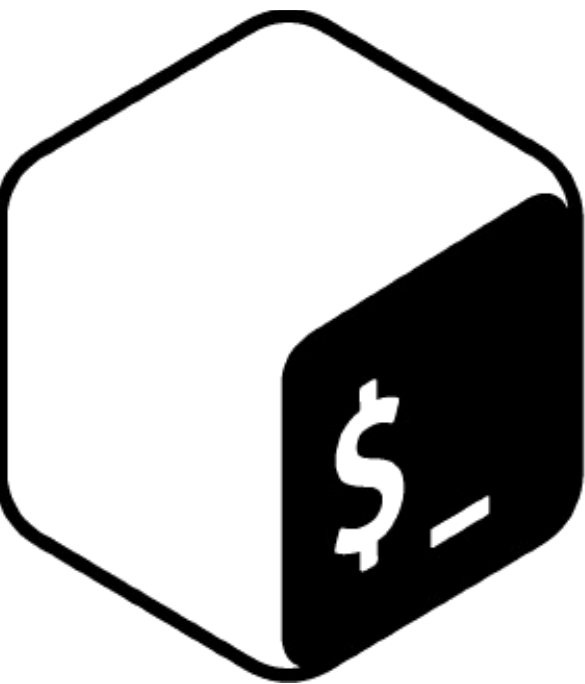


Bash Scripting for Penetration Testing

1. Introduction to Bash Scripting

What is Bash?

- **Bash** (Bourne Again Shell) is a command language interpreter used in Linux/Unix systems.
- You can write Bash scripts to automate tasks like backups, updates, server checks, and more.



BASH
THE BOURNE-AGAIN SHELL

Creating a Bash Script

- The file should start with a **shebang** line: `#!/bin/bash`
- Save your file with a **.sh** extension and make it executable: `chmod +x myscript.sh`

2. Ping

We'll name our initial script *pingscript.sh*. This script will execute a **ping sweep** across our local network by sending **ICMP (Internet Control Message Protocol)** echo requests to remote devices and logging their responses.

While the ping utility is our primary tool for identifying active hosts, note that some systems may be configured to **ignore ICMP requests**, meaning a non-response doesn't always indicate an offline device. Despite this limitation, a ping sweep remains a practical first step in network reconnaissance.

By default, the script requires either an **IP address** or a **hostname** as input for the ping command.

Purpose: Used to check if a host is reachable.

Syntax:

```
File Actions Edit View Help
(sg-learning@SG)-[~]
$ ping -c 4 google.com
PING google.com (142.250.203.238) 56(84) bytes of data.
64 bytes from mrs08s21-in-f14.1e100.net (142.250.203.238): icmp_seq=1 ttl=117 time=47.8 ms
64 bytes from mrs08s21-in-f14.1e100.net (142.250.203.238): icmp_seq=2 ttl=117 time=44.1 ms
64 bytes from mrs08s21-in-f14.1e100.net (142.250.203.238): icmp_seq=3 ttl=117 time=51.3 ms
64 bytes from mrs08s21-in-f14.1e100.net (142.250.203.238): icmp_seq=4 ttl=117 time=44.9 ms

--- google.com ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3007ms
rtt min/avg/max/mdev = 44.130/47.047/51.326/2.827 ms
```

-c 4 → Sends 4 ping packets.

3. A Simple Bash Script

We will now create a basic Bash script to perform host discovery through ICMP pings. As a best practice, we'll first implement a help function to ensure proper script usage. This user guidance should include:

- Command syntax
- Required parameters
- Example invocations

The help documentation serves two critical purposes:

1. It provides immediate assistance to users
2. It establishes script maintainability standards

```
sg-learning@SG: ~$ nano pingscript.sh
GNU nano 8.0 pingscript.sh *
#!/bin/bash
echo "Usage: ./pingscript.sh [network]"
echo "Example: ./pingscript 192.168.1.150"
```

The script begins with the shebang (`#!/bin/bash`), which explicitly declares the Bash interpreter as our execution environment. This ensures consistent behavior across Unix-like systems.

Following the interpreter declaration, we implement user communication through echo statements. These serve two critical functions:

1. **User Guidance:** Informs the operator about the script's expected input
2. **Input Validation:** Prepares for the network parameter that will drive the ping sweep functionality

For example, when executing: `./pingscript.sh 192.168.1.0/24`

The echo outputs will:

- Confirm the target network specification
- Provide visual confirmation before sweep execution

After creating the script, use `chmod` to make it executable so we can run it.

```
(sg-learning@SG)-[~]
$ chmod 744 pingscript.sh
```

Running Our Script

When executing commands in Linux, the system relies on the `PATH` environment variable to locate executables. This includes both:

- Native Linux utilities (e.g., `ls`, `cat`)
- Pentesting tools installed in Kali Linux

The `PATH` variable defines an ordered search path of directories where the system looks for executable files when you enter a command without an absolute path.

To view your current `PATH` configuration:

```
(sg-learning@SG)-[~]
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/usr/games
```

Notice that the `/root` directory is not included in the system's executable path. As a result, we cannot run our Bash script by simply entering `pingscript.sh`. Instead, we need to explicitly specify the current directory using `./pingscript.sh`. When executed this way, the script displays the usage information, as shown below.

```
sg-learning@SG: ~ x sg-learning@SG: ~ x
(sg-learning@SG)-[~]
$ ./pingscript.sh
Usage: ./pingscript.sh [network]
Example: ./pingscript 192.168.1.150
```

Adding Functionality with if Statements

Now let's add in a bit more functionality with an `if` statement

```
GNU nano 8.0 pingscript.sh
#!/bin/bash
if [ "$1" = "" ]
then
echo "Usage: ./pingscript.sh [network]"
echo "Example: ./pingscript 192.168.1.150"
fi
```

In most cases, a script should display usage information only when it is executed incorrectly. For example, our script requires the user to provide a network address as a command-line argument. If this argument is not supplied, we want to notify the user with clear instructions on how to use the script properly.

To implement this behavior, we can use an `if` statement to check whether the required input has been provided. The `if` statement allows us to control the script's behavior based on specific conditions—such as detecting when no command-line argument is given.

The if construct is common across many programming languages, although the syntax may vary. In Bash scripting, an if statement is typically written as `if [condition]`, where condition represents the logic that must evaluate to true for the block to execute.

In our script, we begin by checking whether the first command-line argument is empty. In Bash scripting, `$1` refers to the first argument passed to the script. We use the double equals sign (`==`) to compare values for equality.

Following the if statement, we use the then keyword to define the block of commands that should execute if the condition is true. This block continues until it reaches the fi keyword (which signifies the end of the if statement—"if" spelled backwards).

If the script is executed without providing any command-line argument, the condition evaluates to true because `$1` is indeed empty. In this case, the script executes the commands within the then block, typically displaying usage information or an error message to guide the user.

A terminal window with a dark background. The prompt is `sg-learning@SG: ~`. The user has entered `./pingscript.sh`. The output shows the script's usage and an example: `Usage: ./pingscript.sh [network]` and `Example: ./pingscript.sh 192.168.1.150`.

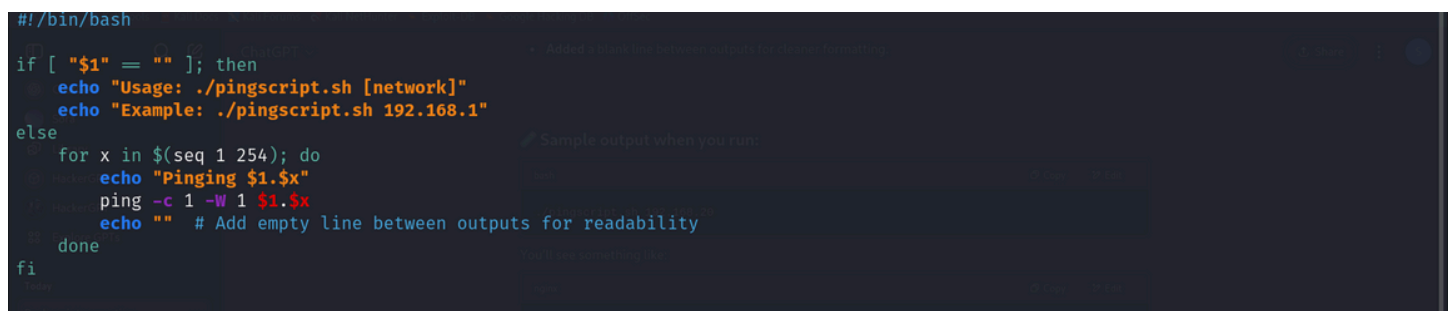
```
sg-learning@SG: ~  
$ ./pingscript.sh  
Usage: ./pingscript.sh [network]  
Example: ./pingscript.sh 192.168.1.150
```

Implementing a for Loop

At this stage, running the script with a valid command-line argument results in no visible output. To make the script functional, we need to add logic that executes when the user provides the required input.

We can enhance the script by incorporating a for loop, which will allow us to iterate over a range of IP addresses or hostnames, performing a specific action, such as pinging each address, when the user runs the script with the correct arguments.

This addition ensures the script responds appropriately when used as intended and performs the necessary operations based on the supplied input.

A code editor window showing a Bash script. The script uses an if statement to check for arguments. If none are provided, it shows usage. If provided, it uses a for loop to ping a range of IP addresses from 1 to 254. Comments are included for readability.

```
#!/bin/bash  
  
if [ "$1" = "" ]; then  
    echo "Usage: ./pingscript.sh [network]"  
    echo "Example: ./pingscript.sh 192.168.1"  
else  
    for x in $(seq 1 254); do  
        echo "Pinging $1.$x"  
        ping -c 1 -W 1 $1.$x  
        echo "" # Add empty line between outputs for readability  
    done  
fi
```

Following the then block, we introduce an else clause to define the behavior when the if condition evaluates as false—in other words, when the user provides a valid command-line argument. In this case, the script should proceed to perform its intended functionality.

Since the goal of the script is to ping all possible hosts within a local network, we need to iterate through the range of potential host addresses. For an IPv4 address, the final octet typically ranges from 1 to 254. To efficiently loop through this sequence, we use a for loop.

The loop structure `for x in \seq 1 254; do` instructs the script to execute the following block of code once for each value between 1 and 254. This approach enables us to perform the same operation—such as `apingcommand`—on each IP address without manually writing code for every instance. The loop concludes with the `done` keyword, which signifies the end of the loop block.

Within the `for` loop, our goal is to ping each IP address in the specified network range. Referring to the ping command's manual (`man page`), we find that the `-c` option allows us to limit the number of echo requests sent. By setting `-c 1`, we ensure that each host is pinged only once, making the scan more efficient.

To target the correct IP addresses, we concatenate the first command-line argument—which represents the first three octets of the IP address (e.g., `192.168.1`)—with the current value of the loop variable. This allows us to dynamically generate the full IP address for each iteration.

`ping -c 1 $1.$x`

Here:

- `$1` represents the first command-line argument (the base IP prefix).
- `$x` is the current number in the loop, ranging from 1 to 254.

As the loop executes, the script constructs and pings addresses such as `192.168.1`, `192.168.1.50`, and continues through to `192.168.1.254`. Once all iterations are complete, the loop ends and the script finishes execution.

When the script is run with the first three octets of the local network passed as an argument, it systematically pings each host in that subnet, as demonstrated below

```
(sg-learning@SG)-[~]
$ ./pingscript.sh 192.168.1
Pinging 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data:
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.61 ms

--- 192.168.1.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.606/1.606/1.606/0.000 ms

Pinging 192.168.1.2
PING 192.168.1.2 (192.168.1.2) 56(84) bytes of data.

--- 192.168.1.2 ping statistics ---
1 packets transmitted, 0 received, 100% packet loss, time 0ms

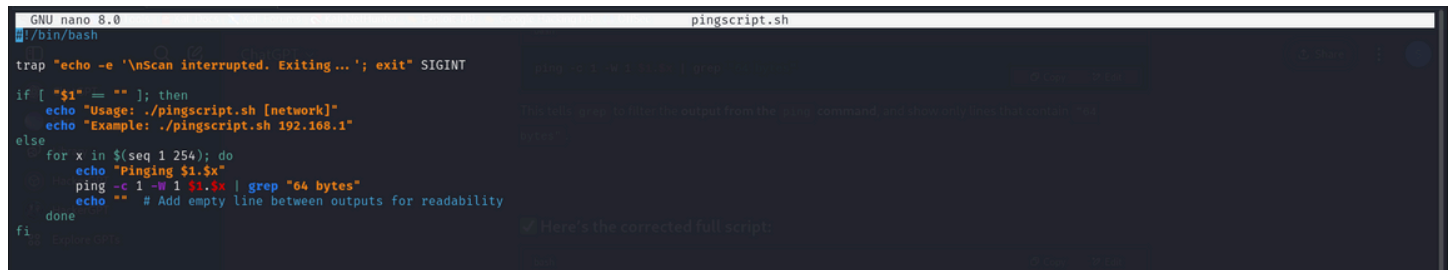
Pinging 192.168.1.3
PING 192.168.1.3 (192.168.1.3) 56(84) bytes of data.
```

Your results may differ depending on the devices currently active on your local network. In the sample output shown, we can observe that the host `192.168.20.1` is online, as indicated by a successful ICMP echo reply. Conversely, the host `192.168.1.50` appears to be offline or unreachable, as the response indicates a "host unreachable" message.

Streamlining the Results

The raw output currently displayed by the script is verbose and difficult to interpret at a glance. Users would have to manually sift through a large amount of information to determine which hosts on the

network are active. To improve usability and readability, we can enhance the script by adding functionality that filters and presents the results in a cleaner, more streamlined format.



```
GNU nano 8.0 pingscript.sh
#!/bin/bash

trap "echo -e '\nScan interrupted. Exiting...'; exit" SIGINT

if [ "$1" = "" ]; then
    echo "Usage: ./pingscript.sh [network]"
    echo "Example: ./pingscript.sh 192.168.1"
else
    for x in $(seq 1 254); do
        echo "Pinging $1.$x"
        ping -c 1 -W 1 $1.$x | grep "64 bytes"
        echo "" # Add empty line between outputs for readability
    done
fi
```

To streamline the output, we filter the results to display only the lines that contain the string "64 bytes", which appears when an ICMP echo reply is successfully received from a host. This indicates that the host is active and responsive.

By incorporating this filter into our script, the output becomes much cleaner, only displaying the relevant lines that confirm which hosts are up. When the updated script is executed, you'll notice that only the responses containing "64 bytes" are printed to the screen, as demonstrated below.



```
(sg-learning@SG)-[~]
$ ./pingscript.sh 192.168.1
Pinging 192.168.1.1
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.78 ms
```

We get indicators only for live hosts; hosts that do not answer are not printed to the screen. But we can make this script even nicer to work with. The point of our ping sweep is to get a list of live hosts. By using the cut command discussed. We can print the IP addresses of only the live hosts, as shown in the terminal.



```
GNU nano 8.0 pingscript.sh
#!/bin/bash

if [ "$1" = "" ]; then
    then
        echo "Usage: ./pingscript.sh [network]"
        echo "Example: ./pingscript.sh 192.168.1,50"
    else
        for x in $(seq 1 254); do
            if ping -c 1 -W 1 $1.$x | grep -q "64 bytes"; then
                echo "$1.$x:"
            fi
        done
    fi
fi
```

We can use a space as the delimiter and grab the fourth field, our IP address, as shown at u



```
(sg-learning@SG)-[~]
$ ./pingscript.sh 192.168.1
192.168.1.1:
```

