PE Structure

CodeEngn Co-Administrator
and
Team Sur3x5F Member

Nick : Deok9
E-mail : DDeok9@gmail.com
HomePage : http://Deok9.Sur3x5f.org
Twitter :@DDeok9

<< Contents >>

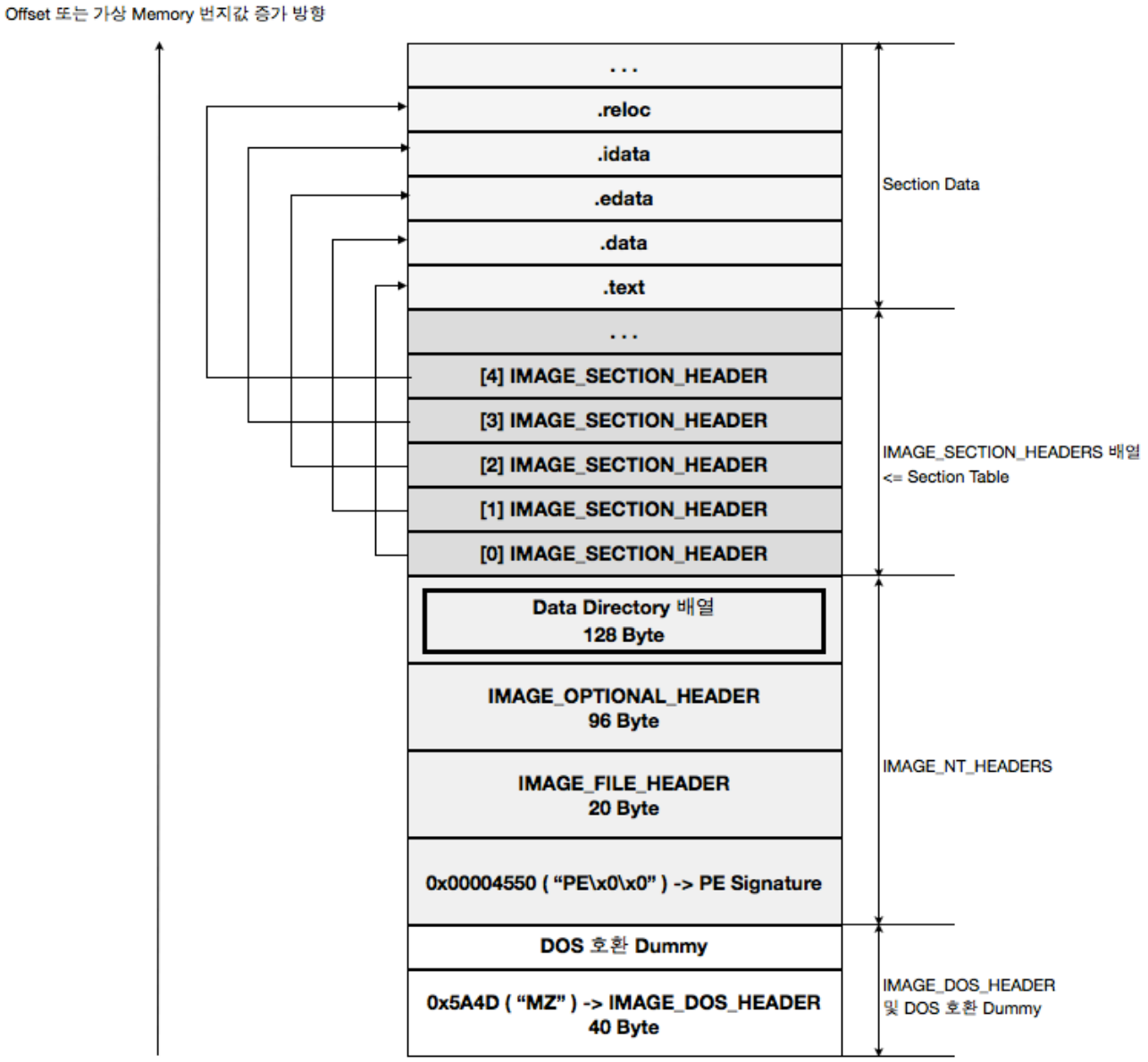
1. PE 분석을 위한 준비

1) PE 란 ?

Portable Executable 의 약자로서, Platform 에 관계없이 Win32 OS 가 돌아가는 System 이면 어디서든 실행 가능하다는 의미를 가진다.

다른 File 과는 달리, PE File 이 Load 될 때 VMM (Virtual Memory Manager) 는 Page File 을 사용하지 않고, PE File 자체를 마치 Page File 처럼 가상 주소공간에 그대로 Mapping 한다.

✱ 이와 같은 성질 때문에 WinNT.H Header File 내의 PE 관련 구조체는 Image 라 칭해져 있다.



[그림 1 - 1 - 1] PE File Format 전체 구조

2) PE 분석을 위한 개념 정리

(i) RVA (Relative Virtual Address)

간단하게 Memory 상에서의 PE 의 시작주소에 대한 Offset 이라고 말할 수 있다.

✱ 실제 주소 번지(가상 주소) = Image Load 시작 번지 + Offset

✱ Image Load 시작 번지는 IMAGE_OPTIONAL_HEADER 구조체의 ImageBase Field 에 지정

(ii) Section

종류	이름	설명
Code	.text	Program 실행을 위한 Code 를 담고 있는 Section
Data	.data	초기화된 전역변수들을 담고 있는 읽고 쓰기 가능한 Section
	.rdata	읽기 전용 Data Section (문자열 표현, C++ 가상함수)
Import API 정보	.idata	Import 할 DLL 과 그 API 들에 대한 정보, IAT 를 가지는 Section
	.didat	Delay-Loading Import Data 를 위한 Section
Export API 정보	.edata	Export 할 API 에 대한 정보를 담고 있는 Section (주로 DLL)
Resource	.rsrc	Window APP Resource 관련 Data 들이 배치되는 Section
재배치 정보	.reloc	기본 재배치 정보를 담고 있는 Section (주로 DLL)
TLS	.tls	Thread 지역 저장소를 위한 Section
Debugging	.debug$P	미리 Compile 된 Header 사용시 OBJ 에만 존재하는 Section

(iii) VAS (Virtual Address Space) 와의 관계

Process 는 자신의 구성원으로 가상 주소 공간 (VAS) 를 가지며, 이를 실제 물리적 기억 장치와 연결시켜 주는것을 VMM (Virtual Memory Manager) 라 한다.

물리적 기억 장치란 RAM 뿐 아니라, Default 로 Windows System 이 설치된 논리적 Disk 의 Root 에 존재하는 Hard-disk 상의 특정 File 인 (PageFile.sys) 를 포함한다.

✱ 해당 가상 주소 공간에 무엇을 읽거나 쓰는 행위는 결국 해당 Paging File 의 특정 Page 에 동일한 행위를 하는 것이 된다.

✱ 가상 주소 공간의 Page 들은 Paging File 에 Mapping 되어야 사용 가능하며, Mapping 된 Page 를 COMMIT 된 Page 라 하고, File 자체가 Paging File 을 대신하는 경우를 MMF 라 한다.

3) PE 분석을 위한 Utility

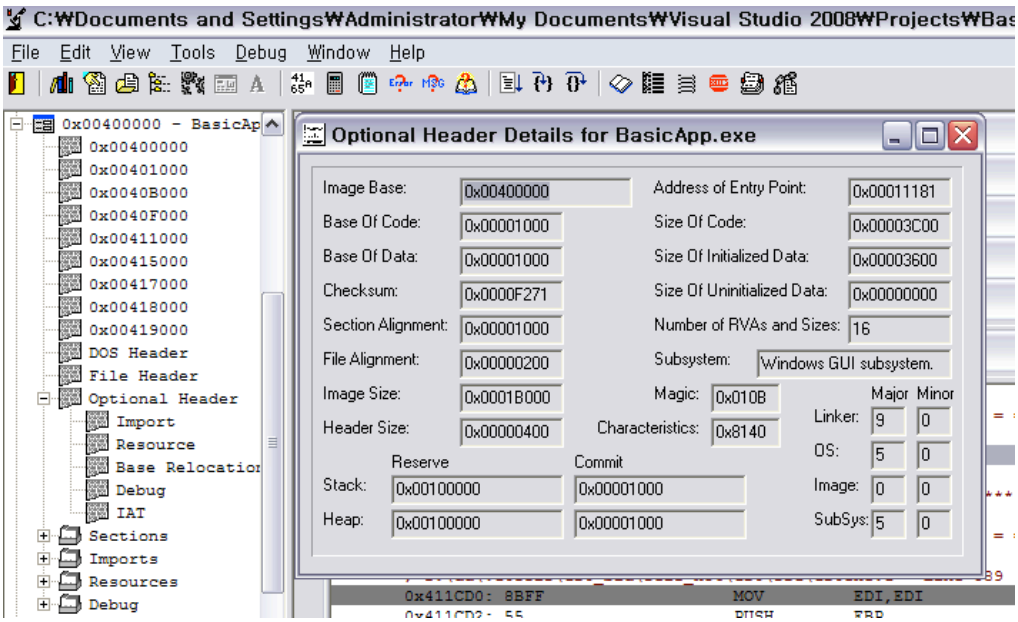
(i) DumpBin.exe

```
C:₩Program Files₩Microsoft Visual Studio 9.0₩vc₩bin>dumpbin
Microsoft (R) COFF/PE Dumper Version 9.00.21022.08
Copyright (C) Microsoft Corporation.  All rights reserved.

사용법: DUMPBIN [options] [files]

   옵션:

      /ALL
      /ARCHIVEMEMBERS
      /CLRHEADER
      /DEPENDENTS
      /DIRECTIVES
      /DISASM[:{BYTES|NOBYTES}]
      /ERRORREPORT:{NONE|PROMPT|QUEUE|SEND}
      /EXPORTS
      /FPO
      /HEADERS
      /IMPORTS[:filename]
      /LINENUMBERS
      /LINKERMEMBER[:{1|2}]
      /LOADCONFIG
      /OUT:filename
      /PDATA
      /PDBPATH[:VERBOSE]
      /RANGE:vaMin[,vaMax]
      /RAWDATA[:{NONE|1|2|4|8}[,#]]
      /RELOCATIONS
```

[그림 1 - 3 - 1] DumpBin.exe

➡ COFF 형태의 이전 PE Format 을 비롯한, 가능한 모든 형태의 PE Format 을 지원

(ii) PEBrowse Professional Interactive



[그림 1 - 3 - 2] PEBrowse Professional Interactive

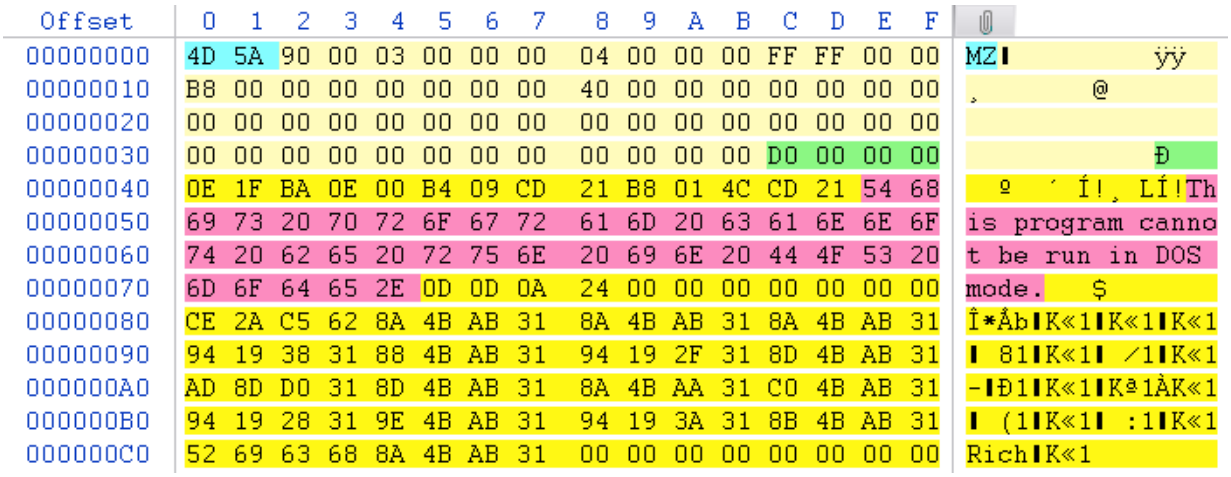
➡ GUI 환경을 지원하며 Debugging 도 가능한 Tool

2. PE Header

1) IMAGE_DOS_HEADER 구조와 DOS Stub

DOS Stub 은 40 Byte 의 IMAGE_DOS_HEADER 구조체와 해당 Program Code 로 이루어 진다.

✱ "This program cannot be run in DOS mode." 란 문장을 출력하기 위한 조그마한 16 bit DOS 용 응용 Program 이다.



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
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	D0	00	00	00
00000040	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
00000050	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
00000060	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
00000070	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
00000080	CE	2A	C5	62	8A	4B	AB	31	8A	4B	AB	31	8A	4B	AB	31
00000090	94	19	38	31	88	4B	AB	31	94	19	2F	31	8D	4B	AB	31
000000A0	AD	8D	D0	31	8D	4B	AB	31	8A	4B	AA	31	C0	4B	AB	31
000000B0	94	19	28	31	9E	4B	AB	31	94	19	3A	31	8B	4B	AB	31
000000C0	52	69	63	68	8A	4B	AB	31	00	00	00	00	00	00	00	00

ASCII
MZ▮ ÿÿ
¸ @
Đ
º ´ Í!¸ LÍ!Th
is program canno
t be run in DOS
mode. $
Î*Åb▮K«1▮K«1▮K«1
▮ 81▮K«1▮ /1▮K«1
-▮Đ1▮K«1▮Kª1ÀK«1
▮ (1▮K«1▮ :1▮K«1
Rich▮K«1

[그림 2 - 1 - 1] IMAGE_DOS_HEADER 와 DOS Stub

```
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
    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, *PIMAGE_DOS_HEADER;
```

[그림 2 - 1 - 2] IMAGE_DOS_HEADER 구조체

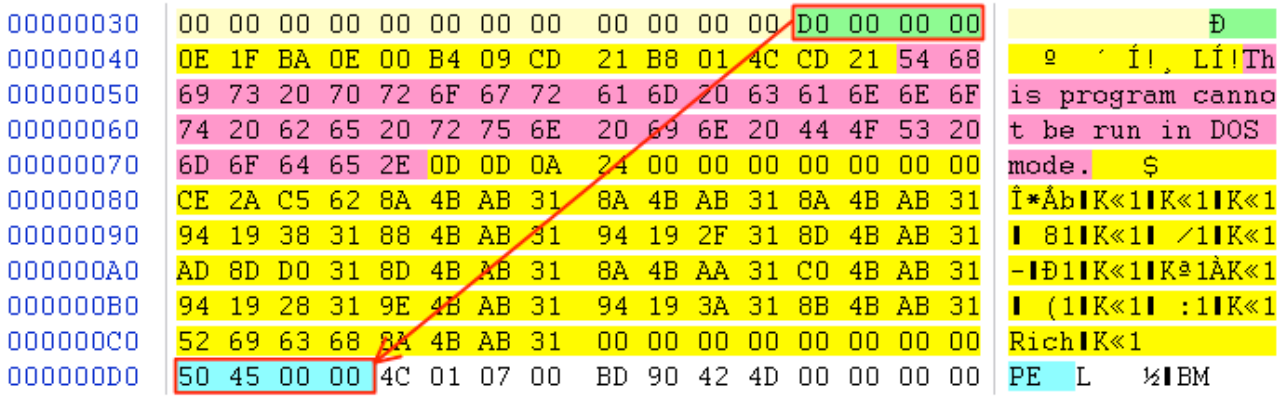
➡ e_magic 은 ASCII "MZ" 로 고정이 되어있다. (DOS 설계자 Mark Zbikowski 에 유래)
➡ e_lfanew 는 실제 PE File 의 시작이라 할 수 있는 IMAGE_NT_HEADER 의 시작 Offset 을 가진다.
➡ PE File 을 열어서 IMAGE_DOS_HEADER 를 읽어들인 다음 e_magic 이 "MZ" 인지 확인한 후 e_lfanew 값을 읽어들여 해당 값만큼 File Pointer 를 이동시키면 PE 분석이 가능하다.

2) IMAGE_NT_HEADERS 구조

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

[그림 2 - 2 - 1] IMAGE_NT_HEADERS 구조체

(i) DWORD Signature

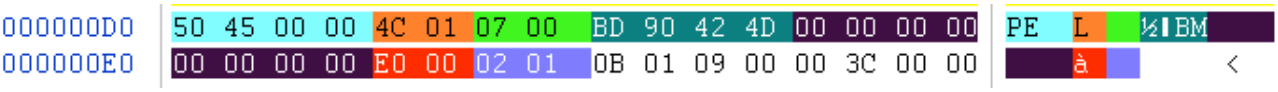


[그림 2 - 2 - 2] DWORD Signature

➡ PE File 을 나타내는 magic number 로써, 4 Byte 이며 항상 "PE\x00\x00" 이다.

✱ IMAGE_DOS_HEADER 의 e_lfanew field 가 가리키는 Offset 으로부터 4 Byte 값이다.

(ii) IMAGE_FILE_HEADER 구조



[그림 2 - 2 - 3] IMAGE_FILE_HEADER

```
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;
```

[그림 2 - 2 - 4] IMAGE_FILE_HEADER 구조체

➡ WORD Machine : 해당 File 의 CPU ID (Intel 386 = 0x014c, Intel 64 = 0x0200)

➡ WORD NumberOfSections : 해당 File 의 Section 수

➡ DWORD TimeDateStamp : 1970년 1월 1일을 기준으로 해당 File 생성 시간을 초로 표현

➡ DWORD PointerToSymbolTable, NumberOfSymbols : COFF Symbol Table의 File Offset 과 그 안의 Symbol 수 (현재 Symbol 크기가 커져 파일로 저장되어 사용됨)

➡ WORD SizeOfOptionalHeader : IMAGE_OPTIONAL_HEADER 의 Byte 수

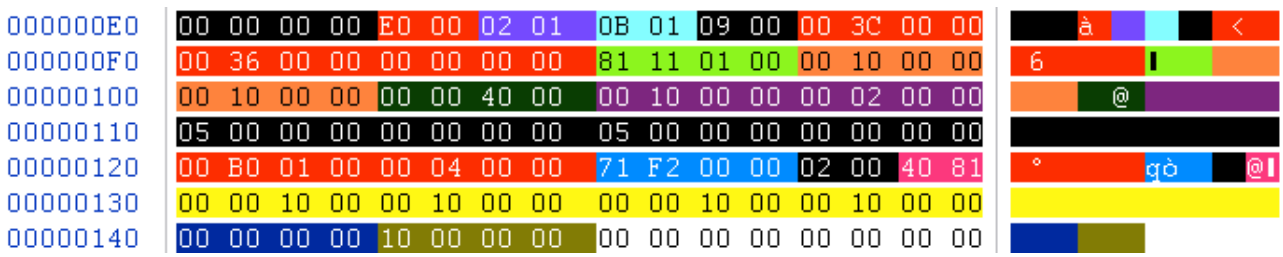
✱32 bit 일 경우 0xE0 (224 Byte), 64 bit 일 경우 0xF0 (240 Byte)

➡ WORD Characteristics : 해당 PE File 에 대한 특정 정보를 나타내는 Flag

MACRO 명 (IMAGE_FILE ~)	값	의미
_RELOCS_STRIPPED	0x0001	재배치 정보가 없음
_EXECUTABLE_IMAGE	0x0002	실행 File Image
_LINE_NUMS_STRIPPED	0x0004	Line 정보가 없음
_LOCAL_SYMS_STRIPPED	0x0008	Local Symbol 이 없음
_AGGRESIVE_WS_TRIM	0x0010	OS 가 적극적으로 WorkingSet 정리
_LARGE_ADDRESS_AWARE	0x0020	Application 이 2G 이상의 주소 제어
_32BIT_MACHINE	0x0100	32 bit Word Machine 필요
_DEBUG_STRIPPED	0x0200	Debug 정보가 .DBG File 에 존재
_REMOVABLE_RUN_FROM_SWAP	0x0400	이동 장치에 존재시 SWAP File 로 고정 Disk 에 Copy 후실행
_NET_RUN_FROM_SWAP	0x0800	Network 에 존재시 SWAP File 로 고정 Disk 에 Copy 후실행
_DLL	0x2000	DLL File
_UP_SYSTEM_ONLY	0x4000	하나의 Processor 장착 기기에만 실행

[표 2 - 2 - 1] PE 특성

(iii) IMAGE_OPTIONAL_HEADER 구조



[그림 2 - 2 - 5] IMAGE_DATA_DIRECTORY 제외한 IMAGE_OPTIONAL_HEADER

```
typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

    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;
```

[그림 2 - 2 - 6] IMAGE_OPTIONAL_HEADER 구조체

➡ 전체 224 Byte 로 구성되어 있으며, 96 Byte 를 차지하는 36 개의 기본 Field 와 8 Byte 크기의 IMAGE_DATA_DIRECTORY 구조체에 대한 Entry 16개 (128 Byte) 로 구성되어 있다.

Field	설명
WORD Magic	IMAGE_OPTIONAL_HEADER 를 나타내는 Signature ✱ 32 bit 일 경우 0x010B, 64 bit 일 경우 0x020B
BYTE Major & Minor LinkerVersion	해당 File 을 만들어낸 Linker 의 Version
DWORD SizeOfCode	IMAGE_SCN_CNT_CODE 속성을 가진 Section들 크기합

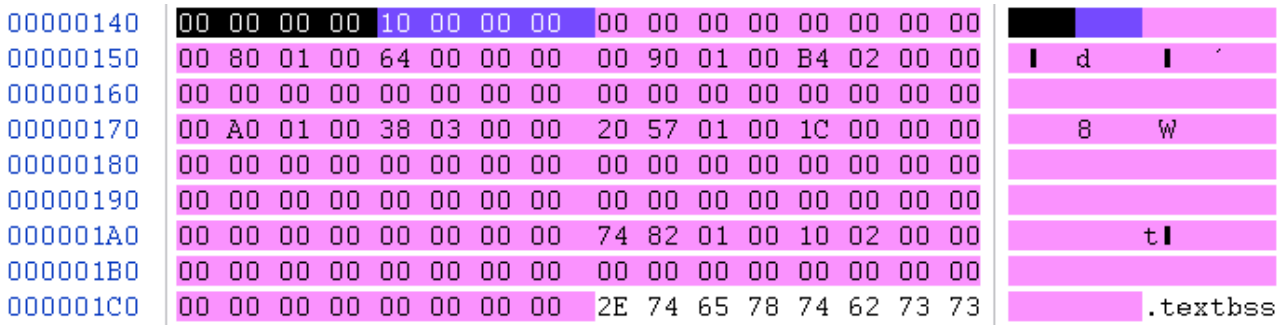
Field	설명
DWORD SizeOfInitializedData	IMAGE_SCN_CNT_ INITIALIZED_DATA 속성을 가진 Section들 크기합 (File 상에서의 정보)
DWORD SizeOfUninitializedData	IMAGE_SCN_CNT_ UNINITIALIZED_DATA 속성을 가진 Section들 크기합 ✱ 일반적으로 일반 Data Section 에 병합되므로 0 으로 Set
DWORD AddressOfEntryPoint	Loader 가 실행을 개시할 주소 ✱ 보통 .text Section 내의 특정번지
DWORD BaseOfCode	Memory 에 Load 될 때, 첫번째 Code Section 시작 RVA ✱ 보통 PE Header 와 Data Section 사이에 존재
DWORD BaseOfData	Memory 에 Load 될 때, Data 의 첫번째 Byte 의 RVA
DWORD ImageBase	해당 PE가 가상주소공간에 Mapping 될 시의 시작주소
DWORD SectionAlignment	Intel 기반의 Window 의 경우 Memory Page 의 크기가 4K 이기 때문에 0x1000 이며, File Alignment 값보다 크거나 같음
DWORD FileAlignment	PE 내에서 Section 들의 정렬 단위 ✱ Disk 의 Sector 단위
WORD Major & Minor OperatingSystemVersion	File 실행 시 필요한 OS 최소 Version
WORD Major & Minor ImageVersion	User 가 나름대로의 Version 주입가능
WORD Major & Minor SubsystemVersion	File 실행 시 필요한 Subsystem 최소 Version
DWORD Win32VersionValue	거의 사용않으며 보통 0
DWORD SizeOfImage	PE 를 Memory 상에 Load 할 때 확보 해야할 충분한 크기 ✱ 반드시 SectionAlignment Field 값의 배수가 되어야 한다.
DWORD SizeOfHeaders	Header, Section Table 들의 크기를 모두 합친 Byte 수 ✱ 반드시 FileAlignment 값의 배수
DWORD CheckSum	IMAGE 의 CheckSum 값 (무결성 검사)
WORD Subsystem	실행 File 이 실행될 환경 자체 (CUI : 3, GUI : 2)

Field	설명
WORD DllChracteristics	DLL 초기화 함수가 호출되어야 하는지를 지시하는 Flag
DWORD SizeOfStackReserve/ Commit & SizeOfHeapReserve/ Commit	Default Stack 과 Heap 을 생성 ✱ Reserve 는 0x00100000 이며 Commit 는 0x00001000
DWORD LoaderFlags	Debugging 자원에 관계된 것으로 0으로 Set
DWORD NumberOfRvaAndSizes	IMAGE_DATA_DIRECTORY 구조체 배열의 원소 개수 ✱ 항상 16개 이기 때문에 0x00000010

[표 2 - 2 - 2] IMAGE_OPTIONAL_HEADER 의 Field

IMAGE_DATA_DIRECTORY 구조체 배열 은 128 Byte (8 * 16) 이며, 여기서 16 은 IMAGE_OPTIONAL_HEADER 의 NumberOfRvaAndSizes 에서 정해진다.

✱ 마지막 Entry (Index 15) 는 항상 0 이므로 총 15개가 사용된다.



[그림 2 - 2 - 7] IMAGE_DATA_DIRECTORY

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

#define IMAGE_NUMBEROF_DIRECTORY_ENTRIES    16
```

[그림 2 - 2 - 8] IMAGE_DATA_DIRECTORY 구조체

➡ VirtualAddress 와 Size 는 각각 해당 Index 의 미리 지정된 Section 또는 Block 의 정보에 대한 시작 주소와 그 크기를 가리키는 RVA 이다.

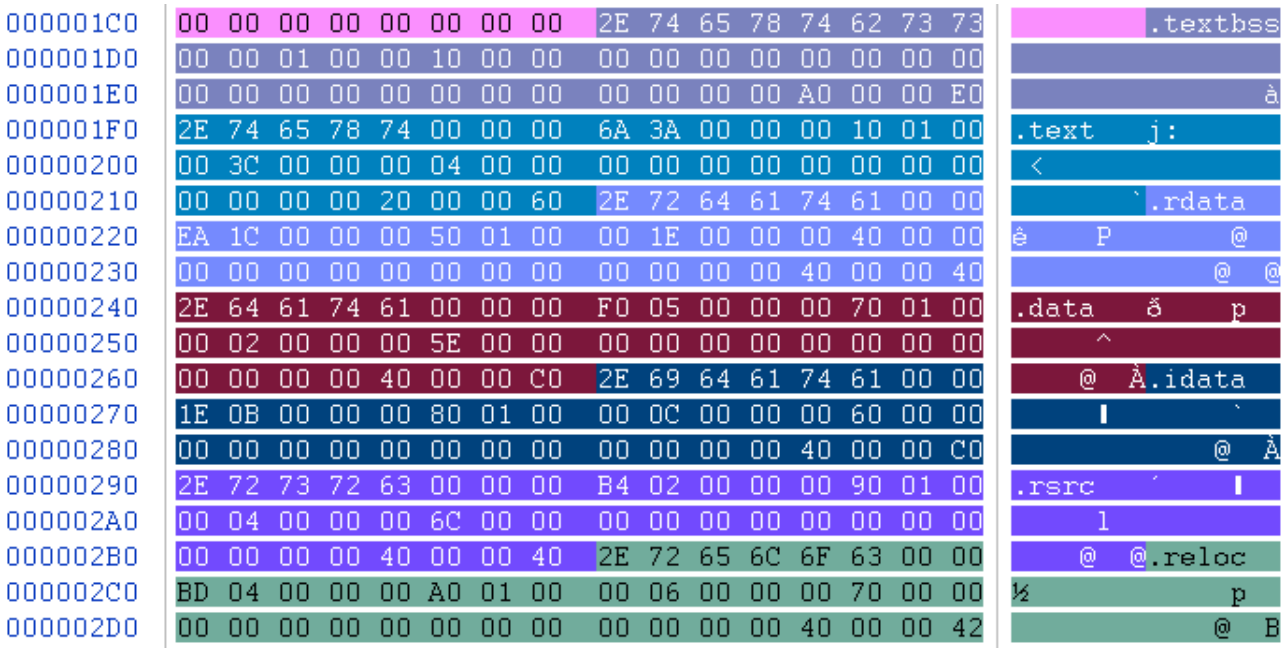
➡ 각 Index 의 의미는 아래 표와 같다.

Entry	Index	설명(~ 시작번지를 가리킨다.)
_EXPORT	0	IMAGE_EXPORT_DIRECTORY 구조체 배열
_IMPORT	1	IMAGE_IMPORT_DIRECTORY 구조체

Entry	Index	설명(~ 시작번지를 가리킨다.)
_RESOURCE	2	IMAGE_RESOURCE_DIRECTORY 구조체
_EXCEPTION	3	IMAGE_RUNTIME_FUCTION_ENTRY 구조체
_SECURITY	4	WIN_CERTIFICATE 구조체
_BASERELOC	5	기준 재배치 정보
_DEBUG	6	IMAGE_DEBUG_DIRECTORY 구조체 배열
_ARCHITECTURE	7	IMAGE_ARCHITECTURE_HEADER 구조체 배열
_GLOBALPTR	8	GP 로 사용되는 RVA 로 IA-64 에서 사용된다.
_TLS	9	Thread Local Storage 초기화 Section
_LOAD_CONFIG	10	IMAGE_LOAD_CONFIG_DIRECTORY 구조체
_BOUND_IMPORT	11	DLL Binding 과 관련된 정보를 담고있다.
_IAT	12	첫번째 IAT 의 시작번지
_DELAY_IMPORT	13	ImgDelayDescr 구조체 배열
_COM_DESCRIPTOR	14	.NET II DLL 용 최상위 정보의 시작 번지

[표 2 - 2 - 3] IMAGE_DATA_DIRECTORY 의 Index 의미

3) IMAGE_SECTION_HEADER 구조



[그림 2 - 3 - 1] IMAGE_SECTION_HEADER

```
#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;

#define IMAGE_SIZEOF_SECTION_HEADER          40
```

[그림 2 - 3 - 2] IMAGE_SECTION_HEADER 구조체

➡ IMAGE_SECTION_HEADER 의 갯수는 IMAGE_FILE_HEADER 의 NumberOfSections Field 값 에 의해 결정된다.

Field	설명
BYTE Name [IMAGE_SIZEOF_SHORT_NAME]	Section 의 ASCII 이름 ✱8 Byte 이상 시에는 문자열을 자름
DWORD PhysicalAddress or Virtual Address	PE 의 경우 Code 와 Data 의 실제 Byte 수를 담음

Field	설명
DWORD VirtualAddress	Memory 상에서 본 Section 의 시작 주소
DWORD SizeOfRawData	Mics.VirtualSize Field 값에 대한 File Alignment 값의 배수로 Round-up 된 값
DWORD PointerToRawData	해당 Section 의 PE File 상에서 시작하는 실제 File Offset 값으로, File Alignment 값의 배수
DWORD PointerToRelocations	재배치 File Offset
DWORD PointerToLinenumbers	COFF Style 의 Line 번호를 위한 File Offset
WORD NumberOfRelocations	IMAGE_RELOCATION 구조체 배열의 원소 개수
WORD NumberOfLinenumbers	IMAGE_LINENUMBER 구조체 배열의 원소 개수
DWORD Characteristics	해당 Section 의 속성을 나타내는 Flag 의 집합 ✱ IMAGE_SCN 으로 시작하는 Flag 들

[표 2 - 3 - 1] IMAGE_SECTION_HEADER 의 Field

PE File 에서 각 Section 의 실제 내용을 확인하고자 한다면, PointerToRawData 가 가리키는 위치로 File Offset 을 이동 시키면 그곳에서 부터 SizeOfRawData 의 Byte 수 만큼이 해당 Section 의 실제 내용이 된다.

실제로 가상 주소 공간에 Mapping 되었을 때의 RVA 는 VirtualAddress 가 되며, Mapping 후의 실제 시작 Pointer 는 VirtualAddress 값에 IMAGE_OPTIONAL_HEADER 의 ImageBase Field 값을 더하면 된다.

✱ Mapping 된 후에 이 Section 이 차지하고 있는 Memory 상의 크기는 VirtualSize 에 있다.

3. Code & Data Section

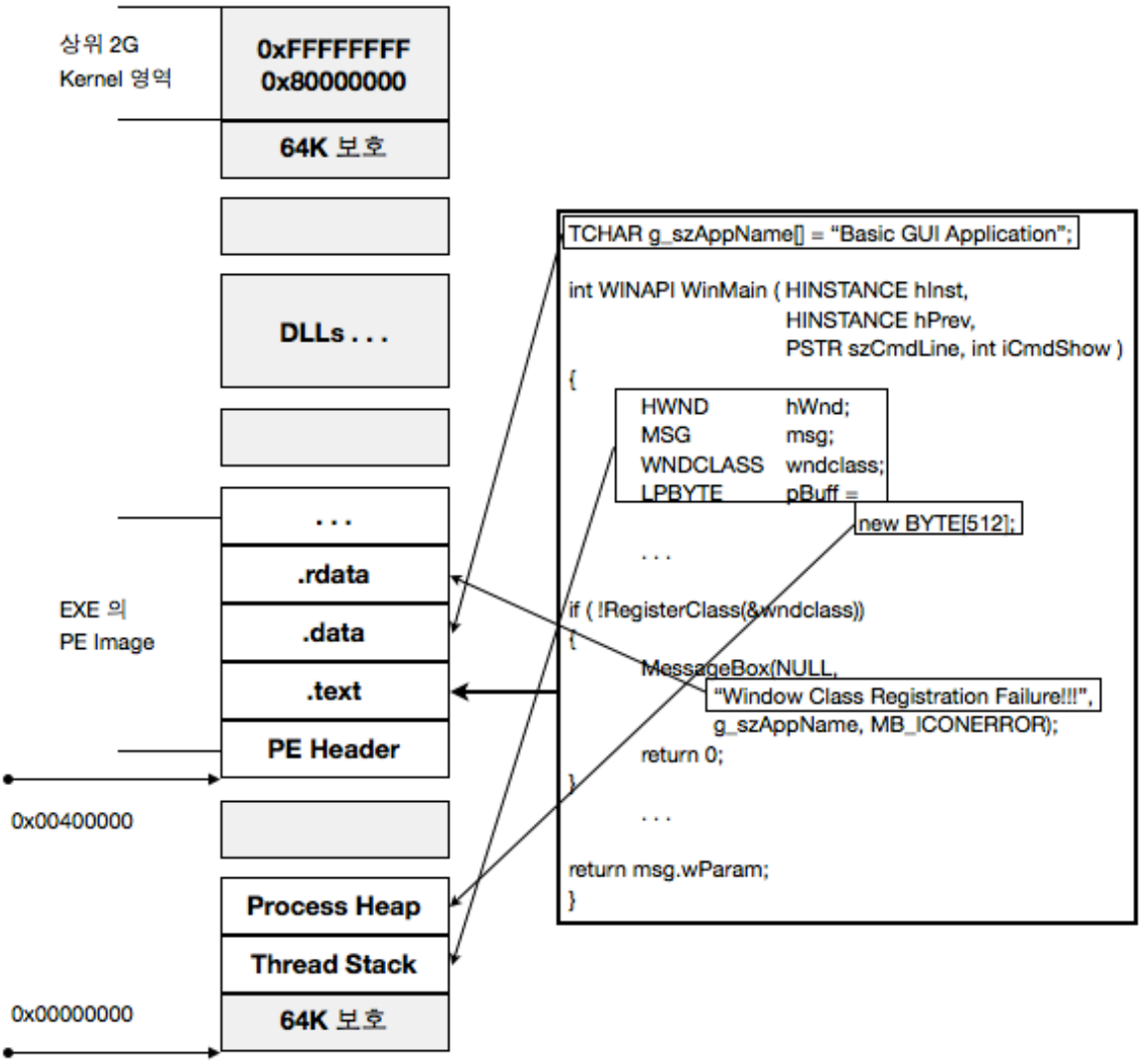
1) Basic

Program Instance 자체에 할당된 Memory 에 대하여 크게 4 영역 으로 나눌 수 있다.

✱ Program Code 가 모여 있는 Code 영역, 전역, 정적 변수 나 문자열 상수 등이 저장되는 Data 영역, 지역 변수나 함수의 매개 변수를 위한 Stack, 이루어지는 동적 Memory 할당을 위한 Heap

Code 영역은 PE 가 가상 주소 공간에 Mapping 되었을 때의 .text Section 이 되며, Data 영역은 가상 주소 공간에 Mapping 된 .data Section 과 .rdata Section 에 해당한다.

✱ PE 의 .text Section 과 .data & .rdata Section 이 Memory 상에 Mapping 되면서 각 Section 자체가 Code 와 Data 영역의 역할을 하게 된다.



[그림 3 - 1 - 1] Win32 Process 의 가상 주소 공간 내에서의 PE

➡ Load 전 4G 의 가상 주소 공간 생성 후 PE 전체 Mapping 에 충분한 공간을 Reserve
➡ Mapping 시 IMAGE_SECTION_HEADER 의 VirtualAddress Field 가 가리키는 값 (기준 주소 + RVA) 에 VirtualSize Field 값 만큼 공간을 Commit 한 후 해당 Section 을 Mapping
➡ Process Heap 과 Thread Stack 은 IMAGE_OPTIONAL_HEADER 의 Field 참조

2) Code Section

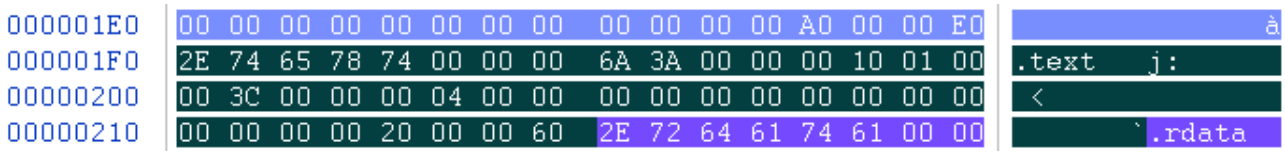
(i) .text Section

IMAGE_SECTION_HEADER 의 .text 라고 되어있는 Entry 가 가리키는 곳에 위치한 Block 으로써, 작성된 해당 PE 내의 모든 Code 가 이곳에 위치하게 된다.

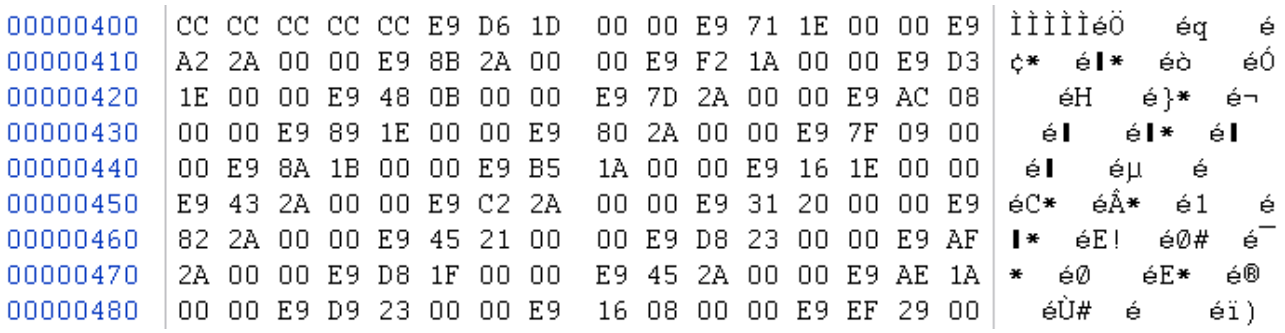
함수로 정의된 각 내용들은 기계어 Code 로 번역된다.

✱ 이는 Thread 가 실행할 하나의 명령 단위를 이루며, 실행될 명령어의 번지 값은 EIP 에 있다.

✱ 기계어 Code 는 PE File 의 .text Section 위에 차례로 기록되어 최종적인 PE File 의 .text Section 을 구성하게 된다.



[그림 3 - 2 - 1] .text Section 의 IMAGE_SECTION_HEADER



[그림 3 - 2 - 2] .text Section

➡ 0x204 부분부터 4 Byte (0x00000400)는 PointerToRawData Field 로써, 해당 PE File 의 .text Section 의 시작 주소가 된다.

➡ .text Section 을 분석한다는 것은 그것의 기계어 Code 를 Assembly 어로 역표현 하는 것

(ii) 예제 (WinMain 호출 지점 찾아가기)

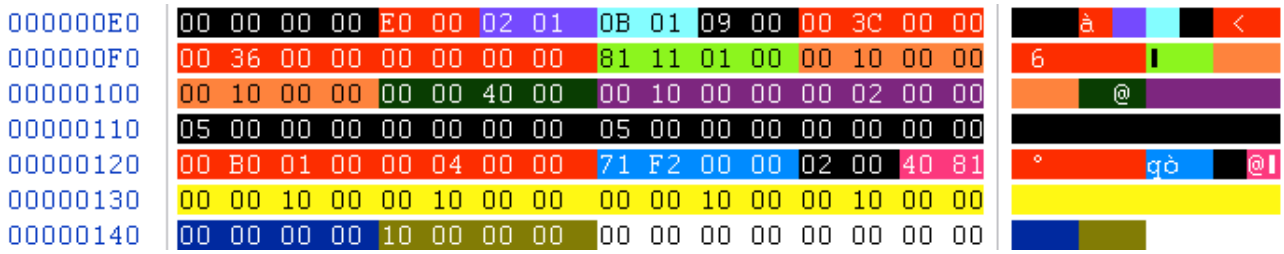
```
1   int WINAPI WinMain ( HINSTANCE hInstance, HINSTANCE hPrevInst,
2                        PSTR szCmdLine, int iCmdShow );
```

[그림 3 - 2 - 3] WinMain 함수

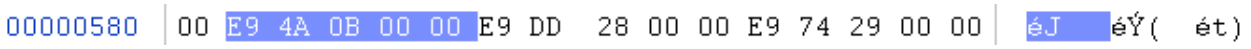
➡ hInstance 는 해당 Program 의 Instance Handle 으로, Process 에 속한 여러가지 Resource 의 준거점 이 된다.

✱ 이 값은 IMAGE_OPTIONAL_HEADER 의 ImageBase Field 에 저장되어 있는 값으로 EXE 일 경우 Default 로 0x00400000, DLL 일 경우 0x00100000 이 된다.

IMAGE_OPTIONAL_HEADER 의 AddressOfEntryPoint Field 는 Loader 가 실행을 개시할 주소를 나타내며, 이는 해당 PE 가 Memory 에 Mapping 된 후 최초 실행하는 Entry 함수이다. 이를 따라가 보겠다.



[그림 3 - 2 - 4] IMAGE_OPTIONAL_HEADER



[그림 3 - 2 - 5] AddressOfEntryPoint 에 해당하는 PE Dump

➡ 0xF8 로부터 4Byte (0x00011181) 지점이 AddressOfEntryPoint Field 이며 Memory 에 Mapping 된 후의 주소는 0x00411181 이 된다.
➡ 해당 주소로 가보면 E9 4A 0B 00 00 이 적혀있다.
✱ 이는 역 Assemble 규칙에 따라서 0xE9 : JMP 에서 0x0B4A 를 더한 번지로 JMP 라는 의미 이며, Memory Mapping 후 JMP 00411CD0 과 같은 것을 알 수 있다.

실제 Memory Mapping 후 확인

```
.text:00411181 ; int __cdecl start()
.text:00411181                 public start
.text:00411181 start           proc near
.text:00411181                 jmp     _WinMainCRTStartup
.text:00411181 start           endp
```

[그림 3 - 2 - 6] 0x00411181

```
.text:00411CD0 ; int __cdecl WinMainCRTStartup()
.text:00411CD0 _WinMainCRTStartup proc near             ; CODE XREF: start↑j
.text:00411CD0                 mov     edi, edi
.text:00411CD2                 push    ebp
.text:00411CD3                 mov     ebp, esp
.text:00411CD5                 call    j____security_init_cookie
.text:00411CDA                 call    __tmainCRTStartup
.text:00411CDF                 pop     ebp
.text:00411CE0                 retn
.text:00411CE0 _WinMainCRTStartup endp
```

[그림 3 - 2 - 7] 0x00411CD0

➡ 결국 Main Thread 는 0x00411CD0 으로 JMP 하여 WinMainCRTStartup 함수를 수행하게 되며, 이 함수를 따라가보면 WinMain 을 호출한다는 것을 확인할 수 있다.

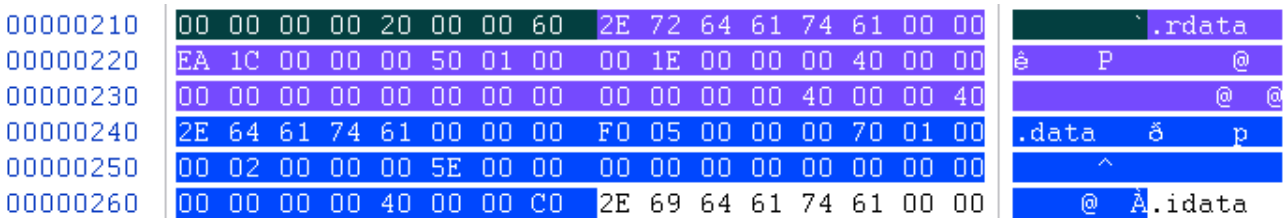
3) Data Section

일반적으로 .data Section 과 읽기 전용인 .rdata Section 그리고 초기화 되지 않은 Data 를 보관하던 .bss -> .textbss Section 이 있다.

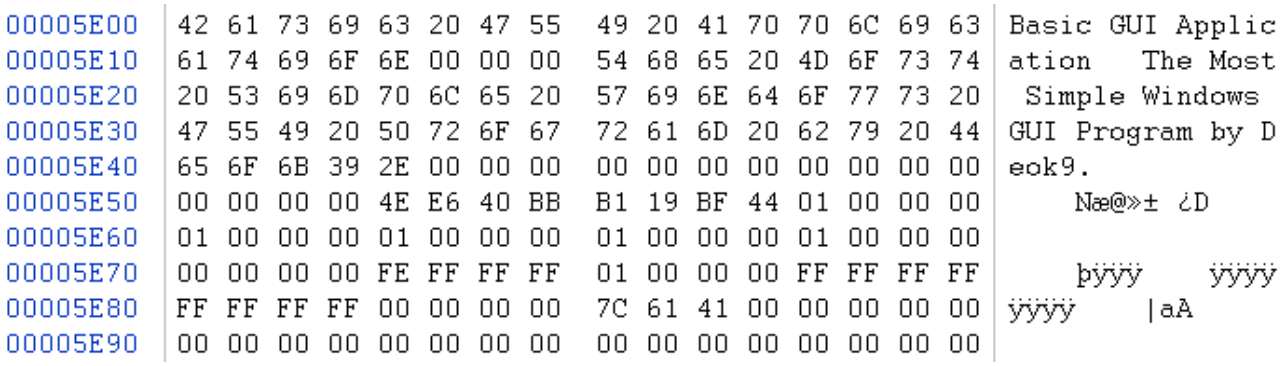
✱ .textbss Section 은 Debugging mode 에서만 생성되며 거의 사용 않는다.

(i) .data Section 과 예제 (지역변수의 초기값 찾아보기)

전역/정적 변수를 정의하게 되면 모두 .data Section 에 위치하게 되며 이는 Process Stack 에 영역이 할당되기 때문에 PE 와는 관계가 없다.



[그림 3 - 3 - 1] .data Section 의 IMAGE_SECTION_HEADER



[그림 3 - 3 - 2] .data Section

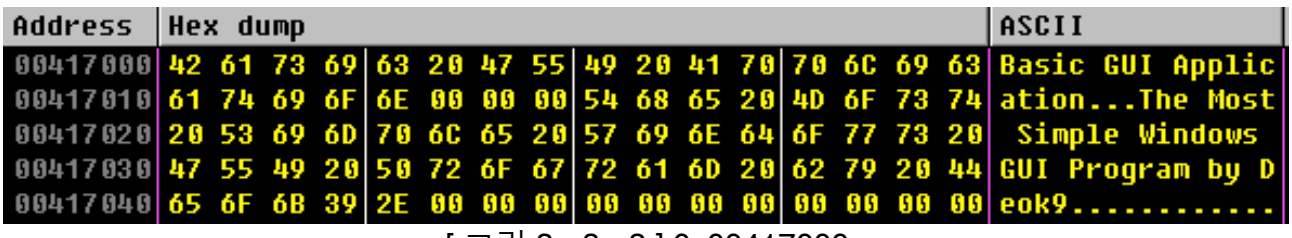
➡ 0x254 부분부터 4 Byte (0x00005E00)는 PointerToRawData Field 로써, 해당 PE File 의 .data Section 의 시작 주소가 되며, 0x520 부분부터 4Byte (0x00000200) 은 SizeOfRawData Field 로써, 해당 Section 의 PE File 상의 크기를 나타낸다.

➡ 두 변수의 VAS 상의 번지값을계산해 보면, 아래와 같이 될 것이다.

✱ 0x00400000 (Base Address) + 0x00005E00 (Data Section Offset) + 0x00017000 (VirtualAddress) - 0x00005E00 (PointerToRawData) = 0x00417000

✱ 0x00400000 (Base Address) + 0x00005E18 (Data Section Offset) + 0x00017000 (VirtualAddress) - 0x00005E00 (PointerToRawData) = 0x00417018

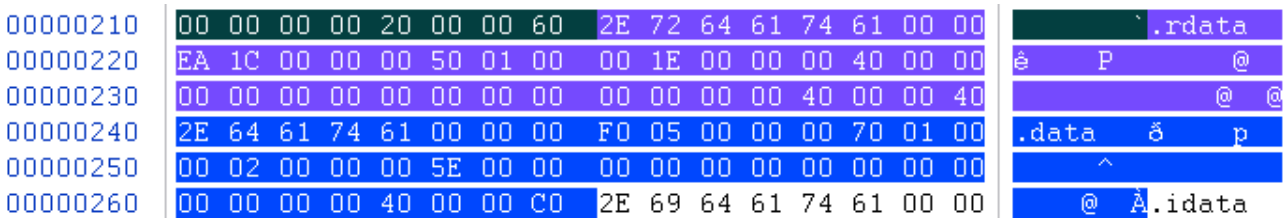
실제 Memory Mapping 후 확인



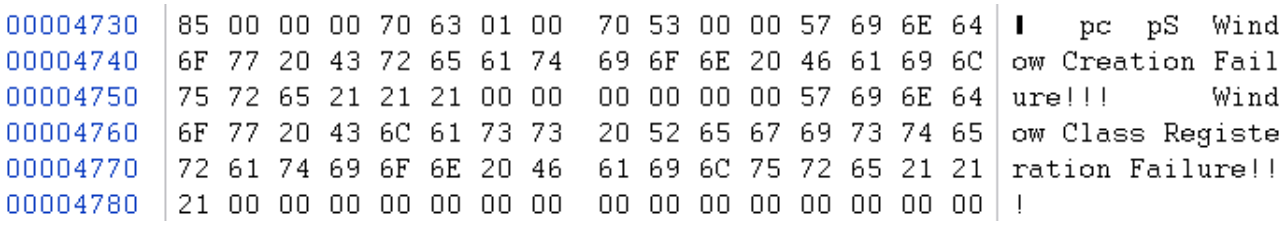
[그림 3 - 3 - 3] 0x00417000

(ii) .rdata Section 과 예제 (문자열 상수 찾아보기)

상수로 정의한 것들, 문자열 Pointer 에 변수를 담지 않고 직접 문자열을 지정해준 경우, Runtime Library 에서 사용하는 Error Message 등이 .rdata Section 에 저장된다.



[그림 3 - 3 - 4] .rdata Section 의 IMAGE_SECTION_HEADER

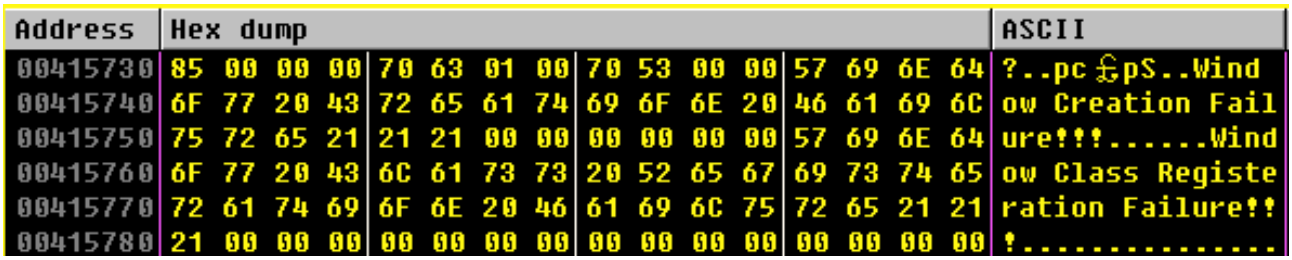


[그림 3 - 3 - 5] .rdata Section

➡ 0x22C 부분부터 4 Byte (0x00004000)는 PointerToRawData Field 로써, 해당 PE File 의 .rdata Section 의 시작 주소가 되며, 0x228 부분부터 4Byte (0x00001E00) 은 SizeOfRawData Field 로써, 해당 Section 의 PE File 상의 크기를 나타낸다.
➡ Characteristics Field (0x40000040) 를.data Section 의 0xC0000040 과 비교해보면 쓰기 속성이 빠진것을 알 수 있다.
➡ 변수의 VAS 상의 번지값을계산해 보면, 아래와 같이 될 것이다.

✱ 0x00400000 (Base Address) + 0x00004730 (Data Section Offset) + 0x00015000 (VirtualAddress) - 0x00004000 (PointerToRawData) = 0x00415730

실제 Memory Mapping 후 확인



[그림 3 - 3 - 6] 0x00415730

Debug Section, DLL 의 Export Section 등 크기가 작고 읽기 전용 속성을 띄는 Section 들은 모두 .rdata Section 에 병합되어 버리고, 뿐만 아니라 다른 Section 의 일부 .rdata Section 에 보관하는 경우도 있다.