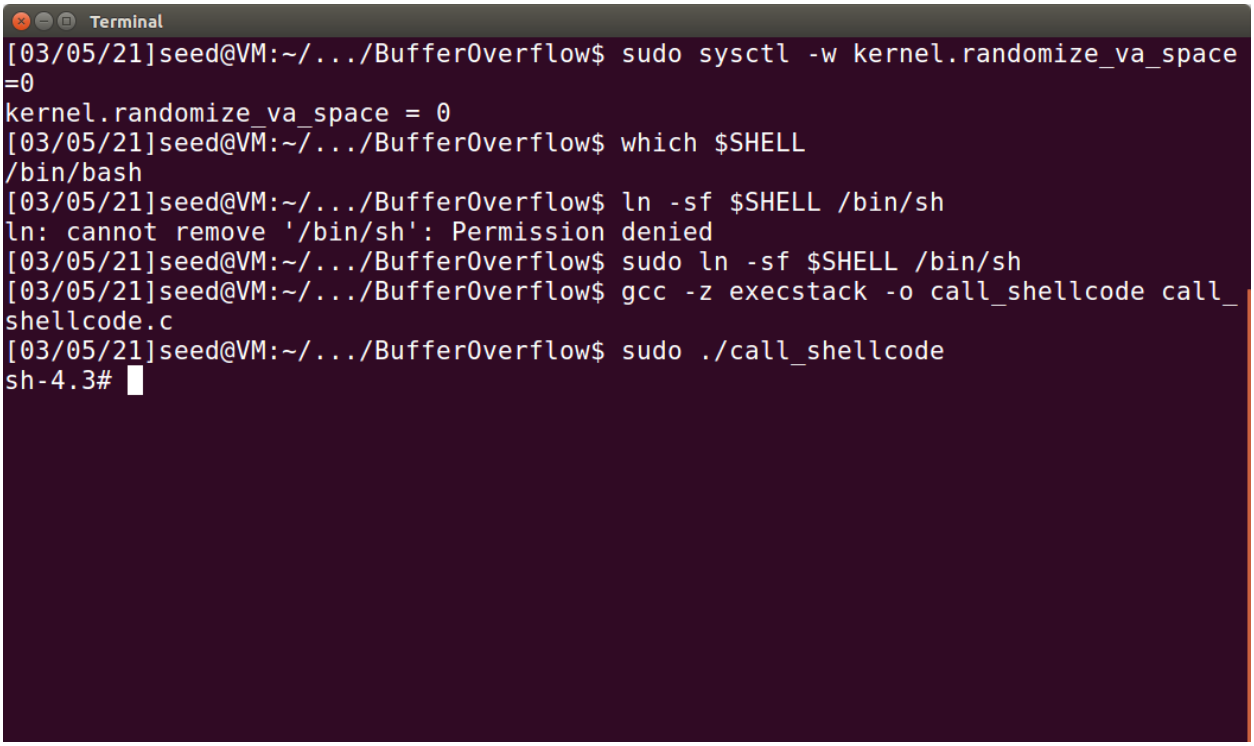
Task 1: Running Shellcode



What I observed:

- In this task, I observe that call_shellcode is trying to execute the assembly instruction which can open /bin/sh.
- The shellcode is copied to the buffer using strcpy(buf, code);
- here is where the shellcode will be loaded to the stack.
- Then when buffer fetch by the Cpu, the shellcode get execute
- We need to put -z execstack to turn off stack guard since OS have mechanism to protect instruction from execution within the stack
- Shellcode is in assembly language, since we need the processor to directly load it from the stack memory, there are no compilation for our shellcode that why it must be in assembly format
- Since we using 32bit machine, each word in the shellcode need to be exactly 32bit in size
- Since we load the shellcode into the buffer using strcpy, the shellcode must eliminate any "\0" that may exists. This prevent strcpy from partially copy the shellcode to the buffer which will fail the attack

Task 2

What we need to exploit buffer-overflow on stack.c:

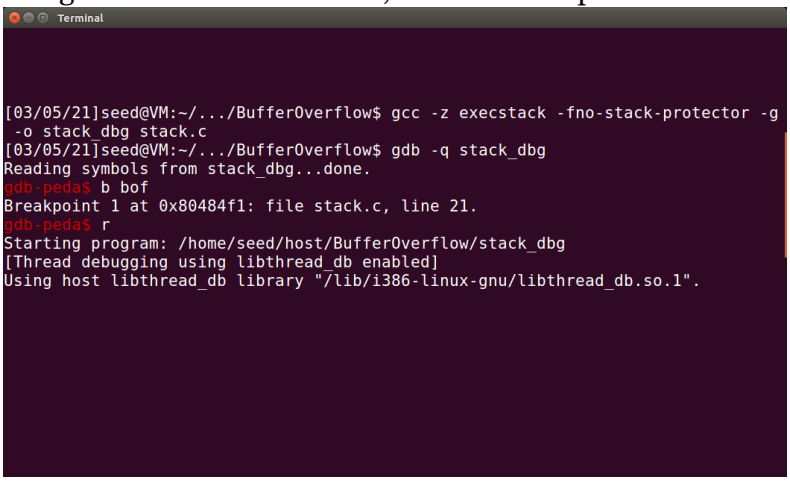
1. Offset from from the bottom of the stack to the address of ebp(stack frame)
2. Offset to the return address on the stack

Before we can calculate the address correctly we need to turn off the address randomization otherwise the starting our stack will be place randomly which make our job harder to determine the correct return address on the stack. We don't want to guess the address

Now, we can calculate the offset if we know the address of the ebp and the address of the buffer, then:

offset = address of ebp – address of buffer

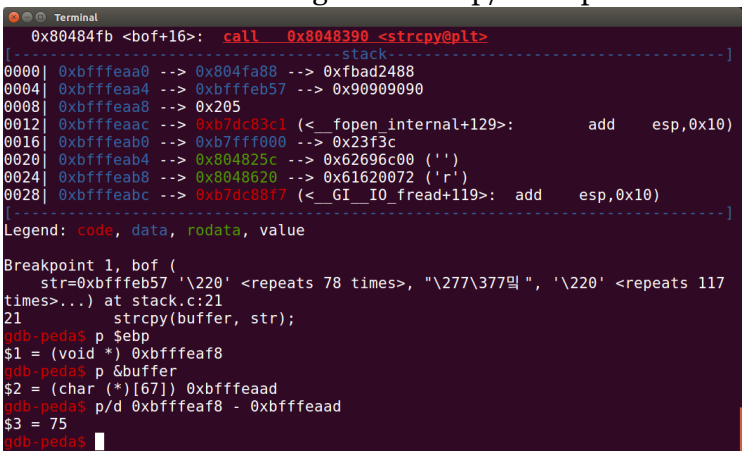
- we can found the address of ebp and buffer after we compile stack with -g for debugging
- Use gdb to examine our stack, make a break point at bof function



-
- Now we can print out the address of ebp by using gdb command
    - p %ebp

  And the address of the buffer using command

- p &buffer

- Calculate the offset using command p/d %ebp - &buffer
- 

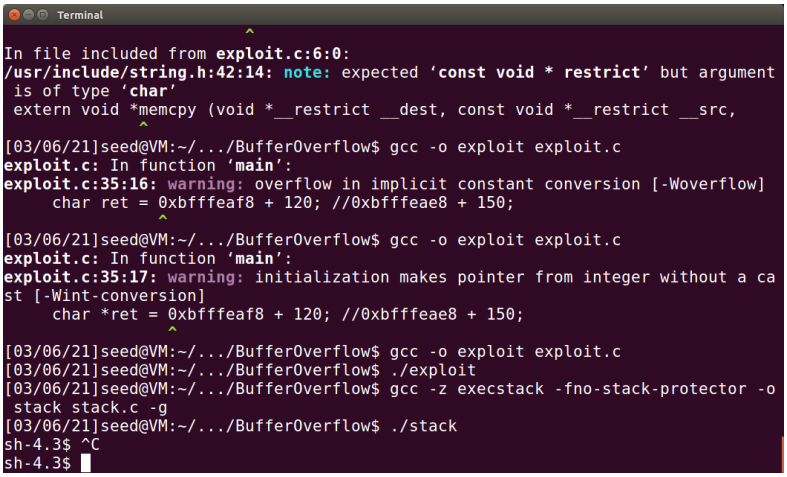
Then, having the offset we can now calculate the offset to the return address by add 4 byte (32bit address of ebp) to the offset. In this case the return address = 75(offset) +4 =79

Writing exploit code:

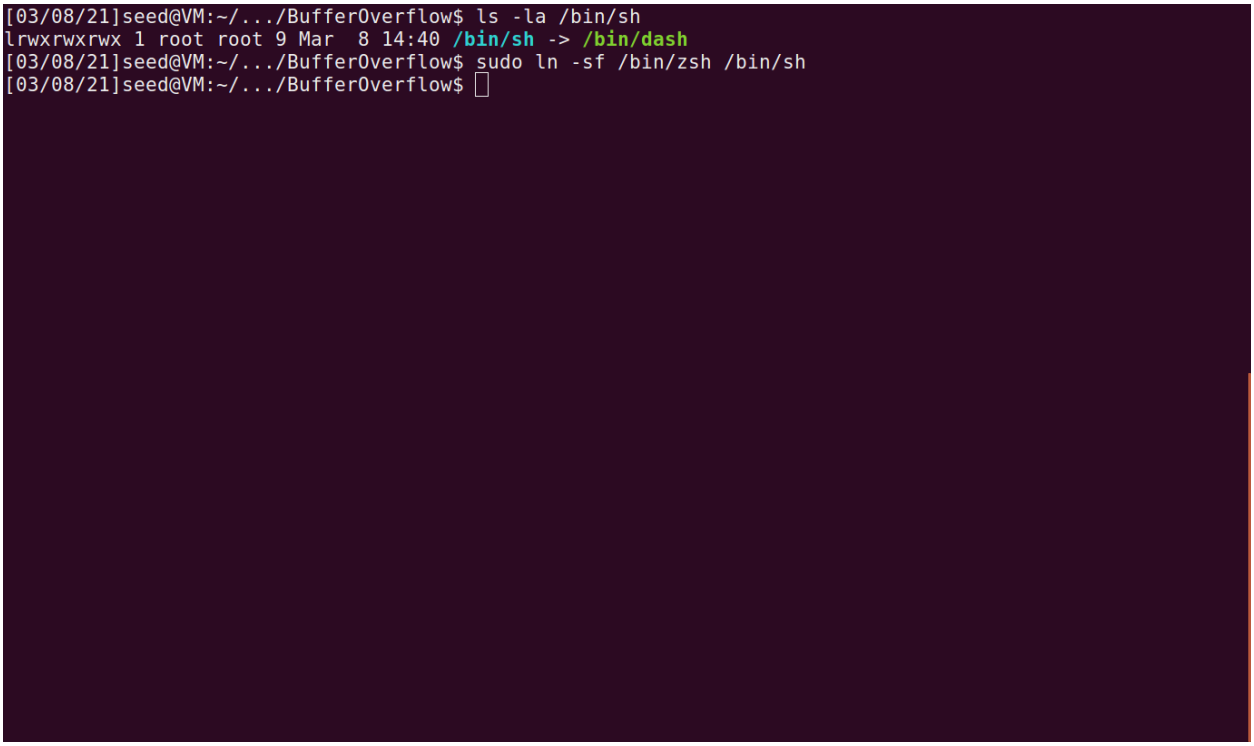
- Put our shellcode at the end of the buffer
- We know our return address is in 0xbfffeaf8($ebp) + 4 but we don't want the program to just return to the first intended address. We need to trick our program to jump to another place by putting some random address(ideally toward our shellcode but it must be before the the starting address of our shellcode) in the return address. I'm using 0xbfffeaf8($ebp) +120
- Put 0xbfffeaf8($ebp) +120 (4 bit) into the buffer at offset 79 which is our return address.

Execute the vulnerability code.

- We need to put -z execstack to turn off stack guard since OS have mechanism to protect instruction from execution within the stack



- Notice that we don’t gain root access because sh sublink to dash shell which drops the privileges when it detect if the real uId and the effective uid is not match
- We can solve this by link sh to another shell program. In this case, I point it to zsh shell



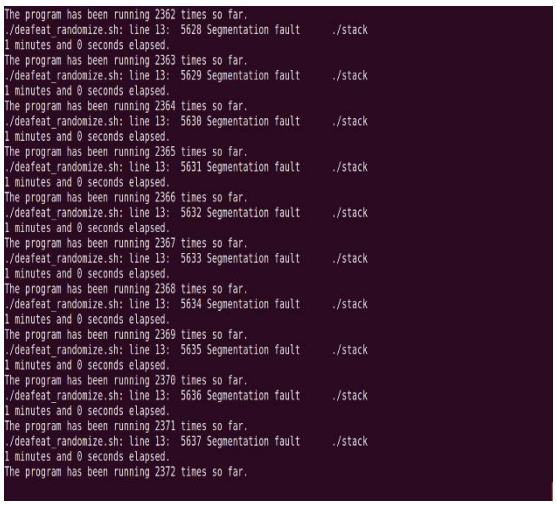
TASK 4

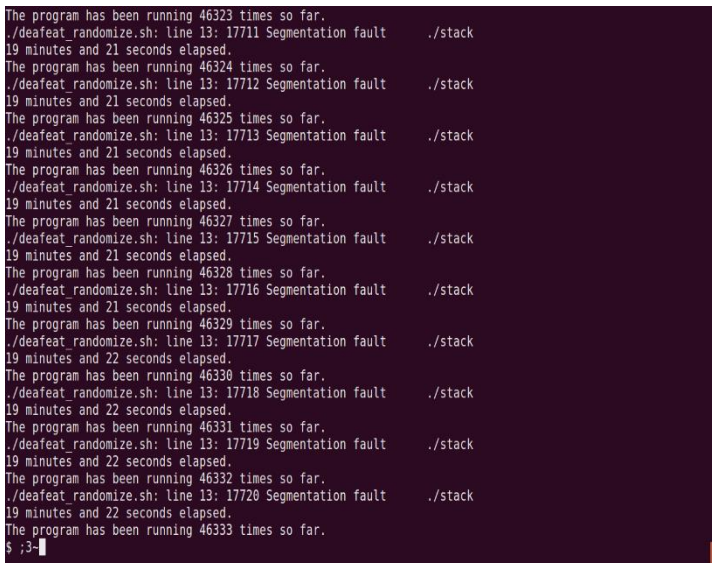
Since the purpose of this task is to use brute force to execute our vulnerability program util it is finally exploited.

We will leave our exploit code as is as task 2. But turn on the stack randomization and run the vulnerability code in in the while loop

With the stack randomization turn on, every time we execute a program, the OS will try to place starting address of the program randomly on the stack (heap as well). So It is really hard if we try to predict the address required for our attack. Since, in 32 bit system, we have 19 bit for stack and 13 bit for heap. Guessing the address with 2^19 possibility could also be done but we are heading to another direction.

Using only fix address that we already set up in task 2, we can run the vulnerability program indefinitely. The OS will shuffle the starting address of program, meaning every time our program get executed, our program will get new starting address on the stack. What we are trying to do is waiting until the OS place our program to the address that we have defined in the exploit code.





What I observed:

- we get segmentation fault on every time the ./stack get executed, except for the last time.
    - This is because our address that we put into the buffer is incorrect. It tried to access some where else which it isn't allowed to go into. we will always get seg fault in this case.
- After roughly 19 minute and 46333 trial, our program exploited. This is because our address defined matches the address on the stack
- 46333 total trial but only 17720 times that we get segmentation fault
    - This maybe because our defined address has landed to somewhere that we allow to go to (ideally not kernel space address)

After using Ctrl+d to terminated the shell, the program still keep running.