**PE(Portable Executable)**

실행파일의 구조 : exe파일 + DLL 파일

플랫폼(CPU)에 상관없이 Wn32 운영체제가 돌아 가는 시스템이면 어디든지 실행 가능하다는 의미에서 붙여진 이름이다.

즉, Intel CPU기반의 윈도우 운영체제가 탑재된 시스템에서 돌아가는 PE 프로그램은 ALPHA CPU기반의 윈도우 운영체제가 탑재된 시스템에서도 실행이 가능하다는 것이다.

**실행 파일의 내용**

Windows 실행 파일은 실행에 필요한 다양한 정보를 PE(Portable Executable)라고 하는 포맷에 따라 저장한다. PE형식은 일부 Unix 시스템에 쓰이는 COFF(Common Object File Format)포맷에 Windows의 독자적인 사양을 추가한 것이다.

PE의 간단한 구조

* MS\_DOS 호환 헤더 /스텁(Stub)
* COFF 헤더, 옵션 헤더, 섹션 테이블, 각 섹션의 데이터

\*) 프로그램 시행에 필요한 정보나 프로그램의 구조에 관한 정보는 헤더에, 프로그램의 코드나 데이터는 섹션 데이터에 저장된다.

**더미 MS\_DOS 프로그램 뒤쪽에 Windows 프로그램이 있다.**

MS-DOS 호환 헤더/스텁 : MS-DOS용 프로그램과의 호환성을 유지하기 위해 남아 있슴(선

두 2byte : 0x4d, 0x5a => ASCII 문자의 ‘MZ’) 로 되어 있슴

본래 이 부분은 Windows 프로그램에서는 전혀 필요가 없으나, 파일 선두에 ‘MZ’부분으

로 실행 파일 여부를 판별하는 경우가 있으므로 삭제해서는 안됨

MS-DOS 호환 헤더에 저장된 오프셋값이 가리키는 위치에 있는 것이 PE 형식 본래의 헤더

이다. 이 부분에는 선두의 4byte에 PE형식임을 나타내는 시그네처(ASCII: 'PE\0\0’)가

있고, 그 뒤에 COFF 헤더와 옵션 헤더가 이어진다.

**PE 로더에 대해**

프로그램이 실행되면 가장먼저 'PE로더'가 그 파일의 PE포멧을 읽어들여서 그 형식에 맞게 메모리에 실행파일을 올리는 일을 수행한다.

그리고 임포트된 함수의 주소를 찾는 역할도 PE로더가 하게 되며 우리는 로더에 대해 많은 지식을 알 필요가 없다.

그것은 PE로더가 하는일은 정해져 있으므로 이를 바꿀 필요도 없으며 바꿀수도 없다고 생각한다. 우리가 얻어내거나 바꿔야 하는 정보는 바로 PE포멧의 구조이다.

PE로더에 대해선 단지 PE포멧을 읽어들여 실행시에 메모리로 로드하는 임무수행자 정도로만 정리하면 될 듯 하다.

ex) PE파일 dump 확인

1. PE파일의 시작 : “MZ” (0x4D, 0x5A)
2. 상당히 복잡해 보이는 암호 코드 같은 바이너리 덤프이지만 정해진 포맷을 가지고 있다.

이를 PE 구조라 한다.

1. **개론**

**1,1 PE 파일의 전체 구조**

[Portable Executable File format]

윈도우즈 운영체제에서 하나의 실행파일(.exe)이나 DLL은 단순한 기계어코드의 덩어리는 아니다.

Portable Executable File Format 이라고 알려진 형태의 파일이다.실행파일(PE File)은 크게 헤더

부분과 Section부분으로 나누어진다.(DLL도 마찬가지 이다.) 위 그림을 보자.

헤더는 다음과 같은 요소로 구성되어 있다.

|  |  |  |
| --- | --- | --- |
| IMAGE\_DOS\_HEADER  (40byte) MS DOS Stub | MS-DOS 시절 사용하던 헤더로 현재는 사용되지 않는다.  “MZ”로 시작 | |
| IMAGE\_NT\_HEADER | Signature | 항상 'PE'로 되어 있다. |
| IMAGE\_FILE\_HEADER  (20byte) |  |
| IMAGE\_OPTIONAL\_HEADER  (96byte) | 가장 중요한 헤더 이다.  Entry Point의 주소, Image의 가상주소(Image base), stack, heap의 크기등의 정보를 가지고 있다. |
| IMAGE\_SECTION\_HEADER | 섹션의 갯수만큼 존재한다. 각 Section의 정보를 담고 있다. | |

헤더 다음은 각 section이라는 요소로 구분되어 있다.

우리가 만든 c 함수들의 기계어 코드는 .text라는 section에 놓인다. 또한 메뉴나 다이얼로그,

Icon등은 .rsrc Section에 놓인다. 또한 exe가 사용하는 각종 DLL에 대한 정보는 .idata section에

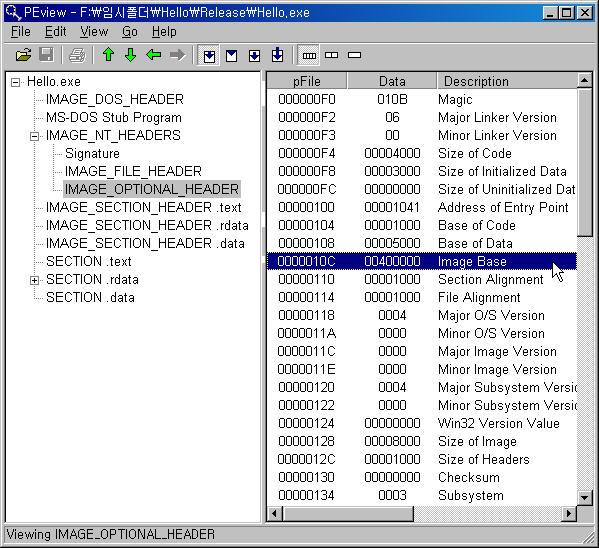
놓인다.

대부분의 표준 섹션들은 '.' 으로 시작된다. 위 표는 몇가지 대표적인 section 들이다.

* 1. **섹션(Section)**

|  |  |  |
| --- | --- | --- |
| 종류 | 이름 | 설명 |
| 코드 | .text | 프로그램을 실행하기 위한 코드를 담고 있는 섹션 |
| 데이터 | .data | 초기화된 전역변수를 담고 있다. 읽고 쓰기 가능 |
| .rdata | 읽기 전용 데이터 섹션. 문자열 표현이나 C++의 가상함수 테이블 저장 |
| .bass | 초기화 되지 않은 전역변수를 담고 있다.  PE 파일 안에는 존재 하지만 가상주소에 매핑 될때는 .data 섹션과 병합된다.( VC++7.0 에서는 .textbss 로 나타난다.) |
| Import API | .idata | 임포트 할 DLL 그 함수들에 대한 정보를 담고 있다.  요즘은 .rdata와 병합되는 경우가 있다. |
| .didat | 지연 로딩(Delay-Loading) 임포트 데이터가 들어 있다.  릴리즈 모드로 링크될때 다른 섹션에 병합된다.  (암시적인 방법과 명시적인 방법의 혼합 로딩 방법) |
| Export API | .edata | Export 할 함수에 대한 정보를 담고 있다. .rdata또는 .text섹션에 병합. |
| 리소스 | .rsrc | 메뉴, 아이콘, 커서, 다이얼로그등의 리소스 관련 data가 들어 있다. |
| 재배치 | .reloc | 재배치 정보를 담고 있는 섹션이다. Exe에는 없고 DLL에만 존재한다.  재배치란 PE이미지를 원하는 기본 주소에 로드하지 못하고 다른 주소에 로드했을 경우 코드 상에서의 관련 주소 참조에 대한 정보를 갱신해야 하는 경우를 말함 |
| TLS | .tls | \_\_declspec(thread)지시어와 함께 선언되는 스레드 지역 저장소를 위한 섹션 |
| C++런타임 | .crt | C++런타임을 지원하기 위한 섹션. 정적 C++객체의 생성자와 소멸자를 호출하는데 이용되는 함수 포인터 등이 있다. |
| Short | .sdata | 전역포인터에 상대적으로 주소 지정될 수 임ㅆ는 읽고 쓰기 가능한 데이터 섹션 |
|  | .srdata |  |
| 예외정보 | .pdata | 예외 정보를 담고 있다. |
| 디버깅 | .debug$s | OBJ 파일에만 존재. 가변 길이 코드 뷰 포맷 심벌레코드의 스트림 |
| .debug$T | OBJ 파일에만 존재. 가변 길이 코드 뷰 포맷 타입 레코드의 스트림 |
| .debug$P | 미리 컴파일 된 헤더를 사용했을 때 OBJ 파일에만 존재하는 섹션 |
| .drectve | OBJ 파일에만 존재 |

* 1. **PE View**



PEView.exe 프로그램을 사용하면 실행파일의 내용을 살펴 볼수 있다.

아래 프로그램을 컴파일해서 Hello.exe라는 실행파일을 만들어 보자.

#include <stdio.h>

void main(){    printf("hello, world\n"); }

위 그림은 Hello.exe를 PEView.exe에서 열었을 때의 화면이다.

1. **Header 분석**

**2.1 IMAGE\_DOS\_HEADER** : 현재는 큰 의미가 없다.

DOS MZ 헤더와 DOS Stub

DOS MZ 헤더 부분에는 DOS에서 실행되기 위해 필요한 정보들이 들어가 있으며

DOS Stub에는 확실하진 않지만 도스에서 실행되는 데이터들이 들어있다.

Windows에선 DOS MZ Header를 편하게 관리할 수 있도록 IMAGE\_DOS\_HEADER라는

구조체를 제공한다. 이 구조체에서 우리가 기억해야할 2개의 중요한 멤버가 있다.

(1) e\_magic : DOS 헤더를 구별하는 식별자. 이 값을 체크해서 우리는 이 파일이 올바  
른 MZ파일인지 검사해야 한다.

모든 실행파일은 파일 가장 첫부분에 'MZ'라는 2바이트의 아스키코드 값

을 가지고 있는데 PE로더는 이 값을 체크해서 맞다면 실행파일을 메모

리에 로드한다.

윈도우즈에서는 이 값을 식별하기 위해 상수를 정의해 놓았는데

IMAGE\_DOS\_SIGNATURE 가 그것이다. 우린 이 값과 e\_magic을 비교해

서 같다면 올바른 MZ 헤더라고 확신할 수 있다.

(2) e\_lfanew : PE헤더가 있는 곳의 offset 값.

\*) Offset과 VA, 그리고 RVA에 대한 개념 이해

PE구조를 익히기 위해선 이 셋은 필수적으로 알아야 한다. Offset 이란 어셈에서 사용되는 용어인데 '특정위치로부터 상대적으로 떨어진 값(Delta)'을 나타내는 말이다.

만약 0x80 이 특정위치이고 현재 0x50 에 있다면 Offset은 0x80 - 0x50 = 0x30 이라고 할 수 있다.

VA는 Virtual Address 를 칭하는 말로 윈도우즈는 메모리를 '가상주소'라는 개념으로 관리하는데 가상주소란 말그대로 실제의 물리적인 주소를 가상주소로 사용한다는 말이다.

가상메모리에서는 0x00000000 부터 0xFFFFFFFF 까지의 메모리를 가상으로 만들고 이 프로그램에서 4바이트가 필요하게 될땐 물리적 메모리를 연속으로 뽑아주는게 아니라

RVA 는 Relative Virtual Address의 약자로서 Offset과 같은 개념이라고 봐도 무방하다. 단지 그 대상이 가상메모리라는 점만 다르다.

IMAGE\_DOS\_HEADER의 e\_lfanew 멤버는 도스헤더로부터 PE헤더까지의 Offset을 담고 있다고 했는데 이제는 무슨뜻인지 알것이라 생각한다.

DOS스텁 부분은 잘 쓰이지 않으므로(사실 쓰이는걸 한번도 못봤다) 그냥 넘어가도록 한다.

\*) e\_lfanew : PE View로 확인해 보면..

E8 00 00 00 🡺 DWORD의 값은 0x000000E8이 됨

해당 주소를 확인해 보면 핵사 “x50 x45가 시작함을 알 수 있다.

즉 이 값을 통해 PE 포멧의 시작 오프셋을 알 수 있게 된다.

[ 정리 ]

1. 실행파일의 제일 첫부분부터 특정 영역까지 PE 포멧으로 나열되어 있으며 그중 DOS Header가 가장 맨 앞에 있다.

2. 해당 파일이 DOS Header로 시작하는지 확인하기 위해서 IMAGE\_DOS\_HEADER 구조체의 e\_magic 멤버를 조사한다(IMAGE\_DOS\_SIGNATURE 상수로 비교한다)

3. PE헤더로 이동하기 위해서 IMAGE\_DOS\_HEADER 구조체의 e\_lfanew 값을 조사한다. 이 멤버는 도스헤더로부터 PE헤더까지의 offset 을 가지고 있다.

* 1. **IMAGE\_NT\_HEADERS**

IMAGE\_NT\_HEADERS 는 PE헤더라고 불리우며 구조체의 구성은 다음과 같다.

* PE의 시작을 알림과 동시에 PE에 관계된 많은 부분을 담고 있다.

typedef struct \_IMAGE\_NT\_HEADERS {

　　DWORD Signature;

　　IMAGE\_FILE\_HEADER FileHeader;

　　IMAGE\_OPTIONAL\_HEADER OptionalHeader;

} IMAGE\_NT\_HEADERS, \*PIMAGE\_NT\_HEADERS;

1. DWORD Signature

이 값은 이 파일이 올바른 PE포멧으로 구성되어 있는지 확인하기 위한 일종의 플래그 값이다. 올바른 PE포멧 파일이라면 항상 값은 4바이트인 PE\0\0 값을 갖는다.

우린 이 값을 체크해서 PE포멧 파일인지 아닌지 구별할 수 있다. 윈도우즈에서는 이 값을 식별하기 위한 IMAGE\_NT\_SIGNATURE 라는 상수를 정의해 놓았다.

// WinNT.h

#define IMAGE\_NT\_SIGNATURE 0x00004550 //PE00

* PE임을 확인하기 위해서는 IMAGE\_DOS\_HEADER 구조체의 e\_lfanew필드가 가리키는 오프셋만큼 파일 포인터를 이동시켜 그 오프셋부터 4byte를 DWORD형으로 읽어들여 그 값을 IMAGE\_NT\_SIGNATURE 매크로 상수와 비교하면 됨

(2) IMAGE\_FILE\_HEADER FileHeader

IMAGE\_FILE\_HEADER 구조체 멤버이다. 이 구조체는 현재 파일이 exe인지 dll인지,, 어느 플랫폼에서 실행되는지, 그리고 후에 설명할 섹션의 갯수가 몇개인지의 정보를 담고 있다.

1. IMAGE\_OPTIONAL\_HEADER OptionalHeader

IMAGE\_OPTIONAL\_HEADER 라는 크기가 224바이트나 되는 멤버가 어마어마하게 많은 구조체이다.

이 구조체엔 앞으로 다루게 될 dll인젝션, API후킹등의 문서를 정확히 이해하기 위해선 필수적으로 알아야 할 핵심적인 멤버들이 포함되어 있는 중요한 구조체이다.

**2.2.1 IMAGE\_FILE\_HEADER**

IMAGE\_NT\_HEADERS의 멤버로 포함되어 있는 IMAGE\_FILE\_HEADER 구조체의 FileHeader 의 정의는 아래와 같다.

typedef struct \_IMAGE\_FILE\_HEADER {

　　WORD Machine;

　　WORD NumberOfSections;

　　DWORD TimeDateStamp;

　　PointerToSymbolTable;

　　DWORD NumberOfSymbols;

　　WORD SizeOfOptionalHeader;

　　WORD Characteristics;

} IMAGE\_FILE\_HEADER, \*PIMAGE\_FILE\_HEADER;

(1) Machine

현재 파일이 어느 플랫폼(CPU)에서 실행되는지의 정보를 가지고 있다.

// WinNT.h

#define IMAGE\_FILE\_MACHINE\_I386 0x014c // Intel 386

#define IMAGE\_FILE\_MACHINE\_IA64 0x0200 //Intel 64

(2) NumberOfSections

섹션을 분석하기 위해 사용되는 값이다. 우리가 파일을 Hex파일등으로 열어 하드코딩시에 이 값을 변경시켜 섹션수를 늘리고 코드를 추가할 수 있다.

즉, IMAGE\_SECTION\_HEADER 구조체 배열의 원소의 개수와 해당 섹션의 개수를 의미함

(3) TimeDateStamp

파일이 생성된 날자와 시간이다. 파일의 등록정보에서 파일이 생성된 날자는 이 값을 참조해서 보여주는 것이다.

(4) SizeOfOptionalHeader

IMAGE\_FILE\_HEADER 바로 다음에 위치한 IMAGE\_OPTIONAL\_HEADER 구조체의 크기이다. 32비트 윈도우즈에선 0xE0 의 어마어마한 크기를 갖는다.

(5) Characteristics

현재 파일이 exe인지 dll파일인지의 플래그를 가지고 있는 변수다.

(6) SizeOfOptionalHeader

IMAGE\_FILE\_HEADER 구조체 바로 다음에 이어서 나오는 IMAGE\_OPTIONAL\_HEADER 구조체의 바이트 수를 나타낸다.

🡺 실행파일의 경우 sizeof(IMAGE\_OPTIONAL\_HEADER)의 크기를 가짐

이 값은 32비트 PE인 경우 0XE0(224byte)이며, 64비트인 경우 0xF0(240byte)이다.

OBJ 파일의 경우는 0임

(7) characteristics

해당 PE 파일에 대한 특정 정보를 나타내는 플래그로서 “WinNT.h”에 다음과 같이 정의되어 있다.

#define IMAGE\_FILE\_RELOCS\_STRIPPED 0x0001 //현재 파일에 재배치 정보가 없다.

#define IMAGE\_FILE\_EXECUTABLE\_IMAGE 0x0002 // 본 파일은 실행 파일 이미지 임

#define IMAGE\_FILE\_LINE\_NUMS\_STRIPPED 0x0004 // 본 파일에 라인 정보가 없다.

//…

Ex) dump를 통한 위치 확인

* IMAGE\_FILE\_HEADER 구조체에 해당하는 부분

0x000000EC ~ 0x000000FF ( 20byte)

최초 4byte : IMAGE\_FILE\_HEADER의 시그네쳐

후 2byte: : 4C 01 🡺 IMAGE\_FILE\_HEADER의 Machine

후 2byte : 03 00 🡺 IMAGE\_FILE\_HEADER의 NumberOfSections

후 4byte : 0D EC F0 40 🡺 IMAGE\_FILE\_HEADER의 TimeDateStamp

기준시인 1790년 1월 1일 오전 9시를 기준으로 링크가 발생한 초단위타임스탬프 -> 1089531101초 -> 2004년 7월 22일 13시 33분 10초

|  |
| --- |
| #include <windows.h>  #include <iostream>  using namespace std;  char pIter[] = "1089531101";  void main()  {  SYSTEMTIME st;  FILETIME ft;  ULARGE\_INTEGER li;  memset(&st, 0x00, sizeof(st));  st.wYear = 1970, st.wMonth = 1, st.wDay = 1, st.wHour = 9;  SystemTimeToFileTime(&st, &ft);  li.HighPart = ft.dwHighDateTime;  li.LowPart = ft.dwLowDateTime;  li.QuadPart += ((unsigned \_\_int64)(\*((LPDWORD)pIter)) \* 10000000L);  ft.dwHighDateTime = li.HighPart;  ft.dwLowDateTime = li.LowPart;  FileTimeToSystemTime(&ft, &st);    char szDateTime[64];  wsprintf(szDateTime, "%04d/%02d/%02d-%02d:%02d:%02d",  st.wYear, st.wMonth, st.wDay, st.wHour, st.wMinute, st.wSecond);  printf("%s\n", szDateTime);  } |

후 4byte\*2 : 00 00 00 00 🡺 IMAGE\_FILE\_HEADER의 PointerToSymbolTable과

00 00 00 00 DWORD NumberOfSymbols

디버깅 관련 정보이고 해당 PE에는 존재하지 않음

후 4byte : E0 00 0F 01 🡺 IMAGE\_FILE\_HEADER의 Characteristics 필드

필드들의 논리 합임

* 1. **IMAGE\_OPTIONAL\_HEADER : 중요한 부분임**

다음과 같은 형태로 이루어져 있다.

typedef struct \_IMAGE\_OPTIONAL\_HEADER {

　　WORD Magic;

　　BYTE MajorLinkerVersion;

　　BYTE MinorLinkerVersion;

　　DWORD SizeOfCode;

　　DWORD SizeOfInitializedData;

　　DWORD SizeOfUninitializedData;

　　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 SzieOfImage;

　　DWORD SizeOfHeaders;

　　DWORD CheckSum;

　　WORD Subsystem;

　　WORD DllCharacteristics;

　　DWORD SizeOfStackReserve;

　　DWORD SizeOfStackCommit;

　　DWORD SizeOfHeapReserve;

　　DWORD SizeOfHeapCommit;

　　DWORD LoaderFlags;

　　DWORD NumberOfRvaAndSizes;

IMAGE\_DATA\_DIRECTORY

DataDirectory[IMAGE\_NUMBEROF\_DIRECTORY\_ENTRIES];

} IMAGE\_OPTIONAL\_HEADER, \*PIMAGE\_OPTIONAL\_HEADER;

구조체의 멤버가 굉장히 많지만 모든 멤버를 숙지할 필요는 없다. 중요하다고 생각되는 멤버만 설명하겠다.

(1) SizeOfCode

코드의 전체 크기이다. 대개 .text 영역(섹션)에 CPU가 실행하는 기계어 코드가 들어있는데 이 코드의 전체 크기이다.

(2) SizeOfInitializedData

섹션들 중에선 읽기가능하고 쓰기가 가능하거나 가능하지 않은,, 프로그램 내에서 쓰이는 변수들이 저장되어 있는 섹션 영역들이 있다(.data나 .idata 같은 영역들). 이 섹션 영역은 섹션의 속성에 따라 데이터가 초기화되어 있는 것들이 있고 그렇지 않은 것들이 있는데 초기화 되어있는 섹션영역들의 총 합이다.

(3) SizeOfUninitializedData

SizeOfInitializedData 와 반대로 생각하면 된다.

(4) AddressOfEntryPoint [중요]

프로그램은 main이나 WinMain,, 혹은 DllMain 에서 시작하게 된다고 배웠을 것이다. 그 위치가 RVA로 저장되어 있다. 주의해야 할점은 프로세스가 가장 시작하는 위치라고 생각해선 안된다. 모든 프로그램은 메인함수가 실행되기 전에 프로그램 실행을 준비하기 위해 Start up 코드를 실행하게 되므로 가장 먼저 시작되는 코드의 위치가 이 RVA가 아니라는 점을 주의하도록 한다.

(5) BaseOfCode [중요]

코드영역의 첫번째 바이트 주소,, 즉 코드영역의 시작 주소이다. 이 값을 AddressOfEntryPonit 와 혼동하지 않도록 한다. 이 둘은 전혀 다르며 항상 AddressOfEntryPoint보다 BaseOfCode의 주소가 앞에 있다. RVA값이라는걸 주의하도록 한다.

(6) BaseOfData [중요]

데이터 영역(대게 .data섹션)의 시작 주소이다. 이 영역은 읽고 쓰기가 가능한 경우가 많다.

역시 RVA값이다.

(7) ImageBase [매우 중요]

이 멤버는 PE파일이 메모리에 매핑될 시작 주소이다. 이는 RVA가 아니라 RVA의 기준이 되는 주소이다. 대게 exe파일의 경우엔 0x00400000 의 값을 갖고 dll은 0x10000000 의 값을 갖는다. 항상 그렇다는건 아니다.

여기서 잠깐 RVA에 짚고 넘어가보자. RVA란 전에도 설명했듯이 기준이 되는 주소로부터부터 얼마만큼 떨어져 있는 위치라고 정의할 수 있다.

PE포멧에서 RVA의 기준은 바로 ImageBase 의 멤버이며 만약 RVA의 값을 가지고 있는 BaseOfCode가 0x00001000h 값이고 ImageBase가 0x00400000h 라고 한다면 PE로더는 코드의 시작주소를 0x00401000h 에 올리게 된다.

즉 해당 파일이 메모리에 올려진 후에 코드의 시작 주소는

ImageBase + BaseOfCode

라는 말이 된다. 간단히 RVA값에 ImageBase를 더해주는게 바로 RVA를 실제 가상주소로 변환하는 방법이다.

여러분이 오류를 범하지 않아야 하는 중요한 사실이 하나 있다!!

RVA는 PE로더에 의해서 메모리에 올려진 후에나 계산해야 하는 것이지, PE로더에 올려지지 않고 헥스에디팅 같은 툴로 분석을 할때 RVA 계산을 해서는 안된다는 점이다!!

PE로더에 의해 올려진 메모리 상태의 PE포멧을 분석하는게 아니라 하드에 저장된 파일 그 자체를 헥스에디팅 등으로 분석한다고 할때 BaseOfCode에 저장되어있는 값의 주소(즉 코드의 시작위치)로 찾아가고자 한다면 BaseOfCode 값에 ImageBase 값을 더해, RVA값을 풀어서 이동해야 하는것이 아니라!! 순수 BaseOfCode 값이 가지고 있는 그 주소 그대로 이동해야 한다.

이해하기 어렵다면 이렇게 생각하라.

파일상태의 PE포멧을 분석할땐 RVA따윈 없다고 무시하고

'PE로더'에 의해 메모리상에 올려진 PE포멧을 분석할땐 RVA값을 가지는 변수엔 무조건 ImageBase를 더한다!! 라고,,

(8) SectionAlignment [중요]

메모리 상에 올려진 후의 섹션의 배치간격이다,, 라고 MSDN에 나와있고 요런식으로만 말해주면 이 멤버가 섭해한다. 좀 더 자세히 알아보자.

VirtualAddress 와 RawAddress 의 차이점이 무엇인지 아는가?

VirtualAddress 는 PE로더에 의해 메모리상에 올려져 있는 상태에서의 주소를 나타내고

RawAddress는 아직 메모리상에 올려져있지 않고 파일상태의 헥스코드의 주소를 나타낸다.

후에 알게되겠지만 섹션헤더부분을 다루게 되면 VirtualAddress 부분이 있고 PointerToRawData 라는 부분이 있다. 이 둘의 차이점은 위에 설명한 것과 같다.

이해하기 어렵다면 .text섹션의 정보를 가지고 있는 섹션헤더가 있다고 생각을 해보자.

이 섹션헤더에는 .text섹션이 위치할 두가지 주소의 정보를 가지고 있다.

왜 위치해야 주소를 두개나 가지고 있을까? 그 이유는 아래와 같다.

첫번째 주소정보는 PE로더가 참조해서 올려야할 가상주소(VirtualAddress)를 위한 주소이고

두번째 주소정보는 메모리에 올려지지 않은 파일(RawAddress)을 위한 주소이다.

.text 섹션의 헤더중 VirtualAddress 멤버는 0x2000 의 값을 가지고 있고,

PointerToRawData(RawAddress) 멤버는 0x1000 의 값을 가지고 있다고 하자.

우리가 그 PE파일을 실행을 하면 PE로더는 섹션헤더에 담겨있는 정보들을 참고해 .text 섹션을 메모리 주소 어딘가에 올려야 한다.

일단 ImageBase가 0x00400000이므로 DOS헤더를 0x00400000에 올린다.

그리고 .text섹션의 헤더중 VirtualAddress 값을 조사하니 0x2000 이란 값이 나왔다.

그러면 PE로더는 0x00400000 에 0x2000 을 더해서 0x00402000 메모리 주소에 .text섹션을 올린다!!

간단하잖은가? 이제 VirtualAddress는 이해됐을 것이라 믿는다!!

이제 우리가 컴파일러가 되어보자.

어쩌고 저쩌고 해서 컴파일러 내부에선 코드를 기계어로 변환시키고 PE포멧에 맞게 코드를 차곡차곡 분석해서 .text영역을 만들었다. 이제 이 .text영역을 파일로 만들어내야 한다. 그런데 이 .text영역을 파일 어디부분에 저장시키지?

바로 이런 상황에서 기준이 되는게 RawAddress다!!

파일상에서 0x1000 번지에 저장을 시키기 위해 PE포멧을 생성하고 PointerToRawData 값을 0x1000으로 셋팅한 다음 .text 섹션영역을 파일상의 0x1000 에 저장시킨다. 그래서 우리는 파일상태로 분석할때 .text 영역의 위치를 알아내기 위해서 단지 PointerToRawData 값을 참조해 그 주소로 이동하면 되는 것이다. 간단하잖은가!!

이제 다시 본 주제로 돌아와서 SectionAlignment 에 대해 알아보자.

SectionAlignment는 섹션이 PE로더에 의해 메모리에 올려질때 항상 이 멤버의 배수값으로 위치한다. 이 값이 0x1000 이라고 한다면 항상 섹션헤더의 VirtualAddress 값은 이 값의 배수가 된다.

0x1000, 0x2000, 0x3000 은 가능하나 0x1500, 0x2004, 0x3600 이런 값은 불가능하다는 소리다.

이게 바로 Alignment 의 개념이다. 머리속에 뿌리를 박아버리자.

그리고 섹션헤더의 VirtualAddress 의 멤버는 항상 SectionAlignment 의 영향을 받는다.

(9) FileAlgnment [중요]

이는 SectionAlignment 와 기본 개념은 똑같지만 그 기준이 PointerToRawData 에 영향을 준다는 것만 다르다. 즉,, 파일상의 섹션위치 간격이다.

(10) SizeOfImage

이미지 파일의 전체크기. 즉 파일의 전체 크기이다.

(11) SizeOfHeaders

도스헤더, 도스스텁, PE헤더, 섹션헤더 모두를 더한 값.

즉 PE포멧의 모든 헤더를 더한 값이다.

(12) SizeOfStackReserve

프로그램에서 사용될 스택을 얼마만큼 예약할지의 바이트 기준 값이다.

(13) SizeOfStackCommit

프로그램에서 사용될 스택을 얼마만큼 commit 할지의 바이트 기준 값이다.

(14) SizeOfHeapReserve

프로그램에서 사용될 힙을 얼마만큼 예약할지의 바이트 기준 값이다.

(15) SizeOfHeapCommit

프로그램에서 사용될 힙을 얼마만큼 commit 할지의 바이트 기준 값이다.

(16) DataDirectory

이 멤버는 Export table, Import table, Resource 영역, Exception 영역, 보안영역, 디버그 영역등을 접근할 수 있는 주소를 가지고 있는 배열이다. 이 멤버는 IMAGE\_DATA\_DIRECTORY 구조체인데 선언은 다음과 같이 되어있다.

typedef \_IMAGE\_DATA\_DIRECTORY {

　　DWORD VirtualAddress;

　　DWORD VirtualSize;

} IMAGE\_DATA\_DIRECTORY, \*PIMAGE\_DATA\_DIRECTORY;

총 14개의 배열로 구성되어 있으며 그 테이블은 아래와 같다.

00　Export table

01　Import table

02　Resource

03　Exception

04　Security

05　Base relocation

06　Debug

07　Copyright string

08　Unknown

09　Thread local storage(TLS)

10　Load configuration

11　Bound Import

12　Import Address Table

13　Delay Import

14　COM descriptor

우리는 Import 함수들을 조사할 때 DataDirectory 의 두번째 배열을 참조하게 된다.

이 배열에는 각각 해당영역으로 갈 수 있는 RVA주소와 그 영역의 크기 정보를 담고 있다.

import table 로 이동하면 IMAGE\_IMPORT\_DESCRIPTOR 구조체를 얻을 수 있는데 이 구조체는 해당 파일이 import 하고 있는 dll과 함수정보들을 가지고 있다. API 후킹시에 사용되는 중요한 테이블이다!!

이로서 IMAGE\_OPTIONAL\_HEADER 에 대한 설명이 끝났다.

옵셔널 헤더 바로 뒤에는 섹션헤더(IMAGE\_SECTION\_HEADER)가 배열로 위치하게 되는데 이는 다음 문서에서 다루기로 한다.

이 문서에서 꼭 숙지해야 할 부분은 RVA의 개념과 VirtualAddress/RawAddress 의 다른 점,, 그리고 DataDirectory 멤버(추후에 더 자세히 설명)이다.

* 1. **IMAGE\_SECTIONL\_HEADER : 중요한 부분임**

섹션헤더란 섹션 테이블이라고도 불리우며 PE포멧의 옵셔널 헤더 바로 뒤에 구조체 배열 형식으로 위치해 있다. 이 구조체는 IMAGE\_SECTION\_HEADER 라는 이름으로 윈도우즈에서 정의해 두었으며 섹션의 갯수가 5개라고 하면 섹션정보가 담겨있는 섹션헤더 배열이 다섯개가 있고 마지막에 null로 채워진 섹션헤더 구조체가 위치하게 된다. 섹션갯수 정보는 IMAGE\_FILE\_HEADER의 NumberOfSections 멤버가 가지고 있다.

섹션헤더의 구조체 정의는 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;

1. Name

섹션의 네임을 나타내는 멤버이다. 이 멤버는 섹션이름을 나타내는 문자열을 저장하게 되는데 IMAGE\_SIZEOF\_SHORT\_NAME 은 8바이트 크기를 나타내는 상수이다. .text나 .data 같은 이름을 갖지만 섹션의 이름으로 섹션의 성격을 파악해선 안된다. 이 멤버의 값은 null일 수도 있으며 섹션이름이 같다고 하여 항상 같은 속성을 지니진 않는다. 섹션의 속성을 파악하기 위해선 Characteristics 멤버를 참조하여야 한다.

2. PhysicalAddress/VirtualSize

PE로더에 의해 이미지가 메모리에 올려진 후에 해당 섹션이 얼마만큼의 크기를 가지고 있게 되는지의 정보이다. 이 멤버는 Misc 유니온 구조체의 멤버이며 이 이름은 변수 의미에 대해 충분한 오해의 소지가 있다. 이는 물리적주소를 의미하는게 아니라 가상주소 상에서 해당 섹션의 크기를 나타내는 멤버라는 점을 주의하기 바란다.

3. VirtualAddress

PE로더에 의해 이미지가 메모리에 올려진 후에 해당 섹션이 어느 주소에 위치하는지의 RVA 주소를 값으로 가지고 있다. 이 멤버는 항상 IMAGE\_OPTIONAL\_HEADER의 멤버인 SectionAlignment 의 배수값을 가진다. PE로더가 메모리에 섹션을 올릴때 이 멤버의 값을 참조하게 된다.

이 멤버의 값이 0x1000h 이고 IMAGE\_OPTIONAL\_HEADER의 ImageBase 값이 0x00400000h 라면 PE로더는 해당 섹션을 0x00401000h 번지에 올린다. 항상 이 값은 ImageBase를 기준으로 하는 RVA값이라는 걸 주의하기 바란다.

4. SizeOfRawData

이 멤버는 Raw data 상에서 해당 섹션에 대한 실제 사용된 크기 정보를 담고 있다. 우리가 이 멤버에 관심을 가져야 하는 이유는 해당 섹션의 빈공간이 얼마나 있는가를 알아내기 위해선 반드시 필요하기 때문이다.

우리가 옵셔널 헤더의 멤버인 SectionAlignment / FileAlignment 에서 설명했듯이 섹션의 영역은 alignment 영향을 받는다고 했다. Raw data 상에서 섹션은 항상 FileAlignment 의 배수값을 갖게 되지만, 그렇다고 해서 섹션은 항상 배분된 영역만큼 모두를 사용하지 않는다. 대게 사용하지 않고 빈 공간이 남아있기 마련이다.

쉽게 풀어서 얘기해보자. 섹션중에서는 CPU 가 실행할 수 있는 기계어 코드가 담겨있는 코드 섹션부분이 있다. 이 코드섹션이 Raw data상에 0x00001000h 주소에 위치해 있고 코드의 크기는 0x00000F00h 이며, FileAlignment 값은 0x00001000h 라고 하자.

코드섹션 위치 : 0x00001000h

코드섹션에서 사용되는 크기 : 0xF00h

FileAlignment 의 값 : 0x1000h

그렇다면 코드섹션에 대한 사이즈는 0xF00h 면 되므로 코드섹션의 크기는 0x00001000h 에서 0x00000F00h 까지인 0xF00h 만 할당하면 될것이다. 하지만 윈도우즈는 이런식으로 섹션공간을 할당하지 않는다. Raw data 상에서는 반드시 FileAlignment 의 배수로 섹션공간을 할당하기 때문에 0x00001000h 에서 0x00002000h 까지인 0x1000h 크기로 코드섹션을 할당하게 된다.

이러한 윈도우즈의 섹션 관리법으로 인해 각 섹션들은 대게 사용하지 않는 빈공간을 가지고 있게 된다. 우리는 이 빈공간에 우리의 코드를 작성함으로서 남이 만든 프로그램을 내 뜻대로 조작할 수 있는것이다!! 물론 우리가 섹션을 직접 추가하거나 섹션테이블의 값을 조정해서 현재 존재하는 섹션의 크기를 늘릴 수도 있다.

섹션을 분석함으로 해서 다른 프로그램의 영역에 내가 원하는 영역을 만들 수도 있다!!

5. PointerToRawData

raw data가 파일상의 어느 주소에 위치에 있는지 나타내는 변수이다.

PointerToRawData 가 0x1000h 이고 FileAlignment 가 0x1000h, SizeOfRawData 가 0x500h 이라면 이 섹션은 0x00001000h 부터 0x00000fffh 까지 이며, 실제 사용되고 있는 영역은 0x00001000h ~ 0x000004ffh 이고 0x00000500h ~ 0x00000fffh 의 공간은 할당된채로 사용되지 않는 영역(빈 영역)이다.

6. Characteristics

해당 섹션에 대한 속성정보를 플래그로 가지고 있다. 윈도우에선 여러가지 속성을 상수 매크로로 정의해 놓았는데 여기선 몇가지만 알아보도록 한다.

IMAGE\_SCN\_CTN\_CODE　　　　　　　 0x00000020　　// 코드로 채워진 섹션

IMAGE\_SCN\_CTN\_INITIALIZED\_DATA 　 0x00000040 　// 데이터가 초기화된 섹션

IMAGE\_SCN\_CTN\_UNINITIALIZED\_DATA 0x00000080 　// 데이터가 비초기화된 섹션

IMAGE\_SCN\_MEM\_EXECUTE 　　　　 0x20000000 　// 코드로서 실행될 수 있는 섹션

IMAGE\_SCN\_MEM\_READ 　　　　　　 0x40000000 　// 읽기 가능영역 섹션

IMAGE\_SCN\_MEM\_WRITE 　　　　　　0x80000000 　// 쓰기 가능영역 섹션

해당 섹션의 속성값을 알아내기 위해선 AND연산을 이용한다. 간단히 해당 섹션이 읽기 속성을 가지고 있는지 검사하기 위해선

if(SectionHeader.Characteristics & IMAGE\_SCN\_MEM\_READ)

또는

if(SectionHeader.Characteristics & 0x40000000)

코드가 참이면 읽기 속성을 가지고 있는 것이고 읽기/쓰기 둘다 가지고 있는지 체크하기 위해선

if(SectionHeader.Characteristics & (IMAGE\_SCN\_MEM\_READ | IMAGE\_SCN\_MEM\_WRITE))

또는

if(SectionHeader.Characteristics & 0xC0000000)

코드가 참인지 검사하면 된다. 하지만 위의 매크로 상수는 특정 헤더파일에 정의되어 있어서 그 헤더파일을 include 해주어야 사용할 수 있는듯 하다. 본인은 그 헤더파일을 모르므로 그냥 헥스코드 값을 그대로 사용한다.

이 속성값을 변경함으로 해서 쓰기가 금지된 섹션에 write할 수도 있고 쓰기 가능한 섹션에 write를 금지 시킬수도 있다.

**【 섹션분석 전반적 과정 】**

1. 파일매핑(CreateFileMapping)을 이용해 PE이미지를 메모리상에 올린다. 이렇게 해두면 메모리를 읽는것같이 접근할 수 있어, PE분석을 하기 용이하다.

2. 먼저 Dos header 영역에 접근해서 e\_lfanew 값을 참조해 NT header 주소로 이동한다.

3. Dos/NT 헤더의 signature 값을 조사해서 타깃파일이 올바른 플래그를 가지고 있는지, 즉 올바른 PE파일인지 검사하고 PE파일이 아니라면 프로그램을 종료한다.

4. IMAGE\_FILE\_HEADER 에서 NumberOfSections 값을 조사해 섹션의 갯수를 알아낸다.

5. IMAGE\_OPTIONAL\_HEADER 에서 SectionAlignment 와 FileAlignment 의 값을 알아낸다.

6. IMAGE\_OPTIONAL\_HEADER 다음에 위치한 IMAGE\_SECTION\_HEADER 로 이동한다.

7. 해당 섹션에 대한 정보를 알아내 화면에 출력한다.

8. 7번의 과정을 NumberOfSections 만큼 반복한다.

9. 파일맵핑시 얻은 자원을 반환하고 프로그램을 종료한다.

자 이제부터 과정을 하나 하나씩 밟아가도록 해보자.

- 1번 과정 / 파일맵을 이용한 PE파일 메모리에 올리기

CreateFileMapping을 이용하여 PE파일 이미지 전체를 메모리에 올린다. 물론 일반 파일읽기 쓰기 함수를 사용해 분석하는 방법도 하지만 파일맵핑을 이용하는 방법이 가장 깔끔한 듯 해서 여기선 이 방법을 사용하였다.

-----------------------------------------------------------------

// 섹션을 조사할 PE파일을 File-mapping을 이용하여 메모리에 올린다.

hFile = CreateFile("C:\\Windows\\system32\\notepad.exe", GENERIC\_READ,

　　　　　　　　FILE\_SHARE\_READ, NULL, OPEN\_EXISTING,

　　　　　　　　FILE\_ATTRIBUTE\_NORMAL, NULL);

hFileMap = CreateFileMapping(hFile, NULL, PAGE\_READONLY, 0, 0, "PE");

pBaseAddr = MapViewOfFile(hFileMap, FILE\_MAP\_READ, 0, 0, 0);

-----------------------------------------------------------------

조사할 파일은 윈도우 내장파일인 메모장으로 하였으며 window xp 기준의 메모장 path이므로 win9x 계열의 윈도우에선 path를 적당히 바꿔주도록 한다. 소스를 보면 알겠지만 모두 읽기 전용으로 파일을 열람한다. 우리는 PE section을 조사할때 쓰기연산은 하지 않을것이다.

그리고 CreateFile시 우리가 섹션을 조사하는 동안 파일변경을 할 수 없도록 FILE\_SHARE\_READ 플래그를 지정한다.

위 코드에는 파일의 가장 첫번째 주소를 pBaseAddr 라는 LPVOID형의 변수에 대입하였다.

앞으로 이 변수를 사용하여 PE파일의 헤더에 접근할 것이다.

- 2번, 3번 과정 / Dos 헤더 영역에서 NT 헤더 영역까지 이동

파일의 시작주소를 담고 있는 pBaseAddr의 주소를 도스헤더에 대입시키고 e\_lfanew 값을 참조해 NT헤더로 이동한 뒤 올바른 Signature를 가지고 있는지 검사한다.

-----------------------------------------------------------------

PIMAGE\_DOS\_HEADER pDosHdr = (PIMAGE\_DOS\_HEADER)pBaseAddr;

PIMAGE\_NT\_HEADERS pNtHdr = (PIMAGE\_NT\_HEADERS)

　　　　　　　　　　　((DWORD)pBaseAddr + (DWORD)pDosHdr->e\_lfanew);

if(IMAGE\_DOS\_SIGNATURE != pDosHdr->e\_magic ||

　　　　IMAGE\_NT\_SIGNATURE != pNtHdr->Signature)

　　restore\_filemapping\_resources(hFile, hFileMap, pBaseAddr);

-----------------------------------------------------------------

코드를 잘 살펴보면 pBaseAddr 는 주소 그대로 PIMAGE\_DOS\_HEADER 에 대입하였다. 파일의 가장 첫부분과 IMAGE\_DOS\_HEADER 구조체가 동일하므로 바로 대입하였는데 이는 포인터,, 즉 주소이므로 포인터 구조체형인 PIMAGE\_DOS\_HEADER를 사용하였다.

그런다음 도스헤더멤버인 e\_lfanew 를 pBaseAddr 에 더해줬는데 e\_lfanew는 PE로 가는 상대주소 값(rva)이기 때문에 반드시 기준이 되는 주소(pBaseAddr)을 더해주어야 한다.

위 코드중 restore\_filemapping\_resources 는 의도되지 않은 상황으로 인해 프로그램을 종료시 파일맵핑시 할당받았던 윈도우 자원을 반환하는 작업을 하는 함수이다.

- 4번, 5번 과정 / IMAGE\_NT\_HEADER에서 섹션에 대한 정보 몇가지를 알아낸다.

-----------------------------------------------------------------

// 섹션의 갯수와 alignment 정보를 알아낸다

iNumberOfSections = pNtHdr->FileHeader.NumberOfSections;

dwSectionAlignment = pNtHdr->OptionalHeader.SectionAlignment;

dwFileAlignment = pNtHdr->OptionalHeader.FileAlignment;

// 알아낸 정보를 출력

printf("- Section Information -\n");

printf(" Number of sections: %d\n", iNumberOfSections);

printf(" Section Alignment : 0x%08X\n", dwSectionAlignment);

printf(" File Alignment : 0x%08X\n", dwFileAlignment);

-----------------------------------------------------------------

어려운 부분은 없다. IMAGE\_FILE\_HEADER 의 NumberOfSections 정보는 후에 모든 섹션의 정보를 조사하기 위해 루프를 돌때 참조하게 되므로 따로 저장해 두었다.

- 6번 과정 / 섹션헤더로 이동

IMAGE\_SECTION\_HEADER 정보는 IMAGE\_OPTIONAL\_HEADER 바로 뒤에 위치하게 된다. 옵셔널 헤더는 IMAGE\_NT\_HEADER 구조체에서 가장 마지막 멤버이기 때문에, 섹션헤더로 이동하는 간단한 방법으로 현재 NT헤더의 위치에 NT헤더 구조체의 크기를 더해 주었다.

물론 IMAGE\_OPTIONAL\_HEADER 의 주소에 IMAGE\_FILE\_HEADER의 멤버인 SizeOfOptionalHeader 값을 더해주어도 상관없다.

아니면

NT헤더주소 + 4(Signature) + IMAGE\_FILE\_HEADER크기 + IMAGE\_OPTIONAL\_HEADER 크기

이런 방법으로 계산해도 된다. 가장 첫번째 방법이 깔끔하므로 여기선 첫번째 방법을 사용하였다.

**------------------------------------------------------------------**

PIMAGE\_SECTION\_HEADER pSecHdr = (PIMAGE\_SECTION\_HEADER)

　　　　　　　　　　　　　　((DWORD)pNtHdr + sizeof(IMAGE\_NT\_HEADERS));

-----------------------------------------------------------------

- 7번, 8번 과정 / 섹션의 정보를 출력하는 과정을 섹션의 갯수만큼 반복한다.

섹션의 정보 중 섹션의 이름, Raw data상의 위치/크기, Virtual data상의 위치/크기, 섹션속성 정보만을 출력한다.

-----------------------------------------------------------------

printf("%8s %10s %10s %10s %10s %10s\n",

　　　　"Name", "Raw Addr", "Raw Size", "Vir Addr", "Vir Size", "Characer");

for(int i = 0; i < iNumberOfSections; i++)

{

　　printf("%8s 0x%08X 0x%08X 0x%08X 0x%08X 0x%08X",

　　　pSecHdr->Name,

　　　pSecHdr->PointerToRawData, pSecHdr->SizeOfRawData,

　　　pSecHdr->VirtualAddress, pSecHdr->Misc.VirtualSize,

　　　pSecHdr->Characteristics);

　　// 섹션 속성 플래그의 종류(Characteristics flags)

　　// IMAGE\_SCN\_CTN\_CODE 0x00000020 코드로 채워진 섹션

　　// IMAGE\_SCN\_CTN\_INITIALIZED\_DATA 0x00000040 데이터가 초기화된 섹션

　　// IMAGE\_SCN\_CTN\_UNINITIALIZED\_DATA 0x00000080 데이터가 비초기화된 섹션

　　// IMAGE\_SCN\_MEM\_EXECUTE 0x20000000 코드로서 실행될 수 있는 섹션

　　// IMAGE\_SCN\_MEM\_READ 0x40000000 읽기 가능영역 섹션

　　// IMAGE\_SCN\_MEM\_WRITE 0x80000000 쓰기 가능영역 섹션

　　// 섹션 속성 검사

　　// if(0x00000040 & pSecHdr->Characteristics)　　printf("/Initialized");

　　// if(0x00000080 & pSecHdr->Characteristics)　　printf("/Uninitialized");

　　if(0x20000000 & pSecHdr->Characteristics)　　printf("/Code");

　　if(0x40000000 & pSecHdr->Characteristics)　　printf("/Read");

　　if(0x80000000 & pSecHdr->Characteristics)　　printf("/Write");

　　puts("");

　　pSecHdr++; // 다음 섹션 정보로 이동

}

-----------------------------------------------------------------

위 코드에서 눈 여겨 봐야할점은 루프를 NumberOfSection 크기만큼 돌렸다는 것이다. 섹션헤더는 섹션의 갯수만큼의 배열로 되어있고 맨 마지막은 null로 채워진 동일 구조체로 되어있다고 해서 null구조체를 만날때까지를 조건으로 루프를 돌려선 안된다. 맨 마지막 섹션구조체 배열이 맨 처음으로 이어지는 경우도 있다고 하니(확인해 본적은 없지만,,) 항상 섹션구조체를 조사할땐 NumberOfSections 값을 사용하도록 하자.

그런 다음, 섹션구조체의 Name(이름), PointerToRawData(Raw상의 섹션위치), SizeOfRawData(실제쓰인크기), VirtualAddress(PE로더 의해 위치할 섹션위치), VirtualSize(현재 섹션을 위해 할당될 가상메모리 크기), Characteristics(섹션 속성)등을 조사한다.

다음 섹션헤더의 이동은 pSecHdr++ 코드를 이용해서 간단히 해결했다.

- 9번 과정 / 파일맵, 파일핸들 반환

파일맵을 위해 할당받은 자원을 다시 윈도우즈에 반환하는 함수인 restore\_filemapping\_resources 함수를 호출한다. 함수정의는 다음과 같다.

-----------------------------------------------------------------

void restore\_filemapping\_resources(HANDLE hFile, HANDLE hFileMap, LPVOID pMapView)

{

　　UnmapViewOfFile(pMapView);

　　CloseHandle(hFileMap);

　　CloseHandle(hFile);

}

-----------------------------------------------------------------