**2.1 Initial Setup:**

The first thing we are instructed to do is to disable Linux’s address space randomization. We do this by setting a flag seen in the image below:

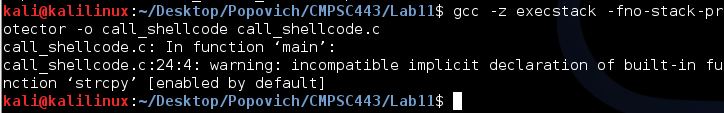
P:\CMPSC443\Git\CMPSC443\Lab11\DerandomizeAddressSpace.PNG

Next, we are reminded to compile with the flag “-fno-stack-protector” and “-z execstack”. This disables the “stack guard” which prevents buffer overflows and allows the stack to be executable.

**2.2 Shellcode:**

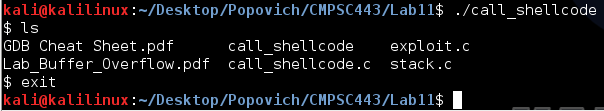
For the next task, we are given some sample code to compile and run. If we are able to successfully compile it and then upon being ran, invoke a shell, this verifies that we have the necessary flags set in Linux and the necessary flags set during compilation.

The first step is to compile the code:



Notice here that Linux warns us of the function ‘strcpy’. This function is what our exploit is based off of.

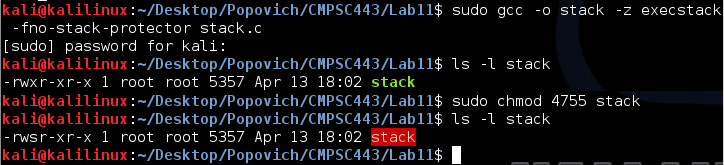
Next, we will run the program:



Success! Upon running it we are greeted with a shell, as demonstrated with the ‘ls’ command.

**2.3 The Vulnerable Program**

This task begins with having us compile some code with root privileges and then sets the program’s ‘setuid’ permission to root. This will allow the program to execute with root privileges no matter who runs it.



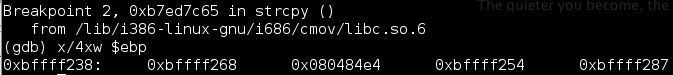
However, this “stack” program has a buffer overflow vulnerability. It first reads an input from a file called “badfile”, and then passes this input to another buffer in the function bof(). The original input can have a maximum length of 517 bytes, but the buffer in bof() has only 12 bytes long. Because strcpy() does not check boundaries, buffer overflow will likely occur. Since this program is a set-root-uid program, if a normal user can exploit this buffer overflow vulnerability, the normal user might be able to get a root shell.

To design the payload, we need to find out two things. First, where the return address is so we can overwrite it. Second, where a copy of our malicious code is being stored so we can direct the return address to that location.

We can use the GNU Project Debugger (gdb) and strategically pause the program to view variable addresses. In specific, we can add a breakpoint when “stack.c” calls “bof”. By stopping here, we can see where in memory the return address is being stored by looking at the ebp. We know that the return address is right above the ebp, which is stored at 0xbffff268. Thus, the return address is at ebp+4 = 0xbffff26c.

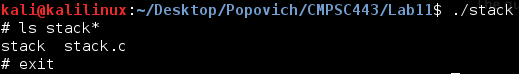
P:\CMPSC443\Git\CMPSC443\Lab11\returnAddressAddress.PNG

We can continue until we hit the “strcpy” function because the buffer, which our malicious code is being copied to, is an argument to the strcpy function. Looking at the first argument, we see that it is a reference to 0xbffff254.



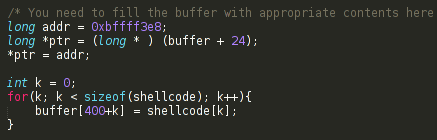
Subtracting 0xbffff254 from 0xbffff26c leaves us with 24. Thus, we have 24 spaces of memory from the start of the buffer to the return address. The 24th byte of our badfile will begin the address to jump to.

What address should we jump to? We can put our malicious code after the 28th byte of the badfile and jump to it! A successful jump can be seen below:

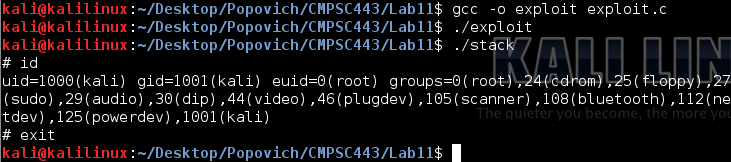


**2.4 Task 1: Exploiting the Vulnerability**

This program has us write the ‘badfile’ that I wrote by hand above. The additions to the exploit file are below:

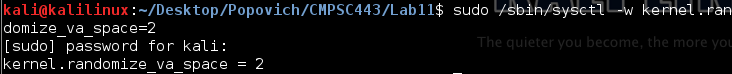


This code replaces a few parts of the badfile’s 0x9090 (NOPs) with the address to return to and the malicious code. After running the previous stack program, it once again, generates a shell:

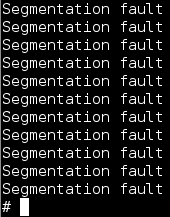


**2.5 Task 2: Address Randomization**

For this task, we are told to turn off address randomization.



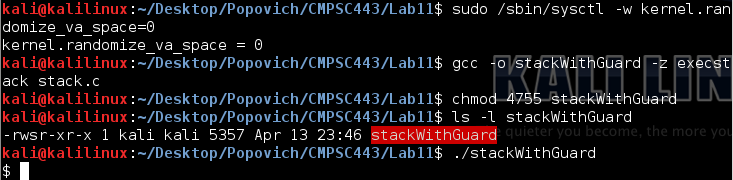
However, our attack should still work! We just have to get lucky! We are told to run our previous stack program but in an infinite loop:



Eventually the address space lines up with what it used to be!

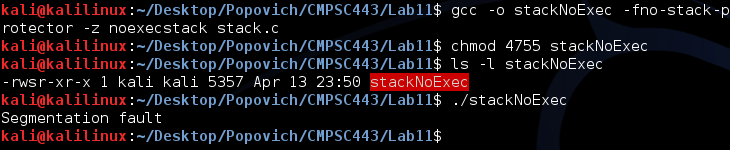
**2.6 Task 3: Stack Guard**

For this task, stack guard is re-enabled while address randomization is disabled. I followed the instructions and surprisingly the buffer overflow still executes!



**2.7 Task 4: Non-executable Stack**

For the last task we disable the executable stack and attempt to complete Task 1 again. However this time, we are blocked!



This task is successful because we are attempting to execute code on the stack. Obviously, by its name, this is prevented. However, as talked about in class, this is not the only way to execute code. The return-to-libc attack is an example that does not execute code on the stack.