- 리버싱 공부 1~8일차 -

pwn&play 박석은

ASEMBLY INFO

EAX : Extended Accumulator Register

- 함수 리턴 값 저장. Win32 API 함수들은 모두 return value 를 EAX에 저장한 후 return 합니다.

EBX : Extended Base Register

- DS segment 에서 Data를 가리킬 때 사용합니다.

ECX : Extended Counter Register

- 반복문 명령어(LOOP)에서 참조 카운트로 사용됩니다. (루프 돌 때마다 ECX를 1씩 감소 시킵니다.)

EDX : Extended Data Register

ESI : Extended Source Register / EDI : Extended Destination Register

- 특정 명령어(LODS, STOS, REP MOVS, etc)와 함께 주로 메모리 복사에 사용됩니다.

EBP : Extended Base Pointer

- 함수가 호출 되었을 때 그 순간의 ESP를 저장하고 있다가, 함수가 return 하기 직전에

다시 ESP에 값을 되돌려줘서 stack이 깨지지 않도록 합니다.

ESP : Extended Stack Pointer

- Stack memory address를 가리킵니다. 어떤 명령어들(PUSH, POP, CALL, RET)은 ESP를 직접 조작하기도 합니다. (stack memory 관리는 프로그램에서 매우 중요하기 때문에 ESP를 다른 용도로 사용하지 않는 것이 좋습니다.

EIP : Extended Instruction Pointer

출처 : reverscore.com

32bit register	16bit register	8bit
EAX	AX	AH/AL
EBX	BX	BH/BL
ECX	CX	CH/CL
EDX	DX	DH/DL
ESI	SI	-
EDI	DI	-
EBP	BP	-
ESP	SP	-
EIP	IP	-

ASSEMBLY INFO : PUSH

Syntax :

PUSH	operand

- PUSH는 값을 stack(스택)에 저장하고 ESP(스택 포인터)가 명령어의 크기만큼 줄어든다.

  그래서 PUSH를 한 후에 ESP는 PUSH된 값을 가리키게 된다.

  ex) push 0(6A 00) 명령이 실행된 후 stack window를 보면 stack의 값이 0이 되는 것을 알 수 있다.

ASSEMBLY INFO : CALL

Syntax :

CALL	something

- CALL 명령은 RVA(Relative Virtual Address, 상대적인 가상 메모리 주소)의 명령을 넣는다.

  call 명령은 call을 실행한 후에 return할 주소를 stack에 넣는다. 그리고 sub program/procedure를 실행한다.

  How to use :

  - CALL 404000(CALL address)
  - CALL EAX // EAX에 있는 주소값을 실행한다.
  - CALL DWORD PTR [EAX] // EAX에 있는 주소값을 실행한다.
  - CALL DWORD PTR [EAX+5] // EAX+5에 있는 주소값을 실행한다.
  - CALL <JMP to API> 또한 CALL address다. 해당 명령을 사용하면 API를 사용할 수 있다.

ASSEMBLY INFO : MOV (Move)

Syntax :

Mov	dest	src

(dest=destination, src=source)

- MOV는 src에서 dest로 값을 복사한다. 그리고 src 값은 명령을 실행하기 전의 값을 유지한다.

- MOVS/MOVSB/MOVSW/MOVSD EDI,ESI:

* Byte/word/dword ESI points에서 EDI 공간에 값을 복사한다.

- Movsx는 Bytes나 word에서 Word나 Dword 크기로 확장한다. 그리고 sign 값은 유지한다.

ex) sign값은 +,- 와 같은 부호. 즉 Byte 크기에서 Word 크기로 확장할 때 부호를 유지한다.

예를 들면 byte값인 0000 0001을 word값인 0000 0000 0000 0001로 확장할 때 부호를 유지한다.

- Movzx는 Bytes나 Word에서 Word나 Dword 크기로 확장한다. 그리고 나머지 영역은 0으로 채운다.

ASSEMBLY INFO : CMP (Compare)

Syntax :

CMP	dest	src

- 두가지 값을 비교하고 그에 맞는 C/O/Z flag를 set한다.

* Carry-flag / Overflow-flag / Zero-flag

ASSEMBLY INFO : TEST

Syntax :

TEST	operand1	operand2

- 논리 AND 연산을 수행할 때 사용된다. (TEST EAX, EAX)

* 값은 저장하지 않는다. Z-Flag는 EAX가 0일 때 set되거나, EAX가 0이 아닐 때 clear된다.

O/C flag는 항상 지워진다.

ASSEMBLY INFO : AND (Logical AND)

Syntax :

AND	dest	src

- AND 명령어는 논리 AND에 두 가지 값으로 사용된다.

 이 명령어는 O-Flag와 C-Flag를 clear하고 Z-Flag를 set한다.

* 연산하고자 하는 값이 둘다 1이면 1로 아니면 0으로 바뀐다.

ASSEMBLY INFO : XOR

Syntax :

XOR	dest	src

- XOR 명령어는 두 값을 배타적 논리 OR로 연결한다.

 이 명령어는 O-Flag와 C-Flag를 clear 하고, Z-Flag를 set한다.

 * 연산하고자 하는 값이 서로 같으면 0, 아니면 1이다.

ASSEMBLY INFO : INC

Syntax :

INC	Register

- INC의 뜻은 1만큼 증가시킨다는 것이다.

- INC는 Z/O Flags를 Set(1) 한다.

조건 점프 명령 정리

명령어	명령어의 의미	명령어가 수행되기 위한 플래그 레지스터와 범용 레지스터의 상태
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF= 0
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JE JZ	Jump if equal Jump if zero	ZF = 1
JNE JNZ	Jump if not equal Jump if not zero	ZF = 0
JB JNAE JC	Jump if below Jump if not above or equal Jump if carry	CF = 1
JNB JAE JNC	Jump if not below Jump if above or equal Jump if not carry	CF = 0
JBE JNA	Jump if below or equal jump if not above	CF = 1 or ZF = 1
JA JNBE	Jump if above Jump if not below or equal	CF = 0 and ZF = 0
JL JNGE	Jump if less Jump if not greater or equal	SF <> OF

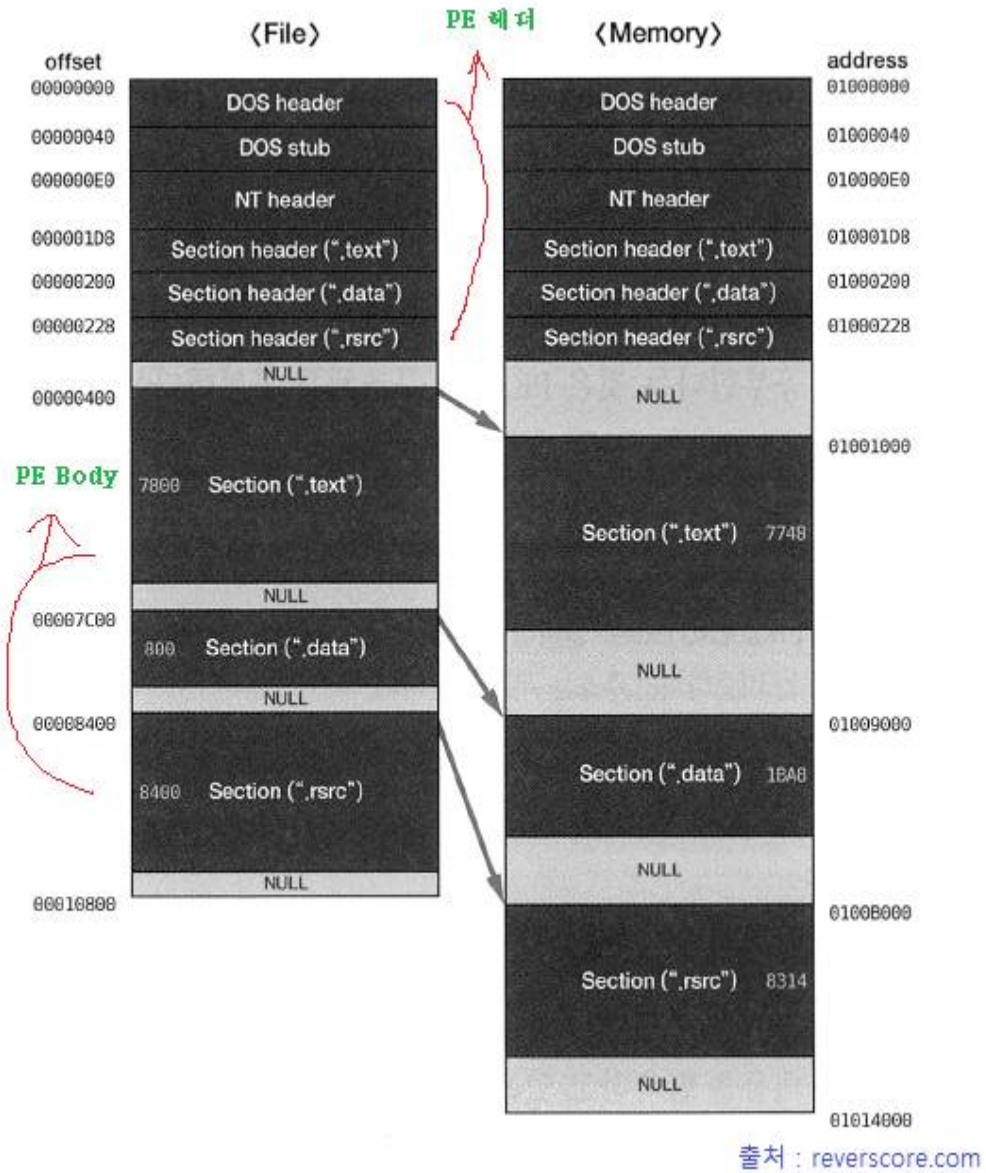
JGE **JNL**	**Jump if greater or equal**	**SF = OF**
JLE **JNG**	**Jump if less or equal** **Jump if not greater**	**ZF = 1 or SF <> OF**
JG **JNLE**	**Jump if greater** **Jump if not less or equal**	**ZF = 0 and SF = OF**
JP **JPE**	**Jump if parity**	**PF = 1**
JNP **JPO**	**Jump if not parity**	**PF = 0**
JCXZ **JECXZ**	**Jump if %CX register is 0** **Jump if %ECX register is 0**	**%CX = 0** **%ECX = 0**

PE File Format

종 류	주요 확장자	종 류	주요 확장자
실행 계열	EXE, SCR	드라이버 계열	SYS, VXD
라이브러리 계열	DLL, OCX, CPL, DRV	오브젝트 파일 계열	OBJ

출처 : reverscore.com

PE 파일이 메모리에 로딩되는 모습



출처 : reverscore.com

<Little Endian & Big Endian>

TYPE	Name	SIZE	빅 엔디언 Style	리틀 엔디언 Style
BYTE	b	1	[12]	[12]
WORD	w	2	[12][34]	[34][12]
DWORD	dw	4	[12][34][56][78]	[78][56][34][12]
char []	str	6	[61][62][63][64][65][00]	[61][62][63][64][65][00]

출처 : reverscore.com

IMAGE_DOS_HEADER

- 도스헤더 구조체 -

```
#include <pshpack2.h>
180a088be Alexa*2365 typedef struct _IMAGE_DOS_HEADER {
                2366     WORD  e_magic;      /* 00: MZ Header signature */
                2367     WORD  e_cblp;       /* 02: Bytes on last page of file */
                2368     WORD  e_cp;         /* 04: Pages in file */
                2369     WORD  e_crlc;       /* 06: Relocations */
                2370     WORD  e_cparhdr;    /* 08: Size of header in paragraphs */
                2371     WORD  e_minalloc;   /* 0a: Minimum extra paragraphs needed */
                2372     WORD  e_maxalloc;   /* 0c: Maximum extra paragraphs needed */
                2373     WORD  e_ss;         /* 0e: Initial (relative) SS value */
                2374     WORD  e_sp;         /* 10: Initial SP value */
                2375     WORD  e_csum;       /* 12: Checksum */
                2376     WORD  e_ip;         /* 14: Initial IP value */
                2377     WORD  e_cs;         /* 16: Initial (relative) CS value */
                2378     WORD  e_lfarlc;     /* 18: File address of relocation table */
                2379     WORD  e_ovno;       /* 1a: Overlay number */
                2380     WORD  e_res[4];     /* 1c: Reserved words */
                2381     WORD  e_oemid;      /* 24: OEM identifier (for e_oeminfo) */
                2382     WORD  e_oeminfo;    /* 26: OEM information; e_oemid specific */
                2383     WORD  e_res2[10];   /* 28: Reserved words */
                2384     DWORD e_lfanew;     /* 3c: Offset to extended header */
                2385 }
```

- 도스헤더 구조체는 보는것과 같이 64byte크기(0x40)로 이루어진 구조체이다.

  중요한 부분은 두 곳이다. 바로 e_magic 과 e_lfanew 이다.

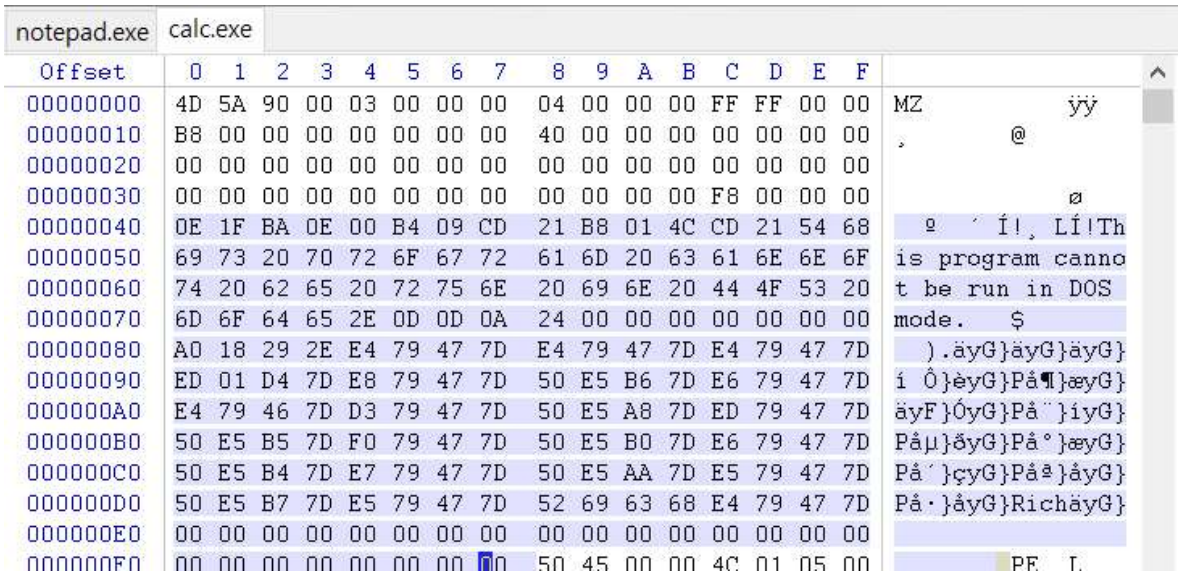
e_magic : 해당 부분은 PE파일이 맞는지 확인할 때 사용되는 필드이다.

PE로더나,헥스에디터를 사용해 exe파일을 열어보면 맨 처음 부분 2byte(2byte인 이유는 word이기 때문) 부분을 보면 4D 5A(MZ)라고 되있는 것을 볼 수 있다. 나는 저 MZ를 검색하다가 마크 즈비코프스키라는 사람의 이름을 땃다는 것을 알게되었는데, 발음하다보면 느끼겠지만 욕처럼 들리기도 한다. LOL..

아무튼 본론으로 돌아가서, PE 파일은 무조건 4D 5A(MZ)로 고정되어있다. 4D 5A가 아니면 PE 파일이 아니다.

e_lfanew : 해당 필드는 IMAGE_NT_HEADER의 START OFFSET을 가리키며, e_magic처럼 값이 고정되어 있지 않고 파일에 따라 변한다. 해당 부분을 참조하면 NT헤더의 주소를 알아낼 수 있다.

Stub Code



위 사진에서 드래그한 부분이 Stub Code이다.

잘 보면 "This program cannot be run is DOS mode."라고 되있는데, 이 부분이 존재하는 이유는 dos-mode에서 파일이 실행되지 않도록 하기 위함이다. 만약 dos-mode에서 실행하면 저 문자열이 출력된다.

IMAGE_NT_HEADER

```
typedef struct _IMAGE_NT_HEADERS {
9a29b0cea Andre*2728   DWORD Signature; /* "PE"\0\0 */   /* 0x00 */
               2729   IMAGE_FILE_HEADER FileHeader;     /* 0x04 */
687337999 Kevin*2730   IMAGE_OPTIONAL_HEADER32 OptionalHeader;   /* 0x18 */
               2731 }
```

위 그림은 NT Header의 구조체이다.

구조체를 보면 2번째 라인에 Signature부분이 있는데, 이 부분의 값(4byte)을 가지고 PE 파일구조인지 구별한다.

그리고 3번째 IMAGE_FILE_HEADER가 있는데 이부분은 다음과 같다.

IMAGE_FILE_HEADER

```
typedef struct _IMAGE_FILE_HEADER {
               2631   WORD  Machine;
               2632   WORD  NumberOfSections;
               2633   DWORD TimeDateStamp;
               2634   DWORD PointerToSymbolTable;
               2635   DWORD NumberOfSymbols;
               2636   WORD  SizeOfOptionalHeader;
               2637   WORD  Characteristics;
               2638 }
```

2631 : Machine 필드는 실행한 파일이 어떤 CPU type에서 동작할 수 있는지 정한다.

2632 : NumberOfSections 필드는 PE 파일을 구성하는 Section의 개수 이다. Section의 개수가 늘어나거나 하면 해당값은 변한다.

*** NumberOfSections 필드는 무조건 0보다 커야한다.(section은 무조건 1개 이상 있기 때문이다.)**

2633 : TimeDateStamp 필드는 PE 파일이 만들어진 시간(이 파일이 컴파일러에 의해 빌드 된 시각)이다.

*** TimeDateStamp는 변조가 쉽고 이 파일을 빌드한 PC의 시간에 따라 시간이 다르므로 맹신은 금물이다.**

2634 : PointerToSymbolTable 필드는 파일의 심볼 내용을 담고 있는 테이블의 오프셋을 저장하고 있다.
없으면 0이다.

2635 : NumberOfSymbols 필드는 심볼 테이블에 저장된 심볼의 수를 저장하고 있다.

이 필드도 PointerToSymbolTable 필드와 같이 0일 수도 있다.

2636 : SizeOfOptionalHeader 필드는 IMAGE_OPTIONAL_HEADER의 크기가 담겨져 있다.

- 32bit에서는 0xE0 , 64bit에서는 0xF0으로 표기되어있다.

2637 : Characteristics 필드는 현재 파일의 형식을 나타준다. (실행파일인지, DLL 파일인지.. 등등)

IMAGE_OPTIONAL_HEADER

```
1typedef struct _IMAGE_OPTIONAL_HEADER {
2                   2687
3                   2688   /* Standard fields */
4                   2689
5         1426c8cb7 Andre*2690   WORD  Magic; /* 0x10b or 0x107 */ /* 0x00 */
6         180a088be Alexa*2691   BYTE  MajorLinkerVersion;
7                   2692   BYTE  MinorLinkerVersion;
8                   2693   DWORD SizeOfCode;
9                   2694   DWORD SizeOfInitializedData;
10                  2695   DWORD SizeOfUninitializedData;
11        1426c8cb7 Andre*2696   DWORD AddressOfEntryPoint;       /* 0x10 */
12        180a088be Alexa*2697   DWORD BaseOfCode;
13                  2698   DWORD BaseOfData;
14                  2699
15                  2700   /* NT additional fields */
16                  2701
17                  2702   DWORD ImageBase;
18        1426c8cb7 Andre*2703   DWORD SectionAlignment;      /* 0x20 */
19        180a088be Alexa*2704   DWORD FileAlignment;
20                  2705   WORD  MajorOperatingSystemVersion;
21                  2706   WORD  MinorOperatingSystemVersion;
22                  2707   WORD  MajorImageVersion;
23                  2708   WORD  MinorImageVersion;
24        1426c8cb7 Andre*2709   WORD  MajorSubsystemVersion;     /* 0x30 */
25        180a088be Alexa*2710   WORD  MinorSubsystemVersion;
26                  2711   DWORD Win32VersionValue;
27                  2712   DWORD SizeOfImage;
28                  2713   DWORD SizeOfHeaders;
29        1426c8cb7 Andre*2714   DWORD CheckSum;          /* 0x40 */
30        180a088be Alexa*2715   WORD  Subsystem;
```

```
31                  2716   WORD  DllCharacteristics;
32                  2717   DWORD SizeOfStackReserve;
33                  2718   DWORD SizeOfStackCommit;
34          1426c8cb7 Andre*2719   DWORD SizeOfHeapReserve;      /* 0x50 */
35          180a088be Alexa*2720   DWORD SizeOfHeapCommit;
36                  2721   DWORD LoaderFlags;
37                  2722   DWORD NumberOfRvaAndSizes;
38          1426c8cb7 Andre*2723   IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
39          9a29b0cea Andre*2724   /* 0xE0 */
40          687337999 Kevin*2725 } IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;
41          180a088be Alexa*2726
42
```

- Magic으로 시작.
- OptionalHeader = 224byte
- DataDirectory = 128byte
- Sizeofcode : 섹션중에 .text라고 되어있는 부분의 실제 크기
  (실제 파일에 기록되는 값의 크기임.)
- SizeofInitializedData : 읽기가 가능한 섹션이 차지하는 크기
- SizeofUninitializedData : SizeofInitializedData의 반대 값
- AddressOfEntryPoint : EP가 올라오는 가상메모리 값에 대한 RVA 값
- BaseOfCode : EP값 이전에 start up code가 오는 경우 그 값만큼 코드값이 줄어든다.
- BaseOfData : data섹션이 시작되는 RVA값
- ImageBase : Dos-header가 오는 자리임.(보통 400000)
- SectionAlignment : 섹션 간 , 가상메모리 간의 간격(주어진 값의 배수!!)
- FileAlignment : 실제 파일에 쓰이는 섹션간의 간격
- SizeOfImage : 가상메모리에 올라갈 총 크기. 헤더와 섹션값을 모두 더한 값
- SizeOfHeader : 모든 헤더값을 더한 값
  SizeOfImage에서 모든 섹션들의 값을 뺀 값.
- NumberOfRvaAndSizes : 10h (DataDirectory(128) / 8 = 16)
  DataDirectory 구조체 멤버마다 각자의 주소를 가리키는 RVA 4byte와
  Size값 4byte를 가진다. 따라서 각각 8byte를 가진다.

<signature 정리>

· 4D 5A --> "MZ"

- dos exe signature.

<API 정리>

CreateFileA : object를 만들거나 열며, object에 접근하여 handle을 돌려준다.

<알아 두면 좋을 것>

* window에서 program은 kernel(low level)제어가 필요 할 때 API function을 호출 한다.

  API는 kernel과 대화하는 것이 허락된다. API는 보통 system dll에 존재한다.

  API를 호출 할 때, 응용프로그램은 API function과 대화하는게 필요하다.

  응용프로그램은 API function으로 넘겨진 parameter로 API와 대화한다.

* 어셈블러에서는 stack에 함수의 파라미터가 반대의 순서로 넣어지는 것을 보게 된다.

  (이유는 그래야 POP할 때 순서대로 나온다. LIFO(Last in First out)

* JNZ는 EAX가 0이 아닐 때 jump한다. TEST EAX, EAX에서 Z-Flag 값이 1이 아니라는 것이다.

- JNZ는 Z-Flag가 1이 아닐 때 jump 한다.

* ascii에서 파일의 끝은 0으로 쓰인다. (10진수로 48 , 16진수로 30)

- 숫자 0은 ascii로 30이다. 혼동하지 말자.

<용어 정리>

· loading or mapping == 적재하다

ex) xx파일이 메모리에 적재(loading or mapping)되다.

· VA(Virtual Address) : 프로세스 가상 메모리의 절대주소.

· RVA(Rela-tive Virtual Address) : 어느 기준 위치(ImageBase)에서부터의 상대주소.

* RVA + ImageBase = VA

· Relocation == 재배치