Introduction

Windows 운영체제의 **PE(Portable Executable) File Format** 에 대해서 아주 상세히 공부해 보도록 하겠습니다.

PE format 을 공부하면서 Windows 운영체제의 가장 핵심적인 부분인
Process, Memory, DLL 등에 대한 내용을 같이 정리할 수 있습니다.

PE(Portable Executable) File Format

PE 파일의 **종류**는 아래와 같습니다.

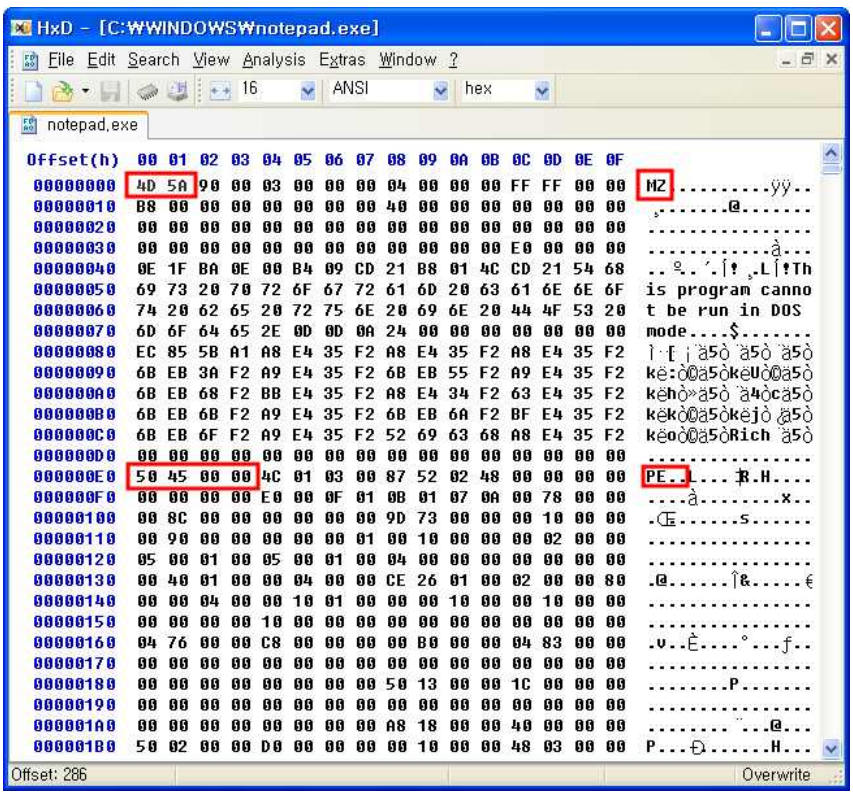
- 실행 파일 계열 : EXE, SCR
- 라이브러리 계열 : DLL, OCX
- 드라이버 계열 : SYS
- 오브젝트 파일 계열 : OBJ

엄밀히 얘기하면 OBJ(오브젝트) 파일을 제외한 모든 파일들은 **실행 가능한 파일** 입니다.

DLL, SYS 파일등은 쉘(Explorer.exe) 에서 직접 실행 할 수는 없지만,
다른 형태의 방법(디버거, 서비스, 기타)을 이용하여 실행이 가능한 파일들입니다.

* PE 공식 스펙 에는 컴파일 결과물인 **OBJ**(오브젝트) 파일도 PE 파일로 간주합니다.
  하지만 OBJ 파일 자체로는 어떠한 형태의 실행도 불가능하므로 리버싱에서 관심을 가질 필요는 없습니다.

간단한 설명을 위해서 노트패드(notepad.exe) 파일을 hex editor 를 이용해서 열어보겠습니다.



<Fig. 1>

<Fig. 1> 은 notepad.exe 파일의 시작 부분이며, **PE 파일의 헤더 (PE header)** 부분입니다.

바로 이 PE header 에 notepad.exe 파일이 실행되기 위해 필요한 모든 정보가 적혀있습니다.

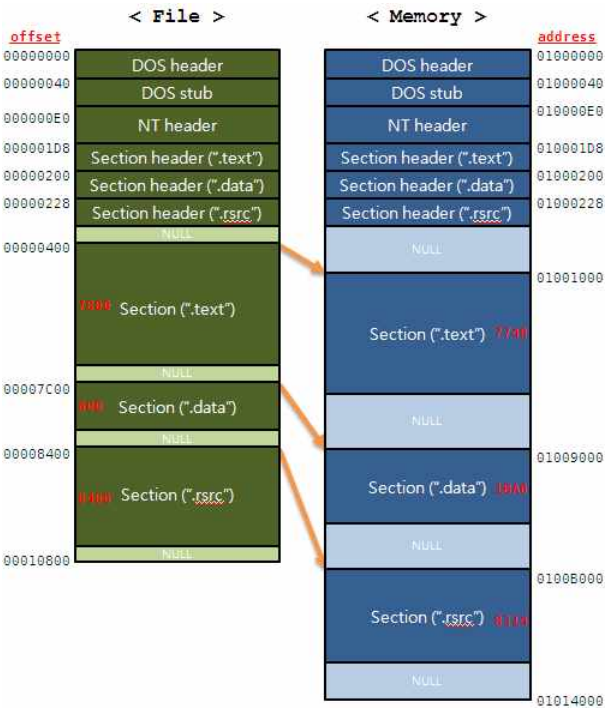
어떻게 메모리에 적재되고, 어디서부터 실행되어야 하며, 실행에 필요한 DLL 들은 어떤것들이 있고,
필요한 stack/heap 메모리의 크기를 얼마로 할지 등등...

수 많은 정보들이 PE header 에 **구조체 형식**으로 저장되어 있습니다.

즉, PE File Format 을 공부한다는 것은 PE header 구조체를 공부한다는 것과 같은 말입니다.

Basic Structure

일반적인 PE 파일의 기본 구조입니다. (notepad.exe)



<Fig. 2>

<Fig. 2> 는 notepad.exe 파일이 메모리에 적재(loading 또는 mapping)될 때의 모습을 나타낸 그림입니다.
많은 내용을 함축하고 있는데요, 하나씩 살펴보겠습니다.

- DOS header 부터 Section header 까지를 **PE Header**, 그 밑의 Section 들을 합쳐서 **PE Body** 라고 합니다.
- 파일에서는 **offset** 으로, 메모리에서는 **VA(Virtual Address)** 로 위치를 표현합니다.
- 파일이 메모리에 로딩되면 모양이 달라집니다. (Section 의 크기, 위치 등)
- 파일의 내용은 보통 코드(".text" 섹션), 데이타(".data" 섹션), 리소스(".rsrc") 섹션에 나뉘어서 저장됩니다.
  반드시 그런것은 아니며 개발도구(VB/VC++/Delphi/etc)와 빌드 옵션에 따라서
  섹션의 이름, 크기, 개수, 저장내용 등은 틀려집니다. 중요한 것은 **섹션이 나뉘어서 저장** 된다는 것입니다.
- **Section Header** 에 각 Section 에 대한 파일/메모리에서의 크기, 위치, 속성 등이 정의 되어 있습니다.

- PE Header 의 끝부분과 각 Section 들의 끝에는 **NULL padding** 이라고 불리우는 영역이 존재합니다.
  컴퓨터에서 파일, 메모리, 네트워크 패킷 등을 처리할 때 효율을 높이기 위해 **최소 기본 단위** 개념을 사용하는데,
  PE 파일에도 같은 개념이 적용된 것입니다.

- 파일/메모리에서 섹션의 시작위치는 각 파일/메모리의 최소 기본 단위의 **배수**에 해당하는 위치여야 하고,
  빈 공간은 NULL 로 채워버립니다. (<Fig. 2> 를 보면 각 섹션의 시작이 이쁘게 딱딱 끊어지는 걸 볼 수 있습니다.)

VA & RVA

VA (Virtual Address) 는 프로세스 가상 메모리의 절대 주소를 말하며,
RVA (Relative Virtual Address) 는 어느 기준위치(ImageBase) 에서부터의 상대 주소를 말합니다.

VA 와 RVA 의 관계는 아래 식과 같습니다.

RVA + ImageBase = VA

PE header 내의 많은 정보는 RVA 형태로 된 것들이 많습니다.
그 이유는 PE 파일(주로 DLL)이 프로세스 가상 메모리의 특정 위치에 로딩되는 순간
이미 그 위치에 다른 PE 파일(DLL) 이 로딩 되어 있을 수 있습니다.

그럴때는 재배치(Relocation) 과정을 통해서 비어 있는 다른 위치에 로딩되어야 하는데,
만약 PE header 정보들이 VA (Virtual Address - 절대주소) 로 되어 있다면 정상적인 엑세스가 이루어지지 않을것입니다.

정보들이 RVA (Relative Virtual Address - 상대주소) 로 되어 있으면 Relocation 이 발생해도
기준위치에 대한 상대주소는 변하지 않기 때문에 아무런 문제없이 원하는 정보에 엑세스 할 수 있을 것입니다.

DOS Header

Microsoft 는 PE 파일 포멧을 만들때 당시에 널리 사용되던 DOS 파일에 대한 **하위 호환성**을 고려해서 만들었습니다.
그 결과로 PE header 의 제일 앞부분에는 기존 DOS EXE header 를
확장시킨 **IMAGE_DOS_HEADER** 구조체가 존재합니다.

```
typedef struct _IMAGE_DOS_HEADER {
    WORD   e_magic;         // DOS signature : 4D5A ("MZ")
    WORD   e_cblp;
    WORD   e_cp;
    WORD   e_crlc;
    WORD   e_cparhdr;
    WORD   e_minalloc;
    WORD   e_maxalloc;
    WORD   e_ss;
    WORD   e_sp;
    WORD   e_csum;
    WORD   e_ip;
    WORD   e_cs;
    WORD   e_lfarlc;
    WORD   e_ovno;
    WORD   e_res[4];
    WORD   e_oemid;
    WORD   e_oeminfo;
    WORD   e_res2[10];
    LONG   e_lfanew;        // offset to NT header
  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

* 출처 : Microsoft 의 Visual C++ 에서 제공하는 winnt.h
```

<IMAGE_DOS_HEADER>

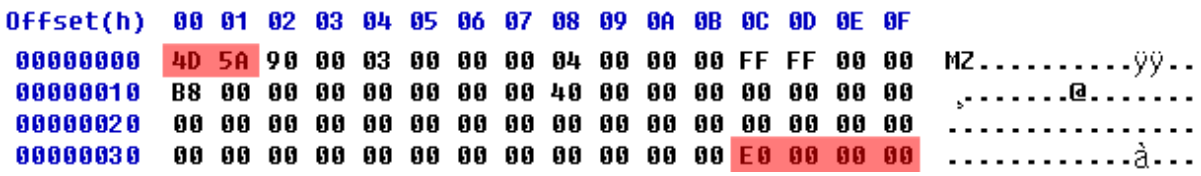
IMAGE_DOS_HEADER 구조체의 크기는 40h 입니다.
이 구조체에서 꼭 알아둬야 할 중요한 멤버는 **e_magic** 과 **e_lfanew** 입니다.

- e_magic : DOS signature (4D5A => ASCII 값 "MZ")
- e_lfanew : NT header 의 옵셋을 표시 (**가변**적인 값을 가짐)

모든 PE 파일은 시작 부분(e_magic)에 DOS signature ("MZ") 가 존재하고,
e_lfanew 값이 가리키는 위치에 NT header 구조체가 존재해야 합니다.
(NT header 구조체의 이름은 **IMAGE_NT_HEADERS** 이며 나중에 소개됩니다.)

* **'MZ'** 는 Microsoft 에서 DOS 실행파일을 설계한 **Mark Zbikowski** 라는 사람의 이니셜입니다.
(출처 : http://en.wikipedia.org/wiki/Mark_Zbikowski)

notepad.exe 를 hex editor 로 열어서 IMAGE_DOS_HEADER 구조체를 확인해 보겠습니다.



Offset(h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	
00000000	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00	MZ.....ÿÿ..
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	E0	00	00	00	.....à...

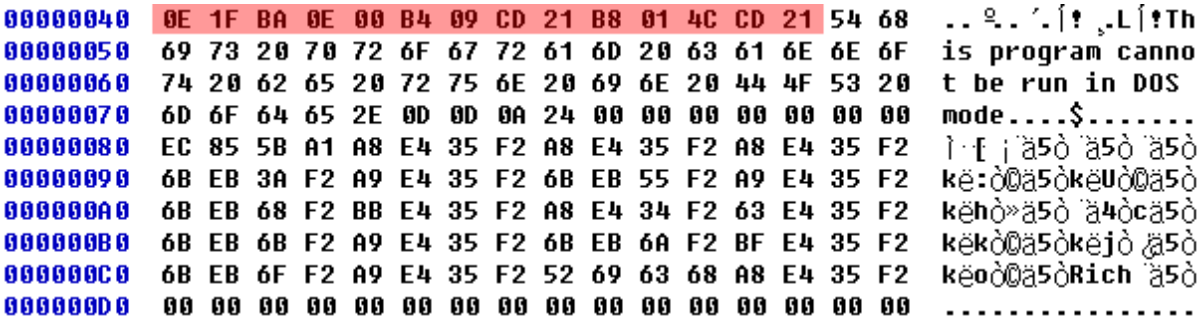
과연 PE 스펙에 맞게 파일 시작 2 byte 는 4D5A 이며, e_lfanew 값은 000000E0 입니다. (E0000000 이 아닙니다.)
(참고 : Intel 계열 PC 는 자료를 역순으로 저장합니다. 이것을 Little Endian 표기법이라고 합니다.)

시험삼아 이 값들을 변경한 후 저장해서 실행해 보세요.
정상 실행되지 않을 것입니다. (PE 스펙에 따라서 더 이상 PE 파일이 아니거든요.)

DOS Stub

DOS Header 밑에는 DOS Stub 이 존재합니다.
DOS Stub 의 존재여부는 **옵션**이며 크기도 일정하지 않습니다.
(DOS Stub 은 없어도 파일 실행에는 문제가 없습니다.)

DOS Stub 은 코드와 데이타의 혼합으로 이루어져 있으며, 아래 그림에 notepad.exe 의 DOS Stub 이 나타나 있습니다.



00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68	..º..´.Í! ¸.LÍ!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	EC	85	5B	A1	A8	E4	35	F2	A8	E4	35	F2	A8	E4	35	F2	ì…[¡¨ä5ò¨ä5ò¨ä5ò
00000090	6B	EB	3A	F2	A9	E4	35	F2	6B	EB	55	F2	A9	E4	35	F2	kë:ò©ä5òkëUò©ä5ò
000000A0	6B	EB	68	F2	BB	E4	35	F2	A8	E4	34	F2	63	E4	35	F2	këhò»ä5ò¨ä4òcä5ò
000000B0	6B	EB	6B	F2	A9	E4	35	F2	6B	EB	6A	F2	BF	E4	35	F2	këkò©ä5òkëjò¿ä5ò
000000C0	6B	EB	6F	F2	A9	E4	35	F2	52	69	63	68	A8	E4	35	F2	këoò©ä5òRich¨ä5ò
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

위 그림에서 붉은색으로 표시된 부분은 **16 bit 어셈블리 명령어** 입니다.
32 bit 윈도우즈에서는 이쪽 명령어가 실행되지 않습니다. (PE 파일로 인식하기 때문에 아예 이쪽 코드를 무시하지요.)

DOS 환경에서 실행하거나, DOS 용 디버거 (debug.exe) 를 이용해서 실행할 수 있습니다.
(DOS EXE 파일로 인식합니다. 이들은 PE 파일 포멧을 모르니까요...)

콘솔 윈도우(cmd.exe)를 띄워서 아래와 같이 명령을 입력합니다.

```
C:\WINDOWS>debug notepad.exe
-u
0D1E:0000 0E          PUSH    CS
0D1E:0001 1F          POP     DS
0D1E:0002 BA0E00      MOV     DX,000E    ; DX = 0E : "This program cannot be run in
DOS mode"
0D1E:0005 B409        MOV     AH,09
0D1E:0007 CD21        INT     21         ; AH = 09 : WriteString()
0D1E:0009 B8014C      MOV     AX,4C01
0D1E:000C CD21        INT     21         ; AX = 4C01 : Exit()
```

코드는 매우 간단합니다. 문자열을 출력하고 종료해버리지요.

즉, notepad.exe 는 32 bit 용 PE 파일이지만, MS-DOS 호환 모드를 가지고 있어서
DOS 환경에서 실행하면 "This program cannot be run in DOS mode" 문자열을 출력하고 종료합니다.

이 특성을 잘 이용하면 하나의 실행(EXE) 파일에 DOS 와 Windows 에서 모두 실행 가능한 파일을 만들 수도 있습니다.
실제로 세계적인 보안업체 McAfee 에서 무료로 배포했던 scan.exe 라는 파일이 이와 같은 특징을 가지고 있었습니다.
(DOS 환경에서는 16 bit DOS 용 코드가, Windows 환경에서는 32 bit Windows 코드가 각각 실행됨.)

앞에서 말씀드린대로 DOS Stub 은 **옵션**이기 때문에, 개발 도구에서 지원해 줘야 합니다.
(VB, VC++, Delphi 등은 DOS Stub 을 기본 지원합니다.)

NT header

NT header 구조체 **IMAGE_NT_HEADERS** 입니다.

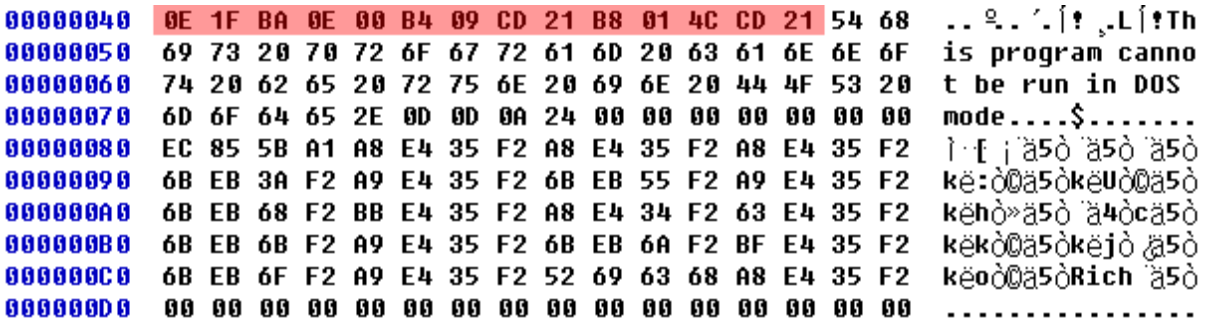
```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;                          // PE Signature : 50450000 ("PE"00)
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

```
* 출처 : Microsoft 의 Visual C++ 에서 제공하는 winnt.h
```

위 구조체는 32 bit 용이며, 64 bit 용은 세번째 멤버가 IMAGE_OPTIONAL_HEADER64 입니다.

IMAGE_NT_HEADER 구조체는 3 개의 멤버로 되어 있는데요,
제일 첫 멤버는 **Signature** 로서 50450000h ("PE"00) 값을 가집니다. (변경불가!)
그리고 **FileHeader** 와 **OptionalHeader** 구조체 멤버가 있습니다.

notepad.exe 의 IMAGE_NT_HEADERS 의 내용을 hex editor 로 살펴보겠습니다.



IMAGE_NT_HEADERS 구조체의 크기는 F8h 입니다. 상당히 큰 구조체 입니다.
FileHeader 와 OptionalHeader 구조체를 하나하나 살펴보겠습니다.

IMAGE_NT_HEADERS - IMAGE_FILE_HEADER

파일의 개략적인 속성을 나타내는 **IMAGE_FILE_HEADER** 구조체 입니다.

```
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;

* 출처 : Microsoft 의 Visual C++ 에서 제공하는 winnt.h
```

IMAGE_FILE_HEADER 구조체에서 아래 4 가지 멤버들이 중요합니다.
(이 값들이 정확히 세팅되어 있지 않으면 파일은 정상적으로 실행되지 않습니다.)

#1. Machine

Machine 넘버는 CPU 별로 고유한 값이며 32 bit Intel 호환 칩은 **14Ch** 의 값을 가집니다.
아래는 winnt.h 파일에 정의된 Machine 넘버의 값들입니다. (일반적인 14Ch 의 값을 기억하면 됩니다.)

```
#define IMAGE_FILE_MACHINE_UNKNOWN           0
#define IMAGE_FILE_MACHINE_I386              0x014c  // Intel 386.
#define IMAGE_FILE_MACHINE_R3000             0x0162  // MIPS little-endian, 0x160 big-endian
#define IMAGE_FILE_MACHINE_R4000             0x0166  // MIPS little-endian
#define IMAGE_FILE_MACHINE_R10000            0x0168  // MIPS little-endian
#define IMAGE_FILE_MACHINE_WCEMIPSV2         0x0169  // MIPS little-endian WCE v2
#define IMAGE_FILE_MACHINE_ALPHA             0x0184  // Alpha_AXP
#define IMAGE_FILE_MACHINE_POWERPC           0x01F0  // IBM PowerPC Little-Endian
#define IMAGE_FILE_MACHINE_SH3               0x01a2  // SH3 little-endian
#define IMAGE_FILE_MACHINE_SH3E              0x01a4  // SH3E little-endian
#define IMAGE_FILE_MACHINE_SH4               0x01a6  // SH4 little-endian
#define IMAGE_FILE_MACHINE_ARM               0x01c0  // ARM Little-Endian
#define IMAGE_FILE_MACHINE_THUMB             0x01c2
#define IMAGE_FILE_MACHINE_IA64              0x0200  // Intel 64
#define IMAGE_FILE_MACHINE_MIPS16            0x0266  // MIPS
#define IMAGE_FILE_MACHINE_MIPSFPU           0x0366  // MIPS
#define IMAGE_FILE_MACHINE_MIPSFPU16         0x0466  // MIPS
#define IMAGE_FILE_MACHINE_ALPHA64           0x0284  // ALPHA64
#define IMAGE_FILE_MACHINE_AXP64             IMAGE_FILE_MACHINE_ALPHA64
```

#2. NumberOfSections

PE 파일은 코드, 데이타, 리소스 등이 각각의 섹션에 나뉘어서 저장된다고 설명드렸습니다.
NumberOfSections 는 바로 그 섹션의 갯수를 나타냅니다.

이 값은 반드시 0 보다 커야 합니다.

정의된 섹션 갯수보다 실제 섹션이 적다면 실행 에러가 발생하며,
정의된 섹션 갯수보다 실제 섹션이 많다면 **정의된 갯수만큼만 인식**됩니다.

#3. SizeOfOptionalHeader

IMAGE_NT_HEADERS 구조체의 마지막 멤버는 IMAGE_OPTIONAL_HEADER32 구조체입니다.
SizeOfOptionalHeader 멤버는 바로 이 IMAGE_OPTIONAL_HEADER32 구조체의 크기를 나타냅니다.

IMAGE_OPTIONAL_HEADER32 는 C 언어의 구조체이기 때문에 이미 그 크기가 결정되어 있습니다.
그런데 Windows 의 PE Loader 는 IMAGE_FILE_HEADER 의 SizeOfOptionalHeader 값을 보고
IMAGE_OPTIONAL_HEADER32 구조체의 크기를 인식합니다.

IMAGE_DOS_HEADER 의 **e_lfanew** 멤버와 IMAGE_FILE_HEADER 의 **SizeOfOptionalHeader** 멤버 때문에
일반적인(상식적인) PE 파일 형식을 벗어나는 일명 **'꽈배기' PE 파일(PE Patch)** 이 만들 수 있습니다.
(나중에 PE Patch 에 대해서 상세히 설명하도록 하겠습니다.)

#4. Characteristics

파일의 속성을 나타내는 값으로써, 실행이 가능한 형태인지(executable or not)
혹은 DLL 파일인지 등의 정보들이 bit OR 형식으로 조합됩니다.

아래는 winnt.h 파일에 정의된 Characteristics 값들입니다. (0002h 와 2000h 의 값을 기억해 두세요.)

```
#define IMAGE_FILE_RELOCS_STRIPPED           0x0001  // Relocation info stripped from file.
#define IMAGE_FILE_EXECUTABLE_IMAGE          0x0002  // File is executable
                                                     // (i.e. no unresolved externel references).
#define IMAGE_FILE_LINE_NUMS_STRIPPED        0x0004  // Line nunbers stripped from file.
#define IMAGE_FILE_LOCAL_SYMS_STRIPPED       0x0008  // Local symbols stripped from file.
#define IMAGE_FILE_AGGRESIVE_WS_TRIM         0x0010  // Agressively trim working set
#define IMAGE_FILE_LARGE_ADDRESS_AWARE       0x0020  // App can handle >2gb addresses
#define IMAGE_FILE_BYTES_REVERSED_LO         0x0080  // Bytes of machine word are reversed.
#define IMAGE_FILE_32BIT_MACHINE             0x0100  // 32 bit word machine.
#define IMAGE_FILE_DEBUG_STRIPPED            0x0200  // Debugging info stripped from
                                                     // file in .DBG file
#define IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP   0x0400  // If Image is on removable media,
                                                     // copy and run from the swap file.
#define IMAGE_FILE_NET_RUN_FROM_SWAP         0x0800  // If Image is on Net,
                                                     // copy and run from the swap file.
#define IMAGE_FILE_SYSTEM                    0x1000  // System File.
#define IMAGE_FILE_DLL                       0x2000  // File is a DLL.
#define IMAGE_FILE_UP_SYSTEM_ONLY            0x4000  // File should only be run on a UP machine
#define IMAGE_FILE_BYTES_REVERSED_HI         0x8000  // Bytes of machine word are reversed.
```

참고로 PE 파일중에 Characteristics 값에 0002h 가 없는 경우(not executable)가 있을까요?
네, 있습니다. 예를 들어 *.obj 와 같은 object 파일이 그런 경우이고, resource DLL 같은 파일이 그런 경우입니다.

이 정도면 IMAGE_FILE_HEADER 의 구조를 이해하는데 부족함이 없을 것입니다.

마지막으로 IMAGE_FILE_HAEDER 의 **TimeDateStamp** 멤버에 대해서 설명드리겠습니다.
이 값은 파일의 실행에 영향을 미치지 않는 값으로써 해당 파일의 빌드 시간을 나타낸 값입니다.
단, 개발 도구에 따라서 이 값을 세팅해주는 도구(VB, VC++)가 있고, 그렇지 않은 도구(Delphi)가 있습니다.
(또한 개발 도구의 옵션에 따라서 달라질 수 있습니다.)

이제 실제로 notepad.exe 의 **IMAGE_FILE_HEADER** 를 확인해 보겠습니다.

```
000000E0  50 45 00 00 4C 01 03 00 87 52 02 48 00 00 00 00  PE..L....‡R.H....
000000F0  00 00 00 00 E0 00 0F 01 0B 01 07 0A 00 78 00 00  ....à........x..
```

위 그림은 hex editor 로 봤을때의 그림이고, 이를 알아보기 쉽게 구조체 멤버로 표현하면 아래와 같습니다.

[IMAGE_FILE_HEADER] - notepad.exe

```
 offset    value    description
-------------------------------------------------------------------------------
000000E4      014C machine
000000E6      0003 number of sections
```

```
000000E8 48025287 time date stamp (Mon Apr 14 03:35:51 2008)
000000EC 00000000 offset to symbol table
000000F0 00000000 number of symbols
000000F4     00E0 size of optional header
000000F6     010F characteristics
                     IMAGE_FILE_RELOCS_STRIPPED
                   IMAGE_FILE_EXECUTABLE_IMAGE
                  IMAGE_FILE_LINE_NUMS_STRIPPED
                  IMAGE_FILE_LOCAL_SYMS_STRIPPED
                  IMAGE_FILE_32BIT_MACHINE
```

IMAGE_NT_HEADERS - IMAGE_OPTIONAL_HEADER32

PE header 구조체 중에서 가장 크기가 큰 **IMAGE_OPTIONAL_HEADER32** 입니다.
(64 bit PE 파일의 경우 IMAGE_OPTIONAL_HEADER64 구조체를 사용합니다.)

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

#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16

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   SizeOfImage;
    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_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
```

* 출처 : Microsoft 의 Visual C++ 에서 제공하는 winnt.h

IMAGE_OPTIONAL_HEADER32 구조체에서 주목해야 할 멤버들은 아래와 같습니다.
이 값들 역시 파일 실행에 필수적인 값들이라서 잘 못 세팅되면 파일이 정상 실행 되지 않습니다.

#1. Magic

IMAGE_OPTIONAL_HEADER32 인 경우 **10Bh**, IMAGE_OPTIONAL_HEADER64 인 경우 **20Bh** 값을 가지게 됩니다.

#2. AddressOfEntryPoint

EP(Entry Point) 의 RVA(Relative Virtual Address) 값을 가지고 있습니다.

#3. ImageBase

프로세스의 가상 메모리는 0 ~ FFFFFFFFh 범위입니다. (32 bit 의 경우)
ImageBase 는 이렇게 광활한 메모리내에서 PE 파일이 로딩(매핑)되는 시작 주소를 나타냅니다.

EXE, DLL 파일은 user memory 영역인 0 ~ 7FFFFFFFh 범위에 위치하고,
SYS 파일은 kernel memory 영역인 80000000h ~ FFFFFFFFh 범위에 위치합니다.

일반적으로 개발 도구(VB/VC++/Delphi)들이 만들어내는 **EXE** 파일의 ImageBase 값은 **00400000h** 이고,
DLL 파일의 ImageBase 값은 **01000000h** 입니다. (물론 다른 값도 가능합니다.)

PE loader 는 PE 파일을 실행시키기 위해 프로세스를 생성하고 파일을 메모리에 로딩(매핑) 시킨 후
EIP 레지스터 값을 ImageBase + AddressOfEntryPoint 값으로 세팅합니다.

#4. SectionAlignment, FileAlignment

PE 파일은 섹션으로 나뉘어져 있는데 파일에서 섹션의 최소단위를 나타내는 것이 FileAlignment 이고
메모리에서 섹션의 최소단위를 나타내는 것이 SectionAlignment 입니다.
(하나의 파일에서 FileAlignment 와 SectionAlignment 의 값은 같을 수도 있고 틀릴 수도 있습니다.)

따라서 파일/메모리의 섹션 크기는 반드시 각각 FileAlignment/SectionAlignment 의 배수가 되어야 합니다.

#5. SizeOfImage

PE 파일이 메모리에 로딩되었을 때 가상 메모리에서 PE Image 가 차지하는 크기를 나타냅니다.
일반적으로 파일의 크기와 메모리에 로딩된 크기는 다릅니다.

(각 섹션의 로딩 위치와 메모리 점유 크기는 나중에 소개할 **Section Header** 에 정의 되어 있습니다.)

#6. SizeOfHeader

PE header 의 전체 크기를 나타냅니다.
이 값 역시 FileAlignment 의 배수 이어야 합니다.

파일 시작에서 SizeOfHeader 옵셋만큼 떨어진 위치에 첫번째 섹션이 위치합니다.

#7. Subsystem

1 : Driver file (*.sys)
2 : GUI (Graphic User Interface) 파일 -> notepad.exe 와 같은 윈도우 기반 어플리케이션
3 : CUI (Console User Interface) 파일 -> cmd.exe 와 같은 콘솔 기반 어플리케이션

#8. NumberOfRvaAndSizes

마지막 멤버인 DataDirectory 배열의 갯수

구조체 정의에 분명히 배열 갯수가 IMAGE_NUMBEROF_DIRECTORY_ENTRIES (16) 이라고 명시 되어 있지만, PE loader 는 NumberOfRvaAndSizes 의 값을 보고 배열의 크기를 인식합니다.

#9. DataDirectory

IMAGE_DATA_DIRECTORY 구조체의 배열로써, 배열의 각 항목마다 정의된 값을 가지게 됩니다.

아래에 각 배열 항목을 나열하였습니다.

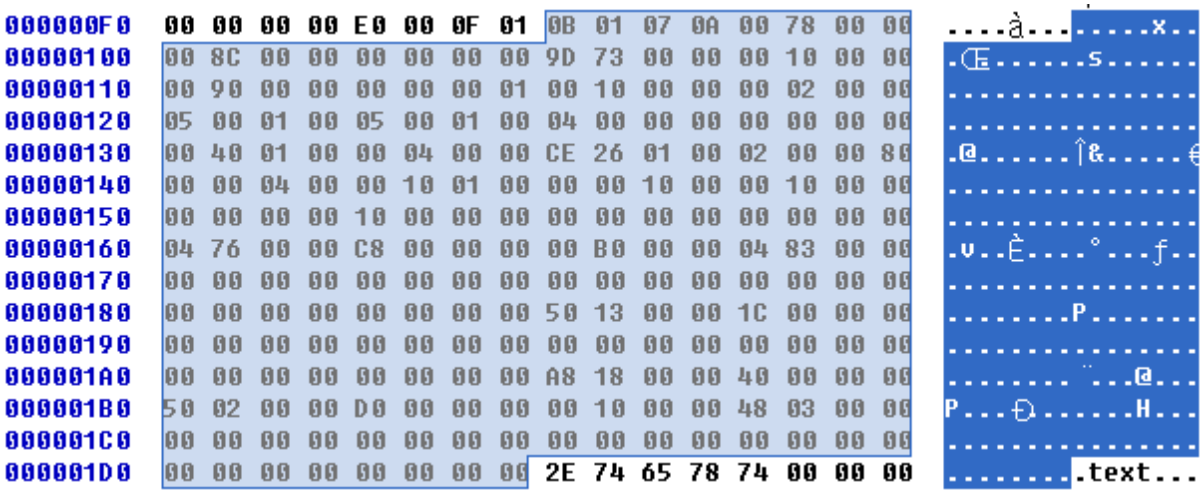
```
DataDirectory[0] = EXPORT Directory
DataDirectory[1] = IMPORT Directory
DataDirectory[2] = RESOURCE Directory
DataDirectory[3] = EXCEPTION Directory
DataDirectory[4] = SECURITY Directory
DataDirectory[5] = BASERELOC Directory
DataDirectory[6] = DEBUG Directory
DataDirectory[7] = COPYRIGHT Directory
DataDirectory[8] = GLOBALPTR Directory
DataDirectory[9] = TLS Directory
DataDirectory[A] = LOAD_CONFIG Directory
DataDirectory[B] = BOUND_IMPORT Directory
DataDirectory[C] = IAT Directory
DataDirectory[D] = DELAY_IMPORT Directory
DataDirectory[E] = COM_DESCRIPTOR Directory
DataDirectory[F] = Reserved Directory
```

여기서 말하는 Directory 란 그냥 어떤 구조체의 배열이라고 생각하시면 됩니다.

빨간색으로 표시한 EXPORT, IMPORT, RESOURCE, TLS Directory 를 눈여겨 보시기 바랍니다.
특히 IMPORT 와 EXPORT Directory 구조는 PE header 에서 매우 중요하기 때문에 나중에 따로 설명하도록 하겠습니다.

나머지는 크게 중요하지 않다고 보시면 됩니다.

이제 실제로 notepad.exe 의 **IMAGE_OPTIONAL_HEADER32** 를 확인해 보겠습니다.



구조체 멤버별 값과 설명은 아래와 같습니다.

[IMAGE_OPTIONAL_HEADER] - notepad.exe

offset	value	description
000000F8	010B	magic
000000FA	07	major linker version
000000FB	0A	minor linker version
000000FC	00007800	size of code
00000100	00008C00	size of initialized data
00000104	00000000	size of uninitialized data
00000108	0000739D	address of entry point
0000010C	00001000	base of code
00000110	00009000	base of data
00000114	01000000	image base
00000118	00001000	section alignment
0000011C	00000200	file alignment
00000120	0005	major OS version
00000122	0001	minor OS version
00000124	0005	major image version
00000126	0001	minor image version
00000128	0004	major subsystem version
0000012A	0000	minor subsystem version
0000012C	00000000	win32 version value
00000130	00014000	size of image
00000134	00000400	size of headers
00000138	000126CE	checksum
0000013C	0002	subsystem
0000013E	8000	DLL characteristics
00000140	00040000	size of stack reserve
00000144	00011000	size of stack commit
00000148	00100000	size of heap reserve
0000014C	00001000	size of heap commit

```
00000150 00000000 loader flags
00000154 00000010 number of directories
00000158 00000000 RVA  of EXPORT Directory
0000015C 00000000 size of EXPORT Directory
00000160 00007604 RVA  of IMPORT Directory
00000164 000000C8 size of IMPORT Directory
00000168 0000B000 RVA  of RESOURCE Directory
0000016C 00008304 size of RESOURCE Directory
00000170 00000000 RVA  of EXCEPTION Directory
00000174 00000000 size of EXCEPTION Directory
00000178 00000000 RVA  of SECURITY Directory
0000017C 00000000 size of SECURITY Directory
00000180 00000000 RVA  of BASERELOC Directory
00000184 00000000 size of BASERELOC Directory
00000188 00001350 RVA  of DEBUG Directory
0000018C 0000001C size of DEBUG Directory
00000190 00000000 RVA  of COPYRIGHT Directory
00000194 00000000 size of COPYRIGHT Directory
00000198 00000000 RVA  of GLOBALPTR Directory
0000019C 00000000 size of GLOBALPTR Directory
000001A0 00000000 RVA  of TLS Directory
000001A4 00000000 size of TLS Directory
000001A8 000018A8 RVA  of LOAD_CONFIG Directory
000001AC 00000040 size of LOAD_CONFIG Directory
000001B0 00000250 RVA  of BOUND_IMPORT Directory
000001B4 000000D0 size of BOUND_IMPORT Directory
000001B8 00001000 RVA  of IAT Directory
000001BC 00000348 size of IAT Directory
000001C0 00000000 RVA  of DELAY_IMPORT Directory
000001C4 00000000 size of DELAY_IMPORT Directory
000001C8 00000000 RVA  of COM_DESCRIPTOR Directory
000001CC 00000000 size of COM_DESCRIPTOR Directory
000001D0 00000000 RVA  of Reserved Directory
000001D4 00000000 size of Reserved Directory
```

Section Header

각 Section 의 **속성(property)**을 정의한 것이 Section Header 입니다.

section header 구조체를 보기 전에 한번 **생각**을 해보겠습니다.

앞서 PE 파일은 code, data, resource 등을 각각의 section 으로 나눠서 저장한다고 설명드렸습니다.
분명 PE 파일 포멧을 설계한 사람들은 어떤 **장점**이 있기 때문에 그랬을 겁니다.

PE 파일을 여러개의 section 구조로 만들었을때 (제가 생각하는) 장점은 바로 프로그램의 **안정성**입니다.

code 와 data 가 하나의 섹션으로 되어 있고 서로 뒤죽박죽 섞여 있다면, (실제로 구현이 가능하긴 합니다.)
그 복잡한은 무시하고라도 안정성에 문제가 생길 수 있습니다.

가령 문자열 data 에 값을 쓰다가 어떤 이유로 overflow 가 발생(버퍼 크기를 초과해서 입력) 했을때

바로 다음의 code (명령어) 를 그대로 덮어써버릴 것입니다. 프로그램은 그대로 뻗어 버리겠죠.

즉, code/data/resource 마다 각각의 성격(특징, 엑세스 권한)이 틀리다는 것을 알게 된 것입니다.

- code - 실행, 읽기 권한
- data - 비실행, 읽기, 쓰기 권한
- resource - 비실행, 읽기 권한

그래서 PE 파일 포멧 설계자들은 비슷한 성격의 자료를 section 이라고 이름 붙인 곳에 모아두기로 결정하였고,
각각의 section 의 속성을 기술할 section header 가 필요하게 된 것입니다.
(section 의 속성에는 file/memory 에서의 시작위치, 크기, 엑세스 권한 등이 있어야 겠지요.)

이제 section header 가 무슨 역할을 하는지 이해 되셨나요?

IMAGE_SECTION_HEADER

---

section header 는 각 section 별 **IMAGE_SECTION_HEADER 구조체의 배열**로 되어있습니다.

```
#define IMAGE_SIZEOF_SHORT_NAME              8

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;

* 출처 : Microsoft 의 Visual C++ 에서 제공하는 winnt.h
```

IMAGE_SECTION_HEADER 구조체에서 알아야 할 중요 멤버는 아래와 같습니다. (나머지는 사용되지 않습니다.)

- VirtualSize : 메모리에서 섹션이 차지하는 크기

- `VirtualAddress` : 메모리에서 섹션의 시작 주소 (RVA)
- `SizeOfRawData` : 파일에서 섹션이 차지하는 크기
- `PointerToRawData` : 파일에서 섹션의 시작 위치
- `Characteristics` : 섹션의 특징 (bit OR)

VirtualAddress 와 PointerToRawData 의 값은 아무 값이나 가질 수 없고,
각각 (IMAGE_OPTIONAL_HEADER32 에 정의된) SectionAlignment 와 FileAlignment 에 맞게 결정됩니다.

VirtualSize 와 SizeOfRawData 는 일반적으로 서로 틀린값을 가집니다.
즉, 파일에서의 섹션 크기와 메모리에 로딩된 섹션의 크기는 틀리다는 얘기가 되는 거죠.

Characteristics 는 아래 값들의 조합(bit OR)으로 이루어 집니다.

```
#define IMAGE_SCN_CNT_CODE                   0x00000020  // Section contains code.
#define IMAGE_SCN_CNT_INITIALIZED_DATA       0x00000040  // Section contains initialized data.
#define IMAGE_SCN_CNT_UNINITIALIZED_DATA     0x00000080  // Section contains uninitialized data.
#define IMAGE_SCN_MEM_EXECUTE                0x20000000  // Section is executable.
#define IMAGE_SCN_MEM_READ                   0x40000000  // Section is readable.
#define IMAGE_SCN_MEM_WRITE                  0x80000000  // Section is writeable.
```

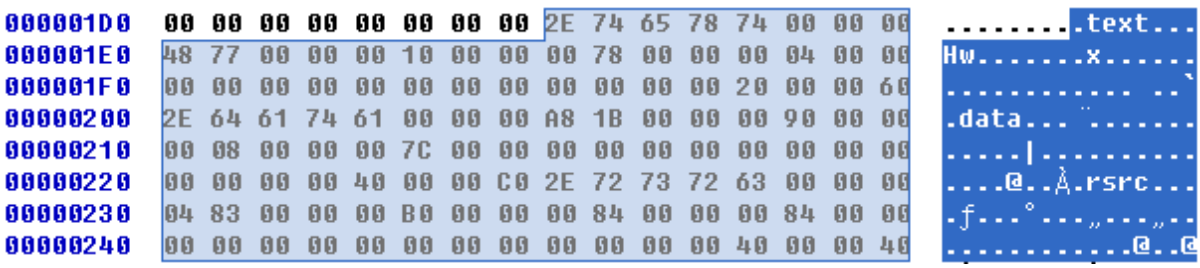
마지막으로 **Name** 항목에 대해서 얘기해보겠습니다.

Name 멤버는 C 언어의 문자열처럼 NULL 로 끝나지 않습니다. 또한 ASCII 값만 와야한다는 제한도 없습니다.
PE 스펙에는 섹션 Name 에 대한 어떠한 명시적인 규칙이 없기 때문에 어떠한 값을 넣어도 되고 심지어 NULL 로 채워도 됩니다.

또한 개발 도구에 따라서 섹션 이름/갯수 등이 달라집니다.

따라서 섹션의 Name 은 그냥 참고용 일뿐 어떤 정보로써 활용하기에는 100% 장담할 수 없습니다.
(데이타 섹션 이름을 ".code" 로 해도 되거든요.)

자 그러면 실제 notepad.exe 의 Section Header 배열을 살펴보죠. (총 3 개의 섹션이 있습니다.)



구조체 멤버별로 살펴보면 아래와 같습니다.

[IMAGE_SECTION_HEADER]

```
 offset   value   description
-------------------------------------------------------------------------------
000001D8 2E746578 Name (.text)
000001DC 74000000
000001E0 00007748 virtual size
000001E4 00001000 RVA
000001E8 00007800 size of raw data
000001EC 00000400 offset to raw data
000001F0 00000000 offset to relocations
000001F4 00000000 offset to line numbers
000001F8     0000 number of relocations
000001FA     0000 number of line numbers
000001FC 60000020 characteristics
                    IMAGE_SCN_CNT_CODE
                    IMAGE_SCN_MEM_EXECUTE
                    IMAGE_SCN_MEM_READ


00000200 2E646174 Name (.data)
00000204 61000000
00000208 00001BA8 virtual size
0000020C 00009000 RVA
00000210 00000800 size of raw data
00000214 00007C00 offset to raw data
00000218 00000000 offset to relocations
0000021C 00000000 offset to line numbers
00000220     0000 number of relocations
00000222     0000 number of line numbers
00000224 C0000040 characteristics
                    IMAGE_SCN_CNT_INITIALIZED_DATA
                    IMAGE_SCN_MEM_READ
                    IMAGE_SCN_MEM_WRITE


00000228 2E727372 Name (.rsrc)
0000022C 63000000
00000230 00008304 virtual size
00000234 0000B000 RVA
00000238 00008400 size of raw data
0000023C 00008400 offset to raw data
00000240 00000000 offset to relocations
00000244 00000000 offset to line numbers
00000248     0000 number of relocations
0000024A     0000 number of line numbers
0000024C 40000040 characteristics
                            IMAGE_SCN_CNT_INITIALIZED_DATA
                            IMAGE_SCN_MEM_READ
```

RVA to RAW

Section Header 를 잘 이해하셨다면 이제부터는 PE 파일이 메모리에 로딩되었을때
각 섹션에서 메모리의 주소(RVA)와 파일 옵셋을 잘 **매핑**할 수 있어야 합니다.

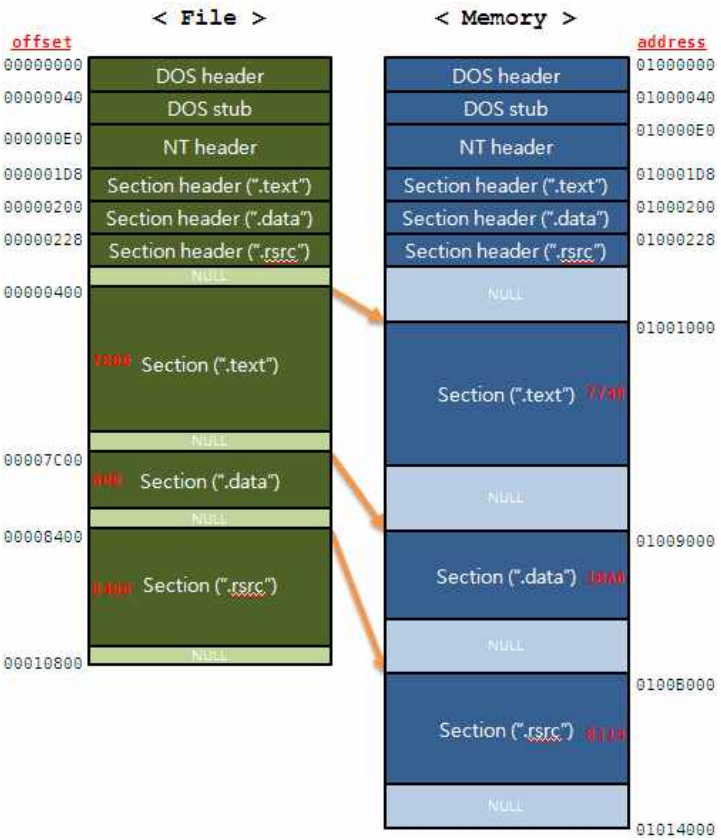
이러한 매핑을 일반적으로 "RVA to RAW" 라고 부릅니다.
방법은 아래와 같습니다.

1) RVA 가 속해 있는 섹션을 찾습니다.
2) 간단한 **비례식**을 사용해서 파일 옵셋(RAW)을 계산합니다.

IMAGE_SECTION_HEADER 구조체에 의하면 **비례식**은 이렇습니다.

```
RAW - PointerToRawData = RVA - VirtualAddress
                  RAW = RVA - VirtualAddress + PointerToRawData
```

간단한 퀴즈를 내보겠습니다.
아래 그림은 notepad.exe 의 File 과 Memory 에서의 모습입니다.
각각 RVA 를 계산해 보세요. (계산기 calc.exe 를 Hex 모드로 세팅하시면 계산이 편합니다.)



Q1) RVA = 5000h 일때 File Offset = ?

A1) 먼저 해당 RVA 값이 속해 있는 섹션을 찾아야 합니다.

=> RVA 5000h 는 첫번째 섹션(".text")에 속해있습니다. (ImageBase 01000000h 를 고려하세요.)

비례식 사용

=> **RAW** = 5000h(RVA) - 1000h(VirtualAddress) + 400h(PointerToRawData) = **4400h**

Q2) RVA = 13314h 일때 File Offset = ?

A2) 해당 RVA 값이 속해 있는 섹션을 찾습니다.

=> 세번째 섹션(".rsrc")에 속해있습니다.

비례식 사용

=> **RAW** = 13314h(RVA) - B000h(VA) + 8400h(PointerToRawData) = **10714h**

Q3) RVA = ABA8h 일때 File Offset = ?

A2) 해당 RVA 값이 속해 있는 섹션을 찾습니다.

=> 두번째 섹션(".data")에 속해있습니다.

비례식 사용

=> **RAW** = ABA8h(RVA) - 9000h(VA) + 7C00h(PointerToRawData) = ~~**97A8h**~~ **(X)**

=> 계산 결과로 RAW = 97A8h 가 나왔지만 이 옵셋은 세번째 섹션(".rsrc")에 속해 있습니다.
RVA 는 두번째 섹션이고, RAW 는 세번째 섹션이라면 말이 안되지요.
이 경우에 **"해당 RVA(ABA8h)에 대한 RAW 값은 정의할 수 없다"** 라고 해야 합니다.
이런 이상한 결과가 나온 이유는 위 경우에 두번째 섹션의 VirtualSize 값이 SizeOfRawData 값 보다 크기 때문입니다.

PE 파일의 섹션에는 Q3) 의 경우와 같이 VirtualSize 와 SizeOfRawData 값이 서로 틀려서 벌어지는 이상하고 재미있는(?) 일들이 많이 있습니다. (앞으로 살펴보게 될 것입니다.)

IAT (Import Address Table)

PE Header 를 처음 배울때 최대 장벽은 **IAT(Import Address Table)** 입니다.

IAT 에는 Windows 운영체제의 **핵심 개념**인 process, memory, DLL 구조 등에 대한 내용이 **함축**되어 있습니다.
즉, IAT 만 잘 이해해도 Windows 운영체제의 근간을 이해한다고 할 수 있습니다.

IAT 란 쉽게 말해서 **프로그램이 어떤 라이브러리에서 어떤 함수를 사용하고 있는지를 기술한 테이블** 입니다.

DLL (Dynamic Linked Library)

IAT 를 설명하기 앞서 Windows OS 의 근간을 이루는 **DLL(Dynamic Linked Library)** 개념을 짚고 넘어가야 합니다.
(뭐든지 이유를 알면 이해하기 쉬운 법이지요...)

DLL 을 우리말로 '동적 연결 라이브러리' 라고 하는데요, 그 이유를 알아 보겠습니다.

16 bit DOS 시절에는 DLL 개념이 없었습니다. 그냥 'Library' 만 존재하였습니다.

예를 들면 C 언어에서 printf() 함수를 사용할 때 컴파일러는 C 라이브러리에서
해당 함수의 binary 코드를 그대로 가져와서 프로그램에 삽입(포함)시켜 버렸습니다.
즉, 실행 파일내에 printf() 함수의 바이너리 코드를 가지고 있는 것입니다.

Windows OS 에서는 **Multi-Tasking** 을 지원하기 때문에 이러한 라이브러리 포함 방식이 **비효율적**이 되어 버렸습니다.

32 bit Windows 환경을 제대로 지원하기 위해 기본적으로 매우 많은 라이브러리 함수(process, memory, window, message, etc)를 사용해야 합니다.

여러 프로그램이 동시에 실행되야 하는 상황에서 모든 프로그램마다 위와 같이 동일한 라이브러리가 포함되어서 실행된다면
심각한 **메모리 낭비**를 불러오게 됩니다. (물론 디스크 공간의 낭비도 무시할 수 없지요.)

그래서 Windows OS 설계자들은 (필요에 의해) 아래와 같은 DLL 개념을 고안해 내었습니다.

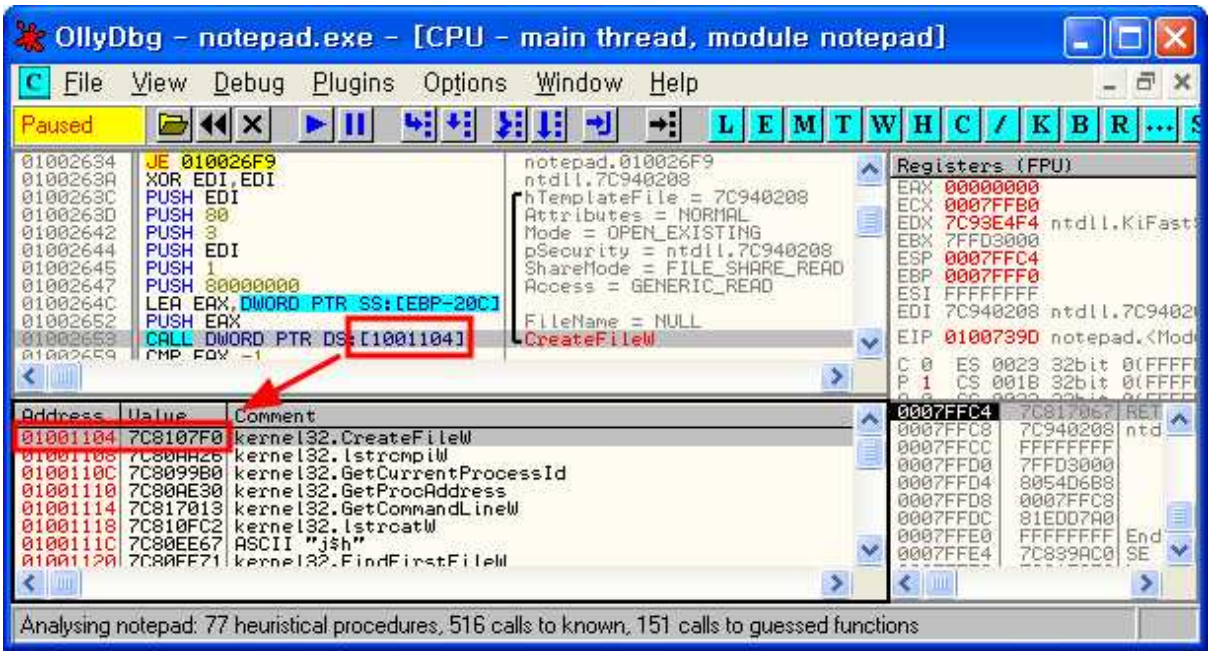
"프로그램내에 라이브러리를 포함시키지 말고 별도의 파일(DLL)로 구성하여 필요할 때마다 불러쓰자."
"일단 한번 로딩된 DLL 의 코드, 리소스는 Memory Mapping 기술로 여러 Process 에서 공유해 쓰자."
"라이브러리가 업데이트 되었을때 해당 DLL 파일만 교체하면 되니 쉽고 편해서 좋다."

실제 DLL 로딩 방식은 2 가지 입니다.
프로그램내에서 사용되는 순간에 로딩하고 사용이 끝나면 메모리에서 해제 시키는 방법(Explicit Linking)과 프로그램 시작할 때 같이 로딩되어 프로그램 종료 할 때 메모리에서 해제되는 방법(Implicit Linking)이 있습니다.

IAT 는 바로 Implicit Linking 에 대한 매카니즘을 제공하는 역할을 합니다.

IAT 의 확인을 위해 OllyDbg 로 notepad.exe 를 열어보겠습니다.

아래 그림은 kernel32.dll 의 CreateFileW 를 호출하는 코드입니다.



CreateFileW 를 호출할 때 직접 호출하지 않고 01001104 주소에 있는 값을 가져와서 호출합니다.
(모든 API 호출은 이런 방식으로 되어 있습니다.)

01001104 주소는 notepad.exe 의 ".text" 섹션 메모리 영역입니다. (더 정확히는 IAT 메모리 영역입니다.)
01001104 주소의 값은 7C8107F0 이며,
7C8107F0 주소가 바로 notepad.exe 프로세스 메모리에 로딩된 kernel32.dll 의 CreateFileW 함수 주소입니다.

여기서 한가지 의문이 생깁니다.
"그냥 **CALL 7C8107F0** 이라고 하면 더 편하고 좋지 않나요?"
컴파일러가 CALL 7C8107F0 이라고 정확히 써줬다면 더 좋지 않냐는 의문이 들 수 있습니다만,
그건 바로 위에서 설명 드렸던 DOS 시절의 방식입니다.

notepad.exe 제작자가 프로그램을 컴파일(생성)하는 순간에는 이 notepad.exe 프로그램이
어떤 Windows(9X, 2K, XP, Vista, etc), 어떤 언어(KOR, ENG, JPN, etc), 어떤 Service Pack 에서
실행 될 지 도저히 알 수 없습니다.

위에서 열거한 모든 환경에서 kernel32.dll 의 버전이 틀려지고, CreateFileW 함수의 위치(주소)가
틀려집니다.
모든 환경에서 CreateFileW 함수 호출을 보장하기 위해서 컴파일러는 CreateFileW 의 실제 주소가 저장될
위치(01001104)를

준비하고 CALL DWORD PTR DS:[1001104] 형식의 명령어를 적어두기만 합니다.

파일이 실행되는 순간 PE Loader 가 01001104 의 위치에 CreateFileW 의 주소를 입력해줍니다.

또 다른 이유는 **DLL Relocation** 때문입니다.
일반적인 DLL 파일의 ImageBase 값은 10000000h 입니다.

예를 들어 어떤 프로그램이 a.dll 과 b.dll 을 사용한다고 했을때,
PE Loader 는 먼저 a.dll 을 ImageBase 값인 메모리 10000000h 에 잘 로딩합니다.
그 다음 b.dll 을 ImageBase 값인 메모리 10000000h 에 로딩하려고 봤더니, 이미 그 주소는 a.dll 이 사용하고 있었습니다.
그래서 PE Loader 는 다른 비어있는 메모리 공간(ex:3E000000h) 을 찾아서 b.dll 을 로딩시켜 줍니다.

이것이 DLL Relocation 이며 실제 주소를 하드코딩 할 수 없는 이유입니다.
또한 PE Header 에서 주소를 나타낼때 VA 를 쓰지 못하고 RVA 를 쓰는 이유이기도 합니다.

* DLL 은 PE Header 에 명시된 ImageBase 에 로딩된다고 보장할 수 없습니다.
  반면에 process 생성 주체가 되는 EXE 파일은 자신의 ImageBase 에 정확히 로딩되지요.
  (자신만의 가상 메모리 공간을 가지기 때문입니다.)

이것은 매우 중요한 설명입니다. 다시 한번 잘 읽어보시기 바랍니다.

이제 IAT 의 역할을 이해할 수 있으실 겁니다.
(아래에서 설명드릴 IAT 구조가 왜 이리 복잡해야 하는지에 대해서도 약간 이해가 되실 겁니다.)

IMAGE_IMPORT_DESCRIPTOR

PE 파일은 자신이 어떤 라이브러리를 Import 하고 있는지 **IMAGE_IMPORT_DESCRIPTOR** 구조체에 명시하고 있습니다.

* Import : library 한테서 서비스(함수)를 제공 받는 일
* Export : library 입장에서 다른 PE 파일에게 서비스(함수)를 제공 하는 일

IMAGE_IMPORT_DESCRIPTOR 구조체는 아래와 같습니다.

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;
        DWORD   OriginalFirstThunk;        // INT(Import Name Table) address (RVA)
    };
    DWORD   TimeDateStamp;
    DWORD   ForwarderChain;
```

```
    DWORD   Name;                           // library name string address (RVA)
    DWORD   FirstThunk;                     // IAT(Import Address Table) address
(RVA)
} IMAGE_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_IMPORT_BY_NAME {
WORD    Hint;                           // ordinal
    BYTE    Name[1];                        // function name string
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

* 출처 : Microsoft 의 Visual C++ 에서 제공하는 winnt.h

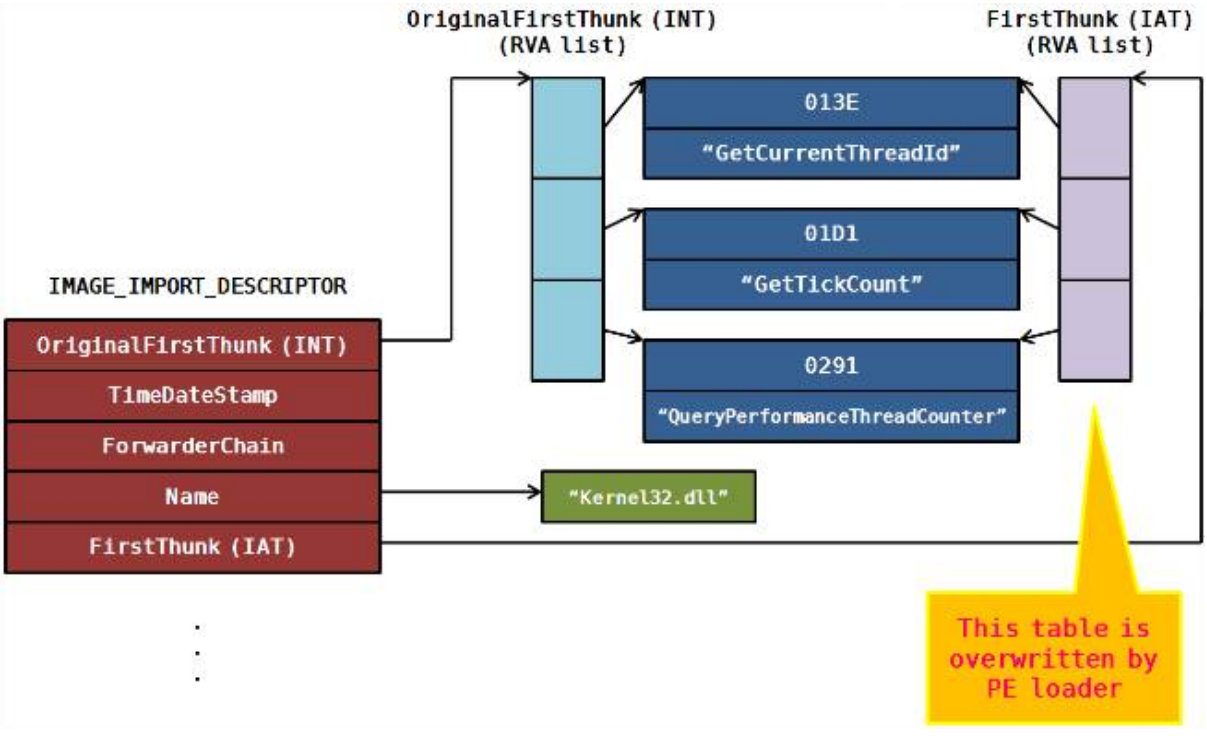
일반적인 프로그램에서는 여러 개의 Library 를 Import 하기 때문에
Library 의 갯수 만큼 위 구조체의 배열 형식으로 존재하게 되며, 구조체 배열의 마지막은 NULL 구조체로 끝나게 됩니다.

IMAGE_IMPORT_DESCRIPTOR 구조체에서 중요한 멤버는 아래와 같습니다. (전부 RVA 값을 가집니다.)

- OriginalFirstThunk : INT(Import Name Table) 의 주소(RVA)
- Name : Library 이름 문자열의 주소(RVA)
- FirstThunk : IAT(Import Address Table) 의 주소(RVA)

* PE Header 에서 'Table' 이라고 하면 '배열' 을 뜻합니다.
* INT 와 IAT 는 long type (4 byte 자료형) 배열이고 NULL 로 끝납니다. (크기가 따로 명시되어 있지 않습니다.)
* INT 에서 각 원소의 값은 IMAGE_IMPORT_BY_NAME 구조체 주소값을 가지고 있습니다.
  (IAT 도 같은 값을 가지는 경우가 있습니다.)
* INT 와 IAT 의 크기는 같아야 합니다.

아래 그림은 notepad.exe 의 kernel32.dll 에 대한 IMAGE_IMPORT_DESCRIPTOR 구조를 표시하고 있습니다.



<Fig. IAT 구조>

PE Loader 가 Import 함수 주소를 IAT 에 입력하는 기본적인 **순서**를 설명드리겠습니다.

1. IID 의 Name 멤버를 읽어서 라이브러리의 이름 문자열("kernel32.dll")을 얻습니다.
2. 해당 라이브러리("kernel32.dll")를 로딩합니다.
3. IID 의 OriginalFirstThunk 멤버를 읽어서 INT 주소를 얻습니다.
4. INT 에서 배열의 값을 하나씩 읽어 해당 IMAGE_IMPORT_BY_NAME 주소(RVA)를 얻습니다.
5. IMAGE_IMPORT_BY_NAME 의 Hint(ordinal) 또는 Name 항목을 이용하여 해당 함수("GetCurrentThreadId")의 시작 주소를 얻습니다.
6. IID 의 FirstThunk(IAT) 멤버를 읽어서 IAT 주소를 얻습니다.
7. 해당 IAT 배열 값에 위에서 구한 함수 주소를 입력합니다.
8. INT 가 끝날때까지 (NULL 을 만날때까지) 위 4 ~ 7 과정을 반복합니다.

위 그림에서는 INT 와 IAT 의 각 원소가 동시에 같은 주소를 가리키고 있지만 그렇지 않은 경우도 많습니다.
(변칙적인 PE 파일에 대해서는 향후 많은 파일을 접해보면서 하나씩 배워 나가야 합니다.)

notepad.exe 를 이용한 실습

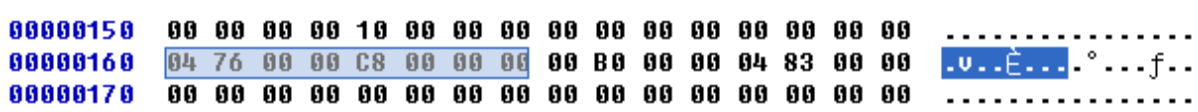
실제로 **notepad.exe** 를 대상으로 하나씩 살펴 보겠습니다.

그런데 실제 IMAGE_IMPORT_DESCRIPTOR 구조체 배열은 PE 파일의 어느 곳에 존재할까요?
PE Header 가 아닌 PE Body 에 위치합니다.

그곳을 찾아가기 위한 정보는 역시 PE Header 에 있습니다.
바로 IMAGE_OPTIONAL_HEADER32.DataDirectory[1].VirtualAddress 값이
실제 IMAGE_IMPORT_DESCRIPTOR 구조체 배열의 시작 주소 입니다. (RVA 값입니다.)

IMAGE_IMPORT_DESCRIPTOR 구조체 배열을 다른 용어로는 **IMPORT Directory Table** 이라고도 합니다.
(위 용어를 전부 알아두셔야 남들과 의사소통이 원활해 집니다.)

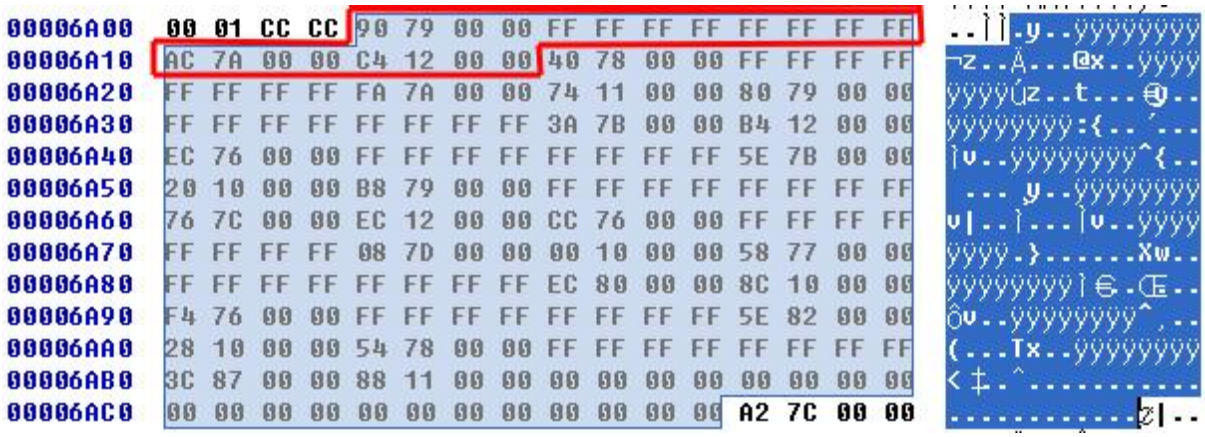
IMAGE_OPTIONAL_HEADER32.DataDirectory[1] 구조체 값은 아래와 같습니다.
(첫번째 4 byte 가 VirtualAddress, 두번째 4 byte 가 Size 멤버입니다.)



```
offset   value   description
-----------------------------------------------
...
00000158 00000000 RVA  of EXPORT Directory
0000015C 00000000 size of EXPORT Directory
00000160 00007604 RVA  of IMPORT Directory
00000164 000000C8 size of IMPORT Directory
00000168 0000B000 RVA  of RESOURCE Directory
0000016C 00008304 size of RESOURCE Directory
...
```

* 위 구조체에 대해 궁금하신 분들께서는 IMAGE_OPTIONAL_HEADER 설명 을 참고하시기 바랍니다.
* DataDirectory 구조체에서 Size 멤버는 중요하지 않습니다. (PE Loader 에서 사용되지 않는 값입니다.)

위 그림에서 보듯이 RVA 가 7604h 이니까 File Offset 은 6A04h 입니다.
파일에서 6A04h 를 보면 아래 그림과 같습니다.



그림에서 파란색으로 표시된 부분이 전부 IMAGE_IMPORT_DESCRIPTOR 구조체 배열이고,
빨간 테두리로 되어 있는 부분은 구조체 배열의 첫번째 원소입니다.
(참고로 배열의 마지막은 NULL 구조체로 되어 있는 것도 확인 할 수 있습니다.)

빨간 테두리의 IMAGE_IMPORT_DESCRIPTOR 구조체를 각 멤버별로 살펴보겠습니다.

```
OriginalFirstThunk (INT) = 7990h (file offset : 6D90h)
TimeDateStamp            = FFFFFFFFh
ForwarderChain           = FFFFFFFFh
Name                     = 7AACh (file offset : 6EACh)
FirstThunk (IAT)         = 12C4h (file offset : 6C4h)
```

우리는 IAT 를 **공부**하는 입장이기 때문에 hex editor 를 이용하여 하나하나 따라가도록 하겠습니다.
(편의를 위해 위 구조체 값(RVA) 를 미리 file offset 으로 변환해 놓았습니다.)

* RVA 를 file offset 으로 변환하는 방법에 대해서는 IMAGE_SECTION_HEADER 설명을 참고하세요.

그럼 순서대로 진행해 볼까요?

1. 라이브러리 이름 (Name)

Name 멤버를 따라가면 쉽게 구할 수 있습니다. (RVA : 7AACh -> file offset : 6EACh)

```
00006E90  70 65 6E 46 69 6C 65 4E 61 6D 65 57 00 00 12 00  penFileNameW....
00006EA0  50 72 69 6E 74 44 6C 67 45 78 57 00 63 6F 6D 64  PrintDlgExW.comd
00006EB0  6C 67 33 32 2E 64 6C 6C 00 00 03 01 53 68 65 6C  lg32.dll....Shel
00006EC0  6C 41 62 6F 75 74 57 00 1F 00 44 72 61 67 46 69  lAboutW...DragFi
```

2. OriginalFirstThunk - INT(Import Name Table)

OriginalFirstThunk 멤버를 따라 갑니다. (RVA : 7990h -> file offset : 6D90h)

```
00006D80  16 7B 00 00 06 7B 00 00 2A 7B 00 00 00 00 00 00  .{...{..*{......
00006D90  7A 7A 00 00 5E 7A 00 00 9E 7A 00 00 50 7A 00 00  zz..^z..žz..Pz..
00006DA0  40 7A 00 00 8A 7A 00 00 6A 7A 00 00 14 7A 00 00  @z..Šz..jz...z..
00006DB0  2C 7A 00 00 00 00 00 00 DC 7B 00 00 D4 7B 00 00  ,z......Ü{..Ô{..
00006DC0  CA 7B 00 00 C2 7B 00 00 B6 7B 00 00 EA 7B 00 00  Ê{..Â{..¶{..ê{..
```

위 그림이 INT 입니다. 주소 배열 형태로 되어 있습니다. (배열의 끝은 NULL 로 되어 있습니다.)
주소값 하나 하나가 각각의 IMAGE_IMPORT_BY_NAME 구조체를 가리키고 있습니다. (<Fig. IAT 구조> 참고)

배열의 첫번째 값인 7A7Ah (RVA) 를 따라가 볼까요?

3. IMAGE_IMPORT_BY_NAME

RVA 값 7A7Ah 는 file offset 으로 6E7Ah 입니다.

```
00006E60  46 69 6E 64 54 65 78 74 57 00 15 00 52 65 70 6C  FindTextW...Repl
00006E70  61 63 65 54 65 78 74 57 00 00 0F 00 50 61 67 65  aceTextW....Page
00006E80  53 65 74 75 70 44 6C 67 57 00 0A 00 47 65 74 4F  SetupDlgW...GetO
00006E90  70 65 6E 46 69 6C 65 4E 61 6D 65 57 00 00 12 00  penFileNameW....
```

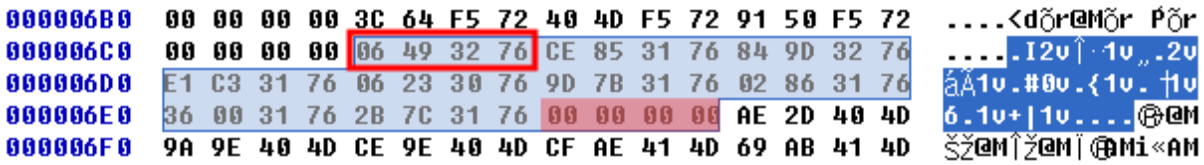
앞에 2 byte 는 Hint (ordinal) 로써 라이브러리에서 함수의 고유번호 입니다.
ordinal 뒤로 "PageSetupDlgW" 함수 이름 문자열이 보이시죠? (문자열 마지막은 '₩0' - C 언어와 동일)

여기까지 정리하면 INT 는 "함수 이름 주소 배열" 인데 첫번째 원소가 가리키는 함수 이름은 "PageSetupDlgW" 였습니다.

이제 IAT 에 해당 함수가 실제 메모리에 매핑된 **주소**를 얻어서 (GetProcAddress API 참고) IAT 에 입력하면 됩니다.

4. FirstThunk - IAT (Import Address Table)

IAT 의 RVA 값은 12C4h 이고 file offset 으로는 6C4h 입니다.



위 그림이 "comdlg32.dll" 라이브러리에 해당하는 IAT 입니다.
INT 와 마찬가지로 주소 배열 형태로 되어 있으며 배열의 끝은 NULL 입니다.

IAT 의 첫번째 원소값은 이미 76324906h 로 하드 코딩되어 있습니다.
notepad.exe 파일이 **메모리에 로딩될 때** 이 값은 위 3 번에서 구한 **정확한 주소값으로 대체** 됩니다.

* 사실 제 시스템(Windows XP SP3) 에서 76324906h 주소는 comdlg32.dll!PageSetupDlgW 함수의 정확한 주소값입니다.
* MS 가 서비스팩을 배포하면서 관련 시스템 파일을 재빌드 할때 이미 정확한 주소를 하드 코딩 한것입니다.
  (일반적인 DLL 은 IAT 에 실제 주소가 하드 코딩되어 있지 않고, INT 와 같은 값을 가지는 경우가 많습니다.)
* 참고로 일반적인 DLL 파일은 ImageBase 가 10000000h 으로 되어 있어서 보통 DLL relocation 이 발생하지만,
  Windows 시스템 DLL 파일들(kernel32, user32, gdi32, etc)은 고유의 ImageBase 가 있어서 DLL relocation 이 발생하지 않습니다.

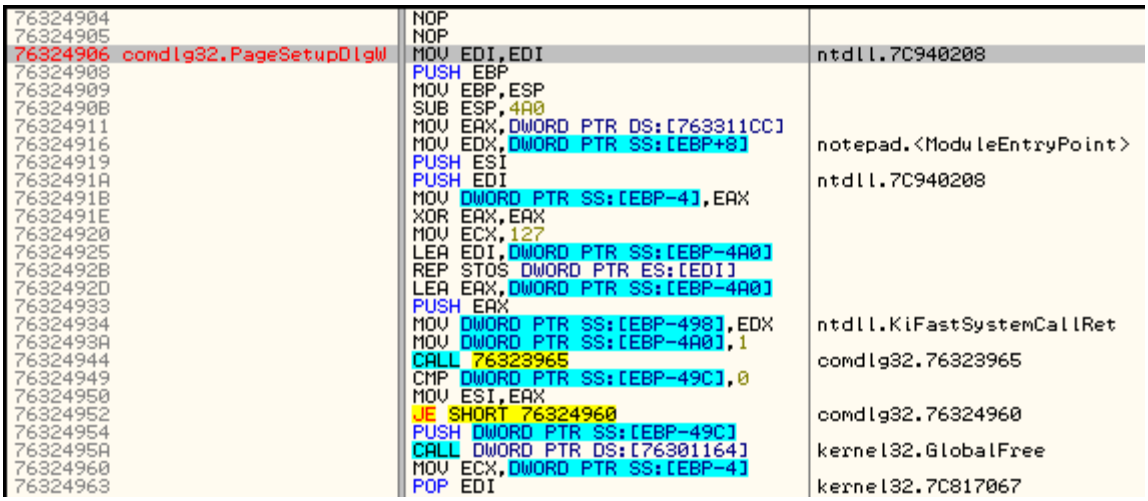
OllyDbg 를 이용해서 notepad.exe 의 IAT 를 확인해 보겠습니다.

Address	Value	Comment
010012B8	72F54D40	WINSPOOL.ClosePrinter
010012BC	72F55091	WINSPOOL.OpenPrinterW
010012C0	00000000	
010012C4	76324906	comdlg32.PageSetupDlgW
010012C8	763185CE	comdlg32.FindTextW
010012CC	76329D84	comdlg32.PrintDlgExW
010012D0	7631C3E1	comdlg32.ChooseFontW
010012D4	76302306	comdlg32.GetFileTitleW
010012D8	76317B9D	comdlg32.GetOpenFileNameW
010012DC	76318602	comdlg32.ReplaceTextW
010012E0	76310036	comdlg32.CommDlgExtendedError
010012E4	76317C2B	comdlg32.GetSaveFileNameW
010012E8	00000000	
010012EC	77BE2DAE	msvcrt._XcptFilter

notepad.exe 의 ImageBase 값은 01000000h 입니다.
따라서 comdlg32.dll!PageSetupDlgW 함수의 IAT 주소는 010012C4h 이며 76324906h 로 정확한 값이 들어와 있습니다.

* XP SP3 notepad.exe 를 다른 OS (2000, Vista, etc) 혹은 다른 ServicePack(SP1, SP2) 에서 실행하면, 010012C4h 주소에는 다른 값이 세팅됩니다. (그 OS 혹은 ServicePack 에 있는 comdlg32.dll!PageSetupDlgW 의 주소)

해당 주소(76324906h)로 가면 아래와 같이 comdlg32.dll 의 PageSetupDlgW 함수 시작이 나타납니다.



```
76324904                          NOP
76324905                          NOP
76324906 comdlg32.PageSetupDlgW   MOV EDI,EDI                          ntdll.7C940208
76324908                          PUSH EBP
76324909                          MOV EBP,ESP
7632490B                          SUB ESP,4A0
76324911                          MOV EAX,DWORD PTR DS:[763311CC]
76324916                          MOV EDX,DWORD PTR SS:[EBP+8]         notepad.<ModuleEntryPoint>
76324919                          PUSH ESI
7632491A                          PUSH EDI                             ntdll.7C940208
7632491B                          MOV DWORD PTR SS:[EBP-4],EAX
7632491E                          XOR EAX,EAX
76324920                          MOV ECX,127
76324925                          LEA EDI,DWORD PTR SS:[EBP-4A0]
7632492B                          REP STOS DWORD PTR ES:[EDI]
7632492D                          LEA EAX,DWORD PTR SS:[EBP-4A0]
76324933                          PUSH EAX
76324934                          MOV DWORD PTR SS:[EBP-498],EDX       ntdll.KiFastSystemCallRet
7632493A                          MOV DWORD PTR SS:[EBP-4A0],1
76324944                          CALL 76323965                        comdlg32.76323965
76324949                          CMP DWORD PTR SS:[EBP-49C],0
76324950                          MOV ESI,EAX
76324952                          JE SHORT 76324960                    comdlg32.76324960
76324954                          PUSH DWORD PTR SS:[EBP-49C]
7632495A                          CALL DWORD PTR DS:[76301164]         kernel32.GlobalFree
76324960                          MOV ECX,DWORD PTR SS:[EBP-4]
76324963                          POP EDI                              kernel32.7C817067
```

EAT (Export Address Table)

Windows 운영체제에서 라이브러리(Library) 란 다른 프로그램에서 불러 쓸 수 있도록
관련 함수들을 모아놓은 파일(DLL/SYS)입니다.

Win32 API 가 대표적인 Library 이며, 그 중에서도 kernel32.dll 파일이 가장 대표적인 Library 파일이라고 할 수 있습니다.

EAT(Export Address Table) 은 라이브러리 파일에서 제공하는 함수를
다른 프로그램에서 가져다 사용할 수 있도록 해주는 매커니즘 입니다.

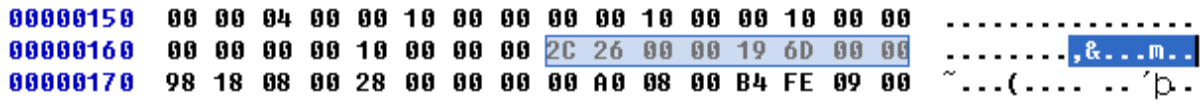
앞서 설명드린 IAT 와 마찬가지로 PE 파일내에 특정 구조체(**IMAGE_EXPORT_DIRECTORY**)에 정보를 저장하고 있습니다.

라이브러리의 EAT 를 설명하는 IMAGE_EXPORT_DIRECTORY 구조체는 PE 파일에 하나만 존재합니다.

* 참고로 IAT 를 설명하는 IMAGE_IMPORT_DESCRIPTOR 구조체는 여러개의 멤버를 가진 배열 형태로 존재합니다.
  왜냐하면 PE 파일은 여러개의 라이브러리를 동시에 Import 할 수 있기 때문이지요

PE 파일내에서 IMAGE_EXPORT_DIRECTORY 구조체의 위치는 PE Header 에서 찾을 수 있습니다.
IMAGE_OPTIONAL_HEADER32.DataDirectory[0].VirtualAddress 값이
실제 IMAGE_EXPORT_DIRECTORY 구조체 배열의 시작 주소 입니다. (RVA 값입니다.)

아래는 kernel32.dll 파일의 IMAGE_OPTIONAL_HEADER32.DataDirectory[0].VirtualAddress 를 보여주고 있습니다.
(첫번째 4 byte 가 VirtualAddress, 두번째 4 byte 가 Size 멤버입니다.)




```
offset    value    description
-----------------------------------------------
...
00000160 00000000 loader flags
00000164 00000010 number of directories
00000168 0000262C RVA  of EXPORT Directory
0000016C 00006D19 size of EXPORT Directory
00000170 00081898 RVA  of IMPORT Directory
00000174 00000028 size of IMPORT Directory
...
```

* IMAGE_OPTIONAL_HEADER32 구조체에 대해서 궁금하신 분은

IMAGE_OPTIONAL_HEADER 설명 을 참고하시기 바랍니다.

RVA 값이 262Ch 이므로 File offset 은 1A2Ch 입니다.
(RVA 와 File offset 간의 변환과정이 잘 이해 안가시는 분은 IMAGE_SECTION_HEADER 설명을 참고하시기 바랍니다.)

IMAGE_EXPORT_DIRECTORY

IMAGE_EXPORT_DIRECTORY 구조체는 아래와 같습니다.

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;          // creation time date stamp
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;                   // address of library file name
DWORD   Base;                   // ordinal base
    DWORD   NumberOfFunctions;      // number of functions
    DWORD   NumberOfNames;          // number of names
    DWORD   AddressOfFunctions;     // address of function start address array
    DWORD   AddressOfNames;         // address of functino name string array
    DWORD   AddressOfNameOrdinals;  // address of ordinal array
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;

* 출처 : Microsoft 의 Visual C++ 에서 제공하는 winnt.h
```

중요 멤버들에 대한 설명입니다. (여기에 나오는 주소는 모두 RVA 입니다.)

NumberOfFunctions : 실제 export 함수 갯수

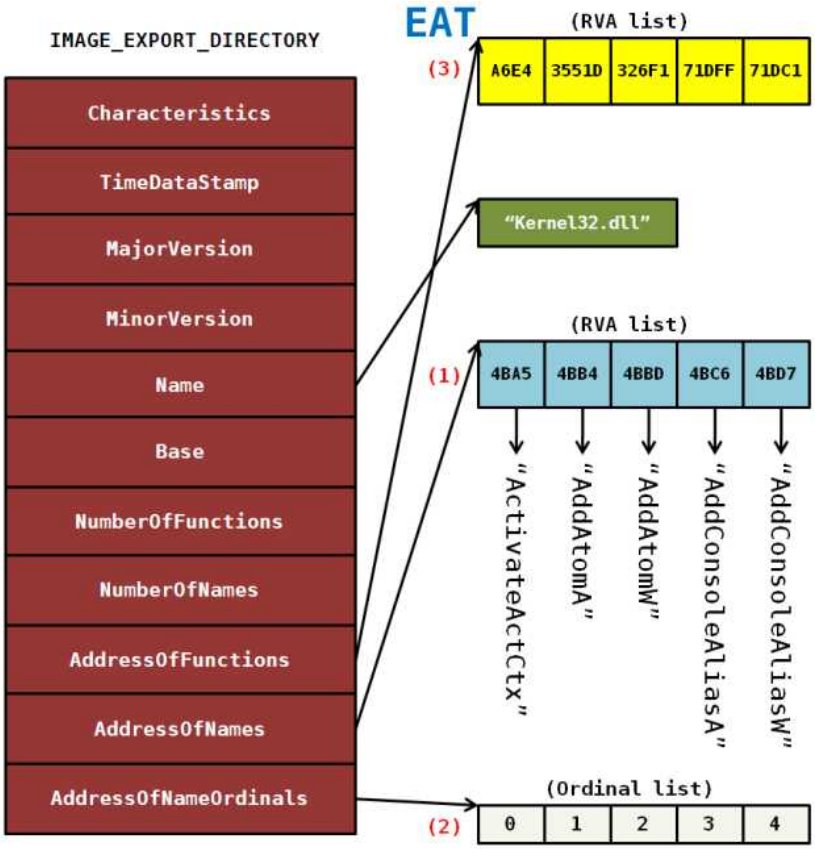
NumberOfNames : export 함수중에서 이름을 가지는 함수 갯수 (<= NumberOfFunctions)

AddressOfFunctions : export 함수들의 시작 위치 배열의 주소 (배열의 원소개수 = NumberOfFunctions)

AddressOfNames : 함수 이름 배열의 주소 (배열의 원소개수 = NumberOfNames)

AddressOfOrdinals : ordinal 배열의 주소 (배열의 원소개수 = NumberOfNames)

아래 그림은 kernel32.dll 파일의 IMAGE_EXPORT_DIRECTORY 의 **구조**를 나타내고 있습니다.



<Fig. EAT 구조>

라이브러리에서 함수 주소를 얻는 API 는 GetProcAddress() 입니다.

GetProcAddress() 함수가 함수 이름을 가지고 어떻게 함수 주소를 얻어내는 **순서**를 설명드리겠습니다.

1. AddressOfNames 멤버를 이용해 "함수 이름 배열" 로 갑니다.
2. "함수 이름 배열"은 문자열 주소가 저장되어 있습니다. 문자열 비교(strcmp)를 통하여 원하는 함수 이름을 찾습니다.
   이 때의 배열 인덱스를 name_index 라고 하겠습니다.
3. AddressOfNameOrdinals 멤버를 이용해 "ordinal 배열" 로 갑니다.
4. "ordinal 배열" 에서 name_index 로 해당 ordinal_index 값을 찾습니다.
5. AddressOfFunctions 멤버를 이용해 "함수 주소 배열 - EAT" 로 갑니다.
6. "함수 주소 배열 - EAT" 에서 아까 구한 ordinal_index 를 배열 인덱스로 하여 원하는 함수의 시작 주소를 얻습니다.

위 <Fig. EAT 구조> 는 kernel32.dll 의 경우를 보여주고 있습니다.

kernel32.dll 은 export 하는 모든 함수에 이름이 존재하며,
AddressOfNameOrdinals 배열의 값이 index = ordinal 형태로 되어있습니다.

하지만 모든 DLL 파일이 이와 같지는 않습니다.
export 하는 함수 중에 이름이 존재하지 않을 수 도 있으며 (ordinal 로만 export 함)
AddressOfNameOrdinals 배열의 값이 index != ordinal 인 경우도 있습니다.

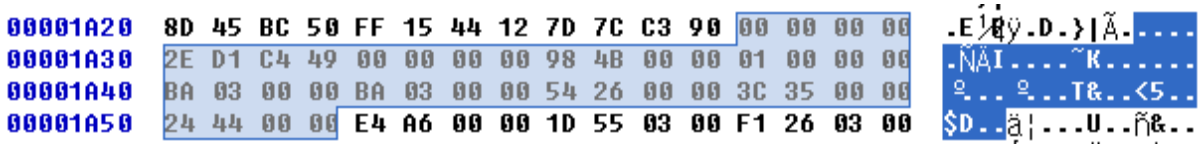
따라서 위 **순서**를 따라야만 정확한 **함수 주소**를 얻을 수 있습니다.

* 참고로 함수 이름 없이 ordinal 로만 export 된 함수의 주소를 찾을 수 도 있습니다.

kernel32.dll 을 이용한 실습

실제 **kernel32.dll** 파일의 EAT 에서 **AddAtomW** 함수 주소를 찾는 실습을 해보겠습니다.
(<Fig. EAT 구조> 를 참고하세요.)

앞에서 kernel32.dll 의 IMPORT_EXPORT_DIRECTORY 구조체 file offset 은 1A2Ch 라고 하였습니다.
hex editor 로 1A2Ch 주소로 갑니다.



각 구조체 멤버별로 나타내 보겠습니다.

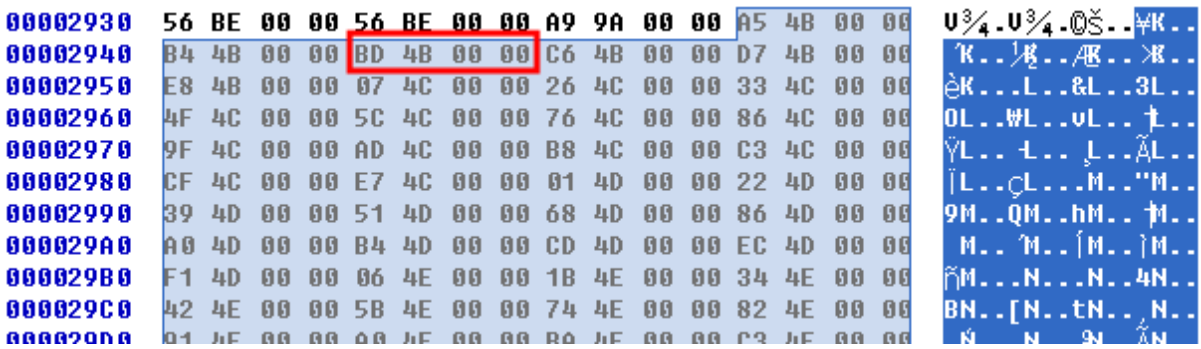
```
Characteristics       = 00000000h
TimeDateStamp         = 49C4D12Eh
MajorVersion          =     0000h
MinorVersion          =     0000h
Name                  = 00004B98h
Base                  = 00000001h
NumberOfFunctions     = 000003BAh
NumberOfNames         = 000003BAh
AddressOfFunctions    = 00002654h
```

```
AddressOfNames        = 0000353Ch
AddressOfNameOrdinals = 00004424h
```

위에서 알려드린 **순서**대로 진행하겠습니다.

1. "함수 이름 배열"

AddressOfNames 멤버의 값은 RVA = 353Ch 이므로 file offset = 293Ch 입니다.

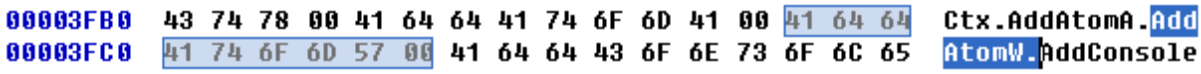


4 byte 의 RVA 로 이루어진 배열입니다. 배열 원소의 갯수는 NumberOfNames (3BAh) 입니다.
저 모든 RVA 값을 하나하나 따라가면 함수 이름 문자열이 나타납니다.

2. 원하는 함수 이름 찾기

설명의 편의를 위해 우리가 찾는 "AddAtomW" 함수 이름 문자열은 배열의 세번째 원소의 값(주소)를 따라가면 됩니다.

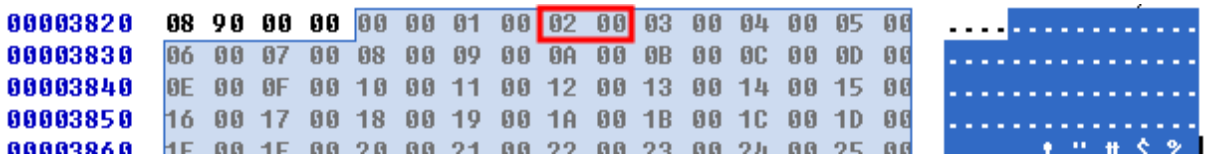
RVA = 4BBDh 이므로 file offset = 3FBDh 입니다.



이때 배열의 인덱스(**index**) 는 **2** 입니다.

3. "Ordinal 배열"

AddressOfNameOrdinals 멤버의 값은 RVA = 4424h 이므로 file offset = 3824h 입니다.



2 byte 의 ordinal 로 이루어진 배열이 나타납니다.

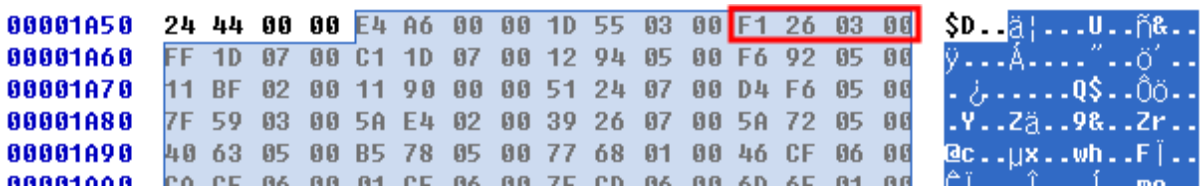
4. ordinal

위에서 구한 index 값 2 를 위의 "ordinal 배열" 에 적용하면 **ordinal 2** 를 구할 수 있습니다.

AddressOfNameOrdinals[index(2)] = ordinal(2)

5. "함수 주소 배열(EAT)"

AddressOfFunctions 멤버의 값은 RVA = 2654h 이므로 file offset = 1A54h 입니다.



4 byte 함수 주소 RVA 배열이 나타납니다.

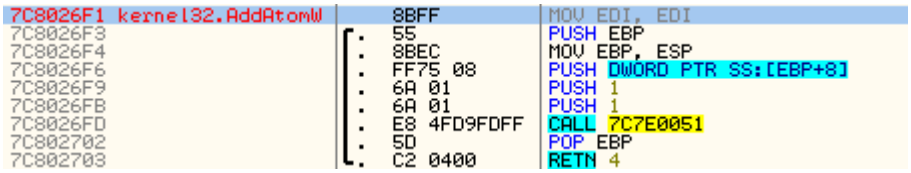
6. AddAtomW 함수 주소

위에서 구한 ordinal_index 를 "함수 주소 배열(EAT)" 에 적용하면 해당 함수의 RVA (000326F1h)를 얻을 수 있습니다.

AddressOfFunctions[ordinal(2)] = 326F1

kernel32.dll 의 ImageBase = 7C7D0000h 입니다.
따라서 "AddAtomW" 함수의 실제 주소(VA)는 **7C8026F1h** 입니다.

OllyDbg 를 이용해서 확인해 보겠습니다.



네, 정확히 7C8026F1h 주소(VA)에 우리가 찾는 "AddAtomW" 함수가 나타납니다.

PE File Format

지금까지 오랜 시간에 걸쳐 PE(Portable Executable) File Format 에 대해 살펴보았습니다.

PE 스펙을 보면 각 구조체 멤버 하나하나 자세히 기술하고 있지만
리버싱에서 주목해야 하는 멤버들만 추려서 설명드렸습니다.

특히 IAT, EAT 에 관한 내용은 실행압축(Run-Time Packer), Anti-Debugging, DLL Injection, API Hooking 등
매우 다양한 중/고급 리버싱 주제들의 기반 지식이 됩니다.

hex editor 와 연필, 종이만 가지고 IAT/EAT 의 주소를 하나하나 계산해서
파일/메모리 에서 실제 주소를 찾는 훈련을 많이 해보시기 바랍니다.

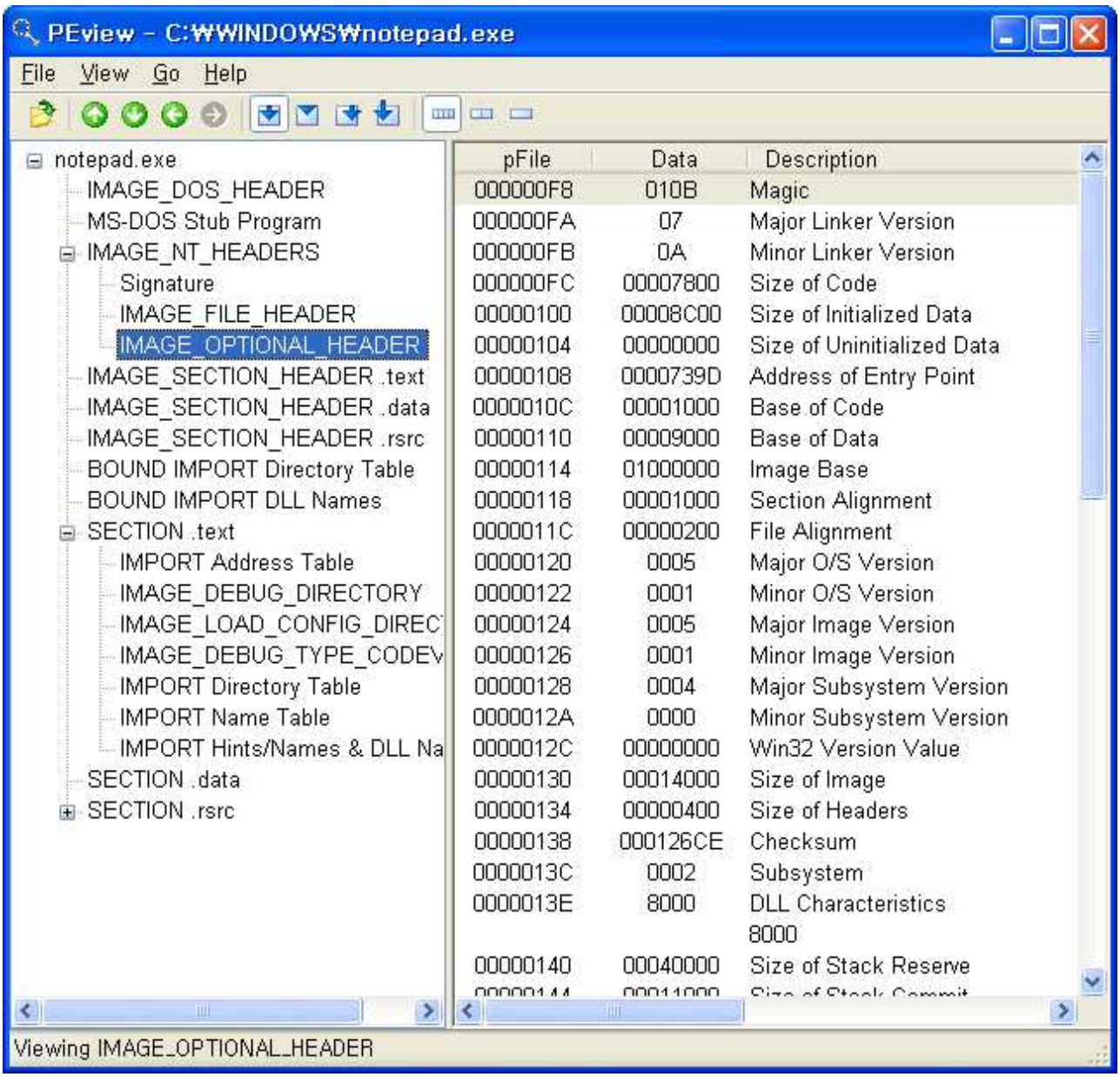
쉽지 않은 내용이지만 그만큼 리버싱에서 중요한 위치를 차지하고 있기 때문에
고급 리버싱을 원하는 분들께서는 반드시 습득하셔야 합니다.

PEView.exe

간단하고 사용하기 편리한 PE Viewer 프로그램(**PEView.exe**)을 소개해 드립니다.
(개인이 만든 무료 공개 SW 입니다.)

http://www.magma.ca/~wjr/PEview.zip

아래는 PEView.exe 의 실행 화면입니다.



PE Header 를 각 구조체 별로 보기 쉽게 표현해주고, RVA <-> File Offset 변환을 간단히 수행해줍니다.
(제가 설명드렸던 내용과 용어 사용에 있어서 약간 틀릴 수 있습니다. 둘 다 익혀두시는게 의사소통에 좋습니다.)

위와 같은 PE Viewer 를 직접 제작해 보시는 것을 추천드립니다.
저 또한 처음 PE Header 를 공부할 때 (검증을 위해) 콘솔 기반의 PE Viewer 를 만들어서 지금까지 잘 사용하고 있습니다.
직접 제작하다 보면 자신이 잘 몰랐거나 잘 못 이해하던 부분을 정확히 파악하고 제대로 공부할 수

있습니다.

PE Patch

PE 스펙은 말 그대로 권장 스펙이기 때문에 각 구조체 내부에 보면 사용되지 않는 많은 멤버들이 많이 있습니다.

또한 말 그대로 스펙만 맞추면 PE 파일이 되기 때문에 일반적인 상식을 벗어나는 PE 파일을 만들어 낼 수 있습니다.

PE Patch 란 바로 그런 PE 파일을 말합니다.
PE 스펙에 어긋나지는 않지만 굉장히 창의적인(?) PE Header 를 가진 파일들입니다.
(정확히 표현하면 PE Header 를 이리저리 꼬아 놨다고 할 수 있습니다.)

PE Patch 만 해도 따로 고급 주제로 다뤄야 할 만큼 (리버싱에 있어서) 넓고도 깊은 분야입니다.

한가지만 소개해 드리겠습니다.
지금까지 배웠던 PE Header 에 대한 상식이 사뿐히 깨지는 경험을 하실 수 있습니다.
(그러나 PE 스펙에 벗어난건 없답니다.)

아래 사이트는 **tiny pe** 라고 가장 작은 크기의 PE 파일을 만드는 내용입니다.

http://blogs.securiteam.com/index.php/archives/675

411 byte 크기의 (정상적인) PE 파일을 만들어 냈습니다.
IMAGE_NT_HEADERS 구조체 크기만 해도 248 byte 라는걸 생각하면 이것은 매우 작은 크기의 PE 파일입니다.

다른 사람들이 계속 도전해서 **304 byte** 크기의 파일까지 나타나게 됩니다.

그리고 마지막으로 어떤 다른 사람이 위 사이트를 본 후 자극을 받아서
아래와 같이 극단적이고 매우 황당한 PE 파일을 만들어 내었습니다.

http://www.phreedom.org/solar/code/tinype/

이곳에 가면 Windows XP 에서 정상 실행되는 **97 byte** 짜리 PE 파일을 다운 받을 수 있습니다.
(2009 년 4 월 현재까지 최고 기록입니다.)

또한 PE Header 와 tiny pe 제작과정에 대한 내용을 자세히 설명하고 있어서 읽어보시면 크게 도움이 되실 겁니다.
(약간의 assembly 언어에 대한 지식이 요구됩니다.)

모두 다운 받아서 하나씩 분석해 보시기 바랍니다. 분명 크게 도움이 됩니다.

Epilogue

이러한 PE patch 파일들은 저뿐만 아니라 일반적인 리버서들의 고정관념을 깨트리는 내용이며
그래서 리버싱 공부가 더 즐겁습니다.

PE Header 에 대해서 다시 한번 강조 하고 싶은 내용은 아래와 같습니다.

- PE 스펙은 그저 스펙일 뿐이다. (만들어 놓고 사용되지 않는 내용이 많다.)
- 내가 지금 알고 있는 PE Header 에 대한 지식도 잘 못된 부분이 있을 수 있다.
  (tiny pe 외에도 PE header 를 조작하는 여러 창의적인 기법들이 계속 쏟아져 나온다.)
- 항상 모르는 부분을 체크해서 보강하자.