Dreamhack memory-leakage Writeup

```c
struct my_page {
    char name[16];
    int age;
};

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);
}
```

```c
int main()
{
    struct my_page my_page;
    char flag_buf[56];
    int idx;

    memset(flag_buf, 0, sizeof(flag_buf));

    initialize();

    while(1) {
        printf("1. Join\n");
        printf("2. Print information\n");
        printf("3. GIVE ME FLAG!\n");
        printf("> ");
        scanf("%d", &idx);
        switch(idx) {
            case 1:
                printf("Name: ");
                read(0, my_page.name, sizeof(my_page.name));

                printf("Age: ");
                scanf("%d", &my_page.age);
                break;
```

```c
            case 2:
                printf("Name: %s\n", my_page.name);
                printf("Age: %d\n", my_page.age);
                break;
            case 3:
                fp = fopen("/flag", "r");
                fread(flag_buf, 1, 56, fp);
                break;
            default:
                break;
        }
    }

}
```

문제에서 준 코드이다. 이 코드로 동작되는 프로그램은 이런 순서로 실행된다.

1. 프로그램 초기화 (initialize) 후 사용자에게 메뉴를 출력한다.
2. 사용자는 메뉴 중 하나를 선택한다:

   1: 사용자 정보를 입력.

   2: 입력된 사용자 정보를 출력.

   3: /flag 파일에서 데이터를 읽어 flag 값을 flag_buf에 저장.
3. 선택된 메뉴에 따라 동작을 수행한 뒤, 반복문으로 돌아간다.
4. 30초가 지나면 프로그램이 종료된다.

문제명은 memory_leakage인데, 여기서 **메모리 누수(memory leak) 현상**은 컴퓨터 프로그램이 필요하지 않은 메모리를 계속 점유하고 있는 현상이다. 할당된 메모리를 사용한 다음 반환하지 않는 것이 누적되면 메모리가 낭비된다. 더 이상 불필요한 메모리가 해제되지 않으면서 메모리 할당을 잘못 관리할 때 발생한다.

즉, 이 문제에서는 코드를 만든 개발자가 의도하지 않은 방향으로 메모리를 활용해야 한다.

```
case 2:
    printf("Name: %s\n", my_page.name);
    printf("Age: %d\n", my_page.age);
    break;
```

입력된 사용자 정보를 출력하는 이 부분을 보았다. printf에서 %s는 문자열을 출력할 때, 널 바이트(₩0)를 만날 때까지 메모리에서 데이터를 읽는다.

즉, 문자열 끝에 널 바이트가 없는 경우, 계속해서 메모리를 읽으면서 예상치 못한 데이터를 출력할 수 있다.

```
switch(idx) {
    case 1:
        printf("Name: ");
        read(0, my_page.name, sizeof(my_page.name));

        printf("Age: ");
        scanf("%d", &my_page.age);
        break;
```

메모리에 사용자 정보를 입력하는 부분인 case 1의 이 부분을 다시 보았다.

```
struct my_page {
    char name[16];
    int age;
};
```

선언부에서 확인할 수 있듯이, case 1에서 입력받는 my_page.name 배열은 read(0, ...)로 최대 16바이트를 입력받는다.

하지만 이 때 널 바이트를 자동으로 추가하지 않는다.

즉 case 1의 입력 데이터의 마지막 바이트에 널 바이트가 없으면 case 2에서 printf는 my_page.name 뒤에 있는 데이터를 계속 읽어 출력하게 된다.

```
{
    struct my_page my_page;
    char flag_buf[56];
    int idx;
```

My_page.name 배열의 크기는 16바이트이며, flag_buf 배열의 크기는 56바이트이다. 하지만 아마도 중간에 있는 my_page.age 배열의 크기는 코드에서 확인할 수 없으므로 직접 확인해야 한다.

Gdb를 이용해서 my_page.age 배열의 크기와 정확한 위치를 알아보려 했으나 거의 처음 다루어보는 거라 정확히 알아낼 수 없었다.

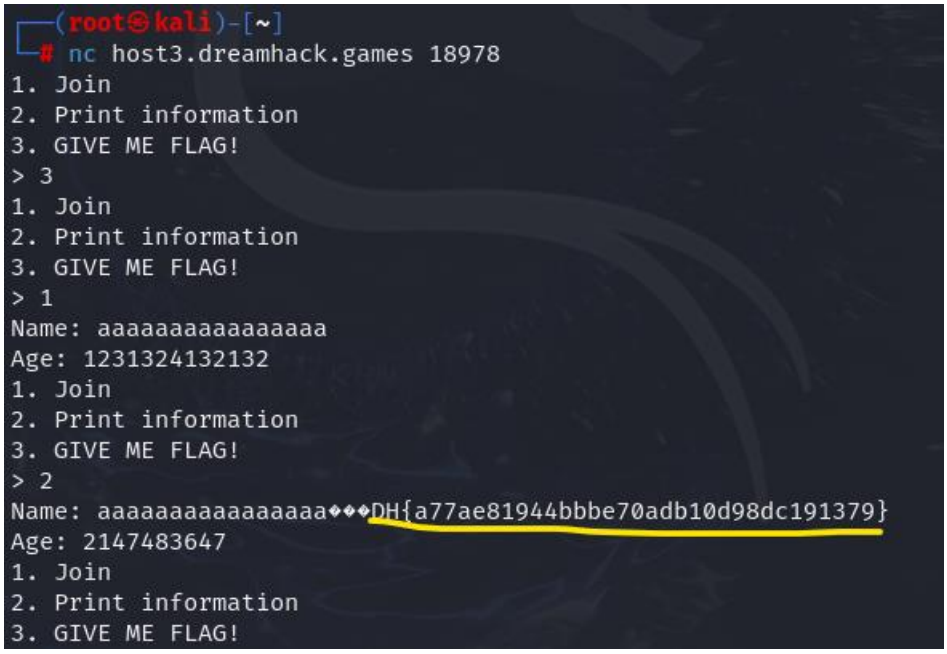
하지만 일단 큰 값을 넣어보면 플래그를 얻을 수 있을 것 같다는 생각이 들었다. 나이를 입력하는 부분이므로 배열의 크기를 그리 크게 설정해 두지는 않았을 것 같았다.

풀이의 정확한 방법을 설명하자면, 일단 %s는 널 바이트를 만날 때까지 데이터를 출력한다. 이때 만약 my_page.name에 널 바이트가 없다면 my_page.name의 끝에서부터 계속 내용을 출력한다. 먼저 메모리 상에서 아마 뒤에 있는 my_page.age의 값을 읽어 출력하고, 여기서도 널 바이트를 만나지 못하면 flag_buf에 저장된 데이터까지 읽어서 출력할 것이다.

즉 my_page.name에 16바이트를 꽉 채워 넣고, 아마 위치상 중간에 있을 my_page.age도 큰 값을 무작위로 넣으면 flag_buf에 저장된 플래그를 얻을 수 있다는 것이다.

이때, flag_buf에 플래그를 저장해 놓아야 하므로 먼저 플래그를 저장하는 case 3 명령을 제일 첫 번째로 시행해야 한다.

실제 실행 화면은 다음과 같다.



이렇게 플래그를 얻을 수 있다.