Linux Shells

Linux Shells

Learned:

- Different shells, such as Bash or Fish, can make running commands easier by offering features like auto-correction and command suggestions.

- You can use variables in Bash to store values and use them in commands.

- Before running a Bash script, you must give it permission to execute using `chmod +x` followed by the script file name.

- To run a script in the current folder, you would use `./scriptname.sh` so the shell knows to look in that directory.

- Loops in Bash scripts let you run a set of commands multiple times automatically.

- You can write scripts that ask for a password or check permissions before executing commands.

Linux has different types of shells available, each with its own features and characteristics.

Multiple shells are installed in different Linux distributions. To see which shell you are using, type the following command:

Current Shell

```
user@tryhackme:~$ echo $SHELL
/bin/bash
```

You can also list down the available shells in your Linux OS. The file `/etc/shells` contains all the installed shells on a Linux system. You can list down the available shells in your Linux OS by typing `cat /etc/shells` in the terminal:

Available Shells

```
user@tryhackme:~$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/bash
```

```
/usr/bin/bash
/bin/rbash
/usr/bin/rbash
/bin/dash
/usr/bin/dash
/usr/bin/tmux
/usr/bin/screen
/bin/zsh
/usr/bin/zsh
```

To switch between these shells, you can type the shell name that is present on your OS, and it will open for you, as can be seen below:

Switch Shell

```
user@tryhackme:~$ zsh
tryhackme%
```

If you want to permanently change your default shell, you can use the command: `chsh -s /usr/bin/zsh`. This will make this shell as the default shell for your terminal.

Bourne Again Shell

Bourne Again Shell (Bash) is the default shell for most Linux distributions. When you open the terminal, bash is present for you to enter commands. Before bash, some shells like sh, ksh, and csh had different capabilities. Bash came as an enhanced replacement for these shells, borrowing capabilities from all of them. This means that it has many of the features of these old shells and some of its unique abilities. Some of the key features provided by bash are listed below:

- Bash is a widely used shell with scripting capabilities.
- It offers a tab completion feature, which means if you are in the middle of completing a command, you can press the tab key on your keyboard. It will automatically complete the command based on a possible match or give you multiple suggestions for completing it.
- Bash keeps a history file and logs all of your commands. You can use the up and down arrow keys to use the previous commands without typing them again. You can also type `history` to display all your previous commands.

Friendly Interactive Shell

Friendly Interactive Shell (Fish) is also not default in most Linux distributions. As its name suggests, it focuses more on user-friendliness than other shells. Some of the key features provided by fish are listed below:

- It offers a very simple syntax, which is feasible for beginner users.

- Unlike bash, it has auto spell correction for the commands you write.
- You can customize the command prompt with some cool themes using fish.
- The syntax highlighting feature of fish colors different parts of a command based on their roles, which can improve the readability of commands. It also helps us to spot errors with their unique colors.
- Fish also provides scripting, tab completion, and command history functionality like the shells mentioned in this task.

Z Shell

Z Shell (Zsh) is not installed by default in most Linux distributions. It is considered a modern shell that combines the functionalities of some previous shells. Some of the key features provided by zsh are listed below:

- Zsh provides advanced tab completion and is also capable of writing scripts.
- Just like fish, it also provides auto spell correction for the commands.
- It offers extensive customization that may make it slower than other shells.
- It also provides tab completion, command history functionality, and several other features.

Feature	Bash	Fish	Zsh
Full Name	The full form of Bash is Bourne Again Shell.	The full form of Fish is Friendly Interactive Shell.	The full form of Zsh is Z Shell.
Scripting	It offers widely compatible scripting with extensive documentation available.	It has limited scripting features as compared to the other two shells.	It offers an excellent level of scripting, combining the traditional capabilities of Bash shell with some extra features.
Tab completion	It has a basic tab completion feature.	It offers advanced tab completion by giving suggestions based on your previous commands.	Its tab completion capability can be extended heavily by using plugins.
Customization	Basic level of customization.	It offers some good customization through interactive tools.	Advanced customization through oh-my-zsh framework.
User friendliness	It is less user-friendly, but being a traditional and widely used shell, its users are quite familiar and comfortable with it.	It is the most user-friendly shell.	It can be highly user-friendly with proper customization.

Feature	Bash	Fish	Zsh
Syntax highlighting	The syntax highlighting feature is not available in this shell.	The syntax highlighting is built-in to this shell.	The syntax highlighting can be used with this shell by introducing some plugins.

A shell script is nothing but a set of commands. Suppose a repetitive task requires you to enter multiple commands using a shell. Instead of entering them one after one on every repetition of that task, which may take more of your time, you can combine them into a script. To execute all those commands, you will only execute the script, and all the commands will be executed. All the shells mentioned in the previous tasks have scripting capabilities. Scripting helps us to automate tasks. Before learning how to write a script, we need to know that even though Linux shells have scripting capabilities, this does not mean that you can only make a script using a shell. Scripting can be done in various programming languages as well. However, the scope of this room is to cover scripting using a shell.

The first step is to open the terminal and select a shell. Let's go with the bash shell, the default, and widely used shell in most distributions.

Unlike the other commands we type in the shell, we first need to create a file using any text editor for the script. The file must be named with an extension `.sh`, the default extension for bash scripts. The following terminal shows the script file creation:

Create Script File

```
user@tryhackme:~$ nano first_script.sh
```

Every script should start from shebang. Shebang is a combination of some characters that are added at the beginning of a script, starting with `#!` followed by the name of the interpreter to use while executing the script. As we are writing our script in bash, let's define it as the interpreter in the shebang.

first_script.sh

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

We are all set to write our first script now. There are some fundamental building blocks of a script that together make an efficient script. Let's learn and utilize these script constructs to write one script ourselves.

Variables

A variable stores a value inside it. Suppose you need to use some complex values, like a URL, a file path, etc., several times in your script. Instead of memorizing and writing them repeatedly, you can store them in a variable and use the variable name wherever you need it.

The script below displays a string on the screen: "Hey, what's your name?" This is done by `echo` command. The second line of the script contains the code `read name`. `read` is used to take input from

the user, and `name` is the variable in which the input would be stored. The last line uses `echo` to display the welcome line for the user, along with its name stored in the variable.

```
# Defining the Interpreter
#!/bin/bash
echo "Hey, what's your name?"
read name
echo "Welcome, $name"
```

Now, save the script by pressing `CTRL+X`. Confirm by pressing `Y` and then `ENTER`.
To execute the script, we first need to make sure that the script has execution permissions. To give these permissions to the script, we can type the following command in our terminal:

Execution Permission to Script

```
user@tryhackme:~$ chmod +x first_script.sh
```

Now that the script has execution permissions use `./` before the script name to execute it. We use `./` before the script to run rather than typing the script name directly because `./` tells the shell to execute the file that is present in the current directory.

Script Execution

```
user@ubuntu:~$ ./first_script.sh
Hey, What's your name?
John
Welcome, John
```

Loops

Loop, as the name suggests, is something that is repeating.

For a general explanation of loops, let's write a loop that will display all numbers starting from 1 to 10 on the screen. First, create a new file named `loop_script.sh`, then enter the code below. Save your file by pressing `CRTL+X`, then confirm with `y` and then `ENTER`.

```
# Defining the Interpreter
#!/bin/bash
for i in {1..10};
do
echo $i
done
```

The first line has the variable `i` that will iterate from 1 to 10 and execute the below code every time. `do` indicates the start of the loop code, and `done` indicates the end. In between them, the code we want to

execute in the loop is to be written. The for loop will take each number in the brackets and assign it to the variable `i` in each iteration. The `echo $i` will display this variable's value every iteration.

Script Execution

```
user@tryhackme:~$ ./loop_script.sh
1
2
3
```

The output of the above terminal is cut to `3` numbers only for demonstration. However, when executed according to the script's logic, it would display the numbers from `1 to 10`.

Conditional Statements

Conditional statements are an essential part of scripting. They help you execute a specific code only when a condition is satisfied; otherwise, you can execute another code. Suppose you want to make a script that shows the user a secret. However, you want it to be shown to only some users, only to the high-authority user. You will create a conditional statement that will first ask the user their name, and if that name matches the high authority user's name, it will display the secret.

First, create a new file named `conditional_script.sh`, then enter the code below. Save your file by pressing `CRTL+X`, then confirm with `y` and then `ENTER`.

```
# Defining the Interpreter
#!/bin/bash
echo "Please enter your name first:"
read name
if [ "$name" = "Stewart" ]; then
        echo "Welcome Stewart! Here is the secret: THM_Script"
else
        echo "Sorry! You are not authorized to access the secret."
fi
```

The above script takes the user's name as input and stores it into a variable (studied in the Variables section). The conditional statement starts with if and compares the value of that variable with the string Stewart; if it's a match, it will display the secret to the user, or else it will not. The fi is used to end the condition.

Following is the terminal showing the script execution when the user name matches the authorized one defined in the script:

conditional_script.sh

```
user@tryhackme:~$ ./conditional_script.sh
Please enter your name first:
```

```
Stewart
Welcome, Stewart! Here is the secret: THM_Script
```

However, the following terminal shows the script execution when the user name does not match the authorized one defined in the script:

conditional_script.sh

```
user@tryhackme:~$ ./conditional_script.sh
Please enter your name first:
Alex
Sorry! You are not authorized to access the secret.
```

Comments

Sometimes, the code can be very lengthy. In this scenario, the code might confuse you when you look at it after some time or share it with somebody. An easy way to resolve this problem is to use comments in different parts of the code. A comment is a sentence that we write in our code just for the sake of our understanding. It is written with a # sign followed by a space and the sentence you need to write.

```
# Defining the Interpreter
#!/bin/bash

# Asking the user to enter a value.
echo "Please enter your name first:"

# Storing the user input value in a variable.
read name

# Checking if the name the user entered is equal to our required name.
if [ "$name" = "Stewart" ]; then

# If it equals the required name, the following line will be displayed.
echo "Welcome Stewart! Here is the secret: THM_Script"

# Defining the sentence to be displayed if the condition fails.
else
        echo "Sorry! You are not authorized to access the secret."
fi
```

The Locker Script

A user has a locker in a bank. To secure the locker, we have to have a script in place that verifies the user before opening it. When executed, the script should ask the user for their name, company name,

and PIN. If the user enters the following details, they should be allowed to enter, or else they should be denied access.

- Username: John
- Company name: Tryhackme
- PIN: 7385

Script

```
# Defining the Interpreter
#!/bin/bash

# Defining the variables
username=""
```

```
companyname=""
pin=""

# Defining the loop
for i in {1..3}; do
# Defining the conditional statements
        if [ "$i" -eq 1 ]; then
                echo "Enter your Username:"
                read username
        elif [ "$i" -eq 2 ]; then
                echo "Enter your Company name:"
                read companyname
        else
                echo "Enter your PIN:"
                read pin
        fi
done

# Checking if the user entered the correct details
if [ "$username" = "John" ] && [ "$companyname" = "Tryhackme" ] && [ "$pin" =
"7385" ]; then
        echo "Authentication Successful. You can now access your locker, John."
else
        echo "Authentication Denied!!"
fi
```

Script Execution

Executing the Locker Script

```
user@tryhackme:~$ ./locker_script.sh
Enter your Username:
John
Enter your Company name:
Tryhackme
Enter your PIN:
1349
Authentication Denied!!
```