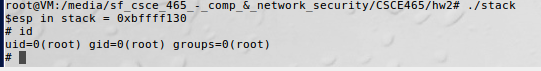
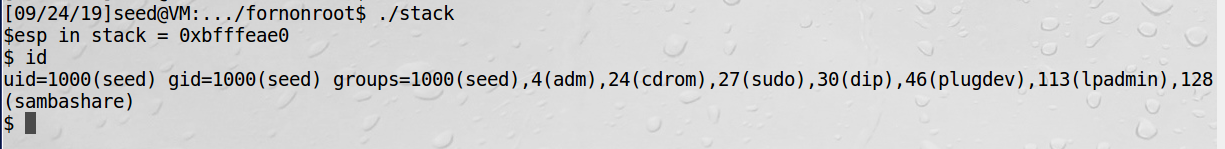
Homework 2:

Buffer Overflow Vulnerability Attack and Defense Lab

## Task 1: Exploiting the Vulnerability





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

## 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:

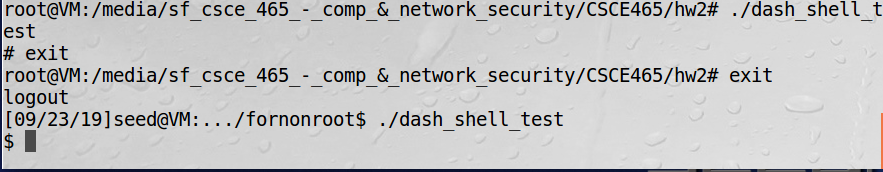


Figure : privilege level of shell is based on adversaries’ privilege level when executed

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.

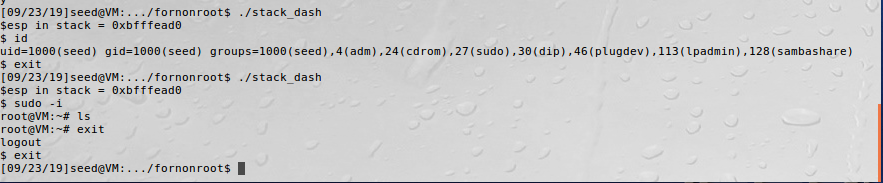


Figure : privilege level is still based on adversaries’ privilege level when executed (see Figure 1)

## 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:

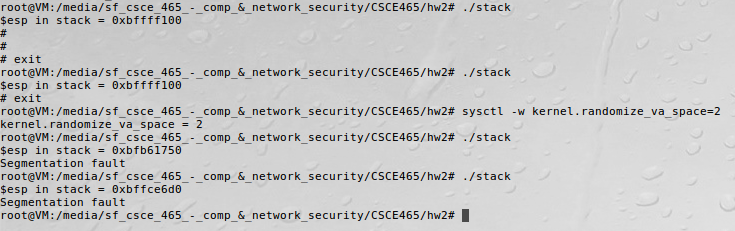


Figure : ASLR affecting absolute address pointer

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 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)

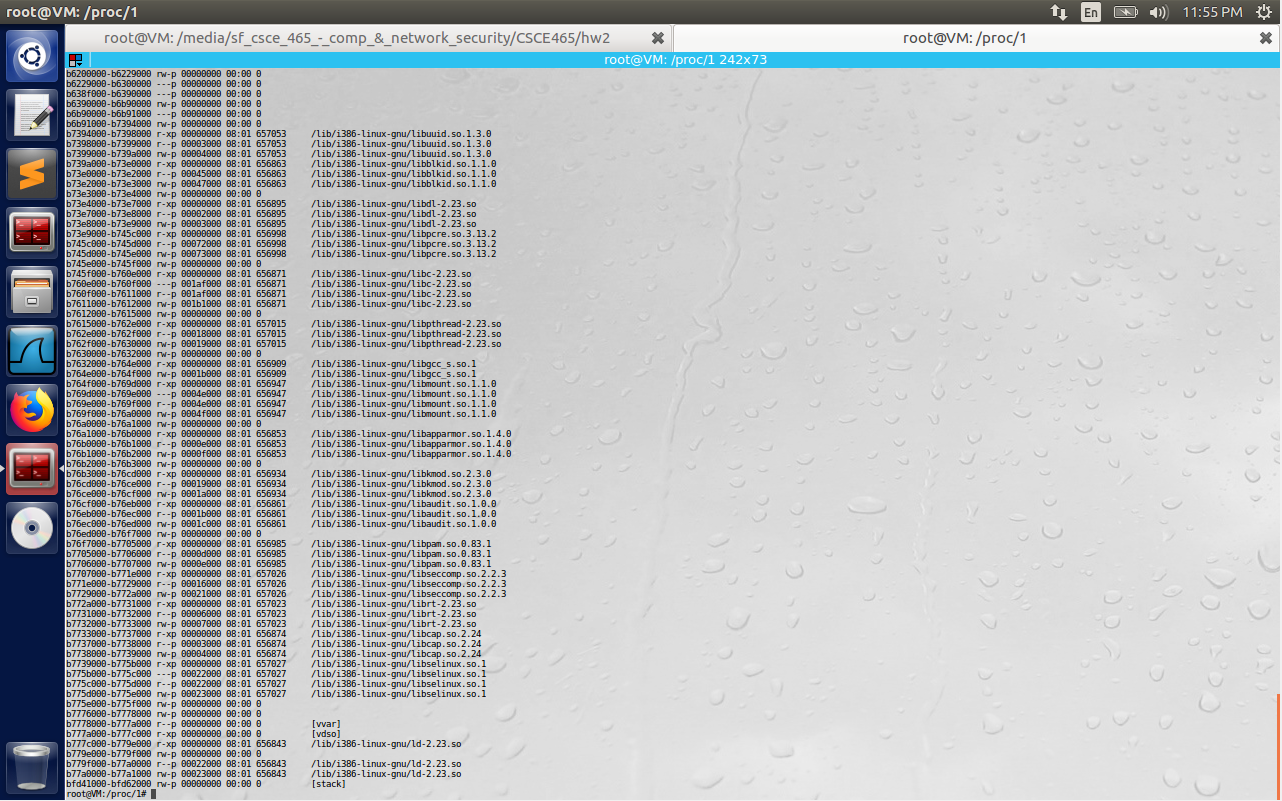


Figure : cat /proc/$proc\_num/maps to see the entire program address mapping

$ 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:

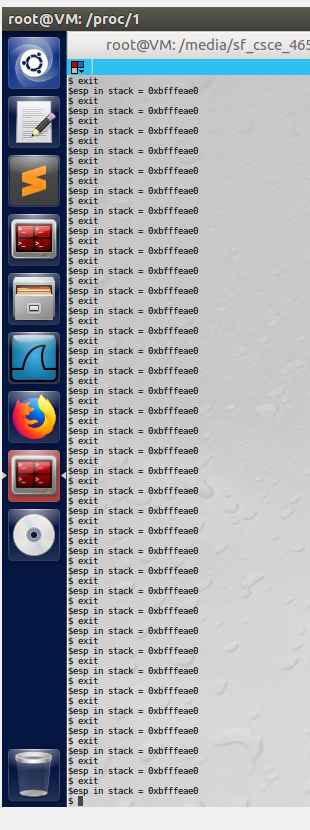


Figure : Countless failed attempts to acquire root

## 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.

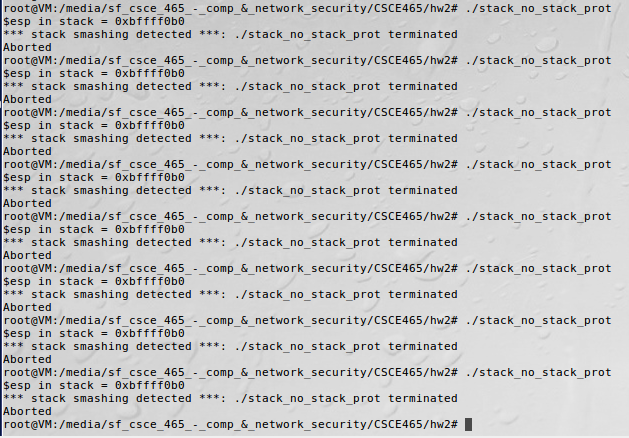


Figure : Stack smashing detected when -fno-stack-protector flag is on

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

## 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.

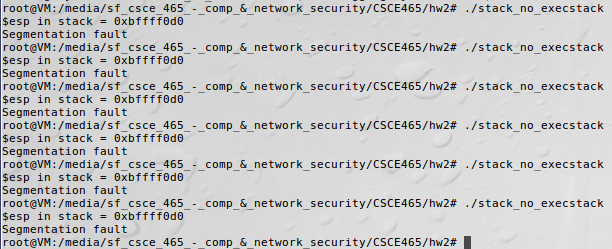


Figure : no execstack flag prevents execution of instructions in stack memory

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):

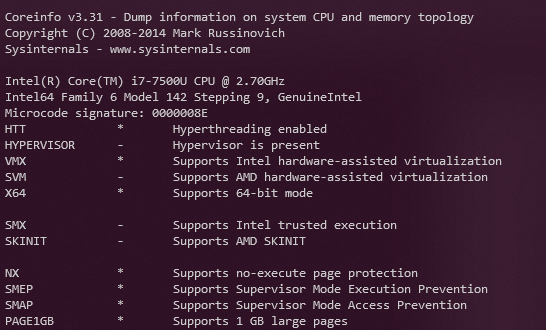


Figure : No-execute page protection is supported

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:

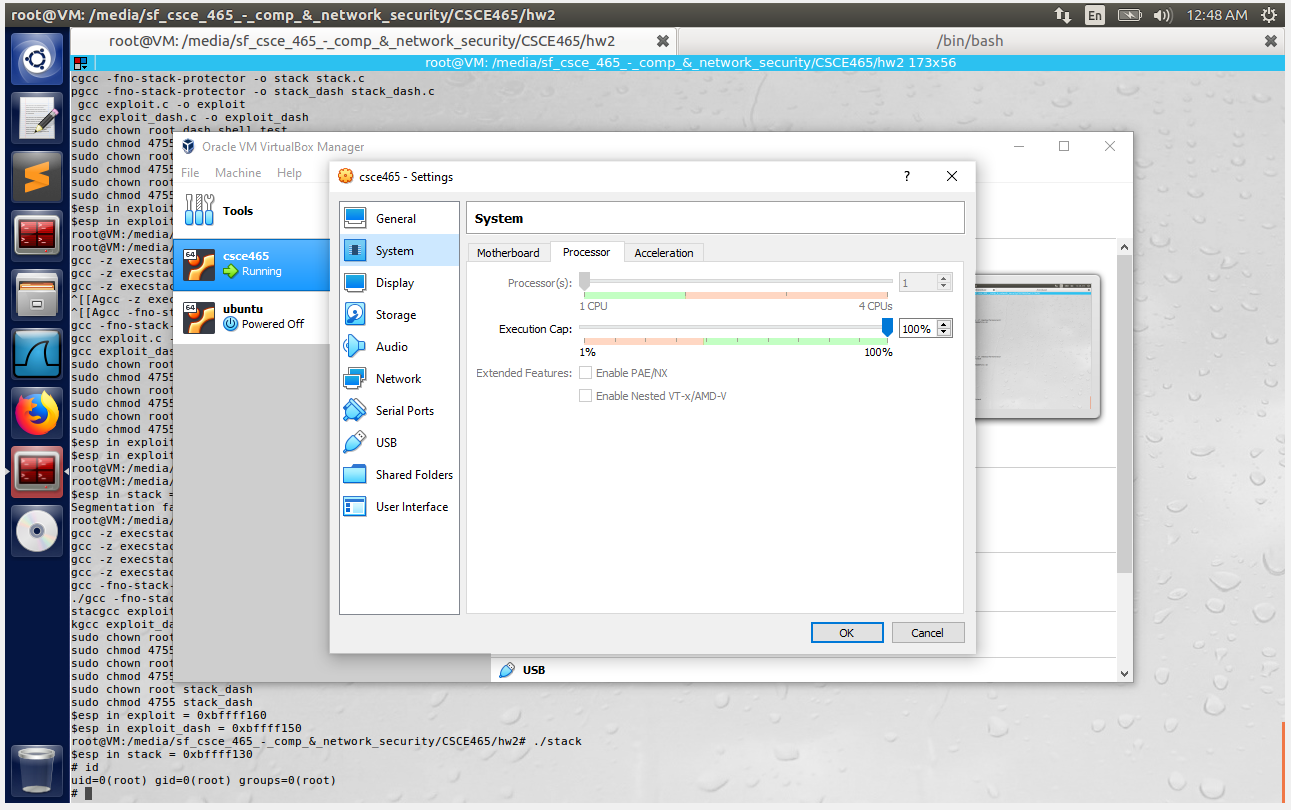


Figure : NX is disabled in my CPU