8

BASH SCRIPTING

Any self-respecting hacker must be able to write scripts. For that matter, any self-respecting Linux administrator must be able to script. Hackers often need to automate commands, sometimes from multiple tools, and this is most efficiently done through short programs they write themselves.

In this chapter, we build a few simple bash shell scripts to start you off with scripting. We'll add capabilities and features as we progress, eventually building a script capable of finding potential attack targets over a range of IP addresses.

To become an *elite* hacker, you also need the ability to script in one of the widely used scripting languages, such as Ruby (Metasploit exploits are written in Ruby), Python (many hacking tools are Python scripts), or Perl (Perl is the best text-manipulation scripting language). I give a brief introduction to Python scripting in Chapter 17.

A Crash Course in Bash

A *shell* is an interface between the user and the operating system that enables you to manipulate files and run commands, utilities, programs, and much more. The advantage of a shell is that you perform these tasks immediately from the computer and not through an abstraction, like a GUI, which allows you to customize your task to your needs. A number of different shells are available for Linux, including the Korn shell, the Z shell, the C shell, and the *B*ourne-*a*gain *sh*ell, more widely known as bash.

Because the bash shell is available on nearly all Linux and UNIX distributions (including macOS and Kali), we'll be using the bash shell, exclusively.

The bash shell can run any system commands, utilities, or applications your usual command line can run, but it also includes some of its own built-in commands. Table 8-1 later in the chapter gives you a reference to some useful commands that reside within the bash shell.

In earlier chapters, you used the `cd`, `pwd`, `set`, and `umask` commands. In this section, you will be using two more commands: the `echo` command, first used in Chapter 7, which displays messages to the screen, and the `read` command, which reads in data and stores it somewhere else. Just learning these two commands alone will enable you to build a simple but powerful tool.

You'll need a text editor to create shell scripts. You can use whichever Linux text editor you like best, including vi, vim, emacs, gedit, kate, and so on. I'll be using Leafpad in these tutorials, as I have in previous chapters. Using a different editor should *not* make any difference in your script or its functionality.

Your First Script: "Hello, Hackers-Arise!"

For your first script, we will start with a simple program that returns a message to the screen that says `"Hello, Hackers-Arise!"` Open your text editor, and let's go.

To start, you need to tell your operating system which interpreter you want to use for the script. To do this, enter a *shebang*, which is a combination of a hash mark and an exclamation mark, like so:

```
#!
```

You then follow the shebang (`#!`) with `/bin/bash` to indicate that you want the operating system to use the bash shell interpreter. As you'll see in later chapters, you could also use the shebang to use other interpreters, such as Perl or Python. Here, you want to use the bash interpreter, so enter the following:

```
#! /bin/bash
```

Next, enter the `echo` command, which tells the system to simply repeat (or *echo*) back to your monitor whatever follows the command.

In this case, we want the system to echo back to us `"Hello, Hackers-Arise!"`, as done in Listing 8-1. Note that the text or message we want to echo back must be in double quotation marks.

```
#! /bin/bash

# This is my first bash script. Wish me luck.

echo "Hello, Hackers-Arise!"
```

Listing 8-1: Your "Hello, Hackers-Arise!" script

Here, you also see a line that's preceded by a hash mark (`#`). This is a *comment*, which is a note you leave to yourself or anyone else reading the code to explain what you're doing in the script. Programmers use comments in every coding language. These comments are not read or executed by the interpreter, so you don't need to worry about messing up your code. They are visible only to humans. The bash shell knows a line is a comment if it starts with the # character.

Now, save this file as *HelloHackersArise* with no extension and exit your text editor.

Setting Execute Permissions

By default, a newly created bash script is not executable even by you, the owner. Let's look at the permissions on our new file in the command line by using `cd` to move into the directory and then entering `ls -l`. It should look something like this:

```
kali >ls -l
--snip--
-rw-r--r-- 1 root root 42 Oct 22 14:32 HelloHackersArise
--snip--
```

As you can see, our new file has `rw-r--r--` (644) permissions. As you learned in Chapter 5, this means the owner of this file only has read (`r`) and write (`w`) permissions, but no execute (`x`) permissions. The group and all other users have only read permissions. We need to give ourselves execute permissions in order to run this script. We change the permissions with the `chmod` command, as you saw in Chapter 5. To give the owner, the group, and all others execute permissions, enter the following:

```
kali >chmod 755 HelloHackersArise
```

Now when we do a long listing on the file, like so, we can see that we have execute permissions:

```
kali >ls -l
--snip--
-rwx r-x r-x 1 root root 42 Oct 22 14:32 HelloHackersArise
--snip--
```

The script is now ready to execute!

Running HelloHackersArise

To run our simple script, enter the following:

```
kali >./HelloHackersArise
```

The `./` before the filename tells the system that we want to execute this script in the file *HelloHackersArise* from the current directory. It also tells the system that if there is another file in another directory named *HelloHackersArise*, please ignore it and only run *HelloHackersArise* in the current directory. It may seem unlikely that there's another file with this name on your system, but it's good practice to use the `./` when executing files, as this localizes the file execution to the current directory and many directories will have duplicate filenames, such as *start* and *setup*.

When we press ENTER, our very simple script returns our message to the monitor:

```
Hello, Hackers-Arise!
```

Success! You just completed your first shell script!

Adding Functionality with Variables and User Input

So, now we have a simple script. All it does is echo back a message to standard output. If we want to create more advanced scripts, we will likely need to add some variables.

A *variable* is an area of storage that can hold something in memory. That "something" might be some letters or words (strings) or numbers. It's known as a variable because the values held within it are changeable; this is an extremely useful feature for adding functionality to a script.

In our next script, we will add functionality to prompt the user for their name, place whatever they input into a variable, then prompt the user for the chapter they're at in this book, and place that keyboard input into a variable. After that, we'll echo a welcome message that includes their name and the chapter back to the user.

Open a new file in your text editor and enter the script shown in Listing 8-2.

```
❶ #! /bin/bash

❷ # This is your second bash script. In this one, you prompt /
  # the user for input, place the input in a variable, and /
  # display the variable contents in a string.

❸ echo "What is your name?"

  read name

❹ echo "What chapter are you on in Linux Basics for Hackers?"

  read chapter

❺ echo "Welcome" $name "to Chapter" $chapter "of Linux Basics for Hackers!"
```

Listing 8-2: A simple script making use of variables

We open with `#! /bin/bash` to tell the system we want to use the bash interpreter for this script ❶. We then add a comment that describes the script and its functionality ❷. After that, we prompt the user for their name and ask the interpreter to read the input and place it into a variable we call `name` ❸. Then we prompt the user to enter the chapter they are currently working through in this book, and we again read the keyboard input into a variable, this time called `chapter` ❹.

In the final line, we construct a line of output that welcomes the reader by their name to the chapter they are on ❺. We use the `echo` command and provide the text we want to display on the screen in double quotes. Then, to fill in the name and chapter number the user entered, we add the variables where they should appear in the message. As noted in Chapter 7, to use the values contained in the variables, you must precede the variable name with the `$` symbol.

Save this file as *WelcomeScript.sh*. The *.sh* extension is the convention for script files. You might have noticed we didn't include the extension earlier; it's not strictly required, and if you leave the extension off, the file will save as a shell script file by default.

Now, let's run this script. Don't forget to give yourself execute permission with `chmod` first; otherwise, the operating system will scold you with a `Permission denied` message.

```
kali >./WelcomeScript.sh
What is your name?
OccupytheWeb
What chapter are you on in Linux Basics for Hackers?
8
Welcome OccupytheWeb to Chapter 8 of Linux Basics for Hackers!
```

As you can see, your script took input from the user, placed it into variables, and then used those inputs to make a greeting for the user.

This is a simple script, but it taught you how to use variables and take input from the keyboard. These are both crucial concepts in scripting that you will need to use in more complex scripts in future.

Your Very First Hacker Script: Scan for Open Ports

Now that you have some basic scripting skills, let's move to some slightly more advanced scripting that has real-world application to hacking. We'll use an example from the world of black hat hacking. Black hat hackers are those with malicious intentions, such as stealing credit card numbers or defacing websites. White hat hackers are those with good intentions, such as helping software developers or system administrators make their systems more secure. Gray hat hackers are those who tend to move between these two extremes.

Before you continue, you need to become familiar with a simple yet essential tool named nmap that comes installed on Kali by default. You've likely heard the name; nmap is used to probe a system to see whether it is connected to the network and finds out what ports are open. From the open ports discovered, you can surmise what services are running on the target system. This is a crucial skill for any hacker or system administrator.

In its simplest form, the syntax for running an nmap scan looks like this:

```
nmap <type of scan><target IP><optionally, target port>
```

Not too difficult. The simplest and most reliable nmap scan is the TCP connect scan, designated with the `-sT` switch in nmap. So, if you wanted to scan IP address 192.168.181.1 with a TCP scan, you would enter the following:

```
nmap -sT 192.168.181.1
```

To take things a step further, if you wanted to perform a TCP scan of address 192.168.181.1, looking to see whether port 3306 (the default port for MySQL) was open, you could enter this:

```
nmap -sT 192.168.181.1 -p 3306
```

Here, `-p` designates the port you want to scan for. Go ahead and try it out now on your Kali system.

Our Task

At the time of this writing, there is a hacker serving time in US federal prison by the name of Max Butler, also known as Max Vision throughout the hacker world. Max was a kind of gray hat hacker. By day, he was an IT security professional in Silicon Valley, and by night, he was stealing and selling credit card numbers on the black market. At one time, he ran the world's largest credit card black market, CardersMarket. Now, Max is serving a 13-year prison term

while at the same time assisting the Computer Emergency Response Team (CERT) in Pittsburgh with defending against hackers.

A few years before Max was caught, he realized that the Aloha Point of Sale (POS) system used by many small restaurants had a technical support backdoor built into it. In this case, the backdoor enabled tech support to assist their clients. Aloha tech support could access the end user's system through port 5505 to provide assistance when the user called for help. Max realized that if he found a system connected to the internet with the Aloha POS system, he could access the system with sysadmin privileges through port 5505. Max was able to enter many of these systems and steal tens of thousands of credit card numbers.

Eventually, Max wanted to find *every* system that had port 5505 open so that he could go from stealing thousands of credit card numbers to stealing millions. Max decided to write a script that would scan millions of IP addresses looking for systems with port 5505 open. Of course, most systems do *not* have port 5505 open so, if they did, it was likely they were running the doomed Aloha POS. He could run this script while at work during the day, then by night hack into those systems identified as having port 5505 open.

Our task is to write a script that will be nearly identical to Max's script, but rather than scan for port 5505 as Max did, our script will scan for systems connected to the ubiquitous online database MySQL. MySQL is an open source database used behind millions of websites; we'll be working with MySQL in Chapter 12. By default, MySQL uses port 3306. Databases are the "Golden Fleece" that nearly every black hat hacker is seeking, as they often contain credit card numbers and personally identifiable information (PII) that is *very* valuable on the black market.

A Simple Scanner

Before we write the script to scan public IPs across the internet, let's take on much a smaller task. Instead of scanning the globe, let's first write a script to scan for port 3306 on a local area network to see whether our script actually works. If it does, we can easily edit it to do the much larger task.

In your text editor, enter the script shown in Listing 8-3.

```
❶ #! /bin/bash

❷ # This script is designed to find hosts with MySQL installed

  nmap ❸-sT 192.168.181.0/24 ❹-p 3306 ❺>/dev/null ❻-oG MySQLscan

❼ cat MySQLscan | grep open > MySQLscan2 ❽

  cat MySQLscan2
```

Listing 8-3: The simplified scanner script

We start with the shebang and the interpreter to use ❶. Let's follow this with a comment to explain what the script does ❷.

Now let's use the `nmap` command to request a TCP scan ❸ on our LAN, looking for port 3306 ❹. (Note that your IP addresses may differ; in your terminal, use the `ifconfig` command on Linux or the `ipconfig` command on Windows to determine your IP address.) To stay stealthy, we also send the standard nmap output that would usually appear on the screen to a special place in Linux, where it disappears ❺. We're doing this on a local machine, so it doesn't matter so much, but if you were to use the script remotely, you'd want to hide the nmap output. We then send the output of the scan to a file named *MySQLscan* in a grep-able format ❻, meaning a format that grep can work on.

The next line displays the *MySQLscan* file we stored the output in and then pipes that output to `grep` to filter for lines that include the keyword `open` ❼. Then we put those lines into a file named *MySQLscan2* ❽.

Finally, you display the contents of the file *MySQLscan2*. This final file should only include lines of output from nmap with hosts that have port 3306 open. Save this file as *MySQLscanner.sh* and give yourself execute permissions with `chmod 755`.

Execute the script, like so:

```
kali >./MySQLscanner.sh

host: 192.168.181.69 () Ports: 3306/open/tcp//mysql///
```

As we can see, this script was able to identify the only IP address on my LAN with MySQL running. Your results may differ, depending on whether any ports are running MySQL installations on your local network, of course.

Improving the MySQL Scanner

Now we want to adapt this script to make it applicable to more than just your own local network. This script would be much easier to use if it could prompt the user for the range of IP addresses they wanted to scan and the port to look for, and then use that input. Remember, you learned how to prompt the user and put their keyboard input into a variable in "Adding Functionality with Variables and User Input" on page 84.

Let's take a look at how you could use variables to make this script more flexible and efficient.

Adding Prompts and Variables to Our Hacker Script

In your text editor, enter the script shown in Listing 8-4.

```
#! /bin/bash

❶ echo "Enter the starting IP address : "
❷ read FirstIP

❸ echo "Enter the last octet of the last IP address : "
  read LastOctetIP
```

```
❹ echo "Enter the port number you want to scan for : "
  read port

❺ nmap -sT $FirstIP-$LastOctetIP -p $port >/dev/null -oG MySQLscan

❻ cat MySQLscan | grep open > MySQLscan2

❼ cat MySQLscan2
```

Listing 8-4: Your advanced MySQL port scanner

The first thing we need to do is replace the specified subnet with an IP address range. We'll create a variable called `FirstIP` and a second variable named `LastOctetIP` to create the range as well as a variable named `port` for the port number (the last octet is the last group of digits after the third period in the IP address. In the IP address 192.168.1.101, the last octet is 101).

NOTE *The name of the variable is irrelevant, but best practice is to use a variable name that helps you remember what the variable holds.*

We also need to prompt the user for these values. We can do this by using the `echo` command that we used in Listing 8-1.

To get a value for the `FirstIP` variable, echo `"Enter the starting IP address : "` to the screen, asking the user for the first IP address they want to scan ❶. Upon seeing this prompt on the screen, the user will enter the first IP address, so we need to capture that input from the user.

We can do this with the `read` command followed by the name of the variable we want to store the input in ❷. This command will put the IP address entered by the user into the variable `FirstIP`. Then we can use that value in `FirstIP` throughout our script.

We'll do the same for the `LastOctetIP` ❸ and `port` ❹ variables by prompting the user to enter the information and then using a `read` command to capture it.

Next, we need to edit the `nmap` command in our script to use the variables we just created and filled. To use the value stored in the variable, we simply preface the variable name with `$`, as in `$port`, for example. So at ❺, we scan a range of IP addresses, starting with the first user-input IP through the second user-input IP, and look for the particular port input by the user. We've used the variables in place of the subnet to scan and the port to determine what to scan for. The redirect symbol > tells the standard nmap output, which usually goes to the screen, to instead go to */dev/null* (*/dev/null* is simply a place to send output so that it disappears). Then, we send the output in a grep-able format to a file we named *MySQLscan*.

The next line remains the same as in our simple scanner: it outputs the contents of the *MySQLscan* file, pipes it to `grep`, where it is filtered for lines that include the keyword `open`, and then sends that output to a new file named *MySQLscan2* ❻. Finally, we display the contents of the *MySQLscan2* file ❼.

If everything works as expected, this script will scan IP addresses from the first input address to the last input address, searching for the input port

and then reporting back with just the IP addresses that have the designated port open. Save your script file as *MySQLscannerAdvanced*, remembering to give yourself execute permission.

A Sample Run

Now we can run our simple scanner script with the variables that determine what IP address range and port to scan without having to edit the script every time we want to run a scan:

```
kali >./MySQLscannerAdvanced.sh
Enter the starting IP address :
192.168.181.0
Enter the last IP address :
192.168.181.255
Enter the port number you want to scan for :
3306
Host: 192.168.181.254 ()Ports:3306/open/tcp//mysql//
```

The script prompts the user for the first IP address, the last IP address, and then the port to scan for. After collecting this info, the script performs the nmap scan and produces a report of all the IP addresses in the range that have the specified port open. As you can see, even the simplest of scripting can create a powerful tool. You'll learn even more about scripting in Chapter 17.

Common Built-in Bash Commands

As promised, Table 8-1 gives you a list of some useful commands built into bash.

Table 8-1: Built-in Bash Commands

Command	Function
`:`	Returns `0` or `true`
`.`	Executes a shell script
`bg`	Puts a job in the background
`break`	Exits the current loop
`cd`	Changes directory
`continue`	Resumes the current loop
`echo`	Displays the command arguments
`eval`	Evaluates the following expression
`exec`	Executes the following command without creating a new process
`exit`	Quits the shell
`export`	Makes a variable or function available to other programs
`fg`	Brings a job to the foreground

Command	Function
`getopts`	Parses arguments to the shell script
`jobs`	Lists background (bg) jobs
`pwd`	Displays the current directory
`read`	Reads a line from standard input
`readonly`	Declares as variable as read-only
`set`	Lists all variables
`shift`	Moves the parameters to the left
`test`	Evaluates arguments
`[`	Performs a conditional test
`times`	Prints the user and system times
`trap`	Traps a signal
`type`	Displays how each argument would be interpreted as a command
`umask`	Changes the default permissions for a new file
`unset`	Deletes values from a variable or function
`wait`	Waits for a background process to complete

Summary

Scripting is an essential skill for any hacker or system administrator. It enables you to automate tasks that would normally take hours of your time, and once the script is saved, it can be used over and over again. Bash scripting is the most basic form of scripting, and you will advance to Python scripting with even more capabilities in Chapter 17.

EXERCISES

Before you move on to Chapter 9, try out the skills you learned from this chapter by completing the following exercises:

1. Create your own greeting script similar to our *HelloHackersArise* script.
2. Create a script similar to *MySQLscanner.sh* but design it to find systems with Microsoft's SQL Server database at port 1433. Call it *MSSQLscanner*.
3. Alter that *MSSQLscanner* script to prompt the user for a starting and ending IP address and the port to search for. Then filter out all the IP addresses where those ports are closed and display only those that are open.