PE FILE 구조와 언패킹의 원리

지선호

kissmefox@gmail.com

수원대학교 보안동아리 FLAG

〈 PE FILE 이란 〉

-win32 운영체제에서 이용되는 파일 형식 (현재 사용되는 대부분의 OS)

-Portable executable , 이식 가능한 실행 파일 형식
: exe , dll , ocx

-윈도우의 바이너리를 분석하기 위한 가장 기본이 되는 지식

: unpacking , API Hooking , DLL injection , 악성코드 분석 ..

〈PE FORMAT 이해를 위한 기본 지식〉

RVA – 이미지가 해당 프로세스의 가상 주소 공간 내에 로드돼었을 때에 그 시작 주소에 대한 상대적 번지 개념. 메모리 상에서의 PE의 시작 주소에 대한 offset 으로 생각하면 됨

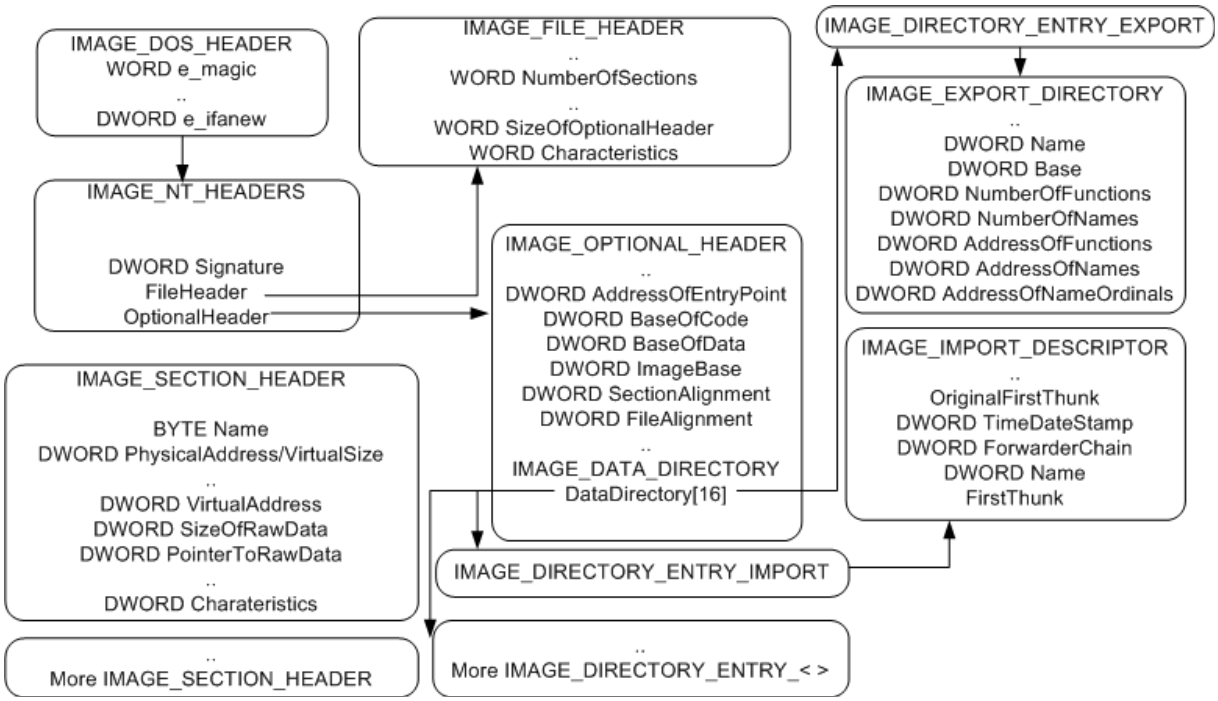
Section – PE 가 가상 주소 공간에 로드된 뒤의 실제 내용(코드와 데이터,Import API,Export API,리소스,재배치정보 TLS 등)을 담고 있는 블록들

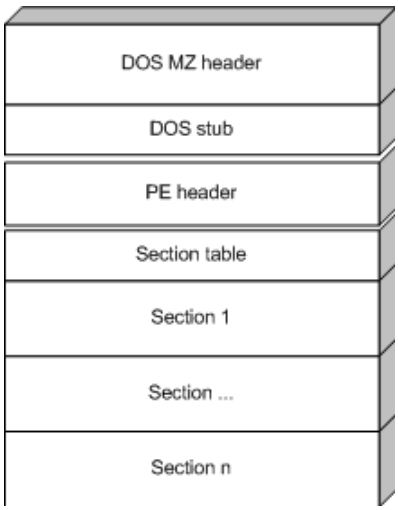
**파일로 존재하는 PE 구조와 가상 번지에 로드되었을 때의 PE 구조의 관계

– 프로세스에 할당되는 4기가바이트의 가상 주소 공간을 유지하기 위해 가상 메모리 관리자(VMM) 는 페이지 파일(pagefile.sys)이라는 스와핑 영역을 하드디스크에 유지하며 이 페이지 파일과의 적절한 스와핑, 매핑을 통해 프로세스에게 4기가의 가상 공간을 실제로 제공해 주는 것처럼 실행 환경을 만들어 주게 됩니다.
하지만 PE 파일이 로드될 때 VMM 은 페이지 파일을 사용하지 않고, PE 파일 자체를 마치 페이지 파일처럼 가상 주소 공간에 그대로 매핑하게 됩니다. 이렇게 파일 자체가 페이지 파일의 역할을 대신하는 경우를 메모리에 매핑된 파일(MMF) 라고 하며 즉 PE 의 경우 MMF 로 해당 파일을 연다고 볼 수 있습니다. WinNT.H 에 정의된 PE Header 의 구조체를 보면 IMAGE 라는 단어를 사용하고 있습니다. 여기서 "IMAGE" 의 의미는 메모리에 매핑된 하드디스크 상의 PE 파일 자체를 말하고 있습니다. 정리하자면, 메모리 상의 로드된 PE 파일 포맷이나, 하드디스크 상에서의 파일로 존재하는 PE 포맷의 모습이 같다고 말할 수 있습니다.

〈 PE FILE HEADER 구조 분석 〉

– Win32 Platform SDK 의 “WinNT.H" 헤더 파일 참조





〈Block 으로 간단히 표현〉

1. DOS MZ header (IMAGE_DOS_HEADER)

:PE 파일의 시작 지점 (PE header 의 offset 을 가짐)
win32 기반 OS에서 PE 파일형식을 실행하면 PE loader는 PE header의 offset을 읽고 바로 DOS 부분을 건너뛰고 PE header 에 접근

–DOS stub
:OS 가 PE 파일 형식을 알지 못할 때 “This program cannot be run in DOS mode" 메시지 출력 , real dos mode 실행 목적으로 사용

주요 필드

e_magic	DOS 헤더를 구별하는 식별자. “MZ" 모든 실행파일은 파일의 가장 첫부분에 이 값을 가지게 됨. PE 로더가 이 값을 체크하여 맞다면 실행파일을 메모리에 로드하게됨
e_lfanew	PE 헤더(IMAGE_NT_HEADER) 가 있는 곳의 offset

–실행파일의 시작부분정도로만 생각하면 됩니다.

2. PE HEADER (IMAGE_NT_HEADER)

: PE 로더가 사용하는 핵심 정보를 담고 있음

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

signature : PE 파일임을 나타내는 매직넘버. 4byte 로 구성되며 항상 "PE₩x0₩x0" 이다. IMAGE_DOS_HEADER 구조체의 e_lfanew 필드가 가리키는 오프셋으로부터의 4byte 값.

FileHeader : IMAGE_FILE_HEADER 구조체 멤버

```
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;
```

주요 필드

Machine	파일이 실행될 CPU 플랫폼
NumberOfSections	파일에 존재하는 섹션의 수
TimeDateStamp	파일이 생성된 시간과 날짜
SizeOfOptionalHeader	IMAGE_OPTIONAL_HEADER 구조체의 크기
Characteristics	해당 PE 파일에 대한 특정 정보를 나타내는 flag

**OptionalHeader : IMAGE_OPTIONAL_HEADER 의 구조체 멤버
Logical Layout 에 대한 정보

```
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;
```

〈IMAGE_OPTIONAL_HEADER 의 구조〉

주요 필드

SizeOfCode	코드영역에 CPU가 실행하는 기계어 코드의 전체 크기
SizeOfInitializedData	Data 가 초기화된 섹션의 총 합
SizeOfUninitializedData	SizeOfInitializedData 의 반대
AddressOfEntryPoint	프로그램의 시작위치(main)의 RVA ** 프로세스가 가장 먼저 시작하는 위치라고 생각하면 안됨!
BaseOfCode	코드영역(주로 .text섹션)의 시작 주소 RVA
BaseOfData	데이터영역(주로 .data섹션)의 시작 주소 RVA

<table>
<tr><td>ImageBase</td><td>PE file 이 메모리에 매핑될 RVA의 기준이 되는 시작 주소
(주로 exe : 0x00400000 dll : 0x10000000)</td></tr>
<tr><td>SectionAlignment</td><td>메모리에 매핑된 후의 섹션의 배치간격. 섹션이 PE로더에 의해 메모리에 올려질때 항상 이 멤버의 배수값으로 위치.
섹션헤더의 VirtualAddress 멤버에 영향을 주게 됨</td></tr>
<tr><td>FileAlgnment</td><td>파일상에서의 섹션의 배치간격. SectionAlignment 와 같은 개념.
섹션헤더의 PointerToRawData 멤버에 영향을 주게 됨</td></tr>
<tr><td>SizeOfImage</td><td>로더가 해당 PE 를 메모리상에 로드할 때 확보해야할 충분한 Size
PE 파일 상에서의 섹션의 배치가 메모리에 매핑되면서 달라질 수 있기 때문에 보통 PE 파일의 크기보다 크다.
값은 반드시 SectionAlignment 필드값이 배수가되어야함</td></tr>
<tr><td>SizeOfHeaders</td><td>PE 포맷의 모든 헤더를 더한 값
FileAlignment 필드 값의 배수가 되어야 함</td></tr>
<tr><td>DataDirectory</td><td>IMAGE_DATA_DIRECTORY 구조체의 멤버

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

VirtualAddress : 데이터 구조체의 RVA
Size : VirtualAddress에서 참조된 데이터 구조체의 크기
총 16개의 배열을 가지고 있음.

00 Export table
01 Import table
02 Resource
03 Exception
04 Security
05 Base relocation
06 Debug
07 Copyright string
08 Unknown
09 Thread local storage(TLS)
10 Load configuration
11 Bound Import
12 Import Address Table
13 Delay Import
14 COM descriptor</td></tr>
</table>

3. Section Table (IMAGE_SECTION_HEADER)

:PE 헤더 바로 뒤에 구조체 배열 형식으로 위치해 있음. 배열의 개수는 FILE HEADER 의 NumberOfSection 멤버값에 의해 결정됨

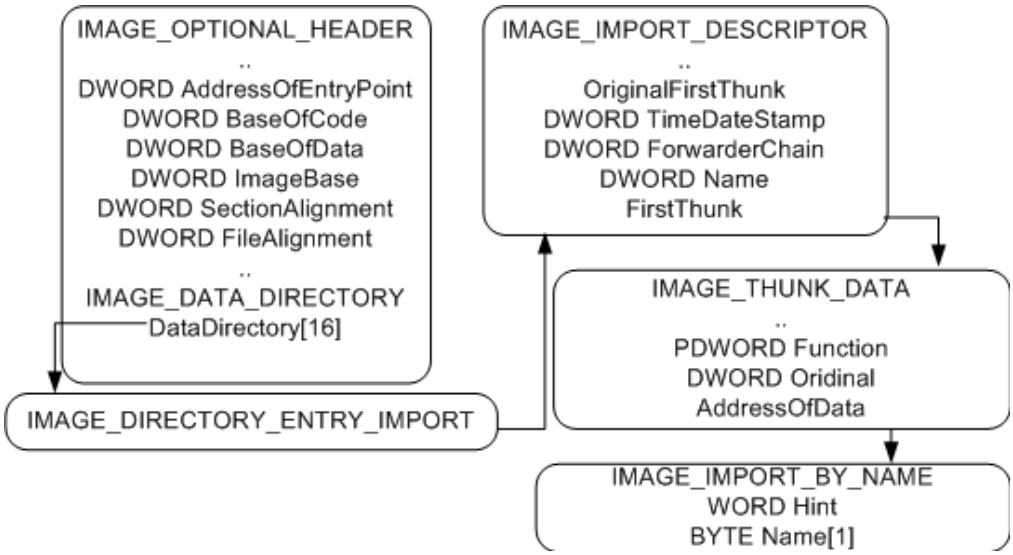
```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE    Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
            DWORD   PhysicalAddress;
            DWORD   VirtualSize;
    } Misc;
    DWORD   VirtualAddress;
    DWORD   SizeOfRawData;
    DWORD   PointerToRawData;
    DWORD   PointerToRelocations;
    DWORD   PointerToLinenumbers;
    WORD    NumberOfRelocations;
    WORD    NumberOfLinenumbers;
    DWORD   Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

주요 필드

Name	섹션의 네임을 나타내는 멤버 *섹션의 속성을 파악하기 위해선 Characteristics 멤버를 참조해야함
PhysicalAddress / VirtualSize	PE로더에의해 이미지가 메모리에 올려진 후에 해당 섹션이 얼마만큼의 크기를 가지는지의 정보. *물리적 주소정보 아님
VirtualAddress	PE로더에의해 이미지가 메모리에 올려진 후에 해당 섹션이 어느 주소에 위치하는지의 RVA 주소값 *IMAGE_OPTIONAL_HEADER 의 멤버인 SectionAlignment 의 배수값을 가짐
SizeOfRawData	Raw Data 상에서 해당 섹션에 실제 사용된 Size 정보 *섹션의 영역은 FileAlignment 의 영향을 받음
PointerToRawData	Raw Data가 파일상의 어느 주소에 위치해 있는지 나타내는 변수
Characteristics	해당섹션에 대한 속성정보 flag 값을 상수 매크로로 정의 IMAGE_SCN_CNT_CODE 0x00000020 :코드로 채워진 섹션 IMAGE_SCN_CNT_INITIALIZED_DATA 0x00000040 :데이터가 초기화된 코드 IMAGE_SCN_CNT_UNINITIALIZED_DATA 0x00000080 :데이터가 비초기화된 섹션 IMAGE_SCN_MEM_EXECUTE 0x20000000 :코드로서 실행될 수 있는 섹션 IMAGE_SCN_MEM_READ 0x40000000 : 읽기 가능역역 섹션 IMAGE_SCN_MEM_WRITE 0x80000000 : 쓰기 가능영역 섹션

4. IMPORT TABLE (IMAGE_IMPORT_DESCRIPTOR)

: Import Section – 사용하고자 하는 익스포트 함수들과 그 DLL 에 대한 정보를 보관하고 있는 곳. 일반적으로 섹션 테이블에는 .idata 라는 이름으로 지정됨



〈import 정보를 가진 구조체의 관계〉

1) OptionalHeader 구조체의 DataDirectory 의 주소를 구한다.
2) DataDirectory의 두 번째 인덱스(import table)에서 VirtualAddress 멤버값을 참조한다.(IMAGE_IMPORT_DESCRIPTOR의 위치)
   *임포트 섹션 헤더의 VirtualAddress 필드와 동일한 값을 가진다
3) IMAGE_IMPORT_DESCRIPTOR 의 구조

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        DWORD   Characteristics;            // 0 for terminating null import descriptor
        DWORD   OriginalFirstThunk;         // RVA to original unbound IAT (PIMAGE_THUNK_DATA)
    };
    DWORD   TimeDateStamp;                  // 0 if not bound,
                                            // -1 if bound, and real date\time stamp
                                            //     in IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT (new BIND)
                                            // O.W. date/time stamp of DLL bound to (Old BIND)

    DWORD   ForwarderChain;                 // -1 if no forwarders
    DWORD   Name;
    DWORD   FirstThunk;                     // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED *PIMAGE_IMPORT_DESCRIPTOR;
```

주요 필드

OriginalFirstThunk	IMAGE_THUNK_DATA 구조체의 배열을 가리키는 RVA 값 IMAGE_THUNK_DATA 배열의 원소는 IMAGE_IMPORT_BY_NAME 이라는 구조체를 가리키는 RVA 값을 가짐
TimeDateStamp	바인딩되지 않았을 경우 0 , 바인딩이후엔 -1 값을 가짐
ForwarerChain	DLL 포워딩과 TimeDateStamp 와 관련있음
Name	임포트된 DLL의 이름을 담고 있는 NULL로 끝나는 ASCII 문자열에 대한 RVA 값
FirstThunk	IMAGE_THUNK_DATA 구조체의 배열을 가리키는 RVA 값 *PE File 이 가상 주소 공간에 매핑되면 IMAGE_THUNK_DATA 는 실제 해당 주소의 함수 포인터를 담게 된다. 이 배열을 Import Address Table(IAT) 이라고 한다. 실제 해당주소가 IAT 에 설정되면 이것을 바인딩되었다고 한다.

-*OriginalFirstThunk , FirstThunk ?

```
typedef struct _IMAGE_THUNK_DATA32 {
    union {
        PBYTE  ForwarderString;
        PDWORD Function;
        DWORD Ordinal;
        PIMAGE_IMPORT_BY_NAME  AddressOfData;
    } u1;
} IMAGE_THUNK_DATA32;
typedef IMAGE_THUNK_DATA32 * PIMAGE_THUNK_DATA32;
```

〈OriginalFirstThunk , FirstThunk 가 가리키는 IMAGE_THUNK_DATA 의 구조〉

ForwarderString	DLL에서 임포트할 함수가 포워딩된 함수일 경우를 위한 필드. *포워딩된 함수의 경우 익스포트 섹션의 익스포트 함수 포인터 내의 엔트리 값이 실제 함수 진입점 주소가 아니라 포워딩된 대상 DLL과 실제 함수를 나타내는 문자열에 대한 주소가 됨
Function	PE file이 메모리에 매핑된 후에 IAT 에 저장된 함수 포인터를 직접 참조할 때 Function 필드를 통해서 참조한다.
Ordinal	모듈 정의 파일을 통해서 구체적으로 서수 파일을 지정하게 되면 링커는 DLL 링크 시 함수명(symbol)을 통한 링크가 아니라 서수를 통해서 사용된 함수를 링크하게 된다. 이때 사용되는 필드
AddressOfData *OriginalFirstThunk 는 정확히 이 값을 가리킨다.	IMAGE_THUNK_DATA 의 값이 IMAGE_IMPORT_BY_NAME 구조체의 시작 번지를 가리킬 때 사용돼는 필드

	typedef struct _IMAGE_IMPORT_BY_NAME { WORD Hint; BYTE Name[1]; } IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME; 〈IMAGE_IMPORT_BY_NAME 의 구조〉 Hint : 서수를 지정하지 않은 경우에 임의로 지정하는 임포트된 함수에 대한 0으로 시작하는 인덱스. 차례대로 1씩 증가. 이 값은 필수요소가 아니며 어떤링커는 이 값을 0으로 만든다. Name[1] : NULL 로 끝나는 아스키 문자열로 이루어진 해당 DLL에 사용된 익스포트 함수의 이름을 나타냄

OriginalFirstThunk –〉 INT (Import Name Table)

&

FirstThunk –〉 IAT (Import Address Table)

– 메모리에 매핑되기 전 INT 와 IAT 는 동일한 위치를 가리킴

– memory 에 매핑이 되면 로더는 OriginalFirstThunk 가 가리키는 INT 배열을 참조해서 임포트할 함수에 대한 정보를 얻어오게 되고, 가상 주소 공간에서 함수 포인터를 획득하게 됨

– 이 함수 포인터는 IAT 에 기록이 되고 이제 FirstThunk 는 IMAGE_THUNK_DATA 의 Function 필드를 통해 실제 함수의 함수 포인터를 이용하여 해당 함수를 호출하게 됨 (이 시점에서 INT 와 IAT 의 값이 달라짐)

5. Export Table (IMAGE_EXPORT_DIRECTORY)

:PE File 이 메모리에 매핑될 때 필요한 DLL들을 해당 프로세스 주소 공간에 로드하고 메인 프로그램에서 필요한 함수를 구하기 위해 로드된 DLL에서 참조하는 곳

-PE 파일이 DLL에서 Export 함수를 참조하는 방식을 정의할 때,
보통 IMAGE_EXPORT_DIRECTORY 구조체의 AddressOfName 필드를 참조하거나 , 함수의 Ordinal 을 이미 알고 AddressOfFunction 필드를 참조하는 두 가지 방식을 가짐

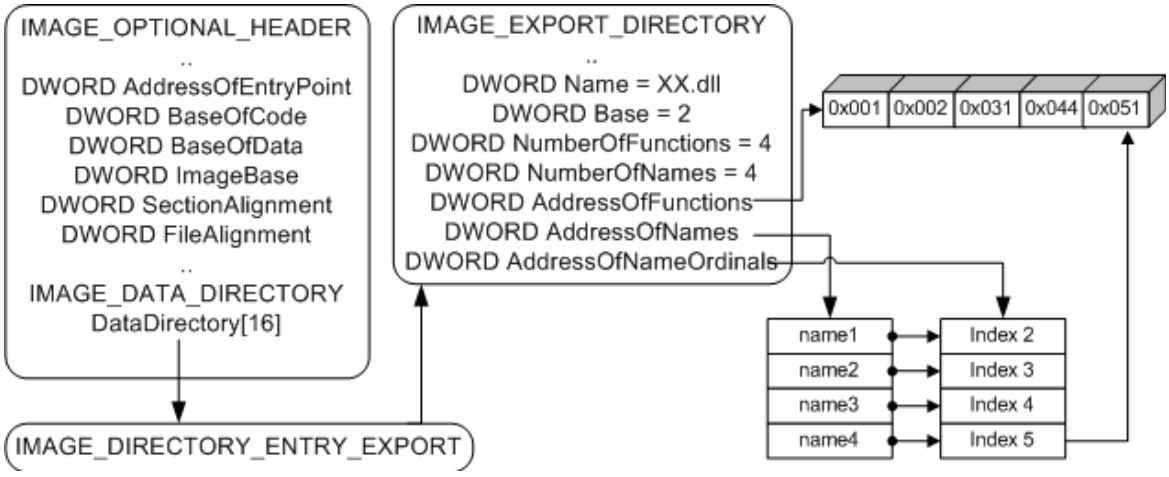
```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;
    DWORD   TimeDateStamp;
    WORD    MajorVersion;
    WORD    MinorVersion;
    DWORD   Name;
    DWORD   Base;
    DWORD   NumberOfFunctions;
    DWORD   NumberOfNames;
    DWORD   AddressOfFunctions;     // RVA from base of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

〈IMAGE_EXPORT_DIREXTORY 구조체〉

주요 필드

Name	해당 DLL 의 이름을 나타내는 아스키코드 문자열의 위치 RVA
Base	익스포트된 함수들에 대한 서수의 시작 번호 AddressOfFunction 배열에 대한 index를 얻기위해 ordinal 을 보정하는데 사용되는 값 *ordinal – base
NumberOfFunctions	AddressOfFunction 필드가 가리키는 RVA의 배열의 원소개수
NumberOfNames	AddressOfNames 필드가 가리키는 RVA의 배열의 원소의 개수와 AddressOfNameOridinals 필드가 가리키는 서수 배열의 원소의 개수를 나타냄. *실제 익스포트 된 함수의 정확한 개수를 나타냄
AddressOfFunctions	익스포트된 함수들의 함수 포인트를 가진 배열을 가리킴. RVA
AddressOfNames	익스포트된 함수의 심볼을 나타내는 문자열 포인터 배열을 가리키는 RVA 값
AddressOfNameOridinals	AddressOfName 배열에 있는 함수 심볼과 연관된 WORD 타입의 배열에 대한 포인터. 이 WORD 타입의 값들은 익스포트된 모든 함수들의 서수를 담고 있음.

-Export 함수의 주소를 찾아가는 과정



〈Export Table 의 구조〉

1) 이름으로 참조할 때

Export Table에 NumberOfNames에서 Name 원소의 개수를 확인

AddressOfName 과 AddressOfNameOrdinals 을 병렬적으로 수행한 후 AddressOfName에서 Name 이 발견되면 AddressOfNameOrdinals에서 이와 연결된 값을 구함.

AddressOfNameOridinals 배열에서 얻은 값을 AddressOfFunction배열의 index 로 사용. 그 값이 그 함수의 RVA

2) Ordinal 로 참조

Export Table에서 Base 값을 구함. Base 값이 서수의 시작번호이므로 Oridinals에서 Base 값을 빼면 AddressOfFunction 배열에 대한 index를 얻게 됨

AddressOfFunction 배열에서 index 값을 이용해 함수의 RVA 를 구함

〈실행압축 (Packing) 이란〉

-일반압축과 같이 여러 파일을 하나로 묶어 압축을 수행하는 것이 아니라, 하나의 실행 파일을 파일의 형태를 그대로 유지하면서 Size 를 줄여 주는 압축방식

-확장자를 그대로 유지하면서 파일의 실행도 전과 같이 이루어짐

-패킹 툴의 종류, 버전과 패킹 방식이 다양함

ASPack –〉 Alexey Solodovnikov Pack, UPX , ASprotect , NeoLite , Armadillo, Exeshield, Pecompect , PEncrypt, CryptFF, DBPE, tElock, Stxe, PE_Patch.AvSpoof, Bat2Exe.BDTmp, Batlite, ExeStealth, JDPack, PECRC, PE_Patch.Elka, Pex, Pingvin, Mmpo, Embedded CAB, Morphine, Eagle, PE-Crypt.Negn, Bat2Exe, PCPEC, FlySFX, Exe2Dll, Teso, PE-Crypt.UC, Polyene, PE-Crypt.UC, PEBundle, CryptFF, DBPE, BitArts.Fusion, PE_Patch.Aklay, TapTrap, CryptZ, PE-Crypt.Moo, PE-Pack, RarSFX, XCR, ZipSFX, DoomPack, NDrop, PECrc32, DebugScript, PE_Patch.Ardurik, PE-Crypt.Wonk, PE_Patch.Upolyx, MEW, PE_Patch.ZiPack, ZiPack, CryptFF.b, MEW 등등...

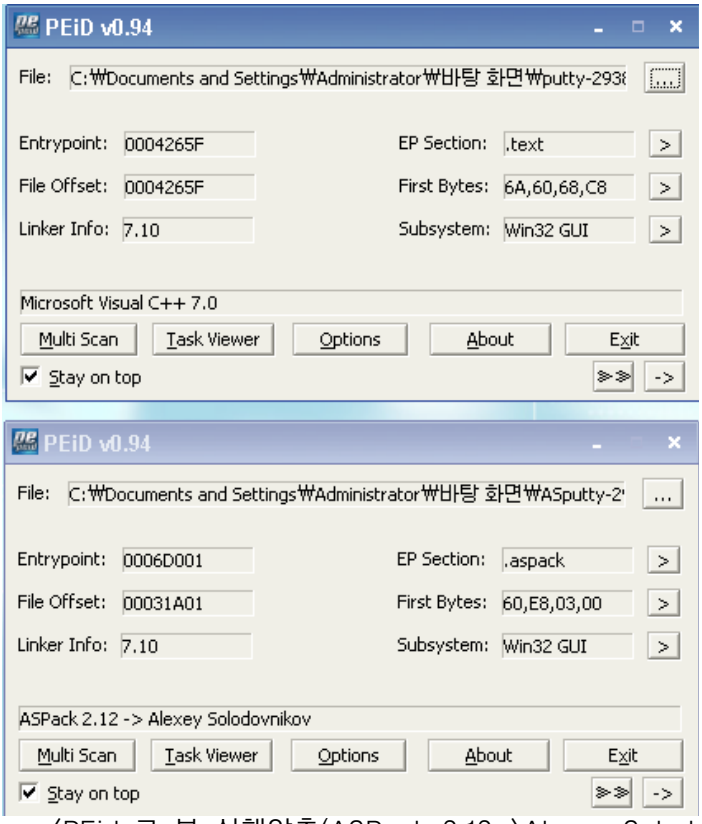
-최근 발견되는 악성 코드들은 대부분 실행 압축과 파일 보호 기법으로 악의적인 코드를 숨기고 있음

*PE file 구조를 이해해야 악성 코드 분석을 위한 실행 압축 해제를 수행할 수 있음

〈실행 압축의 기본적인 원리〉

-packing 을 수행하는 프로그램이 압축을 수행할 프로그램의 실제 Code 및 Data 를 다른 영역에 압축, 저장한 후 프로그램의 Entry Point 를 실행압축 해제 루틴을 먼저 가리키게 한 후, 실행압축 해제(Unpacking) 가 먼저 이루어진 후에 프로그램이 동작하는 방식

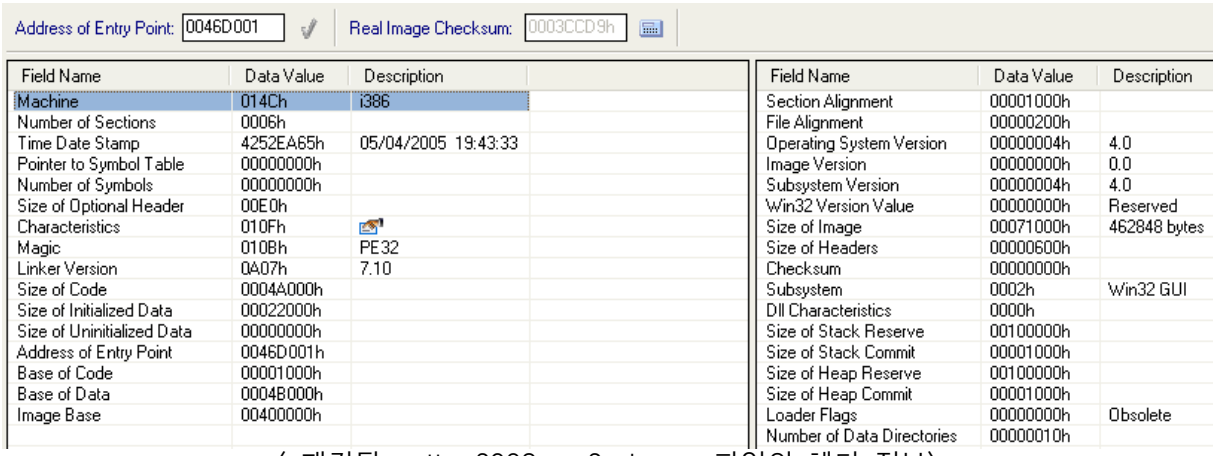
◇Target File : putty-2938-rm2rob.exe



〈PEid 로 본 실행압축(ASPack 2.12-〉Alexey Solodovnikov)된 파일과 원본 파일의 모습〉

* Entry Point 가 다른 위치의 섹션을 가리키고 있습니다.

-PE explorer 를 사용하여 두 파일의 차이점을 분석



Field Name	Data Value	Description
Machine	014Ch	i386
Number of Sections	0006h	
Time Date Stamp	4252EA65h	05/04/2005 19:43:33
Pointer to Symbol Table	00000000h	
Number of Symbols	00000000h	
Size of Optional Header	00E0h	
Characteristics	010Fh	
Magic	010Bh	PE32
Linker Version	0A07h	7.10
Size of Code	0004A000h	
Size of Initialized Data	00022000h	
Size of Uninitialized Data	00000000h	
Address of Entry Point	0046D001h	
Base of Code	00001000h	
Base of Data	0004B000h	
Image Base	00400000h	

Field Name	Data Value	Description
Section Alignment	00001000h	
File Alignment	00000200h	
Operating System Version	00000004h	4.0
Image Version	00000000h	0.0
Subsystem Version	00000004h	4.0
Win32 Version Value	00000000h	Reserved
Size of Image	00071000h	462848 bytes
Size of Headers	00000600h	
Checksum	00000000h	
Subsystem	0002h	Win32 GUI
Dll Characteristics	0000h	
Size of Stack Reserve	00100000h	
Size of Stack Commit	00001000h	
Size of Heap Reserve	00100000h	
Size of Heap Commit	00001000h	
Loader Flags	00000000h	Obsolete
Number of Data Directories	00000010h	

〈 패킹된 putty-2938-rm2rob.exe 파일의 헤더 정보〉

Address of Entry Point: 0044265F Real Image Checksum: 0006A772h

Field Name	Data Value	Description
Machine	014Ch	i386
Number of Sections	0004h	
Time Date Stamp	4252EA65h	05/04/2005 19:43:33
Pointer to Symbol Table	00000000h	
Number of Symbols	00000000h	
Size of Optional Header	00E0h	
Characteristics	010Fh	
Magic	010Bh	PE32
Linker Version	0A07h	7.10
Size of Code	0004A000h	
Size of Initialized Data	00022000h	
Size of Uninitialized Data	00000000h	
Address of Entry Point	0044265Fh	
Base of Code	00001000h	
Base of Data	0004B000h	
Image Base	00400000h	

Field Name	Data Value	Description
Section Alignment	00001000h	
File Alignment	00001000h	
Operating System Version	00000004h	4.0
Image Version	00000000h	0.0
Subsystem Version	00000004h	4.0
Win32 Version Value	00000000h	Reserved
Size of Image	0006D000h	446464 bytes
Size of Headers	00001000h	
Checksum	00000000h	
Subsystem	0002h	Win32 GUI
Dll Characteristics	0000h	
Size of Stack Reserve	00100000h	
Size of Stack Commit	00001000h	
Size of Heap Reserve	00100000h	
Size of Heap Commit	00001000h	
Loader Flags	00000000h	Obsolete
Number of Data Directories	00000010h	

〈원본 putty-2938-rm2rob.exe 파일의 헤더 정보〉

변경된 필드 값

필드네임	패킹된 putty	원본 putty
NumberOfSection	0006h	0004h
AddressOfEntryPoint	0046D001h	0044265Fh
FileAlignment	00000200h	00001000h
SizeOfImage	00071000h	0006D000h
SizeOfHeaders	00000600h	00001000h

: 변경된 Field 값으로 패킹된 이후 새로운 섹션이 2개 추가되었고, EntryPoint의 위치가 변경되었으며, 파일의 사이즈와 헤더의 크기가 변경된 것을 알 수 있습니다. FileAlignment 값으로 파일상에서의 섹션의 배치간격이 줄어든걸 알 수 있습니다.

DATA DIRECTORIES

Export Table 00000000 00000000

Directory Name	Virtual Address	Size
Export Table		
Import Table	0046DFACh	00000214h
Resource Table	0046A000h	00002F28h
Exception Table		
Certificate Table		
Relocation Table	0046DF54h	00000008h
Debug Data		
Architecture-specific data		
Machine Value (MIPS GP)		
TLS Table		
Load Configuration Table		
Bound Import Table		
Import Address Table		
Delay Import Descriptor		
COM+ Runtime Header		
(15) Reserved		00100000h

(패킹된 putty)

DATA DIRECTORIES

Export Table 00000000 00000000

Directory Name	Virtual Address	Size
Export Table		
Import Table	00460DD8h	000000DCh
Resource Table	0046A000h	00002F28h
Exception Table		
Certificate Table		
Relocation Table		
Debug Data		
Architecture-specific data		
Machine Value (MIPS GP)		
TLS Table		
Load Configuration Table	00460D78h	00000048h
Bound Import Table		
Import Address Table	0044B000h	0000045Ch
Delay Import Descriptor		
COM+ Runtime Header		
(15) Reserved		

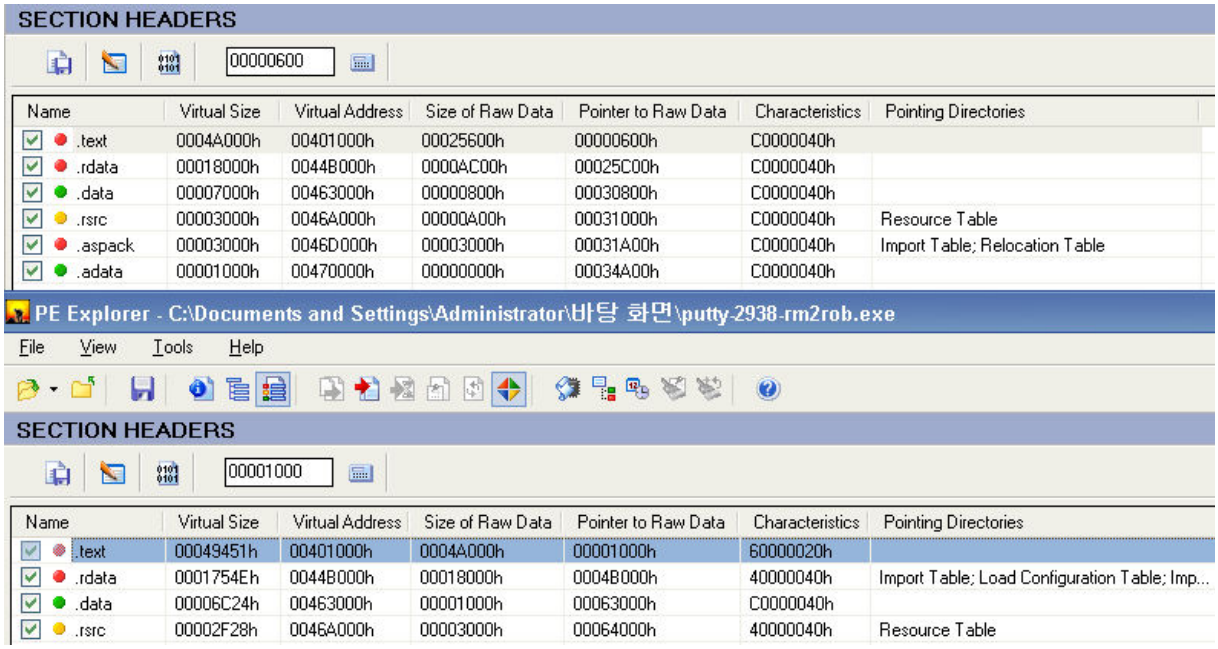
(원본 putty)

변경된 DataDirectory 정보

Directory Name	패킹된 putty (VA / Size)	원본 putty (VA / Size)
Import Table	0046DFACh / 00000214h	00460DD8h / 00002F28h
Relocation Table	0046DF54h / 00000008h	
Load Configuration Table		00460D78h / 00000008h
Import Address Table		0044B000h / 00000045h

:IMAGE_OPTIONAL_HEADER에서 참조하는 DataDirectory 의 정보

패킹된 putty에서 Import Table 의 위치와 크기가 변경되었고 , Configuration Table 과 IAT 정보가 없어지고 재배치 정보가 새로 추가된것을 확인할 수 있습니다



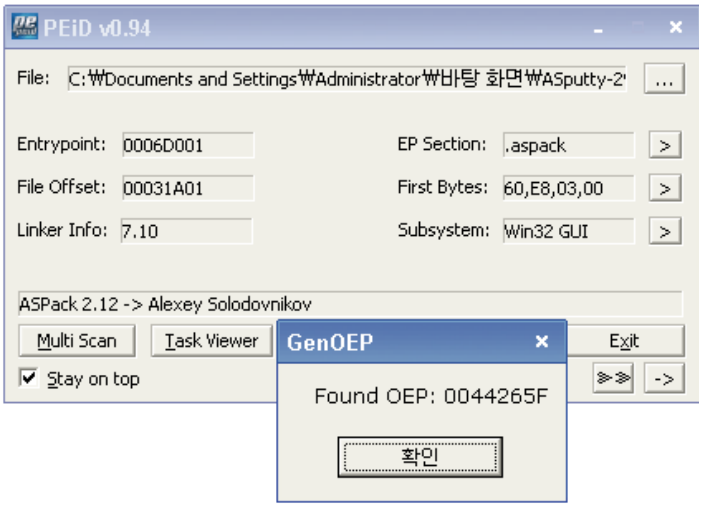
: IMAGE_SECTION_HEADER 구조체의 PointerToRawData 변수 값이 600으로 변경되었고 (FileAlignment 200 의 배수값) 새로 추가된 .aspack 섹션에 임포트 정보와 재배치 정보가 병합되어 있는 것을 알 수 있습니다. 리소스 테이블은 변경사항이 없습니다.

〈실행 압축 해제의 기본원리〉

◇OEP(Original Entry Point) 찾기

- PE file 의 내부 구조가 바뀌어도 결과적으로 실행되는 프로그램의 기본 구조 자체는 변하지 않습니다. 즉 실행압축 된 프로그램이 실행되어 패킹 루틴에 의해 처리가 된 후에는 압축되기 전 상태로 돌아가게 되며 이때 메모리 Dump 를 수행하고 Entry Point를 이 지점으로 수정해주는 것입니다.

1. PEiD plugiin 이용
: PEiD 의 generic OEP Finder 와 같은 플러그인을 이용하여 OEP 주소를 쉽게 찾을 수 있습니다. 그러나 모든 패킹 프로그램의 OEP를 찾아내지는 못합니다.

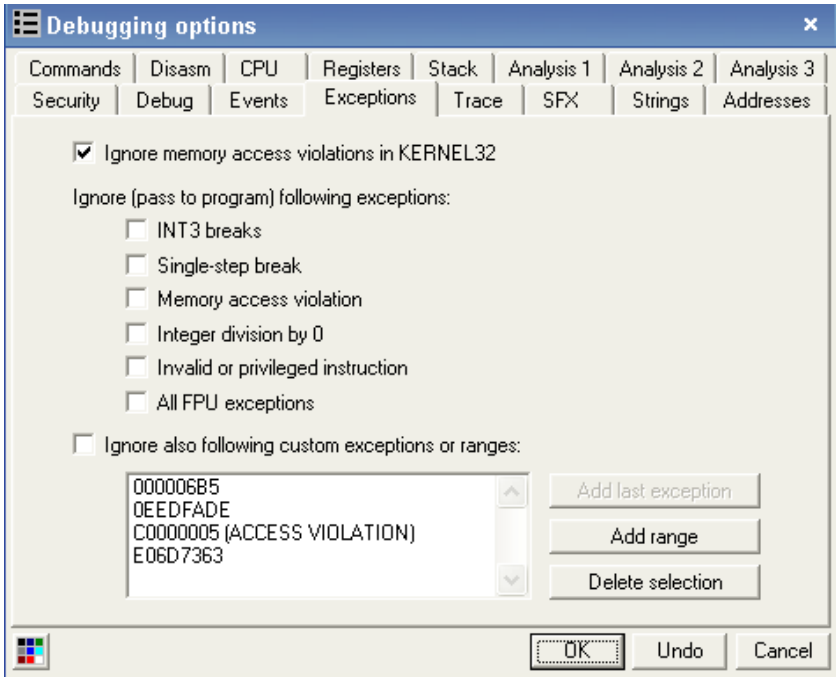


〈PEiD를 이용하여 OEP 를 찾아내는 화면〉

2. Ollydbg 로 MUP (Manual Unpacking) 수행
:Ollydbg 를 사용하여 OEP 를 찾아내는 과정은 패킹 프로그램마다 각자 다른방식의 패킹 알고리즘이 있고, 언패킹을 방지하기위한 트릭이나 안티디버거가 존재하기 때문에 다양한 방법으로 과정을 수행할 수 있습니다.

*SEH 예외처리 이용
-대부분의 패킹 툴은 디버깅을 방지하기 위해 고의로 예외처리를 발생시키는 코드를 삽입합니다. Ollydbg 에서는 예외처리 옵션에서 이와같은 예외가 발생하였

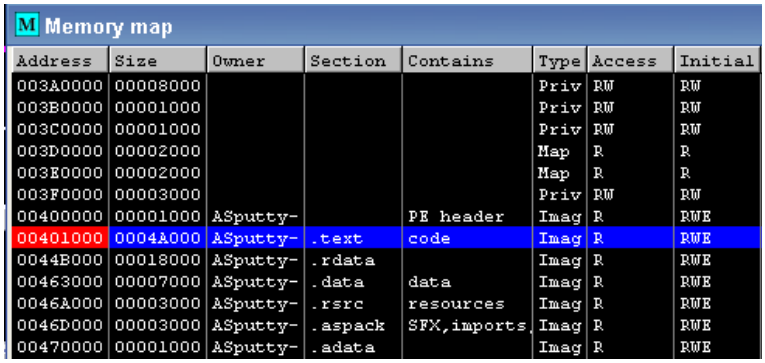
을 때 무시하도록 설정을 변경할 수 있습니다.



Ollydbg 의 Shift + F9 키를 이용하여 exception 이 발생한 코드를 무시하고 디버깅을 진행할 수 있습니다. 디버깅이 진행되다 프로그램이 실행이 되면 SEH 창에서 마지막 SE handler 를 확인한 예외처리 루틴이 위치한 주소로 이동합니다. 그리고 예외처리 루틴을 Reference 한 지점을 검색하여 OEP 에 근접한 곳을 찾아낼 수 있습니다.

*Code Section 접근 확인하기

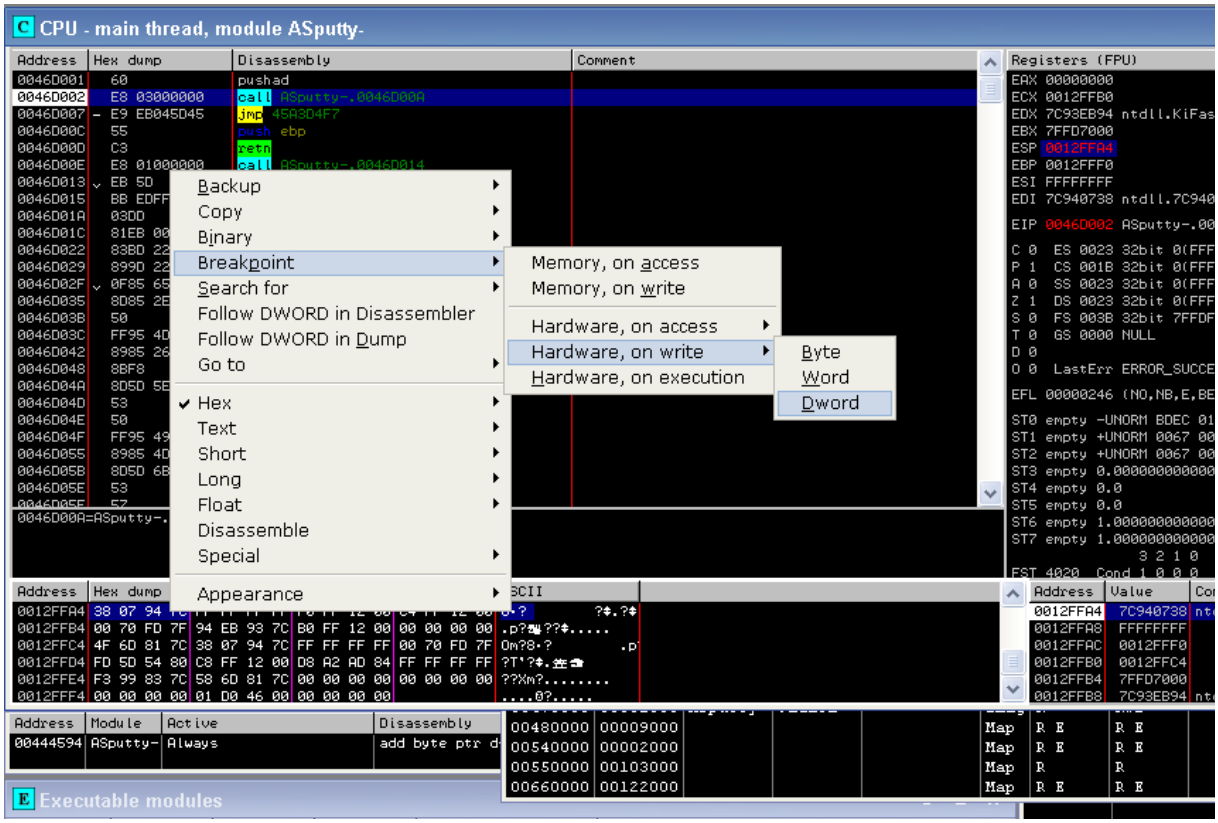
Ollydbg에서 Alt + M 키를 눌러 Memory map 구조를 확인할 수 있습니다. 여기서 코드 섹션에 브레이크 포인트를 설정한 후 실행하면 해당 섹션의 메모리에 접근하는 순간을 잡을 수가 있습니다.



* BreakPoint Hardware.on access Dword 이용하기

PEtite 2.2 나 ASpack 2.12 등으로 패킹된 프로그램을 언패킹하면 패킹 루틴의 시작지점에 pushad 어셈블리 명령어를 볼 수 있습니다. 이 명령어는 모든 레지스터 값을 스택에 저장하는 명령어입니다. 주로 압축이 모두 풀리면 원래의 레지스터들을 복원하기 위해 사용됩니다.

이런 과정을 이용해서 OEP 에 접근할 수가 있습니다. pushad 명령어에서 F8을 한번 눌러 명령을 실행하면 레지스터 값들이 스택에 저장이 되고 ESP 스택포인터는 마지막으로 삽입된 그 값들을 가리키고 있습니다. ESP 를 Follow in Dump 를 실행하여 Hex dump 창에 나타난 값에 4 byte 를 지정해 준 후에 메뉴에서 BreakPoint Hardware.on access Dword 를 선택한후 실행을 하게되면 지정된 메모리에 접근할 때 break 가 되고 popad 와 같은 레지스터를 복원하는 명령어를 볼 수 있습니다. 그후에 OEP 값을 찾아낼 수 있습니다.

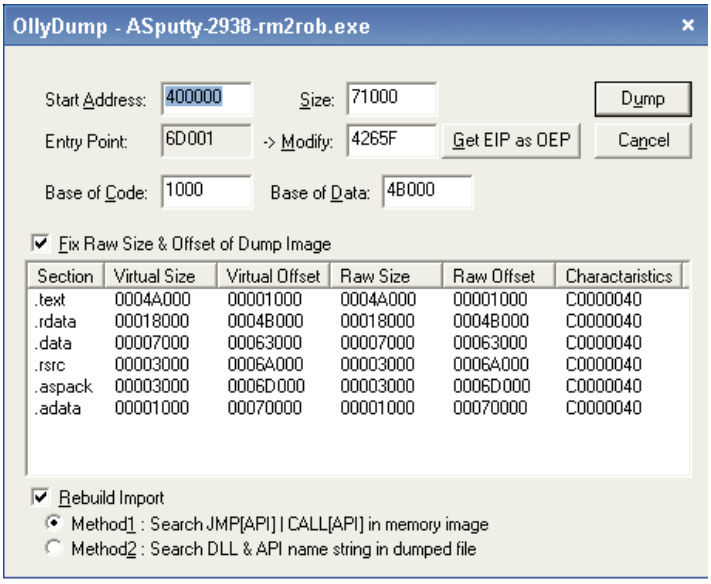


◇IAT(Import Address Table) 복구하기

- 패킹 프로그램 마다 경우가 틀리지만 대부분의 패킹 프로그램은 IAT 정보를 변경하거나 사용할 Import function 들을 에뮬레이션 해주고 IAT엔 쓰레기 값을 넣는 경우가 발생하기 때문에 IAT 정보를 재구성해 주어야 합니다.
  지금은 좋은 툴이 많아서 자동으로 IAT 복구를 수행해주고 있습니다.

1. OllyDump 이용

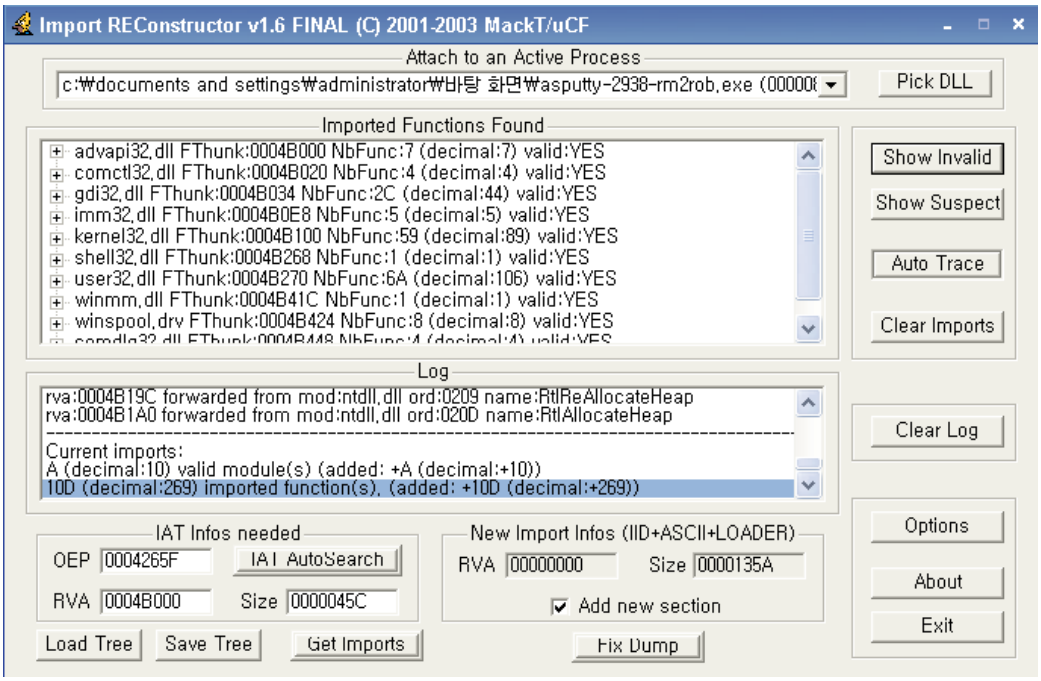
Ollydbg에서 OEP 값을 찾아낸 후에 Ollydump 플러그인을 이용하여 바로 메모리 덤프를 수행할 수 있습니다.



Rebuild Import 를 체크하면 자동으로 IAT 정보를 수정하여 덤프를 수행합니다.

2. ImportREC 이용

ImportREC 를 실행하여 디버깅을 수행중인 패킹된 프로그램을 attach 한 후 찾아낸 OEP 값을 입력해주면 자동으로 RVA 값과 Import Function 들을 찾아줍니다.



Fix Dump를 선택하여 언패킹한 프로그램에 IAT 를 복구해주면 됩니다.

☀참고문서, 사이트☀

★ Windows 시스템 실행파일의 구조와 원리 (이호동 저)
★ OPEN REVERSE FORUMS
https://ampm.ddns.co.kr/~reverse/phpBB/index.php
★ BSW-Powered by vBulletion
http://codediver.kaist.ac.kr/vbulletin/index.php
★ PE Format 완전 분석 (김경곤 (A.K.A. Anasra))