Penetration Testing with Perl

This guide will teach you the fundamentals of penetration testing with Perl, providing an understanding of the mindset of a hacker. In the first few chapters, you will study how to utilize Perl with Linux and the regular expression syntax. After that, you will learn how to use Perl for WAN target analysis, and Internet and external footprinting. You will learn to use Perl for automated web application and site penetration testing. We also cover intelligence gathering techniques from data obtained from footprinting and simple file forensics with file metadata.

By the end of this book, you will bring all of your code together into a simple graphical user interface penetration testing framework. Through this guide, you will have acquired the knowledge to apply Perl programming to any penetration testing phase and learn the importance of applying our technique in the methodology and context of the Penetration Testing Execution Standard.

Who this book is written for

If you are an expert Perl programmer interested in penetration testing or information security, this guide is designed for you. However, it will also be helpful for you even if you have little or no Linux shell experience.

What you will learn from this book

- Develop your knowledge on the methodology, legality, and ethics involved in penetration testing
- Utilize Linux OS, shell, and networking utilities using Perl
- Manipulate and capture LAN network traffic in modern switched environments
- Apply Perl to the Aircrack-ng suite
- Combine external, internal, and Internet footprinting, and open source intelligence gathering
- Perform web penetration testing and automation, including Local and Remote File Inclusion, SQL injection, cross-site scripting, and CMS vulnerability analysis
- Stabilize your tests by using patterns such as Action Wrapper and Black Hole Proxy

Harness the power of Perl to perform professional penetration testing
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 “Perl Programming”
- A synopsis of the book’s content
- More information on Penetration Testing with Perl

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.
I would like to thank my family, Victoria and David Weis, Lynn McLain, and Douglas James Berdeaux. Thank you to my beautiful fiancé, Julie Aluise, <3 and our amazing family, Penelope, Gabriella, Chloe, Bobby, Popchwea, Russel, Petey, Pirate, Jim, and Margaret Aluise for being unconditionally patient and supportive while I was busy writing code. Thank you Thomas Berdeaux for giving me my very own first computer. Thank you Jaime McLain for the inspiration and advice. Thank you Tekk for always being there for our +o. Thank you Brad Carter, Amy, Graem Murd0c, Jered Morgan, Greyarea, Grant Stone, Ben Nichols, Leo Zeygerman, and Jay Turla!
Penetration Testing with Perl

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 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.
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.
Perl Programming

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.

<table>
<thead>
<tr>
<th>Character/quantifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\s</td>
<td>This represents any whitespace character, tab, and space</td>
</tr>
<tr>
<td>\S</td>
<td>This represents any non-whitespace character</td>
</tr>
<tr>
<td>\w</td>
<td>This represents any word character (alphanumeric and underscore)</td>
</tr>
<tr>
<td>\W</td>
<td>This represents any non-word character</td>
</tr>
<tr>
<td>\d</td>
<td>This represents any digit</td>
</tr>
<tr>
<td>\D</td>
<td>This represents any non-digit character</td>
</tr>
<tr>
<td>\n</td>
<td>This represents a new line</td>
</tr>
<tr>
<td>\t</td>
<td>This represents a tab character</td>
</tr>
<tr>
<td>^</td>
<td>This represents an anchor to the beginning of the line</td>
</tr>
<tr>
<td>$</td>
<td>This represents an anchor to the end of the line</td>
</tr>
<tr>
<td>()</td>
<td>This represents a grouping of strings</td>
</tr>
<tr>
<td>(x</td>
<td>y)</td>
</tr>
<tr>
<td>[a-zA-Z]</td>
<td>This represents character classes match a through z OR A-Z</td>
</tr>
<tr>
<td>.</td>
<td>This represents any character at all, even a space</td>
</tr>
<tr>
<td>*</td>
<td>This represents any amount, even 0</td>
</tr>
<tr>
<td>?</td>
<td>This represents 1 or 0</td>
</tr>
<tr>
<td>+</td>
<td>This represents at least 1</td>
</tr>
<tr>
<td>{x}</td>
<td>This indicates that it is at least x times a match</td>
</tr>
<tr>
<td>{x,}</td>
<td>This indicates that it is at least x amount of times a match</td>
</tr>
<tr>
<td>{x,y}</td>
<td>This indicates that it is at least x but no more than y times a match</td>
</tr>
</tbody>
</table>

**Quantifiers**

What if we want to have at least one character match, such as the o character in fo, foo, and fo0000? 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.

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:

```html
<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.

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 L3ea7 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 Le7, 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.

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:

```perl
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:

\[(\text{barbaz})\{2\}\]

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

\^[\text{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:

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

The Perl \texttt{m//}} matching operator

Let's design a simple script that uses the \texttt{curl} Linux program to get a web page and filter its output. \texttt{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, \texttt{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 \texttt{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:

```perl
#!/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:

```perl
#!/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:

```perl
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'
```

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:

```perl
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:

```perl
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:

```perl
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:

```perl
#!/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.
Perl Programming

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:

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

The other snippet is:

```perl
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`:

```perl
#!/usr/bin/perl -w
use strict;
my @passwd = ("123password","mypass00","secr3tPASSWORD","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:

```perl
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:

```perl
#!/usr/bin/perl -w
use strict;
my @passwd = ("123password","mypass00","secr3tPASSWORD","ultraPassword","password_2015","fb_password_011","secret6667","H1KING3343","1337secrets_MS");
print $_ ."\n" foreach(grep{$_ =~ m/\d/} @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:

```perl
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)
```
Perl Programming

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:

```
> 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:

```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");
my $req = HTTP::Request->new(GET => shift);
my $res = $ua->request($req);
my @lines = split(/
/,$res->content);
foreach(@lines){
    print "_\n" if($_ =~ m/<img.+src=('|').*>/);
}
```
Perl Programming

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:

```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");
my $req = HTTP::Request->new(GET => shift);
my $res = $ua->request($req);
my @lines = split(/
/,$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:

```bash
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:

```bash
cpanm Module::Name
```

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

```bash
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:

```bash
root@80211:~ # cpanm -U Regexp::Debugger
Regexp::Debugger contains the following files:
```

[29]
Perl Programming

/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

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.
Where to buy this book
You can buy Penetration Testing with Perl from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.