

# Dreamhack legacyopt Write-up

이름

legacyopt

output.txt

문제 파일을 열어보면 이렇게 legacyopt라는 파일과 output.txt라는 텍스트 파일이 있다.

```
220c6a33204455fb390074013c4156d704316528205156d70b217c14255b6ce10837651234464e
```

Output.txt 파일을 열어보면 이렇게 되어 있다. 무언가의 암호문인 것 같았다.

legacyopt																	Decoded text
Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	7F	45	4C	46	02	01	01	00	00	00	00	00	00	00	00	00	ELF.....
00000010	03	00	3E	00	01	00	00	00	20	11	00	00	00	00	00	00	..>.....
00000020	40	00	00	00	00	00	00	00	48	31	00	00	00	00	00	00	@.....H1.....
00000030	00	00	00	00	40	00	38	00	0D	00	40	00	1D	00	1C	00	....@.8...@.....
00000040	06	00	00	00	04	00	00	00	40	00	00	00	00	00	00	00	.....@.....
00000050	40	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	@.....@.....
00000060	D8	02	00	00	00	00	00	00	D8	02	00	00	00	00	00	00	@.....@.....
00000070	08	00	00	00	00	00	00	00	03	00	00	00	04	00	00	00	.....

Legacyopt 파일의 확장자를 알아보기 위해 hexs 에디터로 파일을 열어보았다. elf파일임을 알 수 있다.

legacyopt.elf

output.txt

Legacyopt 파일의 확장자를 elf로 수정하고 파일을 IDA에서 열어주었다.

```

1 __int64 __fastcall main(int a1, char **a2, char **a3)
2 {
3     unsigned int v3; // eax
4     int i; // [rsp+4h] [rbp-8Ch]
5     char *ptr; // [rsp+8h] [rbp-88h]
6     char s[104]; // [rsp+10h] [rbp-80h] BYREF
7     unsigned __int64 v8; // [rsp+78h] [rbp-18h]
8
9     v8 = __readfsqword(0x28u);
10    ptr = (char *)malloc(0x64uLL);
11    fgets(s, 100, stdin);
12    s[strcspn(s, "\n")] = 0;
13    v3 = strlen(s);
14    sub_1209(ptr, s, v3);
15    for ( i = 0; i < strlen(s); ++i )
16        printf("%02hhx", ptr[i]);
17    free(ptr);
18    return 0LL;
19 }

```

일단 main함수로 가서 디컴파일된 결과를 확인해 보았다.

분석 결과 이 코드가 수행하는 역할은 순서대로 다음과 같다.

## 1. 메모리 할당 및 입력 처리

- **ptr = (char \*)malloc(0x64uLL);** : 100바이트 크기의 메모리를 동적 할당한다. Ptr을 통해 이후 변환된 문자열을 저장.
- **fgets(s, 100, stdin);** : 표준 입력으로부터 최대 100글자의 문자열을 읽어와 s에 저장.
- **s[strcspn(s, "\n")] = 0;** : 입력된 문자열에 개행 문자가 있을 경우 이를 제거하여 문자열의 끝을 표시.

## 2. 변환 함수 sub\_1209 호출

- **v3 = strlen(s);** : 입력받은 문자열의 길이를 계산.
- **sub\_1209(ptr, s, v3);** : 변환 함수 sub\_1209를 호출하여 s 문자열을 ptr에 변환 후 저장.

## 3. 16진수 출력

- **for (i = 0; i < strlen(s); ++i) printf("%02hhx", ptr[i]);** : 변환된 ptr의 내용을 한 바이트씩 읽어와 16진수 형식(%02hhx)으로 출력.

## 4. 메모리 해제

- **free(ptr);** : 동적 할당한 메모리를 해제.

요약하면, 이 코드는 문자열을 입력받아 sub\_1209 함수로 변환한 뒤, 변환된 내용을 16진수 형식으로 출력한다.

암호화 방식을 알아내기 위해 sub\_1209 함수로 이동해 보았다.

```
v3 = a3 + 7;
v4 = a3 + 14;
if ( v3 < 0 )
    v3 = v4;
v32 = v3 >> 3;
result = (unsigned int)(a3 % 8);
switch ( (int)result )
{
    case 0:
        goto LABEL_4;
    case 1:
        goto LABEL_11;
    case 2:
        goto LABEL_10;
    case 3:
        goto LABEL_9;
    case 4:
        goto LABEL_8;
    case 5:
        goto LABEL_7;
    case 6:
        goto LABEL_6;

    case 7:
        while ( 1 )
        {
            v9 = a2++;
            v10 = *v9;
            v11 = a1++;
            *v11 = v10 ^ 0x66;
        }
LABEL_6:
        v12 = a2++;
        v13 = *v12;
        v14 = a1++;
        *v14 = v13 ^ 0x44;
LABEL_7:
        v15 = a2++;
        v16 = *v15;
        v17 = a1++;
        *v17 = v16 ^ 0x11;
LABEL_8:
        v18 = a2++;
        v19 = *v18;
        v20 = a1++;
        *v20 = v19 ^ 0x77;
```

```

LABEL_9:
    v21 = a2++;
    v22 = *v21;
    v23 = a1++;
    *v23 = v22 ^ 0x55;
LABEL_10:
    v24 = a2++;
    v25 = *v24;
    v26 = a1++;
    *v26 = v25 ^ 0x22;
LABEL_11:
    v27 = a2++;
    v28 = *v27;
    result = (unsigned __int64)a1++;
    *(_BYTE *)result = v28 ^ 0x33;
    if ( --v32 <= 0 )
        break;
LABEL_4:
    v6 = a2++;
    v7 = *v6;
    v8 = a1++;
    *v8 = v7 ^ 0x88;
}
break;
default:
    return result;
}
return result;
}

```

해당 함수의 디컴파일 결과이다. (변수 선언 부분은 생략).

분석 결과 이 코드가 수행하는 역할은 순서대로 다음과 같다.

## 1. 처리 반복 횟수 계산

- v3와 v4는 입력 문자열 길이 a3에 각각 7과 14를 더한 값이다.
- 이후 v32에 v3를 8로 나눈 몫을 저장한다. 이 값은 각 문자를 변환할 때 반복할 횟수를 의미한다.

## 2. 시작 위치 결정

- **result = (unsigned int)(a3 % 8);** : a3의 8로 나눈 나머지를 result에 저장하여 특정 케이스(switch)로 분기한다.
- 나머지 값에 따라 case 0부터 case 7까지의 레이블로 이동한다.

### 3. 문자 변환 반복문

- 각 case에 해당하는 레이블에서 **특정 문자를 특정 정수와 XOR 연산**하여 a1에 저장한다. XOR 키 값은 **0x88, 0x66, 0x44, 0x11, 0x77, 0x55, 0x22, 0x33** 이다. 예를 들어, LABEL\_4는 0x88과 XOR 연산을 수행하고, LABEL\_5는 0x66을 사용한다.
- 반복문은 --v32가 0이 될 때까지 실행된다. 이때 입력 문자열을 8바이트 단위로 잘라 각 바이트를 다른 XOR 키로 변환한다.

### 4. 출력

- 최종적으로 변환된 결과는 a1에 저장되어 호출 함수로 보내진다.

요약하자면, sub\_1209함수에서는 a2의 각 바이트를 반복적으로 XOR 연산하여 암호화된 형태로 a1에 저장한다. 이때 XOR 키 값이 반복적으로 순환되어 사용된다.

## Description

Good ol' days of optimization...

문제에 있는 'optimization'이 이러한 암호화 과정과 연관이 있을 거라는 생각이 들어 좀 더 찾아 보았다.

그 결과 이 코드가 **optimization(최적화)중 loop unrolling**이라는 방법을 사용했다는 것을 알게 되었다.

루프 언롤링은 루프의 반복을 줄이기 위해 반복문 내부의 코드를 여러 번 복제하여 수행하는 방법이다. 이렇게 하면 반복 조건을 확인하는 횟수를 줄임으로써 성능 최적화가 가능하다. 그러나 코드 크기가 커진다는 단점이 있다.

루프 언롤링과 관련된 개념으로는 **Duff's device(더프의 장치)**가 있다. 이는 Tom Duff라는 사람이 1983년에 생각해낸 방법인데, 루프 언롤링을 구현하기 위해 반복문의 시작점을 조정하여 조건문을 줄이는 기법이다. 이 기법은 기본적으로 c언어에서 do-while 반복문과 switch문의 특징을 이용한다.

```

send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
               } while (--n > 0);
    }
}

```

위키피디아에서 가져온 실제 duff's device 예시이다. 위 코드는 기본적으로 to에 from의 데이터를 count만큼 복사한다.

- 이때 루프를 8개 항목으로 묶어 반복시키고 있는데, 이렇게 하면 8번 반복되어야 하는 루프는  $8 = 8 * 1$  이므로 1번 반복하게 되고, 80번 반복되어야 하는 루프는  $80 = 8 * 10$ 이므로 10번 반복되게 될 것이다.
- 마찬가지로 10 번 반복되어야 하는 루프는  $10 = 8 * 1 + 2$  이므로 1번 반복되고, 나머지 2 번은 switch에 의해 case 2로 점프하여 2 번 실행될 것이다.
- \*from++과 \*to의 포인터 연산만으로 from 배열의 각 값을 to로 복사하고 있다. 루프 인덱스 n의 감소 빈도를 1/8로 줄여 약간의 최적화를 이끌어낸다.

```

void normal_device(short *to, short *from, int count)
{
    int i = 0;
    for (i = 0; i < count; i++) {
        *to = from[i];
    }
}

```

기능 자체만 본다면 이런 간단한 코드로도 대체해서 쓸 수 있다.

과거 컴퓨터의 경우 조건 비교 연산이 많으면 성능 저하가 생겼기 때문에 이러한 방법으로 성능 최적화를 위해 수동으로 루프 언롤링을 동작 시켰다고 한다. 요즘의 컴파일러는 이 정도의 최적화는 알아서 해주기 때문에 오늘날 실제로 사용할 수 있을 만한 방법은 아니라고 한다.

문제로 돌아가서, sub\_1209 함수에도 Duff's device와 유사한 특징이 보인다.

이 함수는 `switch (a3 % 8)` 문을 통해 입력 문자열의 길이를 8로 나눈 나머지 값을 기준으로 분기하여 반복문의 시작 위치를 조정한다. case문으로 분기된 각 레이블에서 반복적으로 XOR 연산을 수행한다.

좀 더 자세히 살펴보자면 함수에 인자로 넘겨준 데이터의 길이인 `a3`를 이용해 반복문을 수행하는데 이때 가장 처음 반복문에 접근할 때만 `a3 % 8`의 결과를 이용해 반복문 내의 특정 위치로 이동한다. 그리고 그 다음부터는  $(a3 + 7) / 8 - 1$ 회 만큼 반복문 전체를 수행한다. 반복문 내에는 8개의 레이블이 존재하며 각 레이블에 해당하는 부분은 각 레이블에 할당된 특정 바이트와 `a1`의 한 바이트를 XOR 연산한 후에 `a2`에 저장하는 동작을 한다.

이후 복호화 코드를 만들었다.

위의 sub\_1209 함수 분석 과정에서 설명했듯이 0x88, 0x66, 0x44, 0x11, 0x77, 0x55, 0x22, 0x33 순으로 바이트에 대해 XOR연산이 일어난다는 사실을 알 수 있기 때문에 복호화 또한 해당 값을 이용하면 된다.

그런데 이때 해당 함수가 Duff's device의 방법을 차용한 관계로 가장 처음 반복문을 시작할 때는 반복문의 중간에서 시작하기 때문에 처음 `a3 % 8`회의 XOR 연산에 주의를 기울여야 한다.

```
def decrypt(hex_ciphertext):
    ks = [0x88, 0x66, 0x44, 0x11, 0x77, 0x55, 0x22, 0x33]

    vs = [int(hex_ciphertext[i:i+2], 16) for i in range(0, len(hex_ciphertext), 2)]

    decrypted = ''

    r = len(vs) % 8

    for i, v in enumerate(vs):
        decrypted += chr(v ^ ks[(i - r) % 8])

    return decrypted

def main():
    hex_ciphertext = "220c6a33204455fb390074013c4156d704316528205156d70b217c14255b6c"

    decrypted_text = decrypt(hex_ciphertext)
    print("복호화된 암호문:", decrypted_text)

if __name__ == "__main__":
    main()
```

파이썬으로 작성한 복호화 코드이다.

이 코드에서는 먼저 hex\_ciphertext에 output.txt에 적혀져 있던 암호문을 저장한다,

이후 `vs = [int(hex_ciphertext[i:i+2], 16) for i in range(0, len(hex_ciphertext), 2)]`에서 16진수 문자열을 2자리씩 잘라서 각 16진수를 정수로 변환한 후 리스트 vs에 저장한다.

`r = len(vs) % 8`는 암호문의 길이에 대한 남은 값을 계산한다. 이는 XOR 키를 순차적으로 적용하기 위해 필요하다.

`decrypted += chr(v ^ ks[(i - r) % 8])`는 각 바이트에 대해 키 배열을 순환하면서 XOR 연산을 수행하고, 그 결과를 문자로 변환하여 decrypted에 추가한다.

마지막으로 복호화된 결과를 decrypted\_text에 저장하고 출력한다.

```
thon.exe c:/Users/gram/Desktop/legacyoptdecode.py
복호화된 암호문: DH{Duffs_Device_but_use_memcpy_instead}
```

이렇게 플래그를 얻어낼 수 있다.