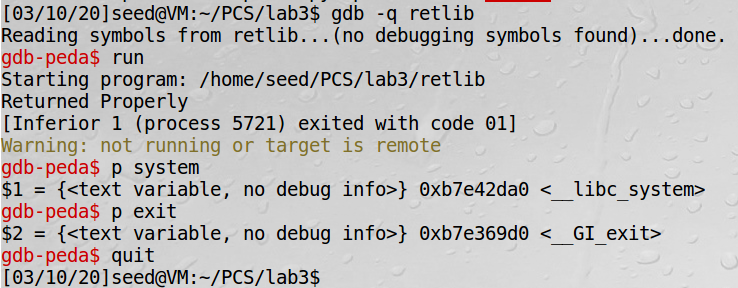
Q0: None

Q1: Address of **system**: 0xb7e42da0; Address of **exit**: 0xb7e369d0

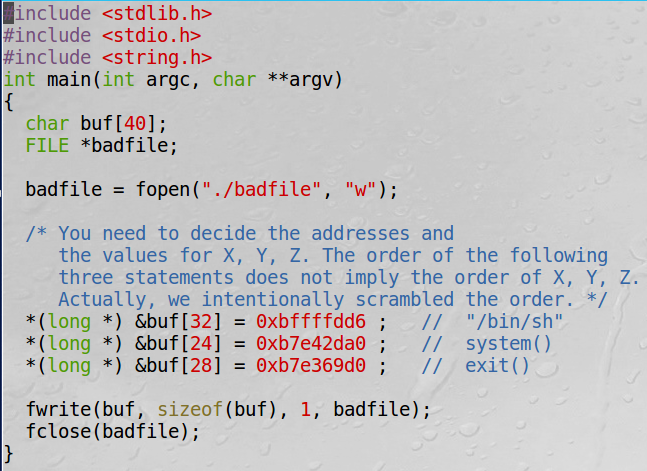


Q2: The address of the environment variable MYSHELL is 0xbffffdd2

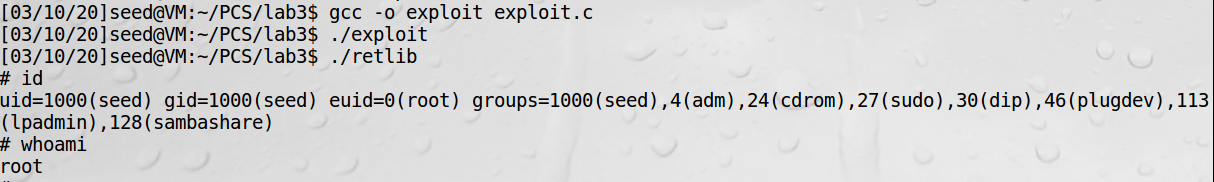


Q3:

1. Code for the exploit program:

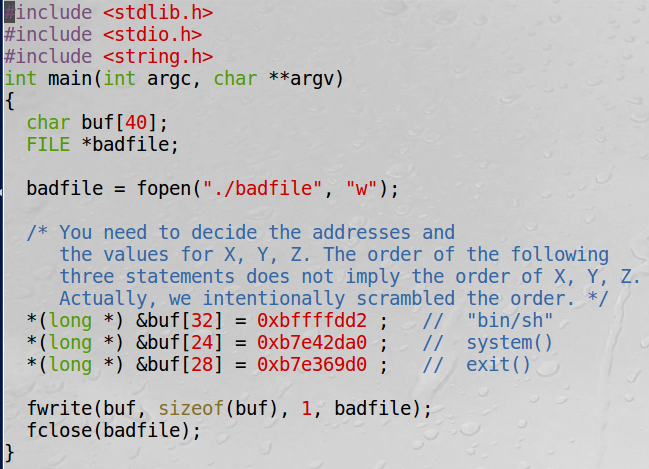


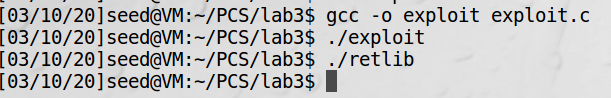
1. A success run of the retlib program.



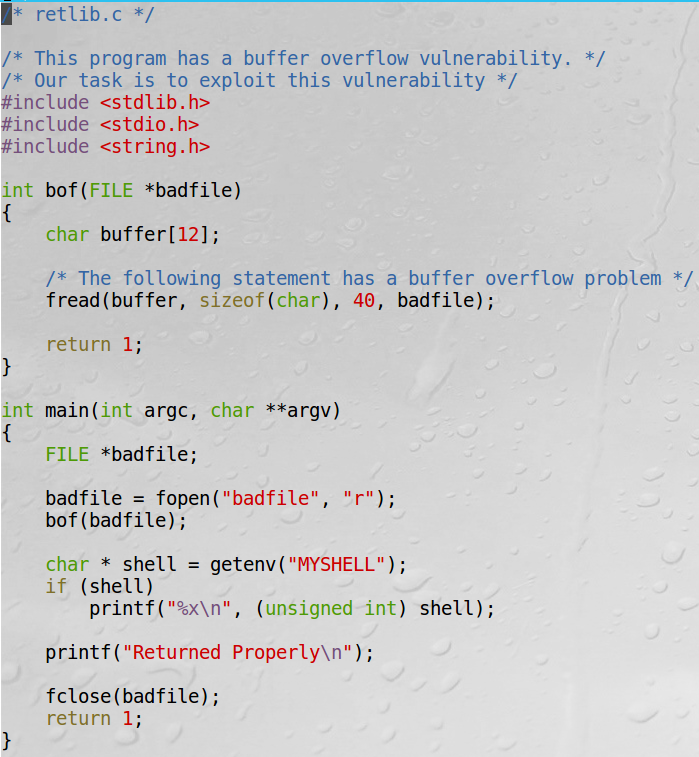
1. Description of how the attack is performed and my reasoning.

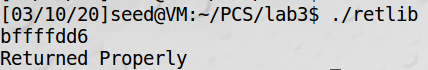
* The high-level idea is to use buffer overflow to overwrite the return address of the **bof** function by the address of the **system** function in order to hijack the control flow.
* Steps:
  + First, find out the address where the **system** and the **exit** functions are stored in the program **retlib**. This was done in Question 1.
  + Next, provide the **system** function with the argument “/bin/sh” at invocation. To achieve this, we created a shell variable **MYSHELL.** This shell variable will be in the memory and we found out its address in Question 2.
  + Then, determine the order to put the three addresses (address to **system**, address to **exit**, and address to shell variable **MYSHELL**) in the stack according to the stack layout. The order is (from low to high): **system**, **exit**, **MYSHELL**, where **system** overwrites the return address of **bof**, **exit** follows right after **system** to provide the return address for **system** after it finishes, and **MYSHELL** right after **exit** to provide **system** with its required argument.
  + Then, find out the address where the return address of **bof** is stored. Note that we don’t have to know the absolute address of the return address. Instead, once we know the offset between the staring address of the local variable **buffer** and the return address, we can construct the content of **badfile** to overwrite the return address using that offset.
  + Finally, overwrite the return address by the address of **system**, followed by the address of **exit**, then the address of **MYSHELL**.
* Interesting findings:
  + The address of **MYSHELL** as found out in Question 2 didn’t work at first.
  + Solutions:
    - One solution is to use trial-and-error, adding (or subtracting) 4 bytes to the address on each trail, since the address of the shell is quite close to what I previously printed out.
      * Trail-and-error: using 0xbffffdd2 didn’t work





* + - * Adding 4 bytes, using 0xbffffdd6, success! (screenshots shown in (1) and (2))
    - Another solution is to put the **getenv()** functioninside the **retlib** program to directly print out the address of the environment variable. This is shown below.
      * However, in reality we usually cannot alter the source code of the vulnerable file. So this is not a valid way to figure out the address of the environment variable.





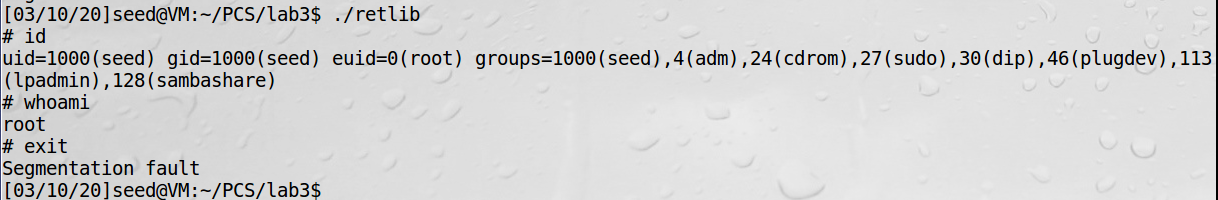
Q4: The attack after changing the file name was unsuccessful. This is because the address of the environment variable **MYSHELL** has changed after the file name changed, and **system** no longer receives ”/bin/sh” as its argument.



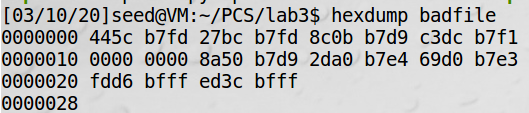
Q5:

1. Determine the offset between buffer and **ebp** register.
2. Determine the location of system() located in **process memory**.
3. Create a **shell variable / environment variable** to hold the string “/bin/sh”
4. The address of above variable is **0xbfffdd6**.
5. Find a **gadget** to load the above address into a register.

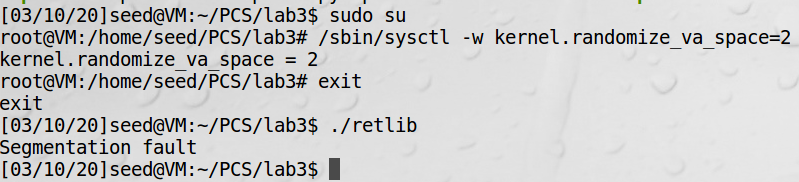
Q6: **exit()** is included because we want the program to terminate normally. That is, after **system()** function finishes, the control flow goes (returns) to **exit()** to terminate the program. Otherwise, after **system()** finishes, it returns to a junk return address and this might cause the program to abort (e.g., cause a segmentation fault). The **exit()** is not mandatory.



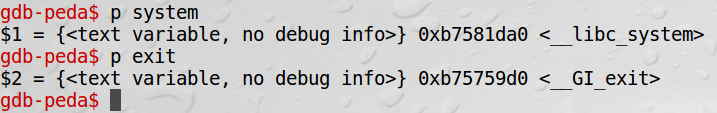
Q7:

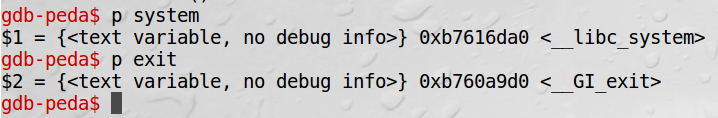


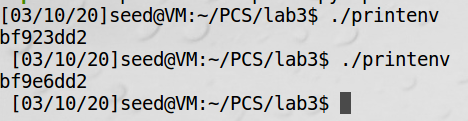
Q8: I could not get a shell. Instead, the program reports segmentation fault.



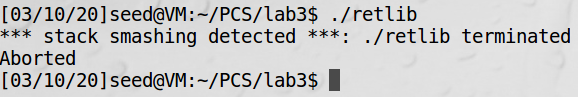
Explanation: Below are two consecutive runs of the **retlib** program and the **printenv** program. We can see that the addresses of **system, exit,** or **MYSHELL** are randomized at each run. This prevents the attacker from first running the program and peeking at the addresses of the functions or environment variables, and later using those fixed addresses to create a badfile that overflows the buffer to conduct the attack. The randomization make ret2libc attack difficult because the attack doesn’t know exactly what the addresses of the essential functions and variables will be at the time he executes the program (conducts the attack).







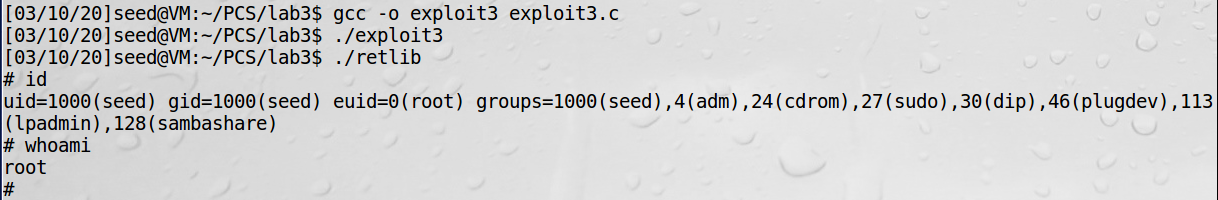
Q9: I could not get a shell. Instead, the program reports “stack smashing detected” and aborted.



Explanation: When stack guard is enabled, it places a canary word next to (before) the return address. Therefore, when a malicious buffer overflow overwrites the return address, it will also have overwritten the canary word. When the function returns, it first checks to see that the canary word is intact before jumping to the return address. In our scenario, the return address as well as the canary word have been overwritten, therefore the stack overflow is detected and the program aborts. The stack guard protection makes ret2libc attack difficult because the attacker has to know what the canary word is in order to overwrites the return address while keeping the canary word intact.

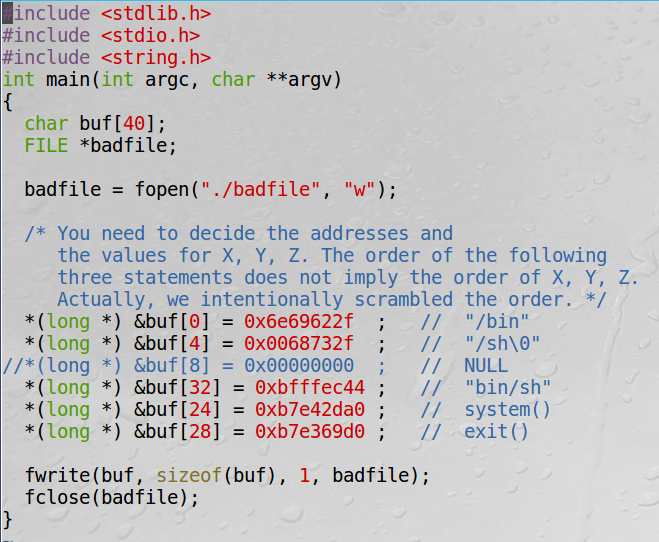
Q10: Instead of storing “/bin/sh” in an environment variable, we can store the argument string directly on the stack (using buffer overflow). Then we only need to figure out the address of the argument on the stack. To be specific, the argument string “/bin/sh” is stored at the first 7 bytes of the buffer **buf**.

A success attack:

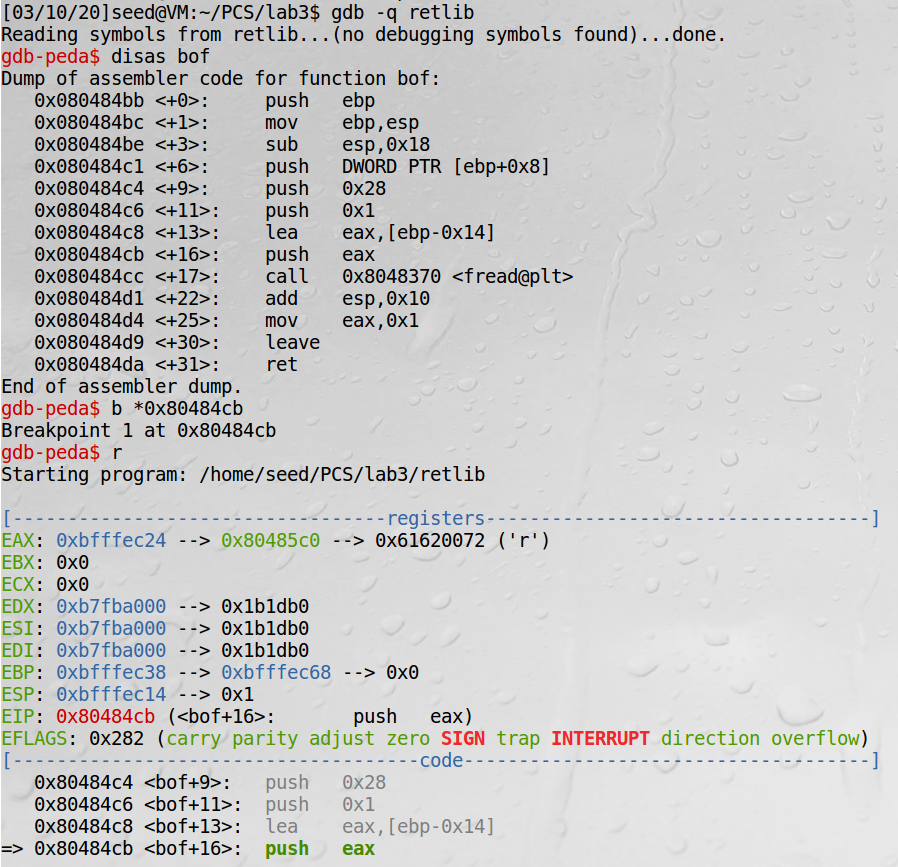


Implementation:

* Store the argument string “/bin/sh”(7 bytes) at the start of the buffer, followed by a zero (1 byte) to terminate the string.



* Run gdb on **retlib** and disassemble **bof** to figure out the approximate starting address of buf.
  + buf is located at $ebp - 14, which is 0xbfffec24.
    - The stack layout is different when running without gdb, but the address of **buf** will be very close. By trail-and-error, I found the address of buf is 0xbfffec44 when running without gdb.





* Finally, overwrite the argument to **system** with the address of the argument string “/bin/sh”, which is the starting address of the buffer **buf** instead of the address of the environment variable **MYSHELL** that we previously used.

Observations:

* In the above implementation, the argument string is put at the start of the buffer (the first 7 bytes). However, if the argument string is put at a higher address in the buffer, then the attack won’t succeed. My guess is that the argument string gets overwritten by the stack for **system**. We are lucky that the stack for **system** didn’t grow low enough to overwrite the first 8 bytes of buffer in the stack for **bof**.
* The address for the buffer **buf** is 0x20 bytes off compared to what I obtained under gdb.