기술문서	08.10.24. 작성

PE 구조 분석

Windows 계열 OS 의 파일 구성 (PE: Portable Executable)

작성자:동명대학교 THINK 강동현(cari2052@gmail.com)

동명대학교 정보보호동아리

0. 시작하면서

본 문서는 파일 구조에 관한 좀더 세심하게 관찰 하기 위한 사람과, 프로그램 역분석 공부를 시작하는 사람들에게 도움이 되고자 하는 동기로 시작하여 작성된 문서이다.

Windows 계열 OS 에서 사용하는 PE 구조에 관한 전체적인 설명을 담고 있으며 각 구조체들의 필드 의미와 설명을 첨가하였다.

본 문서를 이해하려면 아래와 같은 기반 지식이 요구된다.

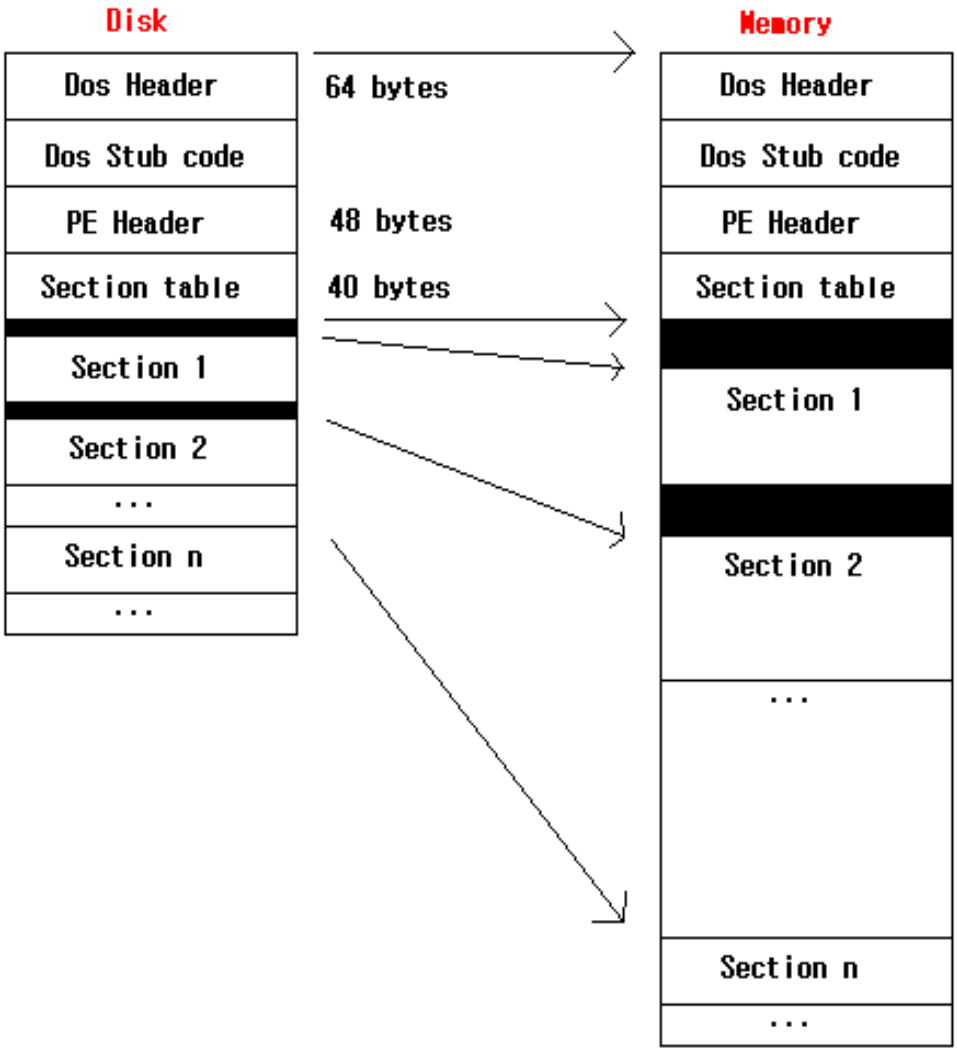
- **구조체, 공용체**
- **HEX 코드에 관한 기본 지식**
- **메모리의 가상주소 개념.**
- Little Endian 과 Big Endian.

본 문서에서 아래와 같은 프로그램을 사용하였다.

- VXI32.exe
- function_if.exe , function_printf.exe (by Visual Studio 2005)

1. PE 구성

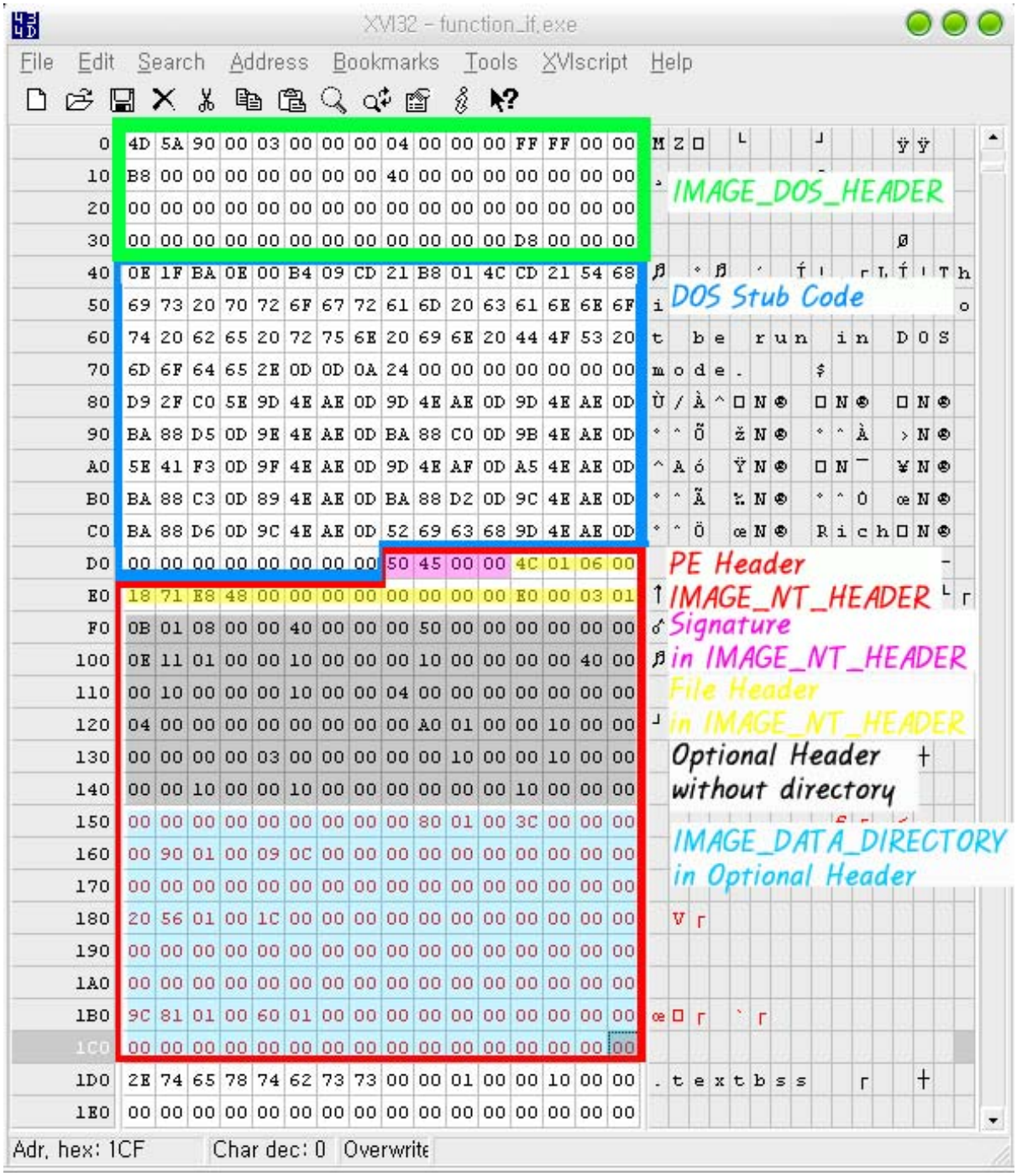
DOS Header, DOS Stub code, PE Header, Section Table + 1 개 이상의 section 으로 이루어져 있다.



<그림 1> PE구조

디스크상의 프로그램 구성과 메모리상의 프로세스의 구성은 비슷하다. 하지만 메모리상 프로세스의 구조의 Section 들은 유동적 공간에 할당 받게 된다.

이 모든 데이터가 파일 형식의 구조로 이루어져 있으며, 내부 HEX 코드들의 각 위치에 해당하는 의미는 미리 정의되어 있는 구조체로 데이터를 인식한다.



<그림 2> 파일 내부 PE구조

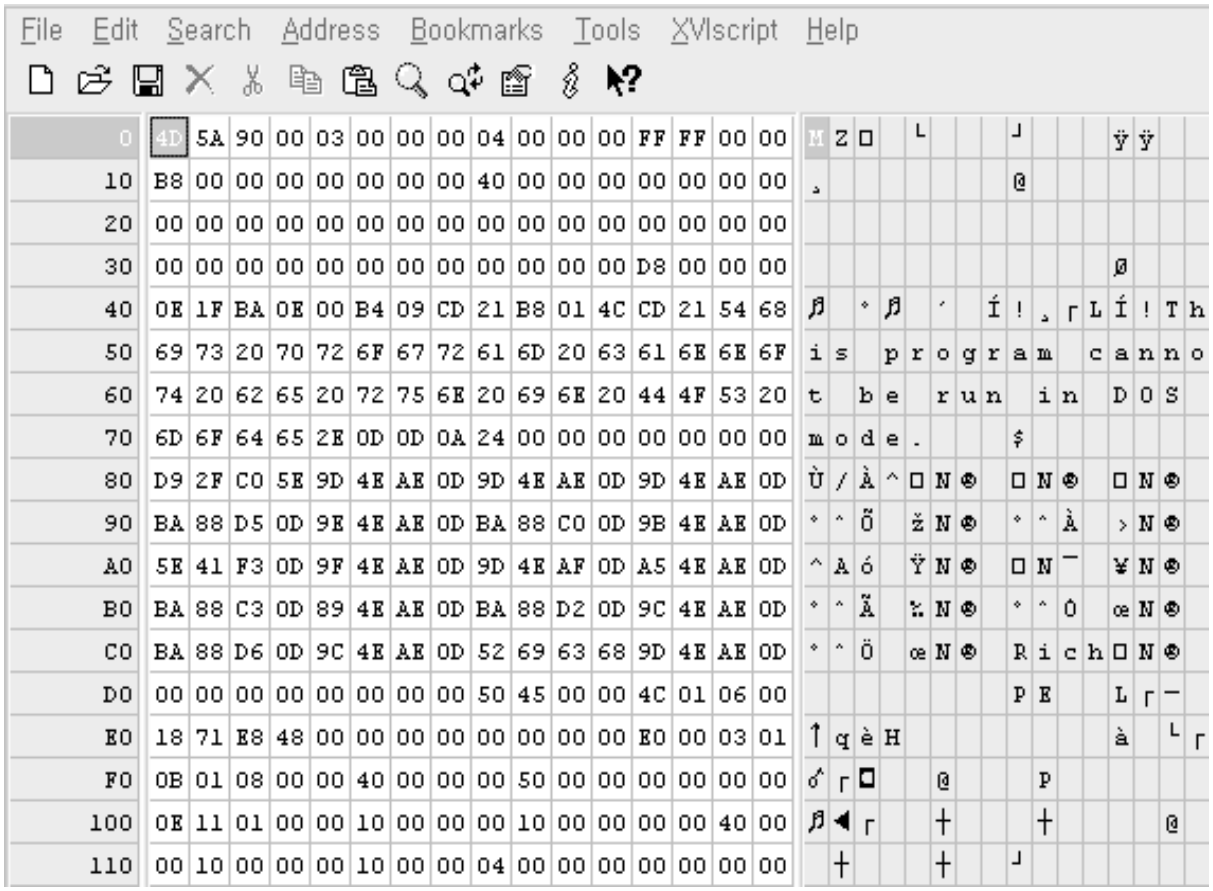
위의 <그림 2>는 디스크 상의 PE 구조(DOS Header, DOS Stub Code, PE Header 등 그 위치)를 표시하고 있다.

위의 <그림 2>의 박스 이후 부터(주소 1D0 부터) 40bytes 까지는 section table 의 영역이다.

2. DOS Header

64bytes 의 고정된 크기를 가지며, 디스크상의 첫부분과 메모리상 ImageBase 에 위치 하고 있다.

(아래의 그림은 XVI32.exe 로 간단한 if 문이 코딩 되어 있는 파일을 열어서 확인한 사진임.)



0	4D	5A	90	00	03	00	00	00	04	00	00	00	FF	FF	00	00
10	B8	00	00	00	00	00	00	00	40	00	00	00	00	00	00	00
20	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
30	00	00	00	00	00	00	00	00	00	00	00	00	D8	00	00	00
40	0E	1F	BA	0E	00	B4	09	CD	21	B8	01	4C	CD	21	54	68
50	69	73	20	70	72	6F	67	72	61	6D	20	63	61	6E	6E	6F
60	74	20	62	65	20	72	75	6E	20	69	6E	20	44	4F	53	20
70	6D	6F	64	65	2E	0D	0D	0A	24	00	00	00	00	00	00	00
80	D9	2F	C0	5E	9D	4E	AE	0D	9D	4E	AE	0D	9D	4E	AE	0D
90	BA	88	D5	0D	9E	4E	AE	0D	BA	88	C0	0D	9B	4E	AE	0D
A0	5E	41	F3	0D	9F	4E	AE	0D	9D	4E	AF	0D	A5	4E	AE	0D
B0	BA	88	C3	0D	89	4E	AE	0D	BA	88	D2	0D	9C	4E	AE	0D
C0	BA	88	D6	0D	9C	4E	AE	0D	52	69	63	68	9D	4E	AE	0D
D0	00	00	00	00	00	00	00	00	50	45	00	00	4C	01	06	00
E0	18	71	E8	48	00	00	00	00	00	00	00	00	E0	00	03	01
F0	0B	01	08	00	00	40	00	00	00	50	00	00	00	00	00	00
100	0E	11	01	00	00	10	00	00	00	10	00	00	00	00	40	00
110	00	10	00	00	00	10	00	00	04	00	00	00	00	00	00	00

<그림 3> MZ in DOS Header

DOS Header 의 처음 2bytes 는 DOS Header 구조체의 e_magic 인자로 대부분 MZ 로 시작한다. 이는 DOS 개발자 가운데 한명인 Mark Zbikowski 의 이니셜로 DOS Header 의 시그니쳐로 사용하고 있다. 그리고 DOS Header 의 마지막 4bytes 는 DOS Hedaer 구조체의 e_lfanew 로 PE Header 의 주소값을 표기하고 있다.

2-1. DOS Header 구조체.

```
typedef struct _IMAGE_DOS_HEADER {
        WORD e_magic;
        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;
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

이전 설명한 첫 2bytes 로 DOS Header 의 식별자와, 마지막의 e_lfanew 값으로 PE Header 주소를 알아볼 수 있다.

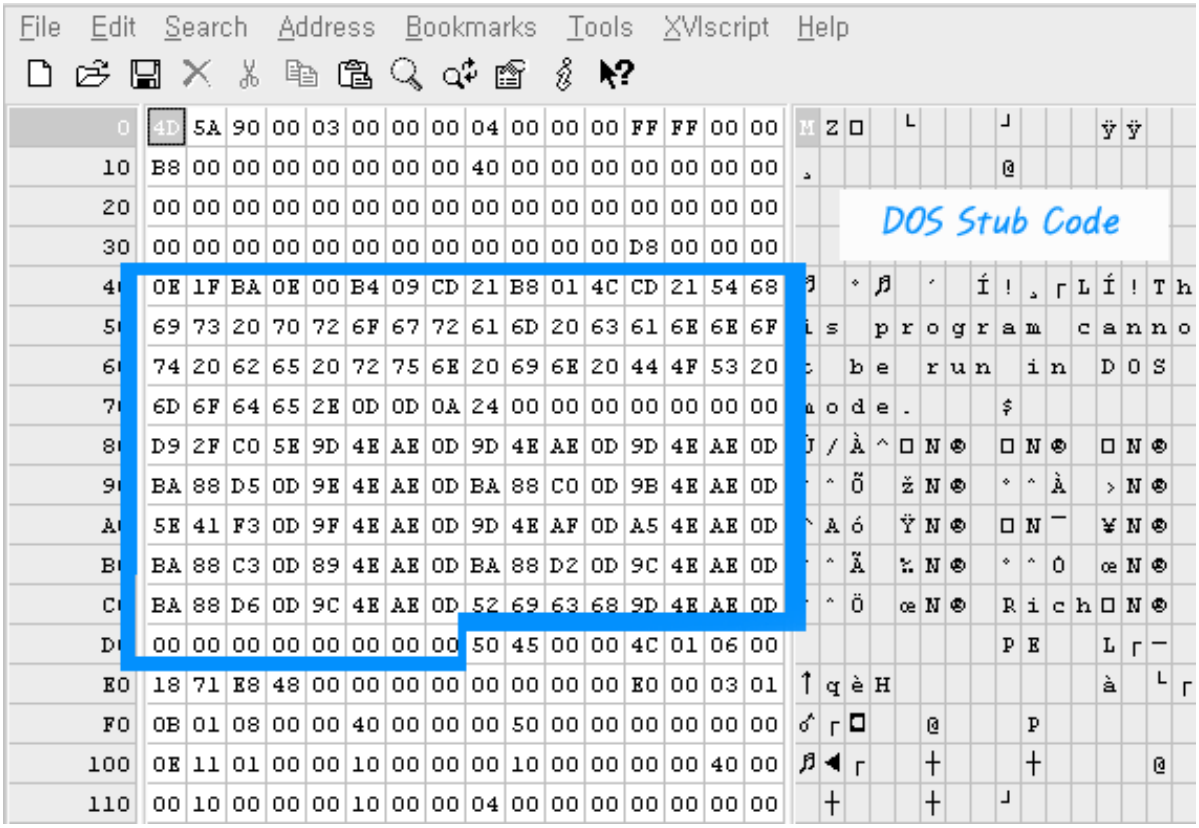
WORD 와 LONG 은 각각 16 비트 32 비트로 2bytes, 4bytes 크기를 가지고 있다. 이를 계산해보면 WORD 형 데이터 분할 30 개, LONG 1 개로 60bytes + 4bytes = 64bytes 의 고정된 공간을 할당 받게 된다는 것을 알 수 있다..

3. DOS Stub Code

시작점(파일의 첫 부분이나 메모리상의 ImageBase)에서부터 DOS Header 의 사이즈인 64bytes 만큼 떨어진곳에서 찾을 수 있다.

DOS mode 에서 실행되며 현재 사용하지 않는 부분이다. 현재는 사용하지 않으며 Hex 에디터들로 분석해보면 "This program cannot be run in DOS mode.."이라는 문자열을 볼 수 있다.

32bit 프로그램을 도스(16bit)에서 실행할 때 위의 문자열을 출력해주는데 쓰이고 있다.



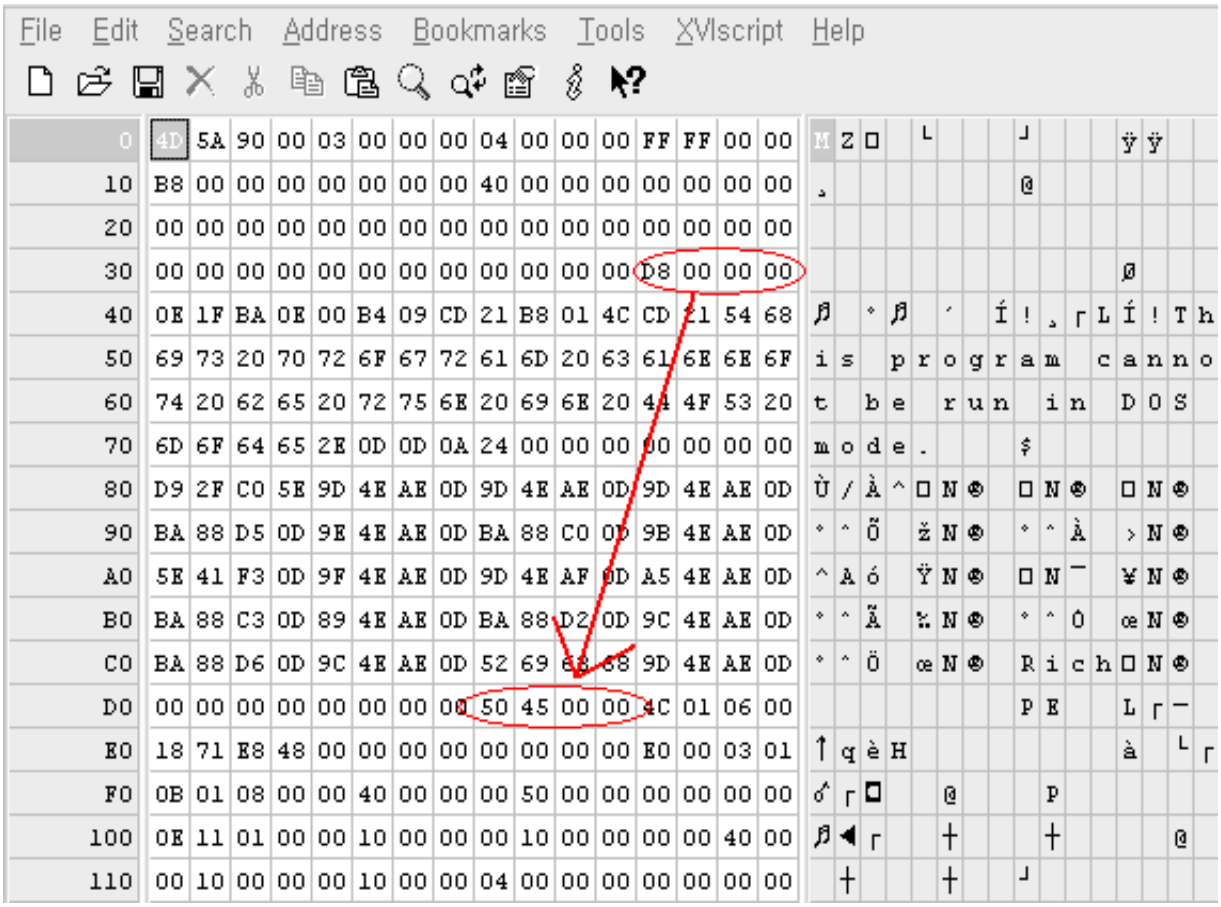
<그림 4> DOS Stub Code

4. PE Header

DOS Header 에 있는 e_lfanew 값을 확인 하여 알 수 있다.

-> e_flanew : 4bytes 크기의 LONG 변수로서 DOS Header 의 마지막에 위치한다. 즉, 64bytes 의 DOS Header 시작점에서 60bytes 만큼 옮기면 e_flanew 값을 확인할 수 있다.

(ps. PE header 는 50 45 00 00 부터 시작한다.)



<그림 5> PE Header 시작 위치

4-1. PE Header 구조체

```
typedef struct _IMAGE_NT_HEADERS{
        DWORD Signature;
        IMAGE_FILE_HEADER FileHeader;
        IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

PE Header 구조체는 Signature, FIleHeader, OptionalHeader 으로 이루어져 있으며, 각각 프로그램의 기본 속성정보가 들어있다.

FileHeader 와 OptionalHeader 는 각각 IMAGE_FILE_HEADER, IMAGE_OPTIONAL_HEADER 구조체로 이루어져 있다.

4-2. IMAGE_FILE_HEADER 구조체

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

PE Header 구조체의 File Header 는 프로그램의 종류,호환 플랫폼 그리고 섹션의 개수 등의 정보가 입력되어 있다.

Machine : 어떤 플랫폼에 사용되는지를 표기.

NumberOfSections: 프로그램의 섹션의 개수.

TimeOfStamp: 파일 생성 날짜

SizeOfOptionalHeader: IMAGE_OPTIONAL_HEADER 구조체의 크기

Characteristics: 파일의 종류에 대한 플래그 (exe 파일 or dll 파일)

4-3. IMAGE_OPTIONAL_HEADER 구조체

```
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;
```

PE 구조체의 Optional Header 는 프로그램의 논리적 구조에 대한 중요한 정보들이 저장되어 있다.
30 개의 필드와 하나의 데이터 디렉토리로 구성되어 있으며 중요한 몇몇 필드만 이해하면 된다.

Magic: Optional Header 를 구분하는 시스니쳐
SizeOfCode : 보통 .text 섹션의 CPU 실행 기계어 코드의 전체크기
SizeOfInitiallzedData : 섹션영역에서 읽기,쓰기가 불가능하고 데이터가 초기화되어 있는 섹션 영역들의 크기
SizeOfUninitiallzedData : SizeOfInitiallzeData 의 반대 개념
AddressOfEntryPoint : PE 파일이 메모리에 로드된 후 가장 먼저 실행되어야 하는 코드의 주소
BaseOfCode : 코드영역의 첫번째 바이트 주소
BaseOfData : .data 섹션의 시작 주소 (RVA 값[1])
ImageBase : PE 파일이 메모리에 매필될 시작 주소.
보통 exe 파일은 0x00400000, dll 파일은 0x10000000 값을 가진다.
SectionAlignment : 메모리상에 올려진 후의 섹션의 배치 간격[2]
FileAlgnment : SectionAlignment 와 유사하며, PointerToRawData 에 기준으로 사용한다.
MajorSubsystemVersion, MinorSubsystemVersion :

[1] RVA(Relative Virtual Address): 기준이 되는 가상주소로부터 얼마만큼 떨어져 있는 위치인지 나타내는 것.
[2] SectionAlignment의 값 단위로 섹션의 크기를 할당한다.

Win32 애플리케이션인 경우 MajorSubsystemVersion 은 4
MinorSubsystem 은 0 이어야 한다.

Subsystem : Console 용 애플리케이션인 경우 0x3,
GUI 용 애플리케이션인 경우 0x2 값을 가진다.
SizeOfImage : 메모리 상에 로드되는 PE 파일의 총 크기
SizeOfHeader : 디스크 상의 모든 헤더의 총 사이즈이다.
SizeOfStackReserve : 프로그램에서 예약할 stack 의 크기
SizeOfStackCommit : 프로그램에서 commit 할 stack 의 크기
SizeOfHeapReserve : 프로그램에서 예약할 Heap 의 크기
SizeOfHeapCommit : 프로그램에서 commit 할 Heap 의 크기
LoaderFlags : NumberOfRvaAndSizes 와 함께 안티 리버싱에 사용.
NumberOfRvaAndSizes : 데이터 디렉토리 엔트리 개수.
DataDirectory : Export table, Import table, Resource 영역, Exception 영역,
디버그 영역 등을 접근할 수 있는 주소정보의 배열.

4-4. IMGAE_DATA_DIRECTORY 구조체

IMAGE_DATA_DIRECTORY 구조체는 IMAGE_OPTIONAL_HEADER 구조체의 마지막 필드이다. #define 되어 있는 IMAGE_NUMBEROF_DIRECTORY_ENTRIES 값을 가져와 배열을 만든다. 보통 16 으로 정의되어 있다.

또한 PE 파일에서 중요한 역할을 하는 개체들의 위치와 크기에 대한 정보를 담고 있다.

IMAGE_NUMBEROF_DIRECTORY_ENTRIES

```
Typedef struct _IMAGE_DATA_DIRECTORY{
        DWORD VirtualAddress;
        DWORD VirtualSize;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Data Directory 구조체 배열의 구성

```
#define IMAGE_DIRECTORY_ENTRY_EXPORT          0
#define IMAGE_DIRECTORY_ENTRY_IMPORT          1
#define IMAGE_DIRECTORY_ENTRY_RESOURCE        2
#define IMAGE_DIRECTORY_ENTRY_EXCEPTION       3
#define IMAGE_DIRECTORY_ENTRY_SECURITY        4
#define IMAGE_DIRECTORY_ENTRY_BASERELOC       5
#define IMAGE_DIRECTORY_ENTRY_DEBUG           6
#define IMAGE_DIRECTORY_ENTRY_ARCHITECTURE    7
#define IMAGE_DIRECTORY_ENTRY_GLOBALPTR       8
#define IMAGE_DIRECTORY_ENTRY_TLS             9
#define IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG     10
#define IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT    11
#define IMAGE_DIRECTORY_ENTRY_IAT             12
#define IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT    13
#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR  14
```

IMAGE_DATA_DIRECTORY 의 DataDirectory 의 마지막 16 번째 배열은 NULL 값을 가지며 이는 메모리상에서 DataDirectory 정의가 끝났음을 의미한다.

5. IMPORT

프로그램이 외부 DLL 파일을 import 하기 위해 사용하는 테이블이다. 외부 DLL 에 대한 정보를 담고 있다.

5-1 IMPORT Table

IMAGE_IMPORT_DESCRIPTOR 구조체로 이루어져 있으며 ILT(Import lookup Table)과 IAT(Import Address Table)에 관한 정보를 가지고 있다.

Optional Header 구조체의 마지막 필드인 DataDirectory배열의 두번째(DataDirectory[1])필드가 이 Import 테이블의 주소를 가리키고 있다.

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

OriginalFirstThunk	: ILT를 가리키는 RVA값이다. IMAGE_THUNK_DATA로 구성된 구조체이다
TimeDateStamp	: 바인딩 전 0, 바인딩 후 -1 로 설정된다.
ForwarderChain	: 바인딩 전 0, 바인딩 후 -1 로 설정된다.
FirstThunk	: IAT를 가리키는 RVA값이다. IMAGE_THUNK_DATA로 구성된 구조체이다. 바인딩 후 실제 함수의 주소로 사용된다.

5-2 IMAGE_THUNK_DATA 구조체

ILT와 IAT가 구성되어 있는 구조체로 바인딩 전에는 ILT와 IAT는 필드의 Ordinal 값을 사용하고, 바인딩 후 IAT는 필드의 Function으로 사용한다. (부록 <1> 참조)

```
typedef struct_IMAGE_THUNK_DATA32
{
        union{
                DWORD ForwarderString;
                DWORD Function;
                DWORD Ordinal;
                DWORD AddressOfData;
        } u1;
} IMAGE_THUNK_DATA32
```

5-3 IMAGE_IMPORT_BY_NAME 구조체

바인딩 전 ILT 와 IAT 는 IMAGE_IMPORT_BY_NAME 구조체에 가리키며, 바인딩 후 IAT 만 실제 함수의 주소를 가리키게 된다.

```
typedef struct _IMAGE_IMPORT_BY_NAME {
        WORD Hint;
       BYTE Name[?];
} IMAGE_IMPORT_BY_NAME,*PIMAGE_IMPORT_BY_NAME;
```

Hint : Import 하는 함수가 추가될 때마다 1 씩 증가한다.

Name : 실제 함수 이름을 가리키는 RVA 값.

6. EXPORT

외부 DLL 파일들을 프로세스의 내부에 로드 하기위해 Export Table 을 사용한다.

6-1. EXPORT Table

```
typedef struct _IMAGE_EXPORT_DIRECTORY{
        DWORD       Characteristics;
        DWORD       TimeDateStamp;
        WORD        MajorVersion;
        WORD        MinorVersion;
        DWORD       Name;
        DWORD       Base;
        DWORD       NumberOfFunctions;
        DWORD       NumberOfNames;
        DWORD       AddressOfFunctions;
        DWORD       AddressOfNames;
        DWORD       AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Name : DLL 의 이름(ASCII 문자열) RVA 값

NumberOfFunctions : AddressOfFunctions 가 가리키는 RVA 배열 수

NumverOfNames : AddressOfNames 가 가리키는 RVA 배열 수

AddressOfFunctions : 함수의 실제 주소를 가리키는 RVA 값.

AddressOfNames : 함수 실제 이름을 가리키는 RVA 값.

AddressOfNameOrdinals : 함수의 Ordinal(서수)값 을 가리키는 RVA 값

6-2 EXPORT 방식

외부 DLL 파일 내부에서 특정 함수를 export 할때,Name 방식과 Ordinal 방식이 있다.

Name 방식은 특정 함수의 이름을 지정하여 해당 함수를 호출하는 방식이며, Ordinal 방식은 DLL 파일에서 16bits 로 각 함수들을 구분지어 사용하는 Ordinal 값을 이용하여 특정 함수를 호출하는 방식이다.

7. Section Table

Section table 은 PE Header 바로 뒤에 위치한다. 즉, 이를 찾으려면 PE Header 시작 주소에서 PE Header 사이즈 만큼 더해주면 그 위치가 Section table 정보가 저장되어 있을 것이다.
PE Header 는 PE signature (4bytes 고정) 과 File Header (20bytes 고정) 그리고 Optional Header (base size : 224bytes) 로 구성되어 있다. 정리하자면, 24bytes + Optional Header 사이즈가 된다.

각 섹션의 위치는 섹션 테이블에 저장되어 있으며 이는 Section Header 를 통해 확인 가능하다. Section 위치는 Section Header 속의 VirtualAddress 와 PointerToRawDate 라는 값에 offset 되어 있다. 각각 VirtualAddress 는 메모리상에서, PointerToRawDate 는 디스크 상에서의 offset 값이다. 즉, VirtualAddress 와 PointerToRawDate 값을 알아보고 섹션의 크기을 알아보면 각 섹션들의 위치와 정보를 알아낼 수 있다.

```
typedef struct _IMAGE_SECTION_HEADER{
        BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
        union{
           DWORD    PhsicalAddress;
           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;
```

VirtualAddress : 로드될 섹션의 가상주소. RAV 값
SizeOfRawData : 파일 상태에서의 섹션의 사이즈 값. file alignment 의 배수이어야 한다.
PointerToRawData : 파일 상태에서의 섹션 시작 위치.
Characteristics : 섹션의 속성값. (읽기, 쓰기, 읽기쓰기, 실행 등과 같은 속성)

속성값계산은 부록<2>의 표에 나와있는 Flag 속성을 보고 적절히 계산하여 사용하면 된다. 예를 들어 속성 INITIALIZED, MEM_READ,MEM_ WRITE 속성을 적용하고 싶다면 0x00000040 + 0x40000000 + 0x80000000 = 0xC0000040 값을 Characteristics 에 적용하면 된다.

8. 맺음말

0.시작하면서 에서 설명했듯이 본 문서는 파일 구조를 좀더 자세하게 알고 싶은 사람, 역분석(리버스) 공부를 시작하는 사람들에게 도움이 되고자 하는 동기로 시작된 문서이다. 하지만 좀더 역분석에 대해 공부한다면 앞으로 OS 에 관한 전반적인 지식을 쌓는 것을 추천한다.

본 문서에서 설명하는 PE 구조와 달리 ELF 구조 역시 기본적인 지식에 해당되므로 가능한 ELF 관련 지식도 쌓을 것을 추천한다.

저자 한마디 :

문서내용 중 잘못된 정보나 궁금한 사항은 E-mail 을 보내주세요. 확인 후 수정하도록 하겠습니다.

9. 참고 자료

MSDN (search: PE)

http://msdn.microsoft.com/en-us/magazine/cc301805.aspx

Zesrever 의 지식펌프

http://zesrever.xstone.org/category/지식뽐뿌질%20II(연재물)/조립하면서%20 배우는%20PE

PE 헤더에서 사용하는 구조체

http://www.openrce.org/reference_library/files/reference/PE%20Format.pdf

10. 부록

부록<1>

Binding

부록<2>

Characteristics

The characteristics of the image. The following values are defined.

Flag	Meaning
0x00000000	Reserved.
0x00000001	Reserved.
0x00000002	Reserved.
0x00000004	Reserved.
IMAGE_SCN_TYPE_NO_PAD 0x00000008	The section should not be padded to the next boundary. This flag is obsolete and is replaced by IMAGE_SCN_ALIGN_1BYTES.
0x00000010	Reserved.
IMAGE_SCN_CNT_CODE 0x00000020	The section contains executable code.
IMAGE_SCN_CNT_INITIALIZED_DATA 0x00000040	The section contains initialized data.
IMAGE_SCN_CNT_UNINITIALIZED_DATA 0x00000080	The section contains uninitialized data.
IMAGE_SCN_LNK_OTHER 0x00000100	Reserved.
IMAGE_SCN_LNK_INFO 0x00000200	The section contains comments or other information. This is valid only for object files.
0x00000400	Reserved.
IMAGE_SCN_LNK_REMOVE 0x00000800	The section will not become part of the image. This is valid only for object files.
IMAGE_SCN_LNK_COMDAT 0x00001000	The section contains COMDAT data. This is valid only for object files.
0x00002000	Reserved.
IMAGE_SCN_NO_DEFER_SPEC_EXC 0x00004000	Reset speculative exceptions handling bits in the TLB entries for this section.
IMAGE_SCN_GPREL 0x00008000	The section contains data referenced through the global pointer.
0x00010000	Reserved.
IMAGE_SCN_MEM_PURGEABLE 0x00020000	Reserved.
IMAGE_SCN_MEM_LOCKED 0x00040000	Reserved.
IMAGE_SCN_MEM_PRELOAD 0x00080000	Reserved.
IMAGE_SCN_ALIGN_1BYTES 0x00100000	Align data on a 1-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_2BYTES 0x00200000	Align data on a 2-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_4BYTES 0x00300000	Align data on a 4-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_8BYTES 0x00400000	Align data on a 8-byte boundary. This is valid only for object files.

IMAGE_SCN_ALIGN_16BYTES 0x00500000	Align data on a 16-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_32BYTES 0x00600000	Align data on a 32-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_64BYTES 0x00700000	Align data on a 64-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_128BYTES 0x00800000	Align data on a 128-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_256BYTES 0x00900000	Align data on a 256-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_512BYTES 0x00A00000	Align data on a 512-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_1024BYTES 0x00B00000	Align data on a 1024-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_2048BYTES 0x00C00000	Align data on a 2048-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_4096BYTES 0x00D00000	Align data on a 4096-byte boundary. This is valid only for object files.
IMAGE_SCN_ALIGN_8192BYTES 0x00E00000	Align data on a 8192-byte boundary. This is valid only for object files.
IMAGE_SCN_LNK_NRELOC_OVFL 0x01000000	The section contains extended relocations. The count of relocations for the section exceeds the 16 bits that is reserved for it in the section header. If the **NumberOfRelocations** field in the section header is 0xffff, the actual relocation count is stored in the **VirtualAddress** field of the first relocation. It is an error if IMAGE_SCN_LNK_NRELOC_OVFL is set and there are fewer than 0xffff relocations in the section.
IMAGE_SCN_MEM_DISCARDABLE 0x02000000	The section can be discarded as needed.
IMAGE_SCN_MEM_NOT_CACHED 0x04000000	The section cannot be cached.
IMAGE_SCN_MEM_NOT_PAGED 0x08000000	The section cannot be paged.
IMAGE_SCN_MEM_SHARED 0x10000000	The section can be shared in memory.
IMAGE_SCN_MEM_EXECUTE 0x20000000	The section can be executed as code.
IMAGE_SCN_MEM_READ 0x40000000	The section can be read.
IMAGE_SCN_MEM_WRITE 0x80000000	The section can be written to.