Daniel Oliveros

Garrett Bogart

Spring 2018 – Independent Study

Return to Libc Vulnerability Lab

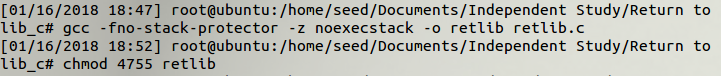
**Background:**

**Task 1:**

For this lab we need to disabling the stack and heap memory randomization.



Then the vulnerable program needs to be to be compiled with the no stack guard and a non-executable stack.

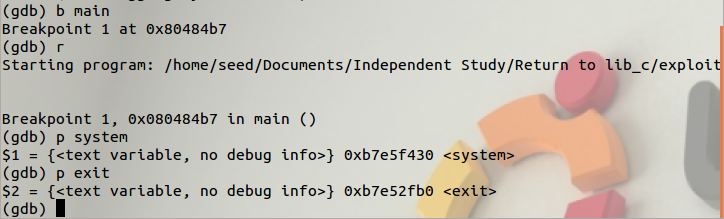


With the non-executable stack activated it is impossible to do the previous buffer overflow attack. Essentially it prevents any code on the stack to be executed, so our NOP sled and shell code would fail to run.

The solution is to realize that we can still run code that is already in memory. Since most programs are reliant upon the libc library, the library is loaded into memory every time a program is run. Within the libc library there is a system function call which can be used to call executable files. In this case we need the /bin/sh executable.

The first step is to find the memory locations of the system function and the exit function. The exit function isn’t that important in terms of the lab. The exit function will have the program exit successfully after the attack is done; otherwise, there is a segmentation fault. The exit function is useful if you want your attack to be undetected.

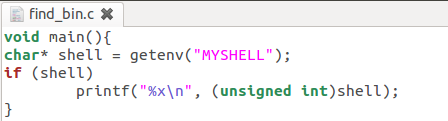
To find the memory location of these functions, execute the exploit executable in the debugger. Then set a breakpoint at main and then run the program. Then use p system and p exit to find the memory address of system and exit.

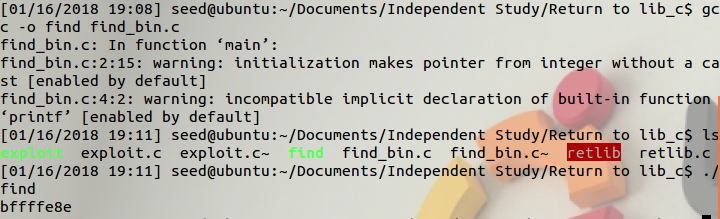


The next trick is to get the /bin/sh into memory, so that it can be executed as well. C programs inherit all environment variables from the shell that executes it. Meaning that we only have to declare /bin/sh as an environment variable and then find its memory address.

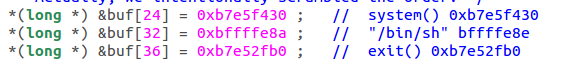


Then using the below program we can find the memory address of the environment variable.





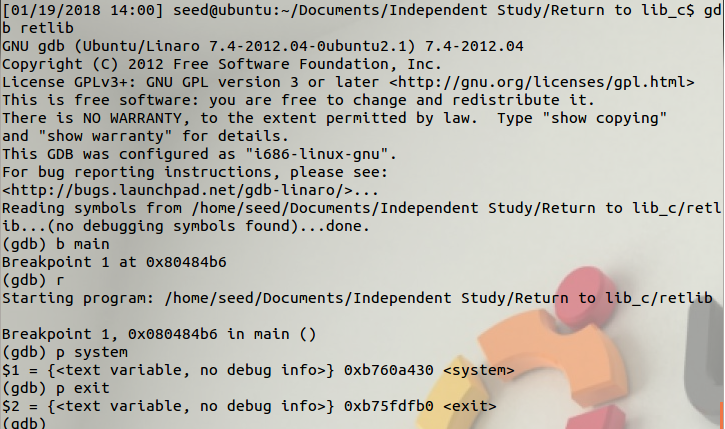
Now we have all three memory addresses that we need for the attack. The most time consuming bug we had was that the /bin/sh memory address was off by four. This is why the commented address is bffffe8e but our actual address turned out to be bffffe8a.

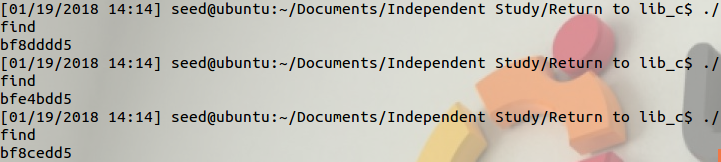




**Task 2**

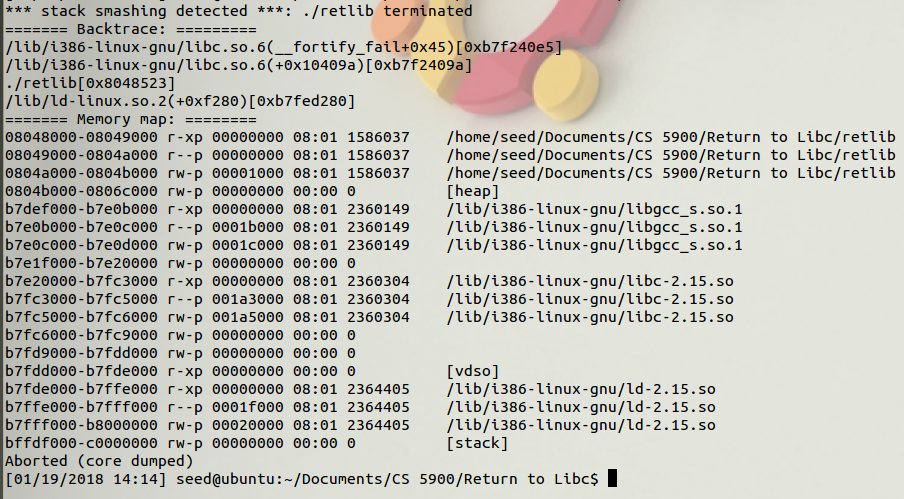
This attack is statistically impossible. All of the addresses are randomize every time, so the odds of getting the two necessary memory locations is infinitely small. The odds of getting all three are even smaller.





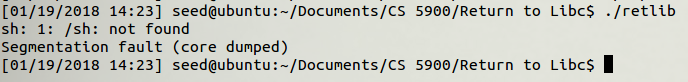
**Task 3:**

This attack is impossible to achieve with this protection either. Since the program detects the changes caused to the memory surrounding the eip, it instantly aborts the process and prints out the following error message



**Issues:**

1. We had to make a change that was never outlined to one of our memory addresses. For some reason it was not storing the location that we needed, so we had to decrease it by 4… This issue took a very long time to fix, as we couldn’t really tell what was going on from the error message, and none of the resources mentioned this as a possible issue.



**Ideas for Improvement:**

1. A great deal of the material in this lab just shows you assembly code and expects you to follow along, which isn’t necessarily ideal. A review session explaining really what you are supposed to be doing would be very useful. Some of the topics that would be worthwhile to mention are:
   1. You will basically emulate a fake stack for your system() call, this means putting the parameters for it (the /bin/sh address) 8 bytes down from your $eip’s location. This is not an easy concept to follow, and would be good to at least give them a general idea of where to begin.