EARLY ACCESS

Black Hat Bash

Bash Scripting for Hackers and Pentesters

Dolev Farhi and Nick Aleks

no starch press®

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

The Early Access program lets you read significant portions of an upcoming book while it's still in the editing and production phases, so you may come across errors or other issues you want to comment on. But while we sincerely appreciate your feedback during a book's EA phase, please use your best discretion when deciding what to report.

At the EA stage, we're most interested in feedback related to content—general comments to the writer, technical errors, versioning concerns, or other high-level issues and observations. As these titles are still in draft form, we already know there may be typos, grammatical mistakes, missing images or captions, layout issues, and instances of placeholder text. No need to report these—they will all be corrected later, during the copyediting, proofreading, and typesetting processes.

If you encounter any errors ("errata") you'd like to report, please fill out **this Google form** so we can review your comments.

BLACK HAT BASH

Dolev Farhi and Nick Aleks

Early Access edition, 08/10/23

ISBN 13: 978-1-7185-0374-8 (print)
ISBN 13: 978-1-7185-0375-5 (ebook)

blisher: illia ollock
ana in ditor: ill Franklin
rod ction ana er: Sabrina lo itallo- on le

CONTENTS

The chapters in red are included in this Early Access PDF.

`

1

BASH BASICS

Bash is a shell scripting language used to interact with components of the Linux operating system. As penetration testers and security practitioners, we frequently write bash scripts to automate a wide variety of tasks, making it an essential tool for hackers. In this chapter, you'll set up your bash development environment, explore useful Linux commands to include in future scripts, and learn the fundamentals of the language's syntax, including variables, arrays, streams, arguments, and operators.

`

Environmental Setup

Before you begin learning bash, you need both a bash shell running in a terminal and a text editor. You can access these on any major operating system by following the instructions in this section.

NOTE Beginning in Chapter 4, you'll run bash commands and complete hacking labs using Kali Linux. If you'd like to set up Kali now, consult the steps included in Chapter 3.

Accessing the Bash Shell

If you're running Linux or macOS, bash should already be available. On Linux, open the Terminal application by entering ALT+CTRL+T. On macOS, you can find the terminal by navigating to the Launchpad icon on the system dock.

Kali and macOS use the Z Shell by default, so when you open a new terminal window, you'll have to enter `exec bash` to switch to a bash shell before you run commands. If you want to change your default shell to bash so you don't have to manually switch shells, you can use the command `chsh -s /bin/bash`.

If you're running Windows, you can make use of the *Windows Subsystem for Linux (WSL)*, which lets you run Linux distributions directly on Windows, providing access to a bash environment without a virtual machine. The official Microsoft WSL documentation page describes how to install it: *https://learn.microsoft.com/en-us/windows/wsl/install*.

An alternative to WSL is Cygwin, which emulates a Linux environment by providing a collection of Linux utilities and system-call functionalities. To install Cygwin, visit *https://www.cygwin.com/install.html* and download the setup file, then follow the installation wizard.

Cygwin installs itself by default to the *C:/cygwin64/* Windows path. To execute your bash scripts, save the scripts in the directory containing your username at *C:/cygwin64/home*. For example, if your username is *david*, you should save your scripts under *C:/cygwin64/home/david*. Then, from the Cygwin terminal, you'll be able to change the directory to the home directory to run your scripts.

`

Installing a Text Editor

To start writing bash scripts, you'll need a text editor, preferably one with handy features such as syntax highlighting built in. You can choose between terminal-based text editors and graphical user interface-based text editors. Terminal-based text editors (such as vi or nano) are useful, because during a penetration test, they may be the only available options when you need to develop a script on the spot.

If you prefer graphical text editors, Sublime Text (*https://www.sublimetext.com/*) is one option you could use. In Sublime Text, you can toggle on the syntax highlighting feature for bash scripts by clicking Plain Text in the bottom-right corner and choosing bash from the drop-down list of languages. If you're using a different text editor, reference its official documentation to learn how to turn on syntax highlighting.

Exploring the Shell

Now that you have a functional bash environment, it's time to learn some basics. Although you'll develop scripts in your text editor, you'll also probably find yourself frequently running single commands in the terminal. This is because you often need to see how a command runs, and what kind of output it produces, before including it in a script. Let's get started by running some bash commands.

Before you begin, the following command will verify that you have bash available:

```
$ bash --version
```

The version in the output will depend on the operating system you are running.

Checking Environment Variables

When running in a terminal, bash loads a set of *environment variables* with every new session that gets invoked. Programs can use these environment variables for various purposes, such as discovering the identity of the user running the script, the location of their home directory, their default shell, and more.

`

To see the list of environment variables set by bash, run the `env` command directly from the shell:

```
$ env

SHELL=/bin/bash
LANGUAGE=en_CA:en
DESKTOP_SESSION=ubuntu
PWD=/home/user
--snip--
```

You can read individual environment variables using the `echo` command. For example, to print the default shell set for the user, use the `SHELL` environment variable preceded by a dollar sign (`$`) and surrounded by curly braces (`{}`). This will cause bash to expand the variable to its assigned value, as shown in Listing 1-1.

```
$ echo ${SHELL}

/bin/bash
```

Listing 1-1 Printing an environment variable to the terminal

Table 1-2 shows a short list of some of the default environment variables available.

Table 1-2 Bash Environment Variables

Variable name	What It Returns
BASH_VERSION	The bash version running
BASHPID	The process ID of the current bash process
GROUPS	A list of groups the running user is a member of
HOSTNAME	The name of the host
OSTYPE	The type of operating system
PWD	The current working directory
RANDOM	A random number between 0 and 32,767
UID	The user ID of the current user
SHELL	The full pathname to the shell

Try checking the values of these environment variables:

```
$ echo ${RANDOM}
8744

$ echo ${UID}
1000

$ echo ${OSTYPE}
linux-gnu
```

`

The full list of environment variables can be found at *https://www.gnu.org/software/bash/manual/html_node/Bash-Variables.html*.

Running Linux Commands

The bash scripts you'll write in this book will run common Linux tools, so if you're not yet familiar with command line navigation and file-modification utilities such as `cd`, `ls`, `chmod`, `mkdir`, and `touch`, try exploring them using the `man` (manual) command. You can insert it before any Linux command to open a terminal-based guide on how to use it and what options it offers, as shown in Listing 1-2.

```
$ man ls

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List information about the FILEs (the current directory by default).
       Sort entries alphabetically if none of -cftuvSUX nor
       --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.
       -a, --all
       do not ignore entries starting with .
--snip--
```

Listing 1-2 The manual page of the `ls` command

Linux commands can accept many types of input on the command line. For example, you can enter `ls` without any arguments to see files and directories, or pass it arguments to, for instance, display the list of files all on one line.

Arguments are passed on the command line using either short form or long form argument syntax, depending on the command in use. *Short-form* syntax uses a single dash (`-`) followed by one or more characters. The following example uses `ls` to list files and directories using a short-form argument syntax:

```
$ ls -l
```

Some commands let you supply multiple arguments by separating the individual arguments or joining them together:

`

```
$ ls -la
$ ls -l -a
```

Note that some commands may throw errors if you attempt to join two arguments using a single dash, so use the `man` command to learn what syntax is permitted.

Some command options may allow you to use *long-form* argument syntax, such as the `--help` command to list the available options. Long-form argument syntax is prepended by the double dash (`--`) symbol:

```
$ ls --help
```

Sometimes, the same command argument supports both short- and long-form argument syntax for convenience. For example, `ls` supports the argument `-a` (all) to display all files including those that are hidden (files starting with a dot in their name are considered hidden files in Linux), but you could also pass the argument `--all`, and the outcome would be identical.

```
$ ls -a
$ ls --all
```

Let's execute some simple Linux commands so you can see the variation of options each offers. First, create a single directory with `mkdir`:

```
$ mkdir directory1
```

Now let's create two directories with `mkdir`:

```
$ mkdir directory directory2
```

Next, list processes with `ps` using short-hand argument syntax, supplying the arguments separately and then together:

```
$ ps -e -f
$ ps -ef
```

Lastly, let's display the available disk space with `df` using long-form argument syntax:

```
$ df --human-readable
```

Later in this book, you'll write your own scripts that can take various arguments.

`

Elements of a Bash Script

In this section, you'll learn the building blocks of a bash script, including how to use comments to document what a script does, how to tell Linux to use a specific interpreter to execute the script, and how to style your scripts for better readability.

Bash doesn't have an official style guide, but we recommend adhering to Google's shell style guide (*https://google.github.io/styleguide/shellguide.html*), which outlines best practices to follow when developing bash code. If you work on a team of penetration testers and have an exploit code repository, using good code styling practices will help your team maintain it.

The Shebang Line

Every script should begin with the *shebang* line, a character sequence that starts with the hash and exclamation mark symbols (`#!`), followed by the full path to the script interpreter. Listing 1-3 shows an example of a shebang line for a typical bash script.

```
#!/bin/bash

--snip--
```

Listing 1-3 The shebang line

The bash interpreter is typically located at */bin/bash*. If you instead wrote scripts in Python or Ruby, your shebang line would include the full path to the Python or Ruby interpreter.

You'll sometimes encounter bash scripts that make use of a shebang line like the following.

```
#!/usr/bin/env bash

--snip--
```

Listing 1-4 A portable shebang line

You may want to use this shebang line because it is more portable than the one in Listing 1-3. Some Linux distributions place the bash interpreter in different system locations, and this shebang line will attempt to find that location. This approach could be particularly useful in penetration tests, where you might not know the location of the bash interpreter on the target machine. For simplicity, however, we'll use the shebang version from Listing 1-3 throughout this book.

`

The shebang line can also take optional arguments to change how the script executes. For example, you could pass the special argument `-x` to your bash shebang, like so:

```
#!/bin/bash -x
```

This option will print all commands and their arguments as they are executed to the terminal. It is useful for debugging scripts as you are developing them.

Another example of an optional argument is `-r`:

```
#!/bin/bash -r
```

This optional argument will create a *restricted bash shell*, which restricts certain potentially dangerous commands that could, for example, navigate to certain directories, change sensitive environment variables, or attempt to turn off the restricted shell from within the script.

Specifying an argument within the shebang line requires modifying the script, but you can also pass arguments to the bash interpreter using the syntax in Listing 1-5.

```
$ bash -r myscript.sh
```

Listing 1-5 Passing an argument to bash

Whether you pass arguments to the bash interpreter on the command line or on the shebang line won't make a difference. The command line option is just an easier way to trigger different modes.

Comments

Comments are parts of a script that the bash interpreter won't treat as code, and they can improve the readability of a program. Imagine that you write a long script and, a few years later, need to modify some of its logic. If you didn't write comments to explain what you did, you might find it quite challenging to remember the purpose of each section.

Comments in bash start with a pound sign (`#`), as shown in Listing 1-6.

```
#!/bin/bash

# This is my first script.
```

Listing 1-6 A comment in a bash script

`

Except for the shebang line, every line that starts with a pound sign is considered a comment. If you wrote the shebang line twice, bash would consider the second one to be a comment.

To write a multiline comment, precede each individual line with the pound sign, as shown in Listing 1-7.

```
#!/bin/bash

# This is my first script!
# Bash scripting is fun...
```

Listing 1-7 A multiline comment

In addition to documenting a script's logic, comments can provide additional metadata, such as who the author is, the script's version, whom to contact for issues, and more. These comments usually appear at the top part of the script, below the shebang line.

Commands

Scripts can be as short as two lines: the shebang line and a Linux command. Let's write a very simple script that prints *Hello World* to the terminal. Open your text editor and enter the following:

```
#!/bin/bash

echo "Hello World!"
```

In this example, we use the shebang statement to specify the interpreter of choice, bash. Then, we use the `echo` command to print the string *Hello World!* to the screen.

Execution

To run the script, save the file as *helloworld.sh*, open the terminal, and navigate to the directory where the script resides. If you saved the file in your home directory, you run the following set of commands:

```
$ cd ~
$ chmod u+x helloworld.sh
$ ./helloworld.sh

Hello World!
```

We use the `cd` command to change directories. The tilde (`~`) represents the home directory of the current running user. Next, we set the executable (`u+x`) permissions using the `chmod` command for

the user who owns the file (in this case, us). Lastly, we run the script using dot slash notation (`./`) followed by the script's name. The dot (`.`) represents the current directory, so we're essentially telling bash to run *helloworld.sh* from the current working directory.

You can also run a bash script with the following syntax:

```
$ bash helloworld.sh
```

Because we specified the `bash` command, the script will run using the bash interpreter and won't require a shebang line. Also, if you use the `bash` command, the script doesn't have to be set with an executable permission (`+x`). In later chapters, you'll learn about the permission model in more depth, and explore its importance in the context of finding misconfigurations in penetration tests.

Debugging

Errors will inevitably occur when you're developing bash scripts. Luckily, debugging the script is quite intuitive. An easy way to check for errors early is by running the script using the `-n` parameter. This parameter will read the commands in the script but won't execute them, so if there are any syntax errors, they will be shown on the screen. you can think of it as a dry-run method to test validity of syntax:

```
bash -n script.sh
```

You can also use the `-x` parameter to turn on verbose mode, which lets you see commands being executed and will help you debug issues as the script executes in real time.

```
bash -x script.sh
```

If you want to start debugging at a given point in the script, you can do this by including the `set` command in the script itself.

```
#!/bin/bash
set -x

--snip--

set +x
```

You can think of `set` as a valve that turns on and off a certain option. The first command sets the debugging mode (`set -x`) while the last command (`set +x`) disables it. Using it, you can avoid

`

generating a massive amount of noise in your terminal in cases when your script is large and contains a specific problem area.

Basic Syntax

At this point, you've written a two-line script that prints the message *Hello World!* to the screen. You've also learned how to run and debug a script. Now you'll learn some bash syntax so you can write more useful scripts.

The most basic bash scripts are just lists of Linux commands collected in a single file. For example, you could write a script that creates resources on a system and then prints information about these resources to the screen (Listing 1-8).

```
#!/bin/bash

# All this script does is create a directory, create a file
# within the directory, and then list the content of the directory.

mkdir directory
touch mydirectory/myfile
ls -l mydirectory
```

Listing 1-8 A basic bash script

In this example, we use `mkdir` to create a directory named *mydirectory*. Next, we use the `touch` command to create a file named *myfile* within the directory. Lastly, we run the `ls -l` command to list the contents of the *mydirectory* directory.

The output of the script looks as follows:

```
--snip--
-rw-r--r-- 1 user user 0 Feb 16 13:37 myfile
```

However, this line-by-line strategy could be improved in several ways. First, when a command runs, bash waits until it finishes before advancing to the next line. This means that if you include a long-running command (such as a file download or large file copy), remaining commands won't be executed until that command has completed. We also have yet to implement any checks to validate that all commands have executed correctly. You'll need to write more intelligent programs to reduce errors during runtime.

Writing sophisticated programs often requires using features like variables, conditions, loops, tests, and so on. For example, what if

we wanted to change this script so that we first check that we have enough space on the disk before attempting to create new files and directories? Or what if we checked whether the directory and file creation actions actually succeeded? This section and the next chapter introduce you to the syntactical elements you'll need to accomplish these tasks.

Variables

Every scripting language has variables. *Variables* are names we assign to memory locations that hold some value, and they act like placeholders or labels. We can directly assign values to variables or execute bash commands and store their output as variable values to use for various purposes.

If you've worked with programming languages, you may know that variables can be of different types, such as integers, strings, and arrays. In bash, variables are untyped; they're all considered character strings. Even so, you'll see that bash allows you to create arrays, access array elements, or perform arithmetic operations so long as the variable value consists of only numbers.

The following rules govern the naming of bash variables:

- They can include alphanumeric characters.
- They cannot start with a numerical character.
- They can contain an underscore (_).
- They cannot contain whitespace.

Assigning and Accessing Variables

Let's assign a variable. Open a terminal and enter the following directly in the command prompt:

```
$ book="black hat bash"
```

We create a variable named `book`, and by using the equal sign (`=`), assign the value `black hat bash` to it. Now we can use this variable in some command. For example, here we use the `echo` command to print the variable to the screen:

```
$ echo "This book's name is ${book}"
This book's name is black hat bash
```

`

In this example, we were able to print the variable by using the `${book}` syntax within an `echo` command. This will expand the `book` variable to its value.

You can also assign the output of a command to a variable using the command substitution syntax `$()`, placing the desired command between the two parentheses. You'll use this syntax often in bash programming. Try running the following:

```
$ root_directory="$(ls -ld /)"
$ echo "${root_directory}"

drwxr-xr-x 1 user user 0 Feb 13 20:12 /
```

We assign the value of the `ls -ld /` command to a variable named `root_directory` and then use `echo` to print the output of the command. In this output, you can see that we were able to get some metadata about the root directory (`/`), such as its type and permission, size, user and group owners, and the timestamp of the last modification.

Note that you shouldn't leave whitespace around the assignment symbol (`=`) when creating a variable. The following variable assignment syntax is considered invalid:

```
book = "this is an invalid variable assignment"
```

Unassigning Variables

You can unassign assigned variables using the `unset` command, as shown here:

```
$ book="Black Hat Bash"
$ unset book
$ echo "${book}"
```

Listing 1-9 Unassigning a variable

If you execute these commands in the terminal, no output will be shown after the `echo` command executes.

Scoping Variables

Global variables are those available to the entire program. But variables in bash can also be *scoped* so that they are only accessible from within a certain block of code. These variables are called *local* variables and are declared using the `local` keyword. The following script shows how local and global variables work:

`

```
#!/bin/bash

PUBLISHER="No Starch Press"

print_name(){
   local name
   name="Black Hat Bash"
   echo "${name} by ${PUBLISHER}"
}

print_name

echo "The variable ${name} will not be printed because it is a local variable."
```

We assign the value `No Starch Press` to the variable `publisher`, then create a function called `print_name()`. (You'll learn more about functions in the next chapter.) Within the function, we declare a local variable called `name` and assign it the value `"Black Hat Bash"`. Then we call the `print_name()` function and attempt to access the `name` variable as part of a sentence to be printed using `echo`.

The `echo` command at the end of the script file will result in an empty variable, as the `name` variable is locally scoped to the `print_name()` function, which means that nothing outside of the function can access it. So, it will simply return without a value.

Save this script, remembering to set the executable permission using `chmod`, and run it using the following command:

```
$ ./local_scope_variable.sh

Black Hat Bash by No Starch Press

The variable will not be printed because it is a local variable.
```

This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch01/local_scope_variable.sh*.

Arithmetic Operators

Arithmetic operators allow you to perform mathematical operations on integers. Table 1-3 shows some of the arithmetic operators available. The full list of the available arithmetic operators can be found at *https://tldp.org/LDP/abs/html/ops.html*.

Table 1-3 Arithmetic Operators

Operator	Description

`

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
+=	Incrementing by a constant
-=	Decrementing by a constant

You can perform these arithmetic operations in bash in a few ways: using the `let` command, using the double parentheses syntax `$((expression))`, and using the `expr` command. Let's consider an example of each method.

Here we perform a multiplication operation using the `let` command:

```
$ let result="4 * 5"
$ echo $result

20
```

This command takes a variable name, then performs an arithmetic calculation to resolve its value. Next, we perform another multiplication operation using the double parentheses syntax:

```
$ result=$((5 * 5))
$ echo $result

25
```

In this case, we perform the calculation within double parentheses. Lastly, we perform an addition operation using the `expr` command:

```
$ result=$(expr 5 + 505)
$ echo $result

510
```

The `expr` command evaluates expressions, which don't have to be arithmetic operations; for example, you might use it to calculate the length of some string. Use `man expr` to learn more about the capabilities of `expr`.

Arrays

Bash allows you to create single-dimension arrays. *Arrays* are a collection of elements that are indexed. You can access these elements using their index numbers, where the first indexed number starts from zero. In bash scripts, you might use arrays whenever you

`

need to iterate over a number of strings and run the same commands on each one.

Here is how to create an array in bash. Save this code to a file named *array.sh* and execute it.

```
#!/bin/bash

# Set an array
IP_ADDRESSES=(192.168.1.1 192.168.1.2 192.168.1.3)

# Print all elements in the array
echo "${IP_ADDRESSES[*]}"

# Print only the first element in the array
echo "${IP_ADDRESSES[0]}"
```

This script uses an array named `IP_ADDRESSES` that contains three IP addresses. The first `echo` command prints all the elements in the array by passing `[*]` to the variable name `IP_ADDRESSES`, which holds the array values. The `*` is a representation of every array element. Lastly, another `echo` command prints just the first element in the array by specifying index zero.

Running this script should produce the following output:

```
$ chmod u+x arrays.sh
$ ./arrays.sh

192.168.1.1 192.168.1.2 192.168.1.3
192.168.1.1
```

As you can see, we were able to get bash to print all elements in the array, as well as just the first element.

You can also delete elements from an array. The following will delete 192.168.1.2 from the array:

```
IP_ADDRESSES=(192.168.1.1 192.168.1.2 192.168.1.3)

unset IP_ADDRESSES[1]
```

You can even swap one of the values with another value. The following code will replace 192.168.1.1 with 192.168.1.10:

```
IP_ADDRESSES[0]="192.168.1.10"
```

You'll find arrays particularly useful when you need to iterate over values and perform actions against them, such as a list of IP addresses to scan (or a list of emails to send a phishing email to).

`

Streams

Streams are files that act as communication channels between a program and its environment. When you interact with a program (whether a built-in Linux utility such as `ls` or `mkdir` or one that you wrote yourself), you're interacting with one or more streams. In bash, there are three standard data streams, as shown in Table 1-4.

Table 1-4 Streams

Stream name	Description	File descriptor number
Standard Input (stdin)	Data coming into some program as input	0
Standard Output (stdout)	Data coming out of a program	1
Standard Error (stderr)	Errors coming out of a program	2

So far, we've run a few commands from the terminal and written and executed a simple script. The generated output was all sent to the *standard output stream (stdout)*, or in other words, your terminal screen.

Scripts can also receive commands as input. When a script is designed to receive input, it reads it from the *standard input stream (stdin)*. Lastly, scripts may display error messages to the screen due to a bug or syntax error in the commands sent to it. These messages are sent to the *standard error stream (stderr)*.

To illustrate streams, we'll use the `mkdir` command to create a few directories and then use `ls` to list the content of the current directory. Open your terminal and execute the following command:

```
$ mkdir directory1 directory2 directory1
mkdir: cannot create directory 'directory1': File exists

$ ls -l
total 1
drwxr-xr-x 1 user user   0 Feb 17 09:45 directory1
drwxr-xr-x 1 user user   0 Feb 17 09:45 directory2
```

Notice that the `mkdir` command generated an error. This is because we passed the directory name *directory1* twice on the command line. So, when `mkdir` ran, it created *directory1*, created *directory2*, and then failed on the third argument because, at that point, *directory1* had already been created. These types of errors are sent to the standard error stream.

`

Next, we executed `ls -l`, which simply listed the directories. The result of the `ls` command succeeded without any specific errors, so it was sent to the standard output stream.

You'll practice working with the standard input stream when we introduce redirection in "Redirection Operators" on page XX.

Control Operators

Control operators in bash are tokens that perform a control function. Table 1-5 gives an overview of control operators.

Table 1-5 Bash Control Operators

Operator	Description
&	Sends a command to the background.
&&	Used as a logical `AND`. The second command in the expression will be evaluated only if the first command evaluated to true.
(and *)*	Used for command grouping
;	Used as a list terminator. A command following the terminator will run after the preceding command has finished, regardless of whether it evaluates to true or not.
;;	Ends a `case` statement.
|	Redirects the output of a command as input to another command.
||	Used as a logical `OR`. The second command will run if the first one evaluates to false.

Let's see some of these control operators in action. The `&` operator sends any command to the background. If you have a list of commands in a shell script, sending the first command to the background will allow bash to continue to the next line even if the previous command hasn't finished its work. Commands that are long-running are often sent to the background to prevent scripts from hanging.

```
#!/bin/bash

# This script will send the sleep command to the background
echo "Sleeping for 10 seconds..."
sleep 10 &

# Creates a file
echo "Creating the file test123"
touch test123

# Delete a file
echo "Deleting the file test123"
rm test123
```

`

You'll learn about sending commands to the background in more depth when we discuss job control in Chapter 2.

The `&&` operator allows us to perform an `AND` operation between two commands. In this case, the file *test123* will be created only if the first command was successful:

```
touch test && touch test123
```

The `()` operator allows us to group commands together so they act a single unit when we need to redirect them together:

```
(ls; ps)
```

This is generally useful when you need to redirect results from multiple commands to some stream, as shown in "Redirection Operations" on page XX.

The `;` operator allows us to run multiple commands irrespective of their exit status:

```
ls; ps; whoami
```

As a result, each command is executed one after the other, as soon as the previous one finishes.

The `||` operator allows us to chain commands together using an `OR` operator:

```
lzl || echo "the lzl command failed"
```

In this example, the `echo` command will be executed only if the first command fails.

Redirection Operators

The three standard streams we highlighted earlier can be redirected from one program to another. *Redirection* is taking some output from one command or script and using it as the input to another. Table 1-6 describes the available redirection operators.

Table 1-6 Redirection Operators

Operator	Description
>	Redirects stdout to a file
>>	Redirects stdout to a file by appending it to the existing content
&> or >&	Redirects stdout and stderr to a file
&>>	Redirects stdout and stderr to a file by appending it to the existing content
<	Redirects input to a command
<<	Called a *here document* or *heredoc*, redirects multiple input lines to a command

`

|	Redirects output of a command as input to another command

Let's practice using redirection operators to see how they work with standard streams. The > operator redirects the standard output stream to a file. Any command that precedes this character will send its output to the specified location. Run the following command directly in your terminal:

```
$ echo "Hello World!" > output.txt
```

We redirected the standard output stream to a file named *output.txt*. To see the content of *output.txt*, simply run the following:

```
$ cat output.txt

Hello World!
```

Next, we'll use the >> operator to append some content to the end of the same file:

```
$ echo "Goodbye!" >> output.txt
$ cat hello_output.txt

Hello World!
Goodbye!
```

Listing 1-10 Append text to a file

If we used > instead of >>, the content of *output.txt* would have been overwritten completely with the *Goodbye!* text.

You can redirect both the standard output stream and the standard error stream to a file using &>. This is useful when you don't want to send any output to the screen and instead save everything in a log file (perhaps for later analysis).

```
ls -l / &> stdout_and_stderr.txt
```

Listing 1-11 Redirecting standard output and standard error streams to a file

To append both the standard output and standard error streams to a file, simply use double chevron (&>>).

What if we wanted to send the standard output stream to one file, and the standard error stream to another? This is also possible using the streams' file descriptor numbers:

```
$ ls -l / 1> stdout.txt 2> stderr.txt
```

Listing 1-12 Redirecting standard output and standard error to separate files

`

You may sometimes find it useful to redirect the standard error stream to a file, as we've done here, so you can log any errors that occur during runtime. For example, the next example runs a non-existent command `lzl`. This should generate bash errors that will be written into the *error.txt* file.

```
$ lzl 2> error.txt
$ cat error.txt

bash: lz: command not found
```

Listing 1-13 Redirecting the standard error stream using its file descriptor number

Notice that you didn't see the error on the screen because bash sent the error to the file instead.

Next, let's use the standard input stream. Run the following command in the shell to supply the contents of *output.txt* as input to the `cat` command:

```
$ cat < output.txt

Hello World!
Goodbye!
```

Listing 1-14 Redirecting standard input to the `cat` command

What if we wanted to redirect multiple lines to a command? *Here document* redirection (`<<`) can help with this:

```
$ cat << EOF
  Black Hat Bash
  by No Starch Press
EOF

Black Hat Bash
by No Starch Press
```

Listing 1-15 A here document

In this example, we pass multiple lines as input to a command. The `EOF` in this example acts as a delimiter marking the start and end points of the input. Here document redirection treats the input as if it were a separate file, preserving line breaks and whitespace.

The *pipe* operator (`|`) redirects the output of one command and uses it as the input of another. For example, we could run the `ls` command on the root directory and then use another command to extract some data from it, as shown here:

```
$ ls -l / | grep "bin"
```

`

```
lrwxrwxrwx   1 root root          7 Mar 10 08:43 bin -> usr/bin
lrwxrwxrwx   1 root root          8 Mar 10 08:43 sbin -> usr/sbin
```

Listing 1-16 Piping a commands output as input to another command

We used `ls` to print the content of the root directory into the standard output stream, then used pipe (`|`) to send it as input to the `grep` command, which filtered out any lines containing the word *bin*.

Positional Arguments

Bash scripts can take positional arguments (also called *parameters*) passed on the command line. Arguments are especially useful, for example, when you want to develop a program that modifies its behavior based on some input passed to it by another program or user. Arguments can also change features of the script such as the output format and how verbose it will be during runtime.

For example, imagine you develop an exploit and send it to a few colleagues, each of whom will use it against a different IP address. Instead of writing a script and asking the user to modify it with their network information, you can write it to take an IP address argument and then act against this input to avoid having to modify the source code in each case.

A bash script can access arguments passed to it on the command line using the variables `$1`, `$2`, and so on. The number represents the order in which the argument was entered. To illustrate this, the following script takes in an argument (an IP address or domain name) and performs a ping test against it using the `ping` utility. Save this file as *ping_with_arguments.sh*:

```
#!/bin/bash

# This script will ping any address provided as an argument.

SCRIPT_NAME="${0}"
TARGET="${1}"

echo "Running the script ${SCRIPT_NAME}..."
echo "Pinging the target: ${TARGET}..."
ping "${TARGET}"
```

Listing 1-17 A pinger command

`

This script assigns the first positional argument to the variable `TARGET`. Notice, also, that the argument `${0}` is assigned to the `SCRIPT_NAME` variable. This argument contains the script's name (in this case, *ping_with_arguments.sh*).

To run this script, use the following commands:

```
$ chmod u+x ping_with_arguments.sh
$./ping_with_arguments.sh nostarch.com

Running the script ping_with_arguments.sh...
Pinging the target nostarch.com...
PING nostarch.com (104.20.120.46) 56(84) bytes of data.

64 bytes from 104.20.120.46 (104.20.120.46): icmp_seq=1 ttl=57 time=6.89 ms
64 bytes from 104.20.120.46 (104.20.120.46): icmp_seq=2 ttl=57 time=4.16 ms
--snip--
```

Listing 1-18 Passing arguments to a script

This script will perform a `ping` command against the domain *nostarch.com* passed to it on the command line. The value was assigned to the `$1` variable; if we passed another argument, it would get assigned to the second variable, `$2`. Use CTRL+C to exit this script, as `ping` may run indefinitely on some operating systems. This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch01/ping_with_arguments.sh*.

What if you wanted to access all arguments? You can do so using the variable `$@`. Also, using `$#`, you can get the total number of arguments passed. The following script demonstrates how this works:

```
#!/bin/bash

echo "The arguments are: $@"
echo "The total number of arguments are: $#"
```

Save this script to a file named *show_args.sh* and run it as follows:

```
$ chmod u+x show_args.sh
$ ./show_args.sh "hello" "world"

The arguments are: hello world
The total number of arguments are: 2
```

Table 1-7 summarizes the variables related to positional arguments.

`

Table 1-7 Special Variables Related to Positional Arguments

Variable	Description
`$0`	The name of the script file
`$1`, `$2`, `$3`, […]	Positional arguments
`$#`	The number of passed positional arguments
`$*`	All positional arguments
`$@`	All positional arguments, where each argument is individually quoted

When a script makes use of `"$*"` with the quotes included, bash will expand arguments into a single word. For instance, the following groups the arguments into one word:

```
$ script.sh "1" "2" "3"
1 2 3
```

When a script makes use of `"$@"` (again including the quotes), it will expand arguments into separate words:

```
$ script.sh "1" "2" "3"
1
2
3
```

In most cases, you will want to use `"$@"` so that every argument is treated as an individual word.

Input Prompting

Some bash scripts don't take any arguments during execution. However, they may need to ask the user for some information in an interactive way and have the response feed into their runtime. In these cases, we can use the `read` command. You often see applications use input prompting when attempting to install some software, asking the user to enter *yes* to proceed or *no* to cancel the operation.

In the following bash script, we ask the user for their first and last name, then print these to the standard output stream.

```
#!/bin/bash

# Takes input from the user and assigns it to variables.
echo "What is your first name?"
read -r firstname

echo "What is your last name?"
read -r lastname

echo "Your first name is ${firstname} and your last name is ${lastname}"
```

`

Listing 1-19 Accepting user input with bash

Save and run this script as *input_prompting.sh*. Notice that you get prompted to enter information that then get printed:

```
$ chmod u+x input_prompting.sh
$./input_prompting

What is your first name?
John

What is your last name?
Doe

Your first name is John and your last name is Doe
```

This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch01/input_prompting.sh*.

Exit Status Codes

Bash commands return *status codes*, which indicate whether the execution of the command succeeded. Status codes fall in the 0–255 range, where 0 means success, 1 means failure, 126 means that the command was found but is not executable, and 127 means the command was not found. The meaning of any other number depends on the specific command being used and the logic it uses.

Checking Status Codes

To see status codes in action, save the following script to a file named *exit_codes.sh* and run it.

```
#!/bin/bash

# Experimenting with status codes

ls -l > /dev/null
echo "The status code of the ls command was: $?"

lzl 2> /dev/null
echo "The status code of the non-existing lzl command was: $?"
```

We use the special variable `$?` with the `echo` command to return the status codes of the executed commands `ls` and `lzl`. We also redirect their standard output and standard error streams to the file */dev/null*, a special device file that discards any data sent to it. When you want to silence commands, you can redirect their output to it.

`

You should see output like the following:

```
$ ./exit_codes.sh

The status code of the ls command was: 0
The status code of the non-existing lzl command was: 127
```

As you can see, we received two distinct status codes, one for each command. The first command returned 0 (success), and the second returned 127 (command not found).

NOTE Use */dev/null* with caution. You may miss out on important errors if you choose to redirect output to it. When in doubt, redirect standard streams such as standard output and standard error to a dedicated log file instead.

To understand why you might want to use status codes, imagine you're trying to download a 1GB file from the internet using bash. It might be wise to first check if the file already exists on the filesystem in case someone ran the script already and retrieved it. Also, you might want to check that you have enough free space on the disk before attempting the download. By running commands and looking at their exit status codes, we can decide whether to proceed with the file download.

Setting a Script's Exit Codes

You can set the exit code of a script using the `exit` command, as shown below:

```
#!/bin/bash

# Sets the exit code of the script to be 223

echo "Exiting with status code: 223"
exit 223
```

Save this script as *set_status_code.sh* and run it on the command line, then use the special variable `$?` to see the status code it returns:

```
$ chmod u+x set_status_code.sh
$ ./set_status_code.sh
Exiting with status code: 223

echo $?
223
```

You can use the `$?` variable to check the exit status of a script, but also of individual commands:

```
$ ps -ef
$ echo $?

0
```

Exit codes are important when you have a series of scripts that call one another. You may have a workflow where one script invokes another depending on a state of a specific condition.

Exercise 1: Recording Your Name and the Date

To practice what you've learned in this chapter, we encourage you to write a script that does the following:

1. Accepts two arguments on the command line and assigns them to variables. The first argument should be your first name and the second should be your last name.
2. Creates a new file named *output.txt*.
3. Writes the current date to *output.txt* using the `date` command. (Bonus points if you can make the `date` command print the date in the *DD-MM-YYYY* format; use `man date` to learn how this works.)
4. Writes your full name to *output.txt*.
5. Makes a backup copy of *output.txt*, named *backup.txt*, using the `cp` command. (Use `man cp` if you aren't sure of the command's syntax.)
6. Prints the content of *output.txt* file to the standard output stream.

An example solution can be found at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch01/exercise_solution.sh*.

Summary

In this chapter, you ran simple Linux commands in the terminal and learned about command options using the `man` command. You also learned how to pass arguments to scripts and execute a sequence of commands from within scripts. We covered the fundamentals of bash, such as how to write basic programs that make use of variables, arrays, redirects, exit codes, arguments, and more. We

`

were also able to prompt the user to enter arbitrary information and use it as part of a script's flow.

`

2

ADVANCED BASH CONCEPTS

This chapter covers bash concepts that can make your scripts more intelligent. You'll learn how to test conditions, use loops, consolidate code into functions, send commands to the background, and more. You'll also learn some ways of customizing your bash environment for penetration testing.

Test Operators

Bash lets us selectively execute commands when certain conditions of interest are met. We can use *test operators* to craft a wide variety of conditions, such as whether one value equals another value, whether a file is of a certain type, or whether one value is greater than another. We often rely on such tests to determine

`

whether to continue running a block of code or not, so being able to construct them is fundamental to bash programming.

There are multiple kinds of test operators. *File test operators* allow us to perform tests against files on the filesystem, such as checking if a file is executable or if some directory exists. Table 2-1 shows a short list of the available tests.

Table 2-1 File Test Operators

Operator	Description
`-d FILE`	Checks whether the file is a directory
`-r FILE`	Checks whether the file is readable
`-x FILE`	Checks whether the file is executable
`-w FILE`	Checks whether the file is writable
`-f FILE`	Checks whether the file is a regular file
`-s FILE`	Checks whether the file size is greater than zero

You can find the full list of file test operators at *https://ss64.com/bash/test.html*.

String comparison operators allow us to perform tests related to strings, such as testing whether one string is equal to another. Table 2-2 shows the string comparison operators.

Table 2-2 String Comparison Operators

Operator	Description
`=`	Checks whether a string is equal to another string
`==`	Synonym of `=` when used within `[[ ]]` constructs
`!=`	Checks whether a string is not equal to another string
`<`	Checks whether a string comes before another string (in alphabetical order)
`>`	Checks whether a string comes after another string (in alphabetical order)
`-z`	Checks whether a string is null
`-n`	Checks whether a string is not null

Integer comparison operators allow us to perform checks on integers, such as if an integer is less than or greater than another. Table 2-3 shows the available operators.

Table 2-3 Integer Comparison Operators

Operator	Description
`-eq`	Checks whether a number is equal to another number
`-ne`	Checks whether a number is not equal to another number
`-ge`	Checks whether a number is greater than or equal to another number
`-gt`	Checks whether a number is greater than another number
`-lt`	Checks whether a number is less than another number
`-le`	Checks whether a number is less than or equal to another number

Let's use these operators in flow-control mechanisms to decide what code to run next.

`

if Conditions

In bash, we can use an `if` condition to execute some code only when a condition is met. Its syntax is as follows:

```
if [[ condition ]]; then
  # do something if the condition is met
else
  # do something if the condition is not met
fi
```

We start with the `if` keyword, followed by a test conditions between double brackets (`[[ ]]`). We then use the `;` character to separate the `if` keyword from the `then` keyword, which allows us to introduce a block of code that runs only if the condition is true.

Next, we use the `else` keyword to introduce a fallback code block that runs if the condition is not met. Note that `else` is optional, and you may not always need it. Finally, we close the `if` condition with the `fi` keyword (which is `if` inversed).

NOTE In some operating systems, such as those often used in containers, the default shell might not necessarily be bash. To account for these cases, you may want to use single brackets (`[…]`) rather than double brackets to enclose your condition. This use of single brackets meets the POSIX standard and should work on almost any Unix derivative, including Linux.

Let's see an `if` condition in practice. Listing 2-1 uses an `if` condition to test whether a file exists, and if not, creates it.

```
#!/bin/bash

FILENAME="flow_control_with_if.txt"

if [[ -f "${FILENAME}" ]]; then
  echo "${FILENAME} already exists."
  exit 1
else
  touch "${FILENAME}"
fi
```

Listing 2-1 An `if` condition to test for the existence of a file

We first create a variable named `FILENAME` containing the name of the file we need. This saves us from having to repeat the filename in the code. We then introduce the `if` statement, which uses a condition that tests for the existence of files with the `-f` file test operator. If this condition is true, we use `echo` to print to the screen a message explaining that the file already exists and then exit the program using the status code `1` (failure). Using the `else` block, which will execute only if the file does not exist, we create the file using the `touch` command.

`

You can download this script from *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/test_if_file_exists.sh*. Save the file and execute it. You should see the *flow_control_with_if.txt* file in your current directory when you run `ls`.

Listing 2-2 shows a different way of achieving the same outcome; it uses the not operator (`!`) to check whether a directory doesn't exist, and if so, creates it. This example has fewer lines of code and eliminates the need for an `else` block altogether.

```
#!/bin/bash

FILENAME="flow_control_with_if.txt"

if [[ ! -f "${FILENAME}" ]]; then
  touch "${FILENAME}"
fi
```

Listing 2-2 An example of a file test using a negative check

Save and run this script. It should create a directory named *downloads* if this directory wasn't already present.

Let's explore `if` conditions that use some of the other kinds of test operators we've covered. Listing 2-3 shows an example of a string comparison test. It tests whether two variables are equal by performing string comparison with the equal-to operator (`==`).

```
#!/bin/bash

VARIABLE_ONE="nostarch"
VARIABLE_TWO="nostarch"

if [[ "${VARIABLE_ONE}" == "${VARIABLE_TWO}" ]]; then
  echo "They are equal!"
else
  echo "They are not equal!"
fi
```

Listing 2-3 A string comparison test comparing two string variables

The script will compare the two variables, both of which have the value `nostarch`, and print `They are equal!` by using the `echo` command. It is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/string_comparison.sh*.

Next is an example of an integer comparison test, which takes two integers and checks which one is the larger number.

```
#!/bin/bash

VARIABLE_ONE="10"
VARIABLE_TWO="20"
```

`

```
if [[ "${VARIABLE_ONE}" -gt "${VARIABLE_TWO}" ]]; then
  echo "${VARIABLE_ONE} is greater than ${VARIABLE_TWO}"
else
  echo "${VARIABLE_ONE} is less than ${VARIABLE_TWO}"
fi
```

We create two variables, `VARIABLE_ONE` and `VARIABLE_TWO`, and assign them values of 10 and 20, respectively. We then use the `-gt` operator to compare the two values and print the one that is greater. This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/integer_comparison.sh*.

Linking Conditions

So far, we've used `if` to check whether a single condition is met. But like most programming languages, we can also use the *or* (`||`) and *and* (`&&`) operators to check for multiple conditions at once.

For example, what if we wanted to check that a file exists and also that its size greater than zero? Listing 2-4 does so.

```
#!/bin/bash

echo "Hello World!" > file.txt

if [[ -f "file.txt" ]] && [[ -s "file.txt" ]]; then
  echo "The file exists and its size is greater than zero".
fi
```

Listing 2-4 Chaining two file-test conditions using an *and* condition

This code writes some content to a file, then checks whether that file exists and whether its size is greater than zero. Both conditions have to be met in order for the `echo` command to be executed. If either returns false, nothing will happen.

To demonstrate an *or* condition, Listing 2-5 checks whether a file is either of file or directory type:

```
#!/bin/bash

DIR_NAME="dir_test"

mkdir "${DIR_NAME}"

if [[ -f "${DIR_NAME}" ]] || [[ -d "${DIR_NAME}" ]]; then
  echo "${DIR_NAME} is either a file or a directory."
fi
```

Listing 2-5 Chaining two file test conditions using *or*

This code first creates a directory, then uses an `if` condition with the *or* (`||`) operator to check whether the variable is a file (`-f`) or directory (`-d`). The second condition should evaluate to true, and the `echo` command should execute.

`

Running Code Only If a Command Succeeds

We can even test the exit code of commands to determine if they were successful or not, in this way:

```
if command; then
  # command was successful
fi

if ! command; then
  # command was unsuccessful
fi
```

Listing 2-6 Testing the exit code of a command

You'll often find yourself using this technique in bash, as commands aren't guaranteed to succeed. Failures could happen for reasons such as these:

- A lack of the necessary permissions when creating resources
- An attempt to execute a command that is not available on the operating system
- The disk being full when downloading a file
- The network being down while executing network utilities

To see how this technique works, execute the following in your terminal:

```
$ if touch test123; then
    echo "OK: file created"
  fi

OK: file created
```

We attempt to create a file. Because the file creation succeeds, we print a message to indicate this.

Using elif

If the first `if` condition fails, you can check for other conditions by using the `elif` keyword (short for *else if*). To show how this works, let's write a program that checks the arguments passed to it on the command line. The script in Listing 2-7 will output a message clarifying whether the argument is a file or a directory.

```
#!/bin/bash

USER_INPUT="${1}"

1 if [[ -z "${USER_INPUT}" ]]; then
    echo you must provide an argument!
    exit 1
  fi

2 if [[ -f "${USER_INPUT}" ]]; then
```

```
    echo "${USER_INPUT} is a file"
3 elif [[ -d "${USER_INPUT}" ]]; then
    echo "${USER_INPUT} is a directory"
  else
  4 echo "${USER_INPUT} is not a file or a directory"
  fi
```

Listing 2-7 Using `if` and `elif` statements

We begin with an `if` statement that checks whether the variable `USER_INPUT` is null 1. This allows us to exit the script early using `exit 1` if we receive no command line arguments from the user. We then begin a second `if` condition that uses the file test operator to check whether the input is a file 2. Below this condition, we use `elif` to test whether the argument is a directory 3. This condition won't be tested unless the file test fails. If neither of these conditions is true, the script responds that the argument is neither a file nor a directory 4.

This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/if_elif.sh*.

Functions

Functions help us reuse blocks of code so we can avoid repeating them. They allow us to run multiple commands and other bash code together by simply entering the function's name. To define a new function, enter a name for it, followed by parentheses `()`. Then place the code you would like the function to run within curly brackets `{}`:

```
#!/bin/bash

say_name(){
  echo "Black Hat Bash"
}
```

Here, we define a function called `say_name` that executes a single `echo` command. To call a function, simply enter its name:

```
say_name
```

If the function is not called, the commands within it won't run.

Returning Values

Like commands and their exit statuses, functions can return values using the `return` keyword. If there is no `return` statement, the function will return the code of the last command it ran. For example, the function in Listing 2-8 returns a different value based on whether the current user is root or not:

```
#!/bin/bash

1 check_if_root(){
```

`

```
  2 if [[ "${EUID}" -eq "0" ]]; then
      return 0
    else
      return 1
    fi
  }

3 is_root=$(check_if_root)
4 if [[ "${is_root}" -eq "0" ]]; then
    echo "user is root!"
  else
    echo "user is not root!"
  fi
```

Listing 2-8 An `if` condition to test whether a function returned true or false

We define the `check_if_root` function 1. Within this function, we use an `if` condition with an integer comparison test 2, accessing the environment variable `EUID` to get the effective running user's ID and checking whether it equals 0. If so, the user is root, and the function returns 0; if not, it returns 1. Next, we call the function and assign the result returned to the variable `is_root` 3. Then we use another integer comparison test to determine this value 4. This code is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/check_root_function.sh*.

Bash scripts that perform privileged actions often check whether the user is root before attempting to install software, create users, delete groups, and so on. Attempting to perform privileged actions on Linux without the necessary privileges will result in errors, so this check helps handle these cases.

Accepting Arguments

In the previous chapter, we covered the passing of arguments to commands on the command line. Functions can also take arguments using the same syntax. For example, the function in Listing 2-9 prints the first three arguments it receives:

```
#!/bin/bash

print_args(){
  echo "first: ${1}, second: ${2}, third: ${3}"
}

1 print_args No Starch Press
```

Listing 2-9 A function with arguments

To call a function with arguments, simply enter its name and the arguments separated by spaces 1. Save this script as *function_with_args.sh* and run it:

```
$ chmod u+x function_with_args.sh
$ ./function_with_args.sh

first: No, second: Starch, third: Press
```

`

You should see similar output.

Loops and Loop Controls

Like many programming languages, bash lets you repeat chunks of code using *loops*. Loops can be particularly useful in your penetration testing adventures because they can help you accomplish tasks such as the following:

- Continuously checking whether an IP address is online after a reboot; once the IP address is detected, stop checking.
- Iterating through a list of hostnames, for example to run a specific exploit against each of them or determine whether there is a firewall protecting them.
- Testing for a certain condition and then running a loop when it is met. For example, checking whether a host is online, and if so, performing a brute force attack against it.

The following sections introduce you to the three kinds of loops in bash, `while`, `until`, and `for`, as well as the `break` and `continue` statements for working with loops.

while

In bash, `while` loops allow you to run a code block until some test returns a successful exit status code. You might use them in penetration testing to continuously perform a port scan on a network and pick up any new hosts that join the network, for example.

Listing 2-10 shows the syntax of a `while` loop.

```
while some_condition; do
   # run commands while the condition is true
done
```

Listing 2-10 A `while` loop

A `while` loop starts with the keyword `while`, followed by an expression that describes the condition. We then surround the code to be executed with the `do` and `done` keywords, which define the start and end of the code block.

You can use `while` loops to run some chunk of code infinitely by using `true` as the condition; because `true` always returns a successful exit code, the code will always run. Let's use a `while` loop to repeatedly print a command to the screen. Save this script to a file named *basic_while.sh* and run it.

```
#!/bin/bash

while true; do
  echo "Looping..."
  sleep 2
done
```

`

You should see the following output:

```
$ chmod u+x basic_while.sh
$ ./basic_while.sh

Looping...
Looping...
--snip--
```

Next, let's write a more sophisticated `while` loop that runs until it finds a specific file on the filesystem (use CTRL+C to stop it from executing at any point):

```
#!/bin/bash

1 SIGNAL_TO_STOP_FILE="stoploop"

2 while [[ ! -f "${SIGNAL_TO_STOP_FILE}" ]]; do
    echo "The file ${SIGNAL_TO_STOP_FILE} does not yet exist..."
    echo "Checking again in 2 seconds..."
    sleep 2
  done

3 echo "File was found! exiting..."
```

At 1, we define a variable representing the name of the file for which the `while` loop 2 checks using a file test operator. The loop won't exit until the condition is satisfied. Once the file is available, the loop will stop, and the script will continue to the `echo` command 3. Save this file as *while_loop.sh* and run it:

```
$ chmod u+x while_loop.sh
$./while_loop.sh

The file stoploop does not yet exist...
Checking again in 2 seconds...
--snip--
```

While the script is running, open a second terminal in the same directory as the script and create the *stoploop* file:

```
$ touch stoploop
```

Once you've done so, you should see the script breaking out of the loop and print the following:

```
File was found! exiting...
```

This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/while_loop.sh*.

until

Although `while` runs so long as the condition succeeds, `until` runs so long as it fails. Listing 2-11 shows the `until` loop syntax.

```
until some_condition; do
```

`

```
    # run some commands until the condition is no longer false
done
```

Listing 2-11 The `until` loop syntax

Listing 2-12 uses `until` to run some commands until a file's size is greater than zero (meaning it is not empty).

```
#!/bin/bash
FILE="output.txt"

touch "${FILE}"
until [[ -s "${FILE}" ]]; do
  echo "$FILE is empty..."
  echo "Checking again in 2 seconds..."
  sleep 2
done

echo "${FILE} appears to have some content in it!"
```

Listing 2-12 An `until` loop to check a file's size

We first create an empty file, then begin a loop that runs until the file is no longer empty. Within the loop, we print some messages to the terminal. Save this file as *until_loop.sh* and run it:

```
$ chmod u+x until_loop.sh
$ ./until_loop.sh

output.txt is empty...
Checking again in 2 seconds...
--snip--
```

At this point, the script has created the file *output.txt*, but it's an empty file. We can check this using the `du` command:

```
$ du -sb output.txt
output.txt
```

Open another terminal and navigate to the location at which your script is saved, then append some content to the file so its size is no longer zero:

```
$ echo "until_loop_will_now_stop!" > output.txt
```

The script should exit the loop, and you should see it print the following:

```
output.txt appears to have some content in it!
```

This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/until_loop.sh*.

for

The `for` loop iterates over a *sequence*, such as a list of filenames, variables, or even group of values generated by running some command. Inside the `for`

`

loop, we define a block of commands that are run against each value in the list, and each value in the list is assigned to a variable name we define.

Listing 2-13 shows the syntax of a `for` loop.

```
for variable_name in LIST; do
  # run some commands for each item in the sequence
done
```

Listing 2-13 The `for` loop syntax

A simple way to use a `for` loop is to execute the same command a number of times. For example, the following code prints the numbers 1 through 10:

```
#!/bin/bash

for index in $(seq 1 10); do
  echo "${index}"
done
```

Save and run this script. You should see the following output:

```
1
2
3
4
5
6
7
8
9
10
```

A more practical example might use a `for` loop to run commands against a bunch of IP addresses passed on the command line. The script in Listing 2-14 retrieves all arguments passed to the script, then iterates through them and prints a message for each:

```
#!/bin/bash

for ip_address in "$@"; do
  echo "Taking some action on IP address ${ip_address}"
done
```

Listing 2-14 A `for` loop to iterate through command line arguments

Save this script as *for_loop_arguments.sh* and run it as follows:

```
$ chmod u+x for_loop_arguments.sh
./for_loop_arguments.sh 10.0.0.1 10.0.0.2 192.168.1.1 192.168.1.2

Taking some action on IP address 10.0.0.1
Taking some action on IP address 10.0.0.2
--snip--
```

`

We can even run a `for` loop on the output of commands such as `ls`. In Listing 2-15, we print the names of all files in the current working directory:

```
#!/bin/bash

for file in $(ls .); do
  echo "File: ${file}"
done
```

Listing 2-15 A `for` loop to iterate through a list of files in the current directory

We use a `for` loop to iterate over the output of the `ls .` command, which lists the files in the current directory. Each file will be assigned to the `file` variable as part of the `for` loop, so we can then use `echo` to print its name. This technique would be useful if we wanted to, for example, perform a file upload of all files in the directory or even rename them in bulk.

The break and continue statements

Loops can run forever or until a condition is met. But you can also exit a loop at any point using the `break` keyword. This keyword provides an alternative to the `exit` command, which would cause the entire script, and not just the loop, to exit. Using `break`, we can leave the loop and advance to the next code block:

```
#!/bin/bash

while true; do
  echo "in the loop"
  break
done

echo "This code block will be reached"
```

In this case, the last `echo` command will be executed.

The `continue` statement is used to jump to the next iteration of a loop. You can use it to skip a certain value in a sequence. To illustrate this, let's create three empty files so we can iterate through them:

```
$ touch example_file1 example_file2 example_file3
```

Next, our `for` loop will write some content to each file, excluding the first one, *example_file1*, which it will leave empty (Listing 2-16).

```
#!/bin/bash

1 for file in example_file*; do
    if [[ "${file}" == "example_file1" ]]; then
      echo "Skipping the first file."
    2 continue
    fi

    echo "${RANDOM}" > "${file}"
  done
```

`

Listing 2-16 The use of `continue` in a `for` loop

We start a `for` loop using the `example_file*` glob, which will expand to match the names of all files starting with *example_file* in the directory where the script runs 1. As a result, the loop should iterate over all three files we created earlier. Within the loop, we use a file test operator to check whether the filename is equal to *example_file1* because we want to skip this file and not make any changes to it. If the condition is met, we use the `continue` statement 2 to proceed to the next iteration, leaving the file unmodified. Later in the loop, we use the `echo` command with the environment variable `${RANDOM}` to generate a random number and write it into the file.

Save this script as *for_loop_continue.sh* and execute it in the same directory as the three files.

```
$ chmod u+x for_loop_continue.sh
$./for_loop_continue.sh

Skipping the first file.
```

If you examine the files, you should see that the first file is empty while the other two contain a random number as a result of the script echoing the value of the `${RANDOM}` environment variable into them.

Case Statements

In bash, `case` statements allow you to test multiple conditions in a cleaner way by using more readable syntax. Often, they help you avoid many `if` conditions, which can become harder to read as they grow in size.

Listing 2-17 shows the `case` statement syntax.

```
case EXPRESSION in
  PATTERN1)
    # do something if the first condition is met
  ;;
  PATTERN2)
    # do something if the second condition is met
  ;;
esac
```

Listing 2-17 The `case` statement syntax

Case statements start with the keyword `case` followed by some expression, such as a variable you want to match a pattern against. `PATTERN1` and `PATTERN2` in this example represent some pattern case (such as a regular expression, a string, or an integer) that you want to compare to the expression. To close a case statement, you use the keyword `esac` (*case* inverted).

Let's take a look at an example `case` statement that checks whether an IP address is present in a specific private network (Listing 2-17).

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

`

```
IP_ADDRESS="${1}"

case ${IP_ADDRESS} in
  192.168.*)
    echo "Network is 192.168.x.x"
  ;;
  10.0.*)
    echo "Network is 10.0.x.x"
  ;;
  *)
    echo "Could not identify the network."
  ;;
esac
```

Listing 2-18 A `case` statement to check an IP address and determine its network

We define a variable that expects one command line argument to be passed (`${1}`) and saves it to the `IP_ADDRESS` variable. We then use a pattern to check whether the `IP_ADDRESS` variable starts with `192.168.` and a second pattern to checks whether it starts with `10.0.`.

We also define a default wildcard pattern using `*`, which returns a default message to the user if nothing else has matched.

Save this file as *case_ip_address_check.sh* and run it:

```
$ chmod u+x case_ip_address_check.sh
$./case_ip_address_check.sh 192.168.12.55
Network is 192.168.x.x

$./case_ip_address_check.sh 212.199.2.2
Could not identify the network.
```

This script is available at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/case_ip_address_check.sh*.

Text Processing and Parsing

One of the most common things you'll find yourself doing in bash is processing text. You can parse text on the command line by running one-off commands, or use a script to store parsed data in a variable that you can act on in some way. In both cases, the skill is important to many scenarios.

To test the commands in this section on your own, download the sample log file from *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/log.txt*. This file is space-separated, and each segment represents a specific data type, such as the client's source IP address, timestamp, HTTP method, HTTP path, HTTP User Agent field, HTTP status code, and more.

`

Filtering with grep

The `grep` command is one of the most popular Linux commands out there today. We use `grep` to filter out information of interest from streams. At its most basic form, you can use it like so (Listing 2-19).

```
$ grep "35.237.4.214" log.txt
```

Listing 2-19 Filtering for a specific string from a file

This `grep` command will read the file and extract any lines containing the IP address 35.237.4.214 from it.

We can even `grep` for multiple patterns simultaneously. The following backslash pipe (`\|`) acts as an *or* condition (Listing 2-20).

```
$ grep "35.237.4.214\|13.66.139.0" log.txt
```

Listing 2-20 Filtering for two specific strings

Alternatively, you could use multiple `grep` patterns with the `-e` argument to accomplish the same thing (Listing 2-21).

```
$ grep -e "35.237.4.214" -e "13.66.139.0" log.txt
```

Listing 2-21 Filtering for two specific strings with grep `-e`

As you learned in Chapter 1, we can use the pipe (`|`) command to provide one command's output as the input to another. In the following example, we run the `ps` command and use `grep` to filter out a specific line. The `ps` command lists the processes on the system:

```
$ ps | grep TTY
```

By default, `grep` is case sensitive. We can make our search case insensitive using the `-i` flag (Listing 2-22).

```
$ ps | grep -i tty
```

Listing 2-22 A case-insensitive search with `grep`

We can also use `grep` to exclude lines containing a certain pattern using the `-v` argument, like in Listing 2-23.

```
$ grep -v "35.237.4.214" log.txt
```

Listing 2-23 Excluding lines containing a string

To print only the matched pattern, and not the entire line at which the matched pattern was found, use `-o` (Listing 2-24).

```
$ grep -o "35.237.4.214" log.txt
```

Listing 2-24 Printing only the matching pattern

`

The command also supports regular expressions, anchoring, grouping, and much more. Use the `man grep` command to read more about its capabilities.

Filtering with awk

The `awk` command is a data processing and extraction Swiss-army knife. You can use it to identify and return specific fields from a file. To see how it works, take another close look at our log file. What if we needed to print just the IP addresses from this file? This is easy to do with `awk` (Listing 2-25).

```
$ awk '{print $1}' log.txt
```

Listing 2-25 Printing the first field

The `$1` represents the first field of every line in the file, where the IP addresses are. By default, `awk` treats spaces or tabs as separators or delimiters.

Using the same syntax, we can print additional fields, such as the timestamps. Listing 2-26 filters the first three fields of every line in the file.

```
$ awk '{print $1,$2,$3}' log.txt
```

Listing 2-26 Printing the first three fields

Using similar syntax, we can print the first and last field simultaneously. In this case, `NF` represents the last field (Listing 2-27).

```
$ awk '{print $1,$NF}' log.txt
```

Listing 2-27 Printing the first and last field

We can also change the default delimiter. For example, if we had a CSV file separated by commas, rather than spaces or tabs, we could pass `awk` the `-F` flag to specify the type of delimiter, as in Listing 2-28.

```
$ awk -F',' '{print $1}' example_csv.txt
```

Listing 2-28 Printing the first field using a comma delimiter

We can even use `awk` to print the first 10 lines of some file. This emulates the behavior of the `head` Linux command. `NR` represents the total number of records and is built into `awk` (Listing 2-29).

```
$ awk 'NR < 10' log.txt
```

Listing 2-29 Printing the first 10 lines of a file

You'll often find it useful to combine `grep` and `awk`. For example, you might want to first find the lines in a file containing the IP address 42.236.10.117 and then print the HTTP paths this IP made a request to (Listing 2-30).

```
$ grep "42.236.10.117" log.txt | awk '{print $7}'
```

Listing 2-30 Filtering an IP address and printing the seventh field, representing HTTP paths

`

The `awk` command is a super powerful tool, and we encourage you to dig deeper into its capabilities by running `man awk` for more information.

Editing Streams with sed

The `sed` (stream editor) command takes actions on text. For example, it can replace the text in a file, modify the text in some command's output, and even delete selected lines from files.

Let's use `sed` to replace any mentions of the word *Mozilla* with the word *Godzilla* in the *log.txt* file. We use its `s` (substitution) command and `g` (global) command to make the substitution across the whole file, rather than to just the first occurrence (Listing 2-31).

```
$ sed 's/Mozilla/Godzilla/g' log.txt
```

Listing 2-31 Replacing a string with another string

This will output the modified version of the file but won't change the original version. You can redirect the output to a new file to save your changes:

```
$ sed 's/Mozilla/Godzilla/g' log.txt > newlog.txt
```

We could also use `sed` to remove any whitespace from the file with the `/ //` syntax, which will replace whitespace with nothing, removing it from the output altogether (Listing 2-32).

```
$ sed 's/ //g' log.txt
```

Listing 2-32 Removing whitespace with `sed`

If you need to delete lines of a file, use the `d` command. In Listing 2-33, `1d` deletes (`d`) the first line (`1`).

```
$ sed '1d' log.txt
```

Listing 2-33 Deleting the first line with `sed`

To delete the last line of a file, use the dollar sign (`$`), which represents the last line, along with `d` (Listing 2-34).

```
$ sed '$d' log.txt
```

Listing 2-34 Deleting the last line with `sed`

You can also delete multiple lines, such as line 5 and 7 (Listing 2-35).

```
$ sed '5,7d' log.txt
```

Listing 2-35 Deleting lines 5 and 7

Lastly, you can print specific line ranges, such as lines 2 through 15 (Listing 2-36).

```
$ sed -n '2,15 p' log.txt
```

`

Listing 2-36 Printing line ranges in a file

When you pass `sed` the `-i` argument, it will make the changes to the file itself rather than create a modified copy (Listing 2-37).

```
sed -i '1d' log.txt
```

Listing 2-37 Making changes to the original file

This rich utility can do a whole lot more. Use the `man sed` command to find additional ways to use `sed`.

Job Control

As you become proficient in bash, you'll start to build complex scripts that take an hour to complete or must run continuously. Not all scripts need to execute in the foreground, blocking execution of other things. Instead, you may want to run certain scripts as background jobs, either because they take a while to complete or because their runtime output isn't interesting and you care only about the end result.

Commands that you run in a terminal occupy that terminal until the command is finished. These commands are considered *foreground jobs*. In Chapter 1, we used the ampersand character (`&`) to send a command to the background. This command then becomes a *background job* that allows us to unblock the execution of other commands.

Managing Background and Foreground Jobs

To practice working with foreground and background jobs, let's run a command directly in the terminal and send it to the background:

```
$ sleep 100 &
```

Notice that we can continue working on the terminal while this `sleep` command runs for 100 seconds. You can verify the spawned process is running by using the `ps` command:

```
$ ps -ef | grep sleep

user    1827    1752 cons0    19:02:29 /usr/bin/sleep
```

Now that this job is in the background, we can use the `jobs` command to see what jobs are currently running (Listing 2-38).

```
$ jobs

[1]+  Running                 sleep 100 &
```

Listing 2-38 Listing jobs

The output shows that the `sleep` command is in `Running` state and that its job ID is `1`.

`

We can migrate the job from the background to the foreground by issuing the `fg` command and the job ID (Listing 2-39).

```
$ fg %1

sleep 100
```

Listing 2-39 Sending a job to the foreground

At this point, the `sleep` command is occupying the terminal, since it's running in the foreground. You can press CTRL+Z to suspend the process, which will produce the following output in the `jobs` table:

```
[1]+  Stopped                 sleep 100
```

To send this job to the background again in a running state, use the `bg` command with the job ID (Listing 2-40).

```
$ bg %1

[1]+ sleep 100 &
```

Listing 2-40 Sending a job to the background

Keeping a Job Running After Logout

Whether you send a job to the background or are running a job in the foreground, the process won't survive if you close the terminal or log out. If you close the terminal, the process will receive a `SIGHUP` signal and terminate.

What if we wanted to keep running a script in the background even after we've logged out of the terminal window or closed it? To do so, we could start a script or command with the `nohup` (no hangup) command prepended (Listing 2-41).

```
$ nohup ./my_script.sh &
```

Listing 2-41 Keeping a script running after closing the terminal or logging out

The `nohup` command will create a file named *nohup.out* with standard output stream data. So, make sure you delete this file if you don't want to leave any traces on the disk.

There are additional ways to run background scripts, such as by plugging into system and service managers like *systemd*, which provide additional features, such as monitoring that the process is running, restarting it if it isn't, and capturing failures. We encourage you to read more about systemd if you have such use-cases at *https://man7.org/linux/man-pages/man1/init.1.html*.

`

Bash Customizations for Penetration Testers

As penetration testers, we often follow standard workflows for all ethical hacking engagements, whether they be consulting work, bug bounty hunting, or red teaming. We can optimize some of this work with a few bash tips and tricks.

Placing Scripts in Searchable Paths

Bash searches for programs within directories defined by the `PATH` environment variable. Commands such as `ls` are always available to you because system and user binaries are located in directories that are part of the `PATH`.

To see your `PATH`, run the following command:

```
$ echo $PATH

/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

The output might change depending on the operating system you use.

When you write a bash script, place it in a directory such as */usr/local/bin*, which, as you can see, is part of the `PATH`. If you don't do this, you have a few other options available to you:

- Call the script directly using the full path.
- Change the directory to the one in which your script lives and execute it from there.
- Use aliases (shown in the next section).
- Adding additional paths to the `PATH` environment variable

The benefit of placing the script in a searchable path is that you can simply call it by its name. You don't have to provide the full path or have the terminal be in the same directory.

Shortening Commands with Aliases

When you find yourself frequently using a long Linux command, you can make use of an *alias* to map it to a shorter custom name that will save you time when you need to run it.

For example, imagine that you often use Nmap with special parameters to scan for all 65,535 ports on a given IP address:

```
nmap -vv -T4 -p- -sV --max-retries 5 localhost
```

This command is quite hard to remember. With aliases, we can make it more accessible on the command line or to our scripts. Here, we assign the command to the alias `quickmap`:

```
$ alias quicknmap="nmap -vv -T4 -p- -sV --max-retries 5 localhost"
```

Now we can run the aliased command using the name of the alias:

```
$ quicknmap
Starting Nmap 7.80 ( https://nmap.org ) at 2023-02-21 22:32 EST
--snip--
PORT    STATE SERVICE
631/tcp open  ipp
```

You can even assign an alias to your own scripts:

```
$ alias helloworld="bash ~/scripts/helloworld.sh"
```

Aliases aren't permanent, but they can be. In the next section, you'll learn how to use bash profiles to make permanent changes to your shell.

Customizing the Bash Run Commands Profile (bashrc)

We can use the *~/.bashrc* file to load functions, variables, and just about any other custom bash code we desire into a new bash session. For example, we can create variables containing information we'll frequently need to access, such as the IP address of a vulnerable host we're testing.

For example, we could append the following to the end of the *~/.bashrc* file. These lines define a few custom variables and save our aliased Nmap command:

```
VULN_HOST=1.0.0.22
VULN_ROUTER=10.0.0.254

alias quicknmap="nmap -vv -T4 -p- -sV --max-retries 5 example.local"
```

The next time you open a terminal, you'll be able to access these values. Make these new values available immediately by re-importing the *~/.bashrc* file using the `source` command:

```
$ source ~/.bashrc

$ echo $VULN_HOST
10.0.0.22

$ echo $VULN_ROUTER
10.0.0.254
```

Now you can use these variables even after you close the terminal and start a new session.

Importing Custom Scripts

Another way to introduce changes to your bash session is to create a dedicated script that contains pentesting-related customizations and have the *~/.bashrc* file import it using the `source` command. To achieve this, create a *~/.pentest.sh* file containing your new logic, and then make a one-time modification to *~/.bashrc* to import it at the end of the file:

```
source ~/.pentest.sh
```

`

Note that you can also source a bash file using the `.` command, as in Listing 42.

```
. ~/.pentest.sh
```

Listing 2-42 An alternative to the `source` command

Capturing Terminal Session Activity

Penetration testing often involves having dozens of terminals open simultaneously, all running many tools that can produce a lot of output. When we find something of interest, we may need some of that output as evidence for later. To avoid losing track of an important piece of information, we can make use of some clever bash.

The `script` command allows us to capture terminal session activity. One way to use it is to load a small bash script that uses `script` to save every session to a file for later inspection. The script might look like this:

```
#!/bin/bash

FILENAME="$(date +%m-%d-%y)_${RANDOM}.log"

if [[ ! -d ~/sessions ]]; then
  mkdir ~/sessions
fi

# Starting a script session
script -f -a "~/sessions/${FILENAME}"
```

Having *~/.bashrc* load this script, as showed earlier, will result in the creation of the *~/sessions* directory containing each terminal session capture in a separate file. The recording stops once you enter `exit` in the terminal or close the entire terminal window.

Exercise 2: Pinging a Domain

In this exercise, we'll write a bash script that accepts two arguments: a name (for example, *mysite*) and a target domain (for example, *nostarch.com*). The script should also be able to do the following:

1. Throw an error if the arguments are missing and exit using the right status code.
2. Ping the domain and return an indication of whether the ping was successful or not.
3. Write the results to a CSV file containing the following information:
    a. The name provided to the script
    b. The target domain provided to the script

`

c. The ping result (either success or failure)

d. The current date and time

Like most tasks in Bash, there are multiple ways to achieve this goal. At *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch02/exercise_solution.sh*, you can find an example solution to this exercise.

Summary

In this chapter, you learned how to perform flow control using conditions, loops, and functions, how to control scripts using jobs, and how to search and parse text. We also highlighted tips and tricks for building more effective and penetration testing workflows using bash features.

3

SETTING UP A HACKING LAB

In this chapter, you'll set up a lab environment containing hacking tools and an intentionally vulnerable target. You'll use this lab in chapter exercises, but you can also turn to it whenever you need to write, stage, and test a bash script before running it against real targets.

The locally deployed target and its assets mimic the production environment of a mock IT hosting company called ACME Infinity Servers, which has its own fake employees, customers, and data. This fabricated internet hosting company and its customers will provide you with a diverse range of intentionally vulnerable applications, user accounts, and infrastructure that you can practice attacking in future chapters.

The lab will be fully contained in a Kali virtual machine. This virtual machine will require the following minimum specifications: at least 4GB of RAM, at least 40GB of storage, and an internet connection.

Security Lab Precautions

Follow these guidelines to reduce the risks associated with building and operating a hacking lab:

Avoid connecting the lab directly to the internet. Hacking lab environments typically run vulnerable code or outdated software. While these vulnerabilities are great for hands-on learning, they could pose risks to your network, computer, and data if they become accessible from the internet. Instead, we recommend working through the book when connected to local networks that you trust or while operating offline.

Deploy the lab in a virtual environment using a hypervisor. Separating the lab environment from your primary operating system is generally a good idea, as it prevents conflicts that could potentially break other software on your computer. We recommend using a virtualization tool to ensure this separation. In the next section, you'll install the lab in a Kali virtual machine.

Take frequent snapshots of your virtual machine. Snapshots are backups of your virtual machine that allow you to restore it to a previous state. Lab environments often won't stay stable after we attack them, so take snapshots whenever your lab is in a stable state.

With these best practices in mind, let's get our hands dirty, and our lab up and running!

Installing Kali

Kali is a Linux distribution created for penetration testing. Based on Debian, it was designed by OffSec. We'll use Kali as our lab's operating system because it comes bundled with some of the libraries, dependencies, and tools we'll need.

Your Kali machine will play two roles in the lab environment: First, it will act as the host responsible for running the target networks and machines against which we'll run our scripts, and secondly, it will serve as the hacking machine from which you'll perform your attacks.

You can find an x64 version of the Kali virtual machine images for the VMware Workstation and Oracle VirtualBox hypervisors at *https://www.kali.org/get-kali*. Pick the hypervisor of your choice and follow the official installation instructions at *https://www.kali.org/docs/installation* to install it.

After completing the installation process, you should see the Kali login screen shown in Figure 3-1. Kali ships with a default user account named *kali* whose password is *kali*.

Figure 3-1 The Kali login screen

After logging in to Kali, you need to make sure it's up to date. To access the terminal, open the **Applications** menu, and in the search bar, enter **terminal emulator**. Click the corresponding application.

Let's use a few commands to update your software repositories and upgrade your installed packages. In the terminal window, enter the following commands:

```
$ sudo apt update -y
$ sudo apt upgrade -y
$ sudo apt dist-upgrade -y
```

When you use `sudo`, Kali will ask for your password. This is the same password you used to log in to the virtual machine, *kali*.

Newer Kali releases use the Z Shell (zsh) by default, so let's ensure that bash is the default shell for the *kali* user with this command:

```
$ sudo usermod --shell /bin/bash kali
```

Next, let's enable our new default shell by running the following command:

```
$ su - kali
```

Moving forward, we'll use this Kali machine for all tasks we cover in the book. We recommend keeping the terminal window open, as you'll need it for additional installations very soon.

Setting Up the Target Environment

Now it's time to install the machines and networks that will make up our simulated corporate target. You can perform this installation in two ways: manually or using an automated script.

We encourage you to set up your lab manually at least once by following the instructions in this section. This will allow you to familiarize yourself with the lab's core components and practice running commands on the command line. However, if you ever need to redeploy the lab from scratch in a fresh installation of Kali, you can do so by executing the *init.sh* script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/lab/init.sh*:

```
$ cd ~/Black-Hat-Bash/lab
$ chmod u+x init.sh
$ ./init.sh
```

This script should install all of the lab's dependencies, containers, and hacking utilities, enabling you to skip the instructions in this section and in "Installing Additional Hacking Tools" on page XX. You must execute the script in a Kali virtual machine that meets the system requirements described in the introduction to this chapter.

Installing Docker and Docker Compose

We'll build the lab environment using Docker, a tool for deploying and managing containers. *Containers* package code and its dependencies so an application can run reliably in various environments. We'll also make use of Docker Compose, a special Docker utility used to orchestrate the building and management of multiple Docker containers using a single YAML file, known as a Compose file.

Let's first configure our sources to use Debian's current stable version of Docker's community edition, *docker-ce*, using the following commands. We use `printf` to add Docker's Advanced Package Tool (APT) repository to the APT package-source database file. The `tee` command reads from standard input stream and writes to a file:

```
$ printf '%s\n' "deb https://download.docker.com/linux/debian bullseye stable" |
sudo tee /etc/apt/sources.list.d/docker-ce.list
```

Next, download and import Docker's keyring to ensure that the repository is validated and all packages installed from that repository are cryptographically verified. We use `curl` to download the key and pipe it to the `gpg` command, which will then store it in the required folder:

```
$ curl -fsSL https://download.docker.com/linux/debian/gpg | sudo gpg --dearmor -o
/etc/apt/trusted.gpg.d/docker-ce-archive-keyring.gpg
```

Finally, run another update to refresh the repository database and install the Docker components:

```
$ sudo apt update -y
```

```
$ sudo apt install docker-ce docker-ce-cli containerd.io -y
```

To verify that you have Docker Compose running correctly, use the following:

```
$ docker compose --help
```

Next, make sure the Docker process will automatically start upon system reboot by running this command:

```
$ sudo systemctl enable docker --now
```

Docker requires the use of `sudo`, which can get a little inconvenient. If you want to avoid having to enter `sudo` before executing Docker-related commands, add the *kali* user to the *docker* Linux group:

```
$ sudo usermod -aG docker $USER
```

Once you've done this, you shouldn't need `sudo` to run Docker commands.

Cloning the Book's Repository

You can find the lab's files in the open source Black Hat Bash repository at *https://github.com/dolevf/Black-Hat-Bash*. This repository contains the Docker configurations needed to build the lab, as well as all of the bash scripts mentioned in the future chapters of this book.

Kali comes preloaded with git, which you can use to clone and download the Black Hat Bash repository. To do so, run the following:

```
$ git clone https://github.com/dolevf/Black-Hat-Bash.git
```

Next, move into the repository's root directory and take a quick look at its contents:

```
$ cd Black-Hat-Bash && ls -l

--snip--
drwxr-xr-x 2 kali kali 4096 Jul 22 23:07 ch01
drwxr-xr-x 2 kali kali 4096 Jul 22 23:07 ch02
drwxr-xr-x 2 kali kali 4096 Jul 22 23:07 ch03
drwxr-xr-x 2 kali kali 4096 Jul 22 23:07 ch04
drwxr-xr-x 2 kali kali 4096 Jul 22 23:07 ch05
--snip--
```

As you can see in the output, the repository's contents are organized into directories for each of the book's chapters. The repository also includes a *lab* directory, which is what we'll use to set up the lab in the next section.

Deploying Docker Containers

The contents of the *lab* directory in the Black Hat Bash repository control all networking infrastructure, machines, and applications used within the lab. This directory includes a *run.sh* script file. By running this script without any arguments, you can see that it is used to deploy, tear down, rebuild, clean, and check the status of our environment:

```
$ cd lab
$ ./run.sh

Usage: ./run.sh deploy | teardown | rebuild | cleanup | status

deploy   | build images and start containers
teardown | stop containers
rebuild  | rebuilds the lab from scratch
cleanup  | stop containers and delete containers and images
status   | check the status of the lab
```

Let's start by creating the lab by using the `deploy` argument. Note that when you execute the deployment, you'll be prompted for your *kali* user password:

```
$ ./run.sh deploy
```

The initial deployment of the lab environment will take a few minutes to complete. To monitor the progress of the installation, you'll need to open a new terminal session and tail the log file located within the repository's lab directory, as so:

```
$ tail -f ~/Black-Hat-Bash/lab/log.txt
```

When the `tail -f` (follow) command is used against a file, it provides a live view of any new lines added to the end of the file. This is useful for keeping an eye on log files, which frequently have new information written to them.

NOTE Because the lab setup downloads software such as operating system images and other applications, this deployment could take some time, depending on the network connection you have and the compute resources allocated to the host running the lab.

Testing and Verifying the Containers

Once the log file indicates the process is complete, it should tell you whether the lab was set up correctly. We can also run a few commands to verify this. First, let's execute a status check using the *run.sh* script, this time with the `status` argument. If all the checks pass, you should get the following output:

```
$ ./run.sh status
Lab is up.
```

We can also list all our lab's running Docker containers using the `docker list` command:

```
$ docker ps --format "{{.Names}}"
p-web-01
p-web-02
p-ftp-01
c-jumpbox-01
c-db-01
c-db-02
c-syslog-01
c-backup-01
c-redis-01
```

You should get a similar output, though the containers won't necessarily be in the same order.

The Network Architecture

The lab consists of nine machines running in Docker containers, as well as two networks. Most of the machines are assigned to one of the two networks, and we'll use them to facilitate various hacking scenarios in future chapters.

The networks within the lab are connected to Kali using Docker's bridged networking mode. Figure 3-2 shows the details of this network architecture.

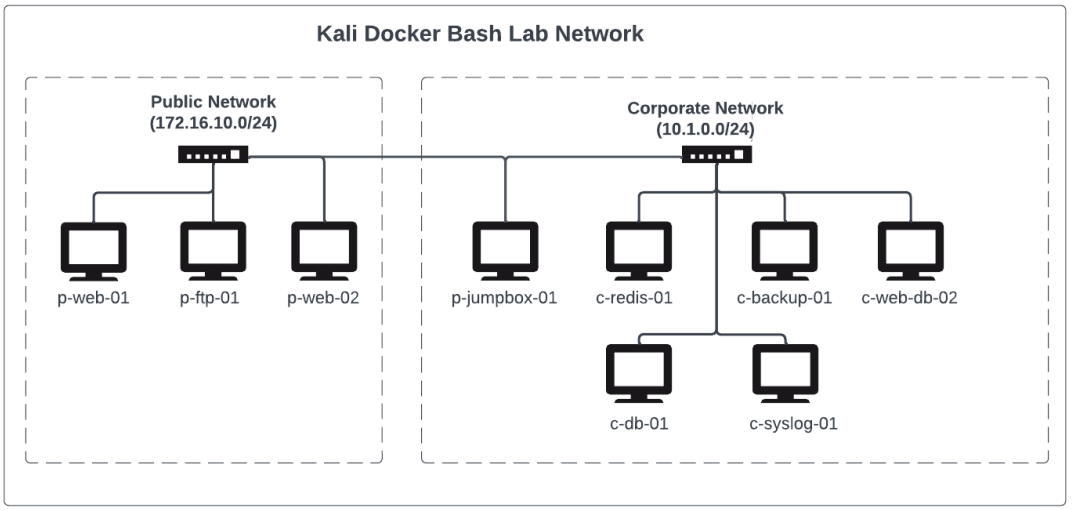

Figure 3-2 The lab's network architecture

You can also find this diagram in the book's repository at *https://github.com/dolevf/Black-Hat-Bash/blob/master/lab/lab-network-diagram.png*.

The Public Network

The network on the left side of Figure 3-2 is the public network, where our fake internet hosting company, ACME Infinity Servers, hosts its customer's websites and resources. The two company websites you'll find in this network belong to ACME Impact Alliance and ACME Hyper Branding.

The public network has an IP address CIDR range of 172.16.10.0/24 and contains four machines (whose names are prefixed with *p-*). It is also public facing, meaning we'll likely test the machines with access to this network before any other, as they constitute possible entry points into the network.

The Corporate Network

The second network is the corporate network. ACME Infinity Servers uses this private network to hosts its supporting infrastructure on the back-end. As you can see, the corporate network has an IP address CIDR range of 10.1.0.0/24 and contains five machines (whose names are prefixed with *c-*). This network is not public facing, meaning the machines in this network don't have internet connectivity, and we won't test them until we're able to take over one or more of the machines on the public network, which will serve as our launchpad to the corporate network.

Kali Network Interfaces

Kali has two network interfaces used to facilitate connections to both lab networks. We can use the *br_public* network interface to access the public network and the *br_corporate* network interface to access the corporate network. You can validate that both interfaces are online and configured to use the correct network address by running the following command:

```
$ ip addr | grep "br_"

--snip--
4: br_public: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:ea:5f:96:9b brd ff:ff:ff:ff:ff:ff
    inet 1 172.16.10.1/24 brd 172.16.10.255 scope global br_public
       valid_lft forever preferred_lft forever
5: br_corporate: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:67:90:5a:95 brd ff:ff:ff:ff:ff:ff
    inet 2 10.1.0.1/24 brd 10.1.0.255 scope global br_corporate
       valid_lft forever preferred_lft forever
```

Verify that the IP addresses match those shown at 1 and 2 before moving on.

The Machines

The nine machines that make up the lab environment follow a simple naming convention. The first character of the name determines what network the machine belongs to. For example, if the machine name starts with a *p*, it belongs to the public network; likewise, if it starts with a *c*, it belongs to the corporate network. The next word describes the machines' functions or main technology stack, such as *web*, *ftp*, *jumpbox*, or *redis*. Finally, a number is used to distinguish similar machines, such as *p-web-01* and *p-web-02*.

Each machines provides us with unique applications, services, and user accounts that we can learn about and break into. Later chapters will describe these machines in more detail, but Table 3-1 provides some high-level information about them.

Table 3-1 Lab Machine Details

Name	Public IP	Corporate IP	Hostname

Kali host	172.16.10.1	10.1.0.1	
p-web-01	172.16.10.10	-	*p-web-01.acme-infinity-servers.com*
p-ftp-01	172.16.10.11	-	*p-ftp-01.acme-infinity-servers.com*
p-web-02	172.16.10.12	10.1.0.11	*p-web-02.acme-infinity-servers.com*
c-jumpbox-01	172.16.10.13	10.1.0.12	*c-jumpbox-01.acme-infinity-servers.com*
c-backup-01	-	10.1.0.13	*c-backup-01.acme-infinity-servers.com*
c-redis-01	-	10.1.0.14	*c-redis-01.acme-infinity-servers.com*
c-db-01	-	10.1.0.15	*c-db-01.acme-infinity-servers.com*
c-db-02	-	10.1.0.16	*c-db-02.acme-infinity-servers.com*
c-syslog-01	-	10.1.0.17	*c-syslog-01.acme-infinity-servers.com*

When you perform penetration tests from Kali, keep in mind that you may sometimes see Kali's own IP addresses, 172.16.10.1 and 10.1.0.1, pop up in certain tool results. We won't be testing those.

Managing the Lab

Now that you've set up your lab and taken a close look at its components, you'll learn how to tear it down, start it, and rebuild it if needed.

Shutting Down

When you're not using the lab environment, it's a good practice to turn it off. To shut down all the containers running in the lab, run the following:

```
$ ./run.sh teardown
```

You should receive a list of all stopped containers, as well as the removed networks and volumes, as shown here:

```
==== Shutdown Started ====
Stopping p-web-02     ... done
Stopping c-jumpbox-01 ... done
--snip--
Removing volume lab_p_web_02_vol
OK: lab has shut down.
```

To restart your containers, simple re-run the `deploy` command mentioned in "Deploying Docker Containers" on page XX.

Removal

To completely remove the lab environment from your Kali machine, you can run the `cleanup` command. This will destroy all containers and their images:

```
$ ./run.sh cleanup

==== Cleanup Started ====
```

```
Cleaning up the Black Hat Bash environment, this may take a few moments...
OK: lab environment has been destroyed.
```

Listing 3-1 Lab cleanup command output

After running the command, you should receive a confirmation that the lab environment has been destroyed.

Rebuilding

When we execute a rebuild, the lab will first shut down all running containers, delete volumes, and remove all container images before running a new deployment. To execute the rebuild, run the following command:

```
$ ./run.sh rebuild
```

If you rebuild the lab, you'll lose any data you saved inside of your containers. Rebuilding is useful when something goes wrong during installation. Maybe, halfway through it, you lost your network connection, and it reported a failed state. The `rebuild` command allows you to wipe and install it from scratch.

Accessing Individual Lab Machines

As you progress through the book, you'll compromise the machines in the lab environment. However, it often takes multiple attempts to obtain full access to a machine. Sometimes, you may need to troubleshoot an issue or reproduce a post-compromise activity, and you won't want to repeat the steps you performed to obtain access.

To gain shell access to any individual lab machine, you can run the following Docker command:

```
$ docker exec -it MACHINE-NAME bash
```

MACHINE-NAME represents the name of a lab machine, such as *p-web-01* or *p-jumpstation-01* (or any other machine from Table 3-1 that starts with *p-* or *c-*). The Docker command will drop you into a bash shell, at which point you can execute any command you like. To exit, simply enter `exit` in the prompt or close the terminal session's window.

We highly recommend you compromise the machines as intended before taking these convenient shortcuts, however.

Installing Additional Hacking Tools

Most of the tools we'll use in this book will come pre-installed in Kali, and we'll introduce them upon first use. However, we'll need several tools that aren't installed by default, so let's install them here. First, create a new directory in which to place your tools:

```
$ cd ~
```

```
$ mkdir tools
```

Wappalyzer

Wappalyzer is a website reconnaissance tool used to identify the technology stack of any website or web application. It can detect the frameworks. platforms, and libraries used by our targets, which will come in handy for us in future chapters when we attempt to discover vulnerabilities in old versions of web application components.

Before you can install Wappalyzer, you need to download its dependencies, Node.js, the *Node Package Manager (NPM)*, and Yarn. Let's start with Nodejs and NPM:

```
$ curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash -
$ sudo apt update
$ sudo apt install nodejs npm -y
```

To verify that Node.js and NPM are properly installed, run the following two commands to get their versions:

```
$ node -v
$ npm -v
```

Next, let's install Yarn and verify that it is installed properly by checking its version:

```
$ sudo npm install --global yarn
$ yarn -v
```

The `--global` flag makes the package available for other applications on the computer to import and use.

Finally, install Wappalyzer from its GitHub repository:

```
$ cd ~/tools
$ git clone https://github.com/wappalyzer/wappalyzer.git
$ cd wappalyzer
$ yarn install
$ yarn run link
```

To verify that it properly installed, try the `help` command:

```
$ node src/drivers/npm/cli.js -h
```

You'll notice that this command is not very intuitive to run, as no part of it indicates that it is related to Wappalyzer. In "Assigning Aliases to Hacking Tools" on page XX, we'll set an alias so we can run the tool by using the command `wappalyzer`.

RustScan

RustScan is a lightning-fast port scanner written in the Rust programming language by Autumn (Bee) Skerritt (@bee_sec_san). Some claim that RustScan can scan all 65,000 ports on a target in seconds!

To install it, you must first install *cargo*, the Rust package manager:

```
$ sudo apt install cargo -y
```

Next, clone the RustScan repository to your *tools* directory:

```
$ cd ~/tools
$ git clone https://github.com/RustScan/RustScan.git
```

Then move into the RustScan directory and build the tool using cargo:

```
$ cd RustScan
$ cargo build --release
```

Once RustScan has been built, run a quick test to ensure that it's working properly. The RustScan binary is located in the *target/release* directory. Try executing a simple `help` command:

```
$ cd ~/tools/RustScan/target/release
$ ./rustscan --help

Fast Port Scanner built in Rust. WARNING Do not use this program against
sensitive infrastructure since the specified
server may not be able to handle this many socket connections at once. - Discord
<http://discord.skerritt.blog> -
GitHub https://github.com/RustScan/RustScan
--snip--
```

Nuclei

Nuclei is a vulnerability scanner written in the Go programming language by ProjectDiscovery, a company that builds open source hacking tools (some of which are extremely popular these days). Nuclei works by sending requests to targets defined by a YAML template file. The hacking community has published thousands of Nuclei templates supporting several protocols, including TCP, DNS, HTTP, Raw Sockets, File, Headless and more. You can find these templates at *https://github.com/projectdiscovery/nuclei-templates*

Install Nuclei by running the following installation command:

```
$ sudo apt install nuclei -y
```

To verify that Nuclei is correctly installed, run a help command:

```
$ nuclei -h

Nuclei is a fast, template based vulnerability scanner focusing
on extensive configurability, massive extensibility and ease of use.

Usage:
  nuclei [flags]

Flags:
TARGET:
   -u, -target string[]       target URLs/hosts to scan
```

The first time you run Nuclei, it will automatically create a *nuclei-templates* directory in the user's home folder and download all of the publicly available nuclei templates.

Dirsearch

Dirsearch is a multi-threaded web directory path-enumeration tool used to find common paths on web servers. Dirsearch is available in Kali's software repositories, so to install it, you can simply run the following command:

```
$ sudo apt install dirsearch -y
```

To verify that Dirsearch is correctly installed, run a help command:

```
$ dirsearch --help
```

Linux_Exploit_Suggester 2

The *Linux Exploit Suggester 2* is a next-generation tool based on the original Linux Exploit Suggester. Written in Perl and developed by Jonathan Donas, it includes several exploits you can use to potentially compromise a local Linux kernel.

To install it, first clone the repository to your *tools* directory:

```
$ cd ~/tools
$ git clone https://github.com/jondonas/linux-exploit-suggester-2.git
```

To verify Linux Exploit Suggester 2 is installed correctly, run a help command:

```
$ cd linux-exploit-suggester-2
$ perl linux-exploit-suggester-2.pl -h
```

Gitjacker

Gitjacker is a data-extraction tool that targets web applications whose *.git* directory has been mistakenly uploaded. Before you can install Gitjacker, you'll first need to install *jq*, a command line JSON processor:

```
$ sudo apt install jq -y
```

Next, get the Gitjacker install script and run it:

PROD: PLEASE BREAK AT A NATURAL PLACE

```
$ curl -s
"https://raw.githubusercontent.com/liamg/gitjacker/master/scripts/install.sh" |
bash
```

Finally, verify that Gitjacker is working properly by running the following help command:

```
$ ~/bin/gitjacker -h
```

LinEnum

LinEnum is a bash script written by Owen Shearing for enumerating local information on a Linux host. We can grab the script from its GitHub repository using `wget`:

```
$ cd ~/tools
$ wget https://raw.githubusercontent.com/rebootuser/LinEnum/master/LinEnum.sh
```

To verify that the script is working correctly, make it executable and run the following help command:

```
$ chmod u+x LinEnum.sh
$ ./LinEnum.sh -h

#############################################################
# Local Linux Enumeration & Privilege Escalation Script #
#############################################################
# www.rebootuser.com | @rebootuser

# Example: ./LinEnum.sh -k keyword -r report -e /tmp/ -t

OPTIONS:
-k      Enter keyword
-e      Enter export location
-s      Supply user password for sudo checks (INSECURE)
-t      Include thorough (lengthy) tests
-r      Enter report name
-h      Displays this help text


Running with no options = limited scans/no output file
#############################################################
```

unix-privesc-check

The *unix-privesc-check* shell script, written by pentestmonkey, collects information from a host in an attempt to find misconfigurations and ways to escalate privileges. The script is written to support many flavors of Linux and UNIX systems and does not require any dependencies, which makes it convenient to both install and run.

By default, the script comes bundled with Kali, and you should find it in */usr/bin/unix-privesc-check*:

```
# which unix-privesc-check

/usr/bin/unix-privesc-check
```

Optionally, you can create a copy of it in the *tools* directory for easier access, should you need to copy it later to any of the lab's machines:

```
$ cp /usr/bin/unix-privesc-check ~/tools
```

If it's not available in your Kali machine, you can also download it directly from APT:

```
$ apt-get install unix-privesc-check -y
```

Verify that you can run it successfully with the following command:

```
$ unix-privesc-check -h

unix-privesc-check ( http://pentestmonkey.net/tools/unix-privesc-check )

Usage: unix-privesc-check { standard | detailed }

"standard" mode: Speed-optimised check of lots of security settings.
--snip--
```

Assigning Aliases to Hacking Tools

Tools that are installed through third-party repositories such as GitHub sometimes won't have setup files that make running them easier. We can assign these tools bash aliases as shorthand references so that we won't need to enter the full directory path every time we run them.

We can assign custom aliases using the following commands. These commands will be written to our *~/.bashrc* file, which will execute when we open a new terminal session:

```
$ echo "alias wappalyzer='node /home/kali/tools/wappalyzer/src/drivers/npm/cli.js'" >> ~/.bashrc
$ echo "alias rustscan='/home/kali/tools/RustScan/target/release/rustscan'" >> ~/.bashrc
$ echo "alias gitjacker='/home/kali/bin/gitjacker'" >> ~/.bashrc
```

Wappalyzer, RustScan, and Gitjacker now have aliases.

NOTE At this point, you should have a fully functioning bash hacking lab. Now would be a good time to take a snapshot of your Kali virtual machine so you can restore it to this clean state. It is a good idea to take snapshots regularly, especially whenever you make significant configuration changes or deploy new tools to your virtual lab.

Summary

In this chapter, you built your hacking lab, which consists of a dedicated Kali virtual machine running several intentionally vulnerable Docker containers and hacking utilities. We also discussed managing your lab environment by tearing it down, cleaning it up, and rebuilding it.

We'll use this lab in all hands-on exercises moving forward. If you encounter problems, we encourage you to keep an eye on the book's GitHub repository (*https://github.com/dolevf/Black-Hat-Bash*), where we maintain the source code responsible for keeping your lab up to date. In the next chapter, you'll make use of these tools to perform reconnaissance and gather information about remote targets.

`

4

RECONNAISSANCE

Every hacking engagement starts with some form of information gathering. In this chapter, we'll perform reconnaissance on targets by writing bash scripts to run various hacking tools. You'll learn how to use bash to automate tasks and chain multiple tools into a single workflow.

In the process, you'll develop an important bash-scripting skill: parsing the output of various tools to extract only the information we need. Your scripts will interact with tools that figure out what hosts are online, what ports are open on those hosts, and what services they are running, then deliver this information to you in the format you require.

`

Perform all hacking activities in your Kali environment against the vulnerable network you set up in Chapter 3.

Creating Reusable Target Lists

The *scope* is the list of systems or resources you're allowed to target. In penetration testing or bug-hunting engagements, the target company might provide you with various types of scopes:

- Individual IP addresses, such as 172.16.10.1 and 172.16.10.2
- Networks, such as 172.16.10.0/24 and 172.16.10.1-172.16.10.254
- Individual domain names, such as *lab.example.com*
- A parent domain name and all of its subdomains, such as **.example.com*

When working with tools such as port and vulnerability scanners, you'll often need to run the same type of scan against all hosts in your scope. This can be hard to do efficiently, however, as each tool uses its own syntax. For instance, one tool might allow you to specify an input file containing a list of targets, while other tools may require you to run the tool against individual addresses.

When working with tools that don't let you provide a wide range of targets, you can use bash to automate this process. In this section, we'll use bash to create IP- and DNS-based target lists that you could feed to scanners.

Generating a List of Consecutive IP Addresses

Imagine that you need to create a file containing a list of IP addresses from 172.16.10.1 to 172.16.10.254. While you could write all 254 addresses by hand, this would be time-consuming. Let's use bash to automate the job! We'll consider three strategies: using the `seq` command in a `for` loop, using brace expansion with `echo`, and using brace expansion with `printf`.

The seq and for Loop Approach

In the `for` loop shown in Listing 4-1, we use the `seq` command to iterate through numbers ranging from 1 to 254 and assign each

`

number to the `ip` variable. After each iteration, we use `echo` to write the IP address to a dedicated file on disk, *172-16-10-hosts.txt*.

```
#!/bin/bash

# generate IP addresses from a given range
for ip in $(seq 1 254); do
  echo "172.16.10.${ip}" >> 172-16-10-hosts.txt
done
```

Listing 4-1 Using a `for` loop to create a list of IP addresses with the `seq` command

You can run this code directly from the command line or save it in a script and then run it. The generated file should look like the following:

```
$ cat 172-16-10-hosts.txt

172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.4
172.16.10.5
--snip--
```

Before moving on, let's consider two other ways to accomplish the same task.

The echo and Brace Expansion Approach

As in most cases, you can achieve the same task in bash using multiple programming approaches. We can generate the IP address list using a simple `echo` command, without running any loops. In Listing 4-2, we use `echo` with brace expansion to generate the strings.

```
$ echo 10.1.0.{1..254}

10.1.0.1 10.1.0.2 10.1.0.3 10.1.0.4 --snip--
```

Listing 4-2 Performing brace expansion with `echo`

You'll notice that this command outputs a list of IP addresses on a single line, separated by spaces. This isn't ideal, as what we really want is each IP address on a separate line. In Listing 4-3, we use `sed` to replace spaces with new line characters (`\n`).

```
$ echo 10.1.0.{1..254} | sed 's/ /\n/g'

10.1.0.1
10.1.0.2
```

```
10.1.0.3
--snip--
```

Listing 4-3 Generating a list of IP addresses with `echo` and `sed`

The printf and Brace Expansion Approach

Alternatively, you can use the `printf` command to generate the same list. Using `printf` won't require piping to `sed`, producing a cleaner output (Listing 4-4).

```
$ printf "10.1.0.%d\n" {1..254}
```

Listing 4-4 Generating a list of IP addresses with `printf`

The `%d` is an integer placeholder, and it will be swapped with the numbers defined in the brace expansion to produce a list of IP addresses from 10.1.0.1 to 10.1.0.254. Now using this list is just a matter of redirecting the output to a new file and using it as an input file.

Compiling a List of Possible Subdomains

Say you're performing a penetration test against a company with the parent domain *example.com*. In this engagement, you're not restricted to any specific IP address or domain name, which means that any asset you find on this parent domain during the information-gathering stage is considered in scope.

Companies tend to host their services and application on dedicated subdomains. These subdomains can be anything, but more often than not, companies use names that make sense to humans and are easy to enter into a web browser. For example, you might find the helpdesk portal at *helpdesk.example.com*, a monitoring system at *monitoring.example.com*, the continuous integration system at *jenkins.example.com*, the email server at *mail.example.com*, and the file transfer server at *ftp.example.com*.

How can we generate a list of possible subdomains for our target? Bash makes this very easy. First, we'll need a list of common subdomains. You can find such a list built into Kali at */usr/share/wordlists/amass/subdomains-top1mil-110000.txt* or */usr/share/wordlists/amass/bitquark_subdomains_top100K.txt*. To look for wordlists on the internet, you could use the following

`

Google search query to search for files on GitHub provided by community members: `subdomain wordlist site:gist.github.com`. This will search GitHub for code snippets (also called *gists*) containing the word *subdomain wordlist*.

For the purposes of this example, we'll use the subdomain list at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch04/subdomains-1000.txt*. Download it and save it in your home directory. The file contains one subdomain per line without an associated parent domain. You'll have to join each subdomain with the target's parent domain to form a *fully qualified domain name (FQDN)*. As in the previous section, we'll show two strategies for accomplishing this task: using a `while` loop and using `sed`.

The while Loop Approach

The script in Listing 4-5 accepts a parent domain and a word list from the user, then prints a list of fully qualified subdomains using the word list we downloaded earlier.

```
#!/bin/bash

DOMAIN="${1}"
FILE="${2}"

# Read the file from standard input and echo the full domain
while read -r subdomain; do
  echo "${subdomain}.${DOMAIN}"
done < "${FILE}"
```

Listing 4-5 Generating a list of subdomains using a `while` loop

The script uses a `while` loop to read the file and assign each line to the `subdomain` variable in turn. The `echo` command then concatenates these two strings together to form a full domain name. Save this script as *generate_subdomains.sh* and provide it with two arguments:

```
$ ./generate_subdomains.sh example.com subdomains-1000.txt

www.example.com
mail.example.com
ftp.example.com
localhost.example.com
webmail.example.com
--snip--
```

`

The first argument is the parent domain and the second is the path to the file containing all possible subdomains.

The sed Approach

We can use `sed` to write content to the end of each line in a file. In Listing 4-6, the command uses the `$` sign to find the end of a line, then replace it with the target domain prefixed with a dot (*.example.com*) to complete the domain name.

```
$ sed 's/$/.example.com/g' subdomains-1000.txt

relay.example.com
files.example.com
newsletter.example.com
```

Listing 4-6 Generating a list of subdomains using `sed`

The `s` at the beginning of the argument to `sed` stands for *substitute*, and `g` means that `sed` will replace all matches in the file, not just the first match. So, in simple words, we substitute the end of each line in the file with *.example.com*. If you save this code to a script, the output should look the same as in the previous example.

Host Discovery

When testing a range of addresses, one of the first things you'll likely want to do is find out information about them. Do they have any open ports? What services are behind those ports, and are they vulnerable to any security flaws? It's possible to answer these questions manually, but this can be challenging if you need to do it for against hundreds or thousands of hosts. Let's use bash to automate network enumeration tasks.

One way to identify live hosts is by attempting to send them network packets and wait for them to return responses. In this section, we'll use bash and additional network utilities to perform host discovery.

ping

At its most basic form, the `ping` command takes one argument: a target IP address or domain name. Run the following command to see its output:

```
$ ping 172.16.10.10
```

`

```
PING 172.16.10.10 (172.16.10.10) 56(84) bytes of data.
64 bytes from 172.16.10.10: icmp_seq=1 ttl=64 time=0.024 ms
64 bytes from 172.16.10.10: icmp_seq=2 ttl=64 time=0.029 ms
64 bytes from 172.16.10.10: icmp_seq=3 ttl=64 time=0.029 ms
```

The `ping` command will run forever, so press CTRL+C to stop its execution.

If you read `ping`'s manual page (by running `man ping`), you'll notice that there is no way to run it against multiple hosts at once. But using bash, we can do this quite easily. The script in Listing 4-7 pings all hosts on the network 172.16.10.0/24.

```
#!/bin/bash
FILE="${1}"

1 while read -r host; do
2   if ping -c 1 -W 1 -w 1 "${host}" &> /dev/null; then
      echo "${host} is up."
    fi
3 done < "${FILE}"
```

Listing 4-7 Pinging multiple hosts using a `while` loop

At 1, we run a `while` loop that reads from the file passed to the script on the command line. This file is assigned to the variable `FILE`. We read each line from the file and assign it to the `host` variable. We then run the `ping` command using the `-c` argument with a value of `1` at 2, which tells the `ping` command to send a ping request only once and exit. By default on Linux, the `ping` command sends ping requests indefinitely until you stop it manually by sending a SIGHUP signal (CTRL+C).

We also use the arguments `-W 1` (to set a timeout in seconds) and `-w 1` (to set a deadline in seconds) to limit how long `ping` will wait to receive a response. This is important because we don't want `ping` to get stuck on an unresponsive IP address; we want it to continue reading from the file until all 254 hosts are tested.

Lastly, we use the standard input stream to read the file and "feed" the while loop with its content 3.

Save this code to a file named *multi_host_ping.sh* and run it while passing the hosts file. You should see that it picks up a few live hosts:

```
$ ./multi_host_ping.sh 172-16-10-hosts.txt
```

`

```
172.16.10.1 is up.
172.16.10.10 is up.
172.16.10.11 is up.
172.16.10.12 is up.
172.16.10.13 is up.
```

The caveat to this host-discovery approach is that certain hosts, especially hardened ones, might not reply to `ping` commands at all. So, if we solely rely on this method for discovery, we might miss out on live hosts on the network.

Also note that commands that run forever by default, such as `ping`, could pose a challenge when integrated into a bash script. In this example, we've explicitly set a few special flags to ensure that our bash script won't hang when it executes `ping`. This is why it's important to first test commands in the terminal before integrating them into your scripts. More often than not, tools have special options to ensure they don't execute forever, such as timeout options.

For tools that don't provide a timeout option, the `timeout` command allows you to run commands and exit after a certain amount of time has passed. You can prepend the `timeout` command to any Linux utility, passing it some interval (in the *seconds, minutes, hours* format). After the time has elapsed, the entire command exits. For example: `timeout 5s ping 8.8.8.8`.

Nmap

Nmap has a special option called `-sn` that performs a ping sweep (and disables port scanning). A ping sweep is a simple technique for finding live hosts on a network by sending them a `ping` command and waiting for a positive response (*ping response*). Since many operating systems will respond to `ping` by default, this technique has proved valuable. The ping sweep in Nmap will essentially make Nmap send Internet Control Message Protocol (ICMP) packets over the network to discover running hosts:

```
$ nmap -sn 172.16.10.0/24

Nmap scan report for 172.16.10.1
Host is up (0.00093s latency).
Nmap scan report for 172.16.10.10
Host is up (0.00020s latency).
Nmap scan report for 172.16.10.11
Host is up (0.00076s latency).
```

```
--snip--
```

There is a lot of text in this output. With a bit of bash magic, we can get a cleaner output by extracting only the IP addresses that were identified as being alive by using the `grep` and `awk` commands (Listing 4-8).

```
$ nmap -sn 172.16.10.0/24 | grep "Nmap scan" | awk -F'report for ' '{print $2}'

172.16.10.1
172.16.10.10
--snip--
```

Listing 4-8 Parsing Nmap's ping scan output with `grep` and `awk`

Using Nmap's built-in ping sweep scan may be more useful than manually wrapping the `ping` utility with bash because you don't have to worry about checking for conditions such as whether the command was successful. Moreover, in penetration tests, you may drop an Nmap binary on more than one type of operating system, and the same syntax will work consistently whether the `ping` utility exists or not.

arp-scan

We can perform penetration testing remotely, from a different network, or from within the same network as the target. In this section, we'll highlight the use of `arp-scan` as a way to find hosts on a network when the test is done locally.

The `arp-scan` utility sends Address Resolution Protocol (ARP) packets to hosts on a network and displays any responses it gets back. The *ARP* communication protocol maps *Media Access Control (MAC)* addresses, which are unique 12-digit hexadecimal addresses assigned to network devices, to the IP addresses on a network. ARP is a Layer 2 protocol in the Open Systems Interconnection (OSI) model, meaning it is useful only when you're on a local network and can't be used to perform a remote scan over the internet.

Note that `arp-scan` requires root privileges to run; this is because it uses functions to read and write packets that require elevated privileges. At its most basic form, you can run it by executing the `arp-scan` command and passing a single IP address as an argument:

```
$ sudo arp-scan 172.16.10.10 -I br_public
```

We also need to tell `arp-scan` which network interface to send packets on, as there are a few network interfaces in Kali. To achieve this, we use the `-I` argument. The `br_public` interface corresponds to the 172.16.10.0/24 network in the lab.

To scan entire networks, you can pass `arp-scan` a CIDR range, such as */24*. For example, the following command scans all IP addresses between 172.16.10.1 and 172.16.10.254:

```
$ sudo arp-scan 172.16.10.0/24 -I br_public
```

Lastly, you can make use of the hosts file you created in "Generating a List of Consecutive IP addresses" on page XX as input to arp-scan:

```
$ sudo arp-scan -f 172-16-10-hosts.txt -I br_public
```

The output generated by `arp-scan` should look like the following:

```
172.16.10.10    02:42:ac:10:0a:0a      (Unknown: locally administered)
172.16.10.11    02:42:ac:10:0a:0b      (Unknown: locally administered)
172.16.10.12    02:42:ac:10:0a:0c      (Unknown: locally administered)
172.16.10.13    02:42:ac:10:0a:0d      (Unknown: locally administered)
```

It consists of three fields: the IP address, the MAC address, and vendor details, identified by the first three octets of the MAC address. In this scan, the tool identified four hosts on the network that responded to ARP packets.

Exercise 3: Receiving Notifications When New Hosts Are Detected

Imagine that you want to be notified whenever a new host appears on the network. For example, maybe you want to know when new laptops or IT assets have connected. This could be useful if you're testing a target in a different time zone, where device users might not be online when you are.

We can use bash to send ourselves an email whenever our script discovers new assets. Listing 4-9 runs a continuous scan to identify new online hosts, adds these to the *172-16-10-hosts.txt* file we created in "Generating a List of Consecutive IP Addresses" on page XX, and notifies us of the discovery. We'll walk through this code, then discuss ways in which you can improve it.

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

`

```
# sends a notification upon new host discovery
KNOWN_HOSTS="172-16-10-hosts.txt"
NETWORK="172.16.10.0/24"
INTERFACE="br_public"
FROM_ADDR="kali@blackhatbash.com"
TO_ADDR="security@blackhatbash.com"

1 while true; do
    echo "Performing an ARP scan against ${NETWORK}..."

  2 sudo arp-scan -x -I ${INTERFACE} ${NETWORK} | while read -r line; do
    3 host="$(echo "${line}" | awk '{print $1}')"
    4 if ! grep -q "${host}" "${KNOWN_HOSTS}"; then
        echo "Found a new host: ${host}!"
      5 echo "${host}" >> "${KNOWN_HOSTS}"
      6 sendemail -f "${FROM_ADDR}" \
          -t "${TO_ADDR}" \
          -u "ARP Scan Notification" \
          -m "A new host was found: ${host}"
      fi
    done

    sleep 10
  done
```

Listing 4-9 Receiving notifications about new `arp-scan` discoveries using `sendmail`

A lot is going on here! First, we set a few variables. We assign the file containing the hosts to look for, *172-16-10-hosts.txt*, to the `KNOWN_HOSTS` variable, and the target network 172.16.10.0/24 to the `NETWORK` variable. We also set the `FROM_ADDR` and `TO_ADDR` variables, which we'll use to send the notification email.

We then run an infinite loop using `while` 1. This loop won't end unless we intentionally break out of it. Within the loop, we run `arp-scan` using the options `-x` to display a plain output (so it's easier to parse) and `-I` to define the network interface `br_public` 2. In the same line, we use a `while read` loop to iterate through the output of `arp-scan`. We use `awk` to parse each IP address in the output and assign it to the `host` variable 3.

At 4, we use an `if` condition to check whether the `host` variable (which represents a host discovered by `arp-scan`) exists in our hosts file. If it does, we don't do anything, but if it doesn't, we write it to the file 5 and send an email notification 6 using the `sendemail` command. Notice that each line in the `sendemail` command ends with a backslash (`\`). When lines are long, bash

allows us to separate them in this way while still treating them as a single command. Line breaks of long code lines make it easier to read. At the end of this process, we use `sleep 10` to wait 10 seconds before running this discovery again.

If you run this script, you should receive an email whenever a new host is discovered. To properly send emails, you'll need to configure a mail transfer agent (MTA) such as Postfix on the system. Refer to the documentation at *https://postfix.org/documentation.html* for more information.

You can download the script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch04/host_monitor_notification.sh*.

Note that the continuous network probing performed by this script isn't very stealthy. To achieve this in a more covert way, try modifying the script in one of the following ways:

- Slow down the probing so it triggers every few hours or arbitrary number of minutes. You can even randomize this interval to make it less predictable.
- Instead of sending notifications over the network, try writing the results to memory if you're running the script from within a compromised network.
- Upload the results to an innocent-looking third-party website. The Living off Trust Sites (LOTS Project) at *https://lots-project.com* maintains an inventory of legitimate websites that corporate networks often allow. Attackers commonly use these to carry out activities such as data exfiltration so that their traffic blends with other legitimate traffic, making it harder for analysts to spot.

Now that we know what hosts are available on the 172.16.10.0/24 network, we recommend removing any unresponsive IP addresses from the *172-16-10-hosts.txt* file to make your future scans faster.

To go further, we encourage you to experiment with other notification delivery methods, such as sending notifications over Slack, Discord, Microsoft Teams, or any other messaging system you use on daily basis. Platforms such as Slack, for example, use the concept of a *webhook*, where a script can make a POST request to a special URL to deliver a custom message to some channel of choice.

`

Port Scanning

Once you've discovered hosts on the network, you can run a port scanner to find their open ports and the services they're running. Let's explore two port scanning tools: Nmap and RustScan.

Scanning Targets with Nmap

Nmap allow us to perform port scanning against single targets or multiple targets at once. In the following example, we use Nmap to perform a port scan of the domain *scanme.nmap.org*.

```
$ nmap scanme.nmap.org
```

Nmap also accepts IP addresses, like so:

```
$ nmap 172.16.10.1
```

When there are no special options provided on the command line to Nmap, it will use the following default settings:

Performs a SYN Scan.
Nmap will use a SYN scan to discover open ports on a target. Also called a half-open scan, a *SYN scan* involves sending a SYN packet and waiting for a response. Nmap won't complete the full TCP handshake (meaning *ACK* won't be sent back), which is why we call this scan half open.

Scans the Top 1000 Ports.
Nmap will scan only popular ports known to be frequently in use, such as TCP ports 21, 22, 80 and 443. It won't scan the entire port range of 0-65,534, to conserve resources.

Scans TCP Ports.
Nmap will scan only TCP ports, not UDP ports.

Nmap allows you to scan multiple targets by passing them on the command line. In the following example, we scan both *localhost* and *scanme.nmap.org* (Listing 4-10).

```
$ nmap localhost scanme.nmap.org
```

Listing 4-10 Passing multiple addresses to Nmap

Nmap can also read targets from a given file using its `-iL` option. The targets must be separated by new lines. Let's use our *172-16-10-hosts.txt* file with Nmap to scan multiple targets.

```
$ nmap -sV -iL 172-16-10-hosts.txt
```

`

```
--snip--
Nmap scan report for 172.16.10.1
Host is up (0.00028s latency).
PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 9.0p1 Debian 1+b2 (protocol 2.0)
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
--snip--

Nmap scan report for 172.16.10.10
Host is up (0.00029s latency).
PORT     STATE SERVICE          VERSION
8081/tcp open  blackice-icecap?
--snip--
```

This scan may take some time to complete due to the use of the `-sV` option, which detects the version of services on each port. As you can see, Nmap returns a few IP addresses and their open ports, including their services and even information related to the operating system running on the host. If we wanted to filter, say, only the open ports, we could do by using `grep`:

```
$ nmap -sV -iL 172-16-10-host.txt | grep open

22/tcp open  ssh
8081/tcp open  blackice-icecap
21/tcp open  ftp
80/tcp open  http
80/tcp open  http
22/tcp open  ssh
--snip--
```

Nmap was able to identify services on several open TCP ports, such as the File Transfer Protocol (FTP) on port 21, Secure Shell (SSH) on port 22, and HyperText Transfer Protocol (HTTP) on port 80. Later in this chapter, we'll take a closer look at each of these services.

Nmap also allows you to pass the `--open` flag on the command line to will show only the ports that were found open:

```
$ nmap -sV -iL 172-16-10-host.txt --open
```

NOTE Kali's own interface IP (172.16.10.1) will be captured in this port scan, since it is part of the hosts file. You can use Nmap's `--exclude` option to exclude this specific IP when performing a network-wide scan: `--exclude 172.16.10.1`. You can also remove it manually from the file for convenience.

`

Use `man nmap` to find out more about Nmap's scanning and filtering capabilities.

Performing Rapid Scans with RustScan

RustScan is becoming more popular in the bug bounty and penetration testing spaces because of its speed and extendibility. The following command runs a port scan using the `rustscan` command. The `-a` (address) argument accepts a single address or an address range:

```
$ rustscan -a 172.16.10.0/24

Open 172.16.10.11:21
Open 172.16.10.1:22
Open 172.16.10.13:22
--snip--
```

RustScan's output is fairly easy to parse with bash. Lines starting with *Open* indicate that an open port was found on a specific IP address. These are followed by the IP address and port separated by a colon.

When you run RustScan, you may notice that the initial output contains banners, author credits, and additional information not directly related to the scan results. Use the `-g` (greppable) option to show only the scanning information. The following command uses the greppable output mode to scan 172.16.10.0/24 on the first 1024 ports (also called *privileged ports*) with the `-r` (range) option:

```
$ rustscan -g -a 172.16.10.0/24 -r 0-1024

172.16.10.11 -> [80]
172.16.10.12 -> [80]
```

Now the output is more `grep` friendly. If we wanted to parse it, all we'd need to do is pass the delimiter `->`, which separates the IP address and port, with `awk`:

```
$ rustscan -g -a 172.16.10.0/24 -r 0-1024 | awk -F'->' '{print $1,$2}'
```

This command outputs two fields, the IP address and the port. If we wanted to get rid of the `[]` surrounding the port number, we can do this with the `tr` command and the `-d` (delete) argument followed by the characters to delete:

```
$ rustscan -g -a 172.16.10.0/24 -r 0-1024 | awk -F'->' '{print $1,$2}' | tr -d '[]'
```

This should return a cleaner output.

`

NOTE Remember that running port scanners in aggressive modes increases the chances of getting caught, especially if the target implements an *Intrusion Detection System* (*IDS*) or *Endpoint Detection and Response* (*EDR*) system. Also, if you scan at a rapid pace, some devices could crash as a result of the network flood.

Exercise 4: Organizing Scan Results by Port Number

It's often useful to sort your scan results into categories of interest. For example, you could dump results for each IP address in a dedicated file or organize the results based on the versions of software found. In this exercise, we'll organize our scan results based on port numbers. Let's write a script that does the following:

1. Runs Nmap against hosts in a file.
2. Uses bash to create individual files whose filenames are open ports.
3. In each file, writes the IP address on which the corresponding port was open.

At the end of this exercise, we'll have a bunch of files, such as *port-22.txt*, *port-80.txt*, and *port-8080.txt*, and in each file, we'll see one or more IP addresses at which that port was found to be open. This can be useful when you have a large number of target hosts and want to attack them in "clusters" by targeting specific protocols associated with given ports. Listing 4-11 shows the script's code.

```
#!/bin/bash
HOSTS_FILE="172-16-10-hosts.txt"
1 NMAP_RESULT=$(nmap -iL ${HOSTS_FILE} --open | grep "Nmap scan report\|tcp open")

# read the nmap output line by line
while read -r line; do
2 if echo "${line}" | grep -q "report for"; then
    ip=$(echo "${line}" | awk -F'for ' '{print $2}')
  else
  3 port=$(echo "${line}" | grep open | awk -F'/' '{print $1}')
  4 file="port-${port}.txt"
  5 echo "${ip}" >> "${file}"
  fi
done <<< "${NMAP_RESULT}"
```

Listing 4-11 Organizing scan results by port using bash

`

We assign the output of the `nmap` command to the variable `NMAP_RESULTS` 1. In this command, we also filter for specific lines containing the words `Nmap scan report` or `tcp open`. These lines are part of Nmap's standard port scan output and they indicate that open ports were found on an IP address.

We use a `while` loop to read `NMAP_RESULT` line by line, checking whether each line contains the string `report for` 2. This line will hold the IP address where ports were found open. If such a line exists, we assign it to the `ip` variable. Then we parse the line to extract the port that was found open 3. At 4, we create the `file` variable to hold the file we'll create on disk with the naming scheme *port-NUMBER.txt*. Lastly, we append the IP address to the file 5.

You can download the script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch04/nmap_to_portfiles.sh*. Save it to a file named *nmap_to_portfiles.sh* and run it. Next, run `ls -l` to see what files were created, and use `cat` to view their contents:

```
$ ls -l

total 24
-rw-r--r-- 1 kali kali 3448 Mar  6 22:18 172-16-10-hosts.txt
-rw-r--r-- 1 kali kali   13 Mar  8 22:34 port-21.txt
-rw-r--r-- 1 kali kali   25 Mar  8 22:34 port-22.txt
--snip--

$ cat port-21.txt

172.16.10.11
```

As you've seen, Nmap's standard output format is a little challenging to parse, but not impossible. It's useful to know that Nmap provides additional output format options we can use to parse it more easily, especially for scripting purposes. One of these options is the `-oG` flag, or the greppable output format. This option is `grep` and `awk` friendly, as you can see in Listing 4-12.

```
$ nmap -iL 172-16-10-hosts.txt --open -oG -

Host: 172.16.10.1 ()    Status: Up
Host: 172.16.10.1 ()    Ports: 22/open/tcp//ssh///    Ignored State: closed (999)
Host: 172.16.10.10 ()   Status: Up
Host: 172.16.10.10 ()   Ports: 8081/open/tcp//blackice-icecap///     Ignored
State: closed (999)
--snip--
```

Listing 4-12 Nmap's greppable output

`

The output now prints the IP address and its open ports on the same line. Nmap has additional format output options such as the `-oX` (XML) output, try to put together a one liner bash script that extracts open ports from an XML output. Open ports in an XML output of Nmap look like the following:

```
$ nmap -iL 172-160-10-hosts.txt --open -oX -

--snip--
<port protocol="tcp" portid="22"><state state="open" reason="syn-ack" reason_ttl="0"/><service
name="ssh" method="table" conf="3"/></port>
--snip--
```

Exercise 5: Detecting a New Open Port on a Given Host

What if we wanted to monitor a host until it opened a certain port? You may find this useful if you're testing an environment in which hosts come up and down frequently. We can do this quite easily with a `while` loop. In Listing 4-13, we continuously check whether a port is open, waiting five seconds between each execution. Once we find an open port, we pass this information to Nmap to perform a service discovery and write the output to a file.

```
#!/bin/bash
RUST_SCAN_BIN="/home/kali/tools/RustScan/target/release/rustscan"
LOG_FILE="watchdog.log"
IP_ADDRESS="$1"
WATCHED_PORT="$2"

service_discovery(){
  local host
  local port
  host="${1}"
  port="${2}"

  nmap -sV -p "${port}" "${host}" >> "${LOG_FILE}" 1
}

2 while true; do
  3 port_scan=$("${RUST_SCAN_BIN}" -a "${IP_ADDRESS}" -g -p "${WATCHED_PORT}")
  4 if [[ -n "${port_scan}" ]]; then
      echo "${IP_ADDRESS} has started responding on port ${WATCHED_PORT}!"
      echo "Performing a service discovery..."
    5 if service_discovery "${IP_ADDRESS}" "${WATCHED_PORT}"; then
        echo "Wrote port scan data to ${LOG_FILE}"
        break
      fi
    else
      echo "Port not yet open or was closed, sleeping for 5 seconds..."
    6 sleep 5
```

`

```
    fi
done
```

Listing 4-13 A watchdog script for newly open ports

At 2 we start an infinite `while` loop. The loop runs the RustScan binary (which is assigned to the variable `RUST_SCAN_BIN`), passing it the `-a` (address) argument containing an IP address we receive on the command line 3. We also pass RustScan the `-g` (greppable) option to produce a format that is `grep` friendly, and the port option (`-p`) to scan a particular port, which we also receive on the command line.

We check the result of the scan 4. If the result is not empty, we pass the IP address and port to the `service_discovery` function 5, which does an Nmap service-version discovery scan (`-sV`) and writes the result to the log file *watchdog.log* 1. If the port scan fails, which means the port is closed, we sleep for five seconds 6. As a result, the process will repeat every five seconds until the port is found open.

You can download this script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch04/port_watchdog.sh*. Save and run it using the following arguments:

```
$ ./port_watchdog.sh 127.0.0.1 3337
```

Since nothing should be running on this port of your localhost, the script should run forever. We can simulate a port-opening event by using Python's built-in *http.server* module, which starts a simple HTTP server:

```
$ python3 -m http.server 3337
```

Now the *port_watchdog.sh* script should show the following:

```
Port is not yet open, sleeping for 5 seconds...
127.0.0.1 has started responding on port 3337!
Performing a service discovery...
Wrote port scan data to watchdog.log
```

You can view the results of the scan by opening the *watchdog.log* file:

```
$ cat watchdog.log
Starting Nmap ( https://nmap.org )
Nmap scan report for 172.16.10.10
Host is up (0.000099s latency).

PORT     STATE SERVICE          VERSION
```

```
3337/tcp open  SimpleHTTPServer
--snip--
```

Using this script, you should be able to identify four IP addresses on the network with open ports: 172.16.10.10 (*p-web-01*) running 8081/TCP, 172.16.10.11 (*p-ftp-01*) running both 21/TCP and 80/TCP, 172.16.10.12 (*p-web-02*) running 80/TCP, and 172.16.10.13 (*p-jumpbox-01*) running 22/TCP.

Banner Grabbing

Learning about the software running on a remote server is a crucial step in a penetration test. In the remainder of this chapter, we'll take a look at how to identify what's behind a port and a service. For example, what web server is running on port 8081, and what technologies does it use to serve content to clients?

Banner grabbing is the process of extracting the information published by remote network services when a connection is established between two parties. Services often transmit these banners to "greet" clients, which can use the information they provide in various ways, such as to ensure they're connecting to the right target. Banners could also include a system admin message of the day (MOTD) or the service's specific running version.

Passive banner grabbing involves looking up banner information using third-party websites. For example, websites such as Shodan (*https://shodan.io*), ZoomEye (*https://zoomeye.org*), and Censys (*https://censys.io*) perform internet-wide scans to map the internet, grabbing banners, versions, website pages, and ports, then create an inventory using this data. We can use such websites to look up banner information without ever interacting with the target server ourselves.

Active banner grabbing is the opposite; it involves establishing a connection to a server and interacting with it directly to receive its banner information. The following network services tend to advertise themselves using banners: web servers, SSH servers, FTP servers, telnet servers, network printers, Internet of Things (IoT) devices, and message queues.

Keep in mind that banners are generally free-form text fields, and they can be changed to mislead clients. For example, an Apache web server could present itself as another type of web server, such as

Nginx. Some organizations even create *honeypot servers* to lure threat actors (or penetration testers). Honeypots make use of deception technologies to masquerade as vulnerable servers, but their real purpose is to detect and analyze attacker activity. More often than not, however, banners transmit default settings that system administrators haven't bothered to change.

Performing Active Banner Grabbing with Netcat

To demonstrate what active banner grabbing looks like, we'll use the following Netcat command to connect to port 21 (FTP) running on the IP address 172.16.10.11 (*p-ftp-01*):

```
$ nc 172.16.10.11 -v 21

172.16.10.11: inverse host lookup failed: Unknown host
(UNKNOWN) [172.16.10.11] 21 (ftp) open
220 (vsFTPd 3.0.5)
```

As you can see, 172.16.10.11 is running the FTP server vsFTPd version 3.0.5. This information may change if the vsFTPd version gets upgraded or downgraded, or if the system administrator decides to disable banner advertisement completely in the FTP server's configuration.

Netcat is a good example of a tool that doesn't natively support probing multiple IP addresses. So, knowing a bit of bash scripting can really help us out. Listing 4-14 will use Netcat to grab banners on port 21 from multiple hosts saved in a file.

```
#!/bin/bash
FILE="${1}"
PORT="${2}"

1 if [[ "$#" -ne 2 ]]; then
    echo "Usage: ${0} <file> <port>"
    exit 1
  fi

2 if [[ ! -f "${FILE}" ]]; then
    echo "File: ${FILE} was not found."
    exit 1
  fi

3 if [[ ! "${PORT}" =~ ^[0-9]+$ ]]; then
    echo "${PORT} must be a number."
    exit 1
  fi

4 while read -r ip; do
```

`

```
    echo "Running netcat on ${ip}:${PORT}"
    result=$(echo -e "\n" | nc -v "${ip}" -w 1 "${PORT}" 2> /dev/null)
5   if [[ -n "${result}" ]]; then
      echo "===================="
      echo "+ IP Address: ${ip}"
      echo "+ Banner: ${result}"
      echo "===================="
    fi
  done < "${FILE}"
```

Listing 4-14 Banner grabbing using Netcat

This script accepts two parameters on the command line: `FILE` and `PORT`. We use an `if` condition to check whether two arguments were indeed passed on the command line 1; if not, we exit with a status code of 1 (fail) and print a usage message indicating how to run the script. We then use another `if` condition to check whether the file provided by the user actually exists on disk using the `-f` test 2.

At 3, we check that the port provided by the user is a number. Anything other than a number will fail. Then we read the host file line by line and run the `nc` (netcat) command on the given port for each 4. Another `if` condition to check whether the command result is not empty 5, meaning a port was found open, and prints the IP address and data that returned from the server.

You can download the script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch04/netcat_banner_grab.sh*.

Detecting HTTP Responses with cURL

You'll often find the popular cURL HTTP client on production systems. When we need to perform banner grabbing on HTTP responses, we could use cURL to send an HTTP request using the HEAD method. The HEAD method allows us to read response headers without fetching the entire response payload from the web server.

Web servers often advertise themselves by setting the `Server` HTTP response header to their name. Sometimes, you may also encounter the running version advertised there. The following command sends an HTTP HEAD request using cURL to the IP address 172.16.10.10:8081 (p-web-01):

```
$ curl -head 172.16.10.10:8081
```

`

```
HTTP/1.1 200 OK
Server: Werkzeug/2.2.3 Python/3.11.1
--snip-
Content-Length: 7176
Connection: close
```

As you can see, the server returns a bunch of headers in the response, one of which is the `Server` header. This header reveals that the remote server is running a Python-based web framework named Werkzeug version 2.2.3, powered by Python version 3.11.1.

Listing 4-15 incorporates this cURL command into a larger script that prompts the user for information with the bash `read` command, then presents the user with a banner.

```
#!/bin/bash
DEFAULT_PORT="80"

read -r -p "Type a target IP address: " ip_address 1
read -r -p "Type a target port (default: 80): " port 2

if [[ -z "${ip_address}" ]]; then 3
  echo "You must provide an IP address."
  exit 1
fi

if [[ -z "${port}" ]]; then 4
  echo "You did not provide a specific port, defaulting to ${DEFAULT_PORT}"
  port="${DEFAULT_PORT}" 5
fi

echo "Attempting to grab the Server header of ${ip_address}"..."

result=$(curl -s --head "${ip_address}:${port}" | grep Server | awk -F':' '{print $2}') 6

echo "Server header for ${ip_address} on port ${port} is: ${result}"
```

Listing 4-15 Extracting the server response header from web servers

This interactive script asks the user to provide details about the target on the command line. First, we use the `read` command to prompt the user to enter an IP address and assign this value to the `ip_address` variable 1. We then ask the user for the desired port number and save that to the `port` variable 2.

At, 3 we check whether the `ip_address` variable length is zero using the `-z` test and exit if this condition is true. Next, we do the same check on the `port` variable 4. This time, if the user didn't provide a port, we use the default HTTP port, 80 5. At 6, we store the

output to the `result` variable. We use `grep` and `awk` to parse the result of `curl` and extract the `Server` header.

You can download the script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch04/curl_banner_grab.sh*. Run it, and when prompted, provide the IP address 172.16.10.10 (*p-web-01*) and port 8081:

```
$ ./curl_banner_grab

Type a target IP address: 172.16.10.10
Type a target port (default: 80): 8081
Attempting to grab the Server header of 172.16.10.10...
Server header for 172.16.10.10 on port 8081 is: Werkzeug/2.2.3 Python/3.11.1
```

As you can see, the script returned the correct information from the target IP address and port. If we didn't specify a port in the terminal, it would have defaulted to port 80. Note that we could have used Netcat to send HTTP HEAD requests, too, but it's useful to know more than one method to achieve a given task.

Using Nmap Scripts

Nmap is more than just a port scanner; we can transform it into a full-fledged vulnerability assessment tool. The *Nmap Scripting Engine (NSE)* allows penetration testers to write scripts in the Lua language to extend Nmap's capabilities. Nmap comes pre-installed with some Lua scripts, as you can see here:

```
$ ls -l /usr/share/nmap/scripts

-rw-r--r-- 1 root root  3901 Oct  6 10:43 acarsd-info.nse
-rw-r--r-- 1 root root  8749 Oct  6 10:43 address-info.nse
-rw-r--r-- 1 root root  3345 Oct  6 10:43 afp-brute.nse
-rw-r--r-- 1 root root  6463 Oct  6 10:43 afp-ls.nse
-rw-r--r-- 1 root root  3345 Oct  6 10:43 afp-brute.nse
-rw-r--r-- 1 root root  6463 Oct  6 10:43 afp-ls.nse
--snip--
```

The *banner.nse* script in the */usr/share/nmap/scripts* folder allows you to grab the banners from many hosts simultaneously. The following bash command performs a banner grab and service discovery (`-sV`) using this script:

```
$ nmap -sV --script=banner.nse -iL 172-16-10-hosts.txt

Nmap scan report for 172.16.10.12
--snip--
PORT   STATE SERVICE VERSION
80/tcp open  http    Apache httpd 2.4.54 ((Debian))
```

```
|_http-server-header: Apache/2.4.54 (Debian)
--snip--
```

When the banner-grabbing script finds a banner, the output line containing that banner will begin with a special character sequence (`|_`). We can filter for this sequence to extract banner information, like so:

```
$ nmap -sV --script=banner.nse -iL 172-16-10-hosts.txt | grep "|_banner\||http-server-header"
```

You may have noticed that, in the case of 172.16.10.10 (*p-web-01*) port 8081, Nmap responded with the following:

```
PORT     STATE SERVICE          VERSION
8081/tcp open  blackice-icecap?
| fingerprint-strings:
--snip--
```

The `blackice-icecap?` value indicates that Nmap was unable to discover the identity of the service definitively. But if you look closely at the `fingerprint-strings` dump, you'll see some HTTP-related information that reveals the same response headers we found when banner grabbing manually using cURL. Specifically, note the Werkzeug web server banner. With a bit of Googling, you'll find that this server runs on Flask, a Python-based web framework.

Detecting Operating Systems

Nmap can also guess the target server's running operating system by using a technique called *TCP/IP fingerprinting*, which is part of its operating system detection scan. This technique identifies the implementation of the operating system's TCP/IP stack by crafting packets in various ways and analyzing the returned responses. Each operating system, such as Linux, Windows, and macOS, implements the TCP/IP stack slightly differently, and Nmap analyzes these subtle differences to identify the running system. In some cases, Nmap may also be able to identify the running kernel version.

To run an operating system detection scan, use the `-O` flag in Nmap. Note that this scan requires `sudo` privileges:

```
$ sudo nmap -O -iL 172-16-10-hosts.txt

--snip--
21/tcp open  ftp
80/tcp open  http
MAC Address: 02:42:AC:10:0A:0B (Unknown)
```

`

```
Device type: general purpose
Running: Linux 4.X|5.X
OS CPE: cpe:/o:linux:linux_kernel:4 cpe:/o:linux:linux_kernel:5
OS details: Linux 4.15 - 5.6
Network Distance: 1 hop
```

Let's create a bash script that can parse this output and sort it by IP address and operating system (Listing 4-16).

```
#!/bin/bash
HOSTS="$*"

1 if [[ "${EUID}" -ne 0 ]]; then
    echo "The Nmap OS detection scan type (-O) requires root privileges."
    exit 1
  fi

2 if [[ "$#" -eq 0 ]]; then
    echo "You must pass an IP or an IP range"
    exit 1
  fi

  echo "Running an OS Detection Scan against ${HOSTS}..."

3 nmap_scan=$(sudo nmap -O ${HOSTS} -oG -)
4 while read -r line; do
    ip=$(echo "${line}" | awk '{print $2}')
    os=$(echo "${line}" | grep OS | awk -F'OS: ' '{print $2}' | sed 's/Seq.*//g')

  5 if [[ -n "${ip}" ]] && [[ -n "${os}" ]]; then
      echo "IP: ${ip} OS: ${os}"
    fi
  done <<< "${nmap_scan}"
```

Listing 4-16 Parsing an operating system detection scan

Because this scan requires root privileges, we check for the effective user's ID 1. If the user ID isn't equal to zero, we exit because there is no point in continuing if the user isn't using root privileges. We then check whether the user passed target hosts as arguments on the command line 2. At 3, we run the Nmap operating system detection scan against these targets, which we've assigned to the `HOSTS` variable.

We use a `while` loop to iterate through the scan results, parsing each line and assigning the IP address in the output to the `ip` variable. We then parse the line a second time to extract the operating system information from Nmap. We clean the output using `sed` so it shows only the operating system, removing everything after the word `Seq`. Next, we check whether both the `ip` and `os`

`

variables are set [5]. If they are, this means we've parsed the output correctly and can finish the script by printing the IP address and the operating system type.

To understand why we parse the output the way we do using `grep`, `awk` and `sed`, run the following command in a separate terminal:

```
$ sudo nmap -O 172.16.10.0/24 -oG -

--snip--
Host: 172.16.10.10 ()   Ports: 8081/open/tcp//blackice-icecap///   Ignored State: closed (999)   OS:
Linux 4.15 - 5.6   Seq Index: 258   IP ID Seq: All zeros
--snip--
```

As you can see, the output is separated by whitespaces. The IP address is found immediately after the first space, and the operating system type comes after the word `OS:` but before the word `Seq`, which is why we needed to extract the text between these two. You can do this parsing in other ways, too, such as with regular expressions; this is just one of way of achieving the task.

You can download the script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch04/os_detection.sh*. Save and run it using the following command (Listing 4-17).

```
$ ./os_detection.sh 172.16.10.0/24

Running an OS Detection Scan against 172.16.10.0/24...
IP: 172.16.10.10 OS: Linux 4.15 - 5.6
IP: 172.16.10.11 OS: Linux 4.15 - 5.6
IP: 172.16.10.12 OS: Linux 4.15 - 5.6
IP: 172.16.10.13 OS: Linux 4.15 - 5.6
IP: 172.16.10.1 OS: Linux 2.6.32
```

Listing 4-17 An operating system detection script that shows only the IP addresses and the operating system

At this point, we've identified a couple of HTTP servers, an FTP server, and an SSH server. Let's take a closer look at the HTTP servers.

Analyzing Websites with Wappalyzer

Wappalyzer is a technology-detection tool tailored to web applications. It has a rich database of signatures for detecting the software running on the remote target, including web frameworks, web servers, databases, operating systems, content management

`

systems, fonts, programming languages, and user interface frameworks.

Let's use Wappalyzer to see what's running on the web applications in the 172.16.10.0/24 network:

```
$ wappalyzer http://172.16.10.10:8081

"urls":{"http://172.16.10.10:8081/":{"status":200}},"technologies":[{"slug":"python",
"name":"Python","description":"Python is an interpreted and general-purpose programming language.",
"confidence":100,"version":"3.11.1","icon":"Python.png"
--snip--
```

Wappalyzer's output is in the *JavaScript Object Notation (JSON)* format, which is composed of keys and values. To parse it, it's helpful to use a tool like `jq` to traverse the JSON structure and extract the information we need. First, take a look at the prettified version of the output using the following command:

```
$ wappalyzer http://172.16.10.10:8081 | jq
```

Next, you'll notice a few fields of interest, specifically the name, the version and the confidence. The name identifies the technology, such as Debian for an operating system. The version identifies the version of that technology, such as Debian 11.6. Confidence is a percentage between 0 and 100. The higher the confidence, the less likely it is to be a false positive.

Let's extract these three pieces of information with `jq`:

```
$ wappalyzer http://172.16.10.10:8081 | jq '.technologies[] | {name, version, confidence}'

{
  "name": "Python",
  "version": "3.11.1",
  "confidence": 100
}
{
  "name": "Tailwind CSS",
  "version": "2.2.19",
  "confidence": 100
}
{
  "name": "Flask",
  "version": "2.2.3",
  "confidence": 100
}
--snip--
```

The `jq` syntax might seem a little odd at first, so let's dissect it. We place the pattern to extract between two single quotes (`'`). Here,

`

we select the `technologies` array (`[]`) key, which contains a bunch of items, each of which is a technology. Then, for each item, we select the `name`, `version`, and `confidence` key names using the `{key_name}` syntax.

Go ahead and run Wappalyzer against every web server we've identified to see what technologies they run. Despite Wappalyzer's confidence level indication, avoid taking the findings at face value. You should always triple-check that what tools report is true.

Summary

In this chapter, we put bash to use in many different ways. We created dynamic target hosts lists; performed host discovery, port scanning, and banner grabbing using multiple tools; created an automated script to notify us of newly discovered hosts; and parsed various tool results. In the next chapter, we'll run vulnerability scanners and fuzzers against these targets.

`

5

VULNERABILITY SCANNING AND FUZZING

In Chapter 4, we identified hosts on a network and a couple of running services, like HTTP, FTP, and SSH. Each of these protocols has its own set of tests we could perform. In this chapter, we'll use specialized tools on the discovered services to find out as much as we can about them.

In the process, you'll use bash to run security testing tools, parse their output, and write custom scripts to scale security testing across many URLs. You'll fuzz with tools such as ffuf and wfuzz, write custom security checks using the Nuclei templating system, extract personally identifiable information from the output of tools, and write your own quick and dirty vulnerability scanners.

`

Scanning Websites Using Nikto

Nikto is a web scanning tool available on Kali. It performs banner grabbing and runs a few basic HTTP security-header checks to determine if the web server uses those security headers, which mitigate known web vulnerabilities such as *cross-site scripting (XSS)*, a client-side injection vulnerability targeting web browsers, and *UI redressing* (also known as *clickjacking*), a vulnerability that lets attackers hijack user clicks by using decoy layers in a web page. These headers indicate to browsers what to do and not do when loading certain resources and opening URLs to protect the user from falling victim to an attack.

After performing these security checks, Nikto also sends requests to possible endpoints on the server using its built-in wordlist of common paths to discover interesting endpoints that could be useful for penetration testers. Let's use it to perform a basic web assessment of the three web servers we've identified on the IP addresses 172.16.10.10 (*p-web-01*), 172.16.10.11 (*p-ftp-01*), and 172.16.10.12 (*p-web-02*).

We'll run a Nikto scan against the web ports we found to be open on the three target IP addresses. Open a terminal and run the following commands one at a time, so you can dissect the output of each IP address:

```
$ nikto -host 172.16.10.10 -port 8081
$ nikto -host 172.16.10.11 -port 80
$ nikto -host 172.16.10.12 -port 80
```

The output for 172.16.10.10 on port 8081 shouldn't yield much interesting information about discovered endpoints, but it should indicate that the server doesn't seem to be hardened, as it doesn't use security headers:

```
+ Server: Werkzeug/2.2.3 Python/3.11.1
+ The anti-clickjacking X-Frame-Options header is not present.
+ The X-XSS-Protection header is not defined. This header can hint to the user agent to protect against some forms of XSS
+ The X-Content-Type-Options header is not set. This could allow the user agent to render the content of the site in a different fashion to the MIME type
--snip--
+ Allowed HTTP Methods: OPTIONS, GET, HEAD
+ 7891 requests: 0 error(s) and 4 item(s) reported on remote host
```

As you can see, Nikto was able to perform a banner grab of the server, as indicated by the line that starts with the word `Server`. It then listed a few missing security headers. These are useful pieces of information, but not enough to take over a server just yet.

`

The IP address 172.16.10.11 on port 80 should give you a similar result, though it also discovered a nice new endpoint, */backup*, and that directory indexing mode is enabled:

```
+ Server: Apache/2.4.55 (Ubuntu)
--snip--
+ OSVDB-3268: /backup/: Directory indexing found.
+ OSVDB-3092: /backup/: This might be interesting...
```

Directory indexing is a server-side setting that lists files located at certain web paths when an index file exists (such as *index.html* or *index.php*). Directory indexing is interesting to find because it could highlight sensitive files in an application, such as configuration files with connection strings, local database files (such as SQLite files) and other environmental files. Open the browser in Kali to *http://172.16.10.11/backup* to see the content of this endpoint (Figure 5-1).

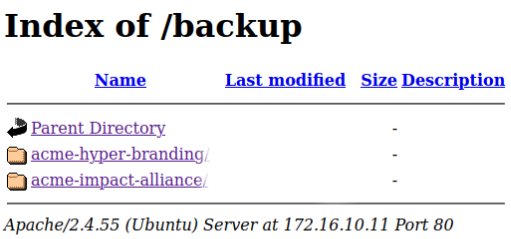

Figure 5-1 Directory indexing found on *172.16.10.11/backup*

Directory indexing lets you browse files in the browser. You can click directories to open them, click files to download them, and so on. On the web page, you should identify two folders: *acme-hyper-branding* and *acme-impact-alliance*. The *acme-hyper-branding* folder appears to contain a file named *app.py*. Download it to Kali so it's available for later inspection by clicking on it.

Building a Directory Indexing Scanner

What if we wanted to run a scan against a list of URLs to check whether directory indexing is enabled on any of them, as well as

`

download all files that they serve? Let's use bash to carry such a task:

```
#!/bin/bash
FILE="${1}"
OUTPUT_FOLDER="${2}"

if [[ ! -s "${FILE}" ]]; then 1
  echo "You must provide a non-empty hosts file as an argument."
  exit 1
fi

if [[ -z "${OUTPUT_FOLDER}" ]]; then
  OUTPUT_FOLDER="data" 5
fi

while read -r line; do
  url=$(echo "${line}" | xargs) 2
  if [[ -n "${url}" ]]; then
    echo "Testing ${url} for Directory indexing..."
    if curl -L -s "${url}" | grep -q -e "Index of /" -e "[PARENTDIR]"; then 3
      echo -e "\t -!- Found Directory Indexing page at ${url}"
      echo -e "\t -!- Beginning a recursive download to the \"${OUTPUT_FOLDER}\" folder..."
      mkdir -p "${OUTPUT_FOLDER}"
      wget -q -r -np -R "index.html*" "${url}" -P "${OUTPUT_FOLDER}" 4
    fi
  fi
done < <(cat "${FILE}")
```

You can download this script from *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch05/directory_indexing_scanner.sh*.

In this script, we define the `FILE` and `OUTPUT_FOLDER` variables. Their assigned values are taken from the arguments the user passes on the command line (`$1` and `$2`). We then fail and exit the script (`exit 1`) if the `FILE` variable is not of the file type and not of length zero (`-s`) 1. If the file has a length of non-zero, it means that the file isn't empty (and some data was written into it).

We then use a `while` loop to read the file at the path assigned to the `FILE` variable. At 2, we ensure that each whitespace character in each line from the file is removed by piping it to the `xargs` command. At 3, we use `curl` to make an HTTP GET request and follow any HTTP redirects (using `-L`). We silence cURL's verbose output (using `-s`) with and pipe it to `grep` to find any instances of the strings `Index of /` and `[PARENTDIR]`. These two strings

exist in directory indexing pages. You can verify this by viewing the source HTML page at *http://172.16.10.11/backup*.

If we find either string, we call the `wget` command [4] with the quiet option (`-q`) to silence verbose output, the recursive option (`-r`) to download files recursively from folders, the no-parent option (`-np`) to ensure we download only files at the same level of hierarchy or below (subfolders), and the reject option (`-R`) to exclude files starting with the word *index.html*. We then use the target folder option (`-P`) to download the content to the path specified by the user calling the script (`OUTPUT_FOLDER` variable). If the user did not provide a destination folder, the script will default to use the *data* folder set at [5].

The *acme-impact-alliance* folder appears to be empty. But is it really? When dealing with web servers, you may run into what seem to be dead ends only to find out that there is something hiding there, just not in a very obvious place. Take note of the empty folder for now; we'll resume this exploration in a little bit.

Identifying Suspicious robots.txt Entries

Continue to the third IP address and explore the results from Nikto:

```
+ Server: Apache/2.4.54 (Debian)
+ Retrieved x-powered-by header: PHP/8.0.28
--snip--
+ Uncommon header 'link' found, with contents: <http://172.16.10.12/wp-json/>; rel="https://api.w.org/"
--snip--
+ Entry '/wp-admin/' in robots.txt returned a non-forbidden or redirect HTTP code (302)
+ Entry '/donate.php' in robots.txt returned a non-forbidden or redirect HTTP code (200)
+ "robots.txt" contains 17 entries which should be manually viewed.
+ /wp-login.php: Wordpress login found
--snip--
```

Nikto was able to find a lot more information this time! Other than the missing security headers (which is extremely common to see in the wild, unfortunately) it also found that the server is running on Apache (Debian), and that it is powered by PHP, a backend programming language commonly used in web applications.

The tool also found an uncommon link that points to *http://172.16.10.12/wp-json* and two suspicious entries in the *robots.txt* file, namely */wp-admin/* and */donate.php*. The *robots.txt* file is a special file used to indicate to web crawlers (such as

BLACK HAT BASH

Dolev Farhi and Nick Aleks

Early Access edition, 08/02/23

ISBN 13: 978-1-7185-0374-8 (print)
ISBN 13: 978-1-7185-0375-5 (ebook)

blisher: illia ollock
ana in ditor: ill Franklin
rod ction ana er: Sabrina lo itallo- on le

BLACK HAT BASH

Dolev Farhi and Nick Aleks

Early Access edition, 08/02/23

ISBN 13: 978-1-7185-0374-8 (print)
ISBN 13: 978-1-7185-0375-5 (ebook)

blisher: illia ollock
ana in ditor: ill Franklin
rod ction ana er: Sabrina lo itallo- on le

CONTENTS

The chapters in **red** are included in this Early Access PDF.

CONTENTS

The chapters in **red** are included in this Early Access PDF.

Google's search engine) which endpoints to index and which to ignore. it hinted that the *robots.txt* file may have more entries than just these two, and advised us to inspect it manually.

Lastly, it also identified another endpoint at */wp-login.php*, which is a login page for WordPress, a known blog platform. Navigate to the main page at *http://172.16.10.12/* to confirm you've identified a blog.

Exercise 6: Automatically Exploring Non-Indexed Endpoints

Nikto advised us to manually explore the *robots.txt* file at *http://172.16.10.12/robots.txt* to identify non-indexed endpoints. Finding these endpoints is useful during a penetration test because we can add them to our list of possible targets to test. If you open this file, you should notice a list of paths:

```
User-agent:  *

Disallow: /cgi-bin/
Disallow: /z/j/
Disallow: /z/c/
Disallow: /stats/
--snip--
Disallow: /manual
Disallow: /manual/*
Disallow: /phpmanual/
Disallow: /category/
Disallow: /donate.php
Disallow: /amount_to_donate.txt
```

We identified some of these endpoints earlier (such as */donate.php* and */wp-admin*), but others we didn't see when scanning with Nikto.

Now that we've found these endpoints, we can use bash to see whether they really exist on the server. Let's put together a script that will perform the following activities: make an HTTP request to *robots.txt*, return the response and iterate over each line, parse the output to extract only the paths, make an additional HTTP request to each path separately, and check what status code each path returns to find out if it exists.

Listing 5-1 is an example script that can help do this work. It relies on a useful cURL feature you'll find handy in your bash scripts: built-in variables you can use when you need to make HTTP requests, such as the size of the request sent

`

(`%{size_request}`), the size of the headers returned in bytes (`%{size_header}`), and more.

```
#!/bin/bash
TARGET_URL="http://172.16.10.12"
ROBOTS_FILE="robots.txt"

1 while read -r line; do
  2 path=$(echo "${line}" | awk -F'Disallow: ' '{print $2}')
  3 if [[ -n "${path}" ]]; then
      url="${TARGET_URL}${path}"
      status_code=$(curl -s -o /dev/null -w "%{http_code}" "${url}")
      echo "URL: ${url} returned a status code of: ${status_code}"
    fi

4 done < <(curl -s "${TARGET_URL}/${ROBOTS_FILE}")
```

Listing 5-1 A bash script that reads *robots.txt* and checks individual paths

At 1 we read the output from the `curl` command at 4 line by line. This command makes an HTTP GET request to *http://172.16.10.12/robots.txt*. We then parse each line and grab the second field (which is separated from the others by a space) to extract the path and assign it to the `path` variable 2. We check that the `path` variable length is greater than zero to ensure we were able to properly parse it at 3. Then we create a `url` variable, which is a string concatenated from the `TARGET_URL` variable plus each path from *robots.txt* file, and make an HTTP request to the URL. We then use the `-w` (write-out) variable `%{http_code}` to extract only the status code from the response returned by the web server.

Try using other cURL variables in your own scripts. The full list of variables can be found here at *https://curl.se/docs/manpage.html* or by running the `man curl` command. You can download the script shown in this section at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch05/curl_fetch_robots_txt.sh*.

Brute-Forcing Directories with dirsearch

Dirsearch is a fast directory brute-forcing tool used to find hidden paths and files on web servers. Written in Python by Mauro Soria, dirsearch provides features such as built-in web directory wordlists, bring-your-own-dictionary options, advanced response filtering, and more. We'll use it to try to identify additional attack vectors and verify that Nikto hasn't missed anything obvious.

`

First, let's rescan 172.16.10.10:8081, which yielded no discovered endpoints when scanned by Nikto. The following dirsearch command uses the `-u` (URL) option to specify a base URL from which to start crawling.

```
$ dirsearch -u http://172.16.10.10:8081/

--snip--

Target: http://172.16.10.10:8081/

[00:14:55] Starting:
[00:15:32] 200 -  371B  - /upload
[00:15:35] 200 -   44B  - /uploads
```

Great! This tool was able to pick up two previously unknown endpoints named */upload* and */uploads*. This is why it's important to double and triple-check your results using more than one tool, and also to manually verify the findings, because tools sometimes produce false positives. If you navigate to the */upload* page, you should see a file-upload form. Take note of this endpoint because we'll test it later in the book.

Let's also use dirsearch to look for attack vectors in what looked like an empty folder at *http://172.16.10.11/backup/acme-impact-alliance*:

```
$ dirsearch -u http://172.16.10.11/backup/acme-impact-alliance/

--snip--
Extensions: php, aspx, jsp, html, js | HTTP method: GET | Threads: 30 | Wordlist size: 10927
Target: http://172.16.10.11/backup/acme-impact-alliance/
--snip--
[22:49:53] Starting:
[22:49:53] 301 -  337B  - /backup/acme-impact-alliance/js  ->  http://172.16.10.11/backup/acme-impact-alliance/js/
[22:49:53] 301 -  339B  - /backup/acme-impact-alliance/.git  ->  http://172.16.10.11/backup/acme-impact-alliance/.git/
--snip--
[22:49:53] 200 -   92B  - /backup/acme-impact-alliance/.git/config
--snip--
```

Dirsearch inspects responses returned from the webserver to identify interesting behaviors that could indicate an existence of an asset. For example, it might note whether a certain URL redirects to a new location (specified by an HTTP status code 301) and the response size in bytes. Sometimes, you can infer information and observe behaviors solely by inspecting this information.

This time, we've identified a subfolder within the *acme-impact-alliance* folder named *.git*. A folder with this name usually indicates

that there is a git repository on the server. Git is a source code management tool, and in this case, it likely manages some source code locally on the remote server.

Use dirsearch to perform another directory brute-force against the second directory, namely */backup/acme-hyper-branding*. Save the result into its own folder, then check them. You should find a git repository there, too.

Exploring git Repositories

When you find a git repository, it's often useful to run a specialized git cloner that pulls the repository and all of its associated metadata so you can inspect it locally. For this task, we'll use Gitjacker.

Cloning the Repository with Gitjacker

Gitjacker's command is pretty simple. The first argument is a URL, and the `-o` (output) argument takes a folder name into which the data will be saved if Gitjacker succeeds at pulling the repository:

```
$ gitjacker http://172.16.10.11/backup/acme-impact-alliance/ -o acme-impact-alliance-git

--snip--
Target:     http://172.16.10.11/backup/acme-impact-alliance/
Output Dir: acme-impact-alliance-git
Operation complete.

Status:           Success
Retrieved Objects: 3242
--snip--
```

As you can see, the tool returned a successful status and a few thousand objects. At this point, you should have a folder named *acme-impact-alliance-git*:

```
$ ls -la ./acme-impact-alliance-git

--snip--
128 -rw-r--r--  1 kali kali 127309 Mar 17 23:15 comment.php
 96 -rw-r--r--  1 kali kali  96284 Mar 17 23:15 comment-template.php
 16 -rw-r--r--  1 kali kali  15006 Mar 17 23:15 compat.php
  4 drwxr-xr-x  2 kali kali   4096 Mar 17 23:15 customize
--snip--
 12 -rw-r--r--  1 kali kali  10707 Mar 17 23:15 customize.php
  4 -rw-r--r--  1 kali kali    705 Mar 17 23:15 donate.php
  4 -rw-r--r--  1 kali kali    355 Mar 17 23:15 robots.txt
--snip--
```

`

Notice some familiar filenames in this list? We saw *donate.php* and *robots.txt* earlier, when we scanned the 172.16.10.12 (*p-web-02*) host.

Viewing the Commits with Git Log

When you run into a git repository, you should attempt a `git log` command to see the history of git code commits made to the repository, as they may include interesting data we could use as attackers. In source code management, a *commit* is a snapshot of the code's state that is taken before the code is pushed to the main repository and made permanent. Commit information could include details about who made the commit and a description of the change (such as whether it was a code addition or deletion):

```
$ cd acme-impact-alliance-git
$ git log

commit 3822fd7a063f3890e78051e56bd280f00cc4180c (HEAD -> master)
Author: Kevin Peterson <kpeterson@acme-impact-alliance.com>
--snip--

    commit code
```

As you can see, we've identified a person who has committed code to the git repository: Kevin Peterson, at *kpeterson@acme-impact-alliance.com*. Take note of this information because this account could exist in other places found during the penetration test.

Try running gitjacker again to hijack the git repository that lives on the second folder, at */backup/acme-hyper-branding*. Then execute another `git log` command to see who committed code to this repository, as we did before. The log should reveal the identity of a second person: Melissa Rogers, at *mrogers@acme-hyper-branding.com*.

You may sometimes run into git repositories with many contributors and many commits. We can use git's built in `--pretty=format` option to extract all this metadata very easily, like so:

```
$ git log --pretty=format:"%an %ae"
```

The `%ae` (author name) and `%ae` (email) fields are built-in placeholders in git that allow you to specify values of interest to include in the output. To see the list of all available variables, reference *https://git-scm.com/docs/pretty-formats#_pretty_formats*.

`

Filtering Git Log Information with Bash

Even without the pretty formatting, bash can filter git log output with a single line:

```
$ git log | grep Author | grep -oP '(?<=Author: ).*' | sort -u | tr -d '<>'
```

This bash code runs `git log`, searches for any lines that start with the word `Author` using `grep`, then pipes it to another `grep` command, which uses regular expressions (`-oP`) to filter anything after the word `Author:` and prints only the words that matched. This filtering leaves us with the git commit author's name and email.

Because the same author could have made multiple commits, we use `sort` to sort the list and to remove any duplicated lines using the `-u` option, leaving us with list free of duplicated entries. Lastly, since the email is surrounded by the characters `<>` by default, we trim these characters using `tr -d '<>'`.

Inspecting Repository Files

The repository contains a file called *app.py*. Let's quickly inspect its contents by viewing it using a text editor. If you take a look at the code in Listing 5-2, you'll see that the file contains web server code written with Python's Flask library.

```
import os, subprocess

from flask import (
    Flask,
    send_from_directory,
    send_file,
    render_template,
    request
)


@app.route('/')

--snip--

@app.route('/files/<path:path>')

--snip--

@app.route('/upload', methods = ['GET', 'POST'])

--snip--

@app.route('/uploads', methods=['GET'])
```

```
    --snip--

    @app.route('/uploads/<path:file_name>', methods=['GET'])

    --snip--
```

Listing 5-2 Flask web server source code

The interesting parts here are the endpoints that are exposed using `@app.route()`. You can see that the application exposes endpoints such as `/`, */files*, */upload*, and */uploads*.

Remember that when we scanned our target IP address range using dirsearch and Nikto, we saw two endpoint named */upload* and */uploads* on 172.16.10.10:8081. Because this Python file also has these endpoints, it's very likely that the source code belongs to the application that is running on the server!

You may be asking yourself why we didn't find the */files* endpoint in our scans. Well, web scanners often rely on response status codes returned by web servers to determine if certain endpoints exist or not. If you run the following cURL command using the `-I` (HEAD request) option, you'll see that the */files* endpoint returns an HTTP error of *404 Not Found*:

```
$ curl -I http://172.16.10.10:8081/files

HTTP/1.1 404 NOT FOUND
--snip--
```

Web scanners will interpret these 404 errors as indicating that an endpoint doesn't exist. The reason we get 404 errors here is that on its own, */files* doesn't serve any requests when called directly. Instead, it will serve requests for any web paths appended to */files*, such as */files/abc.jpg* or */files/salary.docx*.

Using Nuclei

Nuclei is one of the most impressive open source vulnerability scanners released in recent years. Its advantage over other tools stems from its community-powered templating system, which reduces false positives by matching known patterns against responses it receives from network services and files. You can also easily extend it to do custom security checks.

`

Nuclei is a vulnerability scanner, so it can naturally support common network services, such as HTTP, DNS, and network sockets, as well as local file scanning. You can use it to send HTTP requests, DNS queries, and raw bytes over the network, and even scan files to find credentials (for example, when you identify an open git repository and want pull it locally to find secrets).

As of this writing, Nuclei has more than 6,000 templates in its database. In this section, we'll introduce Nuclei and how to use it.

Understanding Templates

Nuclei templates are based on YAML files that define the following high level template structure:

ID
A unique identifier for the template

Metadata
Information about the template, such as description, an author, a severity, and tags (arbitrary labels that can group multiple templates, such as *injection* or *denial-of-service*)

Protocol
The mechanism that the template uses to make its requests; for example, `http` is a protocol type that uses HTTP for web requests

Operators
Used for matching patterns against responses received by a template execution (*matchers*) and extracting data (*extractors*), similar to the filtering performed by tools like `grep`

Listing 5-12 is a simple example of a Nuclei template that uses the HTTP protocol to find the default Apache HTML welcome page. Navigate to *http://172.16.10.11/* to see what this page looks like.

```
  id: detect-apache-welcome-page

1 info:
    name: Apache2 Ubuntu Default Page
    author: Dolev Farhi and Nick Aleks
    severity: info
    tags: apache

  http:
    - method: GET
      path:
      2 - '{{BaseURL}}'
```

`

```
    3 matchers:
        - type: word
          words:
            - "Apache2 Ubuntu Default Page: It works"
          part: body
```

Listing 5-3 An example Nuclei template

We define the template metadata, such as the template's name, author, severity, and so on 1. We then define the `http` (HTTP) protocol, which will instruct Nuclei to use an HTTP client when executing this template 2. We also declare that the template should use the `GET` method. Next, we define a variable that will be swapped with the target URL we'll provide to Nuclei on the command line at scan time. Then we define a single matcher of type `word` 3 and a search pattern to match against the HTTP response body coming back from the server, defined by the word `part: body`.

As a result, when Nuclei performs a scan against an IP address that runs some form of a web server, this template will make a GET request to its base URL (`/`) and look for the string `Apache2 ubuntu Default Page: It works` in the response. If it finds this string in the response's body, the check will be considered successful because the pattern matched.

We encourage you to explore Nuclei's templating system at *https://nuclei.projectdiscovery.io/templating-guide*, as you can easily use Nuclei with Bash to perform continuous assessments.

Writing a Custom Template

Let's write a simple template that finds the git repositories we discovered earlier on 172.16.10.11. In Listing 5-4, we define multiple `BaseURL` paths to represent the two paths we've identified, and using Nuclei's *matchers*, we define a string `ref: refs/heads/master` to match against the response body returned by the scanned server.

```
id: detect-git-repository

info:
  name: Git Repository Finder
  author: Dolev Farhi and Nick Aleks
  severity: info
  tags: git

http:
```

`

```
  - method: GET
    path:
      - '{{BaseURL}}/backup/acme-hyper-branding/.git/HEAD'
      - '{{BaseURL}}/backup/acme-impact-alliance/.git/HEAD'
    matchers:
      - type: word
        words:
          - "ref: refs/heads/master"
        part: body
```

Listing 5-4 Writing a git finder template with Nuclei

This template works just like the one in the previous example, except this time, we provide two paths to check against: */backup/acme-hyper-branding/.git/HEAD* and */backup/acme-impact-alliance/.git/HEAD* . The matcher is the string we expect to see in the `HEAD` file. You can confirm that this is the case by making a cURL request to the git repository at 172.16.10.11:

```
$ curl http://172.16.10.11/backup/acme-hyper-branding/.git/HEAD

ref: refs/heads/master
```

Download this custom Nuclei template from *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch05/git-finder.yaml*.

Applying the Template

Let's run Nuclei against 172.16.10.11 with the custom template we just wrote. Nuclei stores its built in templates in the folder *~/.local/nuclei-templates*. First, run the following command to update Nuclei's template database:

```
$ nuclei -ut
```

Next, save the custom template into the folder *~/.local/nuclei-templates/custom* and give it a name such as *git-finder.yaml*.

In the following command, the `-u` (URL) option specifies the address, and `-t` (template) specifies the path to the template:

```
$ nuclei -u 172.16.10.11 -t ~/.local/nuclei-templates/custom/git-finder.yaml

--snip--
[INF] Targets loaded for scan: 1
[INF] Running httpx on input host
[INF] Found 1 URL from httpx
[detect-git-repository] [http] [info] http://172.16.10.11/backup/acme-hyper-branding/.git/HEAD
[detect-git-repository] [http] [info] http://172.16.10.11/backup/acme-impact-alliance/.git/HEAD
```

`

As you can see, we were able to identify the two git repositories with the custom template.

Running a Full Scan

When not provided with a specific template, Nuclei will use its built-in templates during the scan. Running Nuclei is noisy, so we recommend tailoring the execution to a specific target. For instance, if you know a server is running the Apache web server, you could select just the Apache-related templates by specifying the `-tags` option:

```
$ nuclei -tags apache,git -u 172.16.10.11
```

Run `nuclei -tl` to get a list of all available templates.

Let's run a full Nuclei scan against all three IP addresses in the 172.16.10.0/24 network using all of its built-in templates:

```
$ nuclei -u 172.16.10.10:8081
$ nuclei -u 172.16.10.11
$ nuclei -u 172.16.10.12

--snip--
[tech-detect:google-font-api] [http] [info] http://172.16.10.10:8081
[tech-detect:python] [http] [info] http://172.16.10.10:8081
[http-missing-security-headers:access-control-allow-origin] [http] [info] http://172.16.10.10:8081
[http-missing-security-headers:content-security-policy] [http] [info] http://172.16.10.10:8081
--snip--
```

Nuclei tries to optimize the number of total requests made by using the concept of *clustering*. When multiple templates call the same web path (such as */backup*), Nuclei consolidates these into a single request to reduce network overhead. but it could still send thousands of requests during a single scan. You can control how many requests Nuclei sends by specifying the rate limit option (`-rl`) followed by an integer to specify the number of allowed requests per second.

The full scan results in a lot of findings, so append the output to some file (`>>`) so you can examine them one by one. As you can see, Nuclei can find vulnerabilities, but it can also fingerprint the target server and the technologies that are running on it. It should have highlighted findings we've already seen previously as well as a few new findings. We want to draw your attention to a few specific issues it detected:

- An FTP server with anonymous access enabled on 172.16.10.11

`

port 21

- A WordPress login page at *172.16.10.12/wp-login.php*
- A WordPress user-enumeration vulnerability (CVE-2017-5487) at *172.16.10.12/wp-json/wp/v2/users*

Let's confirm these three findings manually to ensure there are no false positives. Connect to the identified FTP server at 172.16.10.11 by issuing the following `ftp` command. This command will connect to the server using the *anonymous* user and an empty password (note that there is nothing specified after the colon (`:`):

```
$ ftp ftp://anonymous:@172.16.10.11

Connected to 172.16.10.11.
220 (vsFTPd 3.0.5)
331 Please specify the password.
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
200 Switching to Binary mode.
```

We were able to connect! Let's issue an `ls` command to verify that we can list files and directories on the server:

```
ftp> ls
229 Entering Extended Passive Mode (|||33817|)
150 Here comes the directory listing.
drwxr-xr-x    1 0        0            4096 Mar 11 05:23 backup
-rw-r--r--    1 0        0           10671 Mar 11 05:22 index.html
226 Directory send OK.
```

We see an *index.html* file and a *backup* folder. This is the same folder that stores the two git repositories we saw earlier, except now we have access to the FTP server where these files actually live.

Next, open a browser to *http://172.16.10.12/wp-login.php* from your Kali machine. You should see the page in Figure 5-2.

`

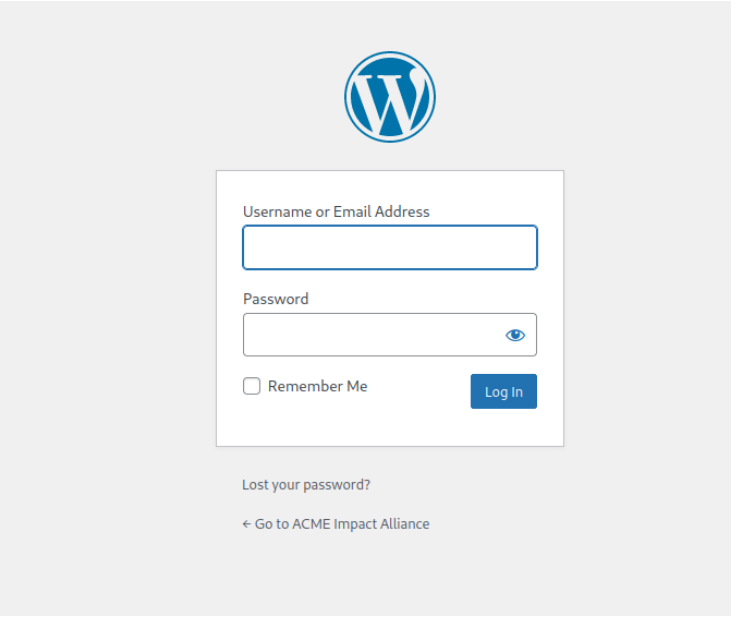

Figure 5-2 The WordPress login page

Lastly, verify the third finding: the WordPress user-enumeration vulnerability, which allows you to gather information about WordPress accounts. By default, every WordPress instance exposes an API endpoint that lists WordPress system users at */wp-json/wp/v2/users*. This endpoint usually doesn't require authentication or authorization, so a simple GET request should return the list of users.

We'll use cURL to send this request and pipe the response to `jq` to prettify the JSON output that comes back. The result should be an array of user data:

```
$ curl -s http://172.16.10.12/wp-json/wp/v2/users/ | jq

[
  {
    "id": 1,
    "name": "jtorres",
    "url": "http://172.16.10.12",
    "description": "",
    "link": "http://172.16.10.12/author/jtorres/",
    "slug": "jtorres",
  },
--snip--
]
```

`

As you can see, there is a single user, *jtorres*, on this blog. This can be a good target to brute-force later on. If this curl command returned many users, you could parse only the usernames with the following `jq` command:

```
$ curl -s http://172.16.10.12/wp-json/wp/v2/users/ | jq .[].name
```

All three findings were true positives, which is great news for us. Let's recap the identities we've identified so far.

Table 5-1 Identity information gathered from git repositories

Source	Name	Email
acme-impact-alliance git repository	Kevin Peterson	kpeterson@acme-impact-alliance.com
acme-hyper-branding git repository	Melissa Rogers	mrogers@acme-hyper-branding.com
WordPress Account	J. Torres	jtorres@acme-impact-alliance.com

Note that in the case of the WordPress account we've identified, we only discovered an account by the name of *jtorres*. Since this was found on the ACME Impact Alliance website, and we already know the email scheme they use (first letter of first name and the last name) it is pretty safe to assume *jtorres* email is *jtorres@acme-impact-alliance.com*. We do not yet know their first name, though.

Exercise 7: Parsing Nuclei's Findings

Nuclei's scan output is a little noisy, and it can be difficult to parse with bash, but not impossible. Nuclei allows you to pass a `-silent` parameter to show only the findings in the output. Before we write a script to parse the output, let's consider Nuclei's output format:

```
[template] [protocol] [severity] url [extractor]
```

Each field is enclosed between brackets `[]` and separated by spaces. The `template` field is a template name (taken from the name of the template file), the `protocol` shows the protocol, such as HTTP, and the `severity` shows the severity of the finding (informational, low, medium, high, or critical). The fourth field is the URL or IP address, and the fifth field is metadata extracted by the template's logic using extractors.

`

Now we should be able to parse this information with bash. Listing 5-5 shows a script to run Nuclei, filter for a specific severity of interest, parse the interesting parts, and email us the results.

```
#!/bin/bash
EMAIL_TO="security@blackhatbash.com"
EMAIL_FROM="nuclei-automation@blackhatbash.com"

for ip_address in "$@"; do
  echo "Testing ${ip_address} with Nuclei..."
1 result=$(nuclei -u "${ip_address}" -silent -severity medium,high,critical)
  if [[ -n "${result}" ]]; then
  2 while read -r line; do
      template=$(echo "${line}" | awk '{print $1}' | tr -d '[]')
      url=$(echo "${line}" | awk '{print $4}')
      echo "Sending an email with the findings ${template} ${url}"
      sendemail -f "${EMAIL_FROM}" \
              3 -t "${EMAIL_TO}" \
                -u "[Nuclei] Vulnerability Found!" \
                -m "${template} - ${url}"

  4 done <<< "${result}"
  fi
done
```

Listing 5-5 Scanning with Nuclei and sending ourselves the results

Let's dissect the code to better understand what it's doing. We use a `for` loop to iterate through values in the `$@` variable, which is a special value you learned about in Chapter 2 that contains the arguments passed to the script on the command line. We assign each argument to the `ip_address` variable.

Next, we run a Nuclei scan, passing it the `-severity` argument to scan for vulnerabilities categorized as either medium, high, or critical, and save the output to the `result` variable 1. At 2, we read the output passed to the `while` loop at 4 line by line. From each line, we extract the first field, using the `tr -d '[]'` command to remove the `[]` characters for a cleaner output. We also extract the fourth field from each line, which is where Nuclei stores the vulnerable URL. At 3 we send an email containing all the relevant information.

To run this script, save it to a file and pass the IP addresses to scan on the command line:

```
$ nuclei-notifier.sh 172.16.10.10:8081 172.16.10.11 172.16.10.12 172.16.10.13
```

`

Note that Nuclei can format the output in JSON format if you use the `-j` option. You can then pipe this output to `jq`, as we did earlier.

You can download this script at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch05/nuclei-notifier.sh*.

Fuzzing for Hidden Files

Now that we've identified potential location of files, let's use fuzzing tools to try to find hidden files on *http://172.16.10.10:8081/files*. *Fuzzers* generate semi-random data to use as part of some payload. When sent to an application, these payloads can trigger anomalous behavior or reveal covert information. You can use fuzzers against web servers to find hidden paths or against local binaries to find vulnerabilities such as buffer overflows or denials of service.

Creating a Tailored Wordlist of Possible Filenames

Fuzzing tools in the context of web application enumeration work best when fed custom wordlists tailored to your target. These lists could contain the name of the company, the individuals you've identified, relevant locations, and so on. These tailored wordlists can help you identify user accounts to attack, network and application services, valid domain names, covert files, email addresses, and web paths, for example.

Let's use bash to write a custom wordlist containing potential filenames of interest:

```
$ echo -e acme-hyper-branding-{0..100}.{txt,csv,pdf,jpg}"\n" | sed 's/ //g' > files_wordlist.txt
```

Listing 5-6 Using brace expansion to create multiple files with various extensions

This command creates files with probable file extensions tailored to our target's name, ACME Hyper Branding. It uses `echo` with brace expansion `{0..100}` to create arbitrary strings from 0 to 100, then appends these to the company name. We also use brace expansion to create multiple file extension types, such as *txt*, *csv*, *pdf*, and *jpg*. The `-e` option for `echo` enables us to interpret backslash (`\`) escapes. This means that `\n` will be interpreted as a new line. We then pipe this output to the `sed` command to remove all whitespaces from the output for a cleaner list.

`

Use `head` to view the created files:

```
$ head files_wordlist.txt

acme-hyper-branding-0.txt
acme-hyper-branding-0.csv
acme-hyper-branding-0.pdf
acme-hyper-branding-0.jpg
acme-hyper-branding-1.txt
acme-hyper-branding-1.csv
acme-hyper-branding-1.pdf
acme-hyper-branding-1.jpg
acme-hyper-branding-2.txt
acme-hyper-branding-2.csv
```

As you can see, this command's output follows the format *acme-hyper-branding-{some_number}.{some_extension}*.

Fuzzing with Ffuf

Ffuf (an acronym for Fuzz Faster U Fool) is a versatile and blazing-fast web fuzzing tool. We'll use fuff to discover potential files under the */files* endpoint that could contain interesting data.

This `ffuf` command uses the `-c` (color) option to highlight the results in the terminal, `-w` (wordlist) to specify a custom word list, `-u` (URL) option to specify a path, and the full URL to the endpoint to fuzz. Let's run ffuf against 172.16.10.10 (*p-web-01*) using the command shown below:

```
$ ffuf -c -w files_wordlist.txt -u http://172.16.10.10:8081/files/FUZZ

:: Method           : GET
:: URL              : http://172.16.10.10:8081/files/FUZZ
:: Wordlist         : FUZZ: files_wordlist.txt
:: Follow redirects : false
:: Calibration      : false
:: Timeout          : 10
:: Threads          : 40
:: Matcher          : Response status: 200,204,301,302,307,401,403,405,500
________________________________________________

acme-hyper-branding-5.csv [Status: 200, Size: 432, Words: 31, Lines: 9, Duration: 32ms]
:: Progress: [405/405] :: Job [1/1] :: 0 req/sec :: Duration: [0:00:00] :: Errors: 0 ::
```

Listing 5-7 Fuzzing with ffuf

Note that the word `FUZZ` at the end of the URL is a placeholder that tells the tool where to inject the words from the wordlist. In essence, it will swap the word `FUZZ` with each line from our file.

`

The output indicates that ffuf has identified that the path *http://172.16.10.10:8081/files/acme-hyper-branding-5.csv* returned a status code of *HTTP 200 OK*. If you look closely at the output, you should see that the fuzzer sent 405 requests in less than a second, which is pretty impressive.

Fuzzing with Wfuzz

Wfuzz is another web fuzzing tool that can do similar things to ffuf. In fact, ffuf is based on Wfuzz. Let's use Wfuzz to perform the same type of word list-based scan (`-w`), then use its filtering capabilities to show only files that receive a response status code of 200 OK (`--sc 200`):

```
$ wfuzz --sc 200 -w files_wordlist.txt http://172.16.10.10:8081/files/FUZZ

--snip--

Target: http://172.16.10.10:8081/files/FUZZ
Total requests: 405

=====================================================================
ID          Response   Lines    Word       Chars       Payload
=====================================================================

000000022: 200        8 L      37 W       432 Ch      "acme-hyper-branding-5.csv"

Total time: 0
Processed Requests: 405
Filtered Requests: 404
Requests/sec.: 0
```

Next, let's use the `wget` command to download this file.

```
$ wget http://172.16.10.10:8081/files/acme-hyper-branding-5.csv

$ cat acme-hyper-branding-5.csv

no, first_name, last_name, designation, email
1, Jacob, Taylor, Founder, jtayoler@acme-hyper-branding.com
2, Sarah, Lewis, Executive Assistance, slewis@acme-hyper-branding.com
3, Nicholas, Young, Influencer, nyoung@acme-hyper-branding.com
4, Lauren, Scott, Influencer, lscott@acme-hyper-branding.com
5, Aaron,Peres, Marketing Lead, aperes@acme-hyper-branding.com
6, Melissa, Rogers, Marketing Lead, mrogers@acme-hyper-branding.com
```

We've identified a table of personally identifiable information (PII), including first and last names, titles, and email addresses. Take notes of every detail we've managed to extract in this chapter; you never know when it will come in handy.

`

Note that fuzzers can cause unintentional denial of service conditions, especially if they are optimized for speed. You may run into applications running on low-powered servers that will crash as a result of running a highly-capable fuzzer against them, so make sure you have explicit permission from the company you're working with to perform such activities.

Assessing SSH Servers with Nmap's Scripting Engine

Nmap contains many NSE scripts that can also help test for vulnerabilities and misconfigurations. All Nmap scripts live in the */usr/share/nmap/scripts* path. When you run Nmap with the `-A` flag, it will blast all NSE scripts at the target, as well as enable operating system detection, version detection, script scanning, and traceroute. This is probably the noisiest scan you can do with Nmap, so never use it when you need to be covert.

In the previous chapter, we identified a server running OpenSSH on 172.16.10.13 (*p-jumpbox-01*). Let's use an NSE script tailored to SSH servers to see what we can discover about the supported authentication methods:

```
$ nmap --script=ssh-auth-methods 172.16.10.13

Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-19 01:53 EDT
--snip--
PORT   STATE SERVICE
22/tcp open  ssh
| ssh-auth-methods:
|   Supported authentication methods:
|     publickey
|_    password

Nmap done: 1 IP address (1 host up) scanned in 0.26 seconds
```

The *ssh-auth-methods* NSE script enumerates the supported authentication methods offered by the SSH server. If *password* is one of them, this means that the server accepts passwords as an authentication mechanism. SSH servers that allow password authentication are prone to brute-force attacks. Later in this chapter, we will perform a brute force against SSH servers.

`

Exercise 8: Combining Multiple Tools to Achieve Your Objective

The goal of this exercise is to write a script that calls several security tools, parses their output, and passes the output to other tools to act on it. Orchestrating multiple tools in this way is a very common task in penetration testing, so we encourage you get comfortable with building such workflows.

Your script should do the following:

1. Accept one or more IP addresses on the command line.
2. Run a port scanner against the IP addresses; which port scanner you use is completely up to you.
3. Identify open ports. If any of them are FTP ports (21/TCP) the script should pass the address to a vulnerability scanner in Step 4.
4. Scan the addresses and ports using Nuclei. You can use templates dedicated to finding issues in FTP servers. Search in the Nuclei templates folder */home/kali/.local/nuclei-templates* for FTP-related templates or use the `-tags ftp` Nuclei flag.
5. Scan the address using Nmap. Use NSE scripts that find vulnerabilities in FTP servers, which you can find under the */usr/share/nmap/scripts* folder. For example, try *ftp-anon.nse*.
6. Parse and write the results to a file with a format of your choice. One example could be an HTML file. The file should include a description of the vulnerability, the relevant IP address and port, the timestamp at which it was found, and the name of the tool that detected the issue. There is no hard requirement about how the data should be presented; one option is to use an HTML table. If you need an example table, you can find one at *https://github.com/dolevf/Black-Hat-Bash/blob/master/ch05/vulnerability_table.html* (open this file in a web browser to view the table). Alternatively, you could write the results to a CSV file.

As you know, there is more than one way to write such script. Only the end result matters, so write it as you see fit!

`

Summary

In this chapter, we wrapped up our reconnaissance activities by performing vulnerability scanning and fuzzing. We also verified the vulnerabilities we discovered, weeding out potential false positives.

Along the way, we used bash scripting to perform several tasks. We scanned for vulnerabilities, wrote custom scripts that can perform recursive downloads from misconfigured webservers, extracted sensitive information from git repositories, and more. We also created custom wordlists using clever bash scripting and orchestrated the execution of multiple security tools to generate a report.

Let's recap what we've identified so far, from a reconnaissance perspective:

1. Hosts running multiple services (HTTP, FTP, SSH) and their versions
2. A web server running WordPress with a login page enabled and a few vulnerabilities, such as user enumeration and an absence of HTTP security headers
3. A web server with a revealing *robots.txt* file containing paths to custom upload forms and a donation page.
4. An anonymous login-enabled FTP server
5. Multiple open git repositories
6. OpenSSH servers that allows password-based logins

In the next chapter, we will use the vulnerabilities identified in this chapter to establish an initial foothold by exploiting vulnerabilities and taking over servers.