Attacking Network Protocols

A Hacker's Guide to Capture, Analysis, and Exploitation

James Forshaw

Foreword by Katie Moussouris

no starch press

ATTACKING NETWORK PROTOCOLS

ATTACKING NETWORK PROTOCOLS

A Hacker's Guide to Capture, Analysis, and Exploitation

by James Forshaw

About the Author

James Forshaw is a renowned computer security researcher at Google Project Zero, with more than ten years of experience in analyzing and exploiting application network protocols. His skills range from cracking game consoles to exposing complex design issues in operating systems, especially Microsoft Windows, which earned him the top bug bounty of $100,000 and placed him as the #1 researcher on Microsoft Security Response Center's (MSRC) published list. He's the creator of the network protocol analysis tool, Canape, which was developed from his years of experience. He's been invited to present his novel security research at global security conferences such as BlackHat, CanSecWest and Chaos Computer Congress.

About the Technical Reviewer

Since the early days of Commodore PET and VIC-20, technology has been a constant companion (and sometimes an obsession!) to Cliff Janzen. Cliff discovered his career passion when he moved to information security in 2008 after a decade of IT operations. Since then, Cliff has had the great fortune to work with and learn from some of the best people in the industry, including Mr. Forshaw and the fine people at No Starch during the production of this book. He is happily employed as a security consultant, doing everything from policy review to penetration tests. He feels lucky to have a career that is also his favorite hobby and a wife who supports him.

BRIEF CONTENTS

CONTENTS IN DETAIL

3
NETWORK PROTOCOL STRUCTURES 37

4
ADVANCED APPLICATION TRAFFIC CAPTURE 63

5
ANALYSIS FROM THE WIRE 79

6
APPLICATION REVERSE ENGINEERING   111

NETWORK PROTOCOL ANALYSIS TOOLKIT   277

INDEX   293

FOREWORD

When I first met James Forshaw, I worked in what
Popular Science described in 2007 as one of the
top ten worst jobs in science: a "Microsoft Security
Grunt." This was the broad-swath label the magazine
used for anyone working in the Microsoft Security
Response Center (MSRC). What positioned our jobs
as worse than "whale-feces researcher" but somehow better than "elephant
vasectomist" on this list (so famous among those of us who suffered in
Redmond, WA, that we made t-shirts) was the relentless drumbeat of
incoming security bug reports in Microsoft products.

It was here in MSRC that James, with his keen and creative eye toward
the uncommon and overlooked, first caught my attention as a security
strategist. James was the author of some of the most interesting security
bug reports. This was no small feat, considering the MSRC was receiving
upwards of 200,000 security bug reports per year from security researchers.
James was finding not only simple bugs—he had taken a look at the .NET

framework and found architecture-level issues. While these architecture-level bugs were harder to address in a simple patch, they were much more valuable to Microsoft and its customers.

Fast-forward to the creation of Microsoft's first bug bounty programs, which I started at the company in June of 2013. We had three programs in that initial batch of bug bounties—programs that promised to pay security researchers like James cash in exchange for reporting the most serious bugs to Microsoft. I knew that for these programs to prove their efficacy, we needed high-quality security bugs to be turned in.

If we built it, there was no guarantee that the bug finders would come. We knew we were competing for some of the most highly skilled bug hunting eyes in the world. Numerous other cash rewards were available, and not all of the bug markets were for defense. Nation-states and criminals had a well-established offense market for bugs and exploits, and Microsoft was relying on the finders who were already coming forward at the rate of 200,000 bug reports per year for free. The bounties were to focus the attention of those friendly, altruistic bug hunters on the problems Microsoft needed the most help with eradicating.

So of course, I called on James and a handful of others, because I was counting on them to deliver the buggy goods. For these first Microsoft bug bounties, we security grunts in the MSRC really wanted vulnerabilities for Internet Explorer (IE) 11 beta, and we wanted something no software vendor had ever tried to set a bug bounty on before: we wanted to know about new exploitation techniques. That latter bounty was known as the Mitigation Bypass Bounty, and worth $100,000 at the time.

I remember sitting with James over a beer in London, trying to get him excited about looking for IE bugs, when he explained that he'd never looked at browser security much before and cautioned me not to expect much from him.

James nevertheless turned in four unique sandbox escapes for IE 11 beta. *Four.*

These sandbox escapes were in areas of the IE code that our internal teams and private external penetration testers had all missed. Sandbox escapes are essential to helping other bugs be more reliably exploitable. James earned bounties for all four bugs, paid for by the IE team itself, plus an extra $5,000 bonus out of my bounty budget. Looking back, I probably should have given him an extra $50,000. Because wow. Not bad for a bug hunter who had never looked at web browser security before.

Just a few months later, I was calling James on the phone from outside a Microsoft cafeteria on a brisk autumn day, absolutely breathless, to tell him that he had just made history. This particular Microsoft Security Grunt couldn't have been more thrilled to deliver the news that his entry for one of the other Microsoft bug bounty programs—the Mitigation Bypass Bounty for $100,000—had been accepted. James Forshaw had found a unique new way to bypass all the platform defenses using architecture-level flaws in the latest operating system and won the very first $100,000 bounty from Microsoft.

On that phone call, as I recall the conversation, he said he pictured me handing him a comically-huge novelty check onstage at Microsoft's internal BlueHat conference. I sent the marketing department a note after that call, and in an instant, "James and the Giant Check" became part of Microsoft and internet history forever.
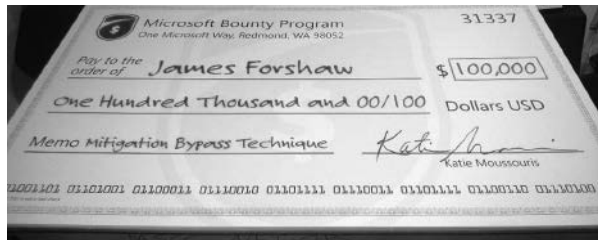
What I am certain readers will gain in the following pages of this book are pieces of James's unparalleled brilliance—the same brilliance that I saw arching across a bug report or four so many years ago. There are precious few security researchers who can find bugs in one advanced technology, and fewer still who can find them in more than one with any consistency. Then there are people like James Forshaw, who can focus on deeper architecture issues with a surgeon's precision. I hope that those reading this book, and any future book by James, treat it like a practical guide to spark that same brilliance and creativity in their own work.

In a bug bounty meeting at Microsoft, when the IE team members were shaking their heads, wondering how they could have missed some of the bugs James reported, I stated simply, "James can see the Lady in the Red Dress, as well as the code that rendered her, in the Matrix." All of those around the table accepted this explanation for the kind of mind at work in James. He could bend any spoon; and by studying his work, if you have an open mind, then so might you.

For all the bug finders in the world, here is your bar, and it is high. For all the untold numbers of security grunts in the world, may all your bug reports be as interesting and valuable as those supplied by the one and only James Forshaw.

Katie Moussouris
Founder and CEO, Luta Security
October 2017

ACKNOWLEDGMENTS

I'd like to thank you for reading my book; I hope you find it enlightening and of practical use. I'm grateful for the contributions from many different people.

I must start by thanking my lovely wife Huayi, who made sure I stuck to writing even if I really didn't want to. Through her encouragement, I finished it in only four years; without her maybe it could have been written in two, but it wouldn't have been as much fun.

Of course, I definitely wouldn't be here today without my amazing parents. Their love and encouragement has led me to become a widely recognized computer security researcher and published author. They bought the family a computer—an Atari 400—when I was young, and they were instrumental in starting my interest in computers and software development. I can't thank them enough for giving me all my opportunities.

Acting as a great counterpoint to my computer nerdiness was my oldest friend, Sam Shearon. Always the more confident and outgoing person and an incredible artist, he made me see a different side to life.

Throughout my career, there have been many colleagues and friends who have made major contributions to my achievements. I must highlight

Richard Neal, a good friend and sometimes line manager who gave me the opportunity to find an interest in computer security, a skill set that suited my mindset.

I also can't forget Mike Jordon who convinced me to start working at Context Information Security in the UK. Along with owners Alex Church and Mark Raeburn, they gave me the time to do impactful security research, build my skills in network protocol analysis, and develop tools such as Canape. This experience of attacking real-world, and typically completely bespoke, network protocols is what much of the content of this book is based on.

I must thank Katie Moussouris for convincing me to go for the Microsoft Mitigation Bypass Bounty, raising my profile massively in the information security world, and of course for giving me a giant novelty check for $100,000 for my troubles.

My increased profile didn't go amiss when the team for Google Project Zero—a group of world leading security researchers with the goal of making the platforms that we all rely on more secure—was being set up. Will Harris mentioned me to the current head of the team, Chris Evans, who convinced me to interview, and soon I was a Googler. Being a member of such an excellent team makes me proud.

Finally, I must thank Bill, Laurel, and Liz at No Starch Press for having the patience to wait for me to finish this book and for giving me solid advice on how to tackle it. I hope that they, and you, are happy with the final result.

INTRODUCTION

When first introduced, the technology that allowed devices to connect to a network was exclusive to large companies and governments. Today, most people carry a fully networked computing device in their pocket, and with the rise of the Internet of Things (IoT), you can add devices such as your fridge and our home's security system to this interconnected world. The security of these connected devices is therefore increasingly important. Although you might not be too concerned about someone disclosing the details of how many yogurts you buy, if your smartphone is compromised over the same network as your fridge, you could lose all your personal and financial information to a malicious attacker.

This book is named *Attacking Network Protocols* because to find security vulnerabilities in a network-connected device, you need to adopt the mind-set of the attacker who wants to exploit those weaknesses. Network protocols communicate with other devices on a network, and because these

protocols must be exposed to a public network and often don't undergo the same level of scrutiny as other components of a device, they're an obvious attack target.

Why Read This Book?

Many books discuss network traffic capture for the purposes of diagnostics and basic network analysis, but they don't focus on the security aspects of the protocols they capture. What makes this book different is that it focuses on analyzing custom protocols to find security vulnerabilities.

This book is for those who are interested in analyzing and attacking network protocols but don't know where to start. The chapters will guide you through learning techniques to capture network traffic, performing analysis of the protocols, and discovering and exploiting security vulnerabilities. The book provides background information on networking and network security, as well as practical examples of protocols to analyze.

Whether you want to attack network protocols to report security vulnerabilities to an application's vendor or just want to know how your latest IoT device communicates, you'll find several topics of interest.

What's in This Book?

This book contains a mix of theoretical and practical chapters. For the practical chapters, I've developed and made available a networking library called Canape Core, which you can use to build your own tools for protocol analysis and exploitation. I've also provided an example networked application called *SuperFunkyChat*, which implements a user-to-user chat protocol. By following the discussions in the chapters, you can use the example application to learn the skills of protocol analysis and attack the sample network protocols. Here is a brief breakdown of each chapter:

Chapter 1: The Basics of Networking
This chapter describes the basics of computer networking with a particular focus on TCP/IP, which forms the basis of application-level network protocols. Subsequent chapters assume that you have a good grasp of the network basics. This chapter also introduces the approach I use to model application protocols. The model breaks down the application protocol into flexible layers and abstracts complex technical detail, allowing you to focus on the bespoke parts of the protocol you're analyzing.

Chapter 2: Capturing Application Traffic
This chapter introduces the concepts of passive and active capture of network traffic, and it's the first chapter to use the Canape Core network libraries for practical tasks.

Chapter 3: Network Protocol Structures

This chapter contains details of the internal structures that are common across network protocols, such as the representation of numbers or human-readable text. When you're analyzing captured network traffic, you can use this knowledge to quickly identify common structures, speeding up your analysis.

Chapter 4: Advanced Application Traffic Capture

This chapter explores a number of more advanced capture techniques that complement the examples in Chapter 2. The advanced capture techniques include configuring Network Address Translation to redirect traffic of interest and spoofing the address resolution protocol.

Chapter 5: Analysis from the Wire

This chapter introduces methods for analyzing captured network traffic using the passive and active techniques described in Chapter 2. In this chapter, we begin using the *SuperFunkyChat* application to generate example traffic.

Chapter 6: Application Reverse Engineering

This chapter describes techniques for reverse engineering network-connected programs. Reverse engineering allows you to analyze a protocol without needing to capture example traffic. These methods also help to identify how custom encryption or obfuscation is implemented so you can better analyze traffic you've captured.

Chapter 7: Network Protocol Security

This chapter provides background information on techniques and cryptographic algorithms used to secure network protocols. Protecting the contents of network traffic from disclosure or tampering as it travels over public networks is of the utmost importance for network protocol security.

Chapter 8: Implementing the Network Protocol

This chapter explains techniques for implementing the application network protocol in your own code so you can test the protocol's behavior to find security weaknesses.

Chapter 9: The Root Causes of Vulnerabilities

This chapter describes common security vulnerabilities you'll encounter in a network protocol. When you understand the root causes of vulnerabilities, you can more easily identify them during analysis.

Chapter 10: Finding and Exploiting Security Vulnerabilities

This chapter describes processes for finding security vulnerabilities based on the root causes in Chapter 9 and demonstrates a number of ways of exploiting them, including developing your own shell code and bypassing exploit mitigations through return-oriented programming.

Appendix: Network Protocol Analysis Toolkit

In the appendix, you'll find descriptions of some of the tools I commonly use when performing network protocol analysis. Many of the tools are described briefly in the main body of the text as well.

How to Use This Book

If you want to start with a refresher on the basics of networking, read Chapter 1 first. When you're familiar with the basics, proceed to Chapters 2, 3, and 5 for practical experience in capturing network traffic and learning the network protocol analysis process.

With the knowledge of the principles of network traffic capture and analysis, you can then move on to Chapters 7 through 10 for practical information on how to find and exploit security vulnerabilities in these protocols. Chapters 4 and 6 contain more advanced information about additional capture techniques and application reverse engineering, so you can read them after you've read the other chapters if you prefer.

For the practical examples, you'll need to install .NET Core (*https://www.microsoft.com/net/core/*), which is a cross-platform version of the .NET runtime from Microsoft that works on Windows, Linux, and macOS. You can then download releases for Canape Core from *https://github.com/tyranid/CANAPE.Core/releases/* and *SuperFunkyChat* from *https://github.com/tyranid/ExampleChatApplication/releases/*; both use .NET Core as the runtime. Links to each site are available with the book's resources at *https://www.nostarch.com/networkprotocols/*.

To execute the example Canape Core scripts, you'll need to use the *CANAPE.Cli* application, which will be in the release package downloaded from the Canape Core Github repository. Execute the script with the following command line, replacing `script.csx` with the name of the script you want to execute.

```
dotnet exec CANAPE.Cli.dll script.csx
```

All example listings for the practical chapters as well as packet captures are available on the book's page at *https://www.nostarch.com/networkprotocols/*. It's best to download these example listings before you begin so you can follow the practical chapters without having to enter a large amount of source code manually.

Contact Me

I'm always interested in receiving feedback, both positive and negative, on my work, and this book is no exception. You can email me at *attacking.network .protocols@gmail.com*. You can also follow me on Twitter *@tiraniddo* or subscribe to my blog at *https://tyranidslair.blogspot.com/* where I post some of my latest advanced security research.

1

THE BASICS OF NETWORKING

To attack network protocols, you need to understand the basics of computer networking. The more you understand how common networks are built and function, the easier it will be to apply that knowledge to capturing, analyzing, and exploiting new protocols.

Throughout this chapter, I'll introduce basic network concepts you'll encounter every day when you're analyzing network protocols. I'll also lay the groundwork for a way to think about network protocols, making it easier to find previously unknown security issues during your analysis.

Network Architecture and Protocols

Let's start by reviewing some basic networking terminology and asking the fundamental question: what is a network? A *network* is a set of two or more computers connected together to share information. It's common to refer to each connected device as a *node* on the network to make the description applicable to a wider range of devices. Figure 1-1 shows a very simple example.
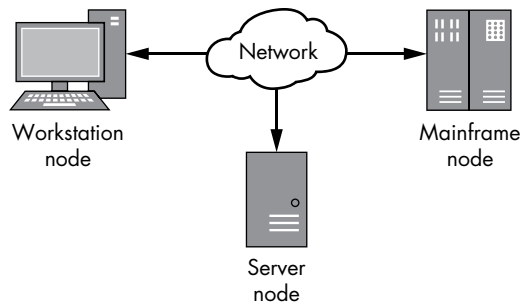
Figure 1-1: A simple network of three nodes

The figure shows three nodes connected with a common network. Each node might have a different operating system or hardware. But as long as each node follows a set of rules, or *network protocol*, it can communicate with the other nodes on the network. To communicate correctly, all nodes on a network must understand the same network protocol.

A network protocol serves many functions, including one or more of the following:

Maintaining session state   Protocols typically implement mechanisms to create new connections and terminate existing connections.

Identifying nodes through addressing   Data must be transmitted to the correct node on a network. Some protocols implement an addressing mechanism to identify specific nodes or groups of nodes.

Controlling flow   The amount of data transferred across a network is limited. Protocols can implement ways of managing data flow to increase throughput and reduce latency.

Guaranteeing the order of transmitted data   Many networks do not guarantee that the order in which the data is sent will match the order in which it's received. A protocol can reorder the data to ensure it's delivered in the correct order.

Detecting and correcting errors   Many networks are not 100 percent reliable; data can become corrupted. It's important to detect corruption and, ideally, correct it.

Formatting and encoding data   Data isn't always in a format suitable for transmitting on the network. A protocol can specify ways of encoding data, such as encoding English text into binary values.

The Internet Protocol Suite

TCP/IP is the de facto protocol that modern networks use. Although you can think of TCP/IP as a single protocol, it's actually a combination of two protocols: the *Transmission Control Protocol (TCP)* and the *Internet Protocol (IP)*. These

two protocols form part of the *Internet Protocol Suite (IPS)*, a conceptual model of how network protocols send network traffic over the internet that breaks down network communication into four layers, as shown in Figure 1-2.
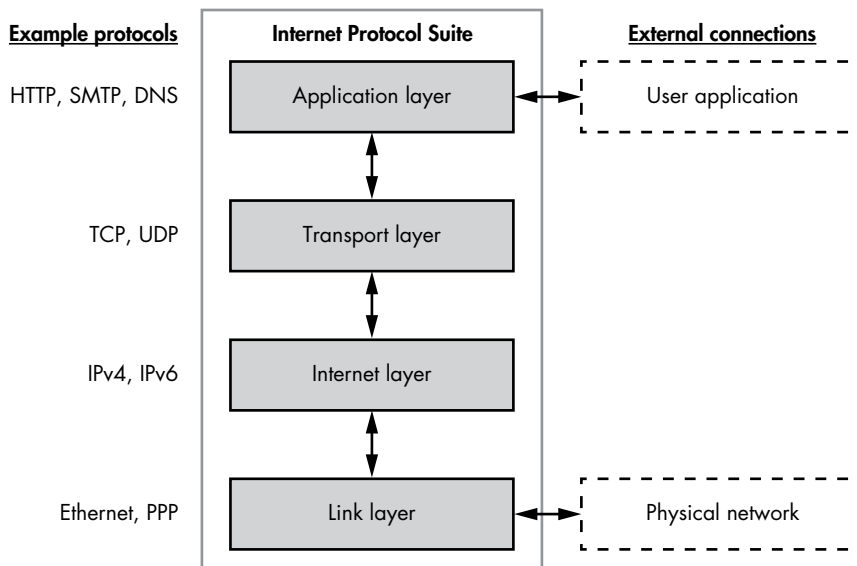
Figure 1-2: Internet Protocol Suite layers

These four layers form a *protocol stack*. The following list explains each layer of the IPS:

Link layer (layer 1)   This layer is the lowest level and describes the physical mechanisms used to transfer information between nodes on a local network. Well-known examples include Ethernet (both wired and wireless) and Point-to-Point Protocol (PPP).

Internet layer (layer 2)   This layer provides the mechanisms for addressing network nodes. Unlike in layer 1, the nodes don't have to be located on the local network. This level contains the IP; on modern networks, the actual protocol used could be either version 4 (IPv4) or version 6 (IPv6).

Transport layer (layer 3)   This layer is responsible for connections between clients and servers, sometimes ensuring the correct order of packets and providing service multiplexing. Service multiplexing allows a single node to support multiple different services by assigning a different number for each service; this number is called a *port*. TCP and the User Datagram Protocol (UDP) operate on this layer.

Application layer (layer 4)   This layer contains network protocols, such as the *HyperText Transport Protocol (HTTP)*, which transfers web page contents; the *Simple Mail Transport Protocol (SMTP)*, which transfers email; and the *Domain Name System (DNS) protocol*, which converts a name to a node on the network. Throughout this book, we'll focus primarily on this layer.

Each layer interacts only with the layer above and below it, but there must be some external interactions with the stack. Figure 1-2 shows two external connections. The link layer interacts with a physical network connection, transmitting data in a physical medium, such as pulses of electricity or light. The application layer interacts with the user application: an *application* is a collection of related functionality that provides a service to a user. Figure 1-3 shows an example of an application that processes email. The service provided by the mail application is the sending and receiving of messages over a network.
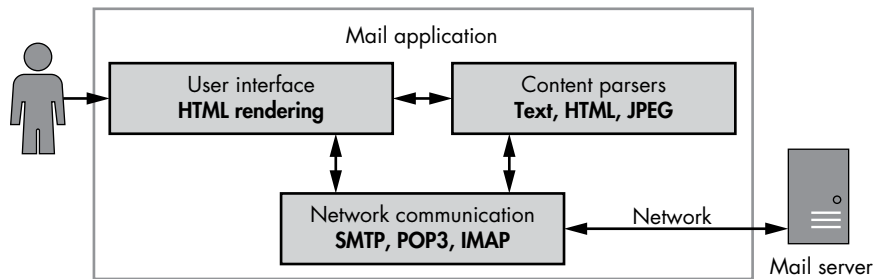
Figure 1-3: Example mail application

Typically, applications contain the following components:

Network communication   This component communicates over the network and processes incoming and outgoing data. For a mail application, the network communication is most likely a standard protocol, such as SMTP or POP3.

Content parsers   Data transferred over a network usually contains content that must be extracted and processed. Content might include textual data, such as the body of an email, or it might be pictures or video.

User interface (UI)   The UI allows the user to view received emails and to create new emails for transmission. In a mail application, the UI might display emails using HTML in a web browser.

Note that the user interacting with the UI doesn't have to be a human being. It could be another application that automates the sending and receiving of emails through a command line tool.

Data Encapsulation

Each layer in the IPS is built on the one below, and each layer is able to encapsulate the data from the layer above so it can move between the layers. Data transmitted by each layer is called a *protocol data unit (PDU)*.

Headers, Footers, and Addresses

The PDU in each layer contains the payload data that is being transmitted. It's common to prefix a *header*—which contains information required

for the payload data to be transmitted, such as the *addresses* of the source and destination nodes on the network—to the payload data. Sometimes a PDU also has a *footer* that is suffixed to the payload data and contains values needed to ensure correct transmission, such as error-checking information. Figure 1-4 shows how the PDUs are laid out in the IPS.
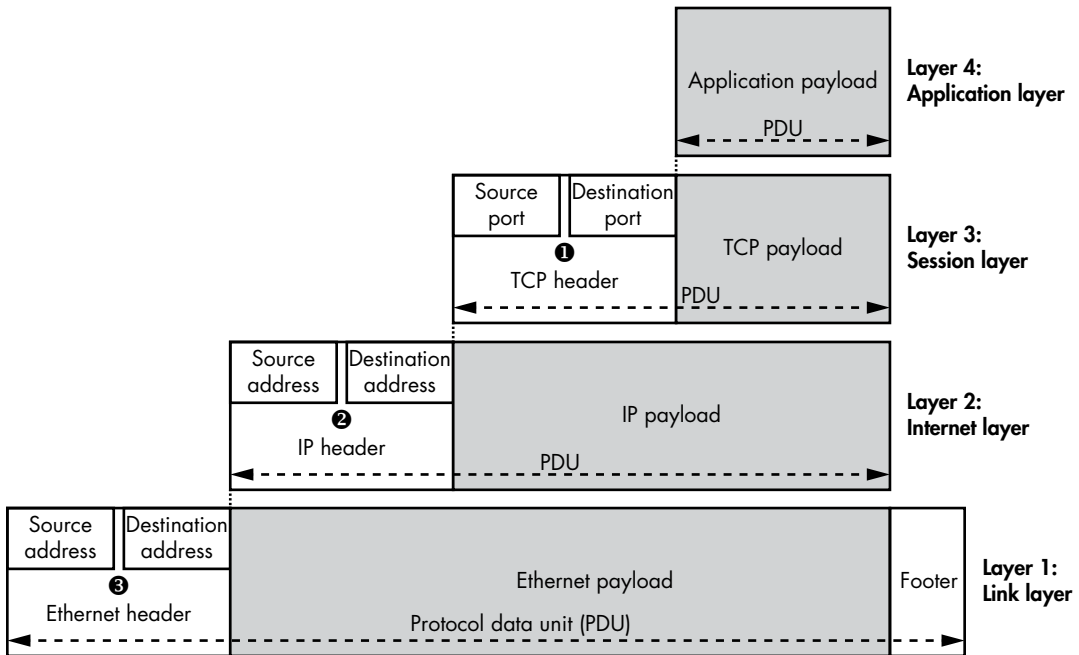
Figure 1-4: IPS data encapsulation

The TCP header contains a source and destination port number ❶. These port numbers allow a single node to have multiple unique network connections. Port numbers for TCP (and UDP) range from 0 to 65535. Most port numbers are assigned as needed to new connections, but some numbers have been given special assignments, such as port 80 for HTTP. (You can find a current list of assigned port numbers in the */etc/services* file on most Unix-like operating systems.) A TCP payload and header are commonly called a *segment*, whereas a UDP payload and header are commonly called a *datagram*.

The IP protocol uses a source and a destination address ❷. The *destination address* allows the data to be sent to a specific node on the network. The *source address* allows the receiver of the data to know which node sent the data and allows the receiver to reply to the sender.

IPv4 uses 32-bit addresses, which you'll typically see written as four numbers separated by dots, such as 192.168.10.1. IPv6 uses 128-bit addresses, because 32-bit addresses aren't sufficient for the number of nodes on modern networks. IPv6 addresses are usually written as hexadecimal numbers separated by colons, such as fe80:0000:0000:0000 :897b:581e:44b0:2057. Long strings of 0000 numbers are collapsed into

two colons. For example, the preceding IPv6 address can also be written as fe80::897b:581e:44b0:2057. An IP payload and header are commonly called a *packet*.

Ethernet also contains source and destination addresses ❸. Ethernet uses a 64-bit value called a *Media Access Control (MAC)* address, which is typically set during manufacture of the Ethernet adapter. You'll usually see MAC addresses written as a series of hexadecimal numbers separated by dashes or colons, such as 0A-00-27-00-00-0E. The Ethernet payload, including the header and footer, is commonly referred to as a *frame*.

Data Transmission

Let's briefly look at how data is transferred from one node to another using the IPS data encapsulation model. Figure 1-5 shows a simple Ethernet network with three nodes.
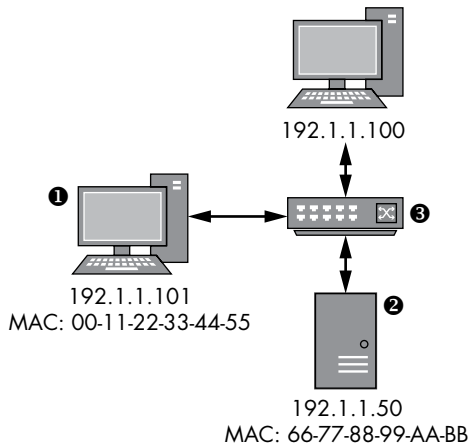
Figure 1-5: A simple Ethernet network

In this example, the node at ❶ with the IP address 192.1.1.101 wants to send data using the IP protocol to the node at ❷ with the IP address 192.1.1.50. (The *switch* device ❸ forwards Ethernet frames between all nodes on the network. The switch doesn't need an IP address because it operates only at the link layer.) Here is what takes place to send data between the two nodes:

1. The operating system network stack node ❶ encapsulates the application and transport layer data and builds an IP packet with a source address of 192.1.1.101 and a destination address of 192.1.1.50.

2. The operating system can at this point encapsulate the IP data as an Ethernet frame, but it might not know the MAC address of the target node. It can request the MAC address for a particular IP address using the Address Resolution Protocol (ARP), which sends a request to all nodes on the network to find the MAC address for the destination IP address.

3. Once the node at ❶ receives an ARP response, it can build the frame, setting the source address to the local MAC address of 00-11-22-33-44-55 and the destination address to 66-77-88-99-AA-BB. The new frame is transmitted on the network and is received by the switch ❸.

4. The switch forwards the frame to the destination node, which unpacks the IP packet and verifies that the destination IP address matches. Then the IP payload data is extracted and passes up the stack to be received by the waiting application.

Network Routing

Ethernet requires that all nodes be directly connected to the same local network. This requirement is a major limitation for a truly global network because it's not practical to physically connect every node to every other node. Rather than require that all nodes be directly connected, the source and destination addresses allow data to be *routed* over different networks until the data reaches the desired destination node, as shown in Figure 1-6.
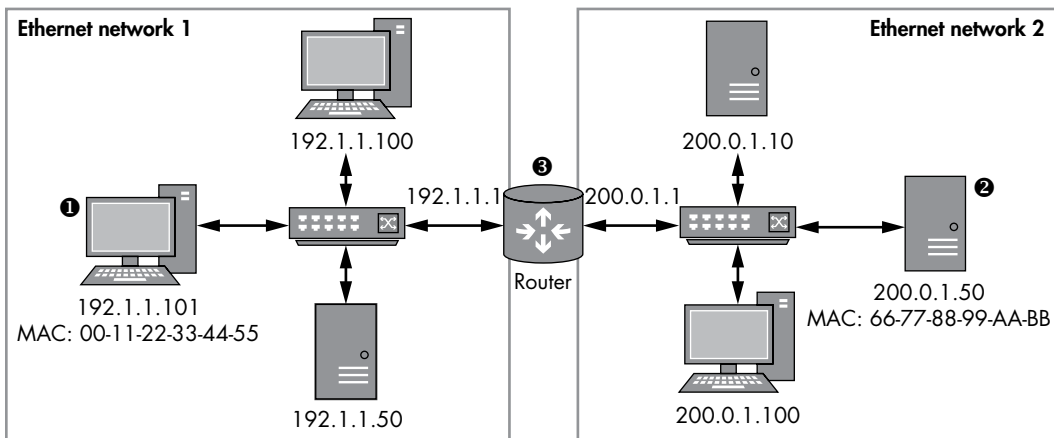
Figure 1-6: An example of a routed network connecting two Ethernet networks

Figure 1-6 shows two Ethernet networks, each with separate IP network address ranges. The following description explains how the IP uses this model to send data from the node at ❶ on network 1 to the node at ❷ on network 2.

1. The operating system network stack node ❶ encapsulates the application and transport layer data, and it builds an IP packet with a source address of 192.1.1.101 and a destination address of 200.0.1.50.

2. The network stack needs to send an Ethernet frame, but because the destination IP address does not exist on any Ethernet network that the node is connected to, the network stack consults its operating system

routing table. In this example, the routing table contains an entry for the IP address 200.0.1.50. The entry indicates that a router ❸ on IP address 192.1.1.1 knows how to get to that destination address.

3. The operating system uses ARP to look up the router's MAC address at 192.1.1.1, and the original IP packet is encapsulated within the Ethernet frame with that MAC address.

4. The router receives the Ethernet frame and unpacks the IP packet. When the router checks the destination IP address, it determines that the IP packet is not destined for the router but for a different node on another connected network. The router looks up the MAC address of 200.0.1.50, encapsulates the original IP packet into the new Ethernet frame, and sends it on to network 2.

5. The destination node receives the Ethernet frame, unpacks the IP packet, and processes its contents.

This routing process might be repeated multiple times. For example, if the router was not directly connected to the network containing the node 200.0.1.50, it would consult its own routing table and determine the next router it could send the IP packet to.

Clearly, it would be impractical for every node on the network to know how to get to every other node on the internet. If there is no explicit routing entry for a destination, the operating system provides a default routing table entry, called the *default gateway*, which contains the IP address of a router that can forward IP packets to their destinations.

My Model for Network Protocol Analysis

The IPS describes how network communication works; however, for analysis purposes, most of the IPS model is not relevant. It's simpler to use my model to understand the behavior of an application network protocol. My model contains three layers, as shown in Figure 1-7, which illustrates how I would analyze an HTTP request.

Here are the three layers of my model:

Content layer   Provides the meaning of what is being communicated. In Figure 1-7, the meaning is making an HTTP request for the file *image.jpg*.

Encoding layer   Provides rules to govern how you represent your content. In this example, the HTTP request is encoded as an HTTP GET request, which specifies the file to retrieve.

Transport layer   Provides rules to govern how data is transferred between the nodes. In the example, the HTTP GET request is sent over a TCP/IP connection to port 80 on the remote node.
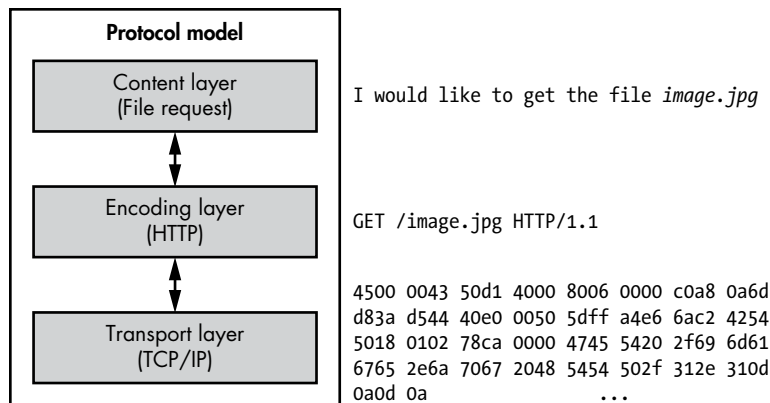
Figure 1-7: My conceptual protocol model

Splitting the model this way reduces complexity with application-specific protocols because it allows us to filter out details of the network protocol that aren't relevant. For example, because we don't really care how TCP/IP is sent to the remote node (we take for granted that it will get there somehow), we simply treat the TCP/IP data as a binary transport that just works.

To understand why the protocol model is useful, consider this protocol example: imagine you're inspecting the network traffic from some malware. You find that the malware uses HTTP to receive commands from the operator via the server. For example, the operator might ask the malware to enumerate all files on the infected computer's hard drive. The list of files can be sent back to the server, at which point the operator can request a specific file to be uploaded.

If we analyze the protocol from the perspective of how the operator would interact with the malware, such as by requesting a file to be uploaded, the new protocol breaks down into the layers shown in Figure 1-8.

Figure 1-8: The conceptual model for a malware protocol using HTTP

The following list explains each layer of the new protocol model:

Content layer   The malicious application is sending a stolen file called *secret.doc* to the server.

Encoding layer   The encoding of the command to send the stolen file is a simple text string with a command SEND followed by the filename and the file data.

Transport layer   The protocol uses an HTTP request parameter to transport the command. It uses the standard percent-encoding mechanism, making it a legal HTTP request.

Notice in this example that we don't consider the HTTP request being sent over TCP/IP; we've combined the encoding and transport layer in Figure 1-7 into just the transport layer in Figure 1-8. Although the malware still uses lower-level protocols, such as TCP/IP, these protocols are not important to the analysis of the malware command to send a file. The reason it's not important is that we can consider HTTP over TCP/IP as a single transport layer that just works and focus specifically on the unique malware commands.

By narrowing our scope to the layers of the protocol that we need to analyze, we avoid a lot of work and focus on the unique aspects of the protocol. On the other hand, if we were to analyze this protocol using the layers in Figure 1-7, we might assume that the malware was simply requesting the file *image.jpg*, because it would appear as though that was all the HTTP request was doing.

Final Words

This chapter provided a quick tour of the networking basics. I discussed the IPS, including some of the protocols you'll encounter in real networks, and described how data is transmitted between nodes on a local network as well as remote networks through routing. Additionally, I described a way to think about application network protocols that should make it easier for you to focus on the unique features of the protocol to speed up its analysis.

In Chapter 2, we'll use these networking basics to guide us in capturing network traffic for analysis. The goal of capturing network traffic is to access the data you need to start the analysis process, identify what protocols are being used, and ultimately discover security issues that you can exploit to compromise the applications using these protocols.

2

CAPTURING APPLICATION TRAFFIC

Surprisingly, capturing useful traffic can be a challenging aspect of protocol analysis. This chapter describes two different capture techniques: *passive* and *active.* Passive capture doesn't directly interact with the traffic. Instead, it extracts the data as it *travels on the wire,* which should be familiar from tools like Wireshark.

You'll find that different applications provide different mechanisms (which have their own advantages and disadvantages) to redirect traffic. Active capture interferes with traffic between a client application and the server; this has great power but can cause some complications. You can think of active capture in terms of proxies or even a man-in-the-middle attack. Let's look at both active and passive techniques in more depth.

Passive Network Traffic Capture

Passive capture is a relatively easy technique: it doesn't typically require any specialist hardware, nor do you usually need to write your own code. Figure 2-1 shows a common scenario: a client and server communicating via Ethernet over a network.
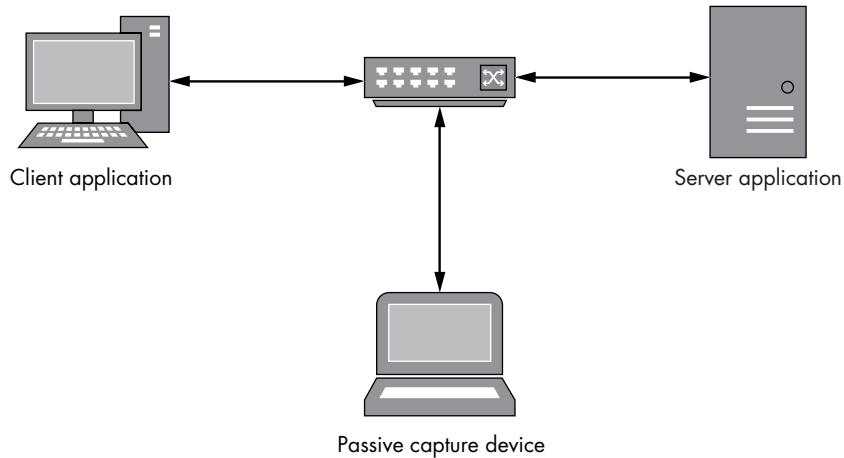
Figure 2-1: An example of passive network capture

Passive network capture can take place either on the network by tapping the traffic as it passes in some way or by sniffing directly on either the client or server host.

Quick Primer for Wireshark

Wireshark is perhaps the most popular packet-sniffing application available. It's cross platform and easy to use, and it comes with many built-in protocol analysis features. In Chapter 5 you'll learn how to write a dissector to aid in protocol analysis, but for now, let's set up Wireshark to capture IP traffic from the network.

To capture traffic from an Ethernet interface (wired or wireless), the capturing device must be in *promiscuous mode.* A device in promiscuous mode receives and processes any Ethernet frame it sees, even if that frame wasn't destined for that interface. Capturing an application running on the same computer is easy: just monitor the outbound network interface or the local loopback interface (better known as localhost). Otherwise, you might need to use networking hardware, such as a hub or a configured switch, to ensure traffic is sent to your network interface.

Figure 2-2 shows the default view when capturing traffic from an Ethernet interface.
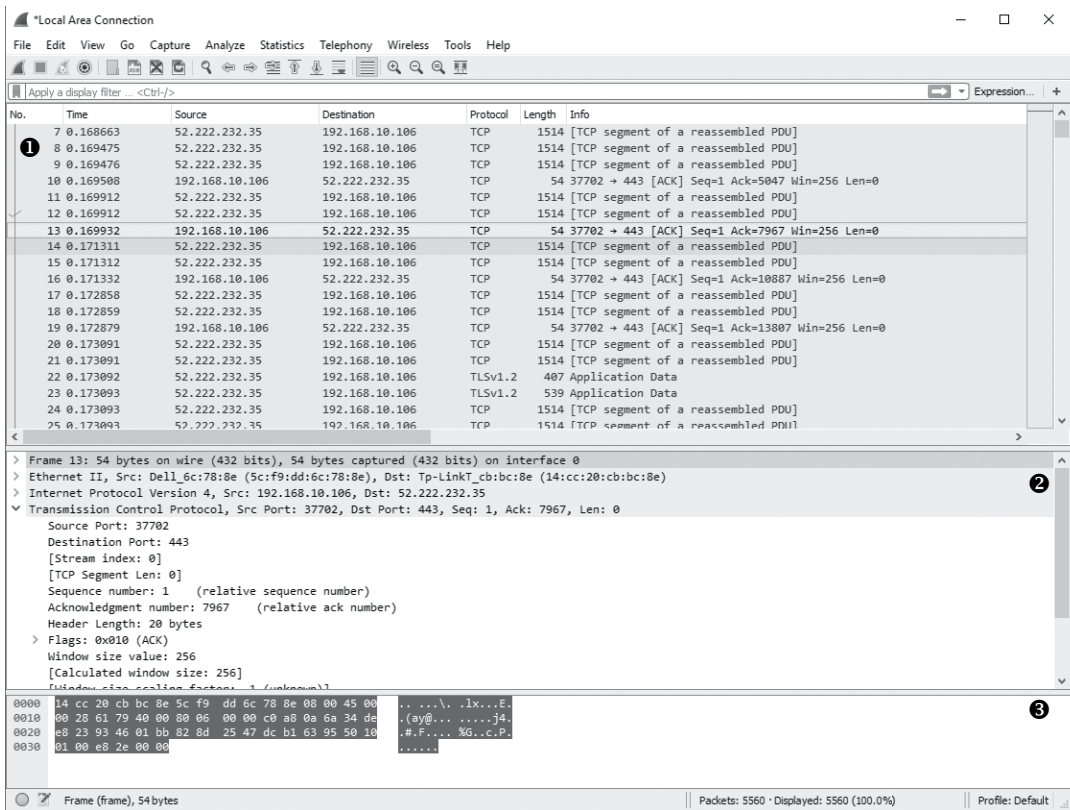
Figure 2-2: The default Wireshark view

There are three main view areas. Area ❶ shows a timeline of raw packets captured off the network. The timeline provides a list of the source and destination IP addresses as well as decoded protocol summary information. Area ❷ provides a dissected view of the packet, separated into distinct protocol layers that correspond to the OSI network stack model. Area ❸ shows the captured packet in its raw form.

The TCP network protocol is stream based and designed to recover from dropped packets or data corruption. Due to the nature of networks and IP, there is no guarantee that packets will be received in a particular order. Therefore, when you are capturing packets, the timeline view might be difficult to interpret. Fortunately, Wireshark offers dissectors for known protocols that will normally reassemble the entire stream and provide all the information in one place. For example, highlight a packet in a TCP connection in the timeline view and then select **Analyze ▸ Follow TCP Stream** from the main menu. A dialog similar to Figure 2-3 should appear. For protocols without a dissector, Wireshark can decode the stream and present it in an easy-to-view dialog.
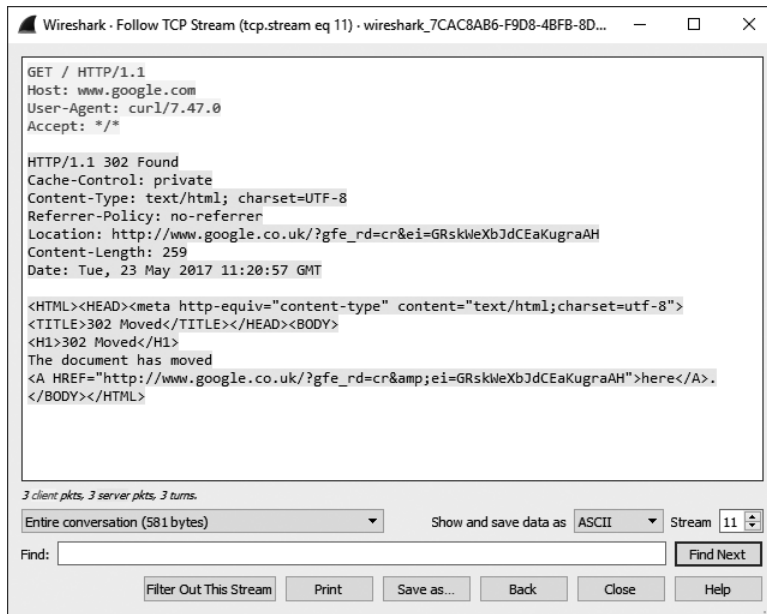
Figure 2-3: Following a TCP stream

Wireshark is a comprehensive tool, and covering all of its features is beyond the scope of this book. If you're not familiar with it, obtain a good reference, such as *Practical Packet Analysis, 3rd Edition* (No Starch Press, 2017), and learn many of its useful features. Wireshark is indispensable for analyzing application network traffic, and it's free under the General Public License (GPL).

Alternative Passive Capture Techniques

Sometimes using a packet sniffer isn't appropriate, for example, in situations when you don't have permission to capture traffic. You might be doing a penetration test on a system with no administrative access or a mobile device with a limited privilege shell. You might also just want to ensure that you look at traffic only for the application you're testing. That's not always easy to do with packet sniffing unless you correlate the traffic based on time. In this section, I'll describe a few techniques for extracting network traffic from a local application without using a packet-sniffing tool.

System Call Tracing

Many modern operating systems provide two modes of execution. *Kernel mode* runs with a high level of privilege and contains code implementing the OS's core functionality. *User mode* is where everyday processes run. The kernel provides services to user mode by exporting a collection of special system calls (see Figure 2-4), allowing users to access files, create processes—and most important for our purposes—connect to networks.
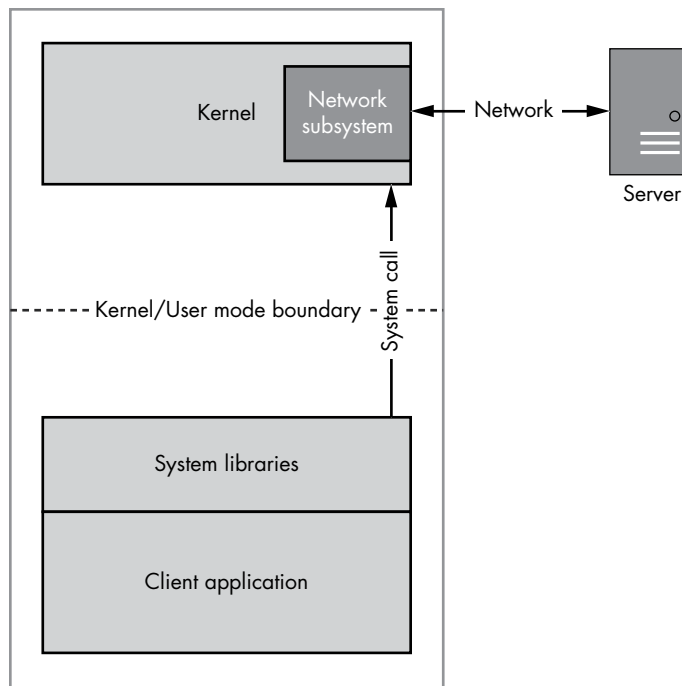
Figure 2-4: An example of user-to-kernel network communication via
system calls

When an application wants to connect to a remote server, it issues
special system calls to the OS's kernel to open a connection. The app then
reads and writes the network data. Depending on the operating system run-
ning your network applications, you can monitor these calls directly to pas-
sively extract data from an application.

Most Unix-like systems implement system calls resembling the
Berkeley Sockets model for network communication. This isn't surpris-
ing, because the IP protocol was originally implemented in the Berkeley
Software Distribution (BSD) 4.2 Unix operating system. This socket imple-
mentation is also part of POSIX, making it the de facto standard. Table 2-1
shows some of the more important system calls in the Berkeley Sockets API.

Table 2-1: Common Unix System Calls for Networking

Name	Description
socket	Creates a new socket file descriptor.
connect	Connects a socket to a known IP address and port.
bind	Binds the socket to a local known IP address and port.
recv, read, recvfrom	Receives data from the network via the socket. The generic function read is for reading from a file descriptor, whereas recv and recvfrom are specific to the socket's API.
send, write, sendfrom	Sends data over the network via the socket.

To learn more about how these system calls work, a great resource is *The TCP/IP Guide* (No Starch Press, 2005). Plenty of online resources are also available, and most Unix-like operating systems include manuals you can view at a terminal using the command man 2 *syscall_name*. Now let's look at how to monitor system calls.

The strace Utility on Linux

In Linux, you can directly monitor system calls from a user program without special permissions, unless the application you want to monitor runs as a privileged user. Many Linux distributions include the handy utility strace, which does most of the work for you. If it isn't installed by default, download it from your distribution's package manager or compile it from source.

Run the following command, replacing */path/to/app* with the application you're testing and *args* with the necessary parameters, to log the network system calls used by that application:

```
$ strace -e trace=network,read,write /path/to/app args
```

Let's monitor a networking application that reads and writes a few strings and look at the output from strace. Listing 2-1 shows four log entries (extraneous logging has been removed from the listing for brevity).

```
$ strace -e trace=network,read,write customapp
--snip--
❶ socket(PF_INET, SOCK_STREAM, IPPROTO_TCP) = 3
❷ connect(3, {sa_family=AF_INET, sin_port=htons(5555),
                        sin_addr=inet_addr("192.168.10.1")}, 16) = 0
❸ write(3, "Hello World!\n", 13)         = 13
❹ read(3, "Boo!\n", 2048)                = 5
```

Listing 2-1: Example output of the strace utility

The first entry ❶ creates a new TCP socket, which is assigned the handle 3. The next entry ❷ shows the connect system call used to make a TCP connection to IP address 192.168.10.1 on port 5555. The application then writes the string Hello World! ❸ before reading out a string Boo! ❹. The output shows it's possible to get a good idea of what an application is doing at the system call level using this utility, even if you don't have high levels of privilege.

Monitoring Network Connections with DTrace

DTrace is a very powerful tool available on many Unix-like systems, including Solaris (where it was originally developed), macOS, and FreeBSD. It allows you to set system-wide probes on special trace providers, including system calls. You configure DTrace by writing scripts in a language with a C-like syntax. For more details on this tool, refer to the DTrace Guide online at *http://www.dtracebook.com/index.php/DTrace_Guide*.

Listing 2-2 shows an example of a script that monitors outbound IP connections using DTrace.

```
/* traceconnect.d - A simple DTrace script to monitor a connect system call */
❶ struct sockaddr_in {
    short           sin_family;
    unsigned short  sin_port;
    in_addr_t       sin_addr;
    char            sin_zero[8];
};

❷ syscall::connect:entry
❸ /arg2 == sizeof(struct sockaddr_in)/
{
  ❹ addr = (struct sockaddr_in*)copyin(arg1, arg2);
  ❺ printf("process:'%s' %s:%d", execname, inet_ntop(2, &addr->sin_addr),
      ntohs(addr->sin_port));
}
```

Listing 2-2: A simple DTrace script to monitor a connect system call

This simple script monitors the connect system call and outputs IPv4 TCP and UDP connections. The system call takes three parameters, represented by arg0, arg1, and arg2 in the DTrace script language, that are initialized for us in the kernel. The arg0 parameter is the socket file descriptor (that we don't need), arg1 is the address of the socket we're connecting to, and arg2 is the length of that address. Parameter 0 is the socket handle, which is not needed in this case. The next parameter is the user process memory address of a socket address structure, which is the address to connect to and can be different sizes depending on the socket type. (For example, IPv4 addresses are smaller than IPv6.) The final parameter is the length of the socket address structure in bytes.

The script defines a sockaddr_in structure that is used for IPv4 connections at ❶; in many cases these structures can be directly copied from the system's C header files. The system call to monitor is specified at ❷. At ❸, a DTrace-specific filter is used to ensure we trace only connect calls where the socket address is the same size as sockaddr_in. At ❹, the sockaddr_in structure is copied from your process into a local structure for DTrace to inspect. At ❺, the process name, the destination IP address, and the port are printed to the console.

To run this script, copy it to a file called *traceconnect.d* and then run the command **dtrace -s traceconnect.d** as the root user. When you use a network-connected application, the output should look like Listing 2-3.

```
process:'Google Chrome'    173.194.78.125:5222
process:'Google Chrome'    173.194.66.95:443
process:'Google Chrome'    217.32.28.199:80
process:'ntpd'             17.72.148.53:123
process:'Mail'             173.194.67.109:993
```

```
process:'syncdefaultsd'     17.167.137.30:443
process:'AddressBookSour'   17.172.192.30:443
```
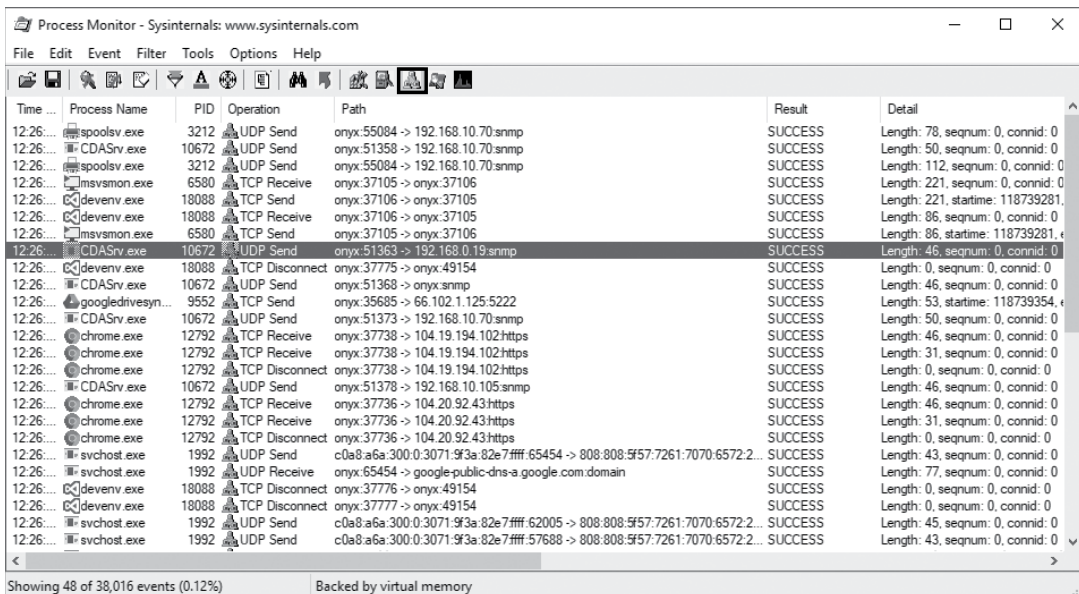
Listing 2-3: Example output from traceconnect.d *script*

The output shows individual connections to IP addresses, printing out the process name, for example `'Google Chrome'`, the IP address, and the port connected to. Unfortunately, the output isn't always as useful as the output from strace on Linux, but DTrace is certainly a valuable tool. This demonstration only scratches the surface of what DTrace can do.

Process Monitor on Windows

In contrast to Unix-like systems, Windows implements its user-mode network functions without direct system calls. The networking stack is exposed through a driver, and establishing a connection uses the file open, read, and write system calls to configure a network socket for use. Even if Windows supported a facility similar to strace, this implementation makes it more difficult to monitor network traffic at the same level as other platforms.

Windows, starting with Vista and later, has supported an event generation framework that allows applications to monitor network activity. Writing your own implementation of this would be quite complex, but fortunately, someone has already written a tool to do it for you: Microsoft's Process Monitor tool. Figure 2-5 shows the main interface when filtering only on network connection events.

Figure 2-5: An example Process Monitor capture

Selecting the filter circled in Figure 2-5 displays only events related to network connections from a monitored process. Details include the hosts involved as well as the protocol and port being used. Although the capture doesn't provide any data associated with the connections, it does offer valuable insight into the network communications the application is establishing. Process Monitor can also capture the state of the current calling stack, which helps you determine where in an application network connections are being made. This will become important in Chapter 6 when we start reverse engineering binaries to work out the network protocol. Figure 2-6 shows a single HTTP connection to a remote server in detail.
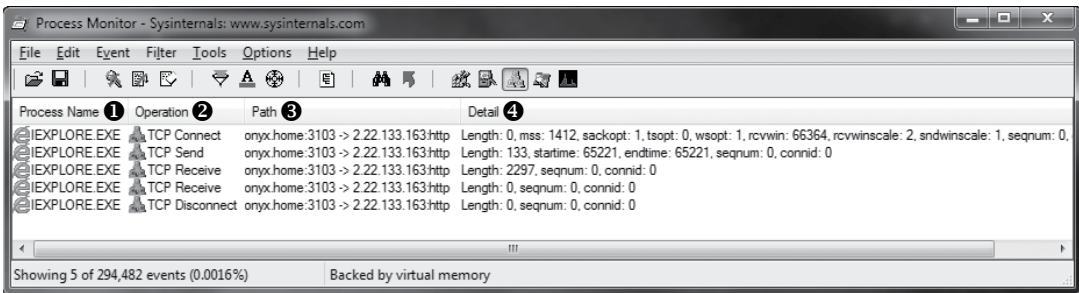
Figure 2-6: A single captured connection

Column ❶ shows the name of the process that established the connection. Column ❷ shows the operation, which in this case is connecting to a remote server, sending the initial HTTP request and receiving a response. Column ❸ indicates the source and destination addresses, and column ❹ provides more in-depth information about the captured event.

Although this solution isn't as helpful as monitoring system calls on other platforms, it's still useful in Windows when you just want to determine the network protocols a particular application is using. You can't capture data using this technique, but once you determine the protocols in use, you can add that information to your analysis through more active network traffic capture.

Advantages and Disadvantages of Passive Capture

The greatest advantage of using passive capture is that it doesn't disrupt the client and server applications' communication. It will not change the destination or source address of traffic, and it doesn't require any modifications or reconfiguration of the applications.

Passive capture might also be the only technique you can use when you don't have direct control over the client or the server. You can usually find a way to listen to the network traffic and capture it with a limited amount of effort. After you've collected your data, you can determine which active capture techniques to use and the best way to attack the protocol you want to analyze.

One major disadvantage of passive network traffic capture is that capture techniques like packet sniffing run at such a low level that it can difficult to interpret what an application received. Tools such as Wireshark certainly help, but if you're analyzing a custom protocol, it might not be possible to easily take apart the protocol without interacting with it directly.

Passive capture also doesn't always make it easy to modify the traffic an application produces. Modifying traffic isn't always necessary, but it's useful when you encounter encrypted protocols, want to disable compression, or need to change the traffic for exploitation.

When analyzing traffic and injecting new packets doesn't yield results, switch tactics and try using active capture techniques.

Active Network Traffic Capture

Active capture differs from passive in that you'll try to influence the flow of the traffic, usually by using a man-in-the-middle attack on the network communication. As shown in Figure 2-7, the device capturing traffic usually sits between the client and server applications, acting as a bridge. This approach has several advantages, including the ability to modify traffic and disable features like encryption or compression, which can make it easier to analyze and exploit a network protocol.

Client application   Man-in-the-middle proxy   Server application
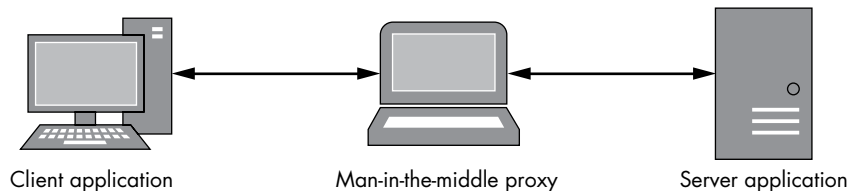
Figure 2-7: A man-in-the-middle proxy

A disadvantage of this approach is that it's usually more difficult because you need to reroute the application's traffic through your active capture system. Active capture can also have unintended, undesirable effects. For example, if you change the network address of the server or client to the proxy, this can cause confusion, resulting in the application sending traffic to the wrong place. Despite these issues, active capture is probably the most valuable technique for analyzing and exploiting application network protocols.

Network Proxies

The most common way to perform a man-in-the-middle attack on network traffic is to force the application to communicate through a proxy service. In this section, I'll explain the relative advantages and disadvantages of some of the common proxy types you can use to capture traffic, analyze that data, and exploit a network protocol. I'll also show you how to get traffic from typical client applications into a proxy.

Port-Forwarding Proxy

Port forwarding is the easiest way to proxy a connection. Just set up a listening server (TCP or UDP) and wait for a new connection. When that new connection is made to the proxy server, it will open a forwarding connection to the real service and logically connect the two, as shown in Figure 2-8.
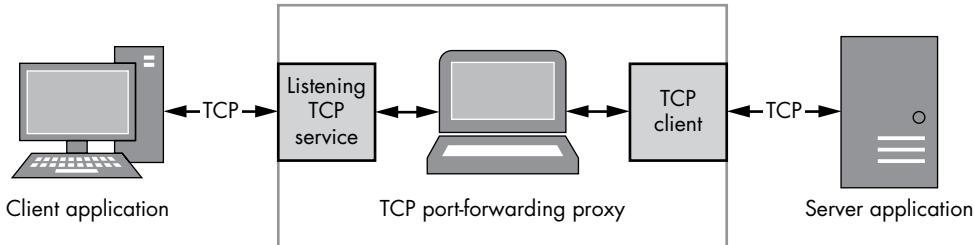
Figure 2-8: Overview of a TCP port-forwarding proxy

Simple Implementation

To create our proxy, we'll use the built-in TCP port forwarder included with the Canape Core libraries. Place the code in Listing 2-4 into a C# script file, changing *LOCALPORT* ❷, *REMOTEHOST* ❸, and *REMOTEPORT* ❹ to appropriate values for your network.

```
// PortFormatProxy.csx - Simple TCP port-forwarding proxy
// Expose methods like WriteLine and WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Create proxy template
var template = new ❶FixedProxyTemplate();
template.LocalPort = ❷LOCALPORT;
template.Host = ❸"REMOTEHOST";
template.Port = ❹REMOTEPORT;

// Create proxy instance and start
❺ var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
❻ service.Stop();

// Dump packets
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
❼ WritePackets(packets);
```

Listing 2-4: A simple TCP port-forwarding proxy example

This very simple script creates an instance of a `FixedProxyTemplate` ❶. Canape Core works on a template model, although if required you can get down and dirty with the low-level network configuration. The script configures the template with the desired local and remote network information. The template is used to create a service instance at ❺; you can think of documents in the framework acting as templates for services. The newly created service is then started; at this point, the network connections are configured. After waiting for a key press, the service is stopped at ❻. Then all the captured packets are written to the console using the `WritePackets()` method ❼.

Running this script should bind an instance of our forwarding proxy to the *LOCALPORT* number for the localhost interface only. When a new TCP connection is made to that port, the proxy code should establish a new connection to *REMOTEHOST* with TCP port *REMOTEPORT* and link the two connections together.

WARNING *Binding a proxy to all network addresses can be risky from a security perspective because proxies written for testing protocols rarely implement robust security mechanisms. Unless you have complete control over the network you are connected to or have no choice, only bind your proxy to the local loopback interface. In Listing 2-4, the default is* `LOCALHOST`*; to bind to all interfaces, set the* `AnyBind` *property to* `true`*.*

Redirecting Traffic to Proxy

With our simple proxy application complete, we now need to direct our application traffic through it.

For a web browser, it's simple enough: to capture a specific request, instead of using the URL form *http://www.domain.com/resource*, use *http://**localhost:localport**/resource*, which pushes the request through your port-forwarding proxy.

Other applications are trickier: you might have to dig into the application's configuration settings. Sometimes, the only setting an application allows you to change is the destination IP address. But this can lead to a chicken-and-egg scenario where you don't know which TCP or UDP ports the application might be using with that address, especially if the application contains complex functions running over multiple different service connections. This occurs with *Remote Procedure Call (RPC)* protocols, such as the Common Object Request Broker Architecture (CORBA). This protocol usually makes an initial network connection to a broker, which acts as a directory of available services. A second connection is then made to the requested service over an instance-specific TCP port.

In this case, a good approach is to use as many network-connected features of the application as possible while monitoring it using passive capture techniques. By doing so, you should uncover the connections that application typically makes, which you can then easily replicate with forwarding proxies.

If the application doesn't support changing its destination, you need to be a bit more creative. If the application resolves the destination server

address via a hostname, you have more options. You could set up a custom DNS server that responds to name requests with the IP address of your proxy. Or you could use the *hosts* file facility, which is available on most operating systems, including Windows, assuming you have control over system files on the device the application is running on.

During hostname resolving, the OS (or the resolving library) first refers to the *hosts* file to see if any local entries exist for that name, making a DNS request only if one is not found. For example, the hosts file in Listing 2-5 redirects the hostnames *www.badgers.com* and *www.domain.com* to *localhost.*

```
# Standard Localhost addresses
127.0.0.1       localhost
::1             localhost

# Following are dummy entries to redirect traffic through the proxy
127.0.0.1       www.badgers.com
127.0.0.1       www.domain.com
```

Listing 2-5: An example hosts *file*

The standard location of the *hosts* file on Unix-like OSes is */etc/hosts*, whereas on Windows it is *C:\Windows\System32\Drivers\etc\hosts.* Obviously, you'll need to replace the path to the Windows folder as necessary for your environment.

NOTE   *Some antivirus and security products track changes to the system's hosts, because changes are a sign of malware. You might need to disable the product's protection if you want to change the* hosts *file.*

Advantages of a Port-Forwarding Proxy

The main advantage of a port-forwarding proxy is its simplicity: you wait for a connection, open a new connection to the original destination, and then pass traffic back and forth between the two. There is no protocol associated with the proxy to deal with, and no special support is required by the application from which you are trying to capture traffic.

A port-forwarding proxy is also the primary way of proxying UDP traffic; because it isn't connection oriented, the implementation of a forwarder for UDP is considerably simpler.

Disadvantages of a Port-Forwarding Proxy

Of course, the simplicity of a port-forwarding proxy also contributes to its disadvantages. Because you are only forwarding traffic from a listening connection to a single destination, multiple instances of a proxy would be required if the application uses multiple protocols on different ports.

For example, consider an application that has a single hostname or IP address for its destination, which you can control either directly by changing it in the application's configuration or by spoofing the hostname. The application then attempts to connect to TCP ports 443 and 1234. Because

you can control the address it connects to, not the ports, you need to set up forwarding proxies for both, even if you are only interested in the traffic running over port 1234.

This proxy can also make it difficult to handle more than one connection to a well-known port. For example, if the port-forwarding proxy is listening on port 1234 and making a connection to *www.domain.com* port 1234, only redirected traffic for the original domain will work as expected. If you wanted to also redirect *www.badgers.com*, things would be more difficult. You can mitigate this if the application supports specifying the destination address and port or by using other techniques, such as Destination Network Address Translation (DNAT), to redirect specific connections to unique forwarding proxies. (Chapter 5 contains more details on DNAT as well as numerous other more advanced network capture techniques.)

Additionally, the protocol might use the destination address for its own purposes. For example, the Host header in HyperText Transport Protocol (HTTP) can be used for Virtual Host decisions, which might make a port-forwarded protocol work differently, or not at all, from a redirected connection. Still, at least for HTTP, I will discuss a workaround for this limitation in "Reverse HTTP Proxy" on page 32.

SOCKS Proxy

Think of a SOCKS proxy as a port-forwarding proxy on steroids. Not only does it forward TCP connections to the desired network location, but all new connections start with a simple handshake protocol that informs the proxy of the ultimate destination rather than having it fixed. It can also support listening connections, which is important for protocols like File Transfer Protocol (FTP) that need to open new local ports for the server to send data to. Figure 2-9 provides an overview of SOCKS proxy.
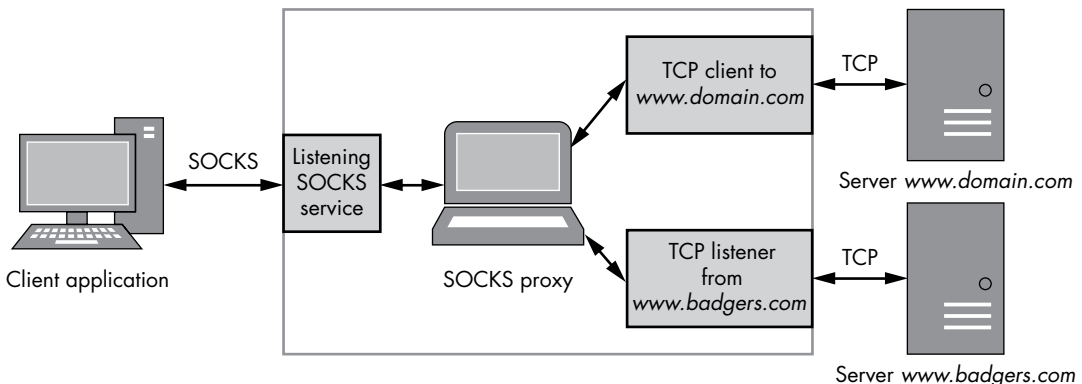
Figure 2-9: Overview of SOCKS proxy

Three common variants of the protocol are currently in use—SOCKS 4, 4a, and 5—and each has its own use. Version 4 is the most commonly supported version of the protocol; however, it supports only IPv4 connections, and the destination address must be specified as a 32-bit IP address. An

update to version 4, version 4a allowed connections by hostname (which is useful if you don't have a DNS server that can resolve IP addresses). Version 5 introduced hostname support, IPv6, UDP forwarding, and improved authentication mechanisms; it is also the only one specified in an RFC (1928).

As an example, a client will send the request shown in Figure 2-10 to establish a SOCKS connection to IP address 10.0.0.1 on port 12345. The USERNAME component is the only method of authentication in SOCKS version 4 (not especially secure, I know). VER represents the version number, which in this case is 4. CMD indicates it wants to connect out (binding to an address is CMD 2), and the TCP port and address are specified in binary form.

VER 0x04	CMD 0x01	TCP PORT 12345	IP ADDRESS 0x10000001	USERNAME "james"	NULL 0x00
Size in octets 1	1	2	4	VARIABLE	1

Figure 2-10: A SOCKS version 4 request

If the connection is successful, it will send back the appropriate response, as shown in Figure 2-11. The RESP field indicates the status of the response; the TCP port and address fields are only significant for binding requests. Then the connection becomes transparent and the client and server directly negotiate with each other; the proxy server only acts to forward traffic in either direction.

VER 0x04	RESP 0x5A	TCP PORT 0	IP ADDRESS 0
Size in octets 1	1	2	4

Figure 2-11: A SOCKS version 4 successful response

Simple Implementation

The Canape Core libraries have built-in support for SOCKS 4, 4a, and 5. Place Listing 2-6 into a C# script file, changing *LOCALPORT* ❷ to the local TCP port you want to listen on for the SOCKS proxy.

SocksProxy.csx
```
// SocksProxy.csx – Simple SOCKS proxy
// Expose methods like WriteLine and WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Create the SOCKS proxy template
❶ var template = new SocksProxyTemplate();
template.LocalPort = ❷LOCALPORT;

// Create proxy instance and start
var service = template.Create();
service.Start();
```

```
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Dump packets
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Listing 2-6: A simple SOCKS proxy example

Listing 2-6 follows the same pattern established with the TCP port-forwarding proxy in Listing 2-4. But in this case, the code at ❶ creates a SOCKS proxy template. The rest of the code is exactly the same.

Redirecting Traffic to Proxy

To determine a way of pushing an application's network traffic through a SOCKS proxy, look in the application first. For example, when you open the proxy settings in Mozilla Firefox, the dialog in Figure 2-12 appears. From there, you can configure Firefox to use a SOCKS proxy.
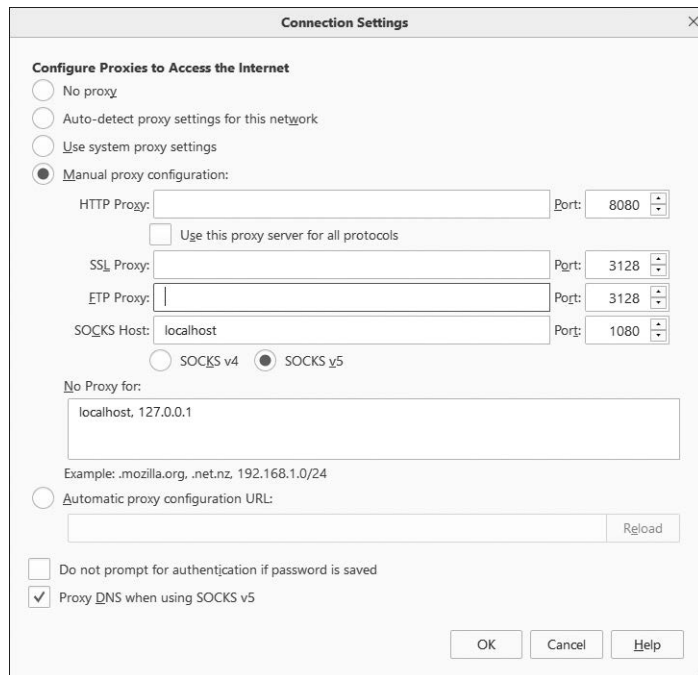
Figure 2-12: Firefox proxy configuration

But sometimes SOCKS support is not immediately obvious. If you are testing a Java application, the Java Runtime accepts command line parameters that enable SOCKS support for any outbound TCP connection. For example, consider the very simple Java application in Listing 2-7, which connects to IP address 192.168.10.1 on port 5555.

SocketClient.java
```
// SocketClient.java - A simple Java TCP socket client
import java.io.PrintWriter;
import java.net.Socket;

public class SocketClient {
    public static void main(String[] args) {
        try {
            Socket s = new Socket("192.168.10.1", 5555);
            PrintWriter out = new PrintWriter(s.getOutputStream(), true);
            out.println("Hello World!");
            s.close();
        } catch(Exception e) {
        }
    }
}
```

Listing 2-7: A simple Java TCP client

When you run this compiled program normally, it would do as you expect. But if on the command line you pass two special system properties, socksProxyHost and socksProxyPort, you can specify a SOCKS proxy for any TCP connection:

```
java –DsocksProxyHost=localhost –DsocksProxyPort=1080 SocketClient
```

This will make the TCP connection through the SOCKS proxy on localhost port 1080.

Another place to look to determine how to push an application's network traffic through a SOCKS proxy is the OS's default proxy. On macOS, navigate to **System Preferences ▸ Network ▸ Advanced ▸ Proxies**. The dialog shown in Figure 2-13 appears. From here, you can configure a system-wide SOCKS proxy or general proxies for other protocols. This won't always work, but it's an easy option worth trying out.

In addition, if the application just will not support a SOCKS proxy natively, certain tools will add that function to arbitrary applications. These tools range from free and open source tools, such as Dante (*https://www.inet.no/dante/*) on Linux, to commercial tools, such as Proxifier (*https://www.proxifier.com/*), which runs on Windows and macOS. In one way or another, they all inject into the application to add SOCKS support and modify the operation of the socket functions.
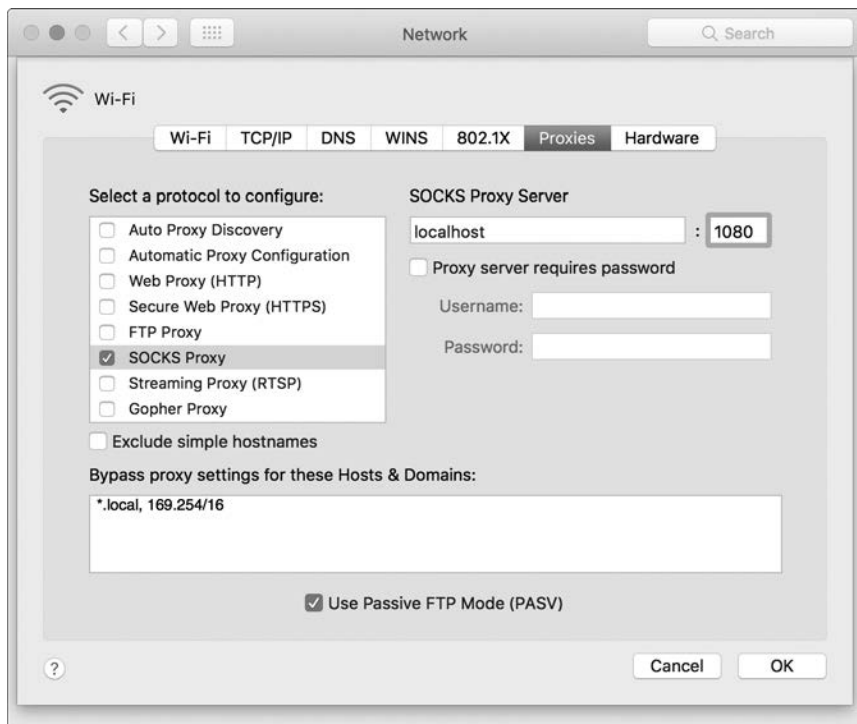
Figure 2-13: A proxy configuration dialog on macOS

Advantages of a SOCKS Proxy

The clear advantage of using a SOCKS proxy, as opposed to using a simple port forwarder, is that it should capture all TCP connections (and potentially some UDP if you are using SOCKS version 5) that an application makes. This is an advantage as long as the OS socket layer is wrapped to effectively push all connections through the proxy.

A SOCKS proxy also generally preserves the destination of the connection from the point of view of the client application. Therefore, if a client application sends in-band data that refers to its endpoint, then the endpoint will be what the server expects. However, this does not preserve the source address. Some protocols, such as FTP, assume they can request ports to be opened on the originating client. The SOCKS protocol provides a facility for binding listening connections but adds to the complexity of the implementation. This makes capture and analysis more difficult because you must consider many different streams of data to and from a server.

Disadvantages of a SOCKS Proxy

The main disadvantage of SOCKS is that support can be inconsistent between applications and platforms. The Windows system proxy supports only SOCKS version 4 proxies, which means it will resolve only local

hostnames. It does not support IPv6 and does not have a robust authentica-
tion mechanism. Generally, you get better support by using a SOCKS tool
to add to an existing application, but this doesn't always work well.

HTTP Proxies

HTTP powers the World Wide Web as well as a myriad of web services and
RESTful protocols. Figure 2-14 provides an overview of an HTTP proxy.
The protocol can also be co-opted as a transport mechanism for non-web
protocols, such as Java's Remote Method Invocation (RMI) or Real Time
Messaging Protocol (RTMP), because it can tunnel though the most restric-
tive firewalls. It is important to understand how HTTP proxying works in
practice, because it will almost certainly be useful for protocol analysis,
even if a web service is not being tested. Existing web application–testing
tools rarely do an ideal job when HTTP is being used out of its original
environment. Sometimes rolling your own implementation of an HTTP
proxy is the only solution.
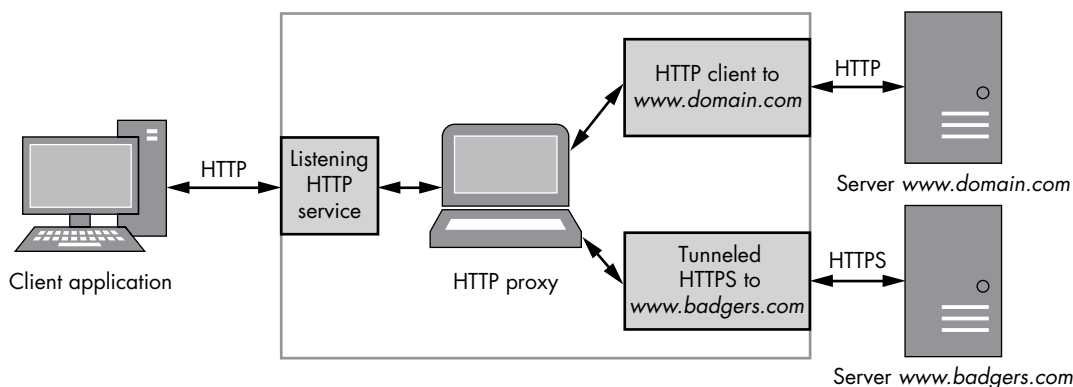
Figure 2-14: Overview of an HTTP proxy

The two main types of HTTP proxy are the forwarding proxy and the
reverse proxy. Each has advantages and disadvantages for the prospective
network protocol analyzer.

Forwarding an HTTP Proxy

The HTTP protocol is specified in RFC 1945 for version 1.0 and RFC 2616
for version 1.1; both versions provide a simple mechanism for proxying
HTTP requests. For example, HTTP 1.1 specifies that the first full line of
a request, the *request line*, has the following format:

```
❶GET ❷/image.jpg HTTP/1.1
```

The method ❶ specifies what to do in that request using familiar
verbs, such as GET, POST, and HEAD. In a proxy request, this does not change
from a normal HTTP connection. The path ❷ is where the proxy request
gets interesting. As is shown, an absolute path indicates the resource that

the method will act upon. Importantly, the path can also be an absolute Uniform Request Identifier (URI). By specifying an absolute URI, a proxy server can establish a new connection to the destination, forwarding all traffic on and returning data back to the client. The proxy can even manipulate the traffic, in a limited fashion, to add authentication, hide version 1.0 servers from 1.1 clients, and add transfer compression along with all manner of other things. However, this flexibility comes with a cost: the proxy server must be able to process the HTTP traffic, which adds massive complexity. For example, the following request line accesses an image resource on a remote server through a proxy:

```
GET http://www.domain.com/image.jpg HTTP/1.1
```

You, the attentive reader, might have identified an issue with this approach to proxying HTTP communication. Because the proxy must be able to access the underlying HTTP protocol, what about HTTPS, which transports HTTP over an encrypted TLS connection? You could break out the encrypted traffic; however, in a normal environment, it is unlikely the HTTP client would trust whatever certificate you provided. Also, TLS is intentionally designed to make it virtually impossible to use a man-in-the-middle attack any other way. Fortunately, this was anticipated, and RFC 2817 provides two solutions: it includes the ability to upgrade an HTTP connection to encryption (there is no need for more details here), and more importantly for our purposes, it specifies the CONNECT HTTP method for creating transparent, tunneled connections over HTTP proxies. As an example, a web browser that wants to establish a proxy connection to an HTTPS site can issue the following request to the proxy:

```
CONNECT www.domain.com:443 HTTP/1.1
```

If the proxy accepts this request, it will make a new TCP connection to the server. On success, it should return the following response:

```
HTTP/1.1 200 Connection Established
```

The TCP connection to the proxy now becomes transparent, and the browser is able to establish the negotiated TLS connection without the proxy getting in the way. Of course, it's worth noting that the proxy is unlikely to verify that TLS is actually being used on this connection. It could be any protocol you like, and this fact is abused by some applications to tunnel out their own binary protocols through HTTP proxies. For this reason, it's common to find deployments of HTTP proxies restricting the ports that can be tunneled to a very limited subset.

Simple Implementation

Once again, the Canape Core libraries include a simple implementation of an HTTP proxy. Unfortunately, they don't support the CONNECT method to

create a transparent tunnel, but it will suffice for demonstration purposes. Place Listing 2-8 into a C# script file, changing *LOCALPORT* ❷ to the local TCP port you want to listen on.

```
// HttpProxy.csx - Simple HTTP proxy
// Expose methods like WriteLine and WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Create proxy template
❶ var template = new HttpProxyTemplate();
template.LocalPort = ❷LOCALPORT;

// Create proxy instance and start
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Dump packets
var packets = service.Packets;
WriteLine("Captured {0} packets:", packets.Count);
WritePackets(packets);
```

Listing 2-8: A simple forward HTTP proxy example

Here we created a forward HTTP Proxy. The code at line ❶ is again only a slight variation from the previous examples, creating an HTTP proxy template.

Redirecting Traffic to Proxy

As with SOCKS proxies, the first port of call will be the application. It's rare for an application that uses the HTTP protocol to not have some sort of proxy configuration. If the application has no specific settings for HTTP proxy support, try the OS configuration, which is in the same place as the SOCKS proxy configuration. For example, on Windows you can access the system proxy settings by selecting Control Panel ▸ Internet Options ▸ Connections ▸ LAN Settings.

Many command line utilities on Unix-like systems, such as curl, wget, and apt, also support setting HTTP proxy configuration through environment variables. If you set the environment variable http_proxy to the URL for the HTTP proxy to use—for example, *http://localhost:3128*—the application will use it. For secure traffic, you can also use *https_proxy*. Some implementations allow special URL schemes, such as *socks4://*, to specify that you want to use a SOCKS proxy.

Advantages of a Forwarding HTTP Proxy

The main advantage of a forwarding HTTP proxy is that if the application uses the HTTP protocol exclusively, all it needs to do to add proxy support is to change the absolute path in the Request Line to an absolute URI and send the data to a listening proxy server. Also, only a few applications that use the HTTP protocol for transport do not already support proxying.

Disadvantages of a Forwarding HTTP Proxy

The requirement of a forwarding HTTP proxy to implement a full HTTP parser to handle the many idiosyncrasies of the protocol adds significant complexity; this complexity might introduce processing issues or, in the worst case, security vulnerabilities. Also, the addition of the proxy destination within the protocol means that it can be more difficult to retrofit HTTP proxy support to an existing application through external techniques, unless you convert connections to use the CONNECT method (which even works for unencrypted HTTP).

Due to the complexities of handling a full HTTP 1.1 connection, it is common for proxies to either disconnect clients after a single request or downgrade communications to version 1.0 (which always closes the response connection after all data has been received). This might break a higher-level protocol that expects to use version 1.1 or request *pipelining*, which is the ability to have multiple requests *in flight* to improve performance or state locality.

Reverse HTTP Proxy

Forwarding proxies are fairly common in environments where an internal client is connecting to an outside network. They act as a security boundary, limiting outbound traffic to a small subset of protocol types. (Let's just ignore the potential security implications of the CONNECT proxy for a moment.) But sometimes you might want to proxy inbound connections, perhaps for load-balancing or security reasons (to prevent exposing your servers directly to the outside world). However, a problem arises if you do this. You have no control over the client. In fact, the client probably doesn't even realize it's connecting to a proxy. This is where the *reverse HTTP proxy* comes in.

Instead of requiring the destination host to be specified in the request line, as with a forwarding proxy, you can abuse the fact that all HTTP 1.1–compliant clients *must* send a Host HTTP header in the request that specifies the original hostname used in the URI of the request. (Note that HTTP 1.0 has no such requirement, but most clients using that version will send the header anyway.) With the Host header information, you can infer the original destination of the request, making a proxy connection to that server, as shown in Listing 2-9.

```
GET /image.jpg HTTP/1.1
User-Agent: Super Funky HTTP Client v1.0
Host: ❶www.domain.com
Accept: */*
```

Listing 2-9: An example HTTP request

Listing 2-9 shows a typical Host header ❶ where the HTTP request was to the URL *http://www.domain.com/image.jpg*. The reverse proxy can easily take this information and reuse it to construct the original destination. Again, because there is a requirement for parsing the HTTP headers, it is more difficult to use for HTTPS traffic that is protected by TLS. Fortunately, most TLS implementations take wildcard certificates where the subject is in the form of **.domain.com* or similar, which would match any subdomain of *domain.com*.

Simple Implementation

Unsurprisingly, the Canape Core libraries include a built-in HTTP reverse proxy implementation, which you can access by changing the template object to *HttpReverseProxyTemplate* from *HttpProxyTemplate*. But for completeness, Listing 2-10 shows a simple implementation. Place the following code in a C# script file, changing `LOCALPORT` ❶ to the local TCP port you want to listen on. If `LOCALPORT` is less than 1024 and you're running this on a Unix-style system, you'll also need to run the script as root.

ReverseHttp
Proxy.csx

```
// ReverseHttpProxy.csx – Simple reverse HTTP proxy
// Expose methods like WriteLine and WritePackets
using static System.Console;
using static CANAPE.Cli.ConsoleUtils;

// Create proxy template
var template = new HttpReverseProxyTemplate();
template.LocalPort = ❶LOCALPORT;

// Create proxy instance and start
var service = template.Create();
service.Start();

WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();

// Dump packets
var packets = service.Packets;
WriteLine("Captured {0} packets:",
    packets.Count);
WritePackets(packets);
```

Listing 2-10: A simple reverse HTTP proxy example

Redirecting Traffic to Your Proxy

The approach to redirecting traffic to a reverse HTTP proxy is similar to that employed for TCP port-forwarding, which is by redirecting the connection to the proxy. But there is a big difference; you can't just change the destination hostname. This would change the Host header, shown in Listing 2-10. If you're not careful, you could cause a proxy loop.[1] Instead, it's best to change the IP address associated with a hostname using the *hosts* file.

But perhaps the application you're testing is running on a device that doesn't allow you to change the *hosts* file. Therefore, setting up a custom DNS server might be the easiest approach, assuming you're able to change the DNS server configuration.

You could use another approach, which is to configure a full DNS server with the appropriate settings. This can be time consuming and error prone; just ask anyone who has ever set up a bind server. Fortunately, existing tools are available to do what we want, which is to return our proxy's IP address in response to a DNS request. Such a tool is *dnsspoof*. To avoid installing another tool, you can do it using Canape's DNS server. The basic DNS server spoofs only a single IP address to all DNS requests (see Listing 2-11). Replace *IPV4ADDRESS* ❶, *IPV6ADDRESS* ❷, and *REVERSEDNS* ❸ with appropriate strings. As with the HTTP Reverse Proxy, you'll need to run this as root on a Unix-like system, as it will try to bind to port 53, which is not usually allowed for normal users. On Windows, there's no such restriction on binding to ports less than 1024.

DnsServer.csx

```
// DnsServer.csx - Simple DNS Server
// Expose console methods like WriteLine at global level.
using static System.Console;

// Create the DNS server template
var template = new DnsServerTemplate();

// Setup the response addresses
template.ResponseAddress = ❶"IPV4ADDRESS";
template.ResponseAddress6 = ❷"IPV6ADDRESS";
template.ReverseDns = ❸"REVERSEDNS";

// Create DNS server instance and start
var service = template.Create();
service.Start();
WriteLine("Created {0}", service);
WriteLine("Press Enter to exit...");
ReadLine();
service.Stop();
```

Listing 2-11: A simple DNS server

---

1. A proxy loop occurs when a proxy repeatedly connects to itself, causing a recursive loop. The outcome can only end in disaster, or at least running out of available resources.

Now if you configure the DNS server for your application to point to your spoofing DNS server, the application should send its traffic through.

Advantage of a Reverse HTTP Proxy

The advantage of a reverse HTTP proxy is that it doesn't require a client application to support a typical forwarding proxy configuration. This is especially useful if the client application is not under your direct control or has a fixed configuration that cannot be easily changed. As long as you can force the original TCP connections to be redirected to the proxy, it's possible to handle requests to multiple different hosts with little difficulty.

Disadvantages of a Reverse HTTP Proxy

The disadvantages of a reverse HTTP proxy are basically the same as for a forwarding proxy. The proxy must be able to parse the HTTP request and handle the idiosyncrasies of the protocol.

Final Words

You've read about passive and active capture techniques in this chapter, but is one better than the other? That depends on the application you're trying to test. Unless you are just monitoring network traffic, it pays to take an active approach. As you continue through this book, you'll realize that active capture has significant benefits for protocol analysis and exploitation. If you have a choice in your application, use SOCKS because it's the easiest approach in many circumstances.

3

NETWORK PROTOCOL STRUCTURES

The old adage "There is nothing new under the sun" holds true when it comes to the way protocols are structured. Binary and text protocols follow common patterns and structures and, once understood, can easily be applied to any new protocol. This chapter details some of these structures and formalizes the way I'll represent them throughout the rest of this book.

In this chapter, I discuss many of the common types of protocol structures. Each is described in detail along with how it is represented in binary- or text-based protocols. By the end of the chapter, you should be able to easily identify these common types in any unknown protocol you analyze.

Once you understand how protocols are structured, you'll also see patterns of exploitable behavior—ways of attacking the network protocol itself. Chapter 10 will provide more detail on finding network protocol issues, but for now we'll just concern ourselves with structure.

Binary Protocol Structures

Binary protocols work at the binary level; the smallest unit of data is a single binary digit. Dealing with single bits is difficult, so we'll use 8-bit units called *octets*, commonly called *bytes*. The octet is the de facto unit of network protocols. Although octets can be broken down into individual bits (for example, to represent a set of flags), we'll treat all network data in 8-bit units, as shown in Figure 3-1.

Bit 7/MSB   Bit 0/LSB

Bit format: [0] 1 0 0 0 0 0 [1] = 0x41/65

Octet format: 0x41

Figure 3-1: Binary data description formats

When showing individual bits, I'll use the *bit format*, which shows bit 7, the *most significant bit (MSB)*, on the left. Bit 0, or the *least significant bit (LSB)*, is on the right. (Some architectures, such as PowerPC, define the bit numbering in the opposite direction.)

Numeric Data

Data values representing numbers are usually at the core of a binary protocol. These values can be integers or decimal values. Numbers can be used to represent the length of data, to identify tag values, or simply to represent a number.

In binary, numeric values can be represented in a few different ways, and a protocol's method of choice depends on the value it's representing. The following sections describe some of the more common formats.

Unsigned Integers

Unsigned integers are the most obvious representation of a binary number. Each bit has a specific value based on its position, and these values are added together to represent the integer. Table 3-1 shows the decimal and hexadecimal values for an 8-bit integer.

Table 3-1: Decimal Bit Values

Bit	Decimal value	Hex value
0	1	0x01
1	2	0x02
2	4	0x04
3	8	0x08
4	16	0x10
5	32	0x20
6	64	0x40
7	128	0x80

Signed Integers

Not all integer values are positive. In some scenarios, negative integers are required—for example, to represent the difference between two integers, you need to take into account that the difference could be negative—and only signed integers can hold negative values. While encoding an unsigned integer seems obvious, the CPU can only work with the same set of bits. Therefore, the CPU requires a way of interpreting the unsigned integer value as signed; the most common signed interpretation is two's complement. The term *two's complement* refers to the way in which the signed integer is represented within a native integer value in the CPU.

Conversion between unsigned and signed values in two's complement is done by taking the bitwise NOT (where a 0 bit is converted to a 1 and 1 is converted to a 0) of the integer and adding 1. For example, Figure 3-2 shows the 8-bit integer 123 converted to its two's complement representation.

```
        MSB              LSB
NOT [   0 1 1 1 1 0 1 1   ]   = 0x7B/123
                 ↓
 +1 [   1 0 0 0 0 1 0 0   ]   = 0x84/-124
                 ↓
  = [   1 0 0 0 0 1 0 1   ]   = 0x85/-123
```
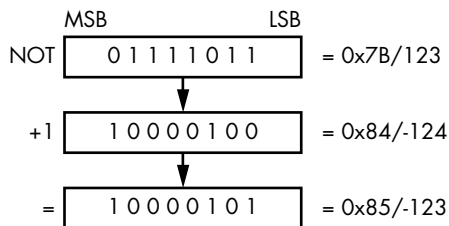
Figure 3-2: The two's complement representation of 123

The two's complement representation has one dangerous security consequence. For example, an 8-bit signed integer has the range –128 to 127, so the magnitude of the minimum is larger than the maximum. If the minimum value is negated, the result is itself; in other words, –(–128) is –128. This can cause calculations to be incorrect in parsed formats, leading to security vulnerabilities. We'll go into more detail in Chapter 10.

Variable-Length Integers

Efficient transfer of network data has historically been very important. Even though today's high-speed networks might make efficiency concerns unnecessary, there are still advantages to reducing a protocol's bandwidth. It can be beneficial to use variable-length integers when the most common integer values being represented are within a very limited range.

For example, consider length fields: when sending blocks of data between 0 and 127 bytes in size, you could use a 7-bit variable integer representation. Figure 3-3 shows a few different encodings for 32-bit words. At most, five octets are required to represent the entire range. But if your protocol tends to assign values between 0 and 127, it will only use one octet, which saves a considerable amount of space.
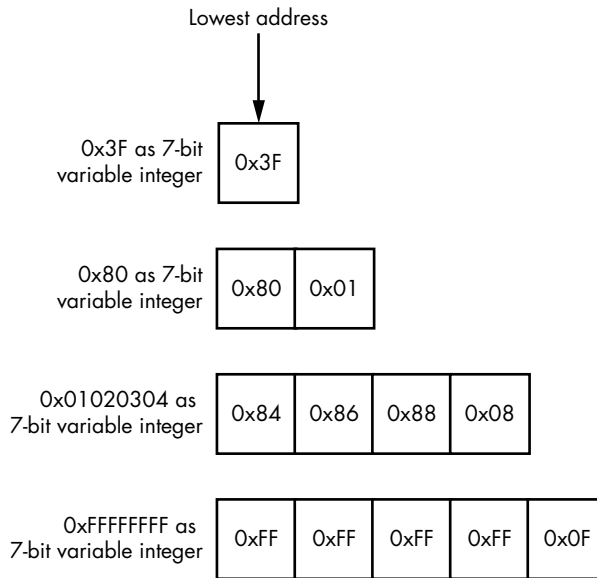
Figure 3-3: Example 7-bit integer encoding

That said, if you parse more than five octets (or even 32 bits), the resulting integer from the parsing operation will depend on the parsing program. Some programs (including those developed in C) will simply drop any bits beyond a given range, whereas other development environments will generate an overflow error. If not handled correctly, this integer overflow might lead to vulnerabilities, such as buffer overflows, which could cause a smaller than expected memory buffer to be allocated, in turn resulting in memory corruption.

Floating-Point Data

Sometimes, integers aren't enough to represent the range of decimal values needed for a protocol. For example, a protocol for a multiplayer computer game might require sending the coordinates of players or objects in the game's virtual world. If this world is large, it would be easy to run up against the limited range of a 32- or even 64-bit fixed-point value.

The format of floating-point integers used most often is the *IEEE format* specified in IEEE Standard for Floating-Point Arithmetic (IEEE 754). Although the standard specifies a number of different binary and even decimal formats for floating-point values, you're likely to encounter only two: a single-precision binary representation, which is a 32-bit value; and a double-precision, 64-bit value. Each format specifies the position and bit size of the significand and exponent. A sign bit is also specified, indicating whether the value is positive or negative. Figure 3-4 shows the general layout of an IEEE floating-point value, and Table 3-2 lists the common exponent and significand sizes.

Sign

IEEE floating-point format

Exponent	Significand

MSB   LSB
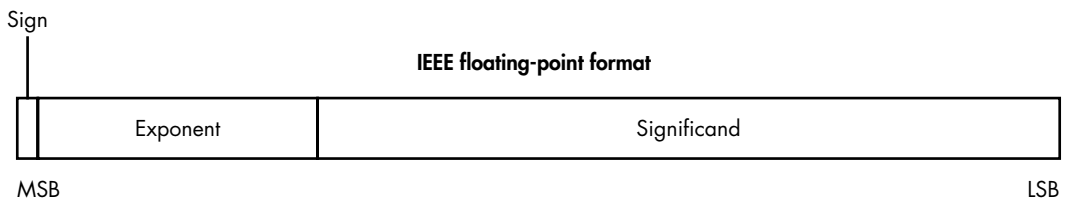
Figure 3-4: Floating-point representation

Table 3-2: Common Float Point Sizes and Ranges

Bit size	Exponent bits	Significand bits	Value range
32	8	23	+/− 3.402823×10^{38}
64	11	52	+/− $1.79769313486232 \times 10^{308}$

Booleans

Because Booleans are very important to computers, it's no surprise to see them reflected in a protocol. Each protocol determines how to represent whether a Boolean value is true or false, but there are some common conventions.

The basic way to represent a Boolean is with a single-bit value. A 0 bit means false and a 1 means true. This is certainly space efficient but not necessarily the simplest way to interface with an underlying application. It's more common to use a single byte for a Boolean value because it's far easier to manipulate. It's also common to use zero to represent false and non-zero to represent true.

Bit Flags

Bit flags are one way to represent specific Boolean states in a protocol. For example, in TCP a set of bit flags is used to determine the current state of a connection. When making a connection, the client sends a packet with the synchronize flag (SYN) set to indicate that the connections should synchronize their timers. The server can then respond with an acknowledgment (ACK) flag to indicate it has received the client request as well as the SYN flag to establish the synchronization with the client. If this handshake used single enumerated values, this dual state would be impossible without a distinct SYN/ACK state.

Binary Endian

The endianness of data is a very important part of interpreting binary protocols correctly. It comes into play whenever a multi-octet value, such as a 32-bit word, is transferred. The endian is an artifact of how computers store data in memory.

Because octets are transmitted sequentially on the network, it's possible to send the most significant octet of a value as the first part of the transmission, as well as the reverse—send the least significant octet first. The order in which octets are sent determines the endianness of the data. Failure to correctly handle the endian format can lead to subtle bugs in the parsing of protocols.

Modern platforms use two main endian formats: big and little. *Big endian* stores the most significant byte at the lowest address, whereas *little endian* stores the least significant byte in that location. Figure 3-5 shows how the 32-bit integer 0x01020304 is stored in both forms.

Lowest address   Highest address

0x01020304
as 32-bit | 0x01 | 0x02 | 0x03 | 0x04 |
big endian word

0x01020304
as 32-bit | 0x04 | 0x03 | 0x02 | 0x01 |
little endian word
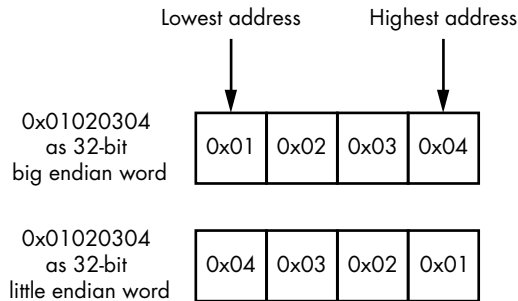
Figure 3-5: Big and little endian word representation

The endianness of a value is commonly referred to as either *network order* or *host order*. Because the Internet RFCs invariably use big endian as the preferred type for all network protocols they specify (unless there are legacy reasons for doing otherwise), big endian is referred as network order. But your computer could be either big or little endian. Processor architectures such as x86 use little endian; others such as SPARC use big endian.

NOTE   *Some processor architectures, including SPARC, ARM, and MIPS, may have onboard logic that specifies the endianness at runtime, usually by toggling a processor control flag. When developing network software, make no assumptions about the endianness of the platform you might be running on. The networking API used to build an application will typically contain convenience functions for converting to and from these orders. Other platforms, such as PDP-11, use a middle endian format where 16-bit words are swapped; however, you're unlikely to ever encounter one in everyday life, so don't dwell on it.*

Text and Human-Readable Data

Along with numeric data, strings are the value type you'll most commonly encounter, whether they're being used for passing authentication credentials or resource paths. When inspecting a protocol designed to send only

English characters, the text will probably be encoded using ASCII. The original ASCII standard defined a 7-bit character set from 0 to 0x7F, which includes most of the characters needed to represent the English language (shown in Figure 3-6).

	Control character	Printable character

Lower 4 bits

Upper 4 bits		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
	1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
	2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
	3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	7	p	q	r	s	t	u	v	w	x	y	z	{			}	~	DEL

Figure 3-6: A 7-bit ASCII table

The ASCII standard was originally developed for text terminals (physical devices with a moving printing head). Control characters were used to send messages to the terminal to move the printing head or to synchronize serial communications between the computer and the terminal. The ASCII character set contains two types of characters: *control* and *printable*. Most of the control characters are relics of those devices and are virtually unused. But some still provide information on modern computers, such as CR and LF, which are used to end lines of text.

The printable characters are the ones you can see. This set of characters consists of many familiar symbols and alphanumeric characters; however, they won't be of much use if you want to represent international characters, of which there are thousands. It's unachievable to represent even a fraction of the possible characters in all the world's languages in a 7-bit number.

Three strategies are commonly employed to counter this limitation: code pages, multibyte character sets, and Unicode. A protocol will either require that you use one of these three ways to represent text, or it will offer an option that an application can select.

Code Pages

The simplest way to extend the ASCII character set is by recognizing that if all your data is stored in octets, 128 unused values (from 128 to 255) can be repurposed for storing extra characters. Although 256 values are not enough to store all the characters in every available language, you have many different ways to use the unused range. Which characters are mapped to which values is typically codified in specifications called *code pages* or *character encodings*.

Multibyte Character Sets

In languages such as Chinese, Japanese, and Korean (collectively referred to as CJK), you simply can't come close to representing the entire written language with 256 characters, even if you use all available space. The solution is to use multibyte character sets combined with ASCII to encode these languages. Common encodings are Shift-JIS for Japanese and GB2312 for simplified Chinese.

 Multibyte character sets allow you to use two or more octets in sequence to encode a desired character, although you'll rarely see them in use. In fact, if you're not working with CJK, you probably won't see them at all. (For the sake of brevity, I won't discuss multibyte character sets any further; plenty of online resources will aid you in decoding them if required.)

Unicode

The Unicode standard, first standardized in 1991, aims to represent all languages within a unified character set. You might think of Unicode as another multibyte character set. But rather than focusing on a specific language, such as Shift-JIS does with Japanese, it tries to encode all written languages, including some archaic and constructed ones, into a single universal character set.

 Unicode defines two related concepts: *character mapping* and *character encoding*. Character mappings include mappings between a numeric value and a character, as well as many other rules and regulations on how characters are used or combined. Character encodings define the way these numeric values are encoded in the underlying file or network protocol. For analysis purposes, it's far more important to know how these numeric values are encoded.

 Each character in Unicode is assigned a *code point* that represents a unique character. Code points are commonly written in the format *U+ABCD*, where *ABCD* is the code point's hexadecimal value. For the sake of compatibility, the first 128 code points match what is specified in ASCII, and the second 128 code points are taken from ISO/IEC 8859-1. The resulting value is encoded using a specific scheme, sometimes referred to as *Universal Character Set (UCS)* or *Unicode Transformation Format (UTF)* encodings. (Subtle differences exist between UCS and UTF formats,

but for the sake of identification and manipulation, these differences are unimportant.) Figure 3-7 shows a simple example of some different Unicode formats.

Code points: Hello = U+0048 - U+0065 - U+006C - U+006C - U+006F

UCS-2/UTF-16 Little endian

0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F	0x00

UCS-2/UTF-16 Big endian

0x00	0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F

UCS-4/UTF-32 Little endian

0x48	0x00	0x00	0x00	0x65	0x00	0x00	0x00	0x6C	0x00	0x00	0x00

0x6C	0x00	0x00	0x00	0x6F	0x00	0x00	0x00

UTF-8

0x48	0x65	0x6C	0x6C	0x6F

Figure 3-7: The string "Hello" in different Unicode encodings

Three common Unicode encodings in use are UTF-16, UTF-32, and UTF-8.

UCS-2/UTF-16

UCS-2/UTF-16 is the native format on modern Microsoft Windows platforms, as well as the Java and .NET virtual machines when they are running code. It encodes code points in sequences of 16-bit integers and has little and big endian variants.

UCS-4/UTF-32

UCS-4/UTF-32 is a common format used in Unix applications because it's the default wide-character format in many C/C++ compilers. It encodes code points in sequences of 32-bit integers and has different endian variants.

UTF-8

UTF-8 is probably the most common format on Unix. It is also the default input and output format for varying platforms and technologies, such as XML. Rather than having a fixed integer size for code points, it encodes them using a simple variable length value. Table 3-3 shows how code points are encoded in UTF-8.

Table 3-3: Encoding Rules for Unicode Code Points in UTF-8

Bits of code point	First code point (U+)	Last code point (U+)	Byte 1	Byte 2	Byte 3	Byte 4
0–7	0000	007F	0xxxxxxx			
8–11	0080	07FF	110xxxxx	10xxxxxx		
12–16	0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx	
17–21	10000	1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx
22–26	200000	3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx
26–31	4000000	7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx

UTF-8 has many advantages. For one, its encoding definition ensures that the ASCII character set, code points U+0000 through U+007F, are encoded using single bytes. This scheme makes this format not only ASCII compatible but also space efficient. In addition, UTF-8 is compatible with C/C++ programs that rely on NUL-terminated strings.

For all of its benefits, UTF-8 does come at a cost, because languages like Chinese and Japanese consume more space than they do in UTF-16. Figure 3-8 shows such a disadvantageous encoding of Chinese characters. But notice that the UTF-8 in this example is still more space efficient than the UTF-32 for the same characters.

Code points: 兔子 = U+5154 - U+5B50

UCS-2/UTF-16 Little endian

0x54	0x51	0x50	0x5B

UCS-2/UTF-16 Big endian

0x51	0x54	0x5B	0x50

UCS-4/UTF-32 Little endian

0x54	0x51	0x00	0x00	0x50	0x5B	0x00	0x00

UTF-8

0xE5	0x85	0x94	0xE5	0xAD	0x90

Figure 3-8: The string "兔子" in different Unicode encodings

NOTE *Incorrect or naive character encoding can be a source of subtle security issues, ranging from bypassing filtering mechanisms (say in a requested resource path) to causing buffer overflows. We'll investigate some of the vulnerabilities associated with character encoding in Chapter 10.*

Variable Binary Length Data

If the protocol developer knows in advance exactly what data must be transmitted, they can ensure that all values within the protocol are of a fixed length. In reality this is quite rare, although even simple authentication credentials would benefit from the ability to specify variable username and password string lengths. Protocols use several strategies to produce variable-length data values: I discuss the most common—terminated data, length-prefixed data, implicit-length data, and padded data—in the following sections.

Terminated Data

You saw an example of variable-length data when variable-length integers were discussed earlier in this chapter. The variable-length integer value was terminated when the octet's MSB was 0. We can extend the concept of terminating values further to elements like strings or data arrays.

A terminated data value has a terminal symbol defined that tells the data parser that the end of the data value has been reached. The terminal symbol is used because it's unlikely to be present in typical data, ensuring that the value isn't terminated prematurely. With string data, the terminating value can be a NUL value (represented by 0) or one of the other control characters in the ASCII set.

If the terminal symbol chosen occurs during normal data transfer, you need to use a mechanism to escape these symbols. With strings, it's common to see the terminating character either prefixed with a backslash (\) or repeated twice to prevent it from being identified as the terminal symbol. This approach is especially useful when a protocol doesn't know ahead of time how long a value is—for example, if it's generated dynamically. Figure 3-9 shows an example of a string terminated by a NUL value.
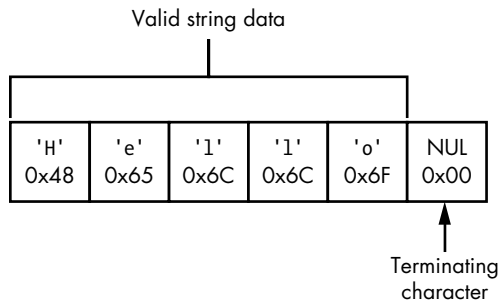
Figure 3-9: "Hello" as a NUL-terminated string

Bounded data is often terminated by a symbol that matches the first character in the variable-length sequence. For example, when using string data, you might find a *quoted string* sandwiched between quotation marks. The initial double quote tells the parser to look for the matching character to end the data. Figure 3-10 shows a string bounded by a pair of double quotes.
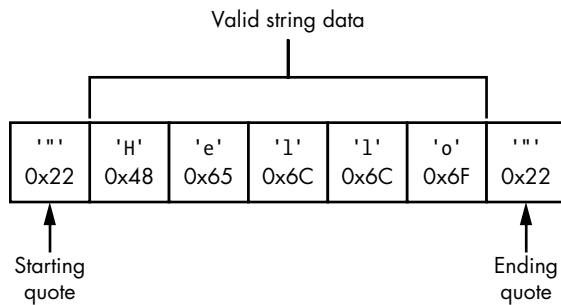
Figure 3-10: "Hello" as a double-quoted bounded string

Length-Prefixed Data

If a data value is known in advance, it's possible to insert its length into the protocol directly. The protocol's parser can read this value and then read the appropriate number of units (say characters or octets) to extract the original value. This is a very common way to specify variable-length data.

The actual size of the *length prefix* is usually not that important, although it should be reasonably representative of the types of data being transmitted. Most protocols won't need to specify the full range of a 32-bit integer; however, you'll often see that size used as a length field, if only because it fits well with most processor architectures and platforms. For example, Figure 3-11 shows a string with an 8-bit length prefix.
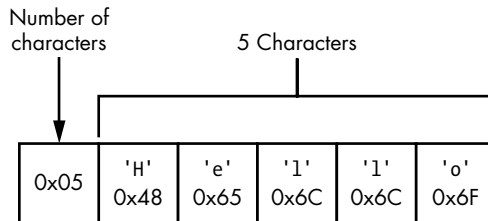
Figure 3-11: "Hello" as a length-prefixed string

Implicit-Length Data

Sometimes the length of the data value is implicit in the values around it. For example, think of a protocol that is sending data back to a client using a connection-oriented protocol such as TCP. Rather than specifying the size of the data up front, the server could close the TCP connection, thus implicitly signifying the end of the data. This is how data is returned in an HTTP version 1.0 response.

Another example would be a higher-level protocol or structure that has already specified the length of a set of values. The parser might extract that higher-level structure first and then read the values contained within it. The protocol could use the fact that this structure has a finite length associated with it to implicitly calculate the length of a value in a similar

fashion to close the connection (without closing it, of course). For example, Figure 3-12 shows a trivial example where a 7-bit variable integer and string are contained within a single block. (Of course, in practice, this can be considerably more complex.)
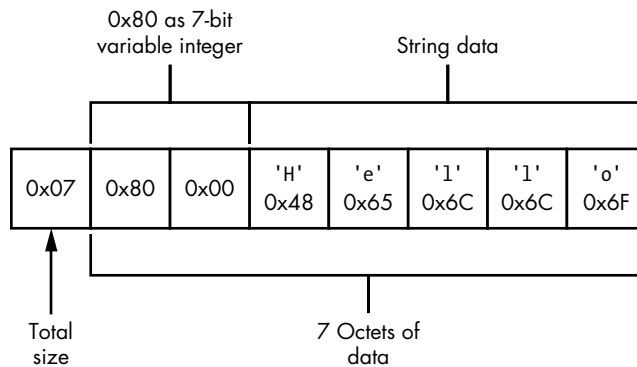
Figure 3-12: "Hello" as an implicit-length string

Padded Data

Padded data is used when there is a maximum upper bound on the length of a value, such as a 32-octet limit. For the sake of simplicity, rather than prefixing the value with a length or having an explicit terminating value, the protocol could instead send the entire fixed-length string but terminate the value by padding the unused data with a known value. Figure 3-13 shows an example.
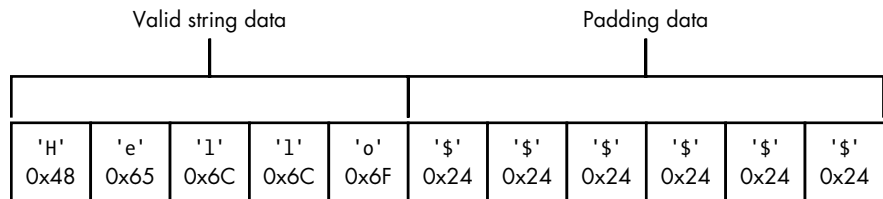
Figure 3-13: "Hello" as a '$' padded string

Dates and Times

It can be very important for a protocol to get the correct date and time. Both can be used as metadata, such as file modification timestamps in a network file protocol, as well as to determine the expiration of authentication credentials. Failure to correctly implement the timestamp might cause serious security issues. The method of date and time representation depends on usage requirements, the platform the applications are running on, and the protocol's space requirements. I discuss two common representations, POSIX/Unix Time and Windows FILETIME, in the following sections.

POSIX/Unix Time

Currently, POSIX/Unix time is stored as a 32-bit signed integer value representing the number of seconds that have elapsed since the Unix epoch, which is usually specified as 00:00:00 (UTC), 1 January 1970. Although this isn't a high-definition timer, it's sufficient for most scenarios. As a 32-bit integer, this value is limited to 03:14:07 (UTC) 19 January 2038, at which point the representation will overflow. Some modern operating systems now use a 64-bit representation to address this problem.

Windows FILETIME

The Windows FILETIME is the date and time format used by Microsoft Windows for its filesystem timestamps. As the only format on Windows with simple binary representation, it also appears in a few different protocols.

The FILETIME format is a 64-bit unsigned integer. One unit of the integer represents a 100 ns interval. The epoch of the format is 00:00:00 (UTC), 1 January 1601. This gives the FILETIME format a larger range than the POSIX/Unix time format.

Tag, Length, Value Pattern

It's easy to imagine how one might send unimportant data using simple protocols, but sending more complex and important data takes some explaining. For example, a protocol that can send different types of structures must have a way to represent the bounds of a structure and its type.

One way to represent data is with a *Tag, Length, Value (TLV) pattern.* The Tag value represents the type of data being sent by the protocol, which is commonly a numeric value (usually an enumerated list of possible values). But the Tag can be anything that provides the data structures with a unique pattern. The Length and Value are variable-length values. The order in which the values appear isn't important; in fact, the Tag might be part of the Value. Figure 3-14 show a couple of ways these values could be arranged.

The Tag value sent can be used to determine how to further process the data. For example, given two types of Tags, one that indicates the authentication credentials to the application and another that represents a message being transmitted to the parser, we must be able to distinguish between the two types of data. One big advantage to this pattern is that it allows us to extend a protocol without breaking applications that have not been updated to support the updated protocol. Because each structure is sent with an associated Tag and Length, a protocol parser could ignore the structures that it doesn't understand.
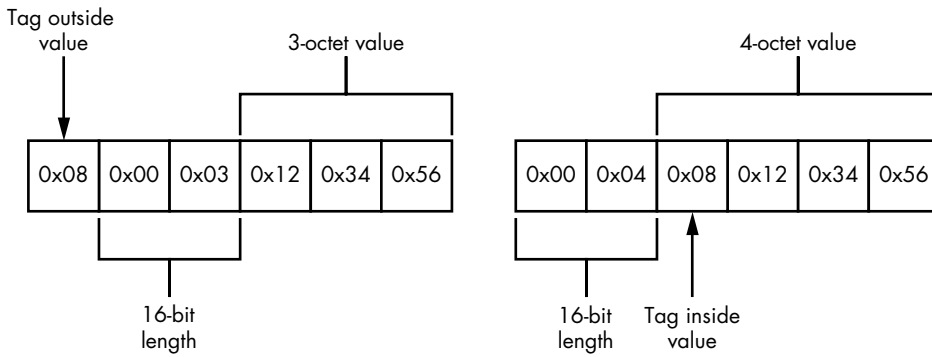
Figure 3-14: Possible TLV arrangements

Multiplexing and Fragmentation

Often in computer communication, multiple tasks must happen at once. For example, consider the Microsoft *Remote Desktop Protocol (RDP)*: a user could be moving the mouse cursor, typing on the keyboard, and transferring files to a remote computer while changes in the display and audio are being transmitted back to the user (see Figure 3-15).
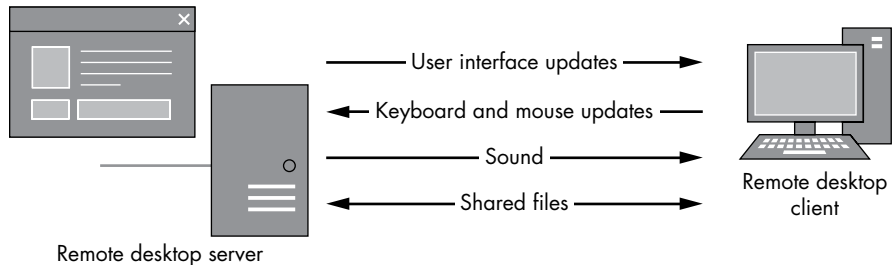
Figure 3-15: Data needs for Remote Desktop Protocol

This complex data transfer would not result in a very rich experience if display updates had to wait for a 10-minute audio file to finish before updating the display. Of course, a workaround would be opening multiple connections to the remote computer, but those would use more resources. Instead, many protocols use *multiplexing,* which allows multiple connections to share the same underlying network connection.

Multiplexing (shown in Figure 3-16) defines an internal *channel* mechanism that allows a single connection to host multiple types of traffic by fragmenting large transmissions into smaller chunks. Multiplexing then combines these chunks into a single connection. When analyzing a protocol, you may need to demultiplex these channels to get the original data back out.
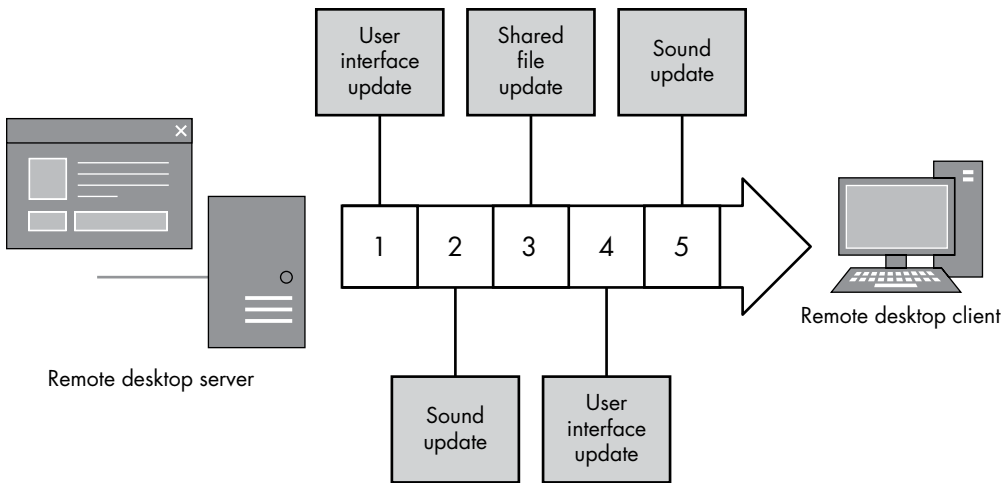
Figure 3-16: Multiplexed RDP data

Unfortunately, some network protocols restrict the type of data that can be transmitted and how large each packet of data can be—a problem commonly encountered when layering protocols. For example, Ethernet defines the maximum size of traffic frames as 1500 octets, and running IP on top of that causes problems because the maximum size of IP packets can be 65536 bytes. *Fragmentation* is designed to solve this problem: it uses a mechanism that allows the network stack to convert large packets into smaller fragments when the application or OS knows that the entire packet cannot be handled by the next layer.

Network Address Information

The representation of network address information in a protocol usually follows a fairly standard format. Because we're almost certainly dealing with TCP or UDP protocols, the most common binary representation is the IP address as either a 4- or 16-octet value (for IPv4 or IPv6) along with a 2-octet port. By convention, these values are typically stored as big endian integer values.

You might also see hostnames sent instead of raw addresses. Because hostnames are just strings, they follow the patterns used for sending variable-length strings, which was discussed earlier in "Variable Binary Length Data" on page 47. Figure 3-17 shows how some of these formats might appear.
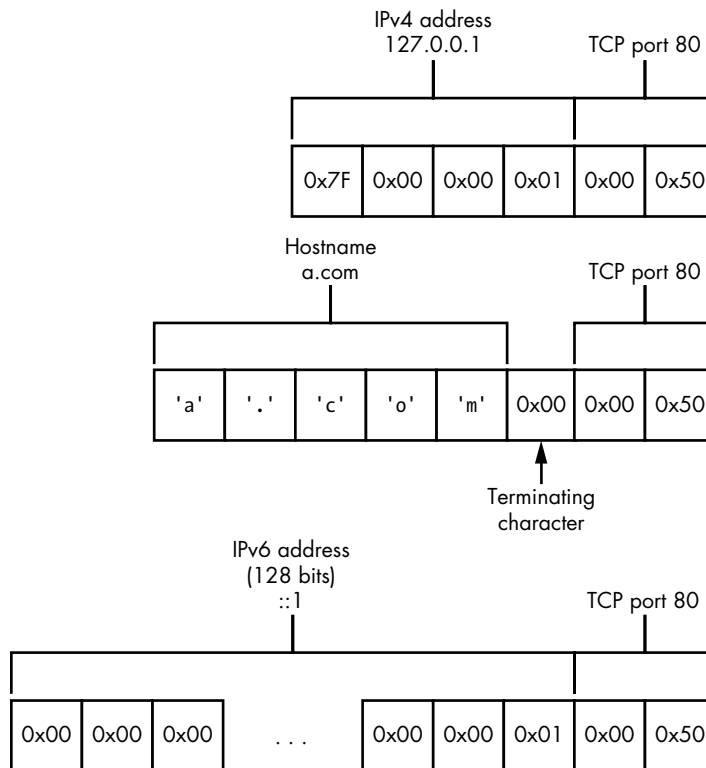
Figure 3-17: Network information in binary

Structured Binary Formats

Although custom network protocols have a habit of reinventing the wheel, sometimes it makes more sense to repurpose existing designs when describing a new protocol. For example, one common format encountered in binary protocols is *Abstract Syntax Notation 1 (ASN.1)*. ASN.1 is the basis for protocols such as the Simple Network Management Protocol (SNMP), and it is the encoding mechanism for all manner of cryptographic values, such as X.509 certificates.

ASN.1 is standardized by the ISO, IEC, and ITU in the X.680 series. It defines an abstract syntax to represent structured data. Data is represented in the protocol depending on the encoding rules, and numerous encodings exist. But you're most likely to encounter the *Distinguished Encoding Rules (DER)*, which is designed to represent ASN.1 structures in a way that cannot be misinterpreted—a useful property for cryptographic protocols. The DER representation is a good example of a TLV protocol.

Rather than going into great detail about ASN.1 (which would take up a fair amount of this book), I give you Listing 3-1, which shows the ASN.1 for X.509 certificates.

```
Certificate  ::=  SEQUENCE  {
    version         [0]  EXPLICIT Version DEFAULT v1,
    serialNumber         CertificateSerialNumber,
    signature            AlgorithmIdentifier,
    issuer               Name,
    validity             Validity,
    subject              Name,
    subjectPublicKeyInfo SubjectPublicKeyInfo,
    issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
    subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
    extensions      [3]  EXPLICIT Extensions OPTIONAL
}
```

Listing 3-1: ASN.1 representation for X.509 certificates

This abstract definition of an X.509 certificate can be represented in any of ASN.1's encoding formats. Listing 3-2 shows a snippet of the DER encoded form dumped as text using the OpenSSL utility.

```
$ openssl asn1parse -in example.cer
    0:d=0  hl=4 l= 539 cons: SEQUENCE
    4:d=1  hl=4 l= 388 cons: SEQUENCE
    8:d=2  hl=2 l=   3 cons: cont [ 0 ]
   10:d=3  hl=2 l=   1 prim: INTEGER          :02
   13:d=2  hl=2 l=  16 prim: INTEGER          :19BB8E9E2F7D60BE48BFE6840B50F7C3
   31:d=2  hl=2 l=  13 cons: SEQUENCE
   33:d=3  hl=2 l=   9 prim: OBJECT           :sha1WithRSAEncryption
   44:d=3  hl=2 l=   0 prim: NULL
   46:d=2  hl=2 l=  17 cons: SEQUENCE
   48:d=3  hl=2 l=  15 cons: SET
   50:d=4  hl=2 l=  13 cons: SEQUENCE
   52:d=5  hl=2 l=   3 prim: OBJECT           :commonName
   57:d=5  hl=2 l=   6 prim: PRINTABLESTRING :democa
```

Listing 3-2: A small sample of X.509 certificate

Text Protocol Structures

Text protocols are a good choice when the main purpose is to transfer text, which is why mail transfer protocols, instant messaging, and news aggregation protocols are usually text based. Text protocols must have structures similar to binary protocols. The reason is that, although their main content differs, both share the goal of transferring data from one place to another.

The following section details some common text protocol structures that you'll likely encounter in the real world.

Numeric Data

Over the millennia, science and written languages have invented ways to represent numeric values in textual format. Of course, computer protocols don't need to be human readable, but why go out of your way just to prevent a protocol from being readable (unless your goal is deliberate obfuscation).

Integers

It's easy to represent integer values using the current character set's representation of the characters 0 through 9 (or A through F if hexadecimal). In this simple representation, size limitations are no concern, and if a number needs to be larger than a binary word size, you can add digits. Of course, you'd better hope that the protocol parser can handle the extra digits or security issues will inevitably occur.

To make a signed number, you add the minus (-) character to the front of the number; the plus (+) symbol for positive numbers is implied.

Decimal Numbers

Decimal numbers are usually defined using human-readable forms. For example, you might write a number as 1.234, using the dot character to separate the integer and fractional components of the number; however, you'll still need to consider the requirement of parsing a value afterward.

Binary representations, such as floating point, can't represent all decimal values precisely with finite precision (just as decimals can't represent numbers like 1/3). This fact can make some values difficult to represent in text format and can cause security issues, especially when values are compared to one another.

Text Booleans

Booleans are easy to represent in text protocols. Usually, they're represented using the words *true* or *false*. But just to be difficult, some protocols might require that words be capitalized exactly to be valid. And sometimes integer values will be used instead of words, such as 0 for false and 1 for true, but not very often.

Dates and Times

At a simple level, it's easy to encode dates and times: just represent them as they would be written in a human-readable language. As long as all applications agree on the representation, that should suffice.

Unfortunately, not everyone can agree on a standard format, so typically many competing date representations are in use. This can be a particularly acute issue in applications such as mail clients, which need to process all manner of international date formats.

Variable-Length Data

All but the most trivial protocols must have a way to separate important text fields so they can be easily interpreted. When a text field is separated out of the original protocol, it's commonly referred to as a *token*. Some protocols specify a fixed length for tokens, but it's far more common to require some type of variable-length data.

Delimited Text

Separating tokens with delimiting characters is a very common way to separate tokens and fields that's simple to understand and easy to construct and parse. Any character can be used as the delimiter (depending on the type of data being transferred), but whitespace is encountered most in human-readable formats. That said, the delimiter doesn't have to be whitespace. For example, the Financial Information Exchange (FIX) protocol delimits tokens using the ASCII Start of Header (SOH) character with a value of 1.

Terminated Text

Protocols that specify a way to separate individual tokens must also have a way to define an End of Command condition. If a protocol is broken into separate lines, the lines must be terminated in some way. Most well-known, text-based Internet protocols are *line oriented*, such as HTTP and IRC; lines typically delimit entire structures, such as the end of a command.

What constitutes the end-of-line character? That depends on whom you ask. OS developers usually define the end-of-line character as either the ASCII *Line Feed (LF)*, which has the value 10; the *Carriage Return (CR)* with the value 13; or the combination CR LF. Protocols such as HTTP and Simple Mail Transfer Protocol (SMTP) specify CR LF as the official end-of-line combination. However, so many incorrect implementations occur that most parsers will also accept a bare LF as the end-of-line indication.

Structured Text Formats

As with structured binary formats such ASN.1, there is normally no reason to reinvent the wheel when you want to represent structured data in a text protocol. You might think of structured text formats as delimited text on steroids, and as such, rules must be in place for how values are represented and hierarchies constructed. With this in mind, I'll describe three formats in common use within real-world text protocols.

Multipurpose Internet Mail Extensions

Originally developed for sending multipart email messages, *Multipurpose Internet Mail Extensions (MIME)* found its way into a number of protocols, such as HTTP. The specification in RFCs 2045, 2046 and 2047, along with numerous other related RFCs, defines a way of encoding multiple discrete attachments in a single MIME-encoded message.

MIME messages separate the body parts by defining a common separator line prefixed with two dashes (--). The message is terminated by following this separator with the same two dashes. Listing 3-3 shows an example of a text message combined with a binary version of the same message.

```
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=MSG_2934894829

This is a message with multiple parts in MIME format.
--MSG_2934894829
Content-Type: text/plain

Hello World!
--MSG_2934894829
Content-Type: application/octet-stream
Content-Transfer-Encoding: base64

PGhObWw+Cjxib2R5PgpIZWxsbyBXb3JsZCEKPC9ib2R5Pgo8L2hObWw+Cg==
--MSG_2934894829--
```

Listing 3-3: A simple MIME message

One of the most common uses of MIME is for Content-Type values, which are usually referred to as *MIME types*. A MIME type is widely used when serving HTTP content and in operating systems to map an application to a particular content type. Each type consists of the form of the data it represents, such as *text* or *application*, in the format of the data. In this case, plain is unencoded text and octet-stream is a series of bytes.

JavaScript Object Notation

JavaScript Object Notation (JSON) was designed as a simple representation for a structure based on the object format provided by the JavaScript programming language. It was originally used to transfer data between a web page in a browser and a backend service, such as in Asynchronous JavaScript and XML (AJAX). Currently, it's commonly used for web service data transfer and all manner of other protocols.

The JSON format is simple: a JSON object is enclosed using the braces ({}) ASCII characters. Within these braces are zero or more member entries, each consisting of a key and a value. For example, Listing 3-4 shows a simple JSON object consisting of an integer index value, "Hello world!" as a string, and an array of strings.

```
{
    "index" : 0,
    "str" : "Hello World!",
    "arr" : [ "A", "B" ]
}
```

Listing 3-4: A simple JSON object

The JSON format was designed for JavaScript processing, and it can be parsed using the "eval" function. Unfortunately, using this function comes with a significant security risk; namely, it's possible to insert arbitrary script code during object creation. Although most modern applications use a parsing library that doesn't need a connection to JavaScript, it's worth ensuring that arbitrary JavaScript code is not executed in the context of the application. The reason is that it could lead to potential security issues, such as *cross-site scripting (XSS)*, a vulnerability where attacker-controlled JavaScript can be executed in the context of another web page, allowing the attacker to access the page's secure resources.

Extensible Markup Language

Extensible Markup Language (XML) is a markup language for describing a structured document format. Developed by the W3C, it's derived from Standard Generalized Markup Language (SGML). It has many similarities to HTML, but it aims to be stricter in its definition in order to simplify parsers and create fewer security issues.[1]

At a basic level, XML consists of elements, attributes, and text. *Elements* are the main structural values. They have a name and can contain child elements or text content. Only one root element is allowed in a single document. *Attributes* are additional name-value pairs that can be assigned to an element. They take the form of `name="Value"`. Text content is just that, text. Text is a child of an element or the value component of an attribute.

Listing 3-5 shows a very simple XML document with elements, attributes, and text values.

```
<value index="0">    <str>Hello World!</str>
    <arr><value>A</value><value>B</value></arr>
</value>
```

Listing 3-5: A simple XML document

All XML data is text; no type information is provided for in the XML specification, so the parser must know what the values represent. Certain specifications, such as XML Schema, aim to remedy this type information deficiency but they are not required in order to process XML content. The XML specification defines a list of well-formed criteria that can be used to determine whether an XML document meets a minimal level of structure.

XML is used in many different places to define the way information is transmitted in a protocol, such as in Rich Site Summary (RSS). It can also be part of a protocol, as in Extensible Messaging and Presence Protocol (XMPP).

---

1. Just ask those who have tried to parse HTML for errant script code how difficult that task can be without a strict format.

Encoding Binary Data

In the early history of computer communication, 8-bit bytes were not the norm. Because most communication was text based and focused on English-speaking countries, it made economic sense to send only 7 bits per byte as required by the ASCII standard. This allowed other bits to provide control for serial link protocols or to improve performance. This history is reflected heavily in some early network protocols, such as the SMTP or Network News Transfer Protocol (NNTP), which assume 7-bit communication channels.

But a 7-bit limitation presents a problem if you want to send that amusing picture to your friend via email or you want to write your mail in a non-English character set. To overcome this limitation, developers devised a number of ways to encode binary data as text, each with varying degrees of efficiency or complexity.

As it turns out, the ability to convert binary content into text still has its advantages. For example, if you wanted to send binary data in a structured text format, such as JSON or XML, you might need to ensure that delimiters were appropriately escaped. Instead, you can choose an existing encoding format, such as Base64, to send the binary data and it will be easily understood on both sides.

Let's look at some of the more common binary-to-text encoding schemes you're likely to encounter when inspecting a text protocol.

Hex Encoding

One of the most naive encoding techniques for binary data is *hex encoding*. In hex encoding, each octet is split into two 4-bit values that are converted to two text characters denoting the hexadecimal representation. The result is a simple representation of the binary in text form, as shown in Figure 3-18.
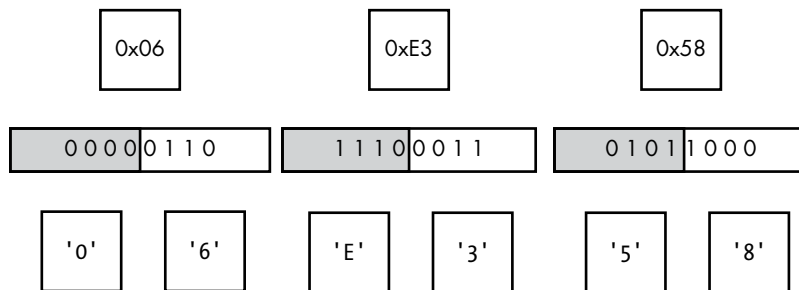
Figure 3-18: Example hex encoding of binary data

Although simple, hex encoding is not space efficient because all binary data automatically becomes 100 percent larger than it was originally. But one advantage is that encoding and decoding operations are fast and simple and little can go wrong, which is definitely beneficial from a security perspective.

HTTP specifies a similar encoding for URLs and some text protocols called *percent encoding*. Rather than all data being encoded, only nonprintable data is converted to hex, and values are signified by prefixing the value with a % character. If percent encoding was used to encode the value in Figure 3-18, you would get %06%E3%58.

Base64

To counter the obvious inefficiencies in hex encoding, we can use Base64, an encoding scheme originally developed as part of the MIME specifications. The *64* in the name refers to the number of characters used to encode the data.

The input binary is separated into individual 6-bit values, enough to represent 0 through 63. This value is then used to look up a corresponding character in an encoding table, as shown in Figure 3-19.

Lower 4 bits

		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Upper 2 bits	0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
	1	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
	2	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
	3	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/

Figure 3-19: Base64 encoding table

But there's a problem with this approach: when 8 bits are divided by 6, 2 bits remain. To counter this problem, the input is taken in units of three octets, because dividing 24 bits by 6 bits produces 4 values. Thus, Base64 encodes 3 bytes into 4, representing an increase of only 33 percent, which is significantly better than the increase produced by hex encoding. Figure 3-20 shows an example of encoding a three-octet sequence into Base64.

But yet another issue is apparent with this strategy. What if you have only one or two octets to encode? Would that not cause the encoding to fail? Base64 gets around this issue by defining a placeholder character, the equal sign (=). If in the encoding process, no valid bits are available to use, the encoder will encode that value as the placeholder. Figure 3-21 shows an example of only one octet being encoded. Note that it generates two placeholder characters. If two octets were encoded, Base64 would generate only one.
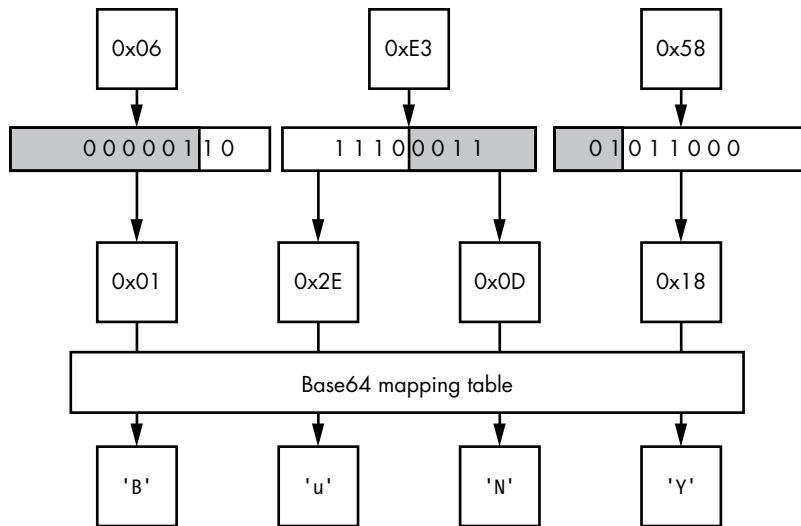
Figure 3-20: Base64 encoding 3 bytes as 4 characters
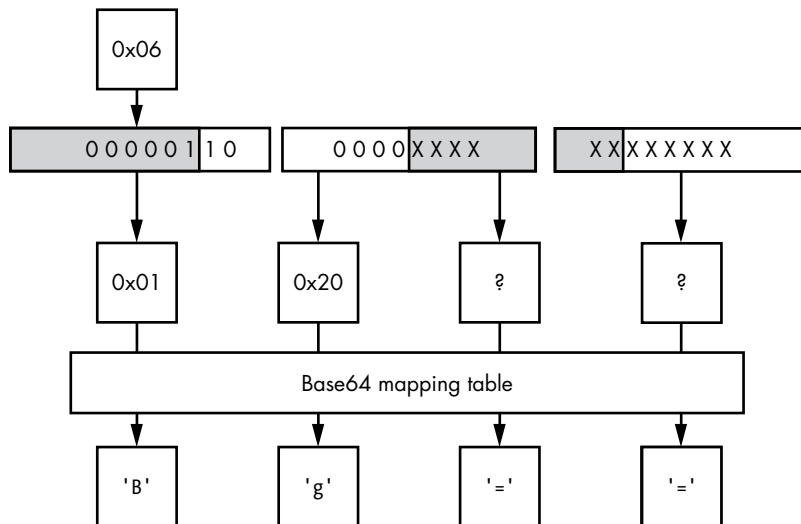
Figure 3-21: Base64 encoding 1 byte as 3 characters

To convert Base64 data back into binary, you simply follow the steps in reverse. But what happens when a non-Base64 character is encountered during the decoding? Well that's up to the application to decide. We can only hope that it makes a secure decision.

Final Words

In this chapter, I defined many ways to represent data values in binary and text protocols and discussed how to represent numeric data, such as integers, in binary. Understanding how octets are transmitted in a protocol is crucial to successfully decoding values. At the same time, it's also important to identify the many ways that variable-length data values can be represented because they are perhaps the most important structure you will encounter within a network protocol. As you analyze more network protocols, you'll see the same structures used repeatedly. Being able to quickly identify the structures is key to easily processing unknown protocols.

In Chapter 4, we'll look at a few real-world protocols and dissect them to see how they match up with the descriptions presented in this chapter.

4

ADVANCED APPLICATION TRAFFIC CAPTURE

Usually, the network traffic-capturing techniques you learned in Chapter 2 should suffice, but occasionally you'll encounter tricky situations that require more advanced ways to capture network traffic. Sometimes, the challenge is an embedded platform that can only be configured with the Dynamic Host Configuration Protocol (DHCP); other times, there may be a network that offers you little control unless you're directly connected to it.

Most of the advanced traffic-capturing techniques discussed in this chapter use existing network infrastructure and protocols to redirect traffic. None of the techniques require specialty hardware; all you'll need are software packages commonly found on various operating systems.

Rerouting Traffic

IP is a *routed* protocol; that is, none of the nodes on the network need to know the exact location of any other nodes. Instead, when one node wants to send traffic to another node that it isn't directly connected to, it sends the traffic to a *gateway* node, which forwards the traffic to the destination. A gateway is also commonly called a *router*, a device that routes traffic from one location to another.

For example, in Figure 4-1, the client 192.168.56.10 is trying to send traffic to the server 10.1.1.10, but the client doesn't have a direct connection to the server. It first sends traffic destined for the server to Router A. In turn, Router A sends the traffic to Router B, which has a direct connection to the target server; Router B passes the traffic on to its final destination.

As with all nodes, the gateway node doesn't know the traffic's exact destination, so it looks up the appropriate next gateway to send to. In this case, Routers A and B only know about the two networks they are directly connected to. To get from the client to the server, the traffic must be routed.
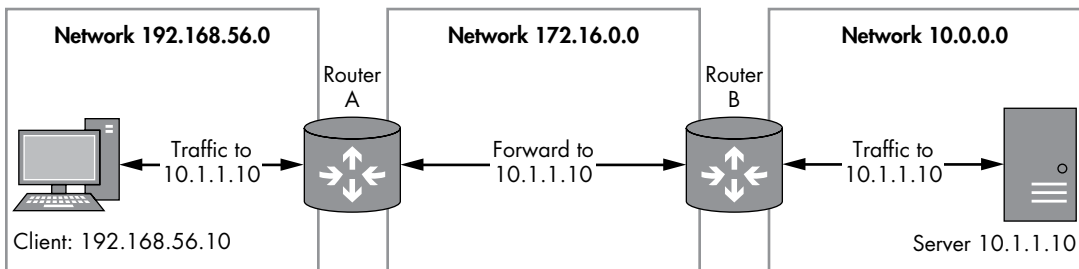
Figure 4-1: An example of routed traffic

Using Traceroute

When tracing a route, you attempt to map the route that the IP traffic will take to a particular destination. Most operating systems have built-in tools to perform a trace, such as traceroute on most Unix-like platforms and tracert on Windows.

Listing 4-1 shows the result of tracing the route to *www.google.com* from a home internet connection.

```
C:\Users\user>tracert www.google.com

Tracing route to www.google.com [173.194.34.176]
over a maximum of 30 hops:

  1     2 ms     2 ms     2 ms  home.local [192.168.1.254]
  2    15 ms    15 ms    15 ms  217.32.146.64
  3    88 ms    15 ms    15 ms  217.32.146.110
  4    16 ms    16 ms    15 ms  217.32.147.194
  5    26 ms    15 ms    15 ms  217.41.168.79
  6    16 ms    26 ms    16 ms  217.41.168.107
```

```
 7    26 ms    15 ms    15 ms  109.159.249.94
 8    18 ms    16 ms    15 ms  109.159.249.17
 9    17 ms    28 ms    16 ms  62.6.201.173
10    17 ms    16 ms    16 ms  195.99.126.105
11    17 ms    17 ms    16 ms  209.85.252.188
12    17 ms    17 ms    17 ms  209.85.253.175
13    27 ms    17 ms    17 ms  lhr14s22-in-f16.1e100.net [173.194.34.176]
```

Listing 4-1: Traceroute to www.google.com using the tracert tool

Each numbered line of output (1, 2, and so on) represents a unique gateway routing traffic to the ultimate destination. The output refers to a maximum number of *hops*. A single hop represents the network between each gateway in the entire route. For example, there's a hop between your machine and the first router, another between that router and the next, and hops all the way to the final destination. If the maximum hop count is exceeded, the traceroute process will stop probing for more routers. The maximum hop can be specified to the trace route tool command line; specify -h NUM on Windows and -m NUM on Unix-style systems.(The output also shows the round-trip time from the machine performing the traceroute and the discovered node.)

Routing Tables

The OS uses *routing tables* to figure out which gateways to send traffic to. A routing table contains a list of destination networks and the gateway to route traffic to. If a network is directly connected to the node sending the network traffic, no gateway is required, and the network traffic can be transmitted directly on the local network.

You can view your computer's routing table by entering the command netstat -r on most Unix-like systems or route print on Windows. Listing 4-2 shows the output from Windows when you execute this command.

```
> route print
```

```
IPv4 Route Table
===========================================================================
Active Routes:
Network Destination        Netmask          Gateway       Interface  Metric
❶          0.0.0.0          0.0.0.0    192.168.1.254    192.168.1.72      10
          127.0.0.0        255.0.0.0         On-link         127.0.0.1     306
          127.0.0.1  255.255.255.255         On-link         127.0.0.1     306
    127.255.255.255  255.255.255.255         On-link         127.0.0.1     306
        192.168.1.0    255.255.255.0         On-link      192.168.1.72     266
       192.168.1.72  255.255.255.255         On-link      192.168.1.72     266
      192.168.1.255  255.255.255.255         On-link      192.168.1.72     266
          224.0.0.0        240.0.0.0         On-link         127.0.0.1     306
          224.0.0.0        240.0.0.0         On-link      192.168.56.1     276
          224.0.0.0        240.0.0.0         On-link      192.168.1.72     266
    255.255.255.255  255.255.255.255         On-link         127.0.0.1     306
    255.255.255.255  255.255.255.255         On-link      192.168.56.1     276
```

```
            255.255.255.255    255.255.255.255              On-link   192.168.1.72       266
===============================================================================
```

Listing 4-2: Example routing table output

As mentioned earlier, one reason routing is used is so that nodes don't need to know the location of all other nodes on the network. But what happens to traffic when the gateway responsible for communicating with the destination network isn't known? In that case, it's common for the routing table to forward all unknown traffic to a *default gateway*. You can see the default gateway at ❶, where the network destination is 0.0.0.0. This destination is a placeholder for the default gateway, which simplifies the management of the routing table. By using a placeholder, the table doesn't need to be changed if the network configuration changes, such as through a DHCP configuration. Traffic sent to any destination that has no known matching route will be sent to the gateway registered for the 0.0.0.0 placeholder address.

How can you use routing to your advantage? Let's consider an embedded system in which the operating system and hardware come as one single device. You might not be able to influence the network configuration in an embedded system as you might not even have access to the underlying operating system, but if you can present your capturing device as a gateway between the system generating the traffic and its ultimate destination, you can capture the traffic on that system.

The following sections discuss ways to configure an OS to act as a gateway to facilitate traffic capture.

Configuring a Router

By default, most operating systems do not route traffic directly between network interfaces. This is mainly to prevent someone on one side of the route from communicating directly with the network addresses on the other side. If routing is not enabled in the OS configuration, any traffic sent to one of the machine's network interfaces that needs to be routed is instead dropped or an error message is sent to the sender. The default configuration is very important for security: imagine the implications if the router controlling your connection to the internet routed traffic from the internet directly to your private network.

Therefore, to enable an OS to perform routing, you need to make some configuration changes as an administrator. Although each OS has different ways of enabling routing, one aspect remains constant: you'll need at least two separate network interfaces installed in your computer to act as a router. In addition, you'll need routes on both sides of the gateway for routing to function correctly. If the destination doesn't have a corresponding route back to the source device, communication might not work as expected. Once routing is enabled, you can configure the network devices

to forward traffic via your new router. By running a tool such as Wireshark on the router, you can capture traffic as it's forwarded between the two network interfaces you configured.

Enabling Routing on Windows

By default, Windows does not enable routing between network interfaces. To enable routing on Windows, you need to modify the system registry. You can do this by using a GUI registry editor, but the easiest way is to run the following command as an administrator from the command prompt:

```
C> reg add HKLM\System\CurrentControlSet\Services\Tcpip\Parameters ^
    /v IPEnableRouter /t REG_DWORD /d 1
```

To turn off routing after you've finished capturing traffic, enter the following command:

```
C> reg add HKLM\System\CurrentControlSet\Services\Tcpip\Parameters ^
    /v IPEnableRouter /t REG_DWORD /d 0
```

You'll also need to reboot between command changes.

WARNING *Be very careful when you're modifying the Windows registry. Incorrect changes could completely break Windows and prevent it from booting! Be sure to make a system backup using a utility like the built-in Windows backup tool before performing any dangerous changes.*

Enabling Routing on *nix

To enable routing on Unix-like operating systems, you simply change the IP routing system setting using the sysctl command. (Note that the instructions for doing so aren't necessarily consistent between systems, but you should be able to easily find specific instructions.)

To enable routing on Linux for IPv4, enter the following command as root (no need to reboot; the change is immediate):

```
# sysctl net.ipv4.conf.all.forwarding=1
```

To enable IPv6 routing on Linux, enter this:

```
# sysctl net.ipv6.conf.all.forwarding=1
```

You can revert the routing configuration by changing 1 to 0 in the previous commands.

To enable routing on macOS, enter the following:

```
> sysctl -w net.inet.ip.forwarding=1
```

Network Address Translation

When trying to capture traffic, you may find that you can capture outbound traffic but not returning traffic. The reason is that an upstream router doesn't know the route to the original source network; therefore, it either drops the traffic entirely or forwards it to an unrelated network. You can mitigate this situation by using *Network Address Translation (NAT)*, a technique that modifies the source and destination address information of IP and higher-layer protocols, such as TCP. NAT is used extensively to extend the limited IPv4 address space by hiding multiple devices behind a single public IP address.

NAT can make network configuration and security easier, too. When NAT is turned on, you can run as many devices behind a single NAT IP address as you like and manage only that public IP address.

Two types of NAT are common today: *Source NAT (SNAT)* and *Destination NAT (DNAT)*. The differences between the two relate to which address is modified during the NAT processing of the network traffic. SNAT (also called *masquerading*) changes the IP source address information; DNAT changes the destination address.

Enabling SNAT

When you want a router to hide multiple machines behind a single IP address, you use SNAT. When SNAT is turned on, as traffic is routed across the external network interface, the source IP address in the packets is rewritten to match the single IP address made available by SNAT.

It can be useful to implement SNAT when you want to route traffic to a network that you don't control because, as you'll recall, both nodes on the network must have appropriate routing information for network traffic to be sent between the nodes. In the worst case, if the routing information is incorrect, traffic will flow in only one direction. Even in the best case, it's likely that you would be able to capture traffic only in one direction; the other direction would be routed through an alternative path.

SNAT addresses this potential problem by changing the source address of the traffic to an IP address that the destination node can route to—typically, the one assigned to the external interface of the router. Thus, the destination node can send traffic back in the direction of the router. Figure 4-2 shows a simple example of SNAT.

Client (10.0.0.1) Router (1.1.1.1) Server (*domain.com*)

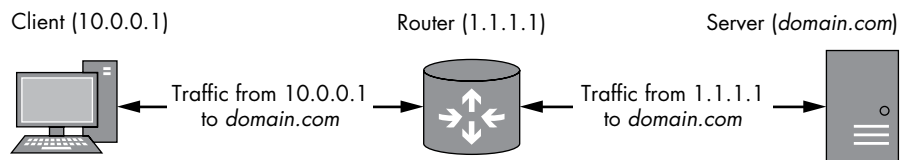Traffic from 10.0.0.1 to *domain.com*   Traffic from 1.1.1.1 to *domain.com*

Figure 4-2: An example of SNAT from a client to a server

When the client wants to send a packet to a server on a different network, it sends it to the router that has been configured with SNAT. When

the router receives the packet from the client, the source address is the client's (10.0.0.1) and the destination is the server (the resolved address of *domain.com*). It's at this point that SNAT is used: the router modifies the source address of the packet to its own (1.1.1.1) and then forwards the packet to the server.

When the server receives this packet, it assumes the packet came from the router; so, when it wants to send a packet back, it sends the packet to 1.1.1.1. The router receives the packet, determines it came from an existing NAT connection (based on destination address and port numbers), and reverts the address change, converting 1.1.1.1 back to the original client address of 10.0.0.1. Finally, the packet can be forwarded back to the original client without the server needing to know about the client or how to route to its network.

Configuring SNAT on Linux

Although you can configure SNAT on Windows and macOS using Internet Connection Sharing, I'll only provide details on how to configure SNAT on Linux because it's the easiest platform to describe and the most flexible when it comes to network configuration.

Before configuring SNAT, you need to do the following:

- Enable IP routing as described earlier in this chapter.
- Find the name of the outbound network interface on which you want to configure SNAT. You can do so by using the `ifconfig` command. The outbound interface might be named something like `eth0`.
- Note the IP address associated with the outbound interface when you use `ifconfig`.

Now you can configure the NAT rules using the `iptables`. (The `iptables` command is most likely already installed on your Linux distribution.) But first, flush any existing NAT rules in `iptables` by entering the following command as the root user:

```
# iptables -t nat -F
```

If the outbound network interface has a fixed address, run the following commands as root to enable SNAT. Replace *INTNAME* with the name of your outbound interface and *INTIP* with the IP address assigned to that interface.

```
# iptables -t nat -A POSTROUTING -o INTNAME -j SNAT --to INTIP
```

However, if the IP address is configured dynamically (perhaps using DHCP or a dial-up connection), use the following command to automatically determine the outbound IP address:

```
# iptables -t nat -A POSTROUTING -o INTNAME -j MASQUERADE
```

Enabling DNAT

DNAT is useful if you want to redirect traffic to a proxy or other service to terminate it, or before forwarding the traffic to its original destination. DNAT rewrites the destination IP address, and optionally, the destination port. You can use DNAT to redirect specific traffic to a different destination, as shown in Figure 4-3, which illustrates traffic being redirected from both the router and the server to a proxy at 192.168.0.10 to perform a man-in-the-middle analysis.
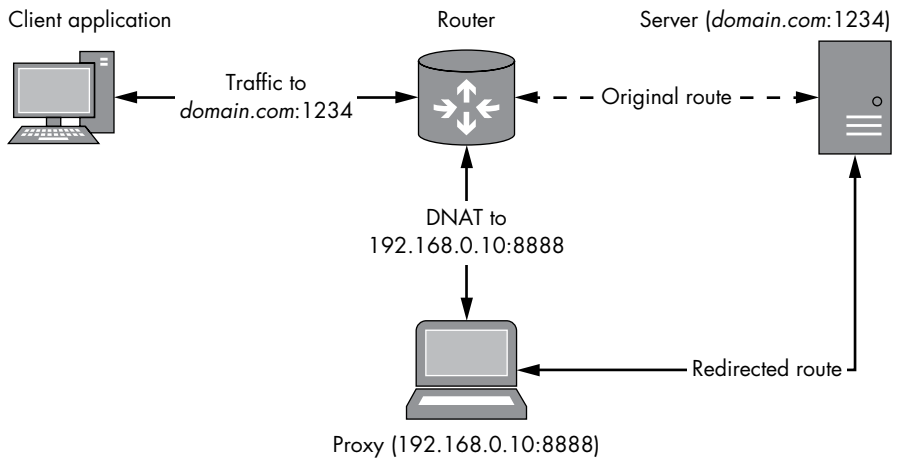
Figure 4-3: An example of DNAT to a proxy

Figure 4-3 shows a client application sending traffic through a router that is destined for *domain.com* on port 1234. When a packet is received at the router, that router would normally just forward the packet to the original destination. But because DNAT is used to change the packet's destination address and port to 192.168.0.10:8888, the router will apply its forwarding rules and send the packet to a proxy machine that can capture the traffic. The proxy then establishes a new connection to the server and forwards any packets sent from the client to the server. All traffic between the original client and the server can be captured and manipulated.

Configuring DNAT depends on the OS the router is running. (If your router is running Windows, you're probably out of luck because the functionality required to support it isn't exposed to the user.) Setup varies considerably between different versions of Unix-like operating systems and macOS, so I'll only show you how to configure DNAT on Linux. First, flush any existing NAT rules by entering the following command:

```
# iptables -t nat -F
```

Next, run the following command as the root user, replacing *ORIGIP* (originating IP) with the IP address to match traffic to and *NEWIP* with the new destination IP address you want that traffic to go to.

```
# iptables -t nat -A PREROUTING -d ORIGIP -j DNAT --to-destination NEWIP
```

The new NAT rule will redirect any packet routed to *ORIGIP* to *NEWIP*. (Because the DNAT occurs prior to the normal routing rules on Linux, it's safe to choose a local network address; the DNAT rule will not affect traffic sent directly from Linux.) To apply the rule only to a specific TCP or UDP, change the command:

```
iptables -t nat -A PREROUTING -p PROTO -d ORIGIP --dport ORIGPORT -j DNAT \
    --to-destination NEWIP:NEWPORT
```

The placeholder *PROTO* (for protocol) should be either tcp or udp depending on the IP protocol being redirected using the DNAT rule. The values for *ORIGIP* (original IP) and *NEWIP* are the same as earlier.

You can also configure *ORIGPORT* (the original port) and *NEWPORT* if you want to change the destination port. If *NEWPORT* is not specified, only the IP address will be changed.

Forwarding Traffic to a Gateway

You've set up your gateway device to capture and modify traffic. Everything appears to be working properly, but there's a problem: you can't easily change the network configuration of the device you want to capture. Also, you have limited ability to change the network configuration the device is connected to. You need some way to reconfigure or trick the sending device into forwarding traffic through your gateway. You could accomplish this by exploiting the local network by spoofing packets for either DHCP or *Address Resolution Protocol (ARP)*.

DHCP Spoofing

DHCP is designed to run on IP networks to distribute network configuration information to nodes automatically. Therefore, if we can spoof DHCP traffic, we can change a node's network configuration remotely. When DHCP is used, the network configuration pushed to a node can include an IP address as well as the default gateway, routing tables, the default DNS servers, and even additional custom parameters. If the device you want to test uses DHCP to configure its network interface, this flexibility makes it very easy to supply a custom configuration that will allow easy network traffic capture.

DHCP uses the UDP protocol to send requests to and from a DHCP service on the local network. Four types of DHCP packets are sent when negotiating the network configuration:

Discover   Sent to all nodes on the IP network to discover a DHCP server

Offer   Sent by the DHCP server to the node that sent the discovery packet to offer a network configuration

Request   Sent by the originating node to confirm its acceptance of the offer

Acknowledgment   Sent by the server to confirm completion of the configuration

The interesting aspect of DHCP is that it uses an unauthenticated, connectionless protocol to perform configuration. Even if an existing DHCP server is on a network, you may be able to spoof the configuration process and change the node's network configuration, including the default gateway address, to one you control. This is called *DHCP spoofing*.

To perform DHCP spoofing, we'll use *Ettercap*, a free tool that's available on most operating systems (although Windows isn't officially supported).

1. On Linux, start Ettercap in graphical mode as the root user:

```
# ettercap -G
```

You should see the Ettercap GUI, as shown in Figure 4-4.
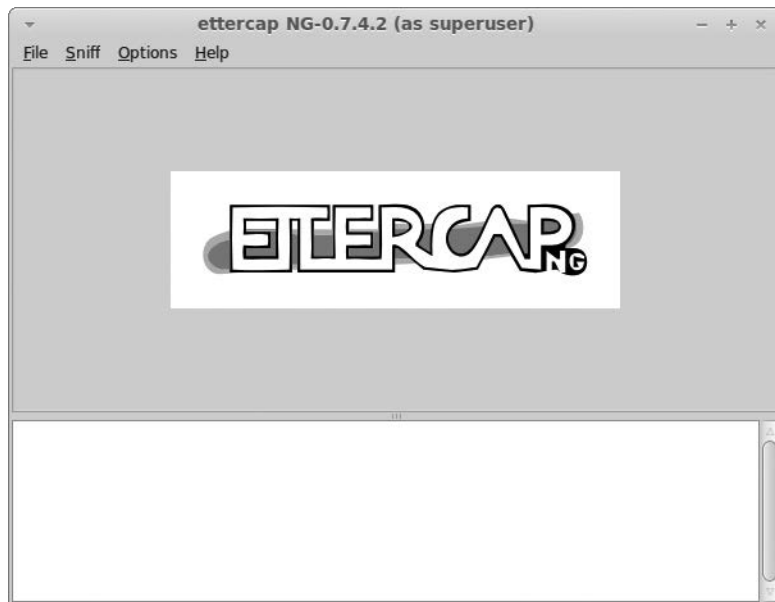
Figure 4-4: The main Ettercap GUI

2. Configure Ettercap's sniffing mode by selecting **Sniff ▸ Unified Sniffing**.

3. The dialog shown in Figure 4-5 should prompt you to select the network interface you want to sniff on. Select the interface connected to the network you want to perform DHCP spoofing on. (Make sure the network interface's network is configured correctly because Ettercap will automatically send the interface's configured IP address as the DHCP default gateway.)

Figure 4-5: Selecting the sniffing interface

4. Enable DHCP spoofing by choosing **Mitm ▸ Dhcp spoofing**. The dialog shown in Figure 4-6 should appear, allowing you to configure the DHCP spoofing options.
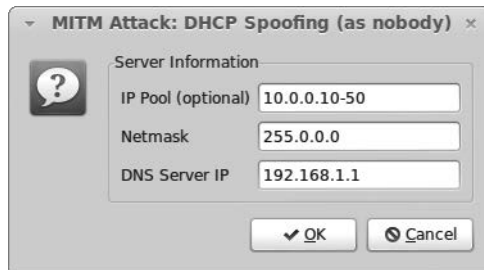
Figure 4-6: Configuring DHCP spoofing

5. The IP Pool field sets the range of IP addresses to hand out for spoofing DHCP requests. Supply a range of IP addresses that you configured for the network interface that is capturing traffic. For example, in Figure 4-6, the IP Pool value is set to 10.0.0.10-50 (the dash indicates all addresses inclusive of each value), so we'll hand out IPs from 10.0.0.10 to 10.0.0.50 inclusive. Configure the Netmask to match your network interface's netmask to prevent conflicts. Specify a DNS server IP of your choice.

6. Start sniffing by choosing **Start ▸ Start sniffing**. If DHCP spoofing is successful on the device, the Ettercap log window should look like Figure 4-7. The crucial line is `fake ACK` sent by Ettercap in response to the DHCP request.
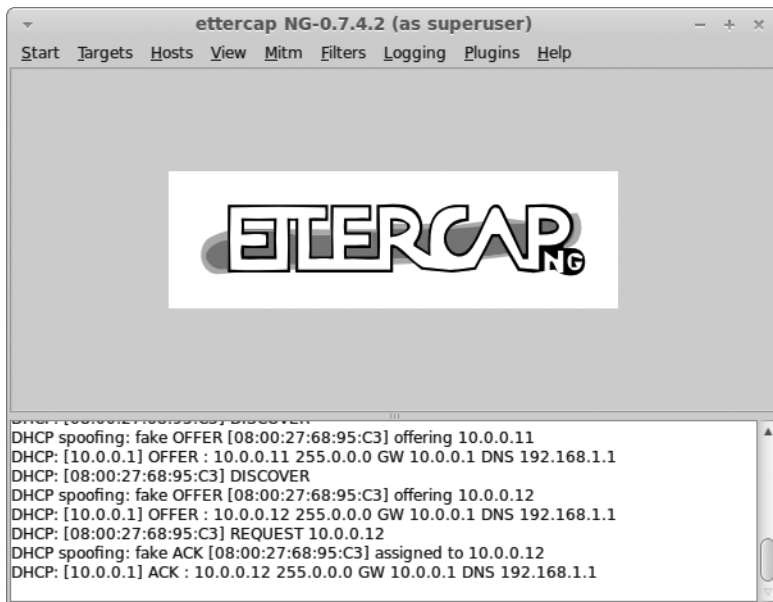
Figure 4-7: Successful DHCP spoofing

That's all there is to DHCP spoofing with Ettercap. It can be very powerful if you don't have any other option and a DHCP server is already on the network you're trying to attack.

ARP Poisoning

ARP is critical to the operation of IP networks running on Ethernet because ARP finds the Ethernet address for a given IP address. Without ARP, it would be very difficult to communicate IP traffic efficiently over Ethernet. Here's how ARP works: when one node wants to communicate with another on the same Ethernet network, it must be able to map the IP address to an Ethernet MAC address (which is how Ethernet knows the destination node to send traffic to). The node generates an ARP request packet (see Figure 4-8) containing the node's 6-byte Ethernet MAC address, its current IP address, and the target node's IP address. The packet is transmitted on the Ethernet network with a destination MAC address of ff:ff:ff:ff:ff:ff, which is the defined broadcast address. Normally, an Ethernet device only processes packets with a destination address that matches its address, but if it receives a packet with the destination MAC address set to the broadcast address, it will process it, too.

If one of the recipients of this broadcasted message has been assigned the target IP address, it can now return an ARP response, as shown in Figure 4-9. This response is almost exactly the same as the request except the sender and target fields are reversed. Because the sender's IP address should correspond to the original requested target IP address, the original

requestor can now extract the sender's MAC address and remember it for future network communication without having to resend the ARP request.

```
⊞ Frame 261: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Ethernet II, Src: CadmusCo_01:62:d7 (08:00:27:01:62:d7), Dst: Broadcast (ff:ff:ff:ff:ff:ff)
⊟ Address Resolution Protocol (request)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (1)
    Sender MAC address: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
    Sender IP address: 192.168.56.101 (192.168.56.101)
    Target MAC address: 00:00:00_00:00:00 (00:00:00:00:00:00)
    Target IP address: 192.168.56.1 (192.168.56.1)
```

Figure 4-8: An example ARP request packet

```
⊞ Frame 262: 42 bytes on wire (336 bits), 42 bytes captured (336 bits) on interface 0
⊞ Ethernet II, Src: CadmusCo_00:f4:8b (08:00:27:00:f4:8b), Dst: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
⊟ Address Resolution Protocol (reply)
    Hardware type: Ethernet (1)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: reply (2)
    Sender MAC address: CadmusCo_00:f4:8b (08:00:27:00:f4:8b)
    Sender IP address: 192.168.56.1 (192.168.56.1)
    Target MAC address: CadmusCo_01:62:d7 (08:00:27:01:62:d7)
    Target IP address: 192.168.56.101 (192.168.56.101)
```

Figure 4-9: An example ARP response

How can you use ARP poisoning to your advantage? As with DHCP, there's no authentication on ARP packets, which are intentionally sent to all nodes on the Ethernet network. Therefore, you can inform the target node you own an IP address and ensure the node forwards traffic to your rogue gateway by sending spoofed ARP packets to poison the target node's ARP cache. You can use Ettercap to spoof the packets, as shown in Figure 4-10.

Network 192.168.100.0
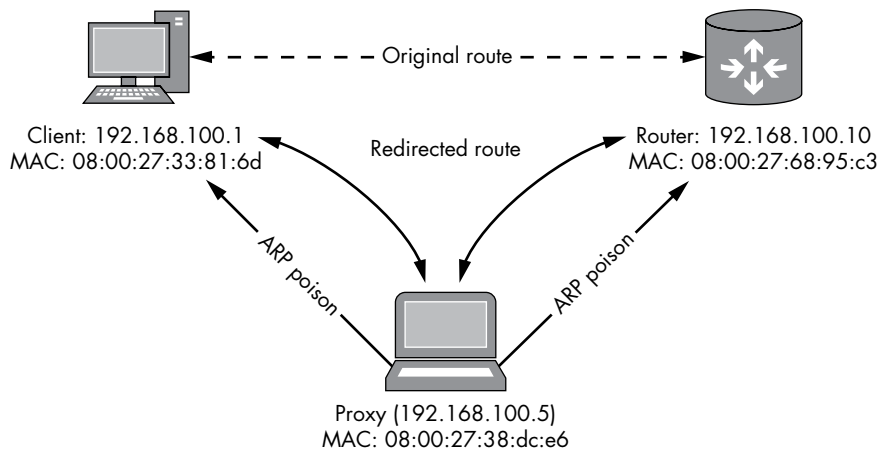
Figure 4-10: ARP poisoning

In Figure 4-10, Ettercap sends spoofed ARP packets to the client and the router on the local network. If spoofing succeeds, these ARP packets will change the cached ARP entries for both devices to point to your proxy.

Be sure to spoof ARP packets to both the client and the router to ensure that you get both sides of the communication. Of course, if all you want is one side of the communication, you only need to poison one or the other node.

To start ARP poisoning, follow these steps:

1. Start Ettercap, and enter **Unified Sniffing** mode as you did with DHCP spoofing.

2. Select the network interface to poison (the one connected to the network with the nodes you want to poison).

3. Configure a list of hosts to ARP poison. The easiest way to get a list of hosts is to let Ettercap scan for you by choosing **Hosts ▸ Scan For Hosts**. Depending on the size of the network, scanning can take from a few seconds to hours. When the scan is complete, choose **Hosts ▸ Host List**; a dialog like the one in Figure 4-11 should appear.
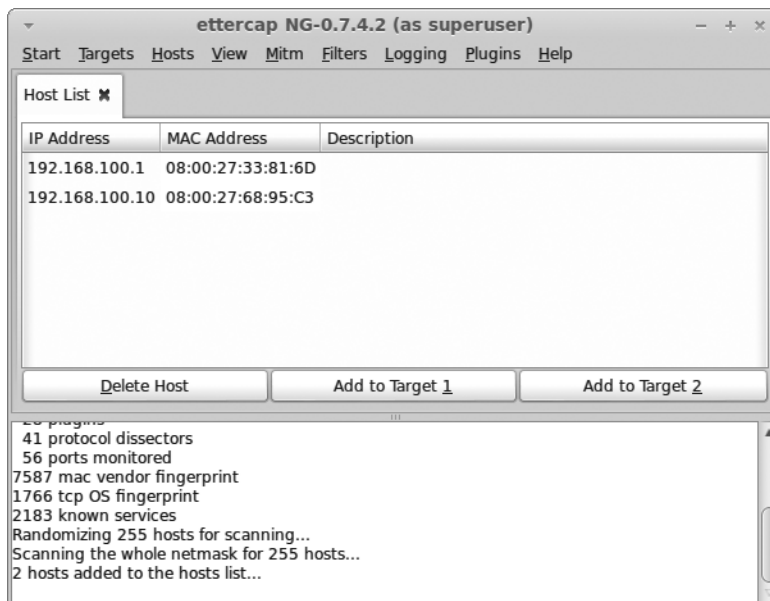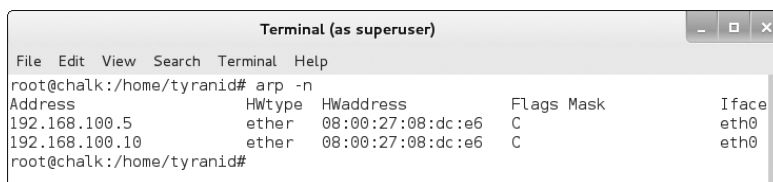
Figure 4-11: A list of discovered hosts

As you can see in Figure 4-11, we've found two hosts. In this case, one is the client node that you want to capture, which is on IP address 192.168.100.1 with a MAC address of 08:00:27:33:81:6d. The other node is the gateway to the internet on IP address 192.168.100.10 with a MAC address of 08:00:27:68:95:c3. Most likely, you'll already know the IP addresses configured for each network device, so you can determine which is the local machine and which is the remote machine.

4. Choose your targets. Select one of the hosts from the list and click **Add to Target 1**; select the other host you want to poison and click **Add to Target 2**. (Target 1 and Target 2 differentiate between the client and the gateway.) This should enable one-way ARP poisoning in which only data sent from Target 1 to Target 2 is rerouted.

5. Start ARP poisoning by choosing **Mitm ▸ ARP poisoning.** A dialog should appear. Accept the defaults and click **OK**. Ettercap should attempt to poison the ARP cache of your chosen targets. ARP poisoning may not work immediately because the ARP cache has to refresh. If poisoning is successful, the client node should look similar to Figure 4-12.

Figure 4-12: Successful ARP poisoning

Figure 4-12 shows the router was poisoned at IP 192.168.100.10, which has had its MAC Hardware address modified to the proxy's MAC address of 08:00:27:08:dc:e6. (For comparison, see the corresponding entry in Figure 4-11.) Now any traffic that is sent from the client to the router will instead be sent to the proxy (shown by the MAC address of 192.168.100.5). The proxy can forward the traffic to the correct destination after capturing or modifying it.

One advantage that ARP poisoning has over DHCP spoofing is that you can redirect nodes on the local network to communicate with your gateway even if the destination is on the local network. ARP poisoning doesn't have to poison the connection between the node and the external gateway if you don't want it to.

Final Words

In this chapter, you've learned a few additional ways to capture and modify traffic between a client and server. I began by describing how to configure your OS as an IP gateway, because if you can forward traffic through your own gateway, you have a number of techniques available to you.

Of course, just getting a device to send traffic to your network capture device isn't always easy, so employing techniques such as DHCP spoofing or ARP poisoning is important to ensure that traffic is sent to your device rather than directly to the internet. Fortunately, as you've seen, you don't need custom tools to do so; all the tools you need are either already included in your operating system (especially if you're running Linux) or easily downloadable.

5

ANALYSIS FROM THE WIRE

In Chapter 2, I discussed how to capture network traffic for analysis. Now it's time to put that knowledge to the test. In this chapter, we'll examine how to analyze captured network protocol traffic from a chat application to understand the protocol in use. If you can determine which features a protocol supports, you can assess its security.

Analysis of an unknown protocol is typically incremental. You begin by capturing network traffic, and then analyze it to try to understand what each part of the traffic represents. Throughout this chapter, I'll show you how to use Wireshark and some custom code to inspect an unknown network protocol. Our approach will include extracting structures and state information.

The Traffic-Producing Application: SuperFunkyChat

The test subject for this chapter is a chat application I've written in C# called SuperFunkyChat, which will run on Windows, Linux, and macOS. Download the latest prebuild applications and source code from the GitHub page at *https://github.com/tyranid/ExampleChatApplication/releases/*; be sure to choose the release binaries appropriate for your platform. (If you're using Mono, choose the .NET version, and so on.) The example client and server console applications for SuperFunkyChat are called ChatClient and ChatServer.

After you've downloaded the application, unpack the release files to a directory on your machine so you can run each application. For the sake of simplicity, all example command lines will use the Windows executable binaries. If you're running under Mono, prefix the command with the path to the main *mono* binary. When running files for .NET Core, prefix the command with the *dotnet* binary. The files for .NET will have a *.dll* extension instead of *.exe*.

Starting the Server

Start the server by running *ChatServer.exe* with no parameters. If successful, it should print some basic information, as shown in Listing 5-1.

```
C:\SuperFunkyChat> ChatServer.exe
ChatServer (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Running server on port 12345 Global Bind False
```

Listing 5-1: Example output from running ChatServer

NOTE *Pay attention to the warning! This application has not been designed to be a secure chat system.*

Notice in Listing 5-1 that the final line prints the port the server is running on (12345 in this case) and whether the server has bound to all interfaces (global). You probably won't need to change the port (--port NUM), but you might need to change whether the application is bound to all interfaces if you want clients and the server to exist on different computers. This is especially important on Windows. It's not easy to capture traffic to the local loopback interface on Windows; if you encounter any difficulties, you may need to run the server on a separate computer or a virtual machine (VM). To bind to all interfaces, specify the --global parameter.

Starting Clients

With the server running, we can start one or more clients. To start a client, run *ChatClient.exe* (see Listing 5-2), specify the username you want to use on the server (the username can be anything you like), and specify the server hostname (for example, localhost). When you run the client, you should see output similar to that shown in Listing 5-2. If you see any errors, make sure

you've set up the server correctly, including requiring binding to all inter-faces or disabling the firewall on the server.

```
C:\SuperFunkyChat> ChatClient.exe USERNAME HOSTNAME
ChatClient (c) 2017 James Forshaw
WARNING: Don't use this for a real chat system!!!
Connecting to localhost:12345
```

Listing 5-2: Example output from running ChatClient

As you start the client, look at the running server: you should see output on the console similar to Listing 5-3, indicating that the client has success-fully sent a "Hello" packet.

```
Connection from 127.0.0.1:49825
Received packet ChatProtocol.HelloProtocolPacket
Hello Packet for User: alice HostName: borax
```

Listing 5-3: The server output when a client connects

Communicating Between Clients

After you've completed the preceding steps successfully, you should be able to connect multiple clients so you can communicate between them. To send a message to all users with the ChatClient, enter the message on the com-mand line and press ENTER.

The ChatClient also supports a few other commands, which all begin with a forward slash (/), as detailed in Table 5-1.

Table 5-1: Commands for the ChatClient Application

Command	Description
/quit [message]	Quit client with optional message
/msg user message	Send a message to a specific user
/list	List other users on the system
/help	Print help information

You're ready to generate traffic between the SuperFunkyChat clients and server. Let's start our analysis by capturing and inspecting some traffic using Wireshark.

A Crash Course in Analysis with Wireshark

In Chapter 2, I introduced Wireshark but didn't go into any detail on how to use Wireshark to analyze rather than simply capture traffic. Because Wireshark is a very powerful and comprehensive tool, I'll only scratch the surface of its functionality here. When you first start Wireshark on Windows, you should see a window similar to the one shown in Figure 5-1.
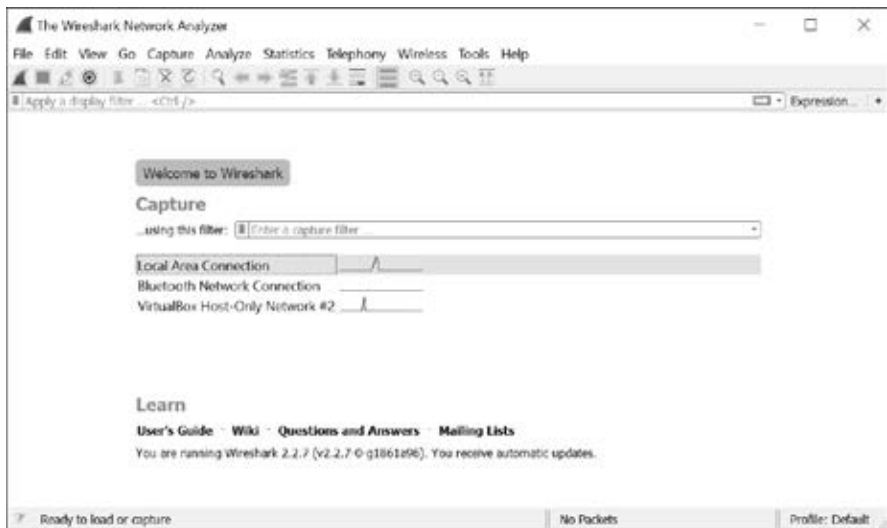
Figure 5-1: The main Wireshark window on Windows

The main window allows you to choose the interface to capture traffic from. To ensure we capture only the traffic we want to analyze, we need to configure some options on the interface. Select **Capture ▶ Options** from the menu. Figure 5-2 shows the options dialog that opens.
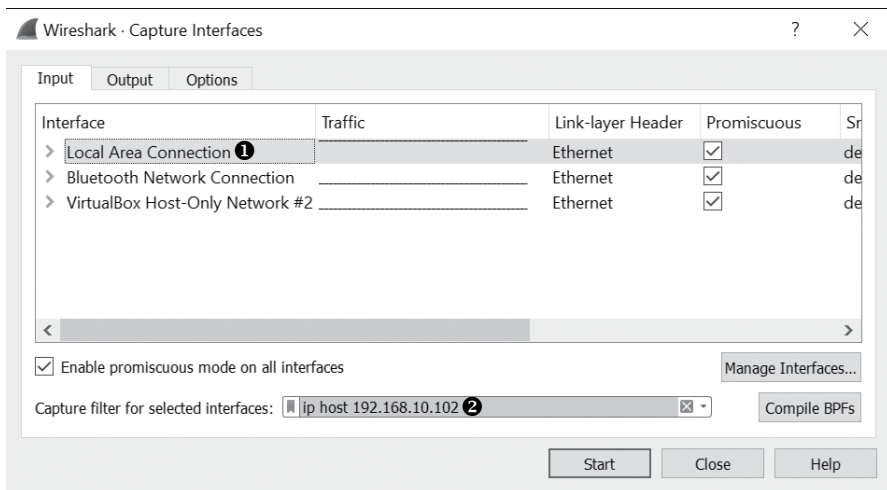
Figure 5-2: The Wireshark Capture Interfaces dialog

Select the network interface you want to capture traffic from, as shown at ❶. Because we're using Windows, choose **Local Area Connection**, which is our main Ethernet connection; we can't easily capture from Localhost. Then set a capture filter ❷. In this case, we specify the filter **ip host 192.168.10.102** to limit capture to traffic to or from the IP address 192.168.10.102. (The IP

address we're using is the chat server's address. Change the IP address as appropriate for your configuration.) Click the **Start** button to begin capturing traffic.

Generating Network Traffic and Capturing Packets

The main approach to packet analysis is to generate as much traffic from the target application as possible to improve your chances of finding its various protocol structures. For example, Listing 5-4 shows a single session with ChatClient for alice.

```
# alice - Session
> Hello There!
< bob: I've just joined from borax
< bob: How are you?
< bob: This is nice isn't it?
< bob: Woo
< Server: 'bob' has quit, they said 'I'm going away now!'
< bob: I've just joined from borax
< bob: Back again for another round.
< Server: 'bob' has quit, they said 'Nope!'
> /quit
< Server: Don't let the door hit you on the way out!
```

Listing 5-4: Single ChatClient session for alice*.*

And Listing 5-5 and Listing 5-6 show two sessions for bob.

```
# bob - Session 1
> How are you?
> This is nice isn't it?
> /list
< User List
< alice - borax
> /msg alice Woo
> /quit
< Server: Don't let the door hit you on the way out!
```

Listing 5-5: First ChatClient session for bob

```
# bob - Session 2
> Back again for another round.
> /quit Nope!
< Server: Don't let the door hit you on the way out!
```

Listing 5-6: Second ChatClient session for bob

We run two sessions for bob so we can capture any connection or disconnection events that might only occur between sessions. In each session, a right angle bracket (>) indicates a command to enter into the ChatClient, and a left angle bracket (<) indicates responses from the server being written to the

console. You can execute the commands to the client for each of these session captures to reproduce the rest of the results in this chapter for analysis.

Now turn to Wireshark. If you've configured Wireshark correctly and bound it to the correct interface, you should start seeing packets being captured, as shown in Figure 5-3.
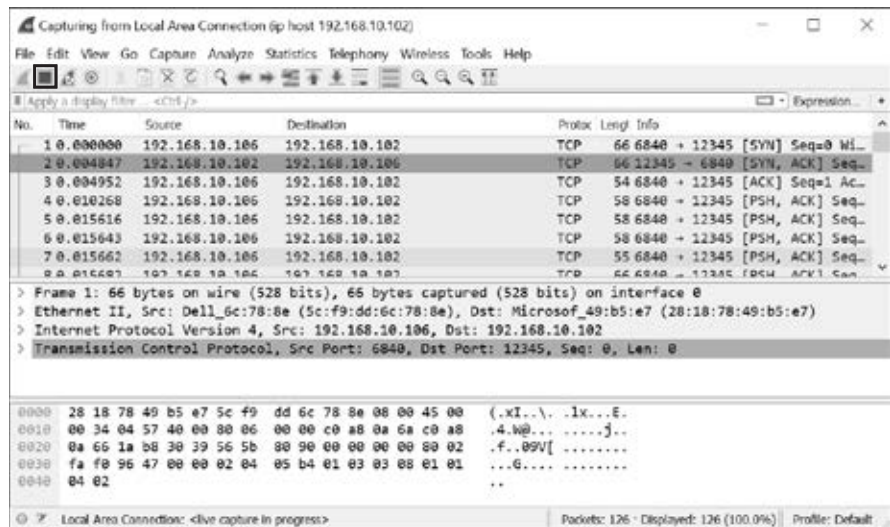
Figure 5-3: Captured traffic in Wireshark

After running the example sessions, stop the capture by clicking the **Stop** button (highlighted) and save the packets for later use if you want.

Basic Analysis

Let's look at the traffic we've captured. To get an overview of the communication that occurred during the capture period, choose among the options on the Statistics menu. For example, choose **Statistics ▸ Conversations**, and you should see a new window displaying high-level conversations such as TCP sessions, as shown in the Conversations window in Figure 5-4.
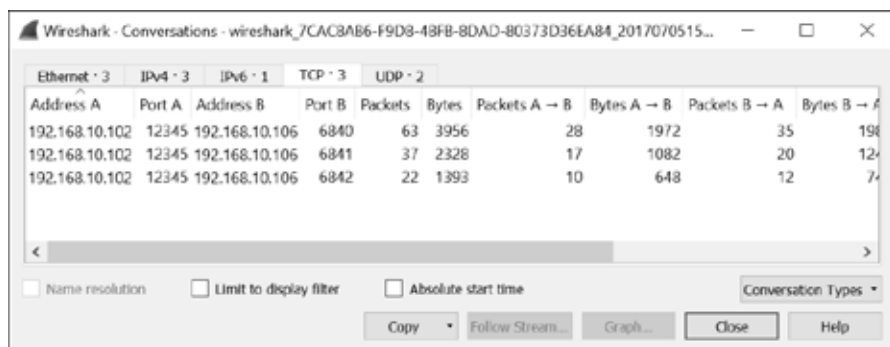
Figure 5-4: The Wireshark Conversations window

The Conversations window shows three separate TCP conversations in the captured traffic. We know that the SuperFunkyChat client application uses port 12345, because we see three separate TCP sessions coming from port 12345. These sessions should correspond to the three client sessions shown in Listing 5-4, Listing 5-5, and Listing 5-6.

Reading the Contents of a TCP Session

To view the captured traffic for a single conversation, select one of the conversations in the Conversations window and click the **Follow Stream** button. A new window displaying the contents of the stream as ASCII text should appear, as shown in Figure 5-5.
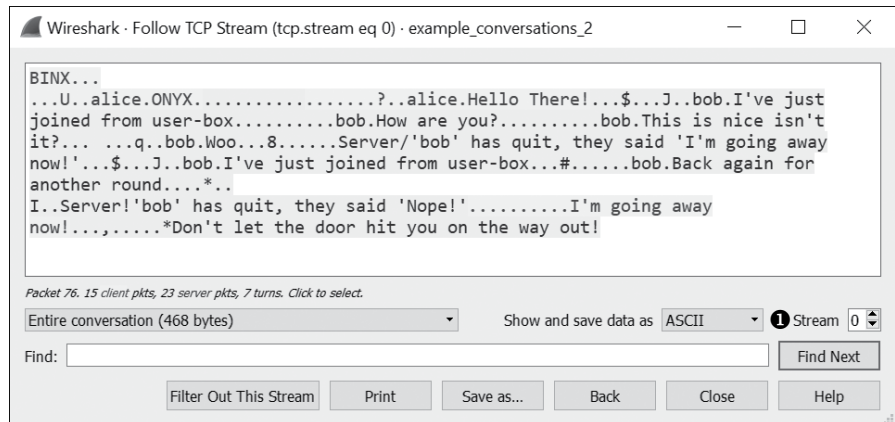
Figure 5-5: Displaying the contents of a TCP session in Wireshark's Follow TCP Stream view

Wireshark replaces data that can't be represented as ASCII characters with a single dot character, but even with that character replacement, it's clear that much of the data is being sent in plaintext. That said, the network protocol is clearly not exclusively a text-based protocol because the control information for the data is nonprintable characters. The only reason we're seeing text is that SuperFunkyChat's primary purpose is to send text messages.

Wireshark shows the inbound and outbound traffic in a session using different colors: pink for outbound traffic and blue for inbound. In a TCP session, outbound traffic is from the client that initiated the TCP session, and inbound traffic is from the TCP server. Because we've captured all traffic to the server, let's look at another conversation. To change the conversation, change the Stream number ❶ in Figure 5-5 to 1. You should now see a different conversation, for example, like the one in Figure 5-6.
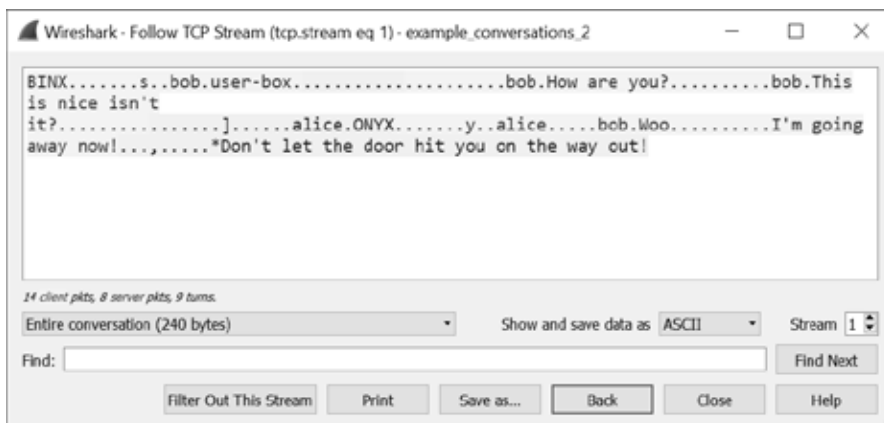
Figure 5-6: A second TCP session from a different client

Compare Figure 5-6 to Figure 5-5; you'll see the details of the two sessions are different. Some text sent by the client (in Figure 5-6), such as "How are you?", is shown as received by the server in Figure 5-5. Next, we'll try to determine what those binary parts of the protocol represent.

Identifying Packet Structure with Hex Dump

At this point, we know that our subject protocol seems to be part binary and part text, which indicates that looking at just the printable text won't be enough to determine all the various structures in the protocol.

To dig in, we first return to Wireshark's Follow TCP Stream view, as shown in Figure 5-5, and change the Show and save data as drop-down menu to the **Hex Dump** option. The stream should now look similar to Figure 5-7.
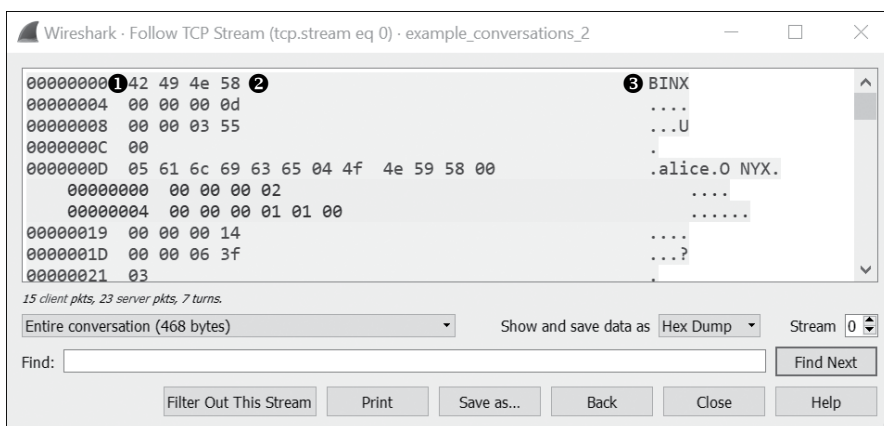
Figure 5-7: The Hex Dump view of the stream

The Hex Dump view shows three columns of information. The column at the very left ❶ is the byte offset into the stream for a particular direction. For example, the byte at 0 is the first byte sent in that direction, the byte 4 is the fifth, and so on. The column in the center ❷ shows the bytes as a hex dump. The column at the right ❸ is the ASCII representation, which we saw previously in Figure 5-5.

Viewing Individual Packets

Notice how the blocks of bytes shown in the center column in Figure 5-7 vary in length. Compare this again to Figure 5-6; you'll see that other than being separated by direction, all data in Figure 5-6 appears as one contiguous block. In contrast, the data in Figure 5-7 might appear as just a few blocks of 4 bytes, then a block of 1 byte, and finally a much longer block containing the main group of text data.

What we're seeing in Wireshark are individual packets: each block is a single TCP packet, or *segment*, containing perhaps only 4 bytes of data. TCP is a stream-based protocol, which means that there are no real boundaries between consecutive blocks of data when you're reading and writing data to a TCP socket. However, from a physical perspective, there's no such thing as a real stream-based network transport protocol. Instead, TCP sends individual packets consisting of a TCP header containing information, such as the source and destination port numbers as well as the data.

In fact, if we return to the main Wireshark window, we can find a packet to prove that Wireshark is displaying single TCP packets. Select **Edit ▸ Find Packet**, and an additional drop-down menu appears in the main window, as shown Figure 5-8.

Figure 5-8: Finding a packet in Wireshark's main window

We'll find the first value shown in Figure 5-7, the string `BINX`. To do this, fill in the Find options as shown in Figure 5-8. The first s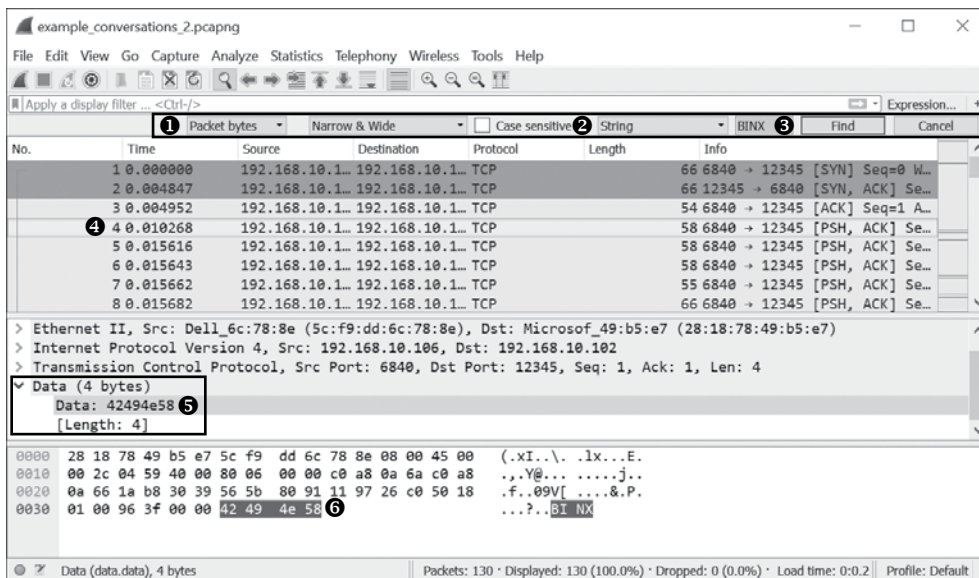election box indicates where in the packet capture to search. Specify that you want to search in the Packet bytes ❶. Leave the second selection box as Narrow & Wide, which indicates that you want to search for both ASCII and Unicode strings. Also leave the Case sensitive box unchecked and specify that you want to look for a String value ❷ in the third drop-down menu. Then enter the string value we want to find, in this case the string `BINX` ❸. Finally, click the **Find** button, and the main window should automatically scroll and highlight the first packet Wireshark finds that contains the `BINX` string ❹. In the middle window at ❺, you should see that the packet contains 4 bytes, and you can see the raw data in the bottom window, which shows that we've found the `BINX` string ❻. We now know that the Hex Dump view Wireshark displays in Figure 5-8 represents packet boundaries because the `BINX` string is in a packet of its own.

Determining the Protocol Structure

To simplify determining the protocol structure, it makes sense to look only at one direction of the network communication. For example, let's just look at the outbound direction (from client to server) in Wireshark. Returning to the Follow TCP Stream view, select the **Hex Dump** option in the Show and save data as drop-down menu. Then select the traffic direction from the client to the server on port 12345 from the drop-down menu at ❶, as shown in Figure 5-9.
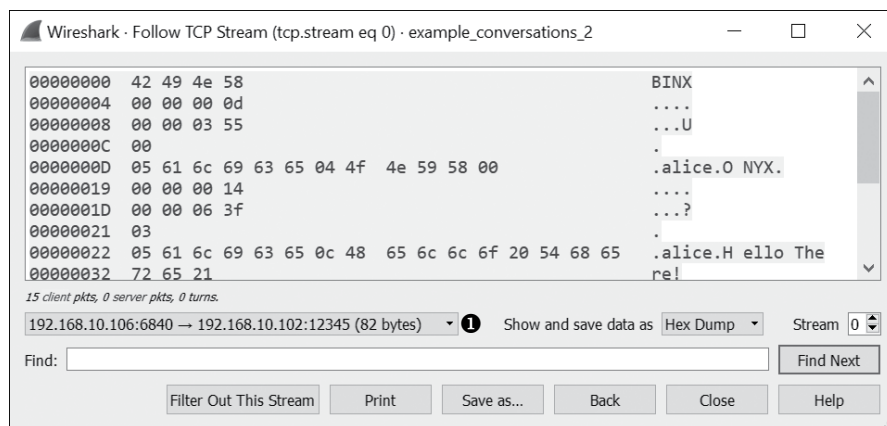
Figure 5-9: A hex dump showing only the outbound direction

Click the **Save as . . .** button to copy the outbound traffic hex dump to a text file to make it easier to inspect. Listing 5-7 shows a small sample of that traffic saved as text.

```
00000000   42 49 4e 58                                         BINX❶
00000004   00 00 00 0d                                         ....❷
00000008   00 00 03 55                                         ...U❸
```

```
0000000C  00                                                       .❹
0000000D  05 61 6c 69 63 65 04 4f  4e 59 58 00                     .alice.O NYX.❺
00000019  00 00 00 14                                              ....
0000001D  00 00 06 3f                                              ...?
00000021  03                                                       .
00000022  05 61 6c 69 63 65 0c 48  65 6c 6c 6f 20 54 68 65         .alice.H ello The
00000032  72 65 21                                                 re!
--snip--
```

Listing 5-7: A snippet of outbound traffic

The outbound stream begins with the four characters BINX ❶. These characters are never repeated in the rest of the data stream, and if you compare different sessions, you'll always find the same four characters at the start of the stream. If I were unfamiliar with this protocol, my intuition at this point would be that this is a magic value sent from the client to the server to tell the server that it's talking to a valid client rather than some other application that happens to have connected to the server's TCP port.

Following the stream, we see that a sequence of four blocks is sent. The blocks at ❷ and ❸ are 4 bytes, the block at ❹ is 1 byte, and the block at ❺ is larger and contains mostly readable text. Let's consider the first block of 4 bytes at ❷. Might these represent a small number, say the integer value 0xD or 13 in decimal?

Recall the discussion of the Tag, Length, Value (TLV) pattern in Chapter 3. TLV is a very simple pattern in which each block of data is delimited by a value representing the length of the data that follows. This pattern is especially important for stream-based protocols, such as those running over TCP, because otherwise the application doesn't know how much data it needs to read from a connection to process the protocol. If we assume that this first value is the length of the data, does this length match the length of the rest of the packet? Let's find out.

Count the total bytes of the blocks at ❷, ❸, ❹, and ❺, which seem to be a single packet, and the result is 21 bytes, which is eight more than the value of 13 we were expecting (the integer value 0xD). The value of the length block might not be counting its own length. If we remove the length block (which is 4 bytes), the result is 17, which is 4 bytes more than the target length but getting closer. We also have the other unknown 4-byte block at ❸ following the potential length, but perhaps that's not counted either. Of course, it's easy to speculate, but facts are more important, so let's do some testing.

Testing Our Assumptions

At this point in such an analysis, I stop staring at a hex dump because it's not the most efficient approach. One way to quickly test whether our assumptions are right is to export the data for the stream and write some simple code to parse the structure. Later in this chapter, we'll write some code for Wireshark to do all of our testing within the GUI, but for now we'll implement the code using Python on the command line.

To get our data into Python, we could add support for reading Wireshark capture files, but for now we'll just export the packet bytes to a file. To export the packets from the dialog shown in Figure 5-9, follow these steps:

1. In the Show and save data as drop-down menu, choose the **Raw** option.
2. Click **Save As** to export the outbound packets to a binary file called *bytes_outbound.bin*.

We also want to export the inbound packets, so change to and select the inbound conversation. Then save the raw inbound bytes using the preceding steps, but name the file *bytes_inbound.bin*.

Now use the XXD tool (or a similar tool) on the command line to be sure that we've successfully dumped the data, as shown in Listing 5-8.

```
$ xxd bytes_outbound.bin
00000000: 4249 4e58 0000 000f 0000 0473 0003 626f  BINX.......s..bo
00000010: 6208 7573 6572 2d62 6f78 0000 0000 1200  b.user-box......
00000020: 0005 8703 0362 6f62 0c48 6f77 2061 7265  .....bob.How are
00000030: 2079 6f75 3f00 0000 1c00 0008 e303 0362   you?..........b
00000040: 6f62 1654 6869 7320 6973 206e 6963 6520  ob.This is nice
00000050: 6973 6e27 7420 6974 3f00 0000 0100 0000  isn't it?.......
00000060: 0606 0000 0013 0000 0479 0505 616c 6963  .........y..alic
00000070: 6500 0000 0303 626f 6203 576f 6f00 0000  e.....bob.Woo...
00000080: 1500 0006 8d02 1349 276d 2067 6f69 6e67  .......I'm going
00000090: 2061 7761 7920 6e6f 7721                  away now!
```

Listing 5-8: The exported packet bytes

Dissecting the Protocol with Python

Now we'll write a simple Python script to dissect the protocol. Because we're just extracting data from a file, we don't need to write any network code; we just need to open the file and read the data. We'll also need to read binary data from the file—specifically, a network byte order integer for the length and unknown 4-byte block.

Performing the Binary Conversion

We can use the built-in Python struct library to do the binary conversions. The script should fail immediately if something doesn't seem right, such as not being able to read all the data we expect from the file. For example, if the length is 100 bytes and we can read only 20 bytes, the read should fail. If no errors occur while parsing the file, we can be more confident that our analysis is correct. Listing 5-9 shows the first implementation, written to work in both Python 2 and 3.

```
from struct import unpack
import sys
import os
```

```
    # Read fixed number of bytes
❶ def read_bytes(f, l):
        bytes = f.read(l)
    ❷ if len(bytes) != l:
            raise Exception("Not enough bytes in stream")
        return bytes

    # Unpack a 4-byte network byte order integer
❸ def read_int(f):
        return unpack("!i", read_bytes(f, 4))[0]

    # Read a single byte
❹ def read_byte(f):
        return ord(read_bytes(f, 1))

    filename = sys.argv[1]
    file_size = os.path.getsize(filename)

    f = open(filename, "rb")
❺ print("Magic: %s" % read_bytes(f, 4))

    # Keep reading until we run out of file
❻ while f.tell() < file_size:
        length = read_int(f)
        unk1 = read_int(f)
        unk2 = read_byte(f)
        data = read_bytes(f, length - 1)
        print("Len: %d, Unk1: %d, Unk2: %d, Data: %s"
            % (length, unk1, unk2, data))
```

Listing 5-9: An example Python script for parsing protocol data

Let's break down the important parts of the script. First, we define some helper functions to read data from the file. The function read_bytes() ❶ reads a fixed number of bytes from the file specified as a parameter. If not enough bytes are in the file to satisfy the read, an exception is thrown to indicate an error ❷. We also define a function read_int() ❸ to read a 4-byte integer from the file in network byte order where the most significant byte of the integer is first in the file, as well as define a function to read a single byte ❹. In the main body of the script, we open a file passed on the command line and first read a 4-byte value ❺, which we expect is the magic value BINX. Then the code enters a loop ❻ while there's still data to read, reading out the length, the two unknown values, and finally the data and then printing the values to the console.

When you run the script in Listing 5-9 and pass it the name of a binary file to open, all data from the file should be parsed and no errors generated if our analysis that the first 4-byte block was the length of the data sent on the network is correct. Listing 5-10 shows example output in Python 3, which does a better job of displaying binary strings than Python 2.

```
$ python3 read_protocol.py bytes_outbound.bin
Magic: b'BINX'
Len: 15, Unk1: 1139, Unk2: 0, Data: b'\x03bob\x08user-box\x00'
Len: 18, Unk1: 1415, Unk2: 3, Data: b'\x03bob\x0cHow are you?'
Len: 28, Unk1: 2275, Unk2: 3, Data: b"\x03bob\x16This is nice isn't it?"
Len: 1, Unk1: 6, Unk2: 6, Data: b''
Len: 19, Unk1: 1145, Unk2: 5, Data: b'\x05alice\x00\x00\x00\x03\x03bob\x03Woo'
Len: 21, Unk1: 1677, Unk2: 2, Data: b"\x13I'm going away now!"
```

Listing 5-10: Example output from running Listing 5-9 against a binary file

Handling Inbound Data

If you ran Listing 5-9 against an exported inbound data set, you would immediately get an error because there's no magic string BINX in the inbound protocol, as shown in Listing 5-11. Of course, this is what we would expect if there were a mistake in our analysis and the length field wasn't quite as simple as we thought.

```
$ python3 read_protocol.py bytes_inbound.bin
Magic: b'\x00\x00\x00\x02'
Length: 1, Unknown1: 16777216, Unknown2: 0, Data: b''
Traceback (most recent call last):
  File "read_protocol.py", line 31, in <module>
    data = read_bytes(f, length - 1)
  File "read_protocol.py", line 9, in read_bytes
    raise Exception("Not enough bytes in stream")
Exception: Not enough bytes in stream
```

Listing 5-11: Error generated by Listing 5-9 on inbound data

We can clear up this error by modifying the script slightly to include a check for the magic value and reset the file pointer if it's not equal to the string BINX. Add the following line just after the file is opened in the original script to reset the file pointer to the start if the magic value is incorrect.

```
if read_bytes(f, 4) != b'BINX': f.seek(0)
```

Now, with this small modification, the script will execute successfully on the inbound data and result in the output shown in Listing 5-12.

```
$ python3 read_protocol.py bytes_inbound.bin
Len: 2, Unk1: 1, Unk2: 1, Data: b'\x00'
Len: 36, Unk1: 3146, Unk2: 3, Data: b"\x03bob\x1eI've just joined from user-box"
Len: 18, Unk1: 1415, Unk2: 3, Data: b'\x03bob\x0cHow are you?'
```

Listing 5-12: Output of modified script on inbound data

Digging into the Unknown Parts of the Protocol

We can use the output in Listing 5-10 and Listing 5-12 to start delving into the unknown parts of the protocol. First, consider the field labeled Unk1. The values it takes seem to be different for every packet, but the values are low, ranging from 1 to 3146.

But the most informative parts of the output are the following two entries, one from the outbound data and one from the inbound.

```
OUTBOUND: Len: 1, Unk1: 6, Unk2: 6, Data: b''
INBOUND:  Len: 2, Unk1: 1, Unk2: 1, Data: b'\x00'
```

Notice that in both entries the value of Unk1 is the same as Unk2. That could be a coincidence, but the fact that both entries have the same value might indicate something important. Also notice that in the second entry the length is 2, which includes the Unk2 value and a 0 data value, whereas the length of the first entry is only 1 with no trailing data after the Unk2 value. Perhaps Unk1 is directly related to the data in the packet? Let's find out.

Calculating the Checksum

It's common to add a checksum to a network protocol. The canonical example of a checksum is just the sum of all the bytes in the data you want to check for errors. If we assume that the unknown value is a *simple* checksum, we can sum all the bytes in the example outbound and inbound packets I highlighted in the preceding section, resulting in the calculated sum shown in Table 5-2.

Table 5-2: Testing Checksum for Example Packets

Unknown value	Data bytes	Sum of data bytes
6	6	6
1	1, 0	1

Although Table 5-2 seems to confirm that the unknown value matches our expectation of a simple checksum for very simple packets, we still need to verify that the checksum works for larger and more complex packets. There are two easy ways to determine whether we've guessed correctly that the unknown value is a checksum over the data. One way is to send simple, incrementing messages from a client (like *A*, then *B*, then *C*, and so on), capture the data, and analyze it. If the checksum is a simple addition, the value should increment by 1 for each incrementing message. The alternative would be to add a function to calculate the checksum to see whether the checksum matches between what was captured on the network and our calculated value.

To test our assumptions, add the code in Listing 5-13 to the script in Listing 5-7 and add a call to it after reading the data to calculate the checksum. Then just compare the value extracted from the network capture as `Unk1` and the calculated value to see whether our calculated checksum matches.

```
def calc_chksum(unk2, data):
    chksum = unk2
    for i in range(len(data)):
        chksum += ord(data[i:i+1])
    return chksum
```

Listing 5-13: Calculating the checksum of a packet

And it does! The numbers calculated match the value of `Unk1`. So, we've discovered the next part of the protocol structure.

Discovering a Tag Value

Now we need to determine what `Unk2` might represent. Because the value of `Unk2` is considered part of the packet's data, it's presumably related to the meaning of what is being sent. However, as we saw at ❹ in Listing 5-7, the value of `Unk2` is being written to the network as a single byte value, which indicates that it's actually separate from the data. Perhaps the value represents the Tag part of a TLV pattern, just as we suspect that Length is the Value part of that construction.

To determine whether `Unk2` is in fact the Tag value and a representation of how to interpret the rest of the data, we'll exercise the ChatClient as much as possible, try all possible commands, and capture the results. We can then perform basic analysis comparing the value of `Unk2` when sending the same type of command to see whether the value of `Unk2` is always the same.

For example, consider the client sessions in Listing 5-4, Listing 5-5, and Listing 5-6. In the session in Listing 5-5, we sent two messages, one after another. We've already analyzed this session using our Python script in Listing 5-10. For simplicity, Listing 5-14 shows only the first three capture packets (with the latest version of the script).

```
Unk2: 0❶, Data: b'\x03bob\x08user-box\x00'
Unk2: 3❷, Data: b'\x03bob\x0cHow are you?'
Unk2: 3❸, Data: b"\x03bob\x16This is nice isn't it?"
*SNIP*
```

Listing 5-14: The first three packets from the session represented by Listing 5-5

The first packet ❶ doesn't correspond to anything we typed into the client session in Listing 5-5. The unknown value is 0. The two messages we then sent in Listing 5-5 are clearly visible as text in the `Data` part of the packets at ❷ and ❸. The `Unk2` values for both of those messages is 3, which is different from the first packet's value of 0. Based on this observation, we can assume that the value of 3 might represent a packet that is sending a message, and if that's the case, we'd expect to find a value of 3 used in every