|  |  |  |
| --- | --- | --- |
| **4주차 1번 과제** | **제출일 : 2024.3.3** | **이 선 용** |

파이썬으로 PEViewer 구현하기

과제1. Windows의 Portable Executable(PE) 파일을 분석하고 내용을 시각적으로 표시하는 PEViewer를 파이썬으로 구현해보기

1. PE 파일의 다양한 구성 요소와 속성들을 해석하고 사용자에게 정보를 제공하는 프로그램을 작성하시오
2. PE파일은 Windows에서 실행 가능한 파일의 표준 형식이며, 이러한 파일은 실행 코드, 리소스 데이터, 메타데이터 등을 포함합니다.
3. 해당 프로그램에 대한 학습 노트를 500자 이상 작성 하시오.

!!주의사항!!

1. 직접 PEview를 다운받아서 임의의 .exe파일을 넣어 확인해보기를 추천함.
2. 사용자로부터 PE파일의 경로를 입력 받음.
3. 각 섹션의 번호가 출력되고, 사용자가 분석하길 원하는 섹션의 번호를 입력할 수 있어야함.
4. 섹션 번호를 입력하면 해당 섹션 번호에 맞게 출력되어야 함.
5. 파일을 읽고 PE 형식을 분석함. 이를 위해 PE파일의 구조에 대한 이해가 필요함( 예: DOS 헤더, NT헤더, 섹션 헤더 등)
6. 파일의 다양한 구성 요소(헤더, 섹션, 리소스 등)의 정보를 16진수로 추출하고 분석할 수 있어야 한다.

# CODE

'''

command에 help를 입력하면 매뉴얼이 출력됩니다.

showall을 입력하면 전체 정보가 출력됩니다.

원하는 정보를 검색하고 싶은 경우 해당하는 커맨드를 입력하세요.

DOS\_HEADER 등 구조체 내 멤버가 많은 헤더의 경우 몇 가지를 생략하였습니다.

학습노트에는 전부 공부한 내용이 기재되어있으니 양해바랍니다.

'''

import pefile

import os

def dos\_header(command):

if(command == "emagic"):

print(pe.DOS\_HEADER.e\_magic)

elif(command == "lfanew"):

print(pe.DOS\_HEADER.e\_lfanew)

elif(command == "cblp"):

print(pe.DOS\_HEADER.e\_cblp)

elif(command == "cp"):

print(pe.DOS\_HEADER.e\_cp)

elif(command == "crlc"):

print(pe.DOS\_HEADER.e\_crlc)

elif(command == "cparhdr"):

print(pe.DOS\_HEADER.e\_cparhdr)

elif(command == "minalloc"):

print(pe.DOS\_HEADER.e\_minalloc)

else:

file\_header(command)

def file\_header(command):

if(command == "machine"):

print(pe.FILE\_HEADER.Machine)

elif(command == "numsection"):

print(pe.FILE\_HEADER.NumberOfSections)

elif(command == "timestamp"):

print(pe.FILE\_HEADER.TimeDateStamp)

elif(command == "szoption"):

print(pe.FILE\_HEADER.SizeOfOptionalHeader)

else:

optional\_header(command)

def optional\_header(command):

if(command == "szcode"):

print(pe.OPTIONAL\_HEADER.SizeOfCode)

elif(command == "epaddr"):

print(pe.OPTIONAL\_HEADER.AddressOfEntryPoint)

elif(command == "imgbase"):

print(pe.OPTIONAL\_HEADER.ImageBase)

elif(command == "dllchar"):

print(pe.OPTIONAL\_HEADER.DllCharacteristics)

else:

section\_header(command)

def section\_header(command):

if(command == "secname"):

for section in pe.sections:

print(section.Name.decode('utf-8'))

elif(command == "secva"):

for section in pe.sections:

print(section.Name.decode('utf-8'), hex(section.VirtualAddress))

elif(command == "secraw"):

for section in pe.sections:

print(section.Name.decode('utf-8'), section.SizeOfRawData)

else:

fix(command)

def fix(command):

if(command == "fiximg"):

imagebase = int(input("Input Value: "))

pe.OPTIONAL\_HEADER.ImageBase = imagebase

elif(command == "fixepaddr"):

epaddr = int(input("Input Value: "))

pe.OPTIONAL\_HEADER.AddressOfEntryPoint = epaddr

else:

etc(command)

def etc(command):

if(command == "exit"):

exit()

elif(command == "clear"):

os.system("clear")

elif(command == "cls"):

os.system("cls")

elif(command == "dump"):

print(pe.dump\_info())

elif(command == "textdump"):

print(pe.get\_memory\_mapped\_image())

elif(command == "help"):

print\_help()

elif(len(command) == 0):

pass

elif(command == "showall"):

showall()

else:

print("Command is not found")

def showall() :

print("\n[DOS Header]")

print("e\_magic: ", pe.DOS\_HEADER.e\_magic)

print("e\_lfanew: ", pe.DOS\_HEADER.e\_lfanew)

print("\n[FILE Header]")

print("Machine: ", pe.FILE\_HEADER.Machine)

print("NumberOfSections: ", pe.FILE\_HEADER.NumberOfSections)

print("TimeDateStamp: ", pe.FILE\_HEADER.TimeDateStamp)

print("SizeOfOptionalHeader: ", pe.FILE\_HEADER.SizeOfOptionalHeader)

print("Characteristics: ", pe.FILE\_HEADER.Characteristics)

print("\n[OPTIONAL Header]")

print("Magic: ", pe.OPTIONAL\_HEADER.Magic)

print("AddressOfEntryPoint: ", pe.OPTIONAL\_HEADER.AddressOfEntryPoint)

print("ImageBase: ", pe.OPTIONAL\_HEADER.ImageBase)

print("SectionAlignment: ", pe.OPTIONAL\_HEADER.SectionAlignment)

print("FileAlignment: ", pe.OPTIONAL\_HEADER.FileAlignment)

print("SizeOfImage: ", pe.OPTIONAL\_HEADER.SizeOfImage)

print("DllCharacteristics: ", pe.OPTIONAL\_HEADER.DllCharacteristics)

print("\n[SECTION Headers]")

for section in pe.sections:

print("\nName: ", section.Name.decode('utf-8'))

print("VirtualSize: ", section.Misc\_VirtualSize)

print("VirtualAddress: ", hex(section.VirtualAddress))

print("SizeOfRawData: ", section.SizeOfRawData)

print("PointerToRawData: ", section.PointerToRawData)

print("Characteristics: ", section.Characteristics)

print("\n[DATA Directories]")

for idx, entry in enumerate(pe.OPTIONAL\_HEADER.DATA\_DIRECTORY):

print("\nDirectory", idx)

print("Size: ", entry.Size)

print("VirtualAddress: ", hex(entry.VirtualAddress))

if hasattr(pe, 'DIRECTORY\_ENTRY\_IMPORT'):

print("\n[IMPORT Table]")

for entry in pe.DIRECTORY\_ENTRY\_IMPORT:

print("\nDLL Name: ", entry.dll.decode('utf-8'))

for imp in entry.imports:

print("Ordinal: ", imp.ordinal)

print("Name: ", imp.name.decode('utf-8') if imp.name else "")

if hasattr(pe, 'DIRECTORY\_ENTRY\_EXPORT'):

print("\n[EXPORT Table]")

for exp in pe.DIRECTORY\_ENTRY\_EXPORT.symbols:

print("\nOrdinal: ", exp.ordinal)

print("Address: ", hex(exp.address))

print("Name: ", exp.name.decode('utf-8') if exp.name else "")

def print\_help():

print("""

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

| Command Comment |

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

| DOS\_HEADER |

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

| emagic Show Machine |

| lfanew Show lfanew |

| cblp Show cblp |

| cp Show cp |

| crlc Show crlc |

| cparhdr Show cparhdr |

| minalloc Show minalloc |

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

| FILE\_HEADER |

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

| machine Show Machine |

| numsection Show NumberOfSections |

| timestamp Show TimeDateStamp |

| szoption Show SizeOfOptionalHeader|

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

| OPTIONAL\_HEADER |

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

| szcode Show SizeOfCode |

| epaddr Show AddressOfEP |

| imgbase Show ImageBase |

| dllchar Show DllCharacteristics |

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

| SECTION\_HEADER |

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

| secname Show Name of Sections |

| secva Show VA of Sections |

| secraw Show SizeOfRawData |

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

| Fix |

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

| fiximg fix Value of ImageBase |

| fixepaddr fix Value of EP Address |

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

| Etc |

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

| clear Linux User clear |

| cls Windows User clear |

| exit Exit the program |

| help Show Command |

| showall Show all Information |

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

""")

filepath = input("file name: ")

pe = pefile.PE(filepath)

while(1):

command = input("command:")

dos\_header(command)

# PE

## PE포맷 이란?

윈도우 운영체제에서 사용되는 실행파일, DLL Object 코드, FON 폰트파일 등을 위한 파일형식임.

OE파일은 윈도우 로더가 실행가능한 코드를 관리하는데 필요한 정보를 캡슐화한 데이터 구조체임.

풀어 쓰자면 PE 말 그대로 옮겨 다니면서 실행시킬 수 있는 파일을 뜻함.

MS에서 다른 운영체제와 이식성을 좋게 하기위해 만든 파일 포맷임.

## PE파일의 종류

실행, 드라이버, 라이브러리, 오브젝트 총 4개의 계열이 있음.

- 실행 계열 : EXE, SCR

- 드라이버 계열 : SYS, VXD

- 라이브러리 계열 : DLL, OCX, CPL, DRV

- 오브젝트 계열 : OBJ

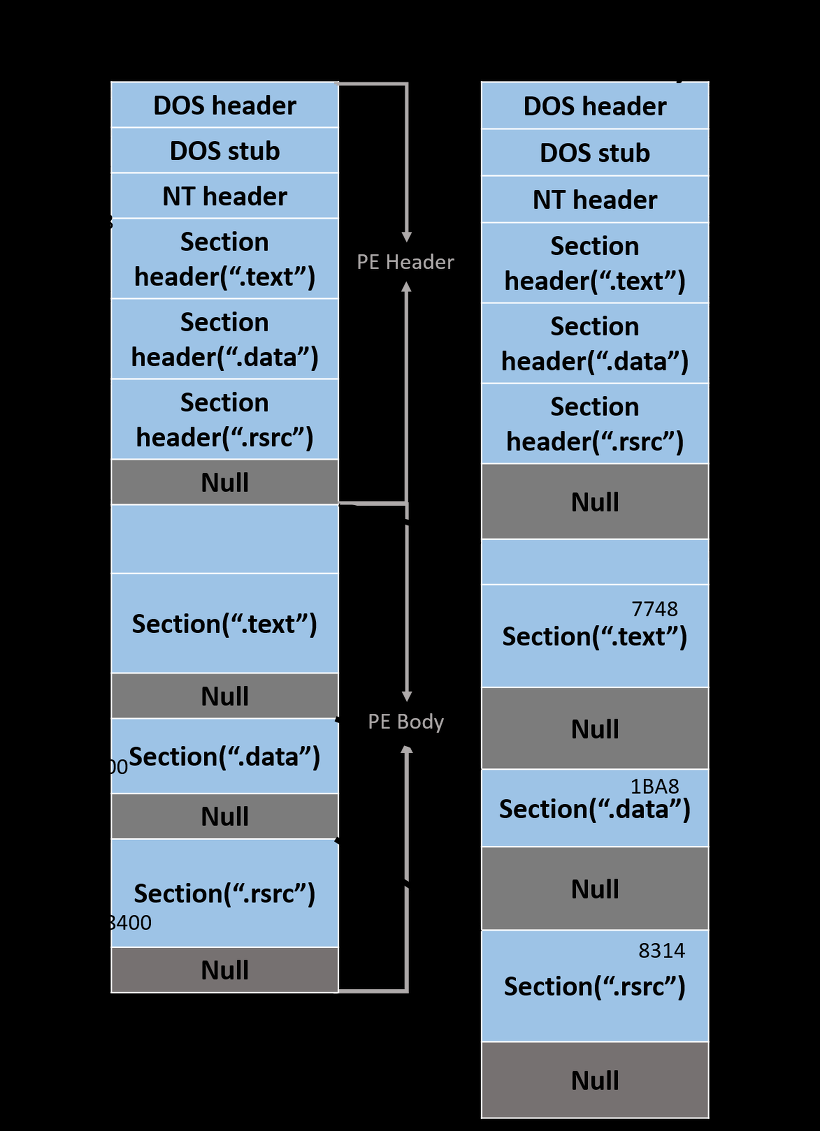
### 알아야 하는 이유?

- PE구조는 파일이 실행되기 위한 모든 정보를 볼 수 있기 때문임.

- 프로그램이 사용하는 API나 어느 메모리 주소에 파일이 로딩되는지 알 수 있기 때문임.

PE 파일의 기본 구조

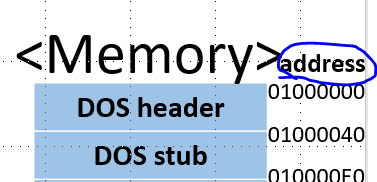
메모리에 적재될 때의 모습



PE 파일(메모장.exe)이 메모리에 로딩되는 모습

PE header : DOS header ~ Section header

PE Body : 그 밑 Section들



메모리에서는 절대 주소로 위치를 표현하며 그것을 VA라고 함.

## NULL 값이 존재하는 이유

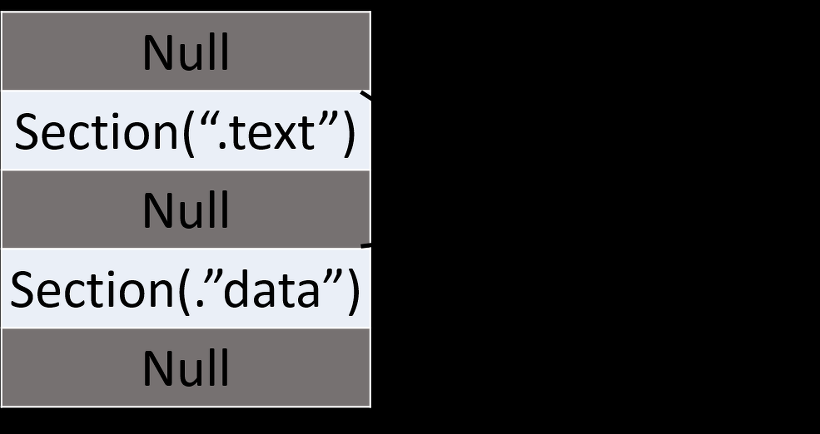
자세히 보면 헤더의 끝부분과 각 섹션 끝에 NULL 값이 붙는 것을 확인할 수 있음.

NULL은 padding이라고 불리기도 함.

즉, 최소 기분단위를 사용하기위해 NULL값이 존재함.

- 최고 기본단위 란? 컴퓨터에서 파일, 메모리, 네트워크 패킷 등을 처리할 때 효율을 높이기 위한 개념

파일 / 메모리에서 섹션의 시작위치는 각각 최소 기본단위의 배수에 해당하는 위치여야하고 빈 공간은 NULL로 채움.



다시 정리를 하자면, 각 섹션의 시작위치는 최소기본단위의 배수에 해당하는 위치여야 하며 그 값이 부족할 경우 NULL값을 줘서 부족한 공간을 채워 줌. 이렇게 해서 최소 기본단위가 적용된다고 말할 수 있음.

## VA & RVA

VA ; Virtual Address(메모리 절대주소)

RVA ; Relative Virtual Address(어느 기준 위치에서부터의 상대 주소)

어떤 기준에서부터 상대주소를 뜻함.

1) 먼저 PE가 메모리에 적재되기 전에 기본 ImageBase는 0임.

2) 그리고 PE안에서는 상대주소로 주소가 적히게 됨.

- 메모리에 적재 시 절대주소가 들어가게 된다면 Relocation이 어렵기 때문임.

- 상대주소가 들어가면 ImageBase에서 얼마만큼 떨어진 곳으로 이동하면 되기 때문에 Relocation이 쉽게 되어 상대주소를 사용함.

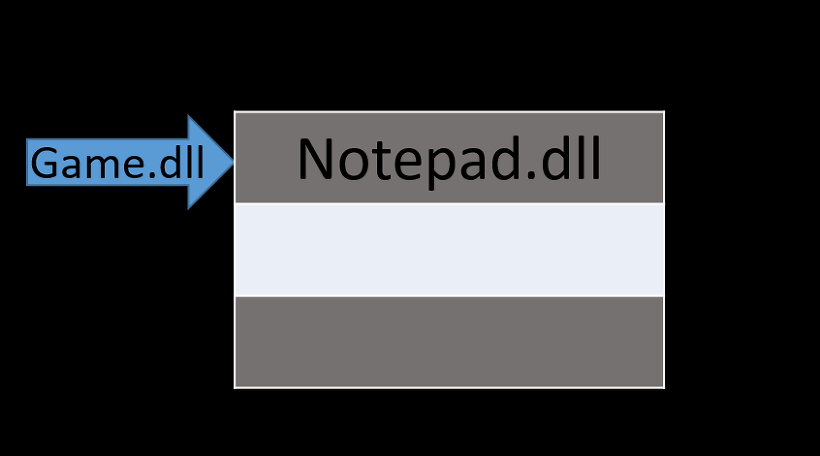
#### VA와 RVA의 관계식

RVA + ImageBase = VA ; 상대주소 + 기준위치 = 절대주소

### 왜 RVA를 사용해야 하는가?

PE 헤더 내의 정보는 RVA 형태로 된 것이 많음.

PE파일(주로 DLL)이 프로세스 가상메모리의 특정위치에 로딩되는 순간 이미 그 위치에 다른 PE파일(DLL)이 로딩되어 있을 수 있음. 이때 재배치를 통해 비어 있는 다른 자리에 로딩되어야 하는데 만약 PE헤더 정보들이 VA로 되어 있다면 정상적인 엑세스가 이루어지지 않았을 것.

game.dll가 00000000이라는 위치에 로딩되는 순간 notepad.dll이 먼저 로딩이 되어버렸음.

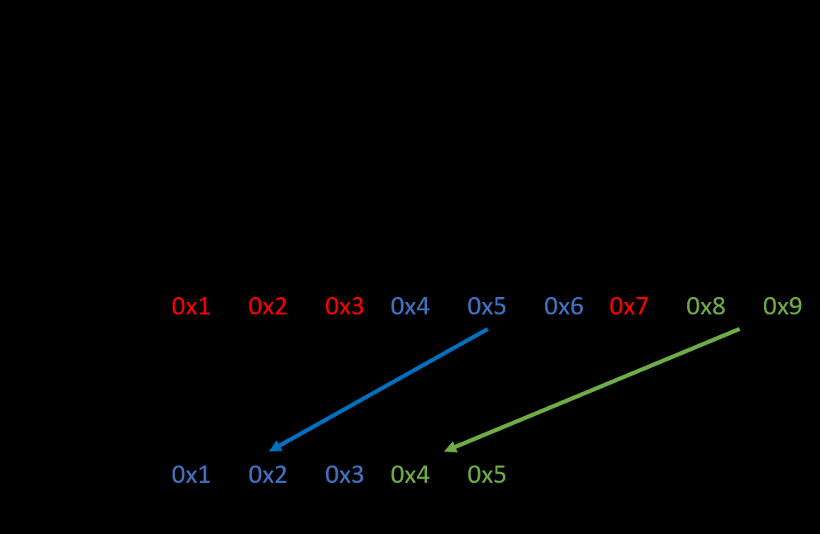
하나의 위치에 2개의 dll파일이 올라갈 수 없기 때문에 이때 재배치를 해줘야 하는 상황이 생기게 됨.

이때 PE헤더 정보들이 VA로 되어 있다면 정상적인 엑세스가 이뤄지지 않기 때문에 RVA로 해서 재배치가 발생해도 기준에 대한 상대주소는 변하지 않기 때문에 문제없이 정보에 엑세스 할 수 있게 됨.

#### 절대주소? 상대주소?

\* 절대주소 : 메모리의 위치를 식별하는 메모리 고유 주소. 즉, 기억장치고유의 주소로서 기억장소를 직접 숫자로 지정함.

\* 상대주소 : 고유주소가 아니며 특정영역에 대한 상대적인 주소를 지정함.

윗 줄 : 물리주소

밑 줄 : 가상주소

빨간색 : 사용 중

파란색, 초록색 : 사용 안함

현재 0x1 ~ 0x3, 0x7이 사용중임. 그리고 중간중간 비어 있는 공간이 있음.

그러나 프로그램은 연속적 메모리를 사용해야 하기 때문에 빈 공간인 0x4 ~ 0x6, 0x8 ~ 0x9(물리)를 사용하면 안됨.

그래서 나온 개념이 가상주소. 이렇게 사용하게 되면

프로그램은 가상메모리인 0x1 ~ 0x3, 0x4~0x5(가상)까지 접근하게 되고 운영체제는 알아서 물리적인 메모리인 0x4~0x6, 0x8~0x9(물리)를 찾아가서 값을 읽어오게 됨.

즉, 물리적인 빈 공간을 가상주소에 연결해서 가상주소에 접근하면 운영체제가 알아서 가상주소와 연결된 물리주소를 찾아가서 값을 읽어 옴.

추가로 가상주소는 0x0 ~ 0xFFFFFFFF 까지 이며 프로그램은 4byte 메모리를 사용해야 함.

## PE header

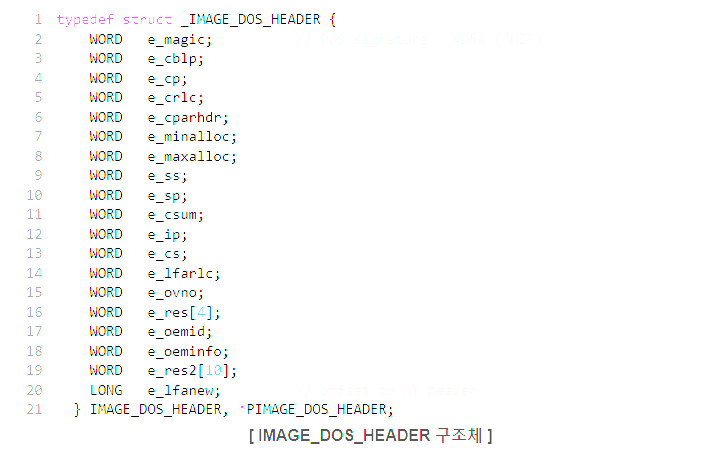
PE헤더는 구조체로 이루어져 있다.

위 사진(PE파일이 메모리에 로딩되는 모습)을 보면 PE헤더의 영역은 DOS header ~ Section header인 것을 확인할 수 있음.

DOS header

Microsoft는 PE파일 포맷을 만들 때 DOS 파일에 대한 하위 호환성을 고려해서 만들었음.

그 결과로 PE헤더 제일 앞 부분에는 기존 DOS EXE Header를 확장시킨 IMAGE\_DOS\_HEADER 구조체가 존재함. 즉 IMAGE\_DOS\_HEADER는 DOS EXE HEADER를 확장시킨 것.



IMAGE\_DOS\_HEADER 구조체의 크기는 40임.

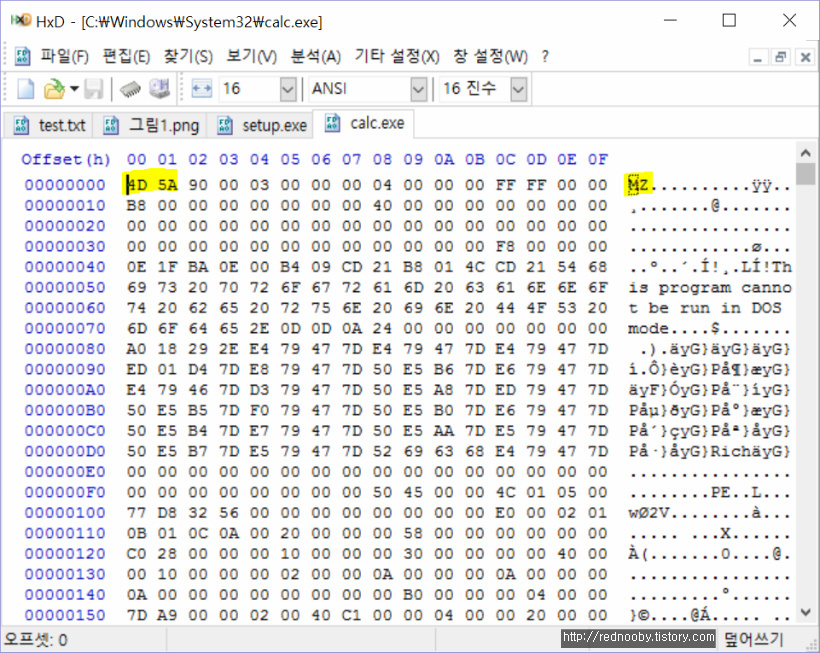
이 구조체에서 가장 중요한 멤버는 e\_magic, e\_lfanew 임.

\* e\_magic : DOS signature(4D5A 이며 ASCII 값으로 "MZ")

\* e\_lfanew : NT header의 옵셋을 표시(파일에 따라 가변적인 값)

모든 PE파일은 시작부분(e\_magic)의 DOS signature(MZ)가 존재하고 e\_lfanew값이 가리키는 위치에 NT header 구조체가 존재해야 함. 그 구조체의 이름이 IMAGE\_NT\_HEADER임.

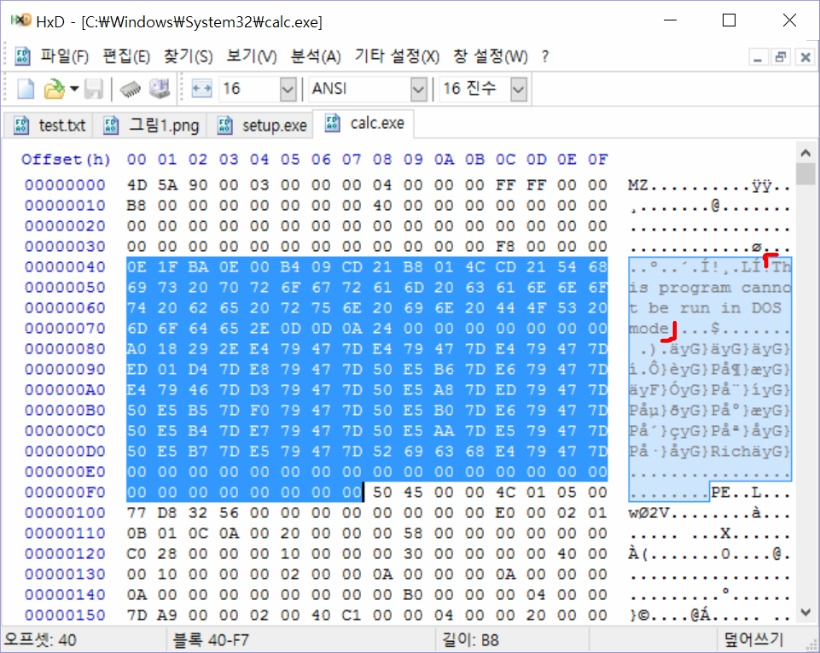
정리 : 모든 pe파일 시작 부분에는 e\_magic이 존재함.



e\_magic에 의해서 맨 처음에 MZ의 값인 4D 5A로 시작하는 것을 확인할 수 있음

## DOS Stub

\* DOS header 밑에 존재하는 DOS Stub임. DOS Stub은 옵션에 의해 존재여부를 결정하며 크기도 일정하지 않음. (DOS Stub이 없어도 파일 실행에는 문제가 없음.)

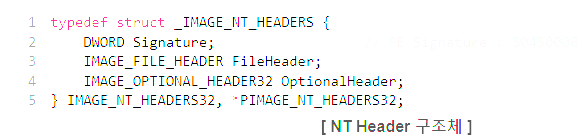


쉽게 말하자면 DOS환경에서 돌아갈 명령어가 16bit 형태로 있으며 분석해보면 This program cannot be run in DOS mode 라는 것으로 보아 이 프로그램은 DOS mode에서 실행될 수 없다고 볼 수 있다.

## NT Header

NT 헤더 구조체는 크게 3개로 구분함.

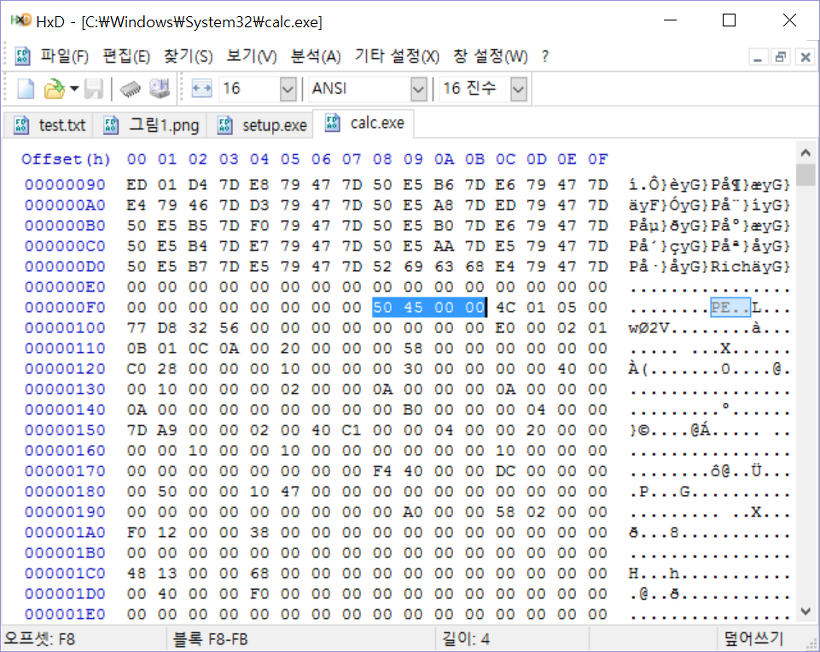
- Signature, IMAGE\_FILE\_HEADER, IMAGE\_OPTIONAL\_HEADER32 임.



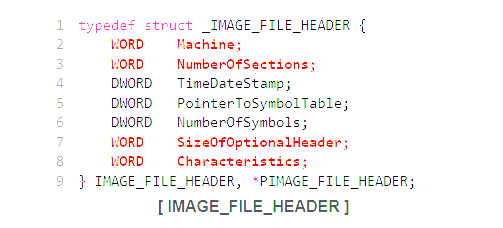
앞서 말했듯 Signature, IMAGE\_FILE\_HEADER, IMAGE\_OPTIONAL\_HEADER32가 정의된 것을 확인할 수 있음.

#### DOS signature

여기부터 NT header구조체의 시작을 판별할 수 있음.



#### IMAGE\_FILE\_HEADER



1. Machine

CPU 별로 고유한 값을 가지면서 IA-32 호환 CPU 14Ch의 값을 IA-64호환 CPU는 200h의 값을 가진다.

- CPU별로 고유한 값을 가진다.

- IA-32 호환 CPU = 14Ch

- IA-64 호환 CPU = 200h

2. NumberOfSections

이 값은 다음에 나오는 Section들의 갯수이며 최소 1개 이상이여야 함.

3. SizeOFOptionalHeader

이 값은 IMAGE\_NT\_HEADERS 구조체의 마지막 구조체 IMAGE\_OPTIONAL\_HEADER32의 구조체 크기를 나타냄.

4. Characteristics

이 값은 파일 속성에 대한 부분

헥사 에디터로 따지면 이 부분이 되겠으며 말 그대로 파일의 속성에 대한 부분임.

여기서 SizeOfOptionalHeader의 값이 E0임. 10진수로 바꾸면 224가 되는데 이것이 헤더의 총 길이가 된다.

#### IMAGE\_OPTIONAL\_HEADER



1. Magic

이 구조체가 32bit요이면 10Bh, 64bit면 20Bh의 값을 가짐

2. AddressOfEntryPoint

EP(Entry Point)의 RVA 주소 값이 들어있음.

쉽게 말해 프로그램에서 최초로 실행되는 코드의 시작주소. 매우 중요한 값임.

3. Image Base

PE파일이 맵핑되는 시작주소를 가리킴.

4. SEctionAlignment

메모리에서의 섹션의 최소단위를 나타냄.

5. FileAlignment

파일에서의 섹션의 최소단위를 나타냄.

6. SizeOfImage

메모리에서의 PE구조 크기를 나타냄.

7. SizeofHeader

PE header의 크기를 나타냄.

8. Sybsystem

-1:드라이버 파일(SYS, VXD)라는 뜻.

-2: GUI파일 이라는 뜻.

-3: CUI파일이라는 뜻.

9. NumberOfRvAndSizes

바로 밑에 있는 IMAGE\_DATA\_DIRECTORY 구조체의 배열 크기를 정함.

#### Section Header

\* 각 Section의 헤더들이 존재하며 Section들은 메타데이터를 저장하고 메모리에 로드될 때 필요한 정보를 포함하고 있다.

1) VirtualSize : 메모리에서 섹션의 크기

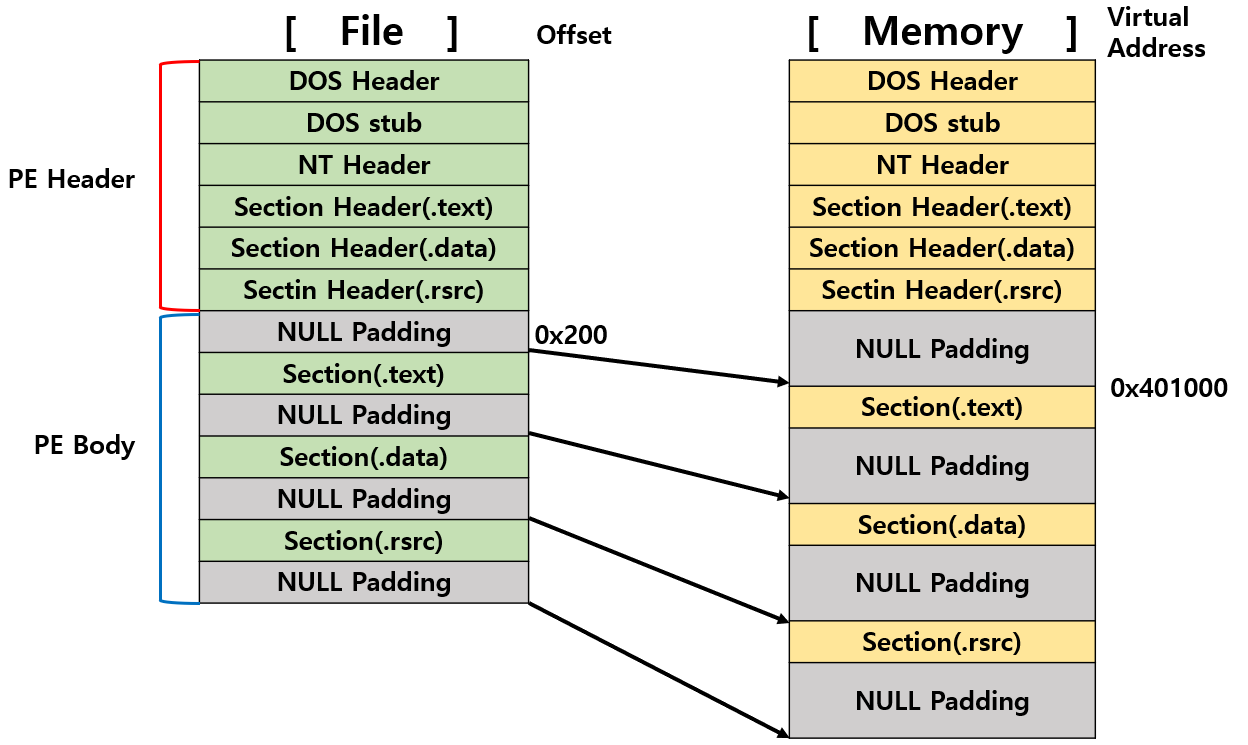
2) VirtualAddress : 메모리에서 섹션의 시작주소(RVA)

3) SizeOfRawData : 파일에서 섹션의 크기

4) PointerToRawData : 파일에서 섹션의 Offset

5) Characteristics : 섹션의 속성, Bit OR연산

## PE 파일 구조



1) PE 구조

\* 다양한 정보들이 PE Header에 구조체 형식으로 저장

\* DOS Header부터 Section Header까지 PE Header이며 밑에 Section들을 합쳐서 PE Body라고함.

\* 파일에서는 offset, 메모리에서는 VA로 위치를 표현.

2) offset, VA, RVA

\* Offset : 파일의 첫 바이트부터 거리를 뜻함

\* VA : 프로세스 가상 메모리의 절대 주소

\* RVA : 기준(Image base)으로부터 상대 주소

PE Header의 많은 정보는 RVA 형태로 된 것들이 존재함.

\* 이유 : PE파일(주로DLL)이 가상 메모리의 특정 위치에 로딩될 때 해당 위치에 다른 PE파일이 로딩되어 있을 수 있으므로 Relocation 과정을 통해 빈 공간에 로딩되어야 한다. 만약, VA로 되어있다면 정상적인 액세스가 이뤄지지 않지만 RVA로 되어있다면 Relocation이 발생해도 ImageBase에 대한 상대주소는 변하지 않기 때문에 문제 없이 엑세스가 가능.

PE 파일의 `IMAGE\_DOS\_HEADER`는 PE 파일이 MS-DOS 환경과 호환성을 유지할 수 있도록 하는 정보를 담고 있음. `IMAGE\_DOS\_HEADER`는 다음과 같은 필드를 포함하고 있음

- `e\_magic` : PE 파일임을 나타내는 표준 시그니처(매직 넘버). 이 값은 'MZ'(0x5A4D)로 설정되어 있습니다.

- `e\_cblp` : 페이지의 마지막 바이트 수.

- `e\_cp` : 파일의 페이지 수.

- `e\_crlc` : 재배치 항목 수.

- `e\_cparhdr` : 헤더 크기(파라그래프 단위).

- `e\_minalloc` : 최소 추가 메모리 요구량(파라그래프 단위).

- `e\_maxalloc` : 최대 추가 메모리 요구량(파라그래프 단위).

- `e\_ss` : 초기 SS 값(상대적인 주소).

- `e\_sp` : 초기 SP 값.

- `e\_csum` : 체크섬.

- `e\_ip` : 초기 IP 값.

- `e\_cs` : 초기 CS 값(상대적인 주소).

- `e\_lfarlc` : 재배치 테이블의 파일 주소.

- `e\_ovno` : 오버레이 번호.

- `e\_res` : 예약된 단어(8개).

- `e\_oemid` : OEM 식별자.

- `e\_oeminfo` : OEM 정보.

- `e\_res2` : 예약된 단어(10개).

- `e\_lfanew` : NT 헤더의 파일 주소.

\* 이 중 `e\_lfanew` 필드는 PE 헤더의 시작 위치를 가리키는 중요한 필드. 이 값은 PE 파일의 구조를 파싱하는 데 필요한 기본 정보를 제공함.