CHAPTER 2

Bash Scripting

In the previous chapter, you learned lots of commands in Linux. Now, let's take your skills to the next level in the command-line tools. In this chapter, you will see how to create scripted commands using Bash based on what you have learned so far.

Why Bash scripting? The universality of Bash gives us, penetration testers, the flexibility of executing powerful terminal commands without the need to install a compiler or an integrated development environment (IDE). To develop a Bash script, all you need is a text editor, and you're good to go.

When should you use Bash scripts? That's an important question to tackle before starting this chapter! Bash is not meant for developing sophisticated tools. If that's what you would like to do, you should use Python instead (Python fundamentals are covered later in this book). Bash is used for quick, small tools that you implement when you want to save time (e.g., to avoid repeating the same commands, you just write them in a Bash script).

This chapter will not only teach you the Bash scripting language, it will go beyond that to show you the ideology of programming as well. If you're new to programming, this is a good starting point for you to understand how programming languages work (they share a lot of similarities).

Here's what you're going to learn in this chapter:

- Printing to the screen using Bash
- Using variables

- Using script parameters
- Handling user input
- Creating functions
- Using conditional `if` statements
- Using `while` and `for` loops

Basic Bash Scripting

Figure 2.1 summarizes all the commands, so you can use it as a reference to grasp all the contents of this chapter. In summary, basic Bash scripting is divided into the following categories:

- Variables
- Functions
- User input
- Script output
- Parameters

Printing to the Screen in Bash

There are two common ways to write into the terminal command-line output using Bash scripting. The first simple method is to use the `echo` command that we saw in the previous chapter (we include the text value inside single quotes or double quotes):

```
$echo 'message to print.'
```

The second method is the `printf` command; this command is more flexible than the `echo` command because it allows you to format the string that you want to print:

```
$printf 'message to print'
```

The previous formula is too simplified; in fact, `printf` allows you to format strings as well (not just for printing; it's more than that). Let's look at an example: if we want to display the number of live hosts in a network, we can use the following pattern:

```
root@kali:~# printf "%s %d\n" "Number of live hosts:" 15
Number of live hosts: 15
```

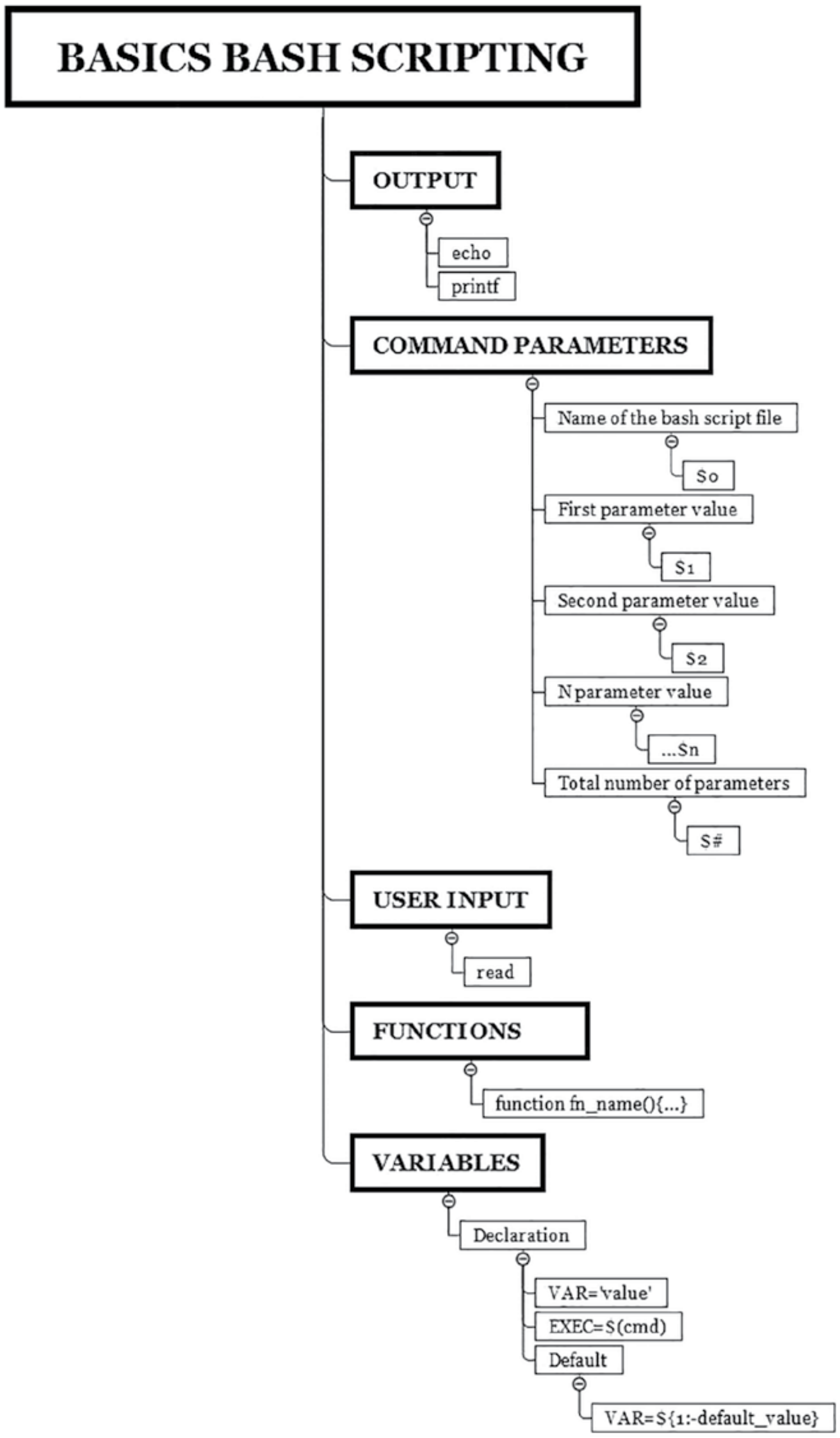


Figure 2.1: Bash Scripting

Let's divide the command so you can understand what's going on:

- `%s`: Means we're inserting a string (text) in this position
- `%d`: Means we're adding a decimal (number) in this position
- `\n`: Means that we want to go to a new line when the print is finished

Also, take note that we are using double quotes instead of single quotes. Double quotes will allow us to be more flexible with string manipulation than the single quotes. So, most of the time, we can use the double quotes for `printf` (we rarely need to use the single quotes).

To format a string using the `printf` command, you can use the following patterns:

- `%s`: String (texts)
- `%d`: Decimal (numbers)
- `%f`: Floating-point (including signed numbers)
- `%x`: Hexadecimal
- `\n`: New line
- `\r`: Carriage return
- `\t`: Horizontal tab

Variables

What is a variable, and why does every programming language use it anyway?

Consider a variable as a storage area where you can save things like strings and numbers. The goal is to reuse them over and over again in your program, and this concept applies to any programming language (not just Bash scripting).

To declare a variable, you give it a name and a value (the value is a string by default). The name of the variable can only contain an alphabetic character or underscore (other programming languages use a different naming convention). For example, if you want to store the IP address of the router in a variable, first you will create a file `var.sh` (Bash script files will end with `.sh`), and inside the file, you'll enter the following:

```
#!/bin/bash
#Simple program with a variable

ROUTERIP="10.0.0.1"

printf "The router IP address: $ROUTERIP\n"
```

Let's explain your first Bash script file:

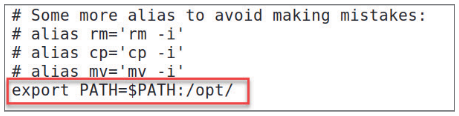
- `#!/bin/bash` is called the *Bash shebang*; we need to include it at the top to tell Kali Linux which interpreter to use to parse the script file (we will use the same concept in Chapter 18, "Pentest Automation with Python," with the Python programming language). The `#` is used in the second line to indicate that it's a comment (a comment is a directive that the creator will leave inside the source code/script for later reference).
- The variable name is called `ROUTERIP`, and its value is 10.0.0.1.
- Finally, we're *printing* the value to the output screen using the `printf` function.

To execute it, make sure to give it the right permissions first (look at the following output to see what happens if you don't). Since we're inside the same directory (`/root`), we will use `./var.sh` to execute it:

```
root@kali:~# ./var.sh
bash: ./var.sh: Permission denied
root@kali:~# chmod +x var.sh
root@kali:~# ./var.sh
The router IP address: 10.0.0.1
```

Congratulations, you just built your first Bash script! Let's say we want this script to *run automatically* without specifying its path anywhere in the system. To do that, we must add it to the `$PATH` variable. In our case, we will add `/opt` to the `$PATH` variable so we can save our custom scripts in this directory.

First, open the `.bashrc` file using any text editor. Once the file is loaded, scroll to the bottom and add the line highlighted in Figure 2.2.



```
# Some more alias to avoid making mistakes:
# alias rm='rm -i'
# alias cp='cp -i'
# alias mv='mv -i'
export PATH=$PATH:/opt/
```

Figure 2.2: Export Config

The changes will append `/opt` to the `$PATH` variable. At this stage, save the file and close all the terminal sessions. Reopen the terminal window and copy the script file to the `/opt` folder. From now on, we don't need to include its path; we just execute it by typing the script name `var.sh` (you don't need to re-execute the `chmod` again; the execution permission has been already set):

```
root@kali:~# cp var.sh /opt/
root@kali:~# cd /opt
root@kali:/opt# ls -la | grep "var.sh"
```

Continues

(continued)

```
-rwxr-xr-x  1 root root       110 Sep 28 11:24 var.sh
root@kali:/opt# var.sh
The router IP address: 10.0.0.1
```

Commands Variable

Sometimes, you might want to execute commands and save their output to a variable. Most of the time, the goal behind this is to manipulate the contents of the command output. Here's a simple command that executes the `ls` command and filters out the filenames that contain the word *simple* using the `grep` command. (Don't worry, you will see more complex scenarios in the upcoming sections of this chapter. For the time being, practice and focus on the fundamentals.)

```
 #!/bin/bash
LS_CMD=$(ls | grep 'simple')
printf "$LS_CMD\n"
```

Here are the script execution results:

```
root@kali:/opt# simplels.sh
simpleadd.sh
simplels.sh
```

Script Parameters

Sometimes, you will need to supply parameters to your Bash script. You will have to separate each parameter with a space, and then you can manipulate those params inside the Bash script. Let's create a simple calculator (`simpleadd.sh`) that adds two numbers:

```
#!/bin/bash
#Simple calculator that adds 2 numbers

#Store the first parameter in num1 variable
NUM1=$1
#Store the second parameter in num2 variable
NUM2=$2
#Store the addition results in the total variable
TOTAL=$(($NUM1 + $NUM2))

echo '########################'
printf "%s %d\n" "The total is =" $TOTAL
echo '########################'
```

You can see in the previous script that we accessed the first parameter using the `$1` syntax and the second parameter using `$2` (you can add as many parameters as you want).

Let's add two numbers together using our new script file (take note that I'm storing my scripts in the `opt` folder from now on):

```
root@kali:/opt# simpleadd.sh 5 2
########################
The total is = 7
########################
```

There is a limitation to the previous script; it can add only two numbers. What if you want to have the flexibility to add two to five numbers? In this case, we can use the default parameter functionality. In other words, by default, all the parameter values are set to zero, and we add them up once a real value is supplied from the script:

```
#!/bin/bash
#Simple calculator that adds until 5 numbers

#Store the first parameter in num1 variable
NUM1=${1:-0}
#Store the second parameter in num2 variable
NUM2=${2:-0}
#Store the third parameter in num3 variable
NUM3=${3:-0}
#Store the fourth parameter in num4 variable
NUM4=${4:-0}
#Store the fifth parameter in num5 variable
NUM5=${5:-0}
#Store the addition results in the total variable
TOTAL=$(($NUM1 + $NUM2 + $NUM3 + $NUM4 + $NUM5))

echo '########################'
printf "%s %d\n" "The total is =" $TOTAL
echo '########################'
```

To understand how it works, let's look at the `NUM1` variable as an example (the same concept applies to the five variables). We will tell it to read the first parameter `{1` from the terminal window, and if it's not supplied by the user, then set it to zero, as in `:-0}`.

Using the default variables, we're not limited to adding five numbers; from now on, we can add as many numbers as we want, but the maximum is five (in the following example, we will add three digits):

```
root@kali:~# simpleadd.sh 2 4 4
########################
The total is = 10
########################
```

TIP **If you want to know the number of parameters supplied in the script, then you can use the `$#` to get the total. Based on the preceding example, the `$#` will be equal to three since we're passing three arguments.**

If you add the following line after the `printf` line:

```
printf "%s %d\n" "The total number of params =" $#
```

you should see the following in the terminal window:

```
root@kali:~# simpleadd.sh 2 4 4
########################
The total is = 10
The total number of params = 3
########################
```

User Input

Another way to interact with the supplied input from the shell script is to use the read function. Again, the best way to explain this is through examples. We will ask the user to enter their first name and last name after which we will print the full name on the screen:

```
  #!/bin/bash

read -p "Please enter your first name:" FIRSTNAME
read -p "Please enter your last name:" LASTNAME

printf "Your full name is: $FIRSTNAME $LASTNAME\n"
```

To execute it, we just enter the script name (we don't need to supply any parameters like we did before). Once we enter the script's name, we will be prompted with the messages defined in the previous script:

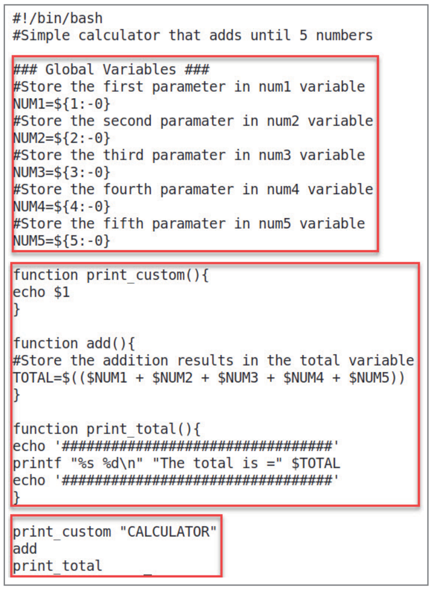
```
root@kali:~# nameprint.sh
Please enter your first name:Gus
Please enter your last name:Khawaja
Your full name is: Gus Khawaja
```

Functions

Functions are a way to organize your Bash script into logical sections instead of having an unorganized structure (programmers call it *spaghetti code*). Let's take the earlier calculator program and reorganize it (refactor it) to make it look better.

This Bash script (in Figure 2.3) is divided into three sections:

- In the first section, we create all the global variables. Global variables are accessible inside any function you create. For example, we are able to use all the `NUM` variables declared in the example inside the `add` function.
- Next, we build the functions by dividing our applications into logical sections. The `print_custom()` function will just print any text that we give it. We're using the `$1` to access the parameter value passed to this function (which is the string `CALCULATOR`).
- Finally, we call each function sequentially (each one by its name). Print the header, add the numbers, and, finally, print the results.



```
#!/bin/bash
#Simple calculator that adds until 5 numbers

### Global Variables ###
#Store the first parameter in num1 variable
NUM1=${1:-0}
#Store the second paramater in num2 variable
NUM2=${2:-0}
#Store the third paramater in num3 variable
NUM3=${3:-0}
#Store the fourth paramater in num4 variable
NUM4=${4:-0}
#Store the fifth paramater in num5 variable
NUM5=${5:-0}

function print_custom(){
echo $1
}

function add(){
#Store the addition results in the total variable
TOTAL=$(($NUM1 + $NUM2 + $NUM3 + $NUM4 + $NUM5))
}

function print_total(){
echo '#################################'
printf "%s %d\n" "The total is =" $TOTAL
echo '#################################'
}

print_custom "CALCULATOR"
add
print_total
```

Figure 2.3: Script Sections

Conditions and Loops

Now that you know the basics of Bash scripting, we can introduce more advanced techniques. When you develop programs in most programming languages (e.g., PHP, Python, C, C++, C#, etc.), including Bash scripting, you will encounter conditions (`if` statements) and loops, as shown in Figure 2.4.

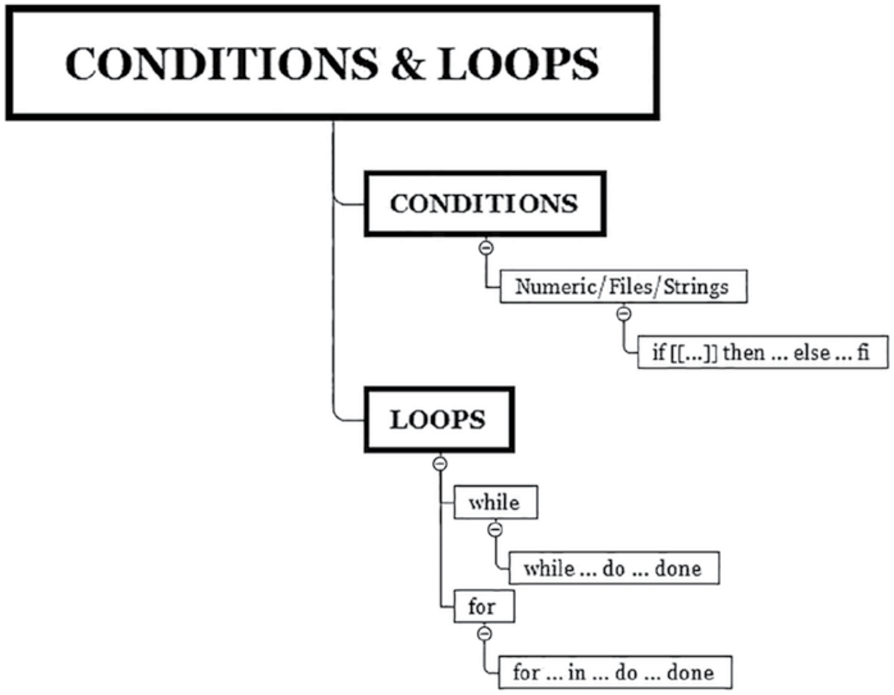


Figure 2.4: Conditions and Loops

Conditions

An `if` statement takes the following pattern:

```
if [[ comparison ]]
then
True, do something
else
False, Do something else
fi
```

If you've been paying attention, you know that the best way to explain this pattern is through examples. Let's develop a program that pings a host using Nmap, and we'll display the state of the machine depending on the condition (the host is up or down):

```
#!/bin/bash
#Ping a host using Nmap

### Global Variables ###
#Store IP address
IP_ADDRESS=$1
```

```
function ping_host(){
ping_cmd=$(nmap -sn $IP_ADDRESS | grep 'Host is up' | cut -d '(' -f 1)
}

function print_status(){
if [[ -z $ping_cmd ]]
then
echo 'Host is down'
else
echo 'Host is up'
fi
}

ping_host
print_status
```

The `nmap` command either returns an empty string text if the host is down or returns the value "Host is up" if it's responding. (Try to execute the full `nmap` command in your terminal window to visualize the difference. If so, replace `$IP_ADDRESS` with a real IP address.) In the `if` condition, the `-z` option will check if the string is empty; if yes, then we print "Host is down" or else we print "Host is up:"

```
root@kali:~# simpleping.sh 10.0.0.11
Host is down
root@kali:~# simpleping.sh 10.0.0.1
Host is up
```

What about other condition statements? In fact, you can compare numbers, strings, or files, as shown in Tables 2.1, 2.2, and 2.3.

Table 2.1: Numerical Conditions

Equal	`[[ x -eq y ]]`
Not equal	`[[ x -ne y ]]`
Less than	`[[ x -lt y ]]`
Greater than	`[[ x -gt y ]]`

Table 2.2: String Conditions

Equal	`[[ str1 == str2 ]]`
Not equal	`[[ str1 != str2 ]]`
Empty string	`[[ -z str ]]`
Not empty string	`[[ -n str ]]`

Table 2.3: File/Directory Conditions

File exists?	`[[ -a filename ]]`
Directory exists?	`[[ -d directoryname ]]`
Readable file?	`[[ -r filename ]]`
Writable file?	`[[ -w filename ]]`
Executable file?	`[[ -x filename ]]`
File not empty?	`[[ -s filename ]]`

Loops

You can write loops in two different ways: using a `while` loop or using a `for` loop. Most of the programming languages use the same pattern for loops. So, if you understand how loops work in Bash, the same concept will apply for Python, for example.

Let's start with a `while` loop that takes the following structure:

```
while [[ condition ]]
do
do something
done
```

The best way to explain a loop is through a counter from 1 to 10. We'll develop a program that displays a progress bar:

```
#!/bin/bash
#Progress bar with a while loop

#Counter
COUNTER=1
#Bar
BAR='##########'

while [[ $COUNTER -lt 11 ]]
do
#Print the bar progress starting from the zero index
echo -ne "\r${BAR:0:COUNTER}"
#Sleep for 1 second
sleep 1
#Increment counter
COUNTER=$(( $COUNTER +1 ))
done
```

Note that the condition (`[[ $COUNTER -lt 11]]`) in the `while` loop follows the same rules as the `if` condition. Since we want the counter to stop at 10, we will use the following mathematical formula: `counter<11`. Each time the counter is incremented, it will display the progress. To make this program more interesting, let it sleep for one second before going into the next number.

On the other hand, the `for` loop will take the following pattern:

```
for ... in [List of items]
do
something
done
```

We will take the same example as before but use it with a `for` loop. You will realize that the `for` loop is more flexible to implement than the `while` loop. (Honestly, I rarely use the `while` loop.) Also, you won't need to increment your index counter; it's done automatically for you:

```
#!/bin/bash
#Progress bar with a For Loop

#Bar
BAR='##########'

for COUNTER in {1..10}
do
#Print the bar progress starting from the zero index
echo -ne "\r${BAR:0:$COUNTER}"
#Sleep for 1 second
sleep 1
done
```

File Iteration

Here's what you should do to simply read a text file in Bash using the `for` loop:

```
for line in $(cat filename)
do
do something
done
```

In the following example, we will save a list of IP addresses in a file called `ips.txt`. Then, we will reuse the Nmap ping program (that we created previously) to check whether every IP address is up or down. On top of that, we will check the DNS name of each IP address:

```
#!/bin/bash
#Ping & get DNS name from a list of IPs saved in a file

#Prompt the user to enter a file name and its path.
read -p "Enter the IP addresses file name / path:" FILE_PATH_NAME
```

Continues

(continued)

```
function check_host(){
        #if not the IP address value is empty
        if [[ -n $IP_ADDRESS ]]
        then
                ping_cmd=$(nmap -sn $IP_ADDRESS| grep 'Host is up' | cut
-d '(' -f 1)
                echo '------------------------------------------------'
                if [[ -z $ping_cmd ]]
                then
                        printf "$IP_ADDRESS is down\n"
                else
                        printf "$IP_ADDRESS is up\n"
                        dns_name
                fi
        fi
}

function dns_name(){
        dns_name=$(host $IP_ADDRESS)
        printf "$dns_name\n"
}

#Iterate through the IP addresses inside the file
for ip in $(cat $FILE_PATH_NAME)
do
        IP_ADDRESS=$ip
        check_host
done
```

If you have followed carefully through this chapter, you should be able to understand everything you see in the previous code. The only difference in this program is that I used Tab spacing to make the script look better. The previous example covers most of what we did so far, including the following:

- User input
- Declaring variables
- Using functions
- Using `if` conditions
- Loop iterations
- Printing to the screen

Summary

I hope you have practiced all the exercises in this chapter, especially if you're new to programming. A lot of the concepts mentioned will apply to many programming languages, so consider the exercises as an opportunity to learn the basics.

I personally use Bash scripting for small and quick scenarios. If you want to build more complex applications, then you can try doing that in Python instead. Don't worry! You will learn about Python at the end of this book so you can tackle any situation you want in your career as a penetration tester.

Finally, this chapter covered a lot of information about Bash scripting. However, there is a lot more information than what is in this chapter. In practice, I use internet search engines to quickly find Bash scripting references. In fact, you don't need memorize everything you learned in this chapter. Remember that this book is a reference on which you can always rely to remember the syntaxes used in each case.