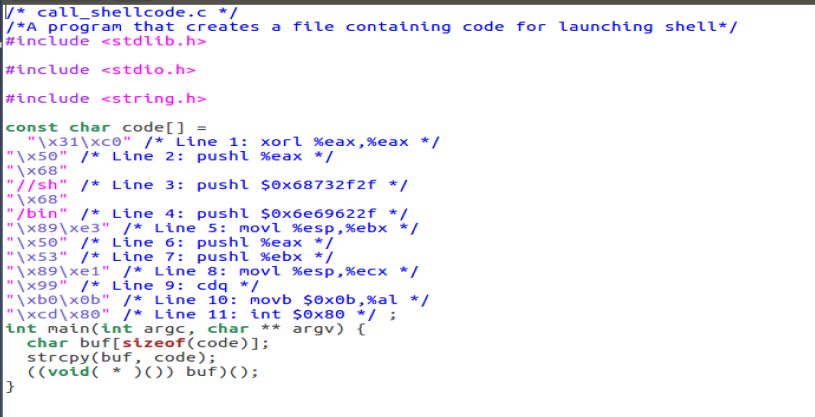
LAB #3 Buffer Overflow Vulnerability

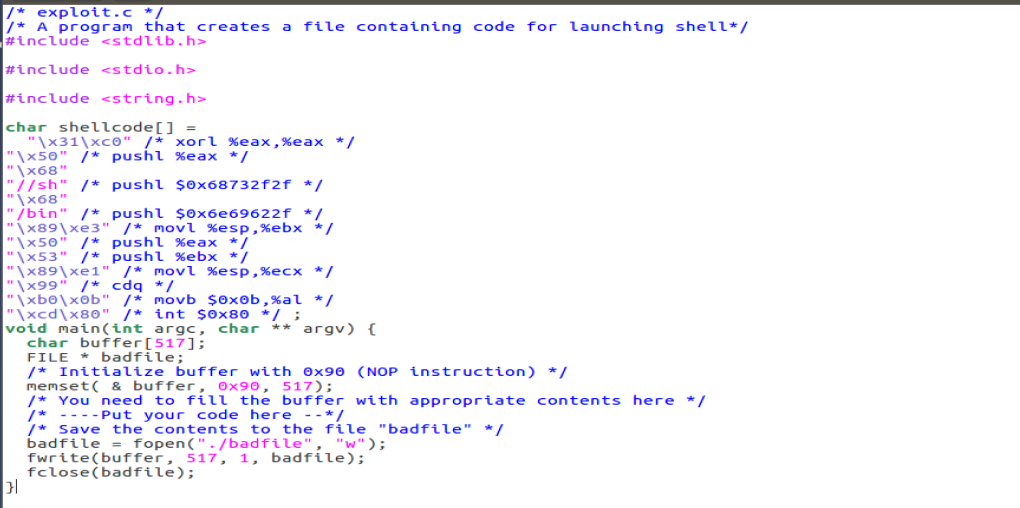
STEPHEN ANTWI

**Running Shellcode.**

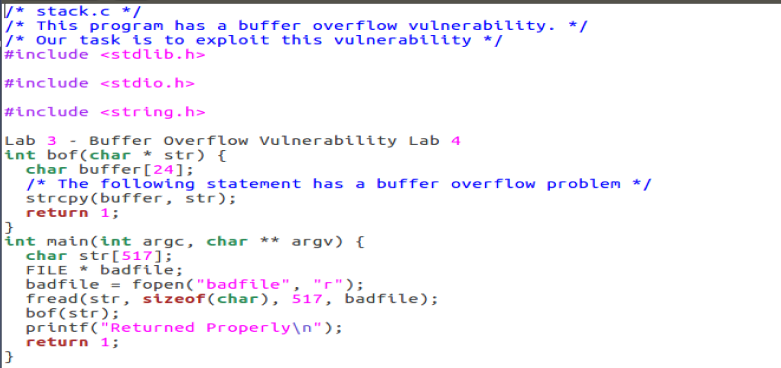
Create the various C++ codes needed for this lab as shown below, while ensuring each code is given its appropriate name.



For instance, the first code is call\_shellcode.c as shown above.



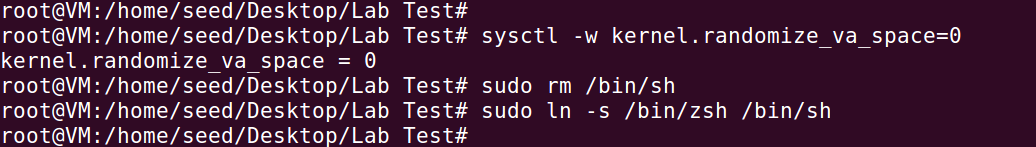
The second one is identified as exploit.c as shown above.



The last one is identified as stack.c as shown above

**Creating and Running the Shell Code**

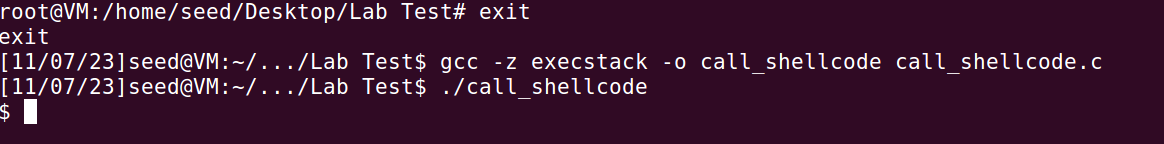
Next would be to open the terminal inside the test folder used for the lab, as shown below



The above countermeasures are executed to make the environment suitable for the lab.

1. The randomization is disabled.
2. The version of Ubuntu in use is version 16, bin/sh needs to be linked to another shell zsh.

Next is the execution of the sell code as shown below.

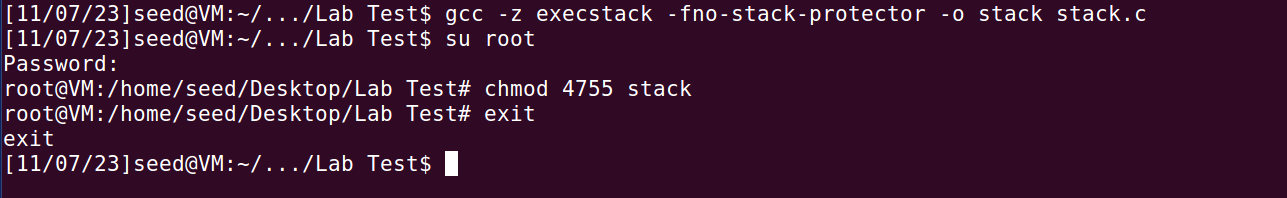


The root account is exited and the shell code is compiled and opened as shown.

Once done this will invoke the shell as shown in the image above.

**Creating the Vulnerable Program**

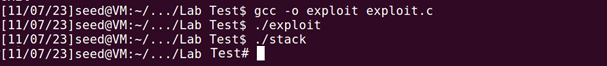
Next would be to create the stack program with a buffer flow vulnerability as shown below and assign a Set-UID with administrative privileges to the stack program.



**Exploiting the vulnerability.**

After the stack is compiled, the next procedure would be to compile the exploit and run the exploit file followed by the stack file created.

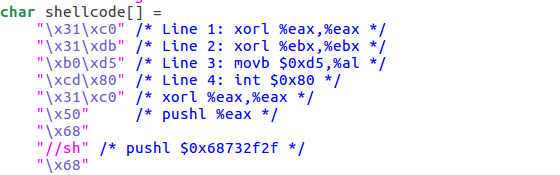
First the exploit.c file is compiled to create the badfile needed for the vulnerable program.



After that, the attack is launched by executing the exploit exe and stack exe as shown above. The exploit file makes it possible for the stack program to trigger a buffer overflow. Hence the user account is granted root privileges as shown above.

**Defeating the countermeasure.**

The following part is embedded in the code to help bypass the countermeasure of the system.

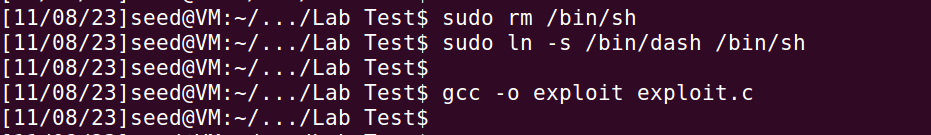


This part of the code is defining a shellcode as a character array. Below is a breakdown of each segment of the code

* The `\x31\xc0`: is similar to the assembly instruction `xorl %eax,%eax`, it adjusts the value of the `eax` register to 0.
* The `\x31\xdb`: is similar to the assembly instruction `xorl %ebx,%ebx`, it adjusts the value of the `ebx` register to 0.
* The `\xb0\xd5`: is similar to the assembly instruction `movb $0xd5,%al`, which moves the value `0xd5` into the `al` register.
* The `\xcd\x80`: is similar to the assembly instruction `int $0x80`, which is the system call instruction on x86 architecture. It is used to make system calls to the kernel.

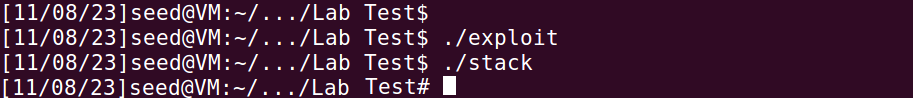
**Stack Guard**

With the code readjusted, the following commands “/bin/sh”, “/bin/zsh” are used to remove the symbolic link and create a new symbolic link to bypass countermeasures in the dash shell.



Afterward, the exploit is recompiled, as shown above. In this instance the stack.c is compiled without disabling Stack Guard.

With that done, the following step would be creating the badfile and launching the attack as shown below.



Like in the first instance, the shell changes, and the attempt results in root shell as shown above.