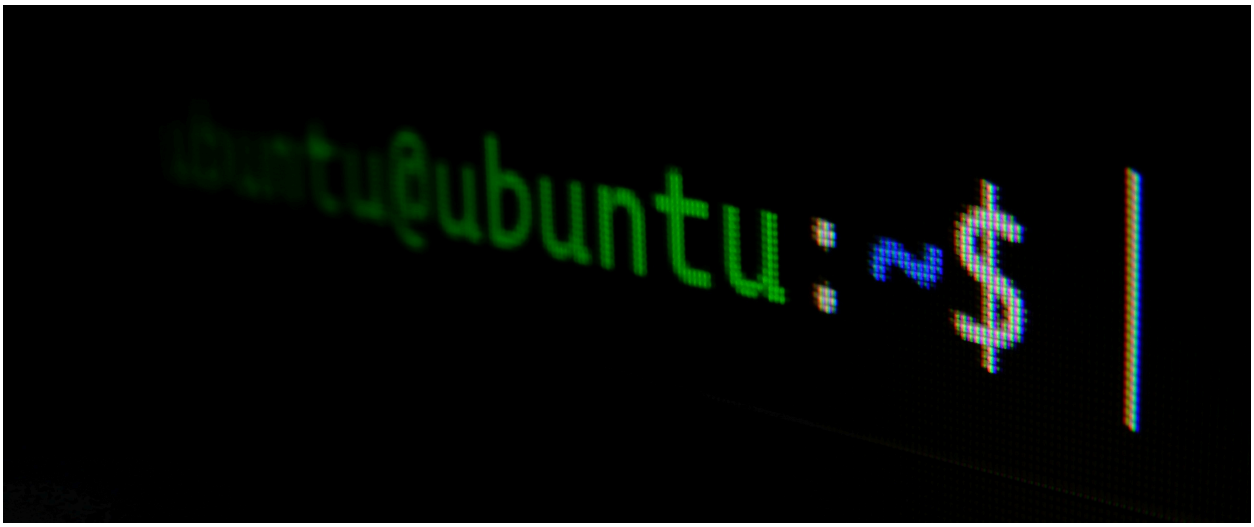


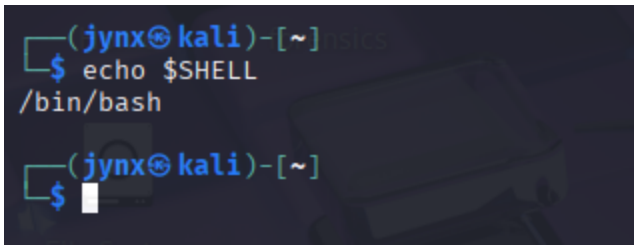
CHAPTER 8 - BASH SCRIPTING

[8.1] INTRODUCTION

Proficient hackers and Linux system administrators should possess ***scripting abilities*** as a fundamental skill. Hackers frequently require **automation** of various commands, often combining multiple tools, which is best accomplished by creating custom short programs. This chapter introduces basic bash shell scripting to help beginners develop these essential automation skills.

Shell is the ***interface*** between user and the operating system, which allows the liberty and flexibility to customize commands and instructions closer to the hardware than a GUI program.
Linux offers various shell options such as the Korn shell, Z shell, C shell, and the Bourne-again shell (commonly called bash). Since bash is universally available across virtually all Linux and UNIX systems, including macOS and Kali distributions, we will focus solely on using the bash in the context of our work in this chapter.

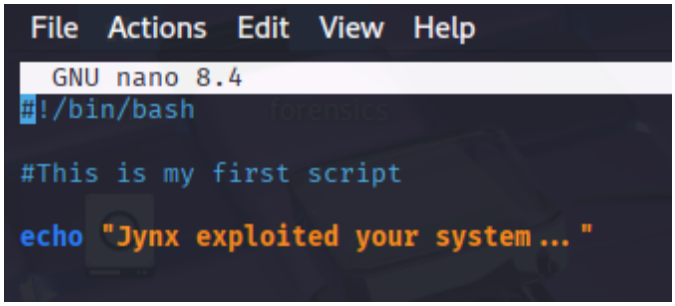
example ***[determine your current shell]:***



▼ [8.2] How to create your FIRST Script ?

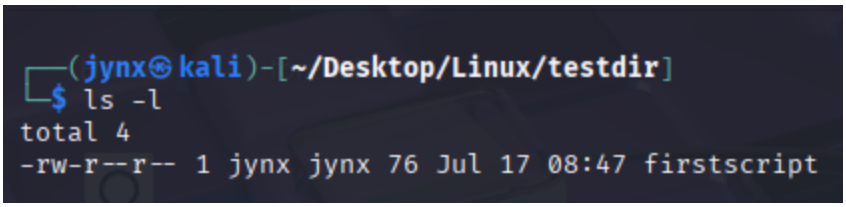
To write a simple script that prints a message on the screen, open any of your supported text-editor, begin the first line with- a ***shebang*** a hash mark + exclamation mark [`#!`] followed by `/bin/bash` essentially, to indicate that the operating sytem to interpret it as a bash shell script. Then a message with the `echo` command.

example :



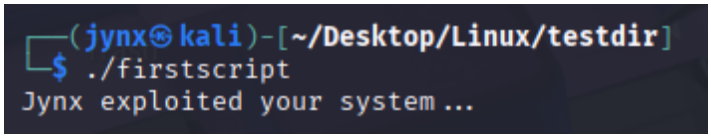
Save the file under a name such as “firstscript”.

Be default, the any shell script is not executable by anyone, not even the owner of the file for obvious security reasons :



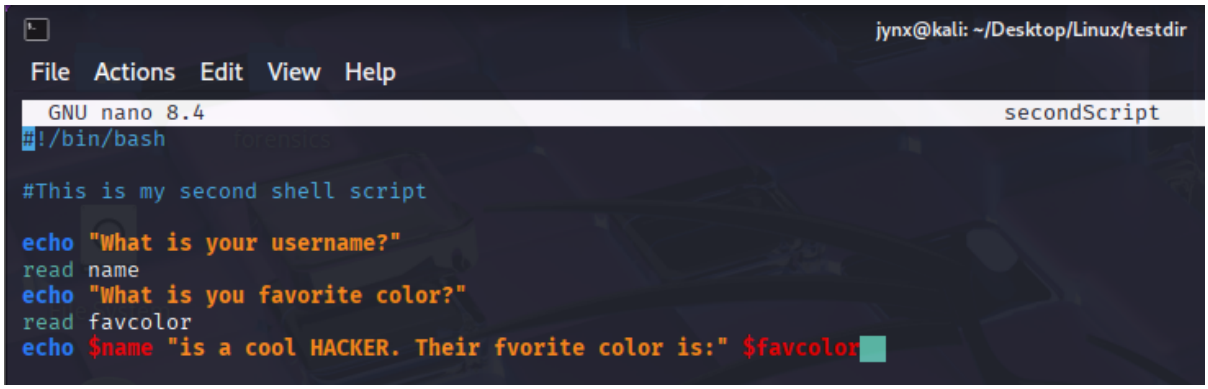
Set the execute permission for owner; `chmod 755 firstscript` [change mode].

Now to finally run your script, enter `./` before the filename to indicate the system to execute the script in the file. Press Enter and see the script return our inserted message.



▼ [8.3] Advance functional SCRIPTS

To create a more advanced style of shell script, we need ***variables.*** To store and manipulate data [strings, numbers, commands, words etc.].
Let's illustrate by prompting the user their name and favorite color then display the same with a message :



Save the file, change the execute permission or owner like we did with our first script and then execute :

```
┌──(jynx㉿kali)-[~/Desktop/Linux/testdir]
└─$ ./secondScript
What is your username?
Jynx
What is you favorite color?
Purple
Jynx is a cool HACKER. Their fvorite color is: Purple

┌──(jynx㉿kali)-[~/Desktop/Linux/testdir]
└─$
```

▼ [8.4] Script to Scan OPEN PORTS

Let's try to scan for open ports on our local system to see where it is closed or open to exploits of any form that we are not aware of.

```
jynx@kali: ~/Desktop/Linux/testdir
File  Actions  Edit  View  Help
  GNU nano 8.4                                    openPorts.sh
#!/bin/bash

#The script scans to find if any ports are open on the current LAN connection

echo "Open ports on local system:"
netstat -tuln | grep LISTEN
```

Make the file executable as we did previously:

```
┌──(jynx㉿kali)-[~/Desktop/Linux/testdir]
└─$ chmod 755 openPorts.sh

┌──(jynx㉿kali)-[~/Desktop/Linux/testdir]
└─$ ./openPorts.sh
Open ports on local system:
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
```

Break down `netstat -tuln | grep LISTEN` :

netstat

- **netstat** is a command-line utility that displays network connections, routing tables, and network interface statistics

- It shows active network connections and listening ports on your system

-tuln (these are combined flags)

Breaking down each flag:

- **t**: Show TCP connections only
  - TCP (Transmission Control Protocol) connections
  - Excludes other protocols like UDP-only entries
- **u**: Show UDP connections only
  - UDP (User Datagram Protocol) connections
  - When combined with -t, it shows both TCP and UDP
- **l**: Show only listening ports
  - "Listening" means the port is open and waiting for incoming connections
  - Excludes established connections
- **n**: Show numerical addresses instead of resolving hostnames
  - Displays IP addresses like `127.0.0.1` instead of `localhost`
  - Shows port numbers like `80` instead of `http`
  - Faster execution since it skips DNS lookups

| (pipe operator)

- Takes the output from the `netstat` command
- Feeds it as input to the next command (`grep`)

grep

- **grep** is a text search utility
- Searches for patterns in text and displays matching lines

LISTEN

- This is the search pattern for grep
- It will only show lines containing the word "LISTEN"
- Filters the netstat output to show only listening services

▼ [8.5] Common Built-In Commands

Some useful commands built into BASH:

Command	Function
`:`	Return 0 or true
`.`	Executes a shell script
`bg`	Puts a job in the background
`break`	Exits the current loop
`continue`	Resumes the current loop
`eval`	Evaluates the following expression
`exec`	Executes the following command without creating a new process
`exit`	Quits the shell
`export`	Makes a variable or function available to other programs
`fg`	Brings a job to the foreground (from bg)
`getopts`	Parses arguments to the shell script
`jobs`	Lists background (bg) jobs
`read`	Reads a line from standard input
`readonly`	Declares a variable as read-only
`set`	Lists all variables
`shift`	Moves the parameters to the left
`test`	Evaluates arguments
`[`	performs a conditional test
`times`	Prints the user and system time
`trap`	Traps a Signal
`type`	Displays how each argument would be interpreted

Command	Function
`unmask`	Changes the default permissions for a new file
`unset`	Deletes values from a variable or function
`wait`	Waits for a background process to complete