Buffer Overflow Lab Report

The BUF\_SIZE for this lab is: 236 bytes.

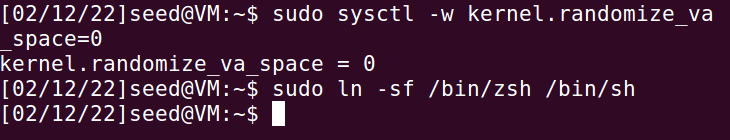
Lab Task 1: Turning off Countermeasures

Address space randomization: $ sudo sysctl -w kernel.randomize\_va\_space=0

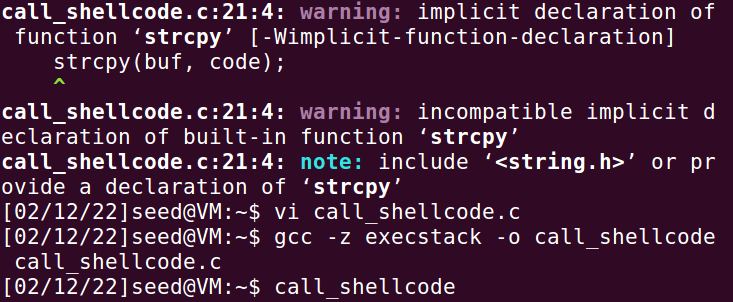
Configuring /bin/sh: $ sudo ln -sf /bin/zsh /bin/sh

Disabling StackGuard Protection (during compilation): -fno-stack-protector

Non-Executable Stack (during compilation): -z noexecstack

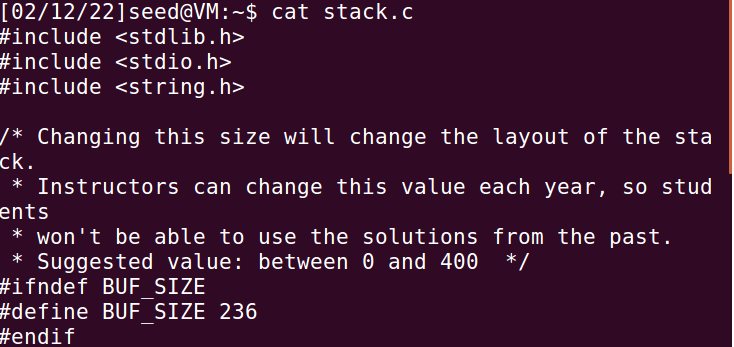


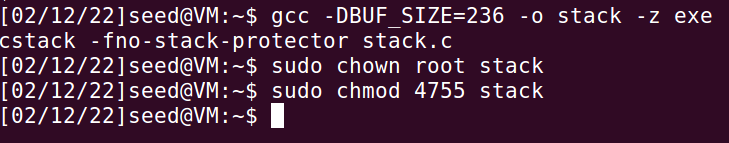
Lab Task 1: Run Shellcode



Observation: The version of the code I downloaded off the website did not have #include <string.h>, so I got two compiler warnings on the first compile. Fixed and continued.

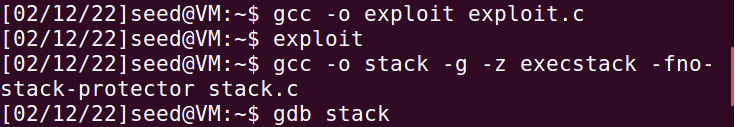
Lab Task 2: Exploiting the Vulnerability



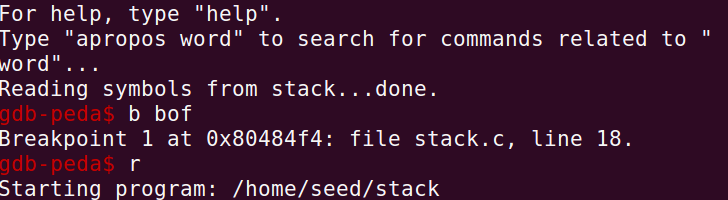
Note: Copied the given program stack.c into the virtual machine, changing the BUF\_SIZE to 236 for this lab. After the fact of manually changing the size, I realized that it could be done from the command line when compiling and changing the settings of the file. 

Lab Task 2.4: Exploiting the Vulnerability

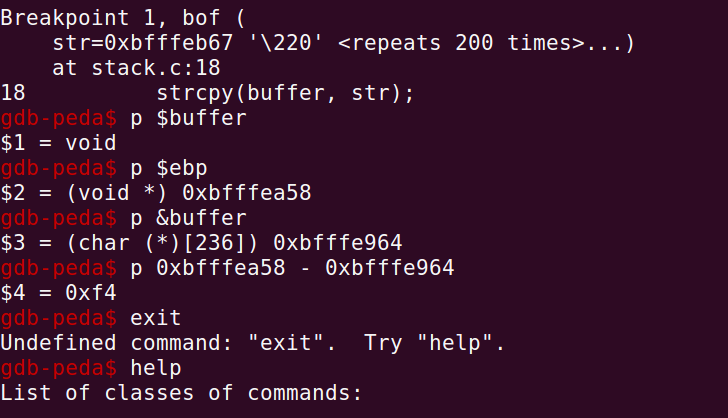




Set up exploit.c and badfile. Then run stack in debugger.

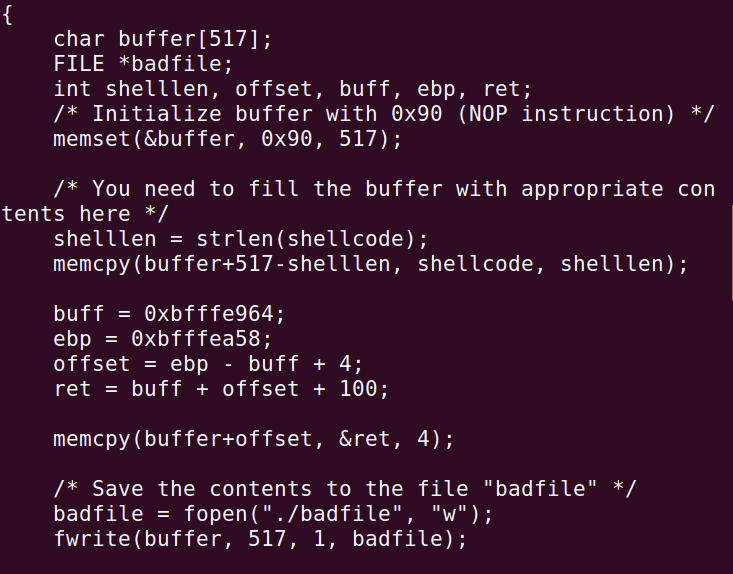


Set a breakpoint at bof since that is where the vulnerability is, then run stack.

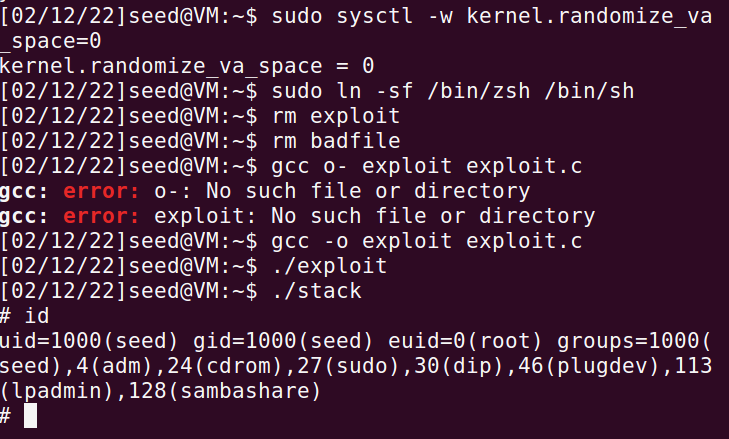


$1 was a typo, but the location of $ebp is 0xbfffea58 and then location of the buffer is 0xbfffe964. We find that $ebp - &buffer = 0xf4, or 244 in decimal. Therefore, the return address needs to be 244+4=248, but any number that lands us into the chain of NOP instructions will work.

Note: To exit debugger, the correct command is q or quit, not exit as I found out.

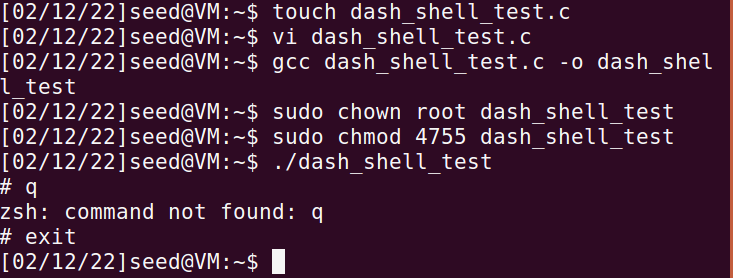


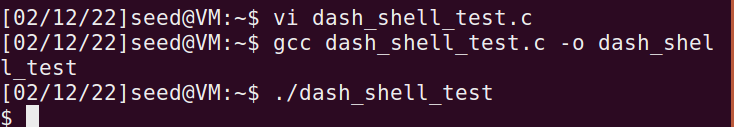
This code calculates the distance to move the pointer with (ebp - buff) + 4 + buffer length + 100 (to ensure that the pointer is in the NOP zone.



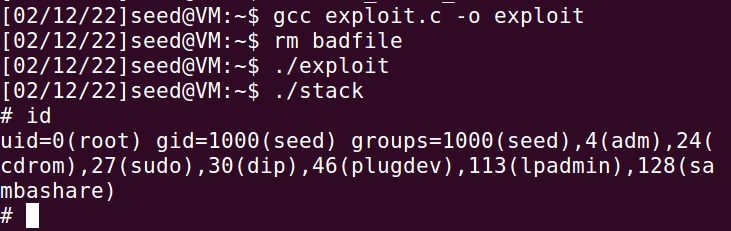
After much confusion (involving several resets and some strange result that make my backspace key add indentation), it worked, and I obtained root access. I’m not sure if the reason the attack wasn’t working was because I failed to remove the countermeasures before trying exploit or if the exploit code was saved improperly, or if there was some other reason. At any rate, it worked in the end.

Lab Task 3: Defeating dash’s Countermeasure



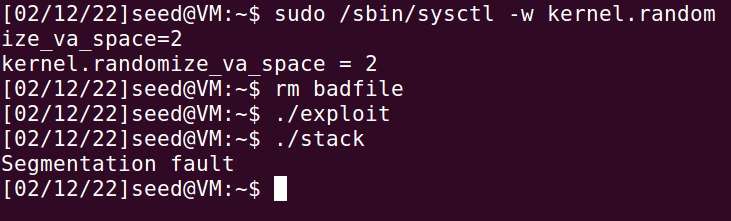
When the test is run with the owner as root, the program will give root permissions. 

This time, the setuid(0); line was not commented out and running the program will no longer give root access. As we can see, the dash countermeasure was successful when the program does not set uid to root before calling execve().

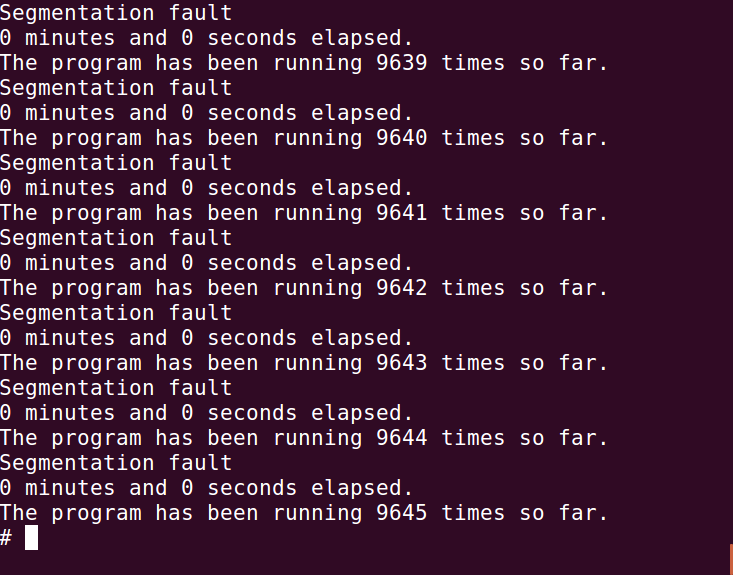


Adding four lines of shellcode to exploit.c and recompiling, I found that the attack works even when /bin/sh is linked to /bin/dash. The attack from task 2 will still get a root shell by overcoming dash’s countermeasure.

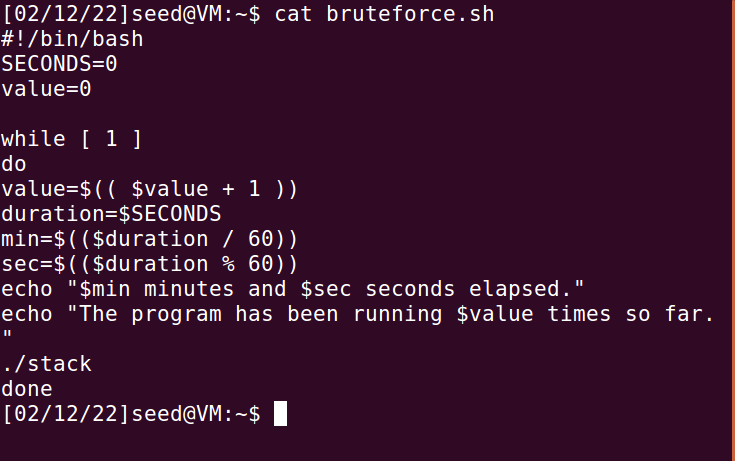
Lab Task 4: Defeating Address Randomization



Attempting the attack from task 2 after activating address randomization leads to a segmentation fault. It is clear that the address has moved and the attack will not work (in one try) since we no longer have the correct address.

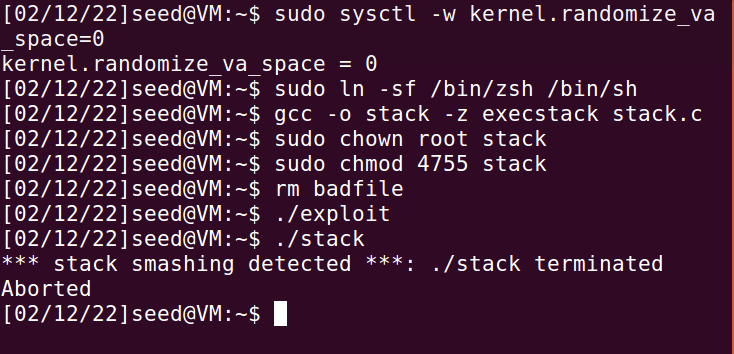


Following the instructions given, I ran a brute force attack to defeat the address randomization. It seems I got lucky, since there should be 524,288 possibilities and I obtained the root shell in 9,645 attempts. The timer seems to have broken, however. The program did not take long to complete, but I believe it was closer to 30 seconds than 0 seconds.



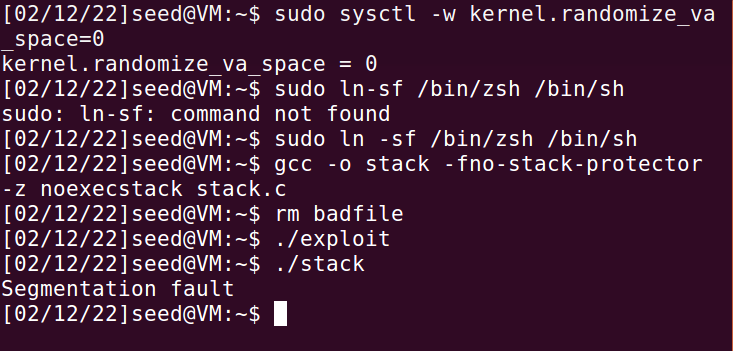
Here is an image of the shell code I used to repeatedly run the vulnerable file. This was copied from the lab pdf document.

Task 5: Turn on the StackGuard Protection



StackGuard can detect the attack and terminates the vulnerable file after reporting that an attempt at stack smashing was made. As a result, the attack failed, and I was not able to obtain a root shell while it was active.

Task 6: Turn on the Non-executable Stack Protection



With non-executable stack protection on, the attack failed, and I was unable to gain a root shell. This is because the shell code cannot be executed at all, preventing this method of buffer overflow attack from being able to gain access. While I am able to write into the stack, being unable to execute the code on the stack heavily neuters this method of attack.