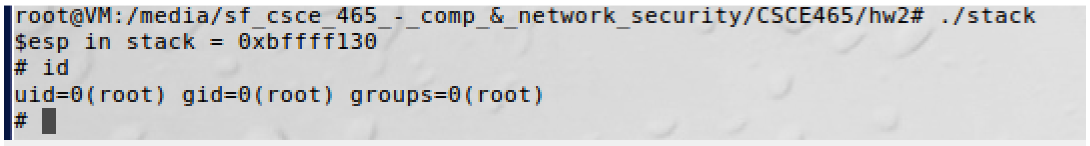
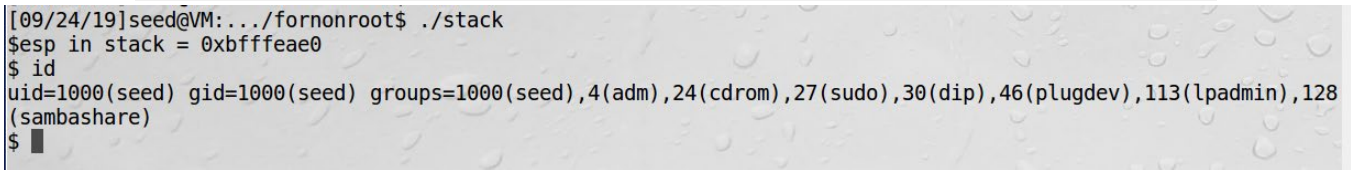
Homework 2: Buffer Overflow Vulnerability Attack and Defense Lab

1.1 Task 1: Exploiting the Vulnerability



```
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack
$esp in stack = 0xbffff130
# id
uid=0(root) gid=0(root) groups=0(root)
#
```



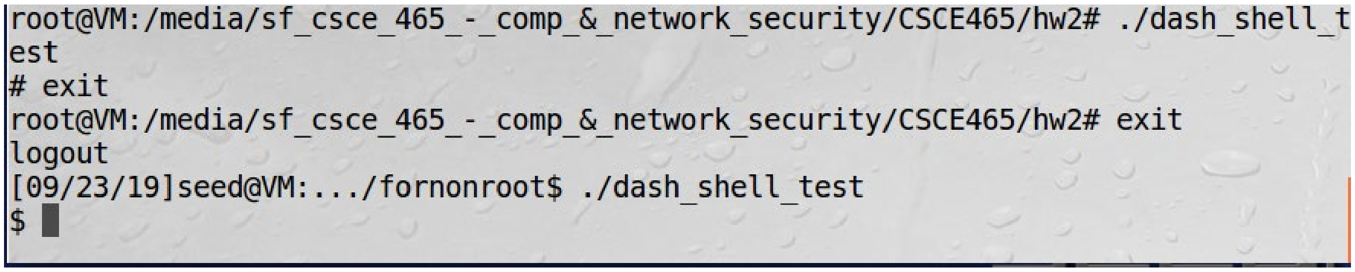
```
[09/24/19]seed@VM:.../fornonroot$ ./stack
$esp in stack = 0xbfffeae0
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128
(sambashare)
$
```

From the above image, it shows that I can only get privilege based on what privilege I was in when executing the program. I have also included the id of the user when I gained shell access

1.2 Task 2: Defeating dash’s Countermeasure

We first comment out the line (`setuid(0)`) and run the program as a `Set-UID` program (the owner should be root); please describe your observations. We then uncomment the line (`setuid(0)`) and run the program again; please describe your observations.

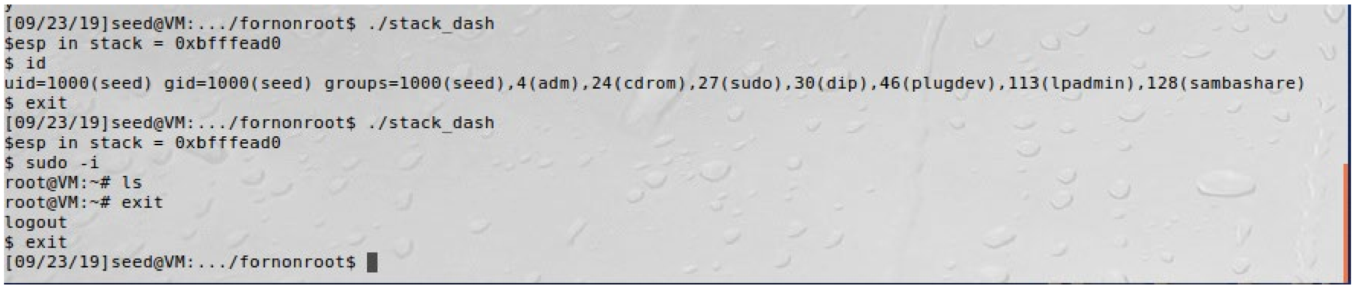
I noticed that running the shell code now give me access based on which privilege level I am executing from. I am able to acquire the shell as a user if I execute it in `seed` user, and root if `root`. The screengrab below shows the output:



```
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./dash_shell_t
est
# exit
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# exit
logout
[09/23/19]seed@VM:.../fornonroot$ ./dash_shell_test
$ 
```

Figure 1: privilege level of shell is based on adversaries’ privilege level when executed

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 system call number), and (3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when `/bin/sh` is linked to `/bin/dash`. Using the above shellcode in `exploit.c`, try the attack from Task 2 again and see if you can get a root shell. Please describe and explain your results.

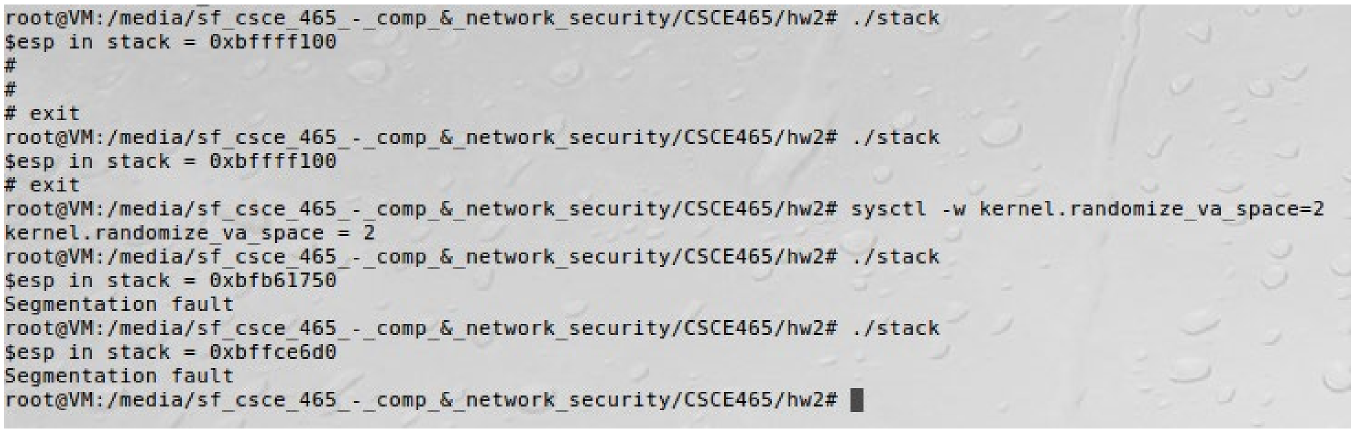


```
[09/23/19]seed@VM:.../fornonroot$ ./stack_dash
$esp in stack = 0xbfffead0
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[09/23/19]seed@VM:.../fornonroot$ ./stack_dash
$esp in stack = 0xbfffead0
$ sudo -i
root@VM:~# ls
root@VM:~# exit
logout
$ exit
[09/23/19]seed@VM:.../fornonroot$ 
```

Figure 2: privilege level is still based on adversaries’ privilege level when executed (see Figure 1)

1.3 Task 3: Address Randomization

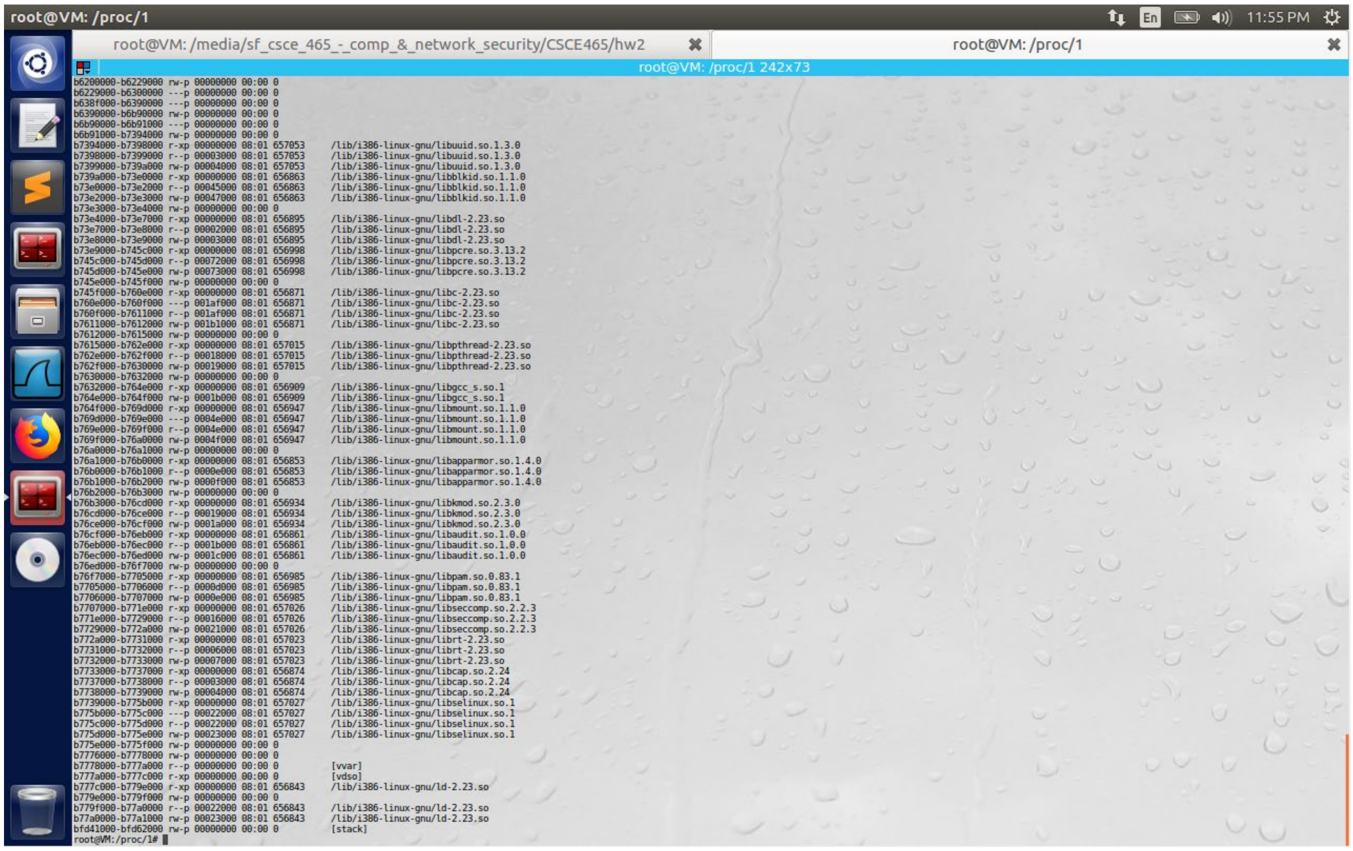
Now, we turn on the Ubuntu's address randomization. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your attacks difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:



```
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack
$esp in stack = 0xbffff100
#
#
# exit
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack
$esp in stack = 0xbffff100
# exit
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack
$esp in stack = 0xbfb61750
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack
$esp in stack = 0xbffce6d0
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2#
```

Figure 3: ASLR affecting absolute address pointer

I noticed that once I activate the ASLR, my program is no longer working. This is because of the linear randomization of the address that made it difficult for adversaries to try and guess where the stack pointer is. The probability of the adversary making an intelligent guess is $\frac{1}{2^{32-x}}$ where x is from restricting stack pointer to a range of 0xb0000000 to 0xbfffffff.
For example, one can see where the libraries/header files are saved in the heap as well as the stack by performing `cat /proc/$proc_num/maps`. An example of this can be found in the image below (figure 4)


root@VM: /proc/1
11:55 PM
root@VM: /media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2
root@VM: /proc/1
root@VM: /proc/1 242x73

```
b6200000-b6229000 rw-p 00000000 00:00 0
b6229000-b6300000 ---p 00000000 00:00 0
b638f000-b6390000 ---p 00000000 00:00 0
b6390000-b6b90000 rw-p 00000000 00:00 0
b6b90000-b6b91000 ---p 00000000 00:00 0
b6b91000-b7394000 rw-p 00000000 00:00 0
b7394000-b7398000 r-xp 00000000 08:01 657053     /lib/i386-linux-gnu/libuuid.so.1.3.0
b7398000-b7399000 r--p 00003000 08:01 657053     /lib/i386-linux-gnu/libuuid.so.1.3.0
b7399000-b739a000 rw-p 00004000 08:01 657053     /lib/i386-linux-gnu/libuuid.so.1.3.0
b739a000-b73e0000 r-xp 00000000 08:01 656863     /lib/i386-linux-gnu/libblkid.so.1.1.0
b73e0000-b73e2000 r--p 00045000 08:01 656863     /lib/i386-linux-gnu/libblkid.so.1.1.0
b73e2000-b73e3000 rw-p 00047000 08:01 656863     /lib/i386-linux-gnu/libblkid.so.1.1.0
b73e3000-b73e4000 rw-p 00000000 00:00 0
b73e4000-b73e7000 r-xp 00000000 08:01 656895     /lib/i386-linux-gnu/libdl-2.23.so
b73e7000-b73e8000 r--p 00002000 08:01 656895     /lib/i386-linux-gnu/libdl-2.23.so
b73e8000-b73e9000 rw-p 00003000 08:01 656895     /lib/i386-linux-gnu/libdl-2.23.so
b73e9000-b745c000 r-xp 00000000 08:01 656998     /lib/i386-linux-gnu/libpcre.so.3.13.2
b745c000-b745d000 r--p 00072000 08:01 656998     /lib/i386-linux-gnu/libpcre.so.3.13.2
b745d000-b745e000 rw-p 00073000 08:01 656998     /lib/i386-linux-gnu/libpcre.so.3.13.2
b745e000-b745f000 rw-p 00000000 00:00 0
b745f000-b760e000 r-xp 00000000 08:01 656871     /lib/i386-linux-gnu/libc-2.23.so
b760e000-b760f000 ---p 001af000 08:01 656871     /lib/i386-linux-gnu/libc-2.23.so
b760f000-b7611000 r--p 001af000 08:01 656871     /lib/i386-linux-gnu/libc-2.23.so
b7611000-b7612000 rw-p 001b1000 08:01 656871     /lib/i386-linux-gnu/libc-2.23.so
b7612000-b7615000 rw-p 00000000 00:00 0
b7615000-b762e000 r-xp 00000000 08:01 657015     /lib/i386-linux-gnu/libpthread-2.23.so
b762e000-b762f000 r--p 00018000 08:01 657015     /lib/i386-linux-gnu/libpthread-2.23.so
b762f000-b7630000 rw-p 00019000 08:01 657015     /lib/i386-linux-gnu/libpthread-2.23.so
b7630000-b7632000 rw-p 00000000 00:00 0
b7632000-b764e000 r-xp 00000000 08:01 656909     /lib/i386-linux-gnu/libgcc_s.so.1
b764e000-b764f000 rw-p 0001b000 08:01 656909     /lib/i386-linux-gnu/libgcc_s.so.1
b764f000-b769d000 r-xp 00000000 08:01 656947     /lib/i386-linux-gnu/libmount.so.1.1.0
b769d000-b769e000 ---p 0004e000 08:01 656947     /lib/i386-linux-gnu/libmount.so.1.1.0
b769e000-b769f000 r--p 0004e000 08:01 656947     /lib/i386-linux-gnu/libmount.so.1.1.0
b769f000-b76a0000 rw-p 0004f000 08:01 656947     /lib/i386-linux-gnu/libmount.so.1.1.0
b76a0000-b76a1000 rw-p 00000000 00:00 0
b76a1000-b76b0000 r-xp 00000000 08:01 656853     /lib/i386-linux-gnu/libapparmor.so.1.4.0
b76b0000-b76b1000 r--p 0000e000 08:01 656853     /lib/i386-linux-gnu/libapparmor.so.1.4.0
b76b1000-b76b2000 rw-p 0000f000 08:01 656853     /lib/i386-linux-gnu/libapparmor.so.1.4.0
b76b2000-b76b3000 rw-p 00000000 00:00 0
b76b3000-b76cd000 r-xp 00000000 08:01 656934     /lib/i386-linux-gnu/libkmod.so.2.3.0
b76cd000-b76ce000 r--p 00019000 08:01 656934     /lib/i386-linux-gnu/libkmod.so.2.3.0
b76ce000-b76cf000 rw-p 0001a000 08:01 656934     /lib/i386-linux-gnu/libkmod.so.2.3.0
b76cf000-b76eb000 r-xp 00000000 08:01 656861     /lib/i386-linux-gnu/libaudit.so.1.0.0
b76eb000-b76ec000 r--p 0001b000 08:01 656861     /lib/i386-linux-gnu/libaudit.so.1.0.0
b76ec000-b76ed000 rw-p 0001c000 08:01 656861     /lib/i386-linux-gnu/libaudit.so.1.0.0
b76ed000-b76f7000 rw-p 00000000 00:00 0
b76f7000-b7705000 r-xp 00000000 08:01 656985     /lib/i386-linux-gnu/libpam.so.0.83.1
b7705000-b7706000 r--p 0000d000 08:01 656985     /lib/i386-linux-gnu/libpam.so.0.83.1
b7706000-b7707000 rw-p 0000e000 08:01 656985     /lib/i386-linux-gnu/libpam.so.0.83.1
b7707000-b771e000 r-xp 00000000 08:01 657026     /lib/i386-linux-gnu/libseccomp.so.2.2.3
b771e000-b7729000 r--p 00016000 08:01 657026     /lib/i386-linux-gnu/libseccomp.so.2.2.3
b7729000-b772a000 rw-p 00021000 08:01 657026     /lib/i386-linux-gnu/libseccomp.so.2.2.3
b772a000-b7731000 r-xp 00000000 08:01 657023     /lib/i386-linux-gnu/librt-2.23.so
b7731000-b7732000 r--p 00006000 08:01 657023     /lib/i386-linux-gnu/librt-2.23.so
b7732000-b7733000 rw-p 00007000 08:01 657023     /lib/i386-linux-gnu/librt-2.23.so
b7733000-b7737000 r-xp 00000000 08:01 656874     /lib/i386-linux-gnu/libcap.so.2.24
b7737000-b7738000 r--p 00003000 08:01 656874     /lib/i386-linux-gnu/libcap.so.2.24
b7738000-b7739000 rw-p 00004000 08:01 656874     /lib/i386-linux-gnu/libcap.so.2.24
b7739000-b775b000 r-xp 00000000 08:01 657027     /lib/i386-linux-gnu/libselinux.so.1
b775b000-b775c000 ---p 00022000 08:01 657027     /lib/i386-linux-gnu/libselinux.so.1
b775c000-b775d000 r--p 00022000 08:01 657027     /lib/i386-linux-gnu/libselinux.so.1
b775d000-b775e000 rw-p 00023000 08:01 657027     /lib/i386-linux-gnu/libselinux.so.1
b775e000-b775f000 rw-p 00000000 00:00 0
b7776000-b7778000 rw-p 00000000 00:00 0
b7778000-b777a000 r--p 00000000 00:00 0          [vvar]
b777a000-b777c000 r-xp 00000000 00:00 0          [vdso]
b777c000-b779e000 r-xp 00000000 08:01 656843     /lib/i386-linux-gnu/ld-2.23.so
b779e000-b779f000 rw-p 00000000 00:00 0
b779f000-b77a0000 r--p 00022000 08:01 656843     /lib/i386-linux-gnu/ld-2.23.so
b77a0000-b77a1000 rw-p 00023000 08:01 656843     /lib/i386-linux-gnu/ld-2.23.so
bfd41000-bfd62000 rw-p 00000000 00:00 0          [stack]
root@VM:/proc/1#
```

Figure 4: cat /proc/$proc_num/maps to see the entire program address mapping

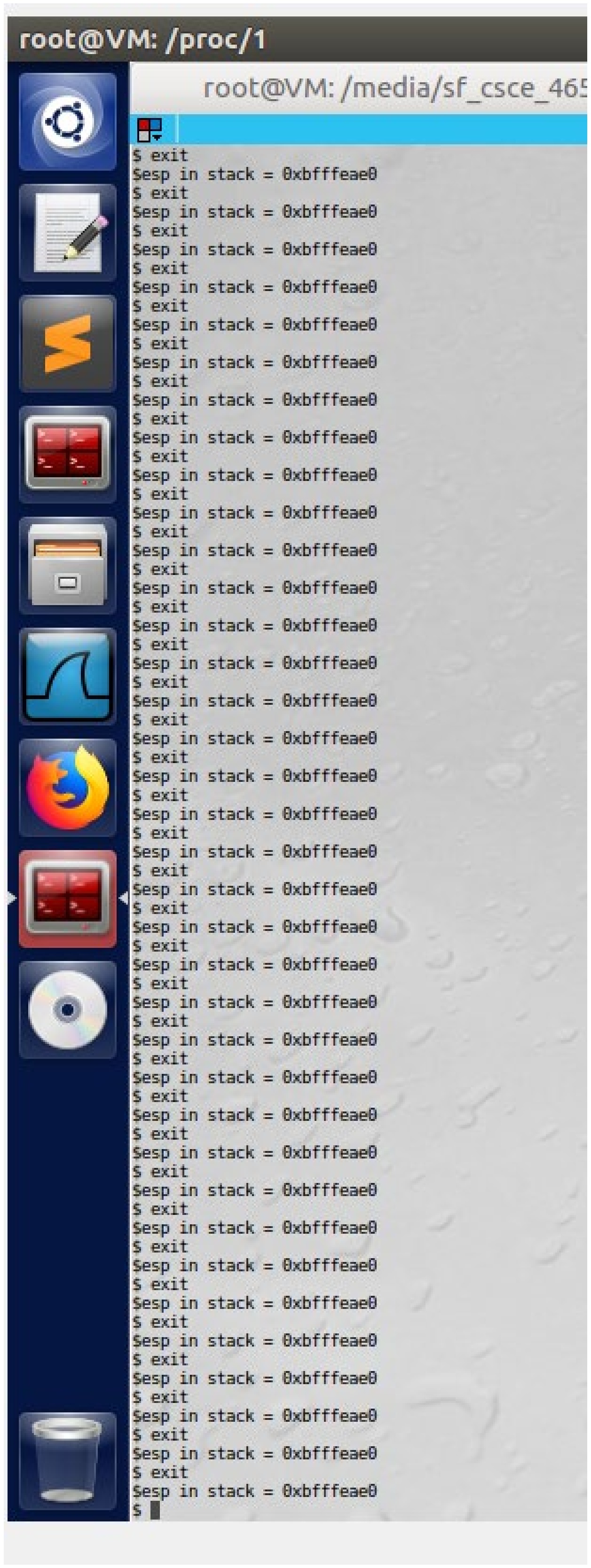
```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

If running the vulnerable code once does not get you the root shell, how about running it for many times? You can run `./stack` in the following loop , and see what will happen. If your exploit program is designed properly, you should be able to get the root shell after a while. You can modify your exploit program to increase the probability of success (i.e., reduce the time that you have to wait).

```
$ sh -c "while [ 1 ]; do ./stack; done;"
```

Running the above code did not seem to help with acquiring root. I have no idea what I am doing wrongly, but I am not acquiring root no matter how long I have tried.

The image below (figure 5) shows the attempt:



```
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ exit
$esp in stack = 0xbfffeae0
$ 
```

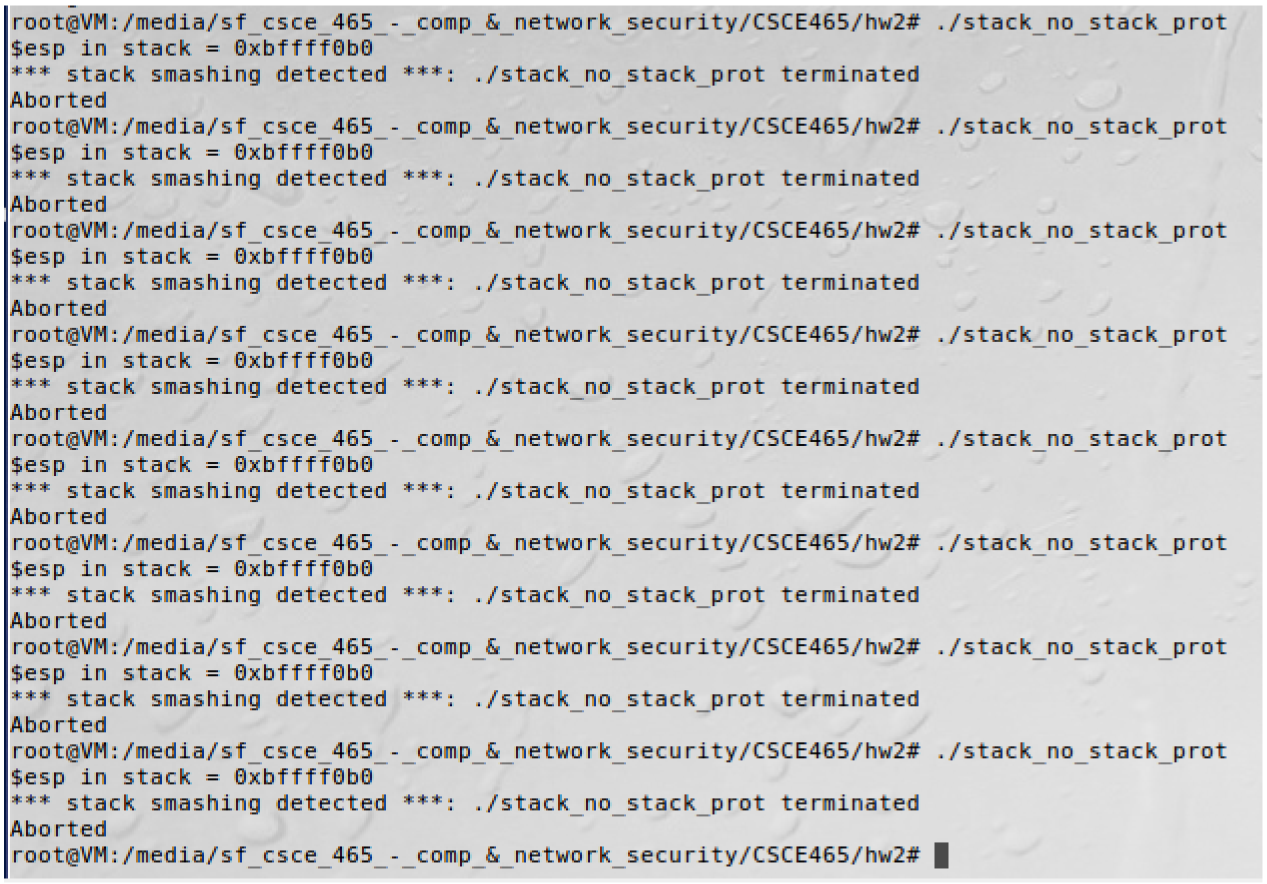
Figure 5: Countless failed attempts to acquire root

1.4 Task 4: Stack Guard

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the "Stack Guard" protection mechanism in GCC when compiling the programs. In this task, you may consider repeating task 1 in the presence of Stack Guard. To do that, you should compile the program without the *-fno-stack-protector'* option. For this task, you will recompile the vulnerable program, stack.c, to use GCC's Stack Guard, execute task 1 again, and report your observations. You may report any error messages you observe.

In the GCC 4.3.3 and newer versions, Stack Guard is enabled by default. Therefore, you have to disable Stack Guard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable Stack Guard.



```
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_stack_prot
$esp in stack = 0xbffff0b0
*** stack smashing detected ***: ./stack_no_stack_prot terminated
Aborted
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2#
```

Figure 6: Stack smashing detected when -fno-stack-protector flag is on

Unable to execute shellcode when -fno-stack-protector flag is activated (see image above, figure 6).

1.5 Task 5: Non-executable Stack

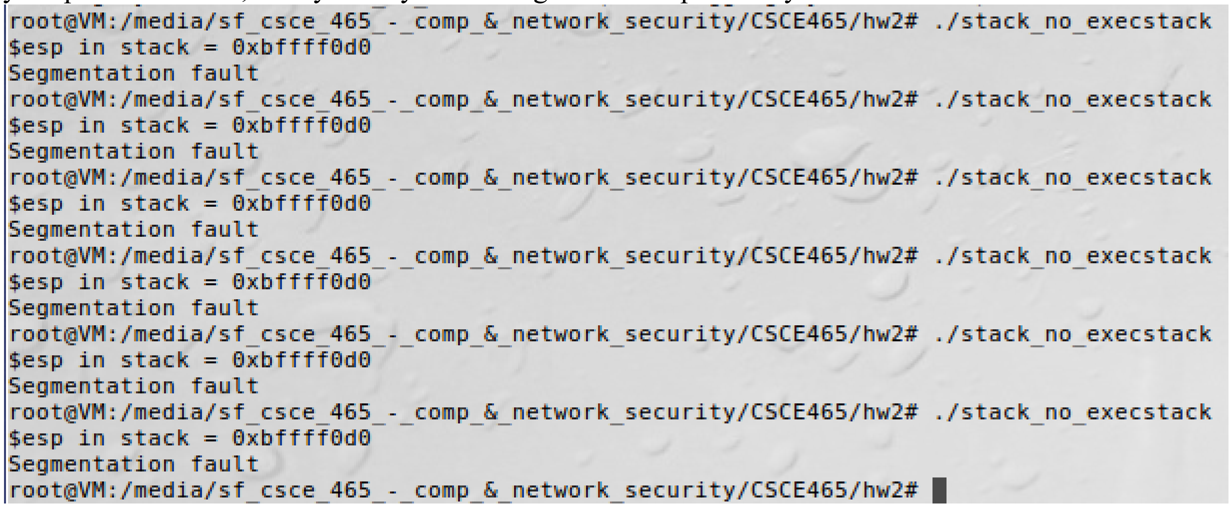
Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we intentionally make stacks executable. In this task, we recompile our vulnerable program using the `noexecstack` option, and repeat the attack in Task 1. Can you get a shell? If not, what is the problem? How does this protection scheme make your attacks difficult. You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the non-executable stack protection.

```
# gcc -o stack -fno-stack-protector -z noexecstack stack.c
```

It should be noted that non-executable stack only makes it impossible to run shellcode on the stack, but it does not prevent buffer-overflow attacks, because there are other ways to run malicious code after exploiting a buffer-overflow vulnerability. The *return-to-libc* attack is an example.

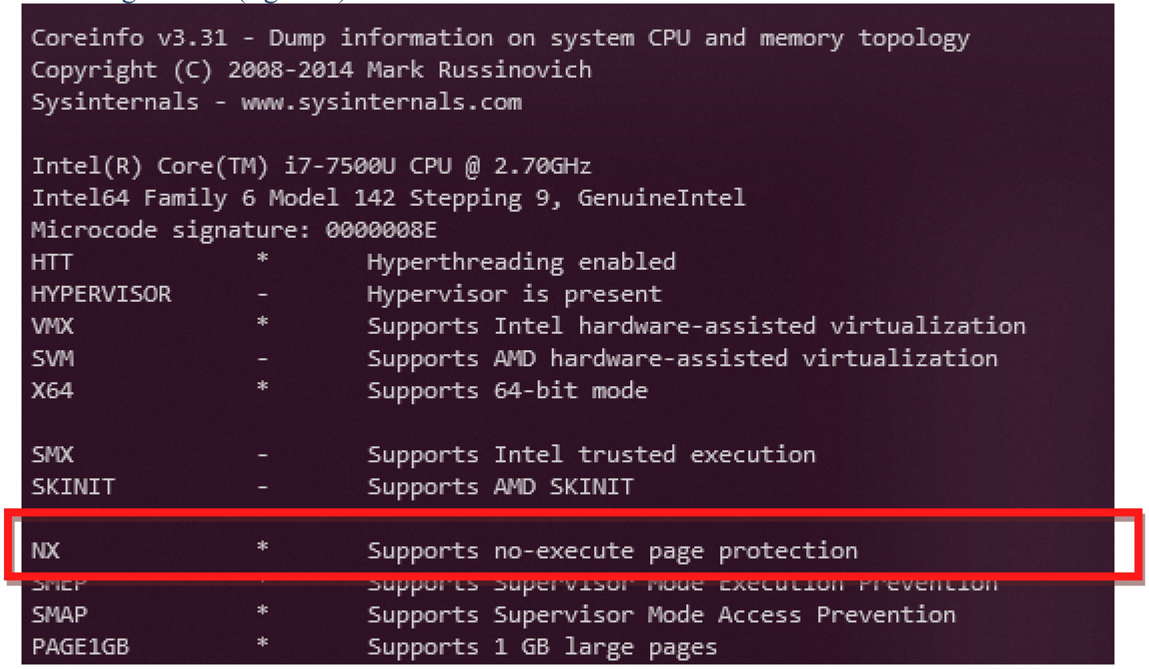
If you are using our Ubuntu 12.04/16.04 VM, whether the non-executable stack protection works or not depends on the CPU and the setting of your virtual machine, because this protection depends on the hardware feature that is provided by CPU. If you find that the non-executable stack protection does not work, check our this document (`https://seedsecuritylabs.org/Labs_16.04/Software/Buffer_Overflow/files/NX.pdf`), and see whether the instruction in the document can help solve your problem. If not, then you may need to figure out the problem yourself.



```
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_execstack
$esp in stack = 0xbffff0d0
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_execstack
$esp in stack = 0xbffff0d0
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_execstack
$esp in stack = 0xbffff0d0
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_execstack
$esp in stack = 0xbffff0d0
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_execstack
$esp in stack = 0xbffff0d0
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack_no_execstack
$esp in stack = 0xbffff0d0
Segmentation fault
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# 
```

Figure 7: no execstack flag prevents execution of instructions in stack memory

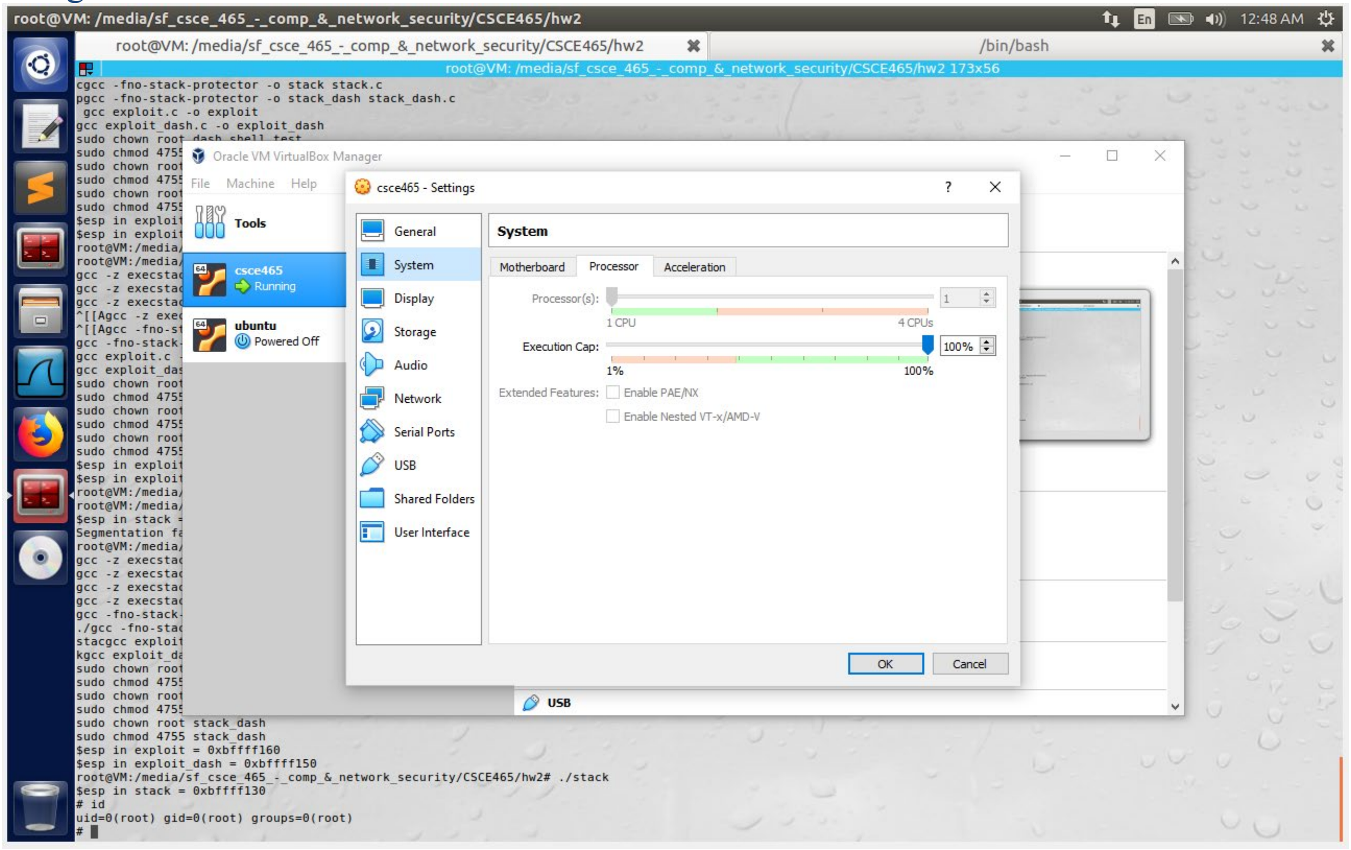
Looking at the page given above, I found my computer's CPU to support no-executable page protection. This is reflected in the image below (figure 8):



```
Coreinfo v3.31 - Dump information on system CPU and memory topology
Copyright (C) 2008-2014 Mark Russinovich
Sysinternals - www.sysinternals.com

Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
Intel64 Family 6 Model 142 Stepping 9, GenuineIntel
Microcode signature: 0000008E
HTT             *       Hyperthreading enabled
HYPERVISOR      -       Hypervisor is present
VMX             *       Supports Intel hardware-assisted virtualization
SVM             -       Supports AMD hardware-assisted virtualization
X64             *       Supports 64-bit mode

SMX             -       Supports Intel trusted execution
SKINIT          -       Supports AMD SKINIT

NX              *       Supports no-execute page protection
SMEP                    Supports Supervisor Mode Execution Prevention
SMAP            *       Supports Supervisor Mode Access Prevention
PAGE1GB         *       Supports 1 GB large pages
```

Figure 8: No-execute page protection is supported

Also, following from the webpage above, I have found that the NX bit was not activated in my computer, yet disabling execstack seems to have made an impact on my program's ability to run shellcode from stack. The image below (Figure 9) describes the option as being disabled:



root@VM: /media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2

```
cgcc -fno-stack-protector -o stack stack.c
pgcc -fno-stack-protector -o stack_dash stack_dash.c
 gcc exploit.c -o exploit
gcc exploit_dash.c -o exploit_dash
sudo chown root dash_shell_test
sudo chmod 4755
sudo chown root
sudo chmod 4755
sudo chown root
sudo chmod 4755
$esp in exploit
$esp in exploit
root@VM:/media/
root@VM:/media/
gcc -z execstac
gcc -z execstac
gcc -z execstac
^[[Agcc -z exec
^[[Agcc -fno-st
gcc -fno-stack-
gcc exploit.c -
gcc exploit_das
sudo chown root
sudo chmod 4755
sudo chown root
sudo chmod 4755
sudo chown root
sudo chmod 4755
$esp in exploit
$esp in exploit
root@VM:/media/
root@VM:/media/
$esp in stack =
Segmentation fa
root@VM:/media/
gcc -z execstac
gcc -z execstac
gcc -z execstac
gcc -z execstac
gcc -fno-stack-
./gcc -fno-stac
stacgcc exploit
kgcc exploit_da
sudo chown root
sudo chmod 4755
sudo chown root
sudo chmod 4755
sudo chown root stack_dash
sudo chmod 4755 stack_dash
$esp in exploit = 0xbffff160
$esp in exploit_dash = 0xbffff150
root@VM:/media/sf_csce_465_-_comp_&_network_security/CSCE465/hw2# ./stack
$esp in stack = 0xbffff130
# id
uid=0(root) gid=0(root) groups=0(root)
# 
```

csce465 - Settings

System

Processor(s): 1

Execution Cap: 100%

Extended Features: ☐ Enable PAE/NX

☐ Enable Nested VT-x/AMD-V

Figure 9: NX is disabled in my CPU