윈도우 프로그래머를 위한 PE 포맷 가이드

실행파일 속으로

목차

License

Copyright © 2007, 신영진
이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr

소개

운영체제의 실행 파일 속에는 많은 비밀이 숨겨져 있다. Windows의 기본 실행 파일 포맷인 PE 포맷의 전체적인 구조에 대해서 살펴보고, DLL의 익스포트, 임포트 함수들을 보여주는 프로그램을 제작해 보자.

연재 가이드

운영체제: 윈도우 2000/XP
개발도구: Visual Studio 2005
기초지식: C/C++, Win32 API
응용분야: IAT 패칭, EAT 패칭 프로그램, PE 분석기

연재 순서

2007. 08. 실행파일 속으로

필자소개

신영진 pop@jiniya.net, http://www.jiniya.net

웰비아닷컴에서 보안 프로그래머로 일하고 있다. 시스템 프로그래밍에 관심이 많으며 다수의 PC 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. C와 C++, Programming에 관한 이야기를 좋아한다.

필자 메모

얼마 전 모 방송사의 게임리그의 결승전이 있었다. 5전 3선승제의 결승전은 접전을 거듭하며 5경기까지 갔고, 5경기도 매우 오랜 시간 싸움 끝에 결말이 났다. 게임을 끝내고 나온 두 선수는 각각 인터뷰를 했고, 패전 선수의 인터뷰 첫 마디는 if였다. 자신이 큰 실수를 하지 않았으면 이길 수 있었다는 요지의 말이었다. 필자가 응원했던 선수라 실망감이 더 클 수 밖에 없었다.

A라는 사건의 결과가 잘못됐을 때 우리가 그 결과에 접근하는 시각은 크게 변명과 반성으로 압축된다. 변명은 결과 이전의 과거 상태에 집착하는 것이고, 반성은 결과 이후의 미래에 대해서 생각하는 것이다. "시간은 과거에서 미래로 흐르고, 지나간 시간은 절대로 다시 돌아오지 않는다."라는 지극히 평범한 상식에 비추어 보아도 변명보다는 반성이 훨씬 더 생산적인 접근 방법이라는 것을 알 수 있다. 이와 관련해서 이외수님 플레이톡(http://playtalk.net/oisoo)에 재미있는 글이 있어서 소개해 본다.

다른 나라와의 축구경기에서 우리 선수들이 부진한 모습을 보이면 해설자들이 그라운드 상태가 엉망이기 때문이라는 둥, 비가 와서 잔디가 미끄럽기 때문이라는 둥 하는 따위의 변명을 상투적으로 늘어 놓는다. 아놔, 상대편 선수들은 명왕성에 가서 따로 경기하고 있냐, 그리고 비는 우리 선수들만 쫓아 다니면서 쏟아지고 있냐. 변명을 많이 할수록 발전은 느려지고 반성을 많이 할수록 발전은 빨라진다. 이것은 개인에게도 적용되는 일종의 법칙이다.

Introduction

Windows 프로그램을 만들다 보면 누구나 한번쯤은 실행 파일에 관련된 궁금증을 가진다. 운영체제는 어떻게 프로그램을 실행하는 것일까? 실행 파일에 사용된 DLL들은 어떻게 로드되는 것일까? DLL의 함수는 어떤 방식으로 호출되는 것일까? 리버싱에 관심이 있는 독자라면 다음과 같은 궁금증을 가져본 적도 있을 것이다. 패커나 언패커는 어떤 식으로 동작하는 것일까? 바이러스는 어떻게 실행 파일을 감염 시키는 것일까? 다른 실행 파일에 임의의 코드를 추가할 순 없을까? 조금 거리가 있지만 이런 것들을 궁금해 하시는 분들도 있다. 자동 풀림 압축 파일은 어떻게 만드는 것일까? 이미지 파일을 입력하면 그것을 보여주는 실행 파일을 생성해내는 프로그램의 원리는 무엇일까?

이 모든 궁금증, 이 모든 질문에 대한 해답은 PE 포맷에 있다. PE는 Portable Executable의 약자로 Windows 운영체제에서 사용하는 표준 실행 파일 포맷이다. 이 포맷은 Windows 95에서 Windows Server 2003까지, 32비트에서 64비트에 이르기까지 전 Windows 운영체제에서 공통적으로 사용된다.

Windows 95때부터 사용된 파일 포맷인 만큼 이 포맷을 분석한 자료는 정말 많다. 구글에서 "PE format"이란 키워드로 검색해 보면 얼마나 많은지 알 수 있다. 대표적인 것으로 참고 자료에 있는 Matt Pietrek의 "Peering Inside the PE: A Tour of the Win32 Portable Executable File Format"와, 이호동님의 "Windows 시스템 실행 파일의 구조와 원리"가 있다. 바이블과 같은 자료이기 때문에 보다 깊이 있는 이해를 하고 싶다면 반드시 읽어 보도록 하자.

우리는 PE 포맷에 대해서 간단하게 분석하고 앞서 열거한 질문에 대한 해답을 하나씩 찾아 볼 것이다. 이번 시간은 그 시작으로 PE 포맷의 개략적인 구조와 임포트, 익스포트 테이블에 관해 다룬다. 이 글을 통해서 실행 파일에서 어떻게 다른 DLL의 함수를 참조하는지, DLL에서는 어떻게 자신의 함수를 외부로 노출시키는지에 대해서 배울 수 있다.

PE 포맷의 전체적인 구조

PE 포맷을 분석하기에 앞 서 가장 먼저 해야 할 일은 간단한 구조의 PE 포맷을 가진 파일을 구하는 것이다. 일반적인 실행 파일이나 DLL의 경우 많은 프로그램에 의존적이고 코드가 복잡하기 때문에 PE 파일의 구조도 복잡하다. <리스트 1>과 <리스트 2>에는 최대한 간단한 구조의 PE 파일을 생성하기 위한 DLL과 EXE 프로그램의 소스 코드가 나와있다. 앞으로 PE 포맷의 분석은 이 두 파일을 기준으로 한다.

리스트 1 dummydll 소스 코드

```
#include <windows.h>

BOOL WINAPI DllEntryPoint(HINSTANCE, DWORD, LPVOID)
{
    return TRUE;
}

extern "C" __declspec(dllexport) int
Plus(int a, int b)
{
    return a + b;
}

extern "C" __declspec(dllexport) void
PrintMsg(const char *msg, DWORD len)
{
    HANDLE out = GetStdHandle(STD_OUTPUT_HANDLE);

    DWORD written = 0;
    WriteConsole(out, msg, len, &written, NULL);
}
```

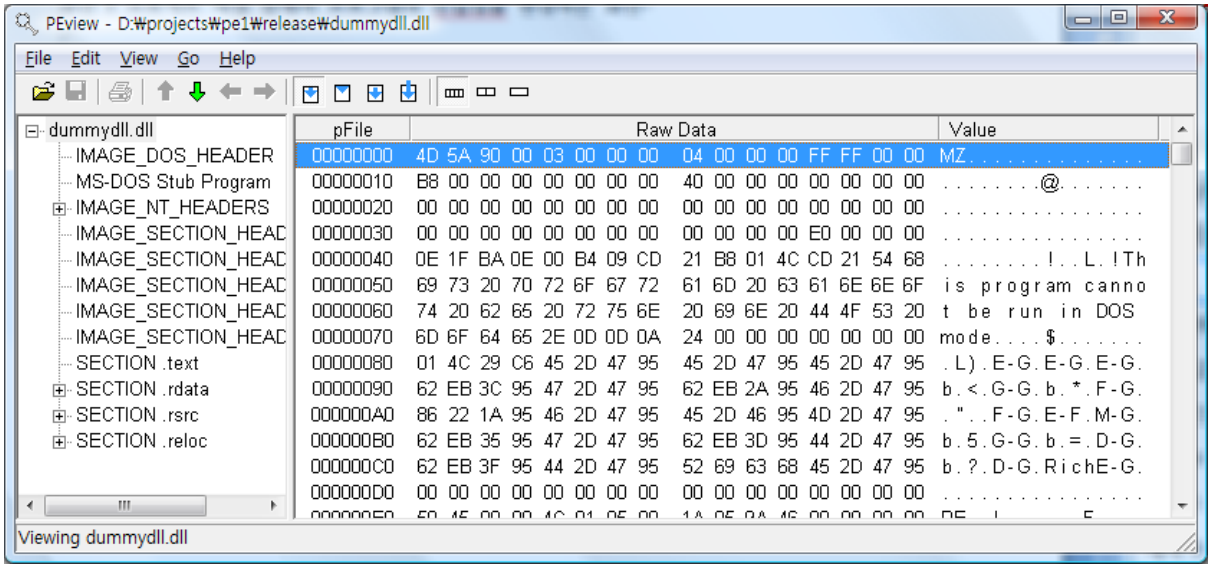
리스트 2 dummyexe 소스

```
#include "stdio.h"
#include <string.h>
```

```
#include <windows.h>
#pragma comment(lib, "dummydll.lib")

extern "C" __declspec(dllimport) int
Plus(int a, int b);

extern "C" __declspec(dllimport) void
PrintMsg(const char *msg, DWORD len);

int EntryPoint()
{
    Plus(3, 4);

    char *buf = "Hello\n";
    PrintMsg(buf, strlen(buf));

    Sleep(1000);
    return 0;
}
```

dummydll, dummyexe의 경우 둘 다 코드를 간단하게 만들기 위해서 CRT 함수를 사용하지 않았다. CRT와의 링크를 완전하게 제거하기 위해서는 프로그램의 진입점또한 main이나 dllmain이 아닌 우리가 만든 함수로 바꾸어야 한다. 두 프로그램을 빌드하기 위해서는 프로젝트 속성 창에서 진입점을 각각 DllEntryPoint, EntryPoint로 바꾸어 주어야 한다.

프로그램을 빌드 했다면 이제 실제로 PE 포맷이 어떻게 생겼는지 알아볼 차례다. 우리는 PEView를 통해서 PE 포맷의 구조를 살펴볼 것이다. PEView는 http://www.magma.ca/~wjr/에서 다운로드 받을 수 있다. PEView를 통해서 dummydll.dll을 불러온 화면이 <화면 1>에 나와있다. 왼쪽 편의 트리가 PE 포맷의 큰 구조를 보여주고, 클릭하면 오른쪽에 각 부분에 대한 파일 내용을 보여준다.



화면 1 PE Viewer를 통해서 dummydll.dll을 불러온 화면

PE 포맷의 전체적인 구조를 그림으로 표현한 것이 <그림 1>에 나와있다. 가장 윗 부분인 DOS 헤더가 파일의 시작 부분이자, 메모리 상의 가장 낮은 번지에 위치한 것이다. 아래쪽으로 갈수록 파일의 뒷부분, 메모리 상의 높은 번지가 된다. PE 포맷의 가장 중요한 개념은 섹션이다. PE 파일

의 경우 DOS 헤더, 스텁 코드, NT 헤더는 모두 공통적으로 사용되는 것들이다. 실제로 실행 파일의 코드와 데이터를 담고 있는 부분은 뒤에 따라 나오는 섹션이다. 이러한 섹션의 개수와 크기는 파일에 따라 다르다.

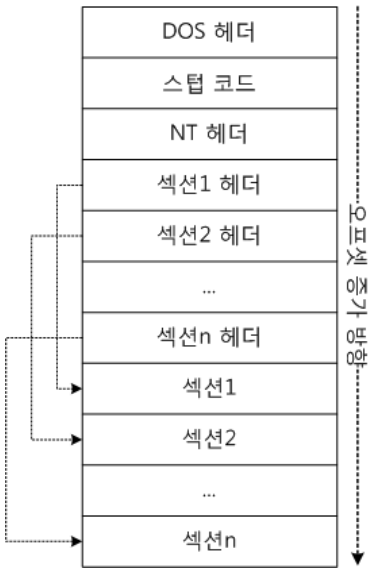


그림 1 PE 포맷 구조

DOS 헤더 및 스텁 코드

PE 이미지의 시작 부분은 DOS 헤더와 스텁 코드로 이루어져 있다. 이 부분은 과거 DOS와의 호환성을 위해서 존재하는 부분이다. Windows 프로그램을 DOS에서 실행하면 "This program cannot run in dos mode."라는 말이 출력되는 것을 본적이 있을 것이다. 이 역할을 하는 부분이 스텁 코드의 역할이다.

DOS 헤더 구조체는 IMAGE_DOS_HEADER로 정의 되어있다. 이 구조체에서는 e_magic과 e_lfanew 필드만 알고 있으면 된다. e_magic 필드는 올바른 DOS 헤더 임을 검증하기 위한 값이다. 이 값이 IMAGE_DOS_SIGNATURE와 일치하면 정상적인 DOS 헤더다. e_lfanew 필드는 NT 헤더를 가리키는 오프셋이다. 중간에 스텁 코드가 있기 때문에 NT 헤더를 읽기 위해서는 이 필드를 읽을 필요가 있다.

NT 헤더

DOS 헤더의 e_lfanew 필드를 따라가면 나오는 것이 NT 헤더다. <리스트 3>에 NT 헤더와 관련된 구조체가 나와있다. IMAGE_NT_HEADERS32는 32비트용 PE 파일의 NT 헤더 구조체다. Signature은 올바른 NT 헤더인지를 검증하기 위한 필드다. IMAGE_NT_SIGNATURE와 일치한다면 제대로 된 NT 헤더다.

NT 헤더에 포함된 FileHeader의 NumberOfSection필드는 NT 헤더 다음에 몇 개의 섹션이 나오는지를 나타낸다. NT 헤더 다음에 나타날 섹션 헤더와 섹션의 개수는 가변적이기 때문이 이 필드에

저장된 값을 기준으로 섹션을 읽어야 한다. 여기 저장된 값이 3이라면 NT 헤더 다음에 3개의 섹션 헤더와 3개의 섹션 데이터가 따라온다는 것을 의미한다. Characteristics 필드는 이미지의 종류를 나타내는 플래그다. Characteristics에 조합해서 사용되는 대표적인 값으론 <표 1>에 나타난 것들이 있다.

표 1 FileHeader의 Characteristics 플래그 의미

플래그	의미
IMAGE_FILE_RELOCS_STRIPPED	파일에 재배치 정보가 없음을 나타낸다.
IMAGE_FILE_DLL	파일이 DLL임을 나타낸다.
IMAGE_FILE_EXECUTABLE_IMAGE	파일이 OBJ, LIB등이 아닌 실행 이미지(EXE, DLL)임을 나타낸다.

OptionalHeader는 이름과는 다르게 중요한 정보를 많이 저장하고 있다. OptionalHeader의 첫 번째 필드인 Magic은 OptionalHeader의 종류를 판별하는데 사용한다. 32비트와 64비트 PE 파일을 구분하는 용도로 보통 사용된다. 32비트 PE 파일의 경우 0x10B가, 64비트 PE 파일의 경우 0x20B가 저장된다. AddressOfEntryPoint는 시작 코드의 번지를 저장하고 있는 RVA 값이다. ImageBase는 이 파일이 로드될 가상 주소를 나타낸다. EXE 파일의 경우는 4G 공간에 자신이 가장 먼저 로드되기 때문에 항상 ImageBase에 지정된 주소에 로드된다. 반면 DLL은 이미 다른 DLL이 자신이 로드하려는 주소를 사용하고 있을 수도 있다. 이 경우에는 비어있는 다른 주소에 DLL이 로드되고, ImageBase 값은 로드된 주소로 변경된다. SizeOfImage는 이 파일을 메모리에 로드하기 위해서 확보해야 하는 공간이다.

리스트 3 NT 헤더 관련 구조체들

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

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;

typedef struct _IMAGE_OPTIONAL_HEADER {
    WORD    Magic;
    BYTE      MajorLinkerVersion;
    BYTE      MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    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;

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

섹션 헤더

PE 파일은 데이터를 종류 별로 나누어서 섹션 단위로 관리한다. 섹션의 개수와 크기는 실행 파일마다 가변적이기 때문에 섹션의 정보를 저장하는 헤더가 필요하다. <리스트 4>에 나와있는 IMAGE_SECTION_HEADER 구조체에 이러한 정보가 저장된다.

리스트 4 섹션 헤더 구조체

```
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;
```

Name 필드는 섹션 이름을 저장하고 있다. VirtualAddress는 섹션이 맵핑될 메모리 번지를 나타낸다. VirtualSize는 메모리 상에서의 섹션 크기를 저장하고 있다. PointerToRawData와 SizeOfRawData는 각각 파일 상에서 섹션의 데이터가 있는 위치와 크기를 저장하고 있다. Characteristics 필드는 이 섹션의 속성을 나타내는 플래그다. <표 2>에 나와있는 값들을 조합해서 사용할 수 있다. 나머지 필드들은 EXE나 DLL 파일에서는 의미가 없는 것들이다.

표 2 섹션 속성 플래그 의미

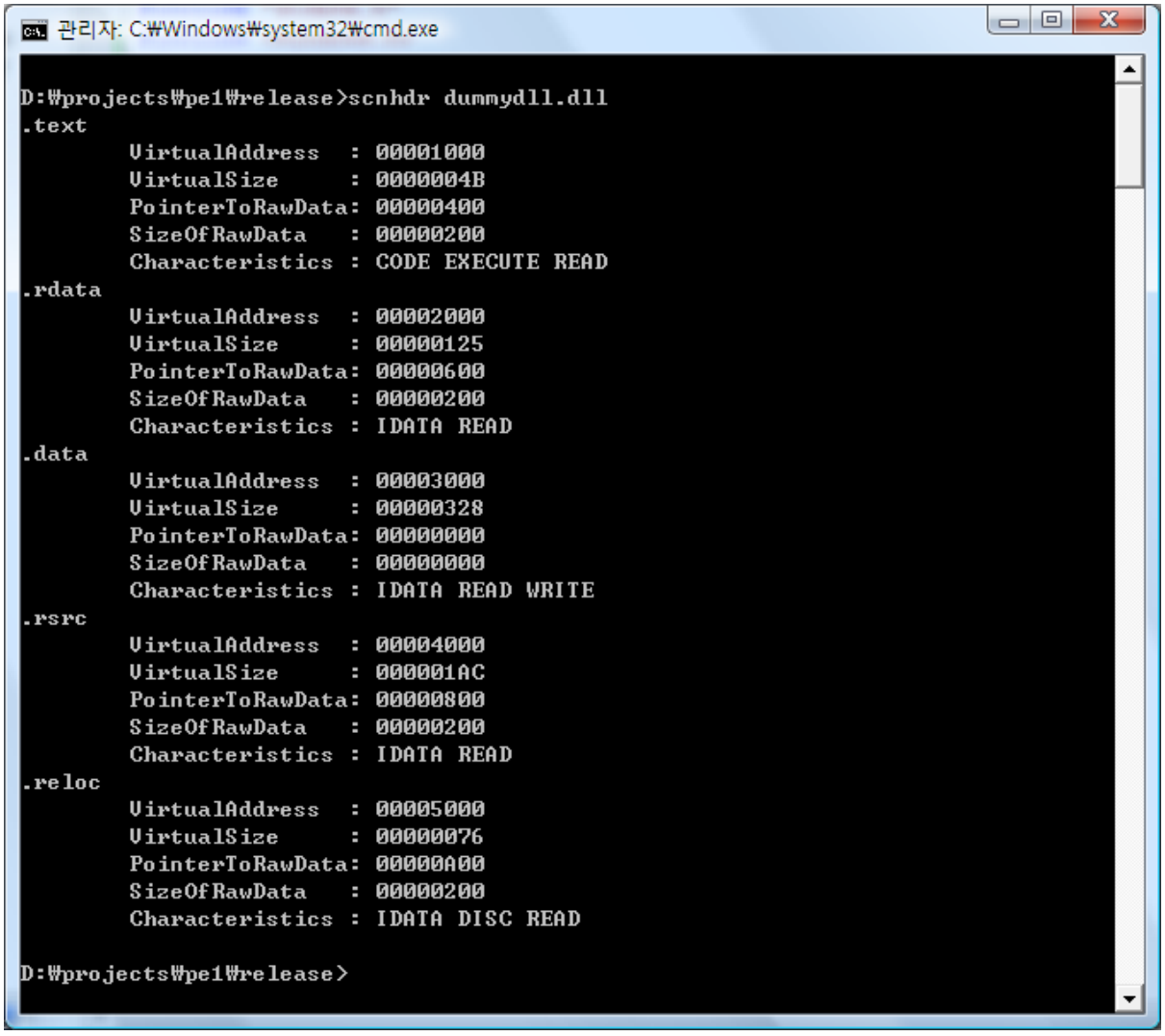
플래그	의미
IMAGE_SCN_CNT_CODE	섹션이 코드 데이터를 포함하고 있음을 나타낸다.
IMAGE_SCN_CNT_INITIALIZED_DATA	섹션이 초가화된 데이터를 포함하고 있음을 나타낸다.
IMAGE_SCN_CNT_UNINITIALIZED_DATA	섹션이 초가화되지 않은 데이터를 포함하고 있음을 나타낸다.
IMAGE_SCN_LNK_INFO	섹션이 링크 정보를 담고 있음을 나타낸다.
IMAGE_SCN_MEM_DISCARDABLE	섹션이 로딩된 후에는 필요 없음을 나타낸다. 초기 로딩 시에만 필요한 재배치 섹션이 여기 해당한다.
IMAGE_SCN_MEM_SHARED	섹션이 공유될 수 있음을 나타낸다.
IMAGE_SCN_MEM_EXECUTE	섹션의 내용을 실행할 수 있음을 나타낸다.
IMAGE_SCN_MEM_READ	섹션의 내용을 읽을 수 있음을 나타낸다.
IMAGE_SCN_MEM_WRITE	섹션에 내용을 기록할 수 있음을 나타낸다.

이미지 파일에 포함된 섹션 정보를 읽어서 출력해주는 프로그램 소스가 <리스트 5>에 나와있다. <화면 2>는 이 프로그램을 통해서 dummydll.dll의 섹션 정보를 확인하고 있는 화면이다. <리스트 5> 코드에서 GetPtr 함수는 <리스트 6>에 코드가 나와있다. 파일을 열고 맵핑해서 사용하는 부분을 유심히 보도록 하자. 이 부분은 이 후 코드에서도 반복적으로 사용되는 부분이다. 파일에서 직접 읽어도 상관은 없지만, 그렇게 할 경우는 fseek, fread등의 코드를 반복적으로 사용해야 하는 불편함이 있다.

리스트 5 섹션 헤더를 출력하는 프로그램 소스

```
#include <stdio.h>
#include <tchar.h>
#include <windows.h>
#include <sstream>
using namespace std;

void GetSectionCharacteristics(stringstream &s, DWORD c)
{
   if(c & IMAGE_SCN_CNT_CODE) s << "CODE ";
   if(c & IMAGE_SCN_CNT_INITIALIZED_DATA) s << "IDATA ";
   if(c & IMAGE_SCN_CNT_UNINITIALIZED_DATA) s << "DATA ";
   if(c & IMAGE_SCN_LNK_INFO) s << "LINKINFO ";
   if(c & IMAGE_SCN_MEM_DISCARDABLE) s << "DISC ";
   if(c & IMAGE_SCN_MEM_SHARED) s << "SHARED ";
   if(c & IMAGE_SCN_MEM_EXECUTE) s << "EXECUTE ";
   if(c & IMAGE_SCN_MEM_READ) s << "READ ";
   if(c & IMAGE_SCN_MEM_WRITE) s << "WRITE ";
}

int _tmain(int argc, _TCHAR* argv[])
{
   if(argc < 2)
      return 0;

   HANDLE h = CreateFile(argv[1], GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
   HANDLE map = CreateFileMapping(h, NULL, PAGE_READONLY, 0, 0, NULL);
   PVOID root = MapViewOfFile(map, FILE_MAP_READ, 0, 0, 0);

   IMAGE_DOS_HEADER *dos = (IMAGE_DOS_HEADER *) root;
   IMAGE_NT_HEADERS *nt = (IMAGE_NT_HEADERS *) GetPtr(dos, dos->e_lfanew);
   IMAGE_SECTION_HEADER *sec
```

```
      = (IMAGE_SECTION_HEADER *) GetPtr(nt, sizeof(IMAGE_NT_HEADERS));

   stringstream s;
   for(int i=0; i<nt->FileHeader.NumberOfSections; ++i)
   {
      printf("%s\n", sec[i].Name);
      printf("\tVirtualAddress  : %08X\n", sec[i].VirtualAddress);
      printf("\tVirtualSize     : %08X\n", sec[i].Misc.VirtualSize);
      printf("\tPointerToRawData: %08X\n", sec[i].PointerToRawData);
      printf("\tSizeOfRawData   : %08X\n", sec[i].SizeOfRawData);

      s.str(string());
      GetSectionCharacteristics(s, sec[i].Characteristics);
      printf("\tCharacteristics : %s\n", s.str().c_str());
   }

$cleanup:
   if(root) UnmapViewOfFile(root);
   if(map) CloseHandle(map);
   if(h != INVALID_HANDLE_VALUE) CloseHandle(h);

   return 0;
}
```



화면 2 scnhdr로 dummydll.dll의 섹션 정보를 확인하는 화면

RVA와 파일 오프셋

PE 파일이 메모리에 로딩되었을 때 DOS 헤더가 위치하는 곳을 베이스 주소라 한다. RVA(Relative Virtual Address)란 이 베이스 주소를 기준으로 한 상대 주소를 말한다. 따라서 메모리에 로딩된 이미지를 기준으로는 아래와 같은 공식이 적용된다.

실제 주소 = 베이스 주소 + RVA

RVA는 메모리에 로딩된 다음에는 쉽게 사용할 수 있지만, 로딩을 하기 전의 파일 단계에서 사용할 때에는 불편하다. 왜냐하면 RVA와 파일 오프셋이 정확하게 일치하지 않기 때문이다. 그 이유는 PE 파일에 존재하는 각 섹션이 파일일 때처럼 선형적으로 그대로 맵핑되지 않고, 각자 자신의 고유 위치에 맵핑 되기 때문이다.

일반적으로 PE 파일을 분석할 때에는 파일을 메모리에 로딩하지 않고, 파일 그대로인 상태에서 분석을 한다. 그래서 파일에 기록된 내용을 정확하게 읽기 위해서는 RVA를 파일 오프셋으로 변환한 다음 파일에서 읽어야 한다.

<리스트 6>에는 포인터 변환을 위한 두 가지 유용한 함수가 나와 있다. GetPtr은 base 주소에서 offset 만큼 떨어진 포인터의 주소를 반환하는 역할을 한다. RVAToOffset 함수는 rva를 파일 오프셋으로 변환하는 기능을 한다.

리스트 6 RVA를 오프셋으로 변환하는 코드

```
inline PVOID GetPtr(PVOID base, DWORD_PTR offset)
{
    return (PVOID) (((DWORD_PTR) base) + offset);
}

inline DWORD_PTR RVAToOffset(PVOID root, DWORD_PTR rva)
{
    IMAGE_DOS_HEADER *dos = (IMAGE_DOS_HEADER *) root;
    IMAGE_NT_HEADERS *nt = (IMAGE_NT_HEADERS *) GetPtr(dos, dos->e_lfanew);
    IMAGE_SECTION_HEADER *sec
        = (IMAGE_SECTION_HEADER *) GetPtr(nt, sizeof(IMAGE_NT_HEADERS));

    for(int i=0; i<nt->FileHeader.NumberOfSections; ++i)
    {
        if(rva >= sec[i].VirtualAddress
            && rva < sec[i].VirtualAddress + sec[i].Misc.VirtualSize)
        {
            return sec[i].PointerToRawData + rva - sec[i].VirtualAddress;
        }
    }

    return 0;
}
```

RVA를 파일 오프셋으로 변환 시키는 원리는 간단하다. rva나 파일 오프셋이나 섹션의 시작 위치에서부터의 오프셋은 동일하다는 점을 이용하는 것이다. rva가 A라는 섹션의 맵핑 범위에 포함되어 있다면 rva - A.VirtualSize는 섹션의 시작 위치에서의 오프셋이 된다. 여기에 섹션이 기록되어 있는 파일 오프셋인 A.PointerToRawData를 더해주면 해당 rva에 대한 파일 오프셋이 된다.

RVAToOffset의 핵심은 모든 섹션을 순회하면서 rva가 어떤 섹션에 포함되어 있는지 판단하는 부분이다.

익스포트 정보

익스포트 정보는 OptionalHeader의 DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]에 들어있다. 이곳의 VirtualAddress는 익스포트 정보가 있는 곳의 RVA를 Size는 해당 정보의 크기를 나타낸다. RVA를 따라가면 나오는 것은 익스포트 정보를 가지고 있는 IMAGE_EXPORT_DIRECTORY란 구조체다. Name은 모듈 이름을, Base는 오디날의 시작 번호를 저장하고 있다. NumberOfFunctions는 모듈이 익스포트하고 있는 함수의 개수를, NumberOfNames는 이름을 통해서 익스포트하고 있는 함수의 개수를 저장하고 있다. AddressOfFunctions, AddressOfNames, AddressOfNameOrdinals는 각각 함수 주소, 함수 이름, 함수 오디날을 저장하고 있는 배열을 가리키는 RVA다.

```
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;
```

백문이 불여일견이란 말처럼 백번 이야기를 듣는 것보다 실제 예를 하나 보는 것이 훨씬 이해하기가 쉽다. <리스트 7>에는 dummydll의 def 파일이 <그림 2>는 dummydll의 익스포트 정보가 그림으로 표시되어 있다. <화면 3>은 dummydll의 익스포트 정보를 출력한 것이다.

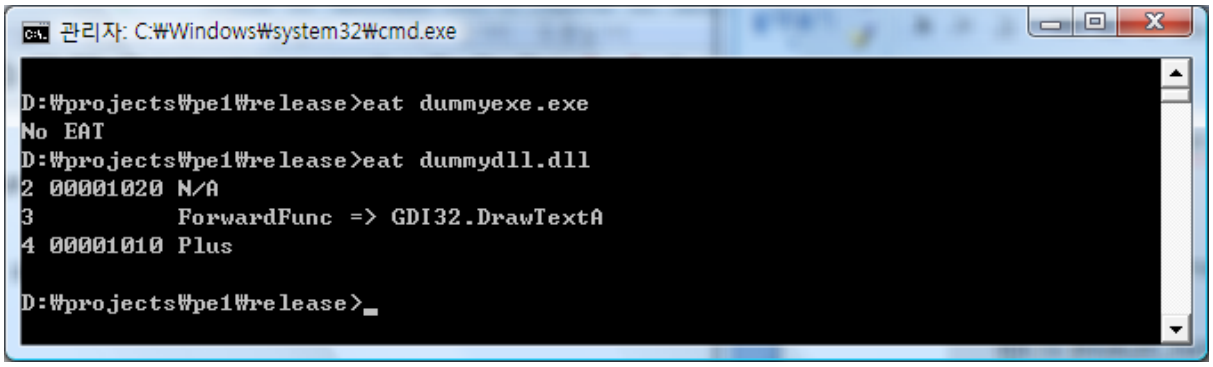
앞서 설명한대로 AddressOfFunctions, AddressOfNameOrdinals, AddressOfNames 모두 각각 배열의 RVA다. 각 배열을 따라가보면 AddressOfFunctions와 AddressOfNames는 DWORD이고, AddressOfNameOrdinals는 WORD 형태의 배열이다.

dummydll에는 총 세 개의 익스포트 함수가 있다. 두 개는 이름을 통해서 익스포트 되었고, PrintMsg는 오디날을 통해서 익스포트 되었다. <그림 2>를 보자. Functions는 예상대로 총 세 개가 있다. 이름을 통해 익스포트된 함수는 두 개 이기 때문에 Names와 NameOrdinals또한 두 개가 된다. Names에 저장된 DWORD값은 RVA로 함수 이름이 저장된 곳을 가리키고 있다. NameOrdinals는 이름을 통해 익스포트된 함수의 오디날을 저장하고 있다. 1, 2이기 때문에 Functions의 두 번째, 세 번째 함수에 대응한다는 것을 알 수 있다. 실제 오디날 값은 이 값에 IMAGE_EXPORT_DIRECTORY 구조체의 Base 값을 더한 것이 된다. 끝으로 Functions 배열에 들어있는 값은 함수 주소의 RVA 값이다. 포워딩된 경우는 해당 정보 문자열을 가리키는 RVA다. 포워딩 됐는지 여부는 RVA 값의 범위가 DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]의 범위 안에 있는지 검사해서 알아낼 수 있다. 해당 범위에 포함된다면 포워딩된 것이고, 그렇지 않다면 정상적으로 함수 주소를 가리키는 RVA로 생각하면 된다. <리스트 8>에는 익스포트 정보를 보여주

는 프로그램 소스가 나와있다. 소스를 그림과 비교해 가면서 살펴보자. AddressOfFunctions 배열의 값이 0인 경우는 함수가 없는 경우다.

리스트 7 dummydll.def

```
LIBRARY "dummydll"
EXPORTS
Plus
PrintMsg @2 NONAME
ForwardFunc=GDI32.DrawTextA
```



화면 3 dummydll과 dummyexe 파일의 익스포트 정보를 출력한 화면

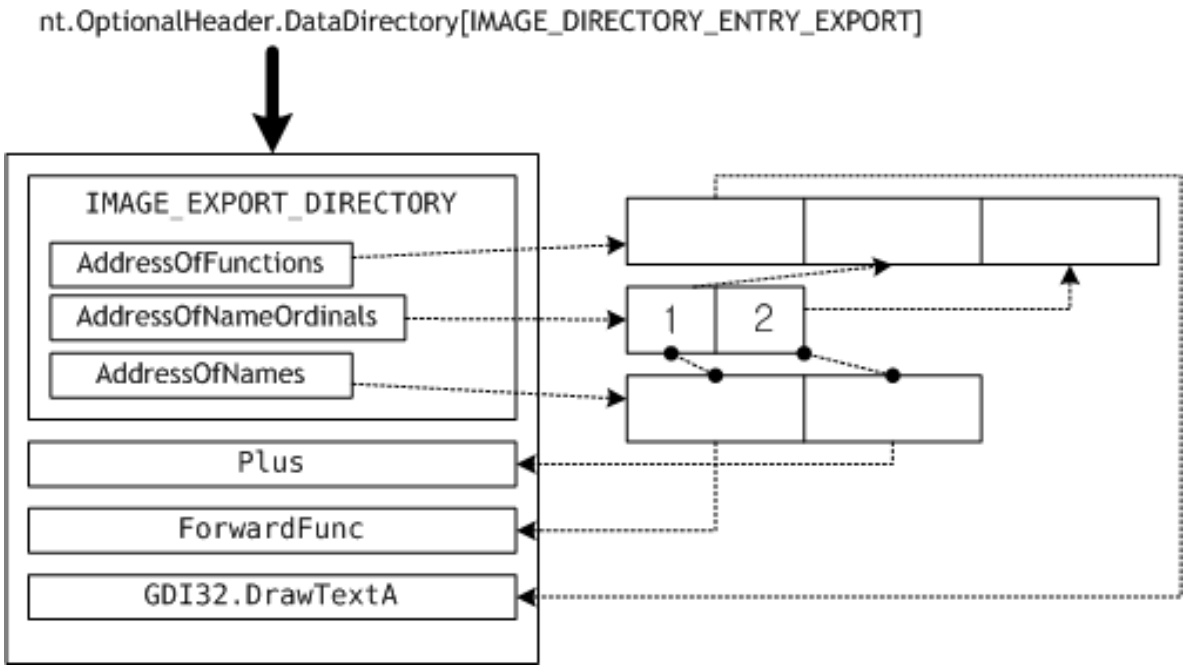


그림 2 dummydll.dll의 익스포트 정보

리스트 8 익스포트 정보를 출력하는 프로그램 소스 코드

```
int _tmain(int argc, _TCHAR* argv[])
{
   // … 중략 …
   IMAGE_DOS_HEADER *dos = (IMAGE_DOS_HEADER *) root;
   IMAGE_NT_HEADERS *nt = (IMAGE_NT_HEADERS *) GetPtr(dos, dos->e_lfanew);
```

```
    DWORD start
        =                                                                   nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
    DWORD end
        =                       start                       +               nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size;

    // 익스포트 정보가 있는지 확인한다.
    if(start == 0)
    {
        printf("No EAT");
        goto $cleanup;
    }

    IMAGE_EXPORT_DIRECTORY *ed
        = (IMAGE_EXPORT_DIRECTORY *) GetPtr(dos, RVAToOffset(root, start));

    // 익스포트 정보를 구한다.
    DWORD    *funcs    =    (DWORD    *)    GetPtr(root,    RVAToOffset(root,    ed-
>AddressOfFunctions));
    DWORD *names = (DWORD *) GetPtr(root, RVAToOffset(root, ed->AddressOfNames));
    WORD    *ordinals    =    (WORD    *)    GetPtr(root,    RVAToOffset(root,    ed-
>AddressOfNameOrdinals));

    char *noname = "N/A", *name, *forward;
    for(DWORD i=0; i<ed->NumberOfFunctions; ++i)
    {
        if(funcs[i] == 0)
            continue;

        // 현재 오디날에 해당하는 함수 이름이 있는지 찾는다.
        name = noname;
        for(DWORD j=0; j<ed->NumberOfNames; ++j)
        {
            if(ordinals[j] == i)
            {
                name = (char *) GetPtr(root, RVAToOffset(root, names[j]));
                break;
            }
        }

        printf("%d ", i + ed->Base);

        // 포워딩된 함수인지 확인한다.
        forward = "";
        if(funcs[i] >= start && funcs[i] < end)
        {
            forward = (char *) GetPtr(root, RVAToOffset(root, funcs[i]));
            printf("%*s %s => %s\n", 8, " ", name, forward);
        }
        else
        {
            printf("%08X %s\n", funcs[i], name);
        }
    }

    // … 중략 …
}
```

임포트 정보

OptionalHeader의 DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT]에는 프로그램에서 임포트해

서 사용하고 있는 DLL들과 함수에 대한 정보가 들어있다. DataDirectory의 VirtualAddress가 가리키는 곳에는 IMAGE_IMPORT_DESCRIPTOR 구조체가 저장되어 있다.

IMAGE_IMPORT_DESCRIPTOR 구조체의 주요 필드로는 Name과 FirstThunk가 있다. Name은 사용하고 있는 DLL의 이름이, FirstThunk는 해당 DLL에서 불러오는 첫 번째 함수에 대한 IMAGE_THUNK_DATA 구조체의 RVA가 저장되어 있다.

IMAGE_THUNK_DATA는 임포트한 함수에 대한 정보를 저장하고 있다. 동일한 데이터 형으로 된 공용체로 구성된 이유는 상황에 맞는 필드명을 쓰기 위해서다. 일반적으로 로딩되기 전에는 AddressOfData나 Ordinal을 저장하기 위한 용도로 사용된다. AddressOfData는 임포트한 함수를 이름으로 연결한 경우다. AddressOfData에 저장된 값은 IMAGE_IMPORT_BY_NAME 구조체가 있는 위치에 대한 RVA다. 함수를 오디날로 연결한 경우는 Ordinal 필드가 사용된다. 둘 중의 어떤 것을 사용하는지를 나타내기 위해서 각 값의 최상위 비트를 사용한다. 최상위 비트가 1인 경우는 하위 31비트를 오디날로 사용한다. 0인 경우는 그 값을 AddressOfData로 사용한다. 로더가 이미지를 로딩하면 이 값을 실제 함수 주소로 덮어 쓴다. 그 때는 Function의 의미로 값이 사용되는 것이다. 여러 가지 의미로 사용됨을 나타내기 위해서 공용체로 묶어 두었지만 동일한 DWORD 값이기 때문에 어떤 필드로 값을 접근하든 결과는 동일하다.

리스트 9 임포트 정보를 위한 구조체

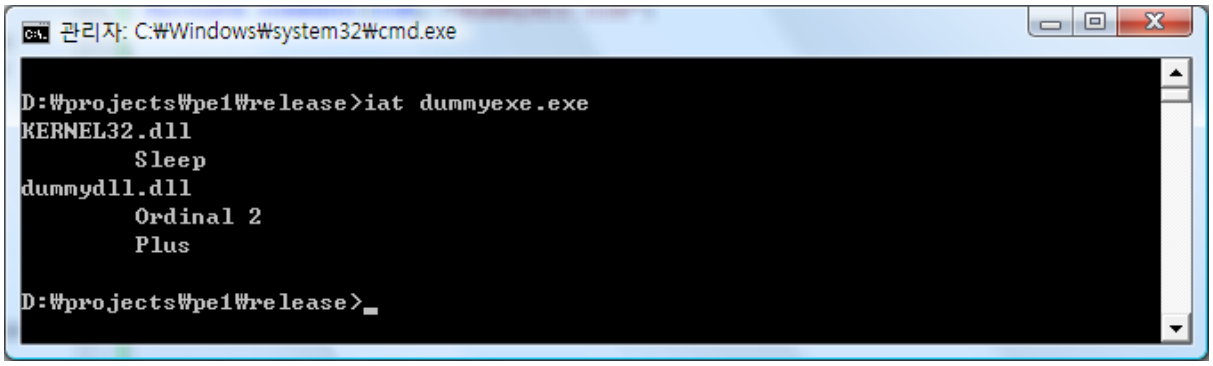
```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
   union {
      DWORD   Characteristics;
      DWORD   OriginalFirstThunk;
   };
   DWORD   TimeDateStamp;

   DWORD   ForwarderChain;
   DWORD   Name;
   DWORD   FirstThunk;
} IMAGE_IMPORT_DESCRIPTOR;

typedef struct _IMAGE_THUNK_DATA32 {
   union {
      DWORD ForwarderString;
      DWORD Function;
      DWORD Ordinal;
      DWORD AddressOfData;
   } u1;
} IMAGE_THUNK_DATA32;

typedef struct _IMAGE_IMPORT_BY_NAME {
   WORD    Hint;
   BYTE    Name[1];
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

한 가지 더 추가로 알아두어야 할 필드가 있다. 바로 OriginalFirstThunk 필드다. 이 필드는 FirstThunk와 동일한 역할을 한다. IMAGE_THUNK_DATA를 가리키는 RVA 값이다. 똑 같은 역할을 하는 필드가 왜 두 개나 있을까? 이유는 간단하다. OriginalFirstThunk는 백업 용이다. 앞서 우리는 이미지가 메모리에 로딩되면 IMAGE_THUNK_DATA의 값이 실제 함수 주소로 덮어 써 진다고 했다. 이렇게 되면 원래 IMAGE_THUNK_DATA를 읽을 수 없게 된다. 이 경우에 OriginalFirstThunk를 따

라가서 데이터를 읽으면 기존의 값을 그대로 읽을 수 있다. PEView로 살펴보면 알겠지만 동일한 테이블이 PE 포맷 내에 두 개가 존재한다. IAT(Import Address Table)과 ILT(Import Lookup Table)이 그것이다. 파일로 존재할 때에는 둘의 내용이 동일하지만 로딩되고 나면 IAT는 실제 주소로 채워지고, ILT는 원래 값을 그대로 가지게 된다.

리스트 10 임포트 정보를 출력하는 프로그램

```
#include <stdio.h>
#include <tchar.h>
#include <windows.h>

int _tmain(int argc, _TCHAR* argv[])
{
   // … 중략 …
   IMAGE_DOS_HEADER *dos = (IMAGE_DOS_HEADER *) root;
   IMAGE_NT_HEADERS *nt = (IMAGE_NT_HEADERS *) GetPtr(dos, dos->e_lfanew);

   // 임포트 정보가 있는지 확인한다.
   DWORD_PTR idescRVA
      =                                                                    nt-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress;
   if(idescRVA == 0)
   {
      printf("No IAT");
      goto $cleanup;
   }

   IMAGE_IMPORT_DESCRIPTOR *idesc
      = (IMAGE_IMPORT_DESCRIPTOR *) GetPtr(dos, RVAToOffset(root, idescRVA));

   IMAGE_THUNK_DATA *itd;

   // 임포트한 DLL 이름을 출력한다.
   while(idesc->Name)
   {
      printf("%s\n", GetPtr(root, RVAToOffset(root, idesc->Name)));

      // 각 DLL에서 임포트한 함수 명을 출력한다.
      itd   =   (IMAGE_THUNK_DATA   *)   GetPtr(root,   RVAToOffset(root,   idesc-
>OriginalFirstThunk));
      while(itd->u1.AddressOfData)
      {
         if(itd->u1.Ordinal & 0x80000000)
         {
            printf("\tOrdinal %u\n", itd->u1.Ordinal & 0x7fffffff);
         }
         else
         {
            DWORD_PTR rva = RVAToOffset(root, itd->u1.AddressOfData);
            PIMAGE_IMPORT_BY_NAME ibn = (PIMAGE_IMPORT_BY_NAME) GetPtr(root, rva);
            printf("\t%s\n", ibn->Name);
         }
         ++itd;
      }

      ++idesc;
   }
   // … 중략 …
}
```



화면 4 dummyexe의 임포트 정보를 출력한 화면

도전 과제

개발에 갓 입문한 새내기 개발자가 자신의 결과물을 자랑하기 위해서 친구에게 파일을 보내고 처음 겪게 되는 난관은 필요한 DLL이 없다는 에러 메시지다. 답답한 마음에 게시판을 찾아가서 물어도 보고, dependency walker란 유틸리티를 다운 받아서 사용해 보기도 한다. 하지만 여전히 어렵다. dependency walker에 나오는 것들 중 무엇을 같이 보내야 하는지도 결정하기가 힘든 것이다. 이러한 개발자를 도와줄 수 있는 간단한 유틸리티를 만들어 보자.

실행 파일을 입력하면 임포트 테이블을 분석해서 필요한 DLL들을 추출해 낸다. 그 중에서 시스템에 기본적으로 설치되지 않는 녀석들만 표시해 주도록 한다. 같이 한 파일로 압축을 해주거나 부가 DLL들을 같은 폴더로 복사해 준다면 사용하기가 더욱 편리할 것이다.

참고자료

"Windows 시스템 실행 파일의 구조와 원리"
이호동저, 한빛미디어

An In-Depth Look into the Win32 Portable Executable File Format
http://msdn.microsoft.com/msdnmag/issues/02/02/PE/

An In-Depth Look into the Win32 Portable Executable File Format, Part 2
http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/

Peering Inside the PE: A Tour of the Win32 Portable Executable File Format
http://msdn2.microsoft.com/en-us/library/ms809762.aspx