

Lab 03

1. uRC4 algorithm

```
# server.py

def PRGA(S):
    i = 0
    j = 0

    while True:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]

        # K = S[(S[i] + S[j]) % 256] # vanilla RC4
        K = S[j] # Charlie's modification for uRC4
        yield K
```

RC4 알고리즘은 Stream Cipher의 일종으로, 키를 통해 S-Box를 만드는 **Key Scheduling Algorithm** (KSA)과 이 S-box를 통해 랜덤한 값을 얻는 **Pseudo-Random Generation Algorithm** (PRGA)으로 이루어져 있다.

uRCA 알고리즘에서는 PRGA의 구현 부분에서 차이가 있는데, 기존의 알고리즘에서는 $K = S[(S[i] + S[j]) \% 256]$ 을 이용해 값을 생성한다. K를 계산할 때, $s[i]$ 와 $s[j]$ 의 값을 이용하기 때문에, K만을 통해서 S-box의 어디에서 값이 왔는지를 알기 어렵다.

하지만, uRCA알고리즘에는 $K = S[j]$ 를 통해 값을 생성한다.

```
def PRGA(S):
    i = 0
    j = 0

    while True:
        i = (i + 1) % 256
        # s_i = S[i]

        j = (j + S[i]) % 256 # we can predict j
        S[i], S[j] = S[j], S[i]
        # S[i], S[j] = ???, s_i

        K = S[j] # s_i
        yield K # s_i
```

위의 알고리즘을 해석해보면, 초기에 설정된 $i = 0$ 으로부터 1증가하면서, $s[i]$ 를 계산해 j 를 업데이트 하고, $s[i]$ 와 $s[j]$ 의 값을 바꾼다. 따라서, $s[j]$ 의 값에 s_i (기존의 $s[i]$ 값)가 들어가는 것을 확인할 수 있다. 이 값이 K 에 전달되어서, `yield`를 통해 출력된다.

따라서, i 번째의 K 값을 얻는다면, 이 keystream 과정에서 생성되는 $s[i]$ 를 i 의 위치에 정확하게 알 수 있고, j 도 구해서 keystream 생성과정을 따라갈 수 있다.

2. exploit server.py

```
# server.py
while True:
    ...
    if choice == 1:
        print("- Give me your plaintext:")
        plaintext = input()
        print(encrypt(keystream, plaintext))
    elif choice == 2:
        print("- Encrypted secret key:")
        print(encrypt(keystream, secret_key))
    ...
```

`server.py`를 보면 oracle을 통해 우리가 입력한 임의의 input을 encrypt해서 보여주는 것을 알 수 있다. 임의의 입력과 그 결과를 알아 낼 수 있으므로, known plaintext attack을 통해 keystream의 출력을 알 수 있을 것이다.

RC4의 암호화 과정은 다음과 같다.

```
# ^ is xor operation
(Ciphertext) = (Plaintext) ^ (Keystream)
```

xor 연산의 성질을 이용해 Plaintext를 전부 `\x00`으로 입력하면, Keystream을 자체를 얻을 수 있을 것이다.

```
# exploit-server.py
# i, j are global variable, leaked_keystream is leaked keystream.

r.sendline(b"\x00" * 0x1000) # dump keystream

...

global i, j
i = 0
j = 0

s_box = [-1] * 256

cnt = 0
```

```

while(1):
    s_i = int(dumped_keystream[cnt * 2:cnt * 2 + 2], 16) # hexstring to int
    i = (i + 1) % 256
    j = (j + s_i) % 256

    s_box[i], s_box[j] = s_box[j], s_i
    cnt += 1
    if cnt == 0x1000:
        break

assert -1 not in s_box

```

얻은 dumped_keystream 이용해 s-box를 위와 같이 복구할 수 있다. s-box의 길이는 256 bytes이지만, key를 생성할 때 같은 위치의 s-box에 접근할 수 있으므로, 모든 s-box의 값을 얻을 수 있도록 충분히 0x1000번 반복했다. s-box가 제대로 복구되었는지를 확인하기 위해서 assert문을 사용했다.

s-box를 복구했으므로, 서버의 s-box과 같은 값을 가지고, 서버가 만드는 임의의 암호문에 대해 key를 만들어 복구할 수 있게된다.

```

def decrypt(keystream, ciphertext):
    plaintext = ""
    for i in range(0, len(ciphertext), 2):
        plaintext += chr(int(ciphertext[i:i+2], 16) ^ next(keystream))
    return plaintext

```

암호화된 secret key를 받아 복구할 수 있다.

```

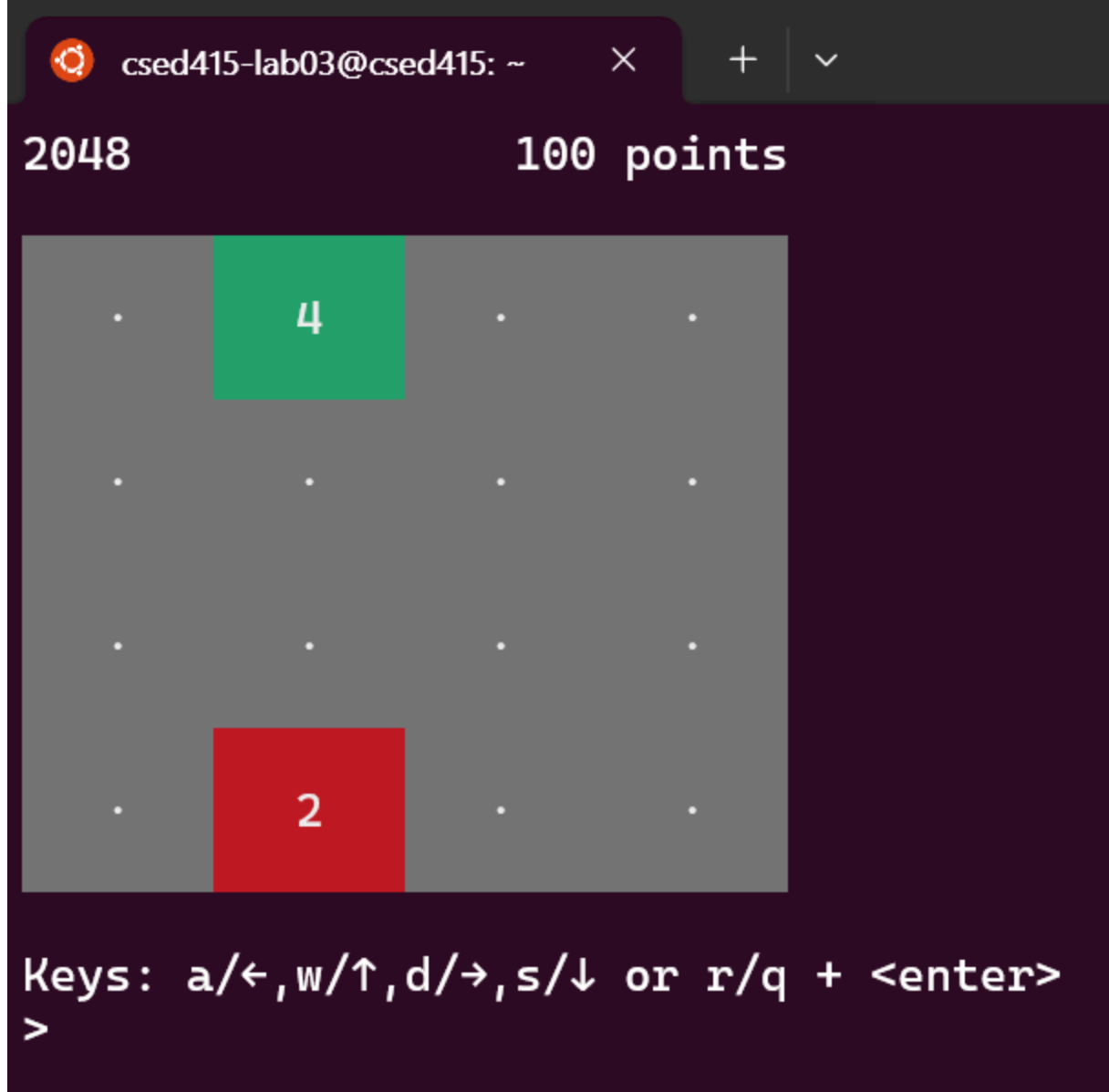
csed415-lab03@csed415:/tmp/jason9751/lab03$ python3 exploit-server.py
[+] Opening connection to localhost on port 10003: Done
e933a52be418ef6c160152d4a7581efeee51eb25ae2452cd0deec507cc957a65

[*] Switching to interactive mode

1. Encrypt your plaintext using uRC4 and print the ciphertext
2. Encrypt the secret key using uRC4 and print
3. Exit
>>> $ █

```

key: e933a52be418ef6c160152d4a7581efeee51eb25ae2452cd0deec507cc957a65



이를 `target` 에 입력 시, 게임이 잘 실행되는 것을 확인할 수 있다.

3. target vulnerability

```
// main.c

short score=100;
...

// in main()
while (true) {
    c = input[0];
    switch(c) {
        ...
        default: {
            score--;
        }
    }
}
```

```

if (success) {
    ...
    if (score > 3932156U) {
        printf("Congratulations! You've scored %lu points!\n", score);
        print_flag("2048");
        exit(0);
    }
}

```

코드를 보면, 방향을 나타내는 wasd와 방향키를 입력하지 않으면 `score`에 1점을 감점하고 있다. 그래서, 우리가 `score`를 음의 방향으로 조절할 수 있게 된다. integer underflow가 발생할 수 있는 상황이라고 볼 수 있을 것이다.

또한, 정상적인 입력을 했을 때 `success`가 활성화 되며, `score`를 비교하는 로직이 실행되며, 여기서 (unsigned int) `3932156U`와 비교가 된다.

https://en.cppreference.com/w/c/language/conversion#Usual_arithmetic_conversions

c언어에서 signed value와 unsigned value를 연산하면 Usual arithmetic conversions에 의해 signed value가 unsigned로 취급되어 계산된다.

위의 코드를 보면 `score`는 `short`의 타입을 가지고 있고, `3932156U`는 `unsigned int`의 타입을 가지고 있으므로, Usual arithmetic conversions와 Implicit conversions에 의해 다음과 같이 작동할 것이다.

```

if ( (unsigned int) score > 3932156U)

```

즉, `score`가 `unsigned int`로 type conversion 한 것과 같은 결과를 낸다.

우리는 위에서 틀린 입력을 통해 `score`를 음수로 만들 수 있는 것을 확인했다. 만약 `score`를 -1로 만들게 되면, 위의 비교 로직에서는 `4294967295`와 동일하게 작동이 되어, 게임이 클리어 하게 된다.

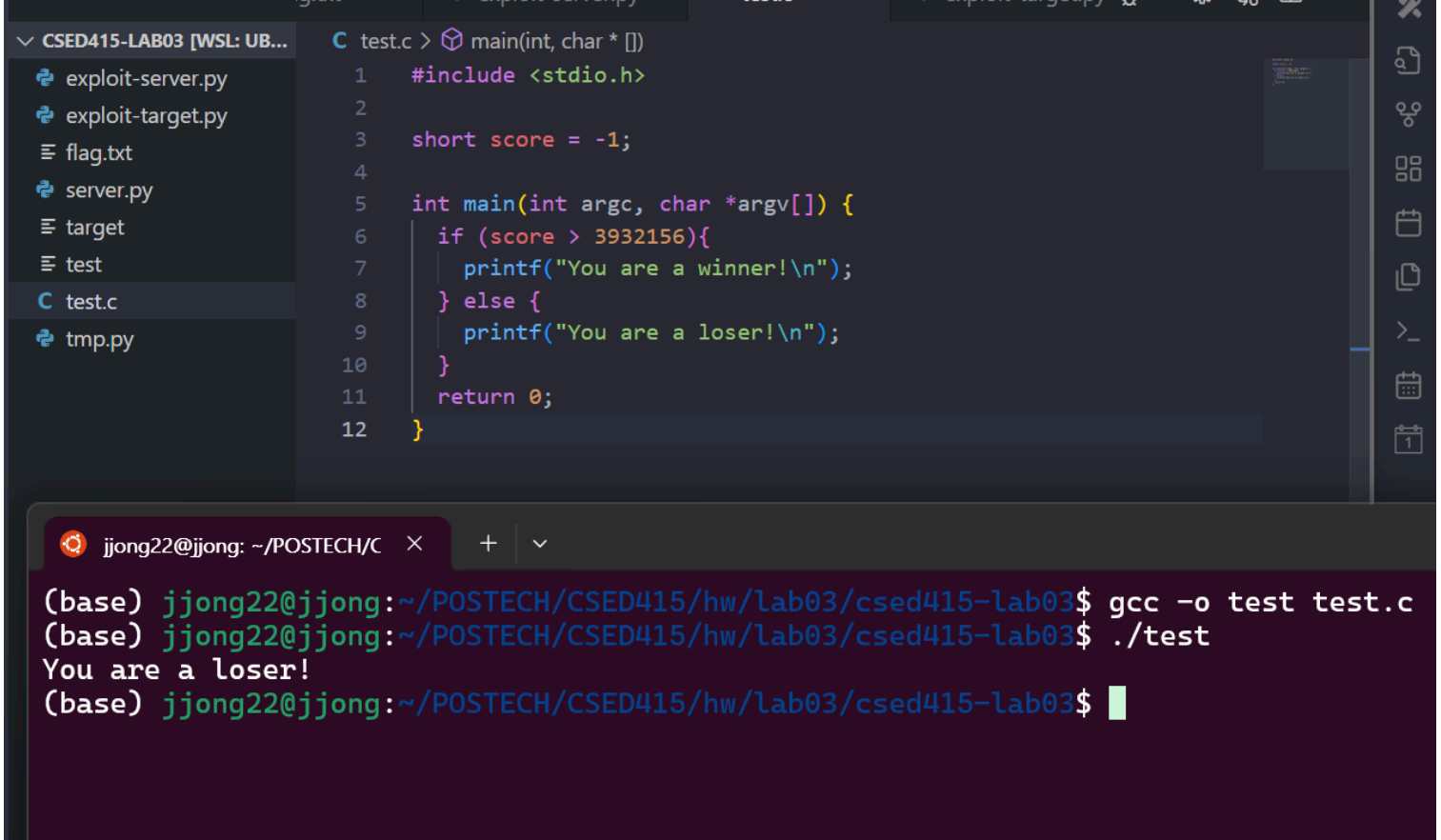
게임에서 틀린 입력을 101번 시도해 `score`를 -1로 만들고, 맞는 입력을 시도하면 게임을 클리어 해 flag를 얻을 수 있다.

```
csed415-lab03@csed415:/tmp/jason9751/lab03$ python3 exploit-target.py  
[+] Starting local process '/home/csed415-lab03/target': pid 311  
[+] Receiving all data: Done (1.02KB)  
[*] Process '/home/csed415-lab03/target' stopped with exit code 0 (pid 311)
```

```
944583A6CFFB89C892AEABE82B57E27862190F85ED1FDE724E96FBB6346F760F  
E61E71742657D3CBC867DA312789589CC7B19B1FC51A92EDAFE7B5F2A61BF702  
AFA9FDC4C6D3BC94E8FD8F905E2DF4B3DC0DD7DA942CF2DBFD9E155BD61FCF87  
F5268221B1EC09B727C0D6EE308D1CADADE4D8F1AB5B52429801684734EF68EF  
F55BCF4D61BA21D71B91FD9B0F5D77A09F0AEDFD24DFD61945376185D2B890F6  
4418B47491A3C23342F0B3004EEC45E2A481123295CB1A485D745E692EE7E235  
54CA347A075A4F4193072084EE6C35F9B3747E0586AEFEFE315D114B065AA101  
0DD15F9E8B6D9C216678454EB3EFEB6A047D66963F03665E4E0A5CF3982F58B1  
9454333EBF1CB2FFC58EF09446FAB7376C023FB4C59EF05A7DD4453C88FA9300  
0707053344468BBBA274E74357A478682BC6C96B5FFF70A746A07D4CE0D027A3  
8F2BCF2E5530580D73FB3DA2DA2619C9D700853F176D8F6BACB411BC36871D7B  
123459DA0F787987C408A2CB39073F28573290CE67913AA984ABA817312F7715  
77C3F1A7A258903CA51F873938116D8EFE0A88F43FEC2F96BF96B4DEBCC4CA26  
4B16E12D997361AEE2F262057EC468415475AEF08A694FBD441EA425B1A967F9  
99CDEA85F0EF6A9CD64D41723A45F4B321A693BCD5BFCCFA34D44AC1EF7410E0  
DA4579E9180BC1773413B137E96635B6EEDB75DEDA8B2B22B74C2B985DBAFF60
```

4. patch the flaw

1. score를 감점하는 로직을 없애면, score가 음수가 될 수 있는 상황이 발생하지 않으므로, 취약점을 패치할 수 있다.
2. 3932156U와 비교하면 unsigned로 취급하여 문제가 발생하므로, 3932156와 비교하면, score가 그대로 signed로 남아있기 때문에, underflow가 발생하지 않게 된다.



The image shows a VS Code editor window with a file explorer on the left and a code editor in the center. The file explorer lists files: exploit-server.py, exploit-target.py, flag.txt, server.py, target, test, test.c, and tmp.py. The code editor shows the content of test.c, which is a C program with a main function that checks a score against 3932156. Below the editor is a terminal window with the following commands and output:

```
jjong22@jjong: ~/POSTECH/C × + v
(base) jjong22@jjong:~/POSTECH/CSED415/hw/lab03/csed415-lab03$ gcc -o test test.c
(base) jjong22@jjong:~/POSTECH/CSED415/hw/lab03/csed415-lab03$ ./test
You are a loser!
(base) jjong22@jjong:~/POSTECH/CSED415/hw/lab03/csed415-lab03$
```

3932156 로 교체한 결과, underflow가 일어나지 않음을 확인할 수 있었다.