

# Reverse Engineering

## Table of Content

<b>Module 1 — RCE 란?</b> .....	1
1-1. RCE 란? .....	2
1-2. RCE 관련 법규 .....	4
1-3. RCE 응용분야 .....	6
<b>Module 2 — RCE 기초</b> .....	9
2-1. CPU 동작방식 .....	10
2-2. CPU 레지스터 .....	12
2-3. Assembly .....	17
2-4. STACK 구조 .....	21
2-5. 함수 호출 규약(Calling Convention).....	23
2-6. SEH(Structured Exception Handling).....	26
2-7. C 코드 컴파일 후 어셈코드 변환 모습 .....	34
2-8. 특정 루틴에서 사용되는 API 함수 .....	40
<b>Module 3 — PE 파일 구조</b> .....	43
3-1. PE 파일 포맷 .....	44
3-2. DOS 헤더 및 DOS Stub Code.....	49
3-3. PE File Header .....	51
3-4. Optional Header .....	54
3-5. Section Table.....	59
3-6. Import Table.....	62
3-7. Export Table.....	69
<b>Module 4 — 분석 기초</b> .....	75
4-1. 분석 도구 설정 .....	76
4-2. CrackMe 분석 실습 .....	84
4-3. KeygenMe 분석 실습 .....	94
<b>Module 5 — MUP(Manual UnPack)</b> .....	101
5-1. Packing / Unpacking .....	102
5-2. Packing 종류 .....	103
5-3. MUP 실습 .....	105
5-4. MUP를 위한 Ollydbg script 작성 .....	114
<b>Module 6 — Anti-Reverse 기법</b> .....	121
6-1. 디버거 탐지 기법 .....	122
6-2. BreakPoint 탐지 기법 .....	133
6-3. TLS Callback .....	145
6-4. Process Attach 기법 .....	150
<b>Module 7 — 분석 실전</b> .....	155
7-1. 온라인 해킹대회 문제 분석 .....	156
7-2. 악성코드 분석 - shadowbot .....	173



## Module 1 — RCE 란?

### Objectives

- 리버스 엔지니어링의 기본 개념
- 리버스 엔지니어링의 법적 문제
- 리버스 엔지니어링 응용분야

### 1-1. RCE 란?

#### RCE란?



#### Reverse Code Engineering

- 어떠한 장치나 객체 혹은 시스템들의 구조나 기능, 그리고 어떻게 작동되는지 등의 기술적인 원리를 분석 함으로써 그것들이 작동되는 과정을 알아내는 것
- 작동하는 모든 순간순간의 세세한 것들까지 분석하는 것을 포함
  - 기계적인 장치
  - 전자적인 부품
  - 소프트웨어적인 프로그램

### Student Notes

리버스 엔지니어링은 여러 분야에서 사용되는 용어로써 인공물로부터 설계 사상이나 지식을 추출하는 행위를 말한다. 본 교재에서는 소프트웨어에 대한 리버스 엔지니어링을 이야기할 것이다.

통상적으로 컴파일된 바이너리(EXE, DLL, SYS 등)를 디스어셈블러라는 도구를 이용하여 어셈블리 코드를 출력한 후 그것을 C 언어 소스형태로 다시 읽겨 적고 적당한 수정을 통해 리버스하고 있는 파일과 동일한 동작을 하는 프로그램을 만드는 것이 있다.

모든 어셈블리 코드를 소스 형태로 읽기지 않고 그냥 동작 방식만을 알아낸다거나 일정 부분만 수정하는 것들도 리버스 엔지니어링이라고 할 수 있는데 예를 들면 바이러스를 분석하는 일은 모든 코드를 알아낼 필요가 없기 때문에 동작 방식만 알아내면 된다. 그리고 크랙처럼 일정 부분만 수정하여 사용제한

을 푸는 것 등도 이에 해당된다.

실행파일을 디스어셈블 하지 않고도 그 실행파일이 만들어내는 데이터 파일이나 패킷 등을 분석하여 똑같이 재구현하는 것도 리버스 엔지니어링이다. 예를 들면 오래 전 PC 게임에서 많이 하던 일인데, HEX 에디터 등으로 세이브 파일을 분석하여 에디트를 만들거나 게임자체를 조작하는 것이 있고, 당나귀와 호환되는 이를 같은 프로그램은 당나귀 프로토콜의 패킷을 분석하여 동일한 동작을 하도록 만들어낸 것이다.

리버스 엔지니어링에서 가장 많이 사용되는 방식은 첫 번째로 이야기했던 바이너리를 디스어셈블 하여 코드를 얻어내는 것이다. 이것을 하기 위해서는 먼저 인텔 어셈블리를 배워야 하고, 물론 C 언어도 알아야 된다.

그런데 여기서 컴파일된 바이너리가 VC 나 gcc 등으로 컴파일 한 것이 대부분이지만 비주얼 베이직으로 컴파일 한 것도 있고 멜파이(파스칼)로 컴파일 한 것도 있을 것이다. 이 바이너리들은 모두 CPU에서 직접 실행되는 것들이기 때문에 디스어셈블 해보면 모두 똑같은 방식으로 되어 있다. 그래서 VC, VB, 멜파이(파스칼)등으로 컴파일 된 것도 디스어셈블 한 뒤 C 소스 코드로 옮길 수 있다. 물론 리버스 하는 사람이 VB 나 파스칼로 옮겨 적을 수도 있을 것이다. 대부분 리버스해서 얻어내는 것들은 프로그램의 로직(알고리즘)이기 때문에 어느 언어로 표현하든 결과는 똑같기 때문이다.

## 1-2. RCE 관련 법규

### RCE 관련 법규



- 상호운용성
- 경쟁(Competition)
- 저작권법(Copyrightlaw)
- The Digital Millennium Copyright Act (DMCA)
  - RIAA의 Felten 교수 연구팀에 대한 위협
  - 드미트리 스클리아로프의 ebook 사건
  - SunnComm사의 CD 복제 방지 시스템

## Student Notes

### 컴퓨터프로그램보호법

제 12 조의 2 (프로그램코드역분석) ①정당한 권원에 의하여 프로그램을 사용하는 자 또는 그의 허락을 받은 자가 호환에 필요한 정보를 쉽게 얻을 수 없고 그 획득이 불가피한 경우 당해 프로그램의 호환에 필요한 부분에 한하여 프로그램저작권자의 허락을 받지 아니하고 프로그램코드역분석을 할 수 있다.  
②제 1 항의 규정에 의한 프로그램코드역분석을 통하여 얻은 정보는 다음 각호의 1에 해당하는 경우에 이를 사용할 수 없다.

1. 호환 목적 외의 다른 목적을 위하여 이용하거나 제 3자에게 제공하는 경우
2. 프로그램코드역분석의 대상이 되는 프로그램과 표현이 실질적으로 유사한 프로그램을 개발·제작·판매하거나 기타의 프로그램저작권을 침해하는 행위에 이용하는 경우[본조신설 2001.1.16][시행일 2001.7.17]]

위 컴퓨터프로그램보호법은 2001년 1월 16일 개정되어 7월 17일부터 시행되는데, 이번 개정에서 가장 중요하게 다루어진 것이 컴퓨터프로그램 역분석의 허용한계를 규정한 것이다. 컴퓨터프로그램의 역분석은 이미 2000년 개정된 컴퓨터프로그램보호법에서도 규정하고 있었는데, 2001년 개정법에서는 저작권제한의 일종으로 리버스엔지니어링을 추가하고, 프로그램코드의 역분석을 규정하고 있다.

상호운용성을 위한 리버스 엔지니어링은 특정 범위 안에서는 대부분 합법적이긴 하다. 하지만 경쟁사의 코드를 사용하는 것은 분명한 저작권 위반이 될 수 있다.

DMCA(The Digital Millennium Copyright Act)은 인터넷에서의 저작권 침해행위를 단속하기 위한 미국법이다. DMCA의 보호를 받는 대상에는 HTML 소스 코드, 본문내용, 사진, 이미지, 해킹된 코드, 불법복제 소프트웨어, 음악, 영화, 웹 페이지 및 이메일 등이 있다. DMCA에 의해서 제재를 받은 대표적인 사례들 몇 가지를 예로 들어보겠다.

### DMCA 가 Sony-BMG 의 "루트킷" 취약성의 공개를 늦추다

Sony-BMG의 CD 복제 방지 기술인 "루트킷(root kit)"이 이용자의 컴퓨터에 보안 문제를 야기하는 것을 발견한 프린스턴의 대학원생 알렉스 해더맨은 저작권 침해 여부를 판단하기 위해서 수주간 발표를 연기할 수 밖에 없었고 때문에 많은 이용자가 위험에 노출된 상태로 오랫동안 있었음

### 펠튼 교수 연구팀 위협을 받다

Secure Digital Music Initiative(SDMI)라는 다산업 집단이 디지털 음악을 보호하는 특정한 워터마킹 기술을 깨는 경연대회를 공개적으로 열었고, 프린스턴 대학의 에드워드 펠튼과 프린스턴과 라이스 대학 그리고 제록스 사의 연구자들이 깼다. 이 결과를 학회에서 발표하려는 것을 SDMI가 저작권법을 근거로 책임을 묻겠다는 협박으로 막았다. 연구자들이 소송을 제기하고 나서야 위협을 철회했다.

### 썬콤사 대학원생 위협하다

SunnComm사가 개발한 CD 복제 방지 시스템을 단지 컴퓨터의 쉬프트 키만 누르고 있으면 무력화 된다는 사실을 발견한 프린스턴 대학원생 알렉스 헐더만이 발표한 보고서에 대해서 SunnComm사가 소송을 제기하겠다는 위협을 함. 후에 SunnComm사는 위협을 포기함.

### 드미트리 스킬리아로프 체포된다

러시아 프로그래머인 드미트리 스킬리아로프는 어도비 사의 이북 포맷의 전자책을 pdf 파일로 바꾸는 소프트웨어를 개발했다는 이유로 수 주간 감옥살이를 하고 다섯달 동안 미국에서 유치장 생활을 했다. 전자책을 pdf로 바꾸는 과정에서 이북 포맷에 포함된 이용 제한 조치가 깨졌다는 이유에서다. 스킬리아로프는 직접적으로 어떤 저작권 침해를 하지도 않았고 저작권 침해를 돋지도 않았지만 미국 라스베가스에서 열린 학회에서 그 소프트웨어를 발표한 것으로 이런 생활을 한 것이다. 결국은 검찰의 기소는 각하되었다.

<출처 : [http://www.eff.org/IP/DMCA/unintended\\_consequences.php](http://www.eff.org/IP/DMCA/unintended_consequences.php)>

### 1-3. RCE 응용분야

## RCE 응용분야



- 악성코드 분석
- DRM 등 보호 알고리즘 분석
- 개발 효율성 증가
- 안전성 테스트
- 응용 프로그램 패치



RCE의 핵심은  
사용자가 원하는대로  
프로그램이 동작하게 만드는 것

### Student Notes

리버스 엔지니어링은 흔히 불법이라는 생각에 응용분야 또한 불법인 것이 아닌가 하는 생각을 할 수 있다. 하지만 실제는 정반대의 상황이 발생할 수 있다. 현재 인터넷을 사용하는 모든 사용자가 가장 신경을 많이 쓰는 부분은 바이러스/웜이나 악성코드일거라 생각한다. 물론 스팸도 만만치 않겠지만 바이러스/웜이나 악성코드(이하 ‘악성코드’라 함)의 경우 직접적으로 내가 사용하고 있는 컴퓨터에 영향을 미치기 때문에 더 많은 신경을 쓰고 백신 프로그램도 설치하고 할 것이다.

이러한 악성코드는 대부분 확장자가 exe인 실행파일의 형태로 배포가 되고 감염자 PC에서 실행 되면서 악성코드 제작자의 의도대로 악의적인 행동을 하게 된다. 그렇다면 이러한 악성코드가 내 PC에서 실행이 되면 어떤 행동들을 하는지 정확하게 파악할 수 있어야지만 악성코드를 찾아내서 내 PC를 안전하게 할 수 있다. 이러한 일련의 행동들을 우리는 리버스 엔지니어링 기법을 통해서 파악할 수 있다. 물론

## Reverse Engineering

악의적인 목적으로 운영체제나 소프트웨어들의 취약점을 찾아서 공격하기 위해 리버스 엔지니어링 기법을 사용할 수도 있겠지만 좋은 의미에서 리버스 엔지니어링을 사용할 수도 있다는 것이다.

대표적인 예로 시그내처 기반으로 동작하는 백신의 경우 새로운 악성코드 발견 시 전문가들에 의해서 악성코드의 일련의 행동들을 알아낸 후 백신 데이터베이스를 업데이트한다. 이 때도 리버스 엔지니어링 기법을 사용하게 된다.

혹은 최근 배포되고 있는 악성코드들이 암호 알고리즘을 사용하여 악성코드 자체를 보호하는 경우가 있는데 이러한 경우에도 리버스 엔지니어링 기법을 사용한다.

DRM의 경우 소프트웨어 복제 방지 기술과 유사한데 차이점이라면 소프트웨어 복제 방지 기술은 소프트웨어 자체를 보호하는 것이고 DRM은 컨텐츠를 보호하는 것이라는 점이다. 예를 들어 온라인에서 음악을 들을 수 있는 사이트가 있고 이 사이트는 클라이언트들이 음악을 듣기 위한 프로그램 설치를 요구할 수 있다. 한 악의적인 사용자가 이 클라이언트 프로그램을 리버스 엔지니어링을 통해서 음원을 다운받을 수 있게 할 수 있다는 것이다.

상용 소프트웨어와 상호호환이 되는 소프트웨어를 개발하는 경우 소프트웨어 라이브러리나 API에 대한 매뉴얼의 내용이 부족한 경우가 많은데 이러한 문제점을 해소하기 위해서 리버스 엔지니어링 기법을 사용하거나 소프트웨어 개발 시 아무것도 없는 상태에서 개발하는 것은 힘들기 때문에 잘 만들어진 소프트웨어에 대해서 리버스 엔지니어링 기법을 사용하여 재사용을 하기도 한다. 물론 공개 소프트웨어의 경우는 소스 코드를 활용할 수 있겠지만 공개되지 않은 소프트웨어의 경우는 리버스 엔지니어링 과정을 거쳐야 할 것이다.

마지막으로 소프트웨어의 안전성과 취약점을 평가하기 위해 리버스 엔지니어링 기법을 사용할 수 있다. 개발된 소프트웨어에 대해 리버스 엔지니어링 기법을 적용하여 소프트웨어에 결함이 있지는 않은지 품질 평가를 할 수 있다.

그리고 응용 프로그램에 대한 패치도 리버스 엔지니어링의 응용분야이다. 취약점이 알려진 프로그램이 있는데 소스코드가 없어서 재컴파일 할 수 없을 경우 바이너리를 패치해야 하고 이 때 리버스 엔지니어링을 하게 된다. 물론 이런 점을 악용하면 크랙을 하는 것이 될 수도 있다.

하지만 리버스 엔지니어링에서 가장 중요한건 개발자의 의도가 아닌 사용자의 의도대로 프로그램이 동작하게 하려고 한다는 것이다.

**Reverse Engineering**

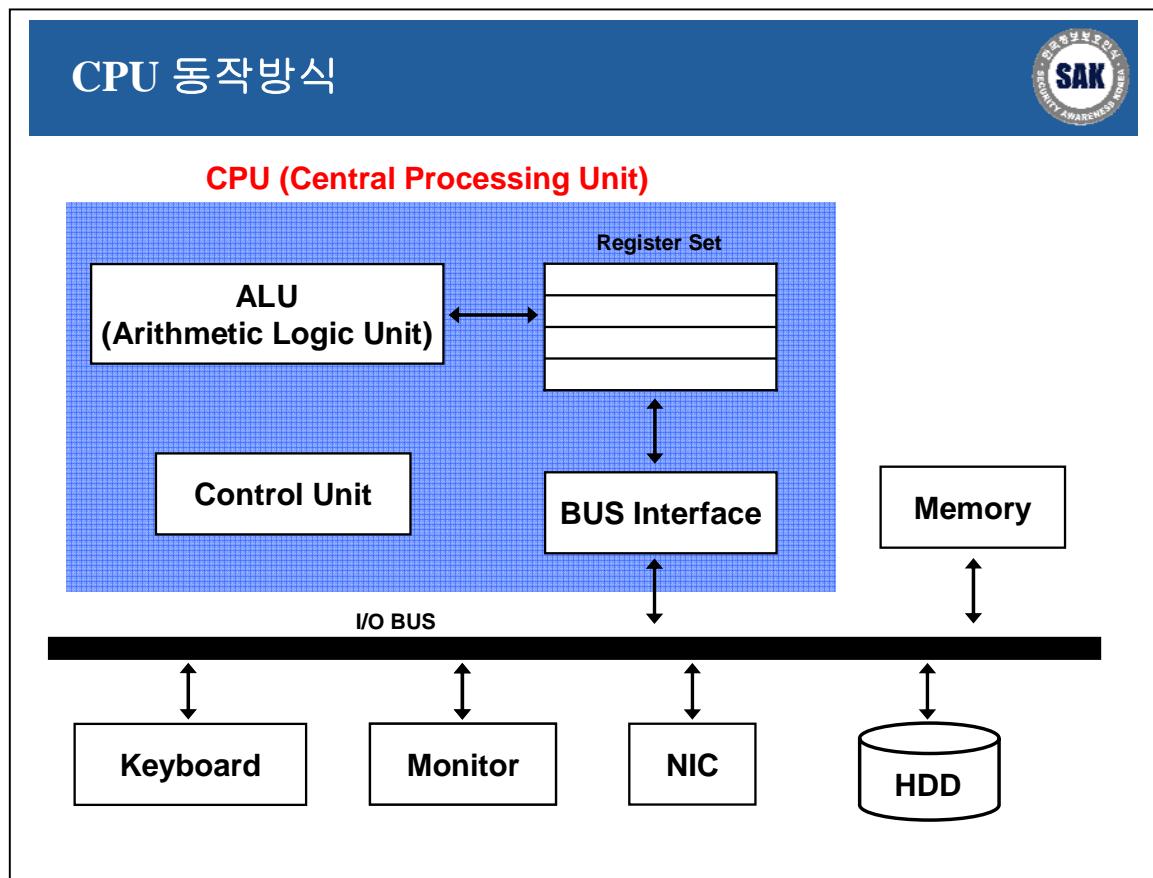


## Module 2 — RCE 기초

### Objectives

- CPU 동작방식
- CPU 레지스터
- 어셈블리 / 디스어셈블리
- STACK 구조
- 함수 호출 규약(Calling Convention)
- SEH(Structured Exception Handling)
- C 코드 컴파일 후 어셈코드 변환 모습
- 특정 루틴에서 사용되는 API 함수

## 2-1. CPU 동작방식



### Student Notes

CPU는 소프트웨어 명령의 실행이 이루어지는 컴퓨터의 부분, 혹은 그 기능을 내장한 칩을 말한다.

CPU는 기계어로 쓰여진 컴퓨터 프로그램의 명령어를 해석하여 실행한다. CPU는 프로그램에 따라 외부에서 정보를 입력하고, 기억하고, 연산하고, 외부로 출력한다. CPU는 컴퓨터 부품과 정보를 교환하면서 컴퓨터 전체의 동작을 제어한다. 기본 구성으로는 레지스터, 산술논리연산장치(ALU: arithmetic logic unit), 제어부(control unit)와 내부 버스 등이 있다.

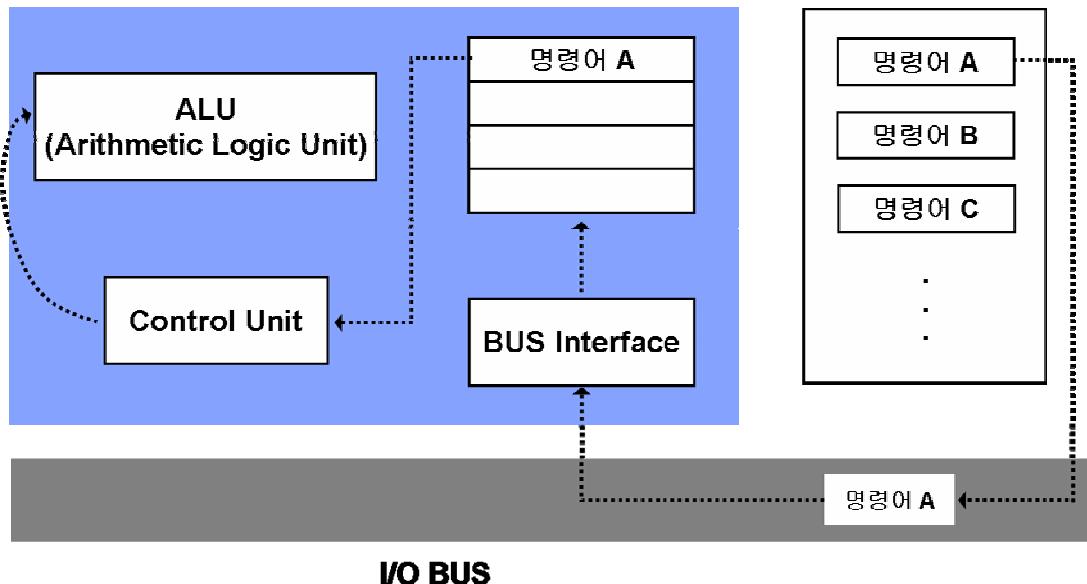
각종 전자 부품과 반도체 칩을 하나의 작은 칩에 내장한 전자 부품을 마이크로프로세서라고 한다. 마이크로프로세서는 전기 밍통에 쓰이는 낮은 성능의 제품부터 컴퓨터에 쓰이는 높은 성능의 제품까지 매우 다양하다. 마이크로프로세서들 가운데 가장 복잡하고 성능이 높은 제품은 컴퓨터의 연산 장치로 쓰인다. 이것을 중앙 처리 장치라고 한다.

- 위키백과사전 -

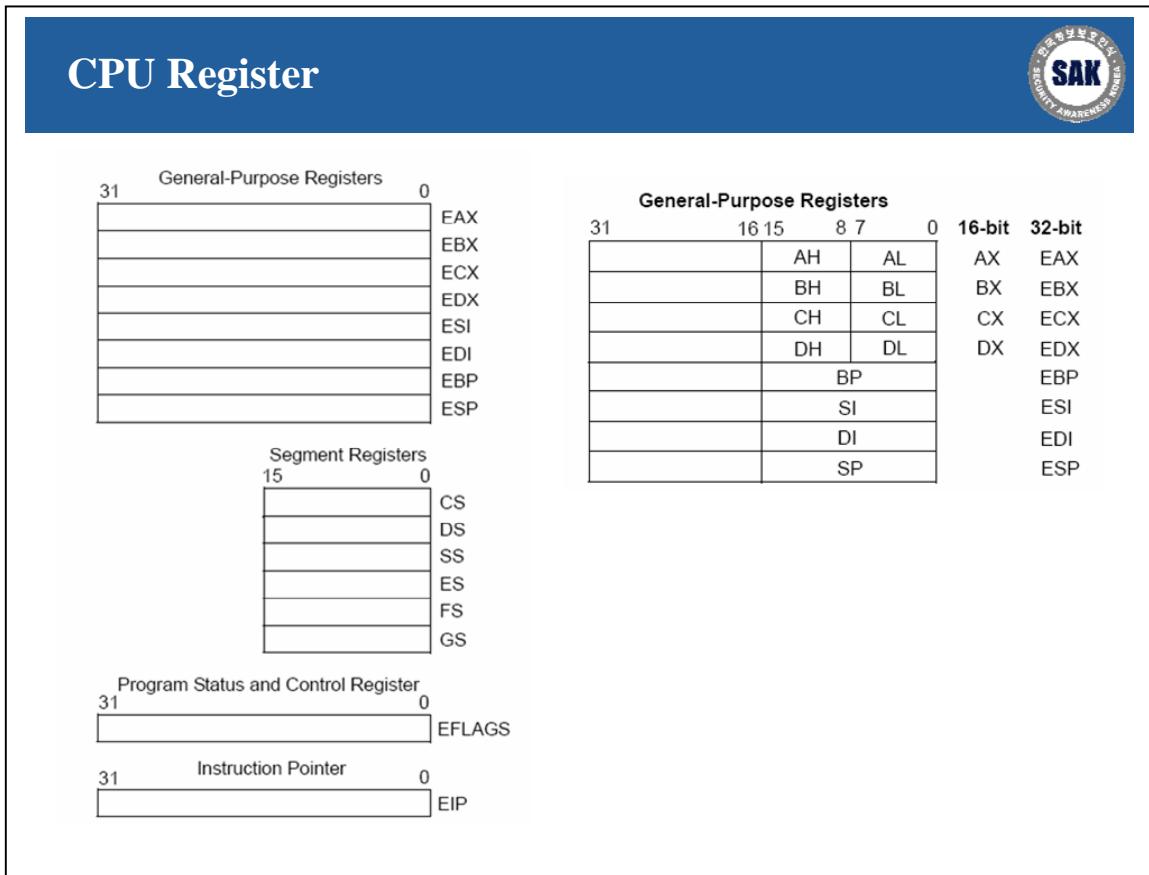
프로그램이 실행되면 실행 프로그램이 메모리에 적재되고 메모리의 내용을 CPU에서 가져와 레지스터에 저장한 후 Control Unit을 통해 ALU에 전달된 후 ALU에서 산술 연산을 하게 된다.

- ALU(산술연산장치) : 두 숫자의 (덧셈, 뺄셈 같은) 산술연산과 (비트적 논리합, 논리곱, 논리합 같은) 논리연산을 계산하는 디지털 회로이다. 산술논리장치는 컴퓨터 중앙처리장치의 기본 설계 블럭이다.
- Register : 컴퓨터의 프로세서 내에서 자료를 보관하는 아주 빠른 기억 장소이다. 일반적으로 현재 계산을 수행중인 값을 저장하는데 사용된다. 대부분의 현대 프로세서는 메인 메모리에서 레지스터로 데이터를 옮겨와 데이터를 처리한 후 그 내용을 다시 레지스터에서 메인 메모리로 저장하는 로드-스토어 설계를 사용하고 있다.

### CPU (Central Processing Unit)



### 2-2. CPU 레지스터



### Student Notes

CPU에서 사용되는 레지스터는 크게 범용 레지스터, 세그먼트 레지스터, EFLAGS 레지스터, CIP 레지스터 네 가지로 구분된다. 기본적으로 32비트(4바이트)의 크기를 갖고 있으며 16비트나 8비트 형태로 나누어서 사용되기도 한다.

General-Purpose Registers				16-bit	32-bit
31	16 15	8 7	0		
		AH	AL	AX	EAX
		BH	BL	BX	EBX
		CH	CL	CX	ECX
		DH	DL	DX	EDX
			BP	EBP	
			SI	ESI	
			DI	EDI	
			SP	ESP	

레지스터에 저장되는 데이터들은 부호가 없는 00000000 부터 FFFFFFFF 까지이다.

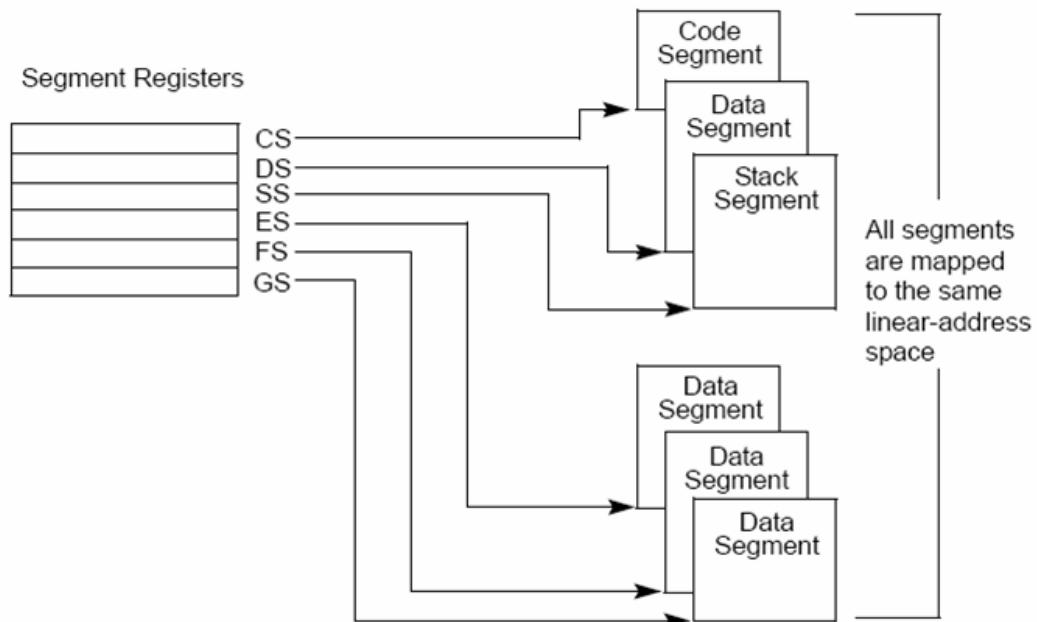
### 범용 레지스터(General-Purpose Register)

범용 레지스터는 논리, 산술 연산에 사용되는 오퍼랜드나 주소 계산을 위한 오픈랜드, 메모리 포인터에 사용된다.

- **EAX(Extended Accumulator Register)** : 산술연산에 사용
- **EBX(Extended Base Register)** : DS 세그먼트에 데이터를 가리키는 역할(주소지정을 확대하기 위한 인덱스로 사용)
- **ECX(Extended Counter Register)** : 루프의 반복 횟수나 좌우방향 시프트 비트 수 기억
- **EDX(Extended Data Register)** : 입출력 동작에서 사용
- **ESI(Extended Source Index)** : 출발지 인덱스에 대한 값 저장
- **EDI(Extended Destination Index)** : 다음 목적지 주소에 대한 값 저장
- **ESP(Extended Stack Pointer)** : 스택 포인터, 스택의 TOP 을 가리킴
- **EBP(Extended Base Pointer)** : 스택 내의 변수값을 읽는데 사용, 스택 프레임의 시작점을 가리킴

### 세그먼트 레지스터(Segment Register)

세그먼트 레지스터는 16 비트 세그먼트 선택자를 가지고 있다. 세그먼트 선택자는 메모리에서 세그먼트를 확인하는 특별한 포인터다.

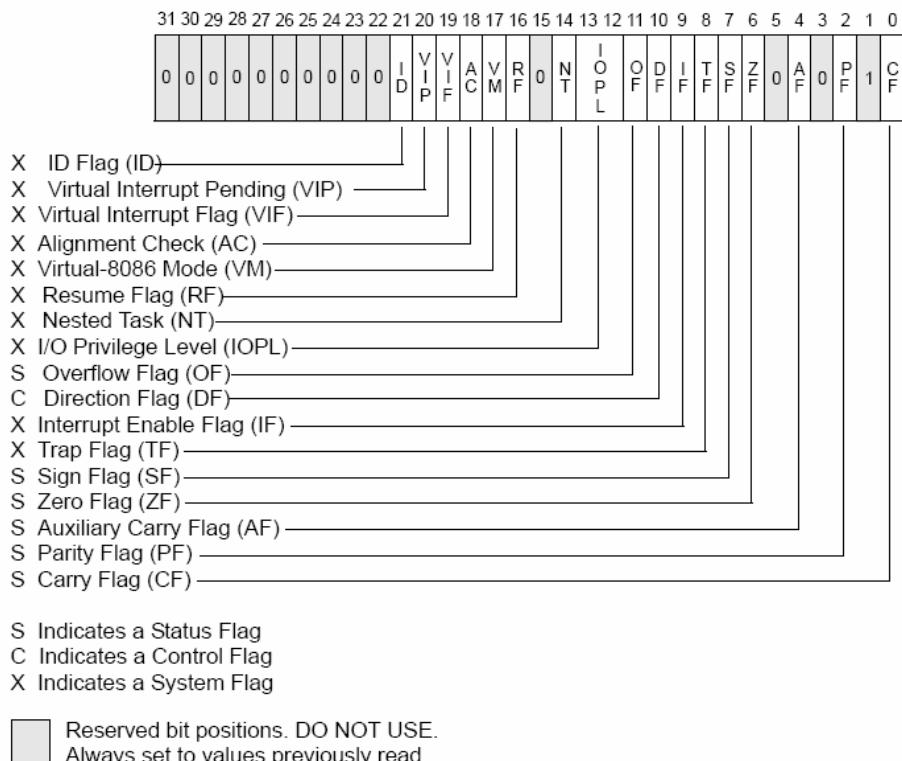


## Reverse Engineering

- **CS** : 코드 세그먼트의 시작 주소를 저장
  - 프로그램에서 명시적으로 로드 될 수는 없으나 프로그램 제어를 변경하는 명령 또는 내부 프로세서 조작에 의해 묵시적으로 로드 될 수 있다.
- **SS** : 스택으로 사용할 메모리 영역의 시작 주소를 저장
  - 명시적으로 로드 될 수 있기 때문에 프로그램에서 이 레지스터를 통해 여러 개의 스택을 셋업하여 그들 간을 상화 전환할 수 있다.
- **DS** : 데이터 세그먼트의 시작 주소를 저장
- **ES** : 여분의 데이터 세그먼트용, 문자열 처리 명령시 목적지 데이터 저장
- **FS/GS** : 여분의 세그먼트용, IA32 프로세서에서만 사용 가능

세그먼트 레지스터가 어떻게 사용되는지는 사용하는 메모리 관리 모델에 따라 달라질 수 있다. 메모리 관리 모델은 크게 Flat 메모리 모델과 Segmented 메모리 모델이 있다. 그 중 윈도우 환경에서 사용되며 일반적으로 많이 사용되는 메모리 모델은 Segmented 메모리 모델이다.

## EFLAGS 레지스터



EFLAGS 레지스터는 상태 플래그, 제어 플래그, 시스템 플래그 그룹으로 구성되어 있으며 1, 3, 5, 15, 22 ~ 31 비트는 예약되어 있으므로 사용할 수 없다.

LAHF, SAHF, PUSHF, PUSHFD, POPF, POPPFD 등의 명령으로 프로시저 스택 혹은 EAX 레지스터로 플래그의 그룹을 이동할 수 있다. EFLAGS 레지스터의 내용은 비트 조작 명령(BT, BTS, BTR, BTC)을 사용해 플래그를 검사 및 변경할 수 있다.

**상태 플래그(Status Flags)** : 산술연산(ADD, SUB, MUL, DIV)을 하는 명령의 결과를 가리킨다.

- **CF(bit 0) Carry flag** : 부호 없는 정수끼리의 산술연산 후 오버플로우가 발생했는지 확인
  - 0 : 오버플로우가 발생하지 않은 경우
  - 1 : 오버플로우가 발생한 경우
- PF(bit 2) Parity flag
- AF(bit 4) Adjust flag
- **ZF(bit 6) Zero flag** : 산술연산 또는 비교동작의 결과를 나타냄
  - 0 : 결과가 0이 아닌 경우
  - 1 : 결과가 0인 경우
- SF(bit 7) Sign flag
- **OF(bit 11) Overflow flag** : 부호 있는 수끼리의 산술연산 후 오버플로우가 발생했는지 확인
  - 0 : 오버플로우가 발생하지 않은 경우
  - 1 : 오버플로우가 발생한 경우(너무 큰 정수나 너무 작은 음수인 경우)

**제어 플래그(Control Flags)**

- DF(bit 10) Direction flag

**시스템 플래그(System Flags)**

- TF(bit 8) Trap flag
- IF(bit 9) Interrupt enable flag
- IOPL(bit 12, 13) I/O privilege level field
- NT(bit 14) Nested task flag
- RF(bit 16) Resume flag
- VM(bit 17) Virtual-8086 mode flag
- AC(bit 18) Alignment check flag
- VIF(bit 19) Virtual interrupt flag
- VIP(bit 20) Virtual interrupt pending flag
- ID(bit 21) Identification flag

리버스 엔지니어링에서 필요한 플래그는 ZF, OF, CF 세 가지 플래그이다.

## **Reverse Engineering**

### **EIP 레지스터**

현재 프로세서가 실행하고 있는 명령 바로 다음에 실행할 명령어의 오프셋을 저장한다. 수행한 명령어 길이만큼 증가된 EIP는 메모리 내의 다음 실행할 명령어를 가리킨다. 실제 모드로 동작할 때는 16 비트 EIP(상위 16 비트는 0)로, 보호 모드로 동작할 때는 32 비트 EIP로 동작한다.

EIP는 소프트웨어의 의해 조작할 수 없고 CALL, JMP, RET 와 같은 control-transfer 명령에 의해서만 영향을 받는다.

## 2-3. Assembly

### Assembly



- Arithmetic Instruction
- Data Transfer Instruction
- Logical Instruction
- String Instruction
- Control Transfer Instruction
- Processor Control Instruction

### Student Notes

어셈블리어는 리버스 엔지니어링을 하는데 중요한 기초 지식 중 하나이다. 어셈블리어 명령어들이 어떤 역할을 하는지에 대해서 이해하고 한 줄 한 줄의 의미보다는 여러 라인이 어떠한 하나의 의미를 가지는지를 파악하는 것이 중요하다.

이번 장에서는 여러 가지 어셈블리 명령어에 대해서 알아보도록 하겠다.

## Reverse Engineering

### Arithmetic Instruction

명령	설명
<b>ADD</b>	캐리를 포함하지 않은 덧셈
<b>SUB</b>	캐리를 포함하지 않은 뺄셈
<b>ADC</b>	캐리를 포함한 덧셈
<b>SBB</b>	캐리를 포함한 뺄셈
<b>CMP</b>	두 개의 오퍼랜드 비교
<b>INC</b>	오퍼랜드 내용을 1 증가
<b>DEC</b>	오퍼랜드 내용을 1 감소
<b>NEG</b>	오퍼랜드의 2의 보수, 즉 부호 반전
<b>AAA</b>	덧셈 결과 AL 값을 UNPACK 10 진수로 보정
<b>DAA</b>	덧셈 결과의 AL 값을 PACK 10 진수로 보정
<b>AAS</b>	뺄셈 결과 AL 값을 UNPCAK 10 진수로 보정
<b>DAS</b>	뺄셈 결과의 AL 값을 PCAK 10 진수로 보정
<b>MUL</b>	AX 와 오퍼랜드를 곱셈하여 결과를 AX 또는 DX:AX에 저장
<b>IMUL</b>	부호화된 곱셈
<b>AAM</b>	곱셈 결과 AX 값을 UNPACK 10 진수로 보정
<b>DIV</b>	AX 또는 DX:AX 내용을 오퍼랜드로 나눔. 몫은 AL, AX 나머지는 AH, DX로 저장
<b>IDIV</b>	부호화된 나눗셈
<b>AAD</b>	나눗셈 결과 AX 값을 UNPACK 10 진수로 보정
<b>CBW</b>	AL의 바이트 데이터를 부호 비트를 포함하여 AX 워드로 확장
<b>CWD</b>	AX의 워드 데이터를 부호를 포함하여 DX:AX의 더블 워드로 변환

### Data Transfer Instruction

명령	설명
<b>MOV</b>	데이터 이동(전송)
<b>PUSH</b>	오퍼랜드 내용을 스택에 쌓음
<b>POP</b>	스택으로부터 값을 가져옴
<b>XCHG</b>	첫 번째 오퍼랜드와 두 번째 오퍼랜드 교환
<b>XLAT</b>	BX:AL 이 지시한 테이블의 내용을 AL로 로드
<b>LEA</b>	메모리 오프셋값을 레지스터로 로드
<b>LDS</b>	REG $\leftarrow$ (MEM), DS $\leftarrow$ (MEM+2)
<b>LES</b>	REG $\leftarrow$ (MEM), ES $\leftarrow$ (MEM+2)
<b>LAHF</b>	플래그의 내용을 AH의 특정 비트로 로드
<b>SAHF</b>	AH의 특정 비트가 플래그 레지스터로 전송
<b>PUSHF</b>	플래그 레지스터의 내용을 스택에 쌓음
<b>POPF</b>	스택으로부터 플래그 레지스터로 가져옴

## Logical Instruction

명령	설명
<b>NOT</b>	오퍼랜드의 1의 보수, 즉 비트 반전
<b>SHL/SAL</b>	왼쪽으로 오퍼랜드만큼 자리 이동(최하위 비트는 0)
<b>SHR</b>	오른쪽으로 오퍼랜드만큼 자리 이동(최상위 비트는 0)
<b>SAR</b>	오른쪽 자리이동, 최상위 비트는 유지
<b>ROL/ROR</b>	왼쪽/오른쪽으로 오퍼랜드만큼 회전 이동
<b>RCL/RCR</b>	캐리를 포함하여 왼쪽/오른쪽으로 오퍼랜드만큼 회전 이동
<b>AND</b>	논리 AND
<b>TEST</b>	첫 번째 오퍼랜드와 두 번째 오퍼랜드를 AND 하여 그 결과로 플래그 세트
<b>OR</b>	논리 OR
<b>XOR</b>	배타 논리 합(OR)

## String Instruction

명령	설명
<b>REP</b>	REP 뒤에 오는 스트링 명령을 CS 가 0이 될 때까지 반복
<b>MOVS</b>	DS:DI 가 지시한 메모리 데이터를 ES:DI 가 지시한 메모리로 전송
<b>COMPS</b>	DS:DI 와 ES:DI 의 내용을 비교하고 결과에 따라 플래그 설정
<b>SCAS</b>	AL 또는 AX 와 ES:DI 가 지시한 메모리 내용 비교하고 결과에 따라 플래그 설정
<b>LODS</b>	SI 내용을 AL 또는 AX 로 로드
<b>STOS</b>	AL 또는 AX 를 ES:DI 가 지시하는 메모리에 저장

## Control Transfer Instruction

명령	설명	플래그의 변화
<b>CALL</b>	프로시저 호출	
<b>JMP</b>	무조건 분기	
<b>RET</b>	CALL 로 스택에 PUSH 된 주소로 복귀	
<b>JE/JZ</b>	결과가 0이면 분기	<b>ZF=1</b>
<b>JL/JNGE</b>	결과가 작으면 분기 (부호화된 수)	<b>SF != OF</b>
<b>JB/JNAE</b>	결과가 작으면 분기 (부호화 안된 수)	<b>CF=1</b>
<b>JLE/JNG</b>	결과가 작거나 같으면 분기 (부호화된 수)	<b>ZF=1 or SF != OF</b>
<b>JBE/JNA</b>	결과가 작거나 같으면 분기 (부호화 안된 수)	<b>CF=1 or ZF=1</b>
<b>JP/JPE</b>	페리티 플래그가 1이면 분기	<b>PF=1</b>
<b>JO</b>	오버플로우가 발생하면 분기	<b>OF=1</b>
<b>JS</b>	부호 플래그가 1이면 분기	<b>SF=1</b>

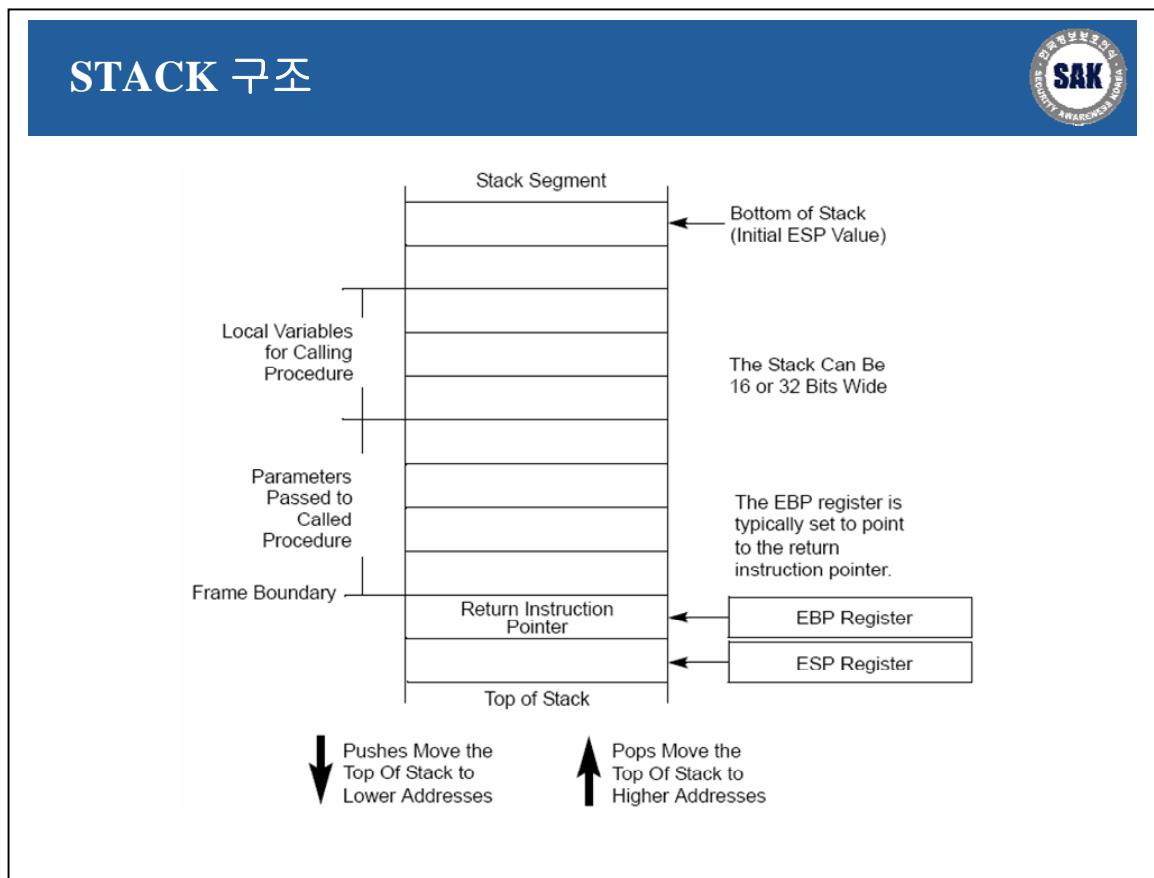
## Reverse Engineering

<b>JC</b>	캐리가 발생하면 분기	<b>CF=1</b>
<b>JNE/JNZ</b>	결과가 0이 아니면 분기	<b>ZF=0</b>
<b>JNL/JGE</b>	결과가 크거나 같으면 분기 (부호화된 수)	<b>SF=OF</b>
<b>JNB/JAE</b>	결과가 크거나 같으면 분기 (부호화 안된 수)	<b>CF=0</b>
<b>JNLE/JG</b>	결과가 크면 분기 (부호화된 수)	<b>ZF=0 and SF=OF</b>
<b>JNBE/JA</b>	결과가 크면 분기 (부호화 안된 수)	<b>CF=0 and ZF=0</b>
<b>JNP/JPO</b>	페리티 플래그가 0이면 분기	<b>PF=0</b>
<b>JNO</b>	오버플로우가 아닌 경우 분기	<b>OF=0</b>
<b>JNS</b>	부호 플래그가 0이면 분기	<b>SF=0</b>
<b>JNC</b>	캐리가 아닌 경우 분기	<b>CF=0</b>
<b>LOOP</b>	CX를 1 감소, 0이 될 때까지 지정된 라벨로 분기	
<b>LOOPZ/LOOPE</b>	CX가 0이 아니면 지정된 라벨로 분기	<b>ZF=1</b>
<b>LOOPNZ/LOOPNE</b>	CX가 0이 아니면 지정된 라벨로 분기	<b>ZF=0</b>
<b>JCXZ</b>	CX가 0이면 분기	<b>CX=0</b>
<b>INT</b>	인터럽트 실행	
<b>INTO</b>	오버플로우가 발생하면 인터럽트 실행	
<b>IRET</b>	인터럽트 복귀 (리턴)	

## Processor Control Instruction

명령	설명
<b>CLC</b>	캐리 플래그 클리어
<b>CMC</b>	캐리 플래그를 반전
<b>CLD</b>	디렉션 플래그를 클리어
<b>CLI</b>	인터럽트 플래그를 클리어
<b>HLT</b>	정지
<b>STC</b>	캐리 플래그 셋
<b>NOP</b>	아무 동작 하지 않음
<b>STD</b>	디렉션 플래그 셋
<b>STI</b>	인터럽트 플래그 셋
<b>WAIT</b>	프로세서를 일시 정지 상태로 한다
<b>ESC</b>	이스케이프 명령

## 2-4. STACK 구조



### Student Notes

스택은 한 쪽 끝에서만 데이터를 넣거나 뺄 수 있는 선형 구조로 되어 있다. 자료를 넣는 것을 PUSH라고 하고 데이터를 꺼내는 것을 POP이라고 하는데 이 때 꺼내지는 데이터는 가장 최근에 넣은 데이터가 된다. 이처럼 나중에 넣은 값이 먼저 나오는 것을 LIFO(Last In First Out)구조라고 한다.

스택은 구현 방법에 따라 네 가지로 구분할 수 있다.

- Full Stack : TOP 이 마지막으로 PUSH 된 데이터를 가리킴
- Empty Stack : TOP 이 다음 데이터가 들어올 곳을 가리킴
- Ascending Stack : 낮은 메모리 주소에서 시작하여 높은 메모리 주소 방향으로 자름
- Descending Stack : 높은 메모리 주소에서 시작하여 낮은 메모리 주소 방향으로 자름

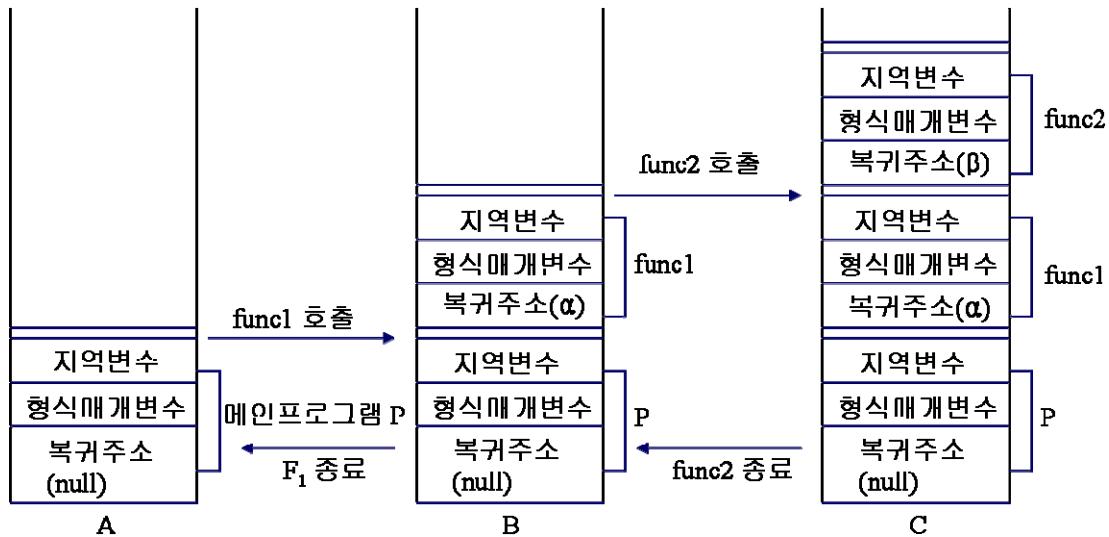
## Reverse Engineering

인텔 아키텍처의 스택은 Full Descending Stack의 형태를 가지고 있다. 즉, 스택의 TOP이 마지막으로 들어온 데이터를 가리키고 있으며 높은 메모리 주소에서 시작하여 낮은 메모리 주소 방향으로 자란다는 뜻이다.

스택에서 주로 사용되는 세 개의 레지스터에는 EBP, ESP, EIP가 있다. EBP는 스택 프레임의 시작점을 가리키는데 함수 내에서 ESP를 통해 스택의 크기를 늘리고 줄일 때 기존의 ESP 값을 백업하는 용도로 사용된다. 그리고 ESP는 스택의 TOP을 가리키는데 스택 포인터 이동시에 사용된다.

EIP는 다음에 실행할 명령의 오프셋을 가지고 있는데 보통 함수 호출 후 복귀 주소를 나타내는 레지스터로써 소프트웨어에 의해 조작될 수 없다.

다음 그림은 시스템에서 프로그램이 실행되고 함수가 호출될 때 스택의 변화를 나타낸 그림이다.



그림에서 보는 것처럼 메인 프로그램이 실행되는 도중에 F1이라는 함수가 호출이 되면 새로운 func1 함수에게 새로운 스택을 할당한다. 이 때 func1 함수의 실행이 끝나면 다시 메인 함수로 돌아오기 위해서 복귀주소를 먼저 스택에 넣어둔다. 이것은 스택의 func1 함수의 실행이 끝나면 func1 함수가 사용하던 스택에 있던 데이터를 전부 비운 후 마지막으로 복귀주소를 보고 메인 함수로 돌아오기 위함이다. 그리고 func1 함수내에서 func2 함수를 호출하는 경우도 같은 방법을 사용한다.

## 2-5. 함수 호출 규약(Calling Convention)

### 함수 호출 규약 (Calling Convention)



- 함수 호출 방법에 대한 규약
  - Argument 전달 방법
    - Stack : C 표준 라이브러리, cdecl, stdcall, Pascal
    - Register : fastcall
  - Argument 전달 순서
    - Right to Left : cdecl, stdcall
    - Left to Right
  - Stack Clearing 방법
    - Caller : cdecl
    - Callee : stdcall
  - Return Value
- 주요 함수 호출 규약
  - \_\_stdcall, \_\_cdecl, \_\_fastcall

### Student Notes

함수 호출 규약은 프로그램이 함수에게 파라미터를 전달하고 결과값을 다시 받는 일련의 표준화된 방법이다. 즉, 함수 호출을 위해서 밟는 절차를 정해둔 것이라고 생각하면 되겠다.

함수의 호출을 구현하기 위해서는 플랫폼이나 프로그래밍 언어에 따라 다를 수 있겠지만 기본적으로 실행할 함수의 코드 위치는 갖고 있는 포인터나 그 값을 가져올 수 있는 함수 이름, 현재 처리하고 있는 함수의 정보를 저장할 공간, 함수의 실행이 끝난 뒤 리턴값을 돌려 받을 공간, 함수의 실행에 필요한 인자들 넘겨줄 공간 등이 필요하다.

함수 호출 규약을 이해하기 위해서 먼저 스택 프레임(Stack Frame)에 대해서 알아보도록 하겠다. 스택 프레임은 호출된 함수가 실행되는 동안 필요한 지역 변수나 호출한 함수의 실행에 필요한 정보가 손실되

## Reverse Engineering

지 않도록 스택에 저장할 때 사용하는 구조를 의미한다. 이렇게 저장해야 하는 정보에는 다음과 같은 것들이 있다.

- 함수의 실행이 종료된 후에 리턴 할 주소
- 지역 변수
- 자신을 호출한 함수의 스택 프레임 위치
- 레지스터 같은 기계 상태
- 예외 처리 리스트 등

스택 프레임은 기본적으로 함수가 호출될 때마다 새로 설정이 되고 Frame Pointer(또는 Base Pointer)를 통해서 참조할 수 있다. 프레임 포인터는 현재 실행되고 있는 함수의 스택 프레임이 스택에서 어느 주소 위치에 있는지 가리키는 값으로 인텔 아키텍처에서는 EBP 레지스터에 저장된다.

함수 호출 규약에는 총 5 가지 규약이 있지만stdcall, cdecl, thiscall, fastcall, naked) 여기서는 자주 사용되는 stdcall과 cdecl 호출 규약에 대해서만 알아보도록 하겠다.

### stdcall

stdcall 호출 규약은 윈도우 플랫폼에서 시스템 API를 호출할 때 사용하는 규약이다. stdcall 호출 규약에서 컴파일러는 멤버 함수의 오른쪽 인자부터 왼쪽 인자 순으로 스택에 집어 넣고, **함수 호출이 종료되면 호출된 함수(Callee)**가 인자를 스택에서 제거하며, 리턴 값이 있는 경우 레지스터 eax를 통해서 리턴 값을 돌려 받는다. 호출된 함수가 스택에서 제거해야 할 인자의 개수를 정확히 알아야 하기 때문에 가변 개수의 인자를 전달할 수 없다. 보통 call 문 다음에 ret n을 이용하여 스택을 정리한다.

```
* .text:00401013      push   3
* .text:00401015      push   2
* .text:00401017      push   1
* .text:00401019      call   _myFunc1@12    ; myFunc1(x,x,x)
* .text:0040101E      pop    ebp
* .text:0040101F      retn
.text:0040101F _main  endp

* .text:00401044      mov    esp, ebp
* .text:00401046      pop    ebp
* .text:00401047      retn  0Ch
.text:00401047 myFunc1  endp
.text:00401047
```

## cdecl

cdecl 호출 규약은 C 컴파일러 혹은 C++ 컴파일러에서 전역 함수나 전역 스태틱 함수, 스태틱 멤버 함수 호출에 기본적으로 사용하는 규약이다. cdecl 호출 규약에서 컴파일러는 멤버 함수의 오른쪽 인자부터 왼쪽 인자 순으로 스택에 집어 넣고, 함수 호출이 종료된 뒤 함수 호출자(Caller)는 스택에 집어 넣은 인자를 제거하며, 리턴 값이 있는 경우 레지스터 eax를 통해서 리턴 값을 돌려 받는다. cdecl 호출 규약에서 는 호출자가 넣은 인자를 제거하게 되어 있으므로, 함수가 가변 개수의 인자를 가져도 안전하게 스택을 정리할 수 있다. 보통 call 문 다음에 add esp, n 을 이용하여 스택을 정리한다.

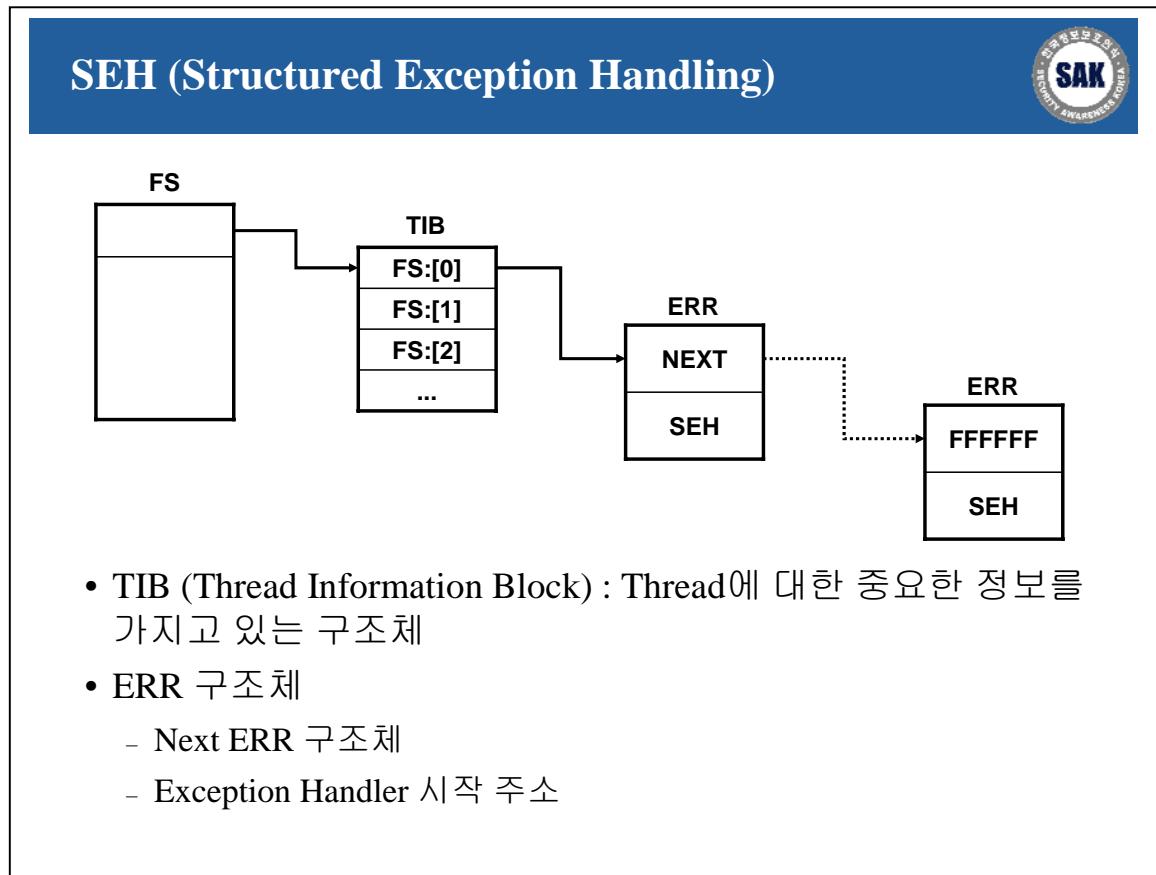
```
* .text:00401078      push   3
* .text:0040107A      push   2
* .text:0040107C      push   1
* .text:0040107E      call    j_myFunc1
* .text:00401083      add    esp, 0Ch

* .text:00401044      mov    esp, ebp
* .text:00401046      pop    ebp
* .text:00401047      retn
.text:00401047 myFunc1
.text:00401047
```

가장 자주 사용되는 함수 호출 규약인 stdcall 과 cdecl 의 가장 큰 차이점은 누가 스택을 정리하느냐 하는 것이다.

stdcall 호출 규약의 경우 호출된 함수(Callee)가 스택을 정리하기 때문에 호출하는 함수(Caller)와 Callee 모두 파라미터의 크기를 알고 있어야 정상적인 처리가 가능하지만 cdecl 호출 규약의 경우 Caller 가 스택을 정리하기 때문에 Callee 는 파리미터의 크기를 정확히 몰라도 된다.

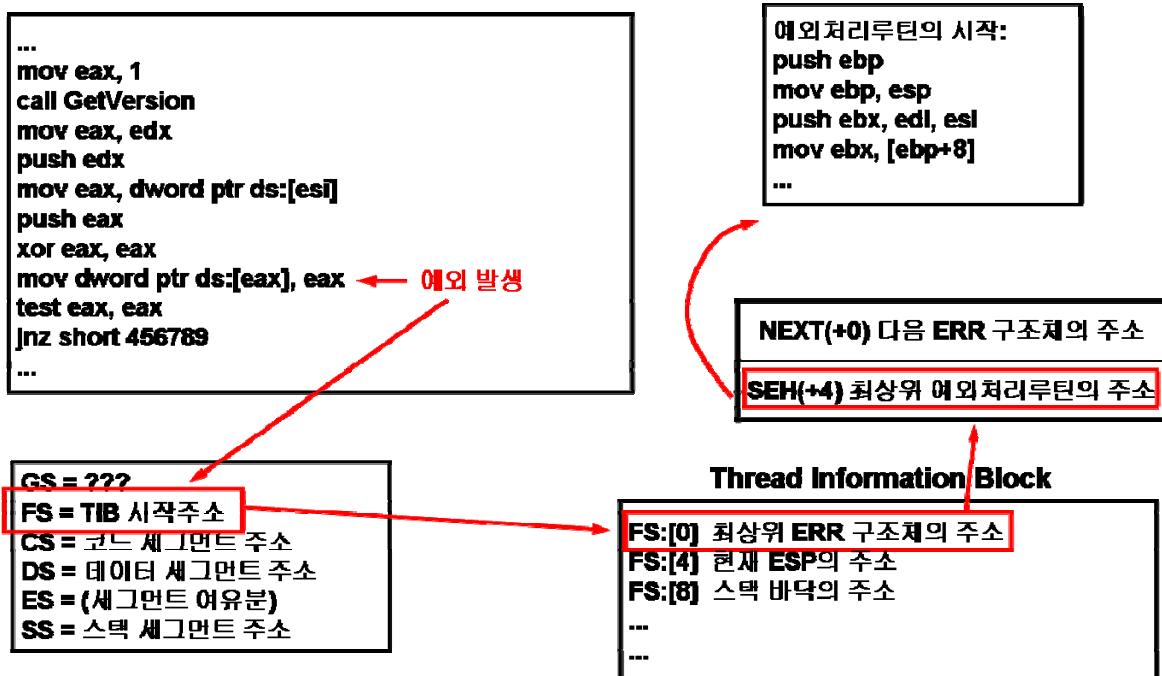
## 2-6. SEH(Structured Exception Handling)



### Student Notes

SEH(Structured Exception Handling)은 윈도우에서 제공하는 예외 처리 방식이다. ‘예외’란 예기치 못하거나 해당 프로세스의 정상적인 실행을 방해하는 이벤트이다. 예외를 발생시키는 상황은 여러 가지가 있는데 대표적인 것들을 0으로 나누기나 숫자 형식의 오버플로우, 잘못된 메모리 주소에 대한 읽기나 쓰기를 시도하는 것들이 있다.

예외 발생시 운영체제는 TIB에서 가리키고 있는 주소로 이동하여 예외를 처리한다. TIB는 쓰래드에 대한 중요한 정보를 가지고 있는 구조체인데 TIB의 첫 번째 멤버인 FS:[0]은 ERR(Exception Registration Record) 구조체를 가리키고 있는 포인터이다. ERR 구조체는 다음 ERR 구조체의 주소와 Exception Handler 시작 주소의 두 개의 멤버를 가지고 있다.



Exception Handler의 프로토 타입은 다음과 같다.

```

__cdecl _except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablishFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext
);

```

\_CONTEXT는 예외 발생시 모든 레지스터의 값을 저장한다. 그리고 EAX 0, ret 시 사용하게 된다.

+0 context 플래그	세그먼트 레지스터
(GetThreadContext API를 이용할 때 사용)	+8C gs register +90 fs register +94 es register +98 ds register
디버그 레지스터	일반 레지스터
+4 debug register #0 +8 debug register #1 +C debug register #2 +10 debug register #3 +14 debug register #6 +18 debug register #7	+9C edi register +A0 esi register +A4 ebx register +A8 edx register +AC ecx register +B0 eax register

## Reverse Engineering

FPU / MMX 레지스터	제어 레지스터
+1C ControlWord +20 StatusWord +24 TagWord +28 ErrorOffset +2C ErrorSelector +30 DataOffset +34 DataSelector +38 FP registers * 8 (각각 10 바이트 차지) +88 Cr0NpxState	+B4 ebp register +B8 eip register +BC cs register +C0 eflags register +C4 esp register +C8 ss register

Winnt.h에 정의되어 있는 EXCEPTION\_RECORD는 다음과 같다.

```
typedef struct _EXCEPTION_RECORD {
    DWORD    ExceptionCode;
    DWORD    ExceptionFlags;
    struct   _EXCEPTION_RECORD *ExceptionRecord;
    PVOID    ExceptionAddress;
    DWORD    NumberParameters;
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

ExceptionCode : 발생한 예외의 종류

- **C0000005h** : 접근할 수 없는 메모리를 Read/Write 하려 할 때
- **C0000094h** : 0으로 나누려 할 때
- **C0000095h** : DIV 할 때 발생하는 오버플로
- **C00000FDh** : 스택의 최대크기를 넘을 때
- **80000001h** : Guard Page 플래그가 셋팅 된 메모리주소를 접근하려 할 때
- **C0000025h** : 더 이상 처리할 수 없는 Exception
- **C0000026h** : 예외처리도중 시스템에 의해 사용되어지는 ExceptionCode
- **80000003h** : INT3
- **80000004h** : Trap 플래그 세트

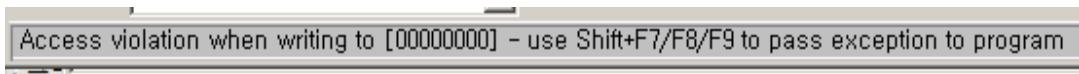
ExceptionFlags : 발생한 예외에 대한 몇 가지 부가적인 정보를 포함

- 0 : 처리할 수 있는 Exception
- 1 : 처리할 수 없는 Exception
- 2 : 스택의 Unwinding하고 있을 때

예제를 통해서 SEH에 대해서 알아보도록 하겠다.

## Reverse Engineering

먼저 예제 프로그램을 Ollydbg에서 열고 실행을 시켜보면 아래쪽 상태바에서 다음과 같은 메시지를 만나게 된다.



예외가 발생한 것을 볼 수 있다. 이 때 SEH chain을 확인하거나 Stack 정보를 확인하면 예외를 처리할 부분이 어느 곳인지 확인할 수 있다.

SEH chain of main thread	
Address	SE handler
0007FF90	00B13313
0007FFE0	kernel32.7C821A60

현재 ERR 구조체를 스택에서 확인해보면 Next SEH는 0007FFE0에 위치해 있고 현재 예외를 처리하는 곳은 00B13313에 있다. 00B13313에 브레이크 포인트를 걸고 SEH에서 어떠한 행동을 하는지 확인해보겠다. 브레이크 포인트 설정 후 Shift + F9를 눌러 진행한다.

SEH chain of main thread	
Address	SE handler
0007FF90	00B13313
0007FFE0	kernel32.7C821A60

브레이크 포인트 설정 후 Shift + F9를 눌러 진행한다. 00B13313에서 멈춘 것을 확인할 수 있다.

CPU - main thread		
Address	Hex dump	Disassembly
00B13313	68 1C33B100	PUSH 0B1331C
00B13318	FF0424	INC DWORD PTR SS:[ESP]
00B1331B	C3	RETN

이 부분부터가 예외 발생시 처리하는 부분이다. 스택의 내용을 확인해보도록 하자.

Address	Value	Comment
0007FBC8	7C968752	RETURN to ntdll.7C968752
0007FBCC	0007FCA8	
0007FBDO	0007FF90	
0007FBD4	0007FCC4	
0007FBD8	0007FC84	
0007FBDC	0007FF90	Pointer to next SEH record
0007FBE0	7C968766	SE handler

예외처리 루틴으로 들어가게 되면 앞서 살펴본 것과 같이 ExceptionRecord, EstablishedFrame, Context, DispatchContext 순서로 스택에 저장된다.

## Reverse Engineering

```

0007FBC8    7C968752    RETURN to ntdll.7C968752 // Return 주소
0007FBCC    0007FCA8    // ExceptionRecord, 환경 저장
0007FBD0    0007FF90    // Frame, ERR 구조체
0007FBD4    0007FCC4    // Context
0007FBD8    0007FC84    // DispatchContext

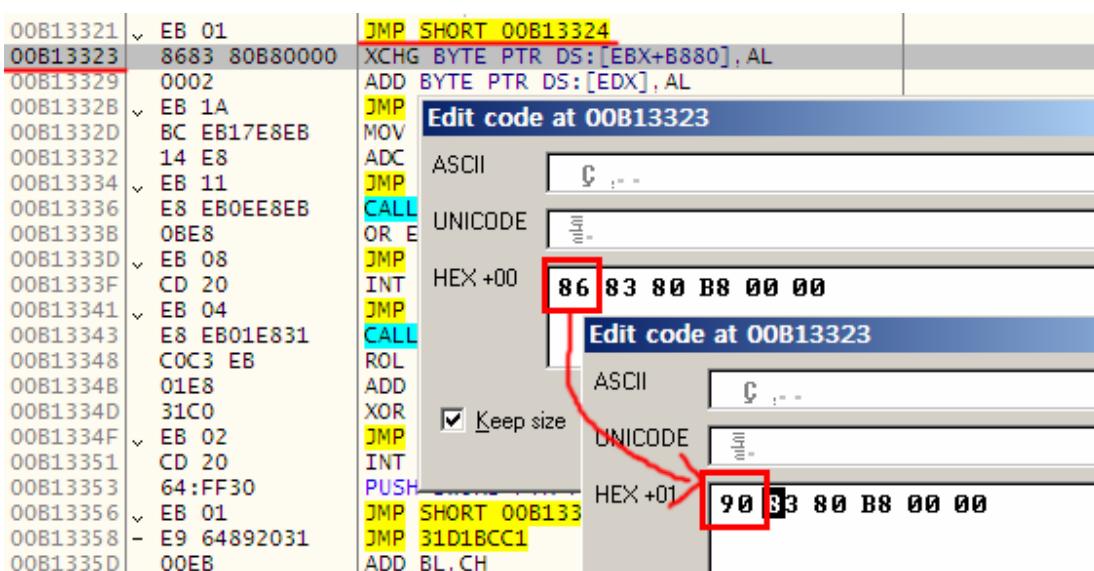
```

현재 위치에서부터 아래쪽으로 코드들을 확인해 보면 뭔가 잘못되었다는 것을 알 수 있다.

Address	Hex dump	Disassembly
00B13313	68 1C33B100	PUSH 0B1331C
00B13318	FF0424	INC DWORD PTR SS:[ESP]
00B1331B	C3	RETN
00B1331C	BC 8B44240C	MOV ESP, 0C24448B
00B13321	EB 01	JMP SHORT 00B13324
00B13323	8683 80B80000	XCHG BYTE PTR DS:[EBX+B880], AL
00B13329	0002	ADD BYTE PTR DS:[EDX], AL
00B1332B	EB 1A	JMP SHORT 00B13347
00B1332D	BC EB17E8EB	MOV ESP, EBE817EB
00B13332	14 E8	ADC AL, 0E8
00B13334	EB 11	JMP SHORT 00B13347
00B13336	E8 E80EE8EB	CALL EC994226
00B13338	0BE8	OR EBP, EAX
00B1333D	EB 08	JMP SHORT 00B13347
00B1333F	CD 20	INT 20

00B13321에서 JMP SHORT 00B13324로 되어 있으므로 00B13324로 점프를 해야 하는데 00B13324가 보이지 않는다는 것은 디버거가 해석을 잘못했기 때문이다. 이렇게 실제 실행되는 내용들이 쓰레기 코드 때문에 보이지 않는다면 이런 부분들을 찾아서 필요 없는 코드들을 전부 NOP(0x90) 처리해 준다.

코드를 수정할 부분에서 바이너리 에디트를 해서(단축키 CTRL + E) 다음과 같이 수정한다.



위와 같이 수정하면 해당 부분의 코드가 다음과 같이 변경된다.

00B13321	EB 01	JMP SHORT 00B13324
00B13323	90	NOP
00B13324	8380 B8000000	ADD DWORD PTR DS:[EAX+B8], 2

이런 코드들을 전부 수정하면 다음과 같다.

CPU - main thread		
Address	Hex dump	Disassembly
00B13313	68 1C33B100	PUSH 0B1331C
00B13318	FF0424	INC DWORD PTR SS:[ESP]
00B1331B	C3	RETN
00B1331C	90	NOP
00B1331D	8B4424 0C	MOV EAX, DWORD PTR SS:[ESP+C]
00B13321	EB 01	JMP SHORT 00B13324
00B13323	90	NOP
00B13324	8380 B8000000	ADD DWORD PTR DS:[EAX+B8], 2
00B1332B	EB 1A	JMP SHORT 00B13347
00B1332D	BC EB17E8EB	MOV ESP, EBE817EB
00B13332	14 E8	ADC AL, 0E8
00B13334	EB 11	JMP SHORT 00B13347
00B13336	E8 EB0EE8EB	CALL EC994226
00B1333B	0BE8	OR EBP, EAX
00B1333D	EB 08	JMP SHORT 00B13347
00B1333F	CD 20	INT 20
00B13341	EB 04	JMP SHORT 00B13347
00B13343	90	NOP
00B13344	90	NOP
00B13345	90	NOP
00B13346	90	NOP
00B13347	31C0	XOR EAX, EAX
00B13349	C3	RETN

수정된 코드를 분석해 보면 SEH 발생시 어떤 행동들을 하는지 확인할 수 있다.

```
/*B13313*/ PUSH 0B1331C
/*B13318*/ INC DWORD PTR SS:[ESP]
/*B1331B*/ RETN
/*B1331C*/ NOP
```

→ 스택에 0B1331C를 넣은 후 ESP에 있는 값을 1 증가 시킨다. 이것은 JMP 0B1331D와 같은 의미이다.

```
/*B1331D*/ MOV EAX,DWORD PTR SS:[ESP+C]
```

→ ESP+C의 주소에 있는 값을 EAX에 저장한다. ESP+C는 0007FD4가 되고 이 주소에 있는 값은 0007FCC4이다. 이 곳은 CONTEXT 내용 있는 곳이다.

## Reverse Engineering

```
EAX 00000000  
ECX 00B13313  
EDX 7C968766 ntdll.7C968766  
EBX 00000000  
ESP 0007FBC8 0007FBC8 + C = 0007FBD4  
EBP 0007FBE8  
ESI 00000000  
EDI 00000000  
EIP 00B1331D
```

Address	Hex dump
0007FBD4	C4 FC 07 00 84 FC 07 00 90 FF 07 00 66 87 96 7C
0007FBE4	90 FF 07 00 90 FC 07 00 23 87 96 7C A8 FC 07 00

```
/*B13321*/ JMP SHORT 00B13324  
/*B13324*/ ADD DWORD PTR DS:[EAX+B8],2
```

→ EAX+B8 은 CONTEXT 구조체를 확인해보면 EIP 레지스터를 나타낸다. 즉, EIP 를 2 만큼 증가한다.

```
/*B1332B*/ JMP SHORT 00B13347
```

→ 00B13347 로 점프

```
/*B1332D*/ MOV ESP,E8E817EB  
/*B13332*/ ADC AL,0E8  
/*B13334*/ JMP SHORT 00B13347  
/*B13336*/ CALL EC994226  
/*B1333B*/ OR EBP,EAX  
/*B1333D*/ JMP SHORT 00B13347  
/*B1333F*/ INT 20  
/*B13341*/ JMP SHORT 00B13347
```

→ 앞에서 00B13347 로 점프하기 때문에 의미 없는 코드들

```
/*B13347*/ XOR EAX,EAX  
/*B13349*/ RETN
```

→ EAX 값을 초기화하고 SEH 처리 완료한다. 다음 SEH 를 호출할 필요 없이 바로 복귀

SEH 처리 루틴이 끝나면 다음과 같은 코드를 만나게 된다.

Address	Hex dump	Disassembly
7C968752	64:8B25 00000000	MOV ESP,DWORD PTR FS:[0]
7C968759	64:8F05 00000000	POP DWORD PTR FS:[0]

이것은 SEH 체인에서 상위 SEH 를 제거하는 과정이다. 그런 후 F9 를 눌러 계속 진행하면 된다.

지금까지의 과정은 SEH 에서 어떤 행동들을 하는지 확인한 것이었다. 이 예제와 같이 다른 행동을 하지

## Reverse Engineering

않고 EIP 만 변경하는 것이라면 문제가 없겠지만 SEH 처리 중간에 다른 행동들을 하게끔 작성된 코드들도 있을 수 있다.

SEH 를 하나씩 처리해 나가다 보면 마지막 SEH 처리에서 다음과 같은 모습을 볼 수 있다.

C CPU - main thread	Address	Hex dump	Disassembly
00B1302D	8B4424 0C		MOV EAX, DWORD PTR SS:[ESP+C]
00B13031	8380 B8000000		ADD DWORD PTR DS:[EAX+B8], 2
00B13038	51		PUSH ECX
00B13039	31C9		XOR ECX, ECX
00B1303B	8948 04		MOV DWORD PTR DS:[EAX+4], ECX
00B1303E	8948 08		MOV DWORD PTR DS:[EAX+8], ECX
00B13041	8948 0C		MOV DWORD PTR DS:[EAX+C], ECX
00B13044	8948 10		MOV DWORD PTR DS:[EAX+10], ECX
00B13047	C740 18 550100		MOV DWORD PTR DS:[EAX+18], 155
00B1304E	59		POP ECX
00B1304F	31C0		XOR EAX, EAX
00B13051	C3		RETN

마지막 SEH 는 EIP 를 2 증가하고 SEH 내부에서 하드웨어 브레이크포인트를 클리어하는 작업을 하는 것을 볼 수 있다. XOR EAX, EAX 는 EAX 값은 0 으로 설정하는 것이고 이것은 현재 SEH 가 발생한 익셉션을 정상적으로 처리했다는 것을 의미한다.

## 2-7. C 코드 컴파일 후 어셈코드 변환 모습

### C 코드 컴파일 후 어셈코드 변환 모습



- for, while, if
- String 관련 함수
  - strcpy
  - strcmp
  - strlen
- 컴파일러는 자동화된 도구이기 때문에 규칙을 가지고 컴파일을 하기 때문에 특정 프로그램을 컴파일 한 후 어셈블리어 코드는 일정한 패턴을 갖는다.

### Student Notes

C 코드를 컴파일 한 후 실행파일을 디버거로 분석하면 어셈블리어로 표현이 되는 것을 볼 수 있고 특정 함수의 경우 동일한 또는 비슷한 패턴의 어셈블리어로 표현되는 것을 볼 수 있다. 우리는 이런 어셈블리어로 된 코드들을 보고 분석을하게 된다. 숙련된 리버서들은 여러 패턴의 코드들을 보기 때문에 어셈블리어 코드만 보고도 금방 어떤 행동들을 하는 코드인지 식별하는 것이 가능하지만 덜 숙련된 리버서들은 조금 힘들 수 있다.

이번 장에서는 몇 가지 함수들을 예제로 하여 어셈블리어 코드와 비교하면서 C 코드가 컴파일된 후에는 어떤 어셈블리어 코드로 변하는지 알아보도록 하겠다.

**Example #1. for**

```
#include <stdio.h>

main(int argc, char *argv[])
{
    int i=0;
    int sum=0;
    for(i=0; i<10; i++)
    {
        sum+=i;
    }
    printf("%d, %d\n", i, sum);
}
```

위 코드를 컴파일하여 실행파일을 만든 후 실행파일을 디버거로 확인하면 다음과 같은 코드들을 볼 수 있다.

Address	Hex dump	Disassembly
0040102F	. C745 F8 00000	MOV DWORD PTR SS:[EBP-8], 0
00401036	. C745 FC 00000	MOV DWORD PTR SS:[EBP-4], 0
0040103D	. EB 09	JMP SHORT for.00401048
0040103F	> 8B45 FC	MOV EAX, DWORD PTR SS:[EBP-4]
00401042	. 83C0 01	ADD EAX, 1
00401045	. 8945 FC	MOV DWORD PTR SS:[EBP-4], EAX
00401048	> 837D FC 0A	CMP DWORD PTR SS:[EBP-4], 0A
0040104C	. 7D 0B	JGE SHORT for.00401059
0040104E	. 8B4D F8	MOV ECX, DWORD PTR SS:[EBP-8]
00401051	. 034D FC	ADD ECX, DWORD PTR SS:[EBP-4]
00401054	. 894D F8	MOV DWORD PTR SS:[EBP-8], ECX
00401057	.^ EB E6	JMP SHORT for.0040103F

루프문을 확인하는 가장 확실한 방법은 v 표시와 ^ 표시, 그리고 > 표시를 확인하는 것이다. 0040103D에서 00401048로 분기하고 조건이 만족할 경우(CMP 결과) 00401059로 그렇지 않을 경우 계속 코드를 진행하다가 00401057에서 0040103F로 분기한다.

## Reverse Engineering

### Example #2. if, strcpy, strcmp, strlen

이번 예제는 if 문과 함께 스트링 관련 함수들(strcpy, strcmp, strlen)이 있는 예제를 살펴보겠다.

```
#include <stdio.h>

void main( int argc, char *argv[ ] )
{
    char src[] = "ForEducationbydemantos";
    char *dst=(char *)malloc( 30 * sizeof(char));
    int cnt=0;

    printf("Input the password : ");
    fgets(dst, 30, stdin);
    dst[strlen(dst)-1] = '\0';

    if( strlen(dst) != strlen(src) ) {
        printf("Not match!!\n");
        exit(0);
    }
    else {
        if( !strcmp(dst, src) ) {
            printf("Good\n");
            exit(0);
        }
        else
            printf("Not match!!\n");
            exit(0);
    }
}
```

위 예제는 비교할 패스워드를 위한 메모리 공간을 할당한 후 입력된 값의 길이를 비교한 후 맞을 경우 입력된 값과 실제 패스워드의 문자열을 비교하는 프로그램이다.

먼저 문자의 길이를 구하는 부분이다.

004013CB	. 05 00000000	ADD EAX, 0
004013D0	> 8B01	MOV EAX, DWORD PTR DS:[ECX]
004013D2	. BA FFFFEF7E	MOV EDX, 7EFEEFFF
004013D7	. 03D0	ADD EDX, EAX
004013D9	. 83F0 FF	XOR EAX, FFFFFFFF
004013DC	. 33C2	XOR EAX, EDX
004013DE	. 83C1 04	ADD ECX, 4
004013E1	. A9 00010181	TEST EAX, 81010100
004013E6	.^ 74 E8	JE SHORT ex.004013D0

## Reverse Engineering

4글자씩 잘라내면서 문자의 길이를 구한다. 함수의 실행 결과는 EAX 레지스터에 저장이 되고 F8로 한 스텝씩 진행하면서 EAX 레지스터를 살펴보면 길이가 저장되는 것을 확인할 수 있다. 아래 코드는 입력된 문자와 비교할 문자의 길이를 구하는 코드들이다.

0040106B	.	83C4 0C	ADD ESP, 0C
0040106E	.	8B4D E4	MOV ECX, DWORD PTR SS:[EBP-1C]
00401071	.	51	PUSH ECX
00401072	.	E8 39030000	CALL ex.strlen
00401077	.	83C4 04	ADD ESP, 4
0040107A	.	8B55 E4	MOV EDX, DWORD PTR SS:[EBP-1C]
0040107D	.	C64402 FF 00	MOV BYTE PTR DS:[EDX+EAX-1], 0
00401082	.	8B45 E4	MOV EAX, DWORD PTR SS:[EBP-1C]
00401085	.	50	PUSH EAX
00401086	.	E8 25030000	CALL ex.strlen
00401088	.	83C4 04	ADD ESP, 4
0040108E	.	8BF0	MOV ESI, EAX
00401090	.	8D4D E8	LEA ECX, DWORD PTR SS:[EBP-18]
00401093	.	51	PUSH ECX
00401094	.	E8 17030000	CALL ex.strlen
00401099	.	83C4 04	ADD ESP, 4
0040109C	.	3BF0	CMP ESI, EAX
0040109E	.~	74 19	JE SHORT ex.004010B9
004010A0	.	68 24004200	PUSH OFFSET ex.??_C@_ON@INJI@Not?5match`
004010A5	.	E8 06060000	CALL ex.printf

0040109E에서 입력된 문자의 길이와 비교할 문자의 길이를 비교해서 같을 경우 004010B9으로 분기하고 그렇지 않을 경우 “Not match!!”라는 메시지를 출력하는 루틴으로 분기하게 된다. 문자의 길이가 같을 경우 입력한 문자와 비교할 문자가 같은지 비교하는 루틴으로 분기한다.(004010B9)

004010B9	>	8D55 E8	LEA EDX, DWORD PTR SS:[EBP-18]
004010BC	.	52	PUSH EDX
004010BD	.	8B45 E4	MOV EAX, DWORD PTR SS:[EBP-1C]
004010C0	.	50	PUSH EAX
004010C1	.	E8 8A000000	CALL ex.strcmp

사용자가 입력한 문자를 EAX에 저장하고 비교할 문자를 EDX에 저장한 후 strcmp 함수를 호출하고 있다. strcmp 함수 내부로 들어가 보면(F7을 눌러 진행) 다음과 같은 코드를 볼 수 있다. 함수 내부에서는 4글자씩 분할한 후 비교를 하고 있다.

먼저 첫 번째 글자를 비교하는 부분이다.

00401160	>	8B02	MOV EAX, DWORD PTR DS:[EDX]
00401162	.	3A01	CMP AL, BYTE PTR DS:[ECX]
00401164	.~	75 2E	JNZ SHORT ex.00401194
00401166	.	0AC0	OR AL, AL
00401168	.~	74 26	JE SHORT ex.00401190

## Reverse Engineering

두 번째 글자를 비교하는 부분이다.

0040116A	. 3A61 01	CMP AH, BYTE PTR DS:[ECX+1]
0040116D	.v 75 25	JNZ SHORT ex.00401194
0040116F	. 0AE4	OR AH, AH
00401171	.v 74 1D	JE SHORT ex.00401190

두 글자 비교 후 SHR 을 이용해서 쉬프트한 후 다시 두 문자를 비교하는 부분이다. 0x10 은 10 진수로 16이고 이것은 16 비트를 의미한다. 즉, 2 바이트만큼 오른쪽으로 쉬프트하므로 4 글자 중 앞에서 비교했던 두 글자를 없애고 나머지 두 글자로 다음 글자를 비교하는 것이다.

00401173	. C1E8 10	SHR EAX, 10
00401176	. 3A41 02	CMP AL, BYTE PTR DS:[ECX+2]
00401179	.v 75 19	JNZ SHORT ex.00401194
0040117B	. 0AC0	OR AL, AL
0040117D	.v 74 11	JE SHORT ex.00401190
0040117F	. 3A61 03	CMP AH, BYTE PTR DS:[ECX+3]
00401182	.v 75 10	JNZ SHORT ex.00401194

처음 4 글자가 같을 경우 ECX 와 EDX 값을 4 씩 더해서 다음 4 글자를 비교하게 된다.

00401184	. 83C1 04	ADD ECX, 4
00401187	. 83C2 04	ADD EDX, 4
0040118A	. 0AE4	OR AH, AH
0040118C	.^ 75 D2	JNZ SHORT ex.00401160
0040118E	. 8BFF	MOV EDI, EDI
00401190	> 33C0	XOR EAX, EAX
00401192	. C3	RETN

위 코드를 F8 진행하면서 확인해보아야 할 부분은 레지스터창과 상태창이다. RETN 전 코드에 브레이크포인트를 설정하고 CTRL+F8 로 실행을 해보면 레지스터창에서 4글자씩 감소하는 것을 확인할 수 있고 상태창에서는 현재 비교할 문자들을 보여주는 것을 확인할 수 있다.

이런 반복문이나 기타 여러 함수들에 대한 어셈블리어 코드들은 자주 보면서 눈에 익숙하게 하는 것이 가장 좋은 방법이다. 자주 이런 코드들을 접하다 보면 자연스럽게 코드들이 눈에 들어 오게 된다.

**Example #3. while**

```
#include <stdio.h>
#include <conio.h>

int main(void) {
    char ch;
    ch = getche();

    while(ch != 'q')
    {
        ch=getche();
    }
    printf("found the q");
    return 0;
}
```

getche 함수에 의해서 사용자의 입력을 받아 들이고 ‘q’라는 글자가 입력이 되면 “found the q”라는 메시지를 출력하면서 끝난다. while 에 의해서 생성된 어셈코드는 아래와 같다.

Address	Hex dump	Disassembly
0040102D	. 8845 FC	MOV BYTE PTR SS:[EBP-4], AL
00401030	> 0FBE45 FC	MOVSX EAX, BYTE PTR SS:[EBP-4]
00401034	. 83F8 71	CMP EAX, 71
00401037	.~ 74 0A	JE SHORT while.00401043
00401039	. E8 52A80000	CALL while._getche
0040103E	. 8845 FC	MOV BYTE PTR SS:[EBP-4], AL
00401041	.^ EB ED	JMP SHORT while.00401030

00401034에서 비교를 하고 같을 경우 00401043으로 분기하여 루프문을 빠져나오지만 그렇지 않을 경우 계속 입력을 받아 들인다. Hex dump 부분에 보이는 v 표시와 ^ 표시, 그리고 > 표시를 잘 보도록 하자. 빨간색으로 화살표가 그려질 경우 분기를 한다는 것이고 회색으로 표시되면 분기되지 않는다는 것이다.

## 2-8. 특정 루틴에서 사용되는 API 함수

### 특정 루틴에서 사용되는 API 함수



- 파일 및 디렉토리 관련
  - CreateFile, ReadFile, WriteFile, SetFilePointer, CopyFile, GetFileAttribute, SetFileAttribute, FindFirstFile, FindNextFile, GetModuleFileName, GetSystemDirectory, GetWindowsDirectory, GetCommandLine, SetCurrentDirectory
- 레지스트리 관련
  - RegCreateKey, RegOpenKeyEx, RegSetValueEx, RegQueryValueEx
- 네트워크 관련
  - WSASStartup, WSAAPI socket, recv, send, listen, accept, gethostbyname, ntohs, inet\_addr, WNetOpenEnum, WNetEnumResource, InternetGetConnectedState, ioctlsocket
- 메모리 관련
  - RtlZeroMemory, GlobalAlloc
- 기타
  - ShellExecute, GetProcAddress, CreateThread

### Student Notes

윈도우 환경에서 exe 파일을 분석할 경우 API 함수들을 많이 알고 있는 것이 분석에 도움이 된다. 디버거를 사용하여 현재 프로그램에서 사용되는 API 함수들을 나열할 수 있는데 함수들의 기능이나 리턴값을 알아야만 프로그램을 디버깅 하는 과정에서 브레이크포인트를 지정할 곳을 쉽게 찾아낼 수 있다. 이번 장에서는 악성코드에서 자주 사용되거나 필히 알아 두어야 할 API 함수들에 대해서 살펴보도록 하겠다.

파일, 디렉토리 관련 함수, 레지스트리 관련 함수, 네트워크 관련 함수, 메모리 관련 함수들로 구분 지어서 보도록 하겠고 기타 함수들에 대해서도 간단히 살펴보도록 하겠다.

## 파일 및 디렉토리 관련 함수

**CreateFile** : 파일을 생성하거나 연다.

**ReadFile** : 파일의 내용을 읽는다.

**WriteFile** : 파일에 내용을 쓴다.

**SetFilePointer** : 파일 포인터를 이동시킨다.

**CopyFile** : 파일을 복사한다.

**GetFileAttribute** : 파일이나 디렉토리의 속성을 저장한다.

**SetFileAttribute** : 파일이나 디렉토리의 속성을 설정한다.

**FindFirstFile** : 디렉토리에서 파일을 찾는다.

**FindNextFile** : 파일 검색을 계속한다. FindFirstFile 함수의 호출에 이어서 검색을 계속한다.

**GetModuleFileName** : 파일의 절대 경로를 저장한다.

**GetSystemDirectory** : 윈도우가 설치된 디렉토리를 저장한다.

**GetWindowsDirectory** : 윈도우가 설치된 디렉토리를 알아낸다.

**GetCommandLine** : 현재 프로세스의 commandline 문자열을 저장한다.

**SetCurrentDirectory** : 현재 프로세스에 대해 현재 디렉토리를 바꾼다.

## 레지스트리 관련 함수

**RegCreateKey** : 레지스트리키를 생성하거나 연다.

**RegOpenKeyEx** : 레지스트리키를 연다.

**RegSetValueEx** : 레지스트리 값은 설정한다.

**RegQueryValueEx** : 타입과 데이터를 저장한다.

## 네트워크 관련 함수

**WSAStartup** : WS2\_32.DLL 을 사용하기 위해 초기화한다.

**socket** : 소켓을 생성한다.

**recv** : 연결되거나 바운드된 소켓으로부터 데이터를 받는다.

**send** : 연결된 소켓으로 데이터를 보낸다.

**listen** : 연결 요청이 들어오는 것을 기다린다.

**accept** : 소켓을 통해 들어오는 연결을 허용한다.

**gethostbyname** : 호스트 정보를 저장한다.

## Reverse Engineering

**ntohs** : network byte order 를 host byte order 로 변환한다.

**inet\_addr** : IN\_ADDR 구조체로 주소를 변환한다.

**WNetOpenEnum** : 열거 할 네트워크 자원에 대해 목록화한다.

**WNetEnumResource** : **WNetOpenEnum** 에서 나열된 데이터를 반복적으로 가져온다.

**InternetGetConnectedState** : 로컬시스템의 연결 상태를 저장한다.

**ioctlsocket** : 소켓의 입출력 모드를 제어한다.

## 메모리 관련 함수

**RtlZeroMemory** : 한 메모리 블록을 0 으로 채운다.

**GlobalAlloc** : Heap 메모리를 할당한다.

## 기타 함수

**ShellExecute** : 프로그램 내에서 다른 프로그램을 실행시키기 위해 사용된다.

**GetProcAddress** : DLL로부터 익스포트된 함수의 주소를 저장

**CreateThread** : 실행 할 쓰레드를 생성한다.

이 외에도 여러 가지 함수들을 알고 있어야 하며 함수들에 대한 자세한 내용은 WIN32.HLP 파일을 참조하면 된다.

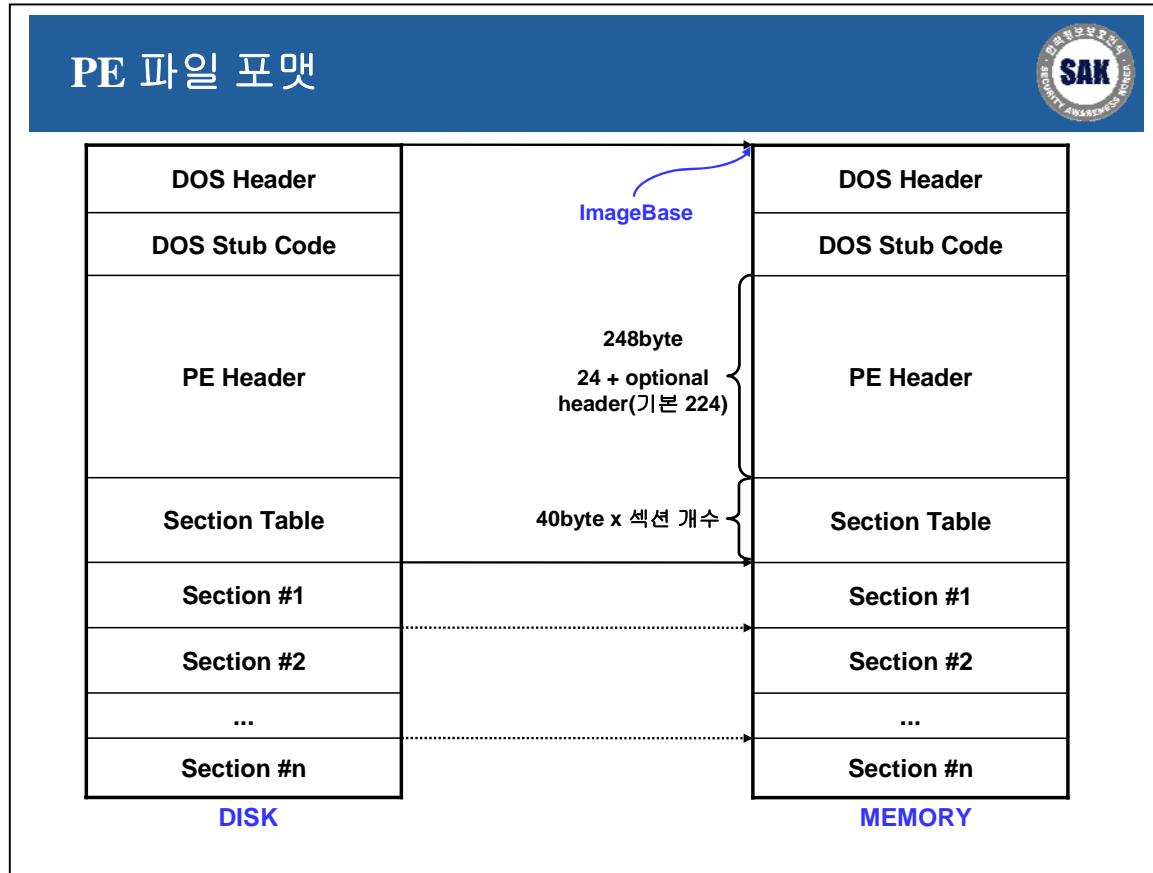
---

## Module 3 — PE 파일 구조

### Objectives

- PE 파일 포맷
- DOS 헤더 및 DOS Stub Code
- PE File Header
- Optional Header
- Section Tables
- Import Table
- Export Table

### 3-1. PE 파일 포맷



### Student Notes

PE 파일 형식(Portable Executable File Format)이란 파일(File)에 담겨 다른 곳에 옮겨져(Portable)도 실행 시킬 수 있도록(Executable) 규정한 형식(Format)이란 뜻이다. Win32 의 기본적인 파일 형식이며 윈도우 운영체제에서 실행되는 프로그램이 모두 PE 파일 형식을 가지고 있다. 이런 실행 프로그램 또는 응용프로그램은 EXE 파일 확장자를 가지는데 이런 EXE 파일이 PE 파일의 대표적인 예이다. 그리고 동적 링크 라이브러리인 DLL 파일도 PE 파일 형식을 가지고 있다.

PE 파일은 운영체제와 상관없이 Win32 플랫폼에서는 공통으로 사용할 수 있다. 즉, 인텔 CPU를 사용하지 않는 경우에도 어떠한 Win32 플랫폼의 PE로더도 이 파일 형식을 사용할 수 있다는 것이다.

Win32 Platform SDK 의 Winnt.h 헤더 파일에 보면 PE 관련 구조체들이 선언되어 있다. 이 구조체에서 파

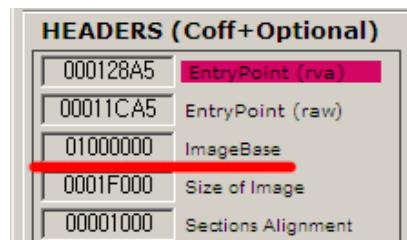
일이라는 명칭 대신 이미지(Image)라는 단어를 사용하는데 이것은 PE 파일이 하드디스크에 파일로 존재하지만 실행되기 위해 메모리로 로드되기 때문이다. 이번 모듈을 학습하는 과정에서 꼭 기억해야 할 것은 “PE 파일은 디스크에서의 모습과 메모리에서의 모습이 거의 같다”는 것이다.

## PE 구성 요소

- DOS Header
- DOS Stub Code
- PE Header
- Section Table
- Section

### DOS Header

디스크상에서 PE 파일은 DOS 헤더로 시작한다. 메모리상에서는 ImageBase에서 시작하기 때문에 DOS 헤더는 ImageBase에서 찾을 수 있다. 파일의 정보는 Stub\_PE 프로그램을 이용하였다.



파일상의 가장 첫 부분과 메모리상에서는 ImageBase의 위치인 0x01000000과 일치하는 것을 확인할 수 있다.

Address	Hex dump	ASCII
01000000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ?L...J...
01000010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	?.....@.....
01000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
01000030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00	.....?..

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ?.....
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	?.....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	E8	00	00	00	.....?..

DOS 헤더는 **4D 5A(MZ)**로 시작한다. MZ는 DOS 개발자인 Mark Zbikowski라는 사람의 이니셜로 DOS 시그내처로 사용된다.

## Reverse Engineering

### PE Header

PE 헤더는 PE 파일의 정보를 가지고 있다. 섹션의 개수나 파일의 속성, optional 헤더의 크기를 비롯하여 ImageBase, 엔트리포인트 주소, 메모리상에 각 섹션이 차지하는 크기, 디스크상에서의 섹션 정렬, PE 파일의 총 사이즈, 디스크 상에서의 헤더의 총 사이즈 등과 같은 정보를 가지고 있다.

PE 헤더의 위치는 DOS 헤더에 있는 **e\_lfanew** 값을 이용하여 확인할 수 있다. **e\_lfanew**는 DOS 헤더의 마지막 4byte에 위치하고 있다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ?.....
00000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	?.....@.....
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
00000030	00	00	00	00	00	00	00	00	00	00	00	00	E8	00	00	00	.....?..
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..?.???.L?Th
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....\$.....
00000080	36	86	40	D7	72	E7	2E	84	72	E7	2E	84	72	E7	2E	84	6????.??.?
00000090	F1	EF	21	84	77	E7	2E	84	F1	EF	73	84	7D	E7	2E	84	標!?.??.???
000000A0	72	E7	2F	84	FE	E7	2E	84	FC	EF	71	84	7D	E7	2E	84	r?.??.??.???
000000B0	FC	EF	4E	84	6E	E7	2E	84	F1	EF	70	84	73	E7	2E	84	標N?.??.??.???
000000C0	F1	EF	74	84	73	E7	2E	84	52	69	63	68	72	E7	2E	84	標t?.??.??.ichr?
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000E0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	03	00	.....PE..L..
000000F0	DA	00	80	3E	00	00	00	00	00	00	00	00	E0	00	0F	01	?>.....?..

### Section Table

섹션 테이블은 섹션의 이름, 섹션의 파일상에서의 위치 및 사이즈, 메모리상에서의 위치 및 사이즈, 속성 값에 대한 정보를 가지고 있다. 섹션 테이블은 PE 헤더 바로 뒤에 위치하며 섹션 테이블의 위치는 PE 헤더 시작 주소에서 PE 헤더의 사이즈를 더해주면 된다. PE 헤더는 기본적으로 248bytes 의 크기를 가지면 경우에 따라서는 변할 수 있다.

섹션 테이블은 각 섹션별로 40bytes 의 크기를 갖는다. 따라서 섹션의 개수가 4개일 경우 섹션 테이블의 크기는  $40\text{bytes} * 4 = 160\text{bytes}$  가 된다.

### Section

섹션은 실제 데이터가 위치하는 공간이다. 각 섹션의 위치는 섹션 테이블에 있는 섹션 헤더에서 확인할 수 있다. 섹션 헤더에는 VirtualAddress 값과 PointerToRawData라는 값이 저장되어 있다. 둘 다 offset 값

## Reverse Engineering

으로 메모리상에서의 위치와 디스크 상에서의 위치를 가리키는 값이다. 이렇게 두 개의 값이 별도로 존재하는 이유는 디스크 상에서의 위치와 메모리상에서의 위치가 다를 수 있다는 의미이다.

d:\tools\wrce\01_rce_example\calc.exe						
	Headers	Dos	Sections	Functions	Resources	Signature
No	Name	VirtualSize	VirtualOffset	RawSize	RawOffset	Characteristics
01	.text	00012B86	00001000	00012C00	00000400	60000020
02	.data	00001034	00014000	00000A00	00013000	C0000040
03	.rsrc	00008780	00016000	00008800	00013A00	40000040

.text 섹션의 디스크상에서의 offset이 0x400임을 확인할 수 있고 .text 섹션이 파일의 시작점부터 0x400만큼 떨어진 곳에 있다는 것을 의미한다. Winhex에서 0x400의 위치를 확인하면 다음과 같다.

offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000003D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000003E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000003F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000400	49	3A	D3	77	62	3C	D3	77	0C	2D	D3	77	00	00	00	00
00000410	7C	18	BA	77	2C	19	BA	77	CC	17	BA	77	00	00	00	00
00000420	80	16	DE	77	85	B0	DE	77	AA	B0	DE	77	20	AF	DE	77
00000430	FD	B0	DE	77	7A	96	DF	77	82	98	DF	77	53	97	DF	77
00000440	5A	1F	E0	77	E3	16	DE	77	4D	3F	DE	77	C2	9A	DF	77
00000450	82	95	DF	77	83	1C	DE	77	4C	3C	E3	77	FB	2D	DE	77

위 정보를 통해 섹션 헤더의 PointerToRawData 값은 해당 섹션의 파일상에서의 시작 위치를 확인하였다. 이번에는 메모리상에서는 어디에 위치하는지 확인해보도록 하겠다.

Address	Hex dump	Data	Comment
010001E0	2E 74 65 71	ASCII ".text"	SECTION
010001E8	B62B0100	DD 00012B86	VirtualSize = 12B86 (76726.)
010001EC	00100000	DD 00001000	VirtualAddress = 1000
010001F0	002C0100	DD 00012C00	SizeOfRawData = 12C00 (76800.)
010001F4	00040000	DD 00000400	PointerToRawData = 400
010001F8	00000000	DD 00000000	PointerToRelocations = 0
010001FC	00000000	DD 00000000	PointerToLineNumbers = 0
01000200	0000	DW 0000	NumberOfRelocations = 0
01000202	0000	DW 0000	NumberOfLineNumbers = 0
01000204	20000060	DD 60000020	Characteristics = CODE EXECUTE READ

메모리상에서의 .text 섹션의 위치는 VirtualAddress에서 확인할 수 있다. VirtualAddress의 값이 0x1000인 것을 확인할 수 있고 이 값은 ImageBase로 부터의 offset 값이다. 이런 offset을 RVA(Relative Virtual Address)라고 부른다. 따라서 .text 섹션의 메모리상에서의 위치는 ImageBase와 VirtualAddress을 합한 값인 0x01001000이 된다. 0x01001000의 위치로 이동하면 다음과 같은 HEX 값을 확인할 수 있다.

# Reverse Engineering

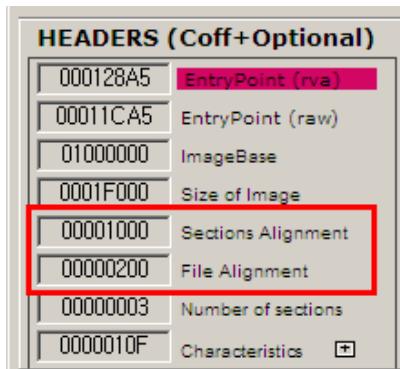
Address	Hex dump	ASCII	
01001000	FA F4 F3 77 86 E0 F3 77 CE 6C F4 77 00 00 00 00	缺?匁???.....	
01001010	81 5F BA 77 39 61 BA 77 21 5E BA 77 00 00 00 00	? <sup>9</sup> a <sup>1</sup> ^ <sup>8</sup> ....	
01001020	7B 1C 82 7C 11 23 82 7C A0 B0 81 7C 0F 51 82 7C	{? <sup>?</sup> ? <sup>?</sup> ?Q?	
01001030	31 98 82 7C 11 21 81 7C 82 27 81 7C A9 21 81 7C	1缺! <sup>1</sup> ?????	
01001040	D5 ED 80 7C DE 24 80 7C AB EF 82 7C F9 23 81 7C	蘇口?D <sup>1</sup> ワ???	
01001050	DA CB 81 7C 6F 3E 82 7C 72 17 86 7C 7A 3D 82 7C	密?o>?-?z=?	
01001060	4A 47 82 7C 77 95 82 7C D1 19 82 7C C7 2F 82 7C	JG?w <sup>8</sup> !????	
01001070	78 3C 82 7C BA 1F 80 7C 04 20 80 7C 9D 2F 82 7C	x<??>J <sup>1</sup> 口??	
01001080	60 20 84 7C C9 22 82 7C F7 F0 81 7C 19 24 82 7C	' ???抱? \$?	

파일상에서 .text 섹션의 시작점의 데이터는 493AD377 이었는데 메모리상에서 .text 섹션의 시작점 데이터는 FAF4F377 로 서로 다른 값인 것을 확인할 수 있다. 이렇게 파일상에서의 섹션의 내용과 메모리상에서의 섹션의 내용이 다를 수 있다는 것을 확인하였다.

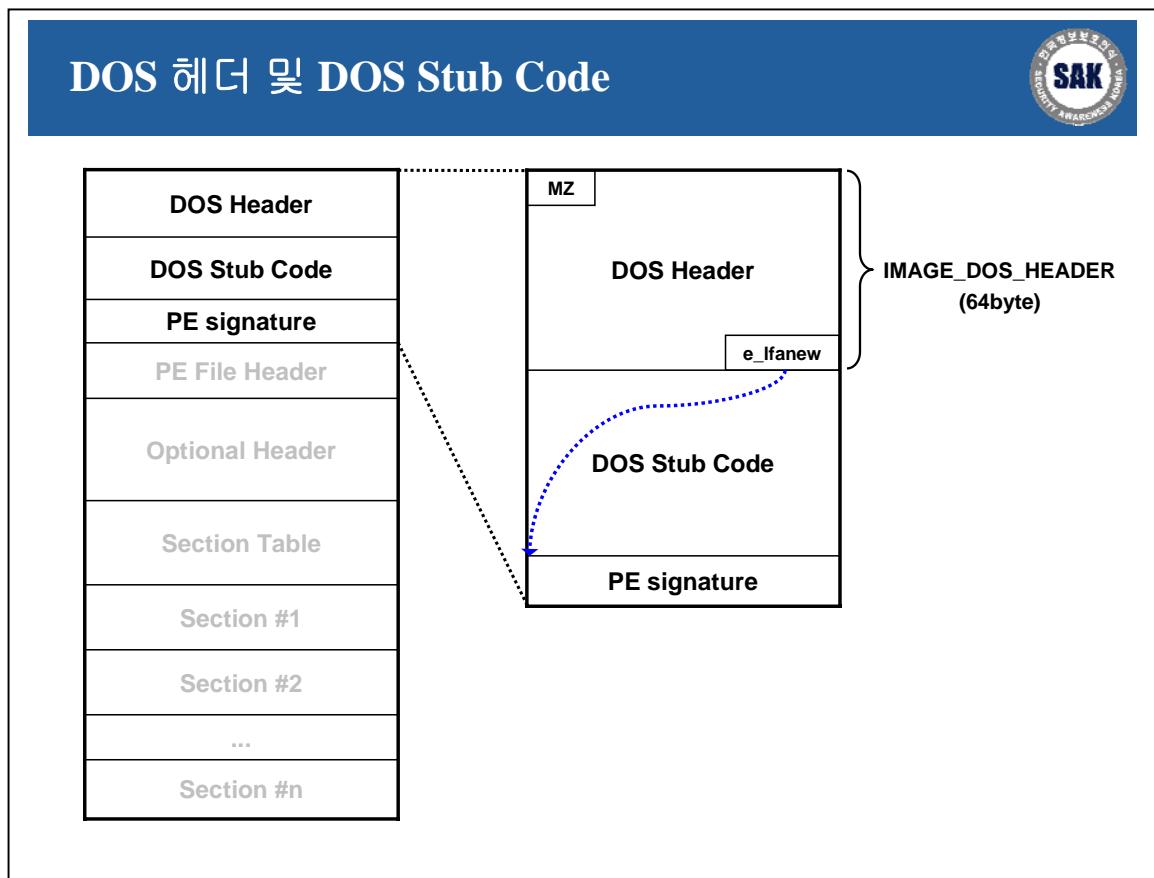
PE 파일은 메모리상에서와 파일상에서 거의 같다는 것을 확인했다. 차이가 발생하는 부분은 섹션인데 이것은 SectionAlignment 와 FileAlignment 와 연관이 있다. 이것은 디스크상에서의 정렬 단위와 메모리 상에서의 정렬 단위가 다르다는 것을 의미한다.

정렬 단위는 뒤에서 나오는 Optional Header 에서 자세히 보도록 하고 간단하게 두 가지만 알아보겠다.

디스크상의 섹션은 **FileAlignment**의 배수가 되는 주소에서 시작  
메모리상의 섹션은 **SectionAlignment**의 배수가 되는 주소에서 시작



### 3-2. DOS 헤더 및 DOS Stub Code



### Student Notes

DOS 헤더는 DOS 와 호환을 위해 사용되는 헤더이다. PE 파일은 DOS 헤더로 시작하고 DOS 헤더는 항상 64bytes 의 크기를 갖는다. 우리가 알아야 할 부분은 가장 처음 2byte 를 차지하는 **e\_magic** 과 마지막 4byte 를 차지하는 **e\_lfanew** 이다. Winnt.h 에 선언되어 있는 IMAGE\_DOS\_HEADER 는 다음과 같다.

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD   e_magic;           // Magic number
    WORD   e_cblp;            // Bytes on last page of file
    WORD   e_cp;              // Pages in file
    WORD   e_crlc;             // Relocations
    WORD   e_cparhdr;          // Size of header in paragraphs
    WORD   e_minalloc;          // Minimum extra paragraphs needed
    WORD   e_maxalloc;          // Maximum extra paragraphs needed
```

## Reverse Engineering

```

WORD    e_ss;           // Initial (relative) SS value
WORD    e_sp;           // Initial SP value
WORD    e_csum;         // Checksum
WORD    e_ip;           // Initial IP value
WORD    e_cs;           // Initial (relative) CS value
WORD    e_lfarlc;       // File address of relocation table
WORD    e_ovno;         // Overlay number
WORD    e_res[4];        // Reserved words
WORD    e_oemid;         // OEM identifier (for e_oeminfo)
WORD    e_oeminfo;       // OEM information; e_oemid specific
WORD    e_res2[10];       // Reserved words
LONG   e_lfanew;        // File address of new exe header
} IMAGE_DOS_HEADER, *PI IMAGE_DOS_HEADER;

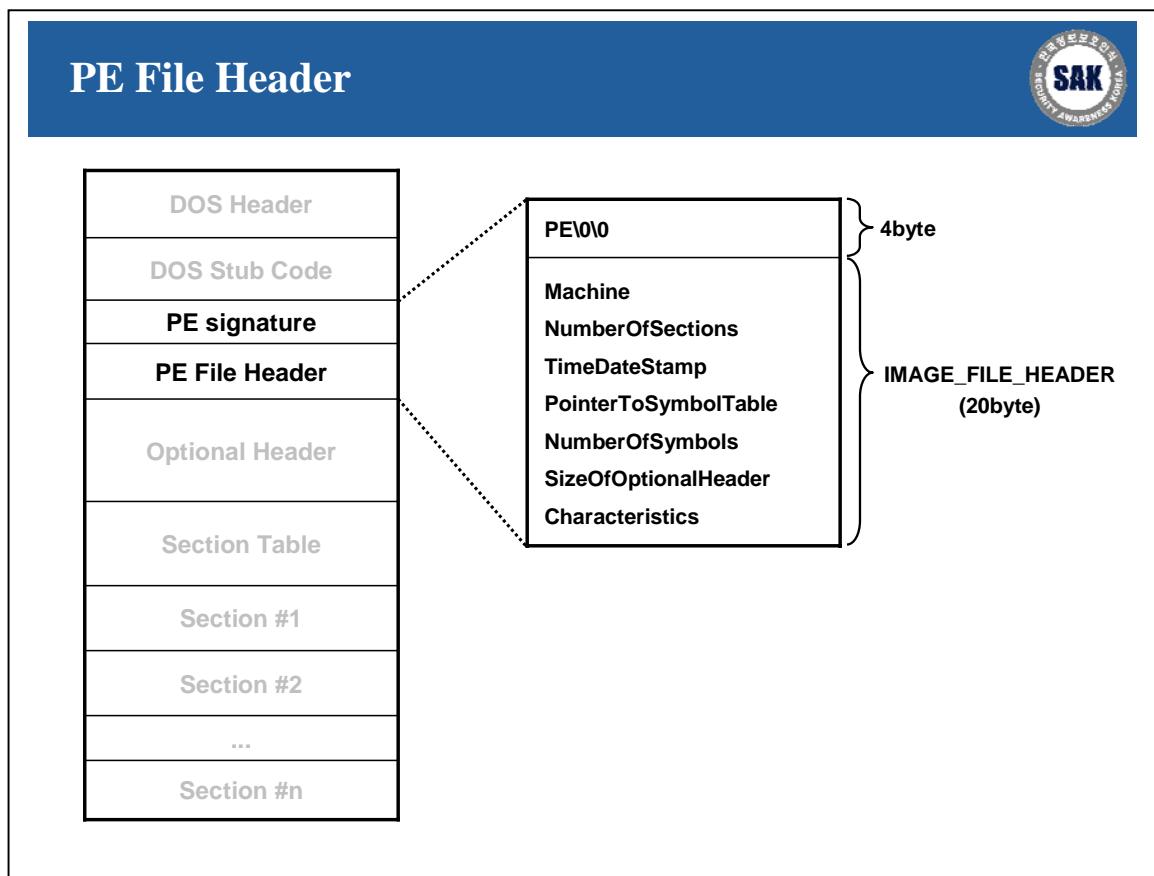
```

**e\_magic**은 DOS 헤더의 시그내처로 4D 5A(MZ) 값을 가진다. **e\_lfanew**는 PE 헤더의 시작점을 가리키는 오프셋 값이다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
000000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ?.....
000000010	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00	?.....@.....
000000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
000000030	00	00	00	00	00	00	00	00	00	00	00	00	E8	00	00	00	.....?..
000000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..?..?..L?Thl.SHE
000000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F	is program canno
000000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20	t be run in DOS
000000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00	mode....\$.....
000000080	36	86	40	D7	72	E7	2E	84	72	E7	2E	84	72	E7	2E	84	6???FIXME?.....
000000090	F1	EF	21	84	77	E7	2E	84	F1	EF	73	84	7D	E7	2E	84	標!FIXME?.....
0000000A0	72	E7	2F	84	FE	E7	2E	84	FC	EF	71	84	7D	E7	2E	84	r?FIXME?.....
0000000B0	FC	EF	4E	84	6E	E7	2E	84	F1	EF	70	84	73	E7	2E	84	註NFIXME?.....
0000000C0	F1	EF	74	84	73	E7	2E	84	52	69	63	68	72	E7	2E	84	標tFIXME?.....
0000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....
0000000E0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	03	00	.....PE...L...

DOS Stub Code는 크게 중요하지 않는 부분이다. 이 부분이 없더라도 프로그램이 실행되는 데는 아무런 문제가 없다. 이 부분에 있는 내용은 윈도우용 응용프로그램을 도스 모드에서 실행시켰을 경우 “This program cannot be run in DOS mode”라는 메시지를 출력하고 프로그램이 종료되는 코드가 삽입되어 있다. 그리고 응용프로그램을 제작할 경우 오브젝트 파일을 링킹할 때 STUB 옵션을 사용하여 원하는 스텝 코드를 삽입할 수도 있다.

### 3-3. PE File Header



### Student Notes

Winnt.h에 선언되어 있는 IMAGE\_NT\_HEADERS는 다음과 같다.

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Si gnature;
    IMAGE_F ILE_HEADER Fi l eHeader;
    IMAGE_OPTI ONAL_HEADER32 Opti onal Header;
} I MAGE_N T_HEADERS32, *PI MAGE_N T_HEADERS32;
```

IMAGE\_NT\_HEADERS에는 PE 시그내처와 IMAGE\_FILE\_HEADER, IMAGE\_OPTIONAL\_HEADER로 구성되어 있다. 첫 번째로 PE 시그내처는 “50 45 00 00”으로 고정되어 있고 두 번째 나오는 IMAGE\_FILE\_HEADER도 20bytes의 고정 사이즈를 갖는다. 마지막에 오는 IMAGE\_OPTIONAL\_HEADER는 224bytes의 사이즈를 갖지만 원칙적으로는 가변 사이즈이다. 이는

## Reverse Engineering

Data Directory 때문인데 Data Directory는 Optional 헤더 부분에서 보도록 하겠다.

Winnt.h에 선언되어 있는 IMAGE\_FILE\_HEADER는 다음과 같다.

```
typedef struct _IMAGE_FILE_HEADER {
    WORD      Machine;
    WORD      NumberOfSections;
    DWORD     TimeDateStamp;
    DWORD     PointerToSymbolTable;
    DWORD     NumberOfSymbols;
    WORD      SizeOfOptionalHeader;
    WORD      Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Winhex에서 보면 다음과 같다.

Offset	0 1 2 3 4 5 6 7 8 9 A B C D E F	
00000000	4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00	MZ?..... .
00000010	B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00	?.....@.....
00000020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000030	00 00 00 00 00 00 00 00 00 00 00 00 E8 00 00 00	.....?..?
00000040	0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68	..?.??.L?Th
00000050	69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F	is program canno
00000060	74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20	t be run in DOS
00000070	6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00	mode....\$.....
00000080	36 86 40 D7 72 E7 2E 84 72 E7 2E 84 72 E7 2E 84	6???.??.??.
00000090	F1 EF 21 84 77 E7 2E 84 F1 EF 73 84 7D E7 2E 84	帳!帳?帳??.
000000A0	72 E7 2F 84 FE E7 2E 84 FC EF 71 84 7D E7 2E 84	r??.?.?.?.
000000B0	FC EF 4E 84 6E E7 2E 84 F1 EF 70 84 73 E7 2E 84	轉N帳?帳??.
000000C0	F1 EF 74 84 73 E7 2E 84 52 69 63 68 72 E7 2E 84	帳t帳?帳ichr?
000000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000000E0	00 00 00 00 00 00 00 00 50 45 00 00 4C 01 03 00	.....PE..L..
000000F0	DA 00 80 3E 00 00 00 00 00 00 00 E0 00 0F 01	?D>.....?..
00000100	0B 01 07 0A 00 2C 01 00 00 92 00 00 00 00 00 00	.....,..?..
00000110	A5 28 01 00 00 10 00 00 00 40 01 00 00 00 00 01	?.....@.....
00000120	00 10 00 00 00 02 00 00 05 00 02 00 05 00 02 00	.....

PE 시그내처는 DOS 헤더의 가장 마지막에 있는 **e\_lfanew**가 가리키고 있는 주소에 있다.

File 헤더에서 꼭 알아두어야 할 필드는 4 가지이다.

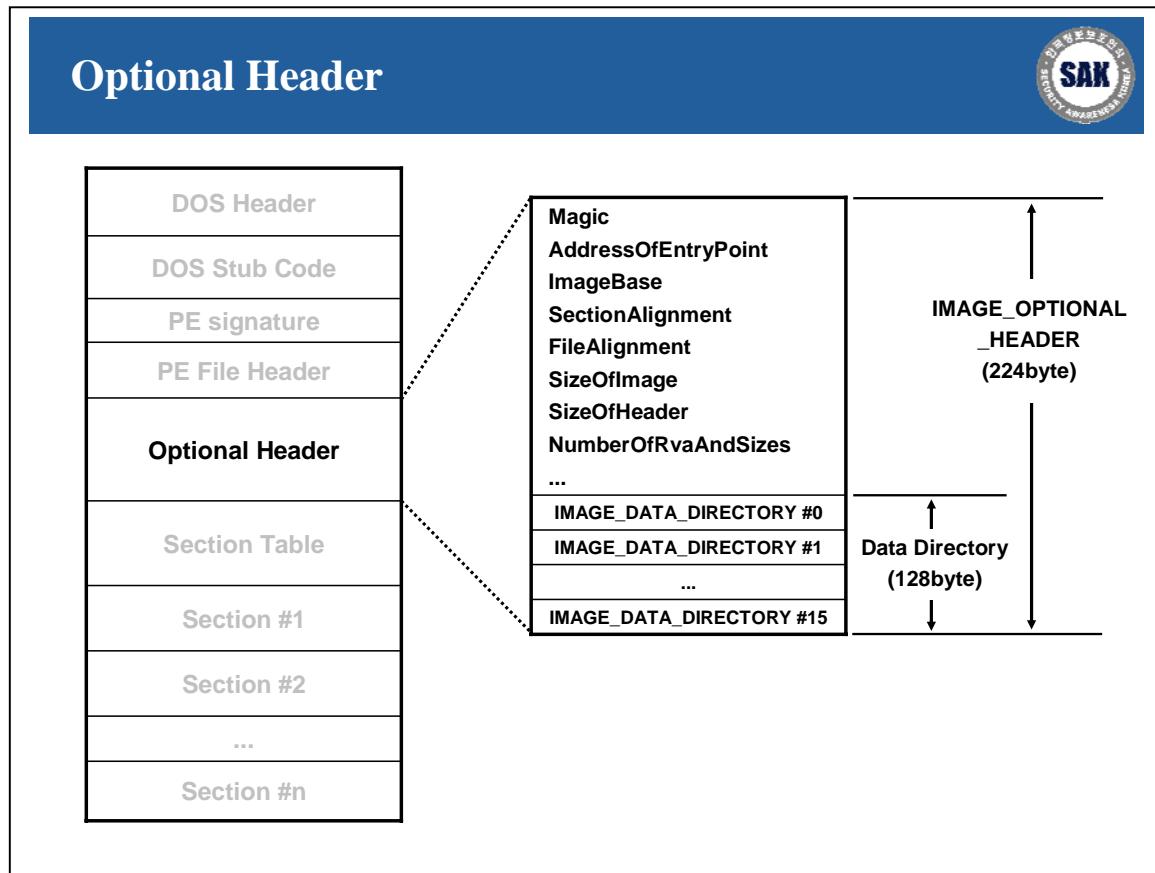
- **Machine** : CPU ID를 나타낸다. IA32의 경우 0x14C가 되고 IA64일 경우 0x200이 되어야 한다.
- **NumberOfSections** : 섹션의 개수를 의미한다.
- **SizeOfOptionalHeader** : 파일 헤더 뒤에 오는 optional 헤더의 크기를 나타낸다. 기본적으로 224bytes의 사이즈를 갖지만 data directory가 생략될 경우 96bytes의 사이즈를 갖게 된다.

- **Characteristics** : 파일의 속성을 나타낸다. 일반적인 실행 파일의 경우 0x10F의 값을 가진다. Winnt.h에 다음과 같은 속성이 정의되어 있다.

#define IMAGE_FILE_RELOCS_STRIPPED	0x0001
#define IMAGE_FILE_EXECUTABLE_IMAGE	0x0002
#define IMAGE_FILE_LINE_NUMS_STRIPPED	0x0004
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED	0x0008
#define IMAGE_FILE.Aggressive_WS_TRIM	0x0010
#define IMAGE_FILE_LARGE_ADDRESS_AWARE	0x0020
#define IMAGE_FILE_BYTES_REVERSED_LO	0x0080
#define IMAGE_FILE_32BIT_MACHINE	0x0100
#define IMAGE_FILE_DEBUG_STRIPPED	0x0200
#define IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP	0x0400
#define IMAGE_FILE_NET_RUN_FROM_SWAP	0x0800
#define IMAGE_FILE_SYSTEM	0x1000
#define IMAGE_FILE_DLL	0x2000
#define IMAGE_FILE_UP_SYSTEM_ONLY	0x4000
#define IMAGE_FILE_BYTES_REVERSED_HI	0x8000

일반적으로 0x10F의 값을 가진다고 했는데 이는 IMAGE\_FILE\_RELOCS\_STRIPPED(0x0001), IMAGE\_FILE\_EXECUTABLE\_IMAGE(0x0002), IMAGE\_FILE\_LINE\_NUMS\_STRIPPED(0x0004), IMAGE\_FILE\_LOCAL\_SYMS\_STRIPPED(0x0008), IMAGE\_FILE\_32BIT\_MACHINE(0x0100)의 속성이 설정되어 있는 것이다.

### 3-4. Optional Header



### Student Notes

Optional 헤더는 PE 파일에서 가장 중요한 부분이라고 할 수 있다. 명칭은 옵션이지만 절대 옵션의 성격을 갖고 있지 않고 프로그램이 메모리상에 로드 되었을 때 시작할 주소라든지 메모리상에서의 정렬 단위와 같이 중요한 정보들을 다수 포함하고 있다. optional 헤더는 30 개의 필드와 1 개의 데이터 딕토리 를 가지고 있다. 하지만 모든 필드를 알아야 할 필요는 없다. 여기서는 중요한 필드에 대해서만 알아보도록 하겠다. 먼저 Winnt.h에 선언되어 있는 IMAGE\_OPTIONAL\_HEADER 를 보도록 하자.

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitialData;
```

```

DWORD  Si ze0fUni ni ti al i zedData;
DWORD  AddressOfEntryPoint;
DWORD  BaseOfCode;
DWORD  BaseOfData;
DWORD  ImageBase;
DWORD  SectionAlignment;
DWORD  FileAlignment;
WORD   MajorOperatingSystemVersion;
WORD   MinorOperatingSystemVersion;
WORD   MajorImageVersion;
WORD   MinorImageVersion;
WORD   MajorSubsystemVersion;
WORD   MinorSubsystemVersion;
DWORD  Win32VersionValue;
DWORD  SizeOfImage;
DWORD  SizeOfHeaders;
DWORD  CheckSum;
WORD   Subsystem;
WORD   DLLCharacteristics;
DWORD  SizeOfStackReserve;
DWORD  SizeOfStackCommit;
DWORD  SizeOfHeapReserve;
DWORD  SizeOfHeapCommit;
DWORD  LoaderFlags;
DWORD  NumberOfRvaAndSizes;
IMAGE_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

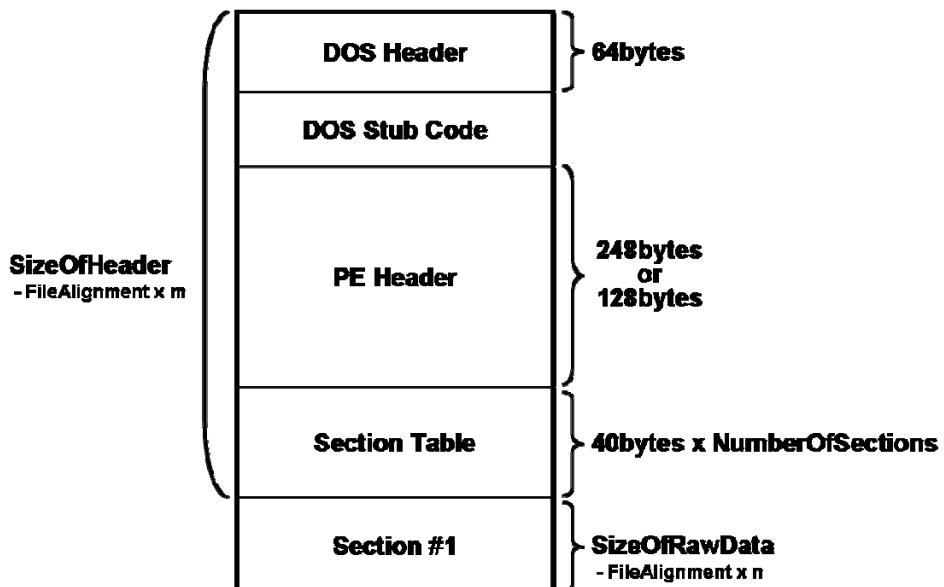
```

Winhex로 보면 다음과 같다.

Offset	0 1 2 3 4 5 6 7 8 9 A B C D E F	
000000C0	F1 EF 74 84 73 E7 2E 84 52 69 63 68 72 E7 2E 84	標幟? ? ichr? ...
000000D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000000E0	00 00 00 00 00 00 00 00 50 45 00 00 4C 01 03 00	..... PE...L...
000000F0	DA 00 80 3E 00 00 00 00 00 00 00 00 E0 00 0F 01	?>.....?
00000100	0B 01 07 0A 00 2C 01 00 00 92 00 00 00 00 00 00	.....,....?
00000110	A5 28 01 00 00 10 00 00 00 40 01 00 00 00 00 01	?.....@.....
00000120	00 10 00 00 00 02 00 00 05 00 02 00 05 00 02 00	.....
00000130	04 00 00 00 00 00 00 00 00 F0 01 00 00 04 00 00	.....
00000140	B1 D9 01 00 02 00 00 80 00 00 04 00 00 20 00 00	.....
00000150	00 00 10 00 00 10 00 00 00 00 00 00 10 00 00 00	.....
00000160	00 00 00 00 00 00 00 00 D4 2F 01 00 8C 00 00 00	.....?..?..?
00000170	00 60 01 00 80 87 00 00 00 00 00 00 00 00 00 00	.`..?.....
00000180	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
00000190	40 12 00 00 1C 00 00 00 00 00 00 00 00 00 00 00	@.....
000001A0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000001B0	48 16 00 00 40 00 00 00 58 02 00 00 80 00 00 00	H...@...X...D...
000001C0	00 10 00 00 3C 02 00 00 00 00 00 00 00 00 00 00	....<.....
000001D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....
000001E0	2E 74 65 78 74 00 00 00 B6 2B 01 00 00 10 00 00	.text...?.....
000001F0	00 2C 01 00 00 04 00 00 00 00 00 00 00 00 00 00	,.....
00000200	00 00 00 00 20 00 00 60 2E 64 61 74 61 00 00 00	.... ..`..data...

## Reverse Engineering

- **Machine** : optional 헤더의 시작 위치에 존재하는 필드로 optional 헤더를 구분하는 시그내처로 사용된다. 0x10B로 고정되어 있다.
- **AddressOfEntryPoint** : 엔트리포인트는 PE 파일이 메모리에 로드된 후 맨 처음으로 실행되는 코드의 주소를 가지고 있다. 이 부분에 지정된 주소값은 가상 주소가 아닌 RVA 값이다. 즉, ImageBase에서부터의 오프셋이다.
- **ImageBase** : PE 파일이 로더에 의해서 로드되는 위치이다. EXE 파일의 경우 가상 메모리 공간에 가장 처음 로드되므로 항상 ImageBase에 로드되지만 DLL의 경우 ImageBase로 지정된 주소 공간이 다른 모듈에 의해서 이미 사용 중인 상황이 발생할 수 있다. 이런 경우 해당 DLL을 다른 곳에 로드하고 재배치 작업을 수행하게 된다. 대부분의 링커는 EXE 파일일 경우 0x00400000으로, DLL일 경우 0x10000000으로 설정한다.
- **SectionAlignment** : 각 섹션이 메모리상에서 차지하는 최소 단위이다. 각 섹션의 시작주소는 언제나 이 SectionAlignment에서 지정된 값의 배수가 되어야 한다. 디폴트값은 0x1000(4096byte)이다.
- **FileAlignment** : 디스크상에서 섹션이 차지하는 최소 단위이다. 각 섹션의 시작주소는 FileAlignment 필드에 지정된 값의 배수가 되어야 한다. 디폴트값은 0x200(512byte)이고 2의 n승의 형태의 값을 사용해야 한다.
- **SizeOfImage** : 메모리상에 로드된 PE 파일의 총 사이즈이고 SectionAlignment의 배수가 되어야 한다.
- **SizeOfHeader** : 디스크상에서의 헤더의 총 사이즈이고 FileAlignment의 배수가 되어야 한다.



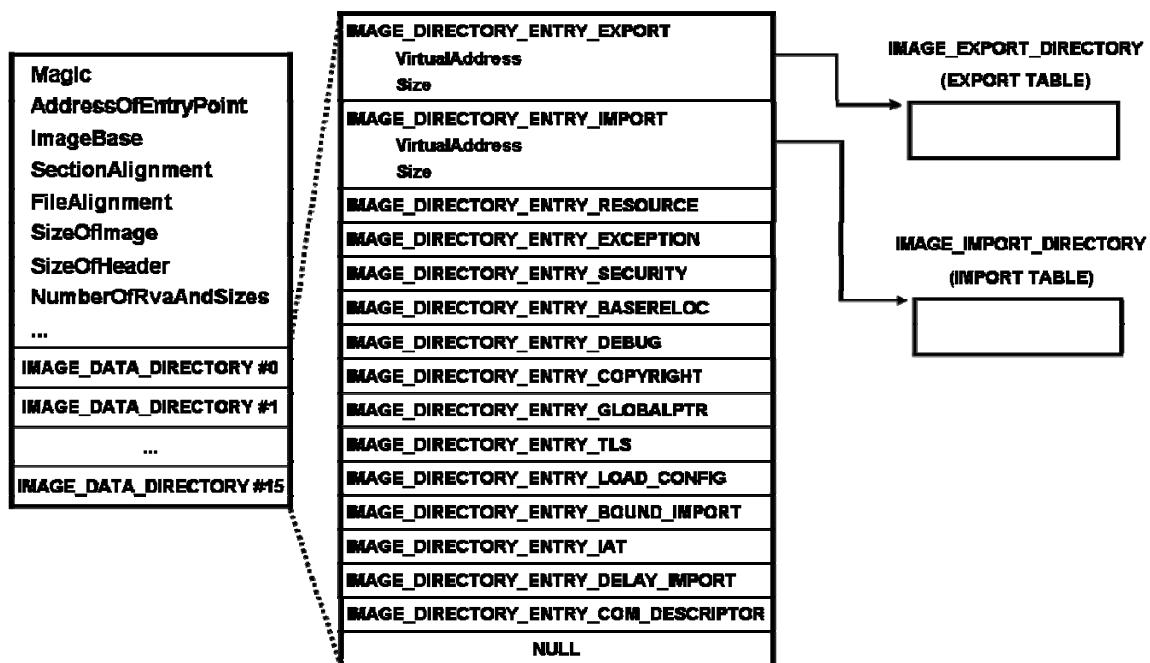
## Data Directory

데이터 딕렉토리는 Optional 헤더 다음에 오는 128bytes 사이즈의 IMAGE\_DATA\_DIRECTORY 구조체의 배열이다. Winnt.h에 다음과 같이 선언되어 있다.

```
typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES 16
```

마지막에 정의된 이 배열의 엔트리의 개수는 Optional 헤더의 마지막 필드인 NumberOfRvaAndSize에 정의되어 있던 값과 같은 값으로 총 16개의 엔트리를 가지고 있다. 데이터 딕렉토리의 각 엘리먼트들은 Export table, Import table 등 PE 파일에서 중요한 역할을 하는 개체들의 주소(VirtualAddress)와 크기(Size)에 대한 정보를 가지고 있다.



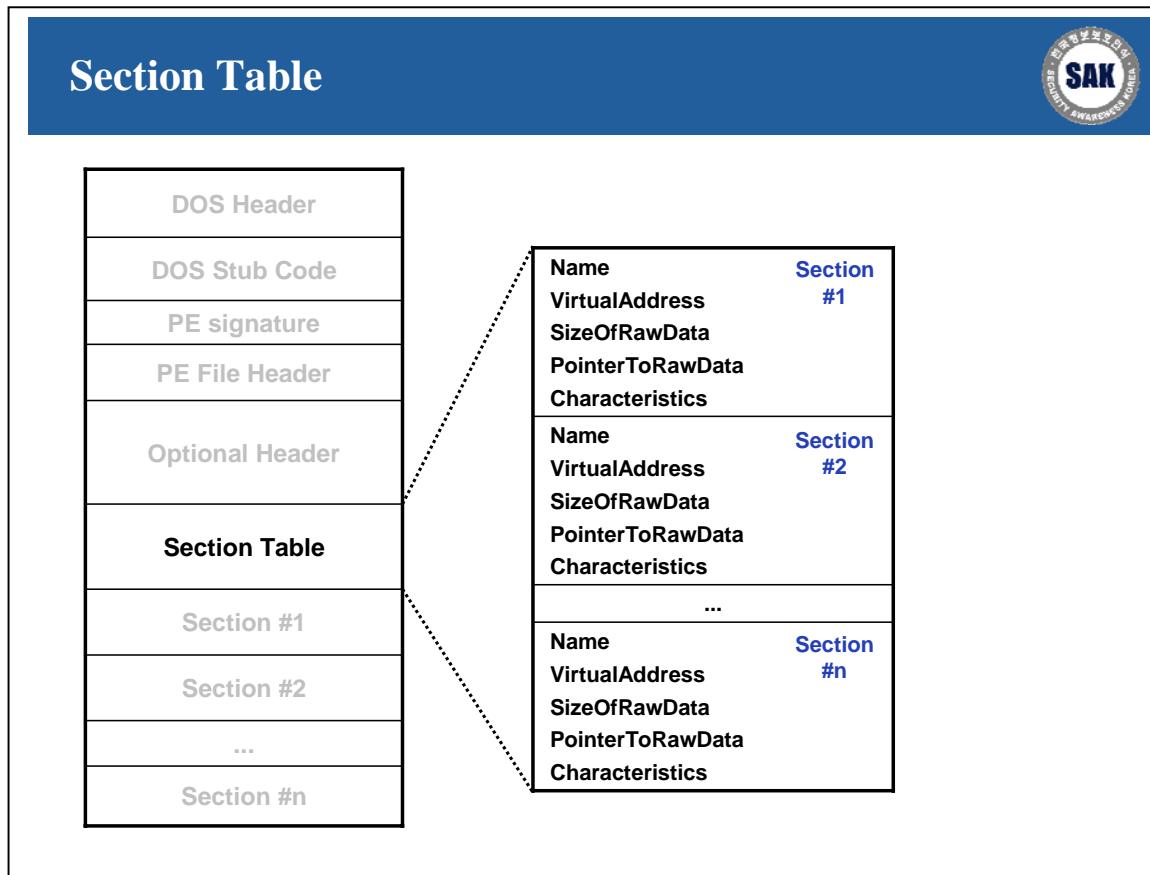
총 16개의 엘리먼트 중 15개가 실제 정보를 가지고 있고 마지막은 0x00으로 채워진 비어 있는 엔트리이다. 15개의 엘리먼트 중 몇 가지 엘리먼트에 대해서만 간단히 알아보도록 하겠다.

- IMAGE\_DIRECTORY\_ENTRY\_EXPORT : EXPORT 테이블의 메모리상에서의 시작점과 크기에 대한 정보를 가지고 있다. EXPORT 테이블은 대부분 DLL에 존재한다.

## **Reverse Engineering**

- IMAGE\_DIRECTORY\_ENTRY\_IMPORT : IMPORT 테이블의 메모리상에서의 시작점과 크기에 대한 정보를 가지고 있다.
- IMAGE\_DIRECTORY\_ENTRY\_BASERELOC : 기준 재배치 정보를 가리킨다. 재배치란 로더가 실행 모듈을 원하는 위치에 위치시키지 못했을 때 코드 상의 포인터 연산과 관련된 주소를 다시 갱신해야 하는 경우를 말한다.
- IMAGE\_DIRECTORY\_ENTRY\_TLS : Thread Local Storage 초기화 섹션에 대한 포인터이다. 앤티 리버싱 기법 중 TLS Callback에서 사용되기 때문에 알아두는 것이 좋다.
- IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT : DLL 바인딩과 관련된 정보를 가지고 있다.
- IMAGE\_DIRECTORY\_ENTRY\_IAT : 첫 번째 임포트 주소 테이블(IAT)의 시작 주소를 가리킨다. 임포트된 각각의 DLL에 대한 IAT는 메모리상에서 연속적으로 나타난다.
- IMAGE\_DIRECTORY\_ENTRY\_DELAY\_IMPORT : 지연 로딩 정보에 대한 포인터이다.

### 3-5. Section Table



### Student Notes

섹션 테이블은 IMAGE\_SECTION\_HEADER 타입의 엘리먼트로 구성된 배열이다. 섹션 헤더는 로더가 각 섹션을 메모리에 로드하고 속성을 설정하는데 필요한 정보들을 가지고 있다.

섹션은 동일한 성질의 데이터가 저장되어 있는 영역이다. 섹션은 윈도우에서 사용하는 메모리 프로텍션 매커니즘과 연관이 있는데 윈도우의 경우 메모리 프로텍션의 최소 단위가 페이지이고 페이지 단위로 여러 속성을 설정해두고 속성에 위배되는 행동을 시도할 때 access violation 을 발생시켜 메모리를 보호한다. 즉, 페이지의 일부는 읽기만 가능하고, 페이지의 일부는 읽고, 쓰기가 가능하도록 설정하는 방식의 프로텍션은 허용하지 않는다는 말이다.

이것은 성질이 다른 데이터들을 하나의 페이지에 담을 수 없다는 것을 의미한다. 그렇다 보니 프로그램

## Reverse Engineering

에 포함된 데이터들 중 읽기와 실행이 가능해야 하는 데이터인 실행코드와 읽고 쓰기가 가능한 데이터, 읽기만 가능한 데이터들을 별도의 페이지에 두어야 하는데 로더의 입장에서는 이를 구분할 방법이 없으므로 섹션이라는 개념을 두어 실행 파일 생성 단계에서 구분해 놓도록 하는 것이다.

Winnt.h에 선언되어 있는 IMAGE\_SECTION\_HEADER는 다음과 같다.

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE      Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD   PhysicalAddress;
        DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLineNumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLineNumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

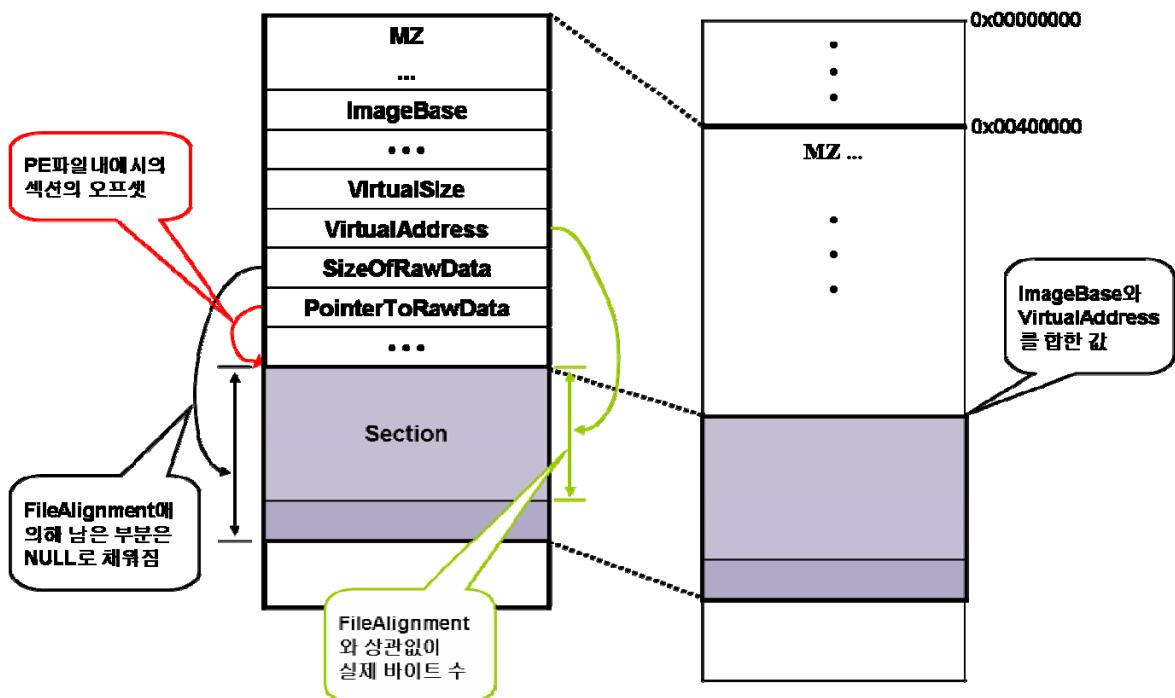
Winhex에서 보면 다음과 같다.

offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000001E0	2E	74	65	78	74	00	00	00	B6	2B	01	00	00	10	00	00
000001F0	00	2C	01	00	00	04	00	00	00	00	00	00	00	00	00	00
00000200	00	00	00	00	20	00	00	60	2E	64	61	74	61	00	00	00
00000210	34	10	00	00	00	40	01	00	00	0A	00	00	00	30	01	00
00000220	00	00	00	00	00	00	00	00	00	00	00	00	00	40	00	00
00000230	00	00	00	00	00	00	00	00	80	87	00	00	00	60	01	00
00000240	2E	72	73	72	63	00	00	00	00	00	00	00	00	00	00	00
00000250	00	88	00	00	00	3A	01	00	00	00	00	00	00	00	00	00
00000260	00	00	00	00	40	00	00	40	98	1F	81	3E	38	00	00	00
00000270	9A	1F	81	3E	45	00	00	00	9A	1F	81	3E	51	00	00	00
00000280	99	1F	81	3E	5C	00	00	00	99	1F	81	3E	69	00	00	00

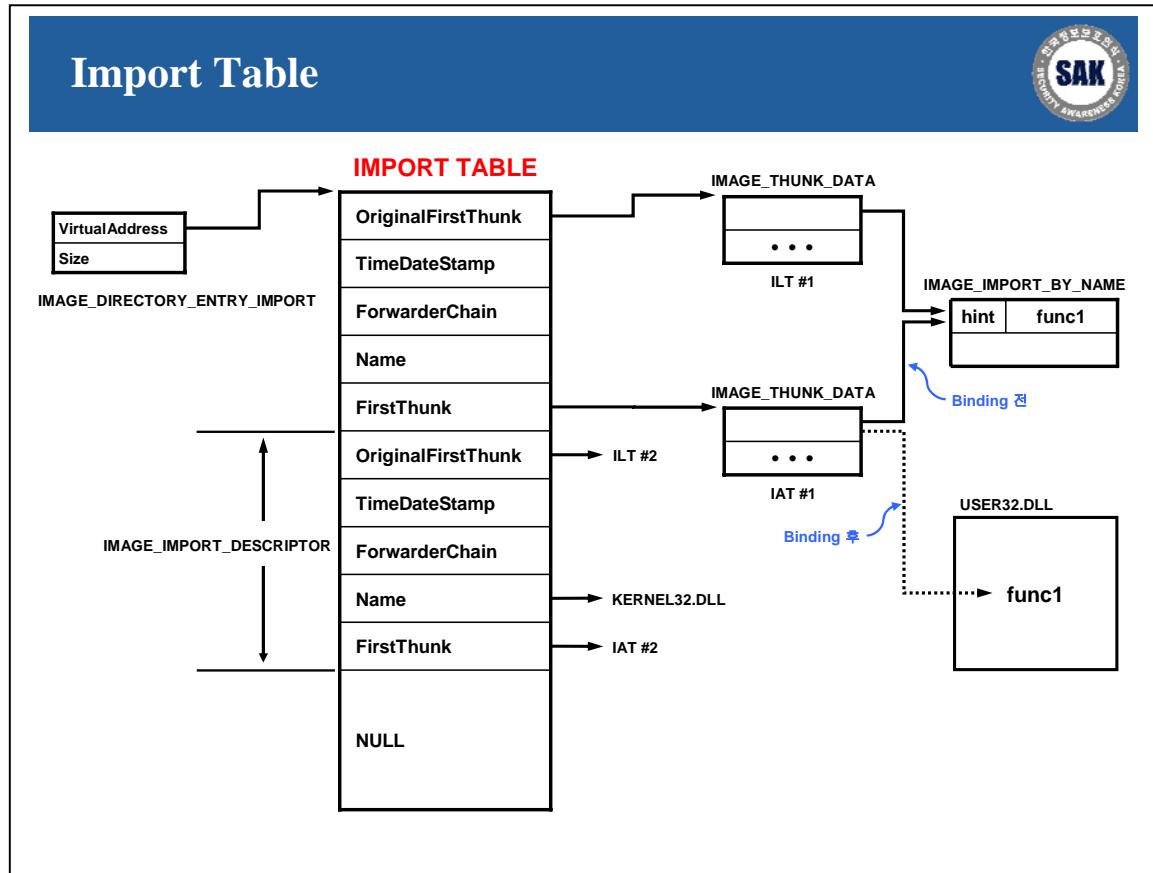
- **Name** : 섹션의 이름이다. 8byte의 크기를 가지고 있지만 이름이 NULL이거나 8byte보다 길어도 상관없다. 8byte 보다 길 경우엔 8byte 만큼만 이름으로 사용하고 나머지는 버리게 된다.
- **VirtualAddress** : 섹션이 로드될 가상 주소(RVA 값)이다.
- **SizeOfRawData** : 파일상에서의 섹션의 사이즈이다. FileAlignment의 배수이어야 한다.
- **PointerToRawData** : 파일상에서의 섹션의 시작 위치를 나타낸다.
- **Characteristics** : 섹션의 속성값이다.

로더는 PointerToRawData가 지정한 곳에서부터 SizeOfRawData 만큼 데이터를 읽어 들여 VirtualAddress

에 맵핑 한 후 Characteristics 에 설정된 속성을 이용하여 페이지 프로텍션을 적용한다.



### 3-6. Import Table



### Student Notes

Import 테이블에는 PE 파일이 실행될 때 외부로부터 가져와서 사용하는 함수들(DLL)의 목록이 들어 있다. 위 그림에서는 USER32.DLL 과 KERNEL32.DLL 을 임포트한 모습을 예로 한 것이다.

DLL 들은 함수를 익스포트하고 EXE 가 그 DLL 로부터 함수를 임포트한다. 익스포트된 함수를 가져온다는 것은 결국 해당 DLL 과 사용하는 함수에 대한 정보를 어딘가에 저장한다는 것을 의미하고 사용하고자 하는 익스포트 함수들과 그 DLL 에 대한 정보를 저장하고 있는 곳이 Import Table 이다. 일반적으로 PE 파일의 섹션 테이블에는 .idata 라는 이름으로 지정된다.

각 IMAGE\_IMPORT\_DESCRIPTOR 엔트리는 임포트한 DLL 의 정보를 가지고 있으며 마지막은 임포트 테이블의 끝을 알리기 위해 NULL 로 채워져 있다.

Winnt.h에 정의되어 있는 IMAGE\_IMPORT\_DESCRIPTOR와 관련된 구조체는 다음과 같다.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD Characteristics;
        DWORD OriginalFirstThunk;
    };
    DWORD TimeDateStamp;
    DWORD ForwarderChain;
    DWORD Name;
    DWORD FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;
```

- **OriginalFirstThunk**: ILT(Import Lookup Table)을 가리키는 RVA 값이다. 앞의 그림에서 보이는 것처럼 ILT는 IMAGE\_THUNK\_DATA로 구성된 배열이다. IMAGE\_THUNK\_DATA는 상황에 따라서 IMAGE\_IMPORT\_BY\_NAME을 가리키기도 하고 함수의 주소를 가리키기도 하며 오디널값으로 사용되거나 포워더로 사용되기도 한다.
- **TimeDateStamp**: 바인딩 전에는 0으로 설정되며 바인딩 후에는 -1로 설정된다.
- **ForwarderChain**: 바인딩 전에는 0으로 설정되며 바인딩 후에는 -1로 설정된다.
- **Name**: 임포트한 DLL의 이름을 가리키는 RVA 값이다.
- **FirstThunk**: IAT(Import Address Table)의 RVA 주소값을 가지고 있다. IAT 역시 ILT처럼 IMAGE\_THUNK\_DATA 배열이고 바인딩 전에는 ILT와 동일한 모습을 갖는다. 하지만 PE 파일이 메모리에 로드된 후에는 로더가 임포트 테이블의 각 엔트리의 이름 정보를 확인한 후 해당 DLL의 익스포트 테이블을 참조하여 함수의 실제 주소를 알아낸다. 그리고 나서 IAT를 실제 함수 주소로 업데이트한다.

IMAGE\_THUNK\_DATA는 다음과 같이 정의되어 있다.

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        PBYTE ForwarderString;
        PDWORD Function;
        DWORD Ordinal;
        PIMAGE_IMPORT_BY_NAME AddressOfData;
    } u1;
} IMAGE_THUNK_DATA32;
```

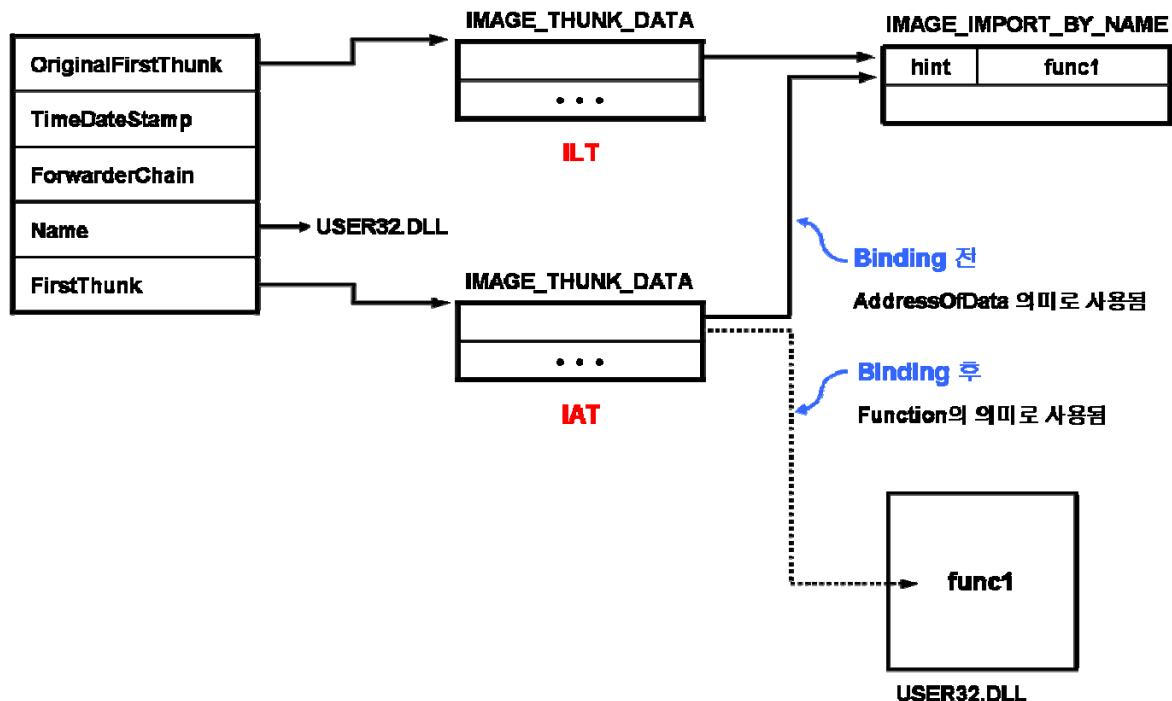
IMAGE\_THUNK\_DATA와 ILT, IAT의 관계를 정리해보면 다음과 같다.

- **ILT**
  - OriginalFirstThunk 가 가리키고 있다.
  - 바인딩 전후의 모습이 변경되지 않는다.

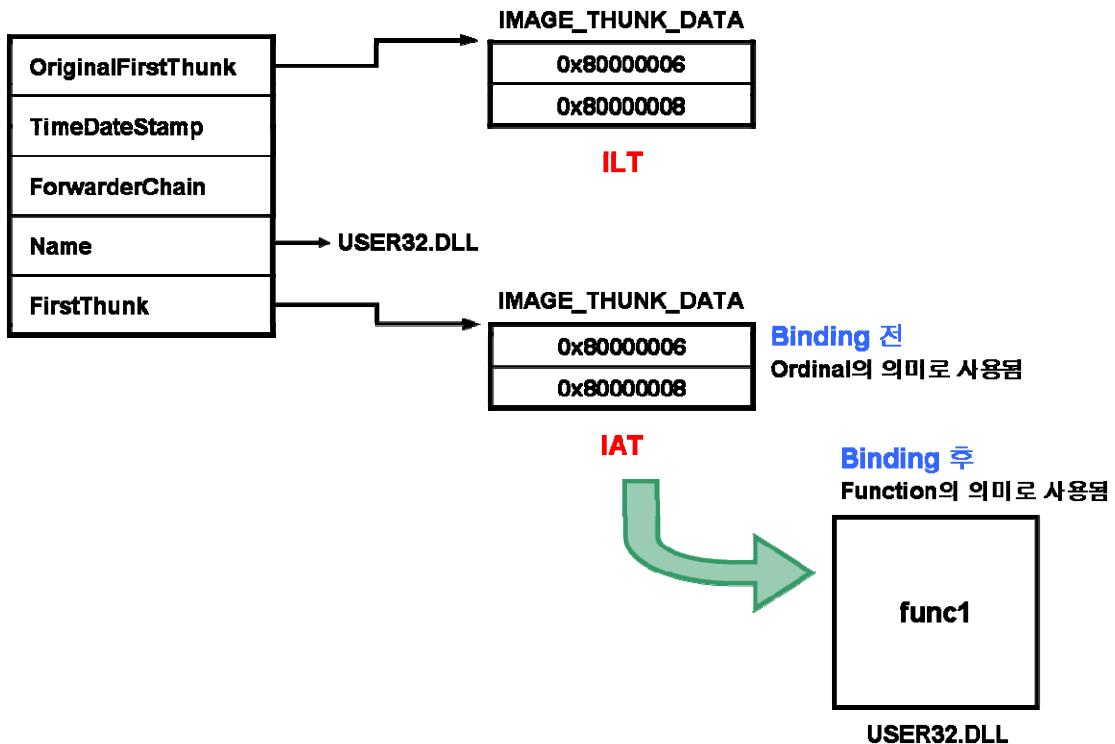
## Reverse Engineering

- IMAGE\_IMPORT\_BY\_NAME 구조체를 가리키거나 Ordinal로 사용되기도 한다.
  - 임포트한 함수에 대한 Ordinal 값을 저장하고 있는 배열이거나 임포트한 함수에 대한 이름을 저장하고 있는 IMAGE\_IMPRT\_BY\_NAME 구조체의 RVA 값으로 이루어진 배열이다.
  - 함수의 의미로 사용되지 않는다.
- 
- IAT
- FirstThunk 가 가리키고 있다.
  - 바인딩 전에는 AddressOfData의 의미로 사용되거나 Ordinal의 의미로 사용된다.
  - IMAGE\_THUNK\_DATA 가 IMAGE\_IMPORT\_BY\_NAME 을 가리키고 있으면 AddressOfData의 의미로 사용된 것이다. IMAGE\_IMPORT\_BY\_NAME 은 임포트할 수 있는 함수의 이름을 저장하고 있는 구조체이다.
  - 최상위 비트값이 1이면 ordinal로 사용된 것이고, 0이면 AddressOfData로 사용된 것이다.
  - 바인딩 후에는 함수의 실제 주소를 가리킨다.

다음 그림은 IMAGE\_THUNK\_DATA 가 AddressOfData로 사용될 경우를 도식화한 것이다.



다음 그림은 IMAGE\_THUNK\_DATA 가 Ordinal로 사용될 경우를 도식화한 것이다.



Winhex에서 임포트 테이블을 확인해보면 다음과 같다.

	OriginalFirstThunk				TimeDateStamp				ForwarderChain							
offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000123D0	00	00	00	00	80	30	01	00	FF	FF	FF	FF	FF	FF	FF	FF
000123E0	D6	34	01	00	20	10	00	00	10	31	01	00	FF	FF	FF	FF
000123F0	FF	F	Name		FirstThunk	00			B0	10	00	00	30	32	01	00
00012400	FF	F				FF			24	36	01	00	D0	11	00	00
00012410	60	30	01	00	FF	FF	FF	FF	FF	FF	FF	FF	AC	36	01	00
00012420	00	10	00	00	70	30	01	00	FF	FF	FF	FF	FF	FF	FF	FF
00012430	E4	36	01	00	10	10	00	00	18	31	01	00	FF	FF	FF	FF
00012440	FF	FF	FF	FF	7C	3B	01	00	B8	10	00	00	00	00	00	00
00012450	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

End of Import Table

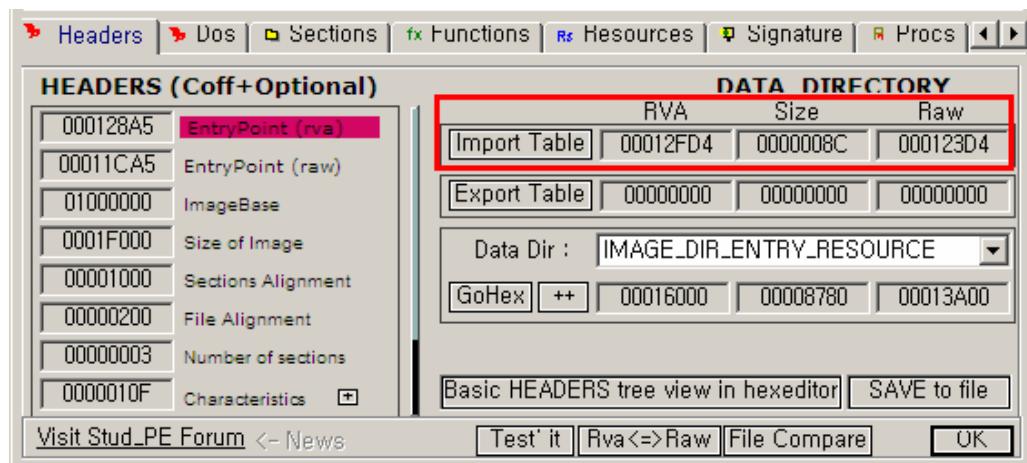
앞서 살펴보았듯이 임포트 테이블의 마지막은 NULL로 채워져 있다. 현재 이 파일은 6개의 DLL을 임포트한다는 것을 알 수 있다.

### Import Table의 위치

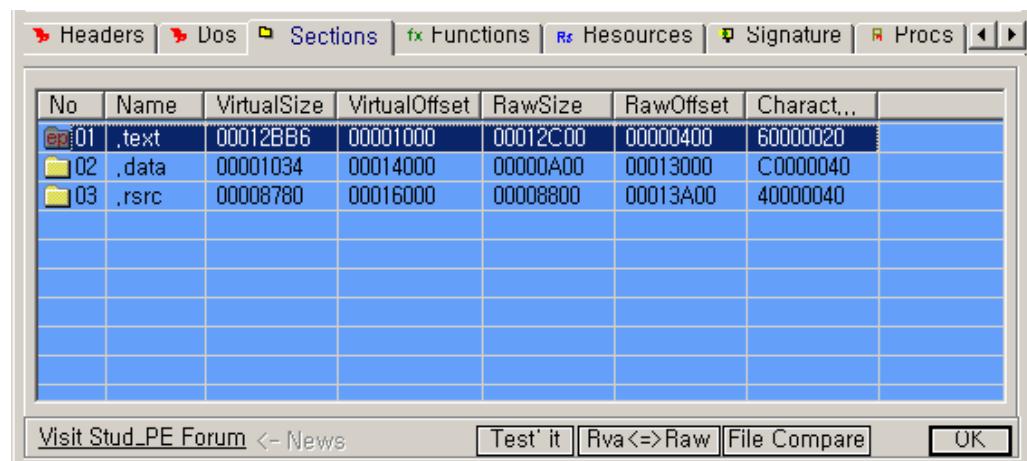
임포트 테이블을 메모리상에서 찾을 때는 IMAGE\_DIRECTORY\_ENTRY\_IMPORT에 지정된

## Reverse Engineering

VirtualAddress 를 찾아보면 확인할 수 있다. 하지만 파일상에서 즉, 메모리에 로드되기 전에 임포트 테이블이 어디에 있는지에 대한 정보는 PE 파일에 저장되어 있지 않다. Stud\_PE 로 확인해보면 임포트 테이블에 대한 정보를 확인할 수 있다.



재미있는 건 실제 PE 파일에 존재하지 않는 정보까지 확인할 수 있다는 것이다. Import Table 에서 Raw라는 항목은 파일상에서의 임포트 테이블의 위치를 나타내는 것이다. 실제 PE 파일에서 데이터 디렉토리에는 RVA 값인 VirtualAddress 와 Size 만 가지고 있다.



임포트 테이블의 RVA 값은 0x12FD4 이므로 .text 섹션에 있다는 것을 알 수 있다. .text 섹션의 RawOffset이 0x400 이므로 파일상에서의 위치는 0x123D4 가 된다. 간단하게 공식을 만들자면 파일상에서 임포트 테이블의 위치는

$$(\text{임포트 테이블의 RVA}) - (\text{임포트 테이블이 위치한 섹션의 VirtualOffset}) + \text{RawOffset}$$

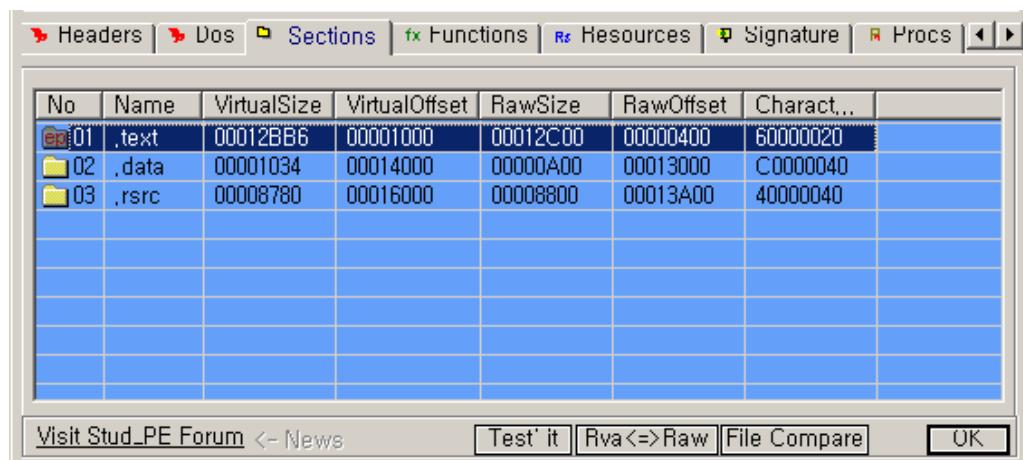
보통 임포트 테이블은 임포트 섹션의 시작점에 위치하지만 항상 그런 것은 아니다. 앞의 예제인 calc.exe에서도 임포트 테이블이 임포트 섹션이 아닌 .text 섹션에 위치하고 있는 것을 확인할 수 있다. 섹션 헤더에 포함된 임포트 섹션에 대한 정보는 파일을 메모리에 로딩하는 과정에서 파일상에서 임포트 섹션을 구분하고 메모리에 로딩하기 위해서 사용하는 것뿐이다.

그리고 PE 파일이 메모리에 로딩되고 나면 IAT(Import Address Table)을 수정해주어야 한다. 따라서 로더는 메모리상에서 임포트 테이블의 위치를 알 수 있어야 하는데 이 정보는 데이터 디렉토리 중 두 번째인 IMAGE DIRECTORY ENTRY IMPORT에 저장되어 있다.

## 임포트된 DLL 확인

앞서 살펴보았던 IMAGE\_IMPORT\_DESCRIPTOR 구조체에 임포트한 DLL의 정보를 가지고 있었다.  
먼저 임포트 테이블의 파일상에서의 위치를 찾아보도록 하자.

임포트 테이블의 RVA 가 0x12FD4 이고 임포트 테이블의 위치는 .text 섹션에 속해 있었다.



$\sum_{i=1}^{16}, (0x12FD4 - 0x1000) + 0x400 = 0x123D4$  가 된다. Winhex 를 통해 파일상에서 확인해보도록 하겠다.

## Reverse Engineering

offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	OriginalFirstThunk	TimeDateStamp	ForwarderChain
000123D0	00	00	00	00	80	30	01	00	FF	FF									
000123E0	D6	34	01	00	20	10	00	00	10	31	01	00	FF	FF	FF	FF	FF	FF	
000123F0	FF	F	Name		FirstThunk		00	00	B0	10	00	00	30	32	01	00			
00012400	FF	F					FF		24	36	01	00	D0	11	00	00			
00012410	60	30	01	00	FF	FF	FF	FF	FF	FF	FF	FF	AC	36	01	00			
00012420	00	10	00	00	70	30	01	00	FF										
00012430	E4	36	01	00	10	10	00	00	18	31	01	00	FF	FF	FF	FF	FF		
00012440	FF	FF	FF	FF	7C	3B	01	00	B8	10	00	00	00	00	00	00	00		
00012450	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		

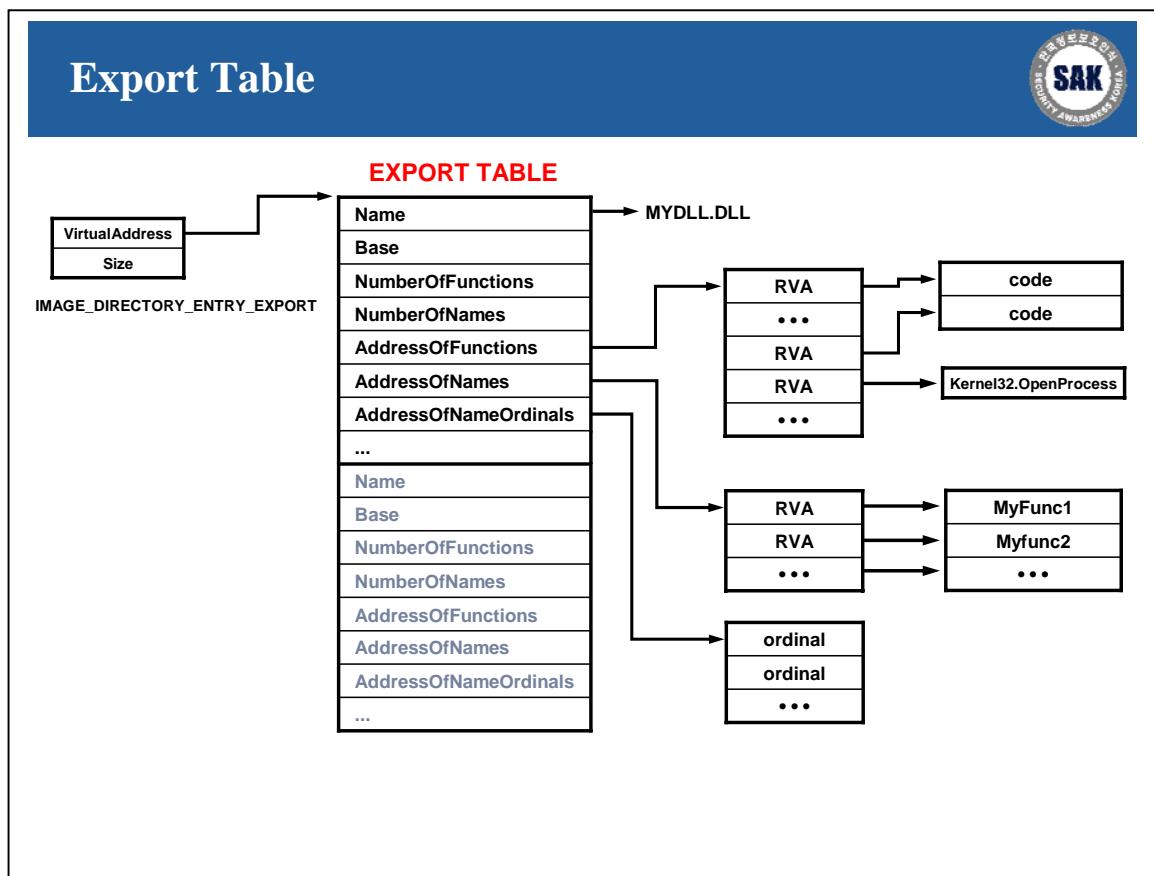
End of Import Table

calc.exe 프로그램이 임포트 할 DLL의 이름을 가리키고 있는 RVA 값은 0x134D6 이다. 이 값은 파일상에서의 오프셋이 아니라 메모리에 로드된 후에 찾아갈 RVA 값이므로 파일상에서는 다른 위치에 있을 것이다. Stud\_PE를 이용하여 섹션 테이블을 살펴보면 0x134D6는 .text 섹션 안에 있는 것을 확인할 수 있다. 결국 Name 은  $0x134D6 - 0x1000 + 0x400 = 0x128D6$ 에서 확인할 수 있다.

000128D0	69	6C	74	65	72	00	4B	45	52	4E	45	4C	33	32	2E	64	iilter.KERNEL32.d
000128E0	6C	6C	00	00	03	01	53	68	65	6C	6C	41	62	6F	75	74	ll....shellAbout

임포트 디렉토리의 첫번째 엔트리는 KERNEL32.dll 인 것을 알 수 있다.

### 3-7. Export Table



### Student Notes

Import 테이블이 외부로부터 가져오는 함수들의 목록이 들어있다면 Export 테이블은 다른 프로그램을 위한 기능을 제공하기 위해 노출하는 함수들의 목록이 들어있다. Export 테이블은 Data Directory 중에서 0 번째 인덱스인 **IMAGE\_DIRECTORY\_ENTRY\_EXPORT**에서 **VirtualAddress** 와 **Size** 를 확인 할 수 있다. Export 테이블은 Import 테이블에 비해 비교적 직관적이고 간단한 구조로 되어 있다. Winnt.h 에는 다음과 같이 정의되어 있다.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD    Characteristics;
    DWORD    TimeDateStamp;
    WORD     MajorVersion;
    WORD     MinorVersion;
```

## Reverse Engineering

```
DWORD    Name;
DWORD    Base;
DWORD    NumberOfFunctions;
DWORD    NumberOfNames;
DWORD    AddressOfFunctions;      // RVA from base of image
DWORD    AddressOfNames;         // RVA from base of image
DWORD    AddressOfNameOrdinals; // RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *IMAGE_EXPORT_DIRECTORY;
```

- **Name** : DLL 이름을 나타내는 ASCII 문자열의 위치를 지시하는 RVA 값
- **Base** : 익스포트된 함수들에 대한 서수의 시작 번호
- **NumberOfFunctions** : AddressOfFunctions 가 가리키는 RVA 배열의 개수
- **NumberOfNames** : AddressOfNames 가 가리키는 RVA 배열의 개수
- **AddressOfFunctions** : 함수의 실제 주소가 담긴 배열(RVA 값)
- **AddressOfNames** : 함수의 심볼을 나타내는 문자열 배열(RVA 값)
- **AddressOfNameOrdinals** : 함수의 서수값의 배열 위치. 실제 해당 함수의 서수는 Base 값을 더해야 한다.

구조체에 선언된 것과 같이 익스포트 테이블은 내보내지는 DLL 이 가지고 있는 함수들의 이름과 위치 (RVA 값), 그리고 서수를 가지고 있다.

## Export Table 의 위치

파일상에서 익스포트 테이블을 찾는 것은 임포트 테이블을 찾는 방법과 동일하다.

### 1. IMAGE\_DIRECTORY\_ENTRY\_EXPORT 에 지정된 VirtualAddress 확인

offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000140	00	00	04	00	00	10	00	00	00	00	10	00	00	10	00	00
00000150	00	00	00	00	10	00	00	00	1C	6A	01	00	9A	4B	00	00
00000160	A8	F1	05	00	50	00	00	00	00	30	06	00	90	9A	02	00

VirtualAddress 는 0x00016A1C 이고 Size 는 0x00004B9A 이다.

### 2. VirtualAddress 를 이용하여 RawOffset 계산

(익스포트 테이블의 RVA) – (익스포트 테이블이 위치한 섹션의 VirtualOffset) + RawOffset

$$0x00016A1C - 0x00001000 + 0x00000400 = 0x00015E1C$$

# Reverse Engineering

이스포트 테이블의 RVA 값은 0x16A1C 이므로 .text 섹션에 있다는 것을 알 수 있다. .text 섹션의 RawOffset이 0x400 이므로 파일상에서의 위치는 0x15E1C 가 된다. Winhex에서 보면 다음과 같다.

TimeDateStamp	MajorVersion	MinorVersion	Name	Base	Characteristics											
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00015E10	E9	84	5A	00	00	8B	C1	E9	8E	5A	00	00	00	00	00	00
00015E20	BE	EA	E6	45	00	00	00	00	D2	86	01	00	01	00	00	00
00015E30	DB	02	00	00	DB	02	00	00	44	6A	01	00	B0	75	01	00
00015E40	1C	81	01	00	16	31	01	00	7A	A1	02	00	99	FE	00	00

AddressOfNameOrdinals

NumberOfFuntions	NumberOfNames	AddressOfFunctions	AddressOfNames
------------------	---------------	--------------------	----------------

그렇다면 함수의 이름과 서수를 가지고 함수 주소를 찾는 방법에 대해서 알아보도록 하겠다.

Name : 0x000186D2

Base : 1

NumberOfFunctions : 0x0000002DB

NumberOfNames : 0x000002DB

AddressOfFunctions : 0x00016A44

AddressOfNames : 0x000175B0

AddressOfNameOrdinals : 0x0001811C

여기에서 사용되는 offset 값은 전부 RVA 이므로 파일상에서 찾을 때는 앞서 했던 계산 과정을 거쳐야 한다.

## Reverse Engineering

Name : 0x000186D2 – 0x1000 + 0x400 = 0x00017AD2

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00017AD0	DA	02	55	53	45	52	33	32	2E	64	6C	6C	00	41	63	74
00017AE0	69	76	61	74	65	4B	65	79	62	6F	61	72	64	4C	61	79

이름이 USER32.dll 이라는 것을 알 수 있다. 이름의 마지막은 NULL (\0)로 끝난다.

서수의 시작번호는 1이고 (Base : 1), 함수의 개수는 731개이고 (NumberOfFunctions : 0x2DB), 함수 이름의 개수도 731개이다. (NumberOfNames : 0x2DB)

실제 함수의 이름이 있는 곳은 AddressOfNames이다.

AddressOfNames : 0x000175B0 – 0x1000 + 0x400 = 0x000169B0

0x000186DD 은 RVA 값이므로 다시 파일상에서의 offset 으로 변환하면 0x00017ADD 이다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
000169B0	DD	86	01	00	F4	86	01	00	05	87	01	00	18	87	01	00
000169C0	23	87	01	00	3D	87	01	00	56	87	01	00	64	87	01	00
000169D0	6D	87	01	00	79	87	01	00	85	87	01	00	9A	87	01	00

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00017AD0	DA	02	55	53	45	52	33	32	2E	64	6C	6C	00	41	63	74
00017AE0	69	76	61	74	65	4B	65	79	62	6F	61	72	64	4C	61	79
00017AF0	6F	75	74	00	41	64	6A	75	73	74	57	69	6E	64	6F	77
00017B00	52	65	63	74	00	41	64	6A	75	73	74	57	69	6E	64	6F

첫 번째 실제 함수의 이름은 ActivateKeyboardLayout 이라는 것을 확인할 수 있다. 그리고 이 함수의 구현 코드가 있는 곳인 AddressOfFunctions 의 파일상의 위치는 0x00016A44 – 0x1000 + 0x400 = 0x00015E44 이다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00015E40	1C	81	01	00	16	31	01	00	7A	A1	02	00	99	FE	00	00
00015E50	79	E4	05	00	AE	92	04	00	31	A9	02	00	4E	B6	05	00

↓  
0x00012516 -> ActivateKeyboardLayout 구현 코드

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00012510	00	90	90	90	90	90	B8	2F	11	00	00	BA	00	03	FE	7F
00012520	FF	12	C2	08	00	90	90	90	90	90	8B	FF	55	8B	EC	8B
00012530	45	OC	3D	E0	03	00	00	0F	84	85	76	01	00	3D	C7	04
00012540	00	00	0F	87	29	B8	FF	FF	FF	75	14	FF	75	10	50	FF
00012550	75	08	E8	60	95	00	00	5D	C2	10	00	FF	75	0C	FF	75

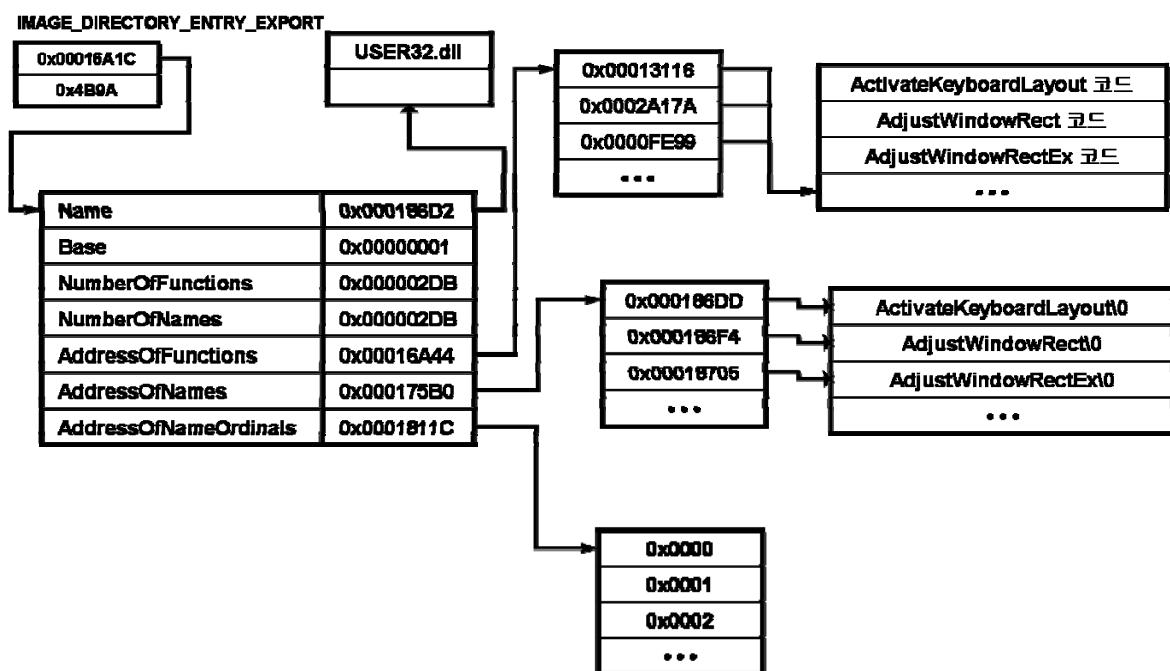
AddressOfNameOrdinals :  $0x0001811C - 0x1000 + 0x400 = 0x0001751C$

$0x1751C$ 의 위치에서부터 NumberOfNameOrdinals가 시작된다. 모든 함수들의 서수를 가지고 있는데 이 서수 배열의 최초 원소의 값은 항상 1이고 그리고 이 배열 값에 Base의 값을 더하면 정확한 서수 값을 획득할 수 있다. 현재 우리가 보고 있는 USER32.dll 파일은 총 731(0x2DB)개의 함수 이름을 가지고 있기 때문에 AddressOfNameOrdinals 또한 731(0x2DB)개를 가지고 있다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00017510	96	B5	01	00	A0	B5	01	00	AB	B5	01	00	00	00	01	00
00017520	02	00	03	00	04	00	05	00	06	00	07	00	08	00	09	00
00017530	0A	00	0B	00	0C	00	0D	00	0E	00	0F	00	10	00	11	00
00017540	12	00	13	00	14	00	15	00	16	00	17	00	18	00	19	00

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00017AA0	C2	02	C3	02	C4	02	C5	02	C6	02	C7	02	C8	02	C9	02
00017AB0	CA	02	CB	02	CC	02	CD	02	CE	02	CF	02	DO	02	D1	02
00017AC0	D2	02	D3	02	D4	02	D5	02	D6	02	D7	02	D8	02	D9	02
00017AD0	DA	02	55	53	45	52	33	32	2E	64	6C	6C	00	41	63	74

위 과정을 도식화하면 다음과 같다.



**Reverse Engineering**



---

## Module 4 — 분석 기초

### Objectives

- 분석 도구 설정
- CrackMe / KeygenMe 실습

## 4-1. 분석 도구 설정

### 분석 도구 설정



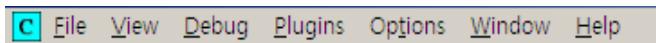
- Debugger
  - Ollydbg
- Disassembler
  - IDA
  - W32DASM
- File Analyzer
  - Stud PE
  - PEiD
  - ImportREConstruction
- HEX Editor
  - WinHex

### Student Notes

분석 도구에는 디버거, 디스어셈블러, 파일 분석기, 헥스 에디터 등 다양한 도구들이 존재한다. 도구들의 사용은 분석을 좀 더 용이하게 해주기 때문에 자주 사용이 되며 본인에게 가장 잘 맞는 도구를 선택하는 것도 중요하다. 여기서는 대표적인 도구들의 사용법을 위주로 소개하고자 한다.

## Ollydbg

<http://www.ollydbg.de/>에서 무료로 배포하고 있고, 바이너리 분석에 사용된다.



File : 디버깅 할 파일 혹은 프로세스를 지정합니다.

View : 디버깅 대상에 대한 각종 정보를 출력합니다.

Debug : 디버깅에 관련된 기능들입니다.

Plugins : 디버깅에 유용한 각종 유ти리티를 플러그인 형태로 제공합니다.

Options : 각종 옵션을 변경합니다.

Window, Help : 화면(창) 설정과 도움말입니다.



프로그램을 restart, close, run 시키거나 실행 과정을 조작할 수 있는 기능들을 제공한다.

어셈코드를 한 줄씩 실행하거나 혹은 특정 상황이 발생할 때까지 실행하거나 할 수 있다.



디버깅을 하는데 필요한 중요한 정보들을 보여준다. 자주 사용하는 것들은

E : Executable modules, 모듈 정보

M : Memory map, 현재 메모리 정보

H : Handles, 핸들 정보

C : CPU – main thread, 메인 디버깅 화면

/ : Patches, 기계어 코드가 변경된 내용을 출력

K : Call stack of main thread

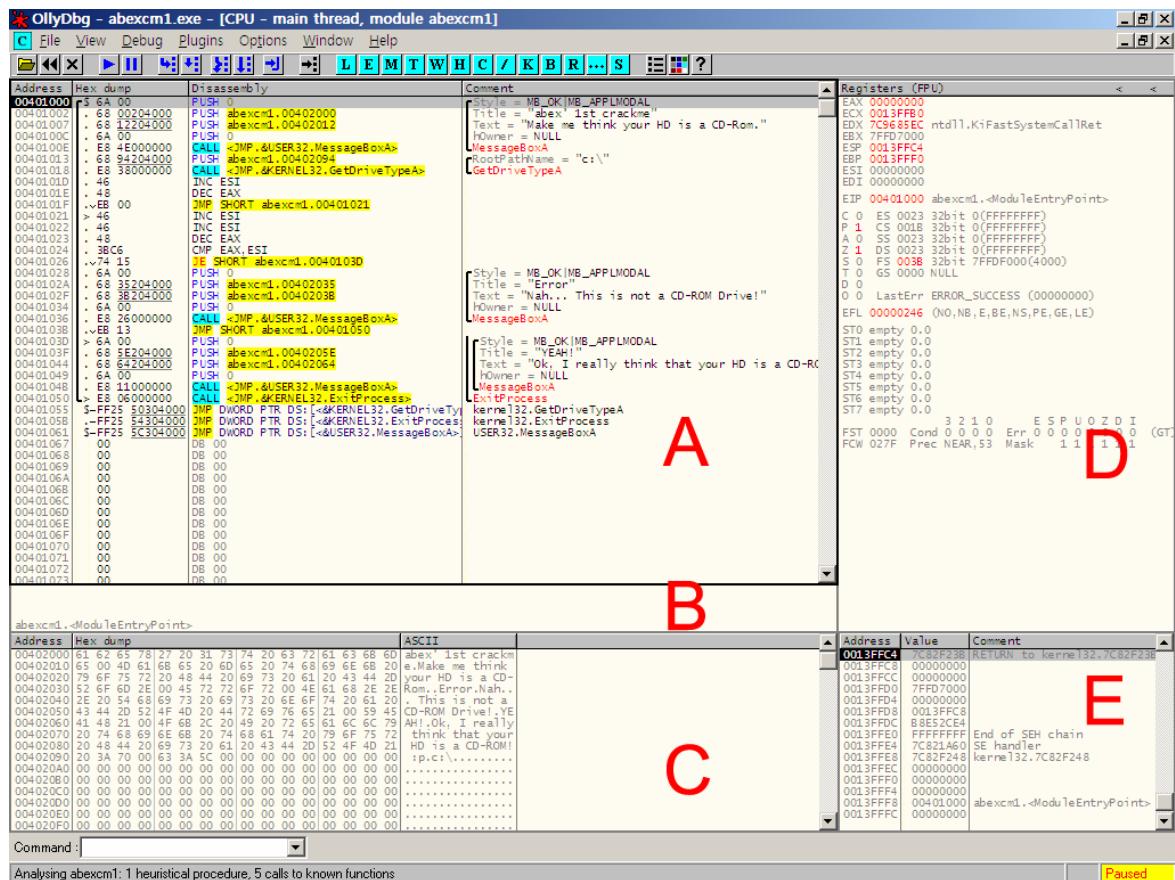
B : Breakpoints, 설정된 브레이크포인트

R : References, 프로그램에 사용된 함수나 문자열 등을 참조할 때 사용

... : Run trace

## Reverse Engineering

다음은 메인 화면에 대해서 알아보도록 하자.



A 부분은 디스어셈블링된 코드와 OP 코드들을 나타낸다. 네 부분으로 나뉘어져 있는데 차례대로 메모리 주소, OP 코드, 디스어셈블된 코드, 코멘트의 내용을 가지고 있다. 이곳에서 프로그램의 흐름이나 실행을 제어해서 분석한다.

B 부분은 상태바이다. A 부분에서 실행되고 있는 각 해당 위치의 offset 값과 변경된 메모리 주소, 레지스터 내용들을 나타낸다.

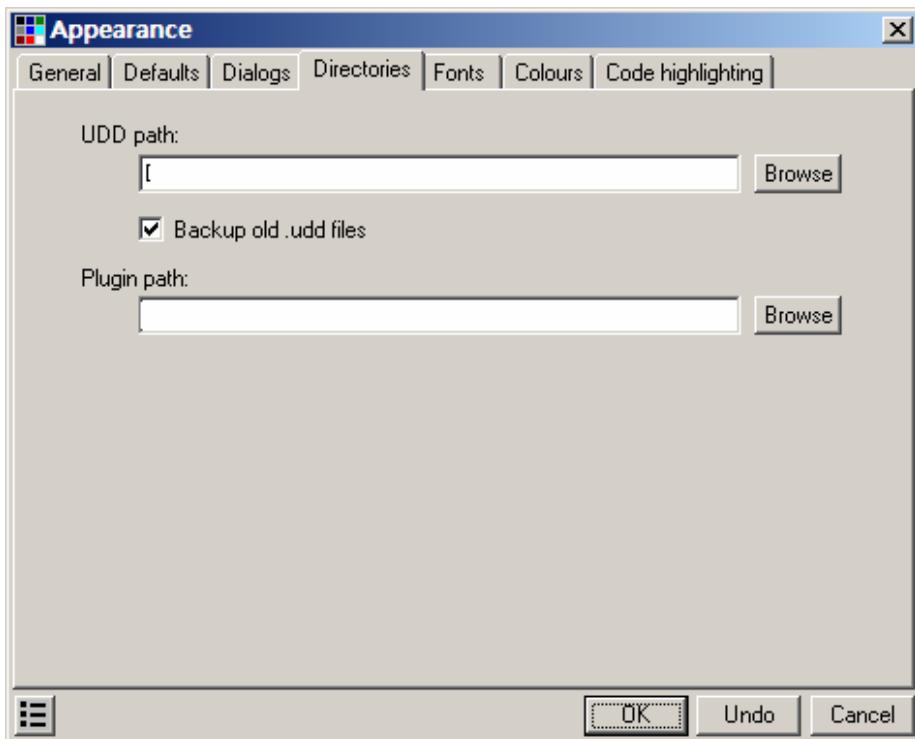
C 부분은 메모리의 값들을 헥사코드와 아스키 코드로 보여주는 부분이다. 그 외 다른 다양한 형태로 값을 확인할 수 있다.

D 부분은 CPU 레지스터들의 정보를 나타낸다. 레지스터 값이나 플래그등을 직접 조작할 수도 있다.

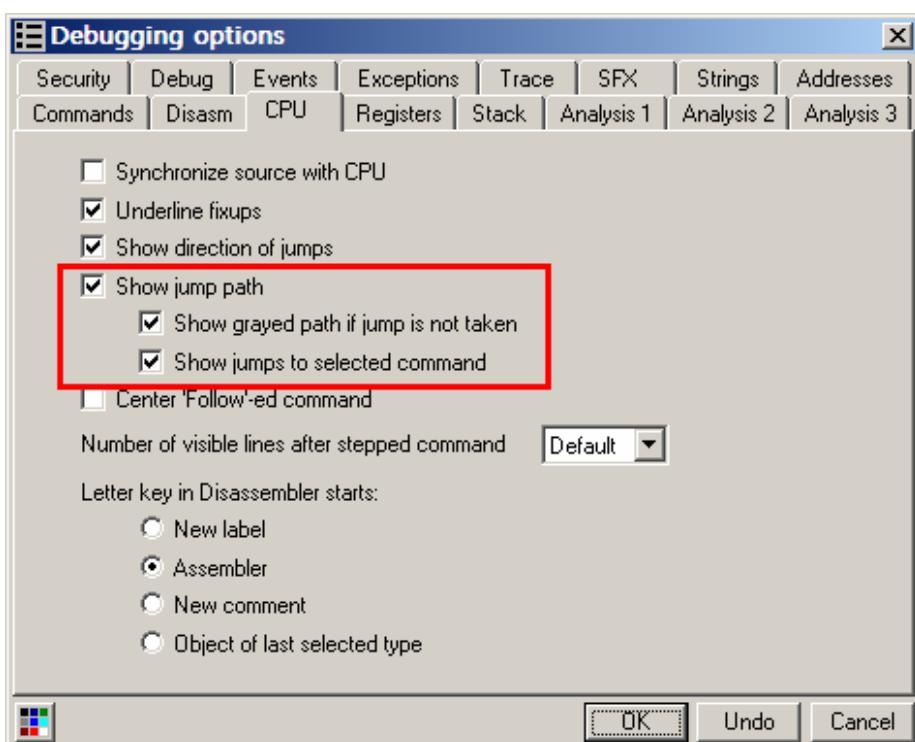
E 부분은 Stack의 내용을 보여준다.

Ollydbg를 이용하여 분석하기 전 몇 가지 설정을 해주어야 한다.

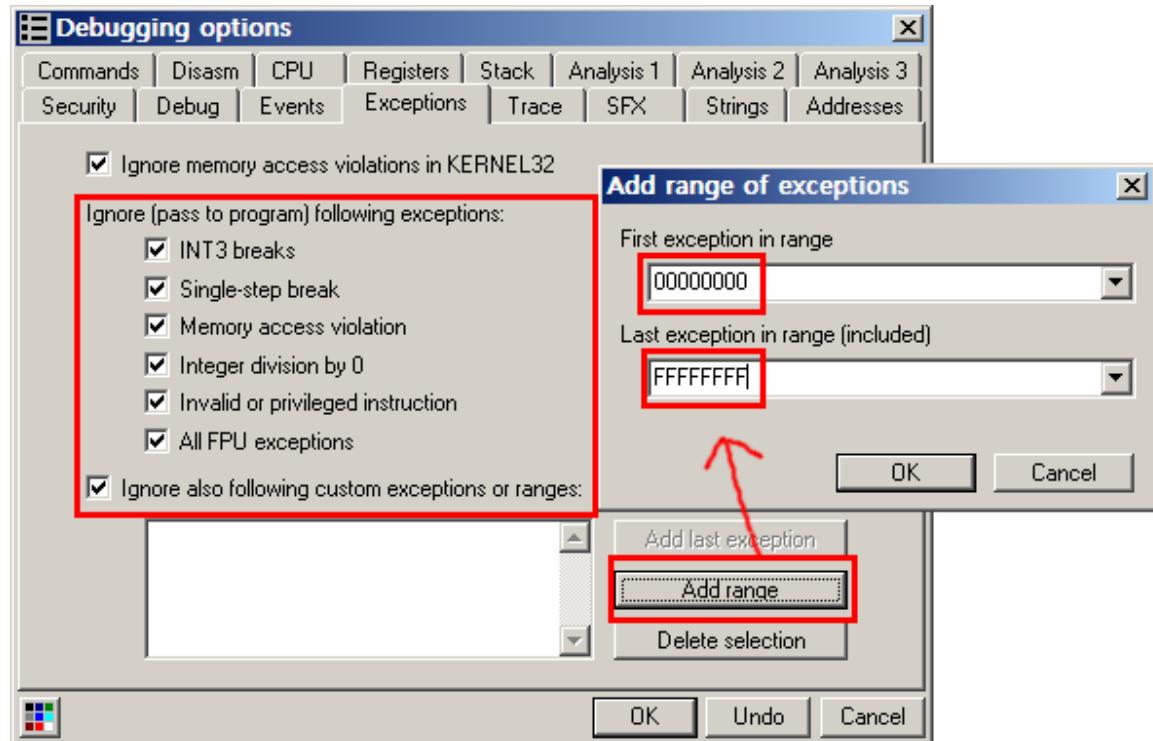
먼저 Option → Appearance에서 udd 폴더와 plugins 폴더를 지정해준다.



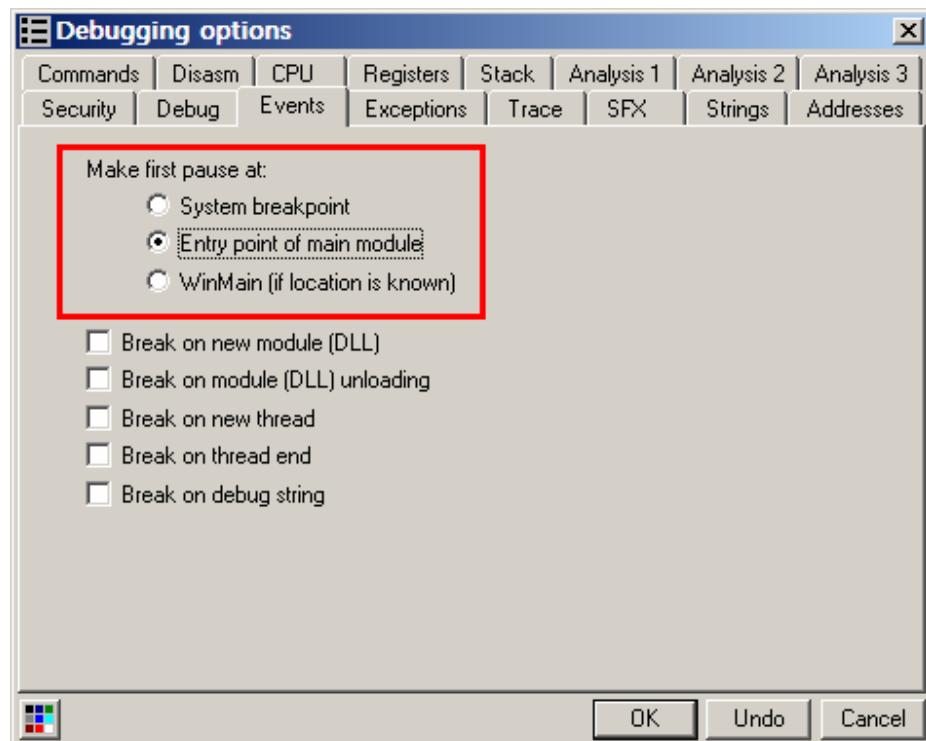
Option → Debugging Option → CPU : jump 구문이 있을 경우 어느 곳으로 점프하는지 보여준다.



## Reverse Engineering



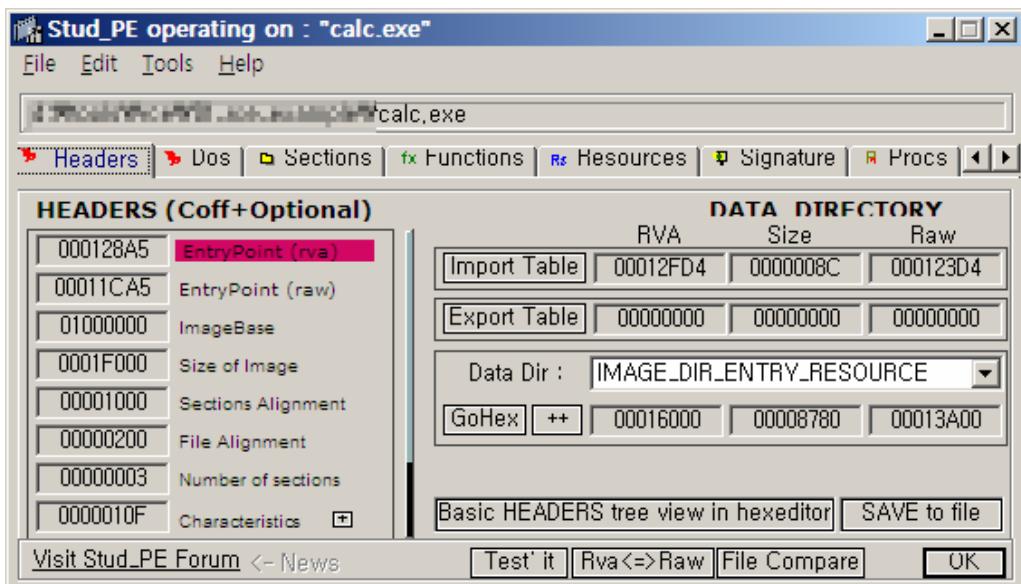
Exception 발생시 어떻게 처리할 것인지와 범위를 지정한다.



프로그램을 디버거로 오픈할 때 어느 곳에서 멈출 것인지 지정한다.

## Stud PE

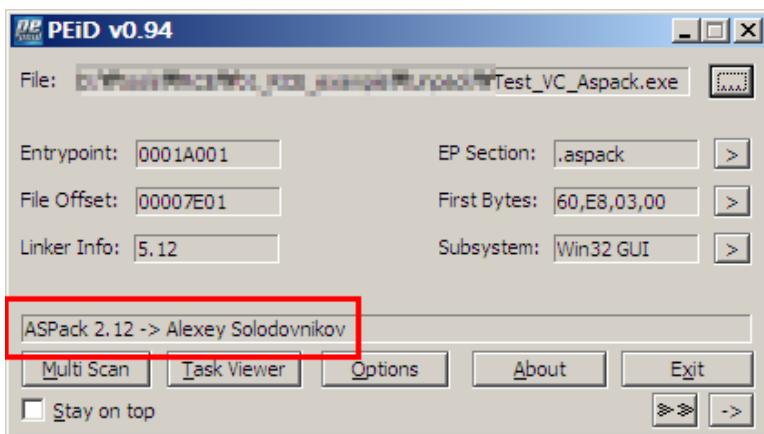
PE 파일의 구조를 분석해 주는 프로그램으로 <http://www.cgsoftlabs.ro/studpe.html>에서 다운 받을 수 있으며 프리웨어이다.



## PEiD

PE 파일의 정보를 볼 수 있는 것은 Stud PE 와 비슷하고 패킹 여부를 확인할 수 있는 기능도 있다.

<http://www.peid.info/>에서 다운로드 받을 수 있다.

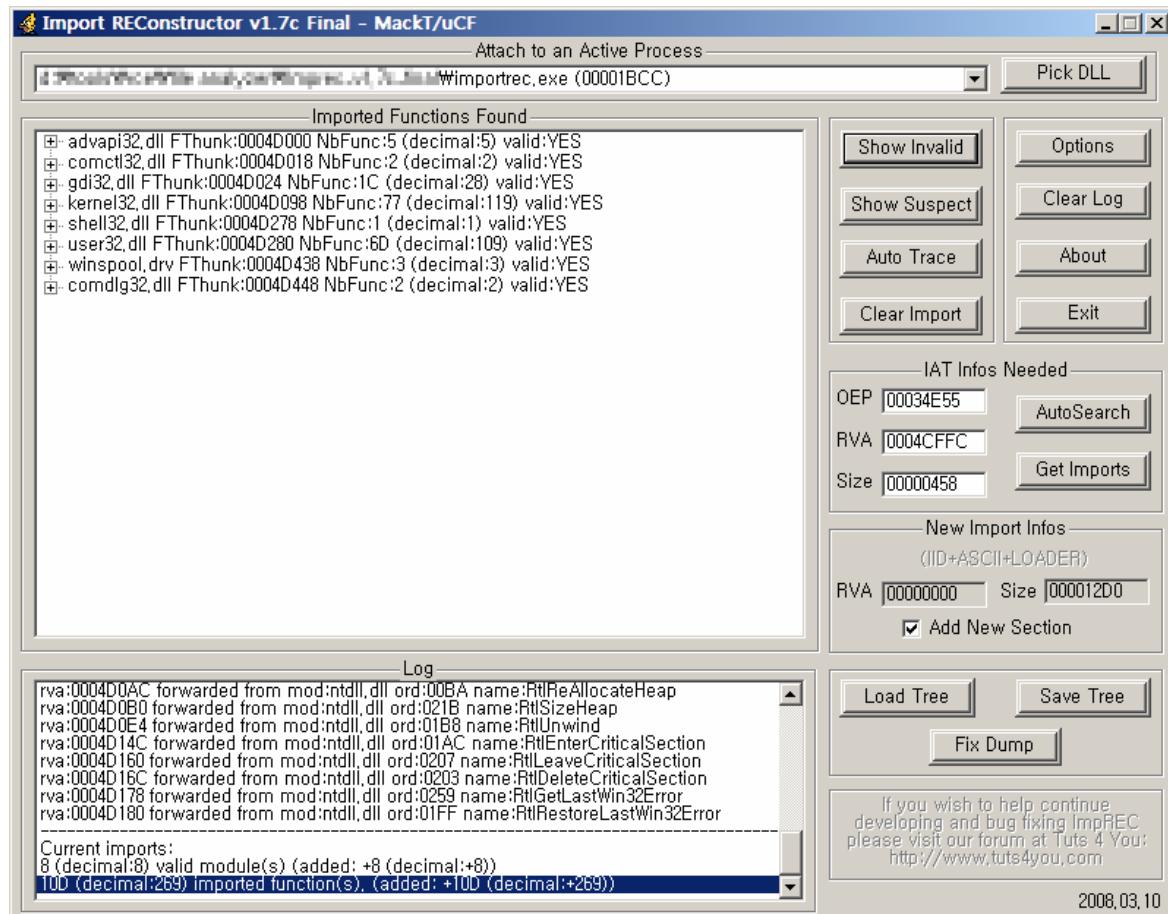


## Reverse Engineering

### ImportREC

IAT를 복구시켜주는 프로그램이다. 패킹이 되어 있는 파일을 언패킹한 후 IAT를 맞춰주지 않으면 정상적으로 실행되지 않기 때문에 IAT를 복구시켜주어야 한다. 이 때 ImportREC을 사용한다.

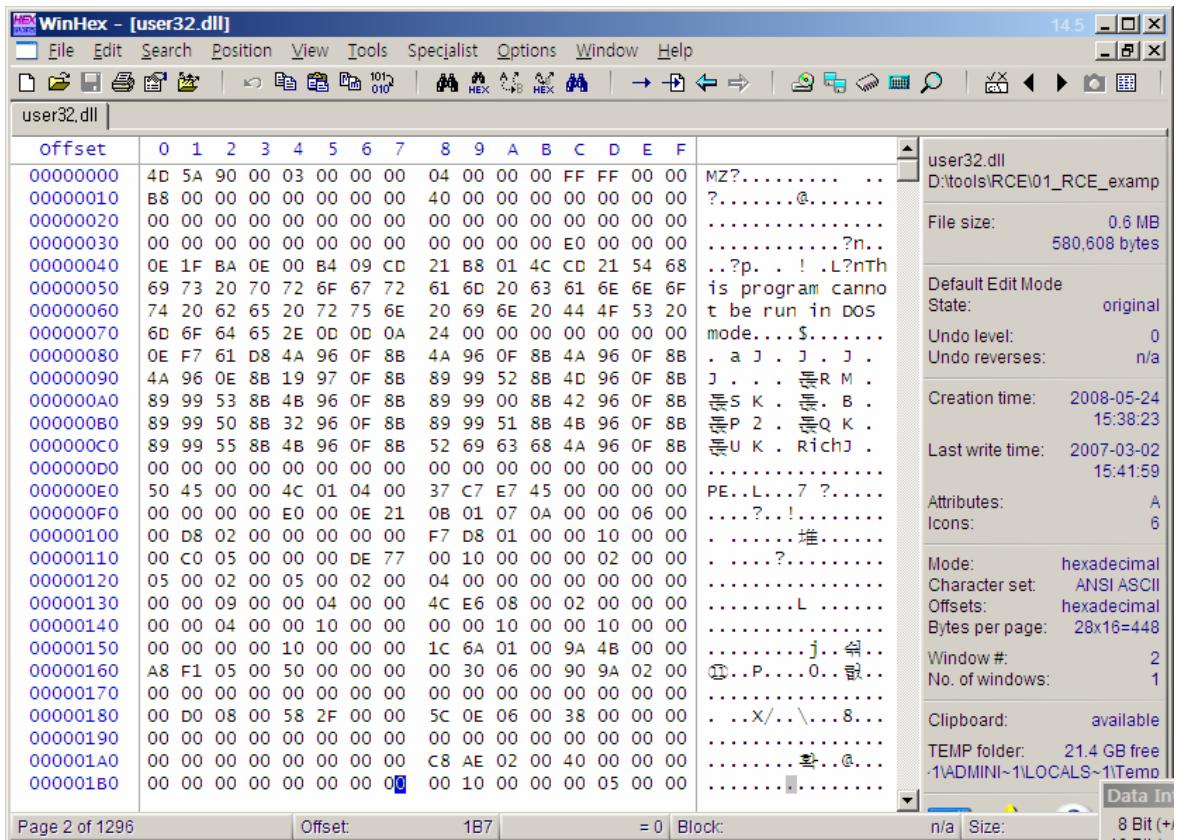
<http://vault.reversers.org/ImpRECDef>에서 다운로드 받을 수 있다.



하지만 모든 상황에서 IAT가 정상적으로 복구되는 것은 아니다. 패커가 의도적으로 IAT를 꼬이게 했을 경우 정상적으로 찾지 못할 수 있다. 이것은 임포트 테이블의 마지막이 0000 0000이라는 점을 이용하여 자동으로 IAT를 복구하는데 정상적인 함수들의 목록 중간에 0000 0000을 삽입하게 되면 그곳이 마지막이라고 ImportREC이 인식하기 때문이다. 이런 경우 수작업으로 IAT의 주소와 사이즈를 맞춰 주어야 한다.

## Winhex

파일을 HEX 형태로 볼 수 있다. 바이너리 파일을 직접 수정할 수도 있다.



## 4-2. CrackMe 분석 실습

### CrackMe 실습



- 인증 우회하기
- Key 값 찾기
- Keyfile 만들기
- 등등

### Student Notes

이번 장에서는 온라인 상에 공개되어 있는 Crackme 예제를 통해 몇 가지 기법들에 대해서 익히도록 하겠다. 온라인 상에 공개된 Crackme 프로그램들의 특징을 살펴보면 인증을 우회하거나 고정된 키값을 찾아서 입력하는 방법, 또는 키 파일을 만들어야 하는 방법 등 여러 가지 방법들을 있다. 물론 Keygen을 만들어서 입력된 이름이나 아이디에 의해 변경되는 키값을 알아내는 방법도 있는데 Keygen에 관한 내용은 다음 장에서 보도록 하겠다.

여기서는 먼저 고정된 키값을 찾아내는 크랙미를 이용하여 실습을 할 것이고 두 번째는 우리가 입력한 값에 의해 키값이 변하는 크랙미에 대해서 분석을 해보도록 하겠다.

## Crackme #1

이 파일을 실행하면 다음과 같이 패스워드를 입력하라고 나오고 아무런 값이나 입력해보도록 하겠다.

```
D:\W00_target>Crackme#1.exe
Please enter the password:
1234567890
Invalid Password
```

이 프로그램에서 요구하는 패스워드와 일치하지 않기 때문에 “Invalid Password”라는 메시지를 출력하고 프로그램이 종료를하게 된다. 여기서 우리가 확인해야 할 부분은 정확한 패스워드를 입력하지 않았을 때 출력되는 “Invalid Password”라는 문구이다. 디버거를 통해서 프로그램을 실행시켜 보자.

코드 창에서 마우스 오른쪽을 클릭하여 Search for → All reference text strings 를 클릭하면 다음과 같이 이 프로그램에서 사용되는 스트링들을 확인할 수 있다.

R Text strings referenced in Crackme#.text		
Address	Disassembly	Text string
00401022	PUSH Crackme#.00407030	ASCII "Please enter the password:"
004010C9	PUSH Crackme#.0040704C	ASCII "Invalid Password"
004010E0	PUSH Crackme#.00407060	ASCII "The password is %s"
00401220	PUSH EBP	(Initial CPU selection)
00402C7A	PUSH Crackme#.00406430	ASCII "<program name unknown>"
00402CBC	PUSH Crackme#.0040642C	ASCII "..."
00402CDO	PUSH Crackme#.00406410	ASCII "Runtime Error! Program: "
00402CEE	PUSH Crackme#.0040640C	ASCII ""
00402D16	PUSH Crackme#.004063E4	ASCII "Microsoft Visual C++ Runtime Library"
0040473A	PUSH Crackme#.00406478	ASCII "user32.dll"
00404751	PUSH Crackme#.0040646C	ASCII "MessageBoxA"
00404762	PUSH Crackme#.0040645C	ASCII "GetActiveWindow"
0040476A	PUSH Crackme#.00406448	ASCII "GetLastActivePopup"

앞에서 보았던 문구가 두 번째 줄에 보이고 아래쪽에 “The password is %s”라는 스트링이 보인다. 아마도 정확한 패스워드를 입력할 경우 “The password is xxxxxxxxxxxx”라고 콘솔에 출력이 될 것이고 그렇지 않을 경우 “Invalid Password”라는 메시지를 출력하는 것일거라 예상할 수 있다. “Invalid Password”라는 스트링을 더블 클릭하면 해당 코드로 이동하게 된다.

## Reverse Engineering

Address	Hex dump	Disassembly	Comment
004010A8	. 83EA 01	SUB EDX, 1	
004010AB	. 8955 DC	MOV DWORD PTR SS:[EBP-24], EDX	
004010AE	> 837D E0 0D	CMP DWORD PTR SS:[EBP-20], 0D	
004010B2	. v 73 28	JNB SHORT Crackme#.004010DC	
004010B4	. 8B45 E0	MOV EAX, DWORD PTR SS:[EBP-20]	
004010B7	. C1E8 02	SHR EAX, 2	
004010BA	. 8B4D F8	MOV ECX, DWORD PTR SS:[EBP-8]	
004010BD	. 8B55 DC	MOV EDX, DWORD PTR SS:[EBP-24]	
004010C0	. 8B0481	MOV EAX, DWORD PTR DS:[ECX+EAX*4]	
004010C3	. 3B4495 E8	CMP EAX, DWORD PTR SS:[EBP+EDX*4-18]	
004010C7	. v 74 11	JE SHORT Crackme#.004010DA	
004010C9	. 68 4C704000	PUSH Crackme#.0040704C	ASCII "Invalid Password"
004010CE	. E8 20000000	CALL Crackme#.004010F3	
004010D3	. 83C4 04	ADD ESP, 4	
004010D6	. 33C0	XOR EAX, EAX	
004010D8	. v EB 15	JMP SHORT Crackme#.004010EF	
004010DA	>^ EB C0	JMP SHORT Crackme#.0040109C	
004010DC	> 8D4D CC	LEA ECX, DWORD PTR SS:[EBP-34]	
004010DF	. 51	PUSH ECX	
004010E0	. 68 60704000	PUSH Crackme#.00407060	
004010E5	. E8 09000000	CALL Crackme#.004010F3	
004010EA	. 83C4 08	ADD ESP, 8	
004010ED	. 33C0	XOR EAX, EAX	
004010EF	> 8BE5	MOV ESP, EBP	
004010F1	. 5D	POP EBP	
004010F2	. C3	RETN	

“Invalid Password”라는 메시지 바로 위라인에 비교하는 부분이 있고 같을 경우 004010DA로 분기하는 것을 볼 수 있다. 004010DA는 다시 0040109C로 분기하기 때문에 아래쪽에 보이는 “The password is %s” 부분으로 이동하지는 않는다. 위쪽으로 조금 더 올라가면 JNB에 의해서 분기하는 코드가 보이는데 조건에 만족하면 004010DC로 분기하여 “The password is %s”라는 메시지를 출력하게 된다. 즉, 현재 보이는 코드보다 윗부분에서 우리가 입력한 패스워드와 프로그램이 요구하는 패스워드를 비교하는 부분이 있을 것이다. 그렇다면 화면을 조금 위로 올려서 패스워드를 입력 받는 부분을 확인해보도록 하겠다.

Address	Hex dump	Disassembly
00401022	. 68 30704000	PUSH Crackme#.00407030
00401027	. E8 7C010000	CALL Crackme#.004011A8
0040102C	. 83C4 04	ADD ESP, 4
0040102F	. 6A 10	PUSH 10
00401031	. 6A 00	PUSH 0
00401033	. 8D45 CC	LEA EAX, DWORD PTR SS:[EBP-34]
00401036	. 50	PUSH EAX
00401037	. E8 14010000	CALL Crackme#.00401150
0040103C	. 83C4 0C	ADD ESP, 0C
0040103F	. C745 E4 0000	MOV DWORD PTR SS:[EBP-1C], 0
00401046	. C745 E0 0000	MOV DWORD PTR SS:[EBP-20], 0
0040104D	. C745 DC 0000	MOV DWORD PTR SS:[EBP-24], 0

PUSH Crackme#.00407030은 “Please enter the password: ”라는 출력할 메시지를 스택에 넣어둔 후 바로 아래에 있는 CALL에 의해서 화면에 출력이 된다. 항상 CALL에 의해서 함수 호출이 끝나면 스택을 잘 보아야 한다. 함수 내부에서 어떠한 연산 작업 후 스택에 데이터들이 저장되기 때문이다. 그리고 이 데이터들은 다음 코드에 영향을 미칠 수 있기 때문이다.

Registers (FPU)
EAX 00000000
ECX 0000001D
EDX 7C9685EC ntdll.KiFastSystemCallRet
EBX 7FFD4000
ESP 0013FF48
EBP 0013FF80
ESI 00000000
EDI 00000000

현재 스택의 값들을 확인하고 다음 코드를 보도록 하겠다.

```
/*40102C*/ ADD ESP,4           // 스택 클리어
/*40102F*/ PUSH 10            // 스택에 10을 저장
/*401031*/ PUSH 0             // 스택에 0을 저장
/*401033*/ LEA EAX,DWORD PTR SS:[EBP-34] // EBP-34의 위치(주소)를 EAX에 저장
/*401036*/ PUSH EAX           // EAX의 값을 스택에 저장
/*401037*/ CALL Crackme#.00401150 // 00401150의 코드를 호출
/*40103C*/ ADD ESP,0C          // 스택 클리어
/*40103F*/ MOV DWORD PTR SS:[EBP-1C],0 // EBP-1C의 위치에 0을 저장
/*401046*/ MOV DWORD PTR SS:[EBP-20],0 // EBP-20의 위치에 0을 저장
/*40104D*/ MOV DWORD PTR SS:[EBP-24],0 // EBP-24의 위치에 0을 저장
```

위 부분은 우리가 코드를 분석하는데 큰 영향을 미치지는 않는 부분들이고 단지 우리가 입력한 패스워드와 프로그램이 가지고 있는 패스워드를 비교하는 연산을 위해서 공간을 만드는 역할을 하는 코드이다.

다음 코드로 진행을 하면 패스워드를 입력할 수 있게 된다.



임의의 패스워드를 입력하고 다음 코드를 살펴보자.

00401054에서 00401140에 위치한 함수를 호출한다. 이 함수는 사용자의 입력값을 받아 들이는 함수이다. 그리고 함수를 호출하고 연산을 수행한 결과는 EAX에 저장된다.

## Reverse Engineering

Address	Hex dump	Disassembly
00401054	> E8 E7000000	CALL Crackme#.00401140
00401059	. 8845 FC	MOV BYTE PTR SS:[EBP-4], AL
0040105C	. 8B4D E4	MOV ECX, DWORD PTR SS:[EBP-1C]
0040105F	. 8A55 FC	MOV DL, BYTE PTR SS:[EBP-4]
00401062	. 88540D CC	MOV BYTE PTR SS:[EBP+ECX-34], DL
00401066	. 8B45 E4	MOV EAX, DWORD PTR SS:[EBP-1C]
00401069	. 83C0 01	ADD EAX, 1
0040106C	. 8945 E4	MOV DWORD PTR SS:[EBP-1C], EAX
0040106F	. 0FBE4D FC	MOVsx ECX, BYTE PTR SS:[EBP-4]
00401073	. 83F9 0A	CMP ECX, 0A
00401076	.~ 74 0E	JE SHORT Crackme#.00401086
00401078	. 0FBE55 FC	MOVsx EDX, BYTE PTR SS:[EBP-4]
0040107C	. 85D2	TEST EDX, EDX
0040107E	.~ 74 06	JE SHORT Crackme#.00401086
00401080	. 837D E4 10	CMP DWORD PTR SS:[EBP-1C], 10
00401084	.^ 72 CE	JB SHORT Crackme#.00401054
00401086	> 8D45 CC	LEA EAX, DWORD PTR SS:[EBP-34]
00401089	. 8945 F8	MOV DWORD PTR SS:[EBP-8], EAX
0040108C	. C745 E0 0000	MOV DWORD PTR SS:[EBP-20], 0
00401093	. C745 DC 0300	MOV DWORD PTR SS:[EBP-24], 3
0040109A	.~ EB 12	JMP SHORT Crackme#.004010AE

Disassembly 부분에서 굵은 선으로 표기된 부분은 루프문을 나타내는 것이다. 즉, 특정 조건을 만족할 때 까지 00401054 부터 00401084 까지 계속 루프를 돌게 된다. 그렇다면 현재 위에 보이는 코드가 어떠한 역할을 하는지 알아보도록 하겠다.

```
/*401059*/ MOV BYTE PTR SS:[EBP-4],AL      // AL에 있는 값을 EBP-4에 1바이트만 저장
/*40105C*/ MOV ECX,DWORD PTR SS:[EBP-1C] // EBP-1C에 있는 값을 ECX에 저장
/*40105F*/ MOV DL,BYTE PTR SS:[EBP-4]      // EBP-4에 있는 값을 DL에 저장
/*401062*/ MOV BYTE PTR SS:[EBP+ECX-34],DL // DL에 있는 값을 EBP+ECX-34에 저장
/*401066*/ MOV EAX,DWORD PTR SS:[EBP-1C] // EBP-1C에 있는 값을 EAX에 저장
/*401069*/ ADD EAX,1                      // EAX 값을 1 증가
/*40106C*/ MOV DWORD PTR SS:[EBP-1C],EAX // EAX에 있는 값을 EBP-1C에 저장
/*40106F*/ MOVsx ECX,BYTE PTR SS:[EBP-4] // EBP-4에 있는 값을 ECX에 저장
/*401073*/ CMP ECX,0A                    // ECX와 0x0A를 비교
/*401076*/ JE SHORT Crackme#.00401086 // 같으면 00401086으로 분기
/*401078*/ MOVsx EDX,BYTE PTR SS:[EBP-4] // EBP-4에 있는 값을 EDX에 저장
/*40107C*/ TEST EDX,EDX                 // EDX와 EDX를 AND 연산
/*40107E*/ JE SHORT Crackme#.00401086 // 같으면 00401086으로 분기
/*401080*/ CMP DWORD PTR SS:[EBP-1C],10 // EBP-1C에 있는 값과 0x10을 비교
/*401084*/ JB SHORT Crackme#.00401054 // EBP-1C의 값이 0x10보다 적으면 00401054
```

로 분기

## Reverse Engineering

계속해서 루프를 돌면서 뭔가를 계산하고 있다. 4 번 정도 루프를 돋 후 헥사값이 있는 부분을 확인해보도록 하자.

Address	Hex dump	[EBP-1C]	[EBP+ECX-34]	ASCII
0013FF40	3E 11 40 00 88 70 40 00 59 10 40 00 71 77 65 72		71	>@. @. Y+@. qwer
0013FF50	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00		00	.....
0013FF60	00 00 00 00 04 00 00 00 00 67 0A 00 00 70 70 69 6E	04	70	....d...g...ppin
0013FF70	72 74 72 69 70 6F 77 65 00 00 00 00 72 20 40 00	70	72	rtripowe....r @.
0013FF80	C0 FF 13 00 E5 12 40 00 01 00 00 00 30 0F 41 00	13	30	?!!.?@. r....0@.
0013FF90	A0 OE 41 00 00 00 00 00 00 00 00 00 C0 FD 7F	00	C0	?A.....

EBP

[EBP-4]

CALL 00401140에 의해 우리가 입력한 값을 한 글자씩 가져와서 EAX에 저장하고 글자수를 세고 있다. 입력한 글자의 개수는 [EBP-1C]에 저장이 되고 현재까지 읽어 들인 글자들은 [EBP+ECX-34]에 저장이 된다. 그리고 현재 읽어 들인 글자는 [EBP-4]에 저장이 된다. 코드에서 보면 BYTE PTR SS라는 부분이 자주 나오는데 이것이 바로 한 글자씩 (1 바이트) 읽어 온다는 증거이다.

0040106F	. OFBE4D FC	MOVsx ECX, BYTE PTR SS:[EBP-4]
00401073	. 83F9 OA	CMP ECX, OA

그리고 ECX에는 위에 보는 것과 같이 [EBP-4]에 있는 값을 저장한 후 0x0A와 비교하고 있다. [EBP-4]는 현재 읽어 들인 글자 하나를 의미하고 0x72(r)와 0x0A를 비교하는 것이다. 0x0A는 LF(Line Feed)로써 사용자가 입력한 글자의 끝을 찾기 위해 사용된 비교 구문이다.

00401080	. 837D E4 10	CMP DWORD PTR SS:[EBP-1C], 10
00401084	.^ 72 CE	JB SHORT Crackme#.00401054

위 부분은 [EBP-1C]에 있는 값과 0x10을 비교하는 부분인데 [EBP-1C]에는 현재까지 읽어 들인 글자의 개수를 나타내므로 사용자가 입력한 값의 길이가 16글자(0x10)보다 작은지 확인하는 부분이다. 16글자보다 많은 경우는 00401054로 분기하지 않고 루프문을 빠져 나오게 된다. 즉, 우리가 입력한 글자를 16글자까지만 인식을 하게 된다.

루프를 빠져 나온 후 레지스터에 저장된 값과 헥사코드 정보에 대한 부분을 확인해 보도록 하겠다.

Address	Hex dump	ASCII
0013FF40	3E 11 40 00 88 70 40 00 59 10 40 00 71 77 65 72	>@. @. Y+@. qwer
0013FF50	74 79 75 69 6F 70 0A 00 00 00 00 00 00 00 00 00	tyuiop.....
0013FF60	00 00 00 00 0B 00 00 00 67 0A 00 00 70 70 69 6E	....d...g...ppin
0013FF70	72 74 72 69 70 6F 77 65 00 00 00 00 0A 20 40 00	rtripowe....@.
0013FF80	C0 FF 13 00 E5 12 40 00 01 00 00 00 30 0F 41 00	?!!.?@. r....0@.
0013FF90	90 OE 41 00 00 00 00 00 00 00 00 E0 FD 7F	?A.....

## Reverse Engineering

레지스터에 저장된 값은 다음과 같다.

Registers (FPU)	
EAX	0000000B
ECX	0000000A
EDX	0041050A
EBX	7FFDE000
ESP	0013FF4C ASCII "qwertyuiop"
EBP	0013FF80
ESI	00000000
EDI	00000000

EAX에 입력한 글자 수(0x0B)가 저장되어 있고 ESP에는 사용자가 입력한 글자를 가지고 있는 주소가 저장되어 있다.

여기까지 알아낸 Crackme#1을 풀기 위한 조건은 16글자 이하라는 것과 EAX에 사용자가 입력한 글자의 개수를 저장해두었다는 것이다. 물론 EAX에 있는 이 값은 어디론가 옮겨져 사용될 것이다. 왜냐하면 다음 루틴에서도 EAX를 써야 하기 때문이다.

루프문을 빠져 나오면 다음과 같은 코드를 볼 수 있다.

00401086	> 8D45 CC	LEA EAX, DWORD PTR SS:[EBP-34]
00401089	. 8945 F8	MOV DWORD PTR SS:[EBP-8], EAX
0040108C	. C745 E0 0000	MOV DWORD PTR SS:[EBP-20], 0
00401093	. C745 DC 0300	MOV DWORD PTR SS:[EBP-24], 3

EBP-34의 주소값을 EAX에 로드하고 EAX에 있는 값을 [EBP-8]의 위치로 옮긴다. 그리고 [EBP-20]와 [EBP-24]는 각각 0과 3이라는 값을 저장한다. 아마도 다음 작업을 위한 준비 단계일 것이다.

[EBP-34]의 주소값을 확인해보니 0013FF4C이고 이 주소에는 사용자가 입력했던 값이 저장되어 있다.

위 코드 실행 후 다음 코드는 004010AE로 분기하는 코드이다. 004010AE로 이동해 보도록 하겠다.

Address	Hex dump	Disassembly
0040109A	.v EB 12	JMP SHORT Crackme#.004010AE
0040109C	> 8B4D E0	MOV ECX, DWORD PTR SS:[EBP-20]
0040109F	. 83C1 04	ADD ECX, 4
004010A2	. 894D E0	MOV DWORD PTR SS:[EBP-20], ECX
004010A5	. 8B55 DC	MOV EDX, DWORD PTR SS:[EBP-24]
004010A8	. 83EA 01	SUB EDX, 1
004010AB	. 8955 DC	MOV DWORD PTR SS:[EBP-24], EDX
004010AE	> 837D E0 0D	CMP DWORD PTR SS:[EBP-20], 0D
004010B2	.v 73 28	JNB SHORT Crackme#.004010DC

[EBP-20]에 있는 값과 0x0D를 비교하여 [EBP-20]의 값이 0x0D보다 적지 않으면 004010DC로 분기한다. 즉, [EBP-20]에 있는 값이 0x0D보다 크거나 같으면 분기한다는 의미이다. 그런데 바로 앞에서

[EBP-20]에 0을 저장하였기 때문에 분기하지 않고 다음 코드를 실행하게 된다.

Address	Hex dump	Disassembly
0040109A	. EB 12	JMP SHORT Crackme#.004010AE
0040109C	> 8B4D E0	MOV ECX, DWORD PTR SS:[EBP-20]
0040109F	. 83C1 04	ADD ECX, 4
004010A2	. 894D E0	MOV DWORD PTR SS:[EBP-20], ECX
004010A5	. 8B55 DC	MOV EDX, DWORD PTR SS:[EBP-24]
004010A8	. 83EA 01	SUB EDX, 1
004010AB	. 8955 DC	MOV DWORD PTR SS:[EBP-24], EDX
004010AE	> 837D E0 OD	CMP DWORD PTR SS:[EBP-20], 0D
004010B2	.> 73 28	JNB SHORT Crackme#.004010DC
004010B4	. 8B45 E0	MOV EAX, DWORD PTR SS:[EBP-20]
004010B7	. C1E8 02	SHR EAX, 2
004010BA	. 8B4D F8	MOV ECX, DWORD PTR SS:[EBP-8]
004010BD	. 8B55 DC	MOV EDX, DWORD PTR SS:[EBP-24]
004010C0	. 8B0481	MOV EAX, DWORD PTR DS:[ECX+EAX*4]
004010C3	. 3B4495 E8	CMP EAX, DWORD PTR SS:[EBP+EDX*4-18]
004010C7	.> 74 11	JE SHORT Crackme#.004010DA
004010C9	. 68 4C704000	PUSH Crackme#.0040704C
004010CE	. E8 20000000	CALL Crackme#.004010F3
004010D3	. 83C4 04	ADD ESP, 4
004010D6	. 33C0	XOR EAX, EAX
004010D8	.> 74 15	JMP SHORT Crackme#.004010EF
004010DA	>> EB C0	JMP SHORT Crackme#.0040109C
004010DC	> 8D4D CC	LEA ECX, DWORD PTR SS:[EBP-34]
004010DF	. 51	PUSH ECX
004010E0	. 68 60704000	PUSH Crackme#.00407060
004010E5	. E8 09000000	CALL Crackme#.004010F3
004010EA	. 83C4 08	ADD ESP, 8
004010ED	. 33C0	XOR EAX, EAX
004010EF	> 8BE5	MOV ESP, EBP
004010F1	. 5D	POP EBP
004010F2	. C3	RETN

다음 코드는 004010B4 부터 시작한다.

```
/*4010B4*/ MOV EAX,DWORD PTR SS:[EBP-20] // EBP-20에 있는 값을 EAX로 저장
/*4010B7*/ SHR EAX,2 // EAX의 값을 오른쪽으로 2 바이트 쉬프트
/*4010BA*/ MOV ECX,DWORD PTR SS:[EBP-8] // EBP-8에 있는 값을 ECX에 저장
/*4010BD*/ MOV EDX,DWORD PTR SS:[EBP-24] // EBP-24에 있는 값을 EDX에 저장
/*4010C0*/ MOV EAX,DWORD PTR DS:[ECX+EAX*4] // ECX+EAX*4에 있는 값을 EAX에 저장
/*4010C3*/ CMP EAX,DWORD PTR SS:[EBP+EDX*4-18] // EBP+EDX*4-18의 값과 EAX의 값 비교
```

위 코드는 사용자가 입력한 패스워드와 프로그램이 가지고 있는 패스워드의 처음 4 글자를 비교하는 부분이다. 헥사코드 부분을 보도록 하자.

## Reverse Engineering

Address	Hex dump	ASCII
0013FF40	3E 11 40 00 88 70 40 00 59 10 40 00 71 77 65 72	>@. @. Y+@. qwer
0013FF50	74 79 75 69 6F 70 0A 00 00 00 00 00 00 00 00 00	tyuiop.....
0013FF60	00 00 00 00 0B 00 00 00 67 0A 00 00 70 70 69 6E	....d...g...ppin
0013FF70	72 74 72 69 70 6F 77 65 00 00 00 00 0A 20 40 00	rtri power .. @.
0013FF80	C0 FF 13 00 E5 1A 40 00 01 00 00 00 30 0F 41 00	?!!.?@. r...0DA.
0013FF90	A0 0E 41 00 00 00 00 00 00 00 00 00 C0 FD 7F	?A.....결□

[EBP+EDX\*4-18]

[ECX+EAX\*4]

004010B4에서부터 004010C3까지의 코드가 처음 실행될 때 ECX 값은 사용자가 입력한 값이 저장되어 있는 주소를 가지고 있으며 EAX는 0을 가지고 있다.([EBP-20]의 값이 0이므로) 그리고 EDX는 [EBP-24]에 있는 값을 가지고 있는데 이전 코드에서 3이라는 값을 저장했었다. 즉, EBP+EDX\*4-18을 하면  $0013FF80 + 3 * 4 - 18 = 0x0013FF80 + 0xC - 0x18 = 0x0013FF74$ 가 된다. 즉, qwer과 powe를 비교한다. 그리고 다음 코드에서 같으면 004010DA로 분기하고 그렇지 않으면 004010C9을 실행한다. 004010C9은 “Invalid Password”라는 메시지를 출력하면서 종료되는 루틴이 있는 곳이다. 만약 사용자가 첫 번째 4글자를 powe라고 입력했다고 가정하고 다음 코드를 살펴보도록 하겠다.

눈치가 빠른 리버서라면 지금 패스워드를 알아보았을 수도 있다. 하지만 좀 더 정확한 루틴을 파악하기 위해 다음 코드를 실행할 때 어떻게 되는지 살펴보자. 첫 번째 4글자가 같을 경우 004010DA로 분기하는데 여기서 다시 0040109C로 분기한다.

Address	Hex dump	Disassembly
0040109C	> 8B4D E0	MOV ECX, DWORD PTR SS:[EBP-20]
0040109F	. 83C1 04	ADD ECX, 4
004010A2	. 894D E0	MOV DWORD PTR SS:[EBP-20], ECX
004010A5	. 8B55 DC	MOV EDX, DWORD PTR SS:[EBP-24]
004010A8	. 83EA 01	SUB EDX, 1
004010AB	. 8955 DC	MOV DWORD PTR SS:[EBP-24], EDX
004010AE	> 837D E0 0D	CMP DWORD PTR SS:[EBP-20], 0D
004010B2	. v 73 28	JNB SHORT Crackme#.004010DC

```

/*40109C*/ MOV ECX,DWORD PTR SS:[EBP-20] // EBP-20에 있는 값을 ECX에 저장
/*40109F*/ ADD ECX,4 // ECX의 값을 4 증가
/*4010A2*/ MOV DWORD PTR SS:[EBP-20],ECX // ECX의 값을 EBP-20에 저장
/*4010A5*/ MOV EDX,DWORD PTR SS:[EBP-24] // EBP-24에 있는 값을 EDX에 저장
/*4010A8*/ SUB EDX,1 // EDX에서 1 감소
/*4010AB*/ MOV DWORD PTR SS:[EBP-24],EDX // EDX의 값을 EBP-24로 저장
/*4010AE*/ CMP DWORD PTR SS:[EBP-20],0D // EBP-20의 값과 0x0D 비교

```

## Reverse Engineering

앞의 코드를 보면 [EBP-20]에는 0이 들어 있기 때문에 ECX를 0으로 초기화한 후 4를 더하고 다시 그 값을 [EBP-20]에 저장한다. 그리고 [EBP-24]에 있는 값을 EDX에 저장한 후 1을 빼고 다시 이 값을 [EBP-24]에 저장한다. 결국 [EBP-20]에는 ECX 값인 4가 들어가고 [EBP-24]에는 EDX 값인 2가 들어간다. 그리고 [EBP-20]은 계속 4씩 증가하는데 0x0D와 비교하는 것은 프로그램이 가지고 있는 패스워드가 13글자이기 때문에 0x0D(13)보다 크면 004010DC로 분기하여 정상적인 패스워드를 출력하게 된다.

그리고 앞서 보았던 004010B4 부터 004010C3 까지 코드를 실행하는데 이때 [EBP-20]에 있던 4를 EAX에 넣은 후 SHR을 통해서 1로 변경이 된다. 그리고 [EBP-8]에 있는 주소를 ECX에 넣고 [EBP-24]에 있는 값을 다시 EDX에 넣는다.

이제 비교할 두 개의 값을 계산을 해보자.

$$ECX + EAX * 4 = 0x0013FF4C + 1 * 4 = 0x0013FF50$$

$$EBP + EDX * 4 - 18 = 0x0013FF80 + 2 * 4 - 0x18 = 0x0013FF80 + 0x08 - 0x18 = 0x0013FF70$$

헥사코드가 있는 부분을 통해서 어떤 값인지 확인해보겠다.

Address	Hex dump	ASCII
0013FF40	3E 11 40 00 88 70 40 00 59 10 40 00 71 77 65 72 >@. @.Y-@.qwer	
0013FF50	74 79 75 69 6F 70 0A 00 00 00 00 00 02 00 00 00 tyuibp.....!....	
0013FF60	04 00 00 00 0B 00 00 00 67 0A 00 00 70 70 69 6E -...d...g...ppin	
0013FF70	72 74 72 69 70 6F 77 65 4C FF 13 00 0A 20 40 00 rtripowle !...@.	
0013FF80	C0 FF 13 00 E5 12 40 00 01 00 00 00 30 0F 41 00 ?!.?@. -...03A.	
0013FF90	A0 0E 41 00 00 00 00 00 00 00 00 00 C0 FD 7F ?A.....!...?@.!	

패턴을 확인해보니 프로그램에서 요구하는 패스워드는 4 글자씩 분할되어 거꾸로 저장이 되어 있고 비교할 때 뒤에서부터 4 글자씩 가져와서 사용자가 입력한 값과 비교를 한다. 이렇게 알아낸 패스워드를 입력해보도록 하자.

```
D:\W00_target>Crackme#1.exe
Please enter the password:
powertripping
The password is powertripping
```

### 4-3. KeygenMe 분석 실습

#### KeygenMe 실습



- 인증을 위한 Key 값 요구시 Key 값 찾기
- Key 값을 생성하는 루틴 확인
- Keygen 작성

#### Student Notes

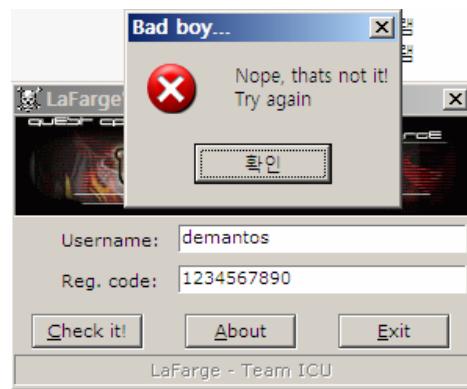
이번에는 단순히 키값을 찾는 것뿐만 아니라 키값을 만들어 내는 Keygenme에 대해서 분석해보도록 하겠다. keygenme는 말 그대로 키를 생성하는 프로그램으로 직접 C나 C++과 같은 프로그래밍 언어를 이용하여 작성할 수도 있고 내부 키젠이라고 해서 프로그램 내부에 키값을 생성하는 루틴을 찾아서 사용자가 잘못 입력하면 정상적인 키값을 보여줄 수 있도록 프로그램을 패치하는 방법을 사용하기도 한다.

본 교재에서는 내부 키젠을 만드는 방법에 대해서 알아보도록 하겠다.

## keygenme#1

이번에 분석 할 예제는 사용자 이름과 인증 코드를 입력하는 문제이다. 본 교재에서는 키젠을 따로 만드는 것이 아닌 내부 키젠을 만드는 방법에 대해서 알아보겠다. 내부키젘이라는 것은 키젠 프로그램을 별도로 만들지 않고 잘못된 인증 코드를 입력할 경우 올바른 인증 코드를 출력하게 해주게끔 해당 프로그램을 패치하는 것을 말한다.

그럼 프로그램을 실행해보도록 하겠다. 프로그램을 실행하고 사용자 이름과 인증 코드를 입력한다.



위와 같이 정상적인 인증 코드를 입력하지 않으면 “Nope, that’s not it! Try again”이라는 메시지를 출력한다. 그럼 디버거를 통해서 파일을 오픈해 보겠다. 먼저 위에 보이는 에러 메시지가 있는 곳을 찾아보겠다. Search for → All referenced text strings를 클릭하면 다음과 같다.

Address	Disassembly	Text string
00401027	CALL <JMP.&kernel32.Sizeof(Initial CPU selection)	
0040106D	ASCII "Dont look at my "	ASCII "#1000"
0040107D	ASCII " fucking code !!!"	ASCII "Bad boy..."
0040108D	ASCII "!",0	ASCII "Username must have at least 4 chars..."
00401098	PUSH keygenme.00406200	ASCII "_r <0<1-ZZ[15,^"
00401158	PUSH keygenme.00406337	ASCII "_r <0<1-ZZ[15,^"
0040115D	PUSH keygenme.0040620A	ASCII "Bad boy..."
0040116E	PUSH keygenme.00406BA4	ASCII "Ummm, no serial entered! U have brain,"
00401192	PUSH keygenme.00406BA4	ASCII "Good boy..."
00401267	PUSH keygenme.00406337	ASCII "Yep. thats the right code! Go write a k
0040126C	PUSH keygenme.00406342	ASCII "Bad boy..."
00401290	PUSH keygenme.0040630C	ASCII "Nope, thats not it!Try again"
00401295	PUSH keygenme.004062DD	ASCII "About..."
004012AA	PUSH keygenme.00406337	ASCII "Coded by LaFarge / ICUProtection: Cust
004012AF	PUSH keygenme.00406318	
004012E4	PUSH keygenme.004062D4	
004012E9	PUSH keygenme.00406231	

우리가 보았던 메시지뿐만 아니라 성공 메시지(Good boy...)도 보인다. 성공 메시지가 있는 부분을 더블 클릭하여 해당 코드로 이동하겠다.

## Reverse Engineering

Address	Hex dump	Disassembly	Comment
0040125C	. E8 DB000000	CALL <JMP.&user32.SendMessageA>	SendMessageA
00401261	. 0BC0	OR EAX, EAX	
00401263	. v 75 16	JNZ SHORT keygenme.00401278	
00401265	. 6A 10	PUSH 10	
00401267	. 68 37634000	PUSH keygenme.00406337	
0040126C	. 68 42634000	PUSH keygenme.00406342	
00401271	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00401274	. E8 B7000000	CALL <JMP.&user32.MessageBoxA>	
00401279	. v EB 41	JMP SHORT keygenme.004012BC	
0040127B	> 68 84654000	PUSH keygenme.00406584	
00401280	. 68 846B4000	PUSH keygenme.00406B84	
00401285	. E8 E8000000	CALL <JMP.&kernel32.lstrcmpA>	
0040128A	. 0BC0	OR EAX, EAX	
0040128C	. v 75 1A	JNZ SHORT keygenme.004012A8	
0040128E	. 6A 40	PUSH 40	
00401290	. 68 0C634000	PUSH keygenme.0040630C	
00401295	. 68 DD624000	PUSH keygenme.004062DD	
0040129A	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
0040129D	. E8 8E000000	CALL <JMP.&user32.MessageBoxA>	
004012A2	. C9	LEAVE	
004012A3	. C2 1000	RETN 10	
004012A6	. v EB 14	JMP SHORT keygenme.004012BC	
004012A8	> 6A 10	PUSH 10	
004012AA	. 68 37634000	PUSH keygenme.00406337	
004012AF	. 68 18634000	PUSH keygenme.00406318	
004012B4	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
004012B7	. E8 74000000	CALL <JMP.&user32.MessageBoxA>	

성공 메시지가 있는 코드로 분기할 것인지 에러 메시지가 있는 코드로 분기할 것인지를 결정하는 부분은 0040128A에 있는 OR EAX, EAX이다. 그리고 그 윗 쪽으로 lstrcmpA라는 함수가 보이고 이 함수에 대해서 우리가 입력한 값과 프로그램이 가지고 있는(또는 계산한) 값을 비교할 것이라고 예상할 수 있다. 그렇다면 우리가 입력한 값은 어디에 저장되어 있으며 프로그램이 가지고 있는 값은 어디에 저장되어 있는지 찾아보도록 하자. 위 코드에서 조금 더 올라가면 다음과 같은 부분을 만날 수 있다.

Address	Hex dump	Disassembly	Comment
00401248	. 6A 64	PUSH 64	ControlID = 64 (100.)
0040124A	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	hWnd
0040124D	. E8 D2000000	CALL <JMP.&user32.GetDlgItem>	GetDlgItem
00401252	. 68 84654000	PUSH keygenme.00406584	TParam = 406584
00401259	. 6A 0D	PUSH 0D	wParam = 40
0040125B	. 50	PUSH EAX	Message = WM_GETTEXT
0040125C	. E8 DB000000	CALL <JMP.&user32.SendMessageA>	hWnd
00401261	. 0BC0	OR EAX, EAX	SendMessageA
00401263	. v 75 16	JNZ SHORT keygenme.00401278	
00401265	. 6A 10	PUSH 10	
00401267	. 68 37634000	PUSH keygenme.00406337	
0040126C	. 68 42634000	PUSH keygenme.00406342	
00401271	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
00401274	. E8 B7000000	CALL <JMP.&user32.MessageBoxA>	
00401279	. v EB 41	JMP SHORT keygenme.004012BC	
0040127B	> 68 84654000	PUSH keygenme.00406584	
00401280	. 68 846B4000	PUSH keygenme.00406B84	
00401285	. E8 E8000000	CALL <JMP.&kernel32.lstrcmpA>	
0040128A	. 0BC0	OR EAX, EAX	
0040128C	. v 75 1A	JNZ SHORT keygenme.004012A8	
0040128E	. 6A 40	PUSH 40	
00401290	. 68 0C634000	PUSH keygenme.0040630C	
00401295	. 68 DD624000	PUSH keygenme.004062DD	
0040129A	. FF75 08	PUSH DWORD PTR SS:[EBP+8]	
0040129D	. E8 8E000000	CALL <JMP.&user32.MessageBoxA>	
004012A2	. C9	LEAVE	
004012A3	. C2 1000	RETN 10	

## Reverse Engineering

00401248에 브레이크포인트를 걸고 F9를 눌러 실행해보자. 그리고 사용자 이름과 인증코드를 입력하고 Check It을 누르면 00401248에서 멈추게 된다. 그리고 F8을 눌러서 코드를 한 줄씩 실행하다가 00401252주소에 있는 코드를 실행하면 스택에서 다음과 같은 값을 볼 수가 있다.

Address	Value	Comment
0013FB90	00406584	ASCII "JXCSUSIQXNUICPRU"
0013FB94	0013FBA0	
0013FB98	00000000	
0013FB9C	000003EC	

/\*401252\*/ PUSH keygenme.00406584

Address	Hex dump	ASCII
00406584	4A 58 43 53 55 53 49 51 58 4E 55 49 43 50 52 55	JXCSUSIQXNUICPRU
00406594	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	.....

우리는 이것을 인증코드라고 예상할 수 있다. 그리고 어디선가 이 값을 만들어 냈을 것이다. 일단 앞에서 진행했던 코드들을 계속 실행해보겠다.

0040127B	> l <sub>68</sub> 84654000	PUSH keygenme.00406584	String2 = "1234567890"
00401280	. 68 846B4000	PUSH keygenme.00406B84	String1 = "JXCS-USIQ-XNUI-CPRU"
00401285	. E8 E8000000	CALL <JMP.&kernel32.1strcmpA>	1strcmpA

우리의 예상이 정확히 맞았다는 것을 알 수 있다. 00401285에서 스트링을 비교하는 `strcmp` 함수를 호출하는데 인자값으로 우리가 입력한 값인 “1234567890”과 스택에서 보았던 “JXCS-USIQ-XNUI-CPRU”을 받아 들여서 비교하고 있다. 그리고 OR EAX, EAX에 의해서 두 개의 인자값이 같으지 확인한다. 같다면 0040128E를 실행할 것이고 같지 않다면 004012A8을 실행할 것이다.

여기서 우리의 목적을 다시 한번 떠 올려보면 잘못된 인증코드를 입력하더라도 올바른 키값을 알아낼 수 있도록 키값을 메시지박스에 띄워주면 된다. 따라서 아래에 보이는 에러 메시지 코드를 키값이 출력될 수 있는 코드로 패치하면 된다.

그렇다면 어느 부분을 고쳐야 할 것인가? 처음에 보았던 화면을 떠올려 보면 “Nope, that's not it! Try again” 메시지가 출력이 되었었는데 이 메시지 대신 올바른 키값을 출력해주면 된다.

`MessageBoxA` 함수는 총 4개의 인자를 받아 들인다. API 함수를 찾아보면 다음과 같다.

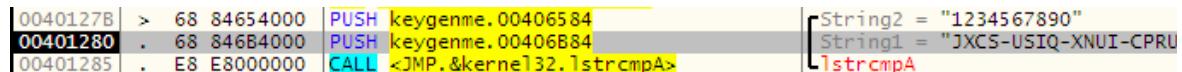
## Reverse Engineering

```
int MessageBox(
    HWND hWnd,           // handle of owner window
    LPCTSTR lpText,      // address of text in message box
    LPCTSTR lpCaption,   // address of title of message box
    UINT uType)          // style of message box
);
```

두 번째와 세 번째 인자가 실제 메시지박스에 출력된다는 것을 알았다. 그럼 코드를 살펴보자.

```
/*4012A8*/ PUSH 10
/*4012AA*/ PUSH keygenme.00406337
/*4012AF*/ PUSH keygenme.00406318
/*4012B4*/ PUSH DWORD PTR SS:[EBP+8]
/*4012B7*/ CALL <JMP.&user32.MessageBoxA>
```

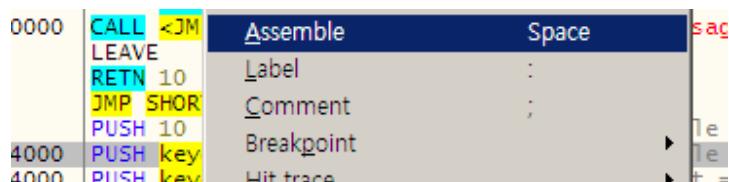
00406337 과 00406318에 있는 문자열들을 가져와서 출력을 해준다. 그럼 00406318에 있는 문자열을 키값을 가지고 있는 주소로 변경하면 여러 메시지 대신 키값을 출력해줄 것이다. 키값을 가지고 있는 주소는 lstrcmpA에서 확인할 수 있다.



00401278 > 68 84654000 | PUSH keygenme.00406584 | String2 = "1234567890"
00401280 . 68 846B4000 | PUSH keygenme.00406B84 | String1 = "JXCS-USIQ-XNUI-CPRU"
00401285 . E8 E8000000 | CALL <JMP.&kernel32.lstrcmpA> | lstrcmpA

00406B84에 키값이 들어 있는 것을 확인할 수 있다. 그럼 이제 코드를 수정해 보자.

해당 코드를 클릭한 후 스페이스를 누르면 어셈블리어 코드를 수정할 수 있다.



변경 전

```
/*4012AA*/ PUSH keygenme.00406337
/*4012AF*/ PUSH keygenme.00406318
```

변경 후

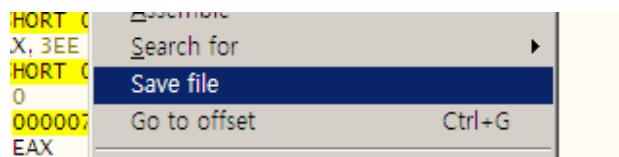
```
/*4012AA*/ PUSH keygenme.0040630C
/*4012AF*/ PUSH keygenme.00406B84
```

마우스 오른쪽 버튼을 클릭하고 Copy to executable → All modifications 를 클릭하여 수정된 코드를 저장

한다. 이 때 Copy all 을 선택한다.



그리고 창이 하나 새로 뜨면 마찬가지로 마우스 오른쪽 클릭해서 Save file 을 한 후 변경된 파일을 다른 이름으로 저장한다.



새로 저장한 파일을 실행한 후 사용자이름을 입력하고 인증코드는 아무값이나 입력한다. 그리고 에러 메시지 대신 키값이 출력이 되는지 확인해보자. 정확하게 패치가 되었는지 확인하려면 사용자 이름을 계속 바꿔가면서 입력해보면 된다.



이렇게 내부키젠을 생성할 수도 있고 또는 프로그램 내에서 키값을 생성하는 알고리즘을 파악한 후 실제 C 와 같은 프로그래밍 언어나 인라인 어셈을 이용하여 키젠을 별도로 작성할 수도 있다. 키젠을 생성하는 방법은 아래 URL 을 참고하기 바란다.

<http://dual5651.hacktizen.com/new/87>

**Reverse Engineering**



---

## Module 5 — MUP(Manual UnPack)

### Objectives

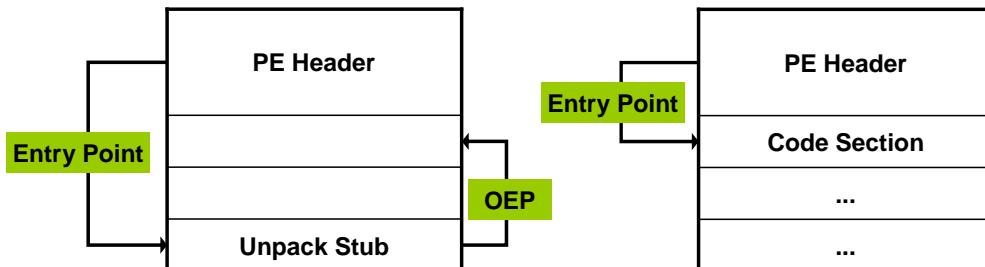
- Packgin / Unpacking
- Packing 종류
- MUP 실습
- MUP 를 위한 Ollydbg script 작성

## 5-1. Packing / Unpacking

### Packing / Unpacking



- Packing
  - 실행파일을 실행할 수 있는 형식 그대로 압축 또는 암호화 하는 과정 (EXE, DLL)
- Unpacking
  - 압축 또는 암호화되어 있는 실행파일(패킹된 파일)을 원상태로 해제 하는 과정



### Student Notes

“Pack”이라는 단어의 사전적 의미를 살펴보면 ‘포장하다’, ‘묶다’의 의미를 가지고 있다. 패킹은 실행파일을 특정 루틴에 의해서 압축을 하거나 암호화하는 과정을 말한다. 반대로 압축되어 있거나 암호화되어 있는 실행파일을 원래의 상태로 푸는 과정을 언패킹이라고 한다.

보통 패킹은 특정 섹션을 추가하여 그 추가된 섹션에 언패킹 루틴을 만들고 파일을 메모리에 로드할 때 메모리에 그대로 매핑을 한다. 패킹된 상태에서의 Entry Point는 언패킹 루틴이 있는 섹션을 가리키고 있고 파일을 실행하면 추가된 섹션에서 언패킹 루틴을 실행하고 코드 섹션으로 점프해서 원래의 Entry Point(OEP)로 진입하게 된다.

## 5-2. Packing 종류

### Packing 종류



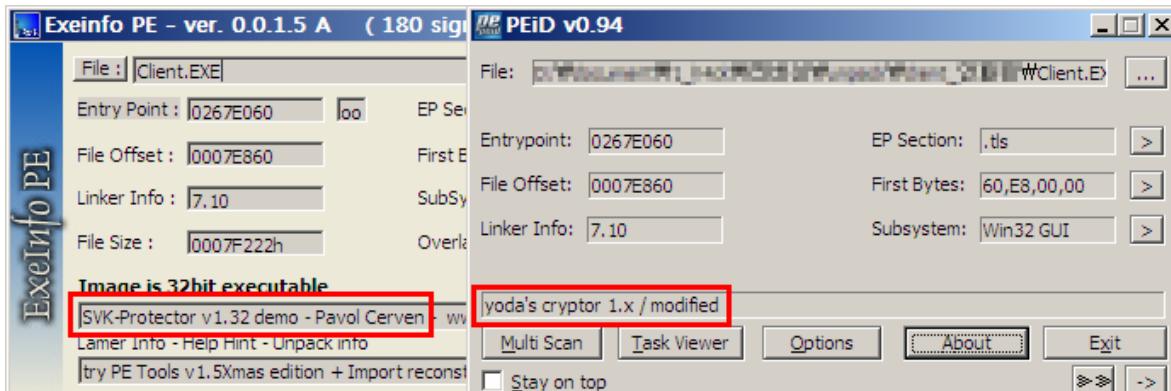
- UPX, ASPack, FSG 기타 등등 매우 많은 Packer들이 존재 종류는 다르지만 Packer의 기본적인 동작원리는 대부분 비슷하다.
- 대상파일을 패킹(암호화)한 후 프로그램의 시작점을 변경하여 그 부분부터 진행(복호화)을 함
- 그런 후 본래의 실행파일의 데이터를 메모리상에 복원해 프로그램을 실행되게 함

### Student Notes

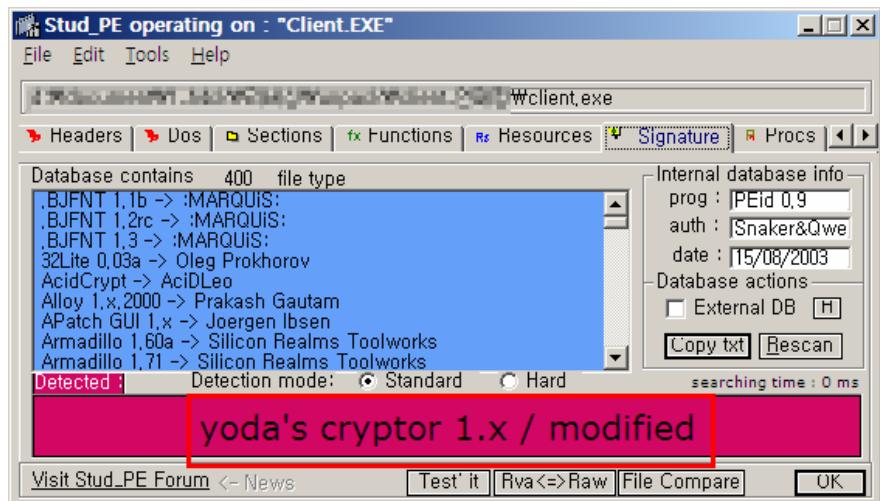
현재 알려져 있는 패커(패킹을 하는 프로그램)는 무수히 많지만 대부분 동작원리는 비슷하다. 경우에 따라서는 이중으로 패킹이 되어 있는 경우도 있다.

대상 프로그램이 패킹이 되었는지 안되어 있는지는 PEiD 나 EXEInfo 와 같은 프로그램을 사용하면 확인 할 수 있다. 하지만 이 프로그램을 통해서 나온 결과가 100% 정확한 것은 아니다. 이유는 패킹 여부를 확인하는 프로그램들이 시그내처 기반으로 검사를 하기 때문이다. 그리고 간혹 같은 파일을 다르게 보여 주는 경우도 있다.

## Reverse Engineering



위에서 보는 것과 같이 같은 파일임에도 불구하고 EXEInfo 는 SVK-Protector v1.32 demo 라고 이야기하고 있고 PEiD 는 yoda's cryptor 1.x / modified 라고 이야기하고 있다. Stud\_PE로 확인해보면 yoda's cryptor 1.x / modified 라고 나온다.



그렇다고 해서 이 파일이 yoda's cryptor 로 패킹이 되었다고 장담할 수는 없다. 시그내처 기반으로 패커를 판단하기 때문에 A라는 패킹 프로그램의 특징을 그대로 살려서 패킹을 하는데 B라는 패킹 프로그램을 사용했다면 위와 같은 프로그램들은 A라는 패커라고 알려줄 것이다. 하지만 실제로는 B라는 패커를 사용해서 프로그램을 패킹한 것이라는 의미이다.

그렇기 때문에 직접 수작업으로 패킹을 푸는 작업(언패킹)을 통해서 실제 OEP를 찾는 방법을 사용하기도 한다.

### 5-3. MUP 실습

#### MUP 실습



##### 1. Packing 여부 확인

PEiD나 EXEInfo와 같은 프로그램 사용

##### 2. OEP를 찾는다.

##### 3. 메모리 덤프

메모리에 로드된 후에는 unpack이 되므로

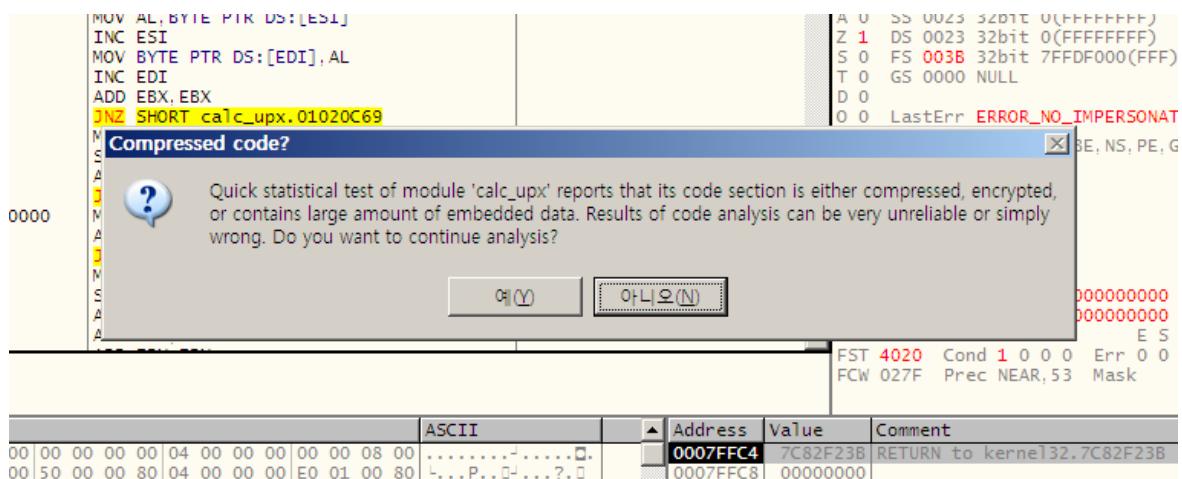
##### 4. IAT 복구

### Student Notes

MUP는 직접 패킹을 풀고 OEP를 찾은 후 덤프를 뜯 후 IAT를 복구시키는 과정을 거친다. 패킹된 프로그램을 실행하면 패킹되지 않은 프로그램과 동일하게 동작하기 때문에 프로그램 실행 후에는 정상적인 코드들이 메모리에 로드가 된다. 메모리에 로드된 후 덤프를 하게 되면 정상적인 프로그램이 덤프가 되지만 IAT는 정상적이지 않다. 따라서 IAT를 복구하는 과정을 거쳐야 한다. 이번 장에서 실습할 파일은 UPX로 패킹되어 있는 파일이다.

## Reverse Engineering

보통 패킹이 되었다는 것은 디버거로 파일을 오픈할 때 알 수 있다.



위와 같이 “Compressed code?”라는 팝업창 뜨면 패킹이 되어 있다는 것을 의심할 수 있다. 그리고 디버거로 패킹된 프로그램을 오픈하면 엔트리 포인트에서 다음과 같은 코드를 확인할 수 있다.

Address	Hex dump	Disassembly
01020C40	60	PUSHAD
01020C41	BE 00A00101	MOV ESI, calc_upx.0101A000

PUSHAD는 범용 레지스터에 들어 있는 값들을 전부 스택에 저장하는 명령이다. 이것이 바로 패킹을 했다는 증거이기도 하다. 패킹이 된 프로그램은 패킹이 되지 않은 프로그램과 동일하게 동작을 한다. 정상적으로 프로그램이 동작하기 위해서는 언패킹 과정을 거치고 OEP로 가서 실제 코드를 실행해야 하기 때문에 프로그램 시작 당시 모든 레지스터의 값들을 스택에 저장하고 언패킹 루틴을 거친 후 다시 POPAD에 의해 스택에 저장되어 있던 범용 레지스터들을 가져와서 실제 코드를 실행하게 된다.

PUSHAD를 했을 때 스택의 변화를 살펴보면 범용레지스터 값들이 스택에 저장되는 것을 확인할 수 있다.

### PUSHAD 실행 전

Address	Hex dump	Disassembly		
01020C40	60	PUSHAD		
01020C41	BE 00A00101	MOV ESI, calc_upx.0101A000		
Registers (FPU)		Address	Value	Comment
EAX	00000000	0007FFC4	7C82F23B	RETURN to kernel32.7C82F23
ECX	0007FFB0	0007FFC8	00000000	
EDX	7C9685EC	ntdll.KiFastSystemCallRet	0007FFCC	00000000
EBX	7FFDB000	0007FFD0	7FFDB000	
ESP	0007FFC4	0007FFD4	00000000	
EBP	0007FFF0	0007FFD8	0007FFC8	
ESI	00000000	0007FFDC	B8602CE4	
EDI	00000000			

### PUSHAD 실행 후

Address	Hex dump	Disassembly		
01020C40	60	PUSHAD		
01020C41	BE 00A00101	MOV ESI, calc_upx.0101A000		
Registers (FPU)		Address	Value	Comment
EAX	00000000	0007FFA4	00000000	
ECX	0007FFB0	0007FFA8	00000000	
EDX	7C9685EC	ntdll.KiFastSystemCallRet	0007FFAC	0007FFF0
EBX	7FFDB000	0007FFB0	0007FFC4	
ESP	0007FFA4	0007FFB4	7FFDB000	
EBP	0007FFF0	0007FFB8	7C9685EC	ntdll.KiFastSystemCallRet
ESI	00000000	0007FFBC	0007FFB0	
EDI	00000000	0007FFC0	00000000	
		0007FFC4	7C82F23B	RETURN to kernel32.7C82F23

POPAD를 통해 저장되어 있던 범용 레지스터들을 꺼내온 후 JMP 구문을 통해서 어디론가 점프하는 것을 볼 수 있다.

Address	Hex dump	Disassembly
01020D8E	61	POPAD
01020D8F	- E9 E116FFFF	JMP calc_upx.01012475

JMP 구문을 따라 F8로 진행을 계속하면 다음과 같이 OEP를 찾을 수 있다.

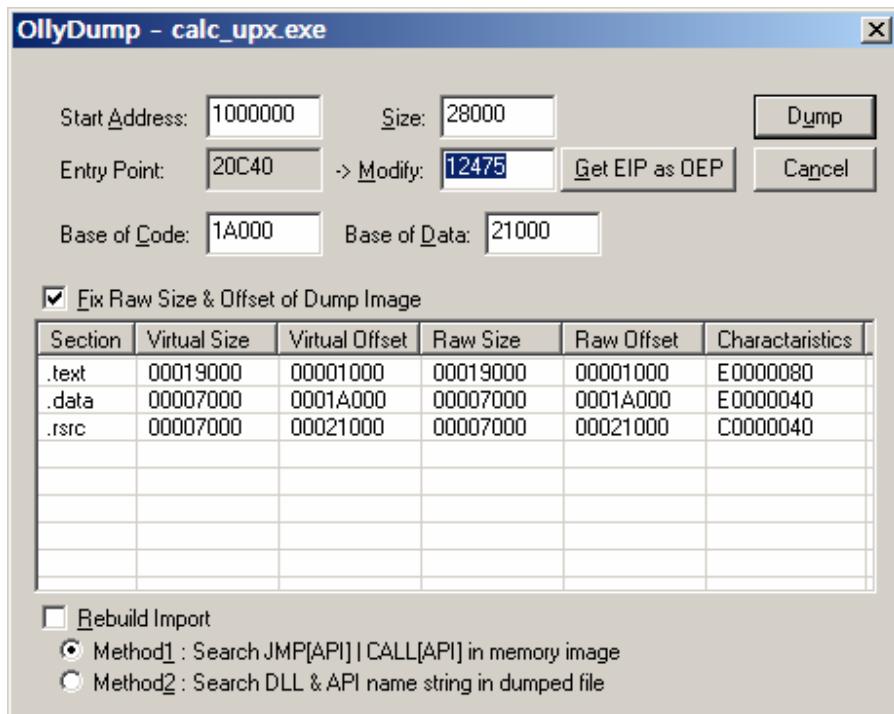
CPU - main thread, module calc_upx		
Address	Hex dump	Disassembly
01012475	6A 70	PUSH 70
01012477	68 E0150001	PUSH calc_upx.010015E0
0101247C	E8 47030000	CALL calc_upx.010127C8
01012481	33DB	XOR EBX, EBX
01012483	53	PUSH EBX
01012484	8B3D 20100001	MOV EDI, DWORD PTR DS:[1001020]
0101248A	FFD7	CALL EDI
0101248C	66:8138 4D5A	CMP WORD PTR DS:[EAX], 5A4D
01012491	v 75 1F	JNZ SHORT calc_upx.010124B2
01012493	8B48 3C	MOV ECX, DWORD PTR DS:[EAX+3C]
01012496	03C8	ADD ECX, EAX

## Reverse Engineering

패킹되지 않은 프로그램을 디버거에서 오픈한 후 앞의 화면과 비교해보면 같다는 것을 알 수 있다.

Address	Hex dump	Disassembly
01012475	\$ 6A 70	PUSH 70
01012477	. 68 E0150001	PUSH calc.010015E0
0101247C	. E8 47030000	CALL calc.010127C8
01012481	. 33DB	XOR EBX, EBX
01012483	. 53	PUSH EBX
01012484	. 8B3D 20100000	MOV EDI, DWORD PTR DS:[<&KERNEL32.GetModi
0101248A	. FFD7	CALL EDI
0101248C	. 66:8138 4D5A	CMP WORD PTR DS:[EAX], 5A4D
01012491	.~ 75 1F	JNZ SHORT calc.010124B2
01012493	. 8B48 3C	MOV ECX, DWORD PTR DS:[EAX+3C]
01012496	. 03C8	ADD ECX, EAX

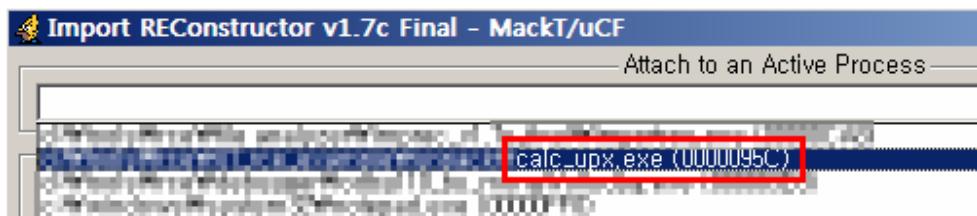
OEP를 찾았으니 덤프를 하도록 하겠다. 마우스 오른쪽 클릭 후 Dump debugged process라는 메뉴를 선택한다. 이 메뉴는 Olly Dump 플러그인이 설치되어 있어야 한다.



현재 메모리에 로드된 상태를 덤프한 것이다. Entry Point 가 20C40에서 12475로 바뀐 것을 확인할 수 있다. 이 OEP를 잘 기억해두자. 그리고 아래쪽에 보면 Rebuild Import 메뉴는 체크를 해제한다. 이 메뉴는 디버거에서 IAT를 복구한다는 의미인데 정상적으로 복구가 안되는 경우가 있기 때문에 체크를 해제한다. IAT를 복구해야만 정상적으로 프로그램이 실행될 텐데 IAT 복구 메뉴를 체크를 해제하면 어떻게 IAT를 복구할 것인가? 걱정하지 마시길 바란다. IAT를 복구는 ImportREConstructor라는 프로그램을 이용할 것이다.

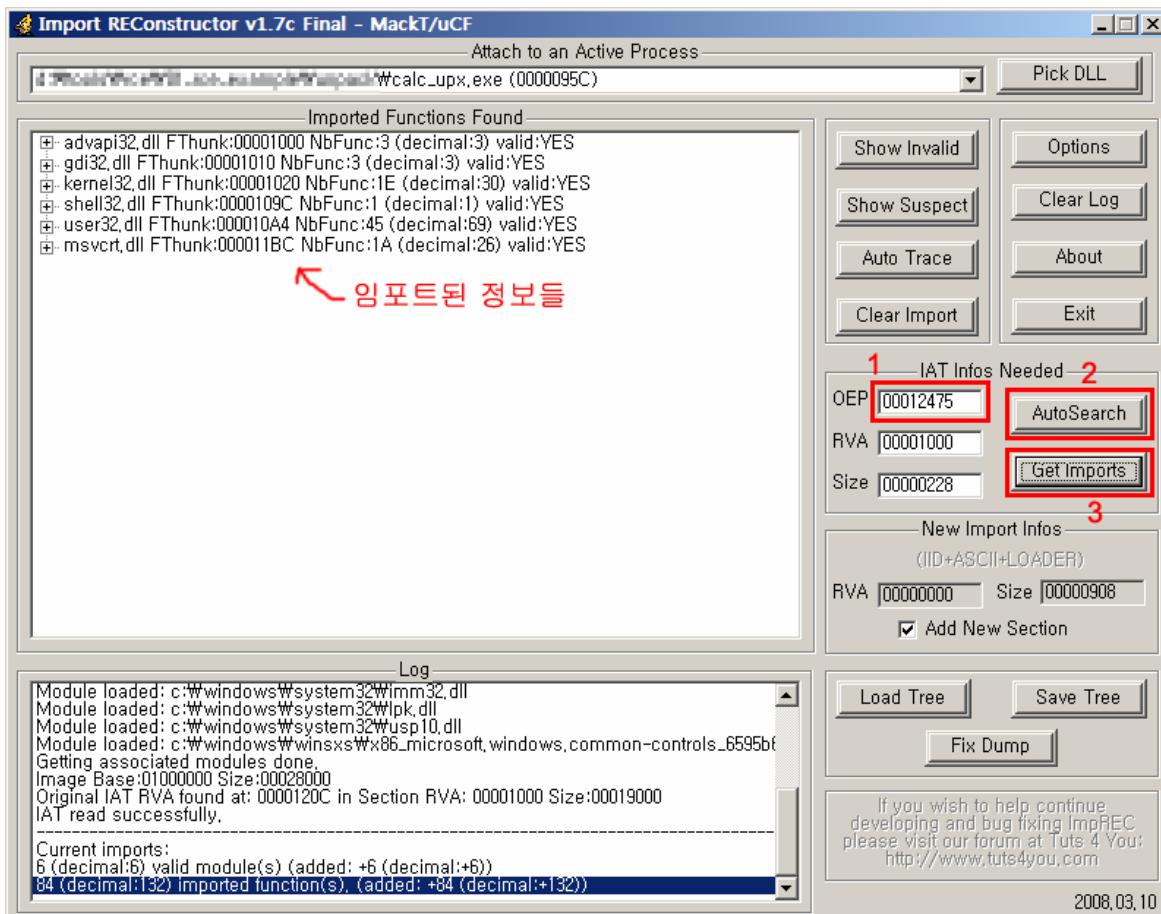
## Reverse Engineering

ImportREC 을 실행한 후 Attach to an Active Process에서 현재 MUP 작업 중인 파일을 선택한다.



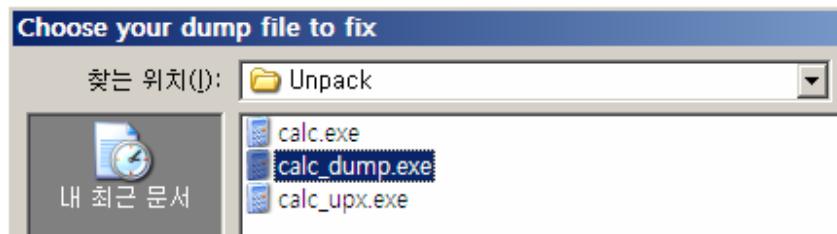
calc\_upx.exe 는 패킹되어 있는 파일이고 이 파일에 정상적인 IAT를 가지고 있을 것이다. 따라서 이 파일을 통해서 IAT 정보를 획득한 후 조금 전에 덤프한 파일에 적용시켜주는 과정을 거쳐 IAT를 복구할 것이다.

먼저 덤프할 때 보았던 OEP를 1 번 부분에 적어 넣고 AutoSearch 를 눌러서 검색을 한 후 Get Imports 를 눌러 인포트 테이블을 가져온다.



## Reverse Engineering

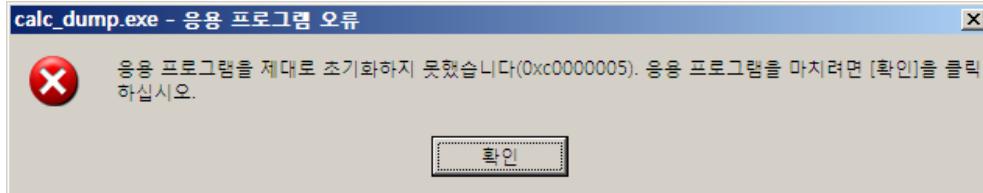
마지막으로 오른쪽 아래에 있는 Fix Dump 를 눌러서 덤프된 파일의 IAT 정보를 복구할 것이다.



저장을 하면 다음과 같이 calc\_dump\_.exe라는 파일명으로 언패킹되고 IAT가 복구된 파일이 생성된다.

이름	크기	종류
calc.exe	112KB	응용 프로그램
calc_dump.exe	160KB	응용 프로그램
calc_dump_.exe	164KB	응용 프로그램
calc_upx.exe	55KB	응용 프로그램

파일의 크기도 다른 것을 확인할 수 있다. calc\_dump.exe는 언패킹만 하고 IAT를 복구하지 않은 파일이기 때문에 실행이 되지 않는다.



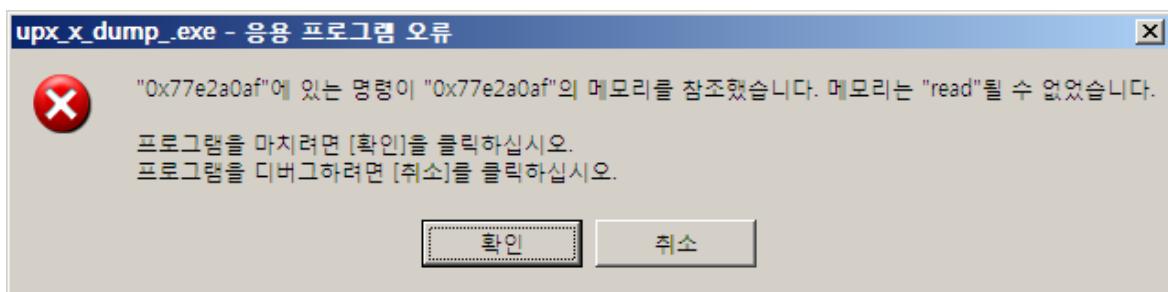
## IAT가 정상적으로 복구되지 않은 경우

PE 포맷 중 임포트 테이블을 살펴보자. 임포트 테이블의 마지막은 NULL 값(00000000)이 들어 있다. 이것을 보고 임포트 테이블의 끝이라는 것을 알 수 있다. ImportREC은 IAT 중에서 00000000 값이 나오면 IAT의 끝으로 판단하고 복구시 크기를 00000000 이 나오는 곳까지만 잡아서 자동으로 계산한다.

이럴 경우 어떤 문제가 발생할까? 정상적으로 임포트되어서 실행되어야 할 함수들이 임포트되지 않게 되고 결국 프로그램이 정상적으로 실행되지 않을 것이다.

## Reverse Engineering

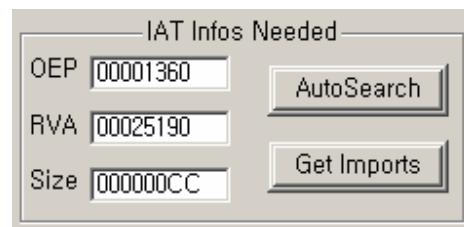
이번에는 이렇게 IAT 가 제대로 복구되지 않았을 경우 수작업으로 IAT 를 복구하는 과정에 대해서 알아보도록 하겠다.



이런 경우 IAT 가 제대로 복구되지 않아서 관련 함수를 “read” 할 수 없어서 발생하는 에러 메시지이다.

먼저 디버거를 통해 IAT 가 어디서부터 어디까지인지 확인해보도록 하자.

ImportREC 을 보면 IAT 에 대한 정보 중 RVA 값은 임포트 테이블의 시작주소를 나타내는 것이고 Size 가 임포트 테이블의 크기를 의미한다.



임포트 테이블의 RVA 값이 00025190 이므로 실제로 메모리상에서는 00425190 이 된다. 이 위치를 확인해보자.

Address	Value	Comment
00425190	00000000	
00425194	7C82474A	kernel32.GetModuleHandleA
00425198	7C81F897	kernel32.FlushFileBuffers
0042519C	7C8312CB	kernel32.SetStdHandle
004251A0	7C82568B	kernel32.SetFilePointer
004251A4	7C82B24B	kernel32.GetStringTypeW
004251A8	7C834A17	kernel32.GetStringTypeA
004251AC	7C823241	kernel32.LCMMapStringW
004251B0	7C8020DC	kernel32.GetStartupInfoA
004251B4	7C82B487	kernel32.GetCommandLineA
004251B8	7C8297CB	kernel32.GetVersion

Command : dd 425190 DD [address]

함수들의 이름이 보인다. ImportREC 은 여기서부터 0xCC 크기만큼 임포트 테이블으로 인식한 것이다.

## Reverse Engineering

0xCC 가 떨어진 곳을 임포트 테이블의 마지막으로 인식한 것이다.

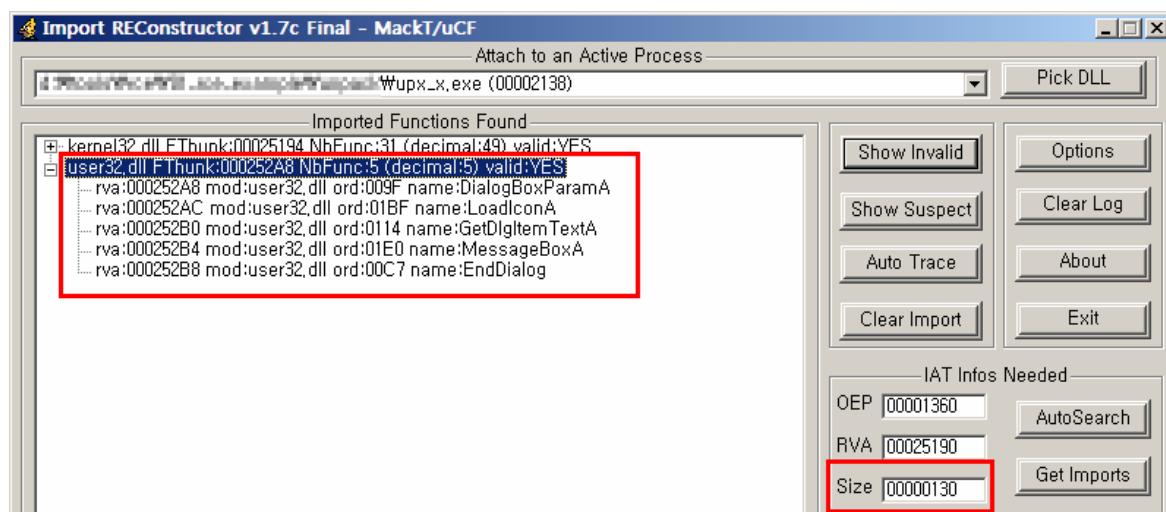
Address	Value	Comment
00425238	7C82794D	kernel32.GetACP
0042523C	7C8087FC	kernel32.GetOEMCP
00425240	7C969FD6	ntdll.RtlAllocateHeap
00425244	7C8245A9	kernel32.VirtualAlloc
00425248	7C96B0DC	ntdll.RtlReAllocateHeap
0042524C	7C822FD6	kernel32.MultiByteToWideChar
00425250	7C832825	kernel32.LCMMapStringA
00425254	7C823E6F	kernel32.CloseHandle
00425258	00000000	
0042525C	00000000	
00425260	00000000	

임포트 테이블의 마지막을 알리는 00000000 이 보인다. 아래로 계속 내려가 보자.

Address	Value	Comment
004252A0	00000000	
004252A4	00000000	
004252A8	77E2A0AF	USER32.DialogBoxParamA
004252AC	77E00AFF	USER32.LoadIconA
004252B0	77E3C516	USER32.GetDlgItemTextA
004252B4	77E2425F	USER32.MessageBoxA
004252B8	77DE97A3	USER32.EndDialog
004252BC	00000000	
004252C0	00000000	
004252C4	00000000	
004252C8	00000000	

또 다른 함수가 보인다. IAT 를 쪼개서 자동 복구를 방해하고 있는 것이다. 따라서 ImportREC에서 IAT 를 복구할 때 사이즈를 다시 계산해서 정확한 사이즈를 입력해주어야 한다. IAT 가 0x425190 부터 0x4252C0 까지이므로 사이즈는 0x130 이 된다.

OEP 와 사이즈를 다시 지정한 후 Get Import 해서 IAT 를 다시 불러온다.

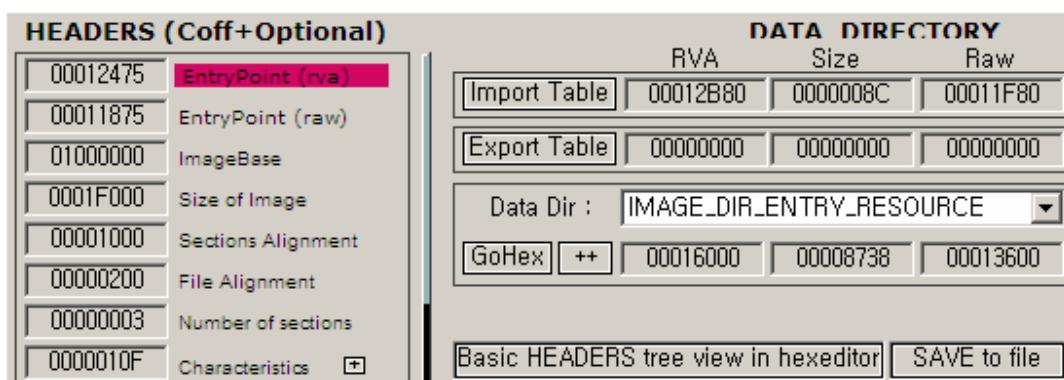


## Reverse Engineering

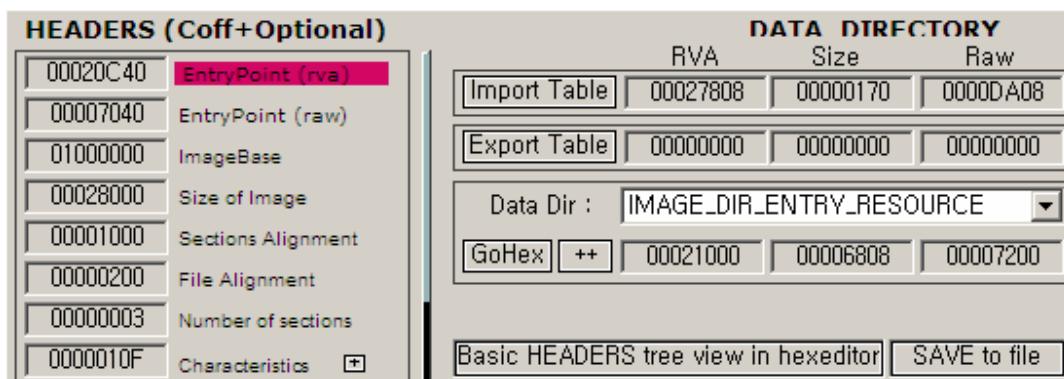
Fix Dump 를 눌러 언패킹한 파일을 지정하여 IAT 를 복구시켜준 후 파일을 실행하면 정상적으로 실행되는 것을 볼 수 있다.

정리를 해보면 언패킹을 하게 되면 원래의 엔트리 포인트(OEP)가 변경된다. 그리고 정상적으로 IAT 를 복구시켜줘야 하는데 ImportREC 과 같은 프로그램에서 자동으로 검색을 하기 때문에 정확하지 않을 수 있다. 마지막으로 패킹 전 프로그램과 패킹 후 프로그램이 어떤 차이를 보이는지 확인해보도록 하자.

### Before Packing



### After Packing



## 5-4. MUP 를 위한 Ollydbg script 작성

### MUP를 위한 Ollydbg script 작성



- MUP 과정을 스크립트로 작성
- Ollydbg script
  - 어셈블리어와 비슷한 형식으로 작성
  - 접두사, 접미사 없이 모든 숫자는 16진수로 인식함  
(10진수로 인식시키고자 할 경우 ':'을 붙임 ex. 10=16.)
  - 변수를 사용하기 위해서는 먼저 선언해야 함
  - 메모리 주소를 가리키기 위해서는 []를 사용함
  - 플래그 비트는 반드시 앞에 '!'을 붙임
  - 연속된 바이트를 하나로 묶어서 사용하기 위해서는 '#' 사이에  
헥사값으로 입력해야 함
  - DWORD값에 대해 '+', '-', '\*', '/', '&', '|', '^', '>', '<' 를 사용할 수 있음
  - 문자열에 대해 '+'를 사용할 수 있음

### Student Notes

앞서 살펴보았던 MUP 는 비교적 간단한 과정을 거쳐서 작업을 할 수 있었다. 하지만 더 복잡한 경우도 많을 것이고 패키에 따라서 언패킹을 하는 과정이 다르기도 하다. MUP 작업을 좀 더 편하게 하기 위해서 스크립트를 작성한 후 실행 할 수 있다. 물론 꼭 MUP 에서만 스크립트를 사용하는 것이 아니고 특정 작업을 하기 위한 스크립트를 작성하기도 한다.

Olly Script 를 사용하기 위해서는 플러그인이 필요하다. 인터넷을 통해 검색을 하면 여러 가지 스크립트를 획득할 수 있고 사용이 가능하다. 여기서는 스크립트를 사용하기 위한 몇 가지 명령어들에 대해서 알아보고 앞서 실습했던 MUP 를 수행할 수 있는 스크립트를 작성해보도록 하겠다.

## Olly Script

**내부 변수**

**\$result**

몇몇 스크립트 명령어들은 결과값을 리턴함  
EAX 레지스터리 대신에 임시변수를 이용함

**\$version**

Ollyscript 의 버전 정보를 나타냄

ex) cmp, \$version, "1.47" //1.47 버전보다 큰가?

**명령어**

**# INC "filename"**

다른 스크립트 파일의 명령어 복사

ex) # INC "text.txt"

**# LOG**

스크립트 결과를 LOG 창에 기록

OllyDbg Log 와 구분 짓기 위해 “→” 로 시작

**ADD dst, src**

문자열을 합쳐서 dst 에 저장

ex) add y, "Times"

**Alloc size**

size 만큼 메모리를 확보한 뒤 시작주소를 \$result 에 저장

ex) alloc 1000, free \$result 1000

**AI / AO**

Animate into / animate over 와 동일

**STI / STO**

Step into / step over 와 동일

**ESTI / ESTO**

Shift + F7 / Shift + F8 과 동일

## **Reverse Engineering**

### **TI / TO**

Trace into / trace over 와 동일

### **TC**

Close Run Trace 와 동일

### **TICND <condition> / TOCND <condition>**

Trace into / trace over 와 동일하나 조건을 만족하면 트레이스를 중지

ex) TICND "eip>40100A"

### **BC Address / BP Address**

Breakpoint Clear / BreakPoint

### **BPCND Address, Condition**

조건에 만족할 경우 BP 활성화

ex) bpcnd 40100A, "EAX==1"

### **BPHWC Address / BPHWCALL**

BP HW Clear / BP HW Clear All

### **BPHWS Address, pattern**

BP HW Set “r”(읽기) “w”(쓰기) “x”(실행) /BPHWS Address

ex) BPHWS 40100A, "r"

### **BPL Address, Expression**

BP of Logging

ex) bpl 40100A, "EAX"

### **BPLCND Address, Expression, Condition**

BP of Logging Condition

ex) bplcnd 40100A, "EAX", "ECX==0"

### **BPMC**

BP Memory Clear

**BPRM Address, size / BPWM Address, size**

BP on Read Memory / BP on Write Memory

ex) BPRM 40100A, FF

ex) BPWM 40100A, FF

**AN Address**

Ollydbg에서 “analyse”와 동일

**ASM Address, Instruction**

Ollydbg에서 assemble 명령과 동일

어셈블한 명령어의 길이가 바이트 단위로 \$result에 저장

ex) asm 40100A, “xor eax, eax”

**CMT address, comment(string of character)**

Ollydbg에서 주석 추가와 동일

ex) cmt eip, “<== OE”

**RUN / PAUSE**

Ollydbg에서 run / pause 와 동일

**RTR / RTU**

Ollydbg에서 “Execute till Return”(Ctrl + F9) / “Run till user code”(Alt + F9)와 동일

**AND dst, src / SUB dst, src**

어셈블리 명령어 and/sub 와 동일

ex) and eax, -1

**CMP dst, src / SCMP dst, src**

어셈블리 명령어 cmp / scmp 와 동일

ex) cmp eax, 1

**MOV dst, src**

어셈블리 명령어 mov 와 동일

**MUL dst, src / DIV dst, src**

어셈블리 명령어 mul/div 와 동일

## **Reverse Engineering**

**OR dst, src / XOR dst, src**

어셈블리 명령어 Or/xor 과 동일

**EOB Label / EOE Label**

Excute On Breakpoint / Execute On Exception

ex) EOB Label / EOE Label1

**FILL Address, Length, Value**

주소에서 길이만큼 입력한 값으로 채움

ex) fill 40100A,10,90

**Find Address, Content**

메모리 주소부터 특정 값을 검색한 후 일치한 값이 있을 경우 \$result에 주소값을, 없을 경우 0을 리턴

ex) find eip, # 6A??E8 #

**FINDOP Address, Content**

주소부터 특정 명령어 바이트로 시작하는 코드를 검색한 후 결과를 \$result에 저장

ex) findop 401000, #61# // find next POPAD

findop 401000, #6A??# // find next PUSH

**FINDMEM what [, Start Address]**

전체 메모리에서 특정값을 검색한 후 일치한 값이 있을 경우 \$result에 주소값을, 없을 경우 0을 리턴

ex) findmem #6A00E8#

findmem #6A00E8#, 00400000

**REPL Address, Find, Replace, Length**

주소의 명령어가 find 한 값과 같은지 비교한 후 같다면 길이만큼 바이트를 대체 (와일드카드 사용가능)

ex) repl eip, #?200 #, #?201 #, 10

**RET**

스크립트의 끝을 나타낸다

**REV val**

Val 값을 역바이트로 \$result에 저장

ex) rev 01020304 // \$result == 04030201

**EXEC/ENDE**

EXECute / END Execute, {}는 치환

```
ex) var x  
    var y  
    mov x, "eax"  
    mov y, "0DEADBEEF"  
    exec  
    mov {x}, {y}  
    mov ecx, {x}  
    ende  
ex) exec  
    push 0  
    call ExitProcess  
    ende  
    ret
```

**JA LABEL / JAE LABEL**

Cmp 이 후에 사용하여야 하며, 어셈블리 ja / jae 와 동일

**JB LABEL / JNB LABEL**

Cmp 이 후에 사용하여야 하며, 어셈블리 ja / jae 와 동일

**JE LABEL / JNE LABEL**

Cmp 이 후에 사용하여야 하며, 어셈블리 ja / jae 와 동일

**JMP LABEL**

어셈블리 JMP 와 동일

**LEN str**

Str 의 길이를 획득함

```
ex) len "NiceJump"  
    msg $RESULT
```

## Reverse Engineering

### MUP 를 위한 스크립트 작성

언패킹을 위한 스크립트를 작성할 때는 수작업의 과정에 해당하는 명령을 찾아서 작성해주면 된다.

앞서 보았던 UPX 언패킹을 생각해보자. PUSHAD를 통해 레지스터를 백업한 후 POPAD로 레지스터를 원상 복귀 시킨 후 OEP로 진입하는 것을 볼 수 있었다.

OEP에 진입하는 바로 그 시점에서 브레이크포인트를 걸어서 멈추게 해야 한다. 또는 OEP 진입 바로 전에 브레이크포인트를 걸어서 멈추게 한 후 Step Over(F8) 해서 OEP에 진입할 수도 있다. 그렇다면 OEP 진입 바로 전에 행해지는 행동은 어떤 것인가?

바로 스택에 백업되어 있던 레지스터들을 다시 가져오기 위해서 ESP에 접근하게 된다. 즉 스택에 접근해서 백업되었던 레지스터들을 복구한 후 OEP에 접근한다는 것이다. 그렇다면 이런 일련의 행동들을 앞에서 봤던 스크립트 명령어들로 구성해 보도록 하겠다.

```
STI          // Step into(F7), PUSHAD를 한번 실행시킨다.  
BPHWS esp, "r" // PUSHAD 후 POPAD 할 때 BP 설정  
RUN          // 프로그램 실행(F9)  
BPHWC        // 설정했던 BP에서 멈춘 후 하드웨어 BP 제거  
STO          // Step over(F8), 멈춘 후 한번 더 진행해서 OEP 진입  
CMT eip, "<- this is OEP" // 현재 위치에 코멘트  
RET          // 종료
```

위 예제는 비교적 간단한 예제이다. UPX에 의해 패킹이 되어 있을 경우 다른 패커에 비해 패턴이 간단하기 때문이다. 위 예제와 똑같이 OEP를 찾는 스크립트인데 약간 다르게 작성된 스크립트를 하나 더 보겠다.

```
EOB Break      // RUN 이후 브레이크 발생시 Break: 에서부터 스크립트 실행  
FINNDOP eip, #61# // OP 코드가 61인 명령어 검색. 즉 POPAD를 검색  
BPHWS $RESULT, "x" // 위 명령의 결과가 $RESULT에 저장되고 그 메모리 주소에  
RUN          // 있는 명령어가 실행되면 브레이크한다.  
  
Break:  
STO          // Step over(F8)  
STO  
BPHWC $RESULT // BP 해제  
RET          // 종료
```

---

## Module 6 — Anti-Reverse 기법

### Objectives

- 디버거 탐지 기법
- Breakpoint 탐지 기법
- TLS Callback
- Process Attach 기법

## 6-1. 디버거 탐지 기법

### 디버거 탐지 기법



**kernel32!IsDebuggerPresent**

**PEB!IsDebugged**

**PEB!NtGlobalFlags**

Heap flags

Vista anti-debug (no name)

**NtQueryInformationProcess**

**kernel32!CheckRemoteDebuggerPresent**

UnhandledExceptionFilter

NtSetInformationThread

kernel32!CloseHandle and NtClose

Self-debugging

Kernel-mode timers

User-mode timers

**kernel32!OutputDebugStringA**

Ctrl-C

Rogue Int3

"Ice" Breakpoint

Interrupt 2Dh

**Timestamp counters**

Popf and the trap flag

Stack Segment register

Debug registers manipulation

Context modification

**TLS-callback**

CC scanning

EntryPoint RVA set to 0

출처 : <http://www.securityfocus.com/infocus/1893>

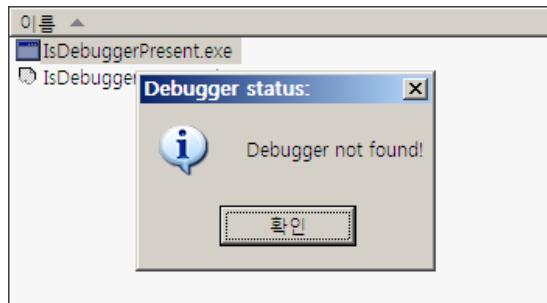
### Student Notes

자신의 프로그램이 디버거에 의해 어떻게 동작하는지 디버거들에 의해서 속속들이 들춰진다면 그리 기분이 썩 좋지는 않을 것이다. 혹은 상용 프로그램이 시리얼을 생성하는 방법들을 디버거를 통해 알아낸 후 키젠을 만들게 되면 이윤을 추구하는 회사로 써는 큰 손실이 될 수 있다. 악성코드 제작자들은 자신이 만든 악성코드를 누군가 디버깅하여 악성코드가 동작하는 과정이나 방식 등을 알아낸다면 제작자의 의도와 부합되지 않을 것이다. 이러한 이유로 자신이 만든 프로그램이 정상적으로 실행되는 것이 아니라 디버거를 통해서 실행되는 것을 탐지하여 프로그램을 종료하는 기법들을 디버거 탐지 기법이라고 한다.

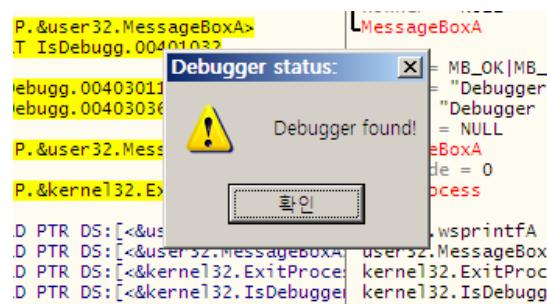
디버거를 탐지하는 기법들을 구현하는 것은 생각보다 간단하다. 디버거들은 위에 보이는 기법들을 눈에 익히고 디버깅 도중 디버거를 탐지하는 루틴이 보이면 그것을 우회하면 된다. 여기서는 몇 가지 기법들에 대해서 알아보도록 하겠다.

## kernel32!IsDebuggerPresent

가장 기본적인 디버거를 탐지하는 기법은 PEB(Process Environment Block)에서 BeingDebugged 플래그가 포함되어 있는지 확인하는 것이다. IsDebuggerPresent()는 디버거가 프로세스를 디버깅하고 있는지 플래그 값으로 확인한다.



IsDebuggerPresent에 함수를 사용한 프로그램을 그냥 실행시키면 위와 같이 Debugger not found!라는 메시지를 보여준다. 하지만 디버거를 통해서 프로그램을 실행시키면 다음과 같이 Debugger found!라는 메시지를 보여주게 된다.



디버거를 통해서 확인을 해보면 다음과 같다.

Address	Hex dump	Disassembly
00401000	\$ E8 47000000	CALL <JMP.&kernel32.IsDebuggerPresent>
00401005	. 83F8 01	CMP EAX, 1
00401008	.. 74 15	JE SHORT IsDebugg.0040101F

IsDebuggerPresent() 함수가 수행되고 그 결과값은 EAX에 저장이 된다. 만약 디버거가 탐지가 되면 IsDebuggerPresent() 함수가 1을 반환하고 디버거가 탐지 되지 않으면 0을 반환한다. CMP에 의해 비교를 한 후 같으면 디버거가 탐지되었다는 메시지를 출력해주는 루틴으로 이동한다. (IsDebugg.0040101F) 위 코드는 다음과 같이 작성할 수도 있다.

## Reverse Engineering

```
call IsDebuggerPresent  
test eax, eax  
jne @DebuggerDetected
```

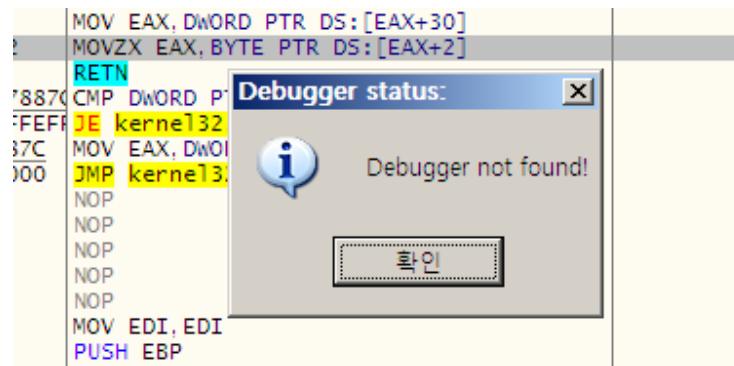
CALL 문을 F7 로 따라서 들어가보면 다음과 같은 코드를 볼 수 있다.

Address	Hex dump	Disassembly
7C81DA00	64:A1 18000000	MOV EAX, DWORD PTR FS:[18]
7C81DA06	8B40 30	MOV EAX, DWORD PTR DS:[EAX+30]
7C81DA09	0FB640 02	MOVZX EAX, BYTE PTR DS:[EAX+2]
7C81DA0D	C3	RETN

이것은 TEB(Thread Environment Block)에 접근하여 PEB 의 주소를 얻은 다음 PEB 를 검사하고 PEB + 2 위치에서 BeingDebugged 플래그를 확인하는 것이다. ⓠ BeingDebugged 플래그를 0 으로 설정하면 IsDebuggerPresent 탐지 기법을 우회할 수 있다.

Address	Hex dump
7FFDB000	00 00 01 00
7FFDB010	00 00 02 00

변경 후 F9 로 실행을 하면 다음과 같이 Debugger not found! 메시지를 확인할 수 있다.



## PEB!IsDebugged

PEB 의 두 번째 바이트를 참조해서 디버거를 탐지한다. 이것은 앞서 보았던 IsDebuggerPresent 와 내부적으로 동작하는 방식이 같다. 간단한 예제만 하나 보도록 하겠다.

```
mov eax, fs:[30h]  
mov eax, byte [eax+2]  
test eax, eax  
jne @DebuggerDetected
```

## PEB!NtGlobalFlags

PEB 의 68 번째 비트에 있는 NtGlobalFlags 를 이용하여 디버거를 탐지한다. 프로세스가 디버깅 중이면 NtGlobalFlags 의 값이 0x70 이고 정상적인 실행 중이라면 0x00 이다. 그리고 프로세스가 디버깅 중이라면 ntdll 의 힙 조작 루틴을 제어하는 몇몇 플래그들이 설정된다. 이 플래그들은 Heap 과 관련된 디버깅에 필요한 옵션들이다.

FLG_HEAP_ENABLE_TAIL_CHECK (Heap Tail Checking)	: 0x10
FLG_HEAP_ENABLE_FREE_CHECK (Heap Free Checking)	: 0x20
FLG_HEAP_VALIDATE_PARAMETERS (Heap Parameter Checking)	: 0x40

NtGlobalFlags 를 이용한 디버거 탐지 기법의 사용한 코드를 보도록 하겠다.

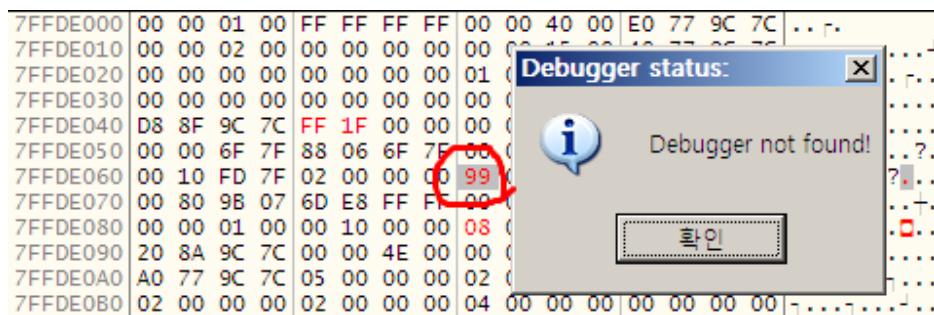
Address	Hex dump	Disassembly
00401000	\$ 64:A1 300000	MOV EAX, DWORD PTR FS:[30]
00401006	. 83C0 68	ADD EAX, 68
00401009	. 8B00	MOV EAX, DWORD PTR DS:[EAX]
0040100B	. 83F8 70	CMP EAX, 70
0040100E	.~ 74 15	JE SHORT NtGlobal.00401025
00401010	. 6A 40	PUSH 40
00401012	. 68 00304000	PUSH NtGlobal.00403000
00401017	. 68 22304000	PUSH NtGlobal.00403022
0040101C	. 6A 00	PUSH 0
0040101E	. E8 23000000	CALL <JMP.&user32.MessageBoxA>
00401023	.~ EB 13	JMP SHORT NtGlobal.00401038
00401025	> 6A 30	PUSH 30
00401027	. 68 11304000	PUSH NtGlobal.00403011
0040102C	. 68 36304000	PUSH NtGlobal.00403036
00401031	. 6A 00	PUSH 0
00401033	. E8 0E000000	CALL <JMP.&user32.MessageBoxA>
00401038	> 6A 00	PUSH 0
0040103A	. E8 0D000000	CALL <JMP.&kernel32.ExitProcess>

FS:[30]은 PEB 를 가리키고 있다. PEB 에서 0x68 만큼 떨어진 곳에 NtGlobalFlags 값이 있다.

Address	Hex dump	ASCII
7FFD5000	00 00 01 00 FF FF FF FF 00 00 00 01	... . . . . . . . .
7FFD5010	00 00 00 00 00 00 00 00 00 00 0A 00	40 77 9C 7C ... . . . . . . @w?
7FF FS:[0]	00 00 00 00 00 00 01 00 00 00	B0 29 DE 77 ..... ??
7FF FS:[2]	00 00 00 00 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 .. . . . . . . . .
7FFD5040	D8 8F 9C 7C 7F 00 00 00 00 00 00 00	??. . . . . . . .
7FFD5050	00 00 6F 7F 88 06 6F 7F 00 00 FA 7F	.. o?o?o? . . ? . ?
7FFD5060	00 10 FD 7F 02 00 00 00 70 00 00 00	00 00 00 00 00 00 00 00 00 00 00 00 .+? . . . . . . . .
7FFD5070	00 80 98 07 6D E8 FF FF 00 00 10 00	.. ?m? . . . . . . . .
7FFD5080	00 00 01 00 00 10 00 00 06 7C 10 00 00 00	... . . . . . . . . . . . .
7FFD5090	20 8A 9C 7C 00 00 43 00 00 FS:[68] 14 00 00 00	ㄻ   . C . . . . . .
7FFD50A0	A0 77 9C 7C 05 00 00 00 02 uu uu u0 CE 0E 00 02	겟?  . . . . . . . .
7FFD50B0	02 00 00 00 02 00 00 00 04 00 00 00 00 00 00 00	.. . . . . . . . . . . .
7FFD50C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	..... . . . . . . . .

## Reverse Engineering

PEB에서 0x68만큼 떨어진 곳의 값이 70인지 아닌지를 확인하는 것이다. 우회하는 방법은 당연히 이 값이 70이 아닌 다른 값으로 설정하면 된다.



## NtQueryInformationProcess

이 구조체의 프로토타입은 다음과 같다.

```
NTSYSAPI NTSTATUS NTAPI NtQueryInformationProcess(
    IN HANDLE ProcessHandle,
    IN PROCESS_INFORMATION_CLASS ProcessInformationClass,
    OUT PVOID ProcessInformation,
    IN ULONG ProcessInformationLength,
    OUT PULONG ReturnLength
);
```

ProcessDebugPort가 7로 설정된 ProcessInformationClass와 함께 호출이 될 때 프로세스가 디버깅 중이라면 시스템은 ProcessInformation을 -1로 설정한다. 즉, 디버거가 아닌 정상적으로 프로그램을 실행시키면 ProcessInformation의 값이 0이고 디버거에 의해 실행이 될 경우에는 -1 값이 설정이 된다. 다음은 NtQueryInformationProcess를 이용하여 디버깅을 탐지하는 코드이다.

Address	Hex dump	Disassembly
00401000	\$ C705 70304000	MOV DWORD PTR DS:[403070], -1
0040100A	. 68 46304000	PUSH NtQueryI.00403046
0040100F	. E8 74000000	CALL <JMP.&kernel32.LoadLibraryA>
00401014	. 68 50304000	PUSH NtQueryI.00403050
00401019	. 50	PUSH EAX
0040101A	. E8 63000000	CALL <JMP.&kernel32GetProcAddress>
0040101F	. A3 6C304000	MOV DWORD PTR DS:[40306C], EAX
00401024	. B8 70304000	MOV EAX, NtQueryI.00403070
00401029	. 50	PUSH EAX
0040102A	. 8BDC	MOV EBX, ESP
0040102C	. 6A 00	PUSH 0
0040102E	. 6A 04	PUSH 4
00401030	. 53	PUSH EBX
00401031	. 6A 07	PUSH 7
00401033	. FF30	PUSH DWORD PTR DS:[EAX]
00401035	. FF15 6C304000	CALL DWORD PTR DS:[40306C]
0040103B	. 58	POP EAX
0040103C	. 85C0	TEST EAX, EAX
0040103E	.v 75 15	JNZ SHORT NtQueryI.00401055

## Reverse Engineering

NtQueryInformationProcess 의 구조체에 들어갈 값들을 설정하는 코드가 보이고 CALL DWORD PTR DS:[40306C]를 통해서 NtQueryInformationProcess 함수를 호출하는 것을 볼 수 있다. 함수 호출 바로 직전의 스택을 살펴보자.

Address	Value	Comment
0013FFAC	FFFFFFFFF	hProcess = FFFFFFFF
0013FFB0	00000007	InfoClass = 7
0013FFB4	0013FFC0	Buffer = 0013FFC0
0013FFB8	00000004	Bufsize = 4
0013FFBC	00000000	pReqsize = NULL
0013FFC0	00403070	NtQueryI.00403070
0013FFC4	7C82F23B	RETURN to kernel32.7C82F23B

함수가 호출이 되고 리턴 값이 eax에 저장이 되고 이 값을 꺼내서 비교하는 것을 볼 수 있다.

Address	Hex dump	Disassembly
00401035	. FF15 6C304000	CALL DWORD PTR DS:[40306C]
0040103B	. 58	POP EAX
0040103C	. 85C0	TEST EAX, EAX
0040103E	.. 75 15	JNZ SHORT NtQueryI.00401055

그렇다면 호출된 함수의 결과값을 0으로 바꿔주면 TEST의 결과가 0이 되고 점프를 하지 않게 되어 디버거를 탐지 루틴을 우회할 수 있다. 0x00401035의 함수 호출 부분에서 Step into(F7)을 이용해서 함수 내부로 들어가 보겠다. 함수 내부로 들어가야만 리턴값을 정확히 알 수 있기 때문이다.

Address	Hex dump	Disassembly
7C96757F	B8 A1000000	MOV EAX, 0A1
7C967584	BA 0003FE7F	MOV EDX, 7FFE0300
7C967589	FF12	CALL DWORD PTR DS:[EDX]
7C96758B	C2 1400	RETN 14

함수 내부로 진입하면 0x7FFE0300 주소에 있는 내용을 호출을 하고 리턴을 한다.

Address	Hex dump
0013FFC0	FF FF FF FF 3B F2 82 7C 00 00 00 00 00 00 00 00
0013FFD0	00 80 FD 7F 00 00 00 00 C8 FF 13 00 E4 6C C1 B8
0013FFE0	FF FF FF FF 60 1A 82 7C 48 F2 82 7C 00 00 00 00
0013FFF0	00 00 00 00 00 00 00 00 00 10 40 00 00 00 00 00

리턴 후 0x0013FFC0에 ProcessInformation의 값이 들어 있고 값이 FFFFFFFF로 되어 있는걸 볼 수 있다. 이것은 디버거를 탐지하는 ProcessInformation 값을 조작하지 못했다는 의미이다. 이 값을 조작을 해보도록 하겠다. 그리고 스택을 살펴보면 스택의 가장 위에 있는 값을 POP EAX에 대해서 가져오게 되는데 FFFFFFFF로 되어 있는걸 볼 수 있다.

Address	Value	Comment
0013FFC0	FFFFFFFFF	
0013FFC4	7C82F23B	RETURN to kernel32.7C82F23B

## Reverse Engineering

NtQueryInformationProcess 함수 진입 후 보이는 아래 코드를 ProcessInformation의 값을 0으로 바꿀 수 있는 코드로 수정하겠다.

Address	Hex dump	Disassembly
7C96757F	B8 A1000000	MOV EAX, 0A1
7C967584	BA 0003FE7F	MOV EDX, 7FFE0300
7C967589	FF12	CALL DWORD PTR DS:[EDX]
7C96758B	C2 1400	RETN 14

총 7 바이트를 차지하고 있다. 그런데 우리가 작성하고자 하는 코드는 9 바이트이다. 따라서 RETN 14의 일부를 덮어쓰게 될 것이다.

Address	Hex dump	Disassembly
7C96757F	B8 A1000000	MOV EAX, 0A1
7C967584	8B4424 0C	MOV EAX, DWORD PTR SS:[ESP+C]
7C967588	C700 00000000	MOV DWORD PTR DS:[EAX], 0
7C96758E	90	NOP
7C96758F	B8 A2000000	MOV EAX, 0A2
7C967594	BA 0003FE7F	MOV EDX, 7FFE0300
7C967599	FF12	CALL DWORD PTR DS:[EDX]
7C96759B	C2 1400	RETN 14

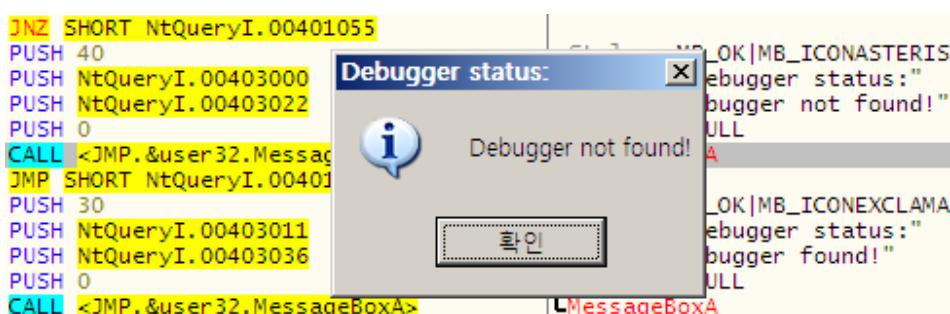
ESP+C에는 ProcessInformation의 주소가 들어 있는 곳이고 이 주소에 0이라는 값을 넣는다. 그리고 앞에서 보았던 큰 의미를 갖기 않는 루틴을 지나 리턴하게 된다.

Address	Hex dump	Disassembly
0040103B	. 58	POP EAX
0040103C	. 85C0	TEST EAX, EAX
0040103E	.~ 75 15	JNZ SHORT NtQueryI.00401055

Address	Value	Comment
0013FFC0	00000000	
0013FFC4	7C82F23B	RETURN to kernel32.7C82F23B

이전과는 다르게 스택의 가장 위에 있는 값이 00000000으로 되어 있다. 이 값을 가져와 TEST로 비교한 후 디버거가 탐지되지 않았다는 메시지를 보여주게 된다.



## kernel32!CheckRemoteDebuggerPresent

CheckRemoteDebuggerPresent 함수는 자신의 프로세스뿐만 아니라 원하는 프로세스가 디버깅 당하는 중인지 까지 확인이 가능한 함수이다.

```
BOOL CheckDebugger(HANDLE hProcess)
{
    BOOL Retval = 0;
    CheckRemoteDebuggerPresent(hProcess, &Retval );
    return Retval ;
}
```

이 함수는 NtQueryInformationProcess 로 연결된다. 이 함수를 이용한 전형적인 코드는 다음과 같다.

Address	Hex dump	Disassembly
00401015	. 68 70304000	PUSH CheckRem.00403070
0040101A	. 6A FF	PUSH -1
0040101C	. FF00	CALL EAX
0040101E	. A1 70304000	MOV EAX, DWORD PTR DS:[403070]
00401023	. 85C0	TEST EAX, EAX
00401025	.. 75 15	JNZ SHORT CheckRem.0040103C

CALL EAX 를 Step into(F7)로 들어가 보면 다음과 같은 코드가 나온다.

Address	Hex dump	Disassembly
7C85C097	8BFF	MOV EDI, EDI
7C85C099	55	PUSH EBP
7C85C09A	8BEC	MOV EBP, ESP
7C85C09C	837D 08 00	CMP DWORD PTR SS:[EBP+8], 0
7C85C0A0	56	PUSH ESI
7C85C0A1	v 74 35	JE SHORT kernel32.7C85C0D8
7C85C0A3	8B75 0C	MOV ESI, DWORD PTR SS:[EBP+C]
7C85C0A6	85F6	TEST ESI, ESI
7C85C0A8	v 74 2E	JE SHORT kernel32.7C85C0D8
7C85C0AA	6A 00	PUSH 0
7C85C0AC	6A 04	PUSH 4
7C85C0AE	8D45 08	LEA EAX, DWORD PTR SS:[EBP+8]
7C85C0B1	50	PUSH EAX
7C85C0B2	6A 07	PUSH 7
7C85C0B4	FF75 08	PUSH DWORD PTR SS:[EBP+8]
7C85C0B7	FF15 B810807C	CALL DWORD PTR DS:[<&ntdll.NtQueryInfor
7C85C0BD	85C0	TEST EAX, EAX
7C85C0BF	v 7D 08	JGE SHORT kernel32.7C85C0C9

EBP+8 에 있는 값과 0 을 비교하고 같으면 아래쪽에 나오는 NtQueryInformationProcess 함수 호출을 뛰어 넘어서 나갈 수 있고 같지 않으면 NtQueryInformationProcess 함수를 호출해야 한다.

NtQueryInformationProcess 함수가 호출이 되면 ProcessInformation 값은 0 으로 패치해야 한다.

NtQueryInformationProcess 에서 봤던 코드와 비슷한 코드들이 보인다. 함수 데코레이션 덕분에 함수명

## Reverse Engineering

이 보여 금방 알 수 있었지만 데코레이션이 안되었더라도 0, 4, 7 을 PUSH 하고 EAX 와 EBP+8 등을 PUSH 하는 것을 봐서 NtQueryInformationProcess 라는 것을 의심해볼 수 있다. ProcessInformation 값을 패기하는 방법은 앞에서 보았으니 여기서는 NtQueryInformationProcess 함수를 거치지 않는 방법을 사용하는 것을 보도록 하자.

/\*7C85C09C\*/ CMP DWORD PTR SS:[EBP+8],0 이 코드를 실행하기 전에 다음과 같이 EBP+8에 있는 값을 수정한다.

Address	Hex dump	Edit data at 0013FFBC
0013FFBC	FF FF FF FF	
0013FFCC	00 00 00 00	
0013FFDC	E4 BC C3 B9	
0013FFEC	00 00 00 00	
0013FFFC	00 00 00 00	

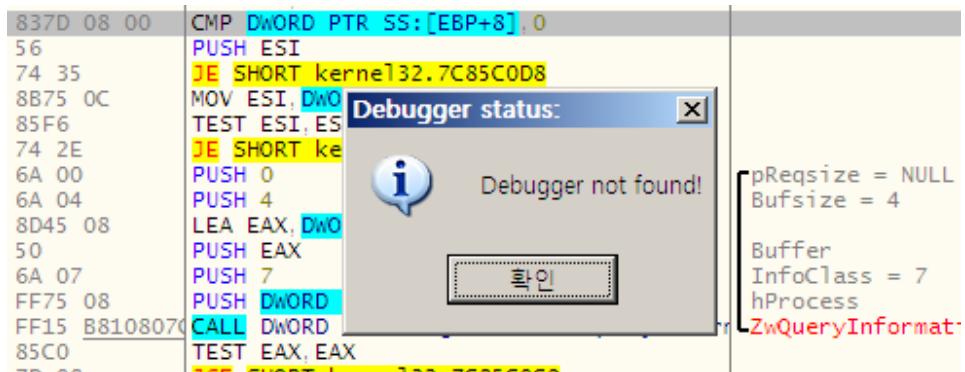
Edit data at 0013FFBC

ASCII: = = = =

UNICODE: = =

HEX +04: 00 00 00 00

수정한 후 F9 를 눌러 실행 하면 Debugger not found! 라는 메시지를 확인할 수 있다.



ProcessInformation 값을 패기하는 것이 아니라 NtQueryInformationProcess 함수 자체를 호출하지 않는 방법을 사용하여 디버거를 탐지하는 루틴을 우회하였다.

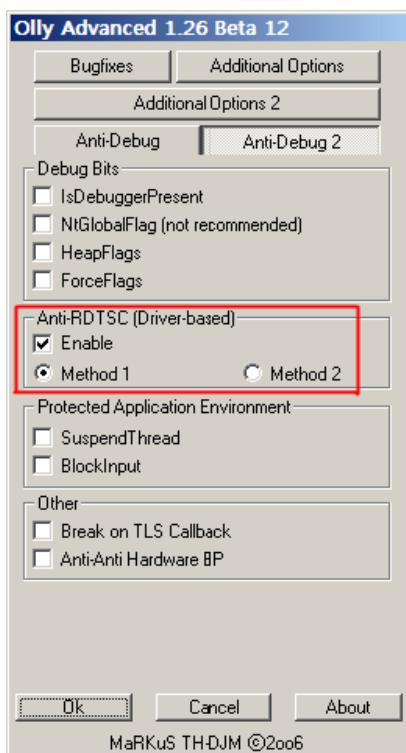
## Timing Check

시간을 이용한 디버거 탐지 기법은 아주 단순한 개념을 사용한다. 일반적으로 실행하는데 걸리는 시간 보다 사용시간이 더 길다면 디버거에서 프로세스를 실행 중이라고 판단한다. 이 때 RDTSC(Read Time Stamp Counter) 명령을 사용해서 시간의 차이를 계산한다. 간단한 예제를 통해서 확인해보도록 하겠다.

Address	Hex dump	Disassembly
00401000	\$ 0F31 →	RDTSC
00401002	. 33C9	XOR ECX, ECX
00401004	. 03C8	ADD ECX, EAX
00401006	. 0F31 →	RDTSC
00401008	. 2BC1	SUB EAX, ECX
0040100A	. 3D FF0F0000	CMP EAX, 0FFF
0040100F	.~ 73 14 →	JNB SHORT RDTSC. 00401025
00401011	. 6A 40	PUSH 40
00401013	. 68 00304000	PUSH RDTSC. 00403000
00401018	. 68 22304000	PUSH RDTSC. 00403022
0040101D	. 6A 00	PUSH 0
0040101F	. E8 16000000	CALL <JMP.&user32.MessageBoxA>
00401024	. C3	RETN
00401025	> 6A 30	PUSH 30
00401027	. 68 11304000	PUSH RDTSC. 00403011
0040102C	. 68 3F304000	PUSH RDTSC. 0040303F
00401031	. 6A 00	PUSH 0
00401033	. E8 02000000	CALL <JMP.&user32.MessageBoxA>
00401038	l. C3	RETN

이 기법은 시간체크하는 코드를 트레이싱하여 우회하는 방법을 사용하면 된다. 증가량을 비교하는 코드가 나오기 전에 브레이크포인트를 걸고 브레이크포인트가 걸릴 때까지 트레이싱한다. 또는 GetTickCount 함수를 불러올 때 브레이크포인트를 설정하여 리턴값을 수정할 수도 있다.

근본적인 우회 방법은 커널 드라이버를 작성해야 하는데 다행스럽게도 Olly Advanced 플러그인을 사용하여 우회 할 수 있다.



Olly Advanced에서 사용하는 방법은 다음과 같다.

1. 4 번 캠트롤 레지스터(CR4)의 TSD(Time Stamp Disable) bit 값을 1로 설정한다. TSD bit 가 1로 설정되면 Ring0 이외의 모드에서 RDTSC 가 실행될 때마다 GP(General Protection) 익셉션이 발생한다.

2. GP 익셉션이 발생하면 GPF 핸들러가 동작한다. GPF 익셉션 넘버는 0xD 이므로 GPF 핸들러의 주소는 IDT 베이스 주소로부터 0xD \* 4byte 만큼 떨어진 곳에 저장되어 있을 것이다.

3. Olly Advanced 플러그인은 바로 IDT에 등록되어 있는 GPF 핸들러를 후킹하여 GPF가 발생한 경우 RDTSC가 원인인지 조사하고 RDTSC 실행이 원인이라면 리턴값을 조작하는 방법으로 timing check를 우회한다.

# Reverse Engineering

## **kernel32!OutputDebugStringA**

이 기법은 유효한 ASCII 스트링과 함께 OutputDebugStringA 를 호출한다. 프로그램이 만약 디버깅 중이라면 반환 값은 매개변수로서 전달된 스트링의 주소가 될 것이다. 정상적인 조건에서는 반환 값이 1 이되어야 한다.

그리고 이 기법은 Ollydbg에서만 가능한 포맷 스트링 공격에 사용될 수 있다.

올리디버거의 취약점을 공격할 수 있으며 현재 사용중인 1.10에서는 패치가 되지 않은 상태이다.

이 외에도 다양한 디버거 탐지 기법들이 있는데 모든 기법들을 수작업으로 우회한다는 것은 상당히 힘든 작업이 될 것이다. 그래서 우리는 올리플러그인을 이용해서 비교적 간단히 디버거 탐지 기법들을 우회할 수 있다. 대표적인 플러그인으로는 앞에서 잠깐 나왔던 Olly Advanced 가 있다. 그리고 HideDebugger 나 Olly Invisible, IsDebuggerPresent 와 같은 플러그인도 있다.

## 6-2. BreakPoint 탐지 기법

### Break Point 탐지 기법



- Hardware Breakpoint
  - DR0 ~ DR3 디버그 레지스터 이용
  - DR7 디버그 레지스터는 브레이크포인트 제어에 사용
- Software Breakpoint
  - 디버거가 직접 관리하는 브레이크포인트
  - 0xCC (INT 3)을 사용
- Instruction Breakpoint
  - 특정 주소 번지의 명령에 설정되는 브레이크포인트
- Memory Breakpoint
  - 특정 메모리 영역에 위치한 데이터를 읽거나 변경할 때 사용

### Student Notes

브레이크포인트는 디버거를 통해 실행 중인 프로그램을 특정 위치나 조건에서 멈추게 하고 싶을 때 사용하는 기법이다. 이런 브레이크포인트를 무력화하는 것도 안티 리버싱 기법 중 하나이다.

여러분들이 만든 프로그램을 누군가가 디버거로 분석을 하면서 특정 위치에 브레이크포인트를 걸어서 중요한 정보를 획득하려고 시도한다고 생각해보자. 당연히 그 브레이크포인트를 삭제해버리고 싶을 것이다.

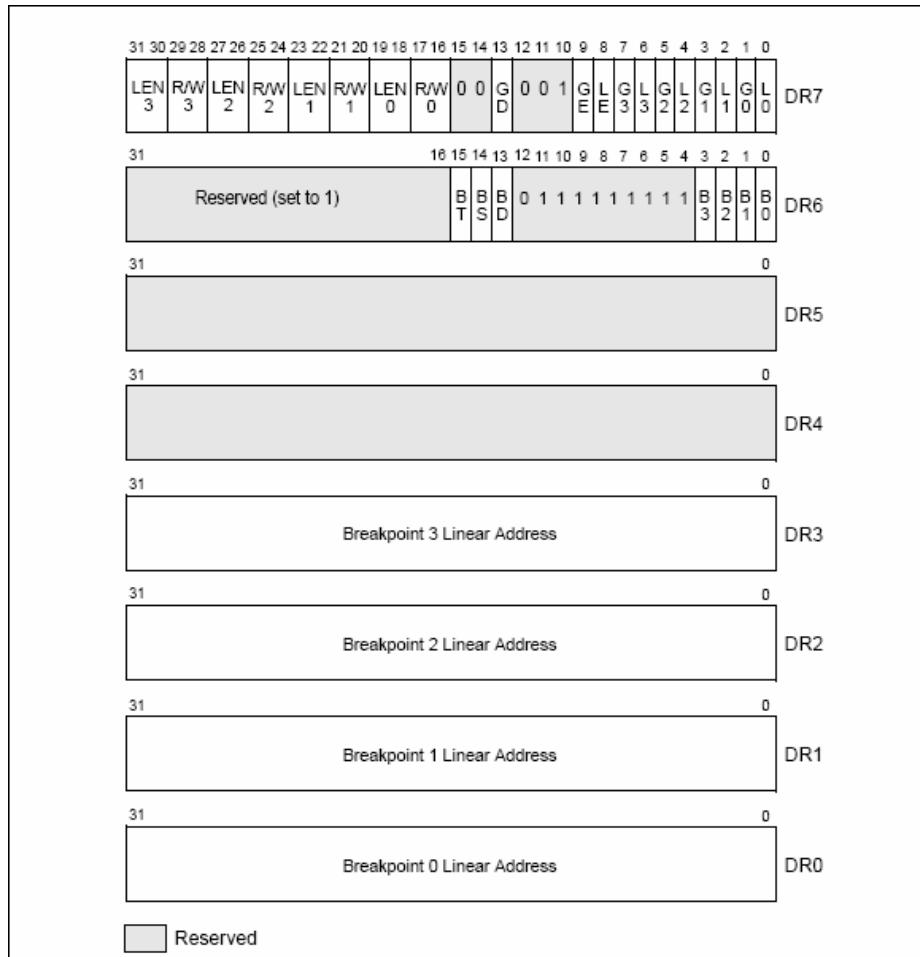
먼저 브레이크포인트의 특징에 대해서 알아보고 이런 브레이크포인트를 탐지하고 무력화시키는 방법과 우회 기법에 대해서 알아보도록 하겠다.

## Reverse Engineering

### 브레이크 포인트 종류

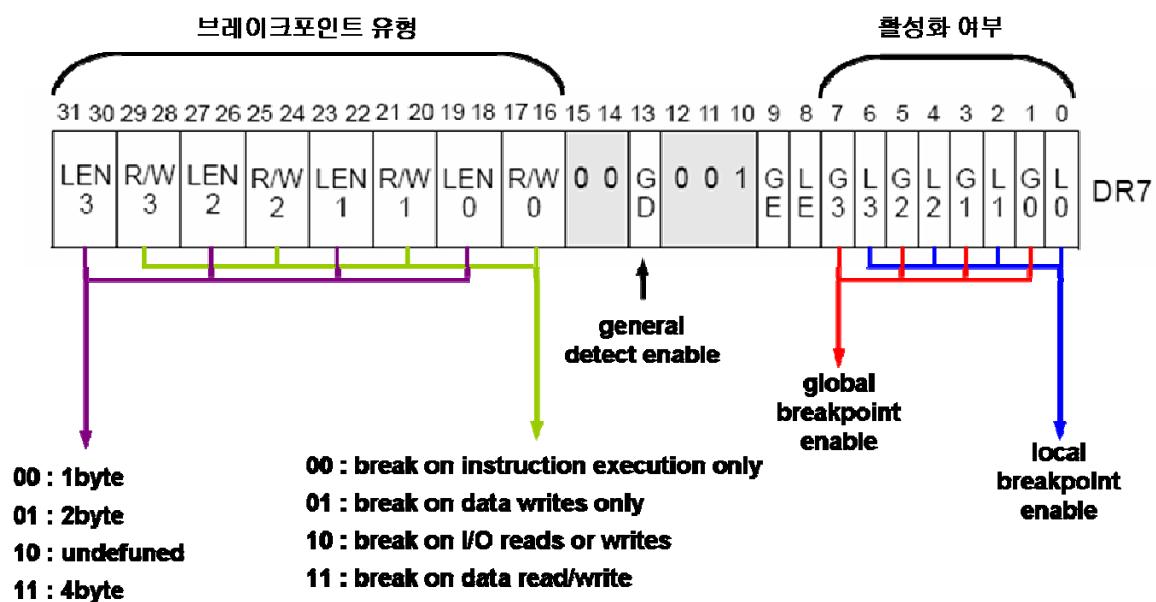
Hardware Breakpoint	DR0 ~ DR3 디버그 레지스터 이용 DR7 디버그 레지스터를 이용하여 브레이크포인트 제어 아키텍처에서 직접 지원하므로 빠르고 정확하다. 레지스터의 개수가 제한적이라는 단점이 있다.
Software Breakpoint	디버거가 직접 관리한다. 소프트웨어 방식으로 구현된 인스트럭션 브레이크포인트는 명령어의 첫 번째 바이트를 0xCC(INT 3)으로 변경한다.
Instruction Breakpoint	특정 주소 번지에 설정되는 브레이크포인트이다. 하드웨어 / 소프트웨어 방식으로 구현된다.
Memory Breakpoint	특정 메모리 주소의 데이터를 읽거나 변경할 때 디버거를 멈출 때 사용한다. 하드웨어 / 소프트웨어 방식으로 구현

### Hardware Breakpoint 탐지 및 우회 기법



IA32에서 제공하는 디버그 레지스터를 이용하는 방법이 하드웨어 브레이크포인트이다. DR0 ~ DR7 까지 모두 8개의 디버그 레지스터를 제공한다.

- DR0 ~ DR3 : 브레이크포인트를 설정할 수 있는 주소를 지정
- DR4 ~ DR5 : 예약된 공간으로 사용하지 않음
- DR6 : 디버그 상태 레지스터, 디버그나 브레이크포인트 예외가 발생했을 때 영향 받는 상태를 알려준다.
- DR7 : 디버그제어 레지스터, 하드웨어 브레이크포인트의 활성화 여부와 유형에 대한 정보를 가지고 있다.



주의 깊게 봐야 할 부분은 브레이크포인트 유형에서 R/W0 ~ R/W3 까지 이다. 이곳에 00 으로 설정되어 있으면 인스트럭션 브레이크포인트이고 01 이나 11 이면 메모리 브레이크포인트를 의미한다는 것이다.

DR7 디버그 레지스터는 위 그림과 같이 각 비트들이 특별한 의미를 가지고 있다. 프로그램들이 위 내용을 확인할 수 있다면 하드웨어 브레이크포인트를 금방 탐지할 수 있을 것이다. 하지만 IA32 매뉴얼에 의하면 디버그 레지스터는 일반적으로 Ring0 모드에서만 접근할 수 있다고 되어 있다. 그런데 윈도우의 경우 SEH를 이용하면 Ring3에서도 디버그 레지스터의 값을 읽거나 변경하는 것이 가능하다.

모듈 2-6에서 보았던 SEH의 프로토타입을 떠올려 의셉션 핸들러의 세 번째 아규먼트는 ContextRecord로써 의셉션이 발생했을 당시의 레지스터들의 값에 접근하거나 변경하는 것이 가능하다. 그리고 \_CONTEXT 구조체 안에는 범용 레지스터를 비롯하여 디버그 레지스터가 포함되어 있다. 즉, 의셉션이

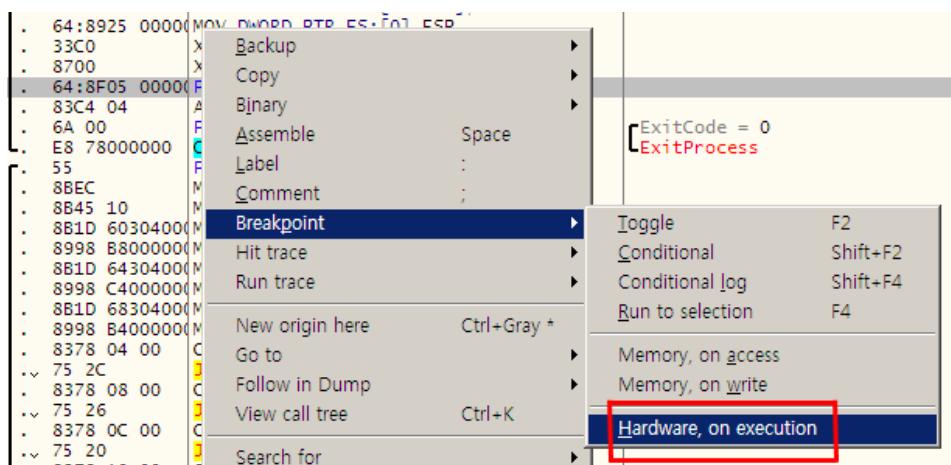
## Reverse Engineering

발생하면 디버그 레지스터를 Ring3 모드에서도 접근하거나 변경이 가능하다는 것이다. 방법은 비교적 간단하다.

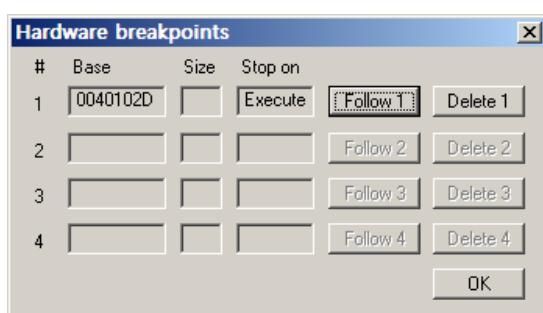
1. SEH 핸들러에 하드웨어 브레이크 포인트 탐지 루틴을 등록
2. 익셉션을 발생
3. 미리 등록해 놓은 SEH 핸들러가 실행
4. 핸들러는 ContextRecord를 이용하여 디버그 레지스터 값을 변경하여 하드웨어 브레이크포인트를 무력화시킨다.

먼저 하드웨어 브레이크포인트를 탐지하는 예제를 보도록 하겠다. 익셉션을 발생시켜 SEH 안으로 들어오게 해서 하드웨어 브레이크포인트가 설정되어 있는지 검사를 한다.

먼저 적당한 곳에 하드웨어 브레이크포인트를 견다.



하드웨어 브레이크포인트 설정 유무를 확인하려면 메뉴에서 Debug → Hardware breakpoints에서 확인할 수 있다.



그리고 F9로 실행한다.

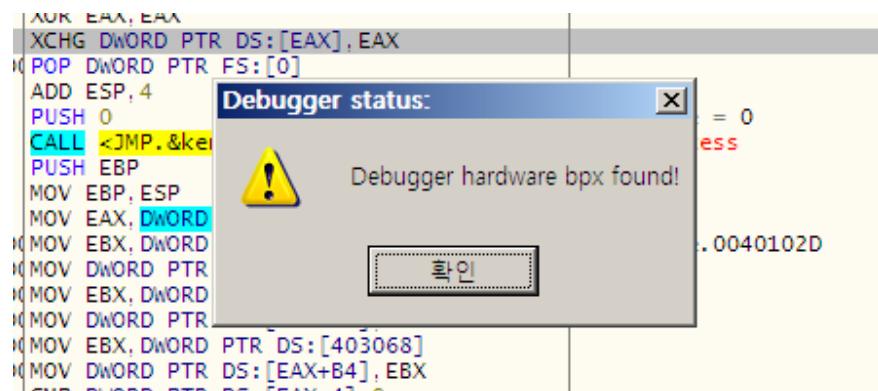
```

00401029 | . 33C0    XOR EAX, EAX
0040102B | . 8700    XCHG DWORD PTR DS:[EAX], EAX ←
0040102D | . 64:8F05 00000 POP DWORD PTR FS:[0]

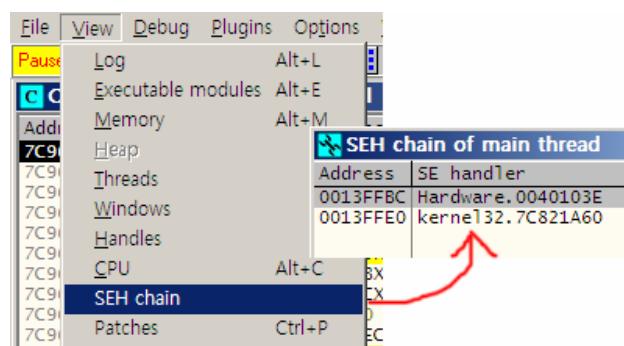
```

Access violation when writing to [00000000] - use Shift+F7/F8/F9 to pass exception to program

화살표로 표시된 부분이 익셉션을 발생시킨 부분이다. 디버거의 하단에 있는 상태바를 확인해보니 Access violation 이 발생한 것을 볼 수 있다. 익셉션이 발생하면 Shift + F7/F8/F9 셋 중 하나를 누르면 된다. Shift + F9를 누르면 익셉션을 처리하고 계속 실행이 되고 Shift + F8을 누르면 익셉션을 처리하는 루틴으로 들어가서 한 단계씩 실행하게 된다. 일단 Shift + F9를 눌러 보자.



역시나 익셉션을 핸들러 내부에 하드웨어 브레이크포인트를 탐지하는 루틴이 들어 있는 것이다. 다시 처음부터 똑같이 진행하되 이번에는 Shift + F9 대신 Shift + F7을 눌러서 진행해보도록 하자. 하드웨어 브레이크포인트는 지워지지 않았으니 다시 지정하지 않아도 상관은 없다. 그런데 언제 나올지 모르는 디버거 탐지 루틴을 마냥 F7을 눌러가면서 찾을 수는 없는 노릇이다. Shift + F7을 눌러 익셉션 처리 루틴으로 들어와서 SEH Chain을 살펴보자. 상단 메뉴에서 View → SEH chain에서 SEH 처리 부분을 찾을 수 있다.



## Reverse Engineering

현재 익셉션을 처리하는 곳이 0x0040103E라는 것을 알 수 있다. 이 주소에서 하드웨어 브레이크가 설정되어 있는지 확인하는 루틴이 들어 있을거라 예상을 하고 브레이크포인트를 설정한다.

SEH chain of main thread	
Address	SE handler
0013FFBC	Hardware.0040103E
0013FFE0	kernel32.7C821A60

F9를 눌러 실행한다.

Address	Hex dump	Disassembly
0040103E	. 55	PUSH EBP
0040103F	. 8BEC	MOV EBP, ESP
00401041	. 8B45 10	MOV EAX, DWORD PTR SS:[EBP+10]
00401044	. 8B1D 60304000	MOV EBX, DWORD PTR DS:[403060]
0040104A	. 8998 B8000000	MOV DWORD PTR DS:[EAX+B8], EBX
00401050	. 8B1D 64304000	MOV EBX, DWORD PTR DS:[403064]
00401056	. 8998 C4000000	MOV DWORD PTR DS:[EAX+C4], EBX
0040105C	. 8B1D 68304000	MOV EBX, DWORD PTR DS:[403068]
00401062	. 8998 B4000000	MOV DWORD PTR DS:[EAX+B4], EBX
00401068	. 8378 04 00	CMP DWORD PTR DS:[EAX+4], 0 ←
0040106C	.~ 75 2C	JNZ SHORT Hardware.0040109A ←
0040106E	.~ 8378 08 00	CMP DWORD PTR DS:[EAX+8], 0 ←
00401072	.~ 75 26	JNZ SHORT Hardware.0040109A ←
00401074	.~ 8378 0C 00	CMP DWORD PTR DS:[EAX+C], 0 ←
00401078	.~ 75 20	JNZ SHORT Hardware.0040109A ←
0040107A	.~ 8378 10 00	CMP DWORD PTR DS:[EAX+10], 0 ←
0040107E	.~ 75 1A	JNZ SHORT Hardware.0040109A ←

우리가 원하던 곳을 찾았다! 화살표로 표시된 부분이 바로 DR0 ~ DR3 까지 하드웨어 브레이크포인트가 설정되어 있는지 확인하는 부분이다. 모두 결과가 0과 같이 않을 경우 0x0040109A로 점프를 한다.

0x0040109A에는 앞에서 보았던 하드웨어 브레이크포인트가 설정되었다는 메시지박스를 띄워주는 함수가 있다.

만약 여기서 하드웨어 브레이크포인트를 무력화시키고자 한다면 SEH 핸들러 내부에서 DR0 ~ DR3 까지의 값을 0으로 만들어버리면 된다.

앞서 보았던 예제의 어셈 코드 중 디버그 레지스터의 설정 유무를 확인하는 루틴은 다음과 같다.

```
CMP DWORD PTR DS:[EAX+4h], 0
JNE @hardware_bpx_found
CMP DWORD PTR DS:[EAX+8h], 0
JNE @hardware_bpx_found
CMP DWORD PTR DS:[EAX+0Ch], 0
JNE @hardware_bpx_found
CMP DWORD PTR DS:[EAX+10h], 0
JNE @hardware_bpx_found
```

모두 비교만 하는데 비교하는 부분을 MOV를 이용하여 0으로 바꿔주면 하드웨어 브레이크 포인트를 무력화시킬 수 있다.

```
MOV DWORD PTR DS:[EAX+4h],0
MOV DWORD PTR DS:[EAX+8h],0
MOV DWORD PTR DS:[EAX+0Ch],0
MOV DWORD PTR DS:[EAX+10h],0
```

이렇게 하드웨어 브레이크포인트를 무력화시키거나 탐지하는 경우는 어떻게 우회할 것인가?

먼저 익셉션을 발생시키는 코드를 NOP로 처리하는 방법이 있다. 하드웨어 브레이크포인트 탐지를 위해 익셉션을 발생시켰다면 익셉션을 발생시키는 코드는 전체 코드가 실행되는데 큰 문제가 없을 것이다.

Address	Hex dump	Disassembly
0040101C	. 8925 64304000	MOV DWORD PTR DS:[403064], ESP
00401022	. 64:8925 000000	MOV DWORD PTR FS:[0], ESP
00401029	90	NOP
0040102A	90	NOP
0040102B	90	NOP
0040102C	90	NOP
0040102D	v EB 55	JMP SHORT Hardware.00401084
0040102F	90	NOP
00401030	90	NOP
00401031	90	NOP
00401032	90	NOP
00401033	90	NOP

본 예제에서는 익셉션을 발생시키는 코드를 NOP 처리하고 바로 다음 메시지 박스를 띄우는 루틴으로 점프하는 구문을 추가하였다.

두 번째 방법은 어느 한 시점에서 브레이크포인트를 탐지하거나 제거한다는 단점을 역으로 이용하면 된다. 이렇게 브레이크포인트를 탐지하거나 제거하는 코드를 찾아낼 수 있다면 탐지 및 제거 코드 이후에 브레이크포인트를 설정하는 것이다. 유저모드 어플리케이션에서 하드웨어 브레이크포인트를 탐지하거나 제거하려면 반드시 SEH를 호출해야 한다는 사실을 이용하는 방법도 있다.

익셉션 핸들러에서 스택을 사용하면 EBP+0x10은 아규먼트로 전달 받은 ContextRecord 값이다. 이러한 코드가 발견될 경우 ContextRecord를 전달 받아서 어떤 행동들을 하는지 살펴볼 필요가 있다. 만약 익셉션 핸들러가 스택을 사용하지 않는다면 ESP+0x0C가 ContextRecord를 의미한다.

그리고 리버싱을 하는 과정에서 SEH chain을 자주 확인하는 것이 좋다.

## Reverse Engineering

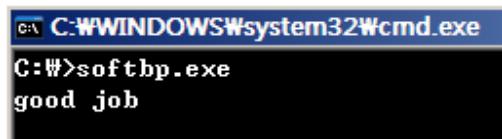
### Software Breakpoint 탐지 및 우회 기법

소프트웨어 브레이크포인트는 명령의 처음 1 바이트를 0xCC(INT 3)으로 변경하여 프로그램의 진행을 멈춘다. 소프트웨어 브레이크포인트를 탐지하는 방법은 크게 두 가지 정도 있다.

- 보호하려는 코드의 처음 1byte 또는 처음 몇 바이트 정도를 검사하여 0xCC 가 있는지 확인한다.
- 보호하려는 코드 전체의 체크섬 값을 계산하여 비교한다.

먼저 특정 위치에 소프트웨어 브레이크포인트가 설정될 경우 프로그램이 비정상적으로 종료되는 예제를 보도록 하겠다.

정상적인 프로그램의 실행은 다음과 같은 결과를 보여준다.



```
C:\>softbp.exe
good job
```

바로 이 “good job”이라는 메시지를 출력해주는 함수에 브레이크포인트를 걸 경우 프로그램이 비정상적으로 종료된다.

디버거로 프로그램을 오픈하고 Step over(F8)로 계속 진행하다 보면 다음과 같은 코드를 볼 수 있다.

Address	Hex dump	Disassembly
00401274	. E8 8CFDFFFF	CALL softbp.00401005
00401279	. 83C4 0C	ADD ESP, 0C
0040127C	. 8945 E4	MOV DWORD PTR SS:[EBP-1C], EAX
0040127F	. 8B55 E4	MOV EDX, DWORD PTR SS:[EBP-1C]
00401282	. 52	PUSH EDX
00401283	. E8 981A0000	CALL softbp.exit

프로그램이 종료(CALL softbp.exit)되기 바로 직전에 스택 클리어링(ADD ESP, 0C)을 볼 수 있다. 00401005 의 주소를 호출하는데 따라가 보도록 하겠다.

Address	Hex dump	Disassembly
00401005	\$v E9 66000000	JMP softbp.main
0040100A	\$v rE9 11000000	JMP softbp.ProtectedFunc

메인 함수를 호출하는 부분과 우리가 보호하고자 하는 함수 모두 보인다. 먼저 ProtectedFunc에 F2를 놀

러 브레이크포인트를 걸고 메인함수를 따라 이동해보겠다.

Address	Hex dump	Disassembly
00401070	> 55	PUSH EBP
00401071	. 8BEC	MOV EBP, ESP
00401073	. 83EC 40	SUB ESP, 40
00401076	. 53	PUSH EBX
00401077	. 56	PUSH ESI
00401078	. 57	PUSH EDI
00401079	. 8D7D C0	LEA EDI, DWORD PTR SS:[EBP-40]
0040107C	. B9 10000000	MOV ECX, 10
00401081	. B8 CCCCCCCC	MOV EAX, CCCCCCCC
00401086	. F3:AB	REP STOS DWORD PTR ES:[EDI]
00401088	. BF 0A104000	MOV EDI, softbp.0040100A
0040108D	. B9 04000000	MOV ECX, 4
00401092	. B8 60060000	MOV EAX, 660
00401097	. C1E8 03	SHR EAX, 3
0040109A	. F2:AE	REPNE SCAS BYTE PTR ES:[EDI]
0040109C	. 85C9	TEST ECX, ECX
0040109E	.~ 74 04	JE SHORT softbp.004010A4
004010A0	. 0F31	RDTSC
004010A2	. 50	PUSH EAX
004010A3	. C3	RETN
004010A4	> E8 61FFFFFF	CALL softbp.0040100A
004010A9	. 5F	POP EDI
004010AA	. 5E	POP ESI
004010AB	. 5B	POP EBX

메인함수에서는 특정 위치에 0xCC 가 있는지 확인을 할 것이다. 그런데 0xCC 는 어디에도 보이지 않는다. 이것은 code permutation 이라는 기법이 적용되어 있는 것인데 다음 코드를 보도록 하자.

0040108D	. B9 04000000	MOV ECX, 4
00401092	. B8 60060000	MOV EAX, 660
00401097	. C1E8 03	SHR EAX, 3

ECX 에 조사할 바이트를 설정한다. 그리고 EAX 에 0x660 을 넣는다. SHR 은 오른쪽으로 쉬프트하는 명령이고 EAX 의 값을 오른쪽으로 3 비트 쉬프트하게 되면 0xCC 가 된다. 이렇게 변경되는 이유는 0x660 을 이진수로 표현하면 0110 0110 0000 이 되고 이 값을 오른쪽으로 3 비트 쉬프트 하면 000011001100 이 된다. 이 값을 다시 16 진수 형태로 변환하면 0xCC 가 되는 것이다. (4 비트씩 분할하면 1100 이 되고 이 값은 10 진수로 12 가 된다. 즉, 16 진수로는 C 가 된다)

즉, 위 코드 부분이 0xCC(INT 3)인 소프트웨어 브레이크포인트가 설정되어 있는지 검사하기 위한 준비하고 보면 된다.

0040109A	. F2:AE	REPNE SCAS BYTE PTR ES:[EDI]
----------	---------	------------------------------

위 코드가 의미하는 것은 EDI 가 가리키는 곳에 EAX 에 저장되어 있는 값이 있는지 찾으라는 것이다. 만약 0xCC 를 찾게 되면 ECX 값은 0 이 되지 않고 찾지 못하면 0 이 된다. 따라서 TEST ECX, ECX 에서

## Reverse Engineering

ECX의 값이 0이면 004010A4로 점프하고 그렇지 않을 경우 아래의 코드를 실행하게 된다.

004010A0	. OF31	RDTSC
004010A2	. 50	PUSH EAX
004010A3	. C3	RETN

RETN은 현재 스택의 가장 윗부분에 있는 주소로 복귀를 하게 된다. RETN을 실행하기 전 스택에는 다음과 같은 데이터들이 저장되어 있다.

Address	Value	Comment
0013FF30	B58C7BB6	
0013FF34	00000000	
0013FF38	00000000	
0013FF3C	7FFD8000	
0013FF40	CCCCCCCC	
0013FF44	CCCCCCCC	
0013FF48	CCCCCCCC	
0013FF4C	CCCCCCCC	

RETN은 POP EIP와 같은 의미이다. 물론 POP EIP와 같은 명령은 존재하지 않지만 이것은 스택의 가장 윗부분에 있는 주소값을 가져와서 그 주소로 복귀한다는 의미이다. 그런데 현재 스택의 가장 윗부분에는 B58C7BB6이라는 엉뚱한 값이 저장되어 있고 결국 예외가 발생하여 프로그램이 비정상 종료를 하게 된다. 보통 함수가 끝나고 복귀할 때 RETN 명령을 만나게 되고 다음과 같이 복귀주소가 스택에 푸시되는 것이 정상적이다.

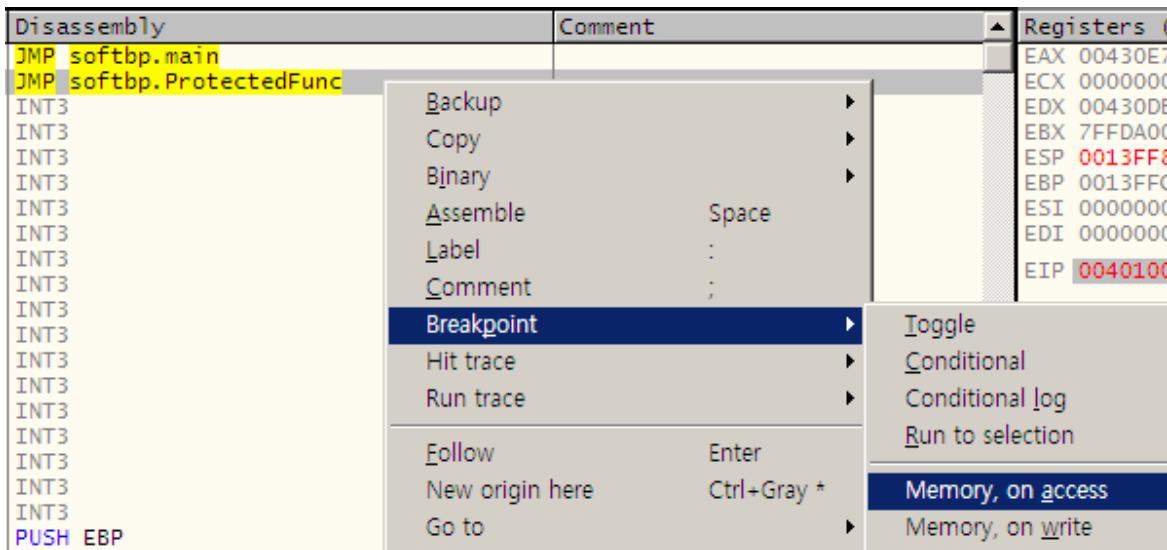
00401055	L. C3	RETN
Address	Value	Comment
0013FF2C	004010A9	RETURN to softbp.main+39 from softbp.00401055

이렇게 복귀주소를 이상한 곳으로 설정되게끔 프로그래밍을 한다면 브레이크포인트가 탐지되고 엉뚱한 곳으로 복귀하게 되어 정상적으로 분석하는 것이 힘들어 진다.

소프트웨어 브레이크포인트 탐지 기법을 우회하는 방법은 하드웨어 브레이크포인트를 사용하는 방법이 있고 메모리 브레이크포인트를 사용하는 방법이 있다. 이 두 가지 방법 모두 실제 코드를 변경하는 것 아니기 때문에 우회가 가능하다.

ProtectedFunc 함수에 인스트럭션 브레이크포인트를 설정하는 것이 아니라 메모리에 접근할 때 멈출 수 있게끔 메모리 브레이크포인트를 설정하면 앞에서 사용한 탐지 기법을 우회 할 수 있다.

## Reverse Engineering



ProtectedFunc에 메모리 브레이크포인트를 설정하고 F9로 실행했더니 다음 그림에서 보는 것과 같은 위치에서 멈추었다.

Address	Hex dump	Disassembly
00401088	. BF 0A104000	MOV EDI, softbp.0040100A
0040108D	. B9 04000000	MOV ECX, 4
00401092	. B8 60060000	MOV EAX, 660
00401097	. C1E8 03	SHR EAX, 3
<b>0040109A</b>	<b>. F2:AE</b>	<b>REPNE SCAS BYTE PTR ES:[EDI]</b>
0040109C	. 85C9	TEST ECX, ECX
0040109E	.~ 74 04	<b>JE SHORT softbp.004010A4</b>
004010A0	. 0F31	RDTSC
004010A2	. 50	PUSH EAX
004010A3	. C3	RETN

앞에서 봤던 코드들이다. 바로 0xCC를 찾는 코드인데 F8로 계속 진행을 해보면 엉뚱한 곳으로 점프하는 구문을 벗어나 정상적인 루틴을 실행한다.

또 한가지의 소프트웨어 브레이크포인트를 우회하는 방법은 Import 테이블을 이용하는 것이다. 즉, 코드에 브레이크포인트를 설정하는 것이 아니라 임포트되는 함수에 브레이크포인트를 설정하는 것이다.

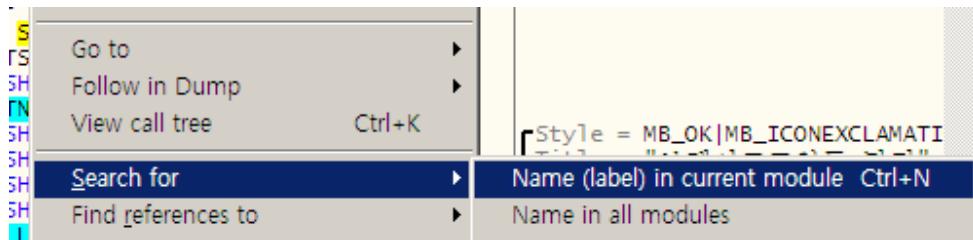
Address	Hex dump	Disassembly
0040102F	. 68 15304000	PUSH Software.00403015
00401034	. 6A 00	PUSH 0
<b>00401036</b>	<b>. E8 07000000</b>	<b>CALL &lt;JMP.&amp;user32.MessageBoxA&gt;</b>
0040103B	. 5B	POP EBX

코드에 브레이크포인트를 설정하고 F9로 실행하면 익셉션이 발생해서 프로세스가 종료된다.

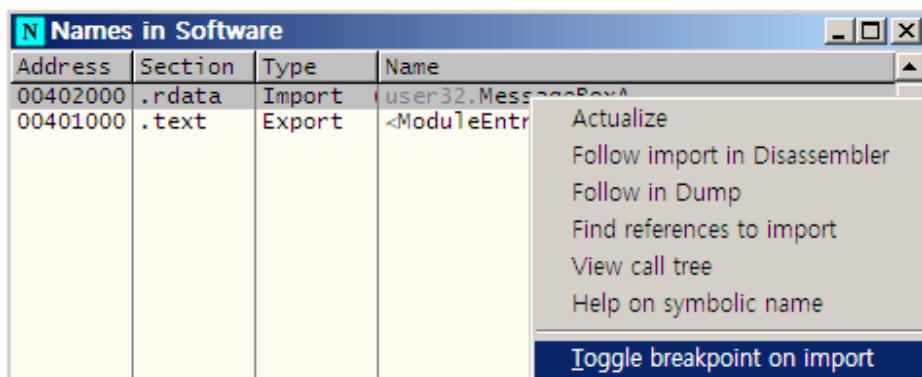
Access violation when executing [75A72A5C] - use Shift+F7/F8/F9 to pass exception to program

## Reverse Engineering

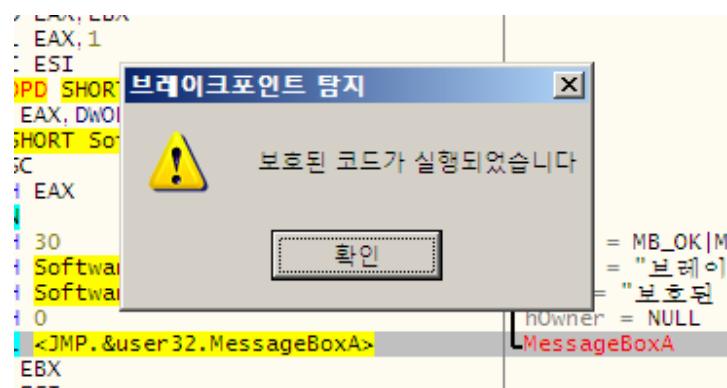
다음과 같이 현재 모듈에서의 이름을 검색한다.(단축키) CTRL+N)



그런 후 아래와 같이 MessageBoxA 함수에 브레이크포인트를 설정한다.



그리고 F9로 실행을 하면 정상적으로 실행이 된다.



### 6-3. TLS Callback

#### TLS Callback



#### TLS (Thread Local Storage) Callback

- 프로세스가 초기화되기 전과 종료된 후에 호출
- 프로그램의 엔트리포인트 이전에 실행
- 디버거 탐지 또는 언패킹 루틴을 TLS Callback에 등록시켜서 디버거로 오픈할 때 프로그램을 종료시킨다.
- PE 헤더 중 Data Directory 엔트리 중 10번째인 IMAGE\_DIRECTORY\_ENTRY\_TLS에 정보가 들어 있다.

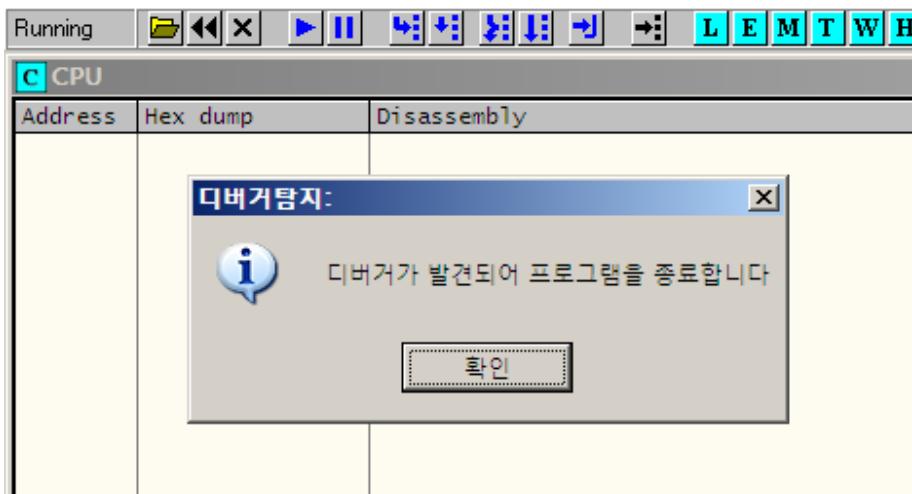
#### Student Notes

TLS Callback 기법은 악성코드나 패커와 같으 프로그램에서 안티 리버싱 기법으로 자주 사용되는 기법이다. TLS Callback은 쓰레드가 생성될 때 로더에 의해서 실행되는 코드이다. 즉, 엔트리포인트에 도달하기 전에 실행되는 것이다.

만약 TLS Callback에 대한 지식이 없다면 상당히 당황스러울 것이다. TLS Callback 부분에 디버거 탐지 루틴이나 언패킹 루틴이 들어 있다면 디버거로 파일을 열면 죽고, OEP를 찾아야 하는데 어느 지점에서 언패킹이 이루어지는지를 찾지 못하기 때문이다.

## Reverse Engineering

디버거에서 예제 파일을 열어보도록 하겠다.



코드가 전혀 보이지 않고 바로 디버거가 발견되어 프로그램을 종료한다는 메시지창이 뜨고 프로세스가 종료된다. 이것은 메인 프로그램이 시작하기 전에 어떠한 함수에 의해서 디버거를 탐지했다는 것을 알 수 있다. TLS 가 무엇인지 알았다면 TLS Callback 에 의해서 디버거가 탐지되었다는 것을 알 수 있다. 즉, 우리는 TLS Callback 함수가 실행하는 위치를 찾아서 디버거 탐지 루틴을 우회하면 된다.

먼저 PE 파일에서 Data Directory 에 있는 TLS 에 대해서 알아보도록 하자. TLS 는 데이터 딕토리 테이블에서 10 번째에 위치하고 있다. Stud\_PE 를 이용하여 예제 파일을 열어 보자.



Data Dir 에서 IMAGE\_DIR\_ENTRY\_TLS 를 선택하면 TLS 에 대한 정보를 확인할 수 있습니다. 앞에서부터 RVA, Size, Raw 이다. PE 파일 구조에서 보았던 것처럼 Raw 값은 Stud-PE 프로그램이 계산해 준 값으로 실제 PE 헤더에는 존재하지 않는 값이다. RVA 값을 이용하여 파일상에서의 위치를 구할 수 있다.

No	Name	VirtualSize	VirtualOffset	RawSize	RawOffset	Characteristics
01	.text	00000062	00001000	00000200	00000400	60000020
02	.rdata	000000AE	00002000	00000200	00000600	40000040
03	.data	00000061	00003000	00000200	00000800	C0000040

TLS 의 RVA – TLS 가 위치하고 있는 섹션의 VirtualOffset + RawOffset = 3060 – 3000 + 800 = 860

## Reverse Engineering

TLS 테이블의 위치는 파일상에서 0x860 이다. Winnt.h에 정의되어 있는 IMAGE\_TLS\_DIRECTORY 는 다음과 같다.

```
typedef struct _IMAGE_TLS_DIRECTORY32 {
    DWORD StartAddressOfRawData;
    DWORD EndAddressOfRawData;
    PDWORD AddressOfIndex;
    PIMAGE_TLSCallback *AddressOfCallbacks;
    DWORD SizeOfZeroFill;
    DWORD Characteristics;
} IMAGE_TLS_DIRECTORY32;
```

네 번째 멤버인 AddressOfCallbacks 에 TLS Callback 의 주소가 들어 있다. Winhex 로 파일을 열어보자.

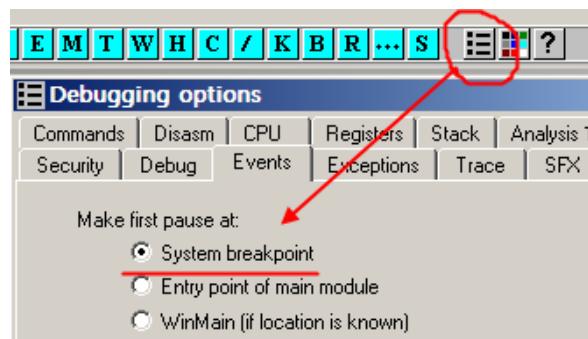
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000860	78	30	40	00	7C	30	40	00	80	30	40	00	84	30	40	00
00000870	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000880	00	00	00	00	1B	10	40	00	00	00	00	00	00	00	00	00

TLS Callback 의 주소는 0x00403084 이다. 이 주소는 RVA 값이 아닌 가상주소이다. 파일상에서 TLS Callback 의 내용을 보려면 한 번 더 계산을 해야 한다. 가상주소가 0x00403084 이므로 RVA 값은 ImageBase 를 뺀 0x3084 가 된다. ( $3084 - 3000 + 800 = 884$ )

TLS 의 파일상에서의 위치는 884 가 된다. Winhex 에서 확인해 보자.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000860	78	30	40	00	7C	30	40	00	80	30	40	00	84	30	40	00
00000870	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000880	00	00	00	00	1B	10	40	00	00	00	00	00	00	00	00	00

TLS Callback 의 주소가 0040101B 이다. 디버거가 프로그램을 시작하는 위치를 메인 모듈의 Entry Point 가 아닌 System breakpoint 로 설정한다. 이것은 프로그램이 로더에 의해서 메모리에 적재되기 전 위치에서부터 디버깅을 시작하겠다는 의미이다.



## Reverse Engineering

디버거에서 프로그램을 시작하면 다음과 같이 System breakpoint에서 멈춘다.

Address	Hex dump	Disassembly
7C95A3E2	C3	RETN
7C95A3E3	90	NOP
7C95A3E4	8BFF	MOV EDI, EDI
7C95A3E6	CC	INT3
7C95A3E7	C3	RETN
7C95A3E8	8BFF	MOV EDI, EDI
7C95A3EA	8B4424 04	MOV EAX, DWORD PTR SS:[ESP+4]
7C95A3EE	CC	INT3
7C95A3EF	C2 0400	RETN 4
7C95A3F2	64:A1 18000000	MOV EAX, DWORD PTR FS:[18]
7C95A3F8	C3	RETN

TLS Callback의 주소인 0040101B로 이동해보자. CTRL+G를 눌러 이동할 주소값을 적는다. 0040101B의 위치로 가면 디버거 탐지 루틴이 바로 보인다.

Address	Hex dump	Disassembly
0040101B	. 803D 60304000	CMP BYTE PTR DS:[403060], 1
00401022	.~ 74 2B	JE SHORT IsDebugg.0040104F
00401024	. C605 60304000	MOV BYTE PTR DS:[403060], 1
0040102B	. E8 2C000000	CALL <JMP.&kernel32.IsDebuggerPresent>
00401030	. 83F8 01	CMP EAX, 1
00401033	.~ 75 1A	JNZ SHORT IsDebugg.0040104F
00401035	. 6A 40	PUSH 40
00401037	. 68 00304000	PUSH IsDebugg.00403000
0040103C	. 68 0D304000	PUSH IsDebugg.0040300D
00401041	. 6A 00	PUSH 0
00401043	. E8 08000000	CALL <JMP.&user32.MessageBoxA>
00401048	. 6A 00	PUSH 0
0040104A	. E8 07000000	CALL <JMP.&kernel32.ExitProcess>
0040104F	> C3	RETN

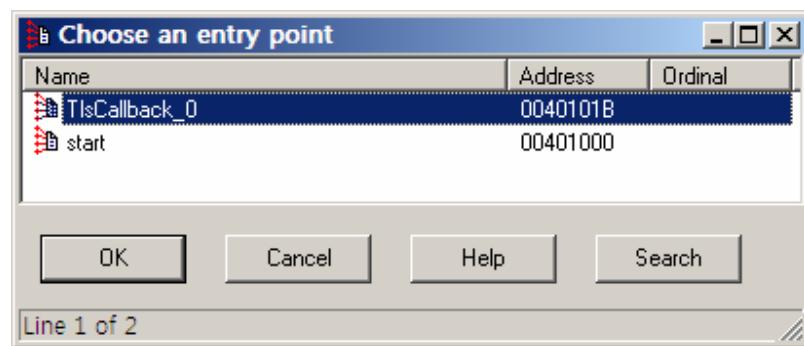
0040101B에 브레이크포인트를 걸고 F9를 눌러 실행을 시킨다. 그리고 IsDebuggerPresent를 우회하면 된다.

이 예제의 경우 TLS Callback에 비교적 간단하고 눈에 잘 보이는 디버거 탐지 루틴이 있지만 다른 방식으로 디버거 탐지 루틴을 복잡하게 구성하거나 쓰레기 코드들을 많이 추가하고, 발견되었을 때도 단순 종료가 아닌 엉뚱한 주소로 점프하게 만들어 디버거가 코드를 잘못 해석하게 만들 수도 있다.

TLS Callback을 우회하는 방법은 IMAGE\_DIRECTORY\_ENTRY\_TLS에 주소값이 들어 있는지 확인하고 있을 경우 TLS 테이블 내에 있는 AddressOfCallbacks의 값을 00000000으로 바꿔주면 된다.

TLS Callback 함수 주소를 알아내는 방법은 위의 방법을 사용해도 되고 IDA를 이용하면 쉽게 찾아낼 수 있다. IDA로 분석 할 파일을 오픈한 후 CTRL+E를 눌러보면 다음과 같이 쉽게 찾을 수 있다.

## Reverse Engineering



## 6-4. Process Attach 기법

### Process Attach 기법



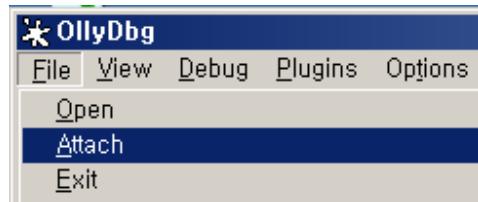
- 일반적으로 혼자서 실행이 불가능한 파일 분석시 사용
- dll이나 ocx 파일 분석시 사용
- 다른 프로그램에 인젝션 되어 정상적인 프로세스를 봐야 하는 경우에도 사용

### Student Notes

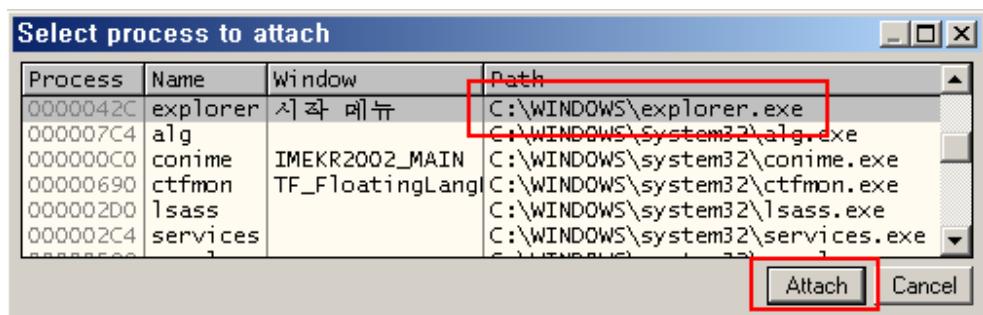
프로세스 어태치 기법은 일반적으로 단독으로 실행이 불가능한 DLL이나 OCX와 같은 파일을 분석할 때 사용되는 기법이다. 악성코드 중 DLL로 배포되면서 특정 프로세스에 인젝션이 되는 경우가 있는데 이런 파일들을 분석할 때 프로세스 어태치 기법을 사용한다.

대표적인 예로 DLL Injection 기법을 사용하는 악성코드의 경우 해당 dll이 인젝션된 프로세스를 찾고 그 프로세스를 통해서 DLL을 attach 시켜서 분석을 하게 된다. 간단한 예제를 통해서 분석 기법에 대해서 알아보도록 하겠다.

디버거를 실행시키고 메뉴에서 File → Attach 를 클릭 한다.



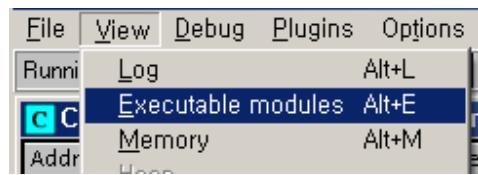
Attach 를 클릭하면 현재 실행되고 있는 프로세스들의 목록을 보여주고 어태치 할 프로세스를 선택하면 된다.



여기서는 explorer.exe 파일을 어태치 해보도록 하겠다. 어태치 할 파일을 선택하면 디버거에서 해당 프로세스의 코드를 읽어 오고 Pause 상태에서 멈추게 된다. 이 때 다시 한번 Start 를 눌러서 시작시켜 준다.



프로세스를 시작시킨 후 현재 이 프로세스와 함께 동작하고 있는 모듈들을 확인한다.



Executable modules 를 클릭하면 해당 프로세스가 사용하고 있는 파일들의 목록이 나온다. 대부분 dll 파일이다. 이 파일들의 목록 중 분석하고자 하는 dll 이 있다면 해당 dll 을 마우스 오른쪽 클릭하여 View names 로 함수들의 이름을 확인할 수 있다.

## Reverse Engineering

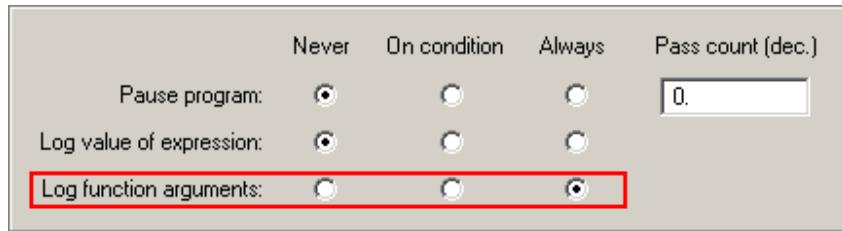
The screenshot shows the Immunity Debugger interface. In the top-left, there's a list of DLL paths under 'Path'. The entry 'C:\WINDOWS\system32\rdshost.dll' is highlighted with a red box. A context menu is open over this entry, listing options like 'Actualize', 'View memory', 'View code in CPU', 'Enter', 'Follow entry', 'Dump data in CPU', 'View names' (which is highlighted with a blue box), 'Ctrl+N', and 'Mark as non-system DLL'.

Address	Section	Type	Name
01644134	.rdata	Import	USER32.BlockInput
01644110	.rdata	Import	USER32.CloseClipboard
01644040	.rdata	Import	KERNEL32.CloseHandle
01644004	.rdata	Import	ADVAPI32.CloseServiceHandle
01644184	.rdata	Import	ole32.CoCreateInstance
01644180	.rdata	Import	ole32.CoInitialize
01644188	.rdata	Import	ole32.CoUninitialize
01644060	.rdata	Import	KERNEL32.CreateMutexA
01644020	.rdata	Import	KERNEL32.CreateProcessA
0164401C	.rdata	Import	KERNEL32.CreateThread
016440D4	.rdata	Import	MSVCRT._CxxFrameHandler
01644064	.rdata	Import	KERNEL32.DisableThreadLibraryCalls
01644080	.rdata	Import	MSVCRT._dllonexit
01644118	.rdata	Import	USER32.EmptyClipboard
01644104	.rdata	Import	PSAPI.EnumProcesses
01644100	.rdata	Import	PSAPI.EnumProcessModules
01644000	.rdata	Import	ADVAPI32.EnumServicesStatusA

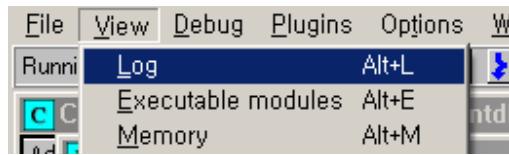
만약 특정 함수가 하는 행동들을 살펴보고 싶다면 해당 함수에 로그 브레이크포인트를 설정한다.

The screenshot shows the Immunity Debugger interface with the 'Imports' list visible. The entry 'MSVCR71.dll strcmp' is highlighted with a red box. A context menu is open over this entry, listing options like 'Actualize', 'Follow import in Disassembler', 'Follow in Dump', 'Find references to import', 'View call tree', 'Help on symbolic name', 'Ctrl+F1', 'Toggle breakpoint on import' (which is highlighted with a blue box), 'Conditional breakpoint on import', 'Conditional log breakpoint on import' (which is also highlighted with a blue box), and 'Set breakpoint on every reference'.

'Conditional log breakpoint on import'는 임포트되는 함수에서 특정 행동이 발생시 브레이크포인트를 설정하는데 코드의 실행이 멈추는 브레이크포인트가 아닌 로그만 남기는 브레이크포인트이다.



위와 같이 프로그램을 멈추지 말고 아규먼트에 대해서만 로그를 남기게끔 설정하여 분석하고자 하는 dll 이 어떤 행동을 하는지 살펴보면 된다. 확인할 때는 View → Log에서 확인하면 된다.



좀 더 원활한 분석을 원한다면 Log 창에서는 지금까지 남아 있던 로그들을 지우고 보는 것이 좋다.

일부 악성코드들은 이런 어태치 기법을 통해서 분석되지 않게 하기 위해 어태치가 발생하면 디버거를 종료시키는 루틴을 가지고 있다. 디버거가 실행중인 프로그램을 어태치하려고 하면 반드시 ntdll.dll 내의 ZwContinue 를 CALL 해야 한다. Anti-Attach 의 기법들 중 ZwContinue 를 후킹해서 다른 루틴(Anti-Debugging 루틴)으로 점프 시키는 기법이 존재하는데 이럴 경우 디버거에서 어태치해서 분석하는 것이 힘들어지게 된다.

Ollydbg 의 플러그인인 AttachAnyway 는 현재 열려있는 모든 프로세스들의 가상메모리공간에서 ZwContinue 의 시작주소를 확인함으로써 후킹 여부를 판단하여 해당 프로그램 존재시 프로그램명을 알려준다. 그리고 후킹코드를 ZwContinue 원래의 코드로 다시 덮어 씌워서 어태치가 가능하게 해준다.

**Reverse Engineering**



---

## Module 7 — 분석 실전

### Objectives

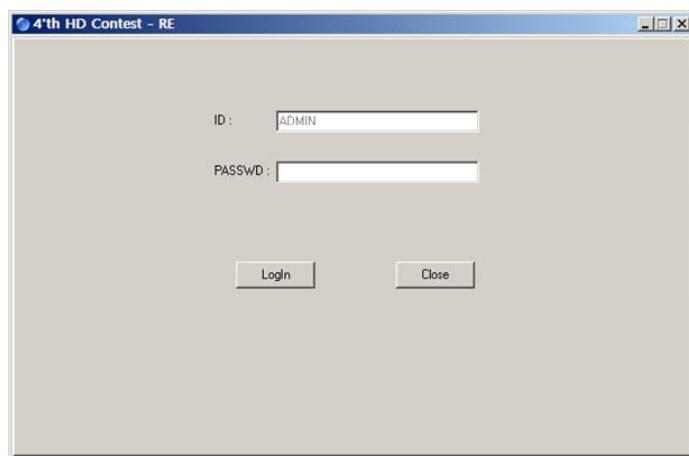
- 온라인 해킹대회 문제 분석
- 악성코드 분석 - shadowbot

### 7-1. 온라인 해킹대회 문제 분석

#### 온라인 해킹대회 문제 분석



- KISA 제4회 해킹방어대회 리버스 엔지니어링 문제
  - Unpacking
  - Steganography



#### Student Notes

이번 모듈에서는 KISA에서 주관하고 있는 해킹방어대회 문제 중 리버스 엔지니어링 기법이 적용되는 문제를 풀어보도록 하겠다. 2007년에 있었던 제 4 회 해킹방어대회 예선 문제 중 스테이지 8을 보도록 하자.

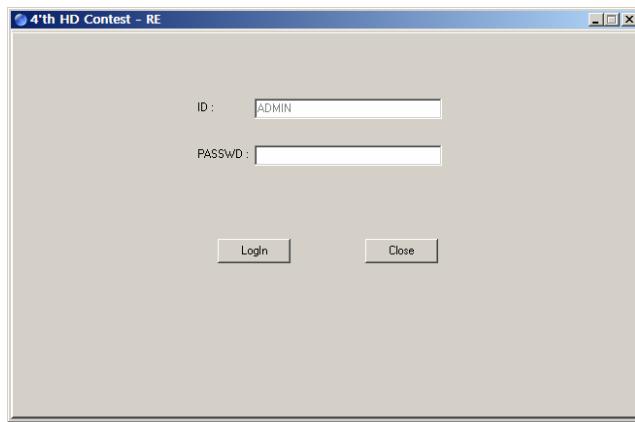
#### 스테이지 8

8 번문제 : 평소 업무시간에 메신저를 즐겼던 하모씨는 최근 사내에 필수 소프트웨어가 설치된 이후로 메신저를 사용할 수 없었다. 해당 프로그램을 크랙하여 그 프로그램의 패스워드를 획득하라.

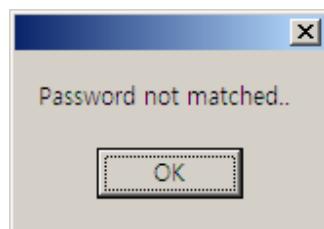
ID: stage8, PASSWORD: 획득한 패스워드

## Reverse Engineering

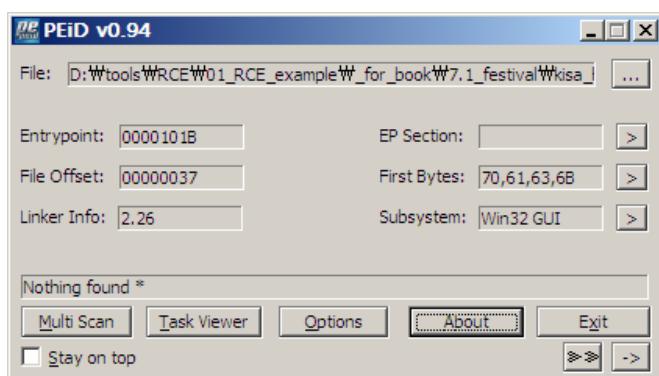
이 문제는 언패킹을 한 후 추출된 파일에서 스테가노그라피 기법을 이용하여 푸는 문제이다.  
먼저 파일을 실행시켜보겠다.



패스워드를 입력해보면 다음과 같이 패스워드가 일치하지 않는다는 메시지를 볼 수 있다.

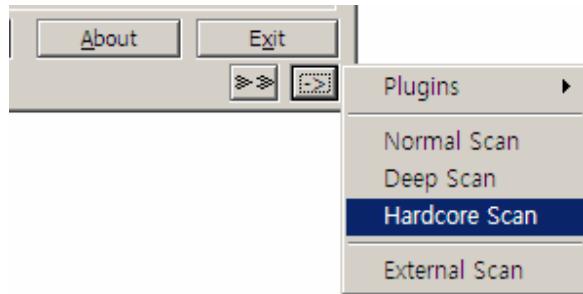


먼저 패킹이 되어 있는지 확인해보겠다. PEiD 를 이용해서 확인한 결과 다음과 같이 Nothing found 라는 메시지를 볼 수 있다.

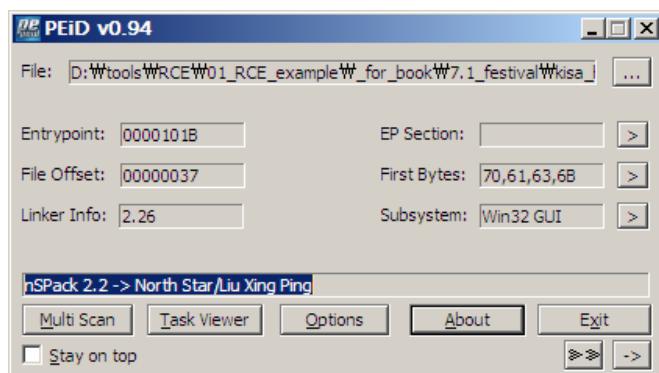


그렇다면 패킹이 되어 있지 않은 것일까? 좀 더 정확한 확인을 위해 오른쪽 하단에 보이는 → 버튼을 클릭해서 Hardcore Scan 을 클릭해 보자.

## Reverse Engineering



이번에는 nSPack 2.2 → North Star/Liu Xing Ping 이라고 나오는 것을 볼 수 있다. 물론 PEiD는 시그내처 기반인기 때문에 userdb.txt 가 업데이트되어야만 nSPack로 패킹되었다는 메시지가 보일 것이다.



그럼 디버거를 통해 파일을 오픈 해보겠다.

Address	Hex	dump	Disassembly	Comment
006E101B	\$- E9 33F11D00		JMP kisa_hd.008C0153	
006E1020	B4		DB B4	
006E1021	09		DB 09	
006E1022	BA		DB BA	
006E1023	0B		DB 0B	
006E1024	01		DB 01	
006E1025	CD		DB CD	
006E1026	21		DB 21	
006E1027	B4		DB B4	
006E1028	4C		DB 4C	
006E1029	CD		DB CD	
006E102A	. 21 70 61 63	ASCII "!	JMP kisa_hd.008C0153	CHAR '!' !
006E103A	. 6B 24 40 00	ASCII "k\$@", 0		CHAR 'L'
006E103E	00		DB 00	
006E103F	00		DB 00	
006E1040	. 50 45 00	ASCII "PE", 0		CHAR 'L'
006E1043	00		DB 00	
006E1044	4C		DB 4C	
006E1045	01		DB 01	

지금까지 보았던 화면과는 좀 다른 화면이 보인다. 이것은 디버거가 정상적으로 코드를 해석하지 못하기 때문이다. 그런데 위에 조금 내려와 보면 “!packed by nspack\$@”라는 문자열이 보이는 것으로 봐서

nspack 으로 패킹이 되어 있다는 것을 알 수 있다.

프로그램을 실행하면 패킹이 풀릴 것이고 메모리에 로드될 것이다. 어느 부분에서 언패킹이 되는지 확인하기 위해 Memory map 을 열어보자. 메모리 맵은 메뉴에서 View → Memory 를 선택하거나 ALT+M 을 눌러서 확인할 수 있다.

Memory map								
Address	Size	Owner	Section	Contains	Type	Access	Initial	
00280000	00041000				Map	R	R	
002D0000	00041000				Map	R	R	
00320000	00006000				Map	R	R	
00330000	00041000				Map	R	R	
006E0000	00001000	kisa_hd		PE header	Image	R	RWE	
006E1000	001DF000	kisa_hd		code	Image	R	RWE	
008C0000	001DF000	kisa_hd		data, imports	Image	R	RWE	
7C800000	00001000	kernel32		PE header	Image	R	RWE	
7C801000	0008A000	kernel32	.text	code, imports	Image	R	RWE	
7C88B000	00005000	kernel32	.data	data	Image	R	RWE	
7C890000	000A7000	kernel32	.rsrc	resources	Image	R	RWE	
7C937000	00007000	kernel32	.reloc	relocations	Image	R	RWE	
7C940000	00001000	ntdll		PE header	Image	R	RWE	
7C941000	00086000	ntdll	.text	code, exports	Image	R	RWE	
7C9C7000	00006000	ntdll	.data	data	Image	R	RWE	
7C9CD000	00048000	ntdll	.rsrc	resources	Image	R	RWE	
7CA15000	00004000	ntdll	.reloc	relocations	Image	R	RWE	

PE 헤더가 006E0000 에 위치한 것으로 보아 ImageBase 가 이 부분이라는 것을 알 수 있다. 물론 패킹된 프로그램의 코드가 이 부분에서부터 시작하는 것이다. 그럼 F9 를 눌러 실행을 시켜서 메모리에 실제 코드가 로드가 되는지 확인해 보자.

Memory map								
Address	Size	Owner	Section	Contains	Type	Access	Initial	
003D0000	00005000				Priv	RW	RW	
003E0000	00001000				Priv	RWE	RWE	
003F0000	00001000				Map	RW	RW	
00400000	001DC000				Priv	RWE	RWE	
005E0000	0000A000				Map	R E	R E	
006A0000	00002000				Map	R E	R E	
006B0000	00002000				Map	R	R	
006C0000	00004000				Priv	RW	RW	
006D0000	00002000				Map	R	R	
006E0000	00001000	kisa_hd		PE header	Image	R	RWE	
006E1000	001DF000	kisa_hd		code	Image	R	RWE	
008C0000	001DF000	kisa_hd		data, imports	Image	R	RWE	
00AA0000	00103000				Map	R	R	
00BB0000	00107000				Map	R E	R E	
00EB0000	00005000				Priv	RW	RW	
010AB000	00001000				Priv	RW	Guar	
010AC000	00004000			stack of thr	Priv	RW	Guar	
010B0000	00004000				Priv	RW	RW	
011B0000	00010000				Map	RW	RW	

## Reverse Engineering

실행을 해보니 ImageBase 로 자주 볼 수 있는 00400000 이 생겼다. 아마도 패킹되었던 부분이 풀리면서 이 주소에 코드가 로드된 것이라 예상해볼 수 있습니다. 그러면 메모리에 공간을 할당하고 이 주소에 쓰기를 할 때를 포착하면 언패킹된 정상적인 코드를 볼 수 있을 것이다.

CTRL+F2 로 다시 시작을 시키고 메모리에 공간을 할당할 때 쓰이는 VirtualAlloc 함수에 브레이크포인트를 설정하여 메모리에 공간을 할당하는 순간을 찾아서 앞에서 보았던 00400000 의 주소가 할당되는지 확인해 보겠다. 먼저 VirtualAlloc 함수가 어떻게 생겼는지 확인해 보도록 하자.

```
LPVOID VirtualAlloc(
    LPVOID lpAddress,           // address of region to reserve or commit
    DWORD dwSize,              // size of region
    DWORD  fAllocationType,     // type of allocation
    DWORD  fProtect          // type of access protection
);
```

첫 번째 인자가 메모리에 할당되는 주소이다. 이것을 확인하도록 하자. 하단에 Command bar 에서 다음과 같이 브레이크포인트를 설정한다.



VirtualAlloc 함수에 브레이크포인트를 설정하니 7C8245A9 라는 주소에 브레이크포인트가 설정되었다. F9 를 눌러 실행을 하고 스택창을 확인하여 00400000 이라는 주소에 메모리 공간을 할당하는 코드를 확인하도록 한다. F9 를 한번 실행하고 VirtualAlloc 함수가 호출될 때 브레이크포인트가 걸릴 때 스택은 다음과 같다.

Address	Value	Comment
0013FF6C	008C01BD	CALL to VirtualAlloc from kisa_hd.008C01B7
0013FF70	00000000	Address = NULL
0013FF74	00001000	Size = 1000 (4096.)
0013FF78	00001000	AllocationType = MEM_COMMIT
0013FF7C	00000040	Protect = PAGE_EXECUTE_READWRITE

위와 같이 Address 가 NULL 이다. 이 Address 가 00400000 인 곳을 찾아야 한다. F9 를 눌러 몇 번 더 실행하다 보면 다음과 같이 Address 가 00400000 인 곳을 찾을 수 있다.

Address	Value	Comment
0013FD6C	006E134C	CALL to VirtualAlloc
0013FD70	00400000	Address = 00400000
0013FD74	001DC000	Size = 1DC000 (1949696.)
0013FD78	00003000	AllocationType = MEM_COMMIT   MEM_RESERVE
0013FD7C	00000040	Protect = PAGE_EXECUTE_READWRITE

## Reverse Engineering

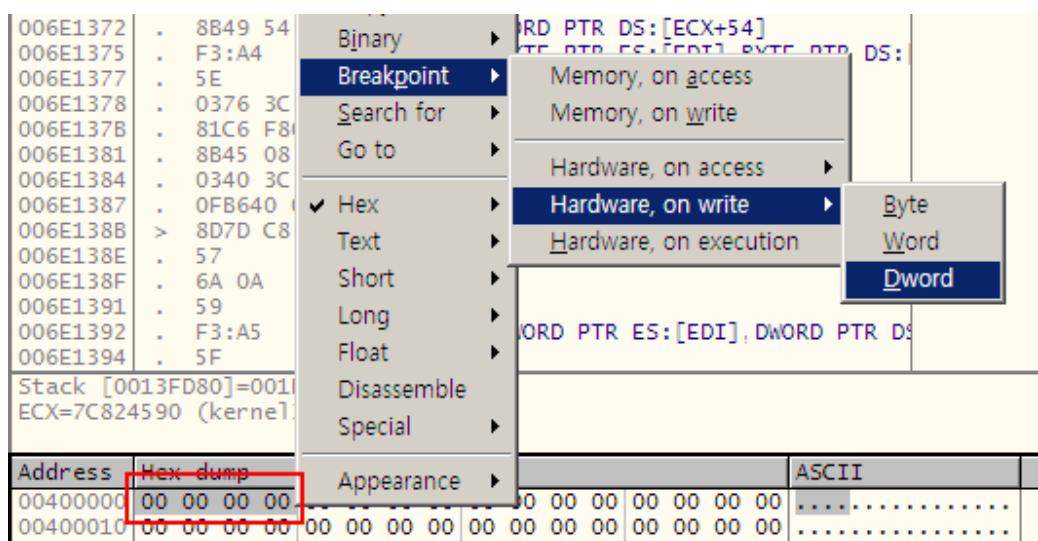
F8 을 눌러 VirtualAlloc 함수를 빠져 나오면 다음과 같이 이상한 문자열들만 보이는 코드 부분에 도달할 수 있다.

Address	Hex dump	Disassembly	Comment
006E134C	59	DB 59	CHAR 'Y'
006E134D	85	DB 85	
006E134E	C0	DB C0	
006E134F	75	DB 75	CHAR 'u'
006E1350	13	DB 13	
006E1351	6A	DB 6A	CHAR 'j'
006E1352	40	DB 40	CHAR '@'
006E1353	68	DB 68	CHAR 'h'

이렇게 보이는 이유는 디버거에서 제대로 해석을 하지 못했기 때문이다. CTRL+A 를 누르면 다시 해석을 해서 화면에 실제 코드들을 보여주게 된다. 그리고 메모리맵을 확인해보니 아직 공간만 할당한 상태이고 코드가 메모리에 로드되기 전이라는 것을 알 수 있다.

Memory map								
Address	Size	Owner	Section	Contains	Type	Access	Initial	
002D0000	00041000				Map	R	R	
00320000	00006000				Map	R	R	
00330000	00041000				Map	R	R	
00400000	001DC000				Priv	RWE	RWE	
006E0000	00001000	kisa_hd		PE header	Image	R	RWE	
006E1000	001DF000	kisa_hd		code	Image	R	RWE	

패키가 프로그램 실행을 위해 언패킹을 하면 이 위치에 코드를 언팩을 하게 된다. 그래서 이 주소에 하드웨어 브레이크포인트를 건다. 이 때 하드웨어 브레이크포인트는 write 로 한다. 왜냐하면 이 주소에 코드를 쓰기(write)를 하는 시점을 잡기 위함이다.



## Reverse Engineering

하드웨어 브레이크포인트가 설정되었으면 F9를 눌러 실행해 본다.

006E1375	. F3:A4	REP MOVS BYTE PTR ES:[EDI], BYTE PTR DS:
006E1377	. 5E	POP ESI
006E1378	. 0376 3C	ADD ESI, DWORD PTR DS:[ESI+3C]
006E137B	. 81C6 F8000000	ADD ESI, OF8
006E1381	. 8B45 08	MOV EAX, DWORD PTR SS:[EBP+8]
006E1384	. 0340 3C	ADD EAX, DWORD PTR DS:[EAX+3C]
006E1387	. 0FB640 06	MOVZX EAX, BYTE PTR DS:[EAX+6]
006E138B	> 8D7D C8	LEA EDI, DWORD PTR SS:[EBP-38]
006E138E	. 57	PUSH EDI
006E138F	. 6A OA	PUSH OA
006E1391	. 59	POP ECX
006E1392	. F3:A5	REP MOVS DWORD PTR ES:[EDI], DWORD PTR DS:
006E1394	. 5F	POP EDI
ECX=000003FF (decimal 1023.)		
DS:[ESI]=[006E1A47]=5A ('Z')		
ES:[EDI]=[00400001]=00		
Address	Hex dump	ASCII
00400000	4D 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   M.....	
00400010	00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   .....	

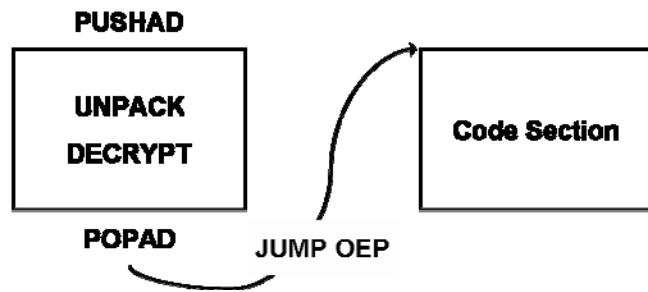
실행을 했더니 방금 전에 할당 받았던 주소인 00400000에 내용이 생겼다. 이 주소에 write를 하고 있기 때문에 브레이크포인트에 의해서 멈추게 된 것이다. 아래쪽으로 계속 내려가서 RETN이 있는 곳에 브레이크포인트를 설정한다. (하드웨어 브레이크포인트는 제거한다)

Address	Hex dump	Disassembly
006E15B6	. 5B	POP EBX
006E15B7	. C9	LEAVE
006E15B8	. C2 0C00	RETN OC
006E15BB	\$ C745 BC 0000	MOV DWORD PTR SS:[EBP-44], 0

그리고 F9를 눌러 실행한다. RETN OC에서 멈출 것이고 F8을 눌러 계속 진행한다.

Address	Hex dump	Disassembly
006E1180	. 5F	POP EDI
006E1181	. 5E	POP ESI
006E1182	. 5D	POP EBP
006E1183	. 83C4 04	ADD ESP, 4
006E1186	. 5B	POP EBX
006E1187	. 5A	POP EDX
006E1188	. 83C4 08	ADD ESP, 8
006E118B	. 894C24 04	MOV DWORD PTR SS:[ESP+4], ECX
006E118F	. FFEO	JMP EAX
006E1191	\$ 55	PUSH EBP
006E1192	. 89E5	MOV EBP, ESP

F8로 진행을 하니 레지스터들을 스택에서 꺼내는 것이 보이고 0045972C로 점프하는 것이 보인다. 보통 패킹된 프로그램들은 PUSHAD를 통해 레지스터들을 스택에 백업한 후 언패킹 루틴이나 복호화 루틴을 거쳐서 실제 코드를 풀어낸 다음 POPAD를 통해서 레지스터들을 복구한다.

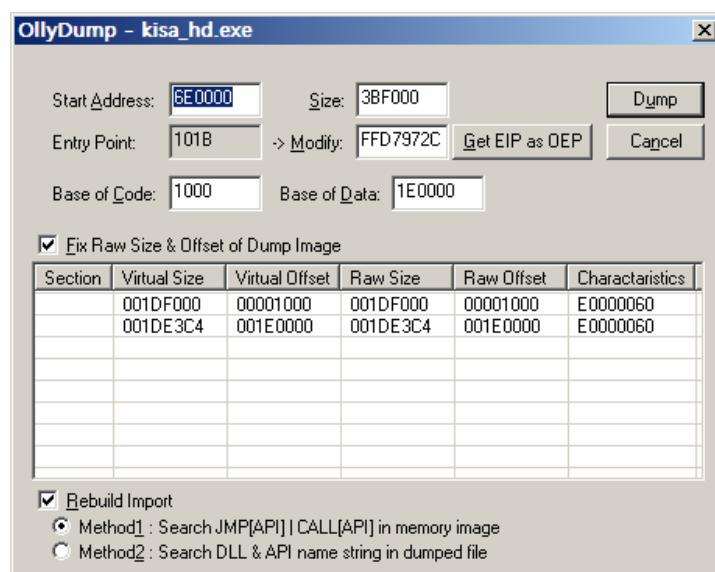


앞의 코드는 POPAD 는 아니지만 POP 명령어로 레지스터들을 복구하는 것이 보이고 JMP EAX 에 의해 서 0045972C 로 점프한다. 바로 이 점프한 후 나오는 부분이 OEP 이다.

Address	Hex dump	Disassembly	Comment
0045972C	55	PUSH EBP	OEP
0045972D	8BEC	MOV EBP, ESP	
0045972F	83C4 F0	ADD ESP, -10	
00459732	B8 4C954500	MOV EAX, 4C954C	
00459737	E8 D8C9FAFF	CALL 00406114	
0045973C	A1 D8B04500	MOV EAX, DWORD PTR DS:[45B0D8]	
00459741	8B00	MOV EAX, DWORD PTR DS:[EAX]	
00459743	E8 50E0FFFF	CALL 00457798	
00459748	8B0D C8B14500	MOV ECX, DWORD PTR DS:[45B1C8]	

일반적으로 OEP 를 찾으면 덤프를 뜯고 ImportREC 같은 프로그램으로 IAT 를 복구하면 된다. 하지만 이 프로그램은 가상으로 메모리를 할당 받았기 때문에 직접 IAT 를 복구해주어야 한다. 그리고 실행을 하게 되면 IAT 는 다른 값으로 채워지기 때문에 채워지기 전 값으로 바꿔준 뒤 덤프를 하면 된다.

현재 위 코드에서 덤프를 뜯기 위해서 코드창에서 Dump debugged process 를 클릭하면 아래 보이는 것과 같이 이상한 주소가 OEP 로 되어 있는 것을 볼 수 있다.



## Reverse Engineering

IAT를 직접 복구하려면 아무 함수나 하나를 찾아서 확인하면 된다. 00459737 주소에서 엔터를 눌러 함수 안으로 들어가 보면 00406120 주소에 CALL 00406050 이 있다. 다시 한번 이 부분에서 엔터를 눌러 들어가 보면 다음처럼 함수들을 찾을 수 있다.

Address	Hex dump	Disassembly	Comment
00406050	- FF25 E4D14500	JMP DWORD PTR DS:[45D1E4]	kernel32.GetModuleHandleA
00406056	8BC0	MOV EAX, EAX	
00406058	- FF25 E0D14500	JMP DWORD PTR DS:[45D1E0]	kernel32.LocalAlloc
0040605E	8BC0	MOV EAX, EAX	
00406060	- FF25 DCD14500	JMP DWORD PTR DS:[45D1DC]	kernel32.TlsGetValue
00406066	8BC0	MOV EAX, EAX	
00406068	- FF25 D8D14500	JMP DWORD PTR DS:[45D1D8]	kernel32.TlsSetValue
0040606E	8BC0	MOV EAX, EAX	

함수가 있는 코드에서 마우스 오른쪽 클릭한 후 Follow in Dump → Memory address 를 눌러서 해당 함수가 있는 곳으로 따라 가보자.

Address	Hex dump	ASCII
0045D1E4	4A 47 82 7C 00 00 00 00 FA F4 F3 77 86 E0 F3 77	JG?...浹?弔?
0045D1F4	CE 6C F4 77 00 00 00 00 94 DA 81 7C 29 55 82 7C	??.?.?.?.U?
0045D204	7B 1C 82 7C D1 6E 82 7C A9 45 82 7C DE 24 80 7C	{???.?.?.
0045D214	50 05 83 7C 7B 78 80 7C 8B 56 82 7C 11 23 82 7C	P ?{x? ?#?
0045D224	DC C2 82 7C 4D AF 81 7C A0 B0 81 7C 4B 18 80 7C	?.?.M?.?.?K↑?
0045D234	CC 16 82 7C 1B B1 82 7C F9 9B 82 7C C6 1D 80 7C	??-.?. ???.
0045D244	AB A3 95 7C 61 78 82 7C 11 21 81 7C F9 23 81 7C	?.?ax?◀!???
0045D254	78 E6 80 7C A9 21 81 7C DA CB 81 7C 88 37 83 7C	x? ???.?
0045D264	54 A7 81 7C 9B BA 81 7C A2 37 83 7C 74 9D 82 7C	T?.?. ?.??.t.?

헥사코드 덤프창에서 마우스 오른쪽을 클릭한 후 Long → Address 를 눌러서 IAT를 확인한다.

Address	Value	Comment
0045D1E4	7C82474A	kernel32.GetModuleHandleA
0045D1E8	00000000	
0045D1EC	77F3F4FA	ADVAPI32.RegQueryValueExA
0045D1F0	77F3E086	ADVAPI32.RegOpenKeyExA
0045D1F4	77F46CCE	ADVAPI32.RegCloseKey
0045D1F8	00000000	
0045D1FC	7C81DA94	kernel32._strcpyA
0045D200	7C825529	kernel32.WriteFile
0045D204	7C821C7B	kernel32.WaitForSingleObject
0045D208	7C826ED1	kernel32.VirtualQuery
0045D20C	7C8245A9	kernel32.VirtualAlloc
0045D210	7C8024DE	kernel32.Sleep
0045D214	7C830550	kernel32.SizeofResource

자, 그럼 IAT가 어디서부터 어디까지인지 확인해보도록 하자. 임포트 테이블의 마지막은 0000 0000 으로 채워져 있다. 하지만 간혹 임포트 테이블 중간 0000 0000 을 삽입하여 리버서들을 혼돈시키거나 귀찮게 하기도 한다. ImportREC과 같은 프로그램은 자동으로 IAT를 복구시켜주는데 이렇게 0000 0000 이 임포트 테이블 중간에 있을 경우 정상적으로 IAT를 복구하지 못하는 경우가 발생하기도 한다.

Address	Value	Comment
0045D118	7C96C988	ntdll.RtlDeleteCriticalSection
0045D11C	7C95A3AB	ntdll.RtlLeaveCriticalSection
0045D120	7C95A360	ntdll.RtlEnterCriticalSection
0045D124	7C827861	kernel32.InitializeCriticalSection
0045D128	7C823FBE	kernel32.VirtualFree
0045D12C	7C8245A9	kernel32.VirtualAlloc
0045D130	7C822419	kernel32.LocalFree
0045D134	7C82239C	kernel32.LocalAlloc
0045D6A4	00000000	
0045D6A8	7C8024DE	kernel32.Sleep
0045D6AC	00000000	
0045D6B0	775C422A	oleaut32.SafeArrayPtrOfIndex
0045D6B4	775C0FC1	oleaut32.SafeArrayGetUBound
0045D6B8	775C100D	oleaut32.SafeArrayGetLBound
0045D6BC	775C131A	oleaut32.SafeArrayCreate
0045D6C0	775C0C6E	oleaut32.VarianChangeType
0045D6C4	775B4518	oleaut32.VarianCopy
0045D6C8	775B42AF	oleaut32.VarianClear
0045D6CC	775B3F70	oleaut32.VarianInit
0045D6D0	00000000	

IAT의 시작 주소는 0045D118이고 마지막 주소는 0045D6D0이다. 그리고 IAT를 복구하기 위한 또 하나의 정보는 OEP인데 OEP는 앞에서 보았듯이 0045972C이다.

그럼 덮어씌워지기 전의 IAT를 복사하기 위해 CTRL+F2를 눌러 프로그램을 다시 실행시키고 VirtualAlloc에 브레이크포인트를 설정한다. 그리고 앞에서 했던 과정들을 그대로 진행하면서 가상 메모리에 주소 공간을 할당한 후까지인 아래 보이는 코드까지 진행한다.

Address	Hex dump	Disassembly
006E134C	. 59	POP ECX
006E134D	. 85C0	TEST EAX, EAX
006E134F	.~ 75 13	JNZ SHORT kisa_hd.006E1364
006E1351	. 6A 40	PUSH 40
006E1353	. 68 00100000	PUSH 1000

그리고 헥사 덤프창에서 앞에서 구했던 IAT의 시작 주소로 이동한다.

현재는 아무런 값도 들어 있지 않다. IAT의 시작주소인 0045D118에 하드웨어 브레이크포인트를 설정한다. (Breakpoint → Hardware, on write → DWROD) 그리고 F9를 한번 눌러 진행한다. 그러면 헥사 덤프 창에 내용이 채워지는 것을 볼 수 있다.

## Reverse Engineering

F8 을 한번 눌러 계속 진행시켜 나머지 IAT 도 채운다.

Address	Hex dump	ASCII
0045D118	42 D7 05 00 5A D7 05 00 72 D7 05 00 8A D7 05 00	B?.Z?.r?. —.N .E . .S .
0045D128	A6 D7 05 00 B4 D7 05 00 C4 D7 05 00 D0 D7 05 00	—.N .E . .S .
0045D138	DE D7 05 00 EC D7 05 00 02 D8 05 00 1A D8 05 00	捨.因.~?.-?
0045D148	32 D8 05 00 42 D8 05 00 58 D8 05 00 6E D8 05 00	2?.B?.X?.n?.
0045D158	7A D8 05 00 86 D8 05 00 98 D8 05 00 AA D8 05 00	z?. —.S . .H .
0045D168	BC D8 05 00 CE D8 05 00 E2 D8 05 00 F8 D8 05 00	술.房.濟.拉.
0045D178	OA D9 05 00 1C D9 05 00 2A D9 05 00 3C D9 05 00	.?.?.*?.<?
0045D188	48 D9 05 00 56 D9 05 00 62 D9 05 00 7E D9 05 00	H?.V?.b?.~?.
0045D198	8A D9 05 00 9C D9 05 00 00 00 00 00 B8 D9 05 00	丟.賴..... —.J .
0045D1A8	CA D9 05 00 D8 D9 05 00 E6 D9 05 00 00 00 00 00	釋.妹.蓮.....
0045D1B8	00 DA 05 00 14 DA 05 00 24 DA 05 00 00 00 00 00	.?.q?.S?.....
0045D1C8	40 DA 05 00 50 DA 05 00 66 DA 05 00 00 00 00 00	@?.P?.f?.....

IAT 가 모두 채워졌다. 이 부분이 원래의 IAT 이다. 0045D118 부터 0045D6D0 까지 복사를 한다.

Address	Hex dump	ASCII
0045D688	FE ED 05 00 0A EE 05 00 1C EE 05 00 2A EE 05 00	..?..*?.
0045D698	3C EE 05 00 4A EE 05 00 60 EE 05 00 00 00 00 00	<?.J?..?.....
0045D6A8	88 EE 05 00 00 00 00 00 9E EE 05 00 B4 EE 05 00	옛.....월..델 .
0045D6B8	CA EE 05 00 E0 EE 05 00 F2 EE 05 00 06 EF 05 00	賴..標..蓮.-?
0045D6C8	14 EF 05 00 24 EF 05 00 00 00 00 00 40 EF 05 00	¶?.S?.....@?.
0045D6D8	58 EF 05 00 70 EF 05 00 82 EF 05 00 94 EF 05 00	X?.p?.云..蓮 .
0045D6E8	AE EF 05 00 CA EF 05 00 EA EF 05 00 00 F0 05 00	?..感..金..?.
0045D6F8	16 F0 05 00 2C F0 05 00 40 F0 05 00 56 F0 05 00	-?..?.@?.V?.
0045D708	6A F0 05 00 7E F0 05 00 90 F0 05 00 A8 F0 05 00	j?..~?.葉..@ .
0045D718	C0 F0 05 00 D8 F0 05 00 E8 F0 05 00 02 F1 05 00	자..萌..寒..-?.
0045D728	16 F1 05 00 2A F1 05 00 00 00 00 00 6B 65 72 6E	-?..?....kern
0045D738	65 6C 33 32   2E 64 6C 6C 00 00 00 00 44 65 6C 65	e132.dll....Delete

그런데 앞에서 분명 0045D6D0 까지였는데 지금 확인해보니 0045D730 까지라고 되어 있다. 앞에서 확인했던 IAT 는 덮어 씌워진 후이기 때문에 실제와 약간 다를 수도 있다. 0045D730 까지 바이너리를 복사한다. ( Binary → Binary copy)

그리고 조금 전에 설정했던 하드웨어 브레이크포인트를 제거 한다. 상단 메뉴에서 View → Hardware breakpoint 에서 제거하면 된다.

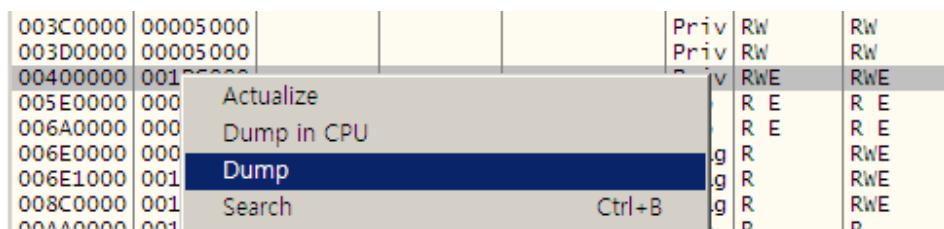
이제 원래 IAT 가 준비가 되었으니 OEP 까지 가보도록 하자. OEP 로 이동한 후 IAT 를 확인해보니 원래의 IAT 대신 새로 구해진 주소값들이 들어가 있는 것을 확인할 수 있다.

Address	Hex dump	ASCII
0045D688	EE B9 DE 77 9A E4 DF 77 01 E4 DF 77 13 E3 DF 77	謹?心?~杏w!!伸w
0045D698	41 C4 DE 77 99 FE DE 77 16 31 DF 77 00 00 00 00	A콤w金?-1?....
0045D6A8	DE 24 80 7C 00 00 00 00 2A 42 5C 77 C1 OF 5C 77	?□ ....*B\w?\w

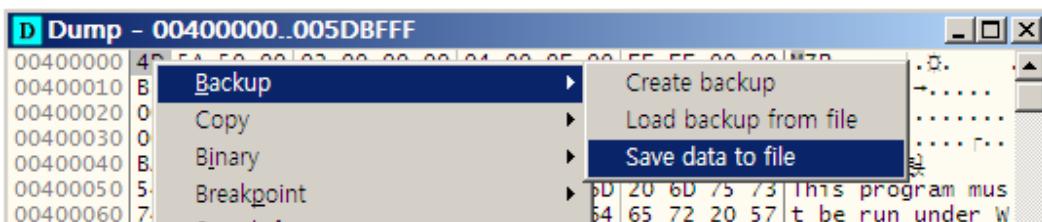
조금 전에 복사했던 바이너리를 이곳에 붙여 넣는다. (Binary → Binary paste)

Address	Hex dump	ASCII
0045D4B8	B4 E6 05 00 C2 E6 05 00 CE E6 05 00 E0 E6 05 00	忒·烈·携·劍··
0045D4C8	FA E6 05 00 16 E7 05 00 28 E7 05 00 38 E7 05 00	見·.-?·(?.8?
0045D4D8	44 E7 05 00 56 E7 05 00 66 E7 05 00 76 E7 05 00	D?,V?,f?,v?,
0045D4E8	86 E7 05 00 94 E7 05 00 A2 E7 05 00 B0 E7 05 00	毒·懈·?.·.結·.
0045D4F8	BE E7 05 00 D0 E7 05 00 E2 E7 05 00 F0 E7 05 00	향·旋·洵·誰·.
0045D508	06 E8 05 00 12 E8 05 00 20 E8 05 00 2E E8 05 00	-?.??.?.?.?
0045D518	3A E8 05 00 46 E8 05 00 58 E8 05 00 6A E8 05 00	:?,F?,X?,j?,
0045D528	76 E8 05 00 84 E8 05 00 90 E8 05 00 A4 E8 05 00	v?,晉·露·.苦·.
0045D538	AE E8 05 00 C0 E8 05 00 D0 E8 05 00 E2 E8 05 00	?·.魄·.秀·.淳·.
0045D548	F0 E8 05 00 FE E8 05 00 1A E9 05 00 2C E9 05 00	烏·.·.-?.,?
0045D558	3C E9 05 00 52 E9 05 00 64 E9 05 00 72 E9 05 00	<?,R?,d?,r?,
0045D568	82 E9 05 00 96 E9 05 00 A6 E9 05 00 BA E9 05 00	是·.晉·.?·.是·.

IAT 를 복구했으니 덤프를 뜹니다. 현재 프로그램이 가상 메모리에 로드되어 있기 때문에 메모리 맵으로 이동해서 덤프를 떠야 한다.

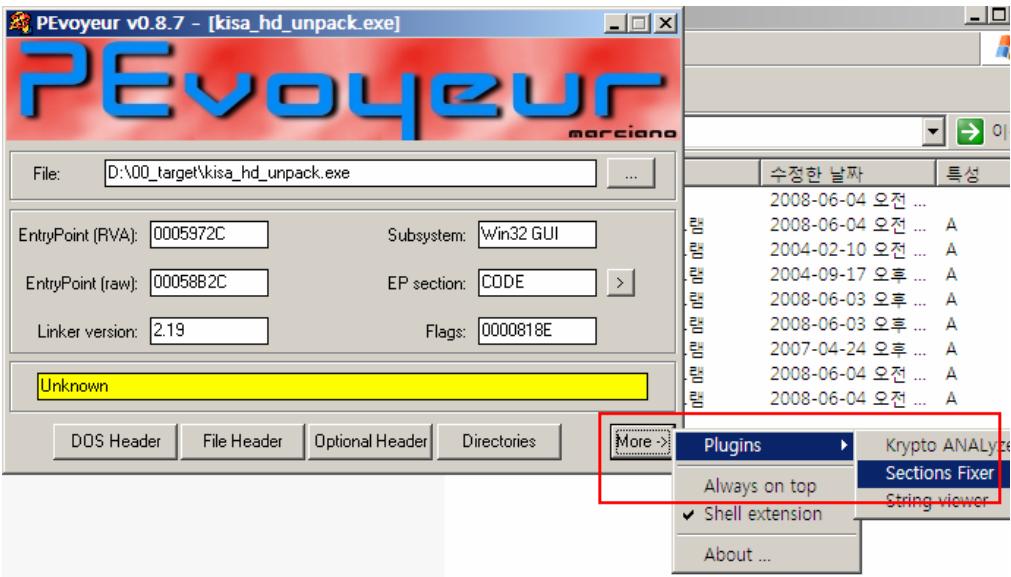


Dump 창이 한 뜰 것이다. 여기서 Backup → Save data to file 을 선택해서 저장한다.

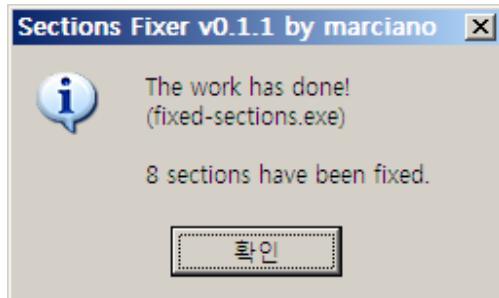


이제 마지막으로 섹션 헤더들의 RVA 값을 맞춰 주어야 한다. PEvoyeur을 이용해서 섹션 헤더들의 RVA 값을 수정하겠다.

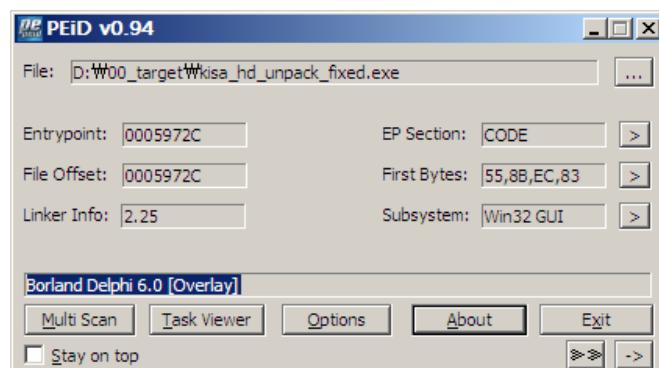
## Reverse Engineering



PEvoyeur에서 오른쪽 아래에 보면 More 옵션에서 Plugins → Sections Fixer을 클릭하면 섹션을 퍽스시켜 준다. 그리고 간단한 정보창이 뜨고 섹션 헤더들의 정보의 FIX가 끝난다.



퍽스된 파일은 fixed-sections.exe라는 파일명으로 저장된다. 실행해보면 원래 파일과 동일한 화면과 동일한 결과가 나온다. 언패킹된 파일은 Borland Delphi 6.0으로 작성된 프로그램이었다.



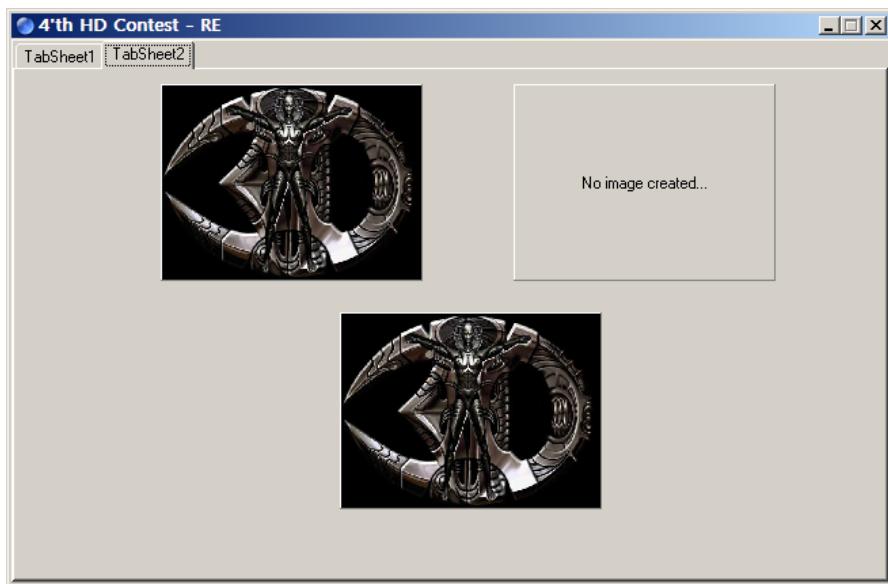
그럼 이제 디버거를 통해 언패킹된 파일을 불러 와서 패스워드를 입력받은 패스워드를 프로그램 가지고 있는 프로그램과 비교한 후 성공/실패 메시지를 뿐려줄 것이다. 하지만 이 파일은 디버거를 통해서 확인해도 올바른 패스워드를 구할 수 없다.

Resource Hacker 를 이용해서 언패킹된 파일을 오픈해서 확인해본 결과 RCDATA 에서 TFORM1 에서 다음과 같은 내용을 확인할 수 있었다.

TabVisible Ⓛ False로 되어 있는 것으로 보아 어떤 템이 숨겨져 있다는 것을 예상할 수 있다. 그리고 아래쪽으로 Picture.Data에 16진수 값이 들어 있다. 아래로 계속 내리면 Picture.Data를 하나 더 만날 수 있다.

# Reverse Engineering

TabVisible 을 True 로 바꾼 후 Compile Script 를 하고 새 파일로 저장한다. 그리고 실행하면 다음과 같이 화면이 바뀌어 있는 것을 볼 수 있다.



그림이 보고 유추할 수 있는 것은 스테가노그라피이다. 육안으로 식별하기에는 두 개의 그림이 똑같다. 따라서 앞에서 보았던 Picture.Data에 있는 16진수 값을 두 개 모두 가져와 Winmerge를 통해 비교해서 서로 다른 곳이 있는지 찾아 보았다. (리눅스에서는 diff 명령을 사용하면 된다)

```
0101010101000000010100000000010101010001010100000101000001010101010101000101  
0000001000000010100010101000101000001010000000100010001010100010001010100  
000001000001010000010101000100000001000100010101000101000101010000000001010  
101000000010000010100000101010101010000010101010101000001010000000001010000000  
010101010000000100000101000001010001010101010001000101010100000001010101010  
101000001010101010100010001010101000100000001010100010001010101010100000101  
0101000101010101010001000100000100000001000101
```

## Reverse Engineering

얼핏 보기에는 2 진 데이터처럼 보이지만 리소스 해커에서 확인했을 때 Picture.Data 의 처음 부분에 알파벳이 있었으므로 16 진수 데이터라는 것을 알 수 있다. 이 내용을 Winhex 를 통해서 저장을 해보겠다.

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	01	01	01	01	01	00	00	00	01	01	00	00	00	00	01	01
00000010	01	01	00	01	01	01	00	00	01	01	00	00	01	01	01	01
00000020	01	01	00	01	01	00	00	00	01	00	00	00	01	01	00	01
00000030	01	01	00	01	01	00	00	01	01	00	00	00	01	00	01	00
00000040	01	01	01	00	01	00	01	01	01	01	00	00	00	01	00	00
00000050	01	01	00	00	01	01	01	00	01	00	00	00	01	00	01	00
00000060	01	01	01	00	01	01	00	01	01	01	00	00	00	00	01	01
00000070	01	01	00	00	00	01	00	00	01	01	00	00	01	01	01	01
00000080	01	01	00	00	01	01	01	01	01	01	00	01	01	00	00	00
00000090	01	01	00	00	00	00	01	01	01	01	00	00	00	01	00	00
000000A0	01	01	00	00	01	01	00	01	01	01	01	01	00	01	00	01
000000B0	01	01	01	00	00	00	01	01	01	01	01	01	01	00	00	01
000000C0	01	01	01	01	00	01	00	01	01	01	01	00	01	00	00	00
000000D0	01	01	01	00	01	00	01	01	01	01	01	01	01	00	00	01
000000E0	01	01	01	00	01	01	01	01	01	01	00	01	00	01	00	00
000000F0	01	00	00	00	01	00	01	01	01	00	01	01	01	01	01	01

두 개의 이미지 파일은 총 0xF7(247)바이트 만큼 차이가 있었다. 01 로 되어 있는 바이트는 1로, 00 으로 되어 있는 바이트는 0으로 고쳐서 다시 확인해보면 다음과 같다.

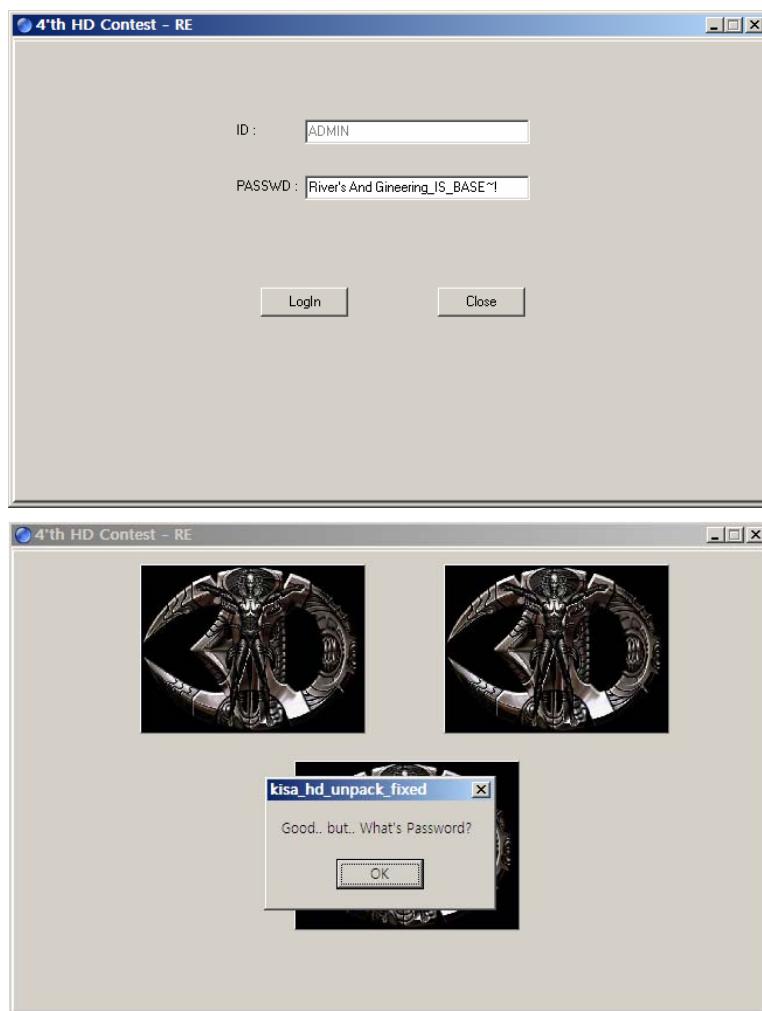
```
11111000 11000011
11011100 11001111
11011000 10001101
11011001 10001010
11101011 11000100
11001110 10001010
11101101 11000011
11000100 11001111
11001111 11011000
11000011 11000100
11001101 11110101
11100011 11111001
11110101 11101000
11101011 11111001
11101111 11010100
10001011
```

첫 번째 비트가 1이기 때문에 이것을 다시 10 진수 형태로 변환한 후 ASCII 테이블에서 해당되는 문자를 찾으면 이상한 문자만 보일 뿐이다. 그리고 만약 위에 보이는 16 진수가 패스워드라면 31 글자를 갖는 패스워드라는 것을 유추해볼 수 있다. 그래서 위 16 진수 데이터를 기반으로 간단한 프로그램을 작성하여 실행시켜 다음과 같은 결과를 얻을 수 있었다.

## Reverse Engineering

```
C:\선택 C:\WINDOWS\system32\cmd.exe
_d{h@*~^Lc i-Jdchh@dc jRD^ROL^Hs,
Pktgp%q"Clf "Eklggpkle lKQ lCQG!#
Qjufq$p#Bmg#Djmffq,jmdWJPWABPF>"!
River's And Gineering_IS_BASE~!
Shwds&r!@oe!Fhodshof^HR^C@RD@
To pct!u&Ghb&AohcctohaYOUYDGUCx'
Unqbu t'Fic'@nibbuni'XNTXEFTBy&
Umrau#w$Ej '$Cmjaavmjc [MW[FEWAz%
```

대부분 알 수 없는 문자들이었고 River's And Gineering\_IS\_BASE~! 가 패스워드라고 예상할 수 있다. 그럼 이 문자열을 입력하면 성공 메시지를 볼 수 있다.



## 7-2. 악성코드 분석 - shadowbot

### 악성코드 분석



- 통제된 환경에서 분석
  - 별도의 네트워크 구축
  - 가상머신 활용
- 분석 툴 준비
  - Filemon, Regmon, TDIMon, TCPView, Wireshark
  - Process Explorer
  - Winalysis
  - PEiD, LordPE
  - Ollydbg, Windbg, IDA
  - ImportREC, Revival
  - Strings, BinText

### Student Notes

악성코드를 분석하는 것은 쉽지 않은 일이다. 노력과 시간이 동시에 필요하며 인내력 또한 필요하다. 그리고 악성코드를 분석하는 과정에서 조그마한 실수로 인해서 악성코드가 네트워크를 통해서 유포되는 일이 발생할 수도 있다. 따라서 악성코드를 분석할 때는 항상 통제된 환경에서 분석을 해야 한다.

가장 좋은 통제된 환경은 네트워크와 연결되지 않은 머신에서 악성코드를 분석하는 것이고 좀 더 원활한 분석을 위해서는 VMware 나 VirtualPC 와 같은 가상 머신에서 악성코드를 분석하는 것이다.

본 교재에서는 VMware 를 이용하여 통제된 환경을 구성하고 shadowbot 이라는 악성코드를 분석해보도록 하겠다. shadowbot에 대한 정보는 다음 URL에서 확인하기 바란다.

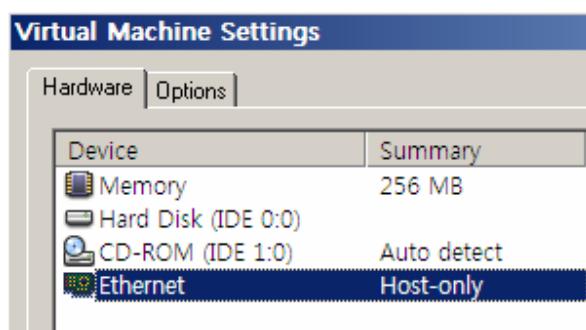
- [http://kr.ahnlab.com/info/smart2u/virus\\_detail\\_7404.html](http://kr.ahnlab.com/info/smart2u/virus_detail_7404.html)
- [http://kr.ahnlab.com/info/smart2u/virus\\_detail\\_7405.html](http://kr.ahnlab.com/info/smart2u/virus_detail_7405.html)

## Reverse Engineering

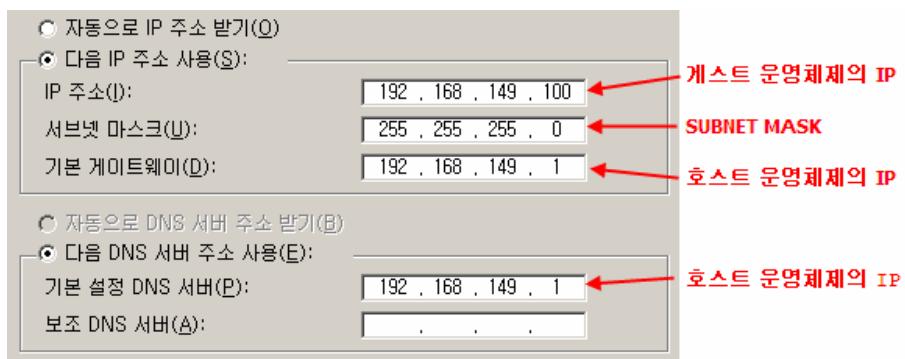
### 통제된 환경 구성

먼저 shadowbot 을 분석하기 위한 통제된 환경을 구성하는 방법에 대해서 알아보도록 하겠다. 본 교재에서는 VMware 를 이용할 것이며 다른 가상 머신이어도 상관없다. 단, 악성코드가 실행될 머신은 외부와 연결이 되지 않아야 한다. 여기서는 분석자가 사용하는 운영체제를 호스트 운영체제, VMware 에 설치된 운영체제를 게스트 운영체제라 부르겠다.

VMware 의 메뉴에서 VM → Settings 에서 Ethernet 을 Host-only 로 설정한다.



그리고 게스트 운영체제에서 네트워크 설정을 다음과 같이 설정한다.



VMware 에서 Host-only 모드는 게스트 운영체제가 호스트 운영체제하고만 통신이 가능한 모드이다. 게스트 운영체제에서 shadowbot 이 실행되고 어떤 패킷을 보내게 된다면 이것을 캡쳐하기 위해 게스트 운영체제의 게이트웨이와 DNS 를 호스트 운영체제로 설정하는 것이다.

## 모니터링 및 분석 툴 준비

shadowbot 실행 시 프로세스는 어떠한 것들이 생기는지, 파일을 어떤 것을 생성하고 접근하는지, 네트워크를 통해 어떤 패킷을 보내는지를 확인하기 위한 모니터링 툴이 필요하다.

### 호스트 운영체제 준비 툴

Wireshark

### 게스트 운영체제 준비 툴

Filemon : 실시간으로 현재 사용되는 파일들에 대한 정보를 볼 수 있다.

Regmon : 실시간으로 현재 레지스트리에 대한 정보를 볼 수 있다.

TDIMon : 실시간으로 현재 모든 TCP/UDP 입출력 상황을 보여준다.

TCPView : 현재 네트워크 연결 정보를 확인할 수 있다.

Process Explorer : 현재 실행중인 프로세스 정보와 해당 프로세스에 관련된 DLL이나 핸들링 정보 제공

Winalysis : 현재 시스템의 상황을 스냅샷을 뜯다.

BinText : 바이너리 파일에서 스트링을 추출해 준다.

Ollydbg, PEiD, ImportREC

모니터링 툴들은 동적 분석시 사용된다. 악성코드를 실행하고 어떤 행동들을 하는지 확인하기 위함이다.

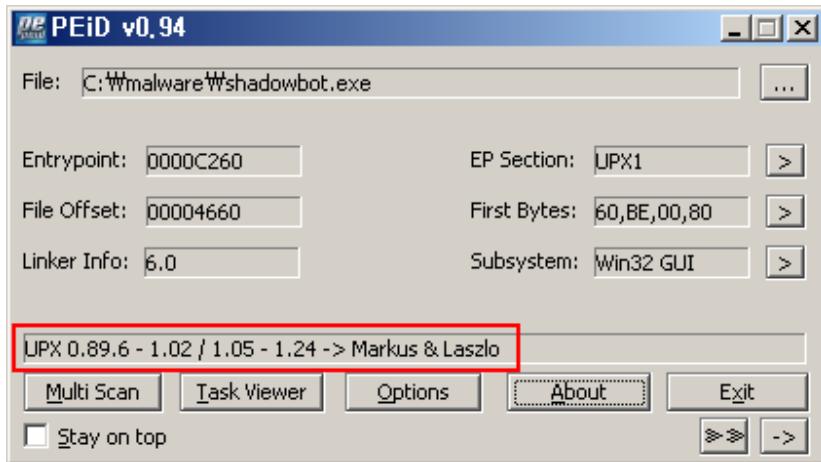
그리고 호스트 운영체제는 게스트 운영체제에서 혹시 외부로 패킷을 보내게 되면 이 패킷을 캡쳐하기 위해 Wireshark나 Ethereal 같은 패킷 캡쳐 프로그램을 실행할 것이다.

## 초기 분석

초기 분석 단계에서는 패킹 여부를 확인하고 만약 패킹이 되어 있다면 언패킹을 해야 한다. 물론 동적 분석시에는 언패킹을 하지 않아도 상관 없지만 정적 분석시에는 언패킹을 한 후에 디버거를 통해서 분석을 해야 한다.

패킹 여부는 PEiD를 통해서 확인할 수 있다. 또는 IDA를 이용하여 확인할 수 있는데 패킹된 파일을 IDA로 열면 경고창이 뜨고 코드의 사이즈가 작고 데이터 사이즈가 크다는 점을 통해서 패킹 여부를 확인할 수 있다. 여기서는 PEiD를 이용하여 패킹 여부를 확인해 보겠다.

## Reverse Engineering



PEiD로 shadowbot을 확인하니 UPX로 패킹되어 있는 것을 확인할 수 있다. 그럼 언패킹을 해보자.  
UPX로 패킹된 파일은 upx 프로그램을 이용하여 패킹 여부를 확인할 수도 있고 언패킹하는 것이 가능하다. 만약 UPX가 아닌 다른 패커로 패킹되었다면 그에 맞는 언패커로 패킹을 풀어야 할 것이고 또는 직접 매뉴얼 언패킹을 해도 상관 없다.

```
C:\#05.upx302w>upx -l c:\malware\shadowbot.exe
          Ultimate Packer for eXecutables
          Copyright (C) 1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007
          UPX 3.02w      Markus Oberhumer, Laszlo Molnar & John Reiser  Dec 16th 2007

          File size        Ratio        Format        Name
          -----        -----
          40960 ->    18944   46.25%    win32/pe    c:\malware\shadowbot.exe

C:\#05.upx302w>
C:\#05.upx302w>upx -d c:\malware\shadowbot.exe -o c:\malware\shadowbot_unpack.exe

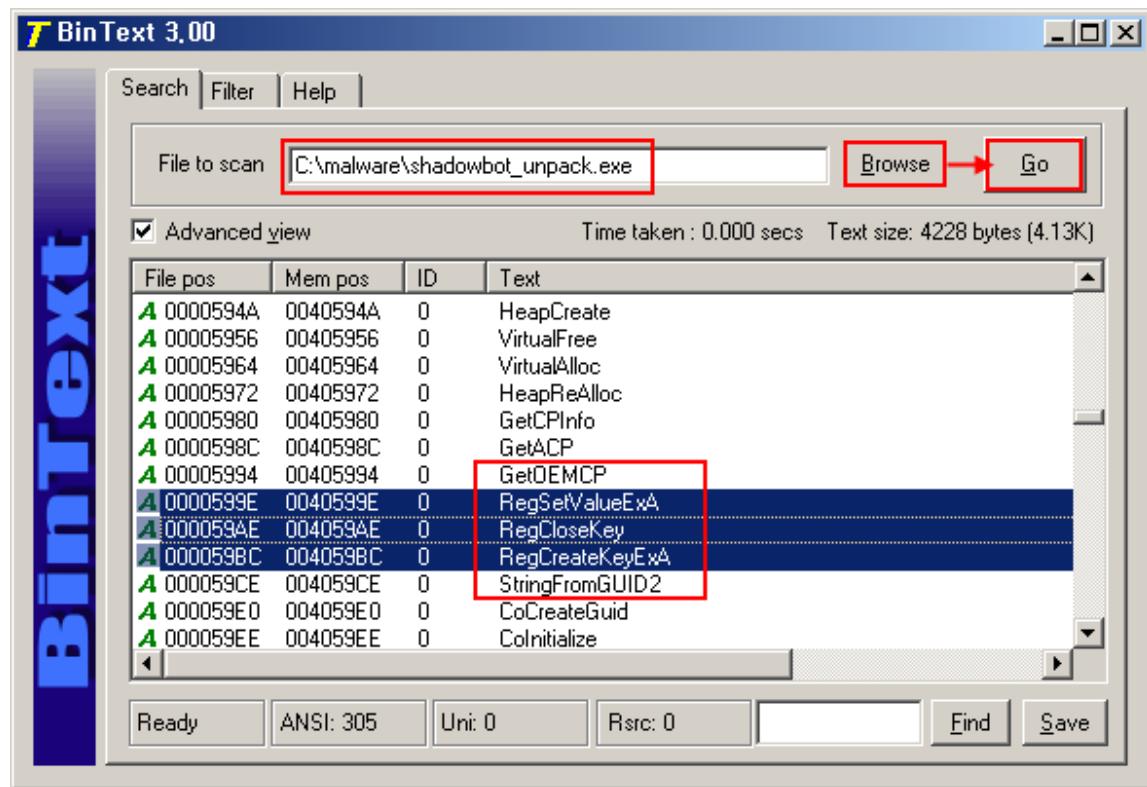
          Ultimate Packer for eXecutables
          Copyright (C) 1996,1997,1998,1999,2000,2001,2002,2003,2004,2005,2006,2007
          UPX 3.02w      Markus Oberhumer, Laszlo Molnar & John Reiser  Dec 16th 2007

          File size        Ratio        Format        Name
          -----        -----
          40960 <-    18944   46.25%    win32/pe    shadowbot_unpack.exe

Unpacked 1 file.
```

언패킹을 해서 shadowbot\_unpack.exe라는 파일로 저장을 하였다. 동적 분석을 마친 후 정적 분석시 이 언패킹된 파일을 가지고 분석을 하도록 할 것이다.

이번에는 BinText 를 이용하여 읽을 수 있는 스트링 추출한다. 여기서 주의할 것은 패킹된 파일을 BinText 에 읽어 와서 스트링을 추출하면 모든 스트링을 다 확인할 수 없다는 것이다. 따라서 언패킹했던 파일을 대상으로 스트링을 추출한다.



Browse 를 눌러 파일을 선택하고 Go 를 누르면 파일 내에 있는 읽을 수 있는 스트링을 보여준다. 위에 보이는 것처럼 함수들의 이름을 확인할 수 있는데 이렇게 임포트되는 함수들의 이름을 보고 이 파일이 실행이 되면 어떠한 행동들을 하는지 짐작할 수 있다. 오른쪽 하단에 Save 를 눌러서 텍스트 파일로 저장한 후 확인하도록 하자. 확인 결과 다음과 같은 함수들과 특이한 문자열들을 확인할 수 있었다.

<b>WriteProcessMemory</b>	// Load 할 DLL 이름
<b>CreateToolhelp32Snapshot</b>	// snapshot 을 찍는다.
<b>CreateRemoteThread</b>	// DLL Injection 4 번째 파라미터가 LoadLibrary 인가?
<b>RegSetValueExA</b>	// 레지스트리 등록
<b>RegCreateKeyExA</b>	// 아마도 채부팅 후에도 적용 받게 설정
<b>CoInitialize</b>	// com 오브젝트와 관련된 것들
<b>CloseClipboard</b>	// 클립보드 관련
<b>OpenClipboard</b>	// 열고 닫고 복사와 관련된 것들

## Reverse Engineering

<b>CloseServiceHandle</b>	// 서비스 닫기. 혹 바이러스 백신을 닫을 수도 있다.
<b>EnumServicesStatusA</b>	// 서비스 관련
<b>OpenSCManagerA</b>	// 서비스 관련 무슨 작업을 기대
<b>InternetReadFile</b>	// 인터넷에서 파일 다운 후 실행
<b>InternetOpenUrlA</b>	// http://~~, ftp://~~
<b>InternetOpenA</b>	
<b>CreateMutexA</b>	// 반복 실행 방지
<b>Explorer.exe</b>	// 인젝션을 할 정상적인 파일 일수도
<b>PRIVMSG %s :pstore %s %s:%s \n</b>	// private message 의 약자
<b>darkjester.xplosionirc.net</b>	// irc server 일수도 있다.
<b>lol lol lol :shadowbot</b>	// 이름의 유래가 된 스트링
<b>PING, PONG, JOIN, PRIVMSG</b>	// IRCBot 일 가능성이 높다.
<b>rdshost.dll</b>	// 인젝션 되는 dll 일까?
<b>\\\photo album.zip</b>	// 압축파일은 뭘까?

위 내용을 토대로 shadowbot에 대한 가정은 IRCBot으로 DLL Injection을 할 것이다. DLL Injection은 아마도 explorer.exe에 시도를 할 것이고 COM Object 관련 작업과 서비스 매니저를 이용해서 어떤 행동들을 할 것이다. 그리고 이 때 사용되는 파일은 rdshost.dll일 것이다. 아직까지는 이런 가설만 세울 수 있다. 좀 더 세부적인 분석을 통해서 이 가설을 확인할 것이다.

그리고 초기 분석의 마지막으로 실행 파일인 shadowbot.exe와 shadowbot\_unpack.exe에 대한 해쉬 값을 계산한다. 이것은 악성코드 중 실행이 되면 자기 자신을 변경하는 악성코드들도 있기 때문이다. 해쉬 값을 계산해 놓은 후 실행하고 후에 이 해쉬 값을 비교해 보면 변경이 되었는지 확인할 수 있다.

```
C:\> C:\Windows\system32\cmd.exe
C:\> malware>md5sum.exe shadowbot.exe > shadowbot.md5
                                                해쉬 데이터 생성
C:\> malware>md5sum.exe shadowbot_unpack.exe >> shadowbot.md5

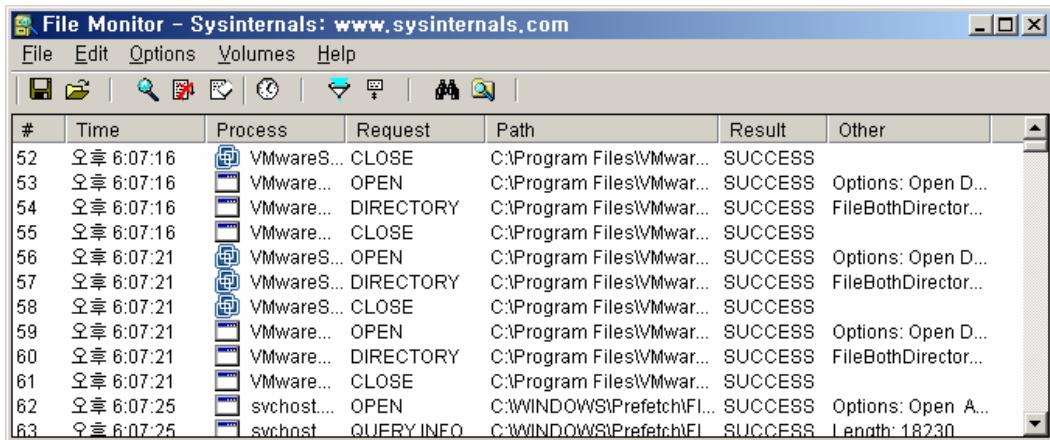
C:\> malware>md5sum.exe -c shadowbot.md5
shadowbot.exe: OK
shadowbot_unpack.exe: OK
                                                해쉬 데이터 비교
```

md5sum.exe 프로그램을 이용해서 해쉬 값을 계산해서 파일에 저장해 둔다. 비교할 때는 -c 옵션을 사용하면 된다. 자, 이제 초기 분석이 끝났고 동적 분석에 들어가보도록 하자.

## 동적 분석

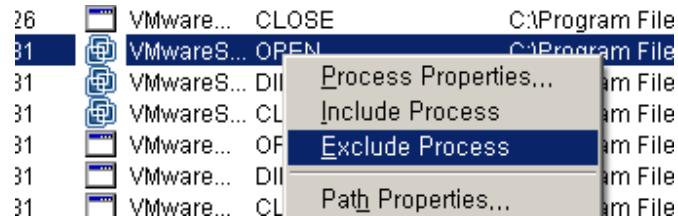
Filemon이나 Regmon은 실행 직후 갑자기 많은 정보들이 출력되는 것을 볼 수 있는데 원활한 분석을 위해서는 Exclude Process를 이용해서 필요 없는 프로세스를 지워주는 것이 좋다.

Filemon을 실행한다.

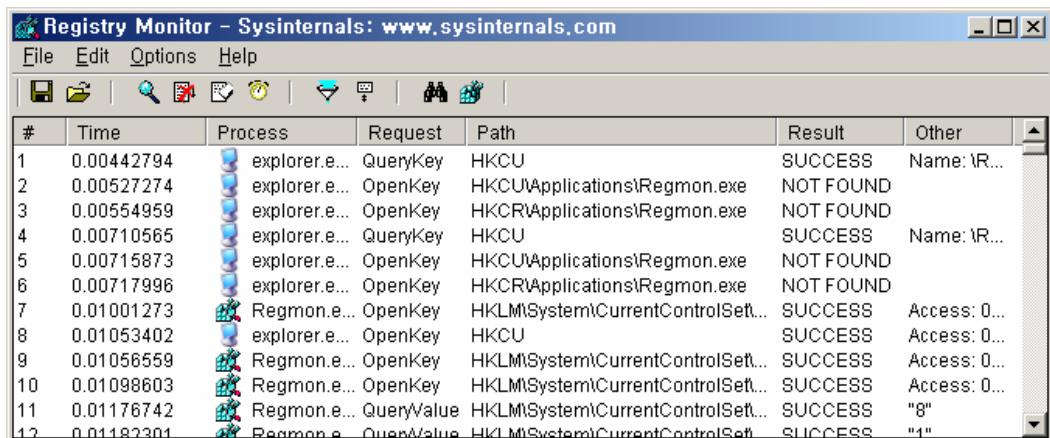


#	Time	Process	Request	Path	Result	Other
52	오후 6:07:16	VMwareS...	CLOSE	C:\Program Files\VMwar...	SUCCESS	
53	오후 6:07:16	VMware...	OPEN	C:\Program Files\VMwar...	SUCCESS	Options: Open D...
54	오후 6:07:16	VMware...	DIRECTORY	C:\Program Files\VMwar...	SUCCESS	FileBothDirector...
55	오후 6:07:16	VMware...	CLOSE	C:\Program Files\VMwar...	SUCCESS	
56	오후 6:07:21	VMwareS...	OPEN	C:\Program Files\VMwar...	SUCCESS	Options: Open D...
57	오후 6:07:21	VMwareS...	DIRECTORY	C:\Program Files\VMwar...	SUCCESS	FileBothDirector...
58	오후 6:07:21	VMwareS...	CLOSE	C:\Program Files\VMwar...	SUCCESS	
59	오후 6:07:21	VMware...	OPEN	C:\Program Files\VMwar...	SUCCESS	Options: Open D...
60	오후 6:07:21	VMware...	DIRECTORY	C:\Program Files\VMwar...	SUCCESS	FileBothDirector...
61	오후 6:07:21	VMware...	CLOSE	C:\Program Files\VMwar...	SUCCESS	
62	오후 6:07:25	svchost...	OPEN	C:\WINDOWS\Prefetch\Fl...	SUCCESS	Options: Open A...
63	오후 6:07:25	svchost	QUERY INFO	C:\WINDOWS\Prefetch\Fl	SUCCESS	Length: 18230

그리고 분석에 필요한 프로세스는 아직 없으니 전부 Exclude Process로 없앤다.



Regmon을 실행한다. 그리고 Filemon에서와 동일하게 모든 프로세스를 Exclude Process로 없앤다.



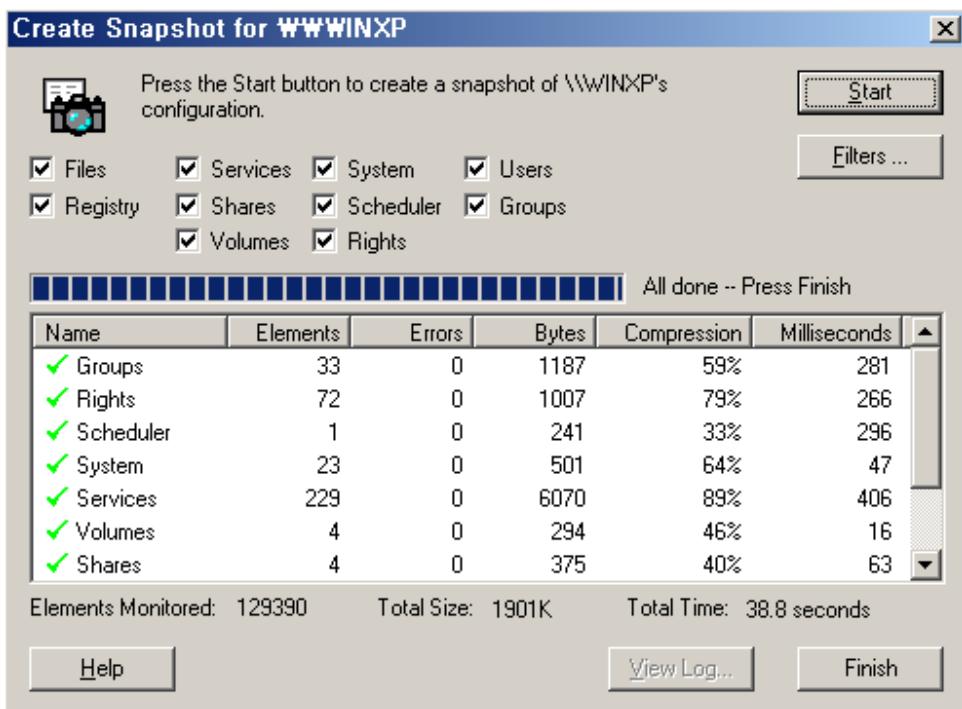
#	Time	Process	Request	Path	Result	Other
1	0.00442794	explorer.e...	QueryKey	HKCU	SUCCESS	Name: \R...
2	0.00527274	explorer.e...	OpenKey	HKCU\Applications\Regmon.exe	NOT FOUND	
3	0.00554959	explorer.e...	OpenKey	HKCR\Applications\Regmon.exe	NOT FOUND	
4	0.00710565	explorer.e...	QueryKey	HKCU	SUCCESS	Name: \R...
5	0.00715873	explorer.e...	OpenKey	HKCU\Applications\Regmon.exe	NOT FOUND	
6	0.00717996	explorer.e...	OpenKey	HKCR\Applications\Regmon.exe	NOT FOUND	
7	0.01001273	Regmon.e...	OpenKey	HKLM\System\CurrentControlSet...	SUCCESS	Access: 0...
8	0.01053402	Regmon.e...	OpenKey	HKCU	SUCCESS	Access: 0...
9	0.01056559	Regmon.e...	OpenKey	HKLM\System\CurrentControlSet...	SUCCESS	Access: 0...
10	0.01098603	Regmon.e...	OpenKey	HKLM\System\CurrentControlSet...	SUCCESS	Access: 0...
11	0.01176742	Regmon.e...	QueryValue	HKLM\System\CurrentControlSet...	SUCCESS	"8"
12	0.01182301	Regmon.e...	QueryValue	HKLM\System\CurrentControlSet...	SUCCESS	"1"

## Reverse Engineering

Winalysis 를 실행하고 Snapshot 을 누른다..

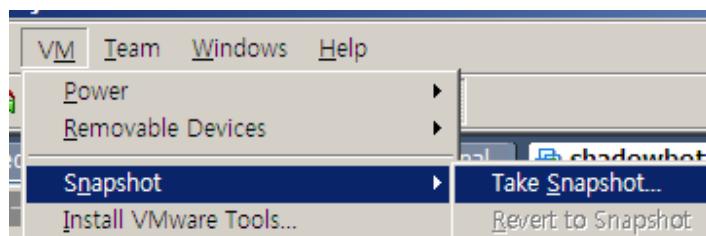


Start 버튼을 눌러 File 부터 시작해서 Service 나 System 등 여러 가지 정보에 대한 스냅샷을 찍는다.



여러 가지 정보들이 스냅샷이 찍혔다.

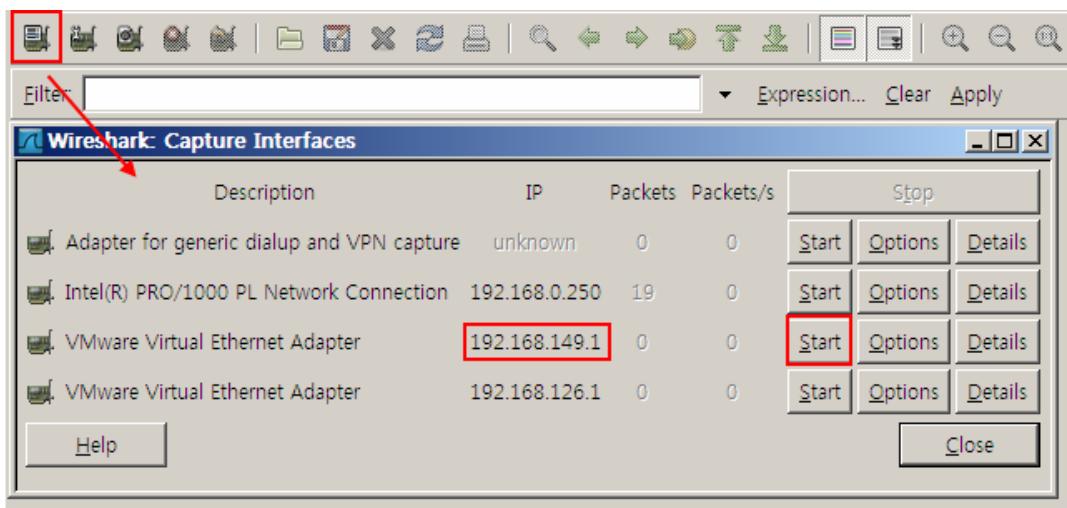
이제 동적 분석을 위한 준비가 끝났다. shadowbot 을 실행해서 어떤 행동들을 하는지 확인하면 되는데 shadowbot 을 실행하기 전에 VMware 에서 Snapshot 을 만들도록 하자. VMware 에서의 스냅샷은 일종의 백업의 의미로 후에 어떤 작업을 하다가 잘못될 경우 다시 이전 상태로 되돌릴 수 있다. 분석을 하는데 걸리는 시간을 단축시킬 수 있는 방법 중 하나이다.



## Reverse Engineering

적당한 이름과 설명을 적어 둔다. 이것은 사용자가 어느 상황에서 스냅샷을 만들었는지 확인하기 위함이다.

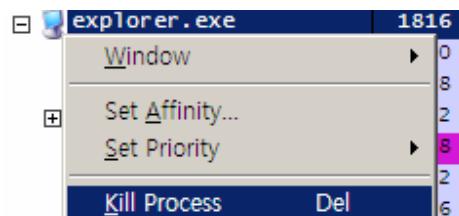
마지막으로 호스트 운영체제에서 패킷 캡쳐 프로그램을 실행시킨다. 이것은 악성코드(shadowbot)가 외부로 패킷을 보내는지 확인하기 위함이다. 여기서는 Wireshark를 사용하겠다.



캡쳐는 처음에 설정했던 게스트 운영체제의 게이트웨이를 사용하면 된다. 만약 shadowbot이 패킷을 외부로 보낸다면 분명 게이트웨이를 통해서 보낼 것이고 여러분은 그 패킷을 잡을 수 있을 것이다. 그럼 Start를 눌러 캡쳐를 시작한다.

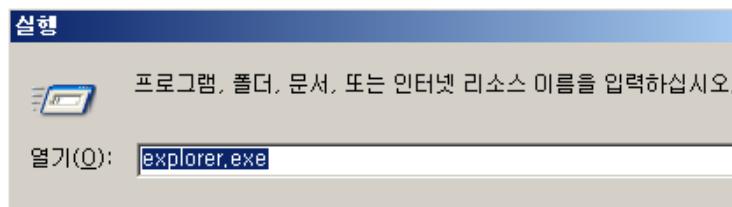
이제 shadowbot을 실행해보도록 하자. 예상치 못했던 일이 발생했다. 실행 중이던 모니터링 프로그램들이 전부 사라졌다. 이것은 shadowbot이 explore.exe의 자식 프로세스를 죽이는 것이다. 이렇게 되면 동적인 분석이 어려울 수 있지만 방법은 있다.

조금 전에 만들어 두었던 VMware 스냅샷으로 돌아가도록 하자. 모니터링 툴을 모두 실행한 상태에서 Process Explorer에서 explorer.exe 프로세스를 죽이고 새롭게 explorer.exe를 실행하면 모니터링 툴들이 explorer.exe의 자식 프로세스가 아니기 때문에 정상적으로 모니터링이 가능하다.



## Reverse Engineering

explorer.exe 프로세스를 죽인 후 메뉴에서 File → Run 을 통해 explore.exe 를 실행한다.



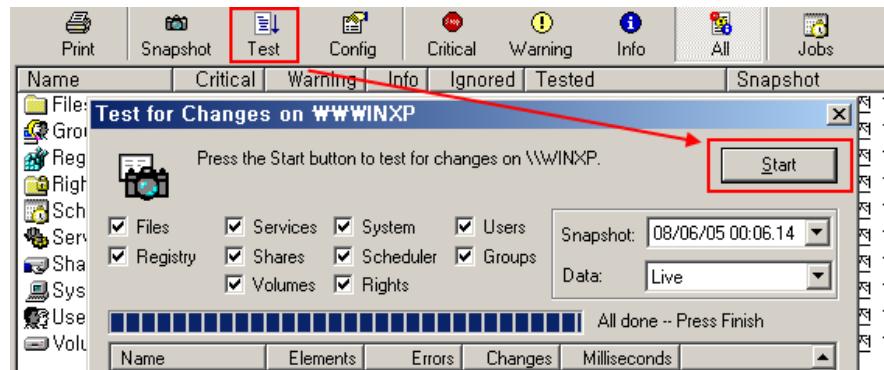
explorer.exe 를 실행하면 다음과 같이 procexp.exe 프로세스의 자식 프로세스로써 explorer.exe 가 실행된다. 이제 shadowbot 이 실행되더라도 모니터링 툴들이 죽는 일은 없을 것이다. 이 상태에서 VMware 스냅샷을 만들어 두는 것을 추천한다. (악성코드 분석 작업은 한치 앞을 내다볼 수 없기 때문에 항상 최악의 상황을 고려하는 것이 좋다)

Filemon.exe	1340	File system monitor	Sysinternals
Regmon.exe	1724	Sysinternals Registry M...	Sysinternals
Tcpview.exe	1468	TCP/UDP endpoint view...	Sysinternals
Winalysis.exe	372	Winalysis Application	Winalysis Software Inc.
procexp.exe	1728	Sysinternals Process E...	Sysinternals
explorer.exe	600	Windows Explorer	Microsoft Corporation

다시 shadowbot 을 실행시켜보겠다. 실행시키자 shadowbot.exe 가 프로세스 익스플로러에 보였다가 사라졌다.

Filemon.exe	1340	File system monitor	Sysinternals
Regmon.exe	1724	Sysinternals Registry M...	Sysinternals
Tcpview.exe	1468	TCP/UDP endpoint view...	Sysinternals
Winalysis.exe	372	Winalysis Application	Winalysis Software Inc.
procexp.exe	1728	Sysinternals Process E...	Sysinternals
explorer.exe	600	Windows Explorer	Microsoft Corporation
shadowbot.exe	1236	3,08	

먼저 Winalysis 의 내용을 확인해보도록 하겠다. 동적 분석 준비를 하면서 스냅샷을 떠 놓았고 shadowbot 을 실행한 후에는 Test 를 눌러 스냅샷과 현재를 비교해야 한다.



그리고 Start 를 누르면 스냅샷과 비교하여 변경된 부분이 있는지 확인한다. 확인 결과 다음에 보이는 것과 같이 File 부분과 Registry 부분이 변경된 것을 볼 수 있다.

Name	Critical	Warning	Info	Ignored
Files	0	0	2	0
Groups	0	0	0	0
Registry	0	1	6	3
Rights	0	0	0	0
Scheduler	0	0	0	0
Services	0	0	0	0
Shares	0	0	0	0
System	0	0	0	0
Users	0	0	0	0
Volumes	0	0	0	0

Filemon 과 Regmon 의 결과도 살펴 보자. 일단 새롭게 생성하거나 변경한 파일이나 레지스트리가 있는지 확인해보도록 하겠다.

Filemon 의 결과 중 파일을 생성하거나(CREATE) 파일에 쓰기(WRITE)하는 부분을 찾아보았다.

```

CREATE C:\WINDOWS\photo album.zip SUCCESS          Options: OverwriteIf Access: All
WRITE  C:\WINDOWS\photo album.zip SUCCESS          Offset: 0 Length: 30
WRITE  C:\WINDOWS\photo album.zip SUCCESS          Offset: 30 Length: 19
READ   C:\malware\shadowbot.exe      SUCCESS      Offset: 0 Length: 1024
WRITE  C:\WINDOWS\photo album.zip SUCCESS          Offset: 49 Length: 1024
READ   C:\malware\shadowbot.exe      SUCCESS      Offset: 1024 Length: 1024
WRITE  C:\WINDOWS\photo album.zip SUCCESS          Offset: 1073 Length: 1024
READ   C:\malware\shadowbot.exe      SUCCESS      Offset: 2048 Length: 1024
WRITE  C:\WINDOWS\photo album.zip SUCCESS          Offset: 2097 Length: 1024
READ   C:\malware\shadowbot.exe      SUCCESS      Offset: 3072 Length: 1024
WRITE  C:\WINDOWS\photo album.zip SUCCESS          Offset: 3121 Length: 1024
READ   C:\malware\shadowbot.exe      SUCCESS      Offset: 4096 Length: 1024
WRITE  C:\WINDOWS\photo album.zip SUCCESS          Offset: 4145 Length: 1024
READ   C:\malware\shadowbot.exe      SUCCESS      Offset: 5120 Length: 1024

```

## Reverse Engineering

```
WRITE C:\WINDOWS\photo album.zip SUCCESS           Offset: 5169 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 6144 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 6193 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 7168 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 7217 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 8192 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 8241 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 9216 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 9265 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 10240 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 10289 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 11264 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 11313 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 12288 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 12337 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 13312 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 13361 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 14336 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 14385 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 15360 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 15409 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 16384 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 16433 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 17408 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 17457 Length: 1024
READ  C:\malware\shadowbot.exe   SUCCESS          Offset: 18432 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 18481 Length: 512
READ  C:\malware\shadowbot.exe   END OF FILE      Offset: 18944 Length: 1024
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 18993 Length: 46
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 19039 Length: 19
WRITE C:\WINDOWS\photo album.zip SUCCESS          Offset: 19058 Length: 22
CLOSE C:\WINDOWS\photo album.zip SUCCESS
CLOSE C:\malware\shadowbot.exe   SUCCESS

CREATE C:\WINDOWS\system32\rdshost.dll  SUCCESS Options: OverwriteIf Access: All
OPEN  C:\WINDOWS\system32\SUCCESS        Options: Open Directory Access: 00000000
WRITE C:\WINDOWS\system32\rdshost.dll  SUCCESS     Offset: 0 Length: 14848
CLOSE C:\WINDOWS\system32\rdshost.dll  SUCCESS
QUERY INFORMATION C:\WINDOWS\system32\rdshost.dll  SUCCESS     Attributes: A
OPEN  C:\WINDOWS\system32\rdshost.dll  SUCCESS     Options: Open Access: Execute
QUERY INFORMATION C:\WINDOWS\system32\rdshost.dll  SUCCESS     Length: 14848
CLOSE C:\WINDOWS\system32\rdshost.dll  SUCCESS
QUERY INFORMATION C:\WINDOWS\system32\rdshost.dll  SUCCESS     Attributes: A
OPEN  C:\WINDOWS\system32\rdshost.dll  SUCCESS     Options: Open Access: Execute
CLOSE C:\WINDOWS\system32\rdshost.dll  SUCCESS
```

C:\WINDOWS에 “photo album.zip”이라는 파일을 생성한 후 shadowbot.exe 파일에서 조금씩 읽어와 “photo album.zip” 파일에 쓰는 것을 볼 수 있다. 그리고 C:\WINDOWS\system32에 rdshost.dll이라는 파일을 생성했다. 이 파일들은 초기 분석 당시 스트링을 추출했을 때 보였던 파일명들이다.

이번에는 Regmon 의 결과에서 Filemon 에서 찾아낸 rdshost.dll 과 photo album.zip 파일을 중심으로 찾아보니 다음과 같은 내용을 찾았다.

```
SetValue HKLM\Software\Microsoft\Windows\CurrentVersion\
ShellServiceObjectDelayLoad\rdshost SUCCESS
```

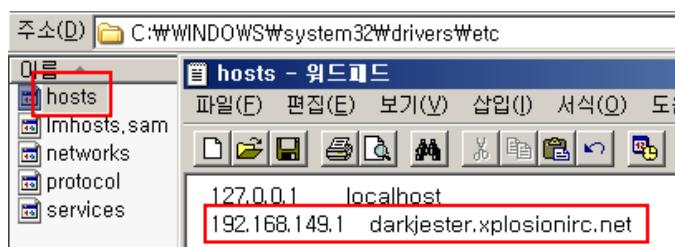
rdshost.dll 을 레지스트리에 등록하고 있다.

Filemon 과 Regmon 의 결과를 통해 shadowbot.exe 가 C:\WINDOWS\photo album.zip 과 C:\WINDOWS\system32\rdshost.dll 을 생성한다는 것을 알 수 있었다.

그리고 호스트 운영체제의 Wireshark 에서 패킷이 캡쳐되었다.

Source	Destination	Protocol	Info
Vmware_c8:3b:b7	Broadcast	ARP	who has 192.168.149.1? Tell 192.168.149.100
Vmware_c0:00:01	Vmware_c8:3b:b7	ARP	192.168.149.1 is at 00:50:56:c0:00:01
192.168.149.100	192.168.149.1	DNS	Standard query A darkjester.xplosionirc.net
192.168.149.1	192.168.149.100	ICMP	Destination unreachable (port unreachable)

darkjester.xplosionirc.net 도메인에 대한 질의를 하고 있는 것을 알 수 있다. DNS 쿼리를 보낸다는 것은 해당 서버에 접속하기 위해 IP 를 받아 오기 위함이다. 따라서 이 URL 로 접속을 하게 되면 좀 더 구체적인 행동들을 할 수도 있다. DNS 설치 대신 게스트 운영체제의 hosts 파일에 이 도메인을 추가해서 다시 실행시켜 보겠다.



게스트 운영체제의 hosts 파일을 수정하고 다시 호스트 운영체제에서 Wireshark 로 패킷을 캡쳐해보니 이번에는 8080 포트로 접속하려고 하는 것을 확인할 수 있다.

Source	Destination	Protocol	Info
192.168.149.100	192.168.149.1	TCP	1032 > 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
192.168.149.1	192.168.149.100	TCP	8080 > 1032 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
192.168.149.100	192.168.149.1	TCP	1032 > 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
192.168.149.1	192.168.149.100	TCP	8080 > 1032 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0
192.168.149.100	192.168.149.1	TCP	1032 > 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=1460
192.168.149.1	192.168.149.100	TCP	8080 > 1032 [RST, ACK] Seq=1 Ack=1 Win=0 Len=0

## Reverse Engineering

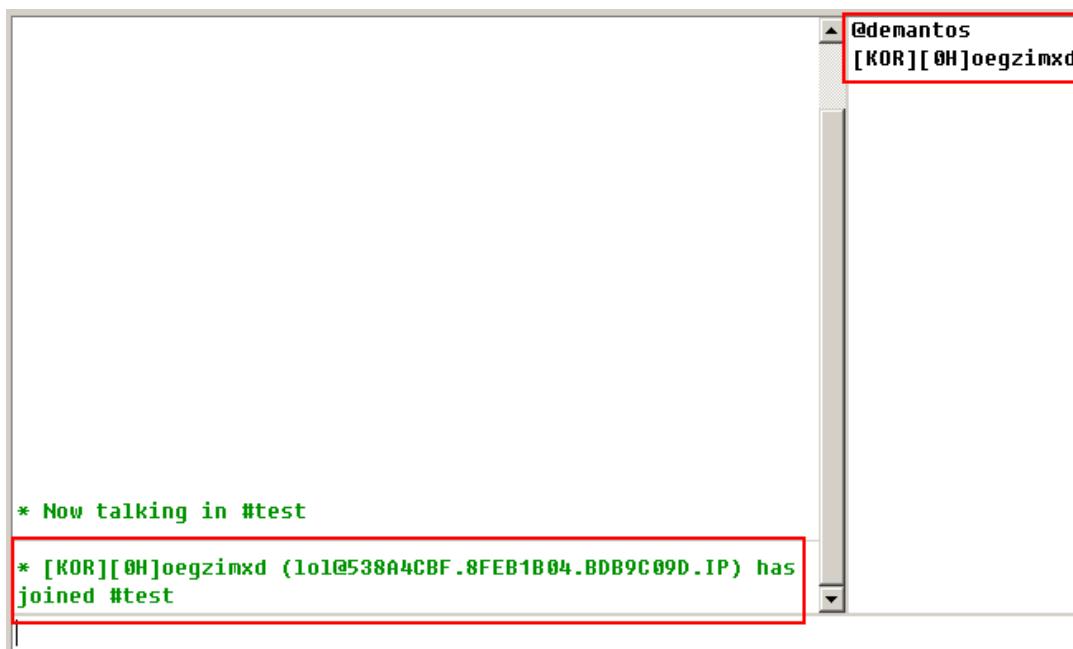
8080 포트로 접속을 시도하였지만 현재 호스트 운영체제는 8080 포트를 열고 있지 않기 때문에 계속 RST 되는 것을 볼 수 있다. 그렇다면 만약 호스트 운영체제가 8080 포트를 열고 연결을 기다리면 뭔가 또 다른 행동들을 볼 수 있을 것이다. 8080 포트를 열고 연결을 기다려야 하기 때문에 netcat 을 사용하도록 하겠다.

```
C:\>nc -l -v -p 8080
D:\>nc -l -v -p 8080
listening on [any] 8080 ...
192.168.149.100: inverse host lookup failed: h_errno 11004: NO_DATA
connect to [192.168.149.1] from <UNKNOWN> [192.168.149.100] 1032: NO_DATA

NICK [KOR][1H]kcqkqkwju
USER lol lol lol :shadowbot
JOIN #test
```

nc 를 통해 포트를 열고 기다렸더니 위와 같은 메시지를 만날 수 있었다. IRC 에서 사용하는 명령어가 보인다. 패킷 캡쳐를 통한 결론은 shadowbot 에 감염이 되면 darkjester.xplosionirc.net 이라는 IRC 서버에 test 라는 방에 접속하려고 한다는 것이다.

그렇다면 실제 IRC 서버를 구축해서 실제로 접속을 하는지 확인해보자. IRC 서버는 UnrealIRCD 를 이용하여 설치한다. 설치 후 unrealircd.conf 라는 파일을 unrealIRCD 가 설치된 디렉토리에 복사한다. 그리고 분석자도 IRC 에 접속해보자.

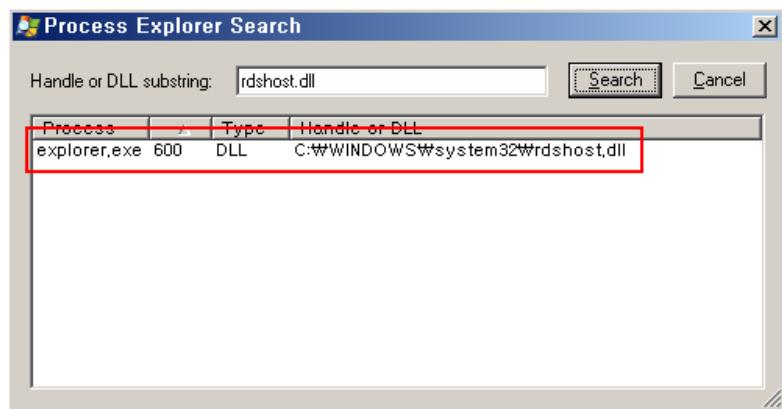


## Reverse Engineering

shadowbot에 감염된 에이전트가 IRC 서버에 접속되어 있는 것이 보인다. shadowbot에 감염된 호스트들은 모두 이 IRC 서버에 접속을 하게 될 것이고 공격자는 특정 명령을 실행하여 감염된 좀비 컴퓨터들을 제어하는 것이 가능할 것이다.

앞서 Filemon과 Regmon을 통해 확인했던 rdshost.dll는 스스로 동작하지 않고 어떤 프로그램과 함께 실행되는 파일이다. 초기 분석에서 스트링 추출 후 내렸던 가설이 rdshost.dll이라는 파일이 explorer.exe에 DLL Injection을 시도한다는 것이었다. 이 가설이 점점 명확해지고 있다.

dll 파일들이 어떤 프로세스와 함께 사용되고 있는지 Process Explorer에서 검색하면 찾을 수 있다. 메뉴에서 Find → Find Handle or DLL을 클릭한다. 그리고 rdshost.dll을 검색하면 다음과 같다.



rdshost.dll은 explorer.exe 프로세스와 함께 동작하고 있다. 원래 explorer.exe는 rdshost.dll을 임포트해서 실행하지 않는다. 그런데 현재 explorer.exe가 rdshost.dll을 임포트해서 사용하고 있다. 즉, shadowbot이 rdshost.dll 파일을 생성하고 explorer.exe에 DLL Injection을 하고 있다는 의미이다.

DLL Injection은 몇 가지 특정 함수들을 사용한다.

- **CreateRemoteThread**
- **WriteProcessMemory**
- **VirtualAllocEx**
- **OpenProcess**
- **LoadLibraryA**

이런 함수들이 사용이 되면 항상 그런 것은 아니지만 DLL Injection의 가능성�이 있다는 것을 의미한다. 따라서 정적 분석을 할 때 위 함수들을 주시하면서 분석을 해야 할 것이다.  
그럼 이제 정적 분석에 들어가도록 하겠다.

## Reverse Engineering

### 정적 분석

초기 분석과 동적 분석의 결과를 토대로 shadowbot.exe 과 rdhost.dll 파일을 디버거를 통해 분석해보도록 하자. 아마도 분석을 해보면 IRCBot 의 기능들을 알 수 있을 것이다.

먼저 shadowbot.exe 부터 분석해보자. shadowbot.exe 를 분석할 때 패킹되어 있던 원본이 아닌 언패킹한 파일을 가지고 하는 것이 좋다. 앞에서 확인했던 DLL Injection 에 사용되는 함수들과 파일을 생성하는 CreateFileA 함수 정도에 브레이크포인트를 설정한다.

R Found intermodular calls			
Address	Disassembly	Destination	
00401415	CALL DWORD PTR DS:[<&ole32.CoCreateGuid>]	ole32.CoCreateGuid	
0040140B	CALL DWORD PTR DS:[<&ole32.CoInitialize>]	ole32.CoInitialize	
00401687	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA>]	kernel32.CreateFileA	
00401973	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA>]	kernel32.CreateFileA	
004019B1	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA>]	kernel32.CreateFileA	
004010F3	CALL DWORD PTR DS:[<&KERNEL32.CreateRemoteThread>]	kernel32.CreateRemoteThread	

B Breakpoints			
Address	Module	Active	Disassembly
00401051	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.OpenProcess>]
0040108E	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.VirtualAllocEx>]
004010B1	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.WriteProcessMemory>]
004010F3	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.CreateRemoteThread>]
00401687	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA>]
00401973	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA>]
004019B1	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA>]
00404097	shadowbo	Always	CALL DWORD PTR DS:[<&KERNEL32.LoadLibraryA>]

F9 를 눌러 실행해 보자. BP 에 처음으로 멈춘 부분은 shadowbot.exe 파일을 오픈하는 CreateFileA 함수이다.

Address	Hex dump	Disassembly	Comment
0040195D	. 6A 00	PUSH 0	
0040195F	. 68 8000	PUSH 80	
00401964	. 6A 03	PUSH 3	
00401966	. 6A 00	PUSH 0	
00401968	. 6A 03	PUSH 3	
0040196A	. 68 0000	PUSH 0	
0040196F	. 8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]	
00401972	. 50	PUSH EAX	
00401973	. FF15 44	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA>]	CreateFileA

Address	Value	Comment
0012F774	0012FD28	FileName = "C:\malware\shadowbot_unpack.e
0012F778	80000000	Access = GENERIC_READ
0012F77C	00000003	ShareMode = FILE_SHARE_READ FILE_SHARE_WRI
0012F780	00000000	pSecurity = NULL
0012F784	00000003	Mode = OPEN_EXISTING
0012F788	00000080	Attributes = NORMAL
0012F78C	00000000	hTemplateFile = NULL

## Reverse Engineering

그 다음에 멈추는 BP는 C:\WINDOWS에 photo album.zip이라는 파일을 생성하는 CreateFileA 함수이다.

Address	Hex dump	Disassembly	Comment
0040199B	> 6A 00	PUSH 0	
0040199D	. 68 8000	PUSH 80	
004019A2	. 6A 02	PUSH 2	
004019A4	. 6A 00	PUSH 0	
004019A6	. 6A 03	PUSH 3	
004019A8	. 68 0000	PUSH 40000000	
004019AD	. 8B4D OC	MOV ECX, DWORD PTR SS:[EBP+C]	
004019B0	. 51	PUSH ECX	
004019B1	. FF15 44	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA]	CreateFileA

Address	Value	Comment
0012F774	0012FE2C	FileName = "C:\WINDOWS\photo album.zip"
0012F778	40000000	Access = GENERIC_WRITE
0012F77C	00000003	ShareMode = FILE_SHARE_READ   FILE_SHARE_WRITE
0012F780	00000000	pSecurity = NULL
0012F784	00000002	Mode = CREATE_ALWAYS
0012F788	00000080	Attributes = NORMAL
0012F78C	00000000	hTemplateFile = NULL

그리고 BP를 설정하지는 않았지만 photo album.zip 파일을 CreateFileA 함수를 통해서 파일을 만든 후 내용을 쓰는(WriteFile) 부분이 있는데 다음과 같다.

Address	Hex dump	Disassembly	Comment
00401BC6	. 50	PUSH EAX	
00401BC7	. FF15 54	CALL DWORD PTR DS:[<&KERNEL32.SetFilePointer]	
00401BCD	> C785 D8	MOV DWORD PTR SS:[EBP-428], 0	
00401BD7	. 6A 00	PUSH 0	
00401BD9	. 8D8D D8	LEA ECX, DWORD PTR SS:[EBP-428]	
00401BDF	. 51	PUSH ECX	
00401BE0	. 68 000400	PUSH 400	
00401BE5	. 8D95 00	LEA EDX, DWORD PTR SS:[EBP-400]	
00401BEB	. 52	PUSH EDX	
00401BEC	. 8B85 B8	MOV EAX, DWORD PTR SS:[EBP-448]	
00401BF2	. 50	PUSH EAX	
00401BF3	. FF15 50	CALL DWORD PTR DS:[<&KERNEL32.ReadFile]	
00401BF9	. 85C0	TEST EAX, EAX	
00401BFB	.~ 75 02	JNZ SHORT shadowbo.00401BFF	
00401BFD	.~ EB 43	JMP SHORT shadowbo.00401C42	
00401BFF	> 83BD D8	CMP DWORD PTR SS:[EBP-428], 0	
00401C06	.~ 75 02	JNZ SHORT shadowbo.00401C0A	
00401C08	.~ EB 38	JMP SHORT shadowbo.00401C42	
00401C0A	> 6A 00	PUSH 0	
00401C0C	. 8D8D D4	LEA ECX, DWORD PTR SS:[EBP-42C]	
00401C12	. 51	PUSH ECX	
00401C13	. 8B95 D8	MOV EDX, DWORD PTR SS:[EBP-428]	
00401C19	. 52	PUSH EDX	
00401C1A	. 8D85 00	LEA EAX, DWORD PTR SS:[EBP-400]	
00401C20	. 50	PUSH EAX	
00401C21	. 8B8D 84	MOV ECX, DWORD PTR SS:[EBP-47C]	
00401C27	. 51	PUSH ECX	
00401C28	. FF15 40	CALL DWORD PTR DS:[<&KERNEL32.WriteFile]	
00401C2E	. 8B95 DC	MOV EDX, DWORD PTR SS:[EBP-424]	
00401C34	. 0395 D8	ADD EDX, DWORD PTR SS:[EBP-428]	
00401C3A	. 8995 DC	MOV DWORD PTR SS:[EBP-424], EDX	
00401C40	.^ EB 8B	JMP SHORT shadowbo.00401BCD	

## Reverse Engineering

그리고 rdhost.dll 파일을 생성하는 부분이다.

Address	Hex dump	Disassembly	Comment
0040166E	. 6A 00	PUSH 0	
00401670	. 68 8000	PUSH 80	
00401675	. 6A 02	PUSH 2	
00401677	. 6A 00	PUSH 0	
00401679	. 6A 02	PUSH 2	
0040167B	. 68 0000	PUSH 40000000	
00401680	. 8D95 EC	LEA EDX, DWORD PTR SS:[EBP-314]	
00401686	. 52	PUSH EDX	
00401687	. FF15 44	CALL DWORD PTR DS:[<&KERNEL32.CreateFileA]	CreateFileA

Address	Value	Comment
0012FC04	0012FC20	FileName = "C:\WINDOWS\system32\rdhost.dll"
0012FC08	40000000	Access = GENERIC_WRITE
0012FC0C	00000002	ShareMode = FILE_SHARE_WRITE
0012FC10	00000000	pSecurity = NULL
0012FC14	00000002	Mode = CREATE_ALWAYS
0012FC18	00000080	Attributes = NORMAL
0012FC1C	00000000	hTemplateFile = NULL

그리고 바로 아래쪽에 rdhost.dll 파일에 내용을 쓰는 부분이 있다.

Address	Hex dump	Disassembly	Comment
0040169C	. 6A 00	PUSH 0	
0040169E	. 8D45 FC	LEA EAX, DWORD PTR SS:[EBP-4]	
004016A1	. 50	PUSH EAX	
004016A2	. 68 003A00	PUSH 3A00	
004016A7	. 68 3060	PUSH shadowbo.00406030	
004016AC	. 8B8D F0	MOV ECX, DWORD PTR SS:[EBP-210]	
004016B2	. 51	PUSH ECX	
004016B3	. FF15 40	CALL DWORD PTR DS:[<&KERNEL32.WriteFile]	WriteFile

Address	Value	Comment
0012FC0C	00000064	hFile = 00000064 (window)
0012FC10	00406030	Buffer = shadowbo.00406030
0012FC14	00003A00	nBytesToWrite = 3A00 (14848.)
0012FC18	0012FF30	pBytesWritten = 0012FF30
0012FC1C	00000000	pOverlapped = NULL

계속 F9로 진행하면 OpenProcess 함수에서 멈춘다. OpenProcess를 통해서 앞에서 생성한 rdhost.dll 파일을 DLL Injection하기 위한 타겟을 확인한다. (3 번째 인자)

Address	Hex dump	Disassembly	Comment
00401049	. 50	PUSH EAX	
0040104A	. 6A 00	PUSH 0	
0040104C	. 68 2A0400	PUSH 42A	
00401051	. FF15 24	CALL DWORD PTR DS:[<&KERNEL32.OpenProcess]	OpenProcess

Address	Value	Comment
0012FA28	0000042A	Access = CREATE_THREAD VM_OPERATION VM_WRI
0012FA2C	00000000	Inheritable = FALSE
0012FA30	00000258	ProcessId = 258

## Reverse Engineering

타겟의 ProcessId 가 0x258 이다. 10 진수로 변환하면 600 이다. 확인해보니 explorer.exe 였다.

이미지 이름	PID	사용자 이름	C...	메모리 ...
explorer.exe	600	demantos	00	16,956 KB
lsass.exe	720	SYSTEM	00	1,320 KB

계속 진행을 하면 가상 메모리 공간을 할당하고 WriteProcessMemory 를 통하여 DLL Injection 에 이용할 DLL 파일을 확인한다.

Address	Hex dump	Disassembly	Comment
0040108E	. FF15 1C	CALL DWORD PTR DS:[<&KERNEL32.VirtualAlloc>]	
00401094	. 8945 D8	MOV DWORD PTR SS:[EBP-28], EAX	
00401097	. 837D D8	CMP DWORD PTR SS:[EBP-28], 0	
0040109B	.~ 75 02	JNZ SHORT shadowbo.0040109F	
0040109D	.~ EB 78	JMP SHORT shadowbo.00401117	
0040109F	> 6A 00	PUSH 0	pBytesWritten = NULL
004010A1	. 8B55 D0	MOV EDX, DWORD PTR SS:[EBP-30]	BytesToWrite
004010A4	. 52	PUSH EDX	Buffer
004010A5	. 8B45 DC	MOV EAX, DWORD PTR SS:[EBP+C]	Address
004010A8	. 50	PUSH EAX	hProcess
004010A9	. 8B4D D8	MOV ECX, DWORD PTR SS:[EBP-28]	WriteProcessMemory
004010AC	. 51	PUSH ECX	
004010AD	. 8B55 E0	MOV EDX, DWORD PTR SS:[EBP-20]	
004010B0	. 52	PUSH EDX	
004010B1	. FF15 18	CALL DWORD PTR DS:[<&KERNEL32.WriteProcessMemory>]	

Address	Value	Comment
0012FA20	00000060	hProcess = 00000060
0012FA24	00DE0000	Address = DE0000
0012FA28	0012FA84	Buffer = 0012FA84
0012FA2C	00000040	BytesToWrite = 40 (64.)
0012FA30	00000000	pBytesWritten = NULL

세 번째 인자가 인젝션 되는 DLL 파일의 이름을 가지고 있는 주소이다. 0012FA84 를 확인해 보면 다음과 같다.

Address	Hex dump	ASCII
0012FA84	43 00 3A 00 5C 00 57 00 49 00 4E 00 44 00 4F 00	C.:.\W.I.N.D.O.
0012FA94	57 00 53 00 5C 00 73 00 79 00 73 00 74 00 65 00	W.S.\s.y.s.t.e.
0012FAA4	6D 00 33 00 32 00 5C 00 72 00 64 00 73 00 68 00	m.3.2.\r.d.s.h.
0012FAB4	6F 00 73 00 74 00 2E 00 64 00 6C 00 6C 00 00 00	o.s.t...d.l.l...
0012FAC4	84 FA 12 00 84 FA 12 00 14 FC 12 00 3D 13 40 00	¶J.¶J.¶J.=!!@.
0012FAD4	58 02 00 00 20 FC 12 00 38 07 94 7C 64 00 00 00	X1.. ?.8•?d...
0012FAE4	28 01 00 00 00 00 00 00 58 02 00 00 00 00 00 00	(r.....X1.....
0012FAF4	00 00 00 00 08 00 00 00 C0 06 00 00 08 00 00 00	....d....?..□...
0012FB04	00 00 00 00 65 78 70 6C 6F 72 65 72 2E 65 78 65	....explorer.exe

Command : db 0012FA84

인젝션되는 파일은 앞에서 생성되었던 C:\WINDOWS\system32\rdshost.dll 이라는 것을 확인할 수 있다. 또는 레지스터 창에서 EAX 레지스터를 확인하면 같은 내용을 확인할 수 있다. 역시나 우리가 앞에서 예상했던 일들이 하나씩 일어나고 있다.

## Reverse Engineering

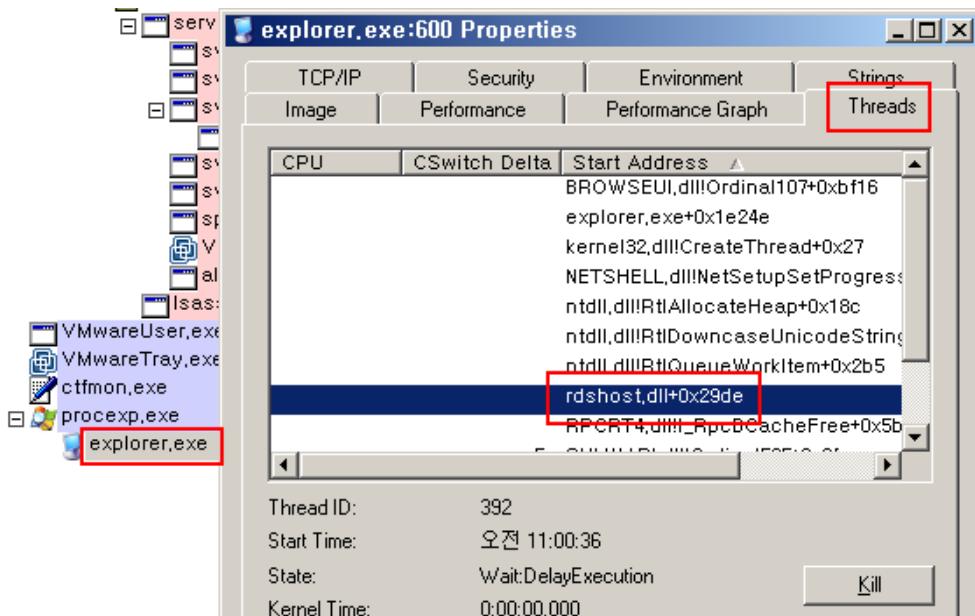
F9를 눌러 계속 진행하면 CreateRemoteThread를 통해서 DLL Injection 공격이 이루어지는 것을 볼 수 있다. 이 함수가 호출이 된 후에 rdhost.dll 파일이 explorer.exe에 인젝션이 된다.

Registers (FPU)

Address	Value	Comment
0012FA18	00000060	
0012FA1C	00000000	
0012FA20	00000000	
0012FA24	7C80ACD3	kernel32.LoadLibraryW
0012FA28	012F0000	
0012FA2C	00000000	

그리고 계속 실행을 시키면 디버거가 죽는다. 앞에서도 봤던 현상인데 explorer.exe의 자식 프로세스로 써 디버거를 동작시켜서 그렇다.(디버거를 explorer.exe의 자식 프로세스로 동작시키지 않았다면 디버거는 아직 그대로 살아 있을 것이다!)

그럼 explorer.exe에 rdhost.dll 파일이 삽입되었는지 확인해보도록 하겠다. Process Explorer에서 explorer.exe를 더블클릭하고 Threads에서 다음과 같은 정보를 확인할 수 있다.

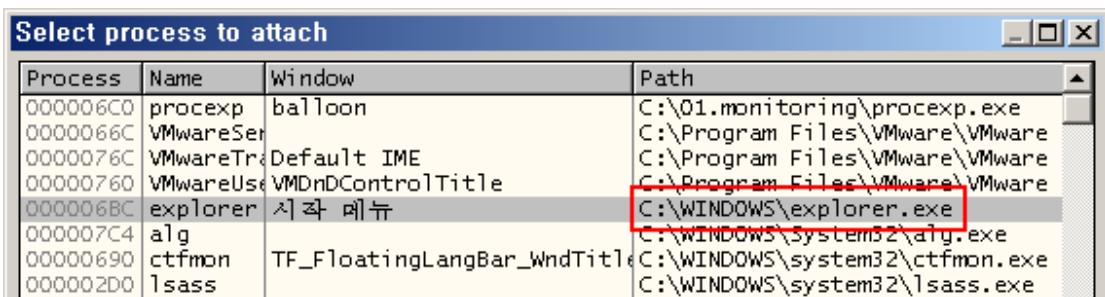


이렇게 rdhost.dll이 DLL Injection을 통해서 explorer.exe의 쓰래드에 삽입되는 것을 확인하였고 이번에는 rdhost.dll의 실질적인 행동인 IRCBot으로써의 기능을 분석해보도록 하겠다.

## IRCBot 기능 분석

이제 정적 분석 중 shadowbot.exe에 대한 분석을 마치고 실제 공격을 담당하는 rdshost.dll 파일을 분석해 보도록 하겠다. DLL의 경우 혼자서 독립적으로 실행되지 않는 파일이기 때문에 rdshost.dll이 인젝션되어 있는 explorer.exe 파일을 attach 시켜서 분석하도록 하겠다.

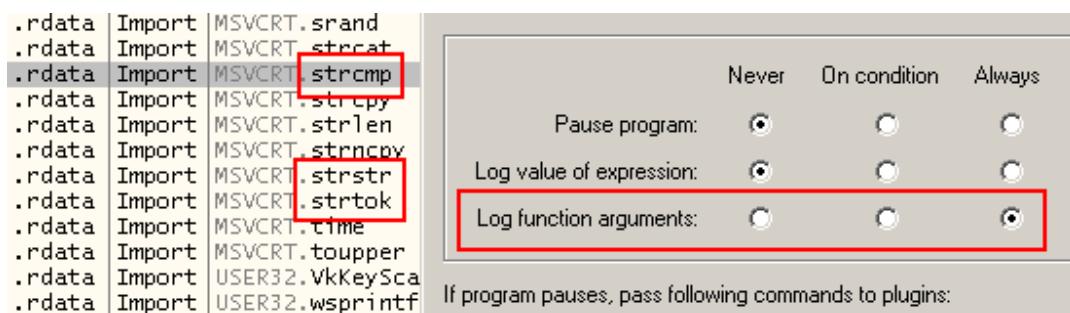
디버거 메뉴에서 File → Attach를 하면 어태치 할 파일을 선택하는데 explorer.exe를 선택한다.



어태치 후 CPU 창에는 별다른 내용이 나오지 않고 상태도 Pause로 되어 있다. 일단 시작을 시키고 메뉴에서 View → Executable modules을 선택한다. (단축키 ALT+E)

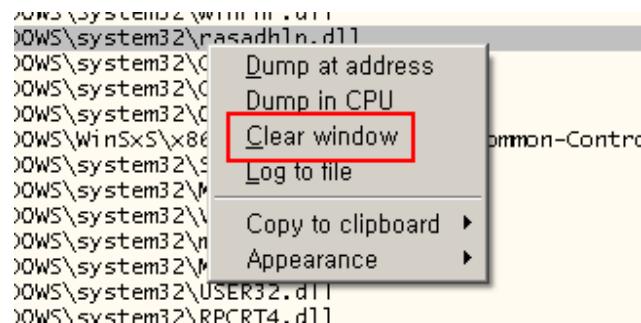
Executable modules					
Base	Size	Entry	Name	File version	Path
76BA0000	00008000	76BA10F1	PSAPI	5.1.2600.2180	C:\WINDOWS\system32\PSAPI.DLL
76F70000	00006000	76F7142B	rasadhlp	5.1.2600.2180	C:\WINDOWS\system32\rasadhlp.dll
100000000	00007000	10003031	rdshost		C:\WINDOWS\system32\rdshost.dll
77D80000	00091000	77D86284	RPCRT4	5.1.2600.2180	C:\WINDOWS\System32\RPCRT4.dll
76E30000	0000E000	76E3245F	rtutils	5.1.2600.2180	C:\WINDOWS\System32\rtutils.dll

rdshost.dll을 찾아서 마우스 오른쪽을 클릭한 후 View names를 해서(단축키 CTRL+N) rdshost 내에 존재하는 함수들 중 문자열과 관련이 있는 함수를 찾아서 conditional log breakpoint를 설정한다. 함수 이름에서 마우스 오른쪽을 클릭해서 Conditional log breakpoint on import를 선택한다.



## Reverse Engineering

상단 메뉴에서 View → Log 를 선택하거나 ALT+L 을 눌러 로그창을 연다. 현재 로그가 남아 있는 것을 볼 수 있을 것이다. IRCBot 에 의한 로그만을 확인하여 보다 정확한 분석을 위해 전부 클리어 한다.

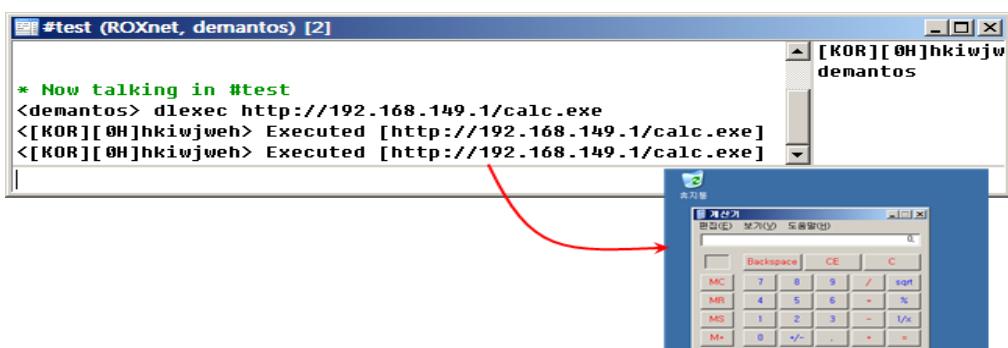


IRC 창에서 아무런 글자나 입력을 한다. 그러면 다음과 같은 로그를 확인할 수 있다.

Address	Message
77BF7C60	CALL to strstr from rdshost.0146282D s1 = ".hi" s2 = "imstart"
77BF7C60	CALL to strstr from rdshost.01462882 s1 = ".hi" s2 = "pstore"
77BF7C60	CALL to strstr from rdshost.014628E6 s1 = ".hi" s2 = "msnfuck"
77BF7C60	CALL to strstr from rdshost.01462905 s1 = ".hi" s2 = "dlexec"

이것은 IRC 를 통해서 메시지를 전달하면 IRCBot 이 가지고 있는 명령어와 비교를 한다는 것을 의미한다. 따라서 shadowbot 에서는 imstart, pstore, msnfuck, dlexec 와 같은 명령어들이 사용된다는 것을 알 수 있다.

dlexec 명령은 지정한 URL 에서 파일을 다운로드해서 실행시키는 명령이다.



## Reverse Engineering

그리고 imstart 는 shadowbot 을 전파하기 위한 명령인데 MSN 메신저를 통해서 photo album.zip 이라는 파일을 전파된다.

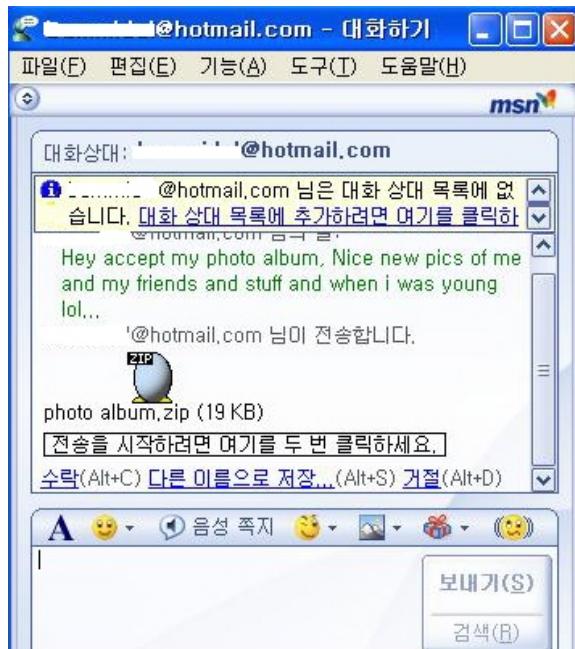


photo album.zip 파일은 photo album2007.pif 파일을 가지고 있는데 이 파일은 shadowbot.exe 파일과 동일한 크기를 가지고 있다.

shadowbot.exe 등록 정보		photo album2007 등록 정보	
일반	호환성	일반	프로그램
	shadowbot.exe		photo album2007
파일 형식:	응용 프로그램	파일 형식:	MS-DOS 프로그램으로 바로
설명:	shadowbot	설명:	photo album2007
위치:	C:\#malware	위치:	C:\#malware
크기:	18,5KB (18,944 바이트)	크기:	18,5KB (18,944 바이트)
디스크 할당 크기:	20,0KB (20,480 바이트)	디스크 할당 크기:	20,0KB (20,480 바이트)

디버거로 열러 보면 shadowbot.exe 와 동일한 파일이라는 것을 알 수 있다. 디버거를 통해 언패킹을 하고 IAT 를 복구했더니 shadowbot.exe 를 실행했을 때와 동일한 증상이 나타났다.

## **Reverse Engineering**

결론을 내려보면

- CreateRemoteThread 함수를 이용한 DLL Injection 기법이 적용된 IRC-BOT 이다.
- 모니터링 툴들을 Kill 시킨다.
- 특정 IRC 서버로 접속을 하기 위해 DNS Query 를 발생 시킨 후 응답이 없을 경우 작업이 정지된다.
- 삽입되는 파일은 %systemroot%system32\rdshost.dll 이다.
- MSN 메신저를 통하여 악성파일(photo album.zip)을 유포 시킨다.
- photo album.zip 은 photo album2007.pif 라는 파일을 가지고 있으며 이 파일은 shadowbot.exe 와 동일한 파일이다.

이렇게 악성코드를 분석할 경우에는 통제된 환경을 구성하여 분석해야 하며 신속하고 정확한 분석을 위해서 동적 분석 기법과 정적 분석 기법을 적절히 혼합하여 분석하는 것이 좋다.