



# Cyber Security Tools – Bash Scripting Task

[Visit our website](#)

# Introduction

Scripting plays a critical role in modern cybersecurity because it allows professionals to respond quickly, work efficiently, and automate tasks that would otherwise require significant manual effort. Cybersecurity operations involve continuous monitoring, rapid investigation, and repeated execution of technical tasks. Without scripting, these activities can become slow, inconsistent, and error-prone. When security teams write scripts, they gain the ability to standardise processes, reduce human mistakes, and perform complex actions with a high degree of precision.

Scripting is not about replacing full programming languages. Instead, it provides a practical, lightweight way to automate system-level tasks directly on the command line. This makes Bash scripting especially valuable because it interacts closely with Linux, which is the foundation for many security tools, penetration testing frameworks, and server environments used in industry.

## Automation in Cybersecurity Operations

Automation is essential in environments where threats evolve quickly and defenders must act at speed. A single security analyst might need to process logs, scan networks, check system configurations, and respond to incidents simultaneously. Performing these tasks manually not only slows down detection but can also allow attackers more time to cause damage. Bash scripts allow analysts to automate repetitive workflows such as log parsing, file integrity checks, permission audits, user account monitoring, and scheduled security scans.

For example, a Bash script can automatically review system logs every hour, extract suspicious authentication attempts, and send an alert if patterns indicate brute-force activity. Instead of relying on a human to perform these checks, automation ensures that surveillance continues 24 hours a day, significantly improving detection speed. Similarly, system administrators can use scripts to update packages, deploy security patches across multiple servers, or run scheduled backups. This reduces the likelihood of forgotten updates or inconsistent configuration, which are common causes of security breaches.

## Efficiency During Incident Response

During an incident, time is one of the most important factors. Analysts often need to collect system information, isolate compromised machines, or extract indicators of compromise from multiple directories. With scripting, these actions can be performed instantly and accurately. A well-written script can gather system logs, check active network connections, identify unusual processes, and package all relevant data for

analysis in a single command. This reduces the time required to investigate an incident and allows defenders to act before an attacker can escalate their access.

Automation also helps security teams scale their response during large attacks. If dozens of systems are affected simultaneously, scripts can run the same investigation or containment action across all machines without requiring individual manual intervention. This ensures a faster, more consistent response that aligns with industry best practices.

## Supporting Security Testing and Ethical Hacking

Cybersecurity practitioners frequently use tools in Kali Linux to test the strength of systems and identify vulnerabilities. Scripting enhances these activities by allowing testers to automate scanning routines, chain commands together, and generate repeatable testing workflows. For example, instead of manually running Nmap, parsing the results, and performing follow-up enumeration, a Bash script can combine these steps into a single automated process. This produces more reliable outcomes and helps testers focus on analysing results rather than repeating the same manual work.

Scripting also enables the creation of custom tools tailored to specific testing scenarios. When existing tools do not provide the exact functionality needed, Bash allows testers to build lightweight utilities that extend their capabilities. This flexibility is one of the reasons scripting knowledge is considered essential for ethical hacking, penetration testing, and digital forensics.

## Enhancing System Hardening and Compliance

Many organisations depend on scripting to maintain secure configurations across their systems. Scripts can enforce policies such as file permissions, password settings, firewall rules, and logging requirements. When used correctly, they help ensure that systems remain aligned with frameworks like CIS Benchmarks, NIST controls, and ISO/IEC 27001 requirements.

For example, a Bash script can check whether unnecessary services are running, confirm that audit logs are active, verify correct file permissions on critical directories, and report any deviations from expected configurations. This automated approach to system hardening reduces risk and supports continuous compliance.

## Scripting as a Core Skill for Cybersecurity Careers

In the cybersecurity industry, Bash scripting is considered a foundational skill because it demonstrates the ability to work effectively with Linux, automate workflows, and solve operational challenges efficiently. Whether specialising in security operations,

penetration testing, digital forensics, system administration, or DevSecOps, scripting enables professionals to adapt to new tools, integrate defensive measures, and build repeatable security processes.

Learning Bash scripting early provides a strong advantage, empowering learners to experiment with automation, understand how operating systems behave, and develop the problem-solving mindset required for advanced cybersecurity tasks. As environments grow more complex and attackers become more sophisticated, scripting becomes not just a convenience but a necessary capability for maintaining strong and agile security operations.

## Bash

Bourne-Again SHell, or Bash, scripts tell your Bash shell what commands it should execute. Scripting can be really powerful if you have a series of commands to execute, and scripting with these typical command-line commands provides you with some useful automation. They're a great tool for merging programs into more intelligent and elegant solutions. In this task, we'll review some basic Bash syntax so that by the end you'll be able to write simple programs using only the Bash shell that comes with most Linux distributions.

Bash is the default shell for many Linux distributions. It provides you with an interface to run commands that interact with the kernel so that you can perform simple to advanced tasks faster than you may be able to with a GUI (if there is a GUI available for those tasks).

Bash scripts are files that are executed line by line, where each line is a command. That's all there is to it!

## Shebangs

Every file in a Unix file system has a set of permissions. One of these permissions is the execute permission. If you want to make a file executable from the terminal, you will need to give it the execute permission by running this command to **change** the **mode** to executable:

```
$ chmod +x [file name]
```

Now that you have an executable file, you need to give the shell interpreter a heads-up of what to run the file with. This is done in the first line of the file with a special line called a "shebang".

If you were writing a Bash script, your shebang would be:

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

When you execute your script, the shell will know that the script must be run with the program you've specified in the shebang, i.e., Bash.

If we wanted to do this with a Python script and have Python run the script rather than Bash, we would first need to know where Python is installed on our system. We could do this using the intuitive command `where python`. Python will probably be located at **/usr/bin/python3**. In this case, our shebang will be: `#!/usr/bin/python3`

## Text editors for bash scripting

When writing Bash scripts on Kali Linux, you can use various text editors. Two common editors are **Nano** and **Vi**.

### Nano

Nano is a simple, user-friendly text editor. Here's how to use it:

1. Open a terminal: Press "Ctrl + Alt + T" or find it in your applications.
2. Create or edit a file: Type `nano filename.sh` to create or edit **filename.sh**.
3. Write your script: Type your Bash script into the editor.
4. Save and exit:
  - Press "Ctrl + O" to save the file.
  - Press "Enter" to confirm.
  - Press "Ctrl + X" to exit Nano.

### Vi

Vi (or Vim) is a more powerful editor but it has a steeper learning curve. Here's a quick guide:

1. Open a terminal: Press "Ctrl + Alt + T" or find it in your applications.
2. Create or edit a file: Type `vi filename.sh` to create or edit **filename.sh**.
3. Switch to insert mode: Press "i" to switch to insert mode and start typing your script.
4. Save and exit:
  - Press "Esc" to exit insert mode.
  - Type `:wq` and press "Enter" to save and exit.

By default, Kali Linux comes with both Nano and Vi installed, so you can choose whichever editor you are most comfortable with.

## File extensions and execution

When writing Bash scripts, it's common to use the **.sh** extension. This extension helps to identify the file as a shell script, making it easier for users and system administrators to recognise its purpose.

Here's a quick reminder of how to make your script executable and to run it:

1. Give execute permission:

```
chmod +x filename.sh
```

2. Run the script:

```
./filename.sh
```

The `./` indicates that the script is located in the current directory. If the script is located in a different directory, you would need to provide the relative or absolute path to the file.

## Variables

In Bash, which is a command-line language used in many Unix-like systems, variables are "weakly typed." This means you don't need to specify what type of data a variable holds (such as numbers or text). Instead, you can just use variables without worrying about their type.

To declare a variable in a Bash script, you use a specific format. You write the name of the variable, followed by an equals sign, and then the value you want to assign to it. For example, if you want to create a variable named `fact` and store the text "Linux is awesome!" in it, you would write:

```
fact="Linux is awesome!"
```

In this statement, `fact` is the name of the variable, and "Linux is awesome!" is the value assigned to it.

When you want to use or display the value of this variable in your script, you refer to it by placing a dollar sign `$` before the variable name. For instance, to print the value stored in the `fact` variable, you would write:

```
echo "Fact:" $fact
```

The `echo` command is used to display text on the screen. In this case, "Fact:" is printed exactly as it appears. The `$fact` portion instructs Bash to substitute the value stored in the `fact` variable into the output.

For instance, if you have a script named **variables.sh** with the following content:

```
#!/bin/bash
fact="Linux is awesome!"
echo "Fact:" $fact
```

The output of running this script will be:

```
vin@localhost:~/Documents/bash scripting>./variables.sh
Fact: Linux is awesome!
vin@localhost:~/Documents/bash scripting>
```

If you want to assign user input to a variable, you can use `read variable_name`:

```
#!/bin/bash

echo "Type in a fact: "
read fact
echo "Fact:" $fact
```

Output:

```
vin@localhost:~/Documents/bash scripting>./variables.sh
Type in a fact:
Linux is awesome!
Fact: Linux is awesome!
vin@localhost:~/Documents/bash scripting>
```

## Conditional statements

Conditional blocks in Bash scripting can take one of four formats:

1. **If blocks:**

```
if [[ condition ]]; then
    statement
fi
```

2. **If/else blocks:**

```
if [[ condition ]]; then
    statement
else
    default
fi
```

### 3. If/elif/else blocks:

```
if [[ condition ]]; then
    statement
elif [[ condition ]]; then
    statement
else
    default
fi
```

### 4. Nested if/else blocks:

```
if [[ condition ]]; then
    statement
else
    if [[ condition ]]; then
        statement
    else
        statement
    fi
fi
```

In these examples, `fi` serves as the ending delimiter for the `if` statement, indicating the conclusion of the `if` block. Recalling the overall structure of an `if` block from above:

```
if [[ condition ]]; then
    statement
fi
```

- `if` initiates the conditional block.
- `then` introduces the statement to be executed if the condition is true.
- `fi` closes the `if` statement.

Within the condition, we can use the following operators:

- `-gt` to represent “greater than”.
- `-lt` to represent “less than”.
- `==` to represent “equality”.
- `-le` to represent “less than or equal to”

To explore additional comparison operators and their uses, you can learn more in [this comprehensive guide](#).

You can combine multiple conditions using:

- The AND operator, `-a`, which requires both conditions to be true.
- The OR operator, `-o`, which requires at least one of the conditions to be true.

Let's look at a simple example program written as a Bash script. If we wanted to write a program that checked if someone was a child, teenager, or adult based on their age, the program would look as follows:

```
#!/bin/bash

# Prompt the user to enter their age
echo "How old are you?"
read age

# Check if the age is less than or equal to 13
if [ $age -le 13 ]; then
    echo "You are a child."
else
    # Check if the age is greater than 13 and less than or equal to 18
    if [ $age -gt 13 -a $age -le 18 ]; then
        echo "You are a teenager."
    else
        # If neither of the above conditions is true, the person is an adult
        echo "You are an adult."
    fi
fi
```

Example output:

```
vin@localhost:~/Documents/bash scripting> ./conditions.sh
How old are you?
15
You are a teenager.
vin@localhost:~/Documents/bash scripting>
```

We can rewrite this program to use `if/elif/else` notation as well, and it would be functionally identical to the version above:

```
#!/bin/bash

# Prompt the user to enter their age
echo "How old are you?"

# Read the user's input and store it in the variable "age"
read age

# Check if the age is less than or equal to 13
if [ $age -le 13 ]; then
    # If the condition is true, print "You are a child."
    echo "You are a child."

# Check if the age is greater than 13 and less than or equal to 18
elif [ $age -gt 13 -a $age -le 18 ]; then
    # If the condition is true, print "You are a teenager."
    echo "You are a teenager."

# If none of the above conditions are true
else
    # Print "You are an adult."
    echo "You are an adult."

# End of the if-elif-else block
fi
```

Recall that the first line, `#!/bin/bash`, is called a shebang, which specifies the path to the Bash interpreter, ensuring that the script runs using Bash.

The script begins by printing the message “How old are you?” to the terminal using the `echo “How old are you?”` line. The following line, `read age`, waits for user input and assigns the entered value to the variable `age`.

Next, the script checks the value of `age` with an `if` statement: `if [ $age -le 13 ]; then`. This condition evaluates whether `age` is less than or equal to 13 (with `-le` meaning “less than or equal to”). If the condition is true, the script prints “You are a child.” using the `echo “You are a child.”` line.

If the condition is false, the script moves to the `else` block. Within this block, there is a nested `if` statement: `if [ $age -gt 13 -a $age -le 18 ]; then`. This checks if `age` is greater than 13 (with `-gt` meaning “greater than”) and less than or equal to 18. If this condition is true, the script prints “You are a teenager.” using the `echo “You are a teenager.”` line.

If neither of the previous conditions is met, the script moves to the final `else` block and prints “You are an adult.” using the `echo “You are an adult.”` line.

Example execution:

```
vin@localhost:~/Documents/bash scripting>./conditions.sh
How old are you?
15
You are a teenager.
vin@localhost:~/Documents/bash scripting>
```

This example shows how the script prompts for the user's age and prints the appropriate message based on the input provided.

## Iterations

You're already familiar with the basic syntax of iterations. Iterations in Bash don't look too different! Loops that run a set number of times are defined as follows:

```
for i in {1..5}; do
    [statement]
done
```

If you ever find yourself in a situation where you need to do one thing to a lot of things all at once (say, change the permissions of a number of files all at once), loops are particularly useful.

## Arrays

Bash gives you access to one-dimensional, associative arrays. To declare an array of strings, you would type:

```
my_array=("object" "item" "thing")
```

If you want to loop through the items in the array, you can do the following:

```
for str in ${my_array[@]}; do
    echo $str
done
```

This prints everything in the loop to the standard output (stdout). It accesses the array elements the same way a Python loop would.

Building on the example above, if you had a large list of file-system objects, you could put them all inside an array, and use an iteration loop to go through them one by one. Helpful, right?



## Take note

The task(s) below is/are **auto-graded**. An auto-graded task still counts towards your progression and graduation. Give it your best attempt and submit it when you are ready.

When you select “Request Review”, the task is automatically complete, you do not need to wait for it to be reviewed by a mentor.

You will then receive an email with a link to a model answer, as well as an overview of the approach taken to reach this answer.

Take some time to review and compare your work against the model answer. This exercise will help solidify your understanding and provide an opportunity for reflection on how to apply these concepts in future projects.

In the same email, you will also receive a link to a survey, which you can use to self-assess your submission.

Once you’ve done that, feel free to progress to the next task.



## Auto-graded task

In this task, you’ll be creating two different scripts.

1. Write a Bash script called **change\_permissions.sh** that changes the permissions of all the objects in a folder to `-rw-r--r--`.
2. Linux users must become adept at finding and learning how to use new commands by using **man pages** and other resources. Review the man page for the **apt** command. Support your understanding with research via web search if necessary. Then, write a Bash script called **manage\_apt.sh** that does the following with **apt**:
  - Uninstalls all unused dependencies.
  - Updates the software database.
  - Updates the entire system.

**Note:** For the second script, you may need to check which account is running the command and have your script notify the user accordingly to avoid any permission errors.

**Important:** Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.



## Share your thoughts

Please take some time to complete this short feedback [form](#) to help us ensure we provide you with the best possible learning experience.

---