2022 리버싱 2조 2주차 보고서

김준서

# 1주차

GoN qual에 나왔던 리버싱 문제 일부 업솔빙

## Run

##### 풀이

main이 플래그를 인코딩하는 데 사용된 바이너리다.

비트 단위로 입력을 받고 1을 만나기까지 연속한 0의 개수에 따라 인코딩을 한다.

연속한 0의 개수를 C라고 했을 때, '1' \* (C를 나타내는데 필요한 비트수) + '0' + (C를 비트로 표현) 가 출력된다.

0을 기준으로 정보가 분리되므로 flag.enc 파일에서 find(0)을 반복하면서 디코딩해주면 플래그가 적힌 png 파일이 나온다.

##### 헤맨 부분

wb 모드로 flag.png 파일 출력을 하기 위해서 int를 byte로 바꿔야 한다.

이 부분에서 `chr(x).encode()` 같이 int -> string -> byte로 변환했었는데, `chr(128).encode() == b'\xc2\x80'` 같은 경우가 발생해서 의도한대로 코드가 동작하지 않았다.

`int.to\_bytes(length, byteorder)` 를 사용해서 int를 바로 byte로 바꿔주니 제대로 된 파일이 나왔다.

## pyc

##### 풀이

prob.pyc 를 디스어셈블해서 분석한다.

`throw` 함수가 `chk.\_\_code\_\_` 에 m을 xor한다. `chk.\_\_code\_\_`에 m을 xor한 후 chk를 분석한다.

chk를 역연산해서 플래그를 구한다.

##### 헤맨 부분

online pyc decompiler에서 prob.pyc가 디컴파일이 안되길래 pyc에 대해 찾아봤다.

pyc 파일의 초기 4바이트에는 매직 넘버가 존재한다. 버전에 따라서 파이썬 코드를 바이트 코드(pyc 파일)로 바꾸는 세부 방식이 다르고, 버전마다 다른 매직 넘버를 가지고 있다. 파이썬 버전에 따른 매직 넘버를 모아놓은 페이지를 찾아서 prob.pyc에 있는 매직 넘버를 검색해봤는데 해당하는 파이썬 버전이 없었다. 그래서 나는 난독화가 사용된 것으로 판단했고, 매직 넘버를 브루트포스해야 되나? 까지 생각하고 풀이를 물어봤다.

prob.pyc에 있는 매직 넘버는 파이썬 3.10 버전에 해당하는 것이었고, 내가 찾았던 페이지에는 최신 버전이 반영이 안 돼있던 거였다. 3.10 버전 디컴파일러는 아직 완전한 게 없다는 듯 해서, [pycdas](https://github.com/zrax/pycdc)를 사용해서 prob.pyc를 디스어셈블하고 직접 해석했다.

유철민

# 1주차

dreamhack에서 리버싱에 대한 이론을 간략히 공부했다.

총 12강이며 모든 강의를 수강했으나 이번 주는 3강의 이론에 대한 요약만 보고한다. 다음주까지 9강의 내용도 요약하여 정리한다. 리눅스 기반 환경에서 컴파일한 바이너리의 실행 흐름을 파악하기 위한 기초 세팅을 만들 준비를 했다.

# stage1 intro

software reverse engineering :

각종 프로그램을 분석해서 기능과 설계를 파악.

cf) 키제네레이터 프로그램, 시리얼 넘버 생성기, 크랙 등

# stage2 binary & analysis

\* binary

이해 :

바이너리의 정의를 알고, 컴파일 과정을 이해

전처리에 대한 이해

컴파일과 최적화 과정

어셈블과 object 파일(리눅스의 ELF/ 윈도우의 PE)

링크에 대한 이해

디스어셈블, 디컴파일의 목적과 결과

프로그램: 컴퓨터가 실행해야 할 명령어의 집합, 바이너리라고도 불림

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

용어 정리 :

전처리: 소스 코드가 컴파일에 필요한 형식으로 가공되는 과정

컴파일: 소스 코드를 어셈블리어로 번역하는 과정

어셈블: 어셈블리 코드를 기계어로 번역하고, 실행 가능한 형식의 변형하는 과정

링크: 여러 개의 목적 파일을 하나로 묶고, 필요한 라이브러리와 연결해주는 과정

디스어셈블: 바이너리를 어셈블리어로 번역하는 과정

디컴파일: 바이너리를 고급 언어로 번역하는 과정

\* analysis

이해 :

소프트웨어를 분석시

1) 큰 구조를 먼저 관찰합니다. 이것만으로 소프트웨어의 실제 동작을 알기 어려울 때가 많다.

2) 프로그램을 실행해보며 동작을 관찰.

3) 기타 수단을 동원하여 특정 부분을 세밀하게 관찰

정적 분석(Static Analysis) / 동적 분석(Dynamic Analysis)

동적인 방법만을 고수하면 큰 구조를 놓칠 수 있고, 반대로 정적인 방법만을 고수하면 실행을 통해 직관적으로 알 수 있는 부분을 분석하느라 시간을 허비할 수 있으므로 상황에 따라 적절한 방법을 선택.

용어 정리 :

정적 분석 : 외적인 관찰만을 통해 정보를 알아내는 것

동적 분석 : 실행을 통해 동작을 분석하는 것

육근우

리버스 엔지니어링 바이블(김병탁)을 읽고 예제 코드를 따라가며 공부

C 함수 호출 규약

\_\_cdecl

함수를 call한 후 add esp, n 식으로 스택을 정리한다. 함수 밖에서. 이를 통해 파라미터의 개수 또한 파악 가능하다.

\_\_stdcall

함수 내에서 retn n 의 형식으로 스택을 정리한다. win32 api는 stdcall을 사용한다.

\_\_fastcall

fastcall은 파라미터가 2개이하일 경우 인자를 ecx와 edx 레지로 이용. 메모리보다 빠름. 함수 호출 전 edx, ecx에 넣는 모습이 보인다면 fastcall

\_\_thiscall

c++ 클래스에서 주로 사용. 현재 객체의 포인터를 ecx에 전달. 이것이 this. 그래서 멤버 변수는 ecx포인터에 오프셋 및 번지를 더하는 식으로 사용한다. 나머지 방법은 stdcall과 동일.

구조체와 API call

변수가 레지스터로 충당이 안된다면 스택으로 공간을 확보한다.

ida를 통해 예제 코드를 짜고 어셈블리 코드를 분석해보았다.

구조체의 크기만큼 스택을 sub하여 공간을 확보한다.

교재에서는 구조체의 크기만큼만 스택을 확보했으나 IDA를 통해 코드를 분석해보니 교재와는 다른 크기만큼 스택을 확보했다.

코드를 분석하며 차이점을 살펴보았다.

교재에서는 push를 통해 파라미터를 집어넣었지만 IDA에선 mov를 통해 esp-n만큼의 위치에 파라미터를 집어넣었다.

이를 통해 IDA에서 보여준 어셈블리 코드는 구조체의 파라미터가 push되는 것까지 미리 계산하여 스택을 확보했음을 알 수 있었다.

그 후 호출되는 코드에서 구조체의 포인터를 가져올 때는 구조체의 크기를 계산한 만큼의 위치를 스택에서 불러오고, 0x44 바이트와 0x 10바이트의 크기를 가진 구조체라는 것을 인지하며 어셈블리 코드를 분석할 시 훨씬 분석이 수월하였다. 또한 함수는 원형이 존재하며 보이는 파라미터가 아니라 추가적인 파라미터를 push할 시 다른 함수를 통해 계산을 하는 것을 생각해야한다.