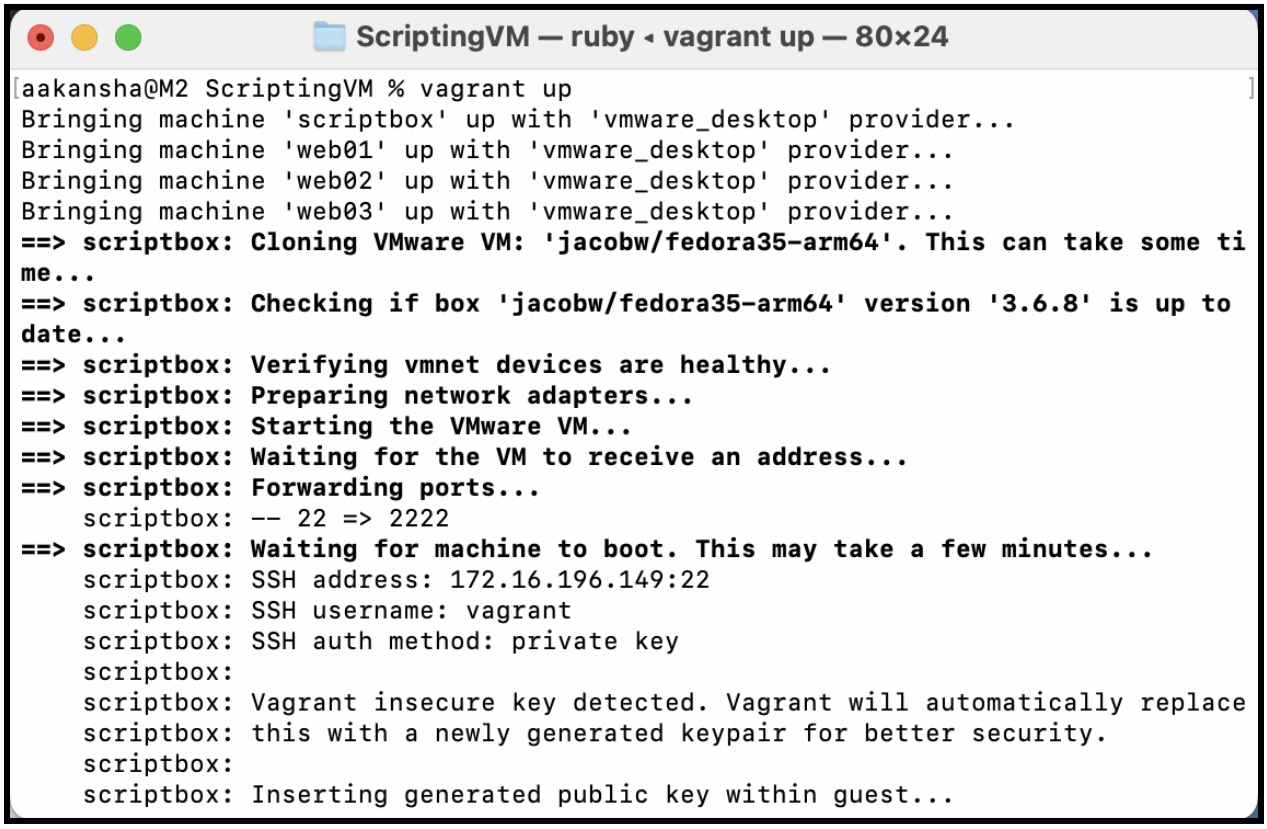
BASH SCRIPTING

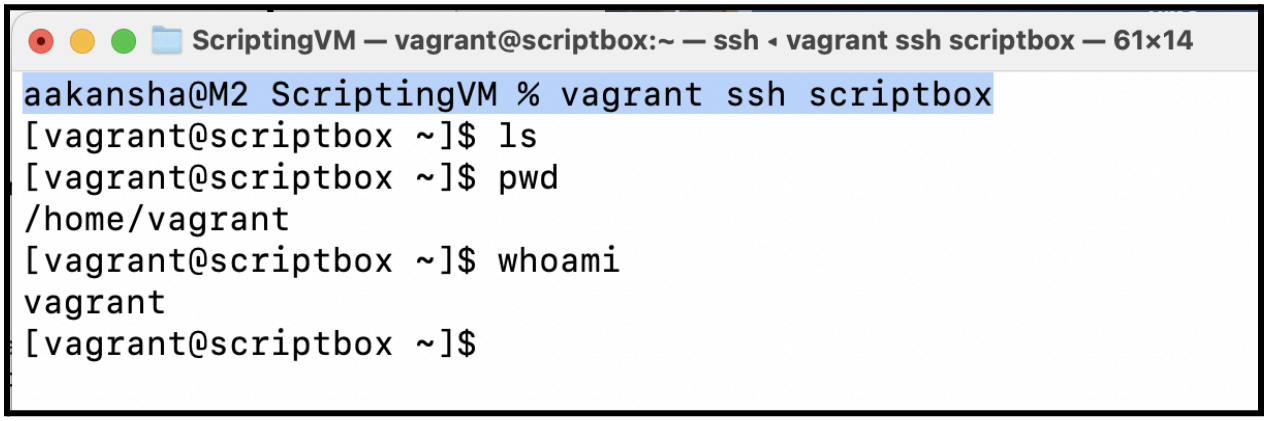
Bring up the VMs

aakansha@M2 ScriptingVM % vagrant up



#Enter into scriptbox vm

aakansha@M2 ScriptingVM % vagrant ssh scriptbox



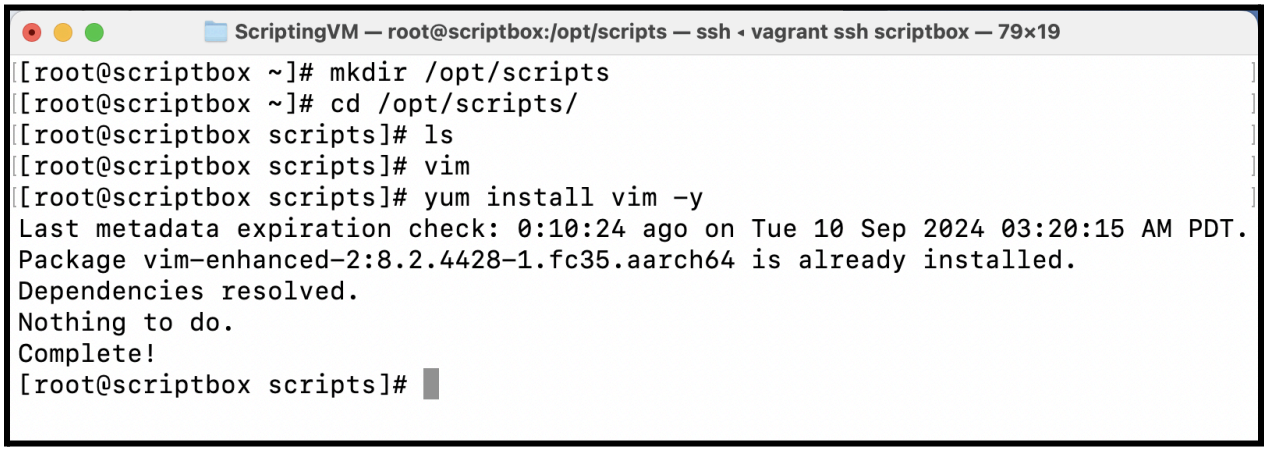
#Become root

[vagrant@scriptbox ~]$ sudo -i

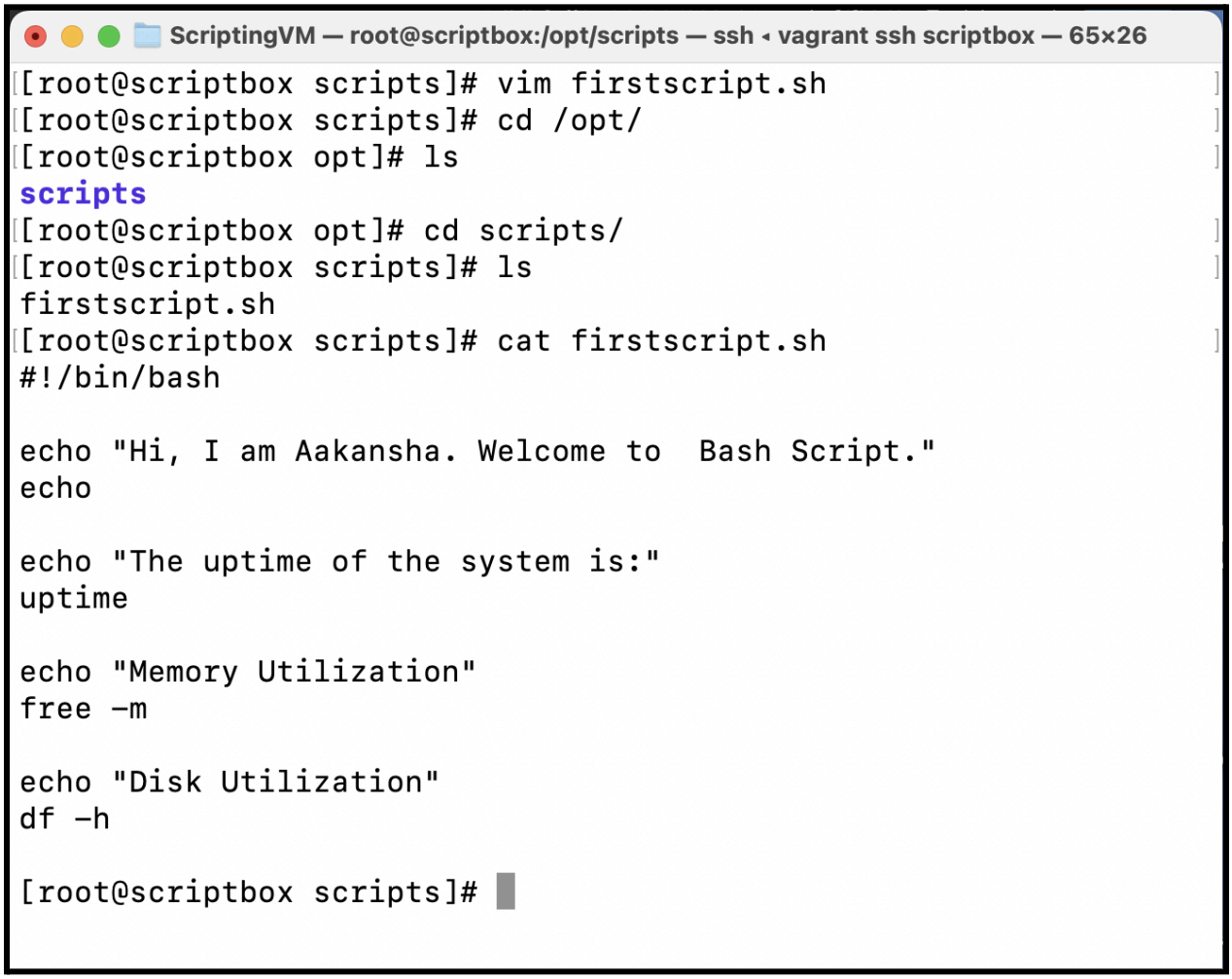
#Writing the first script

#Create a new folder and cd into that

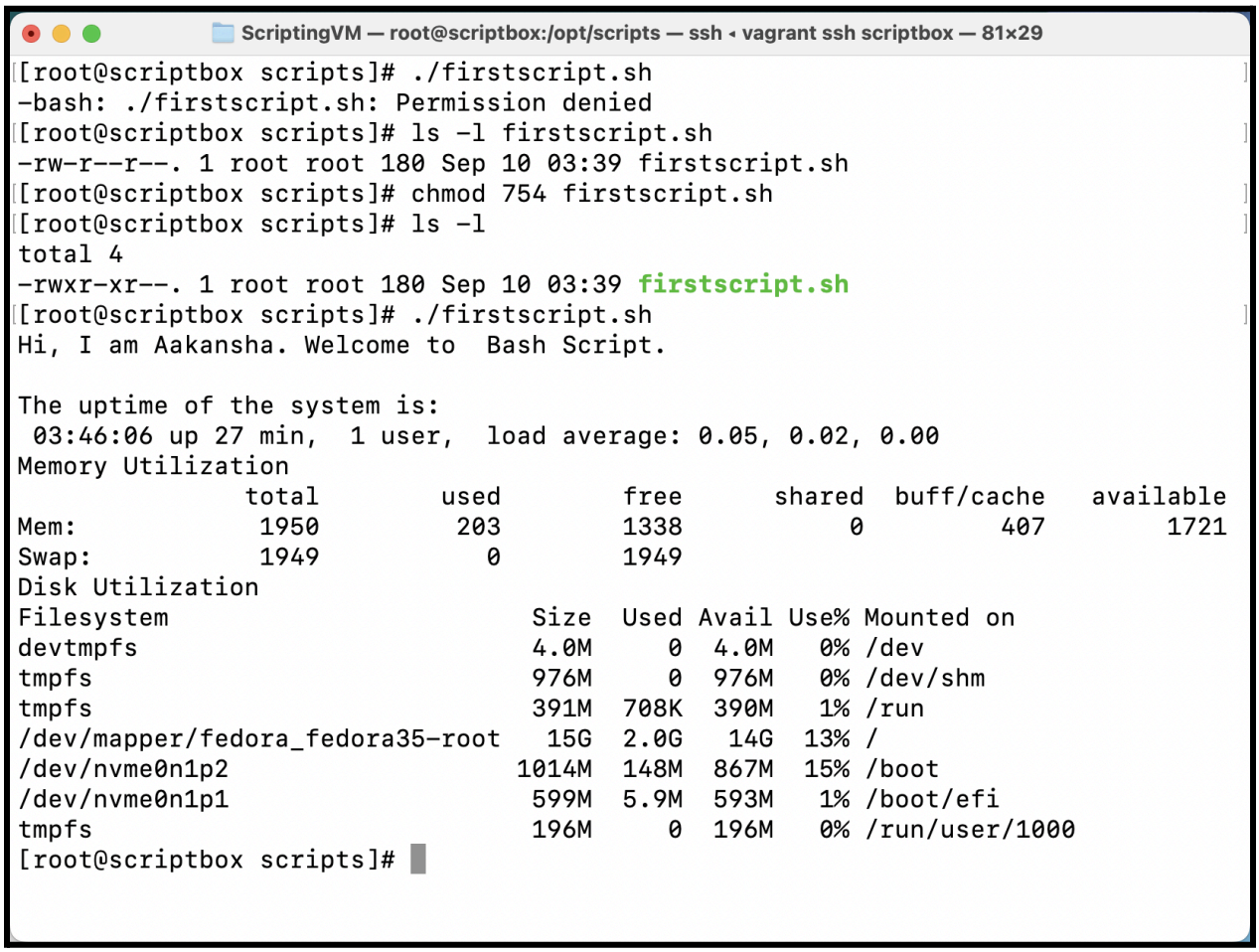
#Check for vim editor



 # Write the script : A script to print system info



#Execute the script



Commands:

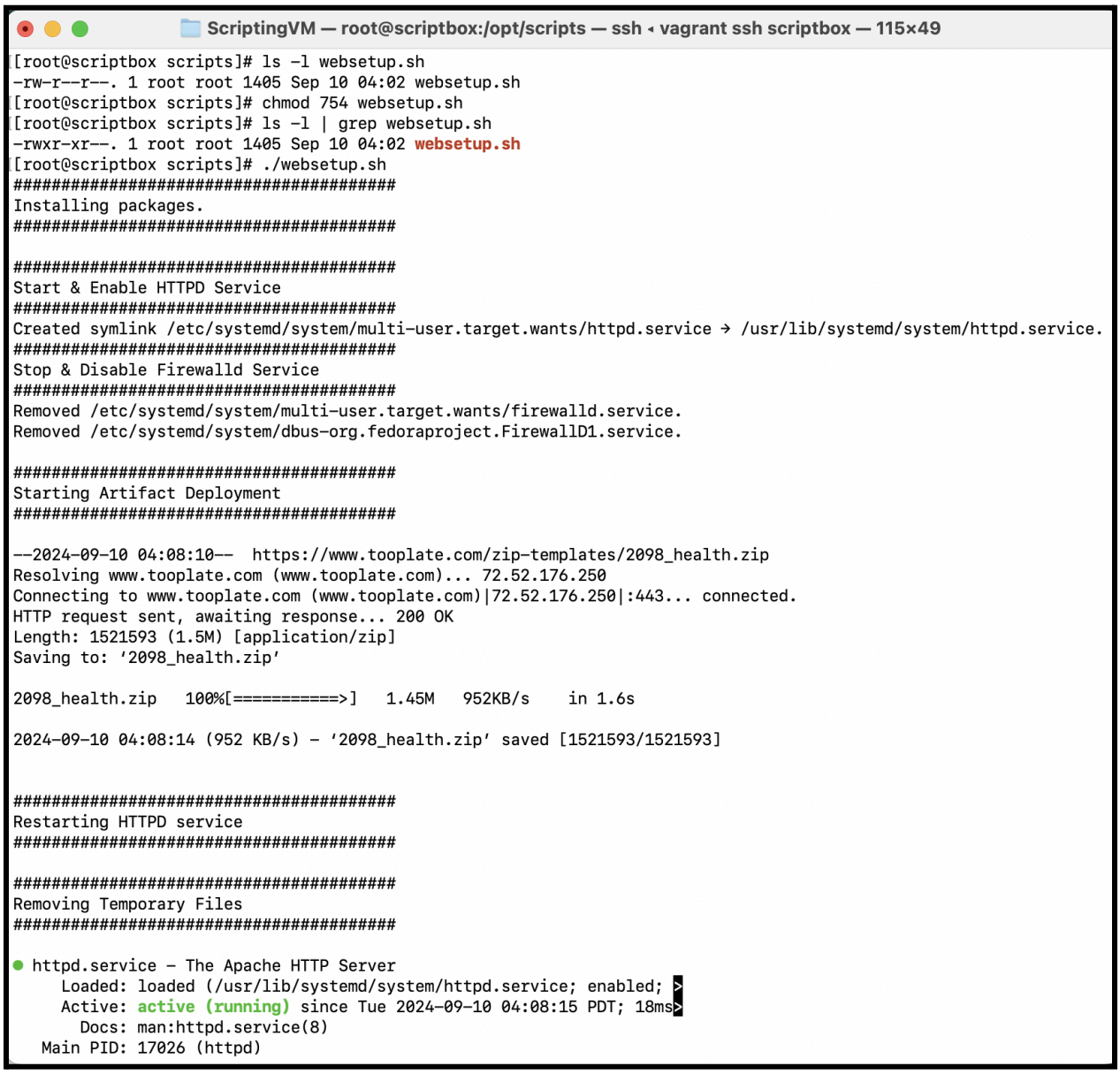
```
[root@scriptbox scripts]# history
    1  cat /etc/hostname
    2  clear
    3  mkdir /opt/scripts
    4  cd /opt/scripts/
    5  ls
    6  vim
    7  yum install vim -y
    8  clear
    9  vim firstscript.sh
   10  cd /opt/
   11  ls
   12  cd scripts/
   13  ls
   14  cat firstscript.sh
   15  clear
   16  ./firstscript.sh
   17  ls -l firstscript.sh
   18  chmod 754 firstscript.sh
   19  ls -l
   20  ./firstscript.sh
   21  clear
[root@scriptbox scripts]#
```

#Writing script for website setup

ScriptingVM — root@scriptbox:/opt/scripts — ssh ◂ vagrant ssh scriptbox — 70×25

```
[root@scriptbox scripts]# vim websetup.sh
[root@scriptbox scripts]# cat websetup.sh
#!/bin/bash

# Installing Dependencies
echo "########################################"
echo "Installing packages."
echo "########################################"
sudo yum install wget unzip httpd -y > /dev/null
echo

# Start & Enable Service
echo "########################################"
echo "Start & Enable HTTPD Service"
echo "########################################"
sudo systemctl start httpd
sudo systemctl enable httpd

# Stop & Disable Service
echo "########################################"
echo "Stop & Disable Firewalld Service"
echo "########################################"
sudo systemctl stop firewalld
sudo systemctl disable firewalld
echo
```

#Executing the script

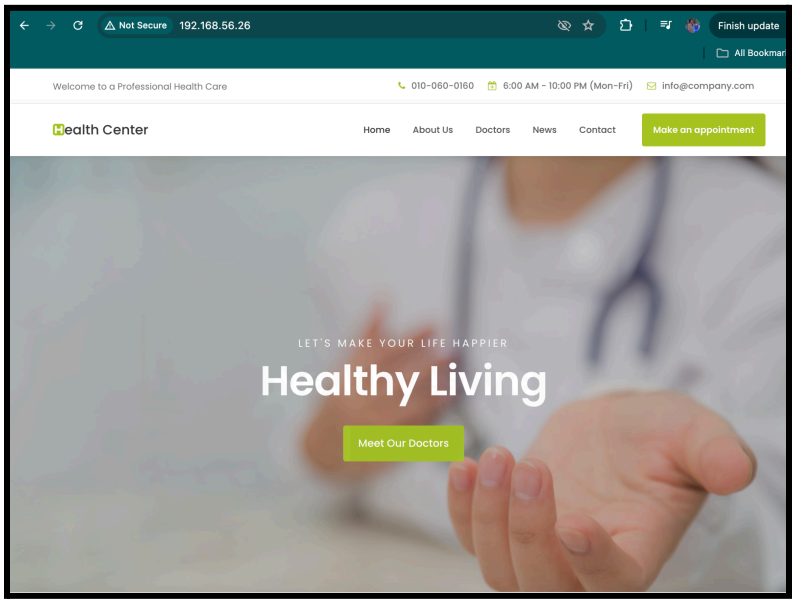


#Let’s test our website on browser using the VM IP

ScriptingVM — root@scriptbox:/opt/scripts — ssh ◂ vagrant ssh scriptbox — 64×21

```
[root@scriptbox scripts]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKN
OWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_cod
el state UP group default qlen 1000
    link/ether 00:0c:29:bd:54:93 brd ff:ff:ff:ff:ff:ff
    altname enp1s1
    inet 172.16.196.149/24 brd 172.16.196.255 scope global dynam
ic noprefixroute eth0
       valid_lft 1385sec preferred_lft 1385sec
    inet6 fe80::bd64:53c8:b0ba:4243/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_cod
el state UP group default qlen 1000
    link/ether 00:0c:29:bd:54:9d brd ff:ff:ff:ff:ff:ff
    altname enp1s4
    inet 192.168.56.26/24 brd 192.168.56.255 scope global nopref
ixroute eth1
```

Open the Browser and paste the IP



Commands:

[root@scriptbox scripts]# history

1. **clear**
2. **vim websetup.sh**
3. **cat websetup.sh**
4. **clear**
5. **ls -l websetup.sh**
6. **chmod 754 websetup.sh**
7. **ls -l | grep websetup.sh**
8. **./websetup.sh**
9. **clear**
10. **ip addr show**

[root@scriptbox scripts]#

#Variables

In Bash scripting, **variables** are used to store and manipulate data. They act as placeholders for values, allowing you to reuse and manage information efficiently within a script.

Key points:

Declaring a Variable: Variables in Bash are created by assigning a value without using the = sign. For example:

```
name="John"
```

- Here, name is the variable holding the value "John". Note that there should be no spaces around the = sign.

Accessing a Variable: To access the value of a variable, you prefix it with a $ symbol:

```
echo $name
```

- This will output John.

No Data Types: Bash variables are untyped, meaning they can hold strings, numbers, or even the output of commands without specifying a type.

Variables are essential for creating flexible and dynamic scripts, enabling better control over script behavior.

#Another Example

ScriptingVM — root@scriptbox:/opt/scripts — ssh ◂ vagrant ssh scriptbox — 79×21

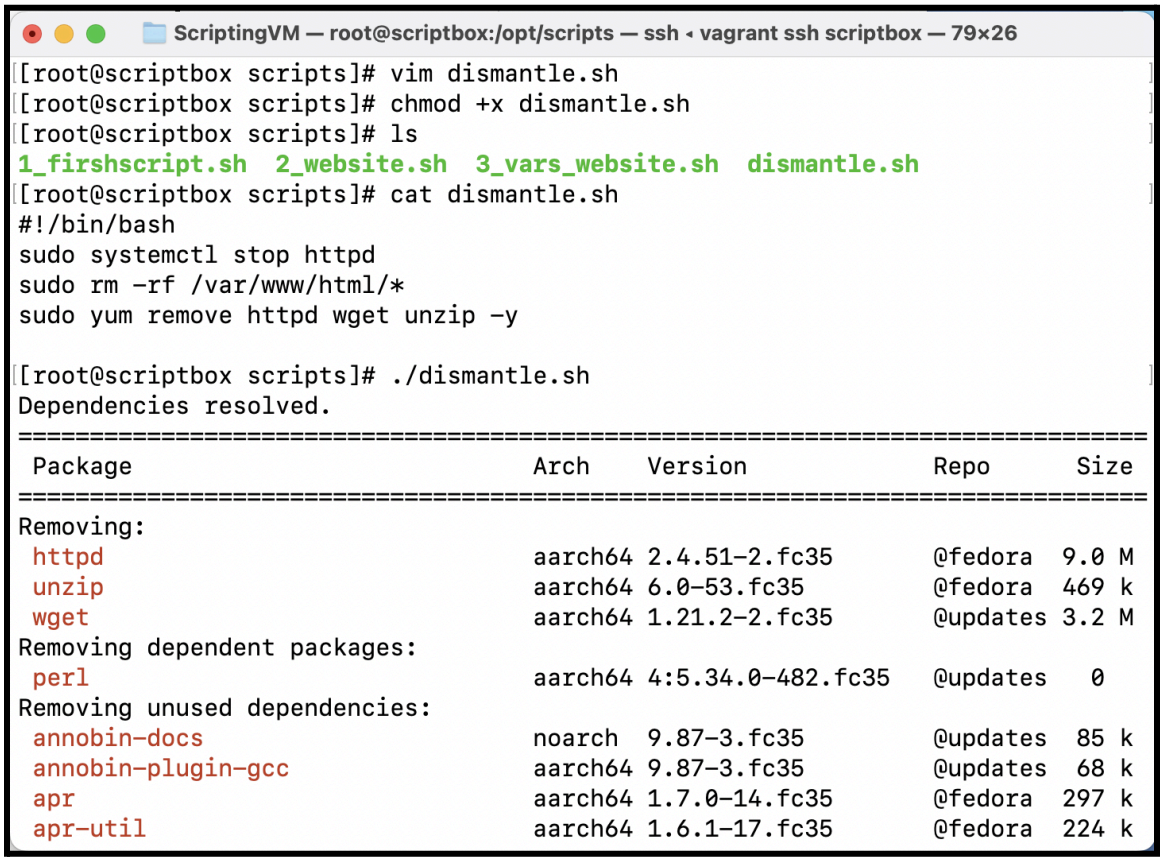
```
[root@scriptbox scripts]# ls
firstscript.sh  websetup.sh
[root@scriptbox scripts]# SKILL="DevOps"
[root@scriptbox scripts]# echo $SKILL
DevOps
[root@scriptbox scripts]# echo SKILL
SKILL
[root@scriptbox scripts]# PACKAGE="httpd wget unzip"
[root@scriptbox scripts]# yum install $PACKAGE -y
Last metadata expiration check: 1:11:08 ago on Tue 10 Sep 2024 03:35:19 AM PDT.
Package httpd-2.4.51-2.fc35.aarch64 is already installed.
Package wget-1.21.2-2.fc35.aarch64 is already installed.
Package unzip-6.0-53.fc35.aarch64 is already installed.
Dependencies resolved.
Nothing to do.
Complete!
[root@scriptbox scripts]#
```

#Let's use our website script with variables

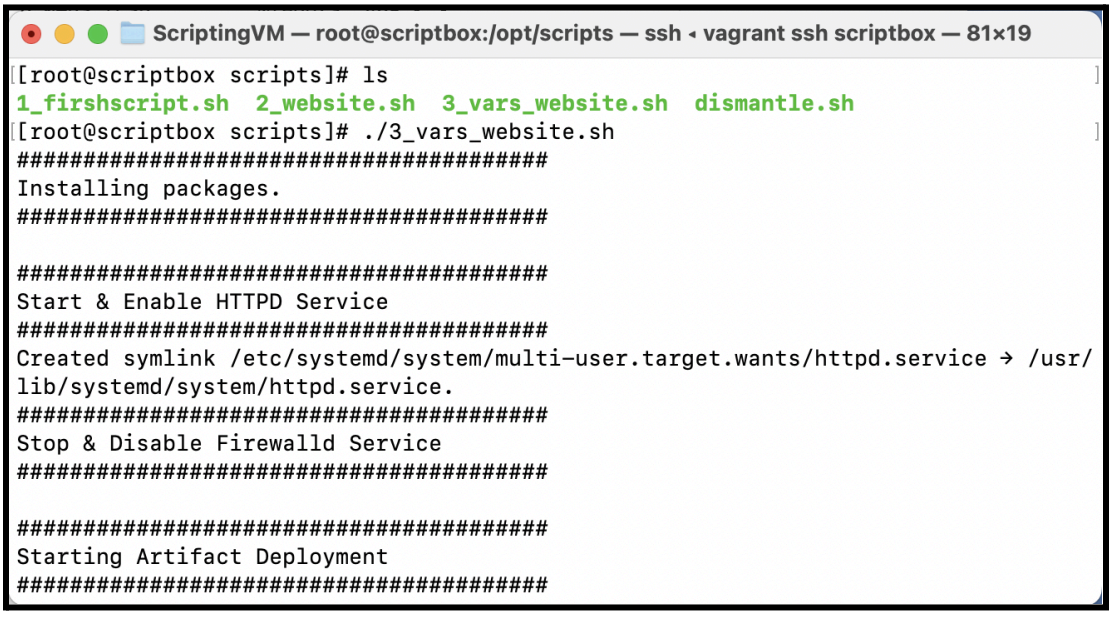
ScriptingVM — root@scriptbox:/opt/scripts — ssh ◂ vagrant ssh scriptbox — 79×26

```
[root@scriptbox scripts]# ls
firstscript.sh  websetup.sh
[root@scriptbox scripts]# mv firstscript.sh 1_firshscript.sh
[root@scriptbox scripts]# mv websetup.sh 2_website.sh
[root@scriptbox scripts]# ls
1_firshscript.sh  2_website.sh
[root@scriptbox scripts]# cp 2_website.sh 3_vars_website.sh
[root@scriptbox scripts]# ls
1_firshscript.sh  2_website.sh  3_vars_website.sh
[root@scriptbox scripts]# vi 3_vars_website.sh
[root@scriptbox scripts]# cat 3_vars_website.sh
#!/bin/bash

# Variable Declaration
PACKAGE="httpd wget unzip"
SVC="httpd"
URL="https://www.tooplate.com/zip-templates/2098_health.zip"
ART_NAME='2098_health'
TEMPDIR="/tmp/webfiles"

# Installing Dependencies
echo "######################################"
echo "Installing packages."
echo "######################################"
sudo yum install $PACKAGE -y > /dev/null
echo
```

#Let's remove all the packages from 2_website.sh to test 3_vars_website.sh website script with variables



#Let's test 3_vars_website.sh



#Working Fine.

#Command line arguments

In Bash scripting, command-line arguments allow you to pass data or parameters to your script when you run it from the terminal. These arguments make your script more dynamic, as you can provide different inputs without modifying the script itself.

Key Concepts:

- $0: Refers to the name of the script itself.
- $1, $2, ...: These represent the first, second, and subsequent command-line arguments passed to the script.
- $#: Represents the total number of command-line arguments passed.
- $@: A list of all the arguments.
- $*: Similar to $@, but handles arguments slightly differently in special cases.

Example:

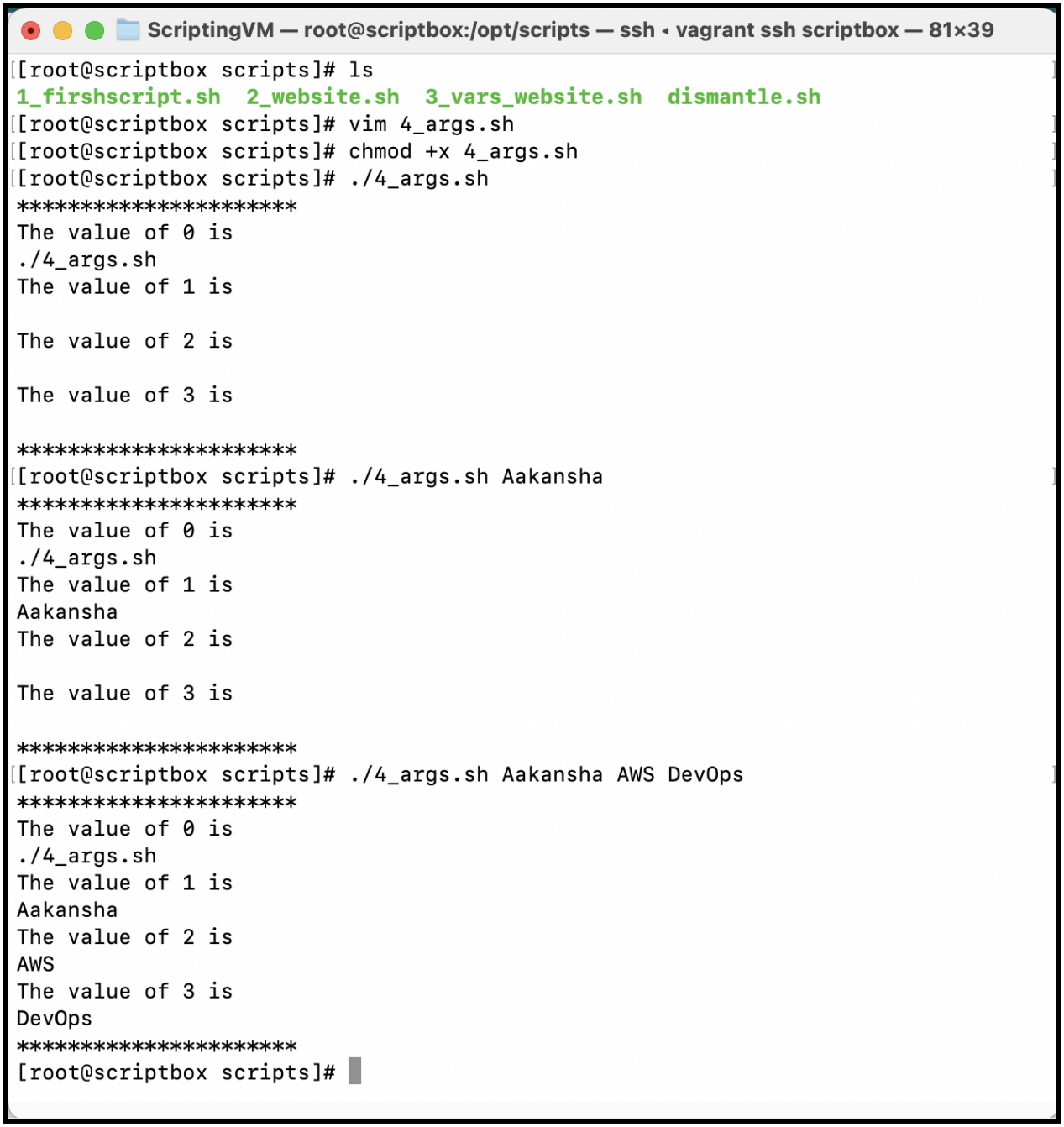
If you run a script like this:

```
./script.sh arg1 arg2
```

- $0 would be ./script.sh
- $1 would be arg1
- $2 would be arg2
- $# would be 2, indicating two arguments were passed.

This allows for flexible scripts that can process different inputs based on the provided arguments.

#Let’s start the practice



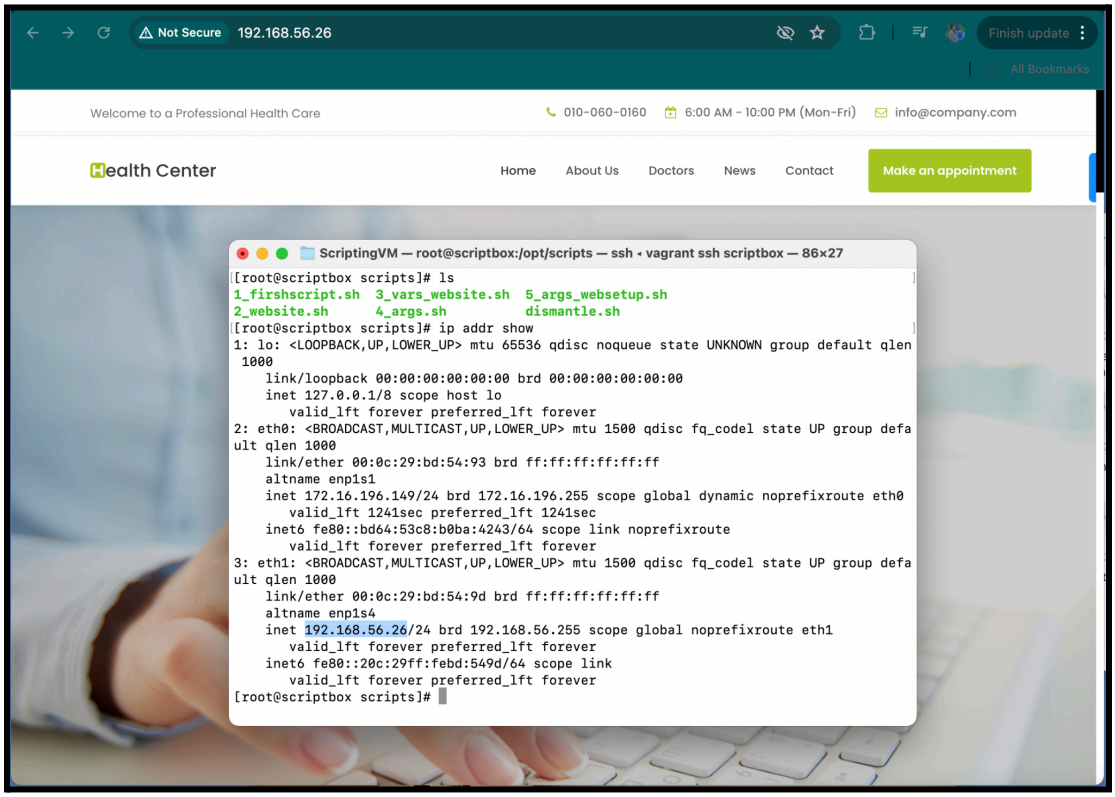
This Bash script prints the values of the script name and the first three command-line arguments passed to it:

- $0: Prints the name of the script.
- $1, $2, $3: Prints the first, second, and third arguments provided when running the script.
- When you run the script with arguments,It will output the script name and the values of arg1, arg2, and arg3 accordingly.

#Let’s see the application of command line arguments in our website setup script

```
ScriptingVM — root@scriptbox:/opt/scripts — ssh ◂ vagrant ssh scriptbox — 86×22
[root@scriptbox scripts]# ls
1_firshscript.sh  2_website.sh  3_vars_website.sh  4_args.sh  dismantle.sh
[root@scriptbox scripts]# cp 3_vars_website.sh 5_args_websetup.sh
[root@scriptbox scripts]# vim 5_args_websetup.sh
[root@scriptbox scripts]# ./dismantle.sh > /dev/null 2>&1
[root@scriptbox scripts]# vim 3_vars_website.sh
[root@scriptbox scripts]# chmod +x 5_args_websetup.sh
[root@scriptbox scripts]# ./5_args_websetup.sh https://www.tooplate.com/zip-templates/
2098_health.zip 2098_health
########################################
Installing packages.
########################################

########################################
Start & Enable HTTPD Service
########################################
Created symlink /etc/systemd/system/multi-user.target.wants/httpd.service → /usr/lib/s
ystemd/system/httpd.service.
########################################
Stop & Disable Firewalld Service
########################################
```

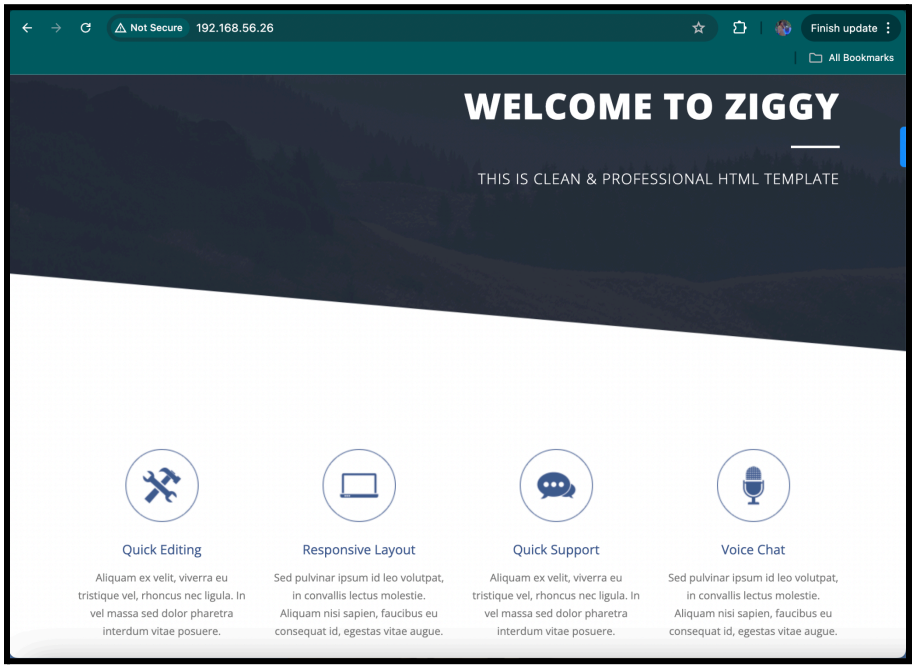
#Output



#Let’s see the application of command line arguments in our website setup script with another url:
https://www.tooplate.com/zip-templates/2091_ziggy.zip

```
ScriptingVM — root@scriptbox:~/scripts — ssh ‹ vagrant ssh scriptbox — 125×33
[root@scriptbox scripts]# ls
1_firstscript.sh  2_websetup.sh  3_vars_websetup.sh  4_args.sh  5_args_websetup.sh  dismantle.sh
[root@scriptbox scripts]# ./5_args_websetup.sh https://www.tooplate.com/zip-templates/2091_ziggy.zip 2091_ziggy > /dev/null
--2024-09-10 22:29:35--  https://www.tooplate.com/zip-templates/2091_ziggy.zip
Resolving www.tooplate.com (www.tooplate.com)... 72.52.176.250
Connecting to www.tooplate.com (www.tooplate.com)|72.52.176.250|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4252660 (4.1M) [application/zip]
Saving to: ‘2091_ziggy.zip’

2091_ziggy.zip                 100%[=====================================================>]   4.05M   453KB/s    in 11s


2024-09-10 22:29:48 (378 KB/s) - ‘2091_ziggy.zip’ saved [4252660/4252660]

[root@scriptbox scripts]# ip addr show
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:bd:54:93 brd ff:ff:ff:ff:ff:ff
    altname enp1s1
    inet 172.16.196.149/24 brd 172.16.196.255 scope global dynamic noprefixroute eth0
       valid_lft 1623sec preferred_lft 1623sec
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 00:0c:29:bd:54:9d brd ff:ff:ff:ff:ff:ff
    altname enp1s4
    inet 192.168.56.26/24 brd 192.168.56.255 scope global noprefixroute eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::20c:29ff:febd:549d/64 scope link
       valid_lft forever preferred_lft forever
[root@scriptbox scripts]#
```

#Output



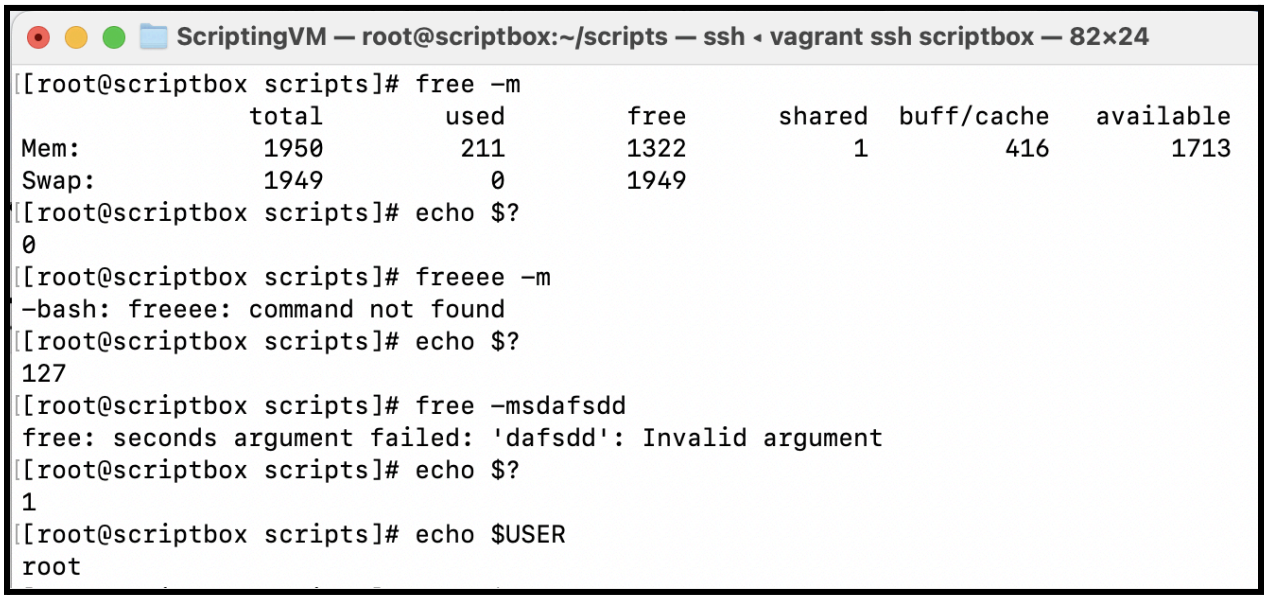
System variables

System variables in Bash scripting are predefined variables that provide information about the system and the environment. These variables are automatically set by the shell and can be accessed within scripts.

Common system variables:

- **$HOME:** The home directory of the current user.
- **$USER:** The username of the current user.
- **$PATH:** A list of directories where the system looks for executable files.
- **$SHELL:** The path to the current shell.
- **$PWD:** The current working directory.

System variables are useful for writing scripts that interact with the environment or require system-specific data.



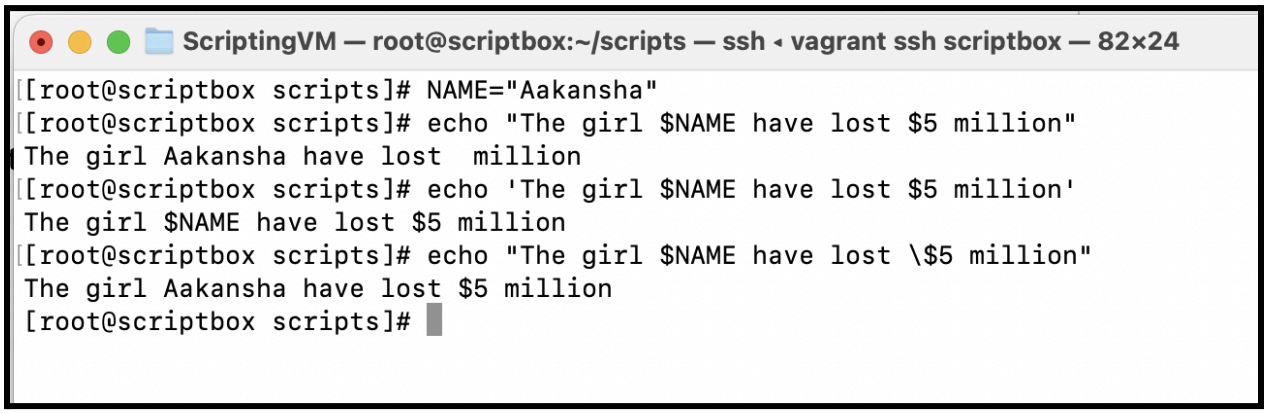
This example demonstrates the exit status in Bash. After running a command, you can check its success or failure using the special variable $?.

- free -m shows memory usage and returns an exit status of 0, indicating success.
- freeee -m gives "command not found" and returns 127, meaning the command doesn't exist.
- free -msdafsdd gives an invalid argument error and returns 1, indicating the command ran but failed due to incorrect input.

The exit status helps in determining whether a command executed successfully (0) or encountered an error (non-zero).

 # Quotes

In Bash scripting, quotes control how variables and special characters are interpreted.



- Double quotes ("): Variables like $NAME are expanded, but special characters like $ can still be escaped with a backslash (\). In the above example, "The girl $NAME" correctly expands to "Aakansha", but \$5 is shown as "$5" when escaped.
- Single quotes ('): Everything inside is treated as literal text, so $NAME and $5 are not expanded or interpreted.

Quotes allows precise control over how strings and variables are handled.

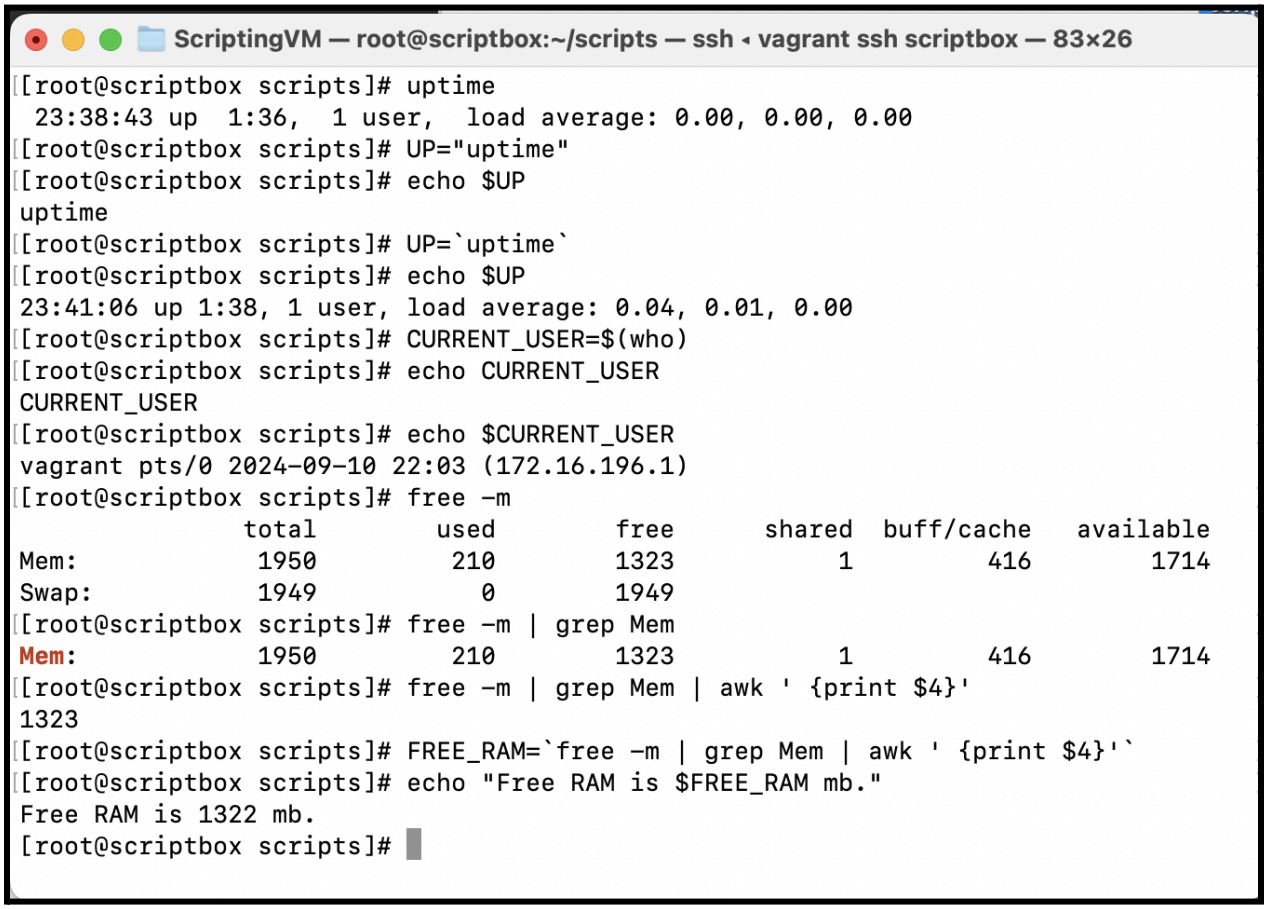
Command substitution

Command substitution in Bash allows you to run a command and use its output as part of another command or assign it to a variable.
Syntax:

- Using backticks: ``command``
- Using `$(command)` (preferred).

```
today=$(date)
```

This assigns the output of the date command to the today variable.
Command substitution is useful for dynamically capturing command output and using it within scripts.

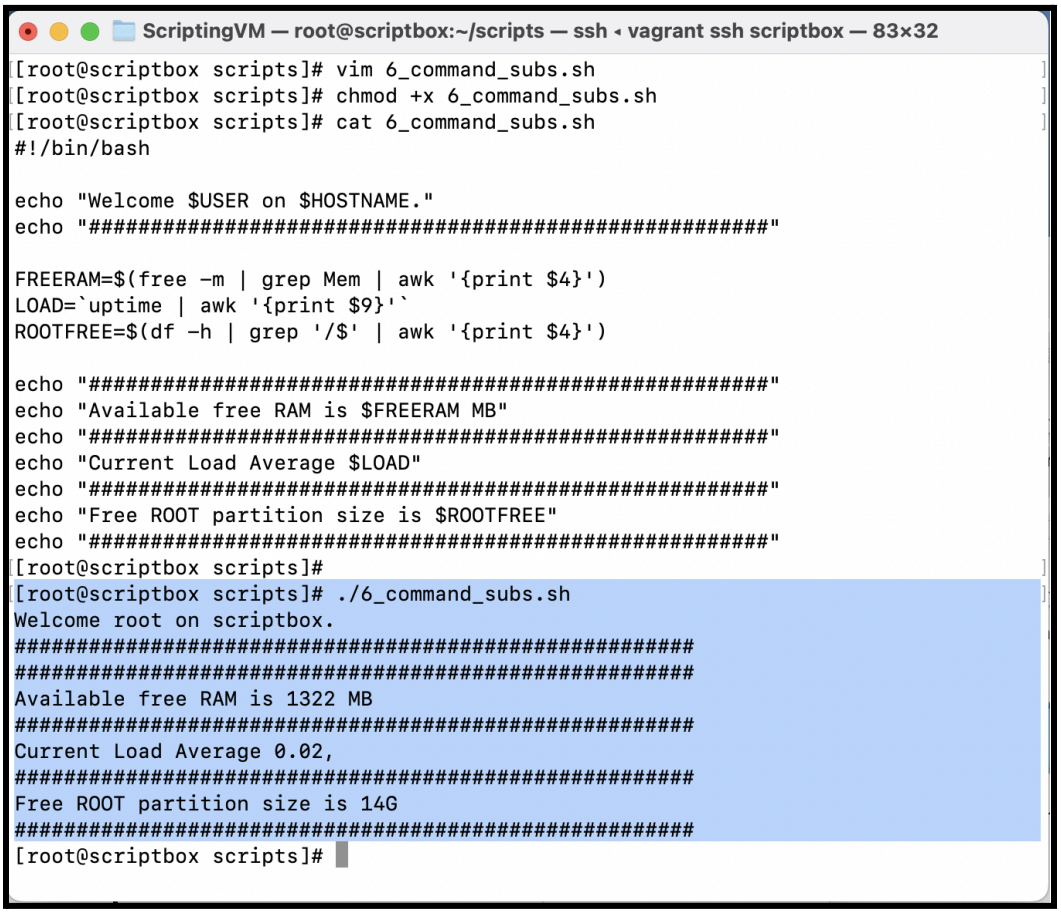


This example demonstrates command substitution and pipelining:

1. **Command substitution:** The output of a command is assigned to a variable, like using `uptime` or $(who) to store the output in UP and CURRENT_USER respectively.
    - UP=\uptime`captures the system uptime and stores it inUP`.
    - CURRENT_USER=$(who) stores the current user information.
2. **Pipelining:** Commands are combined using pipes (|) to process output in stages.
    - free -m | grep Mem | awk '{print $4}' extracts the available memory in MB.
    - This value is stored in FREE_RAM and printed as "Free RAM is $FREE_RAM mb.".

This illustrates how command substitution and pipelines can be used together to process and store command outputs dynamically.

#Let's create a script to print the health of the system using the concept of command substitution and pipelining.



 # Exporting variables

In Bash, exporting variables allows them to be accessible to child processes or scripts spawned from the current shell. Without exporting, variables are only available in the current shell session.

To export a variable:

```
export VAR_NAME=value
```

Once a variable is exported, it can be accessed by any scripts or commands run in the same shell session or any subshells.

Example:

```
NAME="Aakansha"
export NAME
```

Here, NAME is exported and can be used by any child processes, like another script run from the terminal.

Exporting is essential when you need to share environment variables between scripts or commands.

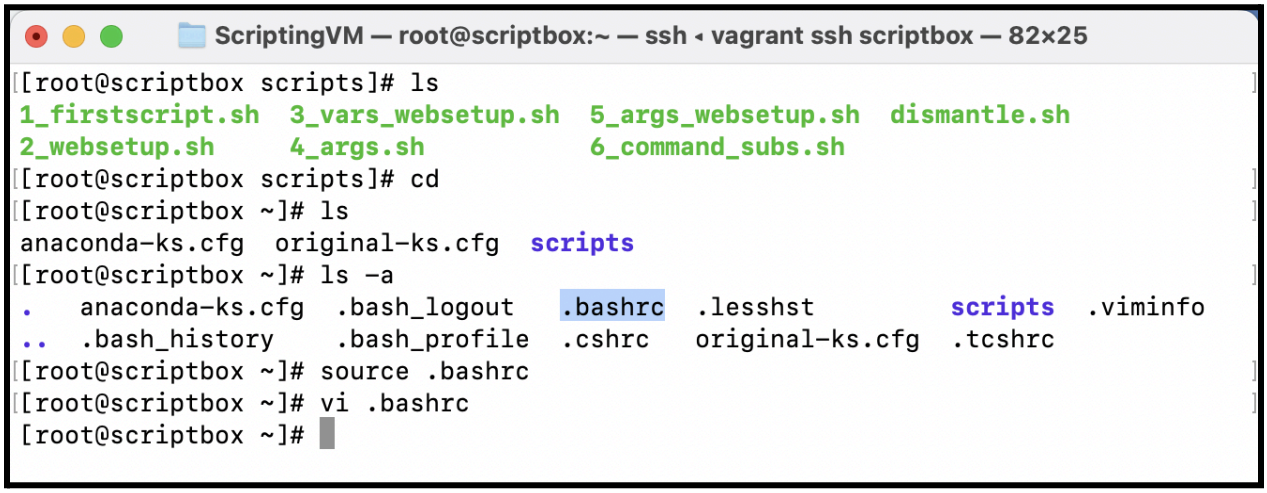
The .bashrc file is commonly used to export variables that need to be available in every Bash session automatically.

By adding export commands in .bashrc, you ensure that those variables are set and exported each time you open a new terminal or start a new session.

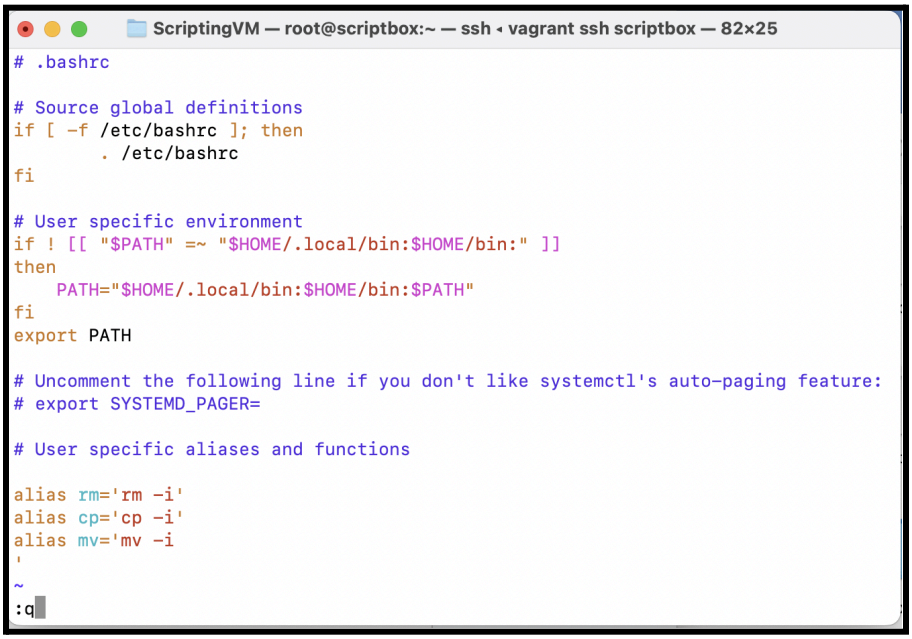
The .bashrc file is a script that runs automatically whenever a new Bash session (non-login) is started, such as opening a terminal. It is used to configure your environment by defining aliases, functions, and environment variables that persist in each session.

Key points:

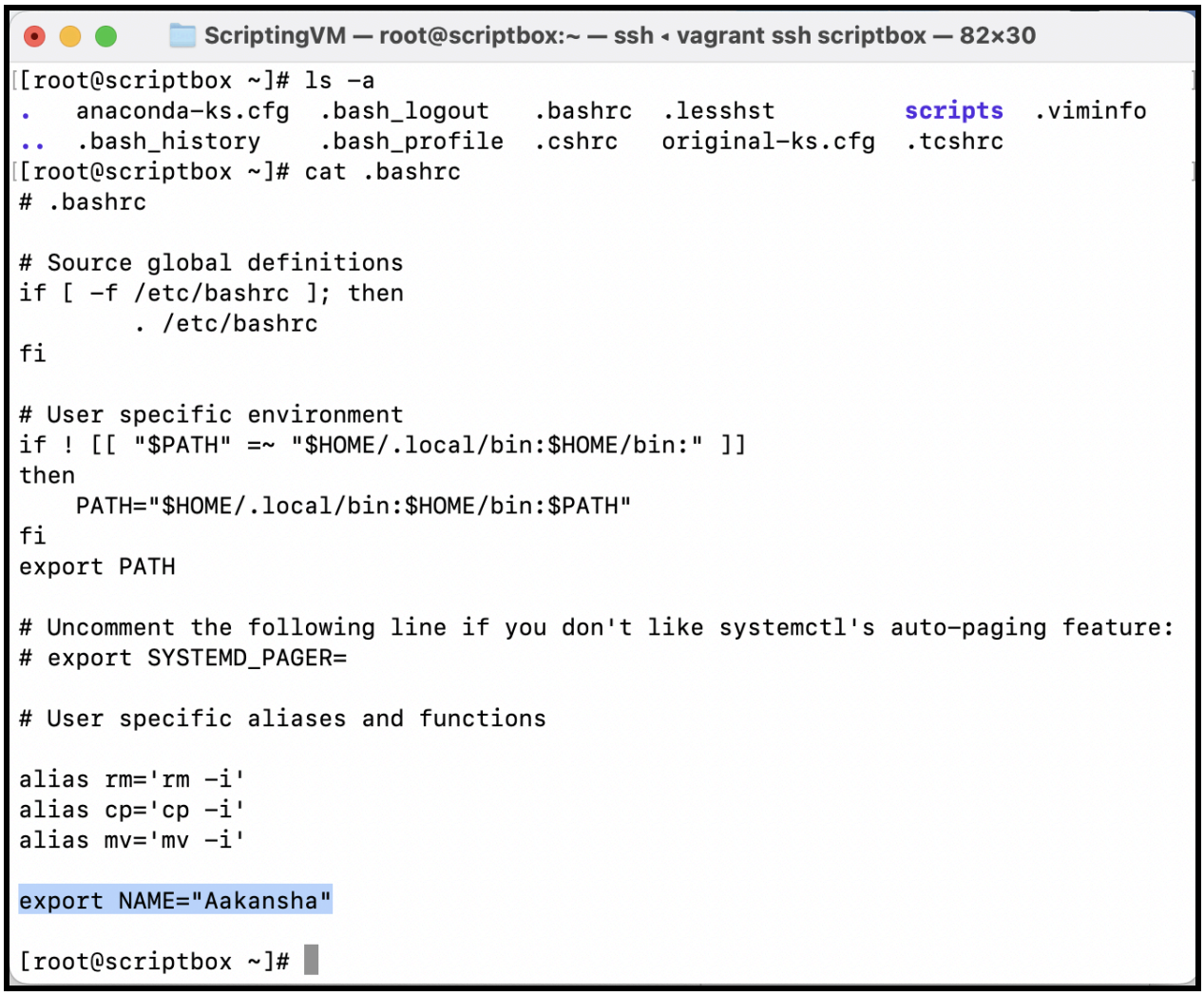
- Located in your home directory (~), it's hidden (indicated by the . before the name).
- You can customize it to set environment variables or add aliases that you want available in every terminal session.
- After editing .bashrc, you need to run source .bashrc to apply changes immediately without restarting the session.



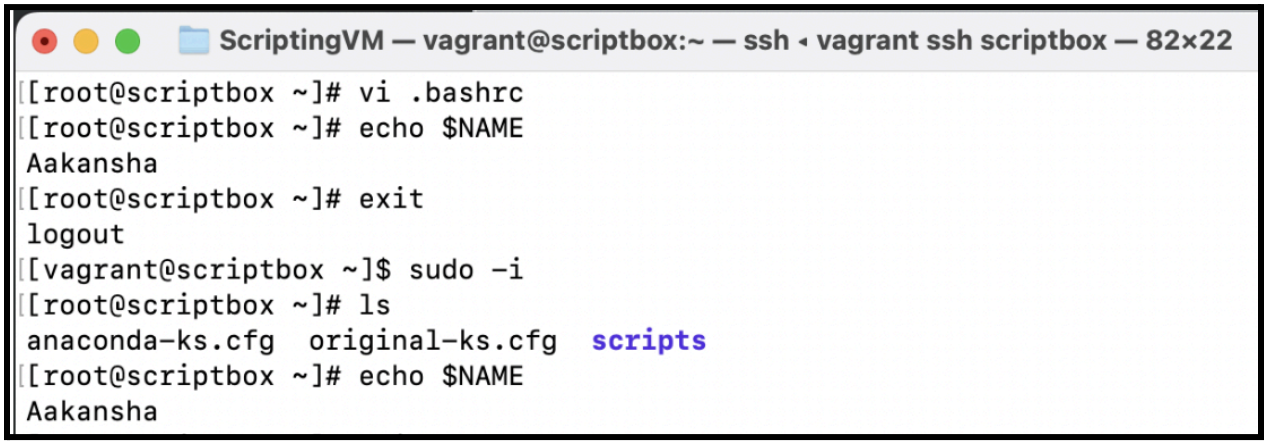
Current Bashrc File:



After updating the bashrc file with export command:

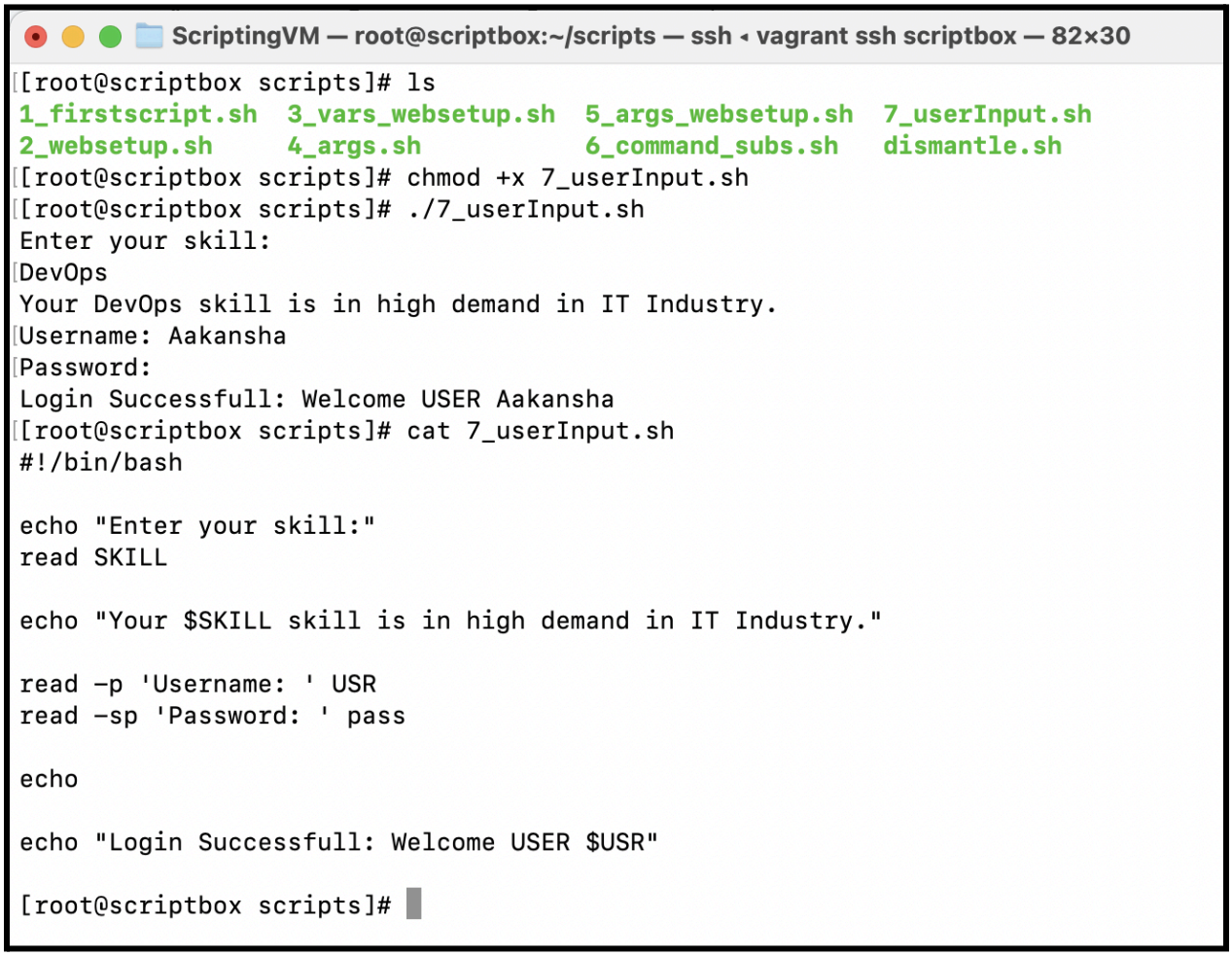


Output:



User Input

In Bash scripting, you can get user input using the read command. It's simple and effective for interactive scripts.



This script prompts the user for input and displays personalized messages based on the input.

1. It asks for the user's skill using read SKILL and then prints a message that their skill is in high demand.
2. It prompts for a **username** and **password**:
    - read -p 'Username: ' takes the username input.
    - read -sp 'Password: ' takes the password input silently (without showing on the screen).
3. After the input, it prints a welcome message using the entered username.

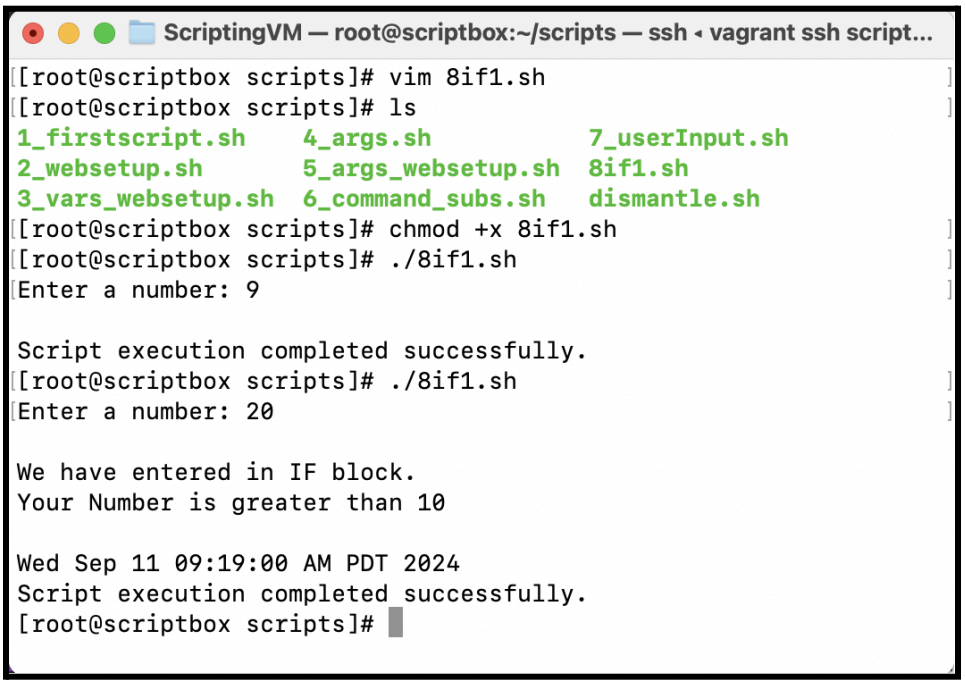
This is an interactive script that handles user input dynamically.

Decision-making

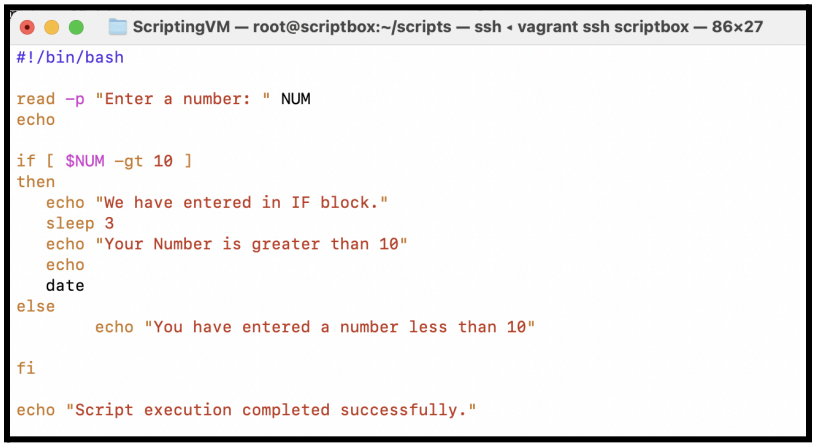
In Bash scripting, decision-making allows you to control the flow of the script based on conditions.

1. If command:

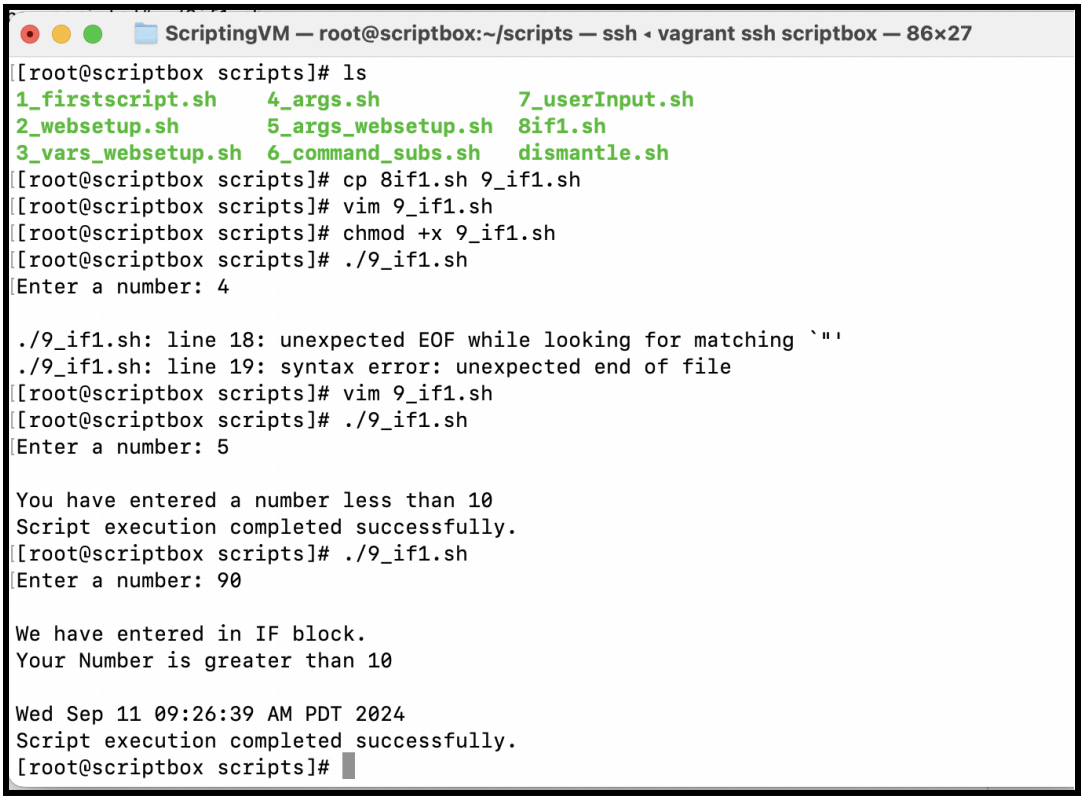
The if command checks a condition, and if it's true, it executes the following commands.



The else statement is used to define a block of code that will execute if none of the if or elif conditions are met. It's the final fallback option when all other conditions are false.

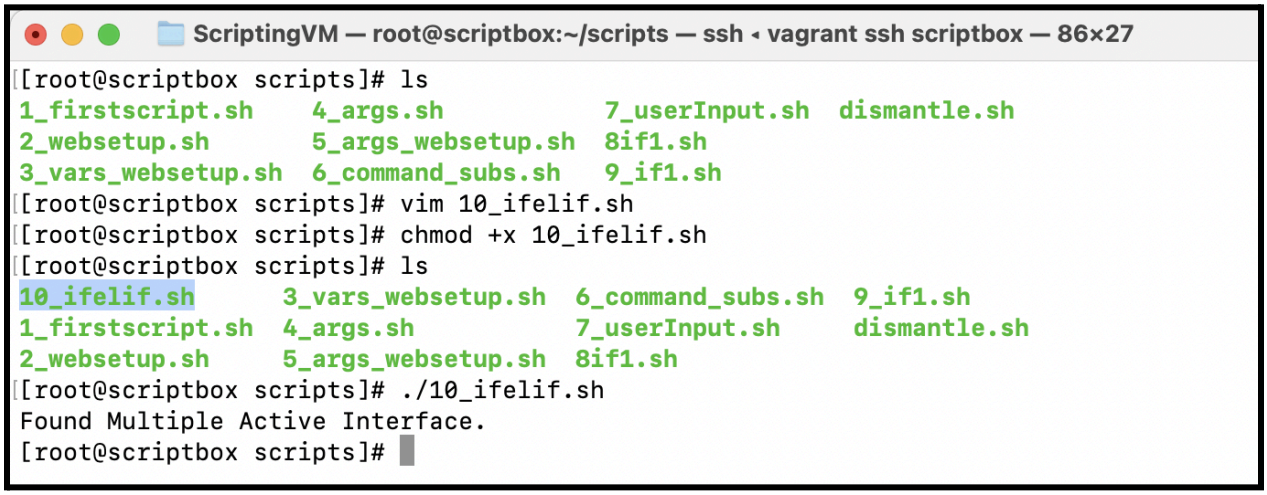


OUTPUT:



2. Elif command:

elif stands for "else if," and it allows you to check multiple conditions.

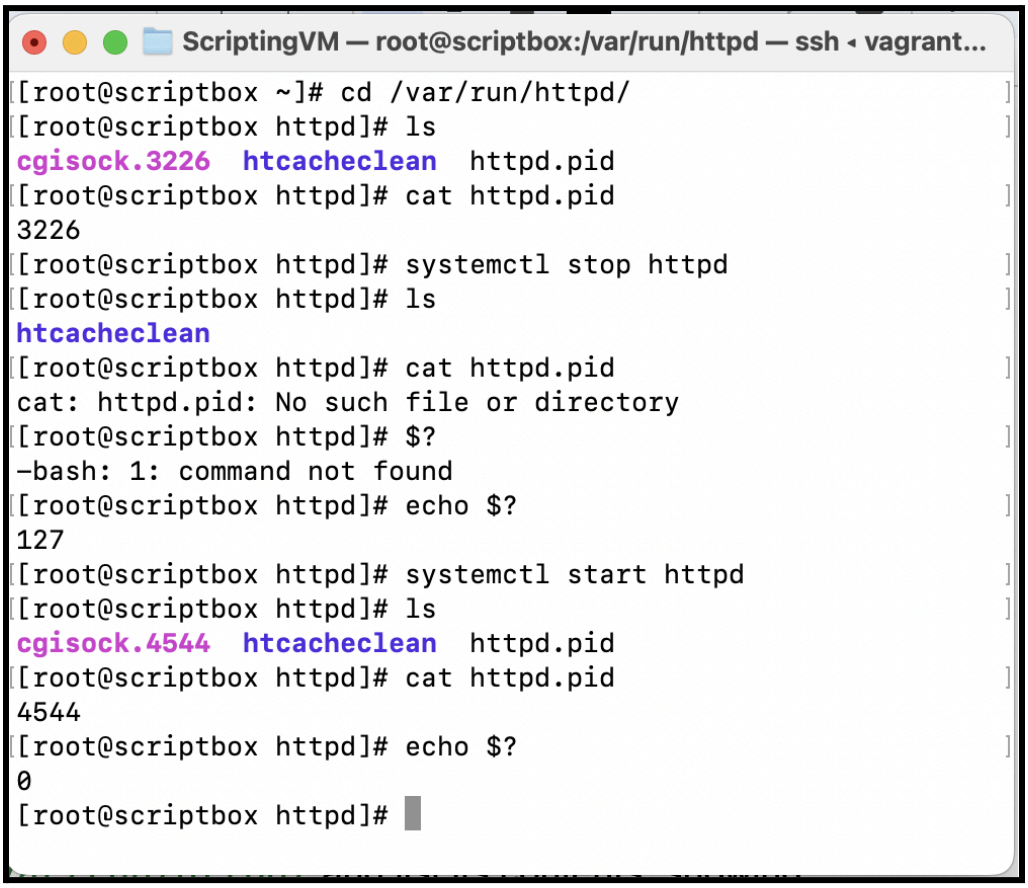


Bash operators

Operator	Description	Example
`-eq`	Equal to (integer comparison)	`[ 5 -eq 5 ]` (true)
`-ne`	Not equal to (integer comparison)	`[ 5 -ne 3 ]` (true)
`-gt`	Greater than	`[ 5 -gt 3 ]` (true)
`-lt`	Less than	`[ 3 -lt 5 ]` (true)
`-ge`	Greater than or equal to	`[ 5 -ge 5 ]` (true)
`-le`	Less than or equal to	`[ 5 -le 5 ]` (true)
`==`	String equality	`[ "abc" == "abc" ]` (true)
`!=`	String inequality	`[ "abc" != "xyz" ]` (true)
`-z`	String is empty	`[ -z "$str" ]`
`-n`	String is not empty	`[ -n "$str" ]`
`&&`	Logical AND	`[ 5 -gt 3 ] && [ 5 -lt 10 ]` (true)
`		`
`!`	Logical NOT	`[ ! 5 -eq 3 ]` (true)
`-d`	Checks if directory exists	`[ -d /path/to/dir ]`
`-f`	Checks if file exists	`[ -f /path/to/file ]`
`-r`	File is readable	`[ -r /path/to/file ]`
`-w`	File is writable	`[ -w /path/to/file ]`
`-x`	File is executable	`[ -x /path/to/file ]`

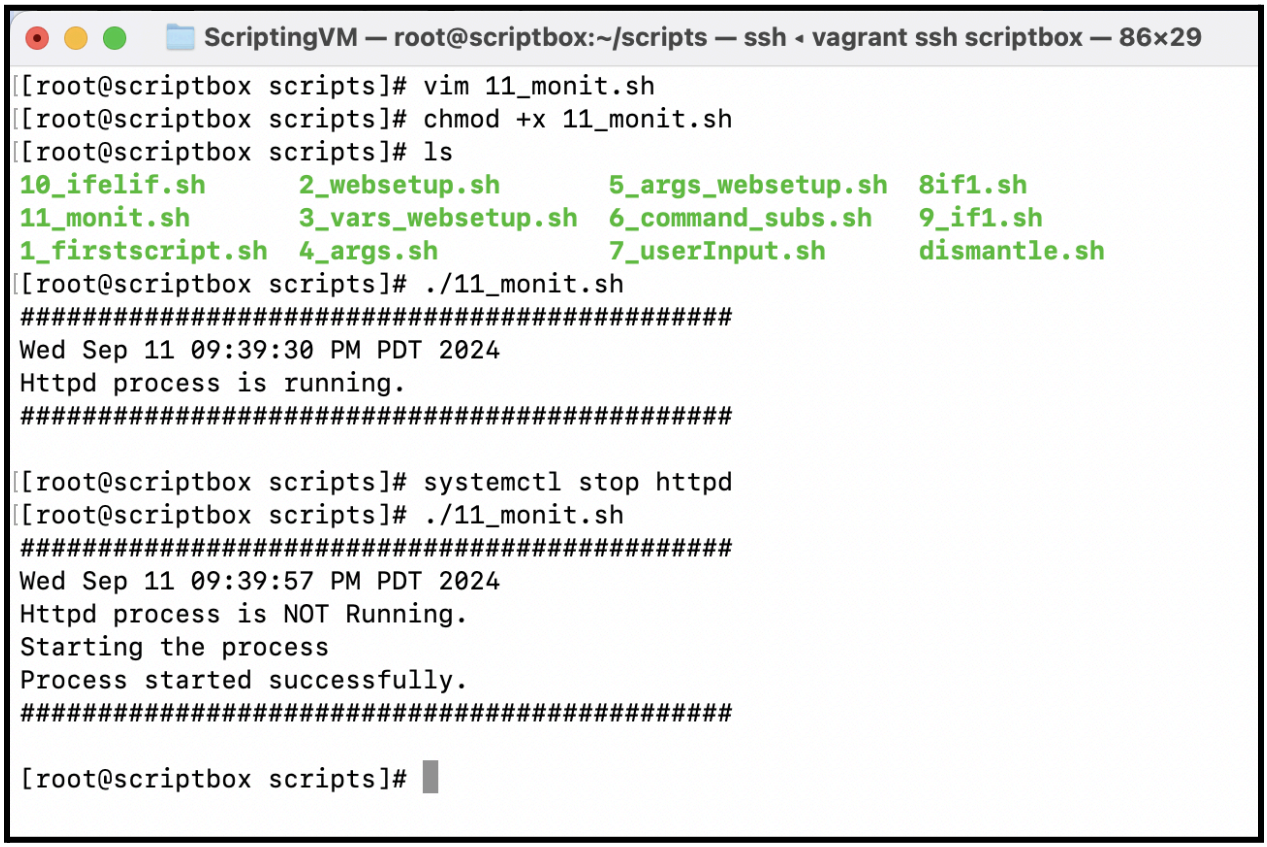
 # Process ID

Whenever there is a process running there will be a PID file created.

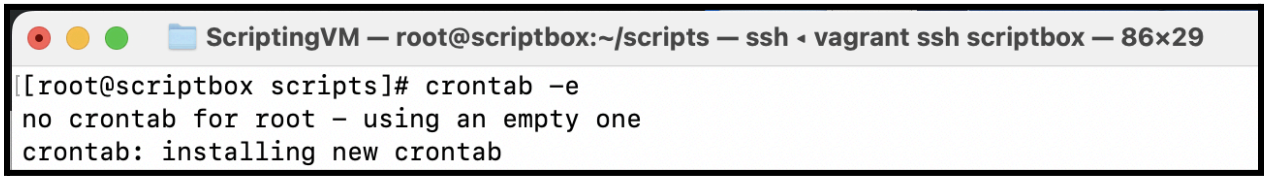


1. We navigate to /var/run/httpd/ and list its contents, showing files related to the HTTP server, including httpd.pid, which contains the process ID (PID) of the running httpd.
2. After stopping, the httpd.pid file is gone because the process has terminated.
3. The usage (echo $?) returns 127, indicating an error.
4. After restarting the httpd service, a new PID (4544) appears in httpd.pid, and the command echo $? returns 0, showing success.

 # Let's use the above concept to make a monitoring script that will detect if the process is running and if it's not running it will start the process.

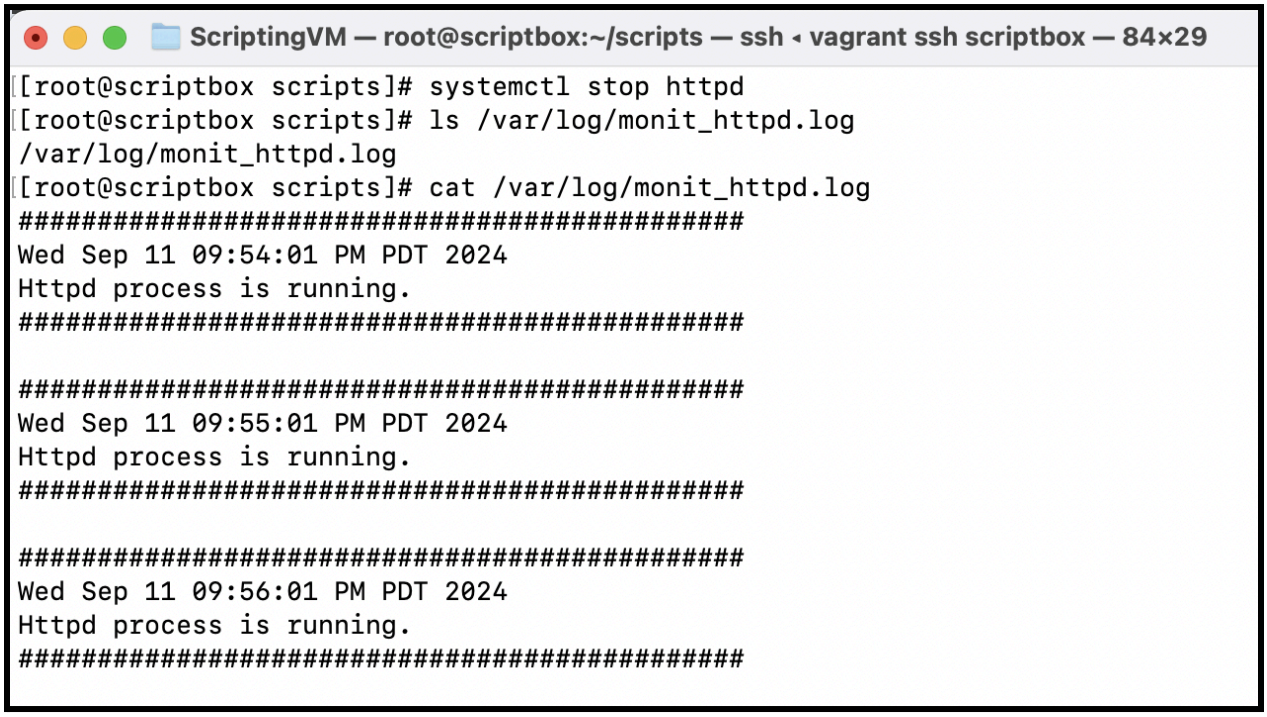


 # Let's automate this process for every minute by scheduling it. First edit the crontab and schedule the job.

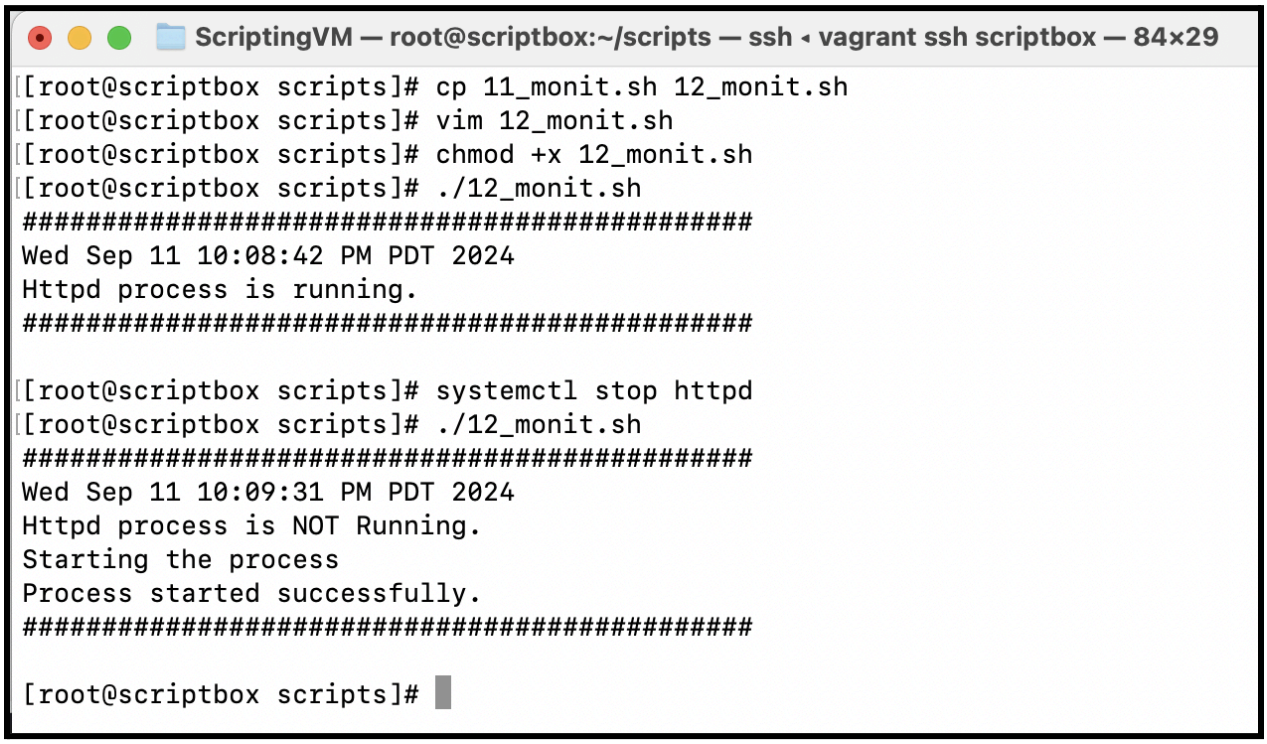


We are scheduling it for every day of the month, every minute.

Let’s test the script and check the log file



Let’s add some creativity to the script using bash operator (-f)



Loops in Bash

In Bash scripting, loops allow you to execute a block of code repeatedly, depending on a condition. The most common types of loops are:

1. For Loop

Executes a block of code for each item in a list or range.

```
for item in 1 2 3 4 5; do
  echo $item
done
```

2. While Loop

Runs as long as the condition remains true.

```
count=1
while [ $count -le 5 ]; do
  echo "Count is $count"
  ((count++))
done
```

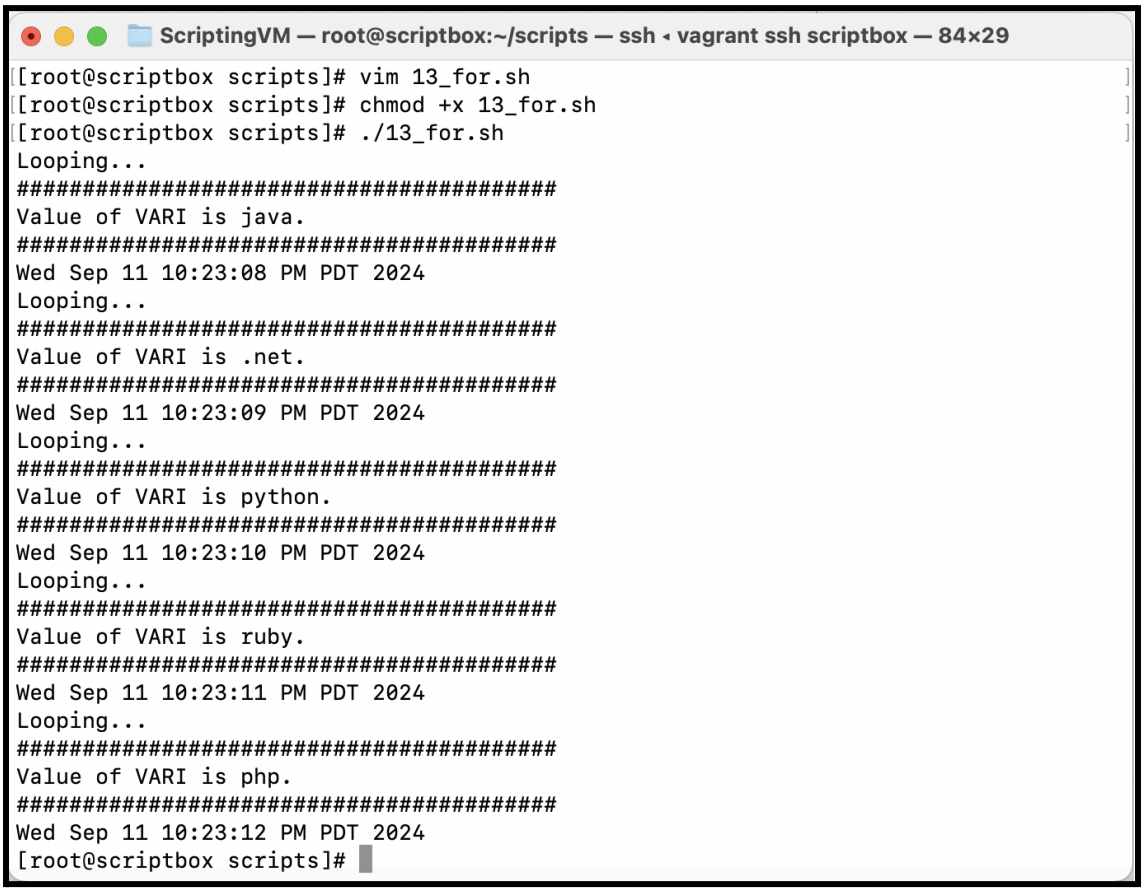
3. Until Loop

Runs until the condition becomes true.

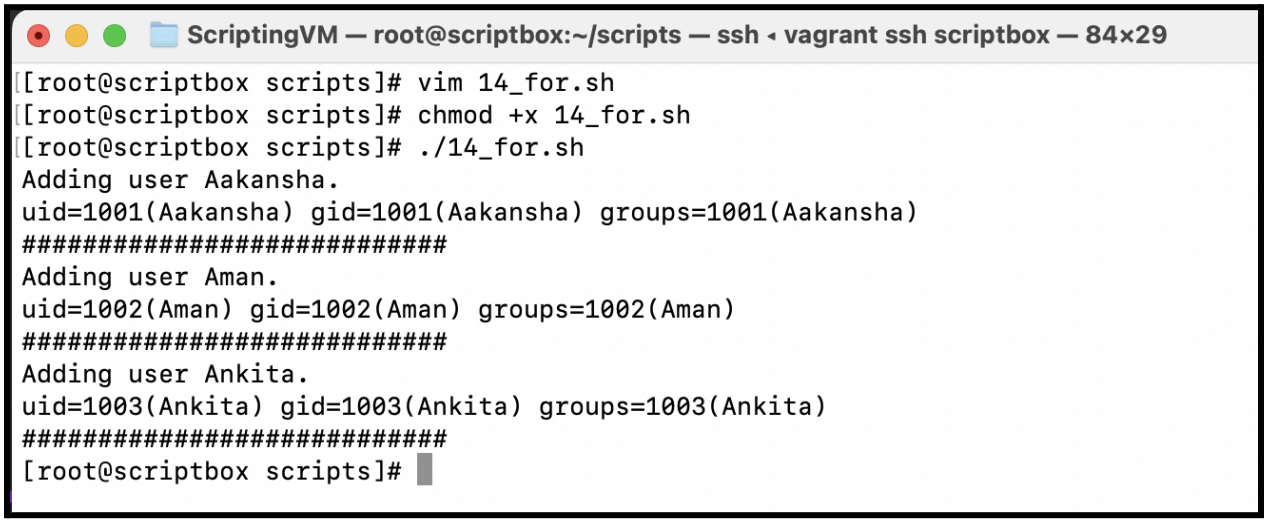
```
count=1
until [ $count -gt 5 ]; do
  echo "Count is $count"
  ((count++))
done
```

#Let’s start the practice using scripts

Example-1 :



Example-2 :



While Loops in Bash

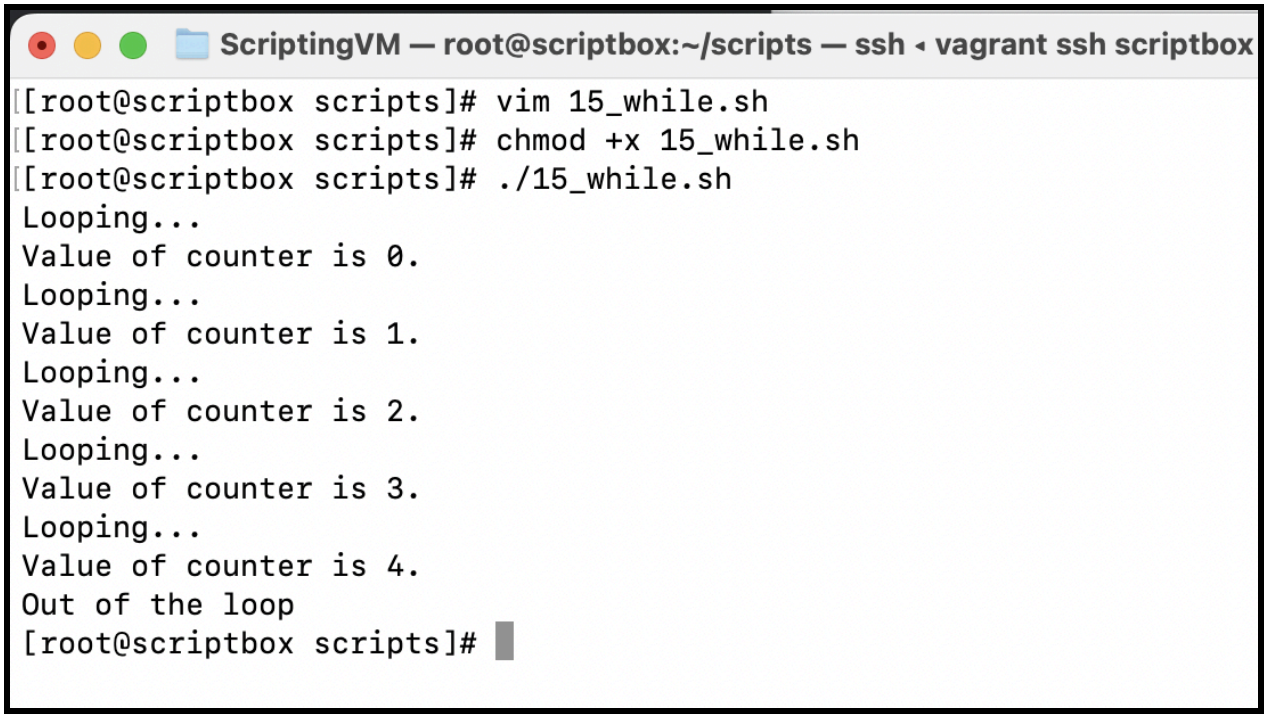
A while loop in Bash repeatedly executes a block of code as long as the given condition is true. The condition is evaluated before each iteration, and if it returns true, the loop continues.

Syntax:

```
while [ condition ]; do
  # Code to be executed
done
```

Let's Start the practice

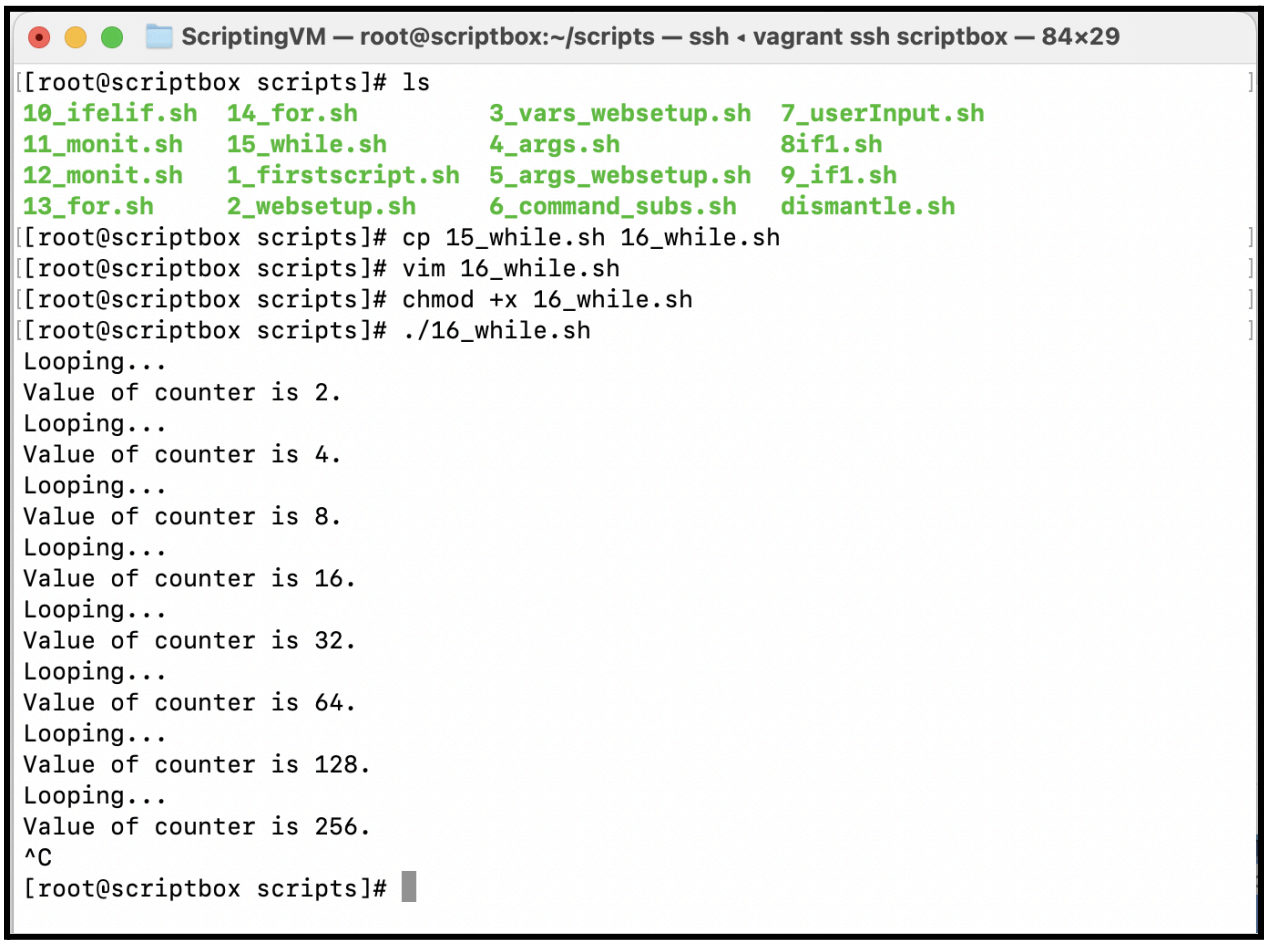
Example-1: Simple while loop

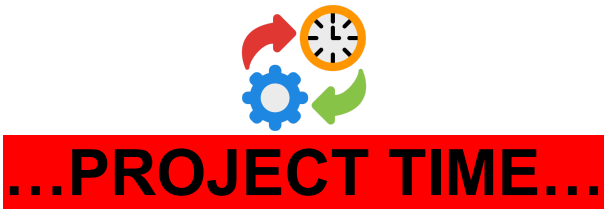


Infinite while loop (until manually stopped)

```
while true; do
  echo "This will run forever"
done
```

Example-2: Infinite while loop





Remote Command Execution

Remote Command Execution (RCE) allows a user to execute commands on a remote machine from a local system. This is commonly done using tools like ssh (Secure Shell) to securely connect and manage remote servers or computers.

How It Works:

- You connect to a remote server using SSH.
- Once authenticated, you can execute commands on the remote machine as if you were physically present at the system.

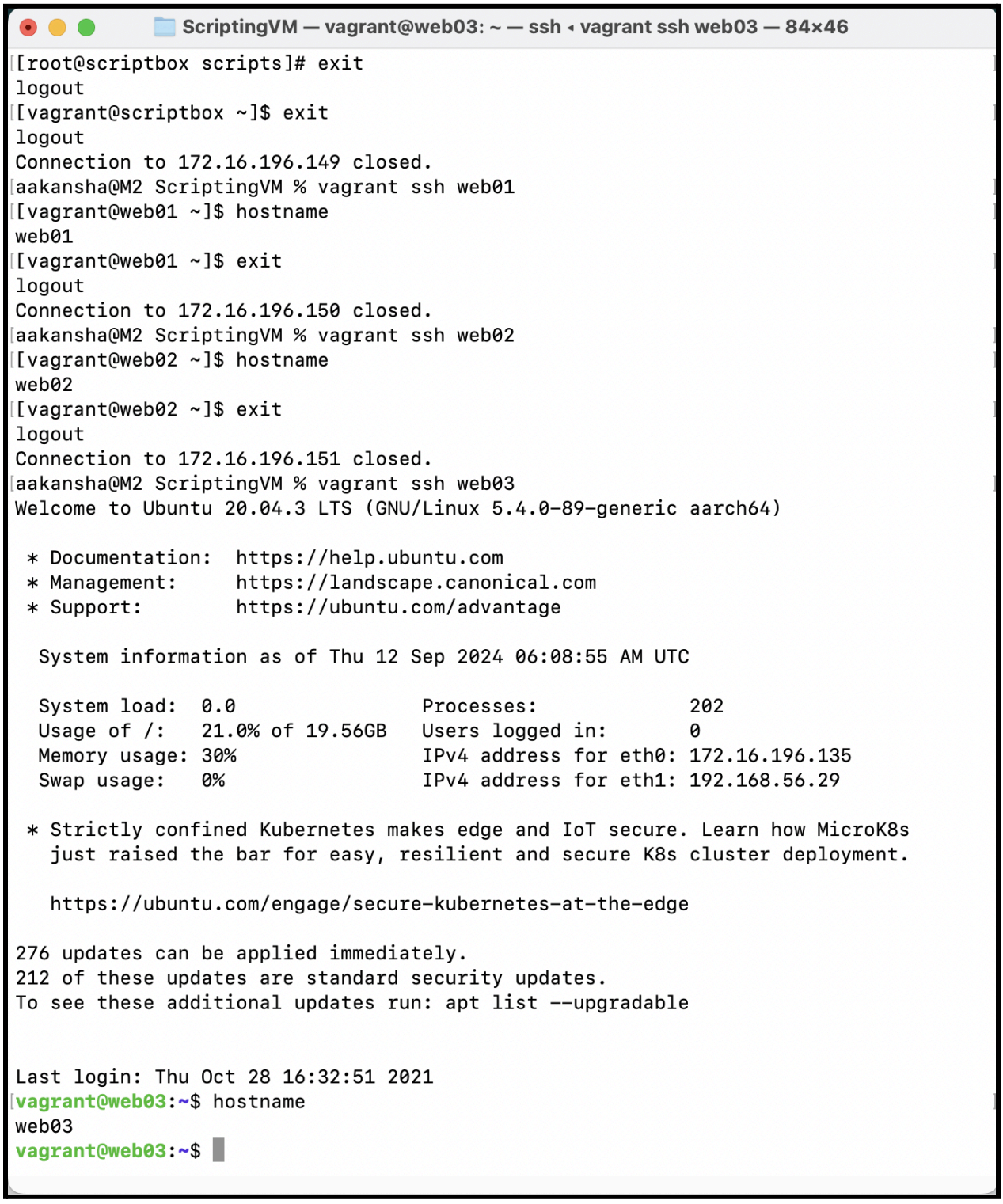
Benefits of Remote Command Execution:

1. **Convenience**:
   Allows system administrators and users to manage multiple machines from a single location, saving time and effort.
2. **Automation**:
   Scripts can be written to execute commands remotely on multiple servers, making it easy to perform tasks like updates, backups, and monitoring across machines.
3. **Security**:
   SSH encrypts all communications, making it a secure way to manage remote systems and execute commands.
4. **Resource Management**:
   Remote command execution helps in monitoring and managing remote system resources without needing to access them physically.
5. **Efficiency**:
   Quick troubleshooting, software installation, and configuration changes can be performed remotely, improving the efficiency of IT management.

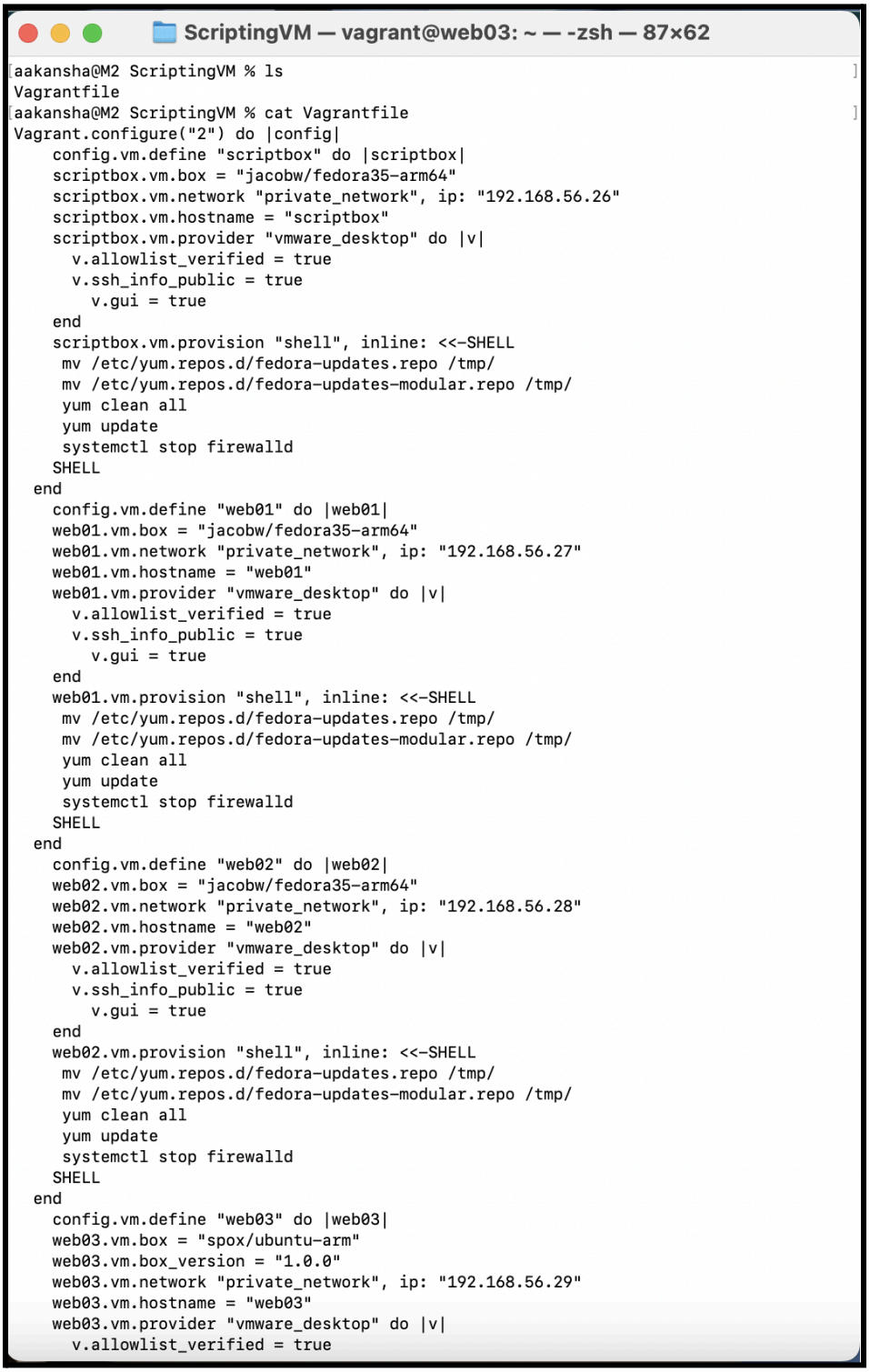
RCE is widely used in system administration, DevOps, and cloud management for effective remote control of resources.

In this project we will see how we can execute commands from script box to the other VMS which were created using this vagrant file.

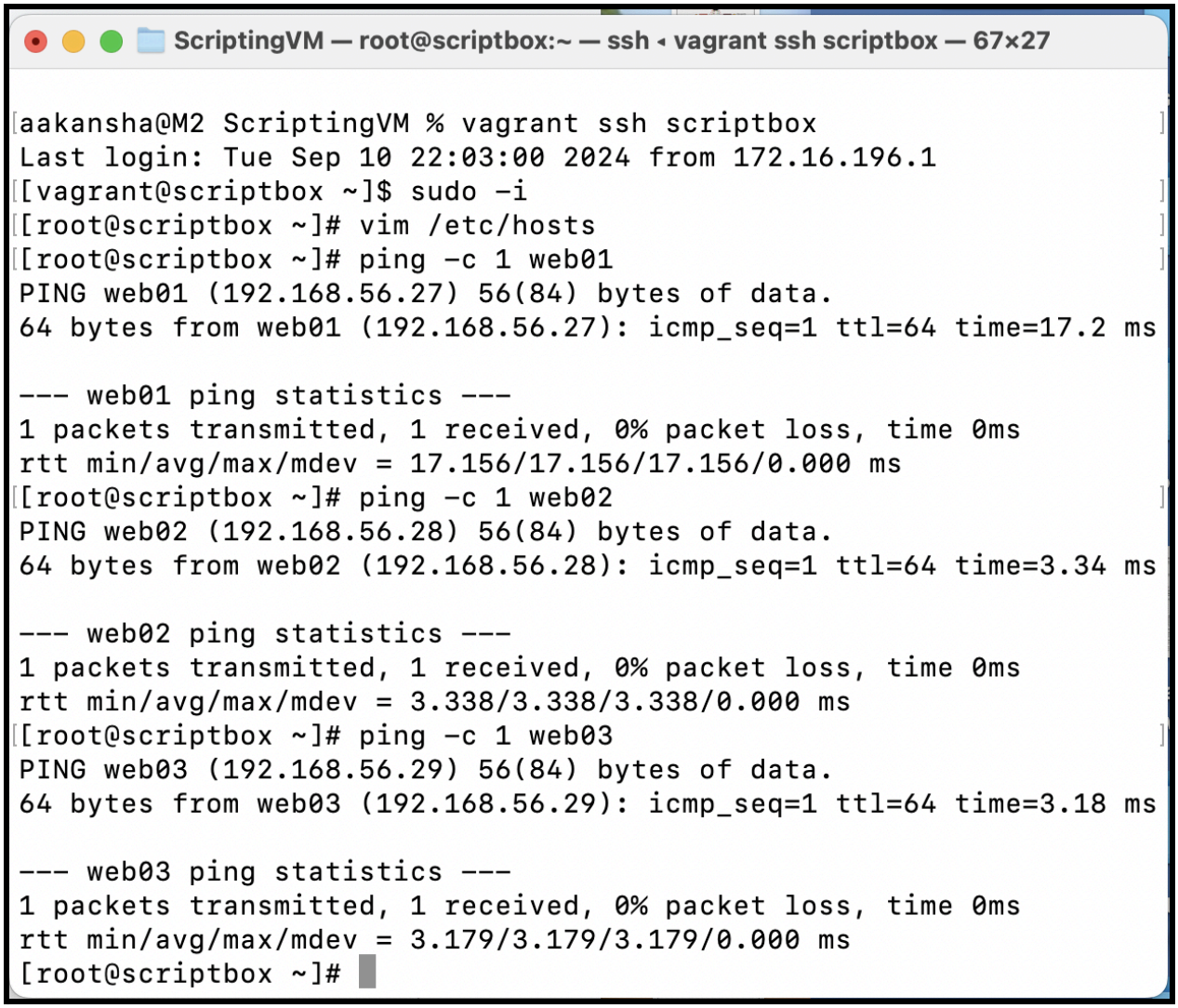
Let's start with checking their hostnames.



Now let’s take note of the IP addresses of these machines to map in the scriptbox



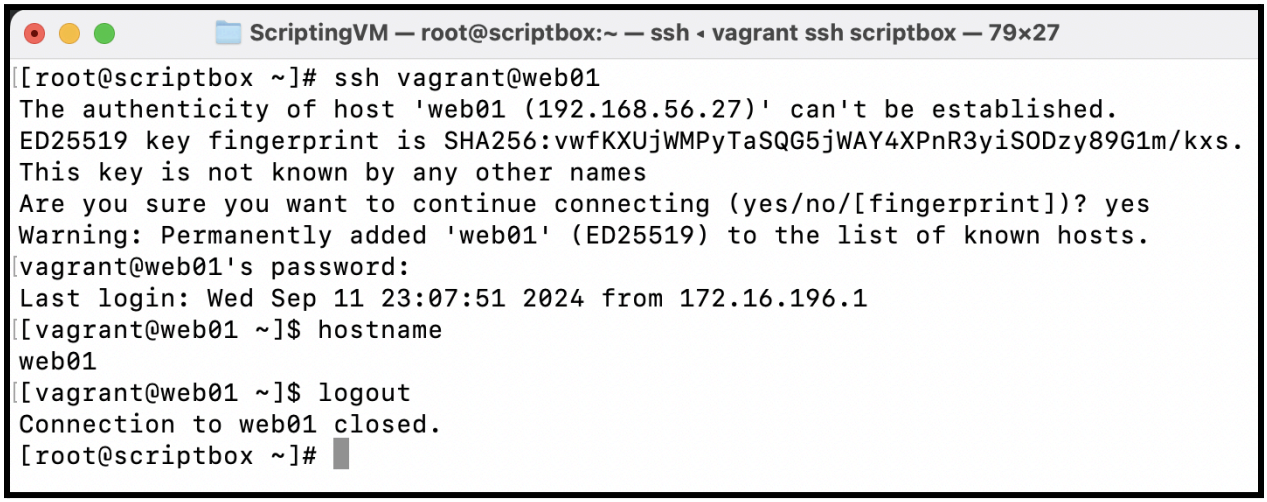
Now Let's save these into the host file of script box



The hostname is resolving successfully to IP addresses.

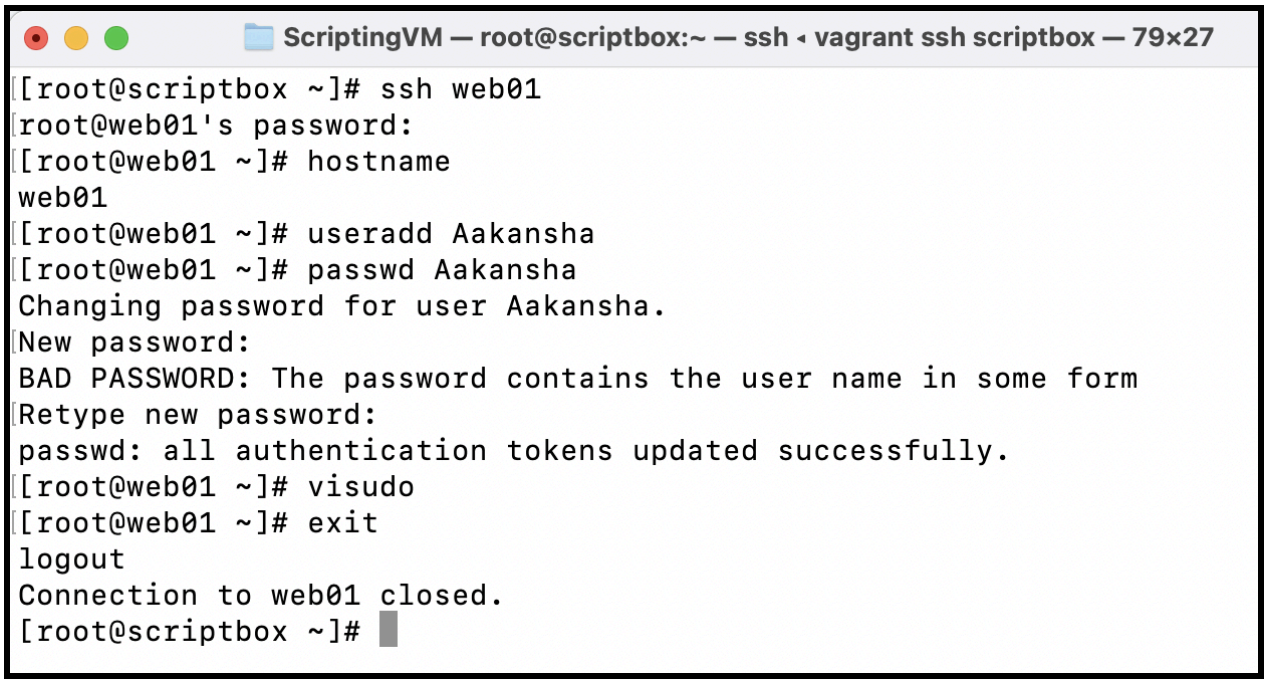
Now let's test the connection and enter the other VMS using ssh

Note: The password for a vagrant user is vagrant

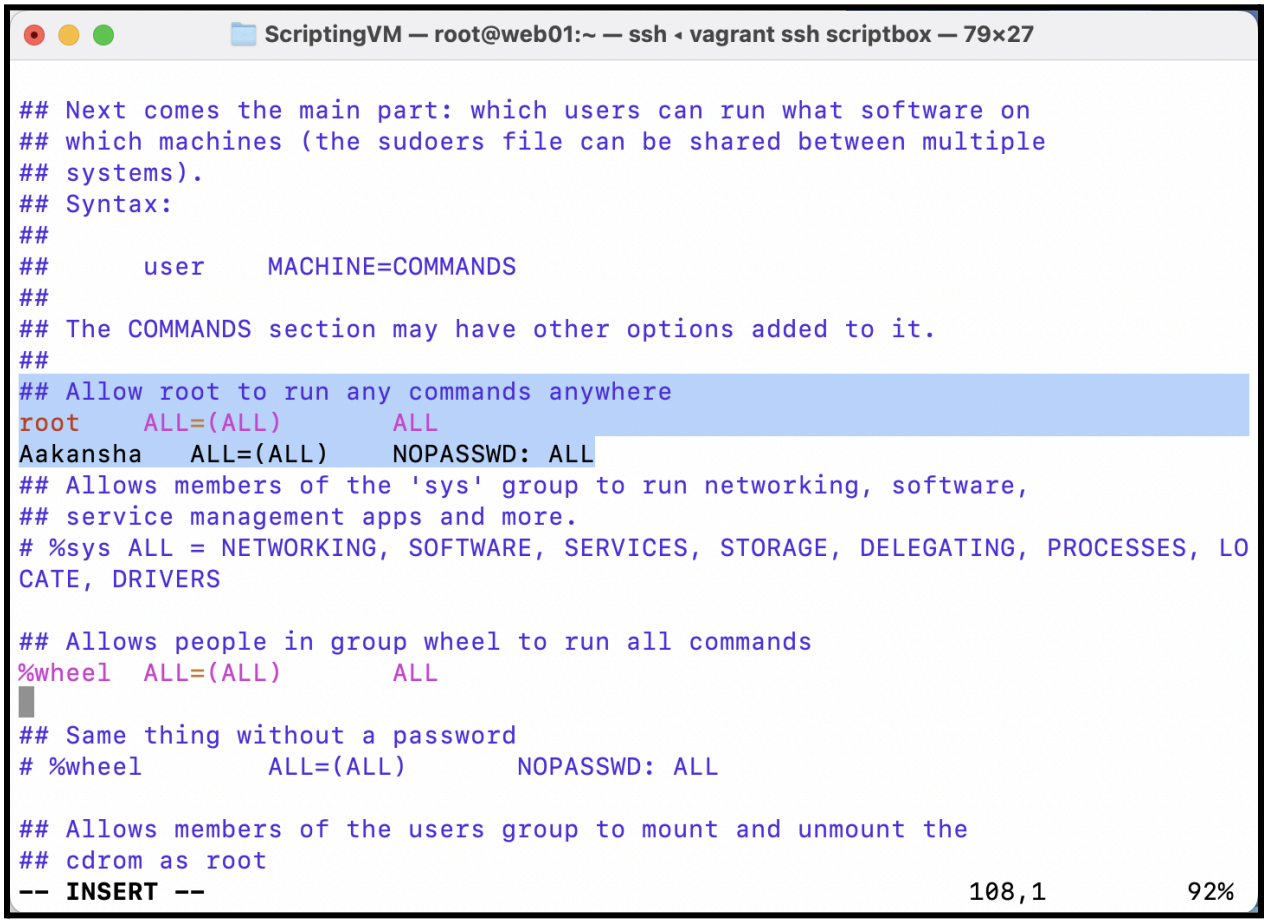


The test was successful. Let's move to the next part.

Now we will not be using the vagrant user. We will create our own user one by one in all the machines.
Also we will be using some system commands and install packages so we will be adding this user to the sudoers file.



Make the change in the file and save and quit



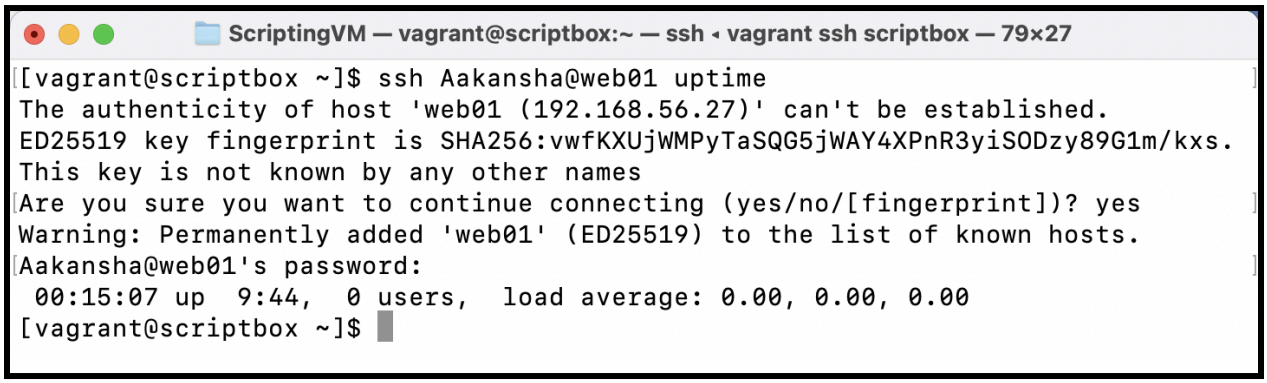
Repeat the same for the other VMS:

- **Web02**
- **Web03**

Note: If you are not able to enter web03 from scriptbox then exit out of it. Then:

- **Do vagrant ssh web03**
- **After getting into the vm open vim /etc/ssh/sshd_config**
- **Find PasswordAuthentication**
- **If it's no change it to yes**
    - **PasswordAuthentication yes**

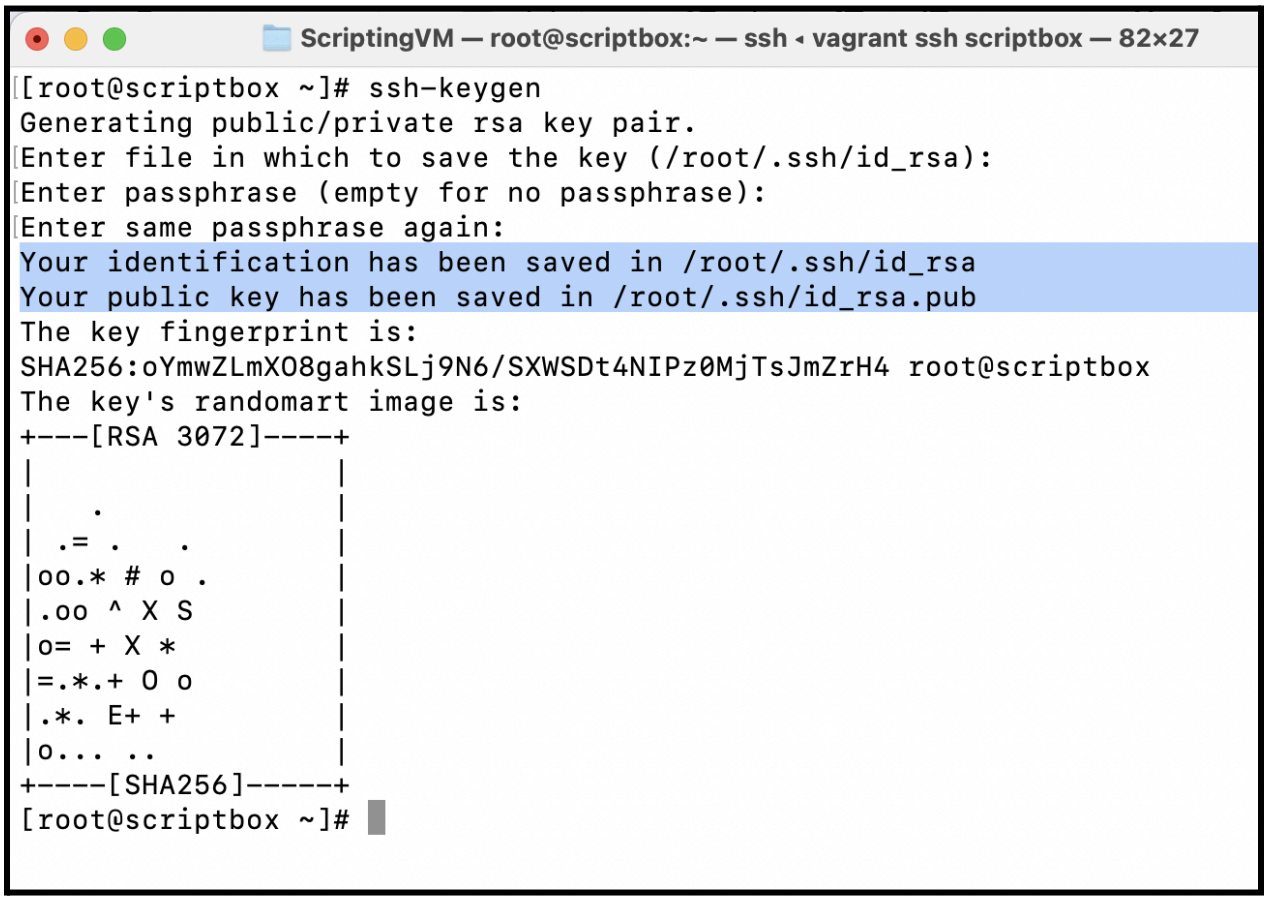
Now let's execute our command remotely



We have executed the command from our scriptbox in web01.

Problem: Asking us to authenticate every time using a password.

Solution: SSH Key Exchange



This is the private key: /root/.ssh/id_rsa

This is the public key: /root/.ssh/id_rsa.pub

(This is like a lock and key, so the public key acts as the lock and private key as the key to the lock.
We are going to put this lock on web01, web02, web03 and we will have this key to open them.)

Now let's lock the VMs with the public key

```
[root@scriptbox ~]# ssh-copy-id Aakansha@web01
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "/root/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now it is to install the new keys
Aakansha@web01's password:

Number of key(s) added: 1

Now try logging into the machine, with:   "ssh 'Aakansha@web01'"
and check to make sure that only the key(s) you wanted were added.
```

Note: Repeat the same for the other VMs

Now let's test it

```
ScriptingVM — root@scriptbox:~ — ssh ◂ vagrant ssh scriptbox — 75×27
[vagrant@scriptbox ~]$ ssh Aakansha@web01 uptime
 01:07:53 up 10:35,  0 users,  load average: 0.04, 0.01, 0.00
[vagrant@scriptbox ~]$ pwd
/home/vagrant
[vagrant@scriptbox ~]$ sudo -i
[root@scriptbox ~]# pwd
/root
[root@scriptbox ~]# ls -a
.                .bash_history  .bashrc   original-ks.cfg  .tcshrc
..               .bash_logout   .cshrc    scripts          .viminfo
anaconda-ks.cfg  .bash_profile  .lesshst  .ssh
[root@scriptbox ~]# ls .ssh/
id_rsa  id_rsa.pub  known_hosts  known_hosts.old
[root@scriptbox ~]#
```

It's not asking for the password now.
What's Happening is everytime we login it's using the id_rsa key to login and execute our command.

Practical Application of RCE

We were running our scripts in the scriptbox, now let's make a framework through which we will run our scripts on our other VMs.

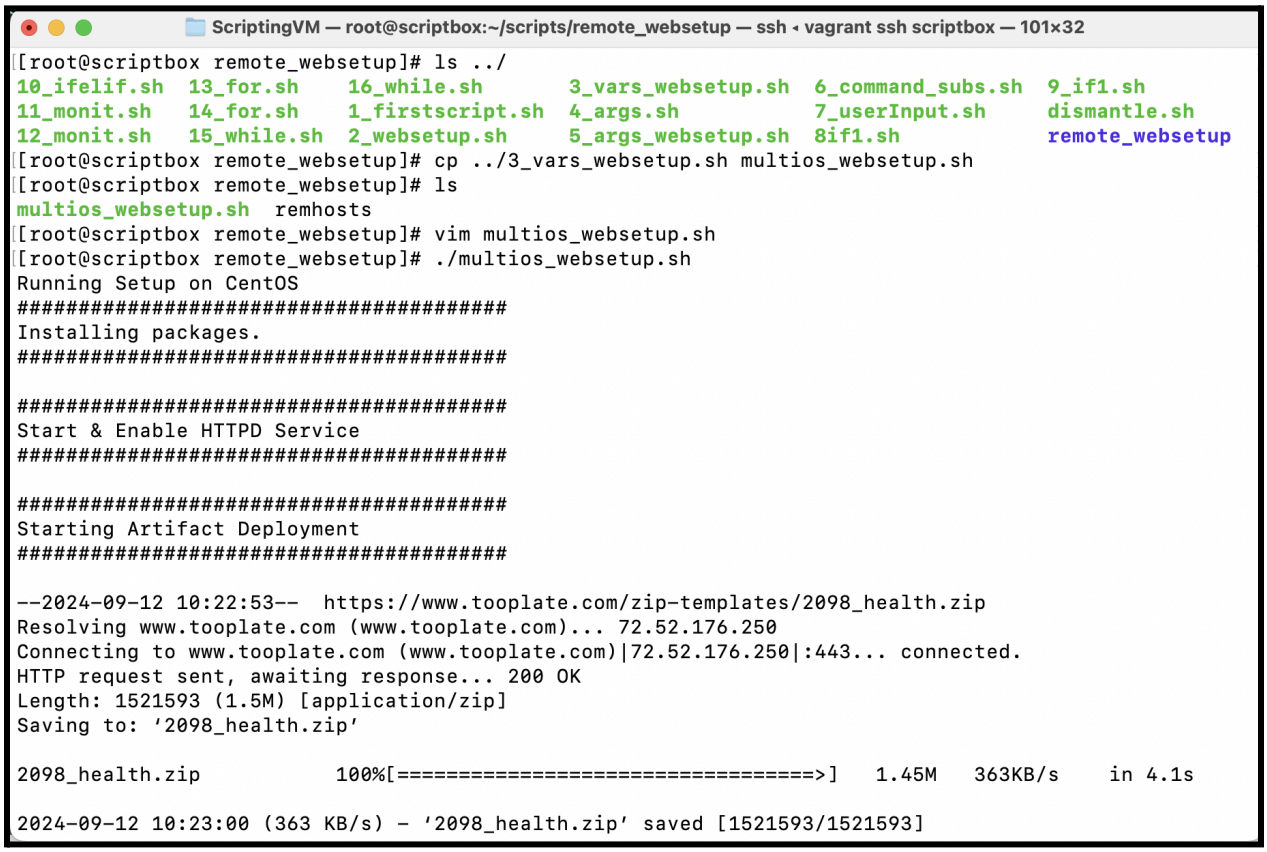
```
ScriptingVM — root@scriptbox:~/scripts/remote_websetup — ssh ◂ vagrant ssh scriptbox — 96×27
[[root@scriptbox ~]# ls
anaconda-ks.cfg  original-ks.cfg  scripts
[[root@scriptbox ~]# cd scripts/
[[root@scriptbox scripts]# ls
10_ifelif.sh  15_while.sh         4_args.sh           9_if1.sh
11_monit.sh   16_while.sh         5_args_websetup.sh  dismantle.sh
12_monit.sh   1_firstscript.sh    6_command_subs.sh
13_for.sh     2_websetup.sh       7_userInput.sh
14_for.sh     3_vars_websetup.sh  8if1.sh
[[root@scriptbox scripts]# mkdir remote_websetup
[[root@scriptbox scripts]# cd remote_websetup/
[[root@scriptbox remote_websetup]# ls
[[root@scriptbox remote_websetup]# vim remhosts
```

Now it's time to check the power of RCE

```
ScriptingVM — root@scriptbox:~/scripts/remote_websetup — ssh ◂ vagrant ssh scriptbox — 101×30
[[root@scriptbox remote_websetup]# ls
remhosts
[[root@scriptbox remote_websetup]# for host in `cat remhosts`; do ssh Aakansha@$host hostname;done
web01
web02
[Aakansha@web03's password:
web03
[[root@scriptbox remote_websetup]# for host in `cat remhosts`; do ssh Aakansha@$host uptime;done
 10:00:15 up 13:07,  0 users,  load average: 0.00, 0.00, 0.00
 10:00:15 up 13:07,  0 users,  load average: 0.01, 0.01, 0.00
[Aakansha@web03's password:
 17:00:20 up 13:06,  0 users,  load average: 0.00, 0.00, 0.00
[[root@scriptbox remote_websetup]# for host in `cat remhosts`; do ssh Aakansha@$host pwd;done
/home/Aakansha
/home/Aakansha
[Aakansha@web03's password:
/home/Aakansha
[[root@scriptbox remote_websetup]# for host in `cat remhosts`; do ssh Aakansha@$host free -m;done
               total        used        free      shared  buff/cache   available
Mem:            1950         221        1360           0         368        1704
Swap:           1949           0        1949
               total        used        free      shared  buff/cache   available
Mem:            1950         222        1356           0         372        1703
Swap:           1949           0        1949
[Aakansha@web03's password:
              total        used        free      shared  buff/cache   available
Mem:            719         166         100           1         453         458
Swap:          1437          17        1420
[root@scriptbox remote_websetup]#
```

We are able to check the hostname, uptime, working directory, and available memory on all servers remotely without needing to physically log into each server individually.

Let's modify our websetup script which was applicable to centos. Now we will make it applicable to other OS.



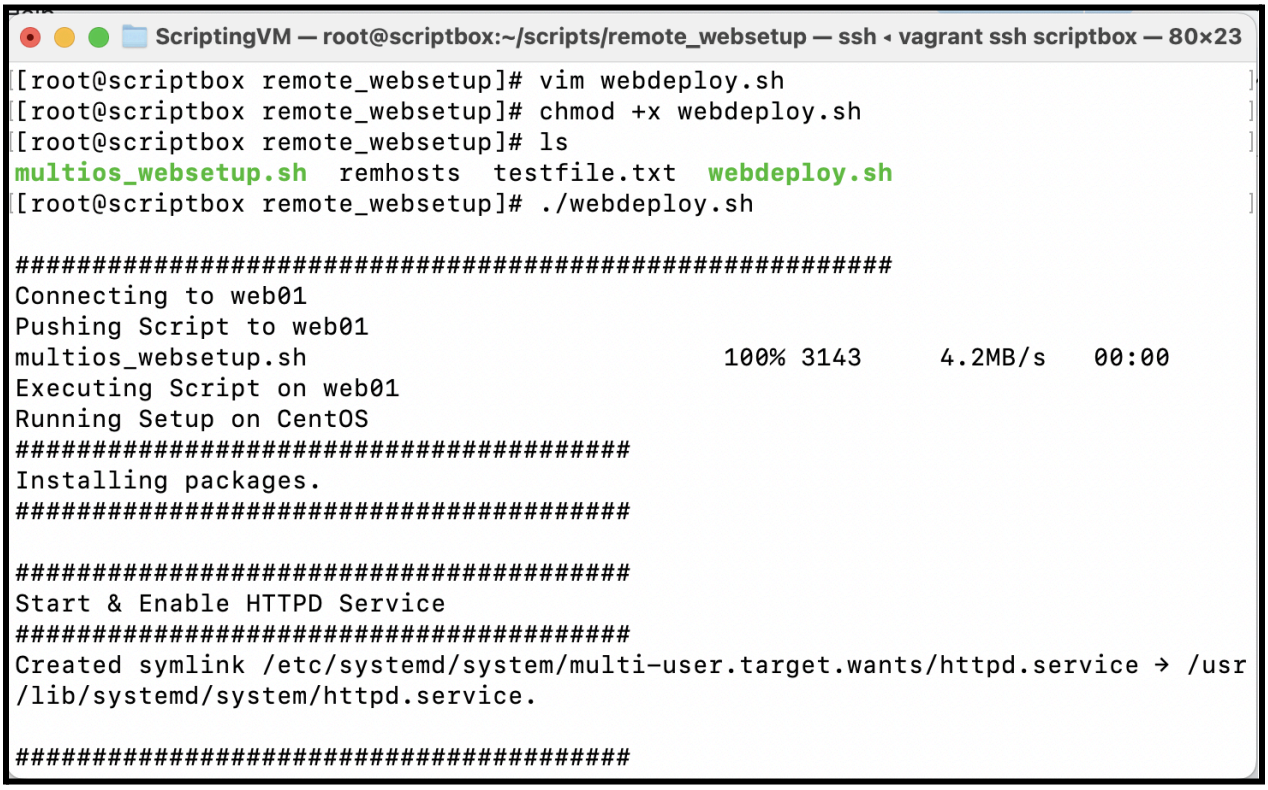
We have checked the script locally and it's working well.

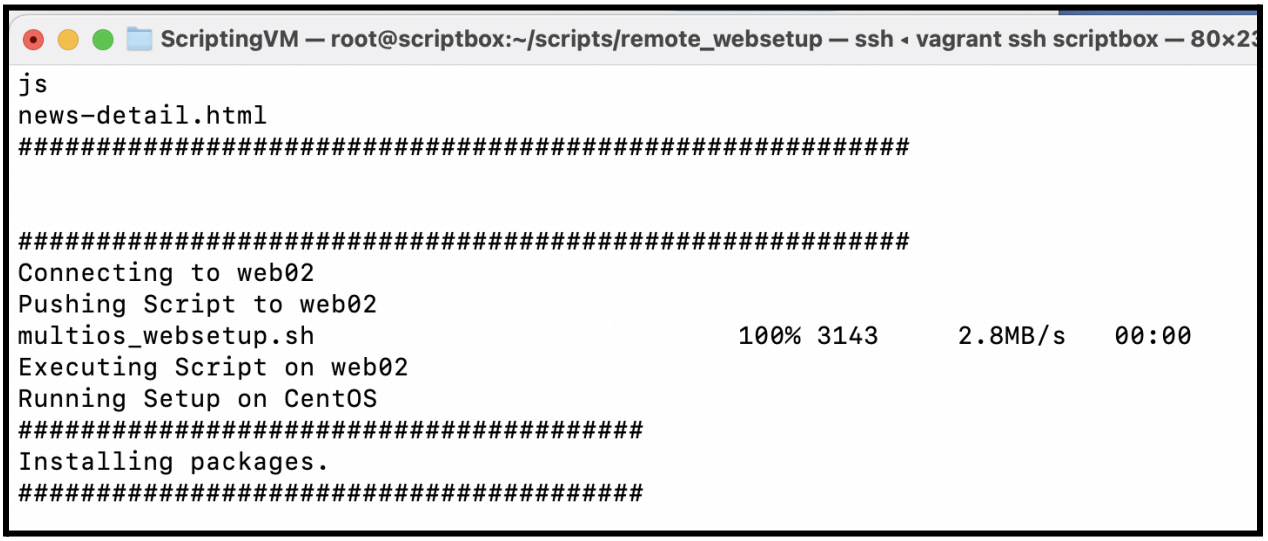
Now let's push it to our VM's.

The scp (Secure Copy Protocol) command is used to securely transfer files between a local host and a remote host, or between two remote hosts. It uses SSH to ensure that data is transferred securely.

In the above example, scp is used to securely copy a file (testfile.txt) from the local machine to a remote host (web01), specifically to the /tmp/ directory on that host.

Now, we will create a script to deploy our multios websetup script to our VMs





Working well with all the VMs. Have doubts …Go check for yourself😀

PROJECT CONCLUSION

Remote Command Execution (RCE) demonstrated the power of managing and controlling multiple systems from a single machine.

In the above examples, we have executed commands on multiple remote servers (web01, web02, web03) from the local machine using SSH.

Let's Summarize the Key Powers of RCE:

1. **Centralized Control:** The admin can log into multiple machines (web01, web02, web03) using a single script and perform administrative tasks like checking hostnames, uptime, or available memory.
2. **Automation:** With a simple loop, commands like hostname, uptime, and free -m are executed across multiple servers in sequence. This allows for quick monitoring and management, saving time compared to manually logging into each system.
3. **Consistency:** Commands executed via RCE ensure consistent results across systems, helping administrators standardize configurations and collect data uniformly.
4. **Security:** Though RCE provides great power, it requires proper authentication. Each time you connect to a server, SSH ensures a secure connection, prompting for a password if necessary.