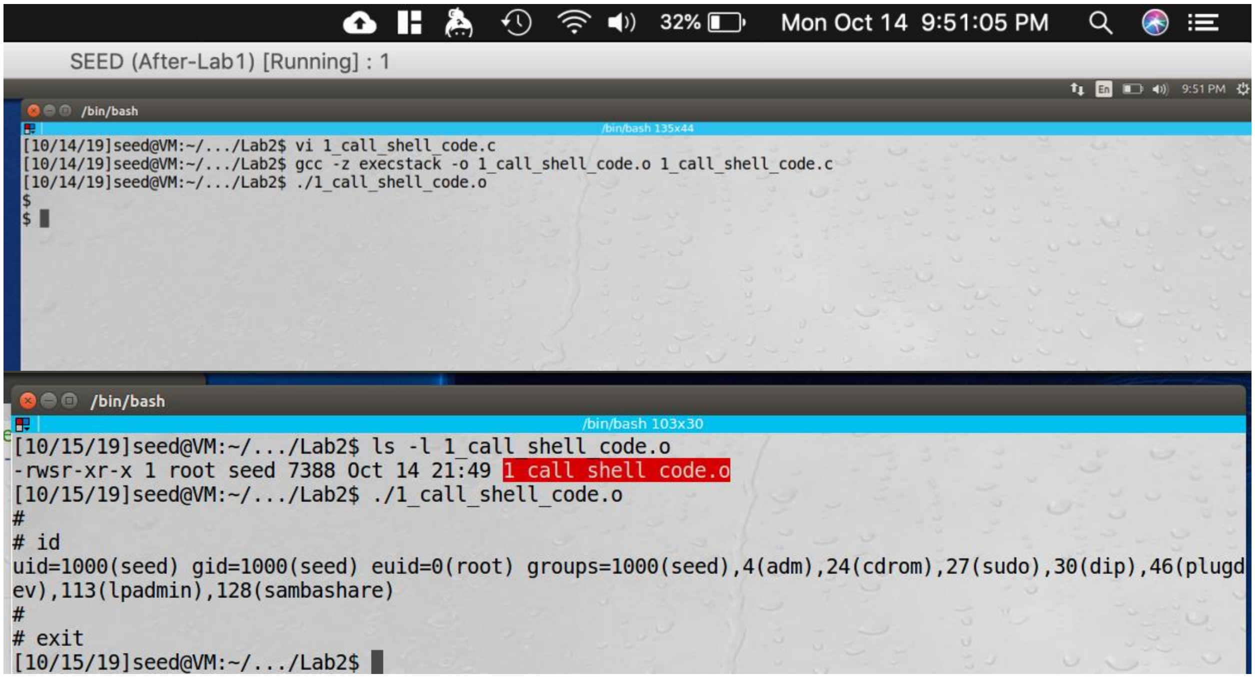
Name: Bharath Karumudi
Assignment: Lab 2 - Buffer Overflow Attack

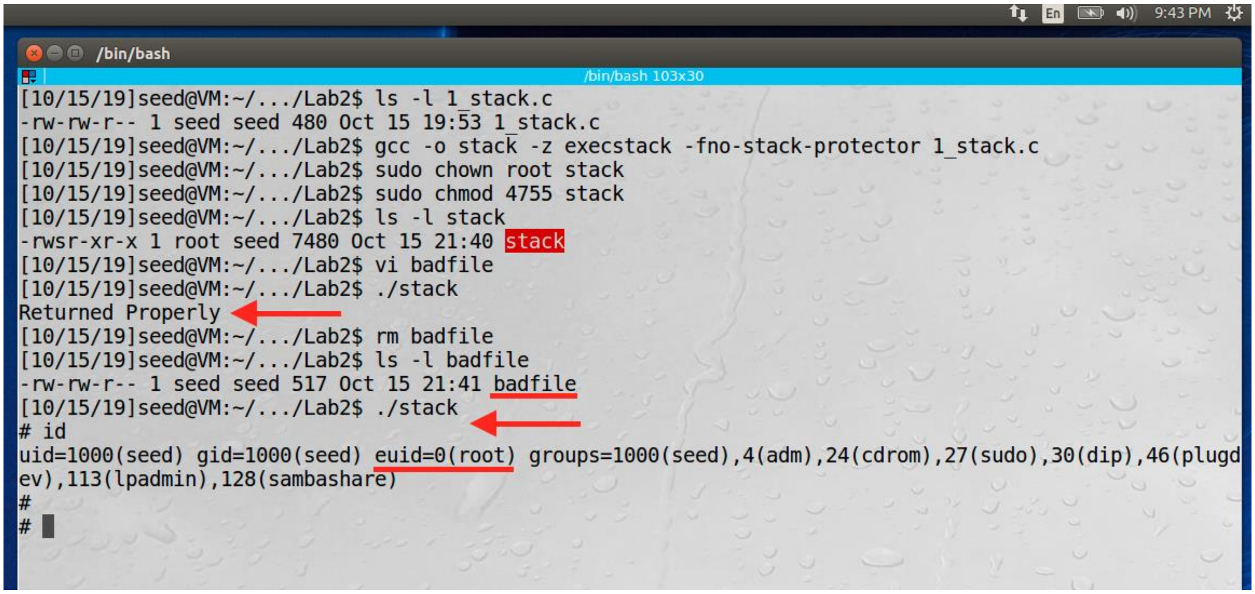
Task1: Running Shellcode



Observation: The given shell code was compiled with executable stack option (-z execstack) to allow the code to be executed from stack and when executed a new shell was created and we can see the new prompt on the terminal. If the same executable was made to set-UID root program, then I got the root shell.

Explanation: The shell code was pushed to the stack which is equivalent to execve("/bin/sh", name, NULL) was executed, which created a new shell.

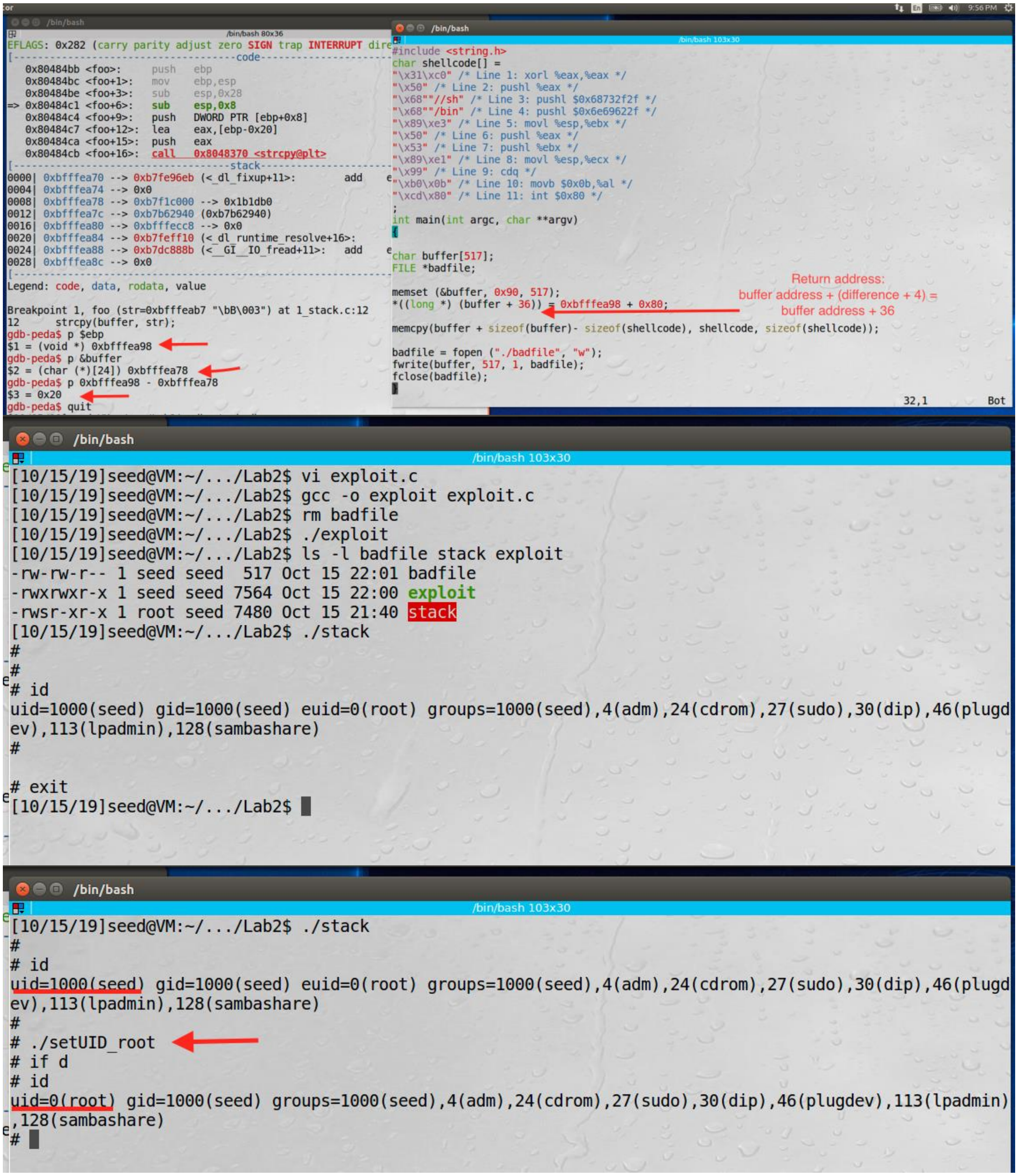
Task 1.1: The Vulnerable Program



Observation: Created the stack.c with the given code and compiled to an executable as stack by disabling the stack protector and enabling the stack executable as shown in the screenshot (-z execstack -fno-stack-protector). Then made the executable (file name: stack) as root owned set-UID program using chown and chmod.
When executed the stack program with a smaller input in the badfile, then it returned as "Returned Properly" and when injected the shell code into the badfile that has the malicious code as input to the program, we got the shell with root as effective id.

Explanation: Because the badfile has malicious code and it was added into the stack and updated the return address to execute the malicious code. So, the malicious code which has shell code, got executed and as the stack is a set-UID program, we got the shell with root privileges (effective id).

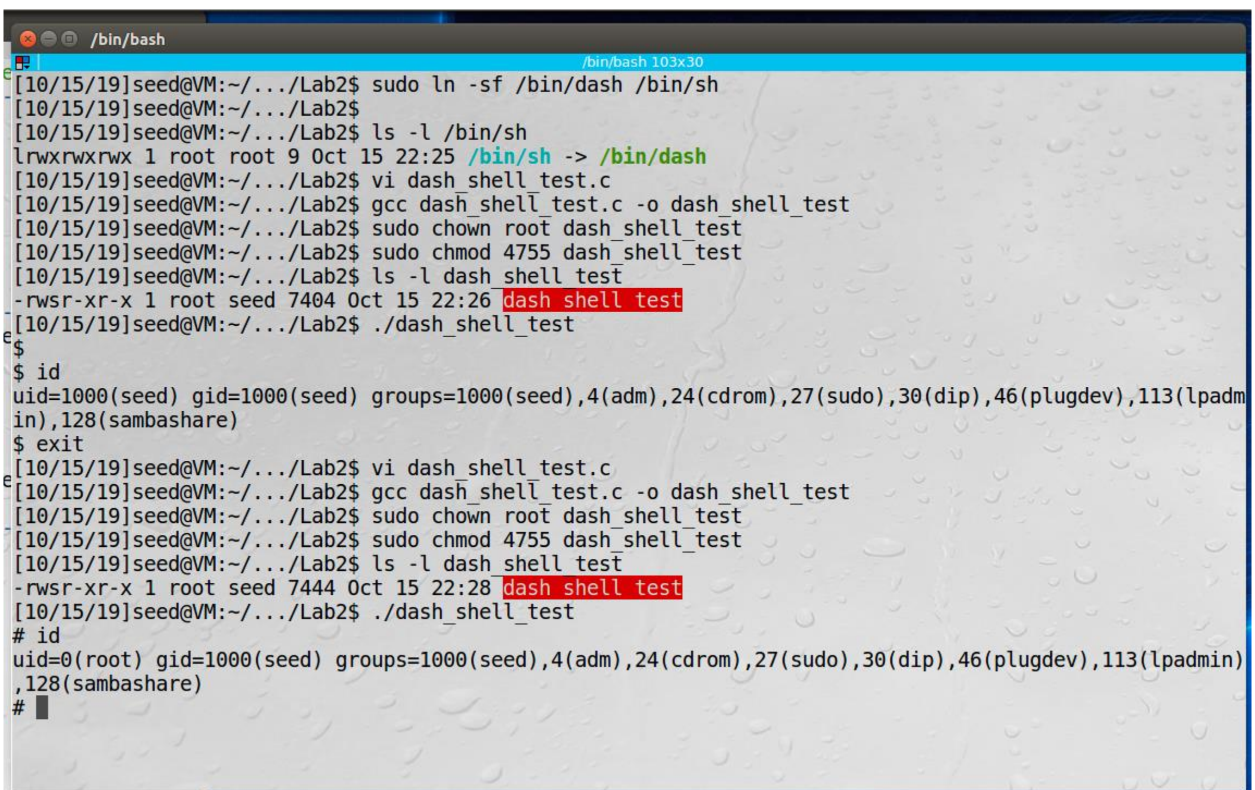
Task 2: Exploiting the Vulnerability



Observation:
Created the complete exploit.c file and compiled to exploit which will create the required badfile and also updates the return address to point to malicious content which was in the badfile.
The badfile created from exploit was given to the stack executable which is a root set-UID program and when executed, we got the root shell and verified with the id command.
(Image 3): Also once inside the root shell as effective id, by executing the setUID_root, which updates the setuid to zero and make a system call to “/bin/sh” and got a new shell with real user id as root.
Note: The return address was calculated using gdb as shown in the first screenshot (left), where the difference is between buffer and edp is 32(0x20), so adding 4 will be 36 which is return address. The new return address was set as 0xbffea98+0x80 where we pushed our malicious code.

Explanation: This was due to buffer overflow vulnerability. The content from badfile which has shell code was pushed to stack and the return address was pointed to execute the badfile content. The setuid(0) and a system call to “/bin/sh” created a new shell with real user id as root.

Task 3: Defeating dash’s Countermeasure



Observation: The /bin/sh symlink is now pointed to dash and created the given program as dash_shell_test by commenting the setuid(0) line, when executed as set-UID program, the shell prompt appeared as a normal user. But when uncommented the code setuid(0) and recompiled the program and executed as set-UID program, this time we can see root shell.

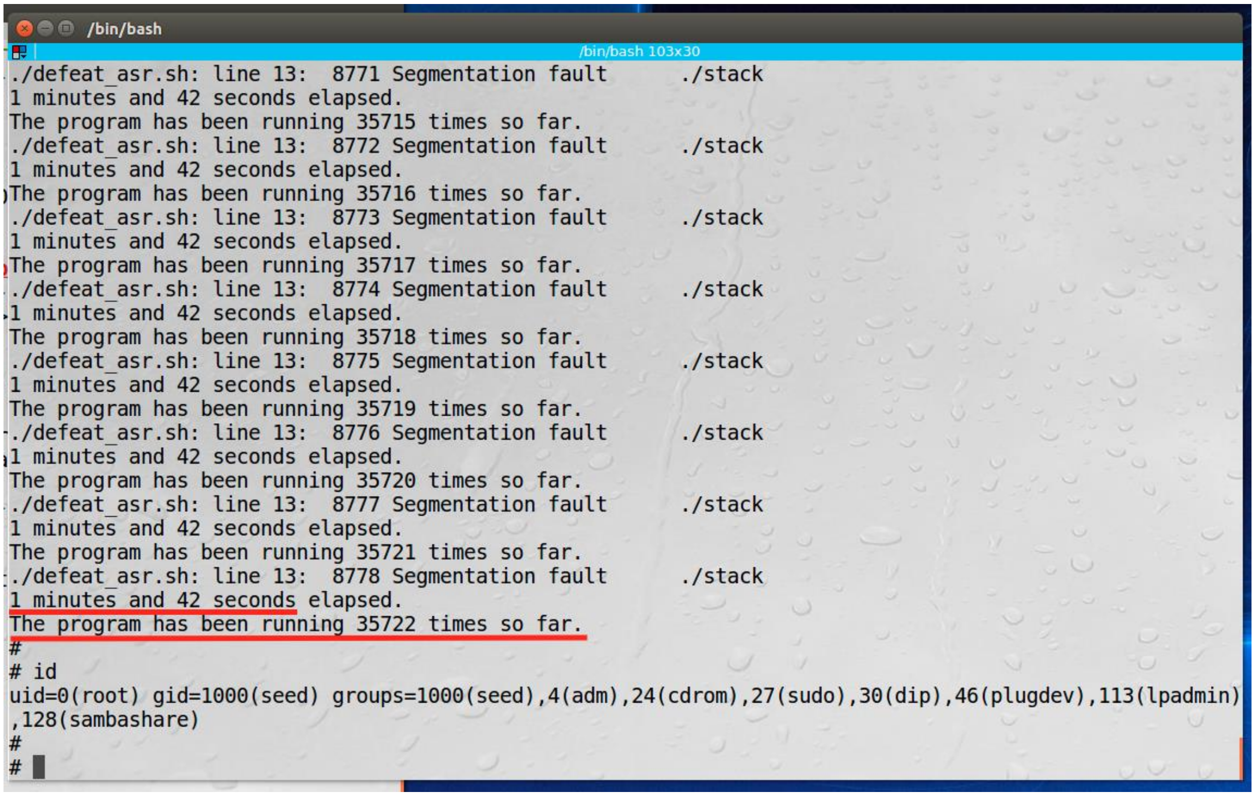
Explanation: The difference in the behavior is due to **setuid(0),** this made the real user id to zero and then we invoked the shell. So, dash was able to allow. Whereas in first case, dash identified that real id is not root and as a security measure, dash will not allow the root shell.

Task 3.1: Running exploit - Defeating dash's Countermeasure

Observation: A new exploit with the name dash_exploit.c was created with the new shell code given in instructions and created the badfile with the new "dash_exploit". The bad file was given to the stack program and when executed got the root shell with effective id also as root.

Explanation: The new shell code has setuid(0) so using the buffer overflow attack, we sent the malicious code to the stack and it got executed and as setuid was made to zero first, the dash countermeasure was bypassed and got the root shell.

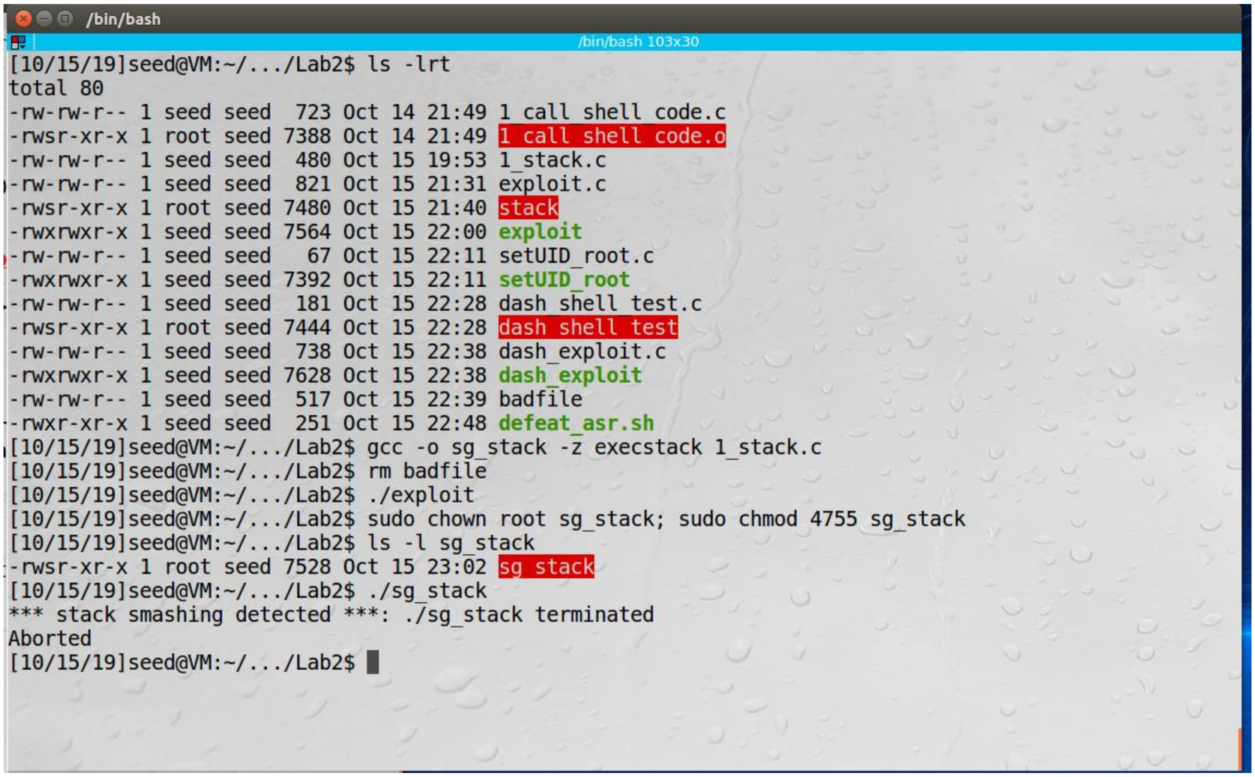
Task 4: Defeating Address Randomization



Observation: The address randomization was disabled and our brute force program: defeat_asr.sh ran for 35,722 times and in 1 min 42 seconds the brute force attack was completed and able to match the address where to execute our malicious code that stack program was introduced through badfile. We can see the root shell with root as real user id was appeared.

Explanation: On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 2^{19} = 524,288 possibilities. Our brute force program defeated in 35,722 runs and broke the address randomization countermeasure and able to match the address where to execute our malicious code that stack program was introduced through badfile.

Task 5: Turn on the StackGuard Protection



Observation: Created a new sg_stack file from Task 1 with StackGaurd protection (by compiling the program without -fno-stack-protector) and made the root owned set-UID program chown and chmod. When tried to do the exploit from Task 2 and executed got the error "*** stack smashing detected ***" and program terminated. The attack was failed.

Explanation: This was due to stack guard protection enabled on the stack and it identified the change in the stack guard value which occurred due to overflow. Once stack identified the change, it terminated the program as a countermeasure.

Task 6: Turn on the Non-executable Stack Protection

```
/bin/bash
/bin/bash 103x30
[10/15/19]seed@VM:~/.../Lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/15/19]seed@VM:~/.../Lab2$ ls -lrt
total 88
-rw-rw-r-- 1 seed seed  723 Oct 14 21:49 1_call_shell_code.c
-rwsr-xr-x 1 root seed 7388 Oct 14 21:49 1_call_shell_code.o
-rw-rw-r-- 1 seed seed  480 Oct 15 19:53 1_stack.c
-rw-rw-r-- 1 seed seed  821 Oct 15 21:31 exploit.c
-rwsr-xr-x 1 root seed 7480 Oct 15 21:40 stack
-rwxrwxr-x 1 seed seed 7564 Oct 15 22:00 exploit
-rw-rw-r-- 1 seed seed   67 Oct 15 22:11 setUID_root.c
-rwxrwxr-x 1 seed seed 7392 Oct 15 22:11 setUID_root
-rw-rw-r-- 1 seed seed  181 Oct 15 22:28 dash_shell_test.c
-rwsr-xr-x 1 root seed 7444 Oct 15 22:28 dash_shell_test
-rw-rw-r-- 1 seed seed  738 Oct 15 22:38 dash_exploit.c
-rwxrwxr-x 1 seed seed 7628 Oct 15 22:38 dash_exploit
-rwxr-xr-x 1 seed seed  251 Oct 15 22:48 defeat_asr.sh
-rwsr-xr-x 1 root seed 7528 Oct 15 23:02 sg_stack
-rw-rw-r-- 1 seed seed  517 Oct 15 23:02 badfile
[10/15/19]seed@VM:~/.../Lab2$ fcc -o nes_stack -fno-stack-protector -z noexecstack 1_stack.c
The program 'fcc' is currently not installed. You can install it by typing:
sudo apt install fcc
[10/15/19]seed@VM:~/.../Lab2$ gcc -o nes_stack -fno-stack-protector -z noexecstack 1_stack.c
[10/15/19]seed@VM:~/.../Lab2$ sudo chown root nes_stack; sudo chmod 4755 nes_stack
[10/15/19]seed@VM:~/.../Lab2$ ls -l nes_stack
-rwsr-xr-x 1 root seed 7480 Oct 15 23:13 nes_stack
[10/15/19]seed@VM:~/.../Lab2$ ./nes_stack
Segmentation fault
[10/15/19]seed@VM:~/.../Lab2$
```

Observation: When recompiled the program by disabling the address randomization using sysctl -w kernel.randomize_va_space=0 and enabling the non-executable stack protection (-z noexecstack) , the program compiled and when tried the attack as in Task 2, encountered the Segmentation fault and attack failed.

Explanation: Because the Stack is now enabled as Non executable, the malicious code in the stack will not be executed. So, we got the Segmentation fault as a countermeasure.