Windows Hook

Jerald Lee
Contact Me : lucid7@paran.com

본 문서는 저자가 Windows Hook을 공부하면서 알게 된 것들을 정리할 목적으로 작성되었습니다. 본인이 Windows System에 대해 아는 것의 거의 없기 때문에 기존에 존재하는 문서들을 짜집기 한 형태의 문서로밖에 만들 수가 없었습니다. 문서를 만들면서 참고한 책, 관련 문서 등이 너무 많아 일일이 다 기술하지 못한 점에 대해 원문 저자들에게 매우 죄송스럽게 생각합니다.
본 문서의 대상은 운영체제와 Win32 API를 어느 정도 알고 있다는 가정하에 쓰여졌습니다. 본 문서에 기술된 일부 기법에 대한 자세한 설명은 책을 참조하시길 바랍니다.

제시된 코드들은 Windows XP Service Pack2, Visual Studio .net 2003에서 테스트 되었습니다. 문서의 내용 중 틀린 곳이나 수정해야 할 부분이 있으면 연락해 주시기 바랍니다.

목 차

1. WINDOWS HOOK 개요 3

1.1. WINDOWS HOOK 3

1.2. WIDOWS 메모리 관리 4

1.3. PE FORMAT 9

1.3.1. RVA(Relative Virtual Address) 10

1.3.2. Section 10

1.3.3. PE Format의 구조 11

1.3.4. PE Format 분석을 위한 도구들 15

1.4. DLL 17

1.4.1. Introduce Dll 17

1.4.2. Dll의 종류 18

1.4.3. 명시적 로딩과 암시적 로딩 18

1.4.4. 익스포트 시 주의할 점 20

1.4.5. 참고문서 23

1. Windows Hook 개요

1.1. Windows Hook

훅(Hook)이라는 단어는 “갈고리”, “낚싯바늘” 또는 “~을 낚싯바늘로 낚다” 라는 사전적 의미를 가지고 있다. 그러므로 Windows Hook이라고 하면 “Windows 라는 운영체제 안에서 무언가를 낚아챈다” 라는 뜻으로 이해할 수 있을 것이다.

MSDN은 Hook을 다음과 같이 정의해 놓고 있다.

> A hook is a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure.

훅이란 메시지 핸들링 매커니즘을 가지고 있는 시스템에 메시지가 타켓 윈도우 프로시저에 도착하기 전에 이들을 모니터링 하기 위한 어플리케이션을 서브 루틴으로써 설치할 수 있는 시점이다. (의역이라 상당히 다른 뜻일 수도 있으니 각자 해석하시기 바랍니다. -__-;)

이러한 Hook을 하기 위한 방법은 여러 가지가 있는데 일반적으로 Win32 API에서 제공되는 SetWindowsHookEx(), UnhookWindowsEx() 함수를 이용한 메시지 후킹을 많이 사용한다.(전역 훅에 속하고 시스템의 성능이 저하된다는 단점이 있지만 많이 애용되는 방법 중의 하나다.) 국내 보안업계에서는 2005년 인터넷 뱅킹 서비스와 관련하여 비슷한 작동을 하는 초보적인 형태의 프로그램을 이용해 “해킹이 가능하다. 고객의 아이디/패스워드를 빼낼 수 있다” 등의 과장된 형태로써 보도가 되어 한 때 사회적 이슈가 된 적이 있었다.
(Windows System Programming을 해 본 사람이면 누구나 다 가능한 일이다. 언론 기사의 쓸데 없는 과잉 보도로 인해 업무량이 급증하면서 신경이 많이 날카로워 진 적이 있다. -__-)

Code Project 의 Three Ways to Inject Your Code into Another Process 문서에서는 Windows Hook을 다음과 같이 세 가지의 형태로 구분해 놓았다.

- Local hooks
  자신의 프로세스에 구성된 스레드들의 메시지 트래픽을 감시
- Remote hooks
  다른 프로세스에 속해진 스레드들의 메시지 트래픽을 감시
- Remote hooks
  현재 시스템에서 동작하는 모든 스레드들의 메시지 트래픽을 감시

Local hooks의 경우를 제외하고는 Windows Hook을 하기 위해서는 Hook을 수행하는 코드가 Dll로 작성되어 타겟 프로세스에 로드되는 형태라야만 가능하다. 이를 Dll Injection이라고 하며 가장 대중적으로 이루어지는 형태의 훅이 아닐까 한다.(IAT를 이용한 훅도 있으며 제프리 아저씨의 책에서 소개된 바 있다. 그 외에도 다양한 방법의 훅이 있으며 본 문서에서 다루어 볼 것이다.)

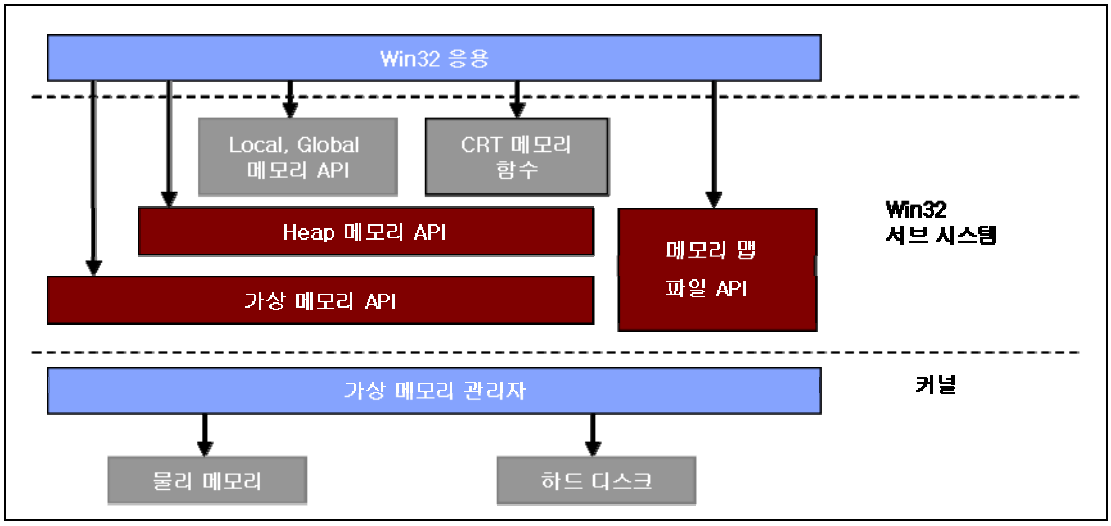
이 Dll Injection을 하는 방법 역시 많은 문서에 여러 가지 형태로 발표 되었으며, 일단 Dll이 성공적으로 로드되면 그 이후에 할 수 있는 작업들은 매우 다양한 형태가 될 것이다.
일반적으로 API Hooking, Message Hooking 등의 제목으로 많은 문서가 존재하고 있으며 구글님께 물어보면 풍부한 양의 관련 문서들을 찾을 수 있다.

1.2. Widows 메모리 관리

본격적인 Dll Injection에 들어가기 전에 알아야 할 많은 주제들 중 먼저 Windows 메모리 관리에 대해 간단하게 정리해 본다.
Windows는 메모리 관리를 위해 크게 다음과 같은 관리 기법을 사용하고 있다.

- 가상 메모리(Virtual Memory)
- 메모리 맵(Memory-mapped)
- 힙(Heap)

이 중에서 가상 메모리에 대해서만 알아본다.



[Win32의 메모리 관리 구조]

프로그램이 실행되기 위해서는 먼저 메모리에 적재 되어야만 한다. 즉, 큰 프로그램을 실행시

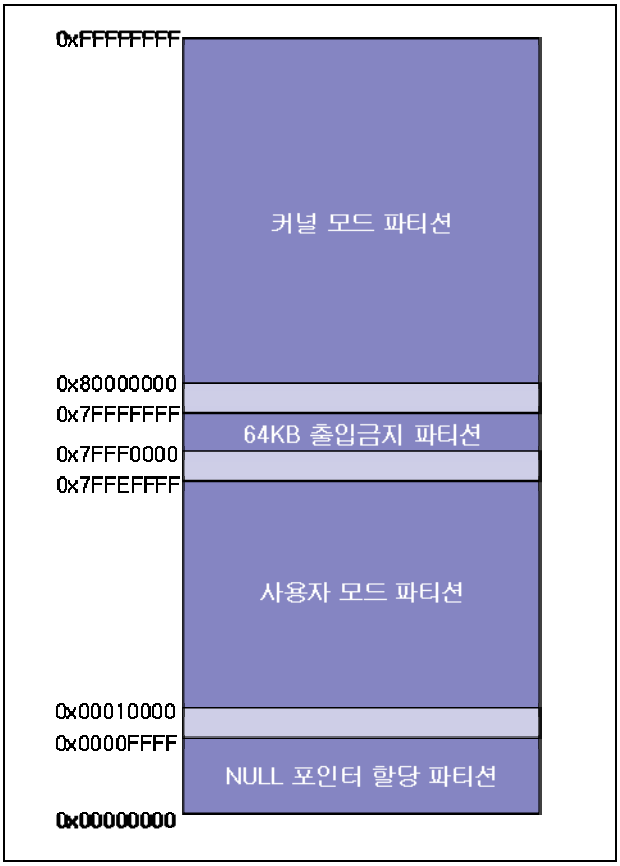
키기 위해서는 그만큼의 메모리가 더 필요하다. 하지만 주 기억장치의 용량에도 한계가 있기 때문에 보조 기억장치를 이용한 가상 메모리 기법이 등장하게 된다.
즉, 가상 메모리란 하드 디스크와 같은 스토리지 시스템을 병합하여 시스템에 한 개 이상의 RAM이 설치된 것처럼 운영체제를 속이는 것이라고 할 수 있으며 32비트 컴퓨터의 경우 최대 인식 가능한 가상의 주소 공간의 크기는 4GB(2^{32})이다.
운영체제가 하드 디스크와 같은 스토리지 시스템을 가상 메모리로서 활용하기 위해 쓰는 기법이 Swapping이며 Unix 및 Windows 모두 사용되고 있다.
Swapping에 대한 자세한 내용은 운영체제 책(공룡책이라 불리는 책이 가장 쉽다)을 참고하기 바란다.

일반적으로 하나의 프로그램이 실행되기 위해서 프로그램 전체가 적재되어야 할 필요는 없으며 그 대상은 아래와 같다.

- 쓰이지 않는 코드(예외 처리와 같은)
- 행렬, 리스트, 테이블 등과 같은 자료구조
- 같은 시간에 동시에 쓰이지 않은 구조

위와 같은 것들은 실행 중인 프로그램에서 사용되지 않을 때 메모리에 적재되어 있을 필요가 없으므로 Swapping을 통해 하드 디스크에 저장되어 있다가 프로그램이 요구할 때 다시 메모리로 적재하게 된다. 이와 같은 가상 메모리를 구현하기 위해 Windows는 Demand Paging 기법을 사용한다.

아래는 가상 메모리의 구조이다.(Windows 2000 기준)



[Windows 2000 가상 메모리 구조]

▪ Null 포인터 할당 파티션

이 파티션은 프로그래머가 널 포인터 할당을 쉽게 알아차리도록 도와주는 파티션이다. 만일 프로세스의 어떤 스레드가 이 파티션의 메모리를 읽거나 쓰려고 한다면 CPU는 접근 위반을 발생시키게 된다.

▪ 사용자 모드 파티션

프로세스의 독립된 주소 공간이 위치하는 곳이며 프로세스는 다른 프로세스의 이 파티션에 위치하는 데이터를 읽거나 쓸 수 없다. 2000 계열에서는 모든 exe, Dll, 메모리 맵 파일들이 이 파티션에 로드된다.

▪ 64KB 출입금지 파티션

2000 계열에만 존재하는 이 파티션은 어떤 파티션에 대한 접근도 Access Violation을 발생하도록 한다. 이는 Microsoft에서 Windows를 구현할 때 커널 모드의 메모리 파티션을 쉽게 보호하기 위해 만들어 놓은 일종의 완충지대라고 할 수 있는 파티션이다.

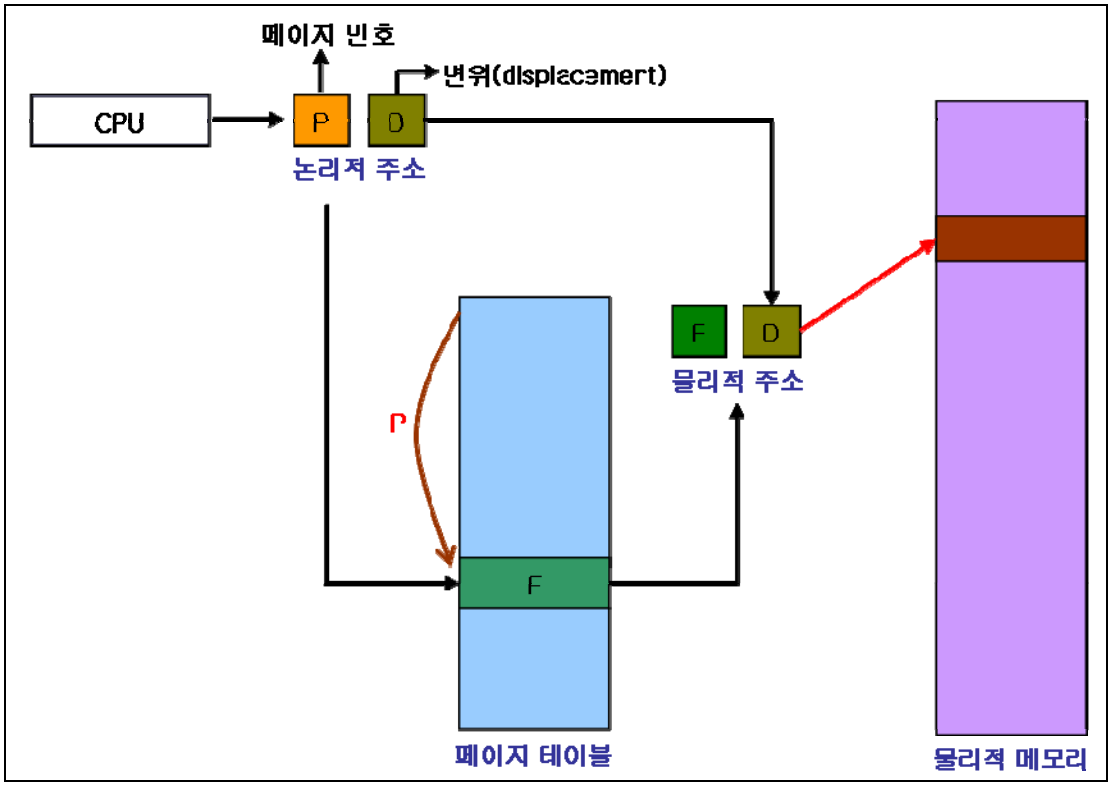
▪ 커널 모드 파티션

이 파티션은 운영체제 자체의 코드가 존재하는 곳이다. 스레드, 메모리, 파일 시스템, 네

트워크, 모든 디바이스 드라이버 등이 여기에 로드된다. 2000 계열에서 이 파티션은 완전히 보호되어 있다.

페이징이란 프로그램 중 자주 사용되지 않는 부분의 작업 메모리를 주기억장치인 메모리로부터 보조기억장치인 하드디스크로 옮기는 방식을 통해, 활용 가능한 메모리 공간을 증가시키기 위한 기법 중의 하나이며 한번에 옮겨지는 메모리 용량 단위를 페이지라고 한다.
페이지의 크기는 프로세서의 구조에 따라 틀리지만 펜티엄 이후 x86 프로세서는 4KB로 고정되어 있다.

페이지 테이블을 이용한 주소 매핑은 아래와 같다.



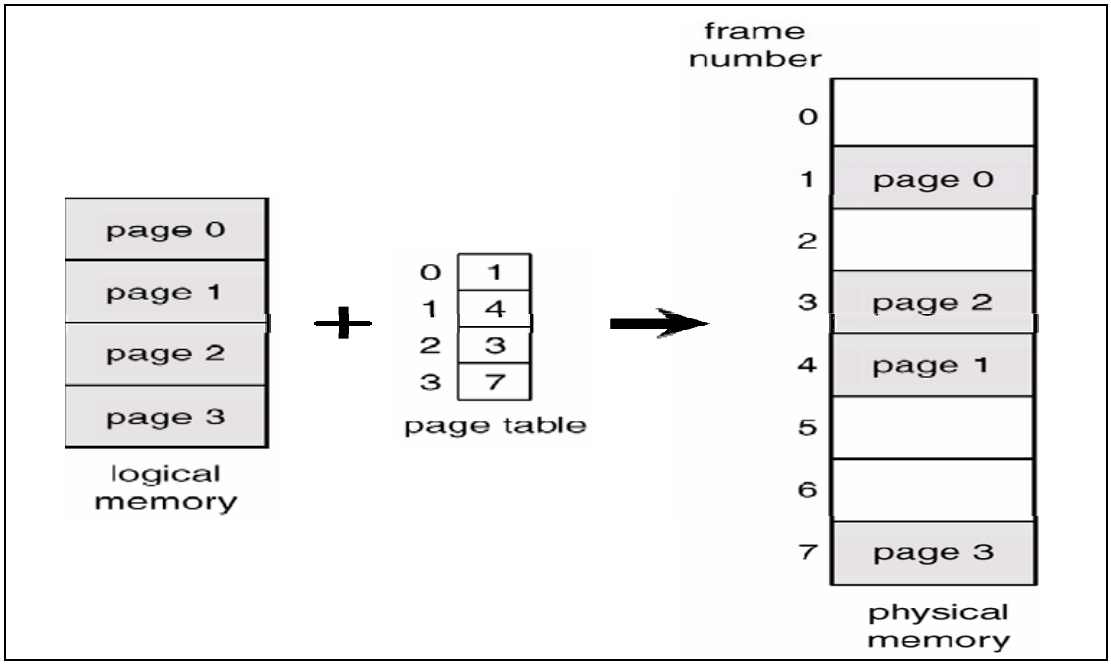
[페이지 테이블을 이용한 주소 매핑 과정]

※ x86의 경우 2단계 페이지 테이블을 사용하며 각 페이지의 크기는 10바이트이다.

메모리 관리장치는 메모리 접근 요청들을 감시하고, 또한 각 주소를 페이지 번호와 페이지 내의 오프셋(그림에서는 변위로 표시되어 있음)으로 분할한다. 메모리 관리장치는 페이지 표에 있는 페이지 번호를 확인하는데, 페이지 표에는 해당 페이지가 메모리에서 디스크로 옮겨졌는지 또는 메모리에 아직 존재하는지에 관한 정보들이 표시되어 있다. 만약 메모리에 존재하는 페이지에 대해 접근하려는 시도가 있는 경우라면, 가상의 주소가 실제 주소로 번역된 후 접근

이 즉시 허용된다. 그러나 요청된 페이지가 이미 디스크로 옮겨진 상태라면, 먼저 메모리에 현존하는 다른 페이지를 디스크로 옮겨서 빈 공간을 만들어야만 한다. 그다음 디스크의 스왑 공간 상에 있던 해당 페이지가 주기억장치인 메모리로 읽혀 들여진다. 이때 그 페이지가 이제는 주기억장치 내에 존재한다는 표시와 함께 메모리 상의 물리적인 주소 등 페이지 표의 정보도 함께 갱신된다.
메모리 관리장치는 주기억장치로 불려 들어온 이후에 내용이 수정된 페이지가 어떤 페이지인지에 관한 정보 기록도 유지한다. 만약 수정이 된 적이 없고 디스크로 복사할 필요가 없는 페이지는 즉시 재사용될 수 있다.

아래의 그림을 보면 페이징 테이블을 이용한 주소의 매핑이 좀 더 명확하게 이해될 것이다.



[페이지 테이블을 이용한 주소 매핑 예]

어떤 새로운 프로세스가 생성되고 주소 공간이 생성될 때 모든 사용 가능한 주소 공간은 해제된 상태로 존재하며 이 주소 공간을 사용하기 위해 Win32 API VritualAlloc() 함수로 필요한 공간을 예약(reserve)하게 된다. 이후 확보된 영역을 사용하기 위해 실제 메모리에 매핑하는 작업이 필요하며 이를 커밋(commit)이라고 한다. 이후 매핑이 끝난 커밋된 페이지는 접근 가능한 상태가 된다.

Windows는 페이징을 이용해 모든 프로세스마다 물리적인 주소가 겹치지 않도록 독립적인 주소 공간을 제공하며 각 프로세스는 서로의 메모리 영역에 접근할 수 없다.
그렇기 때문에 DOS나 16비트 Windows에서 있었던, 하나의 프로그램이 Crash 되었을 때 다른 프로그램도 영향을 받는 일이 없어지게 되었다.(블루스크린)

본 문서에서 말하고자 하는 Dll Injection은 이러한 독립된 프로세스의 주소 공간으로 접근하여 해당 프로세스를 조절하는 것이 목적이며 좁은 의미의 Windows Hook이라고 할 수 있을 것이다.

1.3. PE Format

IAT Table을 이용한 Hook을 이해하기 위해 반드시 알아야만 하는 것이 바로 이 PE Format이다. 사실 IAT Table 변조는 어떻게 보면 Reverse Engineering 분야에 속하기 때문에 다루지 않으려고 했지만 Windows Hook에 대해 소개된 대부분의 방법을 정리해 보자는 취지로 포함시키게 되었다. PE 파일 포맷에 관한 내용은 매우 방대하고 잘 정리된 양질의 문서 또한 매우 많기 때문에 본문에서는 모든 것을 세세하게 설명하지 않고 필요하다고 생각되는 것만 살펴보도록 한다. (어디에 무엇이 있더라~ 정도는 알아두어야만 한다.)

PE는 Portable Executable의 약자로써 이식 가능한 실행 프로그램을 뜻하며 Win32 플랫폼 하에서 공통으로 사용할 수 있음을 뜻한다. 또한 PE는 Microsoft Windows에서 EXE, Dll, OCX 등과 같은 확장자를 가진 파일에 사용되는 포맷으로써 가장 기본적인 파일 형식이라고 할 수 있다. PE Format은 Unix System V 계열에서 사용되는 COFF를 변형시킨 형태로써 발전했으며 PE가 COFF의 변형인 관계로 두 파일 포맷의 구조는 매우 비슷하다.
(최초의 Unix System에서는 a.out, System V 초기버전에서는 COFF, Windows NT는 PE, IBM은 IBM 360, Linux와 Solaris는 ELF 파일 포맷을 사용한다.)

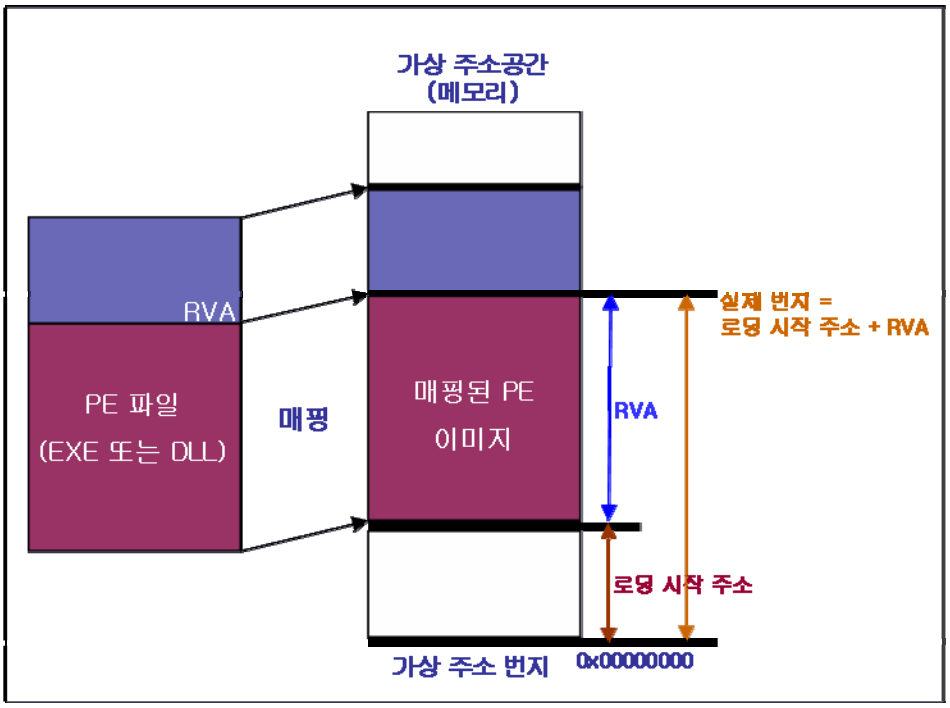
먼저 PE Format에 사용되는 용어들을 살펴보자.

이름	설명
Image File	exe, Dll과 같은 확장자를 가지는 실행 파일. 실행 파일 자체가 가상 주소 공간에 매핑되기 때문에 Image(그림자) 라는 단어를 쓰게 되었으며 이미지 파일이라고 하면 보통 exe 확장자를 가지는 실행 파일을 지칭하기도 한다.
Object File	Object File은 Linker에 입력되는 파일이다. Linker는 Object File에서 Image File을 생성하게 되고 생성된 Image File은 Loader에 의해 메모리로 적재된다.
RVA	Relative Virtual Address의 약자로서 상대적 가상 주소라고 한다. 이미지가 메모리에 로드되었을 때의 그 시작 주소에 대한 상대적 번지를 의미하며 메모리 상에서의 PE의 시작 주소에 대한 오프셋으로 생각하면 된다.
Virtual	이미지 파일의 시작 주소를 빼지 않는 것을 제외하고는 위에서 살펴본

Address(VA)	RVA와 동일하다. 이 주소는 가상 주소라고 불리는데, Windows NT는 각 프로세스를 위해 물리적인 메모리와는 독립적인 별개의 가상 주소 공간을 만들기 때문이다. 가상 주소는 단지 주소를 고려하기 위한 것이며 RVA처럼 예측 가능하지 않다.
Section	PE가 가상 주소 공간에 로드된 뒤의 실제 내용을 담고 있는 블록들이며 일반적으로 코드와 데이터라고 생각할 수 있다. 그리고 프로그램 실행에 관련된 여러 정보들이 이 섹션 안에 배치된다.

1.3.1. RVA(Relative Virtual Address)

위에서 살펴본 대로 RVA는 상대적 가상 주소를 뜻한다. 일반적으로 파일 상태로 2차 메모리인 하드디스크에 존재하는 PE는 가상 메모리의 어떤 위치로 로드될 지 미리 알 수 없지만 사실 NT의 경우 Default로 0x04000000 번지에 이미지의 시작부로부터 로드되며 이런 이미지 로딩 시작 번지의 정보는 PE 파일 내에 존재한다. 따라서 로드된 후의 가상 번지 값을 기준으로하고 그 기준에 대한 상대적 오프셋으로 PE 상에 해당 번지를 기입하는 것이며 RVA를 계산하는 법은 아래와 같다.



[Real Address 계산]

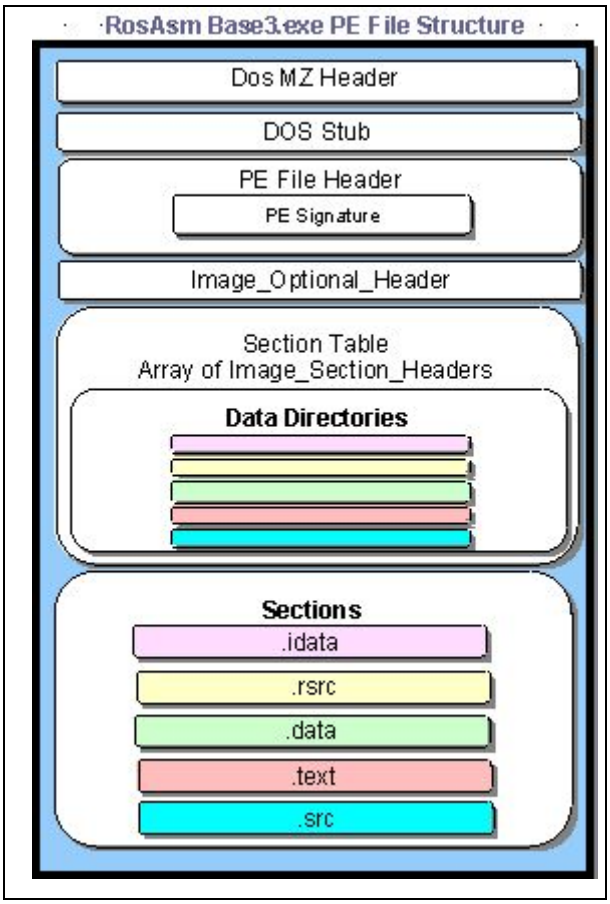
실제 주소 번지(가상 주소) = 이미지 로드 시작 번지 + RVA

1.3.2. Section

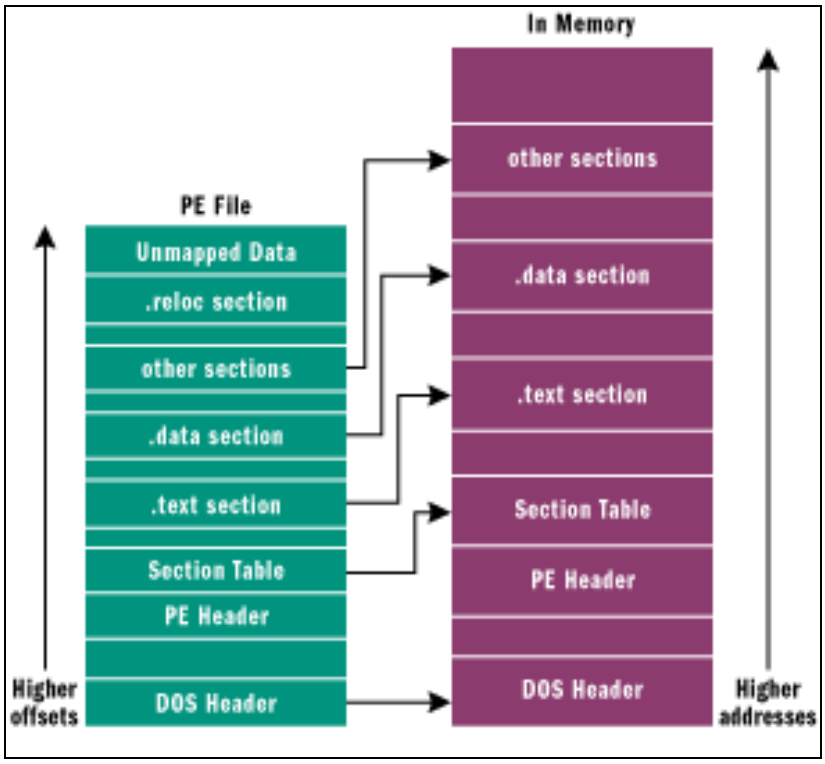
Section은 같은 역할을 하는 데이터를 정리한 블록이며 PE 파일이 메모리에 로드된 후 코드와 데이터, 임포트/익스포트 된 API들, 리소스, 재배치 정보 등과 같이 PE 파일의 실제 정보는 section이라 불리우는 블록으로 나뉘어지게 된다.
Windows에서는 Section을 발견하면 그것을 배치할 가상 주소 영역을 확보해서 Section 데이터를 통째로 복사하며 해당 Section에 대한 정보는 Section Table이나 Header에서 얻는다.

1.3.3. PE Format의 구조

PE 파일 포맷의 전체 구조는 아래 그림과 같다.



[PE 파일 포맷 구조 1]



[PE 파일 포맷 구조 2]

PE 파일 포맷을 구성하는 요소들을 조금 더 자세히 알아보도록 하자. 아래의 내용은 WinNT.h 헤더 파일에 기술되어 있다.

- IMAGE_DOS_HEADER

  "MZ" 문자열로 시작하는 처음부터 40바이트까지의 공간과(그림 1에서는 Dos MZ Header, 그림 2에서는 DOS Header) DOS Stub를 합친 공간이다.

  실행 파일을 바이너리 에디터로 열어보면 처음 시작 부분이 "MZ" 문자열로 시작하는 것을 알 수 있다.

```
00000000h: 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00 ; MZ?........    ..
00000010h: B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00 ; ?.......@.......
00000020h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ; ................
00000030h: 00 00 00 00 00 00 00 00 00 00 00 00 C8 00 00 00 ; ............?..
00000040h: 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68 ; ..?.???L?Th
00000050h: 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F ; is program canno
00000060h: 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20 ; t be run in DOS
00000070h: 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00 ; mode....$.......
00000080h: 16 F3 5B 15 52 92 35 46 52 92 35 46 52 92 35 46 ; .?.R?FR?FR?F
```

[IMAGE_DOS_HEADER]

- IMAGE_NT_HEADER

  실질적인 PE 포맷이 시작되는 곳이며 "PE₩0₩0"(0x00004550)라는 4바이트 문자의 시그

너쳐를 시작으로 IMAGE_FILE_HEADER(20byte), IMAGE_OPTIONAL_HEADER(96byte), Data Directory 배열(128byte)로 구성되어 있다. 그림에서는 단순하게 PE HEADER로 표현되어 있다.

우리가 관심을 가져야 할 부분은 바로 이 Data Directory 배열 부분이다.

Data Directory 배열은 IMAGE_DATA_DIRECTORY 구조체의 배열로 구성되어 있으며 구조는 아래와 같다.

Member	Info inside
0	Export symbols
1	Import symbols
2	Resources
3	Exception
4	Security
5	Base relocation
6	Debug
7	Copyright string
8	Unknown
9	Thread local storage (TLS)
10	Load configuration
11	Bound Import
12	Import Address Table
13	**Delay Import**
14	**COM descriptor**

[Data Directory 배열]

위의 파란색으로 표시된 부분이 보이는가? 이 지겹고도 어려운 PE Format의 설명을 들어온 목적이 바로 12번 Import Address Table 때문이다.

IAT에 관한 사항은 뒤에서 더 자세히 살펴보도록 한다.

- IMAGE_FILE_HEADER

  CPU 타입, 생성 시간과 같은 기본적인 정보를 담고 있다.

- IMAGE_OPTIONAL_HEADER

  ImageBase, AddressOfEntryPoint와 같은 정보를 담고 있다.

▪ Section Table

여기는 IMAGE_SECTION_HEADER들의 배열로 이루어져 있으며
IMAGE_SECTION_HEADER에는 .text, .data, .idata, .reloc 등의 시작 주소, 사이즈와 같은 정보를 포함하고 있다.

▪ Section

PE 포맷이 가상 주소 공간에 로드된 뒤의 실제 데이터를 담고 있는 영역이며 아래의 표와 같은 것을 포함하고 있다.

종류	이름	설명
코드	.text	프로그램을 실행하기 위한 코드를 담고 있는 섹션. CPU 레지스터의 명령 포인터인 ip는 이 섹션 내에 존재하는 번지 값을 담게 된다.
데이터	.data	초기화 된 전역 변수가 저장된다.
	.rdata	읽기 전용 데이터 섹션으로서 문자열 표현이나 c++/com 가상 함수 테이블 등이 .rdata에 배치되는 항목 중의 하나이다.
	.bss	초기화 되지 않은 전역 변수가 저장된다. VC++ 7에서는 .textbss로 나타난다. 실제 PE 파일 내에서는 존재하지만 가상 주소 공간에 매핑될 때에는 보통 .data 섹션에 병합되기 때문에 메모리 상에서는 따로 존재하지 않는다.
임포트 API 정보	.idata	임포트 할 Dll과 그 API에 대한 정보를 담고 있다. IAT가 여기에 존재한다.
	.didat	지연 로딩 임포트 데이터를 위한 섹션이다.
익스포트 API 정보	.edata	익스포트 할 API에 대한 정보를 담고 있다. 이 섹션 역시 .rdata나 .text 섹션에 병합되기 때문에 메모리 상에서는 따로 존재하지 않는다.
리소스	.rsrc	다이얼로그, 아이콘과 같은 리소스 관련 데이터들이 포함된다.
재배치 정보	.reloc	실행 파일에 대한 기본 재배치 정보를 담고 있다.
TLS	.tls	__declspec(thread) 지시어와 함께 선언되는 스레드 지역 저장소를 위한 섹션이다.
C++ 런타임	.crt	생략
Short	.sdata	
	.srdata	
예외 정보	.pdata	
디버깅	.debug$S	
	.debug$T	

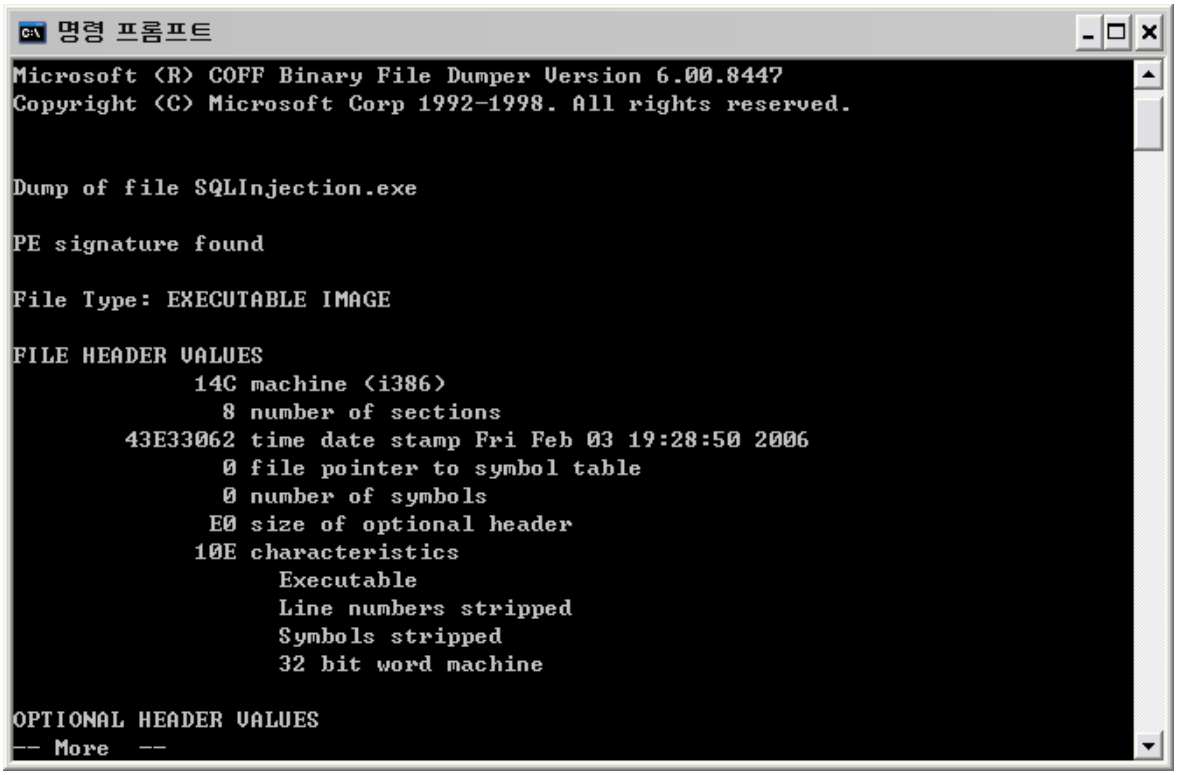
	.debug$P	
Directives	.drectve	

1.3.4. PE Format 분석을 위한 도구들

지금까지 살펴 본 PE 파일 포맷의 구조를 분석하기 위한 유용한 도구들이 많이 있으며 아래와 같다.

- Dumpbin

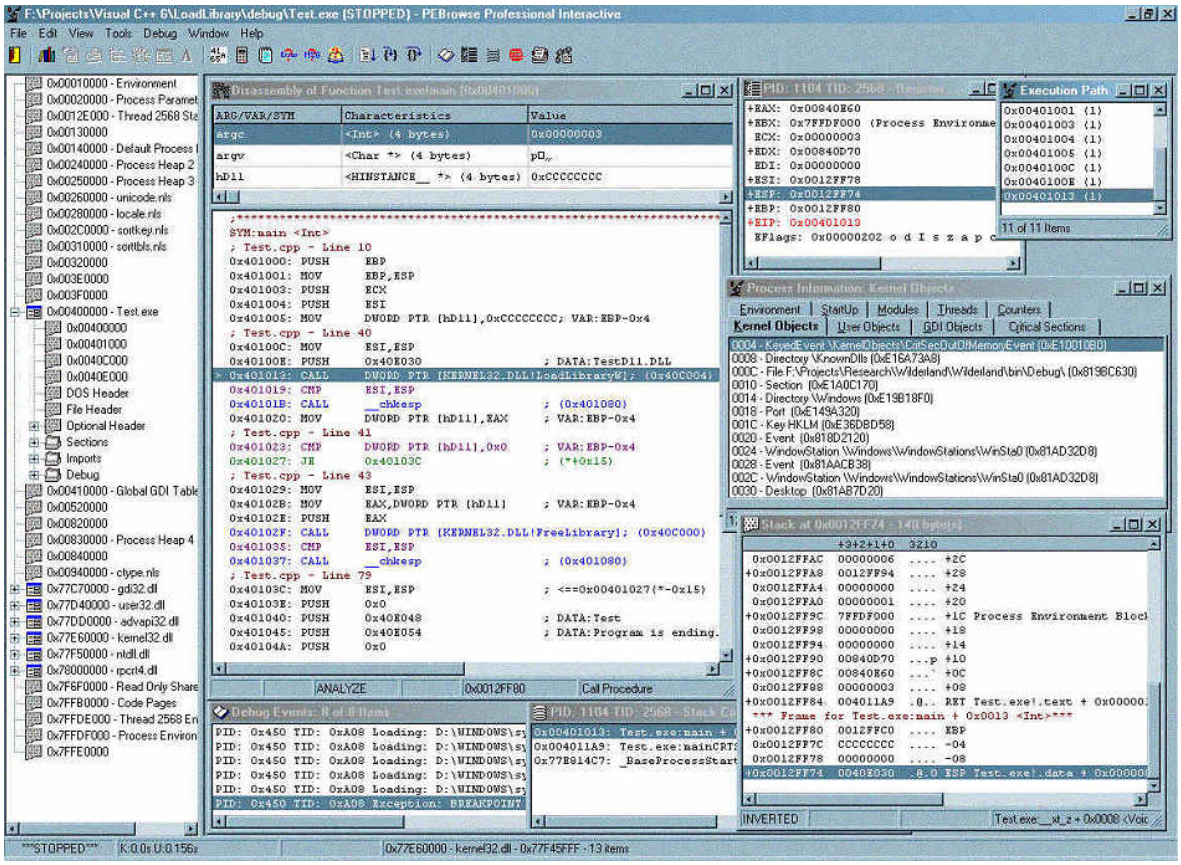
  Visual Studio를 설치 시 같이 제공되는 툴로서 명령 프롬프트 상에서 동작한다.



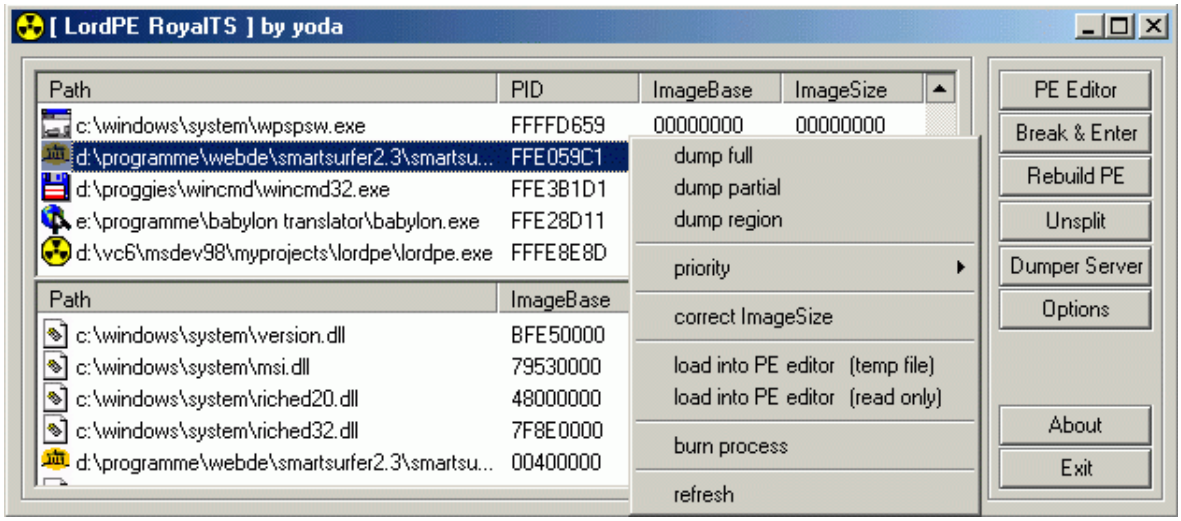
[Dumpbin.exe 실행 모습]

- PE Browser

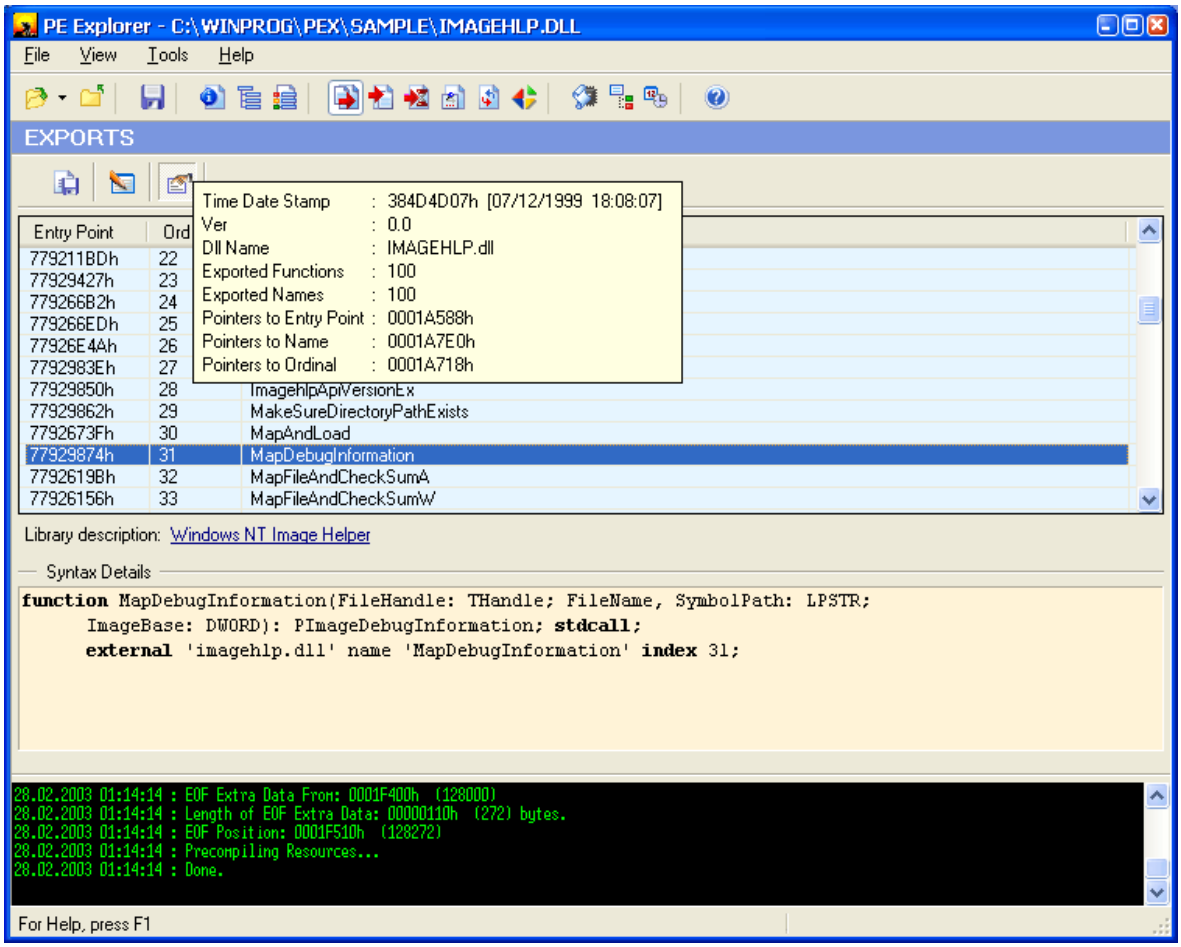
  Smidgeonsoft(http://www.smidgeonsoft.com) 에서 제공하는 Visual 도구



- LordPE



- PE Explorer



그 외 PE iDentifier, pe-scan 등이 있으며 입맛에 따라 골라 쓰면 된다.

1.4. Dll

이번에는 Dll에 대해 살펴보도록 한다.
거듭 말하지만 본인이 Windows System을 잘 모르는지라 정확하고 완전한 설명이 아니라는 것을 강조하고 싶으며 일단 아는 선에서 책과 문서를 참고하여 정리를 해보자 한다.

1.4.1. Introduce Dll

Dll은 Dynamic Link Library의 약자이며 "동적 링크 라이브러리" 라고 부른다.
일반적으로 소스 코드가 컴파일 된 후 링크될 때 라이브러리 파일에서 특정 함수들의 기능이 구현된 부분이 실행 파일에 같이 덧붙여지는 정적 링크 라이브러리와는 달리 Windows System은 프로그램이 실행될 때 Dll 파일로부터 특정 함수들의 기능이 구현된 부분을 동적으로 링크하는 동적 링크 라이브러리를 이용한다.
일단 읽어 온 Dll의 함수는 프로그램 내부 함수처럼 호출할 수 있으며 Dll은 필요한 시점에 메모리로 읽어오고 불필요하면 메모리에서 없앨 수도 있다.

이러한 Dll을 사용시 얻을 수 있는 이점은 다음과 같다.

- 메모리와 하드 디스크를 절약할 수 있다
  정적 링크 라이브러리를 사용하는 경우 공통되는 기능이 여러 프로그램에 중복된 형태로 존재하게 되어 실행파일의 크기가 커지게 된다. Dll 사용 시 여러 프로그램이 하나의 Dll을 공유하는 형태로 실행되기 때문에 메모리와 하드 디스크를 절약할 수 있다.

- 유지보수가 간편하다.
  실행 파일과 독립된 형태로 존재하기 때문에 보안 버그와 같이 패치가 필요할 때 Dll 파일만 패치한 후 배포함으로써 간단히 이루어진다.

1.4.2. Dll의 종류

작성 가능한 Dll의 종류에는 두 가지가 있는데 아래와 같다.

- 일반 Dll
  다른 언어로 작성된 프로그램에서도 쓸 수 있는 범용적인 Dll. 단 외부에서 사용되기 위해 만들어지는 함수는 모두 C의 형태로만 만들어져야 하며 오버로딩, 클래스와 같은 기능은 다른 프로그램에서 호출할 수 없다.

- 확장 Dll
  일반 Dll의 단점인 오버로딩, 클래스와 같은 C++의 특징을 이용할 수 있도록 설계되었지만 C++에서만 이용할 수 있다.

1.4.3. 명시적 로딩과 암시적 로딩

일반적으로 Dll은 변수와 함수들을 외부 어플리케이션이 사용할 수 있도록 제공하는 역할만을 하게 되는데 이렇게 Dll이 변수나 함수를 외부로 제공하는 것을 export라 하고 Dll이 export하는 변수나 함수를 가져 오는 것을 import라고 한다.
이 Dll을 가져오는 방법에 따라 명시적 로딩과 암시적 로딩으로 나뉘게 된다.

- 암시적 로딩
  암시적 로딩 방법은 프로그램 시작 시 해당 Dll을 바로 로드하는 방법이며 MSDN에는 아래와 같이 기술되어 있다.

DLL에 암시적으로 링크하려면 실행 파일에서 DLL 공급자로부터 다음과 같은 항목을 가져와야 합니다.

1. 내보내기 함수 및/또는 C++ 클래스의 선언이 들어 있는 헤더 파일(.H 파일)
2. 링크할 가져오기 라이브러리(.LIB 파일). 링커는 DLL이 빌드될 때 가져오기 라이브러리를 만듭니다.
3. 실제 DLL(.DLL 파일)

DLL을 사용하는 실행 파일에는 내보내기 함수를 호출하는 각 소스 파일마다 내보내기 함수 또는 C++ 클래스가 들어 있는 헤더 파일이 포함되어야 합니다. 코딩 관점에서 볼 때 내보내기 함수를 호출하는 것은 다른 함수 호출과 비슷합니다.
호출 실행 파일을 빌드하려면 가져오기 라이브러리와 링크해야 합니다. 외부 메이크 파일을 사용하는 경우에는 링크할 다른 개체 파일(.OBJ) 또는 라이브러리가 나열된 가져오기 라이브러리의 파일 이름을 지정합니다.
운영 체제에서 호출 실행 파일을 로드할 때 .DLL 파일을 찾을 수 있어야 합니다.

▪ 명시적 로딩
프로그램 실행 중에 Dll을 로드하여 익스포트 된 함수를 사용하는 방법이며 MSDN에는 아래와 같이 기술되어 있다.'

명시적 링크의 경우에는 응용 프로그램이 런타임에 함수를 호출하여 DLL을 명시적으로 로드해야 합니다. DLL에 명시적으로 링크하려면 응용 프로그램은 다음과 같은 작업을 수행해야 합니다.

LoadLibrary 또는 이와 유사한 함수를 호출하여 DLL을 로드하고 모듈 핸들을 가져와야 합니다.
GetProcAddress를 호출하여 응용 프로그램이 호출하려는 각각의 내보내기 함수에 대한 함수 포인터를 가져와야 합니다. 응용 프로그램에서는 포인터를 통해 DLL의 함수를 호출하기 때문에 컴파일러는 외부 참조를 생성하지 않습니다. 따라서 가져오기 라이브러리와 링크할 필요도 없습니다.
DLL을 사용할 필요가 없으면 FreeLibrary를 호출해야 합니다.

다음 코드에서는 이러한 예를 보여 줍니다.

```
typedef UINT (CALLBACK* LPFNDLLFUNC1)(DWORD,UINT);
...

HINSTANCE hDLL;               // Handle to DLL
LPFNDLLFUNC1 lpfnDllFunc1;    // Function pointer
DWORD dwParam1;
UINT  uParam2, uReturnVal;
```

```
hDLL = LoadLibrary("MyDLL");
if (hDLL != NULL)
{
    lpfnDllFunc1 = (LPFNDLLFUNC1)GetProcAddress(hDLL,
                                                "DLLFunc1");
    if (!lpfnDllFunc1)
    {
        // handle the error
        FreeLibrary(hDLL);
        return SOME_ERROR_CODE;
    }
    else
    {
        // call the function
        uReturnVal = lpfnDllFunc1(dwParam1, uParam2);
    }
}
```

1.4.4. 익스포트 시 주의할 점

함수를 익스포트 하는 방법은 다음의 두 가지가 있다.

▪ 모듈 정의 파일을 준비해 EXPORTS 문을 기술한다.

```
EXPORTS
        _GetFuncNumber
```

▪ 소스 코드 내의 함수 정의에 _declspec 키워드로 정의한다.

```
extern “c” _declspec(dllexport) int GetFuncNumber(HWND hWnd)
{
  Do Something…
}
```

대부분의 개발자들이 _declspec를 이용한 방법을 사용할 것이다. 문제는 extern “c”에 있다. 몇몇 dll injection을 소개하는 책, 문서 중에는 이 extern “c”를 없이 작성된 Dll 내의 함수를

GetProcAddress(handle, “함수이름”)와 같이 호출하는 예제가 있는데 당연히 동작하지 않는다. extern “c”로 정의하지 않은 경우 위의 예제에서 GetFuncNumber는 c++ 컴파일러가 링커로 넘길 때의 이름은 @GetFuncNumber$어쩌구저쩌구 가 된다.(c++로 작성된 코드의 이야기이다. 확장자가 c라면 이야기는 틀려진다.)
extern “c”로 정의했을 때의 GetFuncNumber의 내부 이름은 _GetFuncNumber이 되어서 정상적으로 함수를 호출할 수 있다.

export된 함수명은 dumpbin.exe 파일의 /exports 옵션으로 알 수 있다.
아래의 코드와 그 결과를 비교하면 명확히 알 수 있을 것이다.

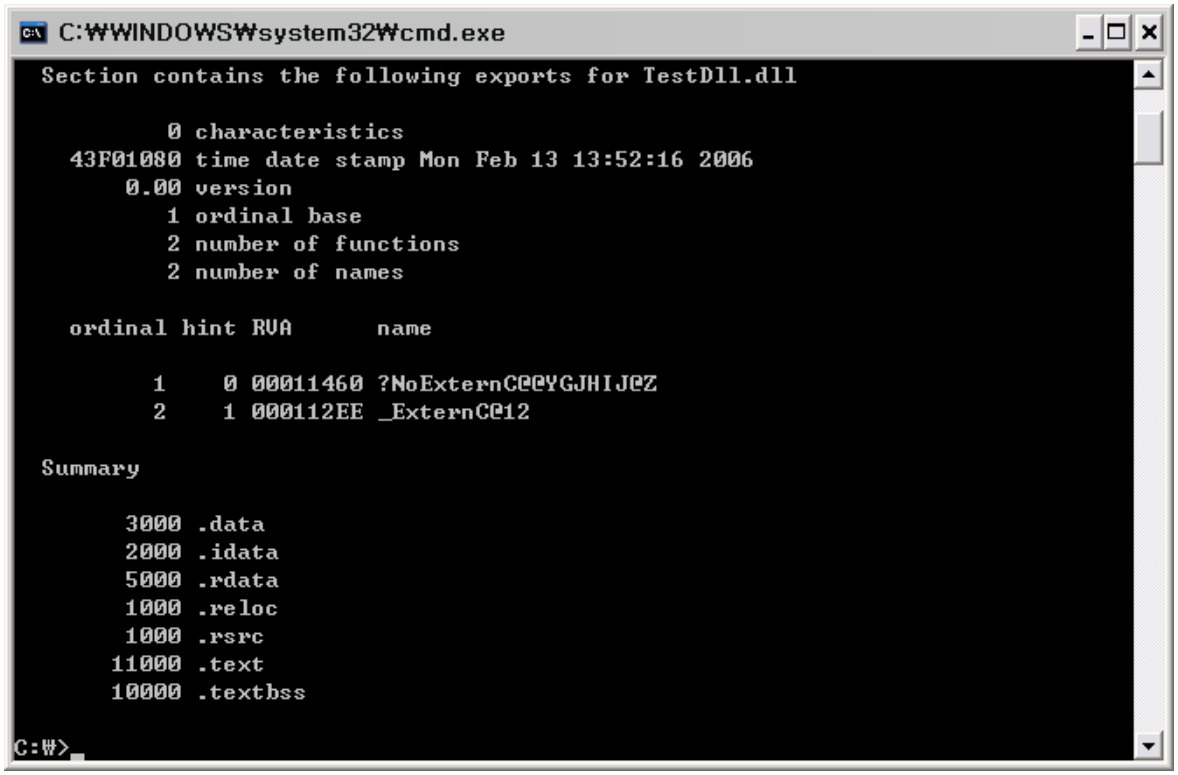
```
#include <windows.h>

HINSTANCE hinst;

BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD dwReason, LPVOID lpReserved) {
	if(dwReason == DLL_PROCESS_ATTACH) {
		hinst = hinstDll;
	}
	return TRUE;
}

__declspec(dllexport) LRESULT CALLBACK NoExternC(INT nCode, WPARAM wp, LPARAM lp)
{
	...
}

extern “C” __declspec(dllexport) LRESULT CALLBACK ExternC(INT nCode, WPARAM wp, LPARAM lp) {
	...
}
```



[export 된 함수 이름]

☞ Visual Studio 2003 환경에서는 위의 그림에서도 보다시피 _function@number 형식으로 나온다. 왜 이런지 알수는 없지만 Visual Studio 6.0 환경에서는 정상적인 _function의 형태로 익스포트 된다.

1.4.5. 참고문서

WebSite

1. Microsoft PE File Pormat
   http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx
   http://msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx
   http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx

2. API Hooking Revealed
   (http://www.codeproject.com/system/HookSys.asp)

Documents

1. Windows 시스템 실행파일의 구조와 원리, 이호동 저, 한빛미디어
2. Debugging Applications for .net and windows, 존 로빈스 저, Microsoft Press