3RD EDITION

PRACTICAL PACKET ANALYSIS

USING *WIRESHARK* TO SOLVE REAL-WORLD NETWORK PROBLEMS

CHRIS SANDERS

no starch press

PRAISE FOR *PRACTICAL PACKET ANALYSIS*

"A wealth of information. Smart, yet very readable, and honestly made me excited to read about packet analysis."
—TechRepublic

"I'd recommend this book to junior network analysts, software developers, and the newly minted CSE/CISSP/etc.—folks that just need to roll up their sleeves and get started troubleshooting network (and security) problems."
—Gunter Ollmann, former Chief Technical Officer of IOActive

"The next time I investigate a slow network, I'll turn to *Practical Packet Analysis*. And that's perhaps the best praise I can offer on any technical book."
—Michael W. Lucas, author of *Absolute FreeBSD* and *Network Flow Analysis*

"An essential book if you are responsible for network administration on any level."
—Linux Pro Magazine

"A wonderful, simple-to-use, and well-laid-out guide."
—ArsGeek.com

"If you need to get the basics of packet analysis down pat, this is a very good place to start."
—StateofSecurity.com

"Very informative and held up to the key word in its title, *practical*. It does a great job of giving readers what they need to know to do packet analysis and then jumps right in with vivid real-life examples of what to do with Wireshark."
—LinuxSecurity.com

"Are there unknown hosts chatting away with each other? Is my machine talking to strangers? You need a packet sniffer to really find the answers to these questions. Wireshark is one of the best tools to do this job, and this book is one of the best ways to learn about that tool."
—Free Software Magazine

"Perfect for the beginner to intermediate."
—Daemon News

PRACTICAL PACKET ANALYSIS

3RD EDITION

Using Wireshark to Solve Real-World Network Problems

by Chris Sanders

no starch press

San Francisco

PRACTICAL PACKET ANALYSIS, 3RD EDITION. Copyright © 2017 by Chris Sanders.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

21 20 19 18 17 1 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-802-0
ISBN-13: 978-1-59327-802-1

Publisher: William Pollock
Production Editor: Serena Yang
Cover Illustration: Octopod Studios
Interior Design: Octopod Studios
Developmental Editor: William Pollock and Jan Cash
Technical Reviewer: Tyler Reguly
Copyeditor: Paula L. Fleming
Compositor: Janelle Ludowise
Proofreader: James Fraleigh
Indexer: BIM Creatives, LLC.

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:
No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com
www.nostarch.com

The Library of Congress has catalogued the first edition as follows:

Sanders, Chris, 1986-
Practical packet analysis : using Wireshark to solve real-world network problems / Chris Sanders.
p. cm.
ISBN-13: 978-1-59327-149-7
ISBN-10: 1-59327-149-2
1. Computer network protocols. 2. Packet switching (Data transmission) I. Title.
TK5105.55.S265 2007
004.6'6--dc22
2007013453

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

"Amazing grace, how sweet the sound
That saved a wretch like me.
I once was lost but now I'm found.
Was blind but now I see."

BRIEF CONTENTS

Acknowledgments xv
Introduction xvii
Chapter 1: Packet Analysis and Network Basics 1
Chapter 2: Tapping into the Wire 17
Chapter 3: Introduction to Wireshark 37
Chapter 4: Working with Captured Packets 53
Chapter 5: Advanced Wireshark Features. 77
Chapter 6: Packet Analysis on the Command Line 103
Chapter 7: Network Layer Protocols 119
Chapter 8: Transport Layer Protocols 151
Chapter 9: Common Upper-Layer Protocols 163
Chapter 10: Basic Real-World Scenarios 199
Chapter 11: Fighting a Slow Network 231
Chapter 12: Packet Analysis for Security. 257
Chapter 13: Wireless Packet Analysis 295
Appendix A: Further Reading 317
Appendix B: Navigating Packets 325
Index 333

CONTENTS IN DETAIL

ACKNOWLEDGMENTS **xv**

INTRODUCTION **xvii**

Why This Book? . . . xviii
Concepts and Approach . . . xviii
How to Use This Book . . . xx
About the Sample Capture Files . . . xx
The Rural Technology Fund . . . xxi
Contacting Me . . . xxi

1
PACKET ANALYSIS AND NETWORK BASICS **1**

Packet Analysis and Packet Sniffers . . . 2
- Evaluating a Packet Sniffer . . . 2
- How Packet Sniffers Work . . . 3

How Computers Communicate . . . 4
- Protocols . . . 4
- The Seven-Layer OSI Model . . . 5
- Network Hardware . . . 10

Traffic Classifications . . . 15
- Broadcast Traffic . . . 15
- Multicast Traffic . . . 16
- Unicast Traffic . . . 16

Final Thoughts . . . 16

2
TAPPING INTO THE WIRE **17**

Living Promiscuously . . . 18
Sniffing Around Hubs . . . 19
Sniffing in a Switched Environment . . . 20
- Port Mirroring . . . 21
- Hubbing Out . . . 23
- Using a Tap . . . 24
- ARP Cache Poisoning . . . 27

Sniffing in a Routed Environment . . . 31
Sniffer Placement in Practice . . . 33

3
INTRODUCTION TO WIRESHARK **37**

A Brief History of Wireshark . . . 37
The Benefits of Wireshark . . . 38

Installing Wireshark 39
Installing on Windows Systems 39
Installing on Linux Systems 41
Installing on OS X Systems 43
Wireshark Fundamentals. 44
Your First Packet Capture 44
Wireshark's Main Window 45
Wireshark Preferences. 46
Packet Color Coding 48
Configuration Files 50
Configuration Profiles 50

4
WORKING WITH CAPTURED PACKETS 53

Working with Capture Files. 53
Saving and Exporting Capture Files. 54
Merging Capture Files 55
Working with Packets. 56
Finding Packets. 56
Marking Packets 57
Printing Packets. 58
Setting Time Display Formats and References. 58
Time Display Formats. 59
Packet Time Referencing 60
Time Shifting 60
Setting Capture Options 61
Input Tab 61
Output Tab. 62
Options Tab 63
Using Filters. 65
Capture Filters 65
Display Filters 71
Saving Filters 74
Adding Display Filters to a Toolbar 75

5
ADVANCED WIRESHARK FEATURES 77

Endpoints and Network Conversations 78
Viewing Endpoint Statistics. 78
Viewing Network Conversations 79
Identifying Top Talkers with Endpoints and Conversations 80
Protocol Hierarchy Statistics. 83
Name Resolution 84
Enabling Name Resolution 84
Potential Drawbacks to Name Resolution 86
Using a Custom hosts File. 86
Manually Initiated Name Resolution 88
Protocol Dissection 88
Changing the Dissector 88
Viewing Dissector Source Code 90

Following Streams 91
Following SSL Streams 92
Packet Lengths 93
Graphing 95
Viewing IO Graphs 95
Round-Trip Time Graphing 98
Flow Graphing 99
Expert Information 99

6
PACKET ANALYSIS ON THE COMMAND LINE 103

Installing TShark 104
Installing tcpdump 105
Capturing and Saving Packets 106
Manipulating Output 109
Name Resolution 111
Applying Filters 113
Time Display Formats in TShark 114
Summary Statistics in TShark 115
Comparing TShark and tcpdump 118

7
NETWORK LAYER PROTOCOLS 119

Address Resolution Protocol (ARP) 120
ARP Packet Structure 121
Packet 1: ARP Request 122
Packet 2: ARP Response 123
Gratuitous ARP 124
Internet Protocol (IP) 125
Internet Protocol Version 4 (IPv4) 125
Internet Protocol Version 6 (IPv6) 133
Internet Control Message Protocol (ICMP) 144
ICMP Packet Structure 144
ICMP Types and Messages 144
Echo Requests and Responses 145
traceroute 147
ICMP Version 6 (ICMPv6) 150

8
TRANSPORT LAYER PROTOCOLS 151

Transmission Control Protocol (TCP) 151
TCP Packet Structure 152
TCP Ports 152
The TCP Three-Way Handshake 155
TCP Teardown 158
TCP Resets 159
User Datagram Protocol (UDP) 160
UDP Packet Structure 161

9
COMMON UPPER-LAYER PROTOCOLS 163

Dynamic Host Configuration Protocol (DHCP) 163
DHCP Packet Structure 164
The DHCP Initialization Process 165
DHCP In-Lease Renewal 170
DHCP Options and Message Types 170
DHCP Version 6 (DHCPv6) 171
Domain Name System (DNS) 173
DNS Packet Structure 173
A Simple DNS Query 174
DNS Question Types 176
DNS Recursion 177
DNS Zone Transfers 181
Hypertext Transfer Protocol (HTTP) 183
Browsing with HTTP 183
Posting Data with HTTP 186
Simple Mail Transfer Protocol (SMTP) 187
Sending and Receiving Email 188
Tracking an Email Message 189
Sending Attachments via SMTP 196
Final Thoughts 198

10
BASIC REAL-WORLD SCENARIOS 199

Missing Web Content 200
Tapping into the Wire 200
Analysis 201
Lessons Learned 204
Unresponsive Weather Service 205
Tapping into the Wire 206
Analysis 206
Lessons Learned 209
No Internet Access 210
Gateway Configuration Problems 210
Unwanted Redirection 213
Upstream Problems 216
Inconsistent Printer 219
Tapping into the Wire 219
Analysis 219
Lessons Learned 222
No Branch Office Connectivity 222
Tapping into the Wire 223
Analysis 223
Lessons Learned 226
Software Data Corruption 226
Tapping into the Wire 226
Analysis 227
Lessons Learned 230
Final Thoughts 230

11
FIGHTING A SLOW NETWORK 231

TCP Error-Recovery Features . . . 232
TCP Retransmissions . . . 232
TCP Duplicate Acknowledgments and Fast Retransmissions. . . . 235
TCP Flow Control . . . 240
Adjusting the Window Size . . . 241
Halting Data Flow with a Zero Window Notification . . . 242
The TCP Sliding Window in Practice . . . 243
Learning from TCP Error-Control and Flow-Control Packets. . . . 247
Locating the Source of High Latency . . . 248
Normal Communications . . . 248
Slow Communications: Wire Latency. . . . 248
Slow Communications: Client Latency . . . 249
Slow Communications: Server Latency . . . 250
Latency Locating Framework. . . . 251
Network Baselining . . . 251
Site Baseline. . . . 252
Host Baseline . . . 253
Application Baseline . . . 254
Additional Notes on Baselines . . . 255
Final Thoughts . . . 255

12
PACKET ANALYSIS FOR SECURITY 257

Reconnaissance . . . 258
SYN Scan . . . 258
Operating System Fingerprinting. . . . 263
Traffic Manipulation . . . 266
ARP Cache Poisoning . . . 267
Session Hijacking . . . 271
Malware. . . . 275
Operation Aurora . . . 275
Remote-Access Trojan . . . 281
Exploit Kit and Ransomware . . . 288
Final Thoughts . . . 294

13
WIRELESS PACKET ANALYSIS 295

Physical Considerations . . . 296
Sniffing One Channel at a Time . . . 296
Wireless Signal Interference . . . 297
Detecting and Analyzing Signal Interference . . . 297
Wireless Card Modes. . . . 298
Sniffing Wirelessly in Windows . . . 300
Configuring AirPcap . . . 300
Capturing Traffic with AirPcap . . . 302
Sniffing Wirelessly in Linux . . . 303
802.11 Packet Structure . . . 304

Adding Wireless-Specific Columns to the Packet List Pane 305
Wireless-Specific Filters 307
Filtering Traffic for a Specific BSS ID 307
Filtering Specific Wireless Packet Types 307
Filtering a Specific Frequency 308
Saving a Wireless Profile 309
Wireless Security 309
Successful WEP Authentication 309
Failed WEP Authentication 311
Successful WPA Authentication 312
Failed WPA Authentication 314
Final Thoughts 315

A
FURTHER READING **317**

Packet Analysis Tools 317
CloudShark 317
WireEdit 318
Cain & Abel 319
Scapy 319
TraceWrangler 319
Tcpreplay 319
NetworkMiner 319
CapTipper 320
ngrep 321
libpcap 321
Npcap 321
hping 321
Python 321
Packet Analysis Resources 321
Wireshark's Home Page 322
Practical Packet Analysis Online Course 322
SANS's Security Intrusion Detection In-Depth Course 322
Chris Sanders's Blog 322
Brad Duncan's Malware Traffic Analysis 322
IANA's Website 323
W. Richard Stevens's TCP/IP Illustrated Series 323
The TCP/IP Guide 323

B
NAVIGATING PACKETS **325**

Packet Representation 326
Using Packet Diagrams 328
Navigating a Mystery Packet 330
Final Thoughts 332

INDEX **333**

ACKNOWLEDGMENTS

I'd like to express sincere gratitude for the people who've supported me and the development of this book.

Ellen, thank you for your unconditional love and for putting up with me pecking away at the keyboard in bed for countless nights while you were trying to sleep.

Mom, even in death the example of kindness you set continues to motivate me. Dad, I learned what hard work was from you and none of this happens without that.

Jason Smith, you're like a brother to me, and I can't thank you enough for being a constant sounding board.

Regarding my coworkers past and present, I'm very fortunate to have surrounded myself with people who've made me a smarter, better person. There's no way I can name everyone, but I want to sincerely thank Dustin, Alek, Martin, Patrick, Chris, Mike, and Grady for supporting me every day and embracing what it means to be servant leaders.

Thanks to Tyler Reguly who served as the primary technical editor. I make stupid mistakes sometimes, and you make me look less stupid. Also, thanks to David Vaughan for providing an extra set of eyes, Jeff Carrell for helping edit the IPv6 content, Brad Duncan for providing a capture file used in the security chapter, and the team at QA Café for providing a Cloudshark license that I used to organize the packet captures for the book.

Of course, I also have to extend thanks to Gerald Combs and the Wireshark development team. It's the dedication of Gerald and hundreds of other developers that makes Wireshark such a great analysis platform. If it weren't for their efforts, information technology and network security would be significantly worse off.

Finally, thanks to Bill, Serena, Anna, Jan, Amanda, Alison, and the rest of the No Starch Press staff for their diligence in editing and producing all three editions of *Practical Packet Analysis*.

INTRODUCTION

This third edition of *Practical Packet Analysis* was written and edited over the course of a year and a half, from late 2015 to early 2017, approximately 6 years after the second edition's release and 10 years since publication of the original. This book contains a significant amount of new content, with completely new capture files and scenarios and an entirely new chapter covering packet analysis from the command line with TShark and tcpdump. If you liked the first two editions, then you'll like this one. It's written in the same tone and breaks down explanations in a simple, understandable manner. If you were hesitant to try out the last two editions because they didn't include the latest information on networking or Wireshark updates, you'll want to read this one because of the expanded content on new network protocols and updated information on Wireshark 2.*x*.

Why This Book?

You may find yourself wondering why you should buy this book as opposed to any other book about packet analysis. The answer lies in the title: *Practical Packet Analysis*. Let's face it—nothing beats real-world experience, and the closest you can come to that experience in a book is through practical examples with real-world scenarios.

The first half of this book gives you the knowledge you'll need to understand packet analysis and Wireshark. The second half of the book is devoted entirely to practical cases that you could easily encounter in day-to-day network management.

Whether you're a network technician, a network administrator, a chief information officer, a desktop technician, or even a network security analyst, you will benefit greatly from understanding and using the packet analysis techniques described in this book.

Concepts and Approach

I'm generally a really laid-back guy, so when I teach a concept, I try to do so in a really laid-back way. This holds true for the language used in this book. It's easy to get lost in technical jargon, but I've tried my best to keep things as casual as possible. I've defined all the terms and concepts clearly and without any added fluff. After all, I'm from the great state of Kentucky, so I try to keep the big words to a minimum. (But you'll have to forgive me for some of the backwoods country verbiage you'll find throughout the text.)

The first several chapters are integral to understanding the rest of the book, so make it a point to master the concepts in these pages first. The second half of the book is purely practical. You may not see these exact scenarios in your workplace, but you will be able to apply the concepts they teach in the situations you do encounter.

Here is a quick breakdown of this book's contents:

Chapter 1: Packet Analysis and Network Basics

What is packet analysis? How does it work? How do you do it? This chapter covers the basics of network communication and packet analysis.

Chapter 2: Tapping into the Wire

This chapter covers the different techniques for placing a packet sniffer on your network.

Chapter 3: Introduction to Wireshark

Here, we'll look at the basics of Wireshark—where to get it, how to use it, what it does, why it's great, and all that good stuff. This edition includes a new discussion about customizing Wireshark with configuration profiles.

Chapter 4: Working with Captured Packets

After you have Wireshark up and running, you'll want to know how to interact with captured packets. This is where you'll learn the basics, including new, more detailed sections on following packet streams and name resolution.

Chapter 5: Advanced Wireshark Features

Once you've learned to crawl, it's time to take off running. This chapter delves into the advanced Wireshark features, taking you under the hood to show you some of the less apparent operations. This includes new, more detailed sections on following packet streams and name resolution.

Chapter 6: Packet Analysis on the Command Line

Wireshark is great, but sometimes you need to leave the comfort of a graphical interface and interact with a packet on the command line. This new chapter shows you how to use TShark and tcpdump, the two best command line packet analysis tools for the job.

Chapter 7: Network Layer Protocols

This chapter shows you what common network layer communication looks like at the packet level by examining ARP, IPv4, IPv6, and ICMP. To troubleshoot these protocols in real-life scenarios, you first need to understand how they work.

Chapter 8: Transport Layer Protocols

Moving up the stack, this chapter discusses the two most common transport protocols, TCP and UDP. The majority of packets you look at will use one of these two protocols, so understanding what they look like at the packet level and how they differ is important.

Chapter 9: Common Upper-Layer Protocols

Continuing with protocol coverage, this chapter shows you what four of the most common upper-layer network communication protocols—HTTP, DNS, DHCP, and SMTP—look like at the packet level.

Chapter 10: Basic Real-World Scenarios

This chapter contains breakdowns of some common traffic and the first set of real-world scenarios. Each scenario is presented in an easy-to-follow format, giving the problem, an analysis, and a solution. These basic scenarios deal with only a few computers and involve a limited amount of analysis—just enough to get your feet wet.

Chapter 11: Fighting a Slow Network

The most common problems network technicians hear about generally involve slow network performance. This chapter is devoted to solving these types of problems.

Chapter 12: Packet Analysis for Security

Network security is the biggest hot-button topic in the information technology area. Chapter 12 shows you some scenarios related to solving security-related issues with packet analysis techniques.

Chapter 13: Wireless Packet Analysis

This chapter is a primer on wireless packet analysis. It discusses the differences between wireless analysis and wired analysis, and it includes some examples of wireless network traffic.

Appendix A: Further Reading

The first appendix of this book suggests some other reference tools and websites that you might find useful as you continue to use the packet analysis techniques you've learned.

Appendix B: Navigating Packets

If you want to dig a little deeper into interpreting individual packets, the second appendix provides an overview of how packet information is stored in binary and how to convert binary into hexadecimal notation. Then it shows you how to dissect packets that are presented in hexadecimal notation with packet diagrams. This is handy if you're going to spend a lot of time analyzing custom protocols or using command line analysis tools.

How to Use This Book

I have intended this book to be used in two ways:

- *As an educational text.* You'll read chapter by chapter, paying particular attention to the real-world scenarios in the later chapters, to gain an understanding of packet analysis.
- *As a reference.* There are some features of Wireshark that you won't use very often, so you may forget how they work. *Practical Packet Analysis* is a great book to have on your bookshelf when you need a quick refresher on how to use a specific feature. When doing packet analysis for your job, you may want to reference the unique charts, diagrams, and methodologies I've provided.

About the Sample Capture Files

All of the capture files used in this book are available from the book's No Starch Press page, *https://www.nostarch.com/packetanalysis3/*. To maximize the potential of this book, download these files and use them as you follow along with the examples.

The Rural Technology Fund

I couldn't write an introduction without mentioning the best thing to come from *Practical Packet Analysis*. Shortly after the release of the first edition of this book, I founded a 501(c)(3) nonprofit organization—the Rural Technology Fund (RTF).

Rural students, even those with excellent grades, often have fewer opportunities for exposure to technology than their city or suburban counterparts. Established in 2008, the RTF is the culmination of one of my biggest dreams. It seeks to reduce the digital divide between rural communities and their urban and suburban counterparts. The RTF does this through targeted scholarship programs, community involvement, donations of educational technology resources to classrooms, and general promotion and advocacy of technology in rural and high-poverty areas.

In 2016, the RTF was able to put technology education resources into the hands of more than 10,000 students in rural and high-poverty areas in the United States. I'm pleased to announce that all of the author's proceeds from this book go directly to the RTF to support these goals. If you want to learn more about the Rural Technology Fund or how you can contribute, visit our website at *http://www.ruraltechfund.org/* or follow us on Twitter @RuralTechFund.

Contacting Me

I'm always thrilled to get feedback from people who read my writing. If you would like to contact me for any reason, you can send all questions, comments, threats, and marriage proposals directly to me at *chris@chrissanders.org*. I also blog regularly at *http://www.chrissanders.org/* and can be followed on Twitter at @chrissanders88.

1

PACKET ANALYSIS AND NETWORK BASICS

A million different things can go wrong with a computer network on any given day—from a simple spyware infection to a complex router configuration error—and it's impossible to solve every problem immediately. The best we can hope for is to be fully prepared with the knowledge and tools we need to respond to these types of issues.

To truly understand network problems, we go to the packet level. All network problems stem from this level, where even the prettiest-looking applications can reveal their horrible implementations and seemingly trustworthy protocols can prove malicious. Here, nothing is hidden from us. Nothing is obscured by misleading menu structures, eye-catching graphics,

or untrustworthy employees—there are no true secrets (only encrypted ones). The more we can do at the packet level, the more we can control our network and solve problems. This is the world of packet analysis.

This book dives into this world headfirst. Through real-world scenarios, you'll learn how to tackle slow network communication, identify application bottlenecks, and even track hackers. By the time you've finished reading this book, you should be able to implement packet analysis techniques that will help you solve even the most difficult problems in your own network.

In this chapter, we'll begin with the basics, focusing on network communication. The material here will help you gain the tools you'll need to examine different scenarios.

Packet Analysis and Packet Sniffers

Packet analysis, often referred to as packet sniffing or protocol analysis, describes the process of capturing and interpreting live data as it flows across a network in order to better understand what is happening on that network. Packet analysis is typically performed by a *packet sniffer*, a tool used to capture raw network data going across the wire.

Packet analysis can help with the following:

- Understanding network characteristics
- Learning who is on a network
- Determining who or what is utilizing available bandwidth
- Identifying peak network usage times
- Identifying malicious activity
- Finding unsecured and bloated applications

There are various types of packet-sniffing programs, including both free and commercial ones. Each program is designed with different goals in mind. A few popular packet analysis programs are tcpdump, OmniPeek, and Wireshark (we'll primarily be using Wireshark in this book). OmniPeek and Wireshark have graphical user interfaces (GUIs), while tcpdump is a command line program.

Evaluating a Packet Sniffer

You need to consider a number of factors when selecting a packet sniffer, including the following:

Supported protocols All packet sniffers can interpret various protocols. Most can interpret common network protocols (such as IPv4 and ICMP), transport protocols (such as TCP and UDP), and even application protocols (such as DNS and HTTP). However, they may not support nontraditional, newer, or more complex protocols (such as IPv6, SMBv2, and SIP). When choosing a sniffer, make sure that it supports the protocols you're going to use.

User friendliness Consider the packet sniffer's layout, ease of installation, and general workflow. The program you choose should fit your level of expertise. If you have very little packet analysis experience, you may want to avoid the more advanced command line packet sniffers like tcpdump. On the other hand, if you are a packet analysis veteran, you may find an advanced program more useful. As you gain experience, you may even find it useful to combine multiple packet-sniffing programs to fit particular scenarios.

Cost The great thing about packet sniffers is that there are many free ones that rival any commercial products. The most notable difference between commercial products and their free alternatives is their reporting engines. Commercial products typically include some form of fancy report generation module, while free applications either lack this capability or offer only very limited reporting.

Program support Even after you have mastered the basics of a sniffing program, you may occasionally need support to solve new problems as they arise. When evaluating available support, look for developer documentation, public forums, and mailing lists. Although there may be a lack of formalized commercial support for free packet-sniffing programs like Wireshark, communities of users and contributors often provide active discussion boards, wikis, and blogs to help you get more out of your packet sniffer.

Source code access Some packet sniffers are open source software. This means that you can view the source code of the program and, in some cases, even suggest and make changes to that source code. If you have a very specific or advanced use case for a sniffing application, this might be an appealing feature. Most commercial applications don't provide source code access.

Operating system support Unfortunately, not all packet sniffers support every operating system. Choose one that will work on all the operating systems that you need to support. If you are a consultant, you may be required to capture and analyze packets on a variety of operating systems, so you'll need a tool that runs on most of them. Also, keep in mind that you'll sometimes capture packets on one machine and review them on another. Variations between operating systems may force you to use a different application for each device.

How Packet Sniffers Work

The packet-sniffing process involves a cooperative effort between software and hardware. This process can be broken down into three steps:

1. **Collection:** First, the packet sniffer collects raw binary data from the wire. Typically this is done by switching the selected network interface into *promiscuous mode*. In this mode, the network card can listen to all traffic on a network segment, not only the traffic that is addressed to it.

2. **Conversion:** Next, the captured binary data is converted into a readable form. This is as far as most advanced command line packet sniffers can go. At this point, the network data can be interpreted only on a very basic level, leaving the majority of the analysis to the end user.
3. **Analysis:** Finally, the packet sniffer conducts an analysis of the captured and converted data. The sniffer verifies the protocol of the captured network data based on the information extracted and begins its analysis of that protocol's specific features.

How Computers Communicate

To fully understand packet analysis, you must know exactly how computers communicate with each other. In this section, we'll examine the basics of network protocols, the Open Systems Interconnections (OSI) model, network data frames, and the hardware that supports it all.

Protocols

Modern networks are made up of a variety of systems running on many different platforms. To communicate between systems, we use a set of common languages called *protocols*. Common protocols include Transmission Control Protocol (TCP), Internet Protocol (IP), Address Resolution Protocol (ARP), and Dynamic Host Configuration Protocol (DHCP). A logical grouping of protocols that work together is called a *protocol stack*.

It might help to think of protocols as similar to the rules that govern human language. Every language has rules such as how to conjugate verbs, how to greet people, and even how to properly thank someone. Protocols work in much the same fashion, allowing us to define how packets should be routed, how to initiate a connection, and how to acknowledge the receipt of data.

A protocol can be extremely simple or highly complex, depending on its function. Although the various protocols can differ significantly, many protocols address the following issues:

Connection initiation Is it the client or server initiating the connection? What information must be exchanged prior to communication?

Negotiation of connection characteristics Is the communication of the protocol encrypted? How are encryption keys transmitted between communicating hosts?

Data formatting How is the data contained within the packet organized? In what order is the data processed by the devices receiving it?

Error detection and correction What happens in the event that a packet takes too long to reach its destination? How does a client recover if it cannot establish communication with a server for a short duration?

Connection termination How does one host signify to the other that communication has ended? What final information must be transmitted in order to gracefully terminate communication?

The Seven-Layer OSI Model

Protocols are separated according to their functions based on the industry-standard OSI reference model. This hierarchical model, with seven distinct layers, is very helpful for understanding network communications. In Figure 1-1, the layers of the OSI model are on the right, and the proper terminology for data at each of these layers is on the left. The application layer at the top represents the programs used to access network resources. The bottom layer is the physical layer, through which the network data travels. The protocols at each layer work together to ensure data is properly handled by the protocols at layers directly above and below.

Figure 1-1: A hierarchical view of the seven layers of the OSI model

NOTE *The OSI model was originally published in 1983 by the International Organization for Standardization (ISO) as a document called ISO 7498. The OSI model is no more than an industry-recommended standard. Protocol developers are not required to follow it exactly. In fact, the OSI model is not the only networking model; for example, some people prefer the Department of Defense (DoD) model, also known as the TCP/IP model.*

Each OSI model layer has a specific function, as follows:

Application layer (layer 7) The topmost layer of the OSI model provides a means for users to access network resources. This is the only layer typically seen by end users, as it provides the interface that is the base for all of their network activities.

Presentation layer (layer 6) This layer transforms the data it receives into a format that can be read by the application layer. The data encoding and decoding done here depends on the application layer protocol that is sending or receiving the data. The presentation layer also handles several forms of encryption and decryption used to secure data.

Session layer (layer 5) This layer manages the *dialogue*, or session, between two computers. It establishes, manages, and terminates this connection among all communicating devices. The session layer is also responsible for establishing whether a connection is duplex (two-way) or half-duplex (one-way) and for gracefully closing a connection between hosts rather than dropping it abruptly.

Transport layer (layer 4) The primary purpose of the transport layer is to provide reliable data transport services to lower layers. Through flow control, segmentation/desegmentation, and error control, the transport layer makes sure data gets from point to point error-free.

Because ensuring reliable data transportation can be extremely cumbersome, the OSI model devotes an entire layer to it. The transport layer utilizes both connection-oriented and connectionless protocols. Certain firewalls and proxy servers operate at this layer.

Network layer (layer 3) This layer, one of the most complex of the OSI layers, is responsible for routing data between physical networks. It sees to the logical addressing of network hosts (for example, through an IP address). It also handles splitting data streams into smaller fragments and, in some cases, error detection. Routers operate at this layer.

Data link layer (layer 2) This layer provides a means of transporting data across a physical network. Its primary purpose is to provide an addressing scheme that can be used to identify physical devices (for example, MAC addresses). Bridges and switches are physical devices that operate at the data link layer.

Physical layer (layer 1) The layer at the bottom of the OSI model is the physical medium through which network data is transferred. This layer defines the physical and electrical nature of all hardware used, including voltages, hubs, network adapters, repeaters, and cabling specifications. The physical layer establishes and terminates connections, provides a means of sharing communication resources, and converts signals from digital to analog and vice versa.

NOTE *A common mnemonic device for remembering the layers of the OSI model is* Please Do Not Throw Sausage Pizza Away. *The first letter of each word refers to each layer of the OSI model, starting with the first layer.*

Table 1-1 lists some of the more common protocols used at each layer of the OSI model.

Table 1-1: Typical Protocols Used at Each Layer of the OSI Model

Layer	Protocols
Application	HTTP, SMTP, FTP, Telnet
Presentation	ASCII, MPEG, JPEG, MIDI
Session	NetBIOS, SAP, SDP, NWLink
Transport	TCP, UDP, SPX
Network	IP, IPX
Data link	Ethernet, Token Ring, FDDI, AppleTalk
Physical	wired, wireless

Although the OSI model is no more than a recommended standard, you should know it by heart as it provides a useful vocabulary for thinking about and describing network problems. As we progress through this book, you will find that router issues soon become “layer 3 problems” and software issues are readily recognized as “layer 7 problems.”

NOTE *A colleague once told me about a user who complained that he could not access a network resource. The issue was the result of the user's entering an incorrect password. My colleague referred to this as a* layer 8 issue. *Layer 8 is the unofficial user layer. This term is commonly used among those who live at the packet level.*

Data Flow Through the OSI Model

The initial data transfer on a network begins at the application layer of the transmitting system. Data works its way down the seven layers of the OSI model until it reaches the physical layer, at which point the physical layer of the transmitting system sends the data to the receiving system. The receiving system picks up the data at its physical layer, and the data proceeds up the layers of the receiving system to the application layer at the top.

Each layer in the OSI model is capable of communicating only with the layers directly above and below it. For example, layer 2 can send and receive data only from layers 1 and 3.

None of the services provided by various protocols at any given level of the OSI is redundant. For example, if a protocol at one layer provides a particular service, then no other protocol at any other layer will provide this same service. Protocols at different levels may have features with similar goals, but they will function a bit differently.

Protocols at corresponding layers on the sending and receiving devices are complementary. So, for example, if a protocol at layer 7 of the sending device is responsible for formatting the data being transmitted, the corresponding protocol at layer 7 of the receiving device is expected to be responsible for reading that formatted data.

Figure 1-2 is a graphical representation of the OSI model as it relates to two communicating devices. You can see communication going from top to bottom on one device and then reversing when it reaches the second device.

Figure 1-2: Protocols working at the same layer on both the sending and receiving systems

Data Encapsulation

The protocols at different layers of the OSI model pass data between each other with the aid of *data encapsulation*. Each layer in the stack is responsible for adding a

header or footer—extra bits of information that allow the layers to communicate—to the data being transferred. For example, when the transport layer receives data from the session layer, the transport layer adds its own header information to that data before passing it to the network layer.

The encapsulation process creates a protocol data unit (PDU), which includes the data being sent and all header or footer information added to it. As data moves down the OSI model and the various protocols add header and footer information, the PDU changes and grows. The PDU is in its final form when it reaches the physical layer, at which point it is sent to the destination device. The receiving device strips the protocol headers and footers from the PDU as the data climbs up the OSI layers in the reverse of the order they were added. Once the PDU reaches the top layer of the OSI model, only the original application layer data remains.

NOTE *The OSI model uses specific terms to describe packaged data at each layer. The physical layer contains bits, the data link layer contains frames, the network layer contains packets, and the transport layer contains segments. The top three layers simply use the term* data. *This nomenclature isn't used much in practice, so we'll generally just use the term* packet *to refer to a complete or partial PDU that includes header and footer information from a few or many layers of the OSI model.*

To illustrate how encapsulation of data works, we'll look at a simplified practical example of a packet being built, transmitted, and received in relation to the OSI model. Keep in mind that as analysts, we don't often talk about the session or presentation layers, so those will be absent in this example (and the rest of this book).

In this scenario, we are attempting to browse to *http://www.google.com/*. First, we must generate a request packet that is transmitted from our source client computer to the destination server computer. This scenario assumes that a TCP/IP communication session has already been initiated. Figure 1-3 illustrates the data encapsulation process in this example.

We begin on our client computer at the application layer. We are browsing to a website, so the application layer protocol being used is HTTP; the HTTP protocol will issue a command to download the file *index.html* from *google.com*.

NOTE *In practice, the browser will request the website document root first, signified by a forward slash (/). When the web server receives this request, it will redirect the browser to whatever file it is configured to serve upon receiving a document root request. This is usually something like* index.html *or* index.php. *We'll cover this more in Chapter 9 when we discuss HTTP.*

Once our application layer protocol has sent the command, our concern is with getting the packet to its destination. The data in our packet is passed down the OSI stack to the transport layer. HTTP is an application layer protocol that uses (or *sits on*) TCP, so TCP serves as the transport layer protocol used to ensure reliable delivery of the packet. A TCP header is generated

and added to the PDU, as shown in the transport layer of Figure 1-3. This TCP header includes sequence numbers and other data that are appended to the packet, ensuring that the packet is properly delivered.

Figure 1-3: A graphical representation of encapsulation of data between client and server

NOTE *We often say that one protocol "sits on" or "rides on" another protocol because of the top-down design of the OSI model. An application protocol such as HTTP provides a particular service and relies on TCP to ensure reliable delivery of its service. Both of those services rely on the IP protocol at the network level to address and deliver their data. Therefore, HTTP sits on TCP, which sits on IP.*

Having done its job, TCP hands the packet off to IP, which is the layer 3 protocol responsible for the logical addressing of the packet. IP creates a header containing logical addressing information, adds it to the PDU, and passes the packet along to the Ethernet on the data link layer. Physical Ethernet addresses are stored in the Ethernet header. The packet is now fully assembled and passed to the physical layer, where it is transmitted as zeros and ones across the network.

The completed packet traverses the network cabling system, eventually reaching the Google web server. The web server begins by reading the packet from the bottom up, meaning that it first reads the data link layer, which contains the physical Ethernet addressing information that the

network card uses to determine that the packet is intended for a particular server. Once this information is processed, the layer 2 information is stripped away, and the layer 3 information is processed.

The layer 3 IP addressing information is read to ensure that the packet is properly addressed and is not fragmented. This data is also stripped away so that the next layer can be processed.

Layer 4 TCP information is now read to ensure that the packet has arrived in sequence. Then the layer 4 header information is stripped away to leave only the application layer data, which can be passed to the web server application hosting the website. In response to this packet from the client, the server should transmit a TCP acknowledgment packet so the client knows its request was received, followed by the *index.html* file.

All packets are built and processed as described in this example, regardless of which protocols are used. But at the same time, keep in mind that not every packet on a network is generated from an application layer protocol, so you will see packets that contain only information from layer 2, 3, or 4 protocols.

Network Hardware

Now it's time to look at network hardware, where the dirty work is done. We'll focus on just a few of the more common pieces of network hardware: hubs, switches, and routers.

Hubs

A *hub* is generally a box with multiple RJ-45 ports, like the NETGEAR hub shown in Figure 1-4. Hubs range from very small 4-port devices to larger 48-port devices designed for rack mounting in a corporate environment.

Figure 1-4: A typical 4-port Ethernet hub

Because hubs can generate a lot of unnecessary network traffic and are capable of operating only in *half-duplex mode* (they cannot send and receive data at the same time), you won't typically see them used in most modern or high-density networks; switches are used instead (discussed in the next section). However, you should know how hubs work, since they will be very important to packet analysis when using the "hubbing out" technique discussed in Chapter 2.

A hub is no more than a *repeating device* that operates on the physical layer of the OSI model. It takes packets sent from one port and transmits (repeats) them to every other port on the device, and it's up to the

receiving device to accept or reject each packet. For example, if a computer on port 1 of a 4-port hub needs to send data to a computer on port 2, the hub sends those packets to ports 2, 3, and 4. The clients connected to ports 3 and 4 examine the destination Media Access Control (MAC) address field in the Ethernet header of the packet and see that the packet is not for them, so they *drop* (discard) the packet. Figure 1-5 illustrates an example in which computer A is transmitting data to computer B. When computer A sends this data, all computers connected to the hub receive it. However, only computer B actually accepts the data; the other computers discard it.

Figure 1-5: The flow of traffic when computer A transmits data to computer B through a hub

As an analogy, suppose that you sent an email with the subject line "Attention all marketing staff" to every employee in your company, rather than to only those people who work in the marketing department. The marketing department employees see the email is for them and open it. The other employees see it's not for them and discard it. You can see how this approach to communication would result in a lot of unnecessary traffic and wasted time, yet this is exactly how a hub functions.

The best alternatives to hubs in production and high-density networks are *switches*, which are *full-duplex devices* that can send and receive data synchronously.

Switches

Like a hub, a switch is designed to repeat packets. However, unlike a hub, rather than broadcasting data to every port, a switch sends data to only the computer for which the data is intended. Switches look just like hubs, as shown in Figure 1-6.

Figure 1-6: A rack-mountable 48-port Ethernet switch

Several larger switches on the market, such as Cisco-branded ones, are managed via specialized, vendor-specific software or web interfaces. These switches are commonly referred to as *managed switches*. Managed switches provide several features that can be useful in network management, including the ability to enable or disable specific ports, view port statistics, make configuration changes, and remotely reboot.

Switches also offer advanced functionality for handling transmitted packets. To be able to communicate directly with specific devices, switches must be able to uniquely identify devices based on their MAC addresses, which means that they must operate on the data link layer of the OSI model.

Switches store the layer 2 address of every connected device in a *CAM table*, which acts as a kind of traffic cop. When a packet is transmitted, the switch reads the layer 2 header information in the packet and, using the CAM table as reference, determines to which port(s) to send the packet. Switches send packets only to specific ports, thus greatly reducing network traffic.

Figure 1-7 illustrates traffic flow through a switch. In this figure, computer A is sending data to only the intended recipient: computer B. Multiple conversations can happen on the network at the same time, but information is communicated directly between the switch and intended recipient, not between the switch and all connected computers.

Figure 1-7: The flow of traffic when computer A transmits data to computer B through a switch

Routers

A *router* is an advanced network device with a much higher level of functionality than a switch or a hub. A router can take many shapes and forms, but most devices have several LED indicator lights on the front and a few network ports on the back, depending on the size of the network. Figure 1-8 shows an example of a small router.

Figure 1-8: A low-level Enterasys router suitable for use in a small to midsized network

Routers operate at layer 3 of the OSI model, where they are responsible for forwarding packets between two or more networks. The process used by routers to direct the flow of traffic among networks is called *routing*. Several types of routing protocols dictate how different types of packets are routed to other networks. Routers commonly use layer 3 addresses (such as IP addresses) to uniquely identify devices on a network.

A good way to illustrate the concept of routing is to use the analogy of a neighborhood with several streets. Think of the houses, with their addresses, as computers. Then think of each street as a network segment. Figure 1-9 illustrates this comparison. From your house, you can easily go visit your neighbors in the other houses on the same street by walking in a straight line from your front door to theirs. In the same way, a switch allows communication among all computers on a network segment.

However, communicating with a neighbor who lives on another street is like communicating with a computer that is not on the same segment. Referring to Figure 1-9, let's say that you're sitting at 502 Vine Street and need to get to 206 Dogwood Lane. In order to do this, you must first turn onto Oak Street and then turn onto Dogwood Lane. Think of this as crossing network segments. If the device at 192.168.0.3 needs to communicate with the device at 192.168.0.54, it must cross a router to get to the 10.100.1.x network and then cross the destination network segment's router before it can get to the destination network segment.

The size and number of routers on a network will typically depend on the network's size and function. Personal and home office networks may have only a small router located at the edge of the network. A large corporate network might have several routers spread throughout various departments, all connecting to one large central router or layer 3 switch (an advanced type of switch that also has built-in functionality to act as a router).

Figure 1-9: Comparison of a routed network to neighborhood streets

As you look at more and more network diagrams, you will come to understand how data flows through these various points. Figure 1-10 shows the layout of a very common form of routed network. In this example, two separate networks are connected via a single router. If a computer on network A wishes to communicate with a computer on network B, the transmitted data must go through the router.

Figure 1-10: The flow of traffic when computer A on one network transmits data to computer X on another network through a router

Traffic Classifications

Network traffic can be classified as one of three types: broadcast, multicast, and unicast. Each classification has a distinct characteristic that determines how packets in that class are handled by networking hardware.

Broadcast Traffic

A *broadcast packet* is a packet that's sent to all ports on a network segment, regardless of whether a given port is a hub or switch.

There are layer 2 and layer 3 forms of broadcast traffic. On layer 2, the MAC address ff:ff:ff:ff:ff:ff is the reserved broadcast address, and any traffic sent to this address is broadcast to the entire network segment. Layer 3 also has a specific broadcast address, but it varies based on the network address range in use.

The highest possible IP address in an IP network range is reserved for use as the broadcast address. For example, if your computer has an address of 192.168.0.20 and a 255.255.255.0 subnet mask, then 192.168.0.255 is the broadcast address (more on IP addressing in Chapter 7).

The extent to which broadcast packets can travel is called the *broadcast domain*, which is the network segment where any computer can directly transmit to another computer without going through a router. In larger networks with multiple hubs or switches connected via different media, broadcast packets transmitted from one switch reach all the ports on all the other switches on the network, as the packets are repeated from switch to switch. Figure 1-11 shows an example of two broadcast domains on a small network. Because each broadcast domain extends until it reaches the router, broadcast packets circulate only within this specified broadcast domain.

Figure 1-11: A broadcast domain extends to everything behind the current routed segment.

Our earlier neighborhood analogy provides good insight into how broadcast domains work, too. You can think of a broadcast domain as being like a neighborhood street where all your neighbors are sitting on their front porch. If you stand on your front porch and yell, the people on your street will be able to hear you. However, if you want to talk to someone on a different street, you need to find a way to speak to that person directly, rather than broadcasting (yelling) from your front porch.

Multicast Traffic

Multicast is a means of transmitting a packet from a single source to multiple destinations simultaneously. The goal of multicasting is to use as little bandwidth as possible. The optimization of this traffic lies in that a stream of data is replicated fewer times along its path to its destination. The exact handling of multicast traffic is highly dependent on its implementation in individual protocols.

The primary method of implementing multicast traffic is via an addressing scheme that joins the packet recipients to a multicast group. This is how IP multicast works. This addressing scheme ensures that the packets cannot be transmitted to computers to which the packets are not destined. In fact, IP devotes an entire range of addresses to multicast. If you see an IP address in the 224.0.0.0 to 239.255.255.255 range, it is most likely handling multicast traffic because these ranges are reserved for that purpose.

Unicast Traffic

A *unicast packet* is transmitted from one computer directly to another. The details of how unicast functions are dependent on the protocol using it. For example, consider a device that wishes to communicate with a web server. This is a one-to-one connection, so this communication process would begin with the client device transmitting a packet to only the web server.

Final Thoughts

This chapter covered the basics of networking that you need as a foundation for packet analysis. You *must* understand what is going on at this level of network communication before you can begin troubleshooting network issues. In Chapter 2, we will look at multiple techniques for capturing the packets you want to analyze.

2

TAPPING INTO THE WIRE

A key decision for effective packet analysis is where to physically position a packet sniffer to appropriately capture the data. Packet analysts often refer to placing the packet sniffer as *sniffing the wire*, *tapping the network*, or *tapping into the wire*.

Unfortunately, sniffing packets isn't as simple as plugging a laptop into a network port and capturing traffic. In fact, it's sometimes more difficult to place a packet sniffer on a network than it is to actually analyze the packets. Sniffer placement is challenging because devices can be connected using a large variety of networking hardware. Figure 2-1 illustrates a typical situation. Because the devices on a modern network (switches and routers) each handle traffic differently, you must take into account the physical setup of the network you are analyzing.

The goal of this chapter is to help you develop an understanding of packet sniffer placement in a variety of different network topologies. But first, let's look at how we're able to see all the packets that cross the wire we're tapping into.

Figure 2-1: Placing your sniffer on the network can be challenging when there are many connections, and getting the data you want can be tricky.

Living Promiscuously

Before you can sniff packets on a network, you need a network interface card (NIC) that supports a promiscuous mode driver. *Promiscuous mode* is what allows a NIC to view all packets crossing the wire.

As you learned in Chapter 1, with network broadcast traffic, it's common for devices to receive packets that are not actually destined for them. For example, the Address Resolution Protocol (ARP), a crucial fixture on any network that we'll examine in depth in Chapter 7, is used to determine which MAC address corresponds to a particular IP address. To find the matching MAC address, a device sends an ARP broadcast packet to every device on its broadcast domain in hopes that the correct one will respond.

A broadcast domain (the network segment where any computer can directly transmit to another computer without going through a router) can consist of several devices, but only the correct recipient device on that domain should be interested in the ARP broadcast packet that's transmitted. It would be terribly inefficient for every device on the network to process the ARP broadcast packet. Instead, if the packet is not destined for the device and therefore isn't useful to it, the device's NIC discards the packet rather than passing it to the CPU for processing.

Discarding packets not destined for the receiving host improves processing efficiency, but it's not so great for packet analysts. As analysts, we typically want to capture *every* packet sent across the wire so we don't risk missing some crucial piece of information.

We can ensure we capture all of the traffic by using the NIC's promiscuous mode. When operating in promiscuous mode, the NIC passes every packet it sees to the host's processor, regardless of addressing. Once the packet makes it to the CPU, a packet-sniffing application can grab it for analysis.

Most modern NICs support promiscuous mode, and Wireshark includes the libpcap/WinPcap driver, which allows it to switch your NIC directly into promiscuous mode from the Wireshark GUI. (We'll talk more about libpcap/WinPcap in Chapter 3.)

For the purposes of this book, you must have a NIC and an operating system that support the use of promiscuous mode. The only time you don't need to sniff in promiscuous mode is when you want to see only the traffic sent directly to the MAC address of the interface from which you are sniffing.

NOTE *Most operating systems (including Windows) will not let you use a NIC in promiscuous mode unless you have elevated user privileges. If you can't legally obtain these privileges on a system, chances are that you shouldn't be performing any type of packet sniffing on that particular network.*

Sniffing Around Hubs

Sniffing on a network that has hubs installed is a dream for any packet analyst. As you learned in Chapter 1, traffic sent through a hub goes through every port connected to that hub. Therefore, to analyze the traffic running through a computer connected to a hub, all you need to do is connect a packet sniffer to an empty port on the hub. You'll be able to see all communication to and from that computer, as well as all communication between any other devices plugged into that hub. As illustrated in Figure 2-2, your visibility window is limitless when your sniffer is connected to a hub-based network.

Figure 2-2: Sniffing on a hub network provides a limitless visibility window.

NOTE *The* visibility window*, as shown in various diagrams throughout this book, represents the devices on the network whose traffic you can see with a packet sniffer.*

Unfortunately for us, hub-based networks are rare because of the headaches they cause network administrators. Since only one device can communicate through a hub at any one time, a connected device must compete for bandwidth with all the other devices trying to communicate. When two or more devices communicate at the same time, packets collide, as shown in Figure 2-3. The result may be packet loss, and the communicating devices may compensate for that loss by retransmitting packets, increasing network congestion. As the level of traffic and number of collisions increase, devices may need to transmit a packet three or four times, and network performance decreases dramatically. It's therefore easy to understand why most modern networks of any size use switches. Although you'll rarely find hubs in use on modern networks, you'll occasionally run into them on networks that support legacy hardware or specialized devices, such as industrial control system (ICS) networks.

Figure 2-3: Collisions occur on a hub network when two or more devices transmit at the same time.

The easiest way to identify whether a hub is in use in a network is to lay eyes on the server room or networking closet. Most hubs are labeled as such. When all else fails, just look in the darkest corner of the server closet for the network hardware with a few inches of dust on it.

Sniffing in a Switched Environment

Switches are the most common type of connection device used in modern networks. They provide an efficient way to transport data via broadcast, unicast, and multicast traffic. Switches allow full-duplex communication, meaning that machines can send and receive data simultaneously.

Unfortunately for packet analysts, switches add complexity. When you connect a sniffer to a port on a switch, you can see only broadcast traffic and the traffic transmitted and received by the device the sniffer is installed on, as shown in Figure 2-4. To capture traffic from a target device on a switched network, you need to take an additional step.

Figure 2-4: The visibility window on a switched network is limited to the port you are plugged into.

There are four primary ways to capture this traffic: port mirroring, hubbing out, using a tap, and ARP cache poisoning.

Port Mirroring

Port mirroring, or *port spanning*, is perhaps the easiest way to capture the traffic from a target device on a switched network. In this type of setup, you must have access to the command line or web management interface of the switch on which the target computer is located. Also, the switch must support port mirroring and have an empty port into which you can plug your sniffer.

To enable port mirroring, you issue a command that forces the switch to copy all traffic on one port to another port. For instance, to capture all the traffic transmitted and received from a device on port 3 of a switch, you could plug your analyzer into port 4 and mirror port 3 to port 4. Figure 2-5 illustrates port mirroring.

Figure 2-5: Port mirroring allows you to expand your visibility window on a switched network.

How you set up port mirroring depends on the manufacturer of your switch. For most enterprise switches, you'll need to log in to a command line interface and configure port mirroring using a specific command. You'll find a list of example port-mirroring commands in Table 2-1.

Table 2-1: Commands Used to Enable Port Mirroring

Manufacturer	Command
Cisco	`set span <source port> <destination port>`
Enterasys	`set port mirroring create <source port> <destination port>`
Nortel	`port-mirroring mode mirror-port <source port> monitor-port <destination port>`

NOTE *Some enterprise switches provide web-based GUIs that offer port mirroring as an option, but these aren't common and aren't standardized. However, if your switch provides an effective way to configure port mirroring through a GUI, by all means use it. Additionally, more small office and home office (SOHO) switches are beginning to provide port-mirroring capabilities, and those are typically configured with a GUI.*

When port mirroring, be aware of the throughput of the ports you are mirroring. Some switch manufacturers allow you to mirror multiple ports to one port, functionality that may be useful when analyzing the communication between two or more devices on a single switch. However, let's consider what can happen using some basic math. If you have a 24-port switch and you mirror 23 full-duplex 100Mbps ports to one port, you have potentially 4,600Mbps flowing to that port. This is well beyond the physical threshold of a single port, so you could cause packet loss or network slowdowns if the traffic reaches a certain level. This is sometimes referred to as oversubscription. In these situations, switches have been known to completely drop excess packets or even "pause" their internal circuitry, preventing communication altogether. Be sure that you don't cause such problems when performing your capture.

Port mirroring may seem like an attractive, low-cost solution for enterprise networks and scenarios in which you need to consistently monitor specific network segments, such as in network security monitoring. However, this technique is usually not reliable enough for such an application. Especially at high throughput levels, port mirroring can provide inconsistent results and cause data loss that can be hard to track down. For such scenarios, you are advised to use a tap, discussed in "Using a Tap" on page 24.

Hubbing Out

Another way to capture the traffic through a target device on a switched network is by *hubbing out*. With this technique, you place the target device and your analyzer system on the same network segment by plugging them both directly into a hub. Many people think of hubbing out as "cheating," but it's really a valid solution when you can't perform port mirroring but still have physical access to the switch the target device is plugged into.

To hub out, all you need is a hub and a few network cables. Once you have your hardware, connect it as follows:

1. Find the switch the target device resides on and unplug the target from the network.
2. Plug the target's network cable into your hub.
3. Plug in another cable that connects your analyzer to the hub.
4. Plug in a network cable from your hub to the network switch to connect the hub to the network.

Now you have put the target device and your analyzer in the same broadcast domain, and all traffic from your target device will be broadcast so that the analyzer can capture those packets, as illustrated in Figure 2-6.

Figure 2-6: Hubbing out isolates your target device and analyzer.

In most situations, hubbing out reduces the duplex of the target device from full (bi-directional) to half (one-directional). While this method isn't the cleanest way to capture packets, it's sometimes your only option when a switch doesn't support port mirroring. But keep in mind that your hub will also require a power connection, which can be difficult to find.

NOTE *As a reminder, it is usually a nice gesture to alert the user of the device that you will be unplugging it, especially if that user happens to be the company CEO!*

FINDING "TRUE" HUBS

When hubbing out, be sure that you're using a true hub and not a falsely labeled switch. Several networking hardware vendors have a bad habit of marketing and selling a device as a "hub" when it actually functions as a low-level switch. If you aren't working with a proven, tested hub, you'll see only your own traffic, not that of the target device.

When you find something you believe is a hub, test it to make sure. The best way to determine whether a device is a true hub is to hook up a pair of computers to it and see whether one computer can sniff traffic between the other computer and various other devices on the network, such as another computer or a printer. If so, you've got a keeper!

Since hubs are so antiquated, they're not mass-produced much anymore. It's almost impossible to buy a true hub off the shelf, so you'll need to be creative in order to find one. A great source is often a surplus auction held by your local school district. Public schools are required to attempt to auction surplus items before disposing of them, and they often have older hardware sitting around. I've seen people walk away from auctions with several hubs for less than the cost of a plate of white beans and cornbread. Alternatively, eBay can be a good source of hubs, but be wary, as you may run into the same issue with mislabeled switches.

Using a Tap

Everybody knows the expression "Why have chicken when you could have steak?" (Or, if you are from the South, "Why have liver loaf when you could have fried bologna?") This choice also applies to hubbing out versus using a tap.

A network *tap* is a hardware device that you can place between two points on your cabling system to capture the packets between those two points. As with hubbing out, you place a piece of hardware on the network that allows you to capture the packets you need. The difference is that rather than using a hub, you use a specialized piece of hardware designed for network analysis.

There are two primary types of network taps: *aggregated* and *nonaggregated*. Both types of taps sit between two devices in order to sniff the communications. The primary difference between an aggregated tap and a nonaggregated tap is that the nonaggregated tap has four ports, as shown in Figure 2-7, and requires separate interfaces for monitoring traffic bidirectionally, while the aggregated tap has only three ports and can monitor bidirectionally with only a single interface.

Figure 2-7: A Barracuda nonaggregated tap

Taps also typically require a power connection, although some include batteries that allow brief stints of packet sniffing.

Aggregated Taps

The aggregated tap is the simplest to use. It has only one physical monitor port for sniffing bidirectional traffic.

To capture all traffic to and from a single computer plugged into a switch using an aggregated tap, follow these steps:

1. Unplug the computer from the switch.
2. Plug one end of a network cable into the computer and plug the other end into the tap's "in" port.
3. Plug one end of another network cable into the tap's "out" port and plug the other end into the network switch.
4. Plug one end of a final cable into the tap's "monitor" port and plug the other end into the computer that is acting as your sniffer.

The aggregated tap should be connected as shown in Figure 2-8. At this point, your sniffer should be capturing all traffic in and out of the computer you've plugged into the tap.

Figure 2-8: Using an aggregated tap to intercept network traffic

Nonaggregated Taps

The nonaggregated tap is slightly more complex than the aggregated type, but it allows a bit more flexibility when capturing traffic. Instead of having a single monitor port that can be used to listen to bidirectional communication, the nonaggregated type has two monitor ports. One monitor port is used for sniffing traffic in one direction (from the computer connected to the tap), and the other monitor port is used for sniffing traffic in the other direction (to the computer connected to the tap).

To capture all traffic to and from a single computer plugged into a switch, follow these steps:

1. Unplug the computer from the switch.
2. Plug one end of a network cable into the computer and plug the other end into the tap's "in" port.
3. Plug one end of another network cable into the tap's "out" port and plug the other end into the network switch.
4. Plug one end of a third network cable into the tap's "monitor A" port and plug the other end into one NIC on the computer that is acting as your sniffer.
5. Plug one end of a final cable into the tap's "monitor B" port and plug the other end into a second NIC on the computer that is acting as your sniffer.

The nonaggregated tap should be connected as shown in Figure 2-9.

Figure 2-9: Using a nonaggregated tap to intercept network traffic

While these examples may make it appear as though you can monitor only a single device using a tap, you can actually monitor many devices by getting creative with your placement of the tap. For example, if you wanted to monitor all the communication between an entire network segment and the Internet, you could place the tap between the switch to which all of the other devices are connected and the network's upstream router. This placement at a network choke point lets you collect the traffic you desire. This strategy is commonly used in security monitoring.

Choosing a Network Tap

Which type of tap is better? In most situations, aggregated taps are preferred because they require less cabling and don't need two NICs on your sniffer computer. However, if you need to capture a high volume of traffic or care about traffic going in only one direction, a nonaggregated tap is a better choice.

You can purchase taps of all sizes, ranging from simple Ethernet taps that run about $150 to enterprise-grade fiber optic taps in the six-figure range. I've used enterprise-grade taps from Ixia (formerly Net Optics), Dualcomm, and Fluke Networks and have been very happy with them, but there are many other great taps available as well. If you're using a tap for an enterprise application, you'll want to be sure the tap has fail-open capability. This means that if the tap malfunctions or dies, packets will still pass through it and network connectivity for the tapped link won't be interrupted.

ARP Cache Poisoning

One of my favorite techniques for tapping into the wire is ARP cache poisoning. We'll cover the ARP protocol in detail in Chapter 7, but a brief explanation is necessary here so you can understand how this technique works.

The ARP Process

Recall from Chapter 1 that the two main types of packet addressing are at layers 2 and 3 of the OSI model. These layer 2 addresses, or MAC addresses, are used in conjunction with whichever layer 3 addressing system you are using. In this book, in accordance with industry-standard terminology, I refer to the layer 3 addressing system as the *IP addressing system*.

All devices on a network communicate with each other on layer 3 using IP addresses. Because switches operate on layer 2 of the OSI model, they are cognizant of only layer 2 MAC addresses, so devices must be able to include this information in packets they construct. When a MAC address is not known, it must be obtained using the known layer 3 IP addresses so traffic can be forwarded to the appropriate device. This translation process is done through the layer 2 protocol ARP.

The ARP process, for computers connected to Ethernet networks, begins when one computer wishes to communicate with another. The transmitting computer first checks its ARP cache to see whether it already has the MAC address associated with the IP address of the destination computer. If it does not, it sends an ARP request to the data link layer broadcast address ff:ff:ff:ff:ff:ff, as discussed in Chapter 1. This broadcast packet is received by every computer on that particular Ethernet segment. The packet basically asks, "Which IP address owns the *xx:xx:xx:xx:xx:xx* MAC address?"

Devices without the destination computer's IP address simply discard this ARP request. The destination machine replies to the packet with its MAC address via an ARP reply. At this point, the original transmitting computer now has the data link layer addressing information it needs to communicate with the remote computer, and it stores that information in its ARP cache for fast retrieval.

How ARP Cache Poisoning Works

ARP cache poisoning, sometimes called *ARP spoofing*, is an advanced form of tapping into the wire on a switched network. It works by sending ARP messages to an Ethernet switch or router with fake MAC (layer 2) addresses in order to intercept the traffic of another computer. Figure 2-10 illustrates this setup.

Figure 2-10: ARP cache poisoning lets you intercept the traffic of your target computer.

This technique is commonly used by attackers to send falsely addressed packets to client systems in order to intercept certain traffic or cause denial-of-service (DoS) attacks on a target. However, it can also be a legitimate way to capture the packets of a target machine on a switched network.

Using Cain & Abel

When attempting to poison the ARP cache, the first step is to acquire the necessary tools and collect some information. For our demonstration, we'll use the popular security tool Cain & Abel from oxid.it (*http://www.oxid.it/*), which supports Windows systems. Download and install it now, according to the directions on the website.

NOTE *When you attempt to download Cain & Abel, there is a good chance that antivirus software or your browser will flag the software as malicious or as a "hacker tool." This tool has multiple uses, including several that could be nefarious. For our purposes, it poses no threat to your system.*

Before you can use Cain & Abel, you'll need to collect certain information, including the IP address of your analyzer system, the remote system from which you wish to capture the traffic, and the router from which the remote system is downstream.

When you first open Cain & Abel, you'll notice a series of tabs near the top of the window. (ARP cache poisoning is only one of Cain & Abel's features.) For our purposes, we'll be working in the Sniffer tab. When you click this tab, you should see an empty table, as shown in Figure 2-11.

Figure 2-11: The Sniffer tab in the Cain & Abel main window

To complete this table, you'll need to activate the program's built-in sniffer and scan your network for hosts. To do so, follow these steps:

1. Click the second icon from the left on the toolbar, which resembles a NIC.
2. You'll be asked to select the interface you wish to sniff. Choose the one that is connected to the network on which you'll be performing your ARP cache poisoning. If this is your first time using Cain & Abel, select this interface and click **OK**. Otherwise, if you've selected an interface in Cain & Abel before, your selection will have been saved, and you'll need to press the NIC icon a second time to select the interface. (Ensure that this button is depressed to activate Cain & Abel's built-in sniffer.)
3. To build a list of available hosts on your network, click the plus (+) button. The MAC Address Scanner dialog appears, as shown in Figure 2-12. The **All hosts in my subnet** radio button should be selected (or you can specify an address range if necessary). Click **OK** to continue.

Figure 2-12: Scanning for MAC addresses using the Cain & Abel network discovery tool

Some Windows 10 users report Cain & Abel is unable to determine the IP address of their network interfaces, which prohibits the completion of this process. If you have this problem, when configuring network interfaces you'll see that the IP address of your interfaces is 0.0.0.0. To remedy this, take the following steps:

1. If Cain & Abel is open, close it.
2. On the desktop search bar, type `ncpa.cpl` to open the Network Connections dialog.
3. Right-click the network interface you'll be sniffing from and click **Properties**.
4. Double-click **Internet Protocol Version 4 (TCP/IPv4)**.
5. Click the **Advanced** button and choose the **DNS** tab.
6. Select the checkbox next to **Use this connection's DNS suffix in DNS registration** to activate it.
7. Click **OK** to exit the open dialogs and relaunch Cain & Abel.

The grid should now be filled with a list of all the hosts on your attached network, along with their MAC addresses, IP addresses, and vendor information. This is the list you'll work from when setting up ARP cache poisoning.

At the bottom of the program window, you should see a set of tabs that will take you to other windows under the Sniffer heading. Now that you have built your host list, you'll be working from the APR (for ARP Poison Routing) tab. Switch to the APR window now by clicking the tab.

Once in the APR window, you are presented with two empty tables. After you've completed the setup steps, the upper table will show the devices involved in your ARP cache poisoning, and the lower one will show all communication between your poisoned machines.

To set up your poisoning, follow these steps:

1. Click in the blank area in the upper portion of the screen. Then click the plus (+) button on the program's standard toolbar.
2. The window that appears has two selection panes. On the left side, you'll see a list of all available hosts on your network. If you click the IP address of the target computer, the pane on the right will show a list of all hosts in the network, except for the target machine's IP address.
3. In the right pane, click the IP address of the router that is directly upstream from the target machine, as shown in Figure 2-13, and click **OK**. The IP addresses of both devices should now be listed in the upper table in the main application window.
4. To complete the process, click the yellow-and-black radiation symbol on the standard toolbar. This will activate Cain & Abel's ARP cache poisoning features and allow your analyzing system to be the middleman for all communications between the target system and its upstream router.

Figure 2-13: Selecting the devices for which you wish to enable ARP cache poisoning

You should now be able to fire up your packet sniffer and begin the analysis process. When you have finished capturing traffic, simply click the yellow-and-black radiation symbol again to stop ARP cache poisoning.

A Word of Caution About ARP Cache Poisoning

A final note on ARP cache poisoning: you should be very aware of the roles of the systems for which you implement this process. For instance, don't use this technique when the target device is something with very high network utilization, such as a file server with a 1Gbps link to the network (especially if your analyzer system provides only a 100Mbps link).

When you reroute traffic using the technique shown in this example, all traffic transmitted and received by the target system must first go through your analyzer system, therefore making your analyzer the bottleneck in the communication process. This rerouting can have a DoS-type effect on the machine you are analyzing, resulting in degraded network performance and faulty analysis data. Traffic congestion can also prohibit SSL-based communication from working as expected.

NOTE *You can avoid having all the traffic go through your analyzer system by using a feature called asymmetric routing. For more information about this technique, see the oxid.it user manual* (http://www.oxid.it/ca_um/topics/apr.htm).

Sniffing in a Routed Environment

All the techniques for tapping into the wire on a switched network are available on routed networks as well. The only major consideration when dealing with routed environments is the importance of sniffer placement when you are troubleshooting a problem that spans multiple network segments.

As you've learned, a device's broadcast domain extends until it reaches a router, at which point the traffic is handed off to the next upstream router. When data must traverse multiple routers, it's important to analyze the traffic on all sides of the router.

For example, consider the problem you might encounter in a network with several segments connected via multiple routers. In this network, each segment communicates with an upstream segment to store and retrieve data. In Figure 2-14, the problem we're trying to solve is that a downstream subnet, network D, can't communicate with any devices on network A.

Figure 2-14: A computer on network D can't communicate with computers on network A.

If you sniff the traffic of a device on network D that is having trouble communicating with devices on other networks, you may clearly see data being transmitted to another segment, but you might not see data coming back. If you rethink the positioning of your sniffer and begin sniffing the traffic in the next upstream network segment (network B), you'll have a clearer picture of what is happening. At this point, you might find that traffic is dropped or routed incorrectly by network B's router. Eventually, this leads you to a router configuration problem that, when corrected, solves your larger dilemma. Although this scenario is a bit broad, the moral of the story is that when dealing with multiple routers and network segments, you may need to move your sniffer around a bit to get the entire picture and pinpoint the problem.

NETWORK MAPS

In our discussion of network placement, we have examined several network maps. A *network map*, or *network diagram*, shows all technical resources on a network and how they are connected.

There is no better way to determine the placement of your packet sniffer than to visualize the network. If you have a network map available, keep it handy, as it will be a valuable asset in the troubleshooting and analysis process. You may even want to make a detailed map of your own network. Remember that sometimes half the battle in troubleshooting is ensuring you are collecting the right data.

Sniffer Placement in Practice

We have looked at four ways to capture network traffic in a switched environment. We can add one more if we simply consider installing a packet-sniffing application on a single device from which we want to capture traffic (the *direct install method*). Given these five methods, it can be a bit confusing to determine which one is the most appropriate. Table 2-2 provides some general guidelines for each method.

As analysts, we need to be as stealthy as possible. In a perfect world, we collect the data we need without leaving a footprint. Just as forensic investigators don't want to contaminate a crime scene, we don't want to contaminate our captured network traffic.

Table 2-2: Guidelines for Packet Sniffing in a Switched Environment

Technique	Guidelines
Port mirroring	• Leaves no network footprint and generates no additional packets. • Can be configured without taking the client offline, which is convenient when mirroring router or server ports. • Requires processing resources from the switch and can be inconsistent at higher throughput levels.
Hubbing out	• Works when you are not concerned about taking the host temporarily offline. • Ineffective when you must capture traffic from multiple hosts, as collisions and dropped packets are likely. • Can result in lost packets on modern 100/1000Mbps hosts because most true hubs are only 10Mbps.

(continued)

Table 2-2 *(continued)*

Technique	Guidelines
Using a tap	• Ideal when you are not concerned about taking the host temporarily offline. • The only option when you need to sniff traffic from a fiber-optic connection. • Preferred solution for enterprise packet capture and continuous monitoring because taps are reliable and can scale to high throughput links. • Since taps are made for the task at hand and are up to par with modern network speeds, this method is superior to hubbing out. • Can be expensive, especially at scale, and so may be cost prohibitive.
ARP cache poisoning	• Considered very sloppy, as it involves injecting packets onto the network to reroute traffic through your sniffer. • When port mirroring is not an option, can be effective for quickly capturing traffic from a device without taking it offline. • Requires great care so as to not impact network functionality.
Direct install	• Usually not recommended because if there is an issue with a host, that issue could cause packets to be dropped or manipulated in such a way that they are not represented accurately. • The NIC of the host doesn't need to be in promiscuous mode. • Best for test environments, examining/baselining performance, and examining capture files created elsewhere.

As we step through practical scenarios in later chapters, we'll discuss the best ways to capture the data we require on a case-by-case basis. For the time being, the flowchart in Figure 2-15 should help you choose the best method for capturing traffic in a given situation. The chart takes different factors into consideration, starting with whether you are capturing packets at home or at work. Remember that this flowchart is simply a general reference and doesn't cover every possible scenario in which you might tap into the wire.

Figure 2-15: A diagram to help determine which method is best for tapping into the wire

3

INTRODUCTION TO WIRESHARK

As mentioned in Chapter 1, several packet-sniffing applications are available for performing network analysis, but we'll focus mostly on Wireshark in this book. This chapter introduces Wireshark.

A Brief History of Wireshark

Wireshark has a very rich history. Gerald Combs, a computer science graduate of the University of Missouri at Kansas City, originally developed it out of necessity. The first version of Combs's application, called Ethereal, was released in 1998 under the GNU Public License (GPL).

Eight years after releasing Ethereal, Combs left his job to pursue other career opportunities. Unfortunately, his employer at that time had full rights to the Ethereal trademarks, and Combs was unable to reach an agreement that would allow him to control the Ethereal brand. Instead, Combs and the rest of the development team rebranded the project as *Wireshark* in mid-2006.

Wireshark has grown dramatically in popularity, and its collaborative development team now boasts more than 500 contributors. The program that exists under the Ethereal name is no longer being developed.

The Benefits of Wireshark

Wireshark offers several benefits that make it appealing for everyday use. Aimed at both the up-and-coming and the expert packet analyst, it offers a variety of features to entice each. Let's examine Wireshark according to the criteria defined in Chapter 1 for selecting a packet-sniffing tool.

Supported protocols Wireshark excels in the number of protocols that it supports—more than 1,000 as of this writing. These range from common ones like IP and DHCP to more advanced proprietary protocols like DNP3 and BitTorrent. And because Wireshark is developed under an open source model, new protocol support is added with each update.

NOTE *In the unlikely event that Wireshark doesn't support a protocol you need, you can code that support yourself. Then you can submit your code to the Wireshark developers for consideration for inclusion in the application. You can learn about what is required to contribute code to the Wireshark project at* https://www.wireshark.org/develop.html.

User-friendliness The Wireshark interface is one of the easiest to understand of any packet-sniffing application. It is GUI based, with clearly written context menus and a straightforward layout. It also provides several features designed to enhance usability, such as protocol-based color coding and detailed graphical representations of raw data. Unlike some of the more complicated command line–driven alternatives, like tcpdump, the Wireshark GUI is accessible to those just entering the world of packet analysis.

Cost Since it's open source and released under the GNU Public License (GPL), Wireshark's pricing can't be beat: it's absolutely free. You can download and use Wireshark for any purpose, whether personal or commercial.

NOTE *Although Wireshark may be free, some people have made the mistake of paying for it by accident. If you search for packet sniffers on eBay, you may be surprised by how many people would love to sell you a "professional enterprise license" for Wireshark for the low, low price of $39.95. If you decide you really want to buy it, give me a call, and we can talk about some oceanfront property in Kentucky I have for sale!*

Program support A software package's level of support can make or break it. Freely distributed software such as Wireshark may not come with any formal support, so the open source community often relies on its user base to provide assistance. Luckily for us, the Wireshark community is one of the most active of any open source project. The Wireshark website links directly to several forms of support, including online documentation; a wiki; FAQs; and a place to sign up for the Wireshark mailing list, which is monitored by most of the program's top developers. Paid support for Wireshark is also available from Riverbed Technology.

Source code access Wireshark is open source software, so you can access the code at any time. This can be useful for troubleshooting application issues, understanding how protocol dissectors work, or making your own contributions.

Operating system support Wireshark supports all major modern operating systems, including Windows, Linux-based, and OS X platforms. You can view a complete list of supported operating systems on the Wireshark home page.

Installing Wireshark

The Wireshark installation process is surprisingly simple. However, before you install Wireshark, make sure that your system meets the following requirements:

- Any modern 32-bit x86 or 64-bit CPU
- 400MB available RAM, but more for larger capture files
- At least 300MB of available storage space, plus space for capture files
- NIC that supports promiscuous mode
- WinPcap/libpcap capture driver

The WinPcap capture driver is the Windows implementation of the pcap packet-capturing application programming interface (API). Simply put, this driver interacts with your operating system to capture raw packet data, apply filters, and switch the NIC in and out of promiscuous mode.

Although you can download WinPcap separately (from *http://www.winpcap.org/*), it is typically better to install WinPcap from the Wireshark installation package, because the included version of WinPcap has been tested to work with Wireshark.

Installing on Windows Systems

The current version of Wireshark is tested to support versions of Windows that are still within their extended support lifetime. As of the writing of this book, that encompasses Windows Vista; Windows 7; Windows 8;

Windows 10; and Windows Servers 2003, 2008, and 2012. While Wireshark will often work on other versions of Windows (like Windows XP), those versions are not officially supported.

The first step when installing Wireshark on Windows is to obtain the latest installation build from the official Wireshark web page, *http://www.wireshark.org/*. Navigate to the Download Wireshark section on the website and choose a release mirror. Once you've downloaded the package, follow these steps:

1. Double-click the *.exe* file to begin installation and then click **Next** in the introductory window.
2. Read the licensing agreement and click **I Agree** if you agree.
3. Select the components of Wireshark you wish to install, as shown in Figure 3-1. For our purposes, you can accept the defaults by clicking **Next**.

Figure 3-1: Choosing the Wireshark components you wish to install

4. Click **Next** in the Additional Tasks window.
5. Select the location where you wish to install Wireshark and click **Next**.
6. When the dialog asks whether you want to install WinPcap, first make sure the **Install WinPcap** box is checked, as shown in Figure 3-2. Then click **Install**. The installation process should begin.
7. About halfway through the Wireshark installation, the WinPcap installation should start. When it does, click **Next** in the introductory window, read the licensing agreement, and click **I Agree**.
8. You'll be given the option to install USBPcap, a utility for collecting data from USB devices. Select the appropriate check box if you wish to do so and click **Next**.

Figure 3-2: Selecting the option to install the WinPcap driver

9. WinPcap and, if you selected it, USBPcap should install on your computer. After this installation is complete, click **Finish**.
10. Wireshark should complete its installation. When it's finished, click **Next**.
11. In the installation confirmation window, click **Finish**.

Installing on Linux Systems

Wireshark works on most modern Unix-based platforms. It can be installed either by using the distributions package manager of choice or by downloading and installing the package appropriate for your distribution. It isn't realistic to cover installation procedures for everyone, so we'll just look at a few.

Typically, for system-wide software, root access is a requirement. However, local software installations compiled from source can usually be installed without root access.

RPM-Based Systems

If you're using Red Hat Linux or a distribution based on it, like CentOS, then it's likely the OS has the Yum package management tool installed by default. If that's the case, you may be able to install Wireshark the quick way by pulling it from the distribution's software repository. To do this, open a console window and enter the following command:

```
$ sudo yum install wireshark
```

If any dependencies are needed, you'll be prompted to install them. If everything completes successfully, then you should be able to run Wireshark from the command line and access it via the GUI.

DEB-Based Systems

Most DEB-based distributions, such as Debian or Ubuntu, include the APT package management tool, which allows you to install Wireshark from the OS software repository. To install Wireshark with this tool, open a console window and enter the following:

```
$ sudo apt-get install wireshark wireshark-qt
```

Once again, you'll be prompted to install any required dependencies to complete the installation.

Compiling from Source

Due to changes in operation system architecture and Wireshark features, the instructions for compiling Wireshark from source might change over time. That's one reason it's recommended to use your operating system package manager to perform the installation. However, if your Linux distribution doesn't use an automated package management software or you require a specialized installation, Wireshark can be installed manually by compiling it from source. To do so, complete the following steps:

1. Download the source package from the Wireshark web page.
2. Extract the archive by entering the following (substituting the filename of your downloaded package as appropriate):

   ```
   $ tar -jxvf <file_name_here>.tar.bz2
   ```

3. Before configuring and installing Wireshark, a few dependencies may be required depending on your chosen Linux flavor. For example, Ubuntu 14.04 requires the installation of a few other packages for Wireshark to work. These can be installed by issuing the following command (you'll need to do this as a root-level user or by invoking `sudo` before the command):

   ```
   $ sudo apt-get install pkg-config bison flex qt5-default libgtk-3-dev
   libpcap-dev qttools5-dev-tools
   ```

4. After installing prerequisites, navigate to the directory where the Wireshark files were extracted.
5. Configure the source so that it will build correctly for your distribution of Linux by using the command **`./configure`**. If you wish to deviate from the default installation options, you can specify those options at this point in the installation. If any dependencies are missing, you'll most likely receive an error. You must install and configure those dependencies before proceeding. If configuration is successful, you should see a message noting success, as shown in Figure 3-3.

```
2. sanders@sanders-dev: ~/wireshark-2.0.0 (ssh)
The Wireshark package has been configured with the following options.
                    Build wireshark : yes (with Qt 5)
                Build wireshark-gtk : yes (with GTK+ 3)
                       Build tshark : yes
                      Build tfshark : no
                     Build capinfos : yes
                      Build captype : yes
                      Build editcap : yes
                      Build dumpcap : yes
                     Build mergecap : yes
                   Build reordercap : yes
                    Build text2pcap : yes
                      Build randpkt : yes
                       Build dftest : yes
                      Build rawshark : yes
                  Build androiddump : yes
                        Build echld : no

   Save files as pcap-ng by default : yes
  Install dumpcap with capabilities : no
             Install dumpcap setuid : no
                  Use dumpcap group : (none)
                         Use plugins : yes
       Use external capture sources : yes
                     Use Lua library : no
                 Build Qt RTP player : no
              Build GTK+ RTP player : no
             Build profile binaries : no
                   Use pcap library : yes
                   Use zlib library : yes
                Use kerberos library : no
                  Use c-ares library : no
                Use GNU ADNS library : no
                 Use SMI MIB library : no
              Use GNU crypto library : no
              Use SSL crypto library : no
           Use IPv6 name resolution : yes
                  Use gnutls library : no
     Use POSIX capabilities library : no
                   Use GeoIP library : no
                       Use nl library : no
               Use SBC codec library : no
```

Figure 3-3: When the `./configure` command is successful, a message is displayed with the selected configurations.

6. Enter the **`make`** command to build the source into a binary.
7. Initiate the final installation with **`sudo make install`**.
8. Run **`sudo /sbin/ldconfig`** to complete the installation.

NOTE *If you run into an error following these steps, you may have to install an additional package.*

Installing on OS X Systems

To install Wireshark on OS X, complete these steps:

1. Download the OS X package from the Wireshark web page.
2. Run the installation utility and proceed through its steps. Once you've accepted the required end user license agreement, you'll have the option to select the installation location.
3. Complete the installation wizard.

Wireshark Fundamentals

Once you've successfully installed Wireshark on your system, you can begin to familiarize yourself with it. Now you finally get to open your fully functioning packet sniffer and see . . . absolutely nothing!

Okay, so Wireshark isn't very interesting when you first open it. For things to really get exciting, you need to get some data.

Your First Packet Capture

To get packet data into Wireshark, you'll perform your first packet capture. You may be thinking, "How am I going to capture packets when nothing is wrong on the network?"

First, there is *always* something wrong on the network. If you don't believe me, then go ahead and send an email to all of your network users and let them know that everything is working perfectly.

Secondly, there doesn't need to be something wrong in order for you to perform packet analysis. In fact, most packet analysts spend more time analyzing problem-free traffic than traffic that they are troubleshooting. After all, you need a baseline for comparison to effectively troubleshoot network traffic. For example, if you ever hope to solve a problem with DHCP by analyzing its traffic, you must understand what the flow of working DHCP traffic looks like.

More broadly, to find anomalies in daily network activity, you must know what normal daily network activity looks like. When your network is running smoothly, your observations become a baseline representing what traffic looks like in a normal state.

So, let's capture some packets!

1. Open Wireshark.
2. From the main drop-down menu, select **Capture** and then **Options**. You should see a dialog listing the various interfaces that can be used to capture packets, along with some basic information about each one (Figure 3-4). Take note of the Traffic heading, which shows a line graph indicating the amount of traffic currently passing through that interface. Peaks on a line tell you that you are actually capturing packets. If you aren't, the line will be flat. You can also expand each interface by clicking the arrow to the left of it to see the addressing information, such as the MAC address or IP address, tied to it.
3. Click the interface you wish to use and click **Start**. Data should begin filling the window.
4. Wait about a minute or so, and when you are ready to stop the capture and view your data, click the **Stop** button from the Capture drop-down menu.

Figure 3-4: Selecting an interface on which to perform your packet capture

Once you have completed these steps and finished the capture process, the Wireshark main window should be alive with data. As a matter of fact, you might be overwhelmed by the amount of data that appears, but it will all start to make sense quickly as we break down the main window of Wireshark one piece at a time.

Wireshark's Main Window

You'll spend most of your time in the Wireshark main window. This is where all of the packets you capture are displayed and broken down into a more understandable format. Using the packet capture you just made, let's take a look at Wireshark's main window, shown in Figure 3-5.

Figure 3-5: The Wireshark main window uses a three-pane design.

The three panes in the main window—Packet List, Packet Details, and Packet Bytes from top to bottom—depend on one another. To view the details of an individual packet in the Packet Details pane, you must first select it in the Packet List pane. When you select a portion of the packet in the Packet Details pane, the Packet Bytes pane displays the bytes that correspond with that portion.

NOTE *Notice that Figure 3-5 lists a few different protocols in the Packet List pane. There is no visual separation of protocols on different layers (other than via color coding); all packets are shown as they are received on the wire.*

Here's what each pane contains:

Packet List The top pane displays a table containing all packets in the current capture file. It has columns containing the packet number, the relative time the packet was captured, the source and destination of the packet, the packet's protocol, and some general information found in the packet.

NOTE *When I refer to* traffic*, I'm referring to all packets displayed in the Packet List pane. When I refer to* DNS traffic *specifically, I mean the DNS protocol packets in the Packet List pane.*

Packet Details The middle pane contains a hierarchical display of information about a single packet and can be collapsed or expanded to show all of the information collected about the individual packet.

Packet Bytes The lower pane—perhaps the most confusing—displays a packet in its raw, unprocessed form; that is, it shows what the packet looks like as it travels across the wire. This is raw information with nothing warm or fuzzy to make it easier to follow. We'll discuss methods for interpreting this type of data in Appendix B.

Wireshark Preferences

Wireshark has several preferences that can be customized to meet your needs. To access Wireshark's preferences, select **Edit** from the main drop-down menu and click **Preferences**. You'll see the Preferences dialog, which contains several customizable options, as shown in Figure 3-6.

Wireshark's preferences are divided into six major sections plus an Advanced section:

Appearance These preferences determine how Wireshark presents data. You can change most options here according to your personal preferences, including whether to save window positions, the layout of the three main panes, the placement of the scroll bar, the placement of the Packet List pane columns, the fonts used to display the captured data, and the background and foreground colors.

Figure 3-6: You can customize Wireshark using the Preferences dialog options.

Capture These preferences allow you to specify options related to the way packets are captured, including your default capture interface, whether to use promiscuous mode by default, and whether to update the Packet List pane in real time.

Filter Expressions Later we will discuss how Wireshark allows you to filter traffic based on specific criteria. This section of the Preferences dialog allows you to create and manage those filters.

Name Resolution Through these preferences, you can activate features of Wireshark that allow it to resolve addresses into more recognizable names (including MAC, network, and transport name resolution) and specify the maximum number of concurrent name resolution requests.

Protocols This section allows you to manipulate options related to the capture and display of the various packets Wireshark is capable of decoding. Not every protocol has configurable preferences, but some have several options that can be changed. These options are best left at their defaults unless you have a specific reason to change them.

Statistics This section provides a few configurable options for Wireshark's statistical features, which will be covered in more depth in Chapter 5.

Advanced Settings that don't fit neatly into any of the previous categories can be found here. Editing these settings is something typically only done by Wireshark power users.

Packet Color Coding

If you are anything like me, you enjoy shiny objects and pretty colors. If so, you probably got excited when you saw all those different colors in the Packet List pane, as in the example in Figure 3-7 (well, the figure is in black and white if you're reading this book in print, but you get the idea). It may seem as if these colors are randomly assigned to each packet, but this isn't the case.

27	1.807280	172.16.16.128	172.16.16.255	NBNS	92 Name query NB ISATAP<00>
28	2.557340	172.16.16.128	172.16.16.255	NBNS	92 Name query NB ISATAP<00>
29	3.009402	172.16.16.128	4.2.2.1	DNS	86 Standard query 0xb86a PTR 128.16.16.172.in-addr.arpa
30	3.050866	4.2.2.1	172.16.16.128	DNS	163 Standard query response 0xb86a No such name
31	3.180870	172.16.16.128	157.166.226.25	TCP	66 2918→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
32	3.241650	157.166.226.25	172.16.16.128	TCP	66 80→2918 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=1406 SACK_PE
33	3.241744	172.16.16.128	157.166.226.25	TCP	54 2918→80 [ACK] Seq=1 Ack=1 Win=16872 Len=0
34	3.241956	172.16.16.128	209.85.225.148	TCP	54 2867→80 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
35	3.242063	172.16.16.128	209.85.225.118	TCP	54 2866→80 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
36	3.242129	172.16.16.128	209.85.225.118	TCP	54 2865→80 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
37	3.242223	172.16.16.128	209.85.225.133	TCP	54 2864→80 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
38	3.242292	172.16.16.128	209.85.225.133	TCP	54 2863→80 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
39	3.242311	172.16.16.128	157.166.226.25	HTTP	804 GET / HTTP/1.1

Figure 3-7: Wireshark's color coding allows for quick protocol identification.

Each packet is displayed in a certain color for a reason. The color can reflect the packet's protocol and specific field values. For example, all UDP traffic is blue and all HTTP traffic is green by default. The color coding allows you to quickly differentiate between various protocols so that you don't need to read the protocol field in the Packet List pane for every packet. You'll find that this greatly speeds up the time it takes to browse through large capture files.

Wireshark makes it easy to see which colors are assigned to each protocol through the Coloring Rules window, shown in Figure 3-8. To open this window, select **View** from the main drop-down menu and click **Coloring Rules**.

Figure 3-8: The Coloring Rules window lets you view and modify the coloring of packets.

Coloring rules are based on Wireshark filters, which we will look at in Chapter 4. Using these filters, you can define your own coloring rules and modify existing ones. For example, to change the background color used for HTTP traffic from the default green to lavender, follow these steps:

1. Open Wireshark and access the Coloring Rules window (**View ▸ Coloring Rules**).
2. Find the HTTP coloring rule in the coloring rules list and select it by clicking it once.
3. You'll see the foreground and background colors at the bottom of the screen, as shown in Figure 3-9.

Figure 3-9: When editing a color filter, you can modify both the foreground and background colors.

4. Click the **Background** button.
5. Select the color you wish to use on the color wheel and click **OK**.
6. Click **OK** once more to accept the changes and return to the main window. The user interface should then reload itself to reflect the updated color scheme.

As you work with Wireshark on your network, you'll begin to notice that you deal with certain protocols more than others. Here's where color-coded packets can make your life a lot easier. For example, if you think that there is a rogue DHCP server on your network handing out IP leases, you could modify the coloring rule for the DHCP protocol so that it shows up in bright yellow (or some other easily identifiable color). This would allow you to pick out all DHCP traffic much more quickly, making your packet analysis more efficient.

NOTE *Not too long ago, I was discussing Wireshark coloring rules during a presentation to a local group of students. One student was relieved to find out he could edit the coloring rules because he was color-blind and had trouble distinguishing certain protocols based on the default coloring. The ability to modify the default coloring rules thus provides some degree of accessibility.*

Configuration Files

It's helpful to understand where Wireshark stores various configuration settings should you ever need to modify those files directly. You can find the location of the Wireshark configuration files by selecting **Help** from the main drop-down menu, choosing **About Wireshark**, and clicking the **Folders** tab. This window is shown in Figure 3-10.

Figure 3-10: Locating Wireshark configuration files

The two most important locations in terms of Wireshark customization are the personal and global configuration directories. The global configuration directory contains all of the default settings for Wireshark and is where the default profile stores its settings. The personal configuration folder contains customized settings and profiles unique to your account. Any new profiles you create will be stored in a subdirectory of the personal configuration folder using whatever name you provide.

The difference between global and personal configuration directories is an important one, because any changes made to the global configuration files will affect every Wireshark user on a system.

Configuration Profiles

After learning about Wireshark's preferences, you may find that sometimes you want to use one set of preferences but then quickly switch to another set to address a different scenario. Instead of making you manually reconfigure your preferences every time this occurs, Wireshark introduced configuration profiles, which allow users to create saved sets of preferences.

A configuration profile stores the following:

- Preferences
- Capture filters
- Display filters
- Coloring rules
- Disabled protocols
- Forced decodes
- Recent settings, such as pane sizes, view menu settings, and column widths
- Protocol-specific tables, such as SNMP users and custom HTTP headers

To view the list of profiles, click **Edit** in the main drop-down menu and choose the **Configuration Profiles** option. Alternatively, you can right-click the profiles section at the bottom-right side of the screen and select the **Manage Profiles** option. When you arrive at the Configuration Profiles window, you'll see that Wireshark comes with a few standard profiles, including the Default, Bluetooth, and Classic profiles shown in Figure 3-11. The Latency Investigation profile is a custom profile I've added and is in plaintext, while the global and default profiles are in italics.

Figure 3-11: Viewing configuration profiles

The Configuration Profiles window allows you to create, copy, delete, and apply configuration profiles. The process of creating a new profile is very simple.

1. Configure Wireshark with the settings you'd like to save to a profile.
2. Proceed to the Configuration Profiles window by clicking **Edit** in the main drop-down menu. Select the **Configuration Profiles** option.
3. Click the plus (+) button and give the profile a descriptive name.
4. Click **OK**.

When you'd like to switch profiles, you can go to the Configuration Profile window, click the profile name, and click **OK**. You can do this more quickly by clicking the Profile heading at the bottom right of the Wireshark window and selecting the profile you'd like to use, as shown in Figure 3-12.

Figure 3-12: Quickly switch between profiles through the Profile heading.

One of the most useful aspects of configuration profiles is that each profile is stored in its own directory with a series of configuration files. This means that you can back up your profiles and share them with others. The folders tab shown in Figure 3-10 provides paths to personal and global configuration file directories. To share a profile with a user on another computer, just copy the folder matching the name of the profile you want to share and paste it into the same directory for the appropriate user on another computer.

While reading along in this book, you may find the need to create a few high-level profiles for general troubleshooting, finding the source of network latency, and investigating security issues. Don't be afraid to use profiles liberally. They are real time-savers when you want to quickly switch a few preference options on or off. I've known people who have used dozens of profiles to address different scenarios with great success.

Now that you have Wireshark up and running, you're ready to do some packet analysis. Chapter 4 describes how you can work with the packets you've captured.

4

WORKING WITH CAPTURED PACKETS

Now that you've been introduced to Wireshark, you're ready to start capturing and analyzing packets. In this chapter, you'll learn how to work with capture files, packets, and time-display formats. We'll also cover more advanced options for capturing packets and dive into the world of filters.

Working with Capture Files

You'll find that a good portion of your packet analysis will happen after your capture. Usually, you'll perform several captures at various times, save them, and analyze them all at once. Therefore, Wireshark allows you to save your capture files to be analyzed later. You can also merge multiple capture files.

Saving and Exporting Capture Files

To save a packet capture, select **File ▸ Save As**. You should see the Save file as dialog, as shown in Figure 4-1. You'll be asked for a location to save your packet capture and for the file format you wish to use. If you don't specify a file format, Wireshark will use the default *.pcapng* file format.

Figure 4-1: The Save file as dialog allows you to save your packet captures.

In many cases, you may only want to save a subset of the packets in your capture. To do so, select **File ▸ Export Specified Packets**. The dialog that appears is shown in Figure 4-2. This is a great way to thin bloated packet-capture files. You can choose to save only packets in a specific number range, marked packets, or packets visible as the result of a display filter (marked packets and filters are discussed later in this chapter).

You can export your Wireshark capture data into several formats for viewing in other media or for importing into other packet analysis tools. Formats include plaintext, PostScript, comma-separated values (CSV), and XML. To export your packet capture in one of these formats, choose **File ▸ Export Packet Dissections** and then select the format for the exported file. You'll see a Save As dialog containing options related to the format you've chosen.

Figure 4-2: The Export Specified Packets dialog allows you to have more granular control over the packets you choose to save.

Merging Capture Files

Certain types of analysis require the ability to merge multiple capture files. This is a common practice when comparing two data streams or combining streams of the same traffic that were captured separately.

To merge capture files, open one of the files you want to merge and choose **File ▸ Merge** to bring up the Merge with capture file dialog, shown in Figure 4-3. Select the new file you wish to merge into the already open file and then select the method to use for merging the files. You can prepend the selected file to the currently open one, append it, or merge the files chronologically based on their timestamps.

Figure 4-3: The Merge with capture file dialog allows you to merge two capture files.

Working with Packets

You will eventually encounter a situation involving a very large number of packets. As the number of packets grows into the thousands and even millions, you will need to navigate through packets more efficiently. For this purpose, Wireshark allows you to find and mark packets that match certain criteria. You can also print packets for easy reference.

Finding Packets

To find packets that match particular criteria, open the Find Packet bar, shown circled in Figure 4-4, by pressing CTRL-F. This bar should appear between the Filter bar and the Packet List pane.

Figure 4-4: Finding packets in Wireshark based on specified criteria—in this case, packets matching the display filter expression `tcp`

This pane offers three options for finding packets:

- The Display filter option allows you to enter an expression-based filter that will find only those packets that satisfy that expression. This option is used in Figure 4-4.
- The Hex value option searches for packets with a hexadecimal value you specify.
- The String option searches for packets with a text string you specify. You can specify the pane the search is performed in or make the search string case sensitive.

Table 4-1 shows examples of these search types.

Table 4-1: Search Types for Finding Packets

Search type	Examples

Display filter	`not ip`
	`ip.addr==192.168.0.1`
	`arp`

Hex value	`00ff`
	`ffff`
	`00ABB1f0`

String	`Workstation1`
	`UserB`
	`domain`

Once you've decided which search type you will use, enter your search criteria in the text box and click **Find** to find the first packet that meets your criterion. To find the next matching packet, click **Find** again or press CTRL-N; find the previous matching packet by pressing CTRL-B.

Marking Packets

After you have found packets that match your criterion, you can mark those of particular interest. For example, marking packets will let you save only these packets. Also, you can find your marked packets quickly by their black background and white text, as shown in Figure 4-5.

Figure 4-5: A marked packet is highlighted on your screen. In this example, the second packet is marked and appears darker.

To mark a packet, either right-click it in the Packet List pane and choose **Mark Packet** from the pop-up or click a packet in the Packet List pane and press CTRL-M. To unmark a packet, toggle this setting off by pressing CTRL-M again. You can mark as many packets as you wish in a capture. To jump forward and backward between marked packets, press SHIFT-CTRL-N and SHIFT-CTRL-B, respectively.

Printing Packets

Although most analysis will take place on the computer screen, you may need to print captured data. I occasionally print out packets and tape them to my desk so I can quickly reference their contents while doing other analysis. Being able to print packets to a PDF file is also very convenient, especially when preparing reports.

To print captured packets, open the Print dialog by choosing **File ▸ Print** from the main menu, as shown in Figure 4-6.

Figure 4-6: The Print dialog allows you to print the packets you specify.

As with the Export Specified Packets dialog, you can print a specific packet range, marked packets only, or packets displayed as the result of a filter. You can also select the level of detail you wish to print for each packet. Once you have selected the options, click **Print**.

Setting Time Display Formats and References

Time is of the essence—especially in packet analysis. Everything that happens on a network is time sensitive, and you will need to examine trends and network latency in capture files frequently. Wireshark supplies several configurable options related to time. In this section, we'll look at time display formats and references.

Time Display Formats

Each packet that Wireshark captures is given a timestamp, which is applied to the packet by the operating system. Wireshark can show the absolute timestamp, which indicates the exact moment when the packet was captured, as well as the time in relation to the last captured packet and the beginning and end of the capture.

Options related to time display are found under the View heading on the main menu. The Time Display Format section, shown in Figure 4-7, lets you configure the presentation format as well as the precision of the time display.

Figure 4-7: Several time display formats are available.

The presentation format options let you choose various settings for time display. These include date and time of day, UTC date and time of day, seconds since epoch, seconds since beginning of capture (the default setting), seconds since previous captured packet, and more.

The precision options allow you to set the time display precision to an automatic setting, which takes the format from the capture file, or to a manual setting, such as seconds, milliseconds, microseconds, and so on. We will be changing these options later in the book, so you should familiarize yourself with them now.

NOTE *When comparing packet data from multiple devices, be sure that the devices are synchronized with the same time source, especially if you are performing forensic analysis or troubleshooting. You can use the Network Time Protocol (NTP) to ensure network devices are synced. When examining packets from devices spanning more than one time zone, consider analyzing packets in UTC instead of local time to avoid confusion when reporting your findings.*

Packet Time Referencing

Packet time referencing allows you to configure a certain packet so that all subsequent time calculations are done in relation to that packet. This feature is particularly handy when you are examining a series of sequential events that are triggered at some point other than the start of the capture file.

To set a time reference to a packet, right-click the reference packet in the Packet List pane and choose **Set/Unset Time Reference**. To toggle this reference off, repeat the same action. You can also toggle a packet as a time reference on and off by selecting the packet you wish to reference in the Packet List pane and pressing CTRL-T.

When you enable a time reference on a packet, the Time column in the Packet List pane will display `*REF*`, as shown in Figure 4-8.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	66	1606 → 80 [SYN] Seq=2082691767 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.030107	74.125.95.104	172.16.16.128	TCP	66	80 → 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 Win=5720 Len=0 MSS=1406…
3	0.030182	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082691768 Ack=2775577374 Win=16872 Len=0
4	*REF*	172.16.16.128	74.125.95.104	HTTP	681	GET / HTTP/1.1
5	0.048778	74.125.95.104	172.16.16.128	TCP	60	80 → 1606 [ACK] Seq=2775577374 Ack=2082692395 Win=6976 Len=0
6	0.070954	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]
7	0.071217	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]
8	0.071247	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082692395 Ack=2775580186 Win=16872 Len=0

Figure 4-8: Packet 4 with the packet time reference toggle enabled

Setting a packet time reference is useful only when the time display format of a capture is set to display the time in relation to the beginning of the capture. Any other setting will produce no usable results and indeed will generate a set of times that can be very confusing.

Time Shifting

In some cases, you might encounter packets from multiple sources that are not synchronized to the same time source. This is especially common when examining capture files taken from two locations that contain the same stream of data. While most administrators desire a state in which every device on their network is synced, it's not uncommon for there to be a few seconds of time skew between certain types of devices. Wireshark provides the ability to shift the timestamp on packets to alleviate this problem during your analysis.

To shift the timestamp on one or more packets, select **Edit ▸ Time Shift** or press CTRL-SHIFT-T. On the Time Shift screen that opens, you can specify a time range to shift the entire capture file by, or you can specify a time to set individual packets to. In the example shown in Figure 4-9, I've chosen to shift the timestamp of every packet in the capture by adding two minutes and five seconds to each packet.

Figure 4-9: The Time Shift dialog

Setting Capture Options

We looked at the Capture Interfaces dialog while walking through a very basic packet capture in the last chapter. Wireshark offers quite a few additional capture options that we didn't address then. To access these options, choose **Capture ▸ Options**.

The Capture Interfaces dialog has a lot of bells and whistles, all designed to give you more flexibility while capturing packets. It's divided into three tabs: Input, Output, and Options. We'll examine each separately.

Input Tab

The main purpose of the Input tab (Figure 4-10) is to display all the interfaces available for capturing packets and some basic information for each interface. This includes the friendly name of the interface provided by the operating system, a traffic graph showing the throughput on the interface, and additional configuration options such as promiscuous mode status and buffer size. At the far right (not pictured), there is also a column for the applied capture filter, which we'll talk about in "Capture Filters" on page 65.

In this section, you can click most of these options and edit them inline. For example, if you want to disable promiscuous mode on an interface, you can click that field and change it from enabled to disabled via the provided drop-down menu.

Figure 4-10: The Capture Interfaces Input options tab

Output Tab

The Output tab (Figure 4-11) allows you to automatically store captured packets in a file, rather than capturing them first and then saving the file. Doing so offers you more flexibility in managing how packets are saved. You can choose to save them as a single file or a file set or even use a ring buffer (which we'll cover in a moment) to manage the number of files created. To enable this option, enter a complete file path and name in the File text box. Alternatively, use the Browse... button to select a directory and provide a filename.

Figure 4-11: The Capture Interfaces Output options tab

When you are capturing a large amount of traffic or performing long-term captures, file sets can prove particularly useful. A *file set* is a grouping of multiple files separated by a particular condition. To save to a file set, check the **Create a new file automatically after...** option.

Wireshark uses various triggers to manage saving to file sets based upon a file size or time condition. To enable one of these triggers, select the radio button next to the size- or time-based option and then specify the value and unit on which to trigger. For instance, you can set a trigger that creates a new file after every 1MB of traffic captured or, as shown in Figure 4-12, after every minute of traffic captured.

Name	Date modified	Type	Size
intervalcapture_00001_20151009141804	10/9/2017 2:19 PM	File	172 KB
intervalcapture_00002_20151009141904	10/9/2017 2:20 PM	File	25 KB
intervalcapture_00003_20151009142004	10/9/2017 2:21 PM	File	3,621 KB
intervalcapture_00004_20151009142104	10/9/2017 2:22 PM	File	52 KB
intervalcapture_00005_20151009142204	10/9/2017 2:23 PM	File	47 KB
intervalcapture_00006_20151009142304	10/9/2017 2:24 PM	File	37 KB

Figure 4-12: A file set created by Wireshark at one-minute intervals

The Use a ring buffer option lets you specify a certain number of files your file set will hold before Wireshark begins to overwrite files. Although the term *ring buffer* has multiple meanings, for our purposes, it is essentially a file set that specifies that once the last file it can hold has been written, when more data must be saved, the first file is overwritten. In other words, it establishes a first in, first out (FIFO) method of writing files. You can check this option and specify the maximum number of files you wish to cycle through. For example, say you choose to use multiple files for your capture with a new file created every hour, and you set your ring buffer to 6. Once the sixth file has been created, the ring buffer will cycle back around and overwrite the first file rather than create a seventh file. This ensures that no more than six files (or in this case, hours) of data will remain on your hard drive, while still allowing new data to be written.

Lastly, the Output tab also lets you specify whether to use the *.pcapng* file format. If you plan to interact with your saved packets using a tool that isn't capable of parsing *.pcapng*, you can select the traditional *.pcap* format.

Options Tab

The Options tab contains a number of other packet-capturing choices, including display, name resolution, and capture termination options, shown in Figure 4-13.

Figure 4-13: The Capture Interfaces Options tab

Display Options

The Display Options section controls how packets are shown as they are being captured. The Update list of packets in real-time option is self-explanatory and can be paired with the Automatically scroll during live capture option. When both of these options are enabled, all captured packets are displayed on the screen, with the most recently captured ones shown instantly.

WARNING *When paired, the Update list of packets in real-time and Automatically scroll during live capture options can be processor intensive, even when you are capturing a modest amount of data. Unless you have a specific need to see the packets in real time, it's best to deselect both options.*

The Show extra capture information dialog option lets you enable or suppress the display of a small window that shows the number and percentage of packets that have been captured, sorted by their protocol. I like to show the capture info dialog since I typically don't allow for the live scrolling of packets during capture.

Name Resolution Settings

The Name Resolution section options allow you to enable automatic MAC (layer 2), network (layer 3), and transport (layer 4) name resolution for your capture. We'll discuss name resolution as a general topic in more depth, including its drawbacks, in Chapter 5.

Stop Capture Settings

The Stop capture automatically after... section lets you stop the running capture when certain conditions are met. As with multiple file sets, you can trigger the capture to stop based on file size and time interval, but you can also trigger on number of packets. These options can be used with the multiple-file options on the Output tab.

Using Filters

Filters allow you to specify which packets you have available for analysis. Simply stated, a filter is an expression that defines criteria for the inclusion or exclusion of packets. If there are packets you don't want to see, you can write a filter that gets rid of them. If there are packets you want to see exclusively, you can write a filter that shows only those packets.

Wireshark offers two main types of filters:

- *Capture filters* are specified when packets are being captured and will capture only those packets that are specified for inclusion/exclusion in the given expression.
- *Display filters* are applied to an existing set of captured packets in order to hide unwanted packets or show desired packets based on the specified expression.

Let's look at capture filters first.

Capture Filters

Capture filters are applied during the packet-capturing process to limit the packets delivered to the analyst from the start. One primary reason for using a capture filter is performance. If you know that you do not need to analyze a particular form of traffic, you can simply filter it out with a capture filter and save the processing power that would typically be used in capturing those packets.

The ability to create custom capture filters comes in handy when you're dealing with large amounts of data. The analysis can be sped up by ensuring that you are looking at only the packet relevant to the issue at hand.

As an example, suppose you are troubleshooting an issue with a service running on port 262, but the server you are analyzing runs several different services on a variety of ports. Finding and analyzing only the traffic on one port would be quite a job in itself. To capture only the traffic on a specific port, you could use a capture filter. To do so, use the Capture Interfaces dialog as follows:

1. Choose the **Capture ▸ Options** button next to the interface on which you want to capture packets. This will open the Capture Interfaces dialog.
2. Find the interface you wish to use and scroll to the Capture Filter option in the far-right column.

3. You can apply the capture filter by clicking in this column to enter an expression. We want our filter to show only traffic inbound and outbound to port 262, so enter **port 262**, as shown in Figure 4-14. (We'll discuss expressions in more detail in the next section.) The color of the cell should turn green, indicating that you've entered a valid expression; it will turn red if the expression is invalid.

Figure 4-14: Creating a capture filter in the Capture Interfaces dialog

4. Once you have set your filter, click **Start** to begin the capture.

You should now see only port 262 traffic and be able to more efficiently analyze this particular data.

Capture/BPF Syntax

Capture filters are applied by libpcap/WinPcap and use the Berkeley Packet Filter (BPF) syntax. This syntax is common in several packet-sniffing applications, mostly because packet-sniffing applications tend to rely on the libpcap/WinPcap libraries, which allow for the use of BPFs. A knowledge of BPF syntax will be crucial as you dig deeper into networks at the packet level.

A filter created using the BPF syntax is called an *expression*, and each expression consists of one or more *primitives*. Primitives consist of one or more *qualifiers* (as listed in Table 4-2), followed by an ID name or number, as shown in Figure 4-15.

Table 4-2: The BPF Qualifiers

Qualifier	Description	Examples
Type	Identifies what the ID name or number refers to	`host`, `net`, `port`
Dir	Specifies a transfer direction to or from the ID name or number	`src`, `dst`
Proto	Restricts the match to a particular protocol	`ether`, `ip`, `tcp`, `udp`, `http`, `ftp`

Figure 4-15: A sample capture filter

Given the components of an expression, a qualifier of `dst host` and an ID of 192.168.0.10 would combine to form a primitive. This primitive alone is an expression that would capture traffic only with a destination IP address of 192.168.0.10.

You can use logical operators to combine primitives to create more advanced expressions. Three logical operators are available:

- Concatenation operator AND (`&&`)
- Alternation operator OR (`||`)
- Negation operator NOT (`!`)

For example, the following expression will capture only traffic with a source IP address of 192.168.0.10 and a source or destination port of 80:

```
src host 192.168.0.10 && port 80
```

Hostname and Addressing Filters

Most filters you create will center on a particular network device or grouping of devices. Depending on the circumstances, filtering can be based on a device's MAC address, IPv4 address, IPv6 address, or DNS hostname.

For example, say you're curious about the traffic of a particular host that is interacting with a server on your network. From the server, you can create a filter using the `host` qualifier that captures all traffic associated with that host's IPv4 address:

```
host 172.16.16.149
```

If you are on an IPv6 network, you would filter based on an IPv6 address using the `host` qualifier, as shown here:

```
host 2001:db8:85a3::8a2e:370:7334
```

You can also filter based on a device's hostname with the `host` qualifier, like so:

```
host testserver2
```

Or, if you're concerned that the IP address for a host might change, you can filter based on its MAC address as well by adding the `ether` protocol qualifier:

```
ether host 00-1a-a0-52-e2-a0
```

The transfer direction qualifiers are often used in conjunction with filters, such as the ones in the previous examples, to capture traffic based on whether it's going to or coming from a host. For example, to capture only traffic coming from a particular host, add the `src` qualifier:

```
src host 172.16.16.149
```

To capture only data destined for 172.16.16.149, use the `dst` qualifier:

```
dst host 172.16.16.149
```

When you don't use a type qualifier (`host`, `net`, or `port`) with a primitive, the `host` qualifier is assumed. Therefore, this expression, which excludes that qualifier, is the equivalent of the preceding example:

```
dst 172.16.16.149
```

Port Filters

In addition to filtering on hosts, you can filter based on the ports used in each packet. Port filtering can be used to filter for services and applications that use known service ports. For example, here's a simple filter to capture traffic only to or from port 8080:

```
port 8080
```

To capture all traffic except that on port 8080, this would work:

```
!port 8080
```

The port filters can be combined with transfer direction qualifiers. For example, to capture only traffic going to the web server listening on the standard HTTP port 80, use the `dst` qualifier:

```
dst port 80
```

Protocol Filters

Protocol filters let you filter packets based on certain protocols. They are used to match non–application layer protocols that can't simply be defined by the use of a certain port. Thus, if you want to see only ICMP traffic, you could use this filter:

```
icmp
```

To see everything but IPv6 traffic, this will do the trick:

```
!ip6
```

Protocol Field Filters

One of the real strengths of the BPF syntax is the ability that it gives us to examine every byte of a protocol header in order to create very specific filters based on that data. The advanced filters that we'll discuss in this section will allow you to retrieve a specific number of bytes from a packet beginning at a particular location.

For example, suppose that we want to filter based on the type field of an ICMP header. The type field is located at the very beginning of a packet, which puts it at offset 0. To identify the location to examine within a packet, specify the byte offset in square brackets next to the protocol qualifier—`icmp[0]` in this example. This specification will return a 1-byte integer value that we can compare against. For instance, to get only ICMP packets that represent destination unreachable (type 3) messages, we use the equal to operator in our filter expression:

```
icmp[0] == 3
```

To examine only ICMP packets that represent an echo request (type 8) or echo reply (type 0), use two primitives with the OR operator:

```
icmp[0] == 8 || icmp[0] == 0
```

These filters work great, but they filter based on only 1 byte of information within a packet header. You can also specify the length of the data to be returned in your filter expression by appending the byte length after the offset number within the square brackets, separated by a colon.

For example, say we want to create a filter that captures all ICMP destination-unreachable, host-unreachable packets, identified by type 3, code 1. These are 1-byte fields, located next to each other at offset 0 of the packet header. To do this, we create a filter that checks 2 bytes of data beginning at offset 0 of the packet header, and we compare that data against the hex value 0301 (type 3, code 1), like this:

```
icmp[0:2] == 0x0301
```

A common scenario is to capture only TCP packets with the RST flag set. We will cover TCP extensively in Chapter 8. For now, you just need to know that the flags of a TCP packet are located at offset 13. This is an interesting field because it is collectively 1 byte in size as the flags field, but each particular flag is identified by a single bit within this byte. As I will discuss further in Appendix B, each bit in a byte represents some base 2 number. The bit the flag is stored in is specified by the number the bit represents, so the first bit would represent 1, the second 2, the third 4, and so on. Multiple flags can be set simultaneously in a TCP packet. Therefore, we can't efficiently filter by using a single `tcp[13]` value because several values may represent the RST bit being set.

Instead, we must specify the location within the byte that we wish to examine by appending a single ampersand (`&`), followed by the number that represents where the flag is stored. The RST flag is at the bit representing the number 4 within this byte, and the fact that this bit is set to 4 tells us that the RST flag is set. The filter looks like this:

```
tcp[13] & 4 == 4
```

To see all packets with the PSH flag set, which is identified by the bit location representing the number 8 in the TCP flags at offset 13, our filter would use that location instead:

```
tcp[13] & 8 == 8
```

Sample Capture Filter Expressions

You will often find that the success or failure of your analysis depends on your ability to create filters appropriate for your current situation. Table 4-3 shows a few common capture filters that you might use frequently.

Table 4-3: Commonly Used Capture Filters

Filter	Description
`tcp[13] & 32 == 32`	TCP packets with the URG flag set
`tcp[13] & 16 == 16`	TCP packets with the ACK flag set
`tcp[13] & 8 == 8`	TCP packets with the PSH flag set
`tcp[13] & 4 == 4`	TCP packets with the RST flag set
`tcp[13] & 2 == 2`	TCP packets with the SYN flag set
`tcp[13] & 1 == 1`	TCP packets with the FIN flag set
`tcp[13] == 18`	TCP SYN-ACK packets
`ether host` *`00:00:00:00:00:00`*	Traffic to or from your MAC address
`!ether host` *`00:00:00:00:00:00`*	Traffic not to or from your MAC address
`broadcast`	Broadcast traffic only
`icmp`	ICMP traffic

Filter	Description
`icmp[0:2] == 0x0301`	ICMP destination unreachable, host unreachable
`ip`	IPv4 traffic only
`ip6`	IPv6 traffic only
`udp`	UDP traffic only

Display Filters

A display filter is one that, when applied to a capture file, tells Wireshark to display only packets that match that filter. You can enter a display filter in the Filter text box above the Packet List pane.

Display filters are used more often than capture filters because they allow you to filter the packet data you see without actually omitting the rest of the data in the capture file. That way, if you need to revert to the original capture, you can simply clear the filter expression. They are also a lot more powerful thanks to Wireshark's extensive library of packet dissectors.

As an example, in some situations, you might use a display filter to clear irrelevant broadcast traffic from a capture file by filtering out ARP broadcasts from the Packet List pane when those packets don't relate to the current problem being analyzed. However, because those ARP broadcast packets may be useful later, it's better to filter them temporarily than it is to delete them.

To filter out all ARP packets in the capture window, place your cursor in the Filter text box at the top of the Packet List pane and enter **`!arp`** to remove all ARP packets from the list (Figure 4-16). To remove the filter, click the **X** button, and to save the filter for later, click the plus (+) button.

Figure 4-16: Creating a display filter using the Filter text box above the Packet List pane

There are two ways to apply display filters. One is to apply them directly using the appropriate syntax, as we did in this example. Another is to use the Display Filter Expression dialog to build your filter iteratively; this is the easier method when you are first starting to use filters. Let's explore both methods, starting with the easier first.

The Display Filter Expression Dialog

The Display Filter Expression dialog, shown in Figure 4-17, makes it easy for novice Wireshark users to create capture and display filters. To access this dialog, click the **Expression** button on the Filter toolbar.

Figure 4-17: The Display Filter Expression dialog allows for the easy creation of filters in Wireshark.

The left side of the dialog lists all possible protocol fields, and these fields specify all possible filter criteria. To create a filter, follow these steps:

1. To view the criteria fields associated with a protocol, expand that protocol by clicking the arrow symbol next to it. Once you find the criterion you want to base your filter on, click to select it.
2. Choose how your selected field will relate to the criterion value you supply. This relation is specified as equal to, greater than, less than, and so on.
3. Create your filter expression by specifying a criterion value that will relate to your selected field. You can define this value or select it from predefined ones programmed into Wireshark.
4. Your complete filter will be displayed at the bottom of the screen. When you've finished, click **OK** to insert it into the filter bar.

The Display Filter Expression dialog is great for novice users, but once you get the hang of things, you'll find that manually entering filter expressions greatly increases your efficiency. The display filter expression syntax structure is simple, yet extremely powerful.

The Filter Expression Syntax Structure

When you begin using Wireshark more, you will want to start using the display filter syntax directly in the main window to save time. Fortunately, the syntax used for display filters follows a standard scheme and is easy to navigate. In most cases, this scheme is protocol-centric and follows the format *`protocol.feature.subfeature`*, as you saw when looking at the Display Filter Expression dialog. Now we will look at a few examples.

You will most often use a capture or display filter to see packets based on a specific protocol alone. For example, say you are troubleshooting a TCP problem and you want to see only TCP traffic in a capture file. If so, a simple `tcp` filter will do the job.

Now let's look at things from the other side of the fence. Imagine that in the course of troubleshooting your TCP problem, you have used the ping utility quite a bit, thereby generating a lot of ICMP traffic. You could remove this ICMP traffic from your capture file with the filter `expression !icmp`.

Comparison operators allow you to compare values. For example, when troubleshooting TCP/IP networks, you will often need to view all packets that reference a particular IP address. The equal to comparison operator (`==`) will allow you to create a filter showing all packets with an IP address of 192.168.0.1:

```
ip.addr==192.168.0.1
```

Now suppose that you need to view only packets that are less than 128 bytes. You can use the less than or equal to operator (`<=`) to accomplish this goal:

```
frame.len<=128
```

Table 4-4 shows Wireshark's comparison operators.

Table 4-4: Wireshark Filter Expression Comparison Operators

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Logical operators allow you to combine multiple filter expressions into one statement, dramatically increasing the effectiveness of your filters.

For example, say that you're interested in displaying only packets to two IP addresses. You can use the `or` operator to create one expression that will display packets containing either IP address, like this:

```
ip.addr==192.168.0.1 or ip.addr==192.168.0.2
```

Table 4-5 lists Wireshark's logical operators.

Table 4-5: Wireshark Filter Expression Logical Operators

Operator	Description
`and`	Both conditions must be true.
`or`	Either one of the conditions must be true.
`xor`	One and only one condition must be true.
`not`	Neither one of the conditions is true.

Sample Display Filter Expressions

Although the concepts related to creating filter expressions are fairly simple, you will need to use several specific keywords and operators when creating new filters for various problems. Table 4-6 shows some of the display filters that I use most often. For a complete list, see the Wireshark display filter reference at *http://www.wireshark.org/docs/dfref/*.

Table 4-6: Commonly Used Display Filters

Filter	Description
`!tcp.port==3389`	Filter out RDP traffic
`tcp.flags.syn==1`	TCP packets with the SYN flag set
`tcp.flags.reset==1`	TCP packets with the RST flag set
`!arp`	Clear ARP traffic
`http`	All HTTP traffic
`tcp.port==23 || tcp.port==21`	Telnet or FTP traffic
`smtp || pop || imap`	Email traffic (SMTP, POP, or IMAP)

Saving Filters

Once you begin creating a lot of capture and display filters, you will find that you use certain ones frequently. Fortunately, you don't need to type these in each time you want to use them, because Wireshark lets you save your filters for later use. To save a custom capture filter, follow these steps:

1. Select **Capture ▸ Capture Filters** to open the Capture Filter dialog.
2. Create a new filter by clicking the plus (+) button on the lower left side of the dialog.

3. Enter a name for your filter in the Filter Name box.
4. Enter the actual filter expression in the Filter String box.
5. Click the **OK** button to save your filter expression in the list.

To save a custom display filter, follow these steps:

1. Type your filter into the Filter bar above the Packet List pane in the main window and click the **ribbon** button on the left side of the bar.
2. Click the **Save this Filter** option, and a list of saved display filters will be presented in a separate dialog. There you can provide a name for your filter before clicking **OK** to save it (Figure 4-18).

Figure 4-18: You can save display filters directly from the main toolbar.

Adding Display Filters to a Toolbar

If you have filters that you find yourself flipping on and off frequently, one of the easiest ways to interact with them is to add filter toggles to the Filter bar just above the Packet List pane. To do this, complete the following steps:

1. Type your filter into the Filter bar above the Packet List pane in the main window and click the plus (+) button on the right side of the bar.
2. A new bar will display below the Filter bar where you can provide a name for your filter in the Label field (Figure 4-19). This is the label that will be used to represent the filter on the toolbar. Once you've input something in this field, click **OK** to create a shortcut to this expression in the Filter toolbar.

Figure 4-19: Adding a filter expression shortcut to the Filter toolbar

As you can see in Figure 4-20, we've created a shortcut to a filter that will quickly show any TCP packets with the RST flag enabled. Additions to the filtering toolbar are saved to your configuration profile (as discussed in Chapter 3), making them a powerful way to enhance your ability to identify problems in packet captures in various scenarios.

Figure 4-20: Filtering using a toolbar shortcut

Wireshark includes several built-in filters that are great examples of what a filter should look like. You'll want to use them (together with the Wireshark help pages) when creating your own filters. We'll use filters in examples throughout this book.

5

ADVANCED WIRESHARK FEATURES

Once you master the basics of Wireshark, the next step is to delve into its analysis and graphing capabilities. In this chapter, we'll look at some of these powerful features, including the Endpoints and Conversations windows, the finer points of name resolution, protocol dissection, stream interpretation, IO graphing, and more.

These features, which are unique to Wireshark as a graphical analysis tool, are useful at multiple stages in the analysis process. Make sure to at least attempt to use all the features listed here before moving on, because we'll revisit them frequently as we look at practical analysis scenarios throughout the rest of the book.

Endpoints and Network Conversations

For network communication to take place, data must be flowing between at least two devices. Each device sending or receiving data on the network represents what Wireshark calls an *endpoint*. The communication between two endpoints is called a *conversation*. Wireshark describes endpoints and conversations based on the attributes of the communication, specifically in terms of the addresses used within various protocols.

Endpoints are identified by multiple addresses, which are assigned at different layers of the OSI model. For example, at the data link layer, an endpoint will have a MAC address, which is a unique address built into the device (although it can be modified, potentially making it no longer required). At the network layer, however, the endpoint will have an IP address, which can be changed at any point. We'll discuss in the next few chapters how these types of addresses are used.

Figure 5-1 shows two examples of how addresses are used to identify endpoints in conversations. Conversation A in the figure consists of two endpoints communicating at the data link (MAC) layer. Endpoint A has a MAC address of 00:ff:ac:ce:0b:de, and Endpoint B has a MAC address of 00:ff:ac:e0:dc:0f. Conversation B is defined by two devices communicating at the network (IP) layer. Endpoint A has an IP address of 192.168.1.25, and Endpoint B has an address of 192.168.1.30.

Figure 5-1: Endpoints and conversations on a network

Let's look at how Wireshark can provide information about network communication on a per endpoint or conversation basis.

Viewing Endpoint Statistics

lotsofweb.pcapng

When analyzing traffic, you may find that you can pinpoint a problem as being at a specific endpoint on a network. For example, open the capture file *lotsofweb.pcapng* and open Wireshark's Endpoints window (**Statistics ▸ Endpoints**). This window shows several helpful statistics for each endpoint, as shown in Figure 5-2, including the address, number of packets, and bytes transmitted and received.

Address	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Latitude	Longitude
0.0.0.0	1	342	1	342	0	0	-	-
4.2.2.1	103	11 k	51	7275	52	4151	-	-
4.2.2.2	2	261	1	174	1	87	-	-
4.23.40.126	451	318 k	234	291 k	217	26 k	-	-
8.18.91.65	9	1241	3	387	6	854	-	-
8.18.95.169	18	3328	7	1321	11	2007	-	-
12.120.63.24	13	4753	6	3737	7	1016	-	-
12.129.199.110	20	5383	8	3332	12	2051	-	-
63.215.202.16	7	2069	2	724	5	1345	-	-
64.4.22.46	16	10 k	10	9347	6	1241	-	-
64.191.203.30	18	7061	6	2943	12	4118	-	-
64.208.21.17	10	2781	5	1295	5	1486	-	-
64.208.21.43	551	357 k	309	280 k	242	77 k	-	-
65.173.218.96	473	331 k	263	305 k	210	25 k	-	-
66.35.45.201	1,106	807 k	596	702 k	510	104 k	-	-
66.227.17.18	56	12 k	28	8577	28	3990	-	-
66.235.142.3	10	2285	4	853	6	1432	-	-
66.235.143.54	16	5217	7	1573	9	3644	-	-
66.235.143.121	10	3234	5	1490	5	1744	-	-
67.192.232.82	15	8056	7	6456	8	1600	-	-

Figure 5-2: The Endpoints window lets you view each endpoint in a capture file.

The tabs at the top of the window (TCP, Ethernet, IPv4, IPv6, and UDP) show the number of endpoints organized by protocol. To display only endpoints for a specific protocol, click one of these tabs. You can add additional protocol-filtering tabs by clicking the Endpoint Types box at the bottom right of the screen and selecting the protocol to add. If you would like to use name resolution to view endpoint addresses (see "Name Resolution" on page 84), check the Name resolution checkbox. If you're dealing with a large capture and want to filter the endpoints displayed, you can apply a display filter in the main Wireshark window and select the Limit to display filter option in the Endpoints window. This option will make the window show only the endpoints matching the display filter.

Another handy feature of the Endpoints window is the ability to filter out specific packets for display in the Packet List pane. This is a quick way to drill down into the packets of an individual endpoint. Right-click an endpoint to select the available filtering options. The dialog that appears will let you show or exclude packets related to the selected input. You can also choose the Colorize option in this dialog to export the endpoint address directly into a colorization rule (coloring rules are discussed in Chapter 4). In this way, you can quickly highlight packets related to a given endpoint so you can spot them quickly during analysis.

Viewing Network Conversations

lotsofweb.pcapng

With *lotsofweb.pcapng* still open, access the Wireshark Conversations window **Statistics ▸ Conversations** (Figure 5-3) to display all the conversations in the capture file. The Conversations window is similar to the Endpoints

window, but the Conversations window shows two addresses per line to represent a conversation, as well as the packets and bytes transmitted to and from each device. The column *Address A* is the origin endpoint, and *Address B* is the destination.

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
0.0.0.0	255.255.255.255	1	342	1	342	0	0	82.137333000	0.000000	N/A	N/A
4.2.2.1	172.16.16.128	16	2101	8	1433	8	668	3.009402000	36.237044	316	147
4.2.2.1	172.16.16.197	61	6699	30	4206	31	2493	16.331275000	58.485202	575	341
4.2.2.1	172.16.16.136	26	2626	13	1636	13	990	27.106391000	33.836085	386	234
4.2.2.2	172.16.16.197	2	261	1	174	1	87	23.098007000	0.023047	60 k	30 k
4.23.40.126	172.16.16.197	451	318 k	234	291 k	217	26 k	73.085870000	13.245934	176 k	16 k
8.18.91.65	172.16.16.128	9	1241	3	387	6	854	3.243355000	63.289076	48	107
8.18.95.169	172.16.16.197	18	3328	7	1321	11	2007	17.862227000	56.918573	185	282
12.120.63.24	172.16.16.128	13	4753	6	3737	7	1016	8.836392000	69.578042	429	116
12.129.199.110	172.16.16.197	20	5383	8	3332	12	2051	74.806613000	11.541974	2309	1421
63.215.202.16	172.16.16.128	7	2069	2	724	5	1345	6.684163000	61.630220	93	174
64.4.22.46	172.16.16.128	16	10 k	10	9347	6	1241	6.681906000	12.392572	6033	801
64.191.203.30	172.16.16.136	18	7061	6	2943	12	4118	60.393104000	3.054116	7708	10 k
64.208.21.17	172.16.16.128	10	2781	5	1295	5	1486	8.800115000	0.232560	44 k	51 k
64.208.21.43	172.16.16.128	551	357 k	309	280 k	242	77 k	6.085472000	72.329769	31 k	8523
65.173.218.96	172.16.16.136	473	331 k	263	305 k	210	25 k	59.432328000	27.290208	89 k	7497
66.35.45.201	172.16.16.136	1,106	807 k	596	702 k	510	104 k	10.306330000	83.442116	67 k	10 k
66.227.17.18	172.16.16.197	56	12 k	28	8577	28	3990	17.882206000	50.526514	1358	631
66.235.142.3	172.16.16.197	10	2285	4	853	6	1432	17.860779000	0.245516	27 k	46 k
66.235.143.54	172.16.16.128	16	5217	7	1573	9	3644	4.475410000	15.134210	831	1926
66.235.143.121	172.16.16.197	10	3234	5	1490	5	1744	73.279308000	9.893837	1204	1410

Figure 5-3: The Conversations window lets you dissect each conversation in a capture file.

The Conversation window is organized by protocol. To see only conversations using a particular protocol, click one of the tabs at the top of the window (as with the Endpoints window) or add other protocol types by clicking the Conversation Types button at the lower right. As with the Endpoints window, you can use name resolution, limit the visible conversations using a display filter, and right-click a specific conversation to create filters based on specific conversations. Conversation-based filters are useful for digging into the details of interesting communication sequences.

Identifying Top Talkers with Endpoints and Conversations

lotsofweb.pcapng

The Endpoints and Conversations windows are helpful in network troubleshooting, especially when you're trying to locate the source of a significant amount of traffic on the network.

As an example, let's look again at *lotsofweb.pcapng*. As the name implies, this capture file contains HTTP traffic generated by multiple clients browsing the internet. Figure 5-4 shows a list of endpoints in this capture file sorted by number of bytes.

Notice that the endpoint responsible for the most traffic (by bytes) is the address 172.16.16.128. This is an internal network address (we'll cover how that is determined in Chapter 7), and, as the device responsible for the most communication in this capture, it is given the designation *top talker*.

Wireshark · Endpoints · lotsofweb

IPv4 · 95 | Ethernet · 12 | IPv6 · 5 | TCP · 358 | UDP · 106

Address	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Latitude	Longitude
172.16.16.128	8,324	7387 k	2790	507 k	5534	6879 k	-	-
74.125.103.163	3,927	4232 k	2882	4173 k	1045	58 k	-	-
172.16.16.136	2,349	1455 k	1137	213 k	1212	1241 k	-	-
172.16.16.197	2,157	1073 k	1107	221 k	1050	851 k	-	-
66.35.45.201	1,106	807 k	596	702 k	510	104 k	-	-
74.125.103.147	608	633 k	435	620 k	173	12 k	-	-
74.125.166.28	553	532 k	382	519 k	171	13 k	-	-
74.125.95.149	543	409 k	336	365 k	207	43 k	-	-
64.208.21.43	551	357 k	309	280 k	242	77 k	-	-
65.173.218.96	473	331 k	263	305 k	210	25 k	-	-
4.23.40.126	451	318 k	234	291 k	217	26 k	-	-
209.85.225.165	294	292 k	211	282 k	83	10 k	-	-
205.203.140.65	363	251 k	235	179 k	128	72 k	-	-
204.160.126.126	449	185 k	206	118 k	243	66 k	-	-
204.160.104.126	327	149 k	166	85 k	161	64 k	-	-
72.32.92.4	387	130 k	190	97 k	197	32 k	-	-

☐ Name resolution ☐ Limit to display filter Endpoint Types

Copy Map Close Help

Figure 5-4: The Endpoints window shows which hosts are talking the most.

The address with the second highest amount of traffic is 74.125.103.163, an external (internet) address. When you encounter external addresses that you don't know anything about, you can search the WHOIS registry to find the registered owner. In this case, the American Registry for Internet Numbers (*https://whois.arin.net/ui/*) reveals that Google owns this IP address, as seen in Figure 5-5.

Network	
Net Range	74.125.0.0 - 74.125.255.255
CIDR	74.125.0.0/16
Name	GOOGLE
Handle	NET-74-125-0-0-1
Parent	NET74 (NET-74-0-0-0-0)
Net Type	Direct Allocation
Origin AS	
Organization	Google Inc. (GOGL)
Registration Date	2007-03-13
Last Updated	2012-02-24
Comments	
RESTful Link	https://whois.arin.net/rest/net/NET-74-125-0-0-1
See Also	Related organization's POC records.
See Also	Related delegations.

Figure 5-5: Viewing WHOIS results for 74.125.103.163 points to a Google IP.

DETERMING IP ADDRESS OWNERSHIP WITH WHOIS

IP address assignments are managed by different entities based on their geographic location. ARIN is responsible for IP address assignment in the United States and some surrounding areas, while AfriNIC manages those in Africa, RIPE handles Europe, and APNIC manages Asia/Pacific. Generally, you would perform a WHOIS for an IP at the website of the registry responsible for that IP. Of course, just by looking at an address, you are unlikely to know which regional registry is responsible for it. Websites like Robtex (*http://robtex.com/*) will do the hard work for you and query the correct registry to provide results. However, if you at first query the wrong registry, you will typically be pointed to the correct one.

Given this information, you could assume either that 172.16.16.128 and 74.125.103.163 are communicating a lot with multiple other devices on their own or that both endpoints are communicating with each other. In fact, as is often the case with top-talking endpoint pairs, the endpoints are communicating with each other. To confirm this, open the Conversations window, select the IPv4 tab, and sort the list by bytes. You should see that these two endpoints comprise the conversation with the highest number of transferred bytes. The pattern of transfer suggests a large download, because the number of bytes transmitted from external Address A (74.125.103.163) is much greater than the number of bytes transmitted from internal Address B (172.16.16.128), as shown in Figure 5-6.

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration
74.125.103.163	172.16.16.128	3,927	4232 k	2882	4173 k	1045	58 k	39.247091000	54.307799
66.35.45.201	172.16.16.136	1,106	807 k	596	702 k	510	104 k	10.306330000	83.442116
74.125.103.147	172.16.16.128	608	633 k	435	620 k	173	12 k	9.966132000	7.539890
74.125.166.28	172.16.16.128	553	532 k	382	519 k	171	13 k	3.242850000	38.430937
64.208.21.43	172.16.16.128	551	357 k	309	280 k	242	77 k	6.085472000	72.329769
65.173.218.96	172.16.16.136	473	331 k	263	305 k	210	25 k	59.432328000	27.290208
74.125.95.149	172.16.16.128	415	323 k	271	289 k	144	33 k	3.243592000	80.884280
4.23.40.126	172.16.16.197	451	318 k	234	291 k	217	26 k	73.085870000	13.245934
172.16.16.128	209.85.225.165	274	288 k	71	8345	203	280 k	4.385288000	48.369102
172.16.16.128	205.203.140.65	363	251 k	128	72 k	235	179 k	1.709231000	76.713264
172.16.16.197	204.160.126.126	449	185 k	243	66 k	206	118 k	16.497808000	69.835435
172.16.16.128	204.160.104.126	327	149 k	161	64 k	166	85 k	3.317446000	11.191162
72.32.92.4	172.16.16.136	387	130 k	190	97 k	197	32 k	14.245523000	36.732188

Figure 5-6: The Conversations window confirms that the two top talkers are communicating with each other.

You can examine this conversation by applying this display filter:

```
ip.addr == 74.125.103.163 && ip.addr == 172.16.16.128
```

If you scroll through the list of packets, you'll see several DNS requests to the *youtube.com* domain in the Info column of the Packet List window. This is consistent with our finding that 74.125.103.163 is a Google-owned IP address, because Google owns YouTube.

You'll see how to use the Endpoints and Conversations windows in practical scenarios throughout the remaining chapters of this book.

Protocol Hierarchy Statistics

lotsofweb.pcapng

When dealing with unfamiliar capture files, you'll sometimes need to determine the distribution of traffic by protocol. That is, what percentage of a capture is TCP, IP, DHCP, and so on? Rather than counting packets and totaling the results, Wireshark's Protocol Hierarchy Statistics window can provide this information for you.

For example, with the *lotsofweb.pcapng* file still open and any previously applied filters cleared, open the Protocol Hierarchy Statistics window, as shown in Figure 5-7, by choosing **Statistics ▸ Protocol Hierarchy**.

Wireshark · Protocol Hierarchy Statistics · lotsofweb

Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s
Frame	100.0	12899	100.0	9931436	847 k	0	0	0
Ethernet	100.0	12899	100.0	9931436	847 k	0	0	0
Internet Protocol Version 6	0.2	32	0.1	5020	428	0	0	0
User Datagram Protocol	0.2	32	0.1	5020	428	0	0	0
Link-local Multicast Name Resolution	0.2	28	0.0	2408	205	28	2408	205
DHCPv6	0.0	2	0.0	308	26	2	308	26
Data	0.0	2	0.0	2304	196	2	2304	196
Internet Protocol Version 4	99.7	12861	99.9	9926164	847 k	0	0	0
User Datagram Protocol	1.7	214	0.3	28932	2468	0	0	0
Simple Network Management Protocol	0.0	4	0.0	476	40	4	476	40
Service Location Protocol	0.0	1	0.0	86	7	1	86	7
NetBIOS Name Service	0.3	43	0.0	3956	337	43	3956	337
Multicast Domain Name System	0.0	6	0.0	800	68	6	800	68
Link-local Multicast Name Resolution	0.2	28	0.0	1848	157	28	1848	157
Hypertext Transfer Protocol	0.2	25	0.1	9395	801	25	9395	801
Domain Name System	0.8	105	0.1	11687	997	105	11687	997
Bootstrap Protocol	0.0	2	0.0	684	58	2	684	58
Transmission Control Protocol	98.0	12645	99.7	9897140	844 k	10905	8908786	760 k
Secure Sockets Layer	0.0	1	0.0	103	8	1	103	8
Malformed Packet	0.0	3	0.0	4380	373	3	4380	373
Hypertext Transfer Protocol	13.5	1736	9.9	983871	83 k	1364	723402	61 k
Portable Network Graphics	0.1	17	0.1	10816	922	16	10469	893
Malformed Packet	0.0	1	0.0	347	29	1	347	29
Media Type	0.2	28	0.2	19284	1645	28	19284	1645
Malformed Packet	0.0	1	0.0	314	26	1	314	26
Line-based text data	1.1	145	1.0	103728	8851	145	103728	8851
JPEG File Interchange Format	0.6	72	0.6	62036	5293	72	62036	5293
JavaScript Object Notation	0.0	3	0.0	2843	242	3	2843	242
eXtensible Markup Language	0.1	11	0.1	5946	507	11	5946	507
Compuserve GIF	0.7	95	0.6	55502	4736	95	55502	4736
Internet Group Management Protocol	0.0	2	0.0	92	7	2	92	7
Address Resolution Protocol	0.0	6	0.0	252	21	6	252	21

No display filter.

Close | Copy | Help

Figure 5-7: The Protocol Hierarchy Statistics window shows the distribution of traffic by protocol.

The Protocol Hierarchy Statistics window gives you a snapshot of the type of activity occurring on a network. In Figure 5-7, 100 percent is Ethernet traffic, 99.7 percent is IPv4, 98 percent is TCP, and 13.5 percent is HTTP from web browsing. This information provides a great way to benchmark your network, especially once you have a mental picture of what your network traffic usually looks like. For instance, if you know that 10 percent of your network traffic is normally ARP traffic, but you see 50 percent ARP traffic in a recent capture, then something might be wrong. In some cases, the mere existence of a protocol could be of interest. If you don't have any devices configured to use Spanning Tree Protocol (STP), seeing it in a protocol hierarchy might mean that a device is misconfigured.

Over time, you'll find that you can use the Protocol Hierarchy Statistics window to profile the users and devices on a network simply by looking at the distribution of protocols in use. For example, a higher amount of HTTP traffic will tell you that there's a lot of web browsing going on. You may also find that you can identify specific devices on the network simply by looking at the traffic from a network segment belonging to a business unit. For example, the IT department might use more administrative protocols such as ICMP or SNMP, customer service might be responsible for a high volume of SMTP (email) traffic, and the pesky intern in the corner might be flooding the network with *World of Warcraft* traffic!

Name Resolution

Network data is sent between endpoints with the help of various alphanumeric addressing systems that are often too long or complicated to remember, such as MAC address 00:16:ce:6e:8b:24, IPv4 address 192.168.47.122, or IPv6 address 2001:db8:a0b:12f0::1. *Name resolution* (also called *name lookup*) converts one identifying address into another, mostly for the sake of making the address easier to remember. For example, it's much easier to remember *google.com* than to remember 216.58.217.238. By associating easy-to-read names with these cryptic addresses, we make them easier to remember and identify.

Enabling Name Resolution

Wireshark can use name resolution when it displays packet data to make analysis easier. To have Wireshark use name resolution, choose **Edit ▸ Preferences ▸ Name Resolution**. This window is shown in Figure 5-8. Here are the primary options available in Wireshark for name resolution:

Resolve MAC addresses Uses the ARP protocol to attempt to convert layer 2 MAC addresses, such as 00:09:5b:01:02:03, into layer 3 addresses, such as 10.100.12.1. If attempts at these conversions fail, Wireshark will use the *ethers* file in its program directory to attempt conversion. Wireshark's last resort is to convert the first 3 bytes of the MAC address into the device's IEEE-specified manufacturer name, such as *Netgear_01:02:03*.

Resolve transport names Attempts to convert a port number into a name associated with it, for example, to display port 80 as *http*. This is handy when you encounter an uncommon port and don't know what service is typically associated with it.

Resolve network (IP) addresses Attempts to convert a layer 3 address, such as 192.168.1.50, into an easy-to-read DNS name, such as *MarketingPC1.domain.com*. This is helpful for identifying the purpose or owner of a system, assuming it has a descriptive name.

Figure 5-8: Enabling name resolution in the Preferences dialog. Only Resolve MAC addresses is selected amongst the first three checkboxes pertaining to types of name resolution.

The Name Resolution preferences dialog in Figure 5-8 includes a few other useful options:

Use captured DNS packet data for address resolution Parses DNS data from captured DNS packets to resolve IP addresses to DNS names.

Use an external network name resolver Allows Wireshark to generate queries to the DNS server used by your analysis machine in order to resolve IP addresses to DNS names. This is helpful if you want to use DNS name resolution but the capture you are analyzing doesn't contain the relevant DNS packets.

Maximum concurrent requests Rate limits the number of concurrent DNS queries that can be outstanding at once. Use this option if your capture will generate a lot of DNS requests and you're concerned about taking up too much bandwidth on your network or DNS server.

Only use the profile "hosts" file Limits DNS resolution to the host file associated with the active Wireshark profile. I'll describe how to use this file later in this section.

The changes made in the Preferences screen will persist after Wireshark is closed and reopened. To make name resolution changes on the fly without them being persistent, toggle name resolution settings on or off by clicking View ▸ Name Resolution on the main drop-down menu. You have the option of enabling or disabling name resolution for physical, transport, and network addresses.

You can leverage the various name resolution tools to make your capture files more readable and to save a lot of time in certain situations. For example, you can use DNS name resolution to help readily identify the name of a computer you are trying to pinpoint as the source of a particular packet.

Potential Drawbacks to Name Resolution

Given its benefits, using name resolution may seem like a no-brainer, but there are some potential drawbacks. First, network name resolution can fail if there is no DNS server available to provide the name associated with an IP address. Name resolution information is not saved with the capture file, so the resolution process must take place every time you open a file. If you capture packets on one network and then open the capture on another network, then your system might not be able to access the DNS servers from the source network and name resolution will fail.

In addition, name resolution requires additional processing overhead. When dealing with a very large capture file, you may want to forgo name resolution to conserve system resources. If you try to open a large capture and find your system struggling to load it or Wireshark crashes, disabling name resolution might help.

One further issue is that network name resolution's reliance on DNS may generate unwanted packets that will cloud your capture file as traffic is sent to DNS servers to resolve addresses. Complicating things further, if the capture file you are analyzing contains malicious IP addresses, attempting to resolve them could generate queries to attacker-controlled infrastructure that could tip off an attacker that you are aware of their actions, possibly making you a target. To reduce the risk of clouding your packet file or of unwittingly communicating with an attacker, disable the Use an external network name resolver option in the Name Resolution Preferences dialog.

Using a Custom hosts File

It can be tedious to keep track of traffic from multiple hosts in large capture files, especially when external host resolution isn't available. One way to help is to manually label systems based on their IP addresses with a Wireshark *hosts* file, which is a text file with a list of IP address to name mappings. You can use a *hosts* file to label addresses in Wireshark with names for quick reference. These names will be shown in the Packet List pane.

To use a *hosts* file, follow these steps:

1. Choose **Edit ▸ Preferences ▸ Name Resolution** and select **Only use the profile "hosts" file**.
2. Create a new file using Windows Notepad or a similar text editor. The file should contain one entry per line with an IP address and the name to resolve to, as shown in Figure 5-9. The name you choose on the right will be what is shown in the packet list window whenever Wireshark encounters the IP address on the left.

Figure 5-9: Creating a Wireshark hosts *file*

3. Save the file as a plaintext file with the name *hosts* to the appropriate directory, as listed below. Be sure that the file has no extension!
   - Windows: *<USERPROFILE>\Application Data\Wireshark\hosts*
   - OS X: */Users/<username>/.wireshark/hosts*
   - Linux: */home/<username>/.wireshark/hosts*

Now open a capture, and any IP addresses in your *hosts* file should resolve to the specified names, as shown in Figure 5-10. Instead of IP addresses in the Source and Destination columns of the packet list window, more meaningful names are shown.

Figure 5-10: Name resolution from a hosts *file in Wireshark*

Using *hosts* files in this way can dramatically improve your ability to recognize certain hosts during analysis. When working with a team of analysts, consider sharing a *hosts* file of known assets among your networking staff. This will help your team quickly recognize systems with static addresses, such as servers and routers.

NOTE *If your* hosts *file doesn't appear to be working, make sure that you haven't accidentally added a file extension to the filename. The file's name should simply be* hosts.

Manually Initiated Name Resolution

Wireshark also has the ability to force name resolution on a temporary, on-demand basis. This is done by right-clicking a packet in the Packet List pane and choosing the Edit Resolved Name option. The window that pops up will allow you to specify a name for an address, like a label. This resolution will be lost once the capture file is closed, making this a quick way to label an address without making any permanent changes that would have to be reverted later. I use this technique often because it is a little easier than manually editing a *hosts* file for every packet capture I look at.

Protocol Dissection

One of Wireshark's biggest strengths is its support for the analysis of over a thousand protocols. Wireshark has this capability because it is open source, thus providing a framework for creating *protocol dissectors*. These allow Wireshark to recognize and decode a protocol into various fields so the protocol can be displayed in the user interface. Wireshark uses several dissectors in unison to interpret each packet. For example, the ICMP protocol dissector allows Wireshark to recognize that an IP packet contains ICMP data, pull out the ICMP type and code, and format those fields for display in the Info column of the Packet List pane.

You can think of a dissector as the translator between raw data and the Wireshark program. For a protocol to be supported by Wireshark, it must have a dissector (or you can write your own).

Changing the Dissector

wrongdissector .pcapng

Wireshark uses dissectors to detect individual protocols and decide how to display network information. Unfortunately, Wireshark doesn't always make the right choices when selecting the dissector to use on a packet. This is especially true when a protocol on the network is using a nonstandard configuration, such as a non-default port (which is often configured by network administrators as a security precaution or by employees trying to circumvent access controls).

When Wireshark incorrectly applies dissectors, it's possible to override this selection. For example, open the trace file *wrongdissector.pcapng*. This file contains a bunch of SSL communication between two computers. SSL is the Secure Socket Layer protocol, which is used for encrypted communication between hosts. Under most normal circumstances, viewing SSL traffic in Wireshark won't yield much usable information due to its encrypted nature. However, there is something definitely wrong here. If you peruse the contents of several of these packets by clicking them and examining the Packet Bytes pane, you will find plaintext traffic. In fact, if you look at packet 4, you will find mention of the FileZilla FTP server application. The next few packets clearly display a request and response for both a username and a password.

If this were actually SSL traffic, you wouldn't be able to read any of the data contained in the packets, and you certainly wouldn't see all the usernames and passwords transmitted in clear text, as in Figure 5-11. Given the information shown here, it's safe to assume that this is probably FTP traffic, rather than SSL traffic. Wireshark is likely interpreting this traffic as SSL because it is using port 443, as seen under the Info column, and port 443 is the standard port used for HTTPS (HTTP over SSL).

Figure 5-11: Plaintext usernames and passwords? This looks more like FTP than SSL!

To fix this problem, you can apply a *forced decode* to Wireshark to use the FTP protocol dissector on these packets. Here are the steps:

1. Right-click an SSL packet (such as packet 30) in the Protocol column and select **Decode As**, which opens a new dialog.
2. Tell Wireshark to decode all TCP port 443 traffic as FTP by selecting TCP port in the Field column, entering 443 in the Value column, and selecting FTP from the drop-down menu in the Current column, as shown in Figure 5-12.

Figure 5-12: The Decode As... dialog allows you to create forced decodes.

3. Click **OK** to see the changes immediately applied to the capture file.

The data will be decoded as FTP traffic so you can analyze it from the Packet List pane without needing to dig deep into individual bytes (Figure 5-13).

Figure 5-13: Viewing properly decoded FTP traffic

The forced decode feature can be used multiple times in the same capture file. Wireshark will keep track of your forced decodes for you in the Decode As... dialog, where you can view and edit all of the forced decodes you have created so far.

By default, forced decodes are not saved when you close a capture. You can remedy this by clicking the Save button in the Decode As... dialog. This will save the protocol-decoding rules to your current Wireshark user profile; they will be applied when you open any capture using that profile. Saved decode rules can be removed by clicking the minus button in the dialog.

It's very easy to save decoding rules and forget about them. This can lead to a lot of confusion when you aren't prepared for it, so be mindful of forced decodes. To prevent myself from falling victim to this oversight, I generally avoid saving forced decodes to my primary Wireshark profile.

Viewing Dissector Source Code

The beauty of working with an open source application is that, if you are confused about why something is happening, you can look at the source code and find out why. This really comes in handy when you are trying to determine why a particular protocol has been interpreted incorrectly, because you can examine individual protocol dissectors.

Examining the source code of protocol dissectors can be done directly from the Wireshark website by clicking the Develop link and clicking Browse the Code. This link will send you to the Wireshark code repository, where you can view the release code for recent Wireshark versions. The protocol dissectors are in the *epan/dissectors* folder, and each dissector is labeled *packets-<protocolname>.c*.

These files can be rather complex, but they all follow a standard template and tend to be commented very well. You don't need to be an expert C programmer to understand the basic function of each dissector. If you want to get a deep understanding of what you are seeing in Wireshark, I recommend taking a look at dissectors for some of the simpler protocols.

Following Streams

http_google.pcapng

One of Wireshark's most satisfying analysis features is its ability to reassemble data from multiple packets into a consolidated, easily readable format, often called a *packet transcript*. So you don't have to view data being sent from client to server in a bunch of small chunks while clicking from packet to packet, *stream following* sorts the data to make it easier to view.

Four types of streams are available to follow:

TCP stream Assembles data from protocols that utilize TCP, such as HTTP and FTP.

UDP stream Assembles data from protocols that utilize UDP, such as DNS.

SSL stream Assembles data from protocols that are encrypted, such as HTTPS. You must supply keys to decrypt the traffic.

HTTP stream Assembles and decompresses data from the HTTP protocol. This is useful when following HTTP data via TCP stream doesn't decode the HTTP payload fully.

As an example, consider a simple HTTP transaction in the file *http_google.pcapng*. Click any of the TCP or HTTP packets in the file, right-click the packet, and choose **Follow TCP Stream**. This will consolidate the TCP stream and open the conversation transcript in a separate window, as in Figure 5-14.

The text displayed in this window is in two colors, with red text (shown here with the lighter gray shading) signifying traffic from source to destination and blue text (shown here with the darker gray shading) identifying traffic in the opposite direction, from destination to source. The color relates to which side initiated the communication. In our example, the client initiated the connection to the web server, so it's displayed in red.

The communication in the TCP stream begins with an initial `GET` request for the web root directory (`/`) and a response from the server that the request was successful in the form of an `HTTP/1.1 200 OK`. A similar pattern is repeated throughout other streams in the packet capture as the client requests individual files and the server responds with them. You are seeing a user browsing to the Google home page, but instead of having to step through every packet, you're able to scroll through the transcript with ease. You're actually seeing what the end user is seeing, but from the inside out.

Figure 5-14: The Follow TCP Stream window reassembles the communication in an easily readable format.

In addition to viewing the raw data in this window, you can search within the text; save it as a file; print it; or choose to view the data in ASCII, EBCDIC, hex, or C array format. These options, which make digging through larger sets of data easier, can be found at the bottom of the Follow Stream window.

Following SSL Streams

Following TCP and UDP streams is a simple two-click operation, but viewing SSL streams in a readable format requires a few additional steps. Because the traffic is encrypted, you are required to supply the private key associated with the server responsible for the encrypted traffic. The method you will use to retrieve this key varies depending on the server technology in use and is beyond the scope of this book, but once you have it, you will have to load it into Wireshark using the following process:

1. Access your Wireshark preferences by clicking **Edit ▸ Preferences**.
2. Expand the **Protocols** section and click the **SSL** protocol heading (shown in Figure 5-15). Click the **Edit** button next to the RSA keys list label.
3. Click the plus (+) button.
4. Supply the required information. This includes the IP address of the server responsible for the encryption, the port, the protocol, the location of the key file, and a password for the key file if one was used.
5. Restart Wireshark.

Figure 5-15: Adding SSL decryption information

Once this process is complete, you should be able to capture encrypted traffic between a client and server. Right-click an HTTPS packet and click **Follow SSL Stream** to see the clear text transcript.

The ability to view packet transcripts is one of the most commonly used analysis features in Wireshark, and you will come to rely on it to quickly determine what specific protocols are being used to do. We'll cover several additional scenarios in later chapters that rely on viewing packet transcripts.

Packet Lengths

download-slow .pcapng

The size of a single packet or group of packets can tell you a lot about a situation. Under normal circumstances, the maximum size of a frame on an Ethernet network is 1,518 bytes. When you subtract the Ethernet, IP, and TCP headers from this number, you are left with 1,460 bytes that can be used for the transmission of a layer 7 protocol header or for data. If you know the minimum requirements for packet transmission, you can begin to look at the distribution of packet lengths in a capture to make educated guesses about the makeup of the traffic. This is immensely helpful for attempting to understand the composition of large capture files. Wireshark provides the Packet Lengths dialog for you to view the distribution of packets based on length.

Let's look at an example by opening the file *download-slow.pcapng*. Once it is open, select **Statistics ▸ Packet Lengths**. The result is the Packet Lengths dialog shown in Figure 5-16.

Topic / Item	Count	Average	Min val	Max val	Rate (ms)	Percent	Burst rate	Burst start
Packet Lengths	10728	988.99	54	1460	0.0614	100%	0.2100	69.617
0-19	0	-	-	-	0.0000	0.00%	-	-
20-39	0	-	-	-	0.0000	0.00%	-	-
40-79	3587	54.07	54	66	0.0205	33.44%	0.0700	31.017
80-159	0	-	-	-	0.0000	0.00%	-	-
160-319	0	-	-	-	0.0000	0.00%	-	-
320-639	1	634.00	634	634	0.0000	0.01%	0.0100	0.178
640-1279	13	761.08	756	822	0.0001	0.12%	0.0400	0.921
1280-2559	7127	1460.00	1458	1460	0.0408	66.43%	0.1400	69.617
2560-5119	0	-	-	-	0.0000	0.00%	-	-
5120 and greater	0	-	-	-	0.0000	0.00%	-	-

Figure 5-16: The Packet Lengths dialog helps you make educated guesses about the traffic in the capture file.

Pay special attention to the row showing statistics for packets ranging from 1,280 to 2,559 bytes. Larger packets like these typically indicate the transfer of data, whereas smaller packets indicate protocol control sequences. In this case, we have a large percentage of bigger packets (66.43 percent). Without seeing the packets in the file, we can make the educated guess that the capture contains one or more transfers of data. This could be in the form of an HTTP download, an FTP upload, or any other type of network communication in which data is transferred between hosts.

Most of the remaining packets (33.44 percent) are in the 40- to 79-byte range. Packets in this range are usually TCP control packets that don't carry data. Let's consider the typical size of protocol headers. The Ethernet header is 14 bytes (plus a 4-byte CRC), the IP header is a minimum of 20 bytes, and a TCP packet with no data or options is also 20 bytes. This means that standard TCP control packets—such as SYN, ACK, RST, and FIN packets—will be around 54 bytes and fall in this range. Of course, the addition of IP or TCP options will increase this size. We'll dig into IP and TCP in Chapters 7 and 8, respectively.

Examining packet lengths is a great way to get a bird's-eye view of a large capture. If there are a lot of large packets, it may be safe to assume that data is being transferred. If the majority of packets are small, indicating that not much data is being passed, you may assume that the capture consists of protocol control commands. These are not hard-and-fast rules, but making such assumptions is helpful before diving deeper.

Graphing

Graphs are the bread and butter of analysis and one of the best ways to get a summary overview of a data set. Wireshark includes several graphing features to assist in understanding capture data, the first of which are its IO graphing capabilities.

Viewing IO Graphs

download-fast .pcapng, download -slow.pcapng, http_espn.pcapng

Wireshark's IO Graph window allows you to graph the throughput of data on a network. You can use such graphs to find spikes and lulls in data throughput, discover performance lags in individual protocols, and compare simultaneous data streams.

To view an example of the IO graph of a computer as it downloads a file from the internet, open *download-fast.pcapng*. Click any TCP packet to highlight it and then select **Statistics ▸ IO Graph**.

The IO Graph window shows a graphical view of the flow of data over time. In the example in Figure 5-17, you can see that the download this graph represents averages around 500 packets per second and stays somewhat consistent throughout its duration before tapering off at the end.

Figure 5-17: The IO graph of the fast download is mostly consistent.

Let's compare this to an example of a slower download. Leaving the current file open, open *download-slow.pcapng* in another instance of Wireshark. Bring up the IO graph of this download, and you'll see a much different story, as shown in Figure 5-18.

Figure 5-18: The IO graph of the slow download is not consistent at all.

This download has a transfer rate between 0 and 100 packets per second, and its rate is far from consistent, sometimes nearing 0 packets per second. You can see these inconsistencies more clearly if you place the IO graphs of the two files next to each other (see Figure 5-19). When comparing two graphs, pay attention to the x-and y-axis values to ensure that you're comparing apples to apples. The scale will automatically adjust based on the number of packets and/or data transmitted, which is a key difference between the two graphs in Figure 5-19. The slower download shows a scale between 0 and 100 packets per second, while the faster download's scale has a range of 0 to 700 packets per second.

Figure 5-19: Viewing multiple IO graphs side by side can be helpful in spotting variance.

The configurable options at the bottom of this window allow you to use multiple unique filters (using the same syntax as for a display or capture filter) and specify display colors for those filters. For instance, you can create filters for specific IP addresses and assign unique colors to them to view the variance in throughput for each device. Let's try that out.

Open *http_espn.pcapng*, which was captured while a device was visiting the ESPN home page. If you look at the Conversations window, you'll see that the top-talking external IP address is 205.234.218.129. From this, we can deduce that this host is likely the primary content provider we are receiving data from when visiting *espn.com*. However, there are also several other IPs participating in conversations, likely because additional content is being downloaded from external content providers and advertisers. We can show the disparity between the direct and third-party content delivery using the IO graph shown in Figure 5-20.

Figure 5-20: An IO graph showing IO of two separate devices.

The two filters applied in this chart are represented by the rows on the bottom of the IO Graph window. The filter named Top Talker shows IO only for the IP address 205.234.218.129, our primary content provider. It will graph this value in black using the stacked-bar style. The second filter, named Everything Else, will show IO for everything in the capture file except for the 205.234.218.129 address and thus includes all of the third-party content providers. This value will be graphed in red (shown here as the lighter gray) using the stacked bar. Notice that we've changed the y-axis unit to bytes per second. With these changes applied, it's very easy to see

the difference between primary and third-party content providers and just how much content is actually from a third-party source. This is a fun exercise to repeat on your frequently visited websites and a useful strategy for comparing the IO of different network hosts.

Round-Trip Time Graphing

download-fast .pcapng

Another graphing feature of Wireshark is the ability to view a plot of round-trip times for a given capture file. The *round-trip time (RTT)* is the time it takes for an acknowledgment to be received for a packet. Effectively, this is the time it took your packet to get to its destination and for the acknowledgment of that packet to be sent back to you. Analysis of RTTs is often done to find slow points or bottlenecks in communication and to determine whether there is any latency.

Let's try out this feature. Open the file *download-fast.pcapng*. View the RTT graph of this file by selecting a TCP packet and then choosing **Statistics ▸ TCP Stream Graphs ▸ Round Trip Time Graph**. The RTT graph for *download-fast.pcapng* is shown in Figure 5-21.

Figure 5-21: The RTT graph of the fast download appears mostly consistent, with only a few stray values.

Each point in the graph represents the RTT of a packet. The default view shows these values sorted by sequence number. You can click a plotted point within the graph to be taken directly to that packet in the Packet List pane.

NOTE *The RTT graph is unidirectional, so it's important to select the proper direction of the traffic you'd like to analyze. If your graph doesn't look like the one in Figure 5-21, you might need to click the Switch Direction button twice.*

It appears as though the RTT graph for the fast download has RTT values mostly under 0.05 seconds, with a few slower points between 0.10 and 0.25 seconds. Although there are quite a few higher values, the majority of the RTT values are okay, so this would be considered an acceptable RTT for a file download. When examining the RTT graph for throughput issues, you want to look for high latency times, which are indicated by multiple points plotted at higher y-axis values.

Flow Graphing

dns_recursivequery_server.pcapng

The flow graph feature is useful for visualizing connections and showing the flow of data over time, information that makes it easier to understand how devices are communicating. A flow graph contains a column-based view of a connection between hosts and organizes the traffic so you can interpret it visually.

To create a flow graph, open the file *dns_recursivequery_server.pcapng* and select **Statistics ▸ Flow Graph**. The resulting graph is shown in Figure 5-22.

Figure 5-22: The TCP flow graph allows us to visualize the connection much better.

This flow graph is a recursive DNS query, which is a DNS query that is received by one host and forwarded to another (we'll cover DNS in Chapter 9). Each vertical line in the graph represents an individual host. The flow graph is a great way to visualize back-and-forth communication between two devices or, as in this example, the relationship between the communication of multiple devices. It's also useful for understanding the normal flow of communication with protocols that you are less experienced with.

Expert Information

download-slow.pcapng

The dissectors for each protocol in Wireshark define *expert info* that can be used to alert you about particular states within packets of that protocol. These states are separated into four categories.

Chat Basic information about the communication

Note Unusual packets that may be part of normal communication

Warning Unusual packets that are most likely not part of normal communication

Error An error in a packet or the dissector interpreting it

For example, open the file *download-slow.pcapng*. Then click **Analyze** and select **Expert Information** to bring up the Expert Information window. Once there, deselect Group by summary to organize the output by severity (see Figure 5-23).

Figure 5-23: The Expert Information window shows information from the expert system programmed within the protocol dissectors.

The window has sections for each classification of information. Here there are no errors, 3 warnings, 19 notes, and 3 chats.

Most of the messages within this capture file are TCP related, simply because the expert information system has traditionally been most used with that protocol. At this time, there are 29 expert info messages configured for TCP, and they will be useful when you are troubleshooting capture files. These messages will flag an individual packet when it meets certain

criteria, as listed below. (The meaning of these messages will become clearer as we study TCP in Chapter 8 and troubleshooting slow networks in Chapter 11.)

Chat Messages

Window Update Sent by a receiver to notify a sender that the size of the TCP receive window has changed.

Note Messages

TCP Retransmission Results from packet loss. Occurs when a duplicate ACK is received or the retransmission timer of a packet expires.

Duplicate ACK When a host doesn't receive the next sequence number it is expecting, it generates a duplicate ACK of the last data it received.

Zero Window Probe Monitors the status of the TCP receive window after a zero window packet has been transmitted (covered in Chapter 11).

Keep Alive ACK Sent in response to keep-alive packets.

Zero Window Probe ACK Sent in response to zero-window-probe packets.

Window Is Full Notifies a transmitting host that the receiver's TCP receive window is full.

Warning Messages

Previous Segment Lost Indicates packet loss. Occurs when an expected sequence number in a data stream is skipped.

ACKed Lost Packet Occurs when an ACK packet is seen but the packet it is acknowledging is not.

Keep Alive Triggered when a connection keep-alive packet is seen.

Zero Window Seen when the size of the TCP receive window is reached and a zero window notice is sent out, requesting that the sender stop sending data.

Out-of-Order Utilizes sequence numbers to detect when packets are received out of sequence.

Fast Retransmission A retransmission that occurs within 20 milliseconds of a duplicate ACK.

Error Messages

No Error Messages

Although some of the features discussed in this chapter may seem as if they would be used only in obscure situations, you will probably find yourself using them more than you might expect. It's important that you familiarize yourself with these windows and options; I will be referencing them a lot in the next few chapters.

6

PACKET ANALYSIS ON THE COMMAND LINE

While many scenarios can be addressed using a GUI, in some cases, using command line tools—such as TShark or tcpdump—is necessary or preferable. Here are some situations in which a command line tool might be used instead of Wireshark:

- Wireshark provides a lot of information at once. By using a command line tool, you can limit displayed information to only pertinent data, such as a single line showing IP addresses.
- Command line tools are best suited for filtering a packet capture file and providing the results directly to another tool using Unix pipes.
- Dealing with a very large capture file can often overwhelm Wireshark because the entire file must be loaded into RAM. Stream processing of large capture files with command line tools can allow you to quickly filter the file down to the relevant packets.
- If you are dealing with a server and don't have access to a graphical tool, you may be forced to rely on command line tools.

In this chapter, I'll demonstrate the features of two common command line packet analysis tools, TShark and tcpdump. I think it's helpful to be familiar with both, but I generally find myself using TShark on Windows systems and tcpdump on Unix systems. If you exclusively use Windows, you may want to skip the parts on tcpdump.

Installing TShark

Terminal-based Wireshark, or TShark, is a packet analysis application that provides a lot of the same functionality as Wireshark but exclusively from a command line interface with no GUI. If you've installed Wireshark, then you likely have TShark as well unless you explicitly chose not to install it during Wireshark installation. You can verify that TShark is installed by following these steps:

1. Open a command prompt. Click the **Start Menu**, enter **`cmd`**, and click **Command Prompt**.
2. Browse to the directory where Wireshark is installed. If you installed it to the default location, you can go there by entering **`cd C:\Program Files\Wireshark`** in the command prompt.
3. Run TShark and print its version information by entering **`tshark -v`**. If TShark isn't installed, you'll get an error saying the command is not recognized. If TShark is installed on your system, you'll get an output with the TShark version information:

```
C:\Program Files\Wireshark>tshark -v
TShark (Wireshark) 2.0.0 (v2.0.0-0-g9a73b82 from master-2.0
--snip--
```

If you didn't install TShark and would like to use it now, you can simply rerun the Wireshark installation and make sure TShark is selected. (It is by default.)

If you'd like to immediately start learning more about TShark's capabilities, you can print the available commands with the `-h` argument. We'll cover some of these commands in this chapter.

```
C:\Program Files\Wireshark>tshark -h
```

Like Wireshark, TShark can run on multiple operating systems, but since it's not dependent on OS-specific graphics libraries, the user experience is more consistent across different OS platforms. Because of this, TShark operates very similarly on Windows, Linux, and OS X. However, there are still some differences in how TShark runs on each platform. In this book, we'll focus on running TShark on Windows because that is the primary operating system it was designed to work with.

Installing tcpdump

While Wireshark is the most popular graphical packet analysis application in the world, tcpdump is by far the most popular command line packet analysis application. Designed to work on Unix-based operating systems, tcpdump is very easy to install via popular package management applications and even comes preinstalled on many flavors of Linux.

Even though the majority of this book is Windows focused, sections on tcpdump are included for Unix users. Specifically, we'll be using Ubuntu 14.04 LTS. If you would like to use tcpdump on a Windows device, then you can download and install its Windows counterpart, WinDump, from *http://www.winpcap.org/windump/*. While the experience of tcpdump and that of WinDump aren't entirely the same, these packet analyzers function similarly. Note, however, that WinDump isn't nearly as actively maintained as tcpdump. As a result, a few newer features might be missing, and security vulnerabilities may exist. (We won't be covering WinDump in this book.)

Ubuntu doesn't come with tcpdump preinstalled, but installing it is very easy thanks to the APT package management system. To install tcpdump, follow these steps:

1. Open a terminal window and run the command **`sudo apt-get update`** to ensure that your package repositories are up-to-date with the latest package versions.
2. Run the command **`sudo apt-get install tcpdump`**.
3. You'll be asked to install a number of prerequisites that are needed to run tcpdump. Allow these installations by typing **`Y`** and pressing ENTER when prompted.
4. Once the installation has completed, run the command **`tcpdump -h`** to execute tcpdump and print its version information. You're ready to start using tcpdump if the command is successful and you see text like this in the terminal window:

```
sanders@ppa:~$ tcpdump -h
tcpdump version 4.5.1
libpcap version 1.5.3
Usage: tcpdump [-aAbdDefhHIJKlLnNOpqRStuUvxX#] [ -B size ] [ -c count ]
            [ -C file_size ] [ -E algo:secret ] [ -F file ] [ -G seconds ]
            [ -i interface ] [ -j tstamptype ] [ -M secret ]
            [ -Q metadata-filter-expression ]
            [ -r file ] [ -s snaplen ] [ -T type ] [ --version ] [ -V file ]
            [ -w file ] [ -W filecount ] [ -y datalinktype ] [ -z command ]
            [ -Z user ] [ expression ]
```

You can print all of tcpdump's available commands by invoking the `man tcpdump` command, like this:

```
sanders@ppa:~$ man tcpdump
```

We'll talk about how to use several of these commands.

Capturing and Saving Packets

The first order of business is to capture packets from the wire and display them on the screen. To start a capture in TShark, simply execute the command **tshark**. This command will start the process of capturing packets from a network interface and dumping them on screen in your terminal window, which will look something like this:

```
C:\Program Files\Wireshark>tshark
  1   0.000000 172.16.16.128 -> 74.125.95.104 TCP 66 1606      80 [SYN]
Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
  2   0.030107 74.125.95.104 -> 172.16.16.128 TCP 66 80      1606 [SYN, ACK]
Seq=0 Ack=1 Win=5720 Len=0 MSS=1406 SACK_PERM=1 WS=64
  3   0.030182 172.16.16.128 -> 74.125.95.104 TCP 54 1606      80 [ACK]
Seq=1 Ack=1 Win=16872 Len=0
  4   0.030248 172.16.16.128 -> 74.125.95.104 HTTP 681 GET / HTTP/1.1
  5   0.079026 74.125.95.104 -> 172.16.16.128 TCP 60 80      1606 [ACK]
Seq=1 Ack=628 Win=6976 Len=0
```

To start a capture in tcpdump, execute the command **tcpdump**. After you run this command, your terminal window should look something like this:

```
sanders@ppa:~$ tcpdump
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
21:18:39.618072 IP 172.16.16.128.slm-api > 74.125.95.104.http: Flags [S],
seq 2082691767, win 8192, options [mss 1460,nop,wscale 2,nop,nop,sackOK],
length 0
21:18:39.648179 IP 74.125.95.104.http > 172.16.16.128.slm-api:
Flags [S.], seq 2775577373, ack 2082691768, win 5720, options [mss
1406,nop,nop,sackOK,nop,wscale 6], length 0
21:18:39.648254 IP 172.16.16.128.slm-api > 74.125.95.104.http: Flags [.],
ack 1, win 4218, length 0
21:18:39.648320 IP 172.16.16.128.slm-api > 74.125.95.104.http: Flags [P.],
seq 1:628, ack 1, win 4218, length 627: HTTP: GET / HTTP/1.1
21:18:39.697098 IP 74.125.95.104.http > 172.16.16.128.slm-api: Flags [.],
ack 628, win 109, length 0
```

NOTE *Since administrative privileges are required to capture packets on Unix systems, you'll likely either have to execute* `tcpdump` *as the root user or use the* `sudo` *command in front of the commands listed in this book. In many cases, you'll probably be accessing your Unix-based system as a user with limited privileges. If you encounter a permissions error while following along, this is probably the reason why.*

Depending on how your system is configured, TShark or tcpdump may not default to the network interface you want to capture traffic from. If that

happens, you will need to specify it. You can list the interfaces available to TShark by using the `-D` argument, which outputs the interfaces as a numbered list, as shown here:

```
C:\Program Files\Wireshark>tshark -D
1. \Device\NPF_{1DE095C2-346D-47E6-B855-11917B74603A} (Local Area Connection*
2)
2. \Device\NPF_{1A494418-97D3-42E8-8C0B-78D79A1F7545} (Ethernet 2)
```

To use a specific interface, use the `-i` argument with the interface's assigned number from the interface list, like this:

```
C:\Program Files\Wireshark>tshark -i 1
```

This command will capture packets exclusively from the interface named Local Area Connection 2, which is assigned the number 1 in the interface list. I recommend always specifying which interface you are capturing from. It's common for virtual machine tools or VPNs to add interfaces, and you want to be certain that the packets you are capturing are coming from the correct source.

On a Linux or OS X system running tcpdump, use the `ifconfig` command to list the available interfaces:

```
sanders@ppa:~$ ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0c:29:1f:a7:55
          inet addr:172.16.16.139  Bcast:172.16.16.255  Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe1f:a755/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5119 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3088 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:876746 (876.7 KB)  TX bytes:538083 (538.0 KB)
```

Specifying the interface is also done by using the `-i` argument:

```
sanders@ppa:~$ tcpdump -i eth0
```

This command will capture packets exclusively from the eth0 interface.

Once you have everything properly configured, you can start capturing packets. If the device you're capturing traffic from is even remotely busy on the network, then you'll probably notice that lines representing individual packets are flying by rather quickly—potentially too quickly for you to read. We can remedy this by saving the packets to a file and then reading only a few of them from that file.

To save collected packets to a file in both tools, use the `-w` argument along with the name of the file. The capture will continue running until you stop it by pressing CTRL-C. The file will be saved to whatever directory the program was executed from, unless otherwise specified.

Here's an example of this command in TShark:

```
C:\Program Files\Wireshark>tshark -i 1 -w packets.pcap
```

This command will write all of the packets captured from the first interface in the interface list to *packets.pcap*.

In tcpdump, the same command would look like this:

```
sanders@ppa:~$ tcpdump -i eth0 -w packets.pcap
```

To read packets back from a saved file, use the `-r` argument along with the name of the file:

```
C:\Program Files\Wireshark>tshark -r packets.pcap
```

This command will read all the packets from *packets.pcap* onto the screen.

The tcpdump command is nearly identical:

```
sanders@ppa:~$ tcpdump -r packets.pcap
```

You may notice that if the file you are attempting to read from contains a lot of packets, you'll encounter a situation similar to the one just described, with the packets scrolling across your screen too fast for you to read. You can limit the number of packets displayed when reading from a file by using the `-c` argument.

For example, the following command will show only the first 10 packets of the capture file in TShark:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -c10
```

In tcpdump, the same argument can be used:

```
sanders@ppa:~$ tcpdump -r packets.pcap -c10
```

The `-c` argument can also be used at capture time. Executing this command will capture only the first 10 packets that are observed. They can also be saved when `-c` is combined with the `-w` argument.

Here's what this command looks like in TShark:

```
C:\Program Files\Wireshark>tshark -i 1 -w packets.pcap -c10
```

And in tcpdump:

```
sanders@ppa:~$ tcpdump -i eth0 -w packets.pcap -c10
```

Manipulating Output

A benefit of using command line tools is that the output is usually considered more carefully. A GUI typically shows you everything and it's up to you to find what you want. Command line tools typically only show the bare minimum and force you to use additional commands to dig deeper. TShark and tcpdump are no different. They both show a single line of output for each packet, requiring you to use additional commands to view information such as protocol details or individual bytes.

In the TShark output, each line represents a single packet, and the format of the line depends on the protocols used in that packet. TShark uses the same dissectors as Wireshark and analyzes packet data in the same way, so TShark output will mirror Wireshark's Packet List pane when the two are run side by side. Because TShark has dissectors for layer 7 protocols, it can provide a lot more information about packets containing headers than can tcpdump.

In tcpdump, each line also represents one packet, which is formatted differently based on the protocol being used. Since tcpdump doesn't use Wireshark's protocol dissectors, layer 7 protocol information isn't interpreted by the tool. This is one of tcpdump's biggest limitations. Instead, single-line packets are formatted based on their transport layer protocol, which is either TCP or UDP (we'll learn more about these in Chapter 8).

TCP packets use this format:

```
[Timestamp] [Layer 3 Protocol] [Source IP].[Source Port] > [Destination IP].
[Destination Port]: [TCP Flags], [TCP Sequence Number], [TCP Acknowledgement
Number], [TCP Windows Size], [Data Length]
```

While UDP packets use this format:

```
[Timestamp] [Layer 3 Protocol] [Source IP].[Source Port] > [Destination IP].
[Destination Port]: [Layer 4 Protocol], [Data Length]
```

These basic one-line summaries are great for quick analysis, but you'll eventually need to perform a deep dive into a packet. In Wireshark, you would do this by clicking a packet in the Packet List pane, which would display information in the Packet Details and Packet Bytes panes. You can access the same information on the command line using a few options.

The simplest way to gain more information about each packet is to increase the verbosity of the output.

In TShark, a capital `V` is used to increase verbosity:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -V
```

This will provide an output similar to Wireshark's Packet Details pane for packets read from the *packets.pcap* capture file. Examples of a packet with normal verbosity (a basic summary) and expanded verbosity (more detailed summaries obtained through the `-V` argument) are shown here.

First the standard output:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -c1
  1   0.000000 172.16.16.172 -> 4.2.2.1      ICMP Echo (ping) request
id=0x0001, seq=17/4352, ttl=128
```

And now a portion of the more in-depth information produced with expanded verbosity:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -V -c1
Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits) on
interface 0
    Interface id: 0 (\Device\NPF_{C30671C1-579D-4F33-9CC0-73EFFFE85A54})
    Encapsulation type: Ethernet (1)
    Arrival Time: Dec 21, 2015 12:52:43.116551000 Eastern Standard Time
    [Time shift for this packet: 0.000000000 seconds]
--snip--
```

In tcpdump, the lowercase v is used to increase verbosity. Unlike TShark, tcpdump allows multiple levels of verbosity to be displayed for each packet. You can add up to three levels of verbosity by appending additional vs, as seen here:

```
sanders@ppa:~$ tcpdump -r packets.pcap -vvv
```

An example of the same packet displayed with normal verbosity and one level of expanded verbosity is shown below. Even with full verbosity, this output isn't nearly as verbose as what TShark produces.

```
sanders@ppa:~$ tcpdump -r packets.pcap -c1
reading from file packets.pcap, link-type EN10MB (Ethernet)
13:26:25.265937 IP 172.16.16.139 > a.resolvers.level3.net: ICMP echo request,
id 1759, seq 150, length 64
sanders@ppa:~$ tcpdump -r packets.pcap -c1 -v
reading from file packets.pcap, link-type EN10MB (Ethernet)
13:26:25.265937 IP (tos 0x0, ttl 64, id 37322, offset 0, flags [DF], proto
ICMP (1), length 84)
    172.16.16.139 > a.resolvers.level3.net: ICMP echo request, id 1759, seq
150, length 64
```

The levels of verbosity available will depend on the protocol of the packet you're examining. While expanded verbosity is useful, it still doesn't show us everything there is to see. TShark and tcpdump store the entire contents of each packet, which can also be viewed in hexadecimal or ASCII form.

In TShark, you can view the hex and ASCII representation of packets by using the -x argument, which can be combined with the r argument to read and display a packet from file:

```
C:\Program Files\Wireshark>tshark -xr packets.pcap
```

This view, which is similar to Wireshark's Packet Bytes pane, is shown in Figure 6-1.

```
Administrator: Command Prompt

C:\Program Files\Wireshark>tshark -xr packets.pcap -c1
0000  c0 c1 c0 17 8c e8 f8 16 54 f8 91 ac 08 00 45 00   ........T.....E.
0010  00 3c 37 d7 00 00 80 01 40 2b ac 10 10 ac 04 02   .<7.....@+......
0020  02 01 08 00 4d 4a 00 01 00 11 61 62 63 64 65 66   ....MJ....abcdef
0030  67 68 69 6a 6b 6c 6d 6e 6f 70 71 72 73 74 75 76   ghijklmnopqrstuv
0040  77 61 62 63 64 65 66 67 68 69                     wabcdefghi
```

Figure 6-1: Viewing raw packets in hex and ASCII in TShark

In tcpdump, you can view the hex and ASCII representation by using the `-X` switch. You can also combine `-X` with the `r` argument to read from a packet file, like this:

```
sanders@ppa:~$ tcpdump -Xr packets.pcap
```

The output from this command is shown in Figure 6-2.

```
1. sanders@ppa: ~ (ssh)
sanders@ppa:~$ tcpdump -Xr packets.pcap -c1
reading from file packets.pcap, link-type EN10MB (Ethernet)
13:26:25.265937 IP 172.16.16.139 > a.resolvers.level3.net: ICMP echo request, id 1759, seq 150, length 64
        0x0000:  4500 0054 91ca 4000 4001 e640 ac10 108b  E..T..@.@..@....
        0x0010:  0402 0201 0800 ab0e 06df 0096 5144 7856  ............QDxV
        0x0020:  0000 0000 b90e 0400 0000 0000 1011 1213  ................
        0x0030:  1415 1617 1819 1a1b 1c1d 1e1f 2021 2223  .............!"#
        0x0040:  2425 2627 2829 2a2b 2c2d 2e2f 3031 3233  $%&'()*+,-./0123
        0x0050:  3435 3637                                4567
```

Figure 6-2: Viewing raw packets in hex and ASCII in tcpdump

tcpdump also lets you get a bit more granular if you need to. You can view only the hexadecimal output using the `-x` (lowercase) argument or only the ASCII output using the `-A` argument.

It's easy to become overwhelmed with data when you start experimenting with these data output options. I find it most efficient to use the least amount of information needed when doing analysis from the command line. Start by viewing packets in their default list view and use more verbose output when you narrow your analysis down to a few interesting packets. This approach will keep you from being overwhelmed with data.

Name Resolution

Like Wireshark, TShark and tcpdump will attempt to perform name resolution to convert addresses and port numbers to names. If you followed along with any of the earlier examples, you may have noticed that this occurs by default. As mentioned previously, I typically prefer to disable this functionality to prevent the possibility of my analysis generating more packets on the wire.

You can disable name resolution in TShark by using the `-n` argument. This argument, like many others, can be combined with other commands to enhance readability:

```
C:\Program Files\Wireshark>tshark -ni 1
```

You can enable or disable certain aspects of name resolution with the `-N` argument. If you use the `-N` argument, all name resolution will be disabled except for any you explicitly enable using the appropriate values. For instance, the following command will enable only transport layer (port name) resolution:

```
C:\Program Files\Wireshark>tshark -i 1 -Nt
```

You can combine multiple values. This command will enable transport layer and MAC resolution:

```
C:\Program Files\Wireshark>tshark -i 1 -Ntm
```

The following values are available when using this option:

- **m** MAC address resolution
- **n** Network address resolution
- **t** Transport layer (port name) resolution
- **N** Use external resolvers
- **C** Concurrent DNS lookups

In tcpdump, using `-n` will disable IP name resolution, and using `-nn` will disable port name resolution as well.

This argument can also be combined with other commands, like this:

```
sanders@ppa:~$ tcpdump -nni eth1
```

The following examples show a packet capture first with port resolution enabled and then with it disabled (`-n`).

```
sanders@ppa:~$ tcpdump -r tcp_ports.pcap -c1
reading from file tcp_ports.pcap, link-type EN10MB (Ethernet)
14:38:34.341715 IP 172.16.16.128.2826 > 212.58.226.142.❶http: Flags [S], seq
3691127924, win 8192, options [mss 1460,nop,wscale 2,nop,nop,sackOK], length 0
sanders@ppa:~$ tcpdump -nr tcp_ports.pcap -c1
reading from file tcp_ports.pcap, link-type EN10MB (Ethernet)
14:38:34.341715 IP 172.16.16.128.2826 > 212.58.226.142.❷80: Flags [S], seq
3691127924, win 8192, options [mss 1460,nop,wscale 2,nop,nop,sackOK], length 0
```

Both of these commands read just the first packet from the capture file *tcp_ports.pcap*. With the first command, port name resolution is on and resolves port 80 to http ❶, but with the second command, the port is just displayed by number ❷.

Applying Filters

Filtering in TShark and tcpdump is very flexible because both allow the use of BPF capture filters. TShark can also use Wireshark display filters. Just as with Wireshark, capture filters in TShark can be used only at capture time, and display filters can be used at capture time or while displaying already captured packets. We'll start by looking at TShark filters.

Capture filters can be applied using the `-f` argument, followed by the BPF syntax you wish to use in quotation marks. This command will only capture and save packets with a destination of port 80 and using the TCP protocol:

```
C:\Program Files\Wireshark>tshark -ni 1 -w packets.pcap -f "tcp port 80"
```

Display filters can be applied using the `-Y` argument, followed by the Wireshark filter syntax you wish to use in quotation marks. This can be applied at capture time like this:

```
C:\Program Files\Wireshark>tshark -ni 1 -w packets.pcap -Y "tcp.dstport == 80"
```

Display filters can be applied on already captured packets using the same argument. This command will display only packets from *packets.pcap* that match the filter:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -Y "tcp.dstport == 80"
```

With tcpdump, you specify filters inline at the end of a command within single quotes. This command will also capture and save only packets destined to TCP port 80:

```
sanders@ppa:~$ tcpdump -nni eth0 -w packets.pcap 'tcp dst port 80'
```

You can specify a filter when reading packets as well. This command will display only packets from *packets.pcap* that match the filter:

```
sanders@ppa:~$ tcpdump -r packets.pcap 'tcp dst port 80'
```

It's important to keep in mind that if the original capture file was created without a filter, then it still contains other packets; you are just limiting what is shown on the screen when reading from an existing file.

What if you have a capture file that contains a large variety of packets, but you want to filter out a subset of them and save that subset to a separate file? You can do this by combining the `-w` and `-r` arguments:

```
sanders@ppa:~$ tcpdump -r packets.pcap 'tcp dst port 80' -w http_packets.pcap
```

This command will read the file *packets.pcap*, filter out only the traffic destined for TCP port 80 (which is used for http), and write those packets to a new file called *http_packets.pcap*. This is a very common technique to

use when you want to maintain a larger source *.pcap* file but only analyze a small portion of it at a time. I frequently use this technique to whittle down very large capture files with tcpdump so that I can analyze a subset of the packets in Wireshark. Smaller capture files are much easier to wrangle.

In addition to specifying a filter inline, tcpdump allows you to reference a BPF file containing a series of filters. This is handy when you'd like to apply an extremely large or complex filter that might otherwise be unwieldy to edit and maintain inline with the tcpdump command. You can specify a filter file using the `-F` argument, like this:

```
sanders@ppa:~$ tcpdump -nni eth0 -F dns_servers.bpf
```

If your file gets too large, you might be tempted to add notes or comments to it to keep track of what each part of the filter does. Keep in mind that a BPF filter file does not allow for comments and will generate an error if anything other than a filtering statement is encountered. Since comments are very helpful for deciphering large filter files, I usually maintain two copies of every file: one for use with tcpdump that doesn't contain comments and one that contains comments for reference.

Time Display Formats in TShark

One thing that often confuses new analysts is the default timestamp used by TShark. It shows packet timestamps in relation to the start of the packet capture. There are times when such timestamping is preferable, but in many cases you may want to see the time the packet was captured, as is the default for tcpdump timestamps. You can get this same output from TShark by using the `-t` argument with the value `ad` for absolute date:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -t ad
```

Here's a comparison of the same packets as before with the default relative timestamps ❶ and absolute timestamps ❷:

```
❶ C:\Program Files\Wireshark>tshark -r packets.pcap -c2
  1   0.000000 172.16.16.172 -> 4.2.2.1      ICMP Echo (ping)
request  id=0x0001, seq=17/4352, ttl=128
  2   0.024500 4.2.2.1 -> 172.16.16.172      ICMP Echo (ping)
reply    id=0x0001, seq=17/4352, ttl=54 (request in 1)

❷ C:\Program Files\Wireshark>tshark -r packets.pcap -t ad -c2
  1 2015-12-21 12:52:43.116551 172.16.16.172 -> 4.2.2.1      ICMP Echo (ping)
request  id=0x0001, seq=17/4352, ttl=128
  2 2015-12-21 12:52:43.141051      4.2.2.1 -> 172.16.16.172 ICMP Echo (ping)
reply    id=0x0001, seq=17/4352, ttl=54 (request in 1)
```

By using the `-t` argument, you can specify any time display format you would find in Wireshark. These formats are shown in Table 6-1.

Table 6-1: Time Display Formats Available in TShark

Value	Timestamp	Example
a	Absolute time the packet was captured (in your time zone)	15:47:58.004669
ad	Absolute time the packet was captured with date (in your time zone)	2015-10-09 15:47:58.004669
d	Delta (time difference) since previous captured packet	0.000140
dd	Delta since previous displayed packet	0.000140
e	Epoch time (seconds since January 1, 1970, UTC)	1444420078.004669
r	Elapsed time between the first packet and the current packet	0.000140
u	Absolute time the packet was captured (UTC)	19:47:58.004669
ud	Absolute time the packet was captured with date (UTC)	2015-10-09 19:47:58.004669

Unfortunately, tcpdump doesn't provide this level of control for manipulating how timestamps are shown.

Summary Statistics in TShark

Another useful TShark feature (and one that sets it apart from tcpdump) is its ability to generate a subset of statistics from a capture file. These statistics mirror many of the capabilities found in Wireshark but provide easy command line access. Statistics are generated by using the `-z` argument and specifying the name of the output you would like to generate. You can view a full listing of available statistics by using this command:

```
C:\Program Files\Wireshark>tshark -z help
```

Many of the features we've already covered are available using the `-z` argument. They include the ability to output endpoint and conversation statistics using this command:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -z conv,ip
```

This command prints a table of statistics with information about the IP conversations in the file *packets.pcap*, as shown in Figure 6-3.

You can also use this argument to view protocol-specific information. As shown in Figure 6-4, you can use the `http,tree` option to see a breakdown of HTTP requests and responses in table form.

```
C:\Program Files\Wireshark>tshark -r packets.pcap -z http,tree
```

```
Administrator: Command Prompt
C:\Program Files\Wireshark>tshark -r packets.pcap -z conv,ip
  1   0.000000 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=17/4352, ttl=128
  2   0.024500      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=17/4352, ttl=54 (request in 1)
  3   1.004292 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=18/4608, ttl=128
  4   1.036842      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=18/4608, ttl=54 (request in 3)
  5   2.015805 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=19/4864, ttl=128
  6   2.045880      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=19/4864, ttl=54 (request in 5)
  7   3.026818 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=20/5120, ttl=128
  8   3.051049      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=20/5120, ttl=54 (request in 7)
  9   4.479931 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=21/5376, ttl=128
 10   4.504257      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=21/5376, ttl=54 (request in 9)
 11   5.483684 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=22/5632, ttl=128
 12   5.508045      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=22/5632, ttl=54 (request in 11)
 13   6.492711 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=23/5888, ttl=128
 14   6.517448      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=23/5888, ttl=54 (request in 13)
 15   7.504493 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=24/6144, ttl=128
 16   7.528843      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=24/6144, ttl=54 (request in 15)
 17  10.880478 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=25/6400, ttl=128
 18  10.904451      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=25/6400, ttl=54 (request in 17)
 19  11.885254 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=26/6656, ttl=128
 20  11.909634      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=26/6656, ttl=54 (request in 19)
 21  12.895091 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=27/6912, ttl=128
 22  12.920183      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=27/6912, ttl=54 (request in 21)
 23  13.904898 172.16.16.172 -> 4.2.2.1      ICMP 74 Echo (ping) request  id=0x0001, seq=28/7168, ttl=128
 24  13.929263      4.2.2.1 -> 172.16.16.172 ICMP 74 Echo (ping) reply    id=0x0001, seq=28/7168, ttl=54 (request in 23)
================================================================================
IPv4 Conversations
Filter:<No Filter>
                                               |       <-      | |       ->      | |     Total     |    Relative    |   Duration   |
                                               | Frames  Bytes | | Frames  Bytes | | Frames  Bytes |     Start      |              |
4.2.2.1               <-> 172.16.16.172            12       888      12       888      24      1776     0.000000000        13.9293
================================================================================
```

Figure 6-3: Using TShark to view conversation statistics

```
Administrator: Command Prompt
==========================================================================================================================
HTTP/Packet Counter:
Topic / Item            Count         Average       Min val       Max val       Rate (ms)     Percent       Burst rate    Burst start
--------------------------------------------------------------------------------------------------------------------------
Total HTTP Packets      1761                                                    0.0203        100%          0.4200        6.651
 HTTP Request Packets   894                                                     0.0103        50.77%        0.2100        6.651
  GET                   871                                                     0.0100        97.43%        0.2100        6.651
  NOTIFY                21                                                      0.0002        2.35%         0.1100        26.951
  SEARCH                2                                                       0.0000        0.22%         0.0100        0.293
 HTTP Response Packets  867                                                     0.0100        49.23%        0.2300        6.886
  3xx: Redirection      479                                                     0.0055        55.25%        0.2200        6.886
   304 Not Modified     457                                                     0.0053        95.41%        0.2200        6.886
   302 Found            22                                                      0.0003        4.59%         0.0500        17.814
  2xx: Success          387                                                     0.0045        44.64%        0.1400        40.264
   200 OK               374                                                     0.0043        96.64%        0.1400        40.264
   204 No Content       13                                                      0.0001        3.36%         0.0200        13.054
  4xx: Client Error     1                                                       0.0000        0.12%         0.0100        22.598
   404 Not Found        1                                                       0.0000        100.00%       0.0100        22.598
  ???: broken           0                                                       0.0000        0.00%         -             -
  5xx: Server Error     0                                                       0.0000        0.00%         -             -
  1xx: Informational    0                                                       0.0000        0.00%         -             -
 Other HTTP Packets     0                                                       0.0000        0.00%         -             -

--------------------------------------------------------------------------------------------------------------------------
```

Figure 6-4: Using TShark to view HTTP request and response statistics

Another useful feature is the ability to view reassembled stream output, similar to what we did earlier by right-clicking packets in Wireshark and choosing the Follow TCP Stream option. To get this output, we have to use the `follow` option and specify the type of stream, the output mode, and which stream we want to display. You can identify a stream with the number assigned to it in the leftmost column when outputting conversation statistics (as seen in Figure 6-3). A command might look like this:

```
C:\Program Files\Wireshark>tshark -r http_google.pcap -z follow,tcp,ascii,0
```

This command will print TCP stream 0 to the screen in ASCII format from the file *http_google.pcap*. The output for this command looks like this:

```
C:\Program Files\Wireshark>tshark -r http_google.pcap -z

--snip--
===================================================================
Follow: tcp,ascii
Filter: tcp.stream eq 0
Node 0: 172.16.16.128:1606
Node 1: 74.125.95.104:80
627
GET / HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.7)
Gecko/20091221 Firefox/3.5.7
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PREF=ID=257913a938e6c248:U=267c896b5f39fb0b:FF=4:LD=e
n:NR=10:TM=1260730654:LM=1265479336:GM=1:S=h1UBGonTuWU3D23L;
NID=31=Z-nhwMjUP63e0tYMTp-3T1igMSPnNS1eM1kN1_DUrnO2zW1cPM4JE3AJec9b_
vG-YFibFXszOApfbhBA1BOX4dKx4L8ZDdeiKwqekgP5_kzELtC2mUHx7RHx3PIttcuZ

        1406
HTTP/1.1 200 OK
Date: Tue, 09 Feb 2010 01:18:37 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 4633
X-XSS-Protection: 0
```

You can also specify which stream you'd like to view by providing the address details. For example, the following command will retrieve a UDP stream for the specified endpoints and ports:

```
C:\Program Files\Wireshark>tshark -r packets.pcap -z follow,udp,ascii,192.168.
1.5:23429❶,4.2.2.1:53❷
```

This command will print the UDP stream for the endpoints 192.168.1.5 on port 23429 ❶ and 4.2.2.1 on port 53 ❷ from *packets.pcap*.

Here are some of my favorite statistical options:

`ip_hosts,tree` Displays every IP address in a capture, along with the rate and percentage of traffic each address is responsible for

`io,phs` Displays a protocol hierarchy showing all protocols found within the capture file

`http,tree` Displays statistics related to HTTP requests and responses

`http_req,tree` Displays statistics for every HTTP request

`smb,srt` Displays statistics related to SMB commands for analyzing Windows communication

`endpoints,wlan` Displays wireless endpoints

`expert` Displays expert information (chats, errors, and so on) from the capture

There are a lot of useful options available using the `-z` argument. It would take far too many pages to cover them all here, but if you plan to use TShark frequently, you should invest time in reviewing the official documentation to learn more about everything that is available. You can find that documentation here: *https://www.wireshark.org/docs/man-pages/tshark.html*.

Comparing TShark and tcpdump

Both command line packet analysis applications we've examined in this chapter are well suited to their respective tasks, and either of them will allow you to accomplish whatever task is at hand with varying degrees of effort. There are a few differences worth highlighting so you can choose the best tool for the job:

Operating system tcpdump is only available for Unix-based operating systems, while TShark can function on Windows and Unix-based systems.

Protocol support Both tools support common layer 3 and 4 protocols, but tcpdump has limited layer 7 protocol support. TShark provides a rich level of layer 7 protocol support because it has access to Wireshark's protocol dissectors.

Analysis features Both tools rely heavily on human analysis to produce meaningful results, but TShark also provides a robust set of analytical and statistical features, similar to those in Wireshark, that can aid analysis when a GUI isn't available.

Tool availability and personal preference are usually the ultimate deciders of which application to use. Fortunately, the tools are similar enough that learning one will inherently teach you something about the other, making you more versatile and increasing the size of your tool kit.

7

NETWORK LAYER PROTOCOLS

Whether you're troubleshooting latency issues, identifying malfunctioning applications, or zeroing in on security threats in order to spot abnormal traffic, you must first understand normal traffic. In the next couple of chapters, you'll learn how normal network traffic works at the packet level as we journey from the bottom of the OSI model all the way to the top. Each protocol section has at least one associated capture file, which you can download and work with directly.

In this chapter, we'll specifically focus on the network layer protocols that are the workhorses of network communication: ARP, IPv4, IPv6, ICMP, and ICMPv6.

The next three chapters on network protocols are arguably the most important chapters in this book. Skipping this discussion would be like making Thanksgiving dinner without preheating the oven. Even if you already have a good grasp of how each protocol functions, give these chapters at least a quick read in order to review the packet structure of each.

Address Resolution Protocol (ARP)

Both logical and physical addresses are used for communication on a network. Logical addresses allow for communication among multiple networks and indirectly connected devices. Physical addresses facilitate communication on a single network segment for devices that are directly connected to each other with a switch. In most cases, these two types of addressing must work together in order for communication to occur.

Consider a scenario in which you wish to communicate with a device on your network. This device may be a server of some sort or just another workstation you need to share files with. The application you are using to initiate the communication is already aware of the remote host's IP address (via DNS, covered in Chapter 9), meaning the system should have all it needs to build the layer 3 through 7 information of the packet it wants to transmit. The only piece of information it needs at this point is the layer 2 data link information containing the MAC address of the target host.

MAC addresses are needed because a switch that interconnects devices on a network uses a *Content Addressable Memory (CAM) table*, which lists the MAC addresses of all devices plugged into each of its ports. When the switch receives traffic destined for a particular MAC address, it uses this table to know which port to send the traffic through. If the destination MAC address is unknown, the transmitting device will first check for the address in its cache; if the address isn't there, then it must be resolved through additional communication on the network.

The resolution process that TCP/IP networking (with IPv4) uses to resolve an IP address to a MAC address is called the *Address Resolution Protocol (ARP)*, which is defined in RFC 826. The ARP resolution process uses only two packets: an ARP request and an ARP response (see Figure 7-1).

NOTE *An RFC, or Request for Comments, is a technical publication from the Internet Engineering Task Force (IETF) and Internet Society (ISOC) and is the mechanism used to define the implementation standards for protocols. You can search for RFC documentation at the RFC Editor home page,* http://www.rfc-editor.org/.

The transmitting computer sends out an ARP request that basically says, "Howdy, everybody. My IP address is 192.168.0.101, and my MAC address is f2:f2:f2:f2:f2:f2. I need to send something to whoever has the IP address 192.168.0.1, but I don't know the hardware address. Will whoever has this IP address please respond with your MAC address?"

This packet is broadcast to every device on the network segment. Any device that doesn't have this IP address simply discards the packet. The device that does have the address sends an ARP reply with an answer such as "Hey, transmitting device, I'm the one you're looking for with the IP address 192.168.0.1. My MAC address is 02:f2:02:f2:02:f2."

Once this resolution process is completed, the transmitting device updates its cache with the MAC-to-IP address association of the receiving device and can begin sending data.

Figure 7-1: The ARP resolution process

NOTE *You can view the ARP table of a Windows host by typing* `arp -a` *from a command prompt.*

Seeing this process in action will help you understand how it works. But before we look at some examples, let's examine the ARP packet header.

ARP Packet Structure

As shown in Figure 7-2, the ARP header includes the following fields:

Hardware Type The layer 2 type used—in most cases, this is Ethernet (type 1)

Protocol Type The higher-layer protocol for which the ARP request is being used

Hardware Address Length The length (in octets/bytes) of the hardware address in use (6 for Ethernet)

Protocol Address Length The length (in octets/bytes) of the logical address of the specified protocol type

Operation The function of the ARP packet: either 1 for a request or 2 for a reply

Address Resolution Protocol (ARP)					
Offsets	Octet	0	1	3	4
Octet	Bit	0–7	8–15	0–7	8–15
0	0	Hardware Type		Protocol Type	
4	32	Hardware Address Length	Protocol Address Length	Operation	
8	64	Sender Hardware Address			
12	96	Sender Hardware Address		Sender Protocol Address	
16	128	Sender Protocol Address		Target Hardware Address	
20	160	Target Hardware Address			
24+	192+	Target Protocol Address			

Figure 7-2: The ARP packet structure

Sender Hardware Address The hardware address of the sender

Sender Protocol Address The sender's upper-layer protocol address

Target Hardware Address The intended receiver's hardware address (all zeroes in ARP requests)

Target Protocol Address The intended receiver's upper-layer protocol address

Now open the file *arp_resolution.pcapng* to see this resolution process in action. We'll focus on each packet individually as we walk through this process.

Packet 1: ARP Request

arp_resolution .pcapng

The first packet is the ARP request, as shown in Figure 7-3. We can confirm that this packet is a true broadcast packet by examining the Ethernet header in Wireshark's Packet Details pane. The packet's destination address is ff:ff:ff:ff:ff:ff ❶. This is the Ethernet broadcast address, and anything sent to it will be broadcast to all devices on the current network segment. The source address of this packet in the Ethernet header is listed as our MAC address ❷.

Given this structure, we can discern that this is indeed an ARP request on an Ethernet network using IPv4. The sender's IP address (192.168.0.114) and MAC address (00:16:ce:6e:8b:24) are listed ❸, as is the IP address of the target (192.168.0.1) ❺. The MAC address of the target—the information we are trying to get—is unknown, so the target MAC is listed as 00:00:00:00:00:00 ❹.

```
Wireshark · Packet 1 · arp_resolution
> Frame 1: 42 bytes on wire (336 bits), 42 bytes captured (336 bits)
v Ethernet II, Src: HonHaiPr_6e:8b:24 (00:16:ce:6e:8b:24), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
  > Destination: Broadcast (ff:ff:ff:ff:ff:ff) ❶
  > Source: HonHaiPr_6e:8b:24 (00:16:ce:6e:8b:24) ❷
    Type: ARP (0x0806)
v Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
❸   Sender MAC address: HonHaiPr_6e:8b:24 (00:16:ce:6e:8b:24)
    Sender IP address: 192.168.0.114
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00) ❹
    Target IP address: 192.168.0.1 ❺

No.: 1 · Time: 0.000000 · Source: HonHaiPr_6e:8b:24 · Destination: Broadcast · Protocol: ARP · Length: 42 · Info: Who has 192.168.0.1? Tell 192.168.0.114
                                                                                    Close   Help
```

Figure 7-3: An ARP request packet

Packet 2: ARP Response

In the response to the initial request (see Figure 7-4), the Ethernet header now has a destination address of the source MAC address from the first packet. The ARP header looks similar to that of the ARP request, with a few changes:

- The packet's operation code (opcode) is now 0x0002 ❶, indicating a reply rather than a request.
- The addressing information is reversed—the sender MAC address and IP address are now the target MAC address and IP address ❸.
- Most important, all the information is present, meaning we now have the MAC address (00:13:46:0b:22:ba) ❷ of our host at 192.168.0.1.

```
Wireshark · Packet 2 · arp_resolution
> Frame 2: 46 bytes on wire (368 bits), 46 bytes captured (368 bits)
v Ethernet II, Src: D-LinkCo_0b:22:ba (00:13:46:0b:22:ba), Dst: HonHaiPr_6e:8b:24 (00:16:ce:6e:8b:24)
  > Destination: HonHaiPr_6e:8b:24 (00:16:ce:6e:8b:24)
  > Source: D-LinkCo_0b:22:ba (00:13:46:0b:22:ba)
    Type: ARP (0x0806)
    Trailer: c0a80072
v Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IPv4 (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2) ❶
    Sender MAC address: D-LinkCo_0b:22:ba (00:13:46:0b:22:ba) ❷
    Sender IP address: 192.168.0.1
❸   Target MAC address: HonHaiPr_6e:8b:24 (00:16:ce:6e:8b:24)
    Target IP address: 192.168.0.114

No.: 2 · Time: 0.004081 · Source: D-LinkCo_0b:22:ba · Destination: HonHaiPr_6e:8b:24 · Protocol: ARP · Length: 46 · Info: 192.168.0.1 is at 00:13:46:0b:22:ba
                                                                                    Close   Help
```

Figure 7-4: An ARP reply packet

Gratuitous ARP

arp_gratuitous .pcapng

Where I come from, when something is done "gratuitously," the word usually carries a negative connotation. A *gratuitous ARP*, however, is a good thing.

In many cases, a device's IP address can change. When this happens, the IP-to-MAC address mappings that hosts on the network have in their caches will be invalid. To prevent this from causing communication errors, a gratuitous ARP packet is transmitted on the network to force any device that receives it to update its cache with the new IP-to-MAC address mapping (see Figure 7-5).

Figure 7-5: The gratuitous ARP process

A few different scenarios can spawn a gratuitous ARP packet. One of the most common is the changing of an IP address. Open the capture file *arp_gratuitous.pcapng*, and you'll see this in action. This file contains only a single packet (see Figure 7-6) because that's all that's involved in gratuitous ARP.

Figure 7-6: A gratuitous ARP packet

Examining the Ethernet header, you can see that this packet is sent as a broadcast so that all hosts on the network receive it ❶. The ARP header looks like an ARP request, except that the sender IP address ❷ and the target IP address ❸ are the same. When received by other hosts on the network, this packet will cause them to update their ARP tables with the new IP-to-MAC address association. Because this ARP packet is unsolicited but results in a client updating its ARP cache, the packet is considered gratuitous.

You'll notice gratuitous ARP packets in a few situations. As mentioned, changing a device's IP address will generate a gratuitous packet. Also, some operating systems will perform a gratuitous ARP on startup. Additionally, some systems use gratuitous ARP packets to support load balancing.

Internet Protocol (IP)

The primary purpose of protocols at layer 3 of the OSI model is to allow for communication between networks. As you just saw, MAC addresses are used for communication on a single network at layer 2. In much the same fashion, layer 3 is responsible for addresses used in internetwork communication. A few protocols can do this, but the most common is the *Internet Protocol (IP)*, which currently has two versions in use—IP version 4 and IP version 6. We'll start by examining IP version 4 (IPv4), which is defined in RFC 791.

Internet Protocol Version 4 (IPv4)

To understand the functionality of IPv4, you need to know how traffic flows between networks. IPv4 is the workhorse of the communication process and is ultimately responsible for carrying data between devices, regardless of where the communication endpoints are located.

A simple network in which all devices are connected via hubs or switches is called a *local area network (LAN)*. When you want to connect two LANs, you can do so with a router. Complex networks can consist of thousands of LANs connected through thousands of routers worldwide. The internet itself is a collection of millions of LANs and routers.

IPv4 Addresses

IPv4 addresses are 32-bit assigned numbers used to uniquely identify devices connected to a network. It's a bit much to expect someone to remember a sequence of ones and zeros that is 32 characters long, so IP addresses are written in *dotted-quad* (or *dotted-decimal*) *notation*.

In dotted-quad notation, each of the four sets of ones and zeros that make up an IP address is converted to base 10 and represented as a number between 0 and 255 in the format *A.B.C.D* (see Figure 7-7). For example, consider the IP address 11000000 10101000 00000000 00000001. This value is obviously a bit much to remember or notate. Fortunately, using dotted-quad notation, we can represent it as 192.168.0.1.

Figure 7-7: Dotted-quad IPv4 address notation

An IP address consists of two parts: a *network portion* and a *host portion*. The network portion identifies the LAN the device is connected to, and the host portion identifies the device itself on that network. The determination of which part of the IP address belongs to the network or host portion is not always the same. This information is communicated by another set of addressing information called the *network mask (netmask)* or sometimes referred to as a *subnet mask*.

NOTE *In this book, when we refer to an IP address, we will always be referring to an IPv4 address. Later in this chapter, we will look at IP version 6, which uses a different set of rules for addressing. Whenever we refer to an IPv6 address, it will be explicitly labeled as such.*

The netmask identifies which part of the IP address belongs to the network portion and which part belongs to the host portion. The netmask number is also 32 bits long, and every bit that is set to a 1 identifies the part of the IP address that is reserved for the network portion. The remaining bits are set to 0 to identify the host portion.

For example, consider the IP address 10.10.1.22, represented in binary as 00001010 00001010 00000001 00010110. To determine the allocation of each section of the IP address, we can apply our netmask. In this case, our netmask is 11111111 11111111 00000000 00000000. This means that the first half of the IP address (10.10 or 00001010 00001010) is reserved for the network portion, and the last half of the IP address (.1.22 or 00000001 00010110) identifies the individual host on this network, as shown in Figure 7-8.

Figure 7-8: The netmask determines the allocation of the bits in an IP address.

As indicated in Figure 7-8, netmasks can also be written in dotted-quad notation. For example, the netmask 11111111 11111111 00000000 00000000 is written as 255.255.0.0.

IP addresses and netmasks are commonly written in *Classless Inter-Domain Routing (CIDR) notation*. In this form, an IP address is written in full, followed by a forward slash (/) and the number of bits that represent the network portion of the IP address. For example, an IP address of 10.10.1.22 and a netmask of 255.255.0.0 would be written in CIDR notation as 10.10.1.22/16.

IPv4 Packet Structure

The source and destination IP addresses are the crucial components of the IPv4 packet header, but that's not all of the IP information you'll find in a packet. The IP header is quite complex compared to the ARP packet we just examined; it includes a lot of extra functionality that helps IP do its job.

As shown in Figure 7-9, the IPv4 header has the following fields:

Version The version of IP being used (this will always be 4 for IPv4).

Header Length The length of the IP header.

Type of Service A precedence flag and type of service flag, which are used by routers to prioritize traffic.

Total Length The length of the IP header and the data included in the packet.

Identification A unique identification number used to identify a packet or sequence of fragmented packets.

Flags Used to identify whether a packet is part of a sequence of fragmented packets.

Fragment Offset If a packet is a fragment, the value of this field is used to reassemble the packets in the correct order.

Time to Live Defines the lifetime of the packet, measured in hops or seconds through routers.

Protocol Identifies the transport layer header that encapsulates the IPv4 header.

Header Checksum An error-detection mechanism used to verify that the contents of the IP header are not damaged or corrupted.

Source IP Address The IP address of the host that sent the packet.

Destination IP Address The IP address of the packet's destination.

Options Reserved for additional IP options. It includes options for source routing and timestamps.

Data The actual data being transmitted with IP.

Internet Protocol Version 4 (IPv4)							
Offsets	Octet	0		1	2		3
Octet	Bit	0–3	4–7	8–15	16–18	19–23	24–31
0	0	Version	Header Length	Type of Service	Total Length		
4	32	Identification			Flags	Fragment Offset	
8	64	Time to Live		Protocol	Header Checksum		
12	96	Source IP Address					
16	128	Destination IP Address					
20	160	Options					
24+	192+	Data					

Figure 7-9: The IPv4 packet structure

Time to Live

ip_ttl_source.pcapng
ip_ttl_dest.pcapng

The *Time to Live (TTL)* value defines a period of time that can elapse or a maximum number of routers a packet can traverse before the packet is discarded for IPv4. A TTL is defined when a packet is created and generally is decremented by 1 every time the packet is forwarded by a router. For example, if a packet has a TTL of 2, the first router it reaches will decrement the TTL to 1 and forward it to the second router. This router will then decrement the TTL to zero, and if the final destination of the packet is not on that network, the packet will be discarded (see Figure 7-10).

Figure 7-10: The TTL of a packet decreases every time it traverses a router.

Why is the TTL value important? Typically, we are concerned about the lifetime of a packet only in terms of the time that it takes to travel from its source to its destination. However, consider a packet that must travel to a host across the internet while traversing dozens of routers. At some point in that packet's path, it could encounter a misconfigured router and lose the path to its final destination. In such a case, the router could do a number of things, one of which could result in the packet's being forwarded around a network in a never-ending loop.

An infinite loop can cause all sorts of issues, but it typically results in the crash of a program or an entire operating system. Theoretically, the same thing could occur with packets on a network. The packets would keep looping between routers. As the number of looping packets increased, the available bandwidth on the network would deplete until a denial of service condition occurred. To prevent this, TTL was created.

Let's look at an example of this in Wireshark. The file *ip_ttl_source.pcapng* contains two ICMP packets. ICMP (discussed later in this chapter) uses IP to deliver packets, as we can see by expanding the IP header section in the Packet Details pane (see Figure 7-11).

Figure 7-11: The IP header of the source packet

You can see that the version of IP being used is version 4 ❶, the IP header length is 20 bytes ❷, the total length of the header and payload is 60 bytes ❸, and the value of the TTL field is 128 ❹.

The primary purpose of an ICMP ping is to test communication between devices. Data is sent from one host to another as a request, and the receiving host should send that data back as a reply. In this file, we have one device with the address of 10.10.0.3 ❺ sending an ICMP request to a device with the address 192.168.0.128 ❻. This initial capture file was created at the source host, 10.10.0.3.

Now open the file *ip_ttl_dest.pcapng*. In this file, the data was captured at the destination host, 192.168.0.128. Expand the IP header of the first packet in this capture to examine its TTL value (see Figure 7-12).

You should immediately notice that the TTL value is 127 ❶, 1 less than the original TTL of 128. Without even knowing the architecture of the network, we can conclude that one router separates these devices and thus the passage through that router reduced the TTL value by 1.

Figure 7-12: The IP header shows us that the TTL has been decremented by 1.

IP Fragmentation

ip_frag_source .pcapng

Packet fragmentation is a feature of IP that permits reliable delivery of data across varying types of networks by splitting a data stream into smaller fragments.

The fragmentation of a packet is based on the *maximum transmission unit (MTU)* size of the layer 2 data link protocol in use and the configuration of the devices using this layer 2 protocol. In most cases, the layer 2 data link protocol in use is Ethernet. Ethernet has a default MTU of 1,500, which means that the maximum packet size that can be transmitted over an Ethernet network is 1,500 bytes (not including the 14-byte Ethernet header itself).

NOTE *Although there are standard MTU settings, the MTU of a device can be reconfigured manually in most cases. An MTU setting is assigned on a per-interface basis and can be modified on Windows and Linux systems, as well as on the interfaces of managed routers.*

When a device prepares to transmit an IP packet, it determines whether it must fragment the packet by comparing the packet's data size to the MTU of the network interface from which the packet will be transmitted. If the data size is greater than the MTU, the packet will be fragmented. Fragmenting a packet involves the following steps:

1. The device splits the data into the number of packets required for successful data transmission.
2. The Total Length field of each IP header is set to the segment size of each fragment.

3. The More fragments flag is set to 1 on all packets in the data stream, except for the last one.
4. The Fragment offset field is set in the IP header of the fragments.
5. The packets are transmitted.

The file *ip_frag_source.pcapng* was taken from a computer with the address 10.10.0.3, transmitting a ping request to a device with the address 192.168.0.128. Notice that the Info column of the Packet List pane lists two fragmented IP packets, followed by the ICMP (ping) request.

Begin by examining the IP header of packet 1 (see Figure 7-13).

Figure 7-13: More fragments and Fragment offset values can indicate a fragmented packet.

You can see that this packet is part of a fragment based on the More fragments and Fragment offset fields. Packets that are fragments will either have a positive Fragment offset value or have the More fragments flag set. In the first packet, the More fragments flag is set ❶, indicating that the receiving device should expect to receive another packet in this sequence. The Fragment offset is set to 0 ❷, indicating that this packet is the first in a series of fragments.

The IP header of the second packet (see Figure 7-14) also has the More fragments flag set ❶, but in this case, the Fragment offset value is 1480 ❷. This is indicative of the 1,500-byte MTU, minus 20 bytes for the IP header.

The third packet (see Figure 7-15) does not have the More fragments flag set ❷, which marks it as the last fragment in the data stream, and the Fragment offset is set to 2960 ❸, the result of 1480 + (1500 – 20). These fragments can all be identified as part of the same series of data because they have the same values in the Identification field of the IP header ❶.

```
Wireshark · Packet 2 · ip_frag_source
> Frame 2: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits)
> Ethernet II, Src: HewlettP_bf:91:ee (00:25:b3:bf:91:ee), Dst: Cisco-Li_66:71:95 (00:21:29:66:71:95)
v Internet Protocol Version 4, Src: 10.10.0.3, Dst: 192.168.0.128
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 1500
    Identification: 0x7474 (29812)
  v Flags: 0x01 (More Fragments)
      0... .... = Reserved bit: Not set
      .0.. .... = Don't fragment: Not set
      ..1. .... = More fragments: Set ❶
    Fragment offset: 1480 ❷
    Time to live: 128
    Protocol: ICMP (1)
  > Header checksum: 0x0000 [validation disabled]
    Source: 10.10.0.3
    Destination: 192.168.0.128
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
    Reassembled IPv4 in frame: 3
> Data (1480 bytes)
No.: 2 · Time: 0.000010 · Source: 10.10.0.3 · Destination: 192.168.0.128 · Protocol: IPv4 … 1514 · Info: Fragmented IP protocol (proto=ICMP 1, off=1480, ID=7474) [Reassembled i
                                                                                    Close   Help
```

Figure 7-14: The Fragment offset value increases based on the size of the packets.

```
Wireshark · Packet 3 · ip_frag_source
> Frame 3: 582 bytes on wire (4656 bits), 582 bytes captured (4656 bits)
> Ethernet II, Src: HewlettP_bf:91:ee (00:25:b3:bf:91:ee), Dst: Cisco-Li_66:71:95 (00:21:29:66:71:95)
v Internet Protocol Version 4, Src: 10.10.0.3, Dst: 192.168.0.128
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 568
    Identification: 0x7474 (29812) ❶
  v Flags: 0x00
      0... .... = Reserved bit: Not set
      .0.. .... = Don't fragment: Not set
      ..0. .... = More fragments: Not set ❷
    Fragment offset: 2960 ❸
    Time to live: 128
    Protocol: ICMP (1)
  > Header checksum: 0x0000 [validation disabled]
    Source: 10.10.0.3
    Destination: 192.168.0.128
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
  > [3 IPv4 Fragments (3508 bytes): #1(1480), #2(1480), #3(548)]
> Internet Control Message Protocol
No.: 3 · Time: 0.000013 · Source: 10.10.0.3 · Destination: 192.168.0.128 · Protocol: ICMP · Length: 582 · Info: Echo (ping) request id=0x0001, seq=57/14592, ttl=128 (reply in
                                                                                    Close   Help
```

Figure 7-15: More fragments is not set, indicating that this fragment is the last.

While it isn't as common to see fragmented packets on a network as it used to be, understanding why packets are fragmented is useful so that when you do encounter them, you can diagnose issues or spot missing fragments.

Internet Protocol Version 6 (IPv6)

When the IPv4 specification was written, nobody had any idea that we would eventually have the number of internet-connected devices that exist today. The maximum IPv4 addressable space was limited to just south of 4.3 billion addresses. The actual amount of addressable space shrinks even further when you subtract ranges reserved for special uses such as testing, broadcast traffic, and RFC1918 internal addresses. While several efforts were made to delay the exhaustion of IPv4 addresses, ultimately the only way to address this limitation was to develop a new version of the IP specification.

Thus, the IPv6 specification was created, with its first version released in 1998 as RFC 2460. This version provided several performance enhancements, including a much larger address space. In this section, we'll look at the IPv6 packet structure and discuss how IPv6 communications differ from those of its predecessor.

IPv6 Addresses

IPv4 addresses were limited to 32 bits, a length that provided an addressable space measured in the billions. IPv6 addresses are 128 bit, providing an addressable space measured in undecillions (a trillion trillion trillion). That's quite an upgrade!

Since IPv6 addresses are 128 bits, they are unwieldy to manage in binary form. Most of the time, an IPv6 address is written in eight groups of 2 bytes in hexadecimal notation, with each group separated by a colon. For example, a very simple IPv6 address looks like this:

```
1111:aaaa:2222:bbbb:3333:cccc:4444:dddd
```

Your first thought is probably the same one many have who are used to remembering IPv4 addresses: IPv6 addresses are virtually impossible to memorize. That is an unfortunate trade-off for a much larger address space.

One feature of IPv6 address notation that will help in some cases is that some groups of zeroes can be collapsed. For example, consider the following IPv6 address:

```
1111:0000:2222:0000:3333:4444:5555:6666
```

You can collapse the grouping containing the zeroes completely so it isn't visible, like this:

```
1111::2222:0000:3333:4444:5555:6666
```

However, you can only collapse a single group of zeroes, so the following address would be invalid:

```
1111::2222::3333:4444:5555:6666
```

Another consideration is that leading zeroes can be dropped from IPv6 addresses. Consider this example in which there are zeroes in front of the fourth, fifth, and six groups:

```
1111:0000:2222:0333:0044:0005:ffff:ffff
```

You could represent the address more efficiently like this:

```
1111::2222:333:44:5:ffff:ffff
```

This isn't quite as easy to use as an IPv4 address, but it's a lot easier to deal with than the longer notation.

An IPv6 address has a network portion and a host portion, often called a *network prefix* and *interface identifier*, respectively. The distribution of these fields varies depending on the classification of the IPv6 communication. IPv6 traffic is broken down into three classifications: unicast, multicast, or anycast. In most cases, you'll probably be working with link-local unicast traffic, which is communication from one device to another inside a network. The format of a link-local unicast IPv6 address is shown in Figure 7-16.

Figure 7-16: The parts of an IPv6 link-local unicast address

Link-local addresses are used when communication is intended for another device within the same network. A link-local address can be identified by having its most significant 10 bits set to 1111111010 and the next 54 bits set to all zeroes. Thus, you can spot a link-local address when the first half is fe80:0000:0000:0000.

The second half of a link-local IPv6 address is the interface ID portion, which uniquely identifies a network interface on an endpoint host. On Ethernet networks, this can be based on the MAC address of the interface. However, a MAC address is only 48 bits. To fill up the entire 64-bit space, the MAC address is cut in half, and the value 0xfffe is added between each half as padding to create a unique identifier. Lastly, the seventh bit of the first byte is inverted. That's a bit complex, but consider the interface ID in Figure 7-17. The original MAC address for the device represented by this ID was 78:31:c1:cb:b2:56. The bytes 0xfffe were added in the middle, and flipping the seventh bit of the first byte changed the *8* to an *a*.

Figure 7-17: The interface ID utilizes an interface MAC address and padding.

IPv6 addresses can be represented with CIDR notation just like IPv4 addresses. In this example, 64 bits of addressable space are represented with a link-local address:

```
fe80:0000:0000:0000:/64
```

The composition of an IPv6 address changes when it is used with global unicast traffic that is routed over the public internet (see Figure 7-18). When used in this manner, a global unicast is identified by having its first 3 bits set to 001, followed by a 45-bit global routing prefix. The global routing prefix, which is assigned to organizations by the Internet Assigned Numbers Authority (IANA), is used to uniquely identify an organization's IP space. The next 16 bits are the subnet ID, which can be used for hierarchical addressing, similar to the netmask portion of an IPv4 address. The final 64 bits are used for the interface ID, just as with link-local unicast addresses. The routing prefix and subnet ID can vary in size.

Figure 7-18: The parts of an IPv6 global unicast address

IPv6 provides a lot more efficiency than IPv4 in terms of routing packets to their destination and making effective use of address space. This efficiency is due to the larger range of addresses available and the use of link-local and global addressing along with unique host identifiers.

NOTE *It's easy for you to visually differentiate IPv6 and IPv4 addresses, but many programs cannot do so. If you need to specify an IPv6 address, some applications, such as browsers or command line utilities, require you to place square brackets around the address, like this: [1111::2222:333:44:5:ffff]. This requirement isn't always documented well and has been a source of frustration for many as they learn IPv6.*

IPv6 Packet Structure

http_ip4and6 .pcapng

The structure of the IPv6 header has grown to support more features, but it was also designed to be easier to parse. Instead of being variable in size with a header length field that needs to be checked to parse the header, headers are now a fixed 40 bytes. Additional options are provided via extension headers. The benefit is that most routers only need to process the 40-byte header to forward the packet along.

As shown in Figure 7-19, the IPv6 header has the following fields:

Version The version of IP being used (this is always 6 for IPv6).

Traffic Class Used to prioritize certain classes of traffic.

Internet Protocol Version 6 (IPv6)							
Offsets	Octet	0		1		2	3
Octet	Bit	0–3	4–7	8–11	12–15	16–23	24–31
0	0	Version	Traffic Class		Flow Label		
4	32	Payload Length				Next Header	Hop Limit
8	64	Source IP Address					
12	96						
16	128						
20	160						
24	192	Destination IP Address					
28	224						
32	256						
36	288						

Figure 7-19: The IPv6 packet structure

Flow Label Used by a source to label a set of packets belonging to the same flow. This field is typically used for quality of service (QoS) management and to ensure packets that are part of the same flow take the same path.

Payload Length The length of the data payload following the IPv6 header.

Next Header Identifies the layer 4 header that encapsulates the IPv6 header. This field replaces the Protocol field in IPv4.

Hop Limit Defines the lifetime of the packet, measured in hops through routers. This field replaces the TTL field in IPv4.

Source IP Address The IP address of the host that sent the packet.

Destination IP Address The IP address of the packet's destination.

Let's compare an IPv4 and an IPv6 packet to examine a few of the differences by looking at *http_ip4and6.pcapng*. In this capture, a web server was configured to listen for both IPv4 and IPv6 connections on the same physical host. A single client configured with both IPv4 and IPv6 addresses browsed to a server using each of its addresses independently and downloaded the *index.php* page using HTTP via the curl application (Figure 7-20).

Upon opening the capture, you should readily see which packets belong to which conversation based on the addresses in the Source and Destination columns in the Packet List area. Packets 1 through 10 represent the IPv4 stream (stream 0), and packets 11 through 20 represent the IPv6 stream (stream 1). You can filter for each of these streams from the Conversations window or by entering `tcp.stream == 0` or `tcp.stream == 1` in the filter bar.

Figure 7-20: Connections between the same physical hosts using different IP versions

We'll cover HTTP, the protocol responsible for serving web pages on the internet, in depth in Chapter 8. In this example, just note that the business of serving web pages remains consistent regardless of which lower-layer network protocol is used. The same can be said of TCP, which also operates in a consistent manner. This is a prime example of encapsulation in action. Although IPv4 and IPv6 function differently, the protocols functioning at different layers are unaffected.

Figure 7-21 provides a side-by-side comparison of two packets with the same function—packets 1 and 11. Both packets are TCP SYN packets designed to initiate a connection from the client to the server. The Ethernet and TCP sections of these packets are nearly identical. However, the IP sections are completely different.

- The source and destination address formats are different ❻❿.
- The IPv4 packet is 74 bytes with a 60-byte total length ❶, which includes both the IPv4 header and payload and a 14-byte Ethernet header. The IPv6 packet is 96 bytes with a 40-byte IPv6 payload ❼ and a separate 40-byte IPv6 header along with the 14-byte Ethernet header. The IPv6 header is 40 bytes, double the IPv4 header's 20 bytes, to accommodate the larger address size.
- IPv4 identifies the protocol with the Protocol field ❹, whereas IPv6 identifies it with the Next header field (which can also be used to specify extension headers) ❽.
- IPv4 has a TTL field ❸, whereas IPv6 accomplishes the same functionality using the Hop limit field ❾.
- IPv4 includes a header checksum value ❺, while IPv6 does not.
- The IPv4 packet is not fragmented, but it still includes values for those options ❷. The IPv6 header doesn't contain this information because, if fragmentation were required, it would be implemented in an extension header.

```
Wireshark · Packet 1 · http_ip4and6
> Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
> Ethernet II, Src: Vmware_d3:46:dd (00:0c:29:d3:46:dd), Dst: Vmware_1f:a7:55 (00:0c:29:1f:a7:55)
v Internet Protocol Version 4, Src: 172.16.16.140, Dst: 172.16.16.139
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  v Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
      0000 00.. = Differentiated Services Codepoint: Default (0)
      .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
❶ Total Length: 60
  Identification: 0xfadf (64223)
  v Flags: 0x02 (Don't Fragment)
❷   0... .... = Reserved bit: Not set
      .1.. .... = Don't fragment: Set
      ..0. .... = More fragments: Not set
  Fragment offset: 0
❸ Time to live: 64
❹ Protocol: TCP (6)
❺ v Header checksum: 0xc6a4 [validation disabled]
      [Good: False]
      [Bad: False]
  Source: 172.16.16.140
❻ Destination: 172.16.16.139
  [Source GeoIP: Unknown]
  [Destination GeoIP: Unknown]
> Transmission Control Protocol, Src Port: 53350 (53350), Dst Port: 80 (80), Seq: 0, Len: 0

No.: 1 · Time: 0.000000 · Source: 172.16.16.140 · Destination: 172.16.16.139 … [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM=1 TSval=417031 TSecr=0 W
                                                                                    Close   Help

Wireshark · Packet 11 · http_ip4and6
> Frame 11: 94 bytes on wire (752 bits), 94 bytes captured (752 bits)
> Ethernet II, Src: Vmware_d3:46:dd (00:0c:29:d3:46:dd), Dst: Vmware_1f:a7:55 (00:0c:29:1f:a7:55)
v Internet Protocol Version 6, Src: 2001:db8:1:2::1002, Dst: 2001:db8:1:2::1000
    0110 .... = Version: 6
  v .... 0000 0000 .... .... .... .... .... = Traffic class: 0x00 (DSCP: CS0, ECN: Not-ECT)
      .... 0000 00.. .... .... .... .... .... = Differentiated Services Codepoint: Default (0)
      .... .... ..00 .... .... .... .... .... = Explicit Congestion Notification: Not ECN-Capabl…
    .... .... .... 0000 0000 0000 0000 0000 = Flowlabel: 0x00000000
❼ Payload length: 40
❽ Next header: TCP (6)
❾ Hop limit: 64
❿ Source: 2001:db8:1:2::1002
  Destination: 2001:db8:1:2::1000
  [Source GeoIP: Unknown]
  [Destination GeoIP: Unknown]
> Transmission Control Protocol, Src Port: 35023 (35023), Dst Port: 80 (80), Seq: 0, Len: 0

No.: 11 · Time: 4.999280 · Source: 2001:db8:1:2::1002 · Destination: 2001:db8… [SYN] Seq=0 Win=28800 Len=0 MSS=1440 SACK_PERM=1 TSval=418281 TSecr=0 W
                                                                                    Close   Help
```

Figure 7-21: A side-by-side comparison of IPv4 (top) and IPv6 (bottom) packets performing the same function

Performing side-by-side comparisons of IPv4 and IPv6 traffic is a great way to fully appreciate the difference between how the two protocols operate.

Neighbor Solicitation and ARP

icmpv6_neighbor_solicitation.pcapng

When we discussed the different classifications of traffic earlier, I listed unicast, multicast, and anycast but did not list broadcast traffic. IPv6 doesn't

support broadcast traffic because broadcast is viewed as an inefficient mechanism for transmission. Because there is no broadcast, ARP can't be used for hosts to find each other on a network. So, how do IPv6 devices find each other?

The answer lies with a new feature called *neighbor solicitation*, a function of Neighbor Discovery Protocol (NDP), which utilizes ICMPv6 (discussed in the last section of this chapter) to do its legwork. To accomplish this task, ICMPv6 uses multicast, a type of communication in which only hosts that subscribe to a data stream will receive and process it. Multicast traffic can be identified quickly because it has its own reserved IP space (ff00::/8).

Although the address resolution process relies on a different protocol, it still uses a very simple request/response workflow. For example, let's consider a scenario in which a host with the IPv6 address 2001:db8:1:2::1003 wants to communicate with another host identified by the address 2001:db8:1:2::1000. Just as with IPv4, the source device must be able to determine the link-layer (MAC) address of the host it wants to communicate with, since this is intra-network communication. This process is described in Figure 7-22.

Figure 7-22: The neighbor solicitation process for address resolution

In this process, the host 2001:db8:1:2::1003 sends a Neighbor Solicitation (ICMPv6 type 135) packet to every device on the network via multicast, asking, "What is the MAC address for the device whose IP address is 2001:db8:1:2::1000? My MAC address is 00:0C:29:2f:80:31."

The device assigned that IPv6 address will receive this multicast transmission and respond to the originating host with a Neighbor Advertisement (ICMPv6 type 136) packet. This packet says, "Hi, my network address is 2001:db8:1:2::1000 and my MAC address is 00:0c:29:1f:a7:55." Once this message is received, communication can begin.

You can see this process in action in the capture file *icmpv6_neighbor_solicitation.pcapng*. This capture embodies the example we've just discussed in which 2001:db8:1:2::1003 wants to communicate with 2001:db8:1:2::1000. Look at the first packet and expand the ICMPv6 portion in the Packet Details window (Figure 7-23) to see that the packet is ICMP type 135 ❷ and was sent from 2001:db8:1:2::1003 to the multicast address ff02::1:ff00:1000 ❶. The source host provided the target IPv6 address it wanted to communicate with ❸, along with its own layer 2 MAC address ❹.

Figure 7-23: A neighbor solicitation packet

The response to the solicitation is found in the second packet in the capture file. Expanding the ICMPv6 portion of the Packet Details window (Figure 7-24) reveals this packet is ICMP type 136 ❷, was sent from 2001:db8:1:2::1000 back to 2001:db8:1:2::1003 ❶, and contains the MAC address 00:0c:29:1f:a7:55 associated with 2001:db8:1:2::1000 ❸.

Figure 7-24: A neighbor advertisement packet

Upon completion of this process, 2001:db8:1:2::1003 and 2001:db8:1:2::1000 begin communicating normally with ICMPv6 echo request and reply packets, indicating the neighbor solicitation process and link-layer address resolution was successful.

IPv6 Fragmentation

ipv6_fragments .pcapng

Fragmentation support was built into the IPv4 header because it ensured packets could traverse all sorts of networks at a time when network MTUs varied wildly. In IPv6, fragmentation is used less, so the options supporting it are not included in the IPv6 header. A device transmitting IPv6 packets is expected to perform a process called *MTU discovery* to determine the maximum size of packets it can send before actually sending them. In the event that a router receives a packet that is too large for the MTU on the network it is forwarding to, it will drop the packet and return an ICMPv6 Packet Too Big (type 2) message to the originating host. Upon receipt, the originating host will attempt to resend the packet with a smaller MTU, if such action is supported by the upper-layer protocol. This process will repeat until a small enough MTU is reached or until the payload can be fragmented no more (Figure 7-25). A router will never be responsible for fragmenting packets on its own; the source device is responsible for determining an appropriate MTU for the transmission path and fragmenting appropriately.

Figure 7-25: IPv6 MTU path discovery

If the upper-layer protocol being used in conjunction with IPv6 can't limit the size of the packet payload, then fragmentation must still be used. A fragmentation extension header can be added to the IPv6 packet to support this scenario. You will find a sample capture showing IPv6 fragmentation in the file named *ipv6_fragments.pcapng*.

Because the receiving device has a smaller MTU than the sending device, there are two fragmented packets to represent each ICMPv6 echo request and reply in the capture file. The fragmentation header from the first packet is shown in Figure 7-26.

Figure 7-26: An IPv6 fragment header extension

The 8-byte extension header contains all the same fragmentation properties that are found in an IPv4 packet, such as a Fragment offset ❷, More Fragments flag ❸, and Identification field ❹. Instead of being present in every packet, it is only added to the end of packets requiring fragmentation. This more efficient process still allows the receiving system to reassemble the fragments appropriately. Additionally, if this extension header is present, the Next header field will point to the extension header rather than the encapsulating protocol ❶.

IPv6 Transitional Protocols

IPv6 addresses a very real problem, but its adoption has been slow because of the effort required to transition network infrastructure to it. To ease this transition, several protocols allow IPv6 communication to be tunneled across networks that support only IPv4 communication. In this respect, tunneling means that IPv6 communication is encapsulated inside of IPv4 communications just as other protocols may be encapsulated. Encapsulation is usually done in one of three ways:

Router to Router Uses a tunnel to encapsulate IPv6 traffic from the transmitting and receiving hosts on their networks over an IPv4 network. This method allows entire networks to communicate in IPv6 over intermediary IPv4 links.

Host to Router Uses encapsulation at the router level to transmit traffic from an IPv6 host over an IPv4 network. This method allows an individual host to communicate in IPv6 to an IPv6 network when the host resides on an IPv4-only network.

Host to Host Uses a tunnel between two endpoints to encapsulate IPv6 traffic between IPv4- or IPv6-capable hosts. This method allows IPv6 endpoints to communicate directly across an IPv4 network.

While this book won't cover transitional protocols in depth, it's helpful to be aware of their existence in case you ever need to investigate them while performing analysis at the packet level. The following are a few common protocols:

6to4 Also known as *IPv6 over IPv4*, this transitional protocol allows IPv6 packets to be transmitted across an IPv4 network. This protocol supports relays and routers to provide router-to-router, host-to-router, and host-to-host IPv6 communication.

Teredo This protocol, used for IPv6 unicast communications over an IPv4 network using NAT (network address translation), works by sending IPv6 packets over IPv4 encapsulated in the UDP transport protocol.

ISATAP This intrasite protocol allows communication between IPv4- and IPv6-only devices within a network in a host-to-host manner.

Internet Control Message Protocol (ICMP)

Internet Control Message Protocol (ICMP) is the utility protocol of TCP/IP, responsible for providing information regarding the availability of devices, services, or routes on a TCP/IP network. Most network-troubleshooting techniques and tools center around common ICMP message types. ICMP is defined in RFC 792.

ICMP Packet Structure

ICMP is part of IP, and it relies on IP to transmit its messages. ICMP contains a relatively small header that changes depending on its purpose. As shown in Figure 7-27, the ICMP header contains the following fields:

Type The type or classification of the ICMP message, based on the RFC specification

Code The subclassification of the ICMP message, based on the RFC specification

Checksum Used to ensure that the contents of the ICMP header and data are intact upon arrival

Variable A portion that varies depending on the Type and Code fields

Internet Control Message Protocol (ICMP)					
Offsets	Octet	0	1	2	3
Octet	Bit	0–7	8–15	16–23	24–31
0	0	Type	Code	Checksum	
4+	32+	Variable			

Figure 7-27: The ICMP header

ICMP Types and Messages

As noted, the structure of an ICMP packet depends on its purpose, as defined by the values in the *Type* and *Code* fields.

You might consider the ICMP Type field the packet's classification and the Code field its subclass. For example, a Type field value of 3 indicates "destination unreachable." While this information alone might not be enough to troubleshoot a problem, if that packet were also to specify a Code field value of 3, indicating "port unreachable," you could conclude that there is an issue with the port with which you are attempting to communicate.

NOTE *For a full list of available ICMP types and codes, see* http://www.iana.org/assignments/icmp-parameters/.

Echo Requests and Responses

icmp_echo.pcapng

ICMP's biggest claim to fame is the ping utility. *Ping* is used to test for connectivity to a device. While ping itself isn't a part of the ICMP spec, it utilizes ICMP to achieve its core functionality.

To use ping, enter `ping ipaddress` at the command prompt, replacing `ipaddress` with the actual IP address of a device on your network. If the target device is turned on, your computer has a communication route to it, and there is no firewall blocking that communication, you should see replies to your `ping` command.

The example in Figure 7-28 shows four successful replies that display their size; round trip time (or RTT), which is the time it takes for the packet to arrive and a response to be received; and TTL used. The Windows utility also provides a summary detailing how many packets were sent, received, and lost. If communication fails, you should see a message telling you why.

Figure 7-28: The `ping` command being used to test connectivity

Basically, the `ping` command sends one packet at a time to a device and listens for a reply to determine whether there is connectivity to that device, as shown in Figure 7-29.

Figure 7-29: The `ping` command involves only two steps.

NOTE *Although ping has long been the bread and butter of IT, its results can be a bit deceiving when host-based firewalls are deployed. Many of today's firewalls limit the ability of a device to respond to ICMP packets. This is great for security, because potential attackers using ping to determine whether a host is accessible might be deterred, but troubleshooting is also more difficult—it can be frustrating to ping a device to test for connectivity and not receive a reply when you know you can communicate with that device.*

The ping utility in action is a great example of simple ICMP communication. The packets in the file *icmp_echo.pcapng* demonstrate what happens when you run ping.

The first packet (see Figure 7-30) shows that host 192.168.100.138 is sending a packet to 192.168.100.1 ❶. When you expand the ICMP portion of this packet, you can determine the ICMP packet type by looking at the Type and Code fields. In this case, the packet is type 8 ❷ and the code is 0 ❸, indicating an echo request. (Wireshark should tell you what the displayed type/code actually is.) This echo (ping) request is the first half of the equation. It is a simple ICMP packet, sent using IP, that contains a small amount of data. Along with the type and code designations and the checksum, we also have a sequence number that is used to pair requests with replies, and there is a random text string in the variable portion of the ICMP packet.

Figure 7-30: An ICMP echo request packet

NOTE *The terms* echo *and* ping *are often used interchangeably, but remember that* ping *is actually the name of a tool. The ping tool is used to send ICMP echo request packets.*

The second packet in this sequence is the reply to our request (see Figure 7-31). The ICMP portion of the packet is type 0 ❶ and code 0 ❷, indicating that this is an echo reply. Because the sequence number and identifier in the second packet match those of the first ❸, we know that this echo reply matches the echo request in the previous packet. Wireshark displays the values of these fields in big-endian (BE) and little-endian (LE) format. In other words, it represents the data in a different order based on how a particular endpoint might process the data. This reply packet also contains the same 32-byte string of data that was transmitted with the initial request ❹. Once this second packet has been received by 192.168.100.138, ping will report success.

Figure 7-31: An ICMP echo reply packet

Note that you can use variations of the ping command to increase the size of the data padding in echo requests, which forces packets to be fragmented for various types of network troubleshooting. This may be necessary when you're troubleshooting networks that require a smaller fragment size.

NOTE *The random text used in an ICMP echo request can be of great interest to a potential attacker. Attackers can use the information in this padding to profile the operating system used on a device. Additionally, attackers can place small bits of data in this field as a method of covert communication.*

traceroute

icmp_traceroute .pcapng

The traceroute utility is used to identify the path from one device to another. On a simple network, a path may go through only a single router or no router at all. On a complex network, however, a packet may need to go through dozens of routers to reach its final destination. Thus, it is crucial to be able to trace the exact path a packet takes from one destination to another in order to troubleshoot communication.

By using ICMP (with a little help from IP), traceroute can map the path packets take. For example, the first packet in the file *icmp_traceroute.pcapng* is pretty similar to the echo request we looked at in the previous section (see Figure 7-32).

In this capture, the packets were generated by running the command `tracert 4.2.2.1`. To use traceroute on Windows, enter `tracert` *`ipaddress`* at the command prompt, replacing *`ipaddress`* with the actual IP address of a device whose path you want to discover. To use traceroute on Linux or Mac, use the command `traceroute` *`ipaddress`*.

Figure 7-32: An ICMP echo request packet with a TTL value of 1

At first glance, this packet appears to be a simple echo request ❸ from 192.168.100.138 to 4.2.2.1 ❶, and everything in the ICMP portion of the packet is identical to the formatting of an echo request packet. However, when you expand the IP header of this packet, you'll notice something odd: the packet's TTL value is set to 1 ❷, meaning that the packet will be dropped at the first router that it hits. Because the destination 4.2.2.1 address is an internet address, we know that there must be at least one router between our source and destination devices, so there is no way this packet will reach its destination. That's good for us, because traceroute relies on the fact that this packet will make it to only the first router it traverses.

The second packet is, as expected, a reply from the first router we reached along the path to our destination (see Figure 7-33). This packet reached this device at 192.168.100.1, its TTL was decremented to 0, and the packet could not be transmitted further, so the router replied with an ICMP response. This packet is type 11 ❶ and code 0 ❷, data that tells us that the destination was unreachable because the packet's TTL was exceeded during transit.

This ICMP packet is sometimes called a *double-headed packet*, because the tail end of its ICMP portion contains a copy of the IP header ❸ and ICMP data ❹ that were sent in the original echo request. This information can prove very useful for troubleshooting.

Figure 7-33: An ICMP response from the first router along the path

This process of sending packets with a TTL value of 1 occurs two more times before we get to packet 7. Here, you see the same thing you saw in the first packet, except that this time, the TTL value in the IP header is set to 2, which ensures the packet will make it to the second hop router before it is dropped. As expected, we receive a reply from the next hop router, 12.180.241.1, with the same ICMP destination unreachable and TTL exceeded messages.

This process continues, with the TTL value increasing by 1, until the destination 4.2.2.1 is reached. Right before that happens, however, you'll see in Figure 7-34 that the request on line 8 timed out. How can a request along the path time out and the process still complete successfully? Typically, this happens when a router is configured to not respond to ICMP requests. The router still receives the request and passes the data forward to the next router, which is why we're able to see the next hop on line 9 in Figure 7-34. It just didn't generate the ICMP time to live exceeded packet as the other hops did. With no response, tracert assumes the request has timed out and moves on to the next one.

To sum up, this traceroute process has communicated with each router along the path, building a map of the route to the destination. An example map is shown in Figure 7-34.

```
Command Prompt

C:\>tracert 4.2.2.1

Tracing route to a.resolvers.level3.net [4.2.2.1]
over a maximum of 30 hops:

  1     1 ms    <1 ms    <1 ms  INTERIORE2000 [172.16.16.1]
  2     1 ms     1 ms     1 ms  192.168.1.1
  3     2 ms     2 ms     1 ms  192.168.0.1
  4    25 ms    21 ms    21 ms  172-127-116-3.lightspeed.tukrga.sbcglobal.net [17
2.127.116.3]
  5    26 ms    26 ms    24 ms  76.201.208.162
  6    28 ms    25 ms    24 ms  12.83.82.181
  7    24 ms    25 ms    26 ms  12.122.117.121
  8     *        *        *     Request timed out.
  9    24 ms    24 ms    24 ms  a.resolvers.level3.net [4.2.2.1]
```

Figure 7-34: A sample output from the traceroute utility

NOTE *The discussion here on traceroute is generally Windows focused because this utility uses ICMP exclusively. The traceroute utility on Linux is a bit more versatile and can utilize other protocols in order to perform route path tracing.*

ICMP Version 6 (ICMPv6)

The updated version of IP relies heavily on ICMP for functions such as neighbor solicitation and path discovery, as demonstrated in earlier examples. *ICMPv6* was established with RFC 4443 to support the feature set needed for IPv6, along with additional enhancements. We don't cover ICMPv6 separately in this book because it uses the same packet structure as do ICMP packets.

ICMPv6 packets are generally classified as either error messages or informational messages. You can find a full list of the available types and codes from IANA here: *http://www.iana.org/assignments/icmpv6-parameters/icmpv6-parameters.xhtml*.

This chapter has introduced you to a few of the most important protocols you will examine during the process of packet analysis. ARP, IP, and ICMP are at the foundation of all network communications, and they are critical to just about every daily task you will perform. In Chapter 8, we will look at common transport layer protocols, TCP and UDP.

8

TRANSPORT LAYER PROTOCOLS

In this chapter, we'll continue to examine individual protocols and how they appear at the packet level. Moving up the OSI model, we'll look at the transport layer and the two most common transport protocols, TCP and UDP.

Transmission Control Protocol (TCP)

The ultimate goal of the *Transmission Control Protocol (TCP)* is to provide end-to-end reliability for the delivery of data. TCP, which is defined in RFC 793, handles data sequencing and error recovery, and ultimately ensures that data gets where it's supposed to go. TCP is considered a *connection-oriented protocol* because it establishes a formal connection before transmitting data, tracks packet delivery, and usually attempts to formally close communication channels when transmission is complete. Many commonly used application-layer protocols rely on TCP and IP to deliver packets to their final destination.

TCP Packet Structure

TCP provides a great deal of functionality, as reflected in the complexity of its header. As shown in Figure 8-1, the following are the TCP header fields:

Source Port The port used to transmit the packet.

Destination Port The port to which the packet will be transmitted.

Sequence Number The number used to identify a TCP segment. This field is used to ensure that parts of a data stream are not missing.

Acknowledgment Number The sequence number that is to be expected in the next packet from the other device taking part in the communication.

Flags The URG, ACK, PSH, RST, SYN, and FIN flags for identifying the type of TCP packet being transmitted.

Window Size The size of the TCP receiver buffer in bytes.

Checksum Used to ensure the contents of the TCP header and data are intact upon arrival.

Urgent Pointer If the URG flag is set, this field is examined for additional instructions for where the CPU should begin reading the data within the packet.

Options Various optional fields that can be specified in a TCP packet.

Transmission Control Protocol (TCP)						
Offsets	Octet	0		1	2	3
Octet	Bit	0–3	4–7	8–15	16–23	24–31
0	0	Source Port			Destination Port	
4	32	Sequence Number				
8	64	Acknowledgment Number				
12	96	Data Offset	Reserved	Flags	Window Size	
16	128	Checksum			Urgent Pointer	
20+	160+	Options				

Figure 8-1: The TCP header

TCP Ports

tcp_ports.pcapng

All TCP communication takes place using source and destination *ports*, which can be found in every TCP header. A port is like the jack on an old telephone switchboard. A switchboard operator would monitor a board of lights and plugs. When a light lit up, he would connect with the caller, ask who she wanted to talk to, and then connect her to the other party by plugging in a cable. Every call needed to have a source port (the caller) and a destination port (the recipient). TCP ports work in much the same fashion.

To transmit data to a particular application on a remote server or device, a TCP packet must know the port the remote service is listening on. If you try to access an application on a port other than the one configured for use, the communication will fail.

The source port in this sequence isn't incredibly important and can be selected randomly. The remote server will simply determine the port to communicate with from the original packet it's sent (see Figure 8-2).

Figure 8-2: TCP uses ports to transmit data.

There are 65,535 ports available for use when communicating with TCP. We typically divide these into two groups:

- The *system port group* (also known as the standard port or well-known port group) is from 1 through 1023 (ignoring 0 because it's reserved). Well-known, established services generally use ports that lie within the system port grouping.
- The *ephemeral port group* is from 1024 through 65535 (although some operating systems have different definitions for this). Only one service can communicate on a port at any given time, so modern operating systems select source ports randomly in an effort to make communications unique. These source ports are typically located in the ephemeral range.

Let's examine a couple of TCP packets and identify the port numbers they are using by opening the file *tcp_ports.pcapng*. In this file, we have the HTTP communication of a client browsing to two websites. As mentioned previously, HTTP uses TCP for communication, making it a great example of standard TCP traffic.

In the first packet in this file (see Figure 8-3), the first two values represent the packet's source port and destination port. This packet is being sent from 172.16.16.128 to 212.58.226.142. The source port is 2826 ❶, an ephemeral port. (Remember that source ports are chosen at random by the operating system, although they can increment from that random selection.) The destination port is a system port, port 80 ❷, the standard port used for web servers using HTTP.

Figure 8-3: The source and destination ports can be found in the TCP header.

Notice that Wireshark labels these ports as slc-systemlog (2826) and http (80). Wireshark maintains a list of ports and their most common uses. Although system ports are primarily the ones with labeled common uses, many ephemeral ports have commonly used services associated with them. The labeling of these ports can be confusing, so it's typically best to disable it by turning off transport name resolution. To do this, go to **Edit ▸ Preferences ▸ Name Resolution** and uncheck Enable Transport Name Resolution. If you wish to leave this option enabled but want to change how Wireshark identifies a certain port, you can do so by modifying the *services* file located in the Wireshark system directory. The contents of this file are based on the IANA common ports listing (see "Using a Custom hosts File" on page 86 for an example of how to edit a name resolution file).

The second packet is sent back from 212.58.226.142 to 172.16.16.128 (see Figure 8-4). As with the IP addresses, the source and destination ports are now also switched ❶.

In most cases, TCP-based communication works the same way: a random source port is chosen to communicate to a known destination port. Once this initial packet is sent, the remote device communicates with the source device using the established ports.

This sample capture file includes one more communication stream. See if you can locate the port numbers it uses for communication.

Figure 8-4: Switching the source and destination port numbers for reverse communication

NOTE *As we progress through this book, you'll learn more about the ports associated with common protocols and services. Eventually, you'll be able to profile services and devices by the ports they use. For a comprehensive list of common ports, look at the* services *file located in the Wireshark system directory.*

The TCP Three-Way Handshake

tcp_handshake .pcapng

All TCP-based communication must begin with a *handshake* between two hosts. This handshake process serves several purposes:

- It allows the transmitting host to ensure that the recipient host is up and able to communicate.
- It lets the transmitting host check that the recipient is listening on the port the transmitting host is attempting to communicate on.
- It allows the transmitting host to send its starting sequence number to the recipient so that both hosts can keep the stream of packets in proper sequence.

The TCP handshake occurs in three steps, as shown in Figure 8-5. In the first step, the device that wants to communicate (host A) sends a TCP packet to its target (host B). This initial packet contains no data other than the lower-layer protocol headers. The TCP header in this packet has the SYN flag set and includes the initial sequence number and maximum segment size (MSS) that will be used for the communication process. Host B responds to this packet by sending a similar packet with the SYN and ACK flags set, along with its initial sequence number. Finally, host A sends one last packet to host B with only the ACK flag set. Once this process is completed, both devices should have all of the information they need to begin communicating properly.

NOTE *TCP packets are often referred to by the flags they have set. For example, rather than refer to a packet as a TCP packet with the SYN flag set, we call that packet a SYN packet. As such, the packets used in the TCP handshake process are referred to as SYN, SYN/ACK, and ACK.*

To see this process in action, open *tcp_handshake.pcapng*. Wireshark includes a feature that replaces the sequence numbers of TCP packets with relative numbers for easier analysis. For our purposes, we'll disable this feature in order to see the actual sequence numbers. To disable this, choose **Edit ▸ Preferences**, expand the **Protocols** heading, and choose **TCP**. In the window, uncheck the box next to **Relative Sequence Numbers** and click **OK**.

Figure 8-5: The TCP three-way handshake

The first packet in this capture represents our initial SYN packet ❷ (see Figure 8-6). The packet is transmitted from 172.16.16.128 on port 2826 to 212.58.226.142 on port 80. We can see here that the sequence number transmitted is 3691127924 ❶.

The second packet in the handshake is the SYN/ACK response ❸ from 212.58.226.142 (see Figure 8-7). This packet also contains this host's initial sequence number (233779340) ❶ and an acknowledgment number (3691127925) ❷. The acknowledgment number shown here is 1 more than the sequence number included in the previous packet, because this field is used to specify the next sequence number the host expects to receive.

The final packet is the ACK ❷ packet sent from 172.16.16.128 (see Figure 8-8). This packet, as expected, contains the sequence number 3691127925 ❶ as defined in the previous packet's Acknowledgment number field.

A handshake occurs before every TCP communication sequence. When you are sorting through a busy capture file in search of the beginning of a communication sequence, the sequence SYN-SYN/ACK-ACK is a great marker.

```
Wireshark · Packet 1 · tcp_handshake
> Frame 1: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
> Ethernet II, Src: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a), Dst: D-Link_21:99:4c (00:05:5d:21:99:4c)
> Internet Protocol Version 4, Src: 172.16.16.128, Dst: 212.58.226.142
v Transmission Control Protocol, Src Port: 2826 (2826), Dst Port: 80 (80), Seq: 3691127924, Len: 0
    Source Port: 2826
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 3691127924 ❶
    Acknowledgment number: 0
    Header Length: 32 bytes
  v Flags: 0x002 (SYN)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...0 .... = Acknowledgment: Not set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
❷   > .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: **********S*]
    Window size value: 8192
    [Calculated window size: 8192]
  > Checksum: 0xcfeb [validation disabled]
    Urgent pointer: 0
  > Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), SACK permitted
No.: 1 · Time: 0.000000 · Source: 172.16.16.128 · Destination: 212.58.226.142 · Protocol: TCP · Length: 66 · Info: 2826 → 80 [SYN] Seq=3691127924 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
                                                                                                   Close   Help
```

Figure 8-6: The initial SYN packet

```
Wireshark · Packet 2 · tcp_handshake
> Frame 2: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
> Ethernet II, Src: D-Link_21:99:4c (00:05:5d:21:99:4c), Dst: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a)
> Internet Protocol Version 4, Src: 212.58.226.142, Dst: 172.16.16.128
v Transmission Control Protocol, Src Port: 80 (80), Dst Port: 2826 (2826), Seq: 233779340, Ack: 3691127925, Len: 0
    Source Port: 80
    Destination Port: 2826
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 233779340 ❶
    Acknowledgment number: 3691127925 ❷
    Header Length: 32 bytes
❸ v Flags: 0x012 (SYN, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
    > .... .... ..1. = Syn: Set
      .... .... ...0 = Fin: Not set
      [TCP Flags: *******A**S*]
    Window size value: 5840
    [Calculated window size: 5840]
  > Checksum: 0x9ac0 [validation disabled]
    Urgent pointer: 0
  > Options: (12 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted, No-Operation (NOP), Window scale
  > [SEQ/ACK analysis]
No.: 2 · Time: 0.132627 · Source: 212.58.226.142 · Destination: 172.16.16.128 · Protocol: TCP · Length: 66 · Info: 80 → 2826 [SYN, ACK] Seq=233779340 Ack=3691127925 Win=5840 Len=0 MSS=1406 SACK_PERM=1 WS=128
                                                                                                   Close   Help
```

Figure 8-7: The SYN/ACK response

```
Wireshark · Packet 3 · tcp_handshake
> Frame 3: 54 bytes on wire (432 bits), 54 bytes captured (432 bits)
> Ethernet II, Src: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a), Dst: D-Link_21:99:4c (00:05:5d:21:99:4c)
> Internet Protocol Version 4, Src: 172.16.16.128, Dst: 212.58.226.142
v Transmission Control Protocol, Src Port: 2826 (2826), Dst Port: 80 (80), Seq: 3691127925, Ack: 233779341, Len: 0
    Source Port: 2826
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 3691127925 ❶
    Acknowledgment number: 233779341
    Header Length: 20 bytes
  v Flags: 0x010 (ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set ❷
      .... .... 0... = Push: Not set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: *******A****]
    Window size value: 4218
    [Calculated window size: 16872]
    [Window size scaling factor: 4]
  > Checksum: 0xe1b2 [validation disabled]
    Urgent pointer: 0
  > [SEQ/ACK analysis]
No.: 3 · Time: 0.132768 · Source: 172.16.16.128 · Destination: 212.58.226.142 · Protocol: TCP · Length: 54 · Info: 2826 → 80 [ACK] Seq=3691127925 Ack=233779341 Win=16872 Len=0
                                                                                    Close   Help
```

Figure 8-8: The final ACK

TCP Teardown

tcp_teardown.pcapng

Most greetings eventually have a good-bye and, in the case of TCP, every handshake has a teardown. The *TCP teardown* is used to gracefully end a connection between two devices after they have finished communicating. This process involves four packets, and it utilizes the FIN flag to signify the end of a connection.

In a teardown sequence, host A tells host B that it is finished communicating by sending a TCP packet with the FIN and ACK flags set. Host B responds with an ACK packet and transmits its own FIN/ACK packet. Host A responds with an ACK packet, ending the communication. This process is illustrated in Figure 8-9.

Figure 8-9: The TCP teardown process

To view this process in Wireshark, open the file *tcp_teardown.pcapng*. Beginning with the first packet in the sequence (see Figure 8-10), you can see that the device at 67.228.110.120 initiates teardown by sending a packet with the FIN and ACK flags set ❶.

Figure 8-10: The FIN/ACK packet initiates the teardown process.

Once this packet is sent, 172.16.16.128 responds with an ACK packet to acknowledge receipt of the first packet, and it sends a FIN/ACK packet. The process is complete when 67.228.110.120 sends a final ACK. At this point, the communication between the two devices ends. If they need to begin communicating again, they will have to complete a new TCP handshake.

TCP Resets

tcp_refuseconnection.pcapng

In an ideal world, every connection would end gracefully with a TCP teardown. In reality, connections often end abruptly. For example, a host may be misconfigured, or a potential attacker may perform a port scan. In these cases, when a packet is sent to a device that is not willing to accept it, a TCP packet with the RST flag set may be sent. The RST flag is used to indicate that a connection was closed abruptly or to refuse a connection attempt.

The file *tcp_refuseconnection.pcapng* displays an example of network traffic that includes an RST packet. The first packet in this file is from the host at 192.168.100.138, which is attempting to communicate with 192.168.100.1 on port 80. What this host doesn't know is that 192.168.100.1 isn't listening on port 80 because it's a Cisco router with no web interface configured. There is no service configured to accept connections on that port. In response to this attempted communication, 192.168.100.1 sends a packet to 192.168.100.138 telling it that communication won't be possible over

port 80. Figure 8-11 shows the abrupt end to this attempted communication in the TCP header of the second packet. The RST packet contains nothing other than RST and ACK flags ❶, and no further communication follows.

```
Wireshark · Packet 2 · tcp_refuseconnection
> Frame 2: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: CiscoInc_4b:c0:7f (00:12:80:4b:c0:7f), Dst: CompalCo_b8:59:b1 (00:16:d4:b8:59:b1)
> Internet Protocol Version 4, Src: 192.168.100.1, Dst: 192.168.100.138
v Transmission Control Protocol, Src Port: 80 (80), Dst Port: 3372 (3372), Seq: 0, Ack: 951057940, Len: 0
    Source Port: 80
    Destination Port: 3372
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0
    Acknowledgment number: 951057940
    Header Length: 20 bytes
❶ v Flags: 0x014 (RST, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Nonce: Not set
      .... 0... .... = Congestion Window Reduced (CWR): Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 0... = Push: Not set
    > .... .... .1.. = Reset: Set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: *******A*R**]
    Window size value: 0
    [Calculated window size: 0]
    [Window size scaling factor: -2 (no window scaling used)]
  > Checksum: 0x21b4 [validation disabled]
    Urgent pointer: 0
  > [SEQ/ACK analysis]
No.: 2 · Time: 0.001316 · Source: 192.168.100.1 · Destination: 192.168.100.138 · Protocol: TCP · Length: 60 · Info: 80 → 3372 [RST, ACK] Seq=0 Ack=951057940 Win=0 Len=0
                                                                                                  Close   Help
```

Figure 8-11: The RST and ACK flags signify the end of communication.

An RST packet ends communication whether it arrives at the beginning of an attempted communication sequence, as in this example, or is sent in the middle of the communication between hosts.

User Datagram Protocol (UDP)

udp_dnsrequest .pcapng

The *User Datagram Protocol (UDP)* is the other layer 4 protocol commonly used on modern networks. While TCP is designed for reliable data delivery with built-in error checking, UDP aims to provide speedy transmission. For this reason, UDP is a best-effort service, commonly referred to as a *connectionless protocol*. A connectionless protocol doesn't formally establish and terminate a connection between hosts, unlike TCP with its handshake and teardown processes.

With a connectionless protocol, which doesn't provide reliable services, it would seem that UDP traffic would be flaky at best. That would be true, except that the protocols that rely on UDP typically have their own built-in reliability services or use certain features of ICMP to make the connection somewhat more reliable. For example, the application-layer protocols DNS and DHCP, which are highly dependent on the speed of packet transmission across a network, use UDP as their transport layer protocol, but they handle error checking and retransmission timers themselves.

UDP Packet Structure

udp_dnsrequest .pcapng

The UDP header is much smaller and simpler than the TCP header. As shown in Figure 8-12, the following are the UDP header fields:

Source Port The port used to transmit the packet

Destination Port The port to which the packet will be transmitted

Packet Length The length of the packet in bytes

Checksum Used to ensure that the contents of the UDP header and data are intact upon arrival

User Datagram Protocol (UDP)					
Offsets	Octet	0	1	2	3
Octet	Bit	0–7	8–15	16–23	24–31
0	0	Source Port		Destination Port	
4	32	Packet Length		Checksum	

Figure 8-12: The UDP header

The file *udp_dnsrequest.pcapng* contains one packet. This packet represents a DNS request, which uses UDP. When you expand the packet's UDP header, you'll see four fields (see Figure 8-13).

Figure 8-13: The contents of a UDP packet are very simple.

The key point to remember is that UDP does not care about reliable delivery. Therefore, any application that uses UDP must take special steps to ensure reliable delivery, if it is necessary. This is in contrast to TCP, which utilizes a formal connection setup and teardown, and has features in place to validate that packets were transmitted successfully.

This chapter has introduced you to the transport layer protocols TCP and UDP. Not unlike network protocols, TCP and UDP are at the core of most of your daily communication, and the ability to analyze them effectively is critical to becoming an effective packet analyst. In Chapter 9, we will look at common application-layer protocols.

9

COMMON UPPER-LAYER PROTOCOLS

In this chapter, we'll continue to examine the functions of individual protocols, as well as what they look like when viewed with Wireshark. We'll discuss five of the most common upper-layer (layer 7) protocols: DHCP, DNS, HTTP, and SMTP.

Dynamic Host Configuration Protocol (DHCP)

In the early days of networking, when a device wanted to communicate over a network, it needed to be assigned an address by hand. As networks grew, this manual process quickly became cumbersome. To solve this problem, Bootstrap Protocol (BOOTP) was created to automatically assign addresses to network-connected devices. BOOTP was later replaced with the more sophisticated Dynamic Host Configuration Protocol (DHCP).

DHCP is an application-layer protocol responsible for allowing a device to automatically obtain an IP address (and addresses of other important network assets, such as DNS servers and routers). Most DHCP servers today also provide other parameters to clients, such as the addresses of the default gateway and DNS servers in use on the network.

DHCP Packet Structure

DHCP packets can carry quite a lot of information to a client. As shown in Figure 9-1, the following fields are present within a DHCP packet:

OpCode Indicates whether the packet is a DHCP request or a DHCP reply

Hardware Type The type of hardware address (10MB Ethernet, IEEE 802, ATM, and so on)

Hardware Length The length of the hardware address

Hops Used by relay agents to assist in finding a DHCP server

Transaction ID A random number used to pair requests with responses

Seconds Elapsed Seconds since the client first requested an address from the DHCP server

Flags The types of traffic the DHCP client can accept (unicast, broadcast, and so on)

Dynamic Host Configuration Protocol (DHCP)					
Offsets	Octet	0	1	2	3
Octet	Bit	0–7	8–15	16–23	24–31
0	0	OpCode	Hardware Type	Hardware Length	Hops
4	32	Transaction ID			
8	64	Seconds Elapsed		Flags	
12	96	Client IP Address			
16	128	Your IP Address			
20	160	Server IP Address			
24	192	Gateway IP Address			
28	224	Client IP Address			
32	256	Client Hardware Address (16 bytes)			
36	288				
40	320				
44	352				
48+	384+	Server Host Name (64 bytes)			
		Boot File (128 bytes)			
		Options			

Figure 9-1: The DHCP packet structure

Client IP Address The client's IP address (derived from the Your IP Address field)

Your IP Address The IP address offered by the DHCP server (ultimately becomes the Client IP Address field value)

Server IP Address The DHCP server's IP address

Gateway IP Address The IP address of the network's default gateway

Client Hardware Address The client's MAC address

Server Host Name The server's host name (optional)

Boot File A boot file for use by DHCP (optional)

Options Used to expand the structure of the DHCP packet to give it more features

The DHCP Initialization Process

dhcp_nolease_initialization.pcapng

The primary goal of DHCP is to assign addresses to clients during the initialization process. The renewal process takes place between a single client and a DHCP server, as shown in the file *dhcp_nolease_initialization.pcapng*. The DHCP initialization process is often referred to as the DORA process because it uses four types of DHCP packets: discover, offer, request, and acknowledgment, as shown in Figure 9-2. Here, we'll take a look at each type of DORA packet.

Figure 9-2: The DHCP DORA process

The Discover Packet

As you can see in the referenced capture file, the first packet is sent from 0.0.0.0 on port 68 to 255.255.255.255 on port 67. The client uses 0.0.0.0 because it does not yet have an IP address. The packet is sent to 255.255.255.255 because this is the network-independent broadcast address, thus ensuring that this packet will be sent out to every device on the network. Because the device doesn't know the address of a DHCP server, this first packet is sent in an attempt to find a DHCP server that will listen.

Examining the Packet Details pane, the first thing we notice is that DHCP relies on UDP as its transport layer protocol. DHCP is very concerned with the speed at which a client receives the information it's requesting. DHCP has its own built-in reliability measures, which means UDP is a perfect fit. You can see the details of the discovery process by examining the first packet's DHCP portion in the Packet Details pane, as shown in Figure 9-3.

Figure 9-3: The DHCP discover packet

NOTE *Because Wireshark still references BOOTP when dealing with DHCP, you'll see a Bootstrap Protocol section in the Packet Details pane, rather than a DHCP section. Nevertheless, I'll refer to this as the packet's* DHCP portion *throughout this book.*

This packet is a request, identified by the (1) in the Message type field ❶. Most of the fields in this discovery packet are either all zeros (as you can see in the IP address fields ❷) or pretty self-explanatory, based on the listing of DHCP fields in the previous section. The meat of this packet is in its four Option fields ❸.

DHCP Message Type This is option type `53`, with length `1` and a value of `Discover (1)`. These values indicate that this is a DHCP discover packet.

Client Identifier This provides additional information about the client requesting an IP address.

Requested IP Address This supplies the IP address the client would like to receive. This can be a previously used IP address or 0.0.0.0 to indicate no preference.

Parameter Request List This lists the different configuration items (IP addresses of other important network devices and other non IP items) the client would like to receive from the DHCP server.

The Offer Packet

The second packet in this file lists valid IP addresses in its IP header, showing a packet traveling from 192.168.1.5 to 192.168.1.10, as shown in Figure 9-4. The client doesn't actually have the 192.168.1.10 address yet, so the server will first attempt to communicate with the client using its hardware address, as provided by ARP. If communication isn't possible, the server will simply broadcast the offer to communicate.

The DHCP portion of this second packet, called the *offer packet*, indicates that the Message type is a reply ❶. This packet contains the same Transaction ID as the previous packet ❷, which tells us that this reply is indeed a response to our original request.

The offer packet is sent by the DHCP server in order to offer its services to the client. It does so by supplying information about itself and the addressing it wants to provide the client. In Figure 9-4, the IP address 192.168.1.10 in the Your (client) IP address field is being offered to the client ❸ from 192.168.1.5 identified by the Next server IP address field ❹.

The first option listed identifies the packet as a `DHCP Offer` ❺. The options that follow are supplied by the server and indicate the additional information it can offer, along with the client's IP address. You can see that it offers the following:

- An IP address lease time of 10 minutes
- A subnet mask of 255.255.255.0
- A broadcast address of 192.168.1.255
- A router address of 192.168.1.254
- A domain name of *mydomain.example*
- Domain name server addresses of 192.168.1.1 and 192.168.1.2

```
Wireshark · Packet 2 · dhcp_nolease_renewal
> Frame 2: 344 bytes on wire (2752 bits), 344 bytes captured (2752 bits) on interface 0
> Ethernet II, Src: Vmware_c0:60:a4 (00:0c:29:c0:60:a4), Dst: Vmware_59:fd:21 (00:0c:29:59:fd:21)
> Internet Protocol Version 4, Src: 192.168.1.5, Dst: 192.168.1.10
> User Datagram Protocol, Src Port: bootps (67), Dst Port: bootpc (68)
v Bootstrap Protocol (Offer)
    Message type: Boot Reply (2) ❶
    Hardware type: Ethernet (0x01)
    Hardware address length: 6
    Hops: 0
    Transaction ID: 0x696ddc66 ❷
    Seconds elapsed: 0
  > Bootp flags: 0x0000 (Unicast)
    Client IP address: 0.0.0.0
  ❸ Your (client) IP address: 192.168.1.10
  ❹ Next server IP address: 192.168.1.5
    Relay agent IP address: 0.0.0.0
    Client MAC address: Vmware_59:fd:21 (00:0c:29:59:fd:21)
    Client hardware address padding: 00000000000000000000
    Server host name not given
    Boot file name not given
    Magic cookie: DHCP
  v Option: (53) DHCP Message Type (Offer) ❺
      Length: 1
      DHCP: Offer (2)
  v Option: (54) DHCP Server Identifier
      Length: 4
      DHCP Server Identifier: 192.168.1.5
  v Option: (51) IP Address Lease Time
      Length: 4
      IP Address Lease Time: (600s) 10 minutes
  v Option: (1) Subnet Mask
      Length: 4
      Subnet Mask: 255.255.255.0
  v Option: (28) Broadcast Address
      Length: 4
      Broadcast Address: 192.168.1.255
  v Option: (3) Router
      Length: 4
      Router: 192.168.1.254
  v Option: (15) Domain Name
      Length: 16
      Domain Name: mydomain.example
  v Option: (6) Domain Name Server
      Length: 8
      Domain Name Server: 192.168.1.1
      Domain Name Server: 192.168.1.2
  v Option: (255) End
      Option End: 255
No.: 2 · Time: 1.002288 · Source: 192.168.1.5 · Destination: 192.168.1.10 · Protocol: DHCP · Info: DHCP Offer - Transaction ID 0x696ddc66
                                                                                    Close   Help
```

Figure 9-4: The DHCP offer packet

The Request Packet

Once the client receives an offer from the DHCP server, it should accept it with a DHCP request packet, as shown in Figure 9-5.

The third packet in this capture still comes from IP address 0.0.0.0, because we have not yet completed the process of obtaining an IP address ❶. The packet now knows the DHCP server it is communicating with.

Figure 9-5: The DHCP request packet

The Message type field shows that this packet is a request ❷, and the Transaction ID field is the same as in the first two packets ❸, indicating they are part of the same process. This packet is similar to the discover packet, in that all of its IP-addressing information is zeroed.

Finally, in the Option fields, we see that this is a `DHCP Request` ❹. Notice that the requested IP address is no longer blank and that the DHCP Server Identifier field also contains an address ❺.

The Acknowledgment Packet

In the final step in this process, the DHCP server sends the requested IP addresses to the client in an acknowledgment packet and records that information in its database, as shown in Figure 9-6. The client now has an IP address and can use it to begin communicating on the network.

Figure 9-6: The DCHP acknowledgment packet

DHCP In-Lease Renewal

dhcp_inlease_renewal.pcapng

When a DHCP server assigns an IP address to a device, it *leases* it to the client. This means that the client is allowed to use the IP address for only a limited amount of time before it must renew the lease. The DORA process just discussed occurs the first time a client gets an IP address or when its lease time has expired. In either case, the device is considered to be *out of lease*.

When a client with an IP address in-lease reboots, it must perform a truncated version of the DORA process in order to reclaim its IP address. This process is called an *in-lease renewal*.

In the case of a lease renewal, the discovery and offer packets are unnecessary. Think of an in-lease renewal as being the same DORA process used in an out-of-lease renewal, but the in-lease renewal doesn't need to do as much, leaving only the request and acknowledgment steps. You'll find a sample capture of an in-lease renewal in the file *dhcp_inlease_renewal.pcapng*.

DHCP Options and Message Types

DHCP's real flexibility lies in its available options. As you've seen, the packet's DHCP options can vary in size and content. The packet's overall size depends on the combination of options used. You can view a full list of the many DHCP options at *http://www.iana.org/assignments/bootp-dhcp-parameters/*.

The only option required in all DHCP packets is the Message type option (option 53). This option identifies how the DHCP client or server will process the information contained within the packet. There are 8 message types, as defined in Table 9-1.

Table 9-1: DHCP Message Types

Type number	Message type	Description
1	Discover	Used by the client to locate available DHCP servers
2	Offer	Sent by the server to the client in response to a discover packet
3	Request	Sent by the client to request the offered parameters from the server
4	Decline	Sent by the client to the server to indicate invalid parameters within a packet
5	ACK	Sent by the server to the client with the configuration parameters requested
6	NAK	Sent by the client to the server to refuse a request for configuration parameters
7	Release	Sent by the client to the server to cancel a lease by releasing its configuration parameters
8	Inform	Sent by the client to the server to ask for configuration parameters when the client already has an IP address

DHCP Version 6 (DHCPv6)

dhcp6_outlease_acquisition.pcapng

If you examine the packet structure for a DHCP packet in Figure 9-1, you'll see that it doesn't provide enough room to support the length required for IPv6 address allocation. Instead of retrofitting DHCP for this purpose, DHCPv6 was devised in RFC3315. Since DHCPv6 isn't built on the concept of BOOTP, its packet format is much simpler (Figure 9-7).

Dynamic Host Configuration Protocol Version 6 (DHCPv6)					
Offsets	Octet	0	1	2	3
Octet	Bit	0–7	8–15	16–23	24–31
0	0	Message Type	Transaction ID		
4+	32+	Options			

Figure 9-7: The DHCPv6 packet structure

The packet structure shown here contains only two static values, which function in the same manner as their DHCP counterparts. The rest of the packet structure varies depending on the message type identified in the first byte. Within the Options section, each option is identified with a 2-byte option code and a 2-byte length field. A full list of message types and option codes that can appear in these fields can be found here: *http://www.iana.org/assignments/dhcpv6-parameters/dhcpv6-parameters.xhtml.*

DHCPv6 accomplishes the same goal as DHCP, but to understand the flow of DHCPv6 communication, we must replace our DORA acronym with a new one, SARR. This process is illustrated in Figure 9-8, which represents a client that is currently out of lease.

Figure 9-8: The DHCPv6 SARR out-of-lease renewal process

The SARR process has four steps:

1. **Solicit**: An initial packet is sent from a client to a special multicast address (ff02::1:2) to attempt to locate available DHCPv6 servers on the network.
2. **Advertise**: An available server responds directly to the client to indicate that it is available to provide addressing and configuration information.
3. **Request**: The client sends a formal request for configuration information to the server via multicast.
4. **Reply**: The server sends all available requested configuration information directly to the client, and the process is complete.

A summary of this process is shown in Figure 9-9, which is taken from the file *dhcp6_outlease_acquisition.pcapng*. In this example, we see the SARR process in action as a new host on the network (fe80::20c:29ff:fe5e:7744) receives configuration information from a DHCPv6 server (fe80::20c:29ff :fe1f:a755). Each packet represents one step of the SARR process, with the initial solicit and advertise packets tied together using the transaction ID 0x9de03f and the request and reply packets associated with the transaction ID 0x2d1603. While it isn't shown in the figure, this communication takes place over ports 546 and 547, which are the standard ports used by DHCPv6.

No	Time	Source	Destination	Protocol	Length	Info
1	0.…	fe80::20c:29ff:fe5e:7744	ff02::1:2	DHCPv6	118	Solicit XID: 0x9de03f CID: 000100011def69bd000c295e7744
2	0.…	fe80::20c:29ff:fe1f:a755	fe80::20c:29ff:fe5e:7744	DHCPv6	166	Advertise XID: 0x9de03f CID: 000100011def69bd000c295e7744 IAA: 2001:db8:1:2::1002
3	1.…	fe80::20c:29ff:fe5e:7744	ff02::1:2	DHCPv6	164	Request XID: 0x2d1603 CID: 000100011def69bd000c295e7744 IAA: 2001:db8:1:2::1002
4	1.…	fe80::20c:29ff:fe1f:a755	fe80::20c:29ff:fe5e:7744	DHCPv6	166	Reply XID: 0x2d1603 CID: 000100011def69bd000c295e7744 IAA: 2001:db8:1:2::1002

Figure 9-9: A client obtaining an IPv6 address via DHCPv6

Overall, the packet structure of DHCPv6 traffic looks a lot different, but most of the same concepts apply. The process still requires some form of DHCP server discovery and a formal retrieval of configuration information. Those transactions are all tied together via transaction identifiers in each pair of packets exchanged between the client and server. IPv6 addressing can't be supported by traditional DHCP mechanisms, so if you have devices getting IPv6 addresses automatically from a server on your network, it's likely that you're already running DHCPv6 services on your network. If you'd like to compare DHCP and DHCPv6 further, I recommend opening the packet captures discussed in this chapter side by side and stepping through them.

Domain Name System (DNS)

The Domain Name System (DNS) is one of the most crucial internet protocols because it is the proverbial molasses that holds the bread together. DNS ties domain names, such as *www.google.com*, to IP addresses, such as 74.125.159.99. When we want to communicate with a networked device and we don't know its IP address, we access that device via its DNS name.

DNS servers store a database of *resource records* of IP address–to–DNS name mappings, which they share with clients and other DNS servers.

NOTE *Because the architecture of DNS servers is complicated, we'll just look at some common types of DNS traffic. You can review the various DNS-related RFCs at* https://www.isc.org/community/rfcs/dns/.

DNS Packet Structure

As you can see in Figure 9-10, the DNS packet structure is somewhat different from that of the packet types we've discussed previously. The following fields can be present within a DNS packet:

DNS ID Number Used to associate DNS queries with DNS responses

Query/Response (QR) Denotes whether the packet is a DNS query or response

OpCode Defines the type of query contained in the message

Authoritative Answers (AA) If this value is set in a response packet, indicates that the response is from a name server with authority over the domain

Truncation (TC) Indicates that the response was truncated because it was too large to fit within the packet

Recursion Desired (RD) When this value is set in a query, indicates that the DNS client requests a recursive query if the target name server doesn't contain the information requested

Recursion Available (RA) If this value is set in a response, indicates that the name server supports recursive queries

Domain Name System (DNS)											
Offsets	Octet	0	1	2					3		
Octet	Bit	0–7	8–15	16–23					24–31		
0	0	DNS ID Number		QR	OpCode	AA	TC	RD	RA	Z	RCode
4	32	Question Count		Answer Count							
8	64	Name Server (Authority) Record Count		Additional Records Count							
12+	96+	Questions Section		Answers Section							
		Authority Section		Additional Information Section							

Figure 9-10: The DNS packet structure

Reserved (Z) Defined by RFC 1035 to be set as all zeros; however, sometimes it's used as an extension of the RCode field

Response Code (RCode) Used in DNS responses to indicate the presence of any errors

Question Count The number of entries in the Questions Section

Answer Count The number of entries in the Answers Section

Name Server (Authority) Record Count The number of name server resource records in the Authority Section

Additional Records Count The number of other resource records in the Additional Information Section

Questions Section Variable-sized section that contains one or more queries for information to be sent to the DNS server

Answers Section Variable-sized section that carries one or more resource records that answer queries

Authority Section Variable-sized section that contains resource records that point to authoritative name servers that can be used to continue the resolution process

Additional Information Section Variable-sized section that contains resource records that hold additional information related to the query that is not absolutely necessary to answer the query

A Simple DNS Query

dns_query_response .pcapng

DNS functions in a query-response format. A client wishing to resolve a DNS name to an IP address sends a *query* to a DNS server, and the server sends the requested information in its *response*. In its simplest form, this process takes two packets, as can be seen in the capture file *dns_query_response.pcapng*.

The first packet, shown in Figure 9-11, is a DNS query sent from the client 192.168.0.114 to the server 205.152.37.23 on port 53, which is the standard port used by DNS.

Figure 9-11: The DNS query packet

When you begin examining the headers in this packet, you'll see that DNS also relies on UDP ❶.

In the DNS portion of the packet, you can see that smaller fields near the beginning of the packet are condensed by Wireshark into a single Flags section. Expand this section, and you'll see that the message is indeed a standard query ❷, that it is not truncated, and that recursion is desired (we'll cover recursion shortly). Only a single question is identified, which can be found by expanding the Queries section. There, you can see the query is for the name *wireshark.org* for a host (type `A`) internet (`IN`) address ❸. This packet is basically asking, "Which IP address is associated with the *wireshark.org* domain?"

The response to this request is in packet 2, as shown in Figure 9-12. Because this packet has an identical identification number ❶, we know that it contains the correct response to the original query.

The Flags section confirms that this is a response and that recursion is available if necessary ❷. This packet contains only one question and one resource record ❸, because it includes the original question in conjunction with its answer. Expanding the Answers section gives us the response to the query: the IP address of *wireshark.org* is 128.121.50.122 ❹. With this information, the client can now construct IP packets and begin communicating with *wireshark.org*.

```
Wireshark · Packet 2 · dns_query_response
> Frame 2: 89 bytes on wire (712 bits), 89 bytes captured (712 bits)
> Ethernet II, Src: D-Link_21:99:4c (00:05:5d:21:99:4c), Dst: HonHaiPr_6e:8b:24 (00:16:ce:6e:8b:24)
> Internet Protocol Version 4, Src: 205.152.37.23, Dst: 192.168.0.114
> User Datagram Protocol, Src Port: 53 (53), Dst Port: 1060 (1060)
v Domain Name System (response)
    [Request In: 1]
    [Time: 0.091164000 seconds]
    Transaction ID: 0x180f ❶
  v Flags: 0x8180 Standard query response, No error ❷
      1... .... .... .... = Response: Message is a response
      .000 0... .... .... = Opcode: Standard query (0)
      .... .0.. .... .... = Authoritative: Server is not an authority for domain
      .... ..0. .... .... = Truncated: Message is not truncated
      .... ...1 .... .... = Recursion desired: Do query recursively
      .... .... 1... .... = Recursion available: Server can do recursive queries
      .... .... .0.. .... = Z: reserved (0)
      .... .... ..0. .... = Answer authenticated: Answer/authority portion was not authenticated by the server
      .... .... ...0 .... = Non-authenticated data: Unacceptable
      .... .... .... 0000 = Reply code: No error (0)
    Questions: 1
❸   Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v wireshark.org: type A, class IN
        Name: wireshark.org
        [Name Length: 13]
        [Label Count: 2]
        Type: A (Host Address) (1)
        Class: IN (0x0001)
  v Answers
    v wireshark.org: type A, class IN, addr 128.121.50.122 ❹
        Name: wireshark.org
        Type: A (Host Address) (1)
        Class: IN (0x0001)
        Time to live: 14400
        Data length: 4
        Address: 128.121.50.122
No.: 2 · Time: 0.091164 · Source: 205.152.37.23 · Destination: 192.168.0.114 · Protocol: DNS · Length: 89 · Info: Standard query response 0x180f A wireshark.org A 128.121.50.122
                                                                                                   Close   Help
```

Figure 9-12: The DNS response packet

DNS Question Types

The Type fields used in DNS queries and responses indicate the resource record type that the query or response is for. Some of the more common message/resource record types are listed in Table 9-2. You'll be seeing these types throughout normal traffic and in this book. (The list in Table 9-2 is brief and by no means exhaustive. To review all DNS resource record types, visit *http://www.iana.org/assignments/dns-parameters/*.)

Table 9-2: Common DNS Resource Record Types

Value	Type	Description
1	A	IPv4 host address
2	NS	Authoritative name server
5	CNAME	Canonical name for an alias
15	MX	Mail exchange

Value	Type	Description
16	TXT	Text string
28	AAAA	IPv6 host address
251	IXFR	Incremental zone transfer
252	AXFR	Full zone transfer

DNS Recursion

dns_recursivequery_client.pcapng, dns_recursivequery_server.pcapng

Due to the hierarchical nature of the internet's DNS structure, DNS servers must be able to communicate with each other in order to answer the queries submitted by clients. While we expect our internal DNS server to know the name-to-IP address mapping of our local intranet server, we can't expect it to know the IP address associated with Google or Dell.

When a DNS server needs to find an IP address, it queries another DNS server on behalf of the client making the request, in effect acting like a client. This process is called *recursion*.

To view the recursion process from both the DNS client and server perspectives, open the file *dns_recursivequery_client.pcapng*. This file contains a capture of a client's DNS traffic file in two packets. The first packet is the initial query sent from the DNS client 172.16.0.8 to its DNS server 172.16.0.102, as shown in Figure 9-13.

Figure 9-13: The DNS query with the Recursion desired bit set

When you expand the DNS portion of this packet, you'll see that this is a standard query for an `A` type record for the DNS name *www.nostarch.com* ❷. To learn more about this packet, expand the Flags section, and you'll see that recursion is desired ❶.

The second packet is what we would expect to see in response to the initial query, as shown in Figure 9-14.

```
Wireshark · Packet 2 · dns_recursivequery_client
> Frame 2: 92 bytes on wire (736 bits), 92 bytes captured (736 bits)
> Ethernet II, Src: Vmware_92:94:9f (00:0c:29:92:94:9f), Dst: HewlettP_bf:91:ee (00:25:b3:bf:91:ee)
> Internet Protocol Version 4, Src: 172.16.0.102, Dst: 172.16.0.8
> User Datagram Protocol, Src Port: 53 (53), Dst Port: 56125 (56125)
v Domain Name System (response)
    [Request In: 1]
    [Time: 0.183134000 seconds]
    Transaction ID: 0x8b34 ❶
  v Flags: 0x8180 Standard query response, No error
      1... .... .... .... = Response: Message is a response
      .000 0... .... .... = Opcode: Standard query (0)
      .... .0.. .... .... = Authoritative: Server is not an authority for domain
      .... ..0. .... .... = Truncated: Message is not truncated
      .... ...1 .... .... = Recursion desired: Do query recursively
      .... .... 1... .... = Recursion available: Server can do recursive queries
      .... .... .0.. .... = Z: reserved (0)
      .... .... ..0. .... = Answer authenticated: Answer/authority portion was not authenticated by the server
      .... .... ...0 .... = Non-authenticated data: Unacceptable
      .... .... .... 0000 = Reply code: No error (0)
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v www.nostarch.com: type A, class IN
        Name: www.nostarch.com
        [Name Length: 16]
        [Label Count: 3]
        Type: A (Host Address) (1)
        Class: IN (0x0001)
  v Answers
    v www.nostarch.com: type A, class IN, addr 72.32.92.4 ❷
        Name: www.nostarch.com
        Type: A (Host Address) (1)
        Class: IN (0x0001)
        Time to live: 3600
        Data length: 4
        Address: 72.32.92.4
No.: 2 · Time: 0.183134 · Source: 172.16.0.102 · Destination: 172.16.0.8 · Protocol: DNS · Length: 92 · Info: Standard query response 0x8b34 A www.nostarch.com A 72.32.92.4
                                                                                                  Close   Help
```

Figure 9-14: The DNS query response

This packet's transaction ID matches that of our query ❶, no errors are listed, and we receive the `A` type resource record associated with *www.nostarch.com* ❷.

We can see that this query was answered by recursion by listening to the DNS server's traffic when the recursion is taking place, as demonstrated in the file *dns_recursivequery_server.pcapng*. This file shows a capture of the traffic on the local DNS server when the query was initiated (Figure 9-15).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.…	172.16.0.8	172.16.0.102	DNS	76	Standard query 0x8b34 A www.nostarch.com
2	0.…	172.16.0.102	4.2.2.1	DNS	76	Standard query 0xf34d A www.nostarch.com
3	0.…	4.2.2.1	172.16.0.102	DNS	92	Standard query response 0xf34d A www.nostarch.com A 72.32.92.4
4	0.…	172.16.0.102	172.16.0.8	DNS	92	Standard query response 0x8b34 A www.nostarch.com A 72.32.92.4

Figure 9-15: DNS recursion from the server's perspective

The first packet is the same initial query we saw in the previous capture file. At this point, the DNS server has received the query, checked its local database, and realized it doesn't know the answer to the question of which IP address goes with the DNS name (*www.nostarch.com*). Because the packet was sent with the Recursion desired bit set, the DNS server can ask another DNS server this question in an attempt to locate the answer, as you can see in the second packet.

In the second packet, the DNS server at 172.16.0.102 transmits a new query to 4.2.2.1 ❶, which is the server to which it is configured to forward upstream requests, as shown in Figure 9-16. This query mirrors the original one, effectively turning the DNS server into a client. We can tell that this is a new query because the transaction ID number differs from the transaction ID number in the previous capture file ❷.

```
Wireshark · Packet 2 · dns_recursivequery_server
> Frame 2: 76 bytes on wire (608 bits), 76 bytes captured (608 bits)
> Ethernet II, Src: Vmware_92:94:9f (00:0c:29:92:94:9f), Dst: CiscoInc_31:07:33 (00:26:0b:31:07:33)
> Internet Protocol Version 4, Src: 172.16.0.102, Dst: 4.2.2.1 ❶
> User Datagram Protocol, Src Port: 62570 (62570), Dst Port: 53 (53)
v Domain Name System (query)
    [Response In: 3]
    Transaction ID: 0xf34d ❷
  > Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v www.nostarch.com: type A, class IN
        Name: www.nostarch.com
        [Name Length: 16]
        [Label Count: 3]
        Type: A (Host Address) (1)
        Class: IN (0x0001)
No.: 2 · Time: 0.000379 · Source: 172.16.0.102 · Destination: 4.2.2.1 · Protocol: DNS · Length: 76 · Info: Standard query 0xf34d A www.nostarch.com
                                                                                    Close    Help
```

Figure 9-16: The recursive DNS query

Once this packet is received by server 4.2.2.1, the local DNS server receives the response shown in Figure 9-17.

```
Wireshark · Packet 3 · dns_recursivequery_server
> Frame 3: 92 bytes on wire (736 bits), 92 bytes captured (736 bits)
> Ethernet II, Src: CiscoInc_31:07:33 (00:26:0b:31:07:33), Dst: Vmware_92:94:9f (00:0c:29:92:94:9f)
> Internet Protocol Version 4, Src: 4.2.2.1, Dst: 172.16.0.102
> User Datagram Protocol, Src Port: 53 (53), Dst Port: 62570 (62570)
v Domain Name System (response)
    [Request In: 2]
    [Time: 0.182223000 seconds]
    Transaction ID: 0xf34d
  > Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 1
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v www.nostarch.com: type A, class IN
        Name: www.nostarch.com
        [Name Length: 16]
        [Label Count: 3]
        Type: A (Host Address) (1)
        Class: IN (0x0001)
  v Answers
    v www.nostarch.com: type A, class IN, addr 72.32.92.4
        Name: www.nostarch.com
        Type: A (Host Address) (1)
        Class: IN (0x0001)
        Time to live: 3600
        Data length: 4
        Address: 72.32.92.4
No.: 3 · Time: 0.182602 · Source: 4.2.2.1 · Destination: 172.16.0.102 · Protocol: DNS · Length: 92 · Info: Standard query response 0xf34d A www.nostarch.com A 72.32.92
                                                                                    Close   Help
```

Figure 9-17: Response to the recursive DNS query

Having received this response, the local DNS server can transmit the fourth and final packet to the DNS client with the information requested.

Although this example shows only one layer of recursion, recursion can occur many times for a single DNS request. Here, we received an answer from the DNS server at 4.2.2.1, but that server could have retransmitted the query recursively to another server in order to find the answer. A simple query can travel all over the world before it finally gets a correct response. Figure 9-18 illustrates the recursive DNS query process.

Figure 9-18: A recursive DNS query

DNS Zone Transfers

dns_axfr.pcapng

A *DNS zone* is the namespace (or group of DNS names) that a DNS server has been delegated to manage. For instance, Emma's Diner might have one DNS server responsible for *emmasdiner.com*. In that case, devices both inside and outside Emma's Diner wishing to resolve *emmasdiner.com* to an IP address would need to contact that DNS server as the authority for that zone. If Emma's Diner were to grow, it could add a second DNS server to handle the email portion of its DNS namespace only, say *mail.emmasdiner.com*, and that server would be the authority for that mail subdomain. Additional DNS servers might be added for subdomains as necessary, as shown in Figure 9-19.

Figure 9-19: DNS zones divide responsibility for namespaces.

A *zone transfer* occurs when zone data is transferred between two devices, typically out of desire for redundancy. For example, in organizations with multiple DNS servers, administrators commonly configure a secondary DNS server to maintain a copy of the primary server's DNS zone information in case the primary server becomes unavailable. There are two types of zone transfers:

Full zone transfer (AXFR) These types of transfers send an entire zone between devices.

Incremental zone transfer (IXFR) These types of transfers send only a portion of the zone information.

The file *dns_axfr.pcapng* contains an example of a full zone transfer between the hosts 172.16.16.164 and 172.16.16.139. When you first look at this file, you may wonder whether you've opened the right one, because rather than UDP packets, you see TCP packets. Although DNS relies on

UDP, it uses TCP for certain tasks, such as zone transfers, because TCP is more reliable for the amount of data being transferred. The first three packets in this capture file are the TCP three-way handshake.

The fourth packet begins the zone transfer request between 172.16.16.164 and 172.16.16.139. This packet doesn't contain any DNS information. It's marked as a "TCP segment of a reassembled PDU" because the data sent in the zone transfer request packet was sent in multiple packets. Packets 4 and 6 contain the packet's data. Packet 5 is the acknowledgment that packet 4 was received. These packets are displayed in this manner because of the way Wireshark parses and displays TCP packets for easier readability. For our purposes, we can reference packet 6 as the complete DNS zone transfer request, as shown in Figure 9-20.

Figure 9-20: DNS full zone transfer request

The zone transfer request is a standard query ❶, but instead of requesting a single record type, it requests the AXFR type ❷, meaning that it wishes to receive the entire DNS zone from the server. The server responds with the zone records in packet 7, as shown in Figure 9-21. As you can see, the zone transfer contains quite a bit of data, and this is one of the simpler examples! With the zone transfer complete, the capture file ends with the TCP connection teardown process.

WARNING *The data contained in a zone transfer can be very dangerous in the wrong hands. For example, by enumerating a single DNS server, you can map a network's entire infrastructure.*

Figure 9-21: The DNS full zone transfer occurring

Hypertext Transfer Protocol (HTTP)

The Hypertext Transfer Protocol is the delivery mechanism of the World Wide Web, allowing web browsers to connect to web servers to view web pages. In most organizations, HTTP represents, by far, the highest percentage of traffic seen going across the wire. Every time you do a Google search, send a tweet, or check University of Kentucky basketball scores on *http://www.espn.com/*, you're using HTTP.

We won't look at the packet structures for an HTTP transfer because there are so many different implementations of the HTTP protocol that the structure may vary wildly. Because of this variance, that exercise is left to you. Here, we'll look at some practical applications of HTTP such as retrieving and posting content.

Browsing with HTTP

http_google.pcapng

HTTP is most commonly used to browse web pages on a web server using a browser. The capture file *http_google.pcapng* shows such an HTTP transfer, using TCP as the transport layer protocol. Communication begins with a three-way handshake between the client 172.16.16.128 and the Google web server 74.125.95.104.

Once communication is established, the first packet is marked as an HTTP packet from the client to the server, as shown in Figure 9-22.

Figure 9-22: The initial HTTP `GET` request packet

The HTTP packet is delivered over TCP to the server's port 80 ❶, the standard port for HTTP communication (several other ports are often used as well, such as 8080 and 8888).

HTTP packets are identified by one of eight request methods as defined in HTTP specification version 1.1 (see *http://www.iana.org/assignments/http-methods/http-methods.xhtml*), which indicate the action the packet's transmitter will perform on the receiver. As shown in Figure 9-22, this packet identifies its method as `GET`, its request Uniform Resource Indicator (URI) as `/`, and the request version as `HTTP/1.1` ❷. This information tells us that the client is sending a request to download (`GET`) the root web directory (`/`) of the web server using version 1.1 of HTTP.

Next, the host sends information about itself to the web server. This information includes things such as the browser (User-Agent) being used, languages accepted by the browser (Accept-Languages), and cookie information (at the bottom of the capture). The server can use this information to determine which data to return to the client in order to ensure compatibility.

When the server receives the HTTP `GET` request in packet 4, it responds with a TCP ACK, acknowledging the packet, and begins transmitting the requested data from packets 6 to 11. HTTP is used only to issue application-layer commands between the client and server. Why do all these HTTP packets show up as TCP under the protocol heading in the packet list? When data transfer begins, the Wireshark packet list window will identify those packets as TCP instead of HTTP since no HTTP request/response

headers are present in those individual packets. Thus, where data transfer is occurring, you see TCP instead of HTTP in the Protocol column. Nonetheless, this is still part of the HTTP communication process.

Data is sent from the server in packets 6 and 7, an acknowledgment from the client in packet 8, two more data packets in packets 9 and 10, and another acknowledgment in packet 11, as shown in Figure 9-23. All of these packets are shown in Wireshark as TCP segments, rather than as HTTP packets, although HTTP is still responsible for their transmission.

No.	Time	Source	Destination	Protocol	Length	Info
6	0.…	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]
7	0.…	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]
8	0.…	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082692395 Ack=2775580186 Win=16872 Len=0
9	0.…	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]
10	0.…	74.125.95.104	172.16.16.128	TCP	156	[TCP segment of a reassembled PDU]
11	0.…	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082692395 Ack=2775581694 Win=16872 Len=0

Figure 9-23: TCP transmitting data between the client browser and web server

Once the data is transferred, Wireshark reassembles the data stream for viewing, as shown in Figure 9-24.

```
Wireshark · Packet 12 · http_google
> Frame 12: 591 bytes on wire (4728 bits), 591 bytes captured (4728 bits)
> Ethernet II, Src: D-Link_21:99:4c (00:05:5d:21:99:4c), Dst: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a)
> Internet Protocol Version 4, Src: 74.125.95.104, Dst: 172.16.16.128
> Transmission Control Protocol, Src Port: 80 (80), Dst Port: 1606 (1606), Seq: 2775581694, Ack: 2082692395, Len: 537
> [5 Reassembled TCP Segments (4857 bytes): #6(1406), #7(1406), #9(1406), #10(102), #12(537)]
v Hypertext Transfer Protocol
  v HTTP/1.1 200 OK\r\n
    > [Expert Info (Chat/Sequence): HTTP/1.1 200 OK\r\n]
      Request Version: HTTP/1.1
    ❶ Status Code: 200
      Response Phrase: OK
    Date: Tue, 09 Feb 2010 01:18:37 GMT\r\n
    Expires: -1\r\n
    Cache-Control: private, max-age=0\r\n
    Content-Type: text/html; charset=UTF-8\r\n
    Content-Encoding: gzip\r\n
    Server: gws\r\n
  v Content-Length: 4633\r\n
      [Content length: 4633]
    X-XSS-Protection: 0\r\n
    \r\n
    [HTTP response 1/1]
    [Time since request: 0.104147000 seconds]
    [Request in frame: 4]
    Content-encoded entity body (gzip): 4633 bytes -> 11308 bytes
> Line-based text data: text/html
No.: 12 · Time: 0.134395 · Source: 74.125.95.104 · Destination: 172.16.16.128 · Protocol: HTTP · Length: 591 · Info: HTTP/1.1 200 OK (text/html)
                                                                                    Close   Help
```

Figure 9-24: Final HTTP packet with response code 200

NOTE *In many instances, you won't be able to see readable HTML data when browsing through the packet list because that data is gzip compressed to increase bandwidth efficiency. This is signified by the Content-Encoding field in the HTTP response from the web server. It's only when you view the full stream that the data is decoded and easily readable.*

HTTP uses a number of predefined response codes to indicate the results of a request method. In this example, we see a packet with status code 200 ❶, which indicates a successful request method. The packet also includes a timestamp and some additional information about the encoding of the content and configuration parameters of the web server. When the client receives this packet, the transaction is complete.

Posting Data with HTTP

http_post.pcapng

Now that we have looked at the process of downloading data from a web server, let's turn our attention to uploading data. The file *http_post.pcapng* contains a very simple example of an upload: a user posting a comment to a website. After the initial three-way handshake, the client (172.16.16.128) sends an HTTP packet to the web server (69.163.176.56), as shown in Figure 9-25.

```
Wireshark · Packet 4 · http_post
> Frame 4: 1175 bytes on wire (9400 bits), 1175 bytes captured (9400 bits)
> Ethernet II, Src: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a), Dst: D-Link_21:99:4c (00:05:5d:21:99:4c)
> Internet Protocol Version 4, Src: 172.16.16.128, Dst: 69.163.176.56
> Transmission Control Protocol, Src Port: 1989 (1989), Dst Port: 80 (80), Seq: 2808074211, Ack: 3740859985, Len: 1121
v Hypertext Transfer Protocol
  v POST /wp-comments-post.php HTTP/1.1\r\n
    > [Expert Info (Chat/Sequence): POST /wp-comments-post.php HTTP/1.1\r\n]
      Request Method: POST ❶
      Request URI: /wp-comments-post.php ❷
      Request Version: HTTP/1.1
    Host: www.chrissanders.org\r\n
    User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.1.7) Gecko/20091221 Firefox/3.5.7\r\n
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8\r\n
    Accept-Language: en-us,en;q=0.5\r\n
    Accept-Encoding: gzip,deflate\r\n
    Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7\r\n
    Keep-Alive: 300\r\n
    Connection: keep-alive\r\n
    Referer: http://www.chrissanders.org/?p=310\r\n
    > [truncated]Cookie: __utma=84195659.500695863.1261144042.1265668706.1265682737.20; __utmz=84195659.1264688282.12.…
    Content-Type: application/x-www-form-urlencoded\r\n
    > Content-Length: 179\r\n
    \r\n
    [Full request URI: http://www.chrissanders.org/wp-comments-post.php]
    [HTTP request 1/2]
    [Response in frame: 6]
    [Next request in frame: 7]
v HTML Form URL Encoded: application/x-www-form-urlencoded ❸
  v Form item: "author" = "Chris Sanders"
      Key: author
      Value: Chris Sanders
  v Form item: "email" = "chris@chrissanders.org"
      Key: email
      Value: chris@chrissanders.org
  v Form item: "url" = "http://www.chrissanders.org"
      Key: url
      Value: http://www.chrissanders.org
No.: 4 · Time: 0.081100 · Source: 172.16.16.128 · Destination: 69.163.176.56 · Protocol: HTTP · Length: 1175 · Info: POST /wp-comments-post.php HTTP/1.1 (application/x-www-form-urlencoded)
                                                                                    Close   Help
```

Figure 9-25: The HTTP `POST` packet

This packet uses the `POST` method ❶ to upload data to a web server for processing. The `POST` method used here specifies the URI `/wp-comments`

-post.php ❷ and the HTTP version of HTTP/1.1. To see the contents of the data posted, expand the HTML Form URL Encoded portion of the packet ❸.

Once the data is transmitted in this POST, an ACK packet is sent. As shown in Figure 9-26, the server responds with packet 6, transmitting the response code 302 ❶, which means "found."

```
Wireshark · Packet 6 · http_post
> Frame 6: 964 bytes on wire (7712 bits), 964 bytes captured (7712 bits)
> Ethernet II, Src: D-Link_21:99:4c (00:05:5d:21:99:4c), Dst: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a)
> Internet Protocol Version 4, Src: 69.163.176.56, Dst: 172.16.16.128
> Transmission Control Protocol, Src Port: 80 (80), Dst Port: 1989 (1989), Seq: 3740859985, Ack: 2808075332, Len: 910
v Hypertext Transfer Protocol
  v HTTP/1.1 302 Found\r\n
    > [Expert Info (Chat/Sequence): HTTP/1.1 302 Found\r\n]
      Request Version: HTTP/1.1
      Status Code: 302 ❶
      Response Phrase: Found
    Date: Tue, 09 Feb 2010 02:30:26 GMT\r\n
    Server: Apache\r\n
    X-Powered-By: PHP/4.4.9\r\n
    Expires: Wed, 11 Jan 1984 05:00:00 GMT\r\n
    Cache-Control: no-cache, must-revalidate, max-age=0\r\n
    Pragma: no-cache\r\n
    Set-Cookie: comment_author_0d7dc802882e903c170f35a2d747915b=Chris+Sanders; expires=Saturday, 22-Jan-11 07:50:27 GMT; path=/\r\n
    Set-Cookie: comment_author_email_0d7dc802882e903c170f35a2d747915b=chris%40chrissanders.org; expires=Saturday, 22-Jan-11 07:50:27 GMT; path=/\r\n
    Set-Cookie: comment_author_url_0d7dc802882e903c170f35a2d747915b=http%3A%2F%2Fwww.chrissanders.org; expires=Saturday, 22-Jan-11 07:50:27 GMT; path=/\r\n
    Last-Modified: Tue, 09 Feb 2010 02:30:27 GMT\r\n
❷   Location: http://www.chrissanders.org/?p=310&cpage=1#comment-103002\r\n
    Vary: Accept-Encoding\r\n
    Content-Encoding: gzip\r\n
    v Content-Length: 20\r\n
        [Content length: 20]
    Keep-Alive: timeout=2, max=100\r\n
    Connection: Keep-Alive\r\n
    Content-Type: text/html\r\n
    \r\n
    [HTTP response 1/2]
    [Time since request: 1.356727000 seconds]
    [Request in frame: 4]
    [Next request in frame: 7]
    [Next response in frame: 18]
    Content-encoded entity body (gzip): 20 bytes -> 0 bytes
No.: 6 · Time: 1.437827 · Source: 69.163.176.56 · Destination: 172.16.16.128 · Protocol: HTTP · Length: 964 · Info: HTTP/1.1 302 Found (text/html)[Malformed Packet]
                                                                                    Close   Help
```

Figure 9-26: HTTP response 302 is used to redirect.

The 302 response code is a common means of redirection in the HTTP world. The Location field in this packet specifies where the client is to be directed ❷. In this case, that location is on the originating web page where the comment was posted. The client performs a new GET request to retrieve content at the new location, which it sends over the next several packets. Finally, the server transmits status code 200, and the communication ends.

Simple Mail Transfer Protocol (SMTP)

If web browsing is the most common activity a user will participate in, sending and receiving email is probably in second place. The *Simple Mail Transfer Protocol (SMTP)*, used by platforms such as Microsoft Exchange and Postfix, is the standard for sending email.

As with HTTP, the structure of an SMTP packet can vary based on the implementation and the set of features supported by the client and server. In this section, we'll review some of the basic functionality of SMTP by examining what sending email looks like at the packet level.

Sending and Receiving Email

The architecture supporting email is similar to the US Postal Service. When you write a letter, you put it in your mailbox, a postal worker picks it up, and it's transported to a post office where it's sorted. From there, the letter is either delivered to another mailbox serviced by that same post office or transported to another post office that is responsible for delivering it. A letter may traverse multiple post offices or even "hub" offices designed exclusively to distribute to post offices in specific geographic regions. This flow of information is illustrated in Figure 9-27.

Figure 9-27: Sending a letter via the postal service

Delivering email works in a very similar manner, but the terminology is a bit different. At the individual user level, the physical mailbox is replaced by a digital mailbox that is responsible for storing and facilitating the sending and receiving of your email. You access this mailbox with a *mail user agent (MUA)*, which is an email client like Microsoft Outlook or Mozilla Thunderbird.

When you send a message, it's sent from your MUA to a *mail transfer agent (MTA)*. The MTA is often referred to as the mail server, with popular mail server applications being Microsoft Exchange or Postfix. If the email being sent is destined for the same domain it came from, the MTA can associate it with the recipient mailbox without any further communication. If the email

is being sent to another domain, the MTA must use DNS to find the location address of the recipient mail server, then transmit the message to it. It's worth noting that the mail server is often made up of other components like a Mail Delivery Agent (MDA) or a Mail Submission Agent (MSA), but from the network standpoint, we'll usually only be interested in the concept of a client and a server. This basic overview is illustrated in Figure 9-28.

Figure 9-28: Sending an email via SMTP

For simplicity's sake, we'll refer to the MUA as the email client and the MTA as the email server.

Tracking an Email Message

With a basic understanding of how email messages are transmitted, we can begin to look at packets that represent this process. Let's start with the scenario outlined in Figure 9-29.

Figure 9-29: Tracking an email from sender to recipient

There are three steps in this scenario:

1. A user sends a message from their workstation (172.16.16.225). The email client transmits the message via SMTP to the local email server (172.16.16.221 / *skynet.local* domain).
2. The local email server receives the message and transmits it to a remote email server (172.16.16.231 / *cyberdyne.local* domain) via SMTP.
3. The remote email server receives the message and associates it with the appropriate mailbox. The email client on a user's workstation (172.16.16.235) retrieves this message using the IMAP protocol.

Step 1: Client to Local Server

mail_sender_client_1.pcapng

We'll begin stepping through this process by reviewing step 1, which is represented by *mail_sender_client_1.pcapng*. The file begins when the user clicks the Send button in their email client, initiating the TCP handshake between their workstation and the local email server in packets 1 through 3.

NOTE *You can ignore any* `ETHERNET FRAME CHECK SEQUENCE INCORRECT` *errors observed while analyzing the packet captures in this section. They are an artifact of the lab environment in which these were created.*

Once a connection is established, SMTP takes over and begins the work of transmitting the user's message to the server. You could examine each SMTP request and response individually by scrolling through each packet and viewing the SMTP section of the Packet Details window, but there is an easier way. Since SMTP is a simple transactional protocol and our example is in clear text, you can follow the TCP stream to view the entire transaction in one window. Do this by right-clicking any packet in the capture and selecting **Follow ▸ TCP Stream**. The resulting stream is shown in Figure 9-30.

With a connection established, the email server sends a service banner to the client in packet 4 to indicate that it is ready to receive a command. In this case, it identifies itself as a Postfix server running on the Ubuntu Linux operating system ❶. It also identifies that it is capable of receiving *Extended SMTP (ESMTP)* commands. ESMTP is an extension to the SMTP specification that allows for additional commands to be used during mail transmission.

The email client responds by issuing the `EHLO` command in packet 5 ❷. `EHLO` is the "Hello" command used to identify the sending host when ESMTP is supported. If ESMTP is not available, the client will revert to the `HELO` command to identify itself. In this example, the sender is identified by its IP address, although a DNS name can be used as well.

In packet 7, the server responds with a list of items that include things like `VRFY`, `STARTTLS`, and `SIZE 10240000` ❸. This list, which reflects commands supported by the SMTP server, is provided so that the client knows what commands it is allowed to use when transmitting the message. This feature negotiation occurs at the beginning of every SMTP transaction before a message is sent. The transmission of the message begins at packet 8 and makes up most of the remainder of this capture.

Figure 9-30: Viewing the TCP stream from the email client to the local server

SMTP is governed by simple commands and parameter values sent from the client, followed by a response code from the server. This is very similar to protocols like HTTP and TELNET and is designed for simplicity. An example request and reply can be seen in packets 8 and 9, where the client issues the `MAIL` command with the parameter `FROM:<sanders@skynet.local> SIZE=556` ❹, and the server responds with response code 250 (Requested mail action okay, completed) and the `2.1.0 Ok` parameter. Here, the client identifies the sender's email address and the size of the message, and the server responds saying that this data was received and is acceptable. A similar transaction happens again in packets 10 and 11, where the client issues the `RCPT` command with the parameter `TO:<sanders@cyberdyne.local>` ❺, and the server responds with another `250 2.1.5 Ok` code.

NOTE *If you'd like to review all the available SMTP commands and parameters, you can do so here:* http://www.iana.org/assignments/mail-parameters/mail-parameters.xhtml. *If you'd like to review the available response codes, that can be done here:* https://www.iana.org/assignments/smtp-enhanced-status-codes/smtp-enhanced-status-codes.xml.

All that is left is to transmit the message itself. The client initiates this process in packet 12 by issuing the `DATA` command. The server responds with code 354 along with a message ❻, which indicates that the server has created a buffer for the message and tells the client to begin transmitting. The line containing the code 354 tells the client to send a dot (`<CR><LF>.<CR><LF>`) to mark the end of the transmission. The message is transmitted in plaintext, and a response code indicating successful transmission is sent. You'll notice the inclusion of some additional information with the message text, including the date, the content type and encoding, and the user agent associated with the transmission. This tells you that the end user who sent this message was using Mozilla Thunderbird ❼.

With transmission complete, the SMTP connection is terminated by the email client by issuing the `QUIT` command with no parameters in packet 18. The server responds in packet 19 with the response code 221 (*<domain>* service closing transmission channel) and the `2.0.0 Bye` parameter ❽. The TCP connection is torn down gracefully in packets 20–23.

mail_sender_server_2.pcapng

Step 2: Local Server to Remote Server

Next we'll examine the same scenario from the perspective of the local email server responsible for the *skynet.local* domain; its address is 172.16.16.221. This capture can be found in the file *mail_sender_server_2.pcapng*, which was taken directly from the email server. As you might expect, the first 20 or so packets mirror the capture in step 1, because they are the same packets captured from another source.

If the sent message was destined for another mailbox in the *skynet.local* domain, we wouldn't see any more SMTP traffic; instead, we would see the retrieval of the message from an email client with the POP3 or IMAP protocol. However, since this message is destined for the *cyberdyne.local* domain, the local SMTP server must transmit the message to the remote SMTP server responsible for that domain. This process begins in packet 22 with a TCP handshake between the local server 172.16.16.221 and the remote mail server 172.16.16.231.

NOTE *In a real-world scenario, an email server locates another server by using a special DNS record type known as a* mail exchange (MX) record. *Since this scenario was created in a lab and the IP address of the remote email server was preconfigured on the local server, we won't see that traffic here. If you're troubleshooting email delivery, you should consider the potential for DNS issues along with email-specific protocol issues.*

With a connection established, we can see in the Packet List window that SMTP is used to deliver the message to the remote server. You can better view this conversation by following the TCP stream for the transaction. It is shown in Figure 9-31. If you need help isolating this connection, apply the filter **`tcp.stream == 1`** in the filter bar.

Figure 9-31: Viewing the TCP stream from the local email server to the remote email server

This transaction is nearly identical to the one in Figure 9-30. Essentially, the message is just being transmitted between servers. The remote server identifies itself as `mail02` ❶, the local server identifies itself as `mail01` ❷, a list of support commands is shared ❸, and the message is transferred in its entirety with a bit of additional data from the previous transaction prepended to the message above the To line ❹. This all occurs between packets 27 and 35, with a TCP teardown closing the communication channel.

The server ultimately doesn't care whether the message is coming from an email client or another SMTP server, so all the same rules and procedures apply (barring any type of access control restrictions). In the real world, a local email server and a remote email server might not support the same feature set or might be based on entirely different platforms. This is why the initial SMTP communication is so important; it allows the recipient

server to transmit its supported feature set to the sender. When an SMTP client or server is aware of the supported features of the recipient server, the SMTP commands can be adjusted so that the message can be transmitted effectively. This capability allows SMTP to be widely usable between any number of client and server technologies, and this is why you don't have to know much about the network infrastructure of the recipient when sending an email.

Step 3: Remote Server to Remote Client

mail_receiver_server_3.pcapng

At this point, our message has reached the remote server responsible for delivering emails to mailboxes in the *cyberdyne.local* domain. We'll now look at a packet capture taken from the perspective of the remote server, *mail_receiver_server_3.pcapng*, shown in Figure 9-32.

Figure 9-32: Viewing the TCP stream from the local email server to the remote email server

Once again, the first 15 packets in this capture look very familiar, as they are a representation of the same message being exchanged, with the source address representing the local email server ❶ and the destination address representing the remote email server ❷. Once this sequence is completed, the SMTP server can associate the message with the appropriate mailbox so that the intended recipient can retrieve it via their email client.

As mentioned earlier, SMTP is primarily used for sending email and is by far the most common protocol for that purpose. Retrieving email from a mailbox on a server is a bit more open-ended, and because of different needs arising in that space, there are several protocols that are designed to support this task. The most prevalent are Post Office Protocol version 3 (POP3) and Internet Message Access Protocol (IMAP). In our example, the remote client retrieves messages from the email server using IMAP in packets 16–34.

We don't cover IMAP in this book, but in this example, it wouldn't do you a ton of good even if we did because the communication is encrypted. If you look at packet 21, you'll see the client (172.16.16.235) send the `STARTTLS` command to the email server (172.16.16.231) in packet 21 ❶, shown in Figure 9-33.

tcp.stream eq 1 | Expression... | +

No.	Time	Source	Destination	Protocol	Length	Info
16	11.748156	172.16.16.235	172.16.16.231	TCP	66	51147 → 143 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
17	11.748191	172.16.16.231	172.16.16.235	TCP	66	143 → 51147 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
18	11.748353	172.16.16.235	172.16.16.231	TCP	60	51147 → 143 [ACK] Seq=1 Ack=1 Win=65536 Len=0
19	11.755638	172.16.16.231	172.16.16.235	IMAP	178	Response: * OK [CAPABILITY IMAP4rev1 LITERAL+ SASL-IR LOGIN-REFERRALS ID ENABLE…
20	11.819470	172.16.16.235	172.16.16.231	TCP	60	51147 → 143 [ACK] Seq=1 Ack=125 Win=65536 Len=0
21	11.871697	172.16.16.235	172.16.16.231	IMAP	66	Request: 1 STARTTLS ❶
22	11.871722	172.16.16.231	172.16.16.235	TCP	54	143 → 51147 [ACK] Seq=125 Ack=13 Win=29312 Len=0
23	11.871904	172.16.16.231	172.16.16.235	IMAP	87	Response: 1 OK Begin TLS negotiation now.
24	11.890004	172.16.16.235	172.16.16.231	TLSv1.2	219	Client Hello
25	11.892786	172.16.16.231	172.16.16.235	TLSv1.2	1447	Server Hello, Certificate, Server Key Exchange, Server Hello Done
26	11.910176	172.16.16.235	172.16.16.231	TLSv1.2	212	Client Key Exchange, Change Cipher Spec, Hello Request, Hello Request ❷
27	11.911283	172.16.16.231	172.16.16.235	TLSv1.2	296	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
28	11.937139	172.16.16.235	172.16.16.231	TLSv1.2	97	Application Data
29	11.937295	172.16.16.231	172.16.16.235	TLSv1.2	238	Application Data ❸

Figure 9-33: The STARTTLS command indicates that the IMAP traffic will be encrypted.

This command informs the server that the client would like to retrieve messages securely using TLS encryption. A secure channel is negotiated between each endpoint in packets 24–27 ❷, and the message is retrieved securely via the *TLS (Transport Layer Security)* protocol in the remaining packets ❸. If you click any of these packets to view the data or attempt to follow the TCP stream (Figure 9-34), you'll find that the contents are unreadable, protecting the email from being intercepted by someone who might be attempting to hijack or sniff traffic maliciously.

With those final packets received, the process of sending a message from a user in one domain to a user in another domain is completed.

Figure 9-34: The IMAP traffic is encrypted as the client downloads the message.

Sending Attachments via SMTP

mail_sender_attachment.pcapng

SMTP was never intended to be a mechanism for transmitting files, but the ease of emailing a file means that it has become the primary sharing mechanism for many. Let's walk through a quick example of what sending a file looks like at the packet level using SMTP.

In the packet capture *mail_sender_attachment.pcapng*, a user is sending an email message from their client (172.16.16.225) to another user on the same network via a local SMTP mail server (172.16.16.221). The message contains a bit of text and includes an image file attachment.

Sending an attachment via SMTP is not too different from sending text. It's all just data to the server, and although some special encoding usually takes place, we still rely on the `DATA` command to get things where they're going. To see this in action, open the capture file and follow the TCP stream for the given SMTP transaction. This stream is pictured in Figure 9-35.

Figure 9-35: A user sending an attachment via SMTP

This example begins like the previous scenarios with service identification and an exchange of supported protocols. When the client is ready to transmit the message, it does so by providing the From and To addresses, and sending the `DATA` command instructs the server to open up a buffer to receive the information. This is where things get a little different.

In the previous example, the client transmitted the text directly to the server, and that was it. In this example, the client must send the plaintext message, as well as the binary data associated with the image attachment. To make this happen, it identifies its Content-Type as `multipart/mixed`, with a boundary of `----------050407080301000500070000` ❶. This tells the server that multiple types of data are being transmitted, each with their own unique MIME type and encoding, and that each type of data will be separated with the boundary value specified. Therefore, when another mail client receives this data, it will know how to interpret the data based on the boundaries and the unique MIME type and encoding specified in each chunk.

In our example, we have two unique parts of this message. The first is the mail text itself, which is identified by the content type `text/plain` ❷. After that, we see a boundary marker and the start of a new part of the message ❸. This part contains the image file and is identified by the content type `image/jpeg` ❹. It's also worth noting that the `Content-Transfer-Encoding` value is set to `base64` ❺, meaning that the data must be converted from base 64 to be parsed. The remainder of the transmission includes the encoded image file ❻.

Whatever you do, don't get this encoding confused with a security feature. Base 64 encoding is almost instantly reversible, and any attacker who intercepts this communication would be able to retrieve the image file without much effort. If you are interested in carving this image file out of the packet capture yourself, there is a similar scenario in which we carve an image from an HTTP-based file transfer in the Remote-Access Trojan section of Chapter 12. Once you've read that, flip back to this capture file and see if you can find out who the user's mysterious new coworker is.

Final Thoughts

This chapter has introduced the most common protocols you will encounter when examining traffic at the application layer. In the following chapters, we'll examine new protocols and additional features of the protocols we've covered here as we explore a wide range of real-world scenarios.

To learn more about individual protocols, read their associated RFCs or have a look at *The TCP/IP Guide* by Charles M. Kozierok (No Starch Press, 2005). Also, see the list of resources in Appendix A.

10

BASIC REAL-WORLD SCENARIOS

Beginning with this chapter, we'll dig into the meat of packet analysis as we use Wireshark to analyze real-world network problems. I'll introduce a series of problem scenarios by describing the context of the problem and providing the information that was available to the analyst at the time. Having laid the groundwork, we'll turn to analysis as I describe the method used to capture the appropriate packets and step you through the process of working toward a diagnosis. Once analysis is complete, I'll point toward potential solutions and give an overview of the lessons learned.

Throughout, remember that analysis is a very dynamic process. Thus, the methods I use to analyze each scenario may not be the same ones that you would use. Everyone approaches problem solving and reasoning through their own lens. The most important thing is that the result of the

analysis solves a problem, but even when it doesn't, it's critical to learn from failures as well. Experience is the thing we get when we don't get what we want, after all.

In addition, most problems discussed in this chapter can probably be solved with methods that don't necessarily involve a packet sniffer, but what's the fun in that? When I was first learning how to analyze packets, I found it helpful to examine typical problems in atypical ways by using packet analysis techniques, which is why I present these scenarios to you.

Missing Web Content

http_espn_fail .pcapng

In the first scenario we'll look at, our user is Packet Pete, a college basketball fan who doesn't keep late hours and usually misses the West Coast games. The first thing he does when he sits down at his workstation every morning is visit *http://www.espn.com/* for the previous night's final scores. When Pete browses to ESPN this morning, he finds that the page is taking a long time to load, and when it finally does, most of the images and content are missing (Figure 10-1). Let's help Pete diagnose this issue.

Figure 10-1: ESPN is failing to load properly.

Tapping into the Wire

This issue is isolated to Pete's workstation and is not affecting any others, so we'll start by capturing packets directly from there. To do this, we'll install Wireshark and capture packets while browsing to the ESPN website. Those packets are found in the file *http_espn_fail.pcapng*.

Analysis

We know Pete's issue is that he's unable to view a website he is browsing to, so we're primarily going to be looking at the HTTP protocol. If you read the previous chapter, you should have a basic understanding of what HTTP traffic between a client and server looks like. A good place to start looking is at the HTTP requests being made to the remote server. You can do this by applying a filter for `GET` requests (using **`http.request.method == "GET"`**), but this can also be done by simply selecting **Statistics ▸ HTTP ▸ Requests** from the main drop-down menu (Figure 10-2).

Figure 10-2: Viewing HTTP requests to ESPN

From this overview, it appears the capture is limited to seven different HTTP requests, and they all look like they are associated with the ESPN website. Each request contains the string `espn` within the domain name, with the exception of *cdn.optimizely.com*, which is a *content delivery network* (CDN) used to deliver advertising to a multitude of sites. It's common to see requests to various CDNs when browsing to websites that host advertisements or other external content.

With no clear leads to follow, the next step is to look at the protocol hierarchy of the capture file by selecting **Statistics ▸ Protocol Hierarchy**. This will allow us to spot unexpected protocols or peculiar distributions of traffic per protocol (Figure 10-3). Keep in mind that the protocol hierarchy screen is based on the currently applied display filter. Be sure to clear the previously applied filter to get the expected results based on the entire packet capture.

Wireshark · Protocol Hierarchy Statistics · http_espn_fail

Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets
Frame	100.0	569	100.0	357205	30 k	0
Ethernet	100.0	569	100.0	357205	30 k	0
Internet Protocol Version 4	100.0	569	100.0	357205	30 k	0
User Datagram Protocol	2.5	14	0.5	1627	136	0
Domain Name System	2.5	14	0.5	1627	136	14
Transmission Control Protocol	97.5	555	99.5	355578	29 k	541
Hypertext Transfer Protocol	2.5	14	2.4	8460	712	7
Portable Network Graphics	0.2	1	0.1	487	41	1
Line-based text data	0.5	3	0.5	1632	137	3
JPEG File Interchange Format	0.5	3	0.8	2962	249	3

No display filter.

Help Copy Close

Figure 10-3: Reviewing the protocol hierarchy of the browsing session

The protocol hierarchy isn't too complex, and we can quickly decipher that there are only two application-layer protocols at work: HTTP and DNS. As you learned in Chapter 9, DNS is used to translate domain names to IP addresses. So, when you browse to a site like *http://www.espn.com/*, your system may need to send out a DNS query to find the IP address of the remote web server if it doesn't already know it. Once a DNS reply with the appropriate IP address comes back, that information can be added to a local cache, and HTTP communication (using TCP) can commence.

Although nothing looks out of the ordinary here, the 14 DNS packets are notable. A DNS request for a single domain name is typically contained in a single packet, and the response also constitutes a single packet (unless it's very large, in which case DNS will utilize TCP). Since there are 14 DNS packets here, it's possible that as many as seven DNS queries were generated (7 queries + 7 replies = 14 packets). Figure 10-2 did show HTTP requests to seven different domains, but Pete only typed a single URL into his browser. Why are all of these extra requests being made?

In a simple world, visiting a web page would be as easy as querying one server and pulling all of its content in a single HTTP conversation. In reality, an individual web page may provide content hosted on multiple servers. All of the text-based content could be in one place, the graphics could be in another, and embedded videos could be in a third. That doesn't include ads, which could be hosted on multiple providers spanning dozens of individual servers. Whenever an HTTP client parses HTML code and finds a reference to content on another host, it will attempt to query that host for the content, which can generate additional DNS queries and HTTP requests. This is exactly what happened here when Pete visited ESPN. While he may have intended to view content only from a single source, references to additional content were found in the HTML code, and his browser automatically requested that content from multiple other domains.

Now that we understand why all of these extra requests exist, our next step is to examine the individual conversations associated with each request (**Statistics ▸ Conversations**). Reviewing the Conversations window (Figure 10-4) provides an important clue.

Address A	Address B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration
4.2.2.1	172.16.16.154	14	1627	7	1106	7	521	0.000000000	0.663869
68.71.212.158	172.16.16.154	13	2032	6	1200	7	832	0.027167000	90.875181
69.31.75.194	172.16.16.154	19	9949	10	8942	9	1007	0.579477000	90.659273
72.21.91.8	172.16.16.154	92	70 k	49	67 k	43	3170	0.526867000	60.553187
72.246.56.35	172.16.16.154	247	196 k	134	188 k	113	8315	0.527902000	90.806341
72.246.56.83	172.16.16.154	30	20 k	15	19 k	15	1518	0.659868000	45.344878
172.16.16.154	199.181.133.61	61	49 k	24	1953	37	47 k	0.238547000	91.083551
172.16.16.154	203.0.113.94	93	6774	93	6774	0	0	0.430071000	94.593597

Figure 10-4: Reviewing IP conversations

We discovered earlier that there were seven DNS requests and seven HTTP requests to match. With that in mind, it would be reasonable to expect that there would also be seven matching IP conversations, but that isn't the case. There are eight. How can that be explained?

One thought might be that the capture was "contaminated" by an additional conversation unrelated to the problem at hand. Ensuring your analysis doesn't suffer due to irrelevant traffic is certainly something you should be cognizant of, but that isn't the issue with this conversation. If you examine each HTTP request and note the IP address the request was sent to, you should be left with one conversation that doesn't have a matching HTTP request. The endpoints for this conversation are Pete's workstation (172.16.16.154) and the remote IP 203.0.113.94. This conversation is represented by the bottom line in Figure 10-4. We note that 6,774 bytes were sent to this unknown host but zero bytes were sent back: that's worth digging into.

If you filter down into this conversation (right-click the conversation and choose **Apply As Filter ▸ Selected ▸ A<->B**), you can apply your knowledge of TCP to identify what's gone wrong (Figure 10-5).

No.	Time	Source	Destination	Protocol	Length	Info
25	0.430071	172.16.16.154	203.0.113.94	TCP	78	64862 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1101093668 TSec…
26	0.430496	172.16.16.154	203.0.113.94	TCP	78	64863 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1101093668 TSec…
27	0.431050	172.16.16.154	203.0.113.94	TCP	78	64864 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1101093669 TSec…
39	0.500663	172.16.16.154	203.0.113.94	TCP	78	64865 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1101093737 TSec…
40	0.500873	172.16.16.154	203.0.113.94	TCP	78	64866 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1101093737 TSec…
70	0.553964	172.16.16.154	203.0.113.94	TCP	78	64869 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1101093787 TSec…
456	1.460006	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64863 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …
457	1.460006	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64862 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …
458	1.461238	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64864 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …
459	1.530278	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64866 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …
460	1.530278	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64865 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …
461	1.580145	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64869 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …
462	2.461157	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64863 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …
463	2.461157	172.16.16.154	203.0.113.94	TCP	78	[TCP Retransmission] 64862 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 …

Figure 10-5: Reviewing the unexpected connection

With normal TCP communication, you expect to see a standard SYN-SYN/ACK-ACK handshake sequence. In this case, Pete's workstation sent a SYN packet to 203.0.113.94, but we never see a SYN/ACK response. Not only this, but Pete's workstation sent multiple SYN packets to no avail, eventually leading his machine to send TCP retransmission packets. We'll

talk more about the specifics of TCP retransmissions in Chapter 11, but the key takeaway here is that one host is sending packets that it never receives a response to. Looking at the Time column, we see that the retransmissions continue for 95 seconds without a response. In network communications, this is slower than molasses.

We have identified seven DNS requests, seven HTTP requests, and eight IP conversations. Since we know that the capture is not contaminated with extra data, it's reasonable to think that the mysterious eighth IP conversation is probably the source of Pete's slowly and incompletely loading web page. For some reason, Pete's workstation is trying to communicate with a device that either doesn't exist or just isn't listening. To understand why this is happening, we won't look at what's in the capture file; instead, we'll consider what isn't there.

When Pete browsed to *http://www.espn.com/*, his browser identified resources hosted on other domains. To retrieve that data, his workstation generated DNS requests to find their IP addresses, then connected to them via TCP so that an HTTP request for the content could be sent. For the conversation with 203.0.113.94, there is no DNS request to be found. So, how did Pete's workstation know about that address?

If you remember our discussion about DNS in Chapter 9 or are otherwise familiar with it, you know that most systems implement some form of DNS caching. This allows them to reference a local DNS-to-IP address mapping that has already been retrieved without having to generate a DNS request every time you visit a domain that you frequently communicate with. Eventually, these DNS-to-IP mappings expire, and a new request must be generated. However, if a DNS-to-IP mapping changes and a device doesn't generate a DNS request to get the new address when visiting the next time, the device will attempt to connect to an address that is no longer valid.

In Pete's case, that is exactly what happened. Pete's workstation already had a cached DNS-to-IP mapping for a domain that hosts content for ESPN. Since this cached entry exists, a DNS request was not generated, and his system attempted to go ahead and connect to the old address. However, that address was no longer configured to respond to requests. As a result, the requests timed out, and the content never loaded.

Fortunately for Pete, clearing his DNS cache manually is possible with a few keystrokes on the command line or in a terminal window. Alternatively, he could also just try again in a few minutes when the DNS cache entry will probably have expired so a new request will be generated.

Lessons Learned

That's a lot of work just to find out that Kentucky beat Duke by 90 points, but we walk away with a deeper understanding of the relationship between network hosts. In this scenario, we were able to work toward a solution by assessing multiple data points related to the requests and conversations

occurring within the capture. From there, we were able to spot a few inconsistencies that took us down a path toward finding the failed communication between the client and one of ESPN's content delivery servers.

In the real world, diagnosing problems is rarely as simple as scrolling through a list of packets and looking for the ones that look funny. Troubleshooting even the simplest problems can result in very large captures that rely on the use of Wireshark's analysis and statistics features to spot anomalies. Getting familiar with this style of analysis is critical to successful troubleshooting at the packet level.

If you'd like to see an example of what normal communication looks like between a web browser and ESPN, try browsing to the site while capturing traffic yourself and see if you can identify all of the servers responsible for delivering content.

Unresponsive Weather Service

weather_broken .pcapng
weather_working .pcapng

Our second scenario once again involves our pal Packet Pete. Among his many hobbies, Pete fancies himself an amateur meteorologist and doesn't go more than a few hours without checking current conditions and the forecast. He doesn't rely solely on the local news forecast though; he actually runs a small weather station outside his home that reports data up to *https://www.wunderground.com/* for aggregation and viewing. Today, Pete went to check his weather station to see how much the temperature had dropped overnight, but found that his station hadn't reported in to Wunderground in over nine hours, since around midnight (Figure 10-6).

Figure 10-6: The weather station hasn't sent a report in nine hours.

Tapping into the Wire

In Pete's network, the weather station mounted on his roof connects to a receiver inside his house through an RF connection. That receiver plugs into his network switch and reports statistics to Wunderground through the internet. This architecture is diagrammed in Figure 10-7.

Figure 10-7: Weather station network architecture

The receiver has a simple web-based management page, but Pete logged into it only to find a cryptic message about the last synchronization time with no additional guidance for troubleshooting—the software doesn't provide any detailed error logging. Since the receiver is the hub of communication for the weather station infrastructure, it makes sense to capture packets transmitted to and from that device to try to diagnose the issue. This is a home network, so port mirroring is probably not an option on the SOHO switch. Our best bet is to use a cheap tap or to perform ARP cache poisoning to intercept these packets. The captured packets are contained in the file *weather_broken.pcapng*.

Analysis

Upon opening the capture file, you'll see that we're dealing with HTTP communication once again. The packet capture is limited to a single conversation between Pete's local weather receiver 172.16.16.154 and an unknown remote device on the internet, 38.102.136.125 (Figure 10-8).

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.154	38.102.136.125	TCP	78	53904 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 TSval=1015238041 TSecr=0 SACK_PERM=1
2	0.087018	38.102.136.125	172.16.16.154	TCP	60	80 → 53904 [SYN, ACK] Seq=0 Ack=1 Win=8190 Len=0 MSS=1360
3	0.087108	172.16.16.154	38.102.136.125	TCP	54	53904 → 80 [ACK] Seq=1 Ack=1 Win=65535 Len=0
4	0.087178	172.16.16.154	38.102.136.125	HTTP	571	GET /weatherstation/updateweatherstation.php?ID=KGAOAKWO2&PASSWORD=00000000&tempf=43.0&humidity=30…
5	0.176462	38.102.136.125	172.16.16.154	HTTP	237	HTTP/1.0 200 OK (text/html)
6	0.176567	172.16.16.154	38.102.136.125	TCP	54	53904 → 80 [ACK] Seq=518 Ack=184 Win=65535 Len=0
7	0.176714	172.16.16.154	38.102.136.125	TCP	54	53904 → 80 [FIN, ACK] Seq=518 Ack=184 Win=65535 Len=0
8	0.262587	38.102.136.125	172.16.16.154	TCP	60	80 → 53904 [FIN, ACK] Seq=184 Ack=519 Win=7673 Len=0
9	0.262656	172.16.16.154	38.102.136.125	TCP	54	53904 → 80 [ACK] Seq=519 Ack=185 Win=65535 Len=0

Figure 10-8: Isolated weather station receiver communication

Before we examine the characteristics of the conversation, let's see if we can identify the unknown IP. Without extensive research, we might not be able to find out whether this is the exact IP address that Pete's weather receiver should be talking to, but we can at least verify that it is part of the Wunderground infrastructure by doing a WHOIS query. You can conduct a WHOIS query through most domain registration or regional internet registry websites, such as *http://whois.arin.net/*. In this case, it looks like the IP belongs to Cogent, an *internet service provider (ISP)* (Figure 10-9). PSINet Inc. is also mentioned here, but a quick search reveals that most PSINet assets were acquired by Cogent in the early 2000s.

Network	
Net Range	38.0.0.0 - 38.255.255.255
CIDR	38.0.0.0/8
Name	COGENT-A
Handle	NET-38-0-0-0-1
Parent	
Net Type	Direct Allocation
Origin AS	AS174
Organization	PSINet, Inc. (PSI)
Registration Date	1991-04-16
Last Updated	2011-05-20
Comments	Reassignment information for this block can be found at rwhois.cogentco.com 4321
RESTful Link	https://whois.arin.net/rest/net/NET-38-0-0-0-1

Function	Point of Contact
Tech	PSI-NISC-ARIN (PSI-NISC-ARIN)

See Also	Related organization's POC records.
See Also	Related delegations.

Figure 10-9: WHOIS data identifies the owner of this IP.

In some cases, if an IP address is registered directly to an organization, the WHOIS query will return that organization's name. However, many times a company will simply utilize IP address space from an ISP without registering it directly to itself. In these cases, another useful tactic is to search for the *autonomous system number (ASN)* that is associated with an IP address. Organizations are required to register for an ASN to support certain types of routing on the public internet. There are a number of ways to look up IP-to-ASN associations (some WHOIS lookups provide it automatically), but I like using Team Cymru's automated lookup tool (*https://asn.cymru.com/*). Using that tool for 38.102.136.125, we see that it is associated with AS 36347, which is associated with "Wunderground – The Weather Channel, LLC, US" (Figure 10-10). That tells us that the device the weather station is communicating with is at least in the right neighborhood. If we were unable to

identify the correct affiliation for this address, it might be worth exploring whether Pete's receiver was talking to the wrong device, but the address checks out.

```
Executing commands. Please be patient!

v4.whois.cymru.com

The server returned 4 line(s).

[Querying v4.whois.cymru.com]
[v4.whois.cymru.com]
AS      | IP              | AS Name
36347   | 38.102.136.125  | WUNDERGROUND - THE WEATHER CHANNEL, LLC,US
```

Figure 10-10: IP-to-ASN lookup for the external IP address

With the unknown host characterized, we can dig into details of the communication. The conversation is relatively short. There is a TCP handshake, a single HTTP `GET` request and response, and a TCP teardown. The handshake and teardown appear to be successful, so whatever issue we are experiencing is probably contained with the HTTP request itself. To examine this closely, we'll follow the TCP stream (Figure 10-11).

```
Wireshark · Follow TCP Stream (tcp.stream eq 0) · weather_broken

GET /weatherstation/updateweatherstation.php?
ID=KGAOAKWO2&PASSWORD=00000000&tempf=43.0&humidity=30&dewptf=13.6&windchillf=43.0&winddir=194&windspeedmph
=0.22&windgustmph=2.46&rainin=0.00&dailyrainin=0.00&weeklyrainin=0.00&monthlyrainin=0.00&yearlyrainin=0.00
&solarradiation=54.14&UV=0&indoortempf=67.1&indoorhumidity=26&baromin=29.32&lowbatt=0&dateutc=2016-1-6%202
1:58:34&softwaretype=Weather%20logger%20V1.0&action=updateraw&realtime=1&rtfreq=5 HTTP/1.1 ❶
Host: 38.102.136.125
User-Agent: curl/7.43.0
Accept: */*

HTTP/1.0 200 OK ❷
Content-type: text/html
Date: Thu, 07 Jan 2016 00:28:39 GMT
Content-Length: 58
Connection: keep-alive

INVALIDPASSWORDID|Password or key and/or id are incorrect ❸

1 client pkt(s), 1 server pkt(s), 1 turn.
Entire conversation (700 bytes)    Show data as ASCII    Stream 0
Find:                                                    Find Next
Hide this stream   Print   Save as...   Close   Help
```

Figure 10-11: Following the TCP stream of the weather receiver communication

The HTTP communication begins with a `GET` request from Pete's weather receiver to Wunderground. No HTTP content was transmitted, but a significant amount of data was transmitted in the URL ❶. Transferring data through the URL query string is common for web applications, and it looks like the receiver is passing weather updates using this mechanism. For instance, you see fields like `tempf=43.0`, `dewptf=13.6`, and `windchillf=43.0`. The Wunderground collection server is parsing the list of fields and parameters from the URL and storing them in a database.

At first glance, everything looks fine with the `GET` request to the Wunderground server. But a look at the corresponding reply shows an error was reported. The server responded with an `HTTP/1.0 200 OK` response

code ❷, indicating that the `GET` request was received and successful, but the body of the response contains a useful message, `INVALIDPASSWORDID|Password or key and/or id are incorrect` ❸.

If you look back up at the request URL, you'll see the first two parameters passed are `ID` and `PASSWORD`. These are used to identify the weather station call sign and authenticate it to the Wunderground server.

In this case, Pete's weather station ID is correct, but his password is not. For some unknown reason, it has been replaced by zeros. Since the last known successful communication was at midnight, it's possible an update was applied or the receiver rebooted and lost the password configuration.

NOTE *While many developers choose to pass parameters in URLs, it's generally frowned upon to do this with passwords as seen here. That's because requested URLs are transmitted in plaintext when using HTTP without added encryption, such as HTTPS. Therefore, a malicious user who happens to be listening on the wire could intercept your password.*

At this point, Pete was able to access his receiver and type in the new password. Shortly thereafter, his weather station began syncing data again. An example of successful weather station communication can be found in *weather_working.pcapng*. The communication stream is shown in Figure 10-12.

Figure 10-12: Successful weather station communication

The password is now correct ❶, and the Wunderground server responds with a `success` message in the HTTP response body ❷.

Lessons Learned

In this scenario, we encountered a third-party service that facilitated network communication by using features available within another protocol (HTTP). Fixing communication problems with third-party services is something you'll encounter often, and packet analysis techniques are very well suited for troubleshooting these services when proper documentation

or error logging isn't available. This is becoming more common now that Internet of Things (IoT) devices, such as this weather station, are popping up all around us.

Fixing such problems requires the ability to inspect unknown traffic sequences and derive how things are supposed to be working. Some applications, such as the HTTP-based weather data transmission in this scenario, are fairly simple. Others are quite complex, requiring multiple transactions, the addition of encryption, or even custom protocols that Wireshark may not natively parse.

As you investigate more third-party services, you'll eventually start learning about common patterns developers use to facilitate network communication. This knowledge will increase your effectiveness when troubleshooting them.

No Internet Access

In many scenarios, you may need to diagnose and solve internet connectivity problems. We'll cover some common problems you might encounter.

Gateway Configuration Problems

nowebaccess1.pcapng

Our next scenario presents a common problem: a user cannot access the internet. We have verified that the user can access all the internal resources of the network, including shares on other workstations and applications hosted on local servers.

The network architecture is straightforward, as all clients and servers connect to a series of simple switches. Internet access is handled through a single router serving as the default gateway, and IP-addressing information is provided by DHCP. This is a very common scenario in small offices.

Tapping into the Wire

To determine the cause of the issue, we can have the user attempt to browse the internet while our sniffer is listening on the wire. We use the information from Chapter 2 (see Figure 2-15) to determine the most appropriate method for placing our sniffer.

The switches on our network don't support port mirroring. We already have to interrupt the user to conduct our test, so we can assume that it is okay to take them offline once again. Even though this isn't a high-throughput scenario, a TAP would be appropriate here if one were available. The resulting file is *nowebaccess1.pcapng*.

Analysis

The traffic capture begins with an ARP request and reply, as shown in Figure 10-13. In packet 1, the user's computer, with a MAC address of 00:25:b3:bf:91:ee and IP address 172.16.0.8, sends an ARP broadcast packet to all computers on the network segment in an attempt to find the MAC address associated with the IP address of its default gateway, 172.16.0.10.

No.	Time	Source	Destination	Protocol	Length	Info
1	04:32:21.445645	00:25:b3:bf:91:ee	ff:ff:ff:ff:ff:ff	ARP	42	Who has 172.16.0.10? Tell 172.16.0.8
2	04:32:21.445735	00:24:81:a1:f6:79	00:25:b3:bf:91:ee	ARP	60	172.16.0.10 is at 00:24:81:a1:f6:79

Figure 10-13: ARP request and reply for the computer's default gateway

A response is received in packet 2, and the user's computer learns that 172.16.0.10 is at 00:24:81:a1:f6:79. Once this reply is received, the computer has a route to a gateway that should be able to direct it to the internet.

Following the ARP reply, the computer must attempt to resolve the DNS name of the website to an IP address using DNS in packet 3. As shown in Figure 10-14, the computer does this by sending a DNS query packet to its primary DNS server, 4.2.2.2 ❶.

Figure 10-14: A DNS query sent to 4.2.2.2

Under normal circumstances, a DNS server would respond to a DNS query very quickly, but that's not the case here. Rather than a response, we see the same DNS query sent a second time to a different destination address. As shown in Figure 10-15, in packet 4, the second DNS query is sent to the secondary DNS server configured on the computer, which is 4.2.2.1 ❶.

Figure 10-15: A second DNS query sent to 4.2.2.1

Again, no reply is received from the DNS server, and the query is sent again 1 second later to 4.2.2.2. This process repeats itself, alternating between the primary ❶ and secondary ❷ configured DNS servers over the next several seconds, as shown in Figure 10-16. The entire process takes around 8 seconds ❸, or until the user's internet browser reports that a website is inaccessible.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	HewlettP_bf:91:ee	Broadcast	ARP	42	Who has 172.16.0.10? Tell 172.16.0.8
2	0.000090	HewlettP_a1:f6:79	HewlettP_bf:91:ee	ARP	60	172.16.0.10 is at 00:24:81:a1:f6:79
3	0.000105	172.16.0.8	4.2.2.2 ❶	DNS	74	Standard query 0x80d1 A www.google.com
4	0.999280	172.16.0.8	4.2.2.1	DNS	74	Standard query 0x80d1 A www.google.com
5	1.999279	172.16.0.8	4.2.2.2	DNS	74	Standard query 0x80d1 A www.google.com
6	3.999372	172.16.0.8	4.2.2.1 ❷	DNS	74	Standard query 0x80d1 A www.google.com
7	3.999393	172.16.0.8	4.2.2.2	DNS	74	Standard query 0x80d1 A www.google.com
8	7.999627	172.16.0.8	4.2.2.1	DNS	74	Standard query 0x80d1 A www.google.com
❸ 9	7.999648	172.16.0.8	4.2.2.2	DNS	74	Standard query 0x80d1 A www.google.com

Figure 10-16: DNS queries are repeated until communication stops.

Based on the packets we've seen, we can begin to pinpoint the source of the problem. First, we see a successful ARP request to what we believe is the default gateway router for the network, so we know that device is online and communicating. We also know that the user's computer is actually transmitting packets on the network, so we can assume there isn't an issue with the protocol stack on the computer itself. The problem clearly begins to occur when the DNS request is made.

In the case of this network, DNS queries are resolved by an external server on the internet (4.2.2.2 or 4.2.2.1). This means that for resolution to take place correctly, the router responsible for routing packets to the internet must successfully forward the DNS queries to the server, and the server must respond. This all must happen before HTTP can be used to request the web page itself.

Because no other users are having issues connecting to the internet, the network router and remote DNS server are probably not the source of the problem. The only thing remaining to investigate is the user's computer itself.

Upon deeper examination of the affected computer, we find that rather than receiving a DHCP-assigned address, the computer has manually assigned addressing information, and the default gateway address is set incorrectly. The address set as the default gateway is not a router and cannot forward the DNS query packets outside the network.

Lessons Learned

The problem in this scenario resulted from a misconfigured client. While the problem itself turned out to be simple, it significantly impacted the user. Troubleshooting a simple misconfiguration like this one could take quite some time for someone lacking knowledge of the network or the ability to perform a quick packet analysis, as we've done here. As you can see, packet analysis is not limited to large and complex problems.

Notice that because we didn't enter the scenario knowing the IP address of the network's gateway router, Wireshark didn't identify the

problem exactly, but it did tell us where to look, saving valuable time. Rather than examining the gateway router, contacting our ISP, or trying to find the resources to troubleshoot the remote DNS server, we were able to focus our troubleshooting efforts on the computer itself, which was, in fact, the source of the problem.

NOTE *Had we been more familiar with this particular network's IP-addressing scheme, analysis could have been even faster. The problem could have been identified immediately once we noticed that the ARP request was sent to an IP address different from that of the gateway router. These simple misconfigurations are often the source of network problems and can typically be resolved quickly with a bit of packet analysis.*

Unwanted Redirection

nowebaccess2.pcapng

In this scenario, we again have a user who is having trouble accessing the internet from their workstation. However, unlike the user in the previous scenario, this user can access the internet. Their problem is that they can't access their home page, *https://www.google.com/*. When the user attempts to reach any domain hosted by Google, they are directed to a browser page that says, "Internet Explorer cannot display the web page." This issue is affecting only this particular user.

As with the previous scenario, this is a small network with a few simple switches and a single router serving as the default gateway.

Tapping into the Wire

To begin our analysis, we have the user attempt to browse to *https://www.google.com/* while we use a tap to listen to the traffic that is generated. The resulting file is *nowebaccess2.pcapng*.

Analysis

The capture begins with an ARP request and reply, as shown in Figure 10-17. In packet 1, the user's computer, with a MAC address of 00:25:b3:bf:91:ee and an IP address of 172.16.0.8, sends an ARP broadcast packet to all computers on the network segment in an attempt to find the MAC address associated with the host's IP address 172.16.0.102. We don't immediately recognize this address.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	00:25:b3:bf:91:ee	ff:ff:ff:ff:ff:ff	ARP	42	Who has 172.16.0.102? Tell 172.16.0.8
2	0.000334	00:21:70:c0:56:f0	00:25:b3:bf:91:ee	ARP	60	172.16.0.102 is at 00:21:70:c0:56:f0

Figure 10-17: ARP request and reply for another device on the network

In packet 2, the user's computer learns that the IP address 172.16.0.102 is at 00:21:70:c0:56:f0. Based on the previous scenario, we might assume that this is the gateway router's address and that address is used so that packets can once again be forwarded to the external DNS server. However, as shown in Figure 10-18, the next packet is not a DNS request but a TCP

packet from 172.16.0.8 to 172.16.0.102. It has the SYN flag set ❸, indicating that this is the first packet in the handshake for a new TCP-based connection between the two hosts.

```
Wireshark · Packet 3 · nowebaccess2
> Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
> Ethernet II, Src: HewlettP_bf:91:ee (00:25:b3:bf:91:ee), Dst: Dell_c0:56:f0 (00:21:70:c0:56:f0)
> Internet Protocol Version 4, Src: 172.16.0.8, Dst: 172.16.0.102 ❶
v Transmission Control Protocol, Src Port: 1074 (1074), Dst Port: 80 (80), Seq: 0, Len: 0
    Source Port: 1074
    Destination Port: 80 ❷
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 32 bytes
  > Flags: 0x002 (SYN) ❸
    Window size value: 8192
    [Calculated window size: 8192]
  > Checksum: 0x58b5 [validation disabled]
    Urgent pointer: 0
  > Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (…
No.: 3 · Time: 0.000349 · Source: 172.16.0.8 · Destination: 172.16.0.102 · Pro…gth: 66 · Info: 1074 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PE
                                                                                    Close   Help
```

Figure 10-18: TCP SYN packet sent from one internal host to another

Notably, the TCP connection attempt is made to port 80 ❷ on 172.16.0.102 ❶, which is typically associated with HTTP traffic.

As shown in Figure 10-19, this connection attempt is abruptly halted when host 172.16.0.102 sends a TCP packet in response (packet 4) with the RST and ACK flags set ❶.

```
Wireshark · Packet 4 · nowebaccess2
> Frame 4: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: Dell_c0:56:f0 (00:21:70:c0:56:f0), Dst: HewlettP_bf:91:ee (00:25:b3:bf:91:ee)
> Internet Protocol Version 4, Src: 172.16.0.102, Dst: 172.16.0.8
v Transmission Control Protocol, Src Port: 80 (80), Dst Port: 1074 (1074), Seq: 1, Ack: 1, Len: 0
    Source Port: 80
    Destination Port: 1074
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1    (relative sequence number)
    Acknowledgment number: 1    (relative ack number)
    Header Length: 20 bytes
  > Flags: 0x014 (RST, ACK) ❶
    Window size value: 0
    [Calculated window size: 0]
    [Window size scaling factor: -1 (unknown)]
  > Checksum: 0xc9aa [validation disabled]
    Urgent pointer: 0
  > [SEQ/ACK analysis]
No.: 4 · Time: 0.000510 · Source: 172.16.0.102 · Destination: 172.16.0.8 · Protocol: TCP · Length: 60 · Info: 80 → 1074 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
                                                                                    Close   Help
```

Figure 10-19: TCP RST packet sent in response to the TCP SYN

Recall from Chapter 8 that a packet with the RST flag set is used to terminate a TCP connection. Here, the host at 172.16.0.8 attempted to establish a TCP connection to the host at 172.16.0.102 on port 80. Unfortunately, because that host has no services configured to listen to requests on port 80, the TCP RST packet is sent to terminate the connection. This process repeats three times before communication finally ends, as shown in Figure 10-20. At this point, the user receives a message in their browser saying that the page can't be displayed.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	HewlettP_bf:91:ee	Broadcast	ARP	42	Who has 172.16.0.102? Tell 172.16.0.8
2	0.000334	Dell_c0:56:f0	HewlettP_bf:91:ee	ARP	60	172.16.0.102 is at 00:21:70:c0:56:f0
3	0.000349	172.16.0.8	172.16.0.102	TCP	66	1074 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
4	0.000510	172.16.0.102	172.16.0.8	TCP	60	80 → 1074 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
5	0.499162	172.16.0.8	172.16.0.102	TCP	66	[TCP Spurious Retransmission] 1074 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
6	0.499362	172.16.0.102	172.16.0.8	TCP	60	80 → 1074 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
7	0.999190	172.16.0.8	172.16.0.102	TCP	62	[TCP Spurious Retransmission] 1074 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 SACK_PERM=1
8	0.999507	172.16.0.102	172.16.0.8	TCP	60	80 → 1074 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

Figure 10-20: The TCP SYN and RST packets are seen three times in total.

After examining the configuration of another network device that is working correctly, we are concerned by the ARP request and reply in packets 1 and 2, because the ARP request isn't for the gateway router's actual MAC address but for some unknown device. Following the ARP request and reply, we would expect to see a DNS query sent to our configured DNS server in order to find the IP address associated with *https://www.google.com/*, but we don't. There are two conditions that could prevent a DNS query from being made:

- The device initiating the connection already has the DNS name-to-IP address mapping in its DNS cache (as in the first scenario in this chapter).
- The device connecting to the DNS name already has the DNS name-to-IP address mapping specified in its *hosts* file.

Upon further examination of the client computer, we find that the computer's *hosts* file has an entry for *https://www.google.com/* associated with the internal IP address 172.16.0.102. This erroneous entry is the source of our user's problems.

A computer will typically use its *hosts* file as the authoritative source for DNS name-to-IP address mappings, and it will check that file before querying an outside source. In this scenario, the user's computer checked its *hosts* file, found the entry for *https://www.google.com/*, and decided that *https://www.google.com/* was actually on its own local network segment. Next, it sent an ARP request to the host, received a response, and attempted to initiate a TCP connection to 172.16.0.102 on port 80. However, because the remote system was not configured as a web server, it wouldn't accept the connection attempts.

Once the *hosts* file entry was removed, the user's computer began communicating correctly and was able to access *https://www.google.com/*.

NOTE *To examine your* hosts *file on a Windows system, open* C:\Windows\System32\drivers\etc\hosts. *On Linux, view* /etc/hosts.

This very common scenario is one that malware has been using for years to redirect users to websites hosting malicious code. Imagine if an attacker were to modify your *hosts* file so that every time you went to do your online banking, you were redirected to a fake site designed to steal your account credentials!

Lessons Learned

As you continue to analyze traffic, you will learn both how the various protocols work and how to break them. In this scenario, the DNS query wasn't sent because the client was misconfigured, not because of any external limitations or misconfigurations.

By examining this problem at the packet level, we were able to quickly spot an IP address that was unknown and to determine that the DNS, a key component of this communication process, was missing. Using this information, we were able to identify the client as the source of the problem.

Upstream Problems

nowebaccess3.pcapng

As with the previous two scenarios, in this scenario, a user complains of no internet access from their workstation. This user has narrowed the issue down to a single website, *https://www.google.com/*. Upon further investigation, we find that this issue is affecting everyone in the organization—no one can access Google domains.

The network is configured as in the two prior scenarios, with a few simple switches and a single router connecting the network to the internet.

Tapping into the Wire

To troubleshoot this issue, we first browse to *https://www.google.com/* to generate traffic. Because this issue is network-wide, ideally any device in the network should be able to reproduce the issue using most capture methods. The file resulting from the capture via a tap is *nowebaccess3.pcapng*.

Analysis

This packet capture begins with DNS traffic instead of the ARP traffic we are used to seeing. Because the first packet in the capture is to an external address, and packet 2 contains a reply from that address, we can assume that the ARP process has already happened and the MAC-to-IP address mapping for our gateway router already exists in the host's ARP cache at 172.16.0.8.

As shown in Figure 10-21, the first packet in the capture is from the host 172.16.0.8 to address 4.2.2.1 ❶ and is a DNS packet ❷. Examining the contents of the packet, we see that it is a query for the A record for *www.google.com* ❸ that will map the DNS name to an IP address.

```
Wireshark · Packet 1 · nowebaccess3
> Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
> Ethernet II, Src: HewlettP_bf:91:ee (00:25:b3:bf:91:ee), Dst: CiscoInc_31:07:33 (00:26:0b:31:07:33)
> Internet Protocol Version 4, Src: 172.16.0.8, Dst: 4.2.2.1❶
> User Datagram Protocol, Src Port: 64417 (64417), Dst Port: 53 (53)
v Domain Name System (query)❷
    [Response In: 2]
    Transaction ID: 0x6138
  > Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  v Queries❸
    v www.google.com: type A, class IN
        Name: www.google.com
        [Name Length: 14]
        [Label Count: 3]
        Type: A (Host Address) (1)
        Class: IN (0x0001)
No.: 1 · Time: 0.000000 · Source: 172.16.0.8 · Destination: 4.2.2.1 · Protocol: DNS · Length: 74 · Info: Standard query 0x6138 A www.google.com
                                                                                    Close    Help
```

Figure 10-21: DNS query for www.google.com *A record*

The response to the query from 4.2.2.1 is the second packet in the capture file, as shown in Figure 10-22. Here, we see that the name server that responded to this request provided multiple answers to the query ❶. At this point, all looks good, and communication is occurring as it should.

```
Wireshark · Packet 2 · nowebaccess3
> Frame 2: 190 bytes on wire (1520 bits), 190 bytes captured (1520 bits)
> Ethernet II, Src: CiscoInc_31:07:33 (00:26:0b:31:07:33), Dst: HewlettP_bf:91:ee (00:25:b3:bf:91:ee)
> Internet Protocol Version 4, Src: 4.2.2.1, Dst: 172.16.0.8
> User Datagram Protocol, Src Port: 53 (53), Dst Port: 64417 (64417)
v Domain Name System (response)
    [Request In: 1]
    [Time: 0.010440000 seconds]
    Transaction ID: 0x6138
  > Flags: 0x8180 Standard query response, No error
    Questions: 1
    Answer RRs: 7
    Authority RRs: 0
    Additional RRs: 0
  > Queries
  v Answers❶
    > www.google.com: type CNAME, class IN, cname www.l.google.com
    > www.l.google.com: type A, class IN, addr 74.125.95.105
    > www.l.google.com: type A, class IN, addr 74.125.95.106
    > www.l.google.com: type A, class IN, addr 74.125.95.147
    > www.l.google.com: type A, class IN, addr 74.125.95.99
    > www.l.google.com: type A, class IN, addr 74.125.95.103
    > www.l.google.com: type A, class IN, addr 74.125.95.104
No.: 2 · Time: 0.010440 · Source: 4.2.2.1 · Destination: 172.16.0.8 · Protocol: DNS ...5.105 A 74.125.95.106 A 74.125.95.147 A 74.125.95.99 A 74.125.95.103 A 74.125.95.1
                                                                                    Close    Help
```

Figure 10-22: DNS reply with multiple A records

Now that the user's computer has determined the web server's IP address, it can attempt to communicate with the server. As shown in Figure 10-23, this process is initiated in packet 3, with a TCP packet sent from 172.16.0.8 to 74.125.95.105 ❶. This destination address comes from the first A record provided in the DNS query response seen in packet 2. The TCP packet has the SYN flag set ❸, and it's attempting to communicate with the remote server on port 80 ❷.

Figure 10-23: The SYN packet is attempting to initiate a connection on port 80.

Because this is a TCP handshake process, we know that we should see a TCP SYN/ACK packet sent in response, but instead, after a short time, another SYN packet is sent from the source to the destination. This process occurs once more after approximately one second, as shown in Figure 10-24, at which point communication stops and the browser reports that the website could not be found.

No.	Time	Source	Destination	Protocol	Length	Info
3	0.014421	172.16.0.8	74.125.95.105	TCP	66	1251 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
4	0.019417	172.16.0.8	74.125.95.105	TCP	66	[TCP Retransmission] 1251 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
5	1.016531	172.16.0.8	74.125.95.105	TCP	66	[TCP Retransmission] 1251 → 80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1

Figure 10-24: The TCP SYN packet is attempted three times with no response received.

As we troubleshoot this scenario, consider that we know that the workstation within our network can connect to the outside world because the DNS query to our external DNS server at 4.2.2.1 is successful. The DNS server responds with what appears to be a valid address, and our hosts attempt to connect to one of those addresses. Also, the local workstation we are attempting to connect from appears to be functioning.

The problem is that the remote server simply isn't responding to our connection requests; a TCP RST packet is not sent. This might occur for several reasons: a misconfigured web server, a corrupted protocol stack on the web server, or a packet-filtering device on the remote network (such

as a firewall). Assuming there is no local packet-filtering device in place, all other potential solutions are on the remote network and beyond our control. In this case, the web server was not functioning correctly, and no attempt to access it succeeded. Once the problem was corrected on Google's end, communication was able to proceed.

Lessons Learned

In this scenario, the problem wasn't one that we could correct. Our analysis determined that the issue wasn't with the hosts on our network, our router, or the external DNS server providing us with name resolution services. The issue lay outside our network infrastructure.

Sometimes discovering that a problem isn't really ours not only relieves stress but also saves face when management comes knocking. I have fought with many ISPs, vendors, and software companies who claim that an issue is not their fault, but as you've just seen, packets don't lie.

Inconsistent Printer

In the next scenario, an IT help desk administrator is having trouble resolving a printing issue. Users in the sales department are reporting that the high-volume printer is malfunctioning. When a user sends a large print job to the printer, it will print several pages and then stop printing before the job is done. Multiple driver configuration changes have been attempted but have been unsuccessful. The help desk staff would like you to ensure that this isn't a network problem.

Tapping into the Wire

inconsistent_printer .pcapng

The common thread in this problem is the printer, so we begin by placing our sniffer as close to the printer as we can. While we can't install Wireshark on the printer itself, the switches used in this network are advanced layer 3 switches, so we can use port mirroring. We'll mirror the port used by the printer to an empty port and connect a laptop with Wireshark installed to this port. Once this setup is complete, we'll have a user send a large print job to the printer so we can monitor the output. The resulting capture file is *inconsistent_printer.pcapng*.

Analysis

A TCP handshake between the network workstation sending the print job (172.16.0.8) and the printer (172.16.0.253) initiates the connection at the start of the capture file. Following the handshake, a 1,460-byte TCP data packet ❶ is sent to the printer in packet 4 (Figure 10-25). The amount of data can be seen in the far right side of the Info column in the Packet List pane or at the bottom of the TCP header information in the Packet Details pane.

Figure 10-25: Data being transmitted to the printer over TCP

Following packet 4, another data packet is sent containing 1,460 bytes of data ❶, as you can see in Figure 10-26. This data is acknowledged by the printer in packet 6 ❷.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.0.8	172.16.0.253	TCP	66	3527 → 9100 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.000166	172.16.0.253	172.16.0.8	TCP	66	9100 → 3527 [SYN, ACK] Seq=0 Ack=1 Win=8760 Len=0 MSS=1460 WS=1 SACK_PERM=1
3	0.000201	172.16.0.8	172.16.0.253	TCP	54	3527 → 9100 [ACK] Seq=1 Ack=1 Win=65700 Len=0
4	0.001637	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=1 Ack=1 Win=65700 Len=1460
5	0.001646	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=1461 Ack=1 Win=65700 Len=1460 ❶
❷ 6	0.005493	172.16.0.253	172.16.0.8	TCP	160	9100 → 3527 [PSH, ACK] Seq=1 Ack=2921 Win=7888 Len=106
7	0.005561	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=2921 Ack=107 Win=65592 Len=1460
8	0.005571	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=4381 Ack=107 Win=65592 Len=1460
9	0.005578	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=5841 Ack=107 Win=65592 Len=1460
10	0.005585	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=7301 Ack=107 Win=65592 Len=1460
11	0.033569	172.16.0.253	172.16.0.8	TCP	60	9100 → 3527 [ACK] Seq=107 Ack=8761 Win=6144 Len=0
12	0.033626	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=8761 Ack=107 Win=65592 Len=1460
13	0.033640	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=10221 Ack=107 Win=65592 Len=1460
14	0.033649	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=11681 Ack=107 Win=65592 Len=1460
15	0.033658	172.16.0.8	172.16.0.253	TCP	1514	3527 → 9100 [ACK] Seq=13141 Ack=107 Win=65592 Len=1460
16	0.098314	172.16.0.253	172.16.0.8	TCP	60	9100 → 3527 [ACK] Seq=107 Ack=14601 Win=4400 Len=0

Figure 10-26: Normal data transmission and TCP acknowledgments

The flow of data continues until the last few packets in the capture are reached. Packet 121 is a TCP retransmission packet, and a sign of trouble, as shown in Figure 10-27.

A TCP retransmission packet is sent when one device sends a TCP packet to a remote device and the remote device doesn't acknowledge the transmission. Once a retransmission threshold is reached, the sending device assumes that the remote device did not receive the data, and it retransmits the packet. This process is repeated a few times before communication effectively stops.

Figure 10-27: These TCP retransmission packets are a sign of a potential problem.

In this scenario, the retransmission is sent from the client workstation to the printer because the printer failed to acknowledge the transmitted data. As shown in Figure 10-27, if you expand the SEQ/ACK analysis portion of the TCP header ❶ along with the additional information beneath it, you can view the details of why this is a retransmission. According to the details processed by Wireshark, packet 121 is a retransmission of packet 120 ❸. Additionally, the retransmission timeout (RTO) for the retransmitted packet was around 5.5 seconds ❷.

When analyzing the delay between packets, you can change the time display format to suit your situation. In this case, because we want to see how long the retransmissions occurred after the previous packet was sent, change this option by selecting **View ▸ Time Display Format** and select **Seconds Since Previous Captured Packet**. Then, as shown in Figure 10-28, you can clearly see that the retransmission in packet 121 occurs 5.5 seconds after the original packet (packet 120) is sent ❶.

No.	Time ❶	Source	Destination	Protocol	Length	Info
121	5.502585	172.16.0.8	172.16.0.253	TCP	1146	[TCP Window Full] [TCP Retransmission] 3527 → 9100 [ACK] Seq=118261 Ack=107 Win=…
122	5.600089	172.16.0.8	172.16.0.253	TCP	1146	[TCP Window Full] [TCP Retransmission] 3527 → 9100 [ACK] Seq=118261 Ack=107 Win=…

Figure 10-28: Viewing the time between packets is useful for troubleshooting.

The next packet is another retransmission of packet 120. The RTO of this packet is 11.10 seconds, which includes the 5.5 seconds from the RTO of the previous packet. A look at the Time column of the Packet List pane

tells us that this retransmission was sent 5.6 seconds after the previous retransmission. This appears to be the last packet in the capture file, and, not coincidentally, the printer stops printing at approximately this time.

In this scenario, we have the benefit of dealing with only two devices inside our own network, so we just need to determine whether the client workstation or the printer is to blame. We can see that data is flowing correctly for quite some time, and then at some point, the printer simply stops responding to the workstation. The workstation gives its best effort to get the data to its destination, as evidenced by the retransmissions, but the effort is met with no response. This issue is reproducible and happens regardless of which computer sends a print job, so we assume the printer is the source of the problem.

After further analysis, we find that the printer's RAM is malfunctioning. When large print jobs are sent to the printer, it prints only a certain number of pages, likely until certain regions of memory are accessed. At that point, the memory issue causes the printer to be unable to accept any new data, and it ceases communication with the host transmitting the print job.

Lessons Learned

Although this printer problem wasn't the result of a network issue, we were able to use Wireshark to pinpoint the problem. Unlike previous scenarios, this one centered solely on TCP traffic. Since TCP is concerned about reliably transmitting data, it often leaves us with useful information when two devices simply stop communicating.

In this case, when communication abruptly stopped, we were able to pinpoint the location of the problem based on nothing more than TCP's built-in retransmission functionality. As we continue through our scenarios, we will often rely on functionality like this to troubleshoot more complex issues.

No Branch Office Connectivity

stranded_clientside.pcapng
stranded_branchdns.pcapng

In this scenario, we have a company with a central headquarters office and a newly deployed remote branch office. The company's IT infrastructure is mostly contained within the central office using a Windows server-based domain. This infrastructure is supported by a domain controller, a DNS server, and an application server used to host web-based software used daily by the organization's employees. The branch office is connected by routers to establish a wide area network (WAN) link. Inside the branch office are user workstations and a slave DNS server that should receive its resource record information from the upstream DNS server at the corporate headquarters. Figure 10-29 shows a map of each office and how the offices are linked together.

Figure 10-29: The relevant components for the stranded branch office issue

The deployment team is rolling out new infrastructure to the branch office when it finds that no one can access the intranet web application server from the branch office network. This server is located at the main office and is accessed through the WAN link. This connectivity issue affects all users at the branch office. All users can access the internet and other resources within the branch.

Tapping into the Wire

Because the problem lies in communication between the main and branch offices, there are a couple of places we could collect data to start tracking down the problem. The problem could be with the clients inside the branch office, so we'll start by port mirroring one of those computers to check what it sees on the wire. Once we've collected that information, we can use it to point toward other collection locations that might help solve the problem. The initial capture file obtained from one of the clients is *stranded_clientside.pcapng*.

Analysis

As shown in Figure 10-30, our first capture file begins when the user at the workstation address 172.16.16.101 attempts to access an application hosted on the headquarter's app server, 172.16.16.200. This capture contains only two packets. It appears as though a DNS request is sent to 172.16.16.251 ❶ for the A record ❸ for `appserver` ❷ in the first packet. This is the DNS name for the server at 172.16.16.200 in the central office.

As you can see in Figure 10-31, the response to this packet is a server failure ❶, which indicates that something is preventing the DNS query from resolving successfully. Notice that this packet does not answer the query ❷ since it is an error (server failure).

```
Wireshark · Packet 1 · stranded_clientside
> Frame 1: 69 bytes on wire (552 bits), 69 bytes captured (552 bits)
> Ethernet II, Src: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a), Dst: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a)
> Internet Protocol Version 4, Src: 172.16.16.101, Dst: 172.16.16.251 ❶
> User Datagram Protocol, Src Port: 56779 (56779), Dst Port: 53 (53)
v Domain Name System (query)
    [Response In: 2]
    Transaction ID: 0x0003
  > Flags: 0x0100 Standard query
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v appservr: type A, class IN
        Name: appservr ❷
        [Name Length: 9]
        [Label Count: 1]
        Type: A (Host Address) (1) ❸
        Class: IN (0x0001)
No.: 1 · Time: 0.000000 · Source: 172.16.16.101 · Destination: 172.16.16.251 · Protocol: DNS · Length: 69 · Info: Standard query 0x0003 A appservr
                                                                                    Close   Help
```

Figure 10-30: Communication begins with a DNS query for the appservr A record.

```
Wireshark · Packet 2 · stranded_clientside
> Frame 2: 69 bytes on wire (552 bits), 69 bytes captured (552 bits)
> Ethernet II, Src: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a), Dst: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a)
> Internet Protocol Version 4, Src: 172.16.16.251, Dst: 172.16.16.101
> User Datagram Protocol, Src Port: 53 (53), Dst Port: 56779 (56779)
v Domain Name System (response)
    [Request In: 1]
    [Time: 0.000346000 seconds]
    Transaction ID: 0x0003
  > Flags: 0x8182 Standard query response, Server failure ❶
    Questions: 1
    Answer RRs: 0 ❷
    Authority RRs: 0
    Additional RRs: 0
  v Queries
    v appservr: type A, class IN
        Name: appservr
        [Name Length: 9]
        [Label Count: 1]
        Type: A (Host Address) (1)
        Class: IN (0x0001)
No.: 2 · Time: 0.000346 · Source: 172.16.16.251 · Destination: 172.16.16.101 · Protocol: DNS · Length: 69 · Info: Standard query response 0x0003 Server failure A appservr
                                                                                    Close   Help
```

Figure 10-31: The query response indicates a problem upstream.

We now know that the communication problem is related to some DNS issue. Because the DNS queries at the branch office are resolved by the on-site DNS server at 172.16.16.251, that's our next stop.

To capture the appropriate traffic from the branch DNS server, we'll leave our sniffer in place and simply change the port-mirroring assignment so that the DNS server's traffic, rather than the workstation's traffic, is now mirrored to our sniffer. The result is the file *stranded_branchdns.pcapng*.

As shown in Figure 10-32, this capture begins with the query and response we saw earlier, along with one additional packet. This additional

packet looks a bit odd because it is attempting to communicate with the primary DNS server at the central office (172.16.16.250) ❶ on the standard DNS server port 53 ❸, but it is not the UDP ❷ we're used to seeing.

Figure 10-32: This SYN packet uses port 53 but is not UDP.

To figure out the purpose of this packet, recall our discussion of DNS in Chapter 9. DNS usually uses UDP, but it uses TCP when the response to a query exceeds a certain size. In that case, we'll see some initial UDP traffic that triggers the TCP traffic. TCP is also used for DNS during a zone transfer, when resource records are transferred between DNS servers, which is likely the case here.

The DNS server at the branch office location is a slave to the DNS server at the central office, meaning that it relies on it in order to receive resource records. The application server that users in the branch office are trying to access is located inside the central office, which means that the central office DNS server is authoritative for that server. For the branch office server to resolve a DNS request for the application server, the DNS resource record for that server must be transferred from the central office DNS server to the branch office DNS server. This is likely the source of the SYN packet in this capture file.

The lack of response to this SYN packet tells us that the DNS problem is the result of a failed zone transfer between the branch and central office DNS servers. Now we can go one step further by figuring out why the zone transfer is failing. The possible culprits for the issue can be narrowed down to the routers between the offices or the central office DNS server itself. To determine which is at fault, we can sniff the traffic of the central office DNS server to see whether the SYN packet is making it to the server.

I haven't included a capture file for the central office DNS server traffic because there was none. The SYN packet never reached the server. Upon dispatching technicians to review the configuration of the routers connecting the two offices, it was found that inbound port 53 traffic on the central office router was configured to allow only UDP traffic and to block inbound

TCP traffic. This simple misconfiguration prevented zone transfers from occurring between servers, thus preventing clients within the branch office from resolving queries for devices in the central office.

Lessons Learned

You can learn a lot about investigating network communication issues by watching crime dramas. When a crime occurs, the detectives begin by interviewing those most affected. Leads that result from that examination are pursued, and the process continues until a culprit is found.

In this scenario, we began by examining the target (the workstation) and established leads by finding the DNS communication issue. Our leads led us to the branch DNS server, then to the central DNS server, and finally to the router, which was the source of the problem.

When performing analysis, try thinking of packets as clues. The clues don't always tell you who committed the crime, but they often take you to the culprit eventually.

Software Data Corruption

tickedoffdeveloper.pcapng

Some of the most frequent arguments in IT are between developers and network administrators. Developers always blame poor network engineering and malfunctioning equipment for program errors. In turn, network administrators tend to blame bad code for network errors and slow communication.

In this scenario, a programmer has developed an application for tracking the sales at multiple stores and reporting back to a central database. To save bandwidth during normal business hours, the application does not update in real time. Data is accumulated throughout the day and then transmitted at night as a comma-separated value (CSV) file to be inserted into the central database.

This newly developed application isn't functioning correctly. The files sent from the stores are being received by the server, but the data being inserted into the database is not correct. Sections are missing, data is in the wrong place, and some portions of the data are missing. Much to the dismay of the network administrator, the programmer blames the network for the issue. They are convinced that the files are becoming corrupted while in transit from the stores to the central data repository. Our goal is to determine whether they are right.

Tapping into the Wire

To collect the data we need, we can capture packets at one of the stores or at the central office. Because the issue is affecting all the stores, it should occur at the central office if it is network related—that is the only common thread among all stores (other than the software itself).

The network switches support port mirroring, so we'll mirror the port the server is plugged into and sniff its traffic. The traffic capture will be isolated to a single instance of a store uploading its CSV file to the collection server. This result is the capture file *tickedoffdeveloper.pcapng*.

Analysis

We know nothing about the application the programmer has developed, other than the basic flow of information on the network. The capture file appears to start with some FTP traffic, so we'll investigate to see whether it is indeed the mechanism that is transporting this file.

Looking at the packet list first (Figure 10-33), we can see that 172.16.16.128 ❶ initiates communication to 172.16.16.121 ❷ with a TCP handshake. Since 172.16.16.128 initiates the communication, we can assume that it is the client and that 172.16.16.121 is the server that compiles and processes the data. Following the handshake completion, we begin seeing FTP requests from the client and responses from the server ❸.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.128 ❶	172.16.16.121 ❷	TCP	66	2555 → 21 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.000071	172.16.16.121	172.16.16.128	TCP	66	21 → 2555 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=1460 WS=256 SACK_PERM=1
3	0.000242	172.16.16.128	172.16.16.121	TCP	60	2555 → 21 [ACK] Seq=1 Ack=1 Win=17520 Len=0
4	0.002749	172.16.16.121	172.16.16.128	FTP	96	Response: 220 FileZilla Server version 0.9.34 beta
5	0.002948	172.16.16.128	172.16.16.121	FTP	70	Request: USER salesxfer
6	0.003396	172.16.16.121	172.16.16.128	FTP	91	Response: 331 Password required for salesxfer
7	0.003514	172.16.16.128	172.16.16.121	FTP	69	Request: PASS p@ssw0rd
8	0.004862	172.16.16.121	172.16.16.128	FTP	69	Response: 230 Logged on

Figure 10-33: The initial communication helps identify the client and server.

We know that some transfer of data should be happening here, so we can use our knowledge of FTP to locate the packet where this transfer begins. The FTP connection and data transfer are initiated by the client, so from 172.16.16.128 we should see the FTP `STOR` command, which is used to upload data to an FTP server. The easiest way to find this command is to build a filter.

Because this capture file is littered with FTP request commands, rather than sorting through the hundreds of protocols and options in the expression builder, we can build the filter we need directly from the Packet List pane. To do so, we first need to select a packet with an FTP request command present. We'll choose packet 5, since it's near the top of our list. Then expand the FTP section in the Packet Details pane and expand the USER section. Right-click the **Request Command: USER** field and select **Prepare a Filter**. Finally, choose **Selected**.

This will prepare a filter for all packets that contain the `FTP USER` request command and put it in the filter dialog. Next, as shown in Figure 10-34, edit the filter by replacing the word `USER` with the word **`STOR`** ❶.

ftp.request.command == "STOR" ❶ Expression... +

No.	Time	Source	Destination	Protocol	Length	Info
❷ 64	4.369659	172.16.16.128	172.16.16.121	FTP	83	Request: STOR store4829-03222010.csv

Figure 10-34: This filter helps identify where data transfer begins.

We could narrow down the filter further by providing the client's IP address and specifying it as the source of the connection by adding `&& ip.src == 172.16.16.128` to the filter, but since this capture file is already isolated to a single client, that isn't necessary here.

Now apply this filter by pressing ENTER, and you'll see that only one instance of the STOR command exists in the capture file, at packet 64 ❷.

Now that we know where data transfer begins, click the packet to select it and clear the filter by clicking the **X** button above the Packet List pane. Your screen should now show all the packets with packet 64 selected.

Examining the capture file beginning with packet 64, we see that this packet specifies the transfer of the file *store4829-03222010.csv* ❶, as shown in Figure 10-35.

Figure 10-35: The CSV file is being transferred using FTP.

The packets following the STOR command use a different port but are identified as part of an FTP-DATA transmission. We've verified that data is being transferred, but we have yet to establish whether the programmer is right or wrong. To do so, we need to show whether the contents of the file are intact after traversing the network, so we'll proceed to extract the transferred file from the captured packets.

When a file is transferred across a network in an unencrypted format, it is broken down into segments and reassembled at its destination. In this scenario, we captured packets as they reached their destination but before they were reassembled. The data is all there; we simply need to reassemble it by extracting the file as a data stream. To perform the reassembly, select any of the packets in the FTP-DATA stream (such as packet 66) and click **Follow TCP Stream**. The results are displayed as shown in Figure 10-36. This looks like a normal CSV-formatted text file containing sales order data.

The data appears because it is being transferred in plaintext over FTP, but we can't be sure that the file is intact based on the stream alone. To reassemble the data so as to extract it in its original format, click the **Save As** button and specify the name of the file as displayed in packet 64. Then click **Save**.

The result of this save operation should be a CSV file that is an exact byte-level copy of the file originally transferred from the store system. The file can be verified by comparing the MD5 hash of the original file with that of the extracted file. The MD5 hashes should be the same, as shown in Figure 10-37.

Figure 10-36: The TCP stream shows what appears to be the data being transferred.

Figure 10-37: The MD5 hashes of the original file and the extracted file are equivalent.

Once the files are compared, we can state that the network is not to blame for the database corruption occurring within the application. The file transferred from the store to the collection server is intact when it reaches the server, so any corruption must be occurring when the file is processed by the application on the server side.

Lessons Learned

One great thing about packet-level analysis is that you don't need to deal with the clutter of applications. Poorly coded applications greatly outnumber the good ones, but at the packet level, none of that matters. In this case, the programmer was concerned about all of the mysterious components their application was dependent upon, but at the end of the day, their complicated data transfer that took hundreds of lines of code is still no more than FTP, TCP, and IP. Using what we know about these basic protocols, we were able to ensure the communication process was flowing correctly and even extract files to prove the soundness of the network. It's crucial to remember that no matter how complex the issue at hand, it still comes down to packets.

Final Thoughts

In this chapter, we've covered several scenarios in which packet analysis allowed us to gain a better understanding of problematic communication. Using basic analysis of common protocols, we were able to track down and solve network problems in a timely manner. While you may not encounter exactly the same scenarios on your network, the analysis techniques presented here should prove useful as you analyze your own unique problems.

11

FIGHTING A SLOW NETWORK

As a network administrator, much of your time will be spent fixing computers and services that are running slower than they should be. But just because someone says that the network is running slowly doesn't mean that the network is to blame.

Before you begin to tackle a slow network, you first need to determine whether the network is in fact running slowly. You'll learn how to do that in this chapter.

We'll begin by discussing the error-recovery and flow-control features of TCP. Then we'll explore how to detect the source of slowness on a network. Finally, we'll look at ways of baselining networks and the devices and services that run on them. Once you have completed this chapter, you should be much better equipped to identify, diagnose, and troubleshoot slow networks.

NOTE *Multiple techniques can be used to troubleshoot slow networks. I've chosen to focus primarily on TCP because most of the time, it is all you'll have to work with. TCP allows you to perform passive retrospective analysis rather than generate additional traffic (unlike ICMP).*

TCP Error-Recovery Features

TCP's error-recovery features are our best tools for locating, diagnosing, and eventually repairing high latency on a network. In terms of computer networking, *latency* is a measure of delay between a packet's transmission and its receipt.

Latency can be measured as one-way (from a single source to a destination) or as round-trip (from a source to a destination and back to the original source). When communication between devices is fast, and the amount of time it takes a packet to get from one point to another is low, the communication is said to have *low latency*. Conversely, when packets take a significant amount of time to travel between a source and destination, the communication is said to have *high latency*. High latency is the number one enemy of all network administrators who value their sanity (and their jobs).

In Chapter 8, we discussed how TCP uses sequence and acknowledgment numbers to ensure the reliable delivery of packets. In this chapter, we'll look at sequence and acknowledgment numbers again to see how TCP responds when high latency causes these numbers to be received out of sequence (or not received at all).

TCP Retransmissions

tcp_retransmissions .pcapng

The ability of a host to retransmit packets is one of TCP's most fundamental error-recovery features. It is designed to combat packet loss.

There are many possible causes of packet loss, including malfunctioning applications, routers under a heavy traffic load, or temporary service outages. Things move fast at the packet level, and often the packet loss is temporary, so it's crucial for TCP to be able to detect and recover from packet loss.

The primary mechanism for determining whether the retransmission of a packet is necessary is the *retransmission timer*. This timer is responsible for maintaining a value called the *retransmission timeout (RTO)*. Whenever a packet is transmitted using TCP, the retransmission timer starts. This timer stops when an ACK for that packet is received. The time between the packet transmission and receipt of the ACK packet is called the *round-trip time (RTT)*. Several of these times are averaged, and that average is used to determine the final RTO value.

Until an RTO value is determined, the transmitting operating system relies on its default configured RTT setting, which is issued for the initial communication between hosts. This is then adjusted based on the RTT of received packets to determine the RTO value.

Once the RTO value has been determined, the retransmission timer is used on every transmitted packet to determine whether packet loss has occurred. Figure 11-1 illustrates the TCP retransmission process.

Figure 11-1: Conceptual view of the TCP retransmission process

When a packet is sent, but the recipient has not sent back a TCP ACK packet, the transmitting host assumes that the original packet was lost and retransmits the original packet. When the retransmission is sent, the RTO value is doubled; if no ACK packet is received before that value is reached, another retransmission will occur. If this retransmission also does not receive an ACK response, the RTO value is doubled again. This process will continue, with the RTO value being doubled for each retransmission, until an ACK packet is received or until the sender reaches the maximum number of retransmission attempts it is configured to send. More details about this process are described in RFC6298.

The maximum number of retransmission attempts depends on the value configured in the transmitting operating system. By default, Windows hosts make a maximum of five retransmission attempts. Most Linux hosts default to a maximum of 15 attempts. This option is configurable in either operating system.

For an example of TCP retransmission, open the file *tcp_retransmissions .pcapng*, which contains six packets. The first packet is shown in Figure 11-2.

Figure 11-2: A simple TCP packet containing data

This packet is a TCP PSH/ACK packet ❷ containing 648 bytes of data ❸ that are sent from 10.3.30.1 to 10.3.71.7 ❶. This is a typical data packet.

Under normal circumstances, you would expect to see a TCP ACK packet in response fairly soon after the first packet is sent. In this case, however, the next packet is a retransmission. You can tell this by looking at the packet in the Packet List pane. The Info column will clearly say `[TCP Retransmission]`, and the packet will appear with red text on a black background. Figure 11-3 shows examples of retransmissions listed in the Packet List pane.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	10.3.30.1	10.3.71.7	TCP	706	1048 → 1043 [PSH, ACK] Seq=1 Ack=1 Win=8760 Len=648 [ETHERNET FRAME CHECK SEQUENCE INCORRECT]
2	0.206000	10.3.30.1	10.3.71.7	TCP	706	[TCP Retransmission] 1048 → 1043 [PSH, ACK] Seq=1 Ack=1 Win=8760 Len=648 [ETHERNET FRAME CHECK SEQUENCE INCORRECT]
3	0.600000	10.3.30.1	10.3.71.7	TCP	706	[TCP Retransmission] 1048 → 1043 [PSH, ACK] Seq=1 Ack=1 Win=8760 Len=648 [ETHERNET FRAME CHECK SEQUENCE INCORRECT]
4	1.200000	10.3.30.1	10.3.71.7	TCP	706	[TCP Retransmission] 1048 → 1043 [PSH, ACK] Seq=1 Ack=1 Win=8760 Len=648 [ETHERNET FRAME CHECK SEQUENCE INCORRECT]
5	2.400000	10.3.30.1	10.3.71.7	TCP	706	[TCP Retransmission] 1048 → 1043 [PSH, ACK] Seq=1 Ack=1 Win=8760 Len=648 [ETHERNET FRAME CHECK SEQUENCE INCORRECT]
6	4.805000	10.3.30.1	10.3.71.7	TCP	706	[TCP Retransmission] 1048 → 1043 [PSH, ACK] Seq=1 Ack=1 Win=8760 Len=648 [ETHERNET FRAME CHECK SEQUENCE INCORRECT]

Figure 11-3: Retransmissions in the Packet List pane

You can also determine whether a packet is a retransmission by examining it in the Packet Details pane, as shown in Figure 11-4.

In the Packet Details pane, notice that the retransmission packet has some additional information included under the SEQ/ACK analysis heading ❶. This useful information is provided by Wireshark and is not contained in the packet itself. The SEQ/ACK analysis tells us that this is indeed a retransmission ❷, that the RTO value is 0.206 seconds ❸, and that the RTO is based on the delta time from packet 1 ❹.

Figure 11-4: An individual retransmission packet

Note that this packet is the same as the original packet (other than the IP identification and Checksum fields). To verify this, compare the Packet Bytes pane of this retransmitted packet with the original one.

Examination of the remaining packets should yield similar results, with the only differences between the packets found in the IP identification and Checksum fields and in the RTO value. To visualize the time lapse between each packet, look at the Time column in the Packet List pane, as shown in Figure 11-5. Here, you see exponential growth in time as the RTO value is doubled after each retransmission.

No.	Time
1	0.000000
2	0.206000
3	0.600000
4	1.200000
5	2.400000
6	4.805000

Figure 11-5: The Time column shows the increase in RTO value.

The TCP retransmission feature is used by the transmitting device to detect and recover from packet loss. Next, we'll examine *TCP duplicate acknowledgments*, a feature that the data recipient uses to detect and recover from packet loss.

TCP Duplicate Acknowledgments and Fast Retransmissions

tcp_dupack.pcapng

A duplicate ACK is a TCP packet sent from a recipient when that recipient receives packets that are out of order. TCP uses the sequence and acknowledgment number fields within its header to reliably ensure that data is received and reassembled in the same order in which it was sent.

NOTE *The proper term for a TCP packet is actually* TCP segment, *but most people refer to it as a packet.*

When a new TCP connection is established, one of the most important pieces of information exchanged during the handshake process is an initial sequence number (ISN). Once the ISN is set for each side of the connection, each subsequently transmitted packet increments the sequence number by the size of its data payload.

Consider a host that has an ISN of 5000 and sends a 500-byte packet to a recipient. Once this packet has been received, the recipient host will respond with a TCP ACK packet with an acknowledgment number of 5500, based on the following formula:

Sequence Number In + Bytes of Data Received = Acknowledgment Number Out

As a result of this calculation, the acknowledgment number returned to the transmitting host is the next sequence number that the recipient expects to receive. An example of this can be seen in Figure 11-6.

Figure 11-6: TCP sequence and acknowledgment numbers

The detection of packet loss by the data recipient is made possible through the sequence numbers. As the recipient tracks the sequence numbers it is receiving, it can determine when it receives sequence numbers that are out of order.

When the recipient receives an unexpected sequence number, it assumes that a packet has been lost in transit. To reassemble data properly, the recipient must have the missing packet, so it resends the ACK packet that contains the lost packet's expected sequence number in order to elicit a retransmission of that packet from the transmitting host.

When the transmitting host receives three duplicate ACKs from the recipient, it assumes that the packet was indeed lost in transit and immediately sends a *fast retransmission*. Once a fast retransmission is triggered, all other packets being transmitted are queued until the fast retransmission packet is sent. This process is depicted in Figure 11-7.

Figure 11-7: Duplicate ACKs from the recipient result in a fast retransmission.

You'll find an example of duplicate ACKs and fast retransmissions in the file *tcp_dupack.pcapng*. The first packet in this capture is shown in Figure 11-8.

Figure 11-8: The ACK showing the next expected sequence number

This packet, a TCP ACK sent from the data recipient (172.31.136.85) to the transmitter (195.81.202.68) ❶, has an acknowledgment of the data sent in the previous packet that is not included in this capture file.

NOTE *By default, Wireshark uses relative sequence numbers to make the analysis of these numbers easier, but the examples and screenshots in the next few sections do not use this feature. To turn off this feature, select* ***Edit ▸ Preferences****. In the Preferences window, select* ***Protocols*** *and then the* ***TCP*** *section. Then uncheck the box next to* ***Relative sequence numbers****.*

The acknowledgment number in this packet is 1310973186 ❷. This should match the sequence number in the next packet received, as shown in Figure 11-9.

Figure 11-9: The sequence number of this packet is not what was expected.

Unfortunately for us and our recipient, the sequence number of the next packet is 1310984130 ❶, which is not what we expect. This out-of-order packet indicates that the expected packet was somehow lost in transit. The recipient host notices that this packet is out of sequence and sends a duplicate ACK in the third packet of this capture, as shown in Figure 11-10.

You can determine that this is a duplicate ACK packet by examining either of the following:

- The Info column in the Packet Details pane. The packet should appear as red text on a black background.
- The Packet Details pane under the SEQ/ACK analysis heading (Figure 11-10). If you expand this heading, you'll find that the packet is listed as a duplicate ACK of packet 1 ❶.

Figure 11-10: The first duplicate ACK packet

The next several packets continue this process, as shown in Figure 11-11.

	No.	Time	Source	Destination	Protocol	Length	Info
	1	0.000000	172.31.136.85	195.81.202.68	TCP	78	38760 → 80 [ACK] Seq=704338729 Ack=1310973186 Win=382 Len=0 TSval=22247173 TSecr=1332354 …
	2	0.000190	195.81.202.68	172.31.136.85	TCP	1434	80 → 38760 [ACK] Seq=1310984130 Ack=704338729 Win=108 Len=1368 TSval=1332354 TSecr=222471…
	3	0.000011	172.31.136.85	195.81.202.68	TCP	78	[TCP Dup ACK 1#1] 38760 → 80 [ACK] Seq=704338729 Ack=1310973186 Win=382 Len=0 TSval=22247…
❶	4	0.000093	195.81.202.68	172.31.136.85	TCP	1434	80 → 38760 [ACK] Seq=1310985498 Ack=704338729 Win=108 Len=1368 TSval=1332354 TSecr=222471…
❷	5	0.000010	172.31.136.85	195.81.202.68	TCP	78	[TCP Dup ACK 1#2] 38760 → 80 [ACK] Seq=704338729 Ack=1310973186 Win=382 Len=0 TSval=22247…
❸	6	0.000121	195.81.202.68	172.31.136.85	TCP	1434	80 → 38760 [ACK] Seq=1310986866 Ack=704338729 Win=108 Len=1368 TSval=1332354 TSecr=222471…
❹	7	0.000010	172.31.136.85	195.81.202.68	TCP	78	[TCP Dup ACK 1#3] 38760 → 80 [ACK] Seq=704338729 Ack=1310973186 Win=382 Len=0 TSval=22247…

Figure 11-11: Additional duplicate ACKs are generated due to out-of-order packets.

The fourth packet in the capture file is another chunk of data sent from the transmitting host with the wrong sequence number ❶. As a result, the recipient host sends its second duplicate ACK ❷. One more packet with the wrong sequence number is received by the recipient ❸. That prompts the transmission of the third and final duplicate ACK ❹.

As soon as the transmitting host receives the third duplicate ACK from the recipient, it is forced to halt all packet transmission and resend the lost packet. Figure 11-12 shows the fast retransmission of the lost packet.

The retransmission packet can once again be found through the Info column in the Packet List pane. As with previous examples, the packet is clearly labeled with red text on a black background. The SEQ/ACK analysis section of this packet (Figure 11-12) tells us that this is suspected to be a fast retransmission ❶. (Again, the information that labels this packet as a fast retransmission is not a value set in the packet itself but rather a feature of Wireshark.) The final packet in the capture is an ACK packet acknowledging receipt of the fast retransmission.

Figure 11-12: Three duplicate ACKs cause this fast retransmission of the lost packet.

NOTE *One feature to consider that may affect the flow of data in TCP communications in which packet loss is present is the Selective Acknowledgment feature. In the packet capture we just examined, Selective ACK was negotiated as an enabled feature during the initial three-way handshake process. As a result, whenever a packet is lost and a duplicate ACK received, only the lost packet has to be retransmitted, even though other packets were received successfully after the lost packet. Had Selective ACK not been enabled, every packet occurring after the lost packet would have had to be retransmitted as well. Selective ACK makes data loss recovery much more efficient. Because most modern TCP/IP stack implementations support Selective ACK, you will find that this feature is usually implemented.*

TCP Flow Control

Retransmissions and duplicate ACKs are reactive TCP functions designed to recover from packet loss. TCP would be a poor protocol indeed if it didn't include some form of proactive method for preventing packet loss.

TCP implements a *sliding-window mechanism* to detect when packet loss may occur and adjust the rate of data transmission to prevent it. The sliding-window mechanism leverages the data recipient's *receive window* to control the flow of data.

The receive window is a value specified by the data recipient and stored in the TCP header (in bytes) that tells the transmitting device how much data the recipient is willing to store in its *TCP buffer space*. This buffer space is where data is stored temporarily until it can be passed up the stack to the

application-layer protocol waiting to process it. As a result, the transmitting host can send only the amount of data specified in the Window size value field at one time. For the transmitter to send more data, the recipient must send an acknowledgment that the previous data was received. It also must clear TCP buffer space by processing the data that is occupying that position. Figure 11-13 illustrates how the receive window works.

Figure 11-13: The receive window keeps the data recipient from getting overwhelmed.

In Figure 11-13, the client is sending data to a server that has communicated a receive window size of 5,000 bytes. The client sends 2,500 bytes, reducing the server's buffer space to 2,500 bytes, and then sends another 2,000 bytes, further reducing the buffer to 500 bytes. The server sends an acknowledgment of this data, and after it processes the data in its buffer, it again has an empty buffer available. This process repeats, with the client sending 3,000 bytes and another 1,000 bytes, reducing the server's buffer to 1,000 bytes. The client once more acknowledges this data and processes the contents of its buffer.

Adjusting the Window Size

This process of adjusting the window size is fairly clear-cut, but it isn't always perfect. Whenever data is received by the TCP stack, an acknowledgment is generated and sent in reply, but the data placed in the recipient's buffer is not always processed immediately.

When a busy server is processing packets from multiple clients, it could quite possibly be slow in clearing its buffer and thus be unable to make room for new data. Without a means of flow control, a full buffer could lead to lost packets and data corruption. Fortunately, when a server becomes too busy to process data at the rate its receive window is advertising, it can adjust the size

of the window. It does this by decreasing the window size value in the TCP header of the ACK packet it is sending back to the hosts that are sending it data. Figure 11-14 shows an example of this.

Figure 11-14: The window size can be adjusted when the server becomes busy.

In Figure 11-14, the server starts with an advertised window size of 5,000 bytes. The client sends 2,000 bytes, followed by another 2,000 bytes, leaving only 1,000 bytes of buffer space available. The server realizes that its buffer is filling up quickly and knows that if data transfer keeps up at this rate, packets will soon be lost. To avoid such a mishap, the server sends an acknowledgment to the client with an updated window size of 1,000 bytes. The client responds by sending less data, and now the rate at which the server can process its buffer contents allows data to flow in a constant manner.

The resizing process works both ways. When the server can process data at a faster rate, it can send an ACK packet with a larger window size.

Halting Data Flow with a Zero Window Notification

Due to a lack of memory, a lack of processing capability, or another problem, a server may no longer process data sent from a client. Such a stoppage could result in dropped packets and a halting of the communication process, but the receive window can minimize negative effects.

When this situation arises, a server can send a packet that contains a window size of zero. When the client receives this packet, it will halt any data transmission but will sometimes keep the connection to the server open with the transmission of *keep-alive packets*. Keep-alive packets can be sent by the client at regular intervals to check the status of the server's

receive window. Once the server can begin processing data again, it will respond with a nonzero window size, and communication will resume. Figure 11-15 illustrates an example of zero window notification.

Figure 11-15: Data transfer stops when the window size is set to 0 bytes.

In Figure 11-15, the server begins receiving data with a 5,000-byte window size. After receiving a total of 4,000 bytes of data from the client, the server begins experiencing a very heavy processor load, and it can no longer process any data from the client. The server then sends a packet with the Window size value field set to 0. The client halts transmission of data and sends a keep-alive packet. After receiving the keep-alive packet, the server responds with a packet notifying the client that it can now receive data and that its window size is 1,000 bytes. The client resumes sending data but at a slower rate than before.

The TCP Sliding Window in Practice

tcp_zerowindowrecovery.pcapng
tcp_zerowindowdead.pcapng

Having covered the theory behind the TCP sliding window, we will now examine it in the capture file *tcp_zerowindowrecovery.pcapng*.

In this file, we begin with several TCP ACK packets traveling from 192.168.0.20 to 192.168.0.30. The main value of interest to us is the Window size value field, which can be seen in both the Info column of the Packet List pane and in the TCP header in the Packet Details pane. You can see immediately that this field's value decreases over the course of the first three packets, as shown in Figure 11-16.

No.	Time ❶	Source	Destination	Protocol	Length	Info ❷
1	0.000000	192.168.0.20	192.168.0.30	TCP	60	2235 → 1720 [ACK] Seq=1422793785 Ack=2710996659 Win=8760 Len=0
2	0.000237	192.168.0.20	192.168.0.30	TCP	60	2235 → 1720 [ACK] Seq=1422793785 Ack=2710999579 Win=5840 Len=0
3	0.000193	192.168.0.20	192.168.0.30	TCP	60	2235 → 1720 [ACK] Seq=1422793785 Ack=2711002499 Win=2920 Len=0

Figure 11-16: The window size of these packets is decreasing.

The window size value goes from 8,760 bytes in the first packet to 5,840 bytes in the second packet and then 2,920 bytes in the third packet ❷. This lowering of the window size value is a classic indicator of increased latency from the host. Notice in the Time column that this happens very quickly ❶. When the window size is lowered this fast, it's common for the window size to drop to zero, which is exactly what happens in the fourth packet, as shown in Figure 11-17.

Figure 11-17: This zero window packet says that the host cannot accept any more data.

The fourth packet is also being sent from 192.168.0.20 to 192.168.0.30, but its purpose is to tell 192.168.0.30 that it can no longer receive any data. The 0 value is seen in the TCP header ❶. Wireshark also tells us that this is a zero window packet in the Info column of the Packet List pane and under the SEQ/ACK analysis section of the TCP header ❷.

Once this zero window packet is sent, the device at 192.168.0.30 will not send any more data until it receives a window update from 192.168.0.20 notifying it that the window size has increased. Luckily for us, the issue causing the zero window condition in this capture file was only temporary. So, a window update is sent in the next packet, shown in Figure 11-18.

In this case, the window size is increased to a very healthy 64,240 bytes ❶. Wireshark once again lets us know that this is a window update under the SEQ/ACK analysis heading.

Once the update packet is received, the host at 192.168.0.30 can begin sending data again, as it does in packets 6 and 7. This entire period of halted data transmission takes place very quickly. Had it lasted only slightly longer, it could have caused a potential hiccup on the network, resulting in a slower or failed data transfer.

```
Wireshark · Packet 5 · tcp_zerowindowrecovery
> Frame 5: 60 bytes on wire (480 bits), 60 bytes captured (480 bits)
> Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware_c0:00:01 (00:50:56:c0:00:01)
> Internet Protocol Version 4, Src: 192.168.0.20, Dst: 192.168.0.30
v Transmission Control Protocol, Src Port: 2235 (2235), Dst Port: 1720 (1720), Seq: 1422793785, Ack: 2711005419, Len: 0
    Source Port: 2235
    Destination Port: 1720
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1422793785
    Acknowledgment number: 2711005419
    Header Length: 20 bytes
  > Flags: 0x010 (ACK)
    Window size value: 64240 ❶
    [Calculated window size: 64240]
    [Window size scaling factor: -1 (unknown)]
  > Checksum: 0x6864 [validation disabled]
    Urgent pointer: 0
  v [SEQ/ACK analysis]
    v [TCP Analysis Flags]
      v [Expert Info (Chat/Sequence): TCP window update]
          [TCP window update]
          [Severity level: Chat]
          [Group: Sequence]
No.: 5 · Time: 0.010005 · Source: 192.168.0.20 · Destination: 192.168.0.30 · Protocol: TCP · Length: 60 · Info: [TCP Window Update] 2235 → 1720 [ACK] Seq=1422793785 Ack=2711005419 Win=64240 Len
                                                                                                  Close   Help
```

Figure 11-18: A TCP window update packet lets the other host know it can begin transmitting again.

For one last look at the sliding window, examine *tcp_zerowindowdead.pcapng*. The first packet in this capture is normal HTTP traffic being sent from 195.81.202.68 to 172.31.136.85. The packet is immediately followed with a zero window packet sent back from 172.31.136.85, as shown in Figure 11-19.

```
Wireshark · Packet 2 · tcp_zerowindowdead
> Frame 2: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
> Ethernet II, Src: HewlettP_a4:c1:c6 (00:16:35:a4:c1:c6), Dst: All-HSRP-routers_01 (00:00:0c:07:ac:01)
> Internet Protocol Version 4, Src: 172.31.136.85, Dst: 195.81.202.68
v Transmission Control Protocol, Src Port: 38760 (38760), Dst Port: 80 (80), Seq: 704338729, Ack: 1310997786, Len: 0
    Source Port: 38760
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 704338729
    Acknowledgment number: 1310997786
    Header Length: 32 bytes
  > Flags: 0x010 (ACK)
    Window size value: 0
    [Calculated window size: 0]
    [Window size scaling factor: -1 (unknown)]
  > Checksum: 0x36d0 [validation disabled]
    Urgent pointer: 0
  > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  v [SEQ/ACK analysis]
      [This is an ACK to the segment in frame: 1]
      [The RTT to ACK the segment was: 0.000029000 seconds]
    v [TCP Analysis Flags]
      > [Expert Info (Warn/Sequence): TCP Zero Window segment]
No.: 2 · Time: 0.000029 · Source: 172.31.136.85 · Destination: 195.81.202.68 · Protocol: TCP ·...Window] 38760 → 80 [ACK] Seq=704338729 Ack=1310997786 Win=0 Len=0 TSval=22248305 TSecr=13
                                                                                                  Close   Help
```

Figure 11-19: A zero window packet halts data transfer.

This looks very similar to the zero window packet shown in Figure 11-17, but the result is much different. Rather than seeing a window update from the 172.31.136.85 host and the resumption of communication, we see a keep-alive packet, as shown in Figure 11-20.

```
Wireshark · Packet 3 · tcp_zerowindowdead
> Frame 3: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
> Ethernet II, Src: CiscoInc_72:15:00 (00:0b:be:72:15:00), Dst: HewlettP_a4:c1:c6 (00:16:35:a4:c1:c6)
> Internet Protocol Version 4, Src: 195.81.202.68, Dst: 172.31.136.85
v Transmission Control Protocol, Src Port: 80 (80), Dst Port: 38760 (38760), Seq: 1310997785, Ack: 704338729, Len: 0
    Source Port: 80
    Destination Port: 38760
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 1310997785
    Acknowledgment number: 704338729
    Header Length: 32 bytes
  > Flags: 0x010 (ACK)
    Window size value: 108
    [Calculated window size: 108]
    [Window size scaling factor: -1 (unknown)]
  > Checksum: 0x3311 [validation disabled]
    Urgent pointer: 0
  > Options: (12 bytes), No-Operation (NOP), No-Operation (NOP), Timestamps
  v [SEQ/ACK analysis]
    v [TCP Analysis Flags]
      v [Expert Info (Note/Sequence): TCP keep-alive segment] ❶
          [TCP keep-alive segment]
          [Severity level: Note]
          [Group: Sequence]
No.: 3 · Time: 3.410576 · Source: 195.81.202.68 · Destination: 172.31.136.85 · Protocol: TCP ·…live] 80 → 38760 [ACK] Seq=1310997785 Ack=704338729 Win=108 Len=0 TSval=1334338 TSecr=2224
                                                                                    Close   Help
```

Figure 11-20: This keep-alive packet ensures the zero window host is still alive.

This packet is marked as a keep-alive by Wireshark under the SEQ/ACK analysis section of the TCP header in the Packet Details pane ❶. The Time column tells us that this packet was sent 3.4 seconds after the last received packet. This process continues several more times, with one host sending a zero window packet and the other sending a keep-alive packet, as shown in Figure 11-21.

No.	Time ❶	Source	Destination	Protocol	Length	Info
2	0.000029	172.31.136.85	195.81.202.68	TCP	66	[TCP ZeroWindow] 38760 → 80 [ACK] Seq=704338729 Ack=1310997786 Win=0 Len=0 TSv…
3	3.410576	195.81.202.68	172.31.136.85	TCP	66	[TCP Keep-Alive] 80 → 38760 [ACK] Seq=1310997785 Ack=704338729 Win=108 Len=0 T…
4	0.000031	172.31.136.85	195.81.202.68	TCP	66	[TCP ZeroWindow] 38760 → 80 [ACK] Seq=704338729 Ack=1310997786 Win=0 Len=0 TSv…
5	6.784127	195.81.202.68	172.31.136.85	TCP	66	[TCP Keep-Alive] 80 → 38760 [ACK] Seq=1310997785 Ack=704338729 Win=108 Len=0 T…
6	0.000029	172.31.136.85	195.81.202.68	TCP	66	[TCP ZeroWindow] 38760 → 80 [ACK] Seq=704338729 Ack=1310997786 Win=0 Len=0 TSv…
7	13.536714	195.81.202.68	172.31.136.85	TCP	66	[TCP Keep-Alive] 80 → 38760 [ACK] Seq=1310997785 Ack=704338729 Win=108 Len=0 T…
8	0.000047	172.31.136.85	195.81.202.68	TCP	66	[TCP ZeroWindow] 38760 → 80 [ACK] Seq=704338729 Ack=1310997786 Win=0 Len=0 TSv…

Figure 11-21: The host and client continue to send zero window and keep-alive packets, respectively.

These keep-alive packets occur at intervals of 3.4, 6.8, and 13.5 seconds ❶. This process can go on for quite a long time, depending on the operating systems of the communicating devices. As you can see by adding up the values in the Time column, the connection is halted for nearly

25 seconds. Imagine attempting to authenticate with a domain controller or download a file from the internet while experiencing a 25-second delay—unacceptable!

Learning from TCP Error-Control and Flow-Control Packets

Let's put retransmission, duplicate ACKs, and the sliding-window mechanism into some context. Here are a few notes to keep in mind when troubleshooting latency issues.

Retransmission Packets

Retransmissions occur because the client has detected that the server is not receiving the data it's sending. Therefore, depending on which side of the communication you are analyzing, you may never see retransmissions. If you are capturing data from the server, and it is truly not receiving the packets being sent and retransmitted from the client, you may be in the dark because you won't see the retransmission packets. If you suspect that you are the victim of packet loss on the server side, consider attempting to capture traffic from the client (if possible) so that you can see whether retransmission packets are present.

Duplicate ACK Packets

I tend to think of a duplicate ACK as the pseudo-opposite of a retransmission, because it is sent when the server detects that a packet from the client it is communicating with was lost in transit. In most cases, you can see duplicate ACKs when capturing traffic on both sides of the communication. Remember that duplicate ACKs are triggered when packets are received out of sequence. For example, if the server received just the first and third of three packets sent, it would send a duplicate ACK to elicit a fast retransmission of the second packet from the client. Since you have received the first and third packets, it's likely that whatever condition caused the second packet to be dropped was only temporary, so the duplicate ACK will likely be sent and received successfully. Of course, this scenario isn't always the case, so when you suspect packet loss on the server side and don't see any duplicate ACKs, consider capturing packets from the client side of the communication.

Zero Window and Keep-Alive Packets

The sliding window relates directly to the server's inability to receive and process data. Any decrease in the window size or a zero window state is a direct result of some issue with the server, so if you see either occurring on the wire, you should focus your investigation there. You will typically see window update packets on both sides of network communications.

Locating the Source of High Latency

In some cases, packet loss may not be the cause of latency. You may find that even though communications between two hosts are slow, that slowness doesn't show the common symptoms of TCP retransmissions or duplicate ACKs. Thus, you need another technique to locate the source of the high latency.

One of the most effective ways to find the source of high latency is to examine the initial connection handshake and the first couple of packets that follow it. For example, consider a simple connection between a client and a web server as the client attempts to browse a site hosted on the web server. We are concerned with the first six packets of this communication sequence, consisting of the TCP handshake, the initial HTTP `GET` request, the acknowledgment of that `GET` request, and the first data packet sent from the server to the client.

NOTE *To follow along with this section, ensure that you have the proper time display format set in Wireshark by selecting* ***View ▸ Time Display Format ▸ Seconds Since Previous Displayed Packet****.*

Normal Communications

latency1.pcapng

We'll discuss network baselines in detail a little later in the chapter. For now, just know that you need a baseline of normal communications to compare with the conditions of high latency. For these examples, we will use the file *latency1.pcapng*. We have already covered the details of the TCP handshake and HTTP communication, so we won't review those topics again. In fact, we won't look at the Packet Details pane at all. All we are really concerned about is the Time column, as shown in Figure 11-22.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	66	1606 → 80 [SYN] Seq=2082691767 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.030107	74.125.95.104	172.16.16.128	TCP	66	80 → 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 Win=5720 Len=0 MSS=1406 SACK_PERM=1 WS=64
3	0.000075	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082691768 Ack=2775577374 Win=16872 Len=0
4	0.000066	172.16.16.128	74.125.95.104	HTTP	681	GET / HTTP/1.1
5	0.048778	74.125.95.104	172.16.16.128	TCP	60	80 → 1606 [ACK] Seq=2775577374 Ack=2082692395 Win=6976 Len=0
6	0.022176	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]

Figure 11-22: This traffic happens very quickly and can be considered normal.

This communication sequence is quite quick, with the entire process taking less than 0.1 seconds.

The next few capture files we'll examine will consist of this same traffic pattern but with differences in the timing of the packets.

Slow Communications: Wire Latency

latency2.pcapng

Now let's turn to the capture file *latency2.pcapng*. Notice that all of the packets are the same except for the time values in two of them, as shown in Figure 11-23.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	66	1606 → 80 [SYN] Seq=2082691767 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.878530	74.125.95.104	172.16.16.128	TCP	66	80 → 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 Win=5720 Len=0 MSS=1406 SACK_PERM=1 WS=64
3	0.016604	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082691768 Ack=2775577374 Win=16872 Len=0
4	0.000335	172.16.16.128	74.125.95.104	HTTP	681	GET / HTTP/1.1
5	1.155228	74.125.95.104	172.16.16.128	TCP	60	80 → 1606 [ACK] Seq=2775577374 Ack=2082692395 Win=6976 Len=0
6	0.015866	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]

Figure 11-23: Packets 2 and 5 show high latency.

As we begin stepping through these six packets, we encounter our first sign of latency immediately. The initial SYN packet is sent by the client (172.16.16.128) to begin the TCP handshake, and a delay of 0.87 seconds is seen before the return SYN/ACK is received from the server (74.125.95.104). This is our first indicator that we are experiencing wire latency, which is caused by a device between the client and server.

We can make the determination that this is wire latency because of the nature of the types of packets being transmitted. When the server receives a SYN packet, a very minimal amount of processing is required to send a reply, because the workload doesn't involve any processing above the transport layer. Even when a server is experiencing a very heavy traffic load, it will typically respond quickly to a SYN packet with a SYN/ACK. This eliminates the server as the potential cause of the high latency.

The client is also eliminated because, at this point, it is not doing any processing beyond simply receiving the SYN/ACK packet. Elimination of both the client and server points us to potential sources of slow communication within the first two packets of this capture.

Continuing, we see that the transmission of the ACK packet that completes the three-way handshake occurs quickly, as does the HTTP `GET` request sent by the client. All of the processing that generates these two packets occurs locally on the client following receipt of the SYN/ACK, so these two packets are expected to be transmitted quickly, as long as the client is not under a heavy processing load.

At packet 5, we see another packet with an incredibly high time value. It appears that after our initial HTTP `GET` request was sent, the ACK packet returned from the server took 1.15 seconds to be received. Upon receipt of the HTTP `GET` request, the server sent a TCP ACK before it began sending data, which once again requires very little processing by the server. This is another sign of wire latency.

Whenever you experience wire latency, you will almost always see it exhibited in both the SYN/ACK during the initial handshake and in other ACK packets throughout the communication. Although this information doesn't tell you the exact source of the high latency on this network, it does tell you that neither client nor server is the source, so you know that the latency is due to some device in between. At this point, you could begin examining the various firewalls, routers, and proxies to locate the culprit.

Slow Communications: Client Latency

latency3.pcapng

The next latency scenario we'll examine is contained in *latency3.pcapng*, as shown in Figure 11-24.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	66	1606 → 80 [SYN] Seq=2082691767 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.023790	74.125.95.104	172.16.16.128	TCP	66	80 → 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 Win=5720 Len=0 MSS=1406 SACK_PERM=1 WS=64
3	0.014894	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082691768 Ack=2775577374 Win=16872 Len=0
4	1.345023	172.16.16.128	74.125.95.104	HTTP	681	GET / HTTP/1.1
5	0.046121	74.125.95.104	172.16.16.128	TCP	60	80 → 1606 [ACK] Seq=2775577374 Ack=2082692395 Win=6976 Len=0
6	0.016182	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]

Figure 11-24: The slow packet in this capture is the initial HTTP `GET`.

This capture begins normally, with the TCP handshake occurring very quickly and without any signs of latency. Everything appears to be fine until packet 4, which is an HTTP `GET` request after the handshake has completed. This packet shows a 1.34-second delay from the previously received packet.

To determine the source of this delay, we need to examine what is occurring between packets 3 and 4. Packet 3 is the final ACK in the TCP handshake sent from the client to the server, and packet 4 is the `GET` request sent from the client to the server. The common thread here is that these are both packets sent by the client and are independent of the server. The `GET` request should occur quickly after the ACK is sent, since all of these actions are centered on the client.

Unfortunately for the end user, the transition from ACK to `GET` doesn't happen quickly. The creation and transmission of the `GET` packet requires processing up to the application layer, and the delay in this processing indicates that the client was unable to perform the action in a timely manner. Thus, the client is ultimately responsible for the high latency in the communication.

Slow Communications: Server Latency

latency4.pcapng

The last latency scenario we'll examine uses the file *latency4.pcapng*, as shown in Figure 11-25. This is an example of server latency.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	172.16.16.128	74.125.95.104	TCP	66	1606 → 80 [SYN] Seq=2082691767 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
2	0.018583	74.125.95.104	172.16.16.128	TCP	66	80 → 1606 [SYN, ACK] Seq=2775577373 Ack=2082691768 Win=5720 Len=0 MSS=1406 SACK_PERM=1 WS=64
3	0.016197	172.16.16.128	74.125.95.104	TCP	54	1606 → 80 [ACK] Seq=2082691768 Ack=2775577374 Win=16872 Len=0
4	0.000172	172.16.16.128	74.125.95.104	HTTP	681	GET / HTTP/1.1
5	0.047936	74.125.95.104	172.16.16.128	TCP	60	80 → 1606 [ACK] Seq=2775577374 Ack=2082692395 Win=6976 Len=0
6	0.982983	74.125.95.104	172.16.16.128	TCP	1460	[TCP segment of a reassembled PDU]

Figure 11-25: High latency isn't exhibited until the last packet of this capture.

In this capture, the TCP handshake process between these two hosts completes flawlessly and quickly, so things begin well. The next couple of packets bring more good news, as the initial `GET` request and response ACK packets are delivered quickly as well. It is not until the last packet in this file that we see signs of high latency.

This sixth packet is the first HTTP data packet sent from the server in response to the `GET` request sent by the client, and it has a slow arrival time of 0.98 seconds after the server sends its TCP ACK for the `GET` request. The transition between packets 5 and 6 is very similar to the transition we saw in the previous scenario between the handshake ACK and `GET` request. However, in this case, the server is the focus of our concern.

Packet 5 is the ACK that the server sends in response to the `GET` request from the client. As soon as that packet has been sent, the server should begin

sending data almost immediately. The accessing, packaging, and transmitting of the data in this packet is done by the HTTP protocol, and because this is an application-layer protocol, a bit of processing is required by the server. The delay in receipt of this packet indicates that the server was unable to process this data in a reasonable amount of time, ultimately pointing to it as the source of latency in this capture file.

Latency Locating Framework

Using six packets, we've managed to locate the source of high network latency between the client and the server in several scenarios. The diagram in Figure 11-26 should help you troubleshoot your own latency issues. These principles can be applied to almost any TCP-based communication.

Figure 11-26: This diagram can be used to troubleshoot your own latency issues.

NOTE *Notice that we have not talked a lot about UDP latency. Because UDP is designed to be quick but unreliable, it doesn't have any built-in features to detect and recover from latency. Instead, it relies on the application-layer protocols (and ICMP) that it's paired with to handle data delivery reliability.*

Network Baselining

When all else fails, your *network baseline* can be one of the most crucial pieces of data you have when troubleshooting slowness on the network.

For our purposes, a network baseline consists of a sample of traffic from various points on the network that includes a large chunk of what we would consider "normal" network traffic. The goal of having a network baseline is for it to serve as a basis of comparison when the network or devices on it are misbehaving.

For example, consider a scenario in which several clients on the network complain of slowness when logging in to a local web application server. If you were to capture this traffic and compare it to a network baseline, you might find that the web server is responding normally but that the external DNS requests resulting from external content embedded in the web application are running twice as slowly as normal.

You might have noticed the slow external DNS server without the aid of a network baseline, but when you are dealing with subtle changes, that may not be the case. Ten DNS queries taking 0.1 seconds longer than normal to process are just as bad as one DNS query taking 1 full second longer than normal, but the former situation is much harder to detect without a network baseline.

Because no two networks are alike, the components of a network baseline can vary drastically. The following sections provide examples of the components of a network baseline. You may find that all of these items apply to your network infrastructure or that very few of them do. Regardless, you should be able to place each component of your baseline inside one of three basic baseline categories: site, host, and application.

Site Baseline

The purpose of the site baseline is to gain an overall snapshot of the traffic at each physical site on your network. Ideally, this would be every segment of the WAN.

Components of this baseline might include the following:

Protocols in Use

To see traffic from all devices, use the Protocol Hierarchy Statistics window (**Statistics ▸ Protocol Hierarchy**) while capturing traffic from all the devices on the network segment at the network edge (router/firewall). Later, you can compare against the hierarchy output to find out whether normally present protocols are missing or new protocols have introduced themselves on the network. You can also use this output to find above ordinary amounts of certain types of traffic based on protocol.

Broadcast Traffic

This includes all broadcast traffic on the network segment. Sniffing at any point within the site should let you capture all of the broadcast traffic, allowing you to know who or what normally sends a lot of broadcast out on the network. Then you can quickly determine whether you have too much (or not enough) broadcasting going on.

Authentication Sequences

These include traffic from authentication processes on random clients to all services, such as Active Directory, web applications, and organization-specific software. Authentication is one area in which services are commonly slow. The baseline allows you to determine whether authentication is to blame for slow communications.

Data Transfer Rate

This usually consists of a measure of a large data transfer from the site to various other sites in the network. You can use the capture summary and graphing features of Wireshark (demonstrated in Chapter 5) to determine the transfer rate and consistency of the connection. This is probably the most important site baseline you can have. Whenever any connection entering or leaving the network segment seems slow, you can perform the same data transfer as in your baseline and compare the results. This will tell you whether the connection is actually slow and will possibly even help you find the area in which the slowness begins.

Host Baseline

You probably don't need to baseline every single host within your network. The host baseline should be performed on only high-traffic or mission-critical servers. Basically, if a slow server will result in angry phone calls from management, you should have a baseline of that host.

Components of the host baseline include the following:

Protocols in Use

This baseline provides a good opportunity to use the Protocol Hierarchy Statistics window while capturing traffic from the host. Later, you can compare against this baseline to find out whether normally present protocols are missing or new protocols have introduced themselves on the host. You can also use this to find unusually large amounts of certain types of traffic based on protocol.

Idle/Busy Traffic

This baseline simply consists of general captures of normal operating traffic during peak and off-peak times. Knowing the number of connections and amount of bandwidth used by those connections at different times of the day will allow you to determine whether slowness is a result of user load or another issue.

Startup/Shutdown

To obtain this baseline, you'll need to create a capture of the traffic generated during the startup and shutdown sequences of the host. If the computer refuses to boot, refuses to shut down, or is abnormally slow during either sequence, you can use this baseline to determine whether the cause is network related.

Authentication Sequences

Getting this baseline requires capturing traffic from authentication processes to all services on the host. Authentication is one area in which services are commonly slow. The baseline allows you to determine whether authentication is to blame for slow communications.

Associations/Dependencies

This baseline consists of a longer-duration capture to determine what other hosts this host is dependent upon (and are dependent upon this host). You can use the Conversations window (**Statistics ▸ Conversations**) to see these associations and dependencies. An example is a SQL Server host on which a web server depends. We are not always aware of the underlying dependencies between hosts, so the host baseline can be used to determine these. From there, you can determine whether a host is not functioning properly due to a malfunctioning or high-latency dependency.

Application Baseline

The final network baseline category is the application baseline. This baseline should be performed on all business-critical network-based applications.

The following are the components of the application baseline:

Protocols in Use

Again, for this baseline, use the Protocol Hierarchy Statistics window in Wireshark, this time while capturing traffic from the host running the application. Later, you can compare against this list to find out whether protocols that the application depends on are functioning incorrectly or not at all.

Startup/Shutdown

This baseline includes a capture of the traffic generated during the startup and shutdown sequences of the application. If the application refuses to start or is abnormally slow during either sequence, you can use this baseline to determine the cause.

Associations/Dependencies

This baseline requires a longer-duration capture in which the Conversations window can be used to determine on which other hosts and applications this application depends. We are not always aware of the underlying dependencies between applications, so this baseline can be used to determine those. From there, you can determine whether an application is not functioning properly due to a malfunctioning or high-latency dependency.

Data Transfer Rate

You can use the capture summary and graphing features of Wireshark to determine the transfer rate and consistency of the connections to the application server during its normal operation. Whenever the application is reported as being slow, you can use this baseline to determine whether the issues being experienced are a result of high utilization or high user load.

Additional Notes on Baselines

Here are a few more points to keep in mind when creating your network baseline:

- When creating your baselines, capture each one at least three times: once during a low-traffic time (early morning), once during a high-traffic time (midafternoon), and once during a no-traffic time (late night).
- When possible, avoid capturing directly from the hosts you are baselining. During periods of high traffic, doing so may put an increased load on the device, hurt its performance, and cause your baseline to be invalid due to dropped packets.
- Your baseline will contain some very intimate information about your network, so be sure to secure it. Store it in a safe place where only the appropriate individuals have access. But at the same time, keep it readily accessible so you can use it when needed. Consider keeping it on a USB flash drive or on an encrypted partition.
- Keep all *.pcap* and *.pcapng* files associated with your baseline and create a cheat sheet of the more commonly referenced values, such as associations or average data transfer rates.

Final Thoughts

This chapter has focused on troubleshooting slow networks. We've covered some of the more useful reliability detection and recovery features of TCP, demonstrated how to locate the source of high latency in network communications, and discussed the importance of a network baseline and some of its components. Using the techniques discussed here, along with some of Wireshark's graphing and analysis features, you should be well equipped to troubleshoot when you get that call complaining that the network is slow.

12

PACKET ANALYSIS FOR SECURITY

Although most of this book focuses on using packet analysis for network troubleshooting, a considerable amount of real-world packet analysis is done for security purposes. For example, an intrusion analyst might review network traffic from potential intruders, or a forensic investigator might attempt to ascertain the extent of a malware infection on a compromised host.

Performing packet analysis while investigating security incidents is always a challenging scenario because it involves the unknown element of an attacker-controlled device. You can't walk over to the attacker's cubicle to ask a question or baseline their normal traffic; all you have to work with is the interaction you can capture between their system and yours. Fortunately, for an attacker to breach one of your systems remotely, they have to interact with the network in some form. Of course, they know that too, so they aren't lacking in tricks to obfuscate their techniques.

In this chapter, we'll take the viewpoint of a security practitioner as we examine different aspects of a system compromise at the network level. We'll cover network reconnaissance, malicious traffic redirection, and common malware techniques. In some cases, we'll take on the role of intrusion analyst as we dissect traffic based on alerts from an intrusion-detection system (IDS). Reading this chapter will provide you with insight into network security that may prove critical, even if you are not presently in a security-focused role.

Reconnaissance

An attacker's first step is often to perform in-depth research on the target system. This step, commonly referred to as *footprinting*, is frequently accomplished using various publicly available resources, such as the target company's website or Google. Once this research is completed, the attacker will typically begin scanning the IP address (or DNS name) of their target for open ports or running services.

Scanning allows the attacker to determine whether the target is alive and reachable. For example, consider a scenario in which bank robbers are planning to steal from the largest bank in the city, located at 123 Main Street. They spend weeks planning an elaborate heist, only to find out upon arrival at the address that the bank has moved to 555 Vine Street. Worse yet, imagine that the robbers plan to walk into the bank during normal business hours, intending to steal from the vault, only to get to the bank and discover it's closed that day. Whether robbing a bank or attacking a network, ensuring that the target is alive and accessible is the first hurdle.

Scanning also tells the attacker on which ports the target is listening. Returning to our bank robbers analogy, consider what would happen if the robbers showed up at the bank with absolutely no knowledge of the building's physical layout. They would have no idea how to gain access to the vault because they wouldn't know the weak points in the bank's physical security.

In this section, we'll discuss a few of the more common scanning techniques used to identify hosts, their open ports, and vulnerabilities on a network.

NOTE *So far, this book has referred to the sides of a connection as the* transmitter *and* receiver *or as the* client *and* server. *This chapter refers to each side of the communication as either the* attacker *or the* target.

SYN Scan

synscan.pcapng

The type of scanning often done first against a system is a *TCP SYN scan*, also known as a *stealth scan* or a *half-open scan*. A SYN scan is the most common type for several reasons:

- It is very fast and reliable.
- It is accurate on all platforms, regardless of TCP stack implementation.
- It is less noisy than other scanning techniques.

The TCP SYN scan relies on the three-way handshake process to determine which ports are open on a target host. The attacker sends a TCP SYN packet to a range of ports on the target, as if trying to establish a channel for normal communication on the ports. Once this packet is received by the target, one of several things may happen, as shown in Figure 12-1.

Figure 12-1: Possible results of a TCP SYN scan

If a service on the target's machine is listening on a port that receives the SYN packet, it will reply to the attacker with a TCP SYN/ACK packet, the second part of the TCP handshake. Now the attacker knows that port is open and a service is listening on it. Under normal circumstances, a final TCP ACK would be sent to complete the connection handshake. In this case, however, the attacker doesn't want that to happen since they won't be communicating with the host further at this point, so the attacker doesn't attempt to complete the TCP handshake.

If no service is listening on a scanned port, the attacker will not receive a SYN/ACK. Depending on the configuration of the target's operating system, the attacker could receive an RST packet in return, indicating that the port is closed. Alternatively, the attacker may receive no response at all. No response could mean that the port is filtered by an intermediate device, such as a firewall or the host itself. On the other hand, it could just be that the response was lost in transit. Thus, while this result typically indicates that the port is closed, it is ultimately inconclusive.

The file *synscan.pcapng* provides a great example of a SYN scan performed with the Nmap tool. Nmap is a robust network-scanning application developed by Gordon "Fyodor" Lyon. It can perform just about any kind of scan you can imagine. You can download Nmap for free from *http://www.nmap.com/download.html*.

Our sample capture contains roughly 2,000 packets, telling us that this scan is of a reasonable size. One of the best ways to ascertain the scope of a scan of this nature is to view the Conversations window, as shown in Figure 12-2. There, you should see only one IPv4 conversation ❶ between the attacker (172.16.0.8) and the target (64.13.134.52). You will also see that there are 1,994 TCP conversations between these two hosts ❷—basically a new conversation for every port pairing involved in the communications.

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
64.13.134.52	65000	172.16.0.8	36050	1	58	0	0	1	58	1.690937000	0.000000	N/A	N/A
64.13.134.52	65000	172.16.0.8	36051	1	58	0	0	1	58	1.820296000	0.000000	N/A	N/A
64.13.134.52	52848	172.16.0.8	36050	1	58	0	0	1	58	1.821670000	0.000000	N/A	N/A
64.13.134.52	52848	172.16.0.8	36051	1	58	0	0	1	58	1.945599000	0.000000	N/A	N/A
64.13.134.52	61532	172.16.0.8	36050	1	58	0	0	1	58	1.946839000	0.000000	N/A	N/A
64.13.134.52	49155	172.16.0.8	36050	1	58	0	0	1	58	2.006228000	0.000000	N/A	N/A
64.13.134.52	61532	172.16.0.8	36051	1	58	0	0	1	58	2.070465000	0.000000	N/A	N/A
64.13.134.52	49155	172.16.0.8	36051	1	58	0	0	1	58	2.130300000	0.000000	N/A	N/A
64.13.134.52	44176	172.16.0.8	36050	1	58	0	0	1	58	2.196298000	0.000000	N/A	N/A
64.13.134.52	49999	172.16.0.8	36050	1	58	0	0	1	58	2.196434000	0.000000	N/A	N/A
64.13.134.52	40911	172.16.0.8	36050	1	58	0	0	1	58	2.255375000	0.000000	N/A	N/A

Figure 12-2: The Conversations window shows the variety of TCP communications taking place.

The scanning is occurring very quickly, so scrolling through the capture file isn't the best way to find the response associated with each initial SYN packet. Several more packets might be sent before a response to the original packet is received. Fortunately, we can create filters to help us find the right traffic.

Using Filters with SYN Scans

As an example of filtering, let's consider the first packet in the capture, which is a SYN packet sent to the target on port 443 (HTTPS). To see whether there was a response to this packet, we can create a filter to show all traffic to and from port 443. Here's how to do this quickly:

1. Select the first packet in the capture file.
2. Expand the TCP header in the Packet Details pane.
3. Right-click the **Destination Port** field, select **Prepare as Filter**, and click **Selected**.
4. This will place a filter in the filter dialog for all packets with the destination port of 443. Now, because we also want all packets from the source port of 443, click in the filter dialog at the top of the screen and erase the *dst* portion of the filter.

The resulting filter will yield two packets, which are both TCP SYN packets sent from attacker to target, as shown in Figure 12-3.

No.	Time	Source	Destination	Protocol	Info
1	0.000000	172.16.0.8	64.13.134.52	TCP	36050 → 443 [SYN] Seq=3713172248 Win=3072 Len=0 MSS=1460
32	0.000065	172.16.0.8	64.13.134.52	TCP	36051 → 443 [SYN] Seq=3713237785 Win=2048 Len=0 MSS=1460

Figure 12-3: Two attempts to establish a connection with SYN packets

NOTE *In this section, packets are shown using the time display format Seconds Since Previous Displayed Packet.*

Since there is no response to either of these packets, it's possible that the response is being filtered by the target host or an intermediary device or that the port is closed. Ultimately, the result of the scan against port 443 is inconclusive.

We can attempt this same technique on another packet to see whether we get different results. To do so, clear your previous filter and select packet 9 in the list. This is a SYN packet to port 53, commonly associated with DNS. Using the method outlined in the previous steps or by modifying your last filter, create a filter that will show all TCP port 53 traffic. When you apply this filter, you should see five packets, as shown in Figure 12-4.

No.	Time	Source	Destination	Protocol	Info
9	0.000052	172.16.0.8	64.13.134.52	TCP	36050 → 53 [SYN] Seq=3713172248 Win=3072 Len=0 MSS=1460
11	0.061832	64.13.134.52	172.16.0.8	TCP	53 → 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 Win=5840 Len=0 MSS=1380
529	0.057126	64.13.134.52	172.16.0.8	TCP	[TCP Retransmission] 53 → 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 Win=5840 Len=0 MSS=1380
2006	3.930109	64.13.134.52	172.16.0.8	TCP	[TCP Retransmission] 53 → 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 Win=5840 Len=0 MSS=1380
2009	10.029025	64.13.134.52	172.16.0.8	TCP	[TCP Retransmission] 53 → 36050 [SYN, ACK] Seq=1117405124 Ack=3713172249 Win=5840 Len=0 MSS=1380

Figure 12-4: Five packets indicating a port is open

The first of these packets is the SYN we selected at the beginning of the capture (packet 9). The second is a response from the target. It's a TCP SYN/ACK—the response expected when setting up the three-way handshake. Under normal circumstances, the next packet would be an ACK from the host that sent the initial SYN. However, in this case, our attacker doesn't want to complete the connection and doesn't send a response. As a result, the target retransmits the SYN/ACK three more times before giving up. Since a SYN/ACK response is received when attempting to communicate with the host on port 53, it's safe to assume that a service is listening on that port.

Let's rinse and repeat this process one more time for packet 13. This is a SYN packet sent to port 113, which is commonly associated with the Ident protocol, often used for IRC identification and authentication services. If you apply the same type of filter to the port listed in this packet, you will see four packets, as shown in Figure 12-5.

No.	Time	Source	Destination	Protocol	Info
13	0.000070	172.16.0.8	64.13.134.52	TCP	36050 → 113 [SYN] Seq=3713172248 Win=4096 Len=0 MSS=1460
14	0.061491	64.13.134.52	172.16.0.8	TCP	113 → 36050 [RST, ACK] Seq=2462244745 Ack=3713172249 Win=0 Len=0
530	0.006942	172.16.0.8	64.13.134.52	TCP	36061 → 113 [SYN] Seq=3696394776 Win=2048 Len=0 MSS=1460
571	0.000827	64.13.134.52	172.16.0.8	TCP	113 → 36061 [RST, ACK] Seq=1027049353 Ack=3696394777 Win=0 Len=0

Figure 12-5: A SYN followed by an RST, indicating the port is closed

The first packet is the initial SYN, which is followed immediately by an RST from the target. This is an indication that the target is not accepting connections on the targeted port and that a service is most likely not running on it.

Identifying Open and Closed Ports

Now that you understand the different types of responses a SYN scan can elicit, you'll want to find a fast method of identifying which ports are open or closed. The answer lies within the Conversations window once again. In this window, you can sort the TCP conversations by packet number, with the highest values at the top, by clicking the Packets column header until the arrow points downward, as shown in Figure 12-6.

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits
172.16.0.8	36050	64.13.134.52	53	5	298	1	58	4	240	0.001913000	21.091302	21	
172.16.0.8	36050	64.13.134.52	80	❶ 5	298	1	58	4	240	0.129000000	21.272180	21	
172.16.0.8	36050	64.13.134.52	22	5	298	1	58	4	240	1.403484000	21.681859	21	
172.16.0.8	36050	64.13.134.52	113	2	118	1	58	1	60	0.065341000	0.061491	7545	
172.16.0.8	36050	64.13.134.52	25	2	118	1	58	1	60	1.403154000	0.062745	7395	
172.16.0.8	36050	64.13.134.52	31337	❷ 2	118	1	58	1	60	1.756214000	0.062293	7448	
172.16.0.8	36061	64.13.134.52	113	2	118	1	58	1	60	3.070317000	0.061814	7506	
172.16.0.8	36050	64.13.134.52	70	2	118	1	58	1	60	4.016755000	0.061231	7577	
64.13.134.52	65000	172.16.0.8	36050	1	58	0	0	1	58	1.690937000	0.000000	N/A	
64.13.134.52	65000	172.16.0.8	36051	1	58	0	0	1	58	1.820296000	0.000000	N/A	
64.13.134.52	52848	172.16.0.8	36050	1	58	0	0	1	58	1.821670000	0.000000	N/A	

Figure 12-6: Finding open ports with the Conversations window

Three scanned ports include five packets in each of their conversations ❶. We know that ports 53, 80, and 22 are open, because these five packets represent the initial SYN, the corresponding SYN/ACK, and the retransmitted SYN/ACKs from the target.

For five ports, only two packets were involved in the communication ❷. The first is the initial SYN, and the second is the RST from the target. These results indicate that ports 113, 25, 31337, 113, and 70 are closed.

The remaining entries in the Conversations window include only one packet, meaning that the target host never responded to the initial SYN. These remaining ports are most likely closed, but we're not sure.

This technique of counting packets worked for this host, but it won't be consistent for all hosts you might scan, so you shouldn't rely on it exclusively. Instead, focus on learning what normal stimulus and response looks like and what abnormal responses to normal stimuli can mean.

Operating System Fingerprinting

An attacker puts a great deal of value on knowing the target's operating system. Knowledge of the operating system helps the attacker configure all their methods of attack correctly for that system. It also allows the attacker to know the location of certain critical files and directories within the target file system, should they succeed in accessing the system.

Operating system fingerprinting is the name given to a group of techniques used to determine the operating system running on a system without having physical access to that system. There are two types of operating system fingerprinting: passive and active.

Passive Fingerprinting

passiveosfingerprinting.pcapng

Using *passive fingerprinting*, you examine certain fields within packets sent from the target to determine the operating system in use. The technique is considered passive because you listen to only the packets the target host is sending and don't actively send any packets to the host yourself. This type of operating system fingerprinting is ideal for attackers because it allows them to be stealthy.

That said, how can we determine which operating system a host is running based on nothing but the packets it sends? This feat is possible due to the lack of standardized values in the specifications defined by protocol RFCs. Although the various fields contained in TCP, UDP, and IP headers are very specific, default values are typically not defined for every field. This means that the TCP/IP stack implementation in each operating system must define its own default values for these fields. Table 12-1 lists some of the more common fields and the default values that can be used to link them to various operating systems. Keep in mind that these values are subject to change with new OS version releases.

Table 12-1: Common Passive Fingerprinting Values

Protocol header	Field	Default value	Platform
IP	Initial time to live	64	NMap, BSD, OS X, Linux
		128	Novell, Windows
		255	Cisco IOS, Palm OS, Solaris
IP	Don't fragment flag	Set	BSD, OS X, Linux, Novell, Windows, Palm OS, Solaris
		Not set	Nmap, Cisco IOS
TCP	Maximum segment size	0	Nmap
		1440–1460	Windows, Novell
		1460	BSD, OS X, Linux, Solaris

(continued)

Table 12-1 *(continued)*

Protocol header	Field	Default value	Platform
TCP	Window size	1024–4096	Nmap
		65535	BSD, OS X
		Variable	Linux
		16384	Novell
		4128	Cisco IOS
		24820	Solaris
		Variable	Windows
TCP	SackOK	Set	Linux, Windows, OS X, OpenBSD
		Not set	Nmap, FreeBSD, Novell, Cisco IOS, Solaris

The packets contained in the file *passiveosfingerprinting.pcapng* are great examples of this technique. There are two packets in this file. Both are TCP SYN packets sent to port 80, but they come from different hosts. Using only the values contained in these packets and referring to Table 12-1, we should be able to determine the operating system architecture in use on each host. The details of each packet are shown in Figure 12-7.

Using Table 12-1 as a reference, we can create Table 12-2, which is a breakdown of the relevant fields in these packets.

Table 12-2: Breakdown of the Operating System Fingerprinting Packets

Protocol header	Field	Packet 1 value	Packet 2 value
IP	Initial time to live	128	64
IP	Don't fragment flag	Set	Set
TCP	Maximum segment size	1,440 bytes	1,460 bytes
TCP	Window size	64,240 bytes	2,920 bytes
TCP	SackOK	Set	Set

Based on these values, we can conclude that packet 1 was most likely sent by a device running Windows and packet 2 was most likely sent by a device running Linux.

Keep in mind that the list of common passive fingerprinting identifying fields in Table 12-1 is by no means exhaustive. There are many quirks that may result in deviations from these expected values. Therefore, you cannot fully rely on the results gained from passive operating system fingerprinting.

```
Wireshark · Packet 1 · passiveosfingerprinting
> Frame 1: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface 0
> Ethernet II, Src: Vmware_f9:74:d8 (00:0c:29:f9:74:d8), Dst: D-Link_21:99:4c (00:05:5d:21:99:4c)
v Internet Protocol Version 4, Src: 172.16.16.134, Dst: 168.143.162.100
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 48
    Identification: 0x4d80 (19840)
  > Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 128
    Protocol: TCP (6)
  > Header checksum: 0xa5bd [validation disabled]
    Source: 172.16.16.134
    Destination: 168.143.162.100
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
v Transmission Control Protocol, Src Port: 1176 (1176), Dst Port: 80 (80), Seq: 2123482830, Len: 0
    Source Port: 1176
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 2123482830
    Acknowledgment number: 0
    Header Length: 28 bytes
  > Flags: 0x002 (SYN)
    Window size value: 64240
    [Calculated window size: 64240]
  > Checksum: 0x3670 [validation disabled]
    Urgent pointer: 0
  v Options: (8 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted
    > Maximum segment size: 1440 bytes
    > No-Operation (NOP)
    > No-Operation (NOP)
    > TCP SACK Permitted Option: True
No.: 1 · Time: 0.000000 · Source: 172.16.16.134 · Destination: 168.143.162.100 · ...h: 62 · Info: 1176 → 80 [SYN] Seq=2123482830 Win=64240 Len=0 MSS=1440 SACK_PER
                                                                                    Close    Help
```

```
Wireshark · Packet 2 · passiveosfingerprinting
> Frame 2: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface 0
> Ethernet II, Src: Vmware_f9:74:d8 (00:0c:29:f9:74:d8), Dst: D-Link_21:99:4c (00:05:5d:21:99:4c)
v Internet Protocol Version 4, Src: 172.16.16.134, Dst: 168.143.162.100
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 48
    Identification: 0x4d80 (19840)
  > Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
  > Header checksum: 0xe5bd [validation disabled]
    Source: 172.16.16.134
    Destination: 168.143.162.100
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
v Transmission Control Protocol, Src Port: 1176 (1176), Dst Port: 80 (80), Seq: 2123482830, Len: 0
    Source Port: 1176
    Destination Port: 80
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 2123482830
    Acknowledgment number: 0
    Header Length: 28 bytes
  > Flags: 0x002 (SYN)
    Window size value: 2920
    [Calculated window size: 2920]
  > Checksum: 0x25e5 [validation disabled]
    Urgent pointer: 0
  v Options: (8 bytes), Maximum segment size, No-Operation (NOP), No-Operation (NOP), SACK permitted
    > Maximum segment size: 1460 bytes
    > No-Operation (NOP)
    > No-Operation (NOP)
    > TCP SACK Permitted Option: True
  > [SEQ/ACK analysis]
No.: 2 · Time: 0.000108 · Source: 172.16.16.134 · Destination: 168.143.162.100 ·...Out-Of-Order] 1176 → 80 [SYN] Seq=2123482830 Win=2920 Len=0 MSS=1460 SACK_PE
                                                                                    Close    Help
```

Figure 12-7: These packets can tell us which operating system they were sent from.

NOTE *In many cases, attackers rely on automated tools to passively identify the operating system of a target. One tool that uses operating system fingerprinting techniques is p0f. This tool analyzes relevant fields in a packet capture and outputs the suspected operating system. Using tools like p0f, you can get not only the operating system architecture but sometimes even the version or patch level of the OS. You can download p0f from* http://lcamtuf.coredump.cx/p0f.shtml.

Active Fingerprinting

activeosfingerprinting .pcapng

When passively monitoring traffic doesn't yield the desired results, a more direct approach—*active fingerprinting*—may be required. Now the attacker actively sends specially crafted packets to the target to elicit replies that will reveal the operating system on the target's machine. Of course, since this approach involves communicating directly with the target, it is not the least bit stealthy, but it can be highly effective.

The file *activeosfingerprinting.pcapng* contains an example of an active operating system fingerprinting scan initiated with the Nmap scanning utility. Several packets in this file are the result of Nmap's sending different probes designed to elicit responses that will allow for operating system identification. Nmap records the responses to these probes and builds a fingerprint, which it compares to a database of values to make a determination.

NOTE *The techniques used by Nmap to actively fingerprint an operating system are quite complex. To learn more about how Nmap performs active operating system fingerprinting, read the definitive guide to Nmap,* Nmap Network Scanning *(2008), by the tool's author, Gordon "Fyodor" Lyon.*

Traffic Manipulation

One of the key points I've tried to show throughout this book is that you can learn a lot about a system or its users by examining the right packets. Thus, it should come as no surprise that attackers often seek to capture these packets themselves. By examining the packets generated by a system, an attacker can learn about the operating system, the applications in use, authentication credentials, and much more.

In this section, we'll examine two techniques at the packet level: how an attacker can use ARP cache poisoning to intercept and capture target traffic and how they can intercept HTTP cookies to perform session-hijacking attacks.

ARP Cache Poisoning

arppoison.pcapng

In Chapter 7, we discussed how the ARP protocol is used to allow devices to map IP addresses to MAC addresses inside of a network, and, in Chapter 2, we discussed how ARP cache poisoning can be a useful technique for tapping into the wire and intercepting traffic from hosts whose packets you need to analyze. When used for legitimate purposes, ARP cache poisoning is very helpful for troubleshooting. However, when this technique is used with malicious intent, it is a lethal form of the *man-in-the-middle (MITM) attack*.

In a MITM attack, an attacker redirects traffic between two hosts in order to intercept or modify data in transit. There are many forms of MITM attacks, including DNS spoofing and SSL hijacking. In ARP cache poisoning, specially crafted ARP packets trick two hosts into thinking they are communicating with each other when, in fact, they are communicating with a third party who is relaying packets as an intermediary. In this way, the illegitimate use of a protocol's normal functionality can be used for malicious purposes.

The file *arppoison.pcapng* contains an example of ARP cache poisoning. When you open it, you'll see that this traffic appears normal at first glance. However, if you follow the packets, you'll see our target, 172.16.0.107, browsing to Google and performing a search. As a result of this search, there is quite a bit of HTTP traffic with some DNS queries mixed in.

We know that ARP cache poisoning is a technique that occurs at layer 2, so if we just casually peruse the packets in the Packet List pane, it may be hard to see any foul play. To give us a leg up, we'll add a couple of columns to the Packet List pane, as follows:

1. Select **Edit ▸ Preferences**.
2. Click **Columns** on the left side of the Preferences window.
3. Click the plus (+) button to add a new column.
4. In the Title area, type `Source MAC` and press ENTER.
5. In the Type drop-down list, select **Hw src addr (resolved)**.
6. Click the newly added entry and drag it so that it is directly after the Source column.
7. Click the plus (+) button to add a new column.
8. In the Title area, type `Dest MAC` and press ENTER.
9. In the Type drop-down list, select **Hw dest addr (resolved)**.
10. Click the newly added entry and drag it so that it is directly after the Destination column.
11. Click **OK** to apply the changes.

When you have completed these steps, your screen should look like Figure 12-8. You should now have two additional columns showing the source and destination MAC addresses of the packets.

Figure 12-8: The column configuration screen with newly added columns for source and destination hardware addresses

If you still have MAC name resolution turned on, you should see that the communicating devices have MAC addresses that indicate Dell and Cisco hardware. This is very important to remember because as we scroll through the capture, we'll see that this changes at packet 54, where we see some peculiar ARP traffic occurring between the Dell host (our target) and a newly introduced HP host (the attacker), as shown in Figure 12-9.

No.	Time	Source	Source MAC	Destination	Dest MAC	Protocol	Info
54	4.171500	HewlettP_bf:91:ee	HewlettP_bf:91:ee	Dell_c0:56:f0 ❶	Dell_c0:56:f0	ARP	Who has 172.16.0.107? Tell 172.16.0.1
❷ 55	0.000053	Dell_c0:56:f0	Dell_c0:56:f0	HewlettP_bf:91:ee	HewlettP_bf:91:ee	ARP	172.16.0.107 is at 00:21:70:c0:56:f0
56	0.000013	HewlettP_bf:91:ee	HewlettP_bf:91:ee	Dell_c0:56:f0	Dell_c0:56:f0	ARP	❸ 172.16.0.1 is at 00:25:b3:bf:91:ee

Figure 12-9: Strange ARP traffic between the Dell device and an HP device

Before proceeding further, note the endpoints involved in this communication, which are listed in Table 12-3.

Table 12-3: Endpoints Being Monitored

Role	Device type	IP address	MAC address
Target	Dell	172.16.0.107	00:21:70:c0:56:f0
Router	Cisco	172.16.0.1	00:26:0b:31:07:33
Attacker	HP	Unknown	00:25:b3:bf:91:ee

But what makes this traffic strange? Recall from our discussion of ARP in Chapter 7 that there are two primary types of ARP packets: a request and a response. The request packet is sent as a broadcast to all hosts on the network in order to find the machine that has the MAC address associated with a particular IP address. Then the machine that replies to the requesting device sends a response as a unicast packet. Given this background, we can identify a few peculiar things in this communication sequence, referring to Figure 12-9.

First, packet 54 is an ARP request sent from the attacker (MAC address 00:25:b3:bf:91:ee) as a unicast packet directly to the target (MAC address 00:21:70:c0:56:f0) ❶. This type of request should be broadcast to all hosts on the network, but this one singles out the target. Also, notice that although this packet is sent from the attacker and includes the attacker's MAC address in the ARP header, it lists the router's IP address rather than its own.

This packet is followed by a response from the target to the attacker containing its MAC address information ❷. The real voodoo here occurs in packet 56, in which the attacker sends a packet to the target with an unsolicited ARP reply telling it that 172.16.0.1 is located at its MAC address, 00:25:b3:bf:91:ee ❸. The problem is that MAC address 172.16.0.1 isn't 00:25:b3:bf:91:ee but is 00:26:0b:31:07:33. We know this because we saw the router at 172.16.0.1 communicating with the target earlier in the packet capture. Since the ARP protocol is inherently insecure (it accepts unsolicited updates to its ARP table), the target will now be sending traffic that should be going to the router to the attacker instead.

NOTE *Because this packet capture was taken from the target's machine, you don't actually see the entire picture. For this attack to work, the attacker must send the same sequence of packets to the router in order to trick it into thinking the attacker is actually the target, but we would need to take another packet capture from the router (or the attacker) to see those packets.*

Once both target and router have been duped, the communication between them flows through the attacker, as illustrated in Figure 12-10.

Figure 12-10: ARP cache poisoning as an MITM attack

Packet 57 confirms the success of this attack. When you compare this packet to one sent before the mysterious ARP traffic, such as packet 40 (see Figure 12-11), you will see that the IP address of the remote server (Google) remains the same ❷ but the target MAC address has changed ❶. This change in MAC address tells us that the traffic is now being routed through the attacker before it gets to the router.

Because this attack is so subtle, it's very difficult to detect. To find it, you typically need the aid of an IDS configured specifically to address it or software running on devices designed to detect sudden changes in ARP table entries. Since you'll most likely use ARP cache poisoning to capture packets on networks you are analyzing, it's important to know how this technique can be used against you as well.

```
Wireshark · Packet 40 · arppoison
> Frame 40: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface 0
v Ethernet II, Src: Dell_c0:56:f0 (00:21:70:c0:56:f0), Dst: CiscoInc_31:07:33 (00:26:0b:31:07:33)
  > Destination: CiscoInc_31:07:33 (00:26:0b:31:07:33) ❶
  > Source: Dell_c0:56:f0 (00:21:70:c0:56:f0)
    Type: IPv4 (0x0800)
v Internet Protocol Version 4, Src: 172.16.0.107, Dst: 74.125.95.147
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0x97bf (38847)
  > Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
  > Header checksum: 0x4c79 [validation disabled]
    Source: 172.16.0.107
    Destination: 74.125.95.147 ❷
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
> Transmission Control Protocol, Src Port: 45692 (45692), Dst Port: 80 (80), Seq: 619079507, Ack: 2814360513, Len: 0
No.: 40 · Time: 0.000013 · Source: 172.16.0.107 · Source MAC: Dell_c0:56:f0 · Destination: 74.125.95.147 · ...col: TCP · Info: 45692 → 80 [ACK] Seq=619079507 Ack=2814360513 Win=6912 Len=0 TSval=586661 TSecr=321.
                                                                                                   Close   Help

Wireshark · Packet 57 · arppoison
> Frame 57: 960 bytes on wire (7680 bits), 960 bytes captured (7680 bits) on interface 0
v Ethernet II, Src: Dell_c0:56:f0 (00:21:70:c0:56:f0), Dst: HewlettP_bf:91:ee (00:25:b3:bf:91:ee)
  > Destination: HewlettP_bf:91:ee (00:25:b3:bf:91:ee) ❶
  > Source: Dell_c0:56:f0 (00:21:70:c0:56:f0)
    Type: IPv4 (0x0800)
v Internet Protocol Version 4, Src: 172.16.0.107, Dst: 74.125.95.147
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 946
    Identification: 0x3af1 (15089)
  > Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (6)
  > Header checksum: 0xa5c9 [validation disabled]
    Source: 172.16.0.107
    Destination: 74.125.95.147 ❷
    [Source GeoIP: Unknown]
    [Destination GeoIP: Unknown]
> Transmission Control Protocol, Src Port: 45691 (45691), Dst Port: 80 (80), Seq: 609865591, Ack: 1647620296, Len: 894
> Hypertext Transfer Protocol
No.: 57 · Time: 1.906795 · Source: 172.16.0.107 · Source MAC: Dell_c0:56:f0 · Destination: 74.125.95.14...qi=&aql=&oq=U&gs_rfai=&fp=57d9c86769d1bf04&tch=1&ech=1&psi=dNQ_TOejlY_6M_Sa8JwH12792515729190 HTT
                                                                                                   Close   Help
```

Figure 12-11: The change in target MAC address shows this attack was a success.

Session Hijacking

sessionhijacking.pcapng

Now that you know how ARP cache poisoning can be used maliciously, I want to demonstrate a technique that can take advantage of it: *session hijacking*. In session hijacking, an attacker compromises an HTTP session cookie, which we'll learn about soon, and uses it to impersonate another user. To accomplish this, an attacker can use ARP cache poisoning to intercept a target's traffic and find relevant session cookie information. The attacker can then use that information to access the target web application as the target user.

This scenario begins with the file *sessionhijacking.pcapng*. This capture contains the traffic of a target (172.16.16.164) communicating with a web application (172.16.16.181). Unbeknownst to the target, they have fallen prey to an attacker (172.16.16.154) who is actively intercepting their communications. These packets were collected from the perspective of the web server, which is likely the same viewpoint a defender would have if a session-hijacking attack were used against their server infrastructure.

NOTE *The web application being accessed here is called Damn Vulnerable Web Application (DVWA). It is intentionally vulnerable to many types of attacks and is used frequently as a teaching tool. If you'd like to learn more about web application attacks or investigate packets associated with them, you can learn more about DVWA at* http://www.dvwa.co.uk/.

The traffic in this capture consists primarily of two conversations. The first is the communication from the target to web server, which can be isolated with the filter **`ip.addr == 172.16.16.164 && ip.addr == 172.16.16.181`**. This communication represents normal web-browsing traffic and isn't particularly special. Of particular interest is the cookie value in the requests. For instance, if you look at a `GET` request such as the one in packet 14, you will find the cookie listed in the Packet Details window, as shown in Figure 12-12. In this case, the cookie identifies the session ID with a PHPSESSID value of `ncobrqrb7fj2a2sinddtk567q4` ❶.

Figure 12-12: Viewing the target's session cookie

Websites use cookies to maintain session awareness for individual hosts. When a new visitor comes to a website, they are issued a session ID that uniquely identifies them (the PHPSESSID). For authentication, many applications wait until a user with a session ID has successfully authenticated

to the app, and then they create a database record recognizing that ID as being representative of an authenticated session. Any user with that ID will be able to access the app with that authentication. Of course, developers want to believe that only a single user would have a specific ID because the IDs are uniquely generated. This method of handling session IDs is insecure, however, because it allows a malicious user to steal another user's ID and use it to impersonate them. There are methods that can be used to prevent session-hijacking techniques, but many websites, including DVWA, are still vulnerable.

The target doesn't realize that their traffic is being intercepted by an attacker or that the attacker has access to the session cookie, as shown in Figure 12-12. All the attacker has to do is communicate with the web server using that cookie value. This task can be accomplished with certain types of proxy servers, but it is made even easier by using browser plugins like Cookie Manager for Chrome. Using this plugin, the attacker can specify the PHPSESSID value obtained from the target's traffic, as shown in Figure 12-13.

Figure 12-13: Using the Cookie Manager plugin to impersonate the target

If you clear the filter previously applied to the capture file and start scrolling down, eventually you'll see the attacker's IP address communicating with the web server. You can limit your view to this communication using the filter **`ip.addr == 172.16.16.154 && ip.addr == 172.16.16.181`**.

Before we dig into this further, let's add a column to show the cookie values in the Packet List pane. If you added columns as part of the previous section on ARP cache poisoning, you should remove those first. Then proceed to use the instructions from the ARP cache-poisoning section to add the new custom column field based on the field name **`http.cookie_pair`**. Once you've added the column, position it after the Destination field. Your screen should look like Figure 12-14.

Figure 12-14: Configuring columns to investigate session hijacking

With the new columns configured, modify the display filter to show only HTTP requests, as TCP communication isn't useful here. The new filter is **`(ip.addr==172.16.16.154 && ip.addr==172.16.16.181) && (http.request.method || http.response.code)`**. The resulting packets are shown in Figure 12-15.

No.	Time	Source	Destination	Cookie pair	Protocol	Info
77	16.563004	172.16.16.154	172.16.16.181	security=low,PHPSESSID=lup70ajeuodkrhrvbmsjtgrd71	HTTP	❶ GET /dvwa/ HTTP/1.1
79	16.565584	172.16.16.181	172.16.16.154		HTTP	HTTP/1.1 302 Found❷
80	16.570187	172.16.16.154	172.16.16.181	security=low,PHPSESSID=lup70ajeuodkrhrvbmsjtgrd71	HTTP	❸ GET /dvwa/login.php HTTP/1.1
81	16.575123	172.16.16.181	172.16.16.154		HTTP	HTTP/1.1 200 OK (text/html)❹
115	60.040166	172.16.16.154	172.16.16.181	security=low,PHPSESSID=ncobrqrb7fj2a2sinddtk567q4	HTTP	❺ GET /dvwa/ HTTP/1.1
118	60.042241	172.16.16.181	172.16.16.154		HTTP	HTTP/1.1 200 OK (text/html)❻
120	64.292056	172.16.16.154	172.16.16.181	security=low,PHPSESSID=ncobrqrb7fj2a2sinddtk567q4	HTTP	❼ GET /dvwa/setup.php HTTP/1.1
122	64.293401	172.16.16.181	172.16.16.154		HTTP	HTTP/1.1 200 OK (text/html)❽

Figure 12-15: The attacker impersonating the target user

You are now looking at communication between the attacker and the server. In the first four packets, the attacker requests the */dvwa/* directory ❶ and receives a 302 response code in return, which is a normal method web servers use to redirect visitors to different URLs on a server. In this case, the attacker gets redirected to the login page at */dvwa/login.php* ❷. The attacker's machine requests the login page ❸, which is returned successfully ❹. Both requests use the session ID *lup70ajeuodkrhrvbmsjtgrd71*.

Following that, there is a new request for the */dvwa/* directory, but this time, take note of the different session ID ❺. The session ID is now *ncobrqrb7fj2a2sinddtk567q4*, which is the same one the target used earlier. This indicates the attacker has manipulated the traffic to use the stolen ID. Instead of being redirected to the login page, the request is met with an HTTP 200 status code, and the page is delivered as the authenticated target would see it ❻. The attacker browses to another page, *dvwa/setup.php*, using the target's ID ❼, and that page also returns successfully ❽. The attacker is browsing the DVWA website as though they were authenticated as the target. This is all without knowing the target's username or password.

This is just one example of how an attacker can turn packet analysis into an offensive tool. In general, it's safe to assume that if an attacker can see the packets associated with your communication, some type of malicious activity can result. This is one reason security professionals advocate for protecting data in transit through encryption.

Malware

While perfectly legitimate software can be used for malicious purposes, *malware* is a term usually reserved for code that is written specifically with malicious intent. Malware can take many shapes and forms, including worms that are self-propagating and trojan horses that masquerade as legitimate software. From a network defender's view, most malware is undiscovered and unknown until it can be captured and analyzed. This analysis involves multiple steps, including one focused on a behavioral analysis of the malware's network communication patterns. In some cases, analysis occurs in a forensic malware reverse-engineering lab, but more often, it occurs in the wild when a security analyst discovers a device on their network that has become infected.

In this section, we will look at a few examples of real malware and its behavior, as observed through packets.

Operation Aurora

aurora.pcapng

In January 2010, Operation Aurora was discovered to have exploited an as of then unknown vulnerability in Internet Explorer. This vulnerability allowed attackers to gain remote control of targeted machines at Google, among other companies.

For this malicious code to be executed, a user simply needed to visit a website using a vulnerable version of Internet Explorer. The attackers then had immediate access to the user's machine with the same privileges as the logged-in user. *Spear phishing*, in which attackers send an email message designed to get recipients to click a link leading to a malicious site, was used to lure the targets.

In the case of Aurora, we pick up this story as soon as the targeted user clicks the link in the spear-phishing email. The resulting packets are contained in the file *aurora.pcapng*.

This capture begins with a three-way handshake between the target (192.168.100.206) and the attacker (192.168.100.202). The initial connection is to port 80, which would lead us to believe this is HTTP traffic. That assumption is confirmed in the fourth packet, which is an HTTP `GET` request for */info* ❶, as shown in Figure 12-16.

As shown in Figure 12-17, the attacker's machine acknowledges receipt of the `GET` request and reports a response code of 302 (Moved Temporarily) in packet 6 ❶, the status code commonly used to redirect a browser to another page, as is the case here. Along with the 302 response code, a Location field specifies the location */info?rFfWELUjLJHpP* ❷.

```
Wireshark · Packet 4 · aurora
> Frame 4: 345 bytes on wire (2760 bits), 345 bytes captured (2760 bits)
> Ethernet II, Src: Vmware_07:ae:27 (00:0c:29:07:ae:27), Dst: HewlettP_bf:91:ee (00:25:b3:bf:91:ee)
> Internet Protocol Version 4, Src: 192.168.100.206, Dst: 192.168.100.202
> Transmission Control Protocol, Src Port: 1031 (1031), Dst Port: 80 (80), Seq: 3982970894, Ack: 3036725423, Len: 291
v Hypertext Transfer Protocol
  > GET /info HTTP/1.1\r\n ❶
    Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, */*\r\n
    Accept-Language: en-us\r\n
    Accept-Encoding: gzip, deflate\r\n
    User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)\r\n
    Host: 192.168.100.202\r\n
    Connection: Keep-Alive\r\n
    \r\n
    [Full request URI: http://192.168.100.202/info]
    [HTTP request 1/3]
    [Response in frame: 6]
    [Next request in frame: 7]
No.: 4 · Time: 0.000465 · Source: 192.168.100.206 · Destination: 192.168.100.202 · Cookie pair: · Protocol: HTTP · Length: 345 · Info: GET /info HTTP/1.1
                                                                                                   Close   Help
```

Figure 12-16: The target makes a `GET` *request for* /info.

```
Wireshark · Packet 6 · aurora
> Frame 6: 191 bytes on wire (1528 bits), 191 bytes captured (1528 bits)
> Ethernet II, Src: HewlettP_bf:91:ee (00:25:b3:bf:91:ee), Dst: Vmware_07:ae:27 (00:0c:29:07:ae:27)
> Internet Protocol Version 4, Src: 192.168.100.202, Dst: 192.168.100.206
> Transmission Control Protocol, Src Port: 80 (80), Dst Port: 1031 (1031), Seq: 3036725423, Ack: 3982971185, Len: 137
v Hypertext Transfer Protocol
  > HTTP/1.1 302 Moved\r\n ❶
    Content-Type: text/html\r\n
    Location: /info?rFfWELUjLJHpP\r\n ❷
    Connection: Keep-Alive\r\n
    Server: Apache\r\n
  > Content-Length: 0\r\n
    \r\n
    [HTTP response 1/3]
    [Time since request: 0.478102000 seconds]
    [Request in frame: 4]
    [Next request in frame: 7]
    [Next response in frame: 18]
No.: 6 · Time: 0.278842 · Source: 192.168.100.202 · Destination: 192.168.100.206 · Cookie pair: · Protocol: HTTP · Length: 191 · Info: HTTP/1.1 302 Moved
                                                                                                   Close   Help
```

Figure 12-17: The client browser is redirected with this packet.

After receiving the HTTP 302 packet, the client initiates another `GET` request to the */info?rFfWELUjLJHpP* URL in packet 7, for which an ACK is received in packet 8. Following the ACK, the next several packets represent data being transferred from the attacker to the target. To take a closer look at that data, right-click one of the packets in the stream, such as packet 9, and select **Follow ▸ TCP Stream**. In this stream output, we see the initial `GET` request, the 302 redirection, and the second `GET` request, as shown in Figure 12-18.

After this, things start getting really strange. The attacker responds to the `GET` request with some very odd-looking content, the first section of which is shown in Figure 12-19.

Figure 12-18: The data stream being transmitted to the client

Figure 12-19: This scrambled content within a `<script>` tag appears to be encoded.

This content appears to be a series of random numbers and letters inside a `<script>` tag ❶. The `<script>` tag is used within HTML to denote the use of a higher-level client-side scripting language whose code is executed on the HTTP client. Within this tag, you normally see various scripting statements. But the gibberish here indicates that the content may be encoded to hide it from detection. Since we know this is exploit traffic, we can assume that this obfuscated section of text contains the hexadecimal padding and shellcode used to exploit the vulnerable service.

NOTE *Script obfuscation is a common technique used by malware to evade detection and hide malicious content. While deobfuscating scripts is beyond the scope of this book, it's a skill you'll develop if you continue to examine malware communication. Many skilled malware analysts can recognize malicious scripts instantly with a quick visual inspection. If you want to challenge yourself, try to manually deobfuscate the script found in this example.*

In the second portion of the content sent from the attacker, shown in Figure 12-20, we finally see some text that is readable. Even without extensive programming knowledge, we can see that this text appears to do some string parsing based on a few variables. This is the last bit of text before the closing `</script>` tag.

Figure 12-20: This portion of the content sent from the server contains readable text and a suspicious iframe.

The last section of data sent from attacker to client, also shown in Figure 12-20, has two parts. The first is the `<span id="vhQYFCtoDnOzUOuxAflDSzVMIHYhjJojAOCHNZtQdlxSPFUeEthCGdRtiIY">` section ❶. The second, contained within the `<span></span>` tags, is `<iframe src="/infowTVeeGDYJWNfsrdrvXiYApnuPoCMjRrSZuKtbVgwuZCXwxKjtEclbPuJPPctcflhsttMRrSyxl.gif" onload="WisgEgTNEfaONekEqaMyAUALLMYW(event)" />` ❷. Once again, this content may be a sign of malicious activity due to the suspiciously long and random strings of unreadable and potentially obfuscated text.

The portion of the code contained within the `<span>` tag is an *iframe*, which is a common method used by attackers to embed additional unexpected content into an HTML page. The `<iframe>` tag creates an inline frame that can go undetected by the user. In this case, the `<iframe>` tag references an oddly named GIF file. As shown in Figure 12-21, when the target's browser sees the reference to this file, it makes a `GET` request for it in packet 21 ❶, and the GIF is sent immediately following that ❷.

No.	Time	Source	Destination	Protocol	Info
21	1.288241	192.168.100.206	192.168.100.202	HTTP	❶ GET /infowTVeeGDYJWNfsrdrvXiYApnuPoCMjRrSZuKtbVgwuZCXwxKjtEclbPuJPPctcflhsttMRrSyxl.gif HTTP/1.1
22	1.488200	192.168.100.202	192.168.100.206	TCP	80 → 1031 [ACK] Seq=3036736951 Ack=3982971911 Win=64518 Len=0
23	1.489366	192.168.100.202	192.168.100.206	HTTP	❷ HTTP/1.1 200 OK (GIF89a) (GIF89a) (image/gif)
24	1.650958	192.168.100.206	192.168.100.202	TCP	1031 → 80 [ACK] Seq=3982971911 Ack=3036737098 Win=64093 Len=0

Figure 12-21: The GIF specified in the iframe is requested and downloaded by the target.

The most peculiar part of this capture occurs at packet 25, when the target initiates a connection back to the attacker on port 4321. Viewing this second stream of communication from the Packet Details pane doesn't yield much information, so we will once again view the TCP stream to get a clearer picture of the data being communicated. Figure 12-22 shows the Follow TCP Stream window output.

In this display, we see something that should set off immediate alarms: a Windows command shell ❶. This shell is sent from the target to the server, indicating that the attacker's exploit attempt succeeded and the payload was dropped. The client transmitted a command shell back to the attacker once the exploit was launched. In this capture, we can even see the attacker interacting with the target by entering the `dir` command ❷ to view a directory listing on the target's machine ❸.

Assuming the exploit has compromised a process running as an administrator or migrated into one, the attacker can do virtually anything they wish to the target's machine. With just a single click, in a matter of seconds, the target has given complete control of their computer to an attacker.

Exploits like this are typically encoded to be unrecognizable when going across the wire to prevent them from being picked up by the network IDS. As such, without prior knowledge of this exploit or even a sample of the exploit code, it might be difficult to tell exactly what is happening on the target's system without further analysis. Luckily, we were able to pick out some telltale signs of malicious code in this packet capture. This includes the obfuscated text in the `<script>` tags, the peculiar iframe, and the command shell seen in plaintext.

Figure 12-22: The attacker is interacting with a command shell through this connection.

Here is a summary of how the Aurora exploit worked here:

- The target receives an email from the attacker that appears to be legitimate, clicks a link within it, and sends a `GET` request to the attacker's malicious site.
- The attacker's web server issues a 302 redirection to the target, and the target's browser automatically issues a `GET` request to the redirected URL.
- The attacker's web server transmits a web page containing obfuscated JavaScript code to the client that includes a vulnerability exploit and an iframe containing a link to a GIF image, which is requested.
- The JavaScript code transmitted earlier is deobfuscated when the page is rendered in the target's browser, and the code executes on their machine, exploiting a vulnerability in Internet Explorer.
- Once the vulnerability is exploited, the payload hidden within the obfuscated code is executed, opening a new session from the target to the attacker on port 4321.
- A command shell is spawned from the payload and shoveled back to the attacker, so that they may interact with it.

From a defender's point of view, we can use this capture file to create a signature for our IDS that might help detect further occurrences of this attack. For example, we might filter on a nonobfuscated part of the capture, such as the plaintext code at the end of the obfuscated text in the `<script>`

tag. Another idea might be to write a signature for all HTTP traffic with a 302 redirection to a site with *info* in the URL. This signature would need some additional tuning to be viable in a production environment, but it's a good start. Of course, it's also important to remember that signatures can be defeated. If the attacker simply changes a few of the strings we've observed here or delivers the exploit through another mechanism, our signatures could be rendered useless. Thus is waged the eternal struggle between attackers and defenders.

NOTE *The ability to create traffic signatures based on malicious traffic samples is a crucial step for someone attempting to defend a network against unknown threats. Analyzing captures such as the one described here are a great way to develop skills in writing those signatures. To learn more about intrusion detection and attack signatures, visit the Snort project at* http://www.snort.org/.

Remote-Access Trojan

ratinfected.pcapng

So far, we've examined security events with some prior knowledge of what is going on. This is a great way to learn what attacks look like, but it's not very real world. In most real-world scenarios, people tasked with defending a network won't examine every packet that goes across the network. Instead, they will use some form of IDS to alert them to anomalies in network traffic that warrant further examination based on a predefined attack signature.

In the next example, we'll begin with a simple alert, as if we're the real-world analyst. In this case, our IDS generates this alert:

```
[**] [1:132456789:2] CyberEYE RAT Session Establishment [**]
[Classification: A Network Trojan was detected] [Priority: 1]
07/18-12:45:04.656854 172.16.0.111:4433 -> 172.16.0.114:6641
TCP TTL:128 TOS:0x0 ID:6526 IpLen:20 DgmLen:54 DF
***AP*** Seq: 0x53BAEB5E  Ack: 0x18874922  Win: 0xFAF0  TcpLen: 20
```

Our next step is to view the signature rule that triggered this alert:

```
alert tcp any any -> $HOME_NET any (msg:"CyberEYE RAT Session Establishment";
content:"|41 4E 41 42 49 4C 47 49 7C|"; classtype:trojan-activity;
sid:132456789; rev:2;)
```

This rule is set to alert whenever it sees a packet from any network entering the internal network with the hexadecimal content `41 4E 41 42 49 4C 47 49 7C`, which converts to *ANA BILGI* in human-readable ASCII. When it is detected, an alert fires, signifying the possible presence of the CyberEYE *remote-access trojan (RAT)*. RATs are malicious programs that run silently on a target's computer and provide a means for the attacker to remotely access the target's machine.

NOTE *CyberEYE is a once popular Turkish-born tool used to create RAT executables and administer compromised hosts. Ironically, the Snort rule seen here fires on the string ANA BILGI, which is Turkish for BASIC INFORMATION.*

Now we'll look at some traffic associated with the alert in *ratinfected .pcapng*. This Snort alert would typically capture only the single packet that triggered the alert, but fortunately we have the entire communication sequence between the hosts. To skip to the punch line, search for the hexadecimal string mentioned in the Snort rule, as follows:

1. Select **Edit ▸ Find Packet** or press CTRL-F.
2. Select the **Hex Value** option from the drop-down menu.
3. Enter the value **`41 4E 41 42 49 4C 47 49 7C`** into the text area.
4. Click **Find**.

As shown in Figure 12-23, you should now see the first occurrence of the hexadecimal string in the data portion of packet 4 ❶.

Figure 12-23: The content string in the Snort alert is first seen here in packet 4.

If you select **Find** several more times, you will see that this string also occurs in packets 5, 10, 32, 156, 280, 405, 531, and 652. Although all of the communication in this capture file is between the attacker (172.16.0.111) and target (172.16.0.114), it appears as though some instances of the string occur in different conversations. While packets 4 and 5 are communicating using ports 4433 and 6641, most of the other instances occur between port 4433 and other randomly selected ephemeral ports. We can confirm that multiple conversations exist by looking at the TCP tab of the Conversations window, as shown in Figure 12-24.

We can visually separate the different conversations in this capture file by colorizing them, as follows:

1. In the filter dialog above the Packet List pane, type the filter **`(tcp.flags .syn == 1) && (tcp.flags.ack == 0)`**. Then press ENTER. This will select the initial SYN packet for each conversation in the traffic.
2. Right-click the first packet and select **Colorize Conversation**.
3. Select **TCP** and then select a color.
4. Repeat this process for the remaining SYN packets, choosing a different color for each.
5. When finished, click **X** to remove the filter.

Address A	Port A	Address B	Port B	Packets	Bytes	Packets A → B	Bytes A → B	Packets B → A	Bytes B → A	Rel Start	Duration	Bits/s A → B	Bits/s B → A
172.16.0.114	6641	172.16.0.111	4433	48	2989	24	1589	24	1400	0.000000000	132.129609	96	84
172.16.0.114	6642	172.16.0.111	4433	10	585	6	343	4	242	0.012008000	132.117839	20	14
172.16.0.114	6643	172.16.0.111	4433	120	91 k	87	89 k	33	1807	74.205235000	0.066042	10 M	218 k
172.16.0.114	6644	172.16.0.111	4433	120	91 k	87	89 k	33	1807	84.209773000	0.070058	10 M	206 k
172.16.0.114	6645	172.16.0.111	4433	121	94 k	89	92 k	32	1753	94.225097000	0.072995	10 M	192 k
172.16.0.114	6646	172.16.0.111	4433	122	94 k	91	93 k	31	1699	104.238408000	0.071781	10 M	189 k
172.16.0.114	6647	172.16.0.111	4433	119	91 k	87	89 k	32	1753	114.238812000	0.070326	10 M	199 k
172.16.0.114	6648	172.16.0.111	4433	119	91 k	87	89 k	32	1753	118.445540000	0.066413	10 M	211 k

Figure 12-24: Three individual conversations exist between the attacker and target.

Having colorized the conversations, we can clear the filter to see how they relate to each other, helping us to track the communication process between the two hosts. The first conversation (ports 6641/4433) is where the communication between the two hosts begins, so it's a good place to start. Right-click any packet within the conversation and select **Follow TCP Stream** to see the data that was transferred, as shown in Figure 12-25.

Figure 12-25: The first conversation yields interesting results.

Immediately, we see that the text string `ANABILGI|556` is sent from the attacker to the target ❶. As a result, the target responds with some basic system information, including the computer name (`CSANDERS-6F7F77`) and the operating system in use (`Windows XP Service Pack 3`) ❷, and begins

repeatedly transmitting the string `BAGLIMI?` back to the attacker ❸. The only communication back from the attacker is the string `CAPSCREEN60` ❹, which appears six times.

This `CAPSCREEN60` string returned by the attacker is interesting, so let's see where it leads. To do so, make sure you've cleared any display filters and search for the text string `CAPSCREEN60` within the packets using the search dialog, specifying the **String** option and ensuring the **Packet bytes** option is selected for where to perform the search.

Upon performing this search, we find the first instance of the string in packet 27. The intriguing thing about this bit of information is that as soon as the string is sent from the attacker to the client, the client acknowledges receipt of the packet, and a new conversation is started in packet 29. You should be able to more easily notice the new conversation starting because of the coloring rules that were applied earlier.

Now, if we follow the TCP stream output of this new conversation (shown in Figure 12-26), we see the familiar string `ANABILGI|12`, followed by the string `SH|556` and, finally, the string `CAPSCREEN|C:\WINDOWS\jpgevhook.dat|84972` ❶. Notice the file path specified after the `CAPSCREEN` string, which is followed by unreadable text. The most intriguing thing here is that the unreadable text is prepended by the string `JFIF` ❷, which a quick Google search will tell you is commonly found at the beginning of JPG files.

Figure 12-26: The attacker appears to be initiating a request for a JPG file.

At this point, it's safe to conclude that the attacker initiated the request to transfer this JPG image. But even more importantly, we are beginning to see a command structure evolve from the traffic. It appears that `CAPSCREEN` is a command sent by the attacker to initiate the transfer of this JPG. In fact, whenever the `CAPSCREEN` command is sent, the result is the same. To verify this, view the TCP stream of each conversation where the `CAPSCREEN` command is present or try using Wireshark's IO graphing feature as follows:

1. Select **Statistics ▸ IO Graph**.
2. Click the plus (+) button to add five lines.
3. Insert the filters **`tcp.stream eq 2`**, **`tcp.stream eq 3`**, **`tcp.stream eq 4`**, **`tcp.stream eq 5`**, and **`tcp.stream eq 6`**, respectively, into the Display Filter of the five newly added lines. Give each one a name as well.
4. Change the y-axis scale for each entry to **Bytes/s**.
5. Click the **Graph 1**, **Graph 2**, **Graph 3**, **Graph 4**, and **Graph 5** buttons to enable the data points for the filters specified.

Figure 12-27 shows the resulting graph.

Figure 12-27: This graph shows that similar activity appears to repeat.

Based on this graph, it appears as though each conversation contains roughly the same amount of data and occurs for the same amount of time. We can now conclude that this activity repeats several times.

You may already have some ideas regarding the content of the JPG image being transferred, so let's see if we can view one of these files. To extract the JPG data from Wireshark, perform the following steps:

1. First, follow the TCP stream of the appropriate packets as we did with Figure 12-25. Packet 29 is a good choice.
2. The communication must be isolated so that we see only the stream of data sent from the target to the attacker. Do this by selecting the arrow next to the drop-down that says **`Entire Conversation (85033 bytes)`**. Be sure to select the appropriate directional traffic, which is **`172.16.0.114:6643 --> 172.16.0.111:4433 (85 kB)`**.
3. In the **Show data as** drop-down, choose **RAW**.
4. Save the data by clicking the **Save As** button, ensuring that you save the file with a *.jpg* file extension.

If you try to open the image now, you may be surprised to find that it won't open. That's because we have one more step to perform. Unlike the scenario in Chapter 10 where we extracted a file cleanly from FTP traffic, the traffic here added some additional content to the data. In this case, the first two lines seen in the TCP stream are actually part of the malware's command sequence, not part of the data that makes up the JPG (see Figure 12-28). When we saved the stream, this extraneous data was also saved. As a result, the file viewer that is looking for a JPG file header is seeing content that doesn't match what it is expecting, and therefore it can't open the image.

Figure 12-28: The extraneous data added by the malware prevents the file from being opened correctly.

Fixing this issue is a painless process, requiring a bit of manipulation with a hex editor. This process is called *file carving*. To carve this file from the exported data, complete the following process:

1. While viewing the TCP Stream in Figure 12-28, click the **Save as** button. Choose a memorable filename and save the file to a location where you can access it again shortly.
2. Download and then install WinHex from *https://www.x-ways.net/winhex/*.
3. Execute WinHex and open the file you just saved from Wireshark.
4. Use your mouse to select all the extraneous data at the beginning of the file. This should be everything occurring before, but not including, the bytes `FF D8 FF E0`, which signify the start of a new JPG file. The bytes to select are highlighted in Figure 12-29.

Figure 12-29: Removing the extraneous bytes from the JPG file

5. Press the **Delete** key to remove the selected data.
6. Click the **Save** button in WinHex's main toolbar to save your changes.

NOTE *I like WinHex for performing this task on Windows, but any hex editor you're familiar with will do.*

With the unneeded bytes of data removed, you should now be able to open the file. It should be clear that the trojan is taking screen captures of the target's desktop and transmitting them back to the attacker (Figure 12-30). After these communication sequences have completed, the communication ends with a normal TCP teardown sequence.

This scenario is a prime example of the thought process an intrusion analyst would follow when analyzing traffic based on an IDS alert:

- Examine the alert and the signature that generated it.
- Confirm whether the signature match was in the traffic in the proper context.
- Examine traffic to find out what the attacker did with the compromised machine.
- Begin containment of the issue before any more sensitive information leaks from the compromised target.

Figure 12-30: The JPG being transferred is a screen capture of the target's computer.

Exploit Kit and Ransomware

cryptowall4_c2.pcapng, ek_to_cryptowall4.pcapng

In our final scenario, we'll look at another investigation that begins with an alert from an IDS. We'll explore the live packets being generated from the infected system and then attempt to trace the source of the compromise. This example will utilize real malware that you would be likely to find infecting a device in your network.

The story beings with an IDS alert generated from Snort in the Sguil console, shown in Figure 12-31. Sguil is a tool used to manage, view, and investigate IDS alerts from one or more sensors. It doesn't provide the most attractive user interface, but it's been around for a while and is a very popular tool for security analysts.

There is a lot of information about this alert available in Sguil. The upper window ❶ shows a summary of the alert. Here you see the time the alert was generated, the source and destination IP addresses and ports, the protocol, and the event message generated from the matching IDS signature. In this case, 192.168.122.145, the local friendly system, is communicating with an unknown external system at 184.170.149.44 over port 80, which is commonly associated with HTTP traffic. The external system is assumed to be hostile since it has shown up in relation to a signature indicating malicious communication and very little is known about it. The signature that matched this traffic is representative of check-in traffic from the CryptoWall malware family, suggesting that a strain of this malware is installed on the friendly system.

Figure 12-31: This IDS alert indicates a CryptoWall 4 infection.

The Sguil console provides the syntax of the matching rule ❷ and the individual packet data that matched the rule ❸. Notice that the packet information is broken down into protocol header and data sections, similar to how packet information is presented in Wireshark. Unfortunately, Sguil only provides information about a single packet that matched, and we need to dig deeper. The next step is to examine the traffic associated with this alert in Wireshark to attempt to validate the traffic and see what's going on. That traffic is contained in the file *cryptowall4_c2.pcapng*.

This packet capture contains the communication that was happening around the time of the alert, and it isn't overly complex. The first conversation occurs in packets 1 through 16, and we can view it easily by following the TCP stream of that conversation (Figure 12-32). At the start of the capture, the local system opens a TCP connection to the hostile host on port 80 and makes a `POST` request to the URL *http://homealldaylong.com/76N1Lm.php?x4tk7t4jo6* ❶ containing a small amount of alphanumeric data ❷. The hostile host responds with an alphanumeric string ❹ and an `HTTP 200 OK` response code ❸ before the connection is terminated gracefully.

Figure 12-32: A small amount of data is being transferred between these hosts via HTTP.

If you look through the rest of the capture file, you'll see that the same sequence repeats between these hosts, with varying amounts of data being transferred each time. Use the filter **`http.request.method == "POST"`** to see three different connections with a similar URL structure (Figure 12-33).

No.	Time	Source	Destination	Protocol	Info
6	0.491136	192.168.122.145	184.170.149.44	HTTP	POST /76N1Lm.php?n=x4tk7t4jo6 HTTP/1.1 (application/x-www-form-urlencoded)
22	15.545562	192.168.122.145	184.170.149.44	HTTP	POST /76N1Lm.php?g=9m822y31lxud7aj HTTP/1.1 (application/x-www-form-urlencoded)
152	41.886948	192.168.122.145	184.170.149.44	HTTP	POST /76N1Lm.php?i=ttfkjb668o38k1z HTTP/1.1 (application/x-www-form-urlencoded)

Figure 12-33: The URL structure shows different data being passed to the same page.

The *76N1Lm.php* portion (the web page) remains the same, but the rest of the content (the parameter and data being passed to the page) varies. The repeating communication sequence combined with the structure of the requests is consistent with command and control (C2) behavior for malware and the signature that generated the alert. It's therefore likely that the local system is infected with CryptoWall, as the signature suggests. You can further verify this by examining a similar sample on the popular CryptoWall Tracker research page: *https://www.cryptowalltracker.org/cryptowall-4.html#networktraffic*.

NOTE *Deciphering the data being communicated between the friendly and hostile system during the C2 sequence would be a little complex for this book. But if you're interested, you can read more about that process here:* https://www.cryptowalltracker.org/communication-protocol.html.

Now that you've verified that malware-based C2 communication is taking place, it's a good idea to remediate the issue and address the infected machine. This is especially important when malware such as CryptoLocker is involved, because it attempts to encrypt the user's data and provides the decryption key only if that user pays a hefty ransom—thus, the term *ransomware* for such malware. The steps to remediate the problem are beyond the scope of this book, but in a real-world scenario, those would be the security analyst's next actions.

A common follow-up question is how the friendly machine got infected in the first place. If this can be determined, you might find other devices that have been infected with other malware in a similar way, or you may be able to develop protection or detection mechanisms to prevent future infection.

The alert packets only showed the active C2 sequence after the infection. In networks performing security monitoring and continuous packet capture, many network sensors are configured to store packet data for a few extra hours or days for forensic investigations. After all, not every organization is equipped to respond to alerts the second they happen. Temporary storage of packets allows us to look at the packet data from the friendly host just before it started the C2 sequence we saw earlier. Those packets are included in the file *ek_to_cryptowall4.pcapng*.

A cursory scroll through this packet capture tells us we have a lot more packets to look through, but they are all HTTP. Since we know how HTTP works already, let's cut to the chase and limit the visible packets to only the requests by using the display filter `http.request`. This shows 11 HTTP requests originating from the friendly host (Figure 12-34).

No.	Time	Source	Destination	Protocol	Info
4	0.534405	192.168.122.145	113.20.11.49	HTTP	GET /index.php/services HTTP/1.1
35	5.265859	192.168.122.145	45.32.238.202	HTTP	GET /contrary/1653873/quite-someone-visitor-nonsense-tonight-sweet-await-gigantic-dance-third HTTP/1.1
39	6.109508	192.168.122.145	45.32.238.202	HTTP	GET /occasional/bX3keHFlYXhmaA HTTP/1.1
123	9.126714	192.168.122.145	45.32.238.202	HTTP	GET /goodness/1854996/earnest-fantastic-thorough-weave-grotesque-forth-awaken-fountain HTTP/1.1
130	14.020289	192.168.122.145	45.32.238.202	HTTP	GET /observation/enVjZ2dtcnpz HTTP/1.1
441	30.245463	192.168.122.145	213.186.33.18	HTTP	POST /VOEHSQ.php?v=x4tk7t4jo6 HTTP/1.1 (application/x-www-form-urlencoded)
456	41.772768	192.168.122.145	184.170.149.44	HTTP	POST /76N1Lm.php?n=x4tk7t4jo6 HTTP/1.1 (application/x-www-form-urlencoded)
472	45.628284	192.168.122.145	213.186.33.18	HTTP	POST /VOEHSQ.php?w=9m822y31lxud7aj HTTP/1.1 (application/x-www-form-urlencoded)
487	56.827194	192.168.122.145	184.170.149.44	HTTP	POST /76N1Lm.php?g=9m822y31lxud7aj HTTP/1.1 (application/x-www-form-urlencoded)
619	71.971402	192.168.122.145	213.186.33.18	HTTP	POST /VOEHSQ.php?h=ttfkjb668o38k1z HTTP/1.1 (application/x-www-form-urlencoded)
634	83.168580	192.168.122.145	184.170.149.44	HTTP	POST /76N1Lm.php?i=ttfkjb668o38k1z HTTP/1.1 (application/x-www-form-urlencoded)

Figure 12-34: There are 11 HTTP requests from the friendly host.

The first request is from the friendly host 192.168.122.145 to an unknown external host 113.20.11.49. An examination of the HTTP portion of this packet (Figure 12-35) tells us that the user requested the page *http://www.sydneygroup.com.au/index.php/services/* ❶ and was referred from a Bing search for *sydneygroup.com.au* ❷. So far, this looks normal.

Next, the friendly host makes four requests to another unknown external host at 45.32.238.202 in packets 35, 39, 123, and 130. As you've seen in earlier examples, it's common for a browser to retrieve content from additional hosts when viewing a web page that stores embedded content or advertisements on third-party servers. This by itself is not worrisome, although the domain in these requests looks somewhat random and suspicious.

Figure 12-35: An HTTP request to an unknown external host

Things get interesting in the `GET` request at packet 39. Following the TCP stream of this exchange (Figure 12-36), you'll notice that a file named *bXJkeHFlYXhmaA* is requested ❶. The name of this file is a little odd, and it doesn't include a file extension either.

Figure 12-36: An oddly named Flash file is downloaded.

Upon closer inspection, we see that the web server identifies the content of this file as *x-shockwave-flash* ❷. Flash is a popular plugin used for streaming media within a browser. It's not abnormal to see Flash content downloaded by a device, but it's worth noting that Flash is notorious for having software vulnerabilities, and it often goes unpatched. The Flash file is downloaded successfully following the request.

After the Flash file is downloaded, there is a request for another similarly named file in packet 130. Following this TCP stream (Figure 12-37), you see a request for a file named *enVjZ2dtcnpz* ❶. The file type isn't identified here with an extension or by the server. The request is followed by the client's downloading a 358,400-byte chunk of unreadable data ❷.

Figure 12-37: Another oddly named file is downloaded, but no file type is identified.

Less than 20 seconds after that file is downloaded, you should see something familiar in the list of HTTP requests from Figure 12-34. Beginning with packet 441, the friendly host starts making HTTP `POST` requests to two different servers using the same C2 pattern observed earlier. It's likely we've identified the source of the infection. The two files that were downloaded were responsible. The first file from the request in packet 39 delivered a Flash exploit, and the second file from the request in packet 130 delivered the malware.

NOTE *You can use malware analysis techniques to decode and analyze the files contained in the packet capture. If you're interested in learning more about reverse engineering malware, I recommend* Practical Malware Analysis *(2012) by Michael Sikorski and Andrew Honig, another No Starch Press book and a personal favorite of mine.*

This scenario represents one of the most common infection techniques. A user was browsing the internet and stumbled onto a site that had been infected with malicious redirection code from an exploit kit. These kits infect legitimate servers and are designed to fingerprint clients to determine their vulnerabilities. The infected page is known as the *kit's landing page*, and its purpose is to redirect the client to another site containing an exploit the kit has determined will be effective against the system.

The packets you've just seen are from the Angler exploit kit, which is perhaps the most frequently observed kit of 2015 and 2016. When the user reached a site that had been infected by Angler, the kit determined the user would be vulnerable to a specific Flash vulnerability. The Flash file was delivered, the system was exploited, and a secondary payload of the CryptoWall malware was downloaded and installed on the host. This entire sequence is depicted in Figure 12-38.

Figure 12-38: The exploit kit infection sequence

Final Thoughts

Entire books could be written on breaking down packet captures in security-related scenarios, analyzing common attacks, and responding to IDS alerts. In this chapter, we've examined some common scanning and enumeration techniques, a common MITM attack, and a couple of examples of how a system might be exploited and what might happen as a result.

13

WIRELESS PACKET ANALYSIS

The world of wireless networking is a bit different from that of traditional wired networking. Although we are still dealing with common communication protocols such as TCP and IP, the game changes a bit when moving to the lowest levels of the OSI model. Here, the data link layer is of special importance due to the nature of wireless networking and the physical layer. Instead of simple wired protocols such as Ethernet, which haven't changed much over time, we have to consider the nuances of wireless protocols such as 802.11, which have evolved pretty quickly. This puts new restrictions on the data we access and how we capture it.

Given these extra considerations, it should come as no surprise that an entire chapter of this book is dedicated to packet capture and analysis on wireless networks. In this chapter, we will discuss exactly why wireless networks are unique when it comes to packet analysis and how to overcome any challenges. Of course, we will be doing this by looking at actual practical examples of wireless network captures.

Physical Considerations

The first thing to consider about capturing and analyzing data transmitted across a wireless network is the physical transmission medium. Until now, we haven't considered the physical layer because we've been communicating over physical cabling. Now we're communicating through invisible airwaves, with packets flying right by us.

Sniffing One Channel at a Time

A consideration distinct to capturing traffic from a wireless local area network (WLAN) is that the wireless spectrum is a shared medium. Unlike wired networks, where each client has its own network cable connected to a switch, the wireless communication medium is the airspace clients share, which is limited in size. A single WLAN will occupy only a portion of the 802.11 spectrum. This allows multiple systems to operate in the same physical area on different portions of the spectrum.

NOTE *Wireless networking is based on the 802.11 standard, developed by the Institute of Electrical and Electronics Engineers (IEEE). Throughout this chapter, the terms* wireless network *and* WLAN *refer to networks that adhere to the 802.11 standard. The most popular versions of this standard are 802.11a, b, g, and n. Each offers a unique set of features and characteristics, with newer standards such as n offering faster speed. They all still use the same spectrum.*

This separation of space is made possible by dividing the spectrum into operation channels. A *channel* is simply a portion of the 802.11 wireless spectrum. In the United States, 11 channels are available (more are allowed in some other countries). This is relevant because, just as a WLAN can operate on only one channel at a time, we can sniff packets on only one channel at a time, as illustrated in Figure 13-1. Therefore, if you are troubleshooting a WLAN operating on channel 6, you must configure your system to capture traffic seen on channel 6.

Figure 13-1: Sniffing wirelessly can be tedious, since it can be done on only one channel at a time.

Traditional wireless sniffing can only be done one channel at a time, with an exception: certain wireless scanning applications use a technique called *channel hopping* to change channels rapidly in order to collect data. One of the most popular tools of this type, Kismet (*http://www.kismetwireless.net/*), can hop up to 10 channels per second, a capability that makes it very effective at sniffing multiple channels at once.

Wireless Signal Interference

With wireless communications, we sometimes can't rely on the integrity of the data being transmitted over the air. It's possible that something will interfere with the signal. Wireless networks include some features to handle interference, but those features don't always work. Therefore, when capturing packets over a wireless network, you must pay close attention to your environment to ensure that there are no significant sources of interference, such as big reflective surfaces, large rigid objects, microwaves, 2.4 GHz phones, thick walls, or high-density surfaces. These can cause packet loss, duplicated packets, and malformed packets.

Interference between channels is also a concern. Although you can sniff only one channel at a time, this comes with a small caveat: several transmission channels are available in the wireless networking spectrum, but because space is limited, there is a slight overlap between channels, as illustrated in Figure 13-2. This means that if there is traffic present on channel 4 and channel 5, and you are sniffing on one of these channels, you will likely capture packets from the other channel. Typically, networks that coexist in the same area are designed to use nonoverlapping channels of 1, 6, and 11, so you probably won't encounter this problem. But just in case, you should understand why it happens.

Figure 13-2: There is overlap between channels due to limited spectrum space.

Detecting and Analyzing Signal Interference

Troubleshooting wireless signal interference isn't something that can be done by looking at packets in Wireshark. If you are going to make a habit or a career out of troubleshooting WLANs, you will surely need to check for signal interference regularly. This task is done with a *spectrum analyzer*, a tool that displays data or interference across the spectrum.

Commercial spectrum analyzers can cost upward of thousands of dollars, but there is a great solution for common everyday use. MetaGeek makes a

USB hardware device, the Wi-Spy, that monitors the entire 802.11 spectrum for signals. When paired with MetaGeek's inSSIDer or Chanalyzer software, this hardware outputs the spectrum graphically to aid in the troubleshooting process. Sample output from Chanalyzer is shown in Figure 13-3.

Figure 13-3: This Chanalyzer output shows four signals equally spaced along the Wi-Fi spectrum.

Wireless Card Modes

Before we start sniffing wireless packets, we need to look at the different modes in which a wireless card can operate as it pertains to packet capture.

Four wireless NIC modes are available:

Managed mode This mode is used when your wireless client connects directly to a wireless access point (WAP). In these cases, the driver associated with the wireless NIC relies on the WAP to manage the entire communication process.

Ad hoc mode This mode is used when you have a wireless network setup in which devices connect directly to each other. In this mode, two wireless clients that want to communicate with each other share the responsibilities that a WAP would normally handle.

Master mode Some higher-end wireless NICs also support master mode. This mode allows the wireless NIC to work in conjunction with specialized driver software in order to allow the computer to act as a WAP for other devices.

Monitor mode This is the most important mode for our purposes. Monitor mode is used when you want your wireless client to stop transmitting and receiving data and instead only listen to the packets flying through the air. For Wireshark to capture wireless packets, your wireless NIC and accompanying driver must support monitor mode (also known as RFMON mode).

Most users use wireless cards in only managed mode or ad hoc mode. A graphical representation of the way each mode operates is shown in Figure 13-4.

Figure 13-4: The different wireless card modes

NOTE *I'm often asked which wireless card I recommend for wireless packet analysis. I use and highly recommend products from ALFA network. Their products are regarded as some of the best on the market for ensuring you are capturing every possible packet, and they're cost-effective and portable. ALFA's products are available through most online computer hardware retailers.*

Sniffing Wirelessly in Windows

Even if you have a wireless NIC that supports monitor mode, most Windows-based wireless NIC drivers won't allow you to change into this mode. This means that you'll only be able to capture packets to and from the wireless interface on the device you're using to connect to the network. To capture packets between all devices on a channel, you'll need extra hardware.

Configuring AirPcap

AirPcap from Riverbed Technologies (*http://www.riverbed.com/*) is designed to overcome the limitations that Windows places on wireless packet analysis. AirPcap is a small USB device that resembles a flash drive, as shown in Figure 13-5. It is designed to capture wireless traffic from one or more specified channels. AirPcap uses the WinPcap driver and a special client configuration utility.

Figure 13-5: The AirPcap device is very compact, making it easy to tote along with a laptop.

The AirPcap configuration program (shown in Figure 13-6) is simple to use, with only a few configurable options:

Interface You can select the device you are using for your capture here. Some advanced analysis scenarios may require you to use more than one AirPcap device to sniff simultaneously on multiple channels.

Blink Led Clicking this button will make the LED lights on the AirPcap device blink. This is primarily used to identify the specific adapter you are using if you have multiple AirPcap devices.

Channel In this field, you select the channel you want AirPcap to listen on.

Figure 13-6: The AirPcap configuration program

Extension Channel Here you can select an extension channel, a feature of 802.11n adapters allowing for the creation of wider channels.

Include 802.11 FCS in Frames By default, some systems strip the last four checksum bits from wireless packets. This checksum, known as a frame check sequence (FCS), is used to ensure that packets have not been corrupted during transmission. Unless you have a specific reason to do otherwise, check this box to include the FCS checksums.

Capture Type The three options here are 802.11 Only, 802.11 + Radio, and 802.11 + PPI. The 802.11 Only option includes the standard 802.11 packet header on all captured packets. The 802.11 + Radio option includes this header and prepends it with a radiotap header, which contains additional information about the packet, such as data rate, frequency, signal level, and noise level. The 802.11 + PPI option adds the Per-Packet Information Header, which contains additional information about 802.11n packets.

FCS Filter Even if you uncheck the Include 802.11 FCS in Frames box, this option lets you filter out packets that FCS determines are corrupted. Use the Valid Frames option to show only those packets that FCS thinks can be received successfully.

WEP Configuration This area (accessible on the Keys tab of the AirPcap Control Panel) allows you to enter WEP decryption keys for the networks you will be sniffing. To be able to interpret data encrypted by WEP, you will need to enter the correct WEP keys into this field. WEP keys are discussed in "Successful WEP Authentication" on page 309.

Capturing Traffic with AirPcap

Once you have AirPcap installed and configured, the capture process should be familiar to you. Just start up Wireshark and select the AirPcap interface to start collecting packets from it (Figure 13-7).

Figure 13-7: Selecting the AirPcap interface to capture packets

Remember that you will be capturing packets from whatever channel you selected in the AirPcap configuration utility. If you don't see the packets you're looking for, it's probably because you're looking on the wrong channel. Change the channel by stopping the active capture, selecting a new channel in the AirPcap configuration utility, and restarting the capture. You can't actively capture packets while you attempt to change the channel.

If you need to verify what channel you're capturing from within Wireshark, an easy way is to view wireless capture statistics. Do this by clicking **Wireless ▸ WLAN Traffic** from the main drop-down menu. The resulting window will show you the devices that were observed and information about them, including the 802.11 channel, as shown in Figure 13-8.

Figure 13-8: The Wireless LAN Statistics window indicates this data was captured by listening to channel 11.

Sniffing Wirelessly in Linux

Sniffing in Linux is simply a matter of enabling monitor mode on the wireless NIC and firing up Wireshark. Unfortunately, the procedure for enabling monitor mode differs with each model of wireless NIC, so I can't offer definitive advice for doing this. In fact, some wireless NICs don't require you to enable monitor mode. Your best bet is to do a quick Google search for your NIC model to determine whether you need to enable it and, if so, how.

One of the more common ways to enable monitor mode in Linux is through its built-in wireless extensions. You can access these wireless extensions with the `iwconfig` command. If you type `iwconfig` from the console, you should see results like this:

```
  $ iwconfig
❶ Eth0  no wireless extensions
  Lo0   no wireless extensions
❷ Eth1  IEEE 802.11g        ESSID: "Tesla Wireless Network"
        Mode: Managed  Frequency: 2.462 GHz  Access Point: 00:02:2D:8B:70:2E
        Bit Rate: 54 Mb/s  Tx-Power-20 dBm  Sensitivity=8/0
        Retry Limit: 7  RTS thr: off  Fragment thr: off
        Power Management: off
        Link Quality=75/100  Signal level=-71 dBm  Noise level=-86 dBm
        Rx invalid nwid: 0  Rx invalid crypt: 0  Rx invalid frag: 0
        Tx excessive retries: 0  Invalid misc: 0  Missed beacon: 2
```

The output from the `iwconfig` command shows that the `Eth1` interface can be configured wirelessly. This is apparent because it shows data for the 802.11g protocol ❷, whereas the interfaces `Eth0` and `Lo0` return the phrase `no wireless extensions` ❶.

Along with all the wireless information this command provides, such as the wireless extended service set ID (ESSID) and frequency, notice that the second line under `Eth1` shows that the mode is currently set to managed. This is what we want to change.

To change the `Eth1` interface to monitor mode, you must be logged in as the root user, either directly or via the switch user (`su`) command, as shown here:

```
$ su
Password: <enter root password here>
```

Once you're root, you can type commands to configure the wireless interface options. To configure `Eth1` to operate in monitor mode, enter this:

```
# iwconfig eth1 mode monitor
```

Once the NIC is in monitor mode, running the `iwconfig` command again should reflect your changes. Now ensure that the `Eth1` interface is operational by entering the following:

```
# iwconfig eth1 up
```

We'll also use the `iwconfig` command to change the channel we are listening on. Change the channel of the `Eth1` interface to channel 3 by entering this:

```
# iwconfig eth1 channel 3
```

NOTE *You can change channels on the fly as you are capturing packets, so don't hesitate to do so at will. The `iwconfig` command can also be scripted to make the process easier.*

When you have completed these configurations, start Wireshark and begin your packet capture.

802.11 Packet Structure

80211beacon.pcapng

The primary difference between wireless and wired packets is the addition of the 802.11 header. This layer 2 header contains extra information about the packet and the medium by which it is transmitted. There are three types of 802.11 packets:

Management These packets are used to establish connectivity between hosts at layer 2. Some important subtypes of management packets include authentication, association, and beacon packets.

Control Control packets allow for delivery of management and data packets, and they are concerned with congestion management. Common subtypes include request-to-send and clear-to-send packets.

Data These packets contain actual data and are the only types of packets that can be forwarded from the wireless network to the wired network.

The type and subtype of a wireless packet determine its structure, so there are a large number of possible structures. We'll examine one such structure by looking at a single packet in the file *80211beacon.pcapng*. This file contains an example of a management packet called a *beacon*, as shown in Figure 13-9.

A beacon is one of the most informative wireless packets you can find. It is sent as a broadcast packet from a WAP across a wireless channel to notify any listening wireless clients that the WAP is available and to define the parameters that must be set in order to connect to it. In our example file, you can see that this packet is defined as a beacon in the Type/Subtype field in the 802.11 header ❶.

A great deal of additional information is found in the 802.11 management frame header, including the following:

Timestamp The time the packet was transmitted

Beacon Interval The interval at which the beacon packet is retransmitted

Capabilities Information Information about the hardware capabilities of the WAP

Figure 13-9: This is an 802.11 beacon packet.

SSID parameter set The SSID (network name) broadcast by the WAP

Supported Rates The data transfer rates supported by the WAP

DS Parameter set The channel on which the WAP is broadcasting

The header also includes the source and destination addresses and vendor-specific information.

Based on this, we can determine quite a few things about the WAP transmitting the beacon in the example file. It is apparent that it is a D-Link device ❷ using the 802.11b standard (B) ❸ on channel 11 ❹.

Although the exact contents and purpose of 802.11 management packets will change, the general structure remains similar to this example.

Adding Wireless-Specific Columns to the Packet List Pane

In previous chapters, we've leveraged Wireshark's flexible interface to add situationally appropriate columns. Before we proceed with any additional wireless analysis, it will be helpful to add the following three columns to the Packet List pane.

- The Channel column, to show the channel on which the packet was collected
- The Signal Strength column, to show the signal strength of a captured packet in dBm
- The Data Rate column, to show the throughput rate of a captured packet

These indicators can be of great help when troubleshooting wireless connections. For instance, even if your wireless client software says you have excellent signal strength, doing a capture and checking these columns may show you a number that does not support this claim.

To add these columns to the Packet List pane, follow these steps:

1. Choose **Edit ▸ Preferences**.
2. Navigate to the Columns section and click +.
3. Type **Channel** in the Title field, select **Custom** in the Type drop-down list, and use the filter **wlan_radio.channel** in the Field Name box.
4. Repeat this process for the Signal Strength and Data Rate columns, titling them appropriately and selecting **wlan_radio.signal_dbm** and **wlan_radio.data_rate**, respectively, in the Field Name drop-down list. Figure 13-10 shows what the Preferences window should look like after you have added all three columns.

Figure 13-10: Adding the IEEE wireless-specific columns in the Packet List pane

5. Click **OK** to save your changes.

Wireless-Specific Filters

We discussed the benefits of capture and display filters in Chapter 4. Filtering traffic in a wired infrastructure is a lot easier since each device has its own dedicated cable. In a wireless network, however, all traffic generated by wireless clients coexists on shared channels, meaning that a capture of any one channel may contain traffic from dozens of clients. This section is devoted to some packet filters that can be used to help you find specific traffic.

Filtering Traffic for a Specific BSS ID

Each WAP in a network has a unique identifying name called its *basic service set identifier (BSS ID)*. This name is sent in every wireless management packet and data packet that the access point transmits.

Once you know the name of the BSS ID you want to examine, all you really need to do is find a packet that has been sent from that particular WAP. Wireshark shows the transmitting WAP in the Info column of the Packet List pane, so finding this information is typically easy.

Once you have a packet from the WAP of interest, find its BSS ID field in the 802.11 header. This is the address on which you will base your filter. After you have found the BSS ID MAC address, you can use this filter:

```
wlan.bssid == 00:11:22:33:44:55
```

And you will see only the traffic flowing through the specified WAP.

Filtering Specific Wireless Packet Types

Earlier in this chapter, we discussed the different types of wireless packets you might see on a network. You'll often need to filter based on these types and subtypes. This can be done with the filters `wlan.fc.type` for specific types and `wc.fc.type_subtype` for specific type or subtype combinations. For instance, to filter for a NULL data packet (a Type 2 Subtype 4 packet in hex), you could use the filter `wlan.fc.type_subtype == 0x24`. Table 13-1 provides a quick reference to some common filters you might need when filtering on 802.11 packet types and subtypes.

Table 13-1: Wireless Types/Subtypes and Associated Filter Syntax

Frame type/subtype	Filter syntax
Management frame	`wlan.fc.type == 0`
Control frame	`wlan.fc.type == 1`
Data frame	`wlan.fc.type == 2`

(continued)

Table 13-1 *(continued)*

Frame type/subtype	Filter syntax
Association request	`wlan.fc.type_subtype == 0x00`
Association response	`wlan.fc.type_subtype == 0x01`
Reassociation request	`wlan.fc.type_subtype == 0x02`
Reassociation response	`wlan.fc.type_subtype == 0x03`
Probe request	`wlan.fc.type_subtype == 0x04`
Probe response	`wlan.fc.type_subtype == 0x05`
Beacon	`wlan.fc.type_subtype == 0x08`
Disassociate	`wlan.fc.type_subtype == 0x0A`
Authentication	`wlan.fc.type_subtype == 0x0B`
Deauthentication	`wlan.fc.type_subtype == 0x0C`
Action frame	`wlan.fc.type_subtype == 0x0D`
Block ACK requests	`wlan.fc.type_subtype == 0x18`
Block ACK	`wlan.fc.type_subtype == 0x19`
Power save poll	`wlan.fc.type_subtype == 0x1A`
Request to send	`wlan.fc.type_subtype == 0x1B`
Clear to send	`wlan.fc.type_subtype == 0x1C`
ACK	`wlan.fc.type_subtype == 0x1D`
Contention free period end	`wlan.fc.type_subtype == 0x1E`
NULL data	`wlan.fc.type_subtype == 0x24`
QoS data	`wlan.fc.type_subtype == 0x28`
Null QoS data	`wlan.fc.type_subtype == 0x2C`

Filtering a Specific Frequency

If you are examining a compilation of traffic that includes packets from multiple channels, it can be very useful to filter based on each individual channel. For instance, if you are expecting to have traffic present on only channels 1 and 6, you can input a filter to show all channel 11 traffic. If you find any traffic, then you'll know that something is wrong—perhaps a misconfiguration or a rogue device. To filter on a specific channel, use this filter syntax:

```
wlan_radio.channel == 11
```

This will show all traffic on channel 11. You can replace the `11` value with the channel you wish to filter. There are hundreds of additional useful filters that you can use for wireless network traffic. You can view additional wireless capture filters on the Wireshark wiki at *http://wiki.wireshark.org/*.

Saving a Wireless Profile

It's a fair bit of work to go through all the trouble of setting up specific columns and saving custom filters for wireless packet analysis. Instead of reconfiguring and removing columns and filters all the time, you can create and save a custom profile to quickly switch between configurations for wired and wireless analysis.

To save a custom profile, first configure wireless columns and filters to your liking. Then right-click the active profile listing at the bottom right of the screen and click **New**. Name the profile **Wireless** and click **OK**.

Wireless Security

The biggest concern when deploying and administering a wireless network is the security of the data transmitted across it. With data flying through the air, free for the taking by anyone who knows how, it's crucial that the data be encrypted. Otherwise, anyone with Wireshark and an AirPcap can see it.

NOTE *When another layer of encryption, such as SSL or SSH, is used, traffic will still be encrypted at that layer, and the user's communication will still be unreadable by a person with a packet sniffer.*

The original preferred method for securing data transmitted over wireless networks was in accordance with the Wired Equivalent Privacy (WEP) standard. WEP was mildly successful for years until several weaknesses were uncovered in its encryption key management. To improve security, new standards were created. These include the Wi-Fi Protected Access (WPA) and the more secure WPA2 standards. Although WPA and WPA2 are fallible, they are considered more secure than WEP.

In this section, we'll look at some WEP and WPA traffic, along with examples of failed authentication attempts.

Successful WEP Authentication

3e80211_WEPauth.pcapng

The file *3e80211_WEPauth.pcapng* contains an example of a successful connection to a WEP-enabled wireless network. The security on this network is set up using a WEP key. This is a key you must provide to the WAP (the wireless access point) in order to authenticate to it and decrypt data sent from it. You can think of this WEP key as a wireless network password.

As shown in Figure 13-11, the capture file begins with a challenge from the WAP (28:c6:8e:ab:96:16) to the wireless client (ac:cf:5c:78:6c:9c) in packet 3 ❶. The purpose of this challenge is to determine whether the wireless client has the correct WEP key. You can see this challenge by expanding the 802.11 header and its tagged parameters.

The wireless client responds, as shown in Figure 13-12, by decrypting the challenge text ❶ with the WEP key and returning it to the WAP in packet 4. The WEP key was provided by the user when attempting to connect to the wireless network.

```
Wireshark · Packet 3 · 3e80211_wepauth
> Frame 3: 184 bytes on wire (1472 bits), 184 bytes captured (1472 bits) on interface 0
> Radiotap Header v0, Length 20
> 802.11 radio information
v IEEE 802.11 Authentication, Flags: ........C
    Type/Subtype: Authentication (0x000b)
  > Frame Control Field: 0xb000
    .000 0001 0011 1010 = Duration: 314 microseconds
    Receiver address: Apple_78:6c:9c (ac:cf:5c:78:6c:9c)
    Destination address: Apple_78:6c:9c (ac:cf:5c:78:6c:9c)
    Transmitter address: Netgear_ab:96:16 (28:c6:8e:ab:96:16)
    Source address: Netgear_ab:96:16 (28:c6:8e:ab:96:16)
    BSS Id: Netgear_ab:96:16 (28:c6:8e:ab:96:16)
    .... .... .... 0000 = Fragment number: 0
    0000 0000 0100 .... = Sequence number: 4
  > Frame check sequence: 0xad79ad88 [correct]
v IEEE 802.11 wireless LAN management frame
  v Fixed parameters (6 bytes)
      Authentication Algorithm: Shared key (1)
      Authentication SEQ: 0x0002
      Status code: Successful (0x0000)
  v Tagged parameters (130 bytes)
    v Tag: Challenge text ❶
        Tag Number: Challenge text (16)
        Tag length: 128
        Challenge Text: 97ebed97ca8a77fddae72796f85d560950a64cad5053e5d9...
No.: 3 · Time: 0.048458 · Source: Netgear_ab:96:16 · Destination: Apple_78:6c:...: Authentication · Type/Subtype: Authentication · Type/Subtype: Authentica
                                                                    Close   Help
```

Figure 13-11: The WAP issues challenge text to the wireless client.

```
Wireshark · Packet 4 · 3e80211_wepauth
> Frame 4: 203 bytes on wire (1624 bits), 203 bytes captured (1624 bits) on interface 0
> Radiotap Header v0, Length 20
> 802.11 radio information
v IEEE 802.11 Authentication, Flags: .p......C
    Type/Subtype: Authentication (0x000b)
  > Frame Control Field: 0xb040
    .000 0001 0011 1010 = Duration: 314 microseconds
    Receiver address: Netgear_ab:96:16 (28:c6:8e:ab:96:16)
    Destination address: Netgear_ab:96:16 (28:c6:8e:ab:96:16)
    Transmitter address: Apple_78:6c:9c (ac:cf:5c:78:6c:9c)
    Source address: Apple_78:6c:9c (ac:cf:5c:78:6c:9c)
    BSS Id: Netgear_ab:96:16 (28:c6:8e:ab:96:16)
    .... .... .... 0000 = Fragment number: 0
    0000 0110 0100 .... = Sequence number: 100
  > Frame check sequence: 0x1afc56d8 [correct]
  > WEP parameters
v Data (147 bytes)
  ❶ Data: e839cc710bf2da1c936cfe6a722fb8e7a8eee78529270d43...
    [Length: 147]
No.: 4 · Time: 0.050464 · Source: Apple_78:6c:9c · Destination: Netgear_ab:96:...: Authentication · Type/Subtype: Authentication · Type/Subtype: Authentica
                                                                    Close   Help
```

Figure 13-12: The wireless client sends the unencrypted challenge text back to the WAP.

The WAP responds to the wireless client in packet 5, as shown in Figure 13-13. The response contains a notification that the authentication process was successful ❶.

Figure 13-13: The WAP alerts the client that authentication was successful.

Finally, after successful authentication, the client can transmit an association request, receive an acknowledgment, and complete the connection process, as shown in Figure 13-14.

No.	Time	Source	Destination	Protocol	Length	Channel	Signal strength (dBm)	Data rate	Info
6	0.052565	Apple_78:6c:9c	Netgear_ab:96:16	802.11	110	1	-40	1	Association Request, SN=101, FN=0, Flags=.....C, SSID=DENVEROFFICE
7	0.053902	Netgear_ab:96:16	Apple_78:6c:9c	802.11	119	1	-17	1	Association Response, SN=6, FN=0, Flags=.....C

Figure 13-14: The authentication process is followed by a simple two-packet association request and response.

Failed WEP Authentication

3e80211_WEPauthfail.pcapng

In our next example, a user enters a WEP key to connect to a WAP. After several seconds, the wireless client utility reports that it was unable to connect to the wireless network but fails to tell why. The resulting file is *3e80211_WEPauthfail.pcapng*.

As with the successful attempt, this communication begins with the WAP's sending challenge text to the wireless client in packet 3. In packet 4, the wireless client sends its response using the WEP key provided by the user.

At this point, we would expect to see a notification that the authentication was successful, but we see something different in packet 5, as shown in Figure 13-15 ❶.

Figure 13-15: This message tells us the authentication was unsuccessful.

This message tells us that the wireless client's response to the challenge text was incorrect and suggests that the WEP key the client used to decrypt the text must have also been incorrect. As a result, the connection process has failed. It must be reattempted with the proper WEP key.

Successful WPA Authentication

3e80211_WPAauth .pcapng

WPA uses a very different authentication mechanism than WEP, but it still relies on the user to enter a key into the wireless client to connect to the network. An example of a successful WPA authentication is found in the file *3e80211_WPAauth.pcapng*.

The first packet in this file is a beacon broadcast from the WAP. Expand the 802.11 header of this packet, look under tagged parameters, and expand the Vendor Specific heading, as shown in Figure 13-16. You should see a section devoted to the WPA attributes of the WAP ❶. This lets us know the version and implementation of WPA that a WAP supports, if any.

Figure 13-16: This beacon lets us know that the WAP supports WPA authentication.

Once the beacon is received, the wireless client (ac:cf:5c:78:6c:9c) broadcasts a probe request in packet 2 that is received by the WAP (28:c6 :8e:ab:96:16), which responds in packet 3. After that, authentication and association requests and responses are generated between the wireless client and WAP in packets 4 through 7. These are similar to the authentication and association packets we saw in the WEP example earlier, but no challenge and response occur here. That exchange happens next.

Things really start to pick up in packet 8. This is where the WPA handshake begins, continuing through packet 11. During the handshake, the WPA challenge and response take place, as shown in Figure 13-17.

No.	Time	Source	Destination	Protocol	Length	Channel	Signal strength (dBm)	Data rate	Info
8	0.…	Netgear_ab:96:16	Apple_78:6c:9c	EAPOL	157	1	-18	24	Key (Message 1 of 4)
9	0.…	Apple_78:6c:9c	Netgear_ab:96:16	EAPOL	183	1	-42	1	Key (Message 2 of 4)
10	0.…	Netgear_ab:96:16	Apple_78:6c:9c	EAPOL	181	1	-18	36	Key (Message 3 of 4)
11	0.…	Apple_78:6c:9c	Netgear_ab:96:16	EAPOL	157	1	-42	1	Key (Message 4 of 4)

Figure 13-17: These packets are a part of the WPA handshake.

There are two challenges and responses. Each can be matched with the other based on the Replay Counter field under the 802.1x Authentication header, as shown in Figure 13-18. Notice that the Replay Counter value for the first two handshake packets is `1` ❶ and for the second two handshake packets is `2` ❷.

Figure 13-18: The Replay Counter field helps us pair challenges and responses.

After the WPA handshake is completed and authentication is successful, data begins transferring between the wireless client and the WAP.

NOTE *This example is from a WAP using WPA with TKIP encryption. TKIP is just one method for encrypting data on WLANs. There are many other types of encryption, and different access points will support different techniques. A WAP using a different encryption method or WPA version will likely exhibit different characteristics at the packet level. You can read the RFC document relating to the technology being used to better decipher what the connection sequence should look like.*

Failed WPA Authentication

3e80211_WPAauthfail.pcapng

As with WEP, we'll look at what happens when a user enters a WPA key and the wireless client utility reports that it was unable to connect to the wireless network without indicating the problem. The resulting file is *3e80211_WPAauthfail.pcapng*.

The capture file begins in a manner identical to the file showing a successful WPA authentication and includes probe, authentication, and association requests. The WPA handshake begins in packet 8, but in this case, there are eight handshake packets instead of the four we saw in the successful authentication attempt.

Packets 8 and 9 represent the first two packets seen in the WPA handshake. In this case, however, the challenge text the client sends back to the WAP is incorrect. As a result, the sequence is repeated in packets 10 and 11, 12 and 13, and 14 and 15, as shown in Figure 13-19. Each request and response can be paired using the Replay Counter value.

No.	Time	Source	Destination	Protocol	Length	Channel	Signal strength (dBm)	Data rate	Info
8	0.073773	Netgear_ab:96:16	Apple_78:6c:9c	EAPOL	157	1	-18	24	Key (Message 1 of 4)
9	0.076510	Apple_78:6c:9c	Netgear_ab:96:16	EAPOL	183	1	-30	1	Key (Message 2 of 4)
10	1.074290	Netgear_ab:96:16	Apple_78:6c:9c	EAPOL	157	1	-19	24	Key (Message 1 of 4)
11	1.076573	Apple_78:6c:9c	Netgear_ab:96:16	EAPOL	183	1	-32	1	Key (Message 2 of 4)
12	2.075292	Netgear_ab:96:16	Apple_78:6c:9c	EAPOL	157	1	-18	36	Key (Message 1 of 4)
13	2.077610	Apple_78:6c:9c	Netgear_ab:96:16	EAPOL	183	1	-29	1	Key (Message 2 of 4)
14	3.077211	Netgear_ab:96:16	Apple_78:6c:9c	EAPOL	157	1	-18	48	Key (Message 1 of 4)
15	3.079537	Apple_78:6c:9c	Netgear_ab:96:16	EAPOL	183	1	-32	1	Key (Message 2 of 4)

Figure 13-19: The additional EAPoL (Extensible Authentication Protocol over LAN) packets here indicate the failed WPA authentication.

Once the handshake process has been attempted and failed four times, the communication is aborted. As shown in Figure 13-20, the wireless client deauthenticates from the WAP in packet 16 ❶.

Figure 13-20: After failing the WPA handshake, the client deauthenticates.

Final Thoughts

Although wireless networks are still considered somewhat insecure, unless a plethora of additional security mechanisms are piled on, that concern hasn't slowed their deployment in various organizational environments. As communication without wires is the new norm, it's crucial to be able to capture and analyze data on wireless networks, as well as wired ones. The skills and concepts taught in this chapter are by no means exhaustive, but they should provide a jump-start for understanding the intricacies of troubleshooting wireless networks with packet analysis.

A

FURTHER READING

Although the tool you've primarily used in this book is Wireshark, a great many additional tools will come in handy when you're performing packet analysis—whether for general troubleshooting, slow networks, security issues, or wireless networks. This appendix lists some useful packet analysis tools and other learning resources.

Packet Analysis Tools

Let's take a look at a few of the tools I've found useful for packet analysis.

CloudShark

CloudShark (developed by QA Café) is my favorite tool for storing, indexing, and sorting packet captures. CloudShark is a commercial web application that serves as a packet capture repository. It allows you to tag packet

captures for quick reference and to add comments in the captures themselves. It even provides some analysis features similar to those in Wireshark (Figure A-1).

Figure A-1: A sample capture file viewed with CloudShark

If you or your organization maintains a large library of packet captures, or you're like me and are always losing your files, then CloudShark can help. I have CloudShark deployed in my network, and I used it to store and organize all the packet captures for this book. You can learn more about CloudShark at *https://www.cloudshark.org/*.

WireEdit

You may need to create specifically formatted packets to support intrusion detection system testing, penetration testing, or network software development. One option is to re-create the scenario that will generate the packets you need in a lab, but doing so can be time-consuming. Another technique is to find a similar packet and manually edit it to match your needs. My favorite tool for this task is WireEdit, a graphical tool that allows you to edit specific values in a packet. The very intuitive user interface is similar to Wireshark's. WireEdit will even recalculate packet checksums so that your packets don't appear invalid when opened in Wireshark. You can learn more about WireEdit at *https://wireedit.com/*.

Cain & Abel

Discussed in Chapter 2, Cain & Abel is one of the better Windows tools for ARP cache poisoning. Cain & Abel is actually a very robust suite of tools, and you will surely be able to find other uses for it as well. It is available from *http://www.oxid.it/cain.html*.

Scapy

Scapy is a very powerful Python library that you can use to create and manipulate packets based on command line scripts within its environment. Simply put, Scapy is the most powerful and flexible packet-crafting application available. You can read more about Scapy, download it, and view sample Scapy scripts at *http://www.secdev.org/projects/scapy/*.

TraceWrangler

Packet captures contain a lot of information about your network. If you need to share a packet capture from your network with a vendor or colleague, you might not want them to have that information. TraceWrangler helps solve this problem by providing the ability to sanitize packet captures by anonymizing the different types of addresses present. It has a few other features, such as the ability to edit and merge capture files, but I primarily use it for sanitization. Download TraceWrangler at *https://www.tracewrangler.com/*.

Tcpreplay

Whenever I have a set of packets that I need to retransmit over the wire to see how a device reacts to them, I use Tcpreplay. This tool is designed specifically to retransmit the packets contained within a packet capture file. Download it from *http://tcpreplay.synfin.net/*.

NetworkMiner

NetworkMiner is a tool primarily used for network forensics, but I've found it useful in a variety of other situations as well. Although it can be used to capture packets, its real strength is how it parses packet capture files. NetworkMiner will take a PCAP file and break it down into the operating systems detected and the sessions between hosts. It even allows you to extract transferred files directly from the capture (Figure A-2). All these features are available in the free version; the commercial version offers a few other helpful features, such as the ability to perform OS fingerprinting, compare findings against a whitelist, and increase the speed of packet capture processing. NetworkMiner is free to download from *http://www.netresec.com/?page=NetworkMiner*.

Figure A-2: Using NetworkMiner to examine files in a packet capture

CapTipper

One thing I hope you've learned in this book is that finding the answers you need often involves looking at the same data in a different way. CapTipper is a tool designed for security practitioners who analyze malicious HTTP traffic (see Figure A-3). It provides a richly featured shell environment that allows the user to interactively explore individual conversations to find redirections, file objects, and malicious content. It also provides a few handy features for interacting with the data you uncover, including the ability to extract gzipped data and submit file hashes to VirusTotal. You can download CapTipper at *https://www.github.com/omriher/CapTipper/*.

Figure A-3: Analyzing an HTTP-based malware delivery with CapTipper

ngrep

If you are familiar with Linux, you've no doubt used grep to search data. ngrep is similar and allows you to perform very specific searches of packet capture data. I mostly use ngrep when capture and display filters won't do the job or get too wildly complex. You can read more about ngrep at *http://ngrep.sourceforge.net/*.

libpcap

If you plan to do any advanced packet parsing or create applications that deal with packets, you'll become very familiar with libpcap. Simply put, libpcap is a portable C/C++ library for network traffic capture. Wireshark, tcpdump, and most other packet analysis applications rely on the libpcap library at some level. You can read more about libpcap at *http://www.tcpdump.org/*.

Npcap

Npcap is the Nmap Project's packet-sniffing library for Windows that is based on WinPcap/libpcap. It is reported to deliver performance increases when capturing packets, and it provides extra security features related to restricting packet capture to administrators and leveraging Windows User Account control. Npcap can be installed as an alternative to WinPCap and used with Wireshark. You can learn more about it here: *https://www.github.com/nmap/npcap/*.

hping

hping is one of the more versatile tools to have in your arsenal. hping is a command line packet-crafting, -editing, and -transmission tool. It supports a variety of protocols and is very quick and intuitive to use. You can download hping from *http://www.hping.org/*.

Python

Python isn't a tool but rather a scripting language that is well worth mentioning. As you become proficient in packet analysis, you'll encounter cases in which no automated tool exists to meet your needs. In those cases, Python is the language of choice for making tools that can do interesting things with packets. You'll also need to know a little Python to interact with the Scapy library. My favorite online resource for learning Python is the popular *Learn Python the Hard Way* series, which can be found here: *https://www.learnpythonthehardway.org/*.

Packet Analysis Resources

From Wireshark's home page to courses and blogs, many resources for packet analysis are available. I'll list a few of my favorites here.

Wireshark's Home Page

The foremost resource for everything related to Wireshark is its home page, *http://www.wireshark.org/*. It has links to software documentation, a very helpful wiki that contains sample capture files, and sign-up information for the Wireshark mailing list. You can also browse to *https://ask.wireshark.org/* to ask questions about things you're seeing in Wireshark or specific features. This community is active and very helpful.

Practical Packet Analysis Online Course

If you like this book, you might also like the online training course that complements it. In the Practical Packet Analysis course, you'll be able to follow along with videos as I go through all the captures in this book and several others. I also provide capture labs where you can test your skills and a discussion forum where you can learn from other students as you progress. This course launches in mid-2017. You can learn more about my training offerings at *http://www.chrissanders.org/training/* and sign up for my mailing list to get notified about training opportunities here: *http://www.chrissanders.org/list/*.

SANS's Security Intrusion Detection In-Depth Course

SANS SEC503: Intrusion Detection In-Depth focuses on the security aspects of packet analysis. Even if you aren't focused on security, the first two days of the course provide a fantastic introduction to packet analysis and tcpdump. It is offered at live events several times a year at locations around the world.

You can read more about SEC503 and other SANS Institute courses at *http://www.sans.org/*.

Chris Sanders's Blog

I occasionally write articles related to packet analysis and post them on my blog at *http://www.chrissanders.org/*. My blog also serves as a portal that links to other articles and books I have written and provides my contact information. You'll also find links to packet captures included in this book and others.

Brad Duncan's Malware Traffic Analysis

My favorite resource for security-related packet captures is Brad Duncan's Malware Traffic Analysis (MTA) site. Brad posts packet captures containing real infection chains multiple times per week. These captures are complete with the associated malware binaries and a description of what is happening. If you want to gain experience dissecting malware infections and learn about current malware techniques, start by downloading some of these captures and trying to make sense of them. You can visit MTA at *http://www.malware-traffic-analysis.net/* or follow Brad on Twitter at @malware_traffic to be alerted when he posts updates.

IANA's Website

The Internet Assigned Numbers Authority (IANA), available at *http://www.iana.org/*, oversees the allocation of IP addresses and protocol number assignments for North America. Its website offers some valuable reference tools, such as the ability to look up port numbers, view information related to top-level domain names, and browse companion sites to find and view RFCs.

W. Richard Stevens's TCP/IP Illustrated Series

Considered the TCP/IP bible by most, W. Richard Stevens's *TCP/IP Illustrated* series (Addison-Wesley, 1994–1996) is a staple on the bookshelves of most who live at the packet level. These are my favorite TCP/IP books, and I consulted these volumes quite a bit while writing this book. A second edition of Volume 1, coauthored with Dr. Keven R. Fall, was published in 2012.

The TCP/IP Guide

The TCP/IP Guide by Charles Kozierok (No Starch Press, 2005) is another reference resource for TCP/IP protocol information. Weighing in at over 1,600 pages, it's very detailed and contains many great diagrams for the visual learner.

B

NAVIGATING PACKETS

In this appendix, we'll examine ways that packets can be represented. We'll look at fully interpreted and hexadecimal representations of packets, as well as how to read and reference packet values using a packet diagram.

Because you'll find a wealth of software that can interpret packet data for you, you could perform packet sniffing and analysis without understanding the information contained in this appendix. But, if you take the time to learn about packet data and how it's structured, you'll be in a much better position to understand what tools like Wireshark are showing you. The less abstraction between you and the data you're analyzing, the better.

Packet Representation

There are many ways a packet can be represented for interpretation. Raw packet data can be represented as binary, a combination of 1s and 0s in base 2, like this:

```
01100000010100110101110000001010110000010000000000001000000000000001000110000010
11010101101110000000000000000000000000000000000000000000000010000001000000000001011
01101000000000000010000001100000000000000000000000001000000010000000000010000000010
```

Binary numbers represent digital information at the lowest level possible, with a 1 representing the presence of an electrical signal and a 0 representing the absence of a signal. Each digit is a bit, and eight bits is a byte. However, binary data is difficult for humans to read and interpret, so we usually convert binary data to hexadecimal, a combination of letters and numbers in base 16. The same packet in hexadecimal looks like this:

```
4500 0034 40f2 4000 8006 535c ac10 1080
4a7d 5f68 0646 0050 7c23 5ab7 0000 0000
8002 2000 0b30 0000 0204 05b4 0103 0302
0101 0402
```

Hexadecimal (also referred to as hex) is a numbering system that uses the numbers 0 through 9 and letters A through F to represent values. It is one of the most common ways that packets are represented because it's concise and can easily be converted to the even more fundamental binary interpretation. In hex, two characters represent a byte, which contains eight bits. Each character within a byte is a *nibble* (4 bits), with the leftmost value being the *higher-order nibble* and the rightmost value being the *lower-order nibble*. Using the example packet, this means that the first byte is 45, the higher-order nibble is 4, and the lower-order nibble is 5.

The position of bytes within a packet is represented using offset notation, starting from zero. Therefore, the first byte in the packet (45) is at position 0x00, the second byte (00) is at 0x01, and the third byte (00) is at 0x02, and so on. The 0x part is saying that hex notation is being used. When referencing a position spanning more than one byte, the number of additional bytes is indicated numerically after a colon. For example, to reference the position of the first four bytes in the example packet (4500 0034), you would use 0x00:4. This explanation will be important when we use packet diagrams to dissect unknown protocols in "Navigating a Mystery Packet" on page 330.

NOTE *The most common mistake I see people make when trying to dissect packets is forgetting to start counting from zero. This is very hard to get used to, since most people are taught to start counting from one. I've been slicing and dicing packets for years, and I still make this mistake. The best advice I can give here is* don't be afraid to count on your fingers. *You might feel like it looks dumb, but there's absolutely no shame in it, especially if it helps you arrive at the correct answer.*

At a much higher level, a tool like Wireshark can represent a packet in a fully interpreted manner by using a protocol dissector, which we'll discuss next. The same packet we just looked at is shown in Figure B-1, fully interpreted by Wireshark.

```
1 0.000000 172.16.16.128 74.125.95.104 TCP 66 1606→80 [SYN] Seq=0 Win=8192 Len=0 MSS=1460 WS=4 SACK_PERM=1
⊞ Frame 1: 66 bytes on wire (528 bits), 66 bytes captured (528 bits)
⊞ Ethernet II, Src: IntelCor_5b:7d:4a (00:21:6a:5b:7d:4a), Dst: D-Link_21:99:4c (00:05:5d:21:99:4c)
⊞ Internet Protocol Version 4, Src: 172.16.16.128 (172.16.16.128), Dst: 74.125.95.104 (74.125.95.104)
⊟ Transmission Control Protocol, Src Port: 1606 (1606), Dst Port: 80 (80), Seq: 0, Len: 0
    Source Port: 1606 (1606)
    Destination Port: 80 (80)
    [Stream index: 0]
    [TCP Segment Len: 0]
    Sequence number: 0    (relative sequence number)
    Acknowledgment number: 0
    Header Length: 32 bytes
  ⊞ .... 0000 0000 0010 = Flags: 0x002 (SYN)
    Window size value: 8192
    [Calculated window size: 8192]
  ⊞ Checksum: 0x0b30 [validation disabled]
    Urgent pointer: 0
  ⊞ Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), SACK permitted

0000  00 05 5d 21 99 4c 00 21  6a 5b 7d 4a 08 00 45 00   ..]!.L.! j[}J..E.
0010  00 34 40 f2 40 00 80 06  53 5c ac 10 10 80 4a 7d   .4@.@... S\....J}
0020  5f 68 06 46 00 50 7c 23  5a b7 00 00 00 00 80 02   _h.F.P|# Z.......
0030  20 00 0b 30 00 00 02 04  05 b4 01 03 03 02 01 01    ..0.... ........
0040  04 02                                              ..
```

Figure B-1: A packet interpreted by Wireshark

Wireshark shows the information in a packet with labels that describe it. Packets don't contain labels, but their data does map to a precise format specified by the protocol standard. Fully interpreting a packet means reading the data based on the protocol standard and dissecting it into labeled, human-friendly text.

Wireshark and similar tools are able to fully interpret packet data because they have protocol dissectors built into them that define the position, length, and values of each field within a protocol. For example, the packet in Figure B-1 is broken into sections based on the Transmission Control Protocol (TCP). Within TCP, there are labeled fields and values. Source Port is one label, and 1606 is its decimal value. This makes it easy to find the information you're looking for when performing analysis. Whenever this option is available to you, it's usually the most efficient way to get the job done.

Wireshark has thousands of dissectors, but you might encounter protocols that Wireshark doesn't know how to interpret. This is often the case with vendor-specific protocols that aren't widely used and custom malware protocols. When this happens, you'll be left with only partially interpreted packets. This is why Wireshark provides the raw hexadecimal packet data at the bottom of the screen by default (see Figure B-1).

More commonly, command line programs like tcpdump that show raw hex don't have nearly as many dissectors. This is especially true for more complex application-layer protocols, which are trickier to parse. Thus, encountering partially interpreted packets is the norm when using this tool. An example of using tcpdump is shown in Figure B-2.

When you are working with partially interpreted packets, you'll have to rely on knowledge of packet structure at a more fundamental level. Wireshark, tcpdump, and most other tools enable this by showing the raw packet data in hex format.

```
1. bash
bash    bash
04:45:53.927963 IP 192.168.110.131.2074 > 192.168.110.138.502: Flags [P.], seq 1104341702:1104341714, ack 37762
64910, win 64710, length 12
        0x0000:  4500 0034 8bfd 4000 8006 1068 c0a8 6e83
        0x0010:  c0a8 6e8a 081a 01f6 41d2 eac6 e115 3ace
        0x0020:  5018 fcc6 0032 0000 00d1 0000 0006 0103
        0x0030:  0001 0001
```

Figure B-2: Partially interpreted packets from tcpdump

Using Packet Diagrams

As we learned in Chapter 1, a packet represents data that is formatted based on the rules of protocols. Because common protocols format packet data in a specific manner so that hardware and software can interpret this data, the packets must follow explicit formatting rules. We can identify this formatting and use it to interpret packet data by using packet diagrams. A *packet diagram* is a graphical representation of a packet that allows an analyst to map bytes within a packet to fields used by any given protocol. Derived from the protocol's RFC specification document, it shows the fields present within the protocol, their length, and their order.

Let's take another look at the example packet diagram for IPv4 we saw in Chapter 7 (provided here for your convenience as Figure B-3).

Internet Protocol Version 4 (IPv4)							
Offsets	Octet	0		1	2		3
Octet	Bit	0–3	4–7	8–15	16–18	19–23	24–31
0	0	Version	Header Length	Type of Service	Total Length		
4	32	Identification			Flags	Fragment Offset	
8	64	Time to Live		Protocol	Header Checksum		
12	96	Source IP Address					
16	128	Destination IP Address					
20	160	Options					
24+	192+	Data					

Figure B-3: A packet diagram for IPv4

In this diagram, the horizontal axis represents individual binary bits that are numbered from 0 to 31. The bits are grouped into 8-bit bytes that are numbered from 0 to 3. The vertical axis also is labeled according to bits and bytes, and each row is divided into 32-bit (or 4-byte) sections. We use the axes to count field positions using offset notation by first reading from the vertical axis to determine which 4-byte section the field resides in, and then counting off each byte in the section using the horizontal axis. The first row consists of the first four bytes, 0 through 3, which are labeled accordingly on the horizontal axis. The second row consists of the next four

bytes, 4 through 7, which can also be counted off using the horizontal axis. Here we start with byte 4, which is byte 0 on the horizontal axis, then byte 5, which corresponds to byte 1 on the horizontal axis, and so on.

For example, we can determine that for IPv4, byte 0x01 is the Type of Service field, since we start at offset 0 and then count to byte 1. On the vertical axis, the first four bytes are in the first row, so we would then use the horizontal axis and start counting from 0 to byte 1. As another example, byte 0x08 is the Time to Live field. Using the vertical axis, we determine that byte 8 is in the third row down, which contains bytes 8 through 11. We then use the horizontal axis to count to byte 8 starting from 0. Since byte 8 is the first in the section, the horizontal axis column is just 0, which is the Time to Live field.

Some fields, such as the Source IP field, span multiple bytes, as we see in 0x12:4. Other fields are divided into nibbles. An example is 0x00, which contains the Version field in the higher-order nibble and the IP Header Length in the lower-order nibble. Byte 0x06 contains even more granularity, with individual bits used to represent specific fields. When a field is a single binary value, it is often referred to as a *flag*. Examples are the Reserved, Don't Fragment, and More Fragments fields in the IPv4 header. A flag can only have a binary value of 1 (true) or 0 (false), so the flag is "set" when the value is 1. The exact implication of a flag setting will vary based on protocol and field.

Let's look at another example in Figure B-4 (you may recognize this diagram from Chapter 8).

Transmission Control Protocol (TCP)						
Offsets	Octet	0		1	2	3
Octet	Bit	0–3	4–7	8–15	16–23	24–31
0	0	Source Port			Destination Port	
4	32	Sequence Number				
8	64	Acknowledgment Number				
12	96	Data Offset	Reserved	Flags	Window Size	
16	128	Checksum			Urgent Pointer	
20+	160+	Options				

Figure B-4: A packet diagram for the TCP

This image shows the TCP header. Looking at this image, we can answer a lot of questions about a TCP packet without knowing exactly what TCP does. Consider an example TCP packet header represented in hex here:

```
0646 0050 7c23 5ab7 0000 0000 8002 2000
0b30 0000 0204 05b4 0103 0302 0101 0402
```

Using the packet diagram, we can locate and interpret specific fields. For example, we can determine the following:

- The Source Port number is at 0x00:2 and has a hex value of 0646 (Decimal: 1606).
- The Destination Port number is at 0x02:2 and has a hex value of 0050 (Decimal: 80).
- The header length is in the Data Offset field at the higher-order nibble of 0x12 and has a hex value of 8.

Let's apply this knowledge by dissecting a mystery packet.

Navigating a Mystery Packet

In Figure B-2, I showed you a packet that was only partially interpreted. You can ascertain through the interpreted portion of the data that this is a TCP/IP packet transmitted between two devices on the same network, but other than that, you don't know much about the data being transmitted. Here's the complete hex output of the packet:

```
4500 0034 8bfd 4000 8006 1068 c0a8 6e83
c0a8 6e8a 081a 01f6 41d2 eac6 e115 3ace
5018 fcc6 0032 0000 00d1 0000 0006 0103
0001 0001
```

A quick count finds that there are 52 bytes in this packet. The packet diagram for IP tells us that the normal size of the IP header is 20 bytes, which is confirmed by looking at the header size value in the lower-order nibble of 0x00. The diagram for the TCP header tells us that it is also 20 bytes if no additional options are present (there aren't here, but we discuss TCP options in more depth in Chapter 8). This means that the first 40 bytes of this output are related to the TCP and IP data that has already been interpreted. This leaves the remaining 12 bytes uninterpreted.

```
00d1 0000 0006 0103 0001 0001
```

Without knowledge of how to navigate packets, this might leave you stumped, but you now know how to apply a packet diagram to the uninterpreted bytes. In this case, the interpreted TCP data tells us that the destination port for this data is 502. Reviewing the ports used by traffic isn't a foolproof method for identifying uninterpreted bytes, but it's a good place to start. A quick Google search reveals that port 502 is most commonly used for Modbus over TCP, which is a protocol used in Industrial Control System (ICS) networks. We can validate this is the case and navigate this packet by comparing the hex output to the packet diagram for Modbus, shown in Figure B-5.

Modbus over TCP					
Offsets	Octet	0	1	2	3
Octet	Bit	0–7	8–15	16–23	24–31
0	0	Transaction Identifier		Protocol Identifier	
4	32	Length		Unit Identifier	Function Code
8+	64+	Variable			

Figure B-5: Packet diagram for Modbus over TCP

This packet diagram was created based on the information in the Modbus implementation guide: *http://www.modbus.org/docs/Modbus_Messaging_Implementation_Guide_V1_0b.pdf*. This tells us that there should be a 7-byte header that includes the Length field at 0x04:2 (relative to the start of the header). Counting to that position, we arrive at a hex value of 0006 (or a decimal value of 6), indicating there should be 6 bytes following that field, which is exactly the case. It appears that this is indeed Modbus over TCP data.

By comparing the packet diagram to the entirety of the hex output, the following information is derived:

- The Transaction Identifier is at 0x00:2 and has a hex value of 00d1. This field is used to pair a request with a response.
- The Protocol Identifier is at 0x02:2 and has a hex value of 0000. This identifies the protocol as Modbus.
- The Length is at 0x04:2 and has a hex value of 0006. This defines the length of the packet data.
- The Unit Identifier is at 0x06 and has a hex value of 01. This is used for intrasystem routing.
- The Function Code is at 0x07 and has a hex value of 03. This is the Read Holding Registers function, which reads a data value from a system.
- Based on the function code value of 3, two more data fields are expected. The Reference Number and Word Count are found at 0x08:4, and each has a hex value of 0001.

The mystery packet can now be fully explained in the context of the Modbus protocol. If you were troubleshooting the system responsible for this packet, this information should be all you need to proceed onward. Even if you never encounter Modbus, this is an example of how you can approach an unknown protocol and uninterpreted packet using a packet diagram.

It's always best practice to be aware of the abstraction between yourself and the data being analyzed. This helps you make sounder and more knowledgeable decisions and allows you to work with packets in a variety of situations. I've found myself in many scenarios in which I've only been

able to use command line–based tools such as tcpdump to analyze packets. Because most of these tools lack dissection for many layer 7 protocols, the ability to manually dissect specific bytes in these packets has been crucial.

NOTE *A colleague once had to help perform incident response in a highly secure environment. He was cleared to review the data he needed to look at, but not to access the specific system the data was stored on. The only thing they could do in the amount of time they had was print out the packets from specific conversations. Thanks to his fundamental knowledge of how packets are built and of how to navigate them, he was able to find the information he needed in the printed data. Of course, the process was slower than cold molasses running down a frozen branch. This is an extreme scenario, but it's a prime example of why universal tool-agnostic knowledge is important.*

For all of these reasons, it's helpful to spend time breaking apart packets in order to gain experience viewing multiple interpretations. I do this enough that I've printed out several common packet diagrams, had them laminated, and keep them beside my desk. I also maintain a digital version on my laptop and tablet for quick reference when traveling. For convenience, I've included several common packet diagrams in the ZIP file containing the packet captures that goes along with this book (*https://www.nostarch.com/packetanalysis3/*).

Final Thoughts

In this appendix, we learned how to interpret packet data in a variety of formats and how to use packet diagrams to navigate uninterpreted packet data. Given this fundamental knowledge, you should have no trouble understanding how to dissect packets regardless of the tool you are using to view packet data.

INDEX

Numerals

6to4 (IPv6 over IPv4) protocol, 143
48-port Ethernet switches, 12
802.11 packet structure
    beacon packets, 304–305
    control packets, 304
    data packets, 304
    management packets, 304

A

AA (Authoritative Answers) field (DNS packets), 173
absolute timestamps, 114
accessibility and connectivity problems
    gateway configuration, 210–213
    troubleshooting, 222–226
    unwanted redirection, 213–216
    upstream problems, 216–219
Acknowledgment Number field (TCP headers), 152
ACK packets
    client latency and, 250
    DHCP initialization process, 169–170
    duplicate, 236–239, 247
    server latency and, 250
    TCP retransmissions, 232–233
    TCP teardown, 158–159
    wire latency and, 249
active fingerprinting, 266
Additional Information Section field (DNS packets), 174
Additional Records Count field (DNS packets), 174
Address Resolution Protocol. *See* ARP
Ad hoc wireless card mode, 298, 299
Advanced preferences (Wireshark), 47
AfriNIC, 82
aggregated network taps, 25
AirPcap
    capturing traffic with, 302
    configuring, 300–301
ALFA network, 300
American Registry for Internet Numbers (ARIN), 81, 82
analysis step (in packet-sniffing process), 4
Answer Count field (DNS packets), 174
Answers Section field (DNS packets), 174
APNIC, 82
Appearance preferences (Wireshark), 46
application baseline, 254–255
Application layer (OSI model), 5
application protocols, 2
ARIN (American Registry for Internet Numbers), 81, 82
ARP (Address Resolution Protocol), 4, 120–121
    cache poisoning, 27–28, 34
        Cain & Abel security tool, 28–31
        traffic manipulation, 267–271
    gratuitous ARP, 124–125
    packet structure, 121–122
    requests, 122–123
    responses, 123, 269
ASCII output, 111
ASN (Autonomous System Number), 207
associations/dependencies, 254
asymmetric routing, 31
attachments, sending via SMTP, 196–198
authentication sequences
    host baseline, 254
    site baseline, 253
Authoritative Answers (AA) field (DNS packets), 173
Authority Section field (DNS packets), 174
Autonomous System Number (ASN), 207
AXFR (full zone transfer), 181

B

basic service set identifier (BSS ID), 307
BE (big-endian) format, 146
beacon packets, 304–305
Berkeley Packet Filter (BPF) syntax, 66–67
big-endian (BE) format, 146
binary system, 326
Blink Led option (AirPcap), 300
Boot File field (DHCP packets), 165
BPF (Berkeley Packet Filter) syntax, 66–67
BPF capture filters, 113–114
broadcast domains, 15–16

broadcast packets, 15–16
broadcast traffic, 15–16
host baseline, 253
site baseline, 252
BSS ID (basic service set identifier), 307

C

C2 (command and control) behavior (malware), 290–291
cache poisoning (ARP spoofing), 27–28, 34
Cain & Abel security tool, 28–31
traffic manipulation, 267–271
Cain & Abel security tool, 28–31, 319
CAM (Content Addressable Memory) tables, 12, 120
CAPSCREEN command, 284–285
CapTipper tool, 320
capture files, 53
merging, 55–56
saving and exporting, 54–55
capture filters, 65–66
addressing, 67–68
BPF, 66–67, 113–114
command line tools and, 113
commonly used, 70–71
hostname, 67–68
port filters, 68
protocol field filters, 69–70
protocol filters, 69
Capture Interfaces dialog (Wireshark), 61–65
Input tab, 61–62
Options tab, 63
Display Options section, 64
Name Resolution section, 64
Stop capture section, 65
Output tab, 62–63
capture packets, 56–58
Capture Interfaces dialog (Wireshark), 61–65
command line tools and, 106–108
finding packets, 56–57
marking packets, 57
packet time referencing, 60
printing packets, 58
time display formats, 59
time shifting, 60–61
Capture preferences (Wireshark), 47
Capture Type options (AirPcap), 301
-c argument (command line tools), 108
CDNs (content delivery networks), 201
Chanalyzer software, 298
Channel column (Wireshark Packet List pane), 306
channel hopping technique, 297
Channel option (AirPcap), 300
channels
defined, 296
signal interference, 297–298
sniffing, 296–297
Chat messages (Wireshark), 100, 101
Checksum field
ICMP headers, 144
TCP headers, 152
UDP headers, 161
Cisco port mirroring commands, 22
Classless Inter-Domain Routing (CIDR) notation, 127
Client Hardware Address field (DHCP packets), 165
Client Identifier option (DHCP discover packets), 167
Client IP Address field (DHCP packets), 165
client latency, 249–250
CloudShark tool, 317–318
Code field (ICMP headers), 144
collection step (in packet-sniffing process), 3
Coloring Rules window (Wireshark), 48–49
Combs, Gerald, 37
command and control (C2) behavior (malware), 290–291
command line tools, 103–104
tcpdump
capturing and saving packets, 106–108
compared to TShark, 118
filters, 113–114
installing, 105
manipulating output, 109–111
name resolution, 111–112
TShark
capturing and saving packets, 106–108
compared to tcpdump, 118
filters, 113–114
installing, 104
manipulating output, 109–111
name resolution, 111–112
summary statistics, 115–118
time display formats, 114–115
comma-separated value (CSV) files, 226–230
comparison operators (Wireshark filter expression), 73
computer communication
network hardware, 10–14
hubs, 10–11
routers, 13–14
switches, 11–12
network traffic, 15–16
broadcast, 15–16
multicast, 16
unicast, 16

OSI model, 5–10
data encapsulation, 8–10
data flow through, 7
protocols, 4–5
configuration files (Wireshark), 50
Configuration Profiles window (Wireshark), 50–52
connectionless protocols, 160–161. *See also* UDP
connection-oriented protocols, 151. *See also* TCP
connectivity problems. *See* accessibility and connectivity problems
Content Addressable Memory (CAM) tables, 12, 120
content delivery networks (CDNs), 201
control packets, 304
conversations
defined, 78
identifying top talkers with, 80–83
viewing, 79–80
Conversations window (Wireshark)
finding open ports with, 262
TCP communications, 260
conversion step (in packet-sniffing process), 4
Cookie Manager plugin, 273
cost
of packet sniffers, 3
of Wireshark, 38
CryptoLocker malware, 291
CryptoWall malware, 289–290, 294
CSV (comma-separated value) files, 226–230
CyberEYE, 281

D

Damn Vulnerable Web Application (DVWA), 272
`-D` argument (command line tools), 107
data encapsulation (OSI model), 7–10
Data field (IPv4 headers), 127
Data link layer (OSI model), 6
data packets, 304
Data Rate column (Wireshark Packet List pane), 306
data representation (packets), 326–328
data transfer
application baseline, 255
site baseline, 253
through OSI model, 7
Decode As... dialog (Wireshark), 89–90
denial-of-service (DoS) attacks, 28, 31
Destination IP Address field
IPv4 headers, 127
IPv6 headers, 136
destination port (TCP), 152–153
Destination Port field
TCP headers, 152
UDP headers, 161
DHCP (Dynamic Host Configuration Protocol), 4, 163–164
DHCPv6, 171–173
initialization process, 165
acknowledgment packets, 169–170
discover packets, 165–167
offer packets, 167–168
request packets, 168–169
in-lease renewal, 170
message types, 171
options, 170–171
packet structure, 164–165
DHCPv6, 171–173
direct install method, 33, 34
discarding (dropping) packets, 11
discover packets (DHCP initialization process), 165–167
Display filter option (Wireshark), 57
display filters, 71–74, 113
DNS (Domain Name System)
packet structure, 173–174
query, 174–176
question types, 176–177
recursion, 177–180
zone transfers, 181–183
DORA process (DHCP), 165
acknowledgment packets, 169–170
discover packets, 165–167
offer packets, 167–168
request packets, 168–169
DoS (denial-of-service) attacks, 28, 31
dotted-quad (dotted-decimal) notation (IPv4 addresses), 125
double-headed packets (ICMP), 148
dropping (discarding) packets, 11
Duncan, Brad, 322
duplicate acknowledgments (TCP), 235–240
duplicate ACK packets, 236–239, 247
DVWA (Damn Vulnerable Web Application), 272
Dynamic Host Configuration Protocol. *See* DHCP

E

EAPoL (Extensible Authentication Protocol over LAN) packets, 314
echo request packets (ICMP), 145–147
email. *See* SMTP
encapsulation, 8–10, 143
endpoints
defined, 78
identifying top talkers with, 80–83
viewing statistics, 78–79

Enterasys router, 13, 22
enterprise-grade fiber optic taps, 27
ephemeral port group, 153
Error messages (Wireshark), 100, 101
error-recovery features (TCP)
    duplicate acknowledgments, 235–240
    retransmissions, 232–235
ESMTP (Extended SMTP), 190
ESSID (extended service set ID), 303
Ethereal application, 37–38
Ethernet taps, 27
expert information (Wireshark), 99–101
    Chat messages, 100, 101
    Error messages, 100, 101
    Note messages, 100, 101
    Warning messages, 100, 101
exploit kits, 294
exporting
    capture files, 54–55
    endpoint address into colorization rule, 79
Export Specified Packets dialog (Wireshark), 55
expressions, 66. *See also* capture filters
extended service set ID (ESSID), 303
Extended SMTP (ESMTP), 190
Extensible Authentication Protocol over LAN (EAPoL) packets, 314
Extension Channel option (AirPcap), 301

F

fail-open capability (network taps), 27
`-f` argument (command line tools), 113
fast retransmission packets, 236–237
FCS Filter option (AirPcap), 301
FIFO (first in, first out) method, 63
file carving process, 287
file formats (Wireshark), 54
file sets, 63
Filter Expressions preferences (Wireshark), 47
filters
    adding to toolbar, 75–76
    capture filters, 65–71
    display filters, 71–74, 113
    saving, 74–75
    tcpdump, 113–114
    TShark, 113–114
    wireless-specific, 307–308
        filtering specific frequency, 308
        filtering specific wireless packet types, 307–308
        filtering traffic for BSS ID, 307
finding packets, 56–57
Find Packet bar (Wireshark), 56–57
first in, first out (FIFO) method, 63
flags, 329
Flags field
    DHCP packets, 164
    IPv4 headers, 127
    TCP headers, 152
flow control features (TCP), 240
    adjusting window size, 241–242
    sliding window, 243–247
    zero window notification, 242–243
flow graphing, 99
Flow Label field (IPv6 headers), 136
footprinting
    operating system fingerprinting, 263–266
        active fingerprinting, 266
        passive fingerprinting, 263–266
    SYN scan, 258–262
        identifying open and closed ports, 262
        using filters with, 260–262
forced decodes (Wireshark), 88–90
fragmentation (IPv6), 141–143
Fragment Offset field (IPv4 headers), 127
full-duplex devices (switches), 11–12
full zone transfer (AXFR), 181

G

gateway configuration problems, 210–213
    analysis, 210–212
    conclusions, 212–213
    tapping into the wire, 210
Gateway IP Address field (DHCP packets), 165
`GET` requests (HTTP), 184, 187, 208–209, 249–250, 275–276, 280
GNU Public License (GPL), 37, 38
graphing
    flow, 99
    IO graphs, 95–98
    round-trip time, 98–99
gratuitous ARP, 124–125

H

half-duplex mode (hubs), 10
half-open scan (SYN scan), 258–262
    identifying open and closed ports, 262
    using filters with, 260–262
handshakes (TCP), 155–158
Hardware Address Length field (ARP headers), 121
Hardware Length field (DHCP packets), 164
Hardware Type field
    ARP headers, 121
    DHCP packets, 164

-h argument (command line tools), 104
Header Checksum field (IPv4 headers), 127
Header Length field (IPv4 headers), 127
help resources, 321–323
  Brad Duncan's Malware Traffic Analysis site, 322
  Chris Sanders's Blog, 322
  Internet Assigned Numbers Authority, 323
  Practical Packet Analysis online course, 322
  SANS SEC503: Intrusion Detection In-Depth course, 322
  The TCP/IP Guide, 323
  Wireshark's Home Page, 322
  W. Richard Stevens's *TCP/IP Illustrated* series, 323
hexadecimal system, 111, 326
Hex value option (Wireshark), 57
higher-order nibble, 326
high latency
  defined, 232
  locating source of, 248–251
Hop Limit field (IPv6 headers), 136
Hops field (DHCP packets), 164
host baseline, 253–254
hostname (capture filters), 67–68
host portion (interface identifier), 126, 134
hosts file, 86–87, 215–216
Host to Host encapsulation, 143
Host to Router encapsulation, 143
hping tool, 321
HTTP (Hypertext Transfer Protocol), 8–9
  browsing with, 183–186
  GET requests, 184, 187, 208–209, 249–250, 275–276, 280
  posting data with, 186–187
  <script> tags, 278
  session hijacking attacks, 271–275
   tags, 279
  streams, 91
hubbing out, 23–24, 33
hub networks, 19–20
hubs, 10–11, 24
Hypertext Transfer Protocol. *See* HTTP

I

IANA (Internet Assigned Numbers Authority), 323
-i argument (command line tools), 107
ICMP (Internet Control Message Protocol), 144–150
  ICMPv6, 150
  packet structure, 144
  requests and responses, 145–147
  traceroute utility, 147–150
  types and messages, 144
ICMP protocol dissector, 88
ICMP version 6 (ICMPv6), 150
ICS (Industrial Control System) networks, 330
Identification field (IPv4 headers), 127
IDS (intrusion-detection system), 258
IETF (Internet Engineering Task Force), 120
ifconfig command, 107
<iframe> tags, 279
IMAP (Internet Message Access Protocol), 195
Include 802.11 FCS in Frames option (AirPcap), 301
incremental zone transfer (IXFR), 181
Industrial Control System (ICS) networks, 330
infrastructure problems, 222
  analysis, 223–226
  conclusions, 226
  tapping into the wire, 223
initialization process (DHCP), 165
  acknowledgment packets, 169–170
  discover packets, 165–167
  offer packets, 167–168
  request packets, 168–169
initial sequence number (ISN), 236
in-lease renewal (DHCP), 170
installing
  tcpdump, 105
  TShark, 104
  Wireshark
    on Linux systems, 41–43
    on OS X systems, 43
    on Windows systems, 39–41
interface identifier (host portion), 126, 134
Interface option (AirPcap), 300
internet accessibility and connectivity problems
  gateway configuration problems, 210–213
  troubleshooting, 222–226
  unwanted redirection, 213–216
  upstream problems, 216–219
Internet Assigned Numbers Authority (IANA), 323
Internet Control Message Protocol. *See* ICMP
Internet Engineering Task Force (IETF), 120
Internet Message Access Protocol (IMAP), 195
Internet of Things (IoT) devices, 210
Internet Protocol addresses. *See* IP addresses
Internet Society (ISOC), 120
intrusion-detection system (IDS), 258

IO graphs, 95–98
IoT (Internet of Things) devices, 210
IP addresses
determining ownership of, 82
hostnames, 67–68
IPv4, 125–127
IPv6, 133–135
IPv4 (Internet Protocol version 4), 4, 125–132, 328–329
addresses, 125–127
packet fragmentation, 130–132
packet structure, 127–128
Time to Live value, 128–130
IPv6 (Internet Protocol version 6), 4, 133–143
addresses, 133–135
fragmentation, 141–143
neighbor solicitation, 138–141
packet structure, 135–138
transitional protocols, 143
IPv6 over IPv4 (6to4) protocol, 143
ISATAP protocol, 143
ISN (initial sequence number), 236
ISOC (Internet Society), 120
`iwconfig` command (Linux), 303–304
IXFR (incremental zone transfer), 181

K

keep-alive packets, 242, 246, 247
Kismet tool, 297
kit's landing page, 294
Kozierok, Charles, 323

L

LANs (local area networks), 125
latency, 231
locating source of high
client latency, 249–250
latency locating framework, 251
normal communications, 248
server latency, 250–251
wire latency, 248–249
network baselining, 251–255
application baseline, 254–255
host baseline, 253–254
site baseline, 252–253
TCP error-recovery features, 232–240
TCP flow control, 240–247
troubleshooting, 247
layer 7 (upper-layer) protocols
DHCP, 163–173
DHCPv6, 171–173
initialization process, 165–170
in-lease renewal, 170
message types, 171
options, 170–171
packet structure, 164–165
DNS, 173–183
packet structure, 173–174
query, 174–176
question types, 176–177
recursion, 177–180
zone transfers, 181–183
HTTP, 183–187
browsing with, 183–186
posting data with, 186–187
SMTP, 187–198
sending and receiving email, 188–189
sending attachments via, 196–198
tracking email messages, 189–196
LE (little-endian) format, 146
libpcap tool, 321
link-local addresses (IPv6), 134
Linux
`ifconfig` command, 107
installing Wireshark, 41–43
compiling from source, 42–43
DEB-based systems, 42
RPM-based systems, 41
sniffing wirelessly, 303–304
little-endian (LE) format, 146
local area networks (LANs), 125
logical operators, 67, 74
lower-order nibble, 326
low latency, 232

M

MAC (Media Access Control) addresses, 11, 134
MAC Address Scanner dialog box (Cain & Abel tool), 29
mail delivery agent (MDA), 189
mail submission agent (MSA), 189
mail transfer agent (MTA), 188–189
mail user agent (MUA), 188
main window (Wireshark), 45–46
malware
Brad Duncan's Malware Traffic Analysis site, 322
Operation Aurora, 275–281
remote-access trojans, 281–288
Malware Traffic Analysis (MTA) site, 322
managed switches, 12
Managed wireless card mode, 298, 299
management packets, 304
man-in-the-middle (MITM) attacks, 267, 270
marking packets, 57
Master wireless card mode, 298, 299

Maximum concurrent requests option (Wireshark), 85
maximum transmission unit (MTU), 130
MDA (mail delivery agent), 189
Media Access Control (MAC) addresses, 11, 134
merging capture files, 55–56
messages
  ICMP, 144
  Wireshark, 100–101
Message Type options (DHCP discover packets), 167, 171
message types (DHCP), 171
MetaGeek spectrum analyzer, 297–298
missing web content, 200–205
  analysis, 201–204
  conclusions, 204–205
  tapping into the wire, 200
MITM (man-in-the-middle) attacks, 267, 270
Modbus protocol, 330–331
Monitor (RFMON) wireless card mode, 298, 299
More fragments offset value (IP fragmentation), 131–132
MSA (mail submission agent), 189
MTA (mail transfer agent), 188–189
MTA (Malware Traffic Analysis) site, 322
MTU (maximum transmission unit), 130
MTU discovery process, 141–142
MUA (mail user agent), 188
multicast traffic, 16
mystery packets, 330–332

N

name resolution (name lookup)
  enabling, 84–86
  hosts file, 86–87
  manually initiated, 88
  potential drawbacks to, 86
  tcpdump, 111–112
  TShark, 111–112
Name Resolution preferences (Wireshark), 47, 84–88
Name Server (Authority) Record Count field (DNS packets), 174
`-n` argument (command line tools), 112
`-N` argument (command line tools), 112
navigating packets, 325
  data representation, 326–328
  mystery packets, 330–332
  packet diagrams, 328–330
Neighbor Advertisement packets, 140
Neighbor Discovery Protocol (NDP), 139
neighbor solicitation (IPv6), 138–141
NETGEAR hubs, 10
netmask (network mask), 126
network baselining, 251
  application baseline, 254–255
  host baseline, 253–254
  site baseline, 252–253
network diagrams (network maps), 33
network hardware, 10–14
  hubs, 10–11
  routers, 13–14
  switches, 11–12
network interface cards (NICs), 18–19, 298–299
Network layer (OSI model), 6
network layer protocols, 2, 119
  ARP, 120–121
    gratuitous ARP, 124–125
    packet structure, 121–122
    requests, 122–123
    responses, 123
  ICMP, 144–150
    ICMPv6, 150
    packet structure, 144
    requests and responses, 145–147
    traceroute utility, 147–150
    types and messages, 144
  IP, 125–143
    IPv4, 125–132
    IPv6, 133–143
network maps (network diagrams), 33
network mask (netmask), 126
NetworkMiner tool, 319–320
network portion (network prefix), 126, 134
network taps, 24–27, 34
Network Time Protocol (NTP), 60
network traffic, 15–16
  broadcast, 15–16, 252–253
  multicast, 16
  unicast, 16
Next Header field (IPv6 headers), 136
ngrep tool, 321
nibbles, 326
NICs (network interface cards), 18–19, 298–299
Nmap tool, 260, 266
nonaggregated network taps, 25–27
Nortel, 22
Note messages (Wireshark), 100, 101
Npcap tool, 321
NTP (Network Time Protocol), 60

O

offer packets (DHCP initialization process), 167–168
OmniPeek, 2
Only use the profile “hosts” file option (Wireshark), 86

OpCode field
DHCP packets, 164
DNS packets, 173
Open Systems Interconnections model. *See* OSI model
operating system fingerprinting
active fingerprinting, 266
passive fingerprinting, 263–266
operating system support
command line tools, 118
Wireshark, 39
Operation Aurora, 275–281
Operation field (ARP headers), 121
Options field
DHCP packets, 165
IPv4 headers, 127
TCP headers, 152
OSI (Open Systems Interconnections) model, 5
data encapsulation, 7–10
data flow through, 7
OS X
`ifconfig` command, 107
installing Wireshark on, 43
output (of command line tools), 109–111

P

packet analysis, 1–2
computer communication, 4–14
network hardware, 10–14
network traffic, 15–16
OSI model, 5–10
protocols, 4–5
packet sniffers, 2–4
Packet Bytes pane (Wireshark), 45, 46, 111
packet capture, 44–45
packet color coding, 48–49
Packet Details pane (Wireshark), 45, 46, 238
packet diagrams, 328–330
packet fragmentation (IPv4), 130–132
Packet Length field (UDP headers), 161
packet lengths, 93–94
Packet List pane (Wireshark), 45, 46
adding wireless-specific columns to, 305–306
retransmission packets, 239
packet sniffers, 2–4. *See also* sniffer placement
evaluating, 2–3
packet-sniffing process, 3–4
analysis, 4
collection, 3
conversion, 4
packet structure
802.11, 304–305
ARP, 121–122
DHCP, 164–165
DNS, 173–174
ICMP, 144
IPv4, 127–128
IPv6, 135–138
TCP, 152
UDP, 161
packet time referencing, 60
packet transcript, 91
Parameter Request List option (DHCP discover packets), 167
passive fingerprinting, 263–266
Payload Length field (IPv6 headers), 136
.pcapng file format, 54
PDUs (protocol data units), 8
Physical layer (OSI model), 6
physical transmission medium (wireless packet analysis), 296–298
signal interference, 297–298
sniffing channels, 296–297
ping utility, 145–146
POP3 (Post Office Protocol version 3), 195
port filters, 68
port mirroring (port spanning), 21–22, 33
ports
SYN scan, 258–262
TCP, 152–155
posting data with HTTP, 186–187
Post Office Protocol version 3 (POP3), 195
Practical Packet Analysis online course, 322
preferences management (Wireshark), 46–47, 84–86
Presentation layer (OSI model), 5
primitives, 66
printing
capture packets, 58
printer problems, 219–222
program support (Wireshark), 39
promiscuous mode, 3, 18–19
Protocol Address Length field (ARP headers), 121
protocol data units (PDUs), 8
protocol dissectors, 88–91, 327
changing, 88–90
viewing source code, 90–91
Protocol field (IPv4 headers), 127
protocol field filters, 69–70
protocol filters, 69
Protocol Hierarchy Statistics window (Wireshark), 83–84
application baseline, 254
host baseline, 253
site baseline, 252
protocols, 4–5. *See also names of specific protocols*
application, 2
Modbus, 330–331

OSI model, 6–7
Wireshark, 38
Protocols preferences (Wireshark), 47
protocol stack, 4
protocol support (command line tools), 118
Protocol Type field (ARP headers), 121
Python tool, 321

Q

QR (Query/Response) field (DNS packets), 173
qualifiers (BPF syntax), 66
queries (DNS), 174–176
Query/Response (QR) field (DNS packets), 173
Question Count field (DNS packets), 174
Questions Section field (DNS packets), 174
question types (DNS), 176–177

R

RA (Recursion Available) field (DNS packets), 173
ransomware, 288–294
-r argument (command line tools), 108
RAT (remote-access trojans), 281–288
RCode (Response Code) field (DNS packets), 174
RD (Recursion Desired) field (DNS packets), 173
receive window (server), 241–243
reconnaissance. *See* footprinting
recursion (DNS), 177–180
Recursion Available (RA) field (DNS packets), 173
Recursion Desired (RD) field (DNS packets), 173
relative timestamps, 114
remote-access trojans (RAT), 281–288
repeating devices (hubs), 10
Requested IP Address option (DHCP discover packets), 167
request packets
ARP, 122–123, 269
DHCP initialization process, 168–169
ICMP, 145–147
Reserved (Z) field (DNS packets), 174
resets (TCP), 159–160
Resolve MAC addresses option (Wireshark), 84
Resolve network (IP) addresses option (Wireshark), 85
Resolve transport names option (Wireshark), 85
Response Code (RCode) field (DNS packets), 174
response packets
ARP, 123, 269
ICMP, 145–147
retransmission packets, 247
retransmissions (TCP), 232–235
retransmission timeout (RTO), 232–235
retransmission timer, 232
RFMON (Monitor) wireless card mode, 298, 299
ring buffer, 63
RIPE, 82
RJ-45 ports (hubs), 10
Robtex, 82
round-trip time (RTT), 98–99, 145, 232
routed environment, 31–32
routers, 13–14
Router to Router encapsulation, 143
RST packets, 159–160
RTO (retransmission timeout), 232–235
RTT (round-trip time), 98–99, 145, 232

S

SANS SEC503: Intrusion Detection In-Depth course, 322
SARR process (DHCPv6), 172–173
saving
capture files, 54–55
filters, 74–75
packets, 106–108
wireless profile, 309
scanning techniques, 258–262
Scapy tool, 319
<script> tags (HTML), 278
Seconds Elapsed field (DHCP packets), 164
Secure Socket Layer (SSL) protocol, 88
security, 257–258
exploit kit, 294
footprinting
operating system fingerprinting, 263–266
SYN scan, 258–262
malware
Operation Aurora, 275–281
remote-access trojans, 281–288
ransomware, 288–294
traffic manipulation, 266
ARP cache poisoning, 267–271
session hijacking, 271–275
wireless
WEP authentication, 309–312
WPA authentication, 312–314
Sender Hardware Address field (ARP headers), 122
Sender Protocol Address field (ARP headers), 122
Sequence Number field (TCP headers), 152

Server Host Name field (DHCP packets), 165
Server IP Address field (DHCP packets), 165
server latency, 250–251
session hijacking, 271–275
Session layer (OSI model), 5
Sguil tool, 288–289
signal interference, 297–298
Signal Strength column (Wireshark Packet List pane), 306
Simple Mail Transfer Protocol. *See* SMTP
site baseline, 252–253
sliding-window mechanism (TCP), 240–241, 243–247. *See also* flow control features
slow networks, 231
  locating source of high latency
    client latency, 249–250
    latency locating framework, 251
    normal communications, 248
    server latency, 250–251
    wire latency, 248–249
  network baselining, 251
    application baseline, 254–255
    host baseline, 253–254
    site baseline, 252–253
  TCP error-recovery features, 232–240
  TCP flow control, 240–247
  troubleshooting, 247
small office and home office (SOHO) switches, 22
SMTP (Simple Mail Transfer Protocol), 187–198
  sending and receiving email, 188–189
  sending attachments via, 196–198
  tracking email messages, 189–196
sniffer placement, 17. *See also* tapping into the wire
  determining best method, 35
  hub network, 19–20
  promiscuous mode, 18–19
  routed environment, 31–32
  switched environment, 20–31, 33–34
    ARP cache poisoning, 27–31, 34
    direct install method, 33, 34
    hubbing out, 23–24, 33
    network taps, 24–27, 34
    port mirroring, 21–22, 33
Sniffer tab (Cain & Abel tool), 28–29
sniffing wirelessly
  in Linux, 303–304
  sniffing channels, 296–297
  in Windows, 300–302
Snort alert, 281–282
software data corruption, 226–230
  analysis, 226–230
  conclusions, 230
  tapping into the wire, 226
SOHO (small office and home office) switches, 22
source code access
  packet sniffers, 3
  Wireshark, 39
Source IP Address field
  IPv4 headers, 127
  IPv6 headers, 136
source port (TCP), 152–153
Source Port field
  TCP headers, 152
  UDP headers, 161
 tags (HTML), 279
Spanning Tree Protocol (STP), 84
spear phishing, 275
spectrum analyzer, 297–298
SSL (Secure Socket Layer) protocol, 88
SSL streams
  defined, 91
  following, 92–93
standard port (system port group), 153
startup/shutdown
  application baseline, 254
  host baseline, 253
Statistics preferences (Wireshark), 47
stealth scan (SYN scan), 258–262
  identifying open and closed ports, 262
  using filters with, 260–262
Stevens, W. Richard, 323
STP (Spanning Tree Protocol), 84
stream following, 91–93
String option (Wireshark), 57
subnet mask (network mask), 126
summary statistics (TShark), 115–118
switched environment, sniffing in, 20–31, 33–34
  ARP cache poisoning, 27–31, 34
  direct install method, 33, 34
  hubbing out, 23–24, 33
  network taps, 24–27, 34
  port mirroring, 21–22, 33
switches, 11–12
SYN scan, 258–262
  identifying open and closed ports, 262
  using filters with, 260–262
system port group (standard port; well-known port group), 153

T

tapping into the wire, 17. *See also* sniffer placement
  determining best method, 35
  gateway configuration problems, 210
  hub network, 19–20
  infrastructure problems, 223
  missing web content, 200

printer problems, 219
promiscuous mode, 18–19
routed environment, 31–32
software data corruption, 226
switched environment, 20–31, 33–34
ARP cache poisoning, 27–31, 34
direct install method, 33, 34
hubbing out, 23–24, 33
network taps, 24–27, 34
port mirroring, 21–22, 33
unresponsive weather service, 206
unwanted redirection, 213
upstream problems, 216
Target Hardware Address field (ARP headers), 122
Target Protocol Address field (ARP headers), 122
-t argument (command line tools), 114
TC (Truncation) field (DNS packets), 173
TCP (Transmission Control Protocol), 4, 109, 151–160
buffer space, 240–241
error-recovery features
duplicate acknowledgments, 235–240
retransmissions, 232–235
flow control features, 240
adjusting window size, 241–242
sliding window, 243–247
zero window notification, 242–243
handshakes, 155–158
HTTP protocol and, 9
packet structure, 152
ports, 152–155
resets, 159–160
streams, 91
teardown process, 158–159
tcpdump, 2
capturing and saving packets, 106–108
compared to TShark, 118
filters, 113–114
installing, 105
manipulating output, 109–111
name resolution, 111–112
The TCP/IP Guide (Kozierok), 323
TCP/IP Illustrated series (Stevens), 323
Tcpreplay tool, 319
TCP SYN scan (SYN scan), 258–262
identifying open and closed ports, 262
using filters with, 260–262
teardown process (TCP), 158–159
Teredo protocol, 143
Terminal-based Wireshark. *See* TShark
time display formats
capture packets, 59
TShark, 114–115
time shifting (capture packets), 60–61
Time to Live value. *See* TTL value
TKIP encryption, 314
TLS (Transport Layer Security) protocol, 195
toolbar (Wireshark), adding filters to, 75–76
tools, 317–321. *See also* command line tools
Cain & Abel security tool, 319
CapTipper, 320
CloudShark, 317–318
hping, 321
libpcap, 321
NetworkMiner, 319–320
ngrep, 321
Npcap, 321
Python, 321
Scapy, 319
Tcpreplay, 319
TraceWrangler, 319
WireEdit, 318
top talkers, 80–83
Total Length field (IPv4 headers), 127
traceroute utility (ICMP), 147–150
TraceWrangler tool, 319
Traffic Class field (IPv6 headers), 135
traffic manipulation, 266–275. *See also* network traffic
ARP cache poisoning, 267–271
session hijacking, 271–275
Transaction ID field (DHCP packets), 164
transitional protocols (IPv6), 143
Transmission Control Protocol. *See* TCP
Transport layer (OSI model), 5–6
transport layer protocols. *See also* TCP
choosing sniffer, 2
UDP, 160–161
Transport Layer Security (TLS) protocol, 195
troubleshooting
inconsistent printers, 219–222
internet accessibility problems, 210–219
missing web content, 200–205
no branch office connectivity, 222–226
slow networks, 247
software data corruption, 226–230
unresponsive weather service, 205–210
Truncation (TC) field (DNS packets), 173
TShark (Terminal-based Wireshark)
capturing and saving packets, 106–108
compared to tcpdump, 118
filters, 113–114
installing, 104
manipulating output, 109–111
name resolution, 111–112
summary statistics, 115–118
time display formats, 114–115
TTL (Time to Live) value
ICMP packets, 148–149
IPv4, 128–130

Type field (ICMP headers), 144
Type of Service field (IPv4 headers), 127

U

Ubuntu 14.04 LTS, 105
UDP (User Datagram Protocol), 109, 160–161
    latency and, 251
    packet structure, 161
    streams, 91
unicast packet, 16
Unix systems
    capturing and saving packets, 106–108
    filters, 113–114
    installing, 105
    manipulating output, 109–111
    name resolution, 111–112
unwanted redirection, 213–216
    analysis, 213–216
    conclusions, 216
    tapping into the wire, 213
upper-layer protocols. *See* layer 7 protocols
upstream problems, 216–219
    analysis, 216–219
    conclusions, 219
    tapping into the wire, 216
Urgent Pointer field (TCP headers), 152
USBPcap, 40–41
Use an external network name resolver option (Wireshark), 85
Use captured DNS packet data for address resolution option (Wireshark), 85
User Datagram Protocol. *See* UDP

V

`-V` argument (command line tools), 109–110
`-v` argument (command line tools), 110
Variable field (ICMP headers), 144
Version field
    IPv4 headers, 127
    IPv6 headers, 135
visibility window
    defined, 19
    on switched network, 21

W

WAN (wide area network) link, 222–223
WAPs (wireless access points)
    802.11 packet structure, 304–305
    basic service set identifier, 307
    filtering traffic for BSS ID, 307
    WEP authentication, 309–312
    wireless NIC modes, 298
    WPA authentication, 312–314
`-w` argument (command line tools), 107–108
Warning messages (Wireshark), 100, 101
weather service, unresponsive, 205–210
    analysis, 206–209
    conclusions, 209–210
    tapping into the wire, 206
well-known port group (system port group), 153
WEP (Wired Equivalent Privacy) authentication, 309–312
WEP Configuration option (AirPcap), 301
WHOIS query, 207
WHOIS registry, 81, 82
wide area network (WAN) link, 222–223
Wi-Fi Protected Access. *See* WPA
Windows
    installing Wireshark, 39–41
    sniffing wirelessly, 300–302
    TShark
        capturing and saving packets, 106–108
        compared to tcpdump, 118
        filters, 113–114
        installing, 104
        manipulating output, 109–111
        name resolution, 111–112
        summary statistics, 115–118
        time display formats, 114–115
    WinDump, 105
Window Size field (TCP headers), 152
WinPcap capture driver, 39
Wired Equivalent Privacy (WEP) authentication, 309–312
WireEdit tool, 318
wire latency, 248–249
wireless access points. *See* WAPs
wireless card modes
    Ad hoc mode, 298, 299
    Managed mode, 298, 299
    Master mode, 298, 299
    Monitor mode, 298, 299
Wireless LAN Statistics window (Wireshark), 302
wireless local area networks (WLANs), 296. *See also* wireless packet analysis
wireless packet analysis
    802.11 packet structure, 304–305
    adding wireless-specific columns to Packet List pane, 305–306
    filters, 307–308
    physical considerations
        signal interference, 297–298
        sniffing channels, 296–297
    saving wireless profile, 309
    security, 309–315

sniffing wirelessly
in Linux, 303–304
in Windows, 300–302
wireless card modes, 298–299
wireless sniffing
in Linux, 303–304
sniffing channels, 296–297
in Windows, 300–302
Wireshark, 2, 37
advanced features, 77–101
big-endian format, 146
configuration files, 50
configuration profiles, 50–52
conversations, 78–83, 260, 262
cost, 38
endpoints, 78–83
expert information, 99–101
graphing, 95–99
Home Page, 322
installing, 39–43
on Linux systems, 41–43
on OS X systems, 43
on Windows systems, 39–41
main window, 45–46
name resolution, 84–88
operating system support, 39
Packet Bytes pane, 45, 46, 111
packet capture, 44–45
packet color coding, 48–49
Packet Details pane, 45, 46, 238
packet lengths, 93–94
Packet List pane, 45, 46, 239, 305–306
preferences, 46–47
program support, 39
protocol dissectors, 88–91
protocol hierarchy statistics, 83–84, 252, 253, 254
source code access, 39
stream following, 91–93
supported protocols, 38
user-friendliness, 38
Wi-Spy spectrum analyzer, 298
WLANs (wireless local area networks), 296. *See also* wireless packet analysis
WPA (Wi-Fi Protected Access), 309, 312–314

X

-x argument (command line tools), 110

Y

-Y argument (command line tools), 113
Your IP Address field (DHCP packets), 165

Z

Z (Reserved) field (DNS packets), 174
-z argument (command line tools), 115, 118
zero window packet (TCP), 242–245, 247

The Electronic Frontier Foundation (EFF) is the leading organization defending civil liberties in the digital world. We defend free speech on the Internet, fight illegal surveillance, promote the rights of innovators to develop new digital technologies, and work to ensure that the rights and freedoms we enjoy are enhanced — rather than eroded — as our use of technology grows.

EFF.ORG

ELECTRONIC FRONTIER FOUNDATION

Protecting Rights and Promoting Freedom on the Electronic Frontier

DON'T JUST STARE AT CAPTURED PACKETS. ANALYZE THEM.

Download the capture files used in this book from *nostarch.com/packetanalysis3/*

It's easy to capture packets with Wireshark, the world's most popular network sniffer, whether off the wire or from the air. But how do you use those packets to understand what's happening on your network?

Updated to cover Wireshark 2.*x*, the third edition of *Practical Packet Analysis* will teach you to make sense of your packet captures so that you can better troubleshoot network problems. You'll find added coverage of IPv6 and SMTP, a new chapter on the powerful command line packet analyzers tcpdump and TShark, and an appendix on how to read and reference packet values using a packet map.

Practical Packet Analysis will show you how to:

- Monitor your network in real time and tap live network communications
- Build customized capture and display filters
- Use packet analysis to troubleshoot and resolve common network problems, like loss of connectivity, DNS issues, and slow speeds
- Explore modern exploits and malware at the packet level
- Extract files sent across a network from packet captures
- Graph traffic patterns to visualize the data flowing across your network
- Use advanced Wireshark features to understand confusing captures
- Build statistics and reports to help you better explain technical network information to non-techies

No matter what your level of experience is, *Practical Packet Analysis* will show you how to use Wireshark to make sense of any network and get things done.

ABOUT THE AUTHOR

Chris Sanders is a computer security consultant, researcher, and educator. He is the author of *Applied Network Security Monitoring* and blogs regularly at *ChrisSanders.org*. Chris uses packet analysis daily to catch bad guys and find evil.

The author's royalties from this book will be donated to the Rural Technology Fund (*http://ruraltechfund.org/*).

COVERS WIRESHARK 2.*X*

no starch press

THE FINEST IN GEEK ENTERTAINMENT™

www.nostarch.com

"I LIE FLAT."

This book uses a durable binding that won't snap shut.

$49.95 *($57.95 CDN)*

ISBN: 978-1-59327-802-1

54995

9 781593 278021

6 89145 78020 8

SHELVE IN: NETWORKING/SECURITY