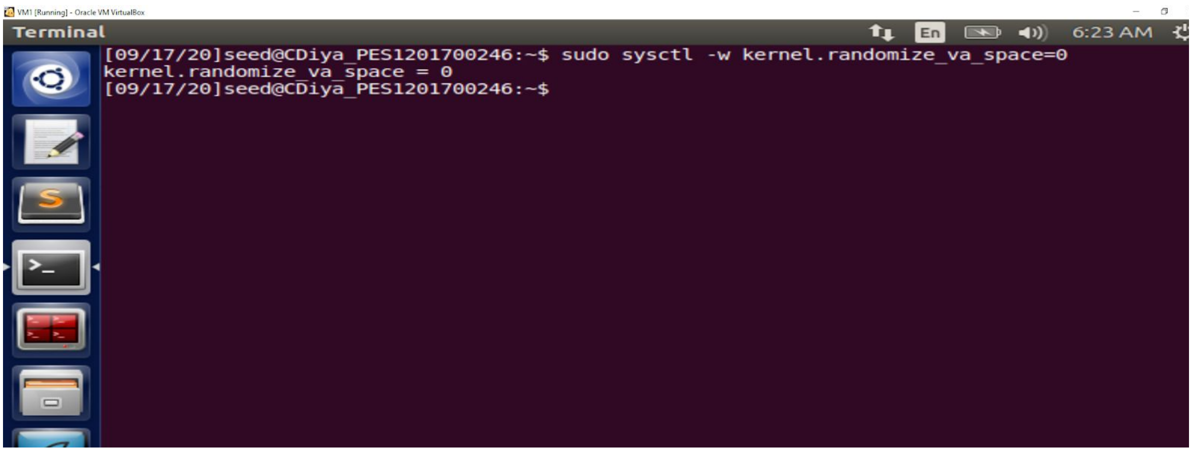
Information Security
LABORATORY 3

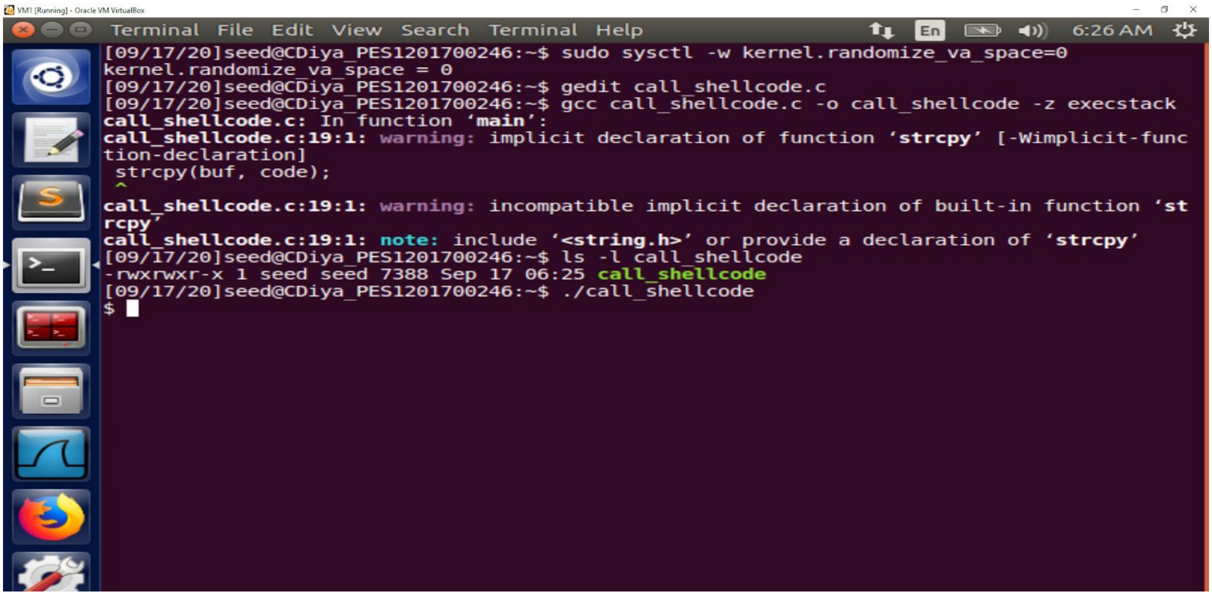
Buffer Overflow Vulnerability Lab

C Diya
PES1201700246

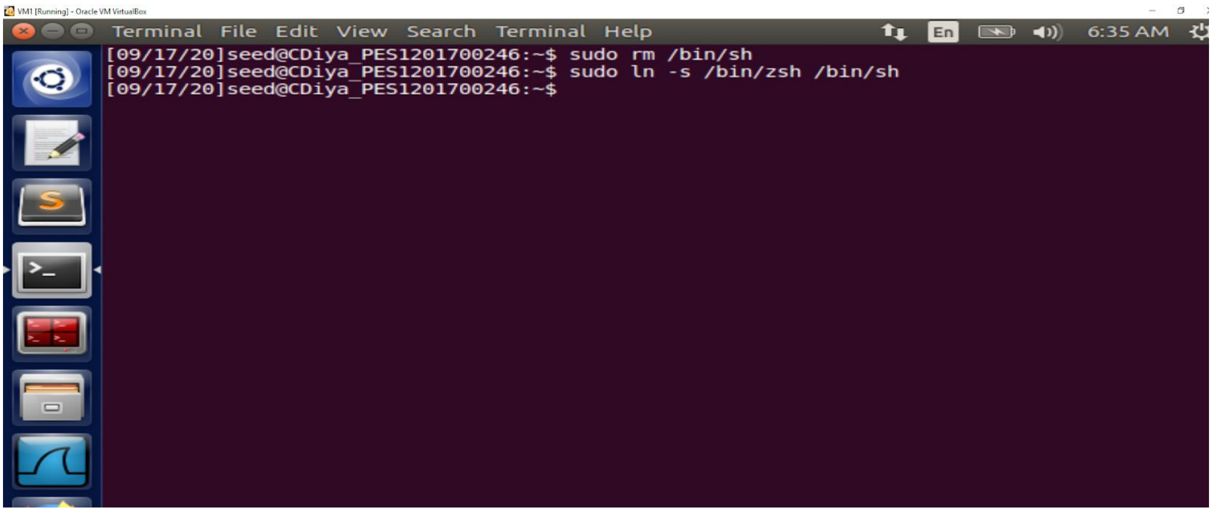
Task 1: Turning Off Countermeasures



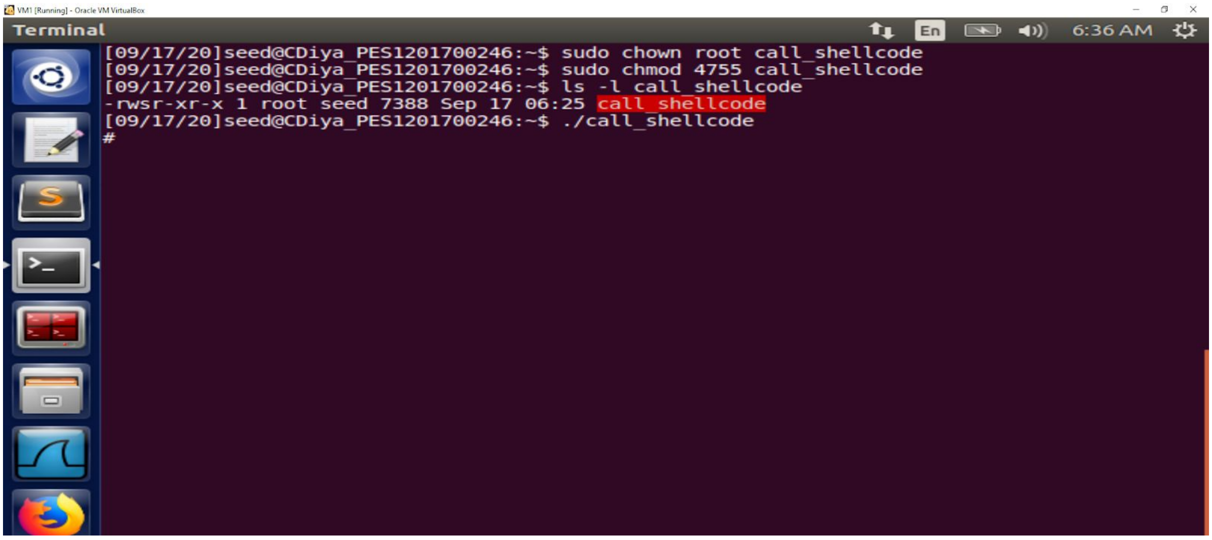
Observation: The screenshot above shows the disabling of the address space randomization measure so as to ensure that attacks occur.
Linux based systems implement several security mechanisms to make the buffer-overflow attack difficult. To simplify the attacks, these measures should be disabled. Ubuntu and Linux-based systems use address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult.



Observation: The code above shows the creation of a shellcode. A shellcode is the code to launch a shell. While compiling the program, the execstack option must be used, which allows code to be executed from the stack, without which the program will fail. The '$' which is the shell mode(not in root mode) is generated when the program is executed.

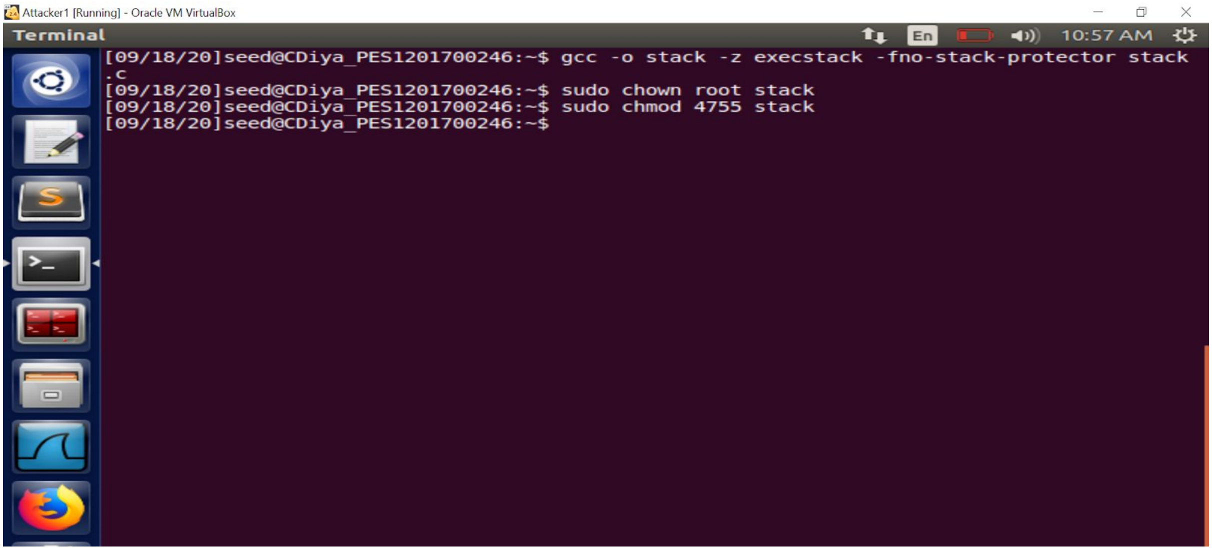


Observation: /bin/sh is a symbolic link that points to bin/dash. The dash shell has a countermeasure that prevents itself from being executed in a Set-UID process.Since /bin/sh makes the attack more difficult , /bin/sh is made to point to another shell zsh.



Observation: The shellcode is executed in root mode using chown root and 4755 to gain shell in root mode. The '#' generated on execution of the above program shows that the shell has gained root mode.

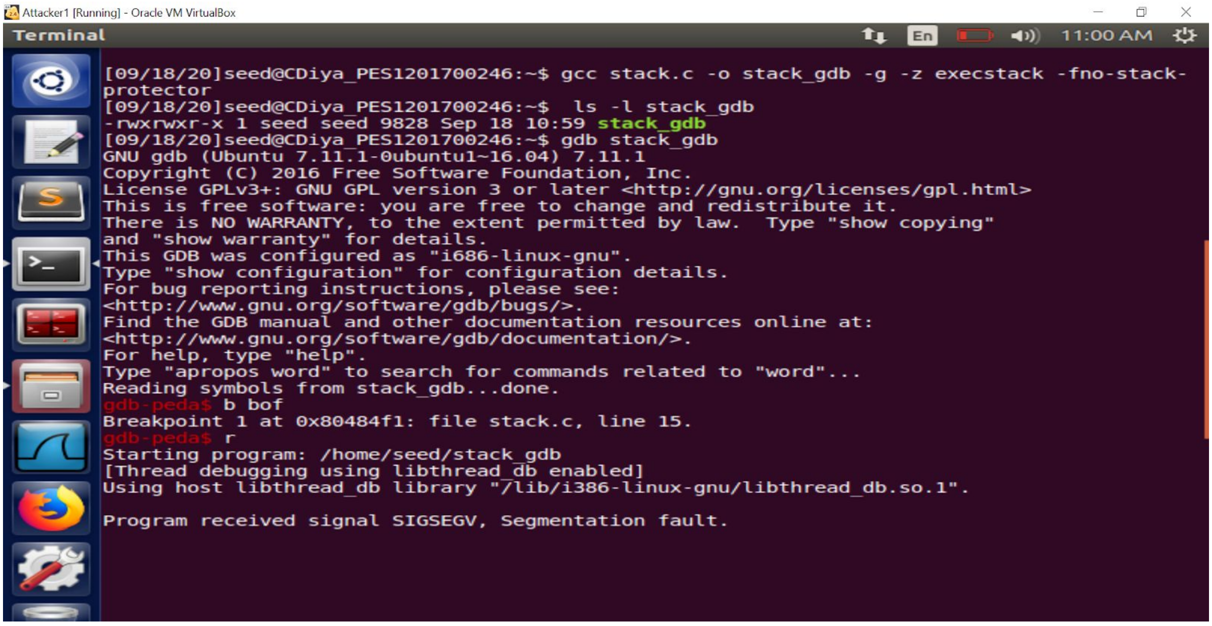
Task 2: Vulnerable Program



Observation: The above program is compiled while ensuring to disable the StackGuard and the non-executable stack protections using the -fno-stack-protector and "-z execstack" options. After the compilation, the program is made a root-owned Set-UID program.
The program above uses strcpy() which does not check boundaries, thus, buffer overflow will occur which results in a buffer overflow vulnerability. Since

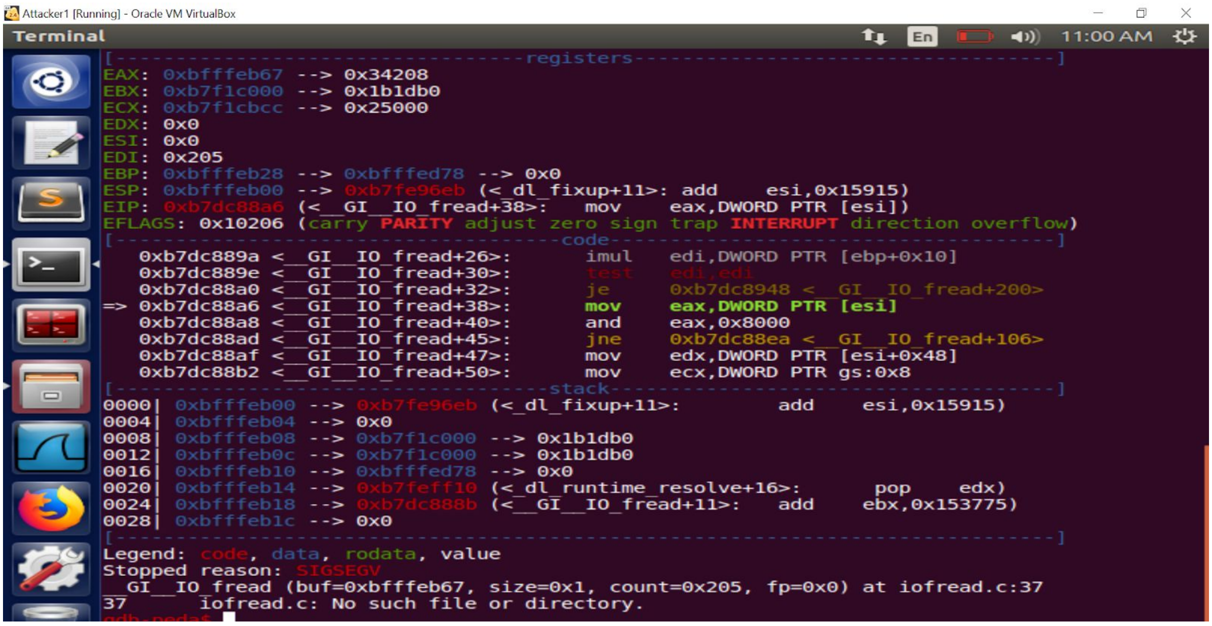
this program is a root-owned Set-UID program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. The program creates the contents for a badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be generated.

Task 3: Exploiting the Vulnerability



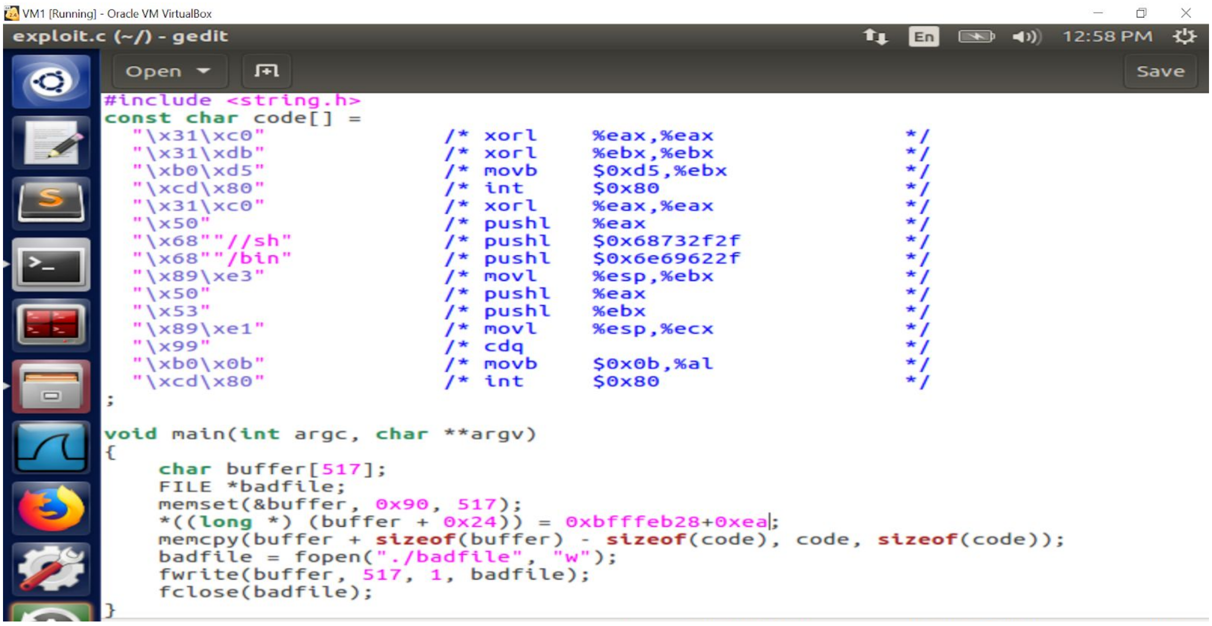
```
[09/18/20]seed@CDiya_PES1201700246:~$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
[09/18/20]seed@CDiya_PES1201700246:~$  ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9828 Sep 18 10:59 stack_gdb
[09/18/20]seed@CDiya_PES1201700246:~$ gdb stack_gdb
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 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_gdb...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 15.
gdb-peda$ r
Starting program: /home/seed/stack_gdb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
```

Observation: To find the address of the buffer variable in the bof() method, the above commands are executed. The stack program and gdb are used.

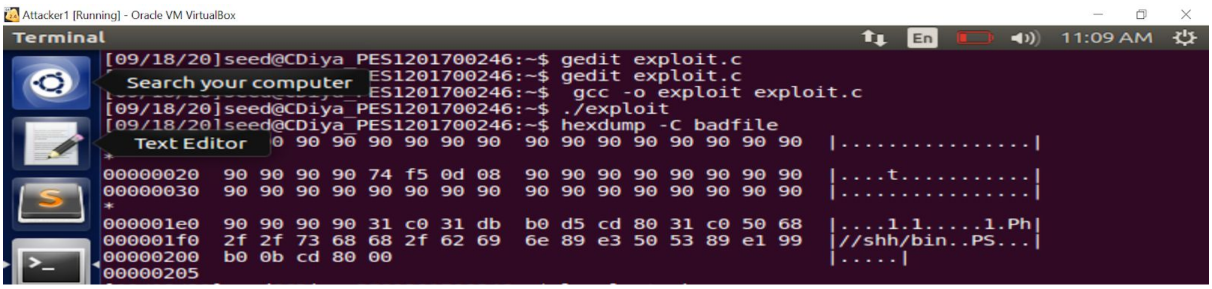


```
[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb67 --> 0x34208
EBX: 0xb7f1c000 --> 0x1b1db0
ECX: 0xb7f1cbcc --> 0x25000
EDX: 0x0
ESI: 0x0
EDI: 0x205
EBP: 0xbfffeb28 --> 0xbfffed78 --> 0x0
ESP: 0xbfffeb00 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0xb7dc88a6 (<__GI__IO_fread+38>:   mov    eax,DWORD PTR [esi])
EFLAGS: 0x10206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0xb7dc889a <__GI__IO_fread+26>:      imul   edi,DWORD PTR [ebp+0x10]
   0xb7dc889e <__GI__IO_fread+30>:      test   edi,edi
   0xb7dc88a0 <__GI__IO_fread+32>:      je     0xb7dc8948 <__GI__IO_fread+200>
=> 0xb7dc88a6 <__GI__IO_fread+38>:      mov    eax,DWORD PTR [esi]
   0xb7dc88a8 <__GI__IO_fread+40>:      and    eax,0x8000
   0xb7dc88ad <__GI__IO_fread+45>:      jne    0xb7dc88ea <__GI__IO_fread+106>
   0xb7dc88af <__GI__IO_fread+47>:      mov    edx,DWORD PTR [esi+0x48]
   0xb7dc88b2 <__GI__IO_fread+50>:      mov    ecx,DWORD PTR gs:0x8
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb00 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb04 --> 0x0
0008| 0xbfffeb08 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb0c --> 0xb7f1c000 --> 0x1b1db0
0016| 0xbfffeb10 --> 0xbfffed78 --> 0x0
0020| 0xbfffeb14 --> 0xb7feff10 (<_dl_runtime_resolve+16>:      pop    edx)
0024| 0xbfffeb18 --> 0xb7dc888b (<__GI__IO_fread+11>:   add    ebx,0x153775)
0028| 0xbfffeb1c --> 0x0
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
__GI__IO_fread (buf=0xbfffeb67, size=0x1, count=0x205, fp=0x0) at iofread.c:37
37      iofread.c: No such file or directory.
gdb-peda$
```

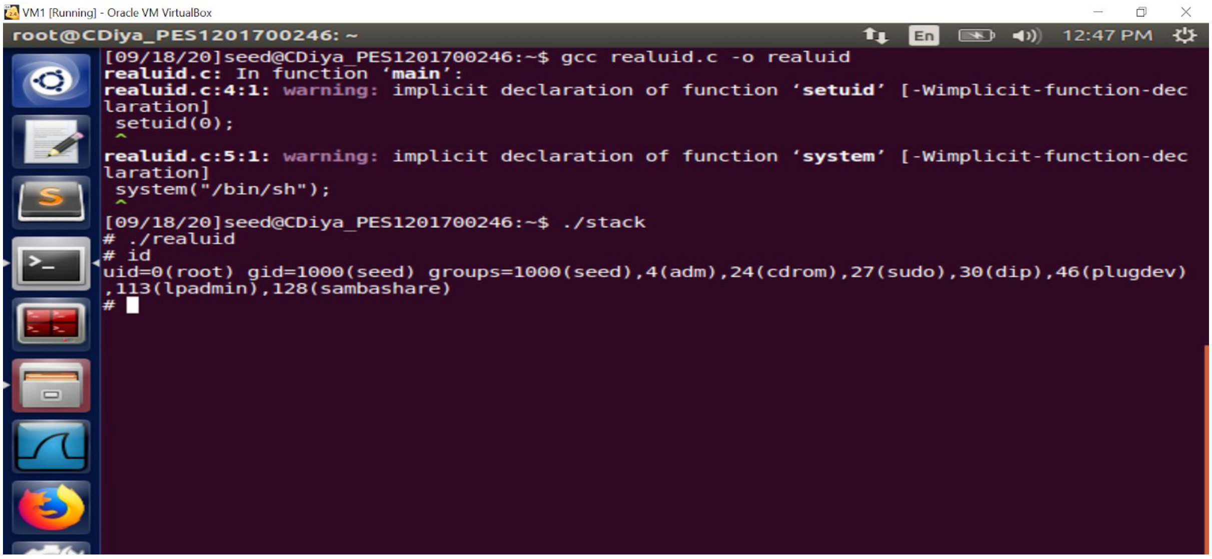
Observation: The $ebp, $buffer value is printed out.



Observation: The $ebp value is is added to the exploit.c program



Observation: The program above uses the exploit.c program which constructs the contents of badfile. Thus, on compiling the exploit.c program, the badfile content has been created as shown above.

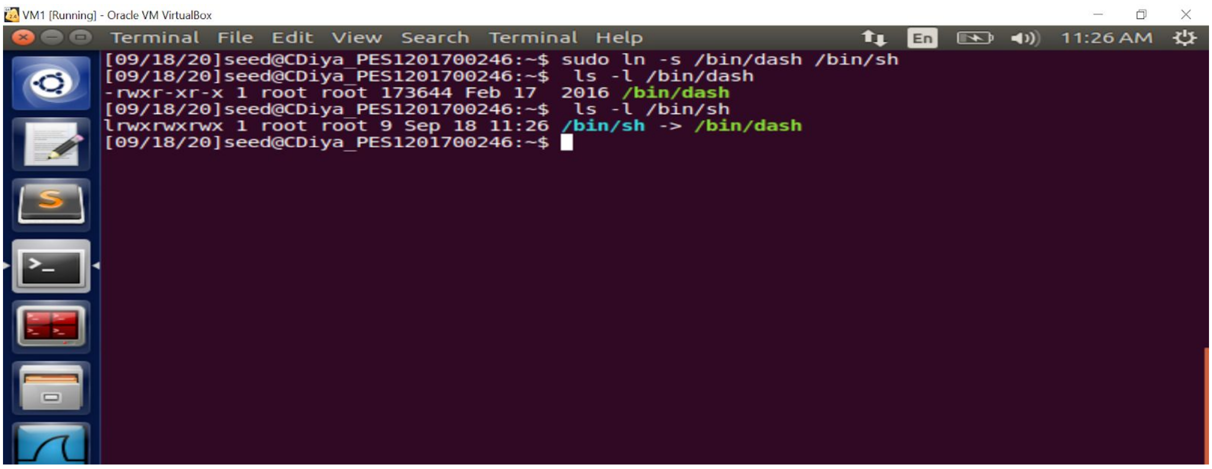


Observation: the uid=0 is observed on realuid

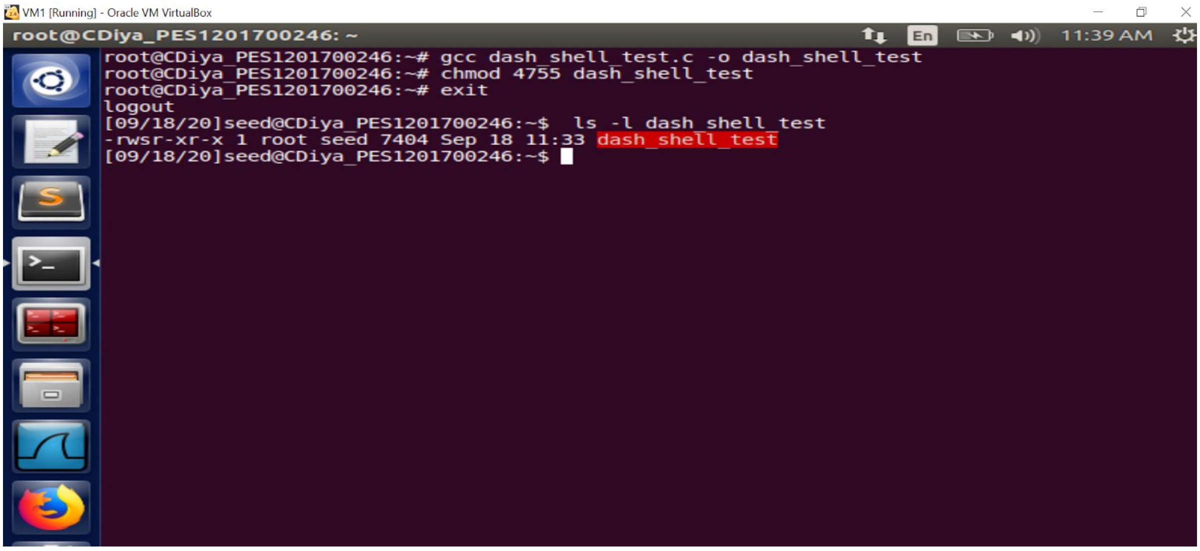
Then run the vulnerable program stack. The buffer will be overflowed in stack.c, which is compiled with the StackGuard protection disabled.

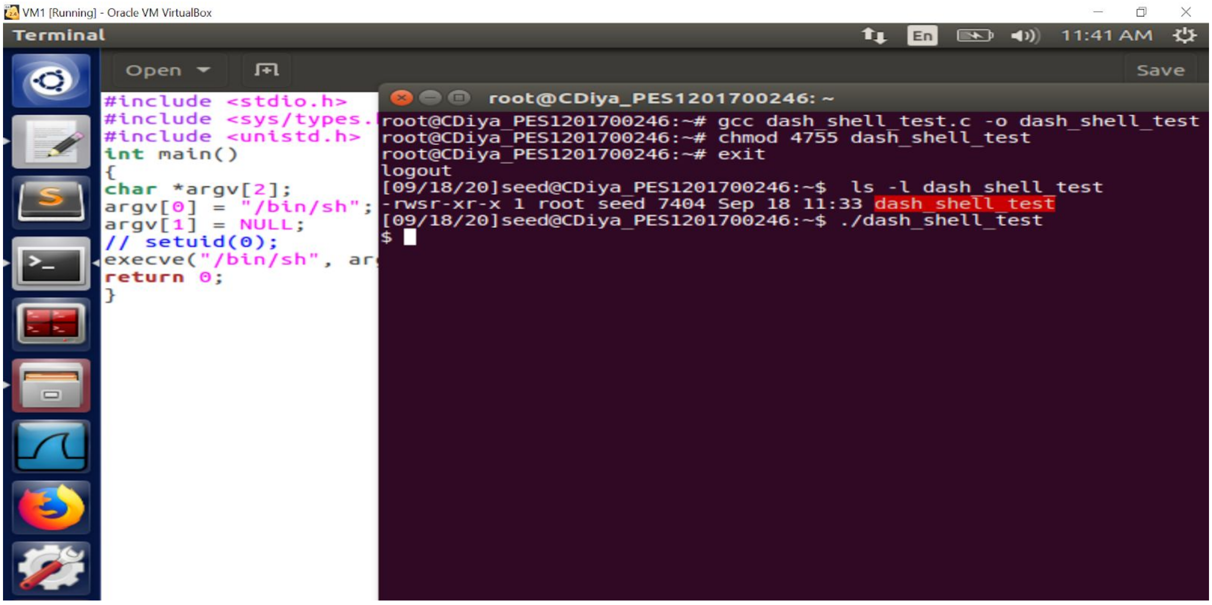
Many commands will behave differently if they are executed as Set-UID root processes, instead of just as root processes, because they recognize that the real user id is not root. Thus, the aim is to run the following program to turn the real user id to root. This would generate a real root process

Task 4: Defeating dash's Countermeasure

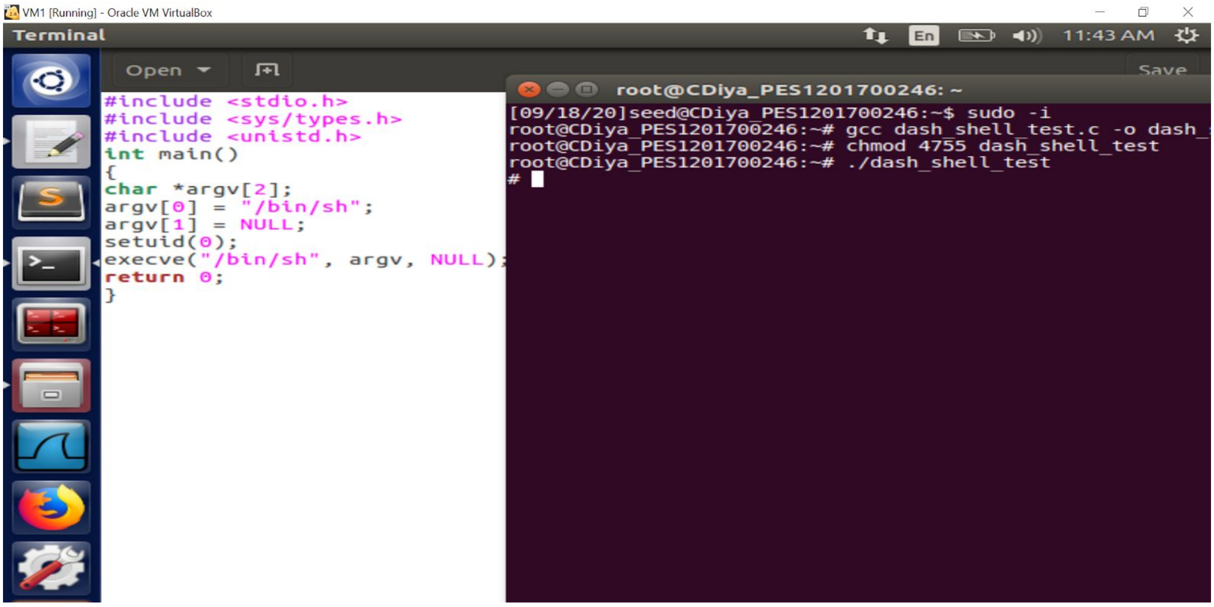


Observation: The link from /bin/sh points back to bin/dash. The dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID.The countermeasure implemented in dash can be defeated which will be tested.



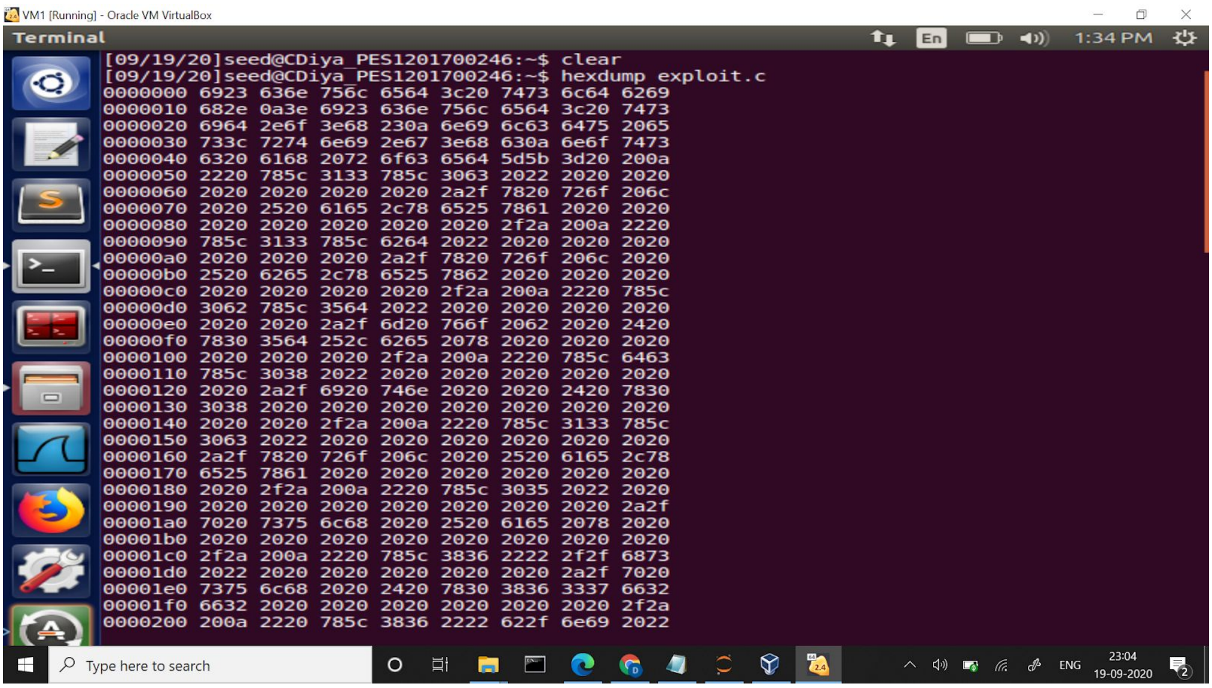


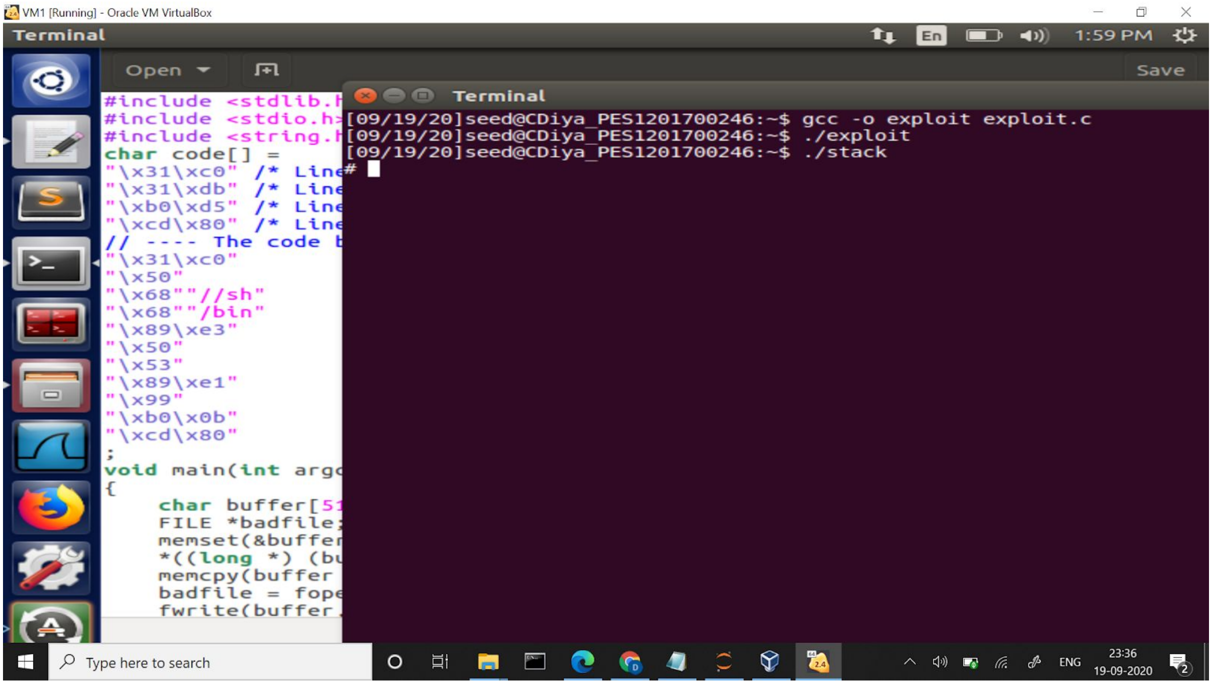
Observation: When setuid(0) is commented, it can be seen that $ appears which shows that the shell is not in root mode.



Observation: When setuid(0) is uncommented, it can be seen that # appears which shows that the shell is in root mode.

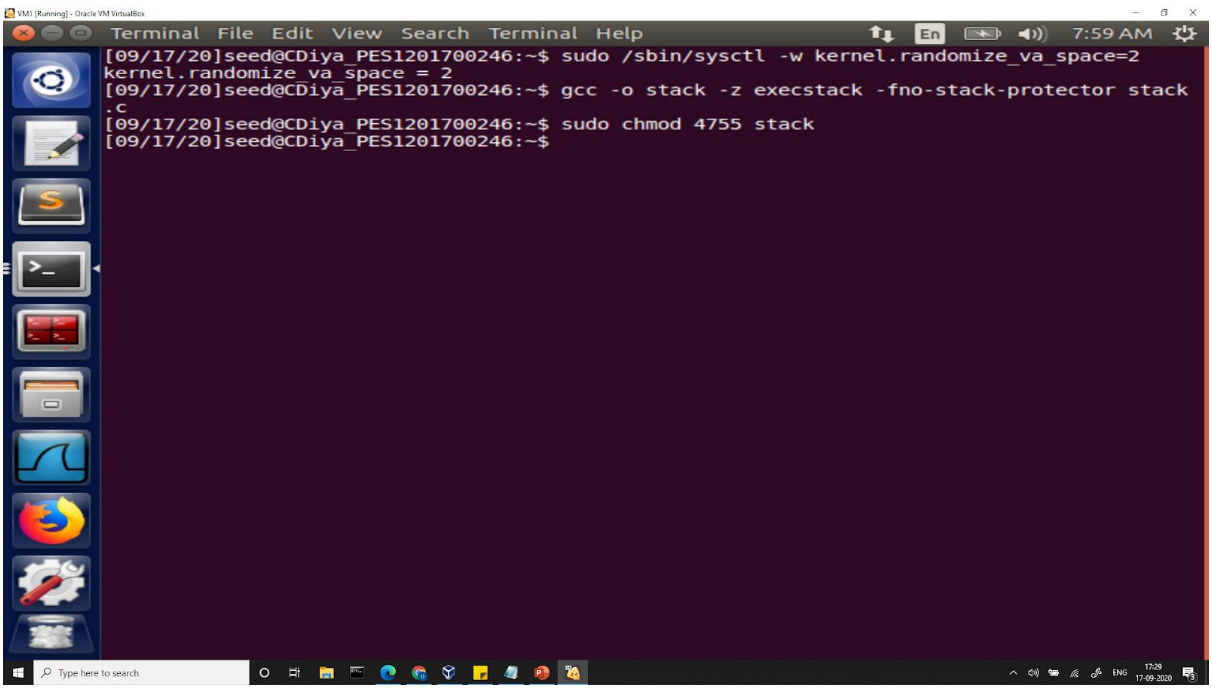
This approach is to change the real user ID of the victim process to zero before invoking the dash program. This is done by invoking setuid(0) before executing execve() in the shellcode.



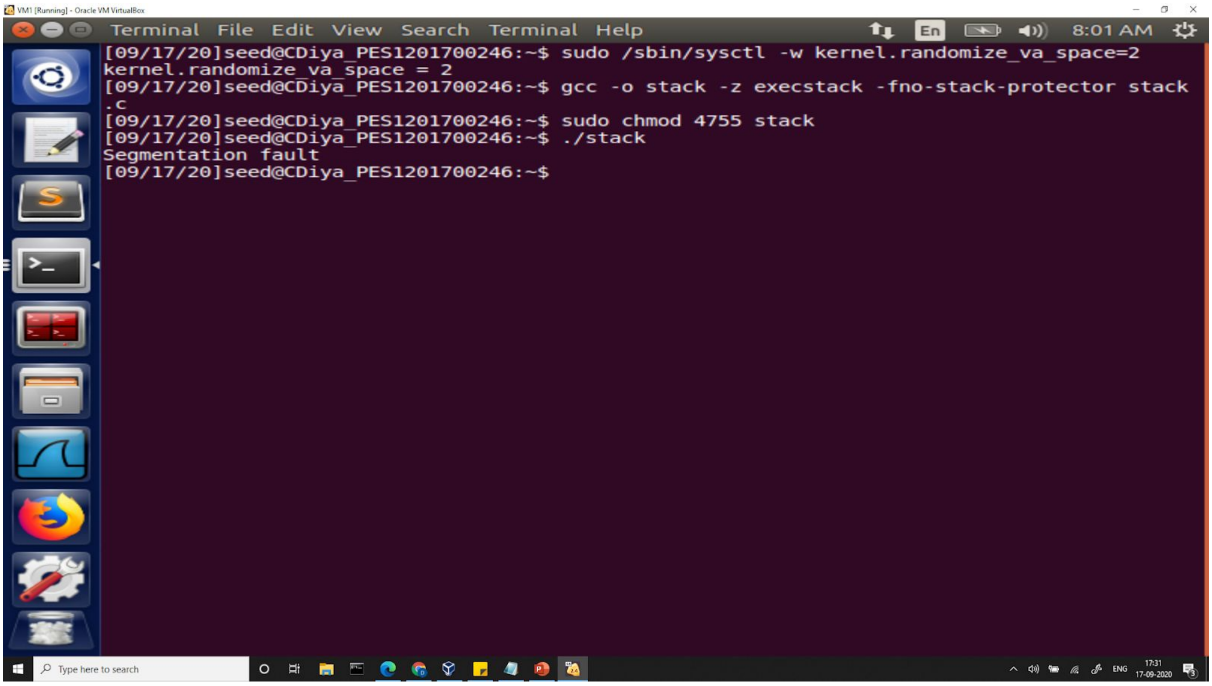


Observation: The screenshot above shows the root mode shell is obtained when the updated shellcode adds 4 instructions. Thus, the attack is successful.

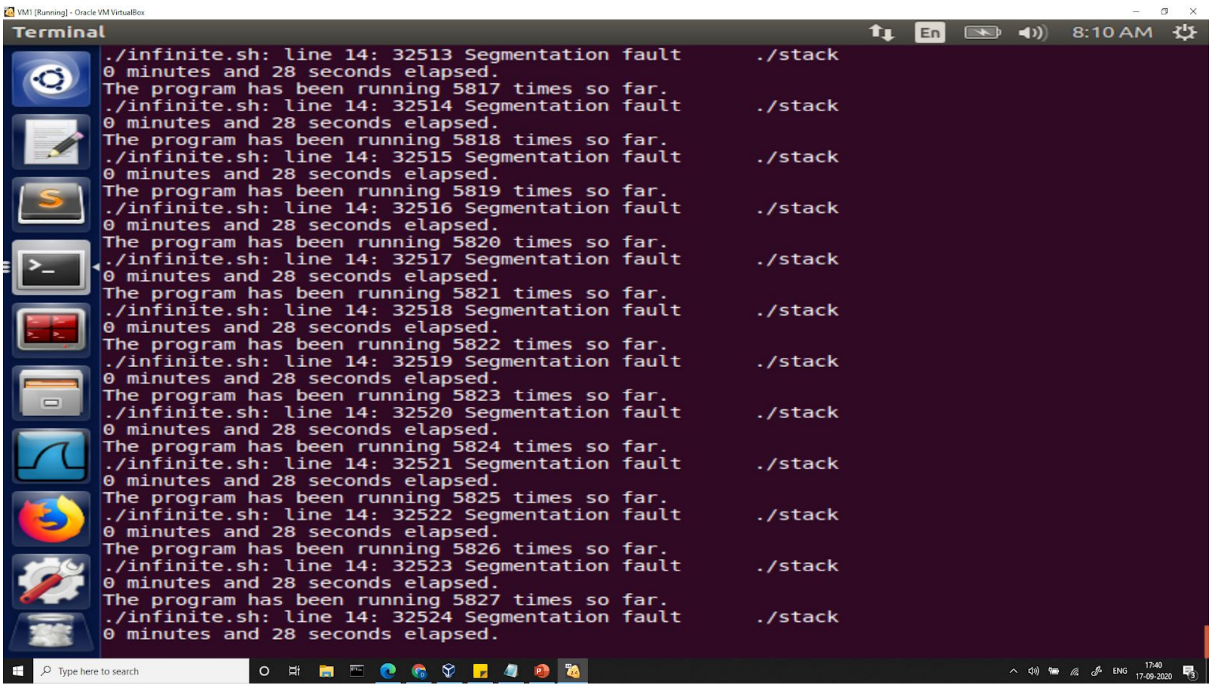
Task 5: Defeating Address Randomization



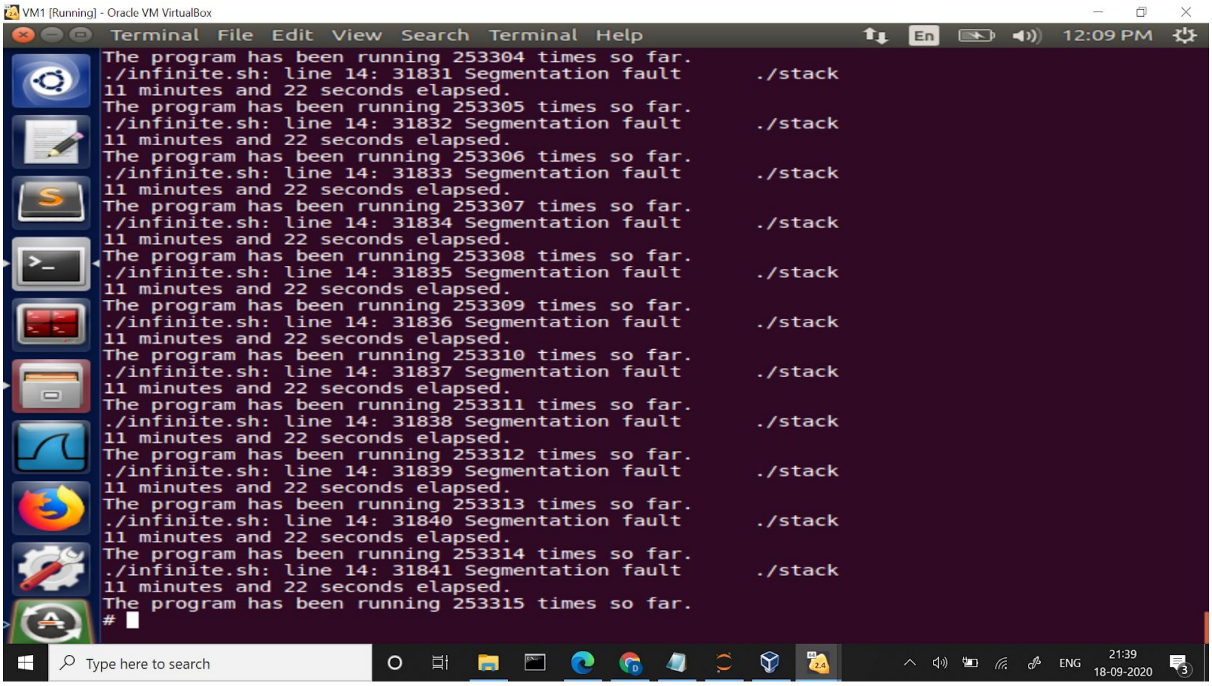
Observation: Ubuntu and Linux-based systems use address space randomization to randomize the starting address of heap and stack. Thus, the screenshot above shows the enabling of this measure so as to ensure that attacks occur using brute force.



Observation: Segmentation fault is encountered on running the stack program. This is because the address space randomization has been enabled which makes guessing exact address difficult

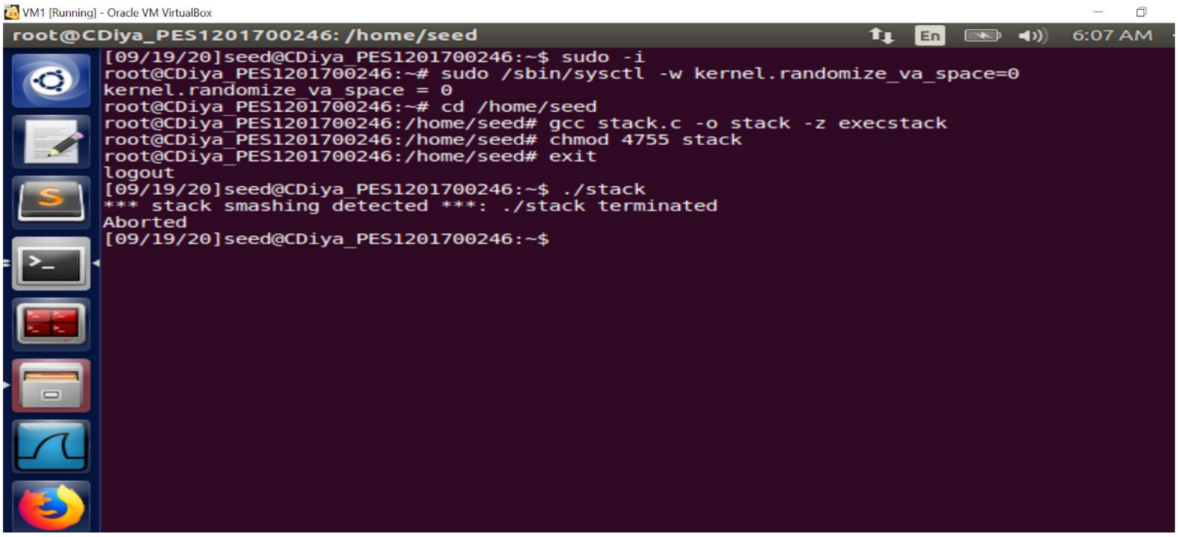


Observation: On running the infinite.sh script, due to the brute force approach, on 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 2^19 = 524, 288 possibilities. Thus, the program uses a brute-force approach to attack the vulnerable program repeatedly, hoping that the address put in the badfile can eventually be correct.



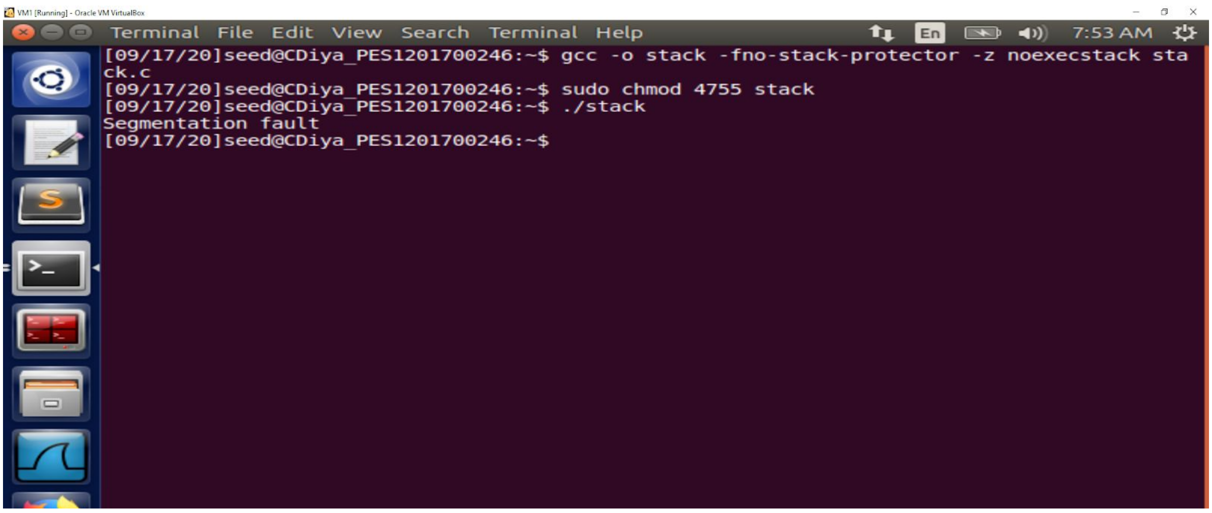
Observation: After 11 minutes, the exact address was obtained which enabled the shell in root mode stopping the program. Thus, the attack succeeded and the shell script stopped to give a root mode.

Task 6: Turn on the StackGuard Protection



Observation: From the screenshot above, it can be seen that the attack does not occur and a 'stack smashing detected: Aborted' message is seen. The GCC compiler uses The StackGuard Protection Scheme which implements a security mechanism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. This feature is enabled which prevents the attack from happening.

Task 7: Turn on the Non-executable Stack Protection



Observation:

Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult?

No shell has been created using noexecstack while compiling the program. The output is a segmentation fault. The reason for this is that stacks are set to be non-executable which prevent the attack from happening.
This simply makes the stack portion of a user process's virtual address space non-executable, so that attack code injected onto the stack cannot be executed.
Thus, using a non-executable protection option stack only makes it impossible to run shellcode on the stack which makes the program fail.