SSDT Hooking

Last Update : 2007년 1월 4일

Written by Jerald Lee

Contact Me : lucid78@gmail.com

본 문서는 커널모드 후킹 기술 중의 하나인 SSDT 후킹에 대해 정리한 것입니다. 제가 알고있는 지식이 너무 짧아 가급적이면 다음에 언제봐도 쉽게 이해할 수 있을 정도로 쉽게 쓸려고 노력하였습니다. 제가 작성하였던 기존의 Windows Hook 시리즈가 유저모드에서의 후킹을 다루었던 반면 본 SSDT는 커널모드에서의 후킹을 사용하므로 디바이스 드라이버를 다루는 부분이 포함되어 있습니다. 윈도우 디바이스 드라이버의 경우 많은 부분을 상세히 설명하지는 못했으나 코드 작성에 필요한 부분을 중점으로 설명하였습니다. 다양한 분야의 방대한 자료들 중에서 필요한 것만 골라 요약하다보니 두서없이 써내려간 부분도 많이 있습니다. 읽으시는 분들의 양해를 부탁드립니다.

이번에도 역시 기존에 나와있는 여러 문서들을 짜집기 하는 형태로 작성되었으며 기술하지 못한 원문 저자들에게 매우 죄송할 따름입니다.

본 문서는 읽으시는 분들이 어느 정도 Windows API를 알고 있다는 가정 하에 쓰여졌습니다. 본 문서에서 자세히 다루지 못한 내용은 참고 자료를 살펴보시기 바랍니다.

제시된 코드들은 Windows XP Professional Service Pack 2, Windows Server 2003 SP1 DDK[1]에서 테스트되었습니다.

문서의 내용 중 틀린 곳이나 수정할 곳이 있으면 연락해주시기 바랍니다.

[1] http://www.microsoft.com/whdc/devtools/ddk/default.mspx

목차

그림 목차

1. Windows System의 구조

먼저 Windows System에 대해 알아봅니다. 이 문서를 읽으시는 많은 분들이 이미 해당 내용을 잘 알고 계실테지만 만약을 위해 간략하게 설명해봅니다.

Windows는 유저모드(User Mode)와 커널모드(Kernel Mode)로 구분되어 프로세스가 실행됩니다. 커널모드에서 동작하는 프로세스만이 컴퓨터에 장착된 모든 메모리와 하드웨어에 대한 직접적인 접근이 가능하며 디바이스 드라이버가 가장 대표적인 예라고 할 수 있습니다. 일반적인 윈도우 프로그램의 경우 유저모드 프로세스가 작동하는 것이므로 직접 하드웨어 장치나 메모리에 접근할 수는 없습니다. 하드웨어 장치나 메모리에 접근하기 위해 유저모드 프로그램은 시스템 서비스(API라고 이해하시면 될 듯합니다.)를 호출하게 되고 운영체제는 trap을 발생시켜 커널모드로의 스위칭을 위한 스레드를 호출하게 됩니다. 이후 이 스레드에서 하드웨어 장치 또는 메모리에 접근하게 됩니다. 이를 context switch(공룡책에는 문맥 교환이라고 번역되어 있습니다.)라고 합니다.

정확한 시점은 알 수 없지만(-_-;) 어느 시점 이후부터 최근에 제작되는 OS는 대부분 마이크로 커널의 형태를 가집니다.(좀 더 엄밀하게 말해서 수정된 마이크로 커널의 형태를 가집니다.) 운영체제에서 프로세스 스케쥴링이나 메모리 관리 등과 같은 가장 핵심적이고 필수적인 부분만 커널에 구현이 되어 있고 기타 기능은 서브시스템에서 관리하는 형태를 띄게 됩니다. 아래 그림 1은 Windows Internal에서 발췌한 간단한 형태의 Windows 구조입니다.

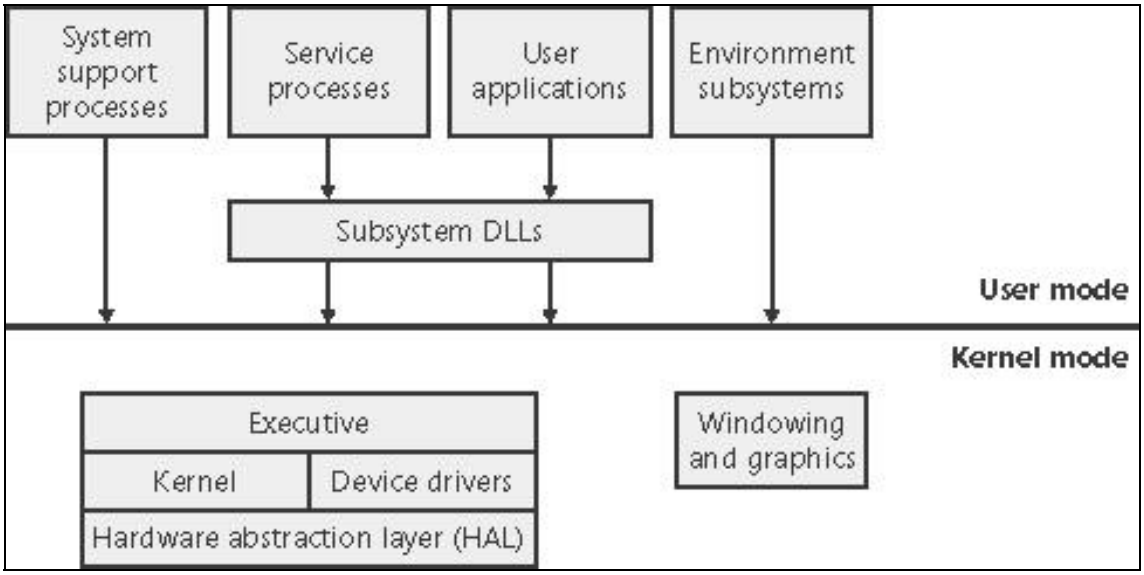


그림 1. **Simplified Windows architecture**

2. Native API

위에서도 잠시 설명했지만 유저모드 프로세스는 하드웨어 장치나 메모리에 접근하기 위해 커널모드로 스위칭을 해야 합니다. 일반적으로 사용하는 Win32 API는 이러한 기능을 자동으로 수행하는데 사실 Win32 API는 운영체제에서 제공하는 서브시스템 중의 하나입니다.

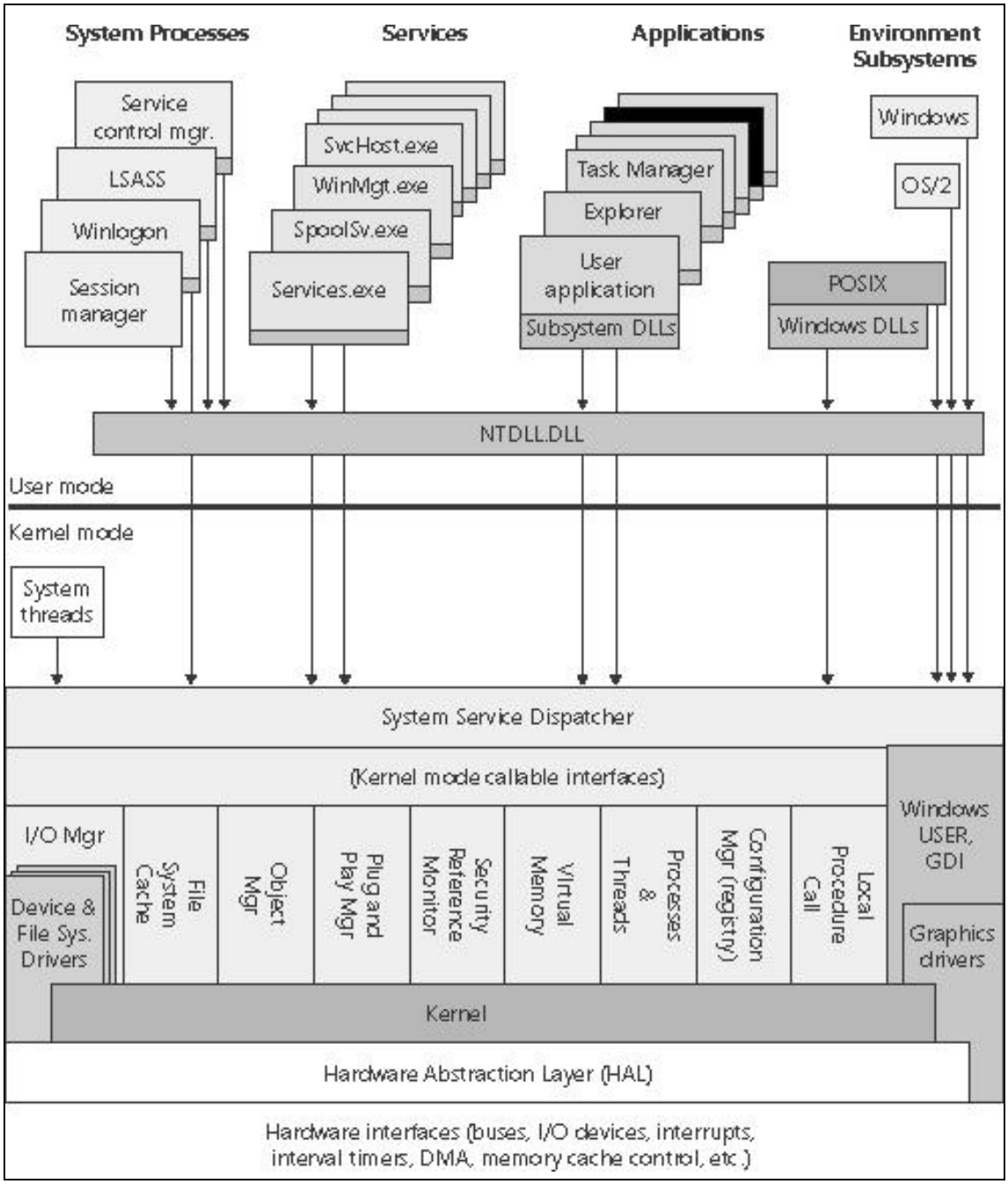


그림 2. Windows Architecture

위의 그림 2에서 살펴볼 수 있듯이 Win32, OS/2, Wow, POSIX 등은 모두 서브시스템 환경의 일부이

며 이는 운영체제가 다른 기종에서 작성된 프로그램들의 호환을 위해 제공하는 것들입니다. 각각의 서브시스템들은 커널모드에서 동작하는 서비스(시스템 서비스라고 합니다.)들을 이용하여 하드웨어 장치나 메모리에 접근하게 되며 이 때 사용되는 커널모드 서비스들은 Native API라고 불리는 함수들을 호출하여 이러한 작업들을 수행하게 됩니다.

조금 더 구체적으로 알아봅시다.

Microsoft는 보안을 이유로 유저모드에서 동작하는 코드가 직접 커널모드에서 동작하는 코드를 call 하는 것을 허용하지 않습니다.

MFC 개발자들이 디렉토리 구조를 알기 위해서 자주 사용하는 FindFirstFile(), FindNextFile() 함수가 있습니다.

FindFirstFile()로 디렉토리를 찾은 후 디렉토리 내 모든 구조를 FindNextFile()이 찾게 됩니다. 프로그래머는 단순히 위의 두 API를 호출함으로써 임의의 디렉토리 구조를 알 수 있지만 실제 운영체제 내부에서는 더 많은 작업들이 수행되게 됩니다.

즉, 유저모드 코드 FindNextFile()은 NtQueryDirectoryFile()이라는 Native API를 호출하고 NtQueryDirectoryFile()은 동일한 이름의 커널모드 코드의 Wrapper 역할을 함으로써 디렉토리 구조를 얻을려는 목적을 이루게 됩니다.

Visual Studio를 설치하면 기본적으로 설치되는 Dependency Walker을 이용해 ntdll.dll을 열어보면 동일한 이름을 가진(Nt, Zw로 구분되는) 함수가 존재함을 볼 수 있습니다.

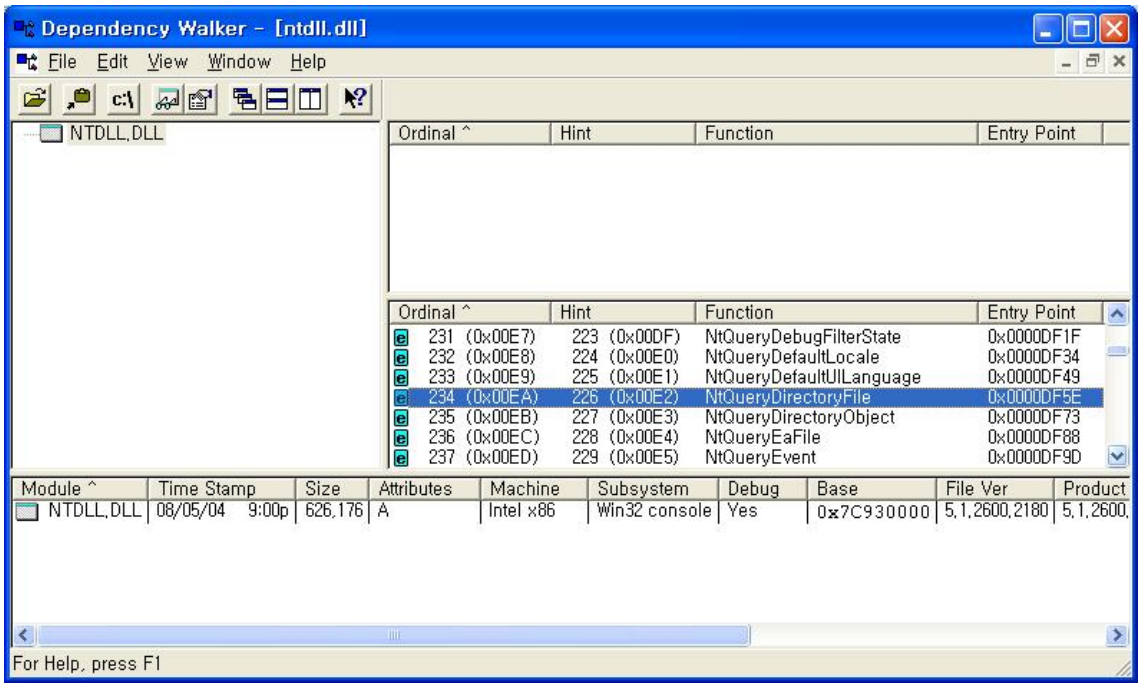


그림 3. NtQueryDirectoryFile() API

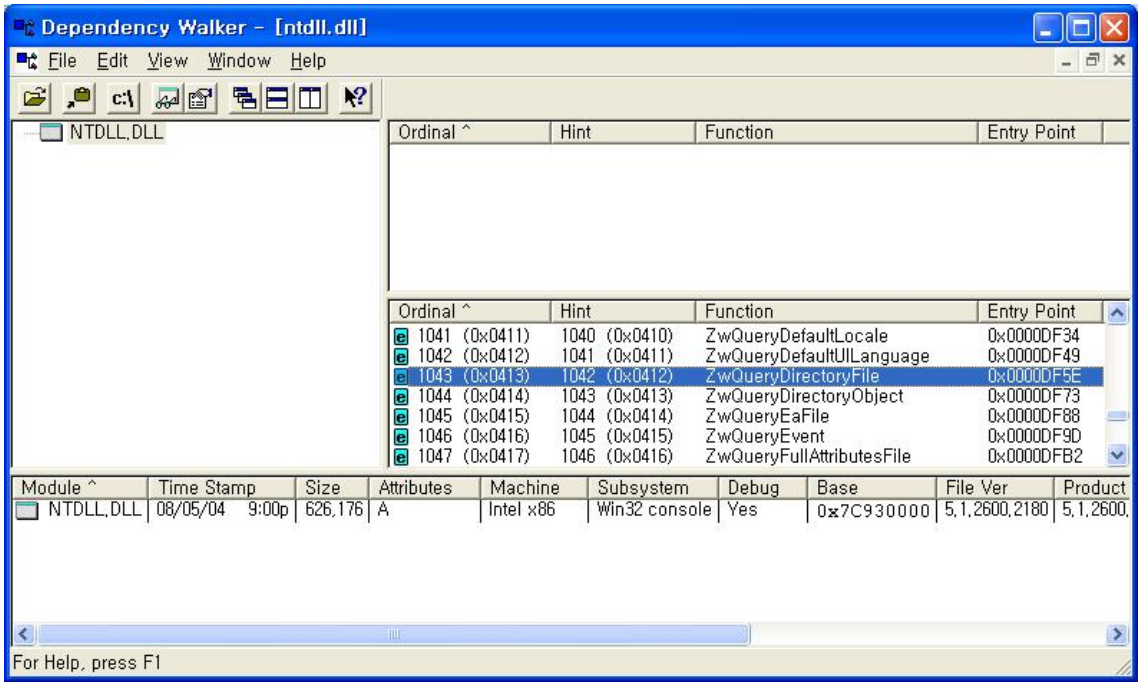


그림 4. ZwQueryDirerctoryFile() API

Native API는 운영체제에서 사용하기 위해 만들어진 것이며 대부분이 문서화되어 있지 않습니다. 하지만 수많은 해커들의 노력에 의해 많은 수의 Native API가 Undocument API 라는 이름으로 문서화되기에 이르렀습니다.

Native API들은 보통 "Nt" 또는 "Zt"로 시작하는 이름을 가집니다. 유저모드에서 동작하는 프로그램이 Native API를 이용하려면 ntdll.dll 파일을 통해야만 하며 ntdll.dll 파일은 모든 시스템 서비스들의 진입점을 포함하고 있습니다. MSDN에 따르면 Zw로 시작하는 함수의 경우 해당 함수를 호출한 프로세스에 대한 접근 권한 체크를 하지 않습니다. 좀 더 자세한 설명은 아래의 URL을 참고하시기 바랍니다.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/Kernel_r/hh/Kernel_r/k111_80b1882a-8617-45d4-a783-dbc3bfc9aad4.xml.asp

이 과정은 그림 5와 같습니다.

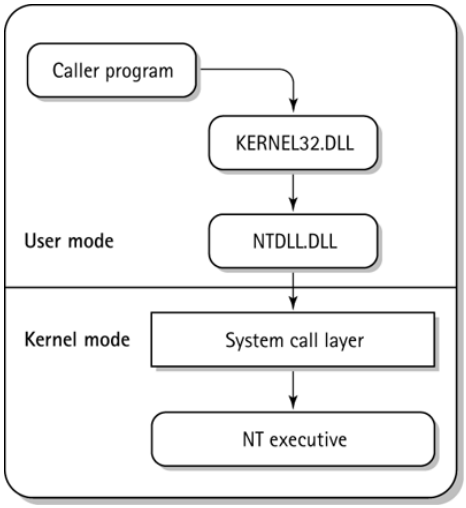


그림 5. Native API 진입 단계

위의 그림 5과 같이 ntdll.dll은 커널모드로 진입하기 위해 인터럽트를 사용합니다. INT 2E는 XP 이전 버전의 운영체제에서 사용되던 방법이며 성능상의 이유로 XP부터는 SYSENTER을 사용합니다. INT 2E는 소프트웨어 인터럽트, SYSENTER는 하드웨어 인터럽트라고 생각하시면 될 듯합니다. 이 내용은 다음 절에서 다루도록 하겠습니다.

그럼 실제 함수는 어떻게 동작하도록 되어있는지 살펴봅시다.
위에서 잠깐 언급했었던 NtReadFile을 예로 들어보도록 하겠습니다. 이 작업은 드라이버스튜디오 또는 WinDbg가 준비되어 있다면 가능합니다. 본 문서에서는 드라이버 스튜디오를 이용하였습니다. 여기서부터는 Devguru의 “Attack Native API”의 내용을 참고하였음을 미리 밝힙니다.

먼저 유저모드에서의 함수 동작을 알기 위해 Softice를 실행시키기 전 NtReadFile 함수가 들어있는 ntdll.dll을 로드해야 합니다.(앞에서도 이야기 했지만 ntdll.dll 파일에 유저모드에서 커널모드로의 진입점이 존재합니다). 이 함수는 디폴트로 로드되지 않습니다.
아래 그림 6은 Driver Studio에 포함된 Symbol Loader 프로그램을 이용해 ntdll.dll 을 로드하는 화면을 캡쳐한 것입니다.

File – LoadExports 를 클릭하여 ntdll.dll 을 로드합니다.

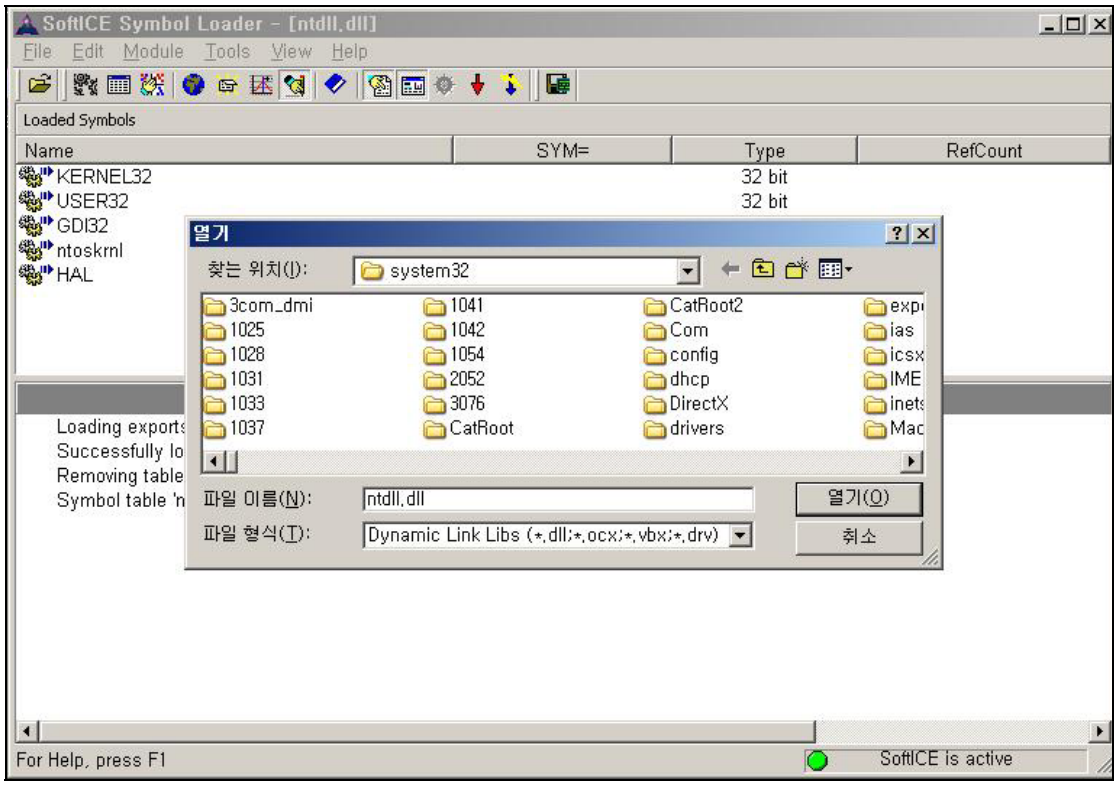


그림 6. ntdll Load

로드에 성공하면 아래 그림 7에서 빨간색 부분처럼 ntdll 부분이 생성됩니다.

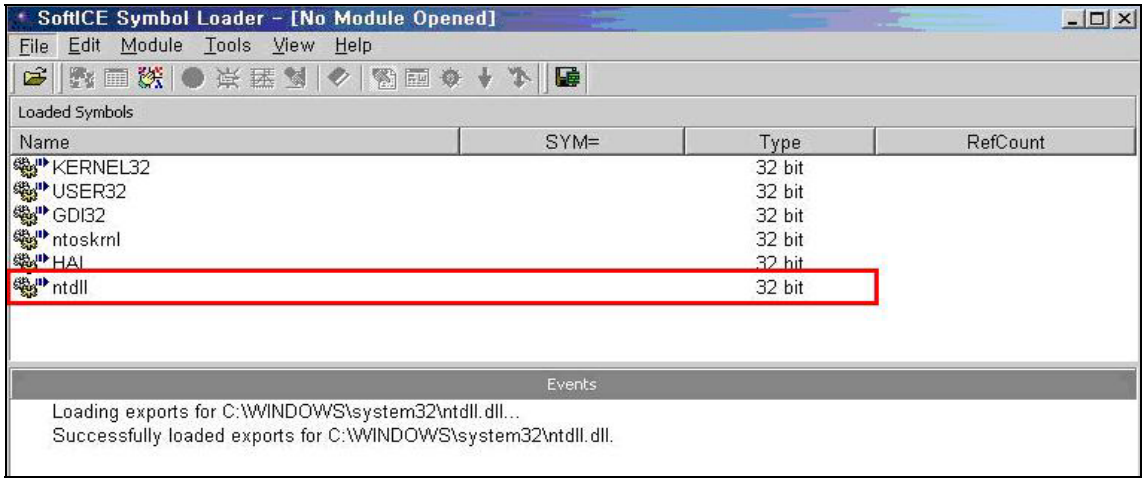


그림 7. ntdll Load에 성공한 화면

이제 Softice를 실행시킵니다. Softice가 성공적으로 실행된 후 Ctrl + D 를 누르면 SoftIce 화면으로 진입하게 됩니다.

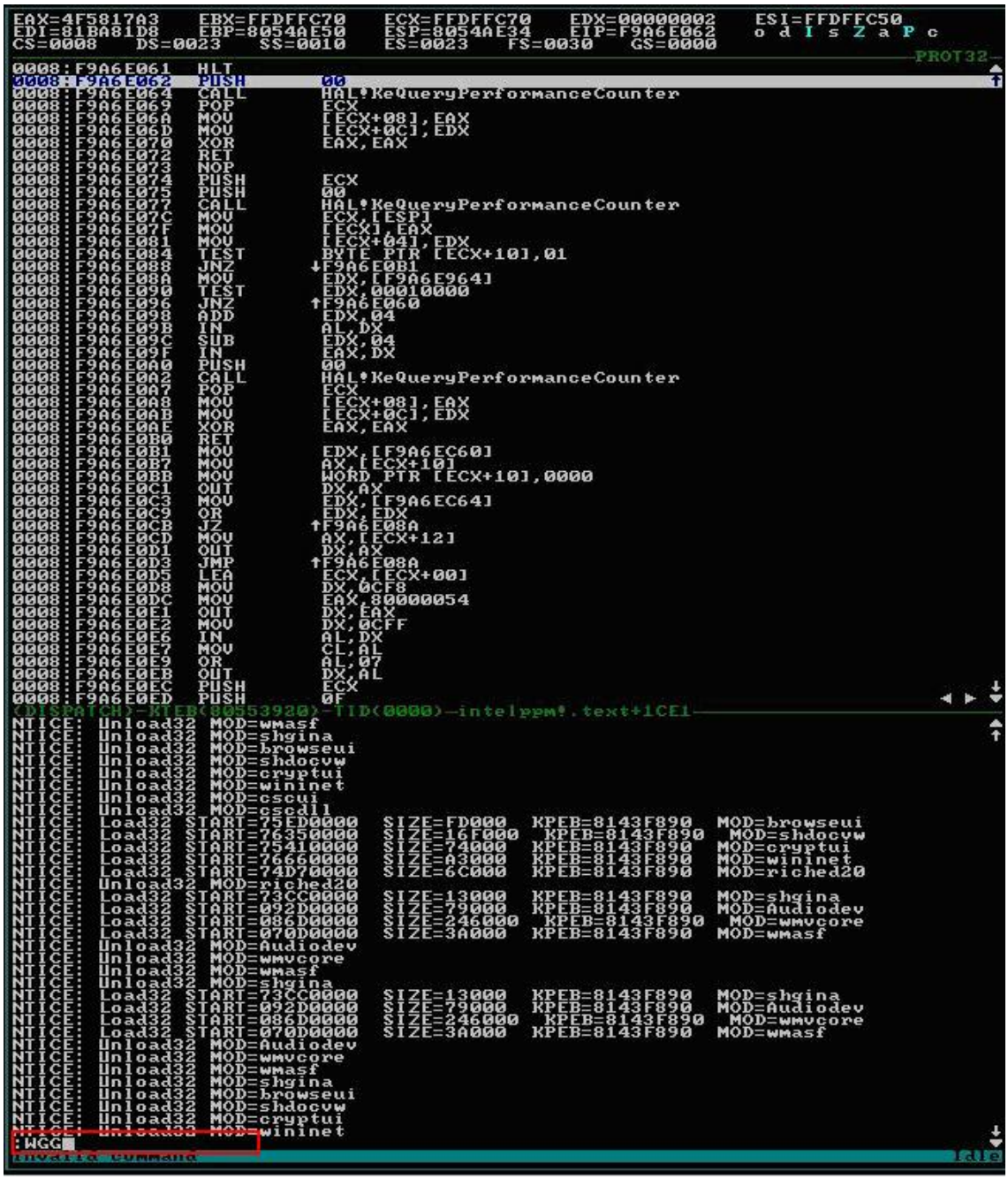


그림 8. Softice로 진입한 화면

위의 그림 8에서 빨간색으로 표시된 부분이 명령창입니다. Vmware를 통한 이미지 캡쳐라 실제 명령어를 수행한 결과 화면은 캡쳐할 수 없어서 텍스트 형태로 결과를 보이도록 하겠습니다.

먼저 명령어 창에 u ntdll!NtReadFile을 입력한 결과입니다.

```
: u ntdll!NtReadFile
ntdll!NtReadFile
```

```
mov     eax,    000000b7
mov     edx,    ntdll!KiFastSystemCall
call    edx
ret     0024
```

```
: u ntdll!KiFastSystemCall
ntdll!KiFastSystemCall
mov     edx,    esp
sysenter
```

eax 레지스터에 NtReadFile의 주소인 0xb7 값을 넣고 KiFastSystemCall을 호출합니다.
KiFastSystemCall에서는 edx에 esp 값을 넣은 후 SysEnter을 호출하게 됩니다.

다음으로 ZwReadFile을 살펴보겠습니다. 명령창에 u ntdll!ZwReadFile을 입력하면 NtReadFile 값이 나오는 것을 확인할 수 있습니다.
즉, Wrapper인 ntdll을 통한(유저모드에서의) NtReadFile, ZwReadFile은 결국 NtReadFile 함수를 호출하게 됩니다. 유저모드에서는 커널모드 함수인 ZwReadFile을 호출하더라도 결국 NtReadFile을 호출하게 된다는 말입니다.

커널모드(Ntoskrnl.lib) 에서의 NtReadFile, ZwReadFile은 모두 정상적으로 해당 함수를 실행합니다.
명령 창에서 직접 u ntoskrnl!NtReadFile, u ntoskrnl!ZwReadFile을 실행해보시기 바랍니다.

3. System Service Descriptor Table

Windows는 수많은 테이블들의 집합이라고 할 수 있습니다. 앞에서 살펴보았듯이 인터럽트가 발생하면 해당 인터럽트의 종류에 따른 결과를 반환하게 됩니다. 인터럽트가 발생했을 때 어떤 시스템 서비스가 호출되어야 하는지에 대한 판단을 하기 위해 윈도우는 테이블을 참조하게 됩니다.(정확히 얘기하면 CPU는 해당 시스템 서비스들이 위치하고 있는 메모리 상의 주소를 알아야 할 필요가 있습니다. 하지만 모든 주소를 내부적으로 저장할 수 없기 때문에 CPU는 테이블이라는 자료 구조를 사용하는 것이며 윈도우, 즉 운영체제는 이를 이용하는 것입니다. 좀 더 자세하고 정확한 내용은 Windows Internals를 참고하시기 바랍니다.) 이런 테이블들(CPU가 참조하는)에는 여러 종류의 테이블이 있으며 아래와 같습니다.

- GDT : Global Descriptor Table
- LDT : Local Descriptor Table
- IDT : Interrupt Descriptor Table

이런 테이블들을 통칭하여 System Service Descriptor Table이라고 부릅니다.

운영체제 또한 자신만의 테이블을 만들어 참조하는 방법을 사용하는데 운영체제에서 구현된 중요한 테이블 중에 아래의 테이블이 있습니다.

- SSDT : System Service Dispatch Table

바로 이 문서의 목표인 SSDT 테이블입니다. 이 테이블은 시스템에서 이용 가능한 모든 시스템 서비스들의 주소를 가지고 있으며 인터럽트 발생 시 운영체제는 이 테이블을 참고하여 적절한 결과 값을 돌려주게 됩니다. 이 테이블을 간단히 조작/변경함으로써 시스템의 모든 서비스를 다룰 수 있기 때문에 커널 루트킷, 안티 바이러스 프로그램 양쪽 모두에 사용됩니다. 커널 루트킷의 경우에는 프로세스/파일/디렉토리 은닉 등을 위해 사용됩니다.
SSDT 를 이용한 후킹은 GDT, IDT 등과 복합 형태의 후킹 코드로서 이용되기도 합니다.

여기서 앞 절에서 잠시 나왔던 INT 2E와 SYSENTER을 알아봅시다.
커널 모드의 System Call을 위해 Intel 기반 하에서 Windows는 INT 2E 또는 SYSENTER을 이용합니다.
Windows 2000과 이전 버전의 Windows는 유저모드에서 커널모드로 진입하기 위해 소프트웨어 인터럽트인 INT 2E를 이용했습니다. 인터럽트가 발생하면 0x2E 값은 Interrupt Descriptor Table(IDT)에서 Processor가 참조하는 offset 값이 됩니다.(일반적으로 EAX 레지스터에 들어가는 값들은 대부분 Offset으로 쓰입니다.) 이 Offset이 가리키는 테이블의 값은 System Service Dispatcher(일반적으로 KiSystemService라고 합니다.)의 주소입니다. CPU는 Instruction Pointer Register(IP Register)에 이 값을 Load하고 System Service Dispatcher가 실행되게 됩니다. System Service Dispatcher는 System Service Dispatch Table(SSDT)에 기록된 System Service를 실행하게 되며(실행할 System Service의

구분은 EAX 레지스터를 참조하며 EDX 레지스터에서 시스템 서비스에 전달된 매개변수의 목록-메모리 주소-을 참조합니다. Windows NT의 경우 EDX 레지스터가 아니라 EBX 레지스터를 사용합니다.)
유저모드에서 호출된 API가 재호출되게 됩니다. 즉, 커널모드에서 API가 수행되게 됩니다.

Windows XP에서는 INT 2E 대신 SYSENTER이라고 하는 Instruction을 사용하게 됩니다. INTEL은 SYSENTER Instruction을 CPU Instruction Set에 포함시킴으로써 성능 상의 향상을 꾀했습니다. 이 Instruction은 펜티엄 2 이상의 CPU에서 지원하기 시작하였으며 Windows 2000에서는 INT 2E와 SYSENTER을 같이 사용했으나 XP부터는 SYSENTER만을 사용하게 됩니다. 윈도우는 이 Instruction을 지원하기 위해 부팅할 때 명령과 연결된 레지스터에서 커널의 System Service Dispatcher 루틴의 주소를 저장합니다.
ntdll.dll이 SYSENTER Instruction을 발생시키면 Processor는 IA32_SYSENTER_EIP 라는 특수 Register에 저장된 값, 즉 System Service Dispatcher의 주소를 체크하여 IP Register에 Load 하고 실행시킵니다. INT 2E와 마찬가지로 실행할 System Service의 구분은 EAX 레지스터에 저장된 값을 참조하며 전달되는 매개변수의 정보는 EDX 레지스터를 참조합니다. 유저모드로 되돌아가기 위해서는 일반적으로 SYSEXIT Instruction을 사용합니다.

AMD의 경우 SYSCALL이라는 SYSENTER과 비슷한 Instruction을 사용하는데 이는 생략하도록 하겠습니다. 더 자세한 정보는 Windows Internals 와 참고자료 web)5를 살펴보시기 바랍니다.

위에서 설명한 과정을 도식하면 아래와 같습니다.

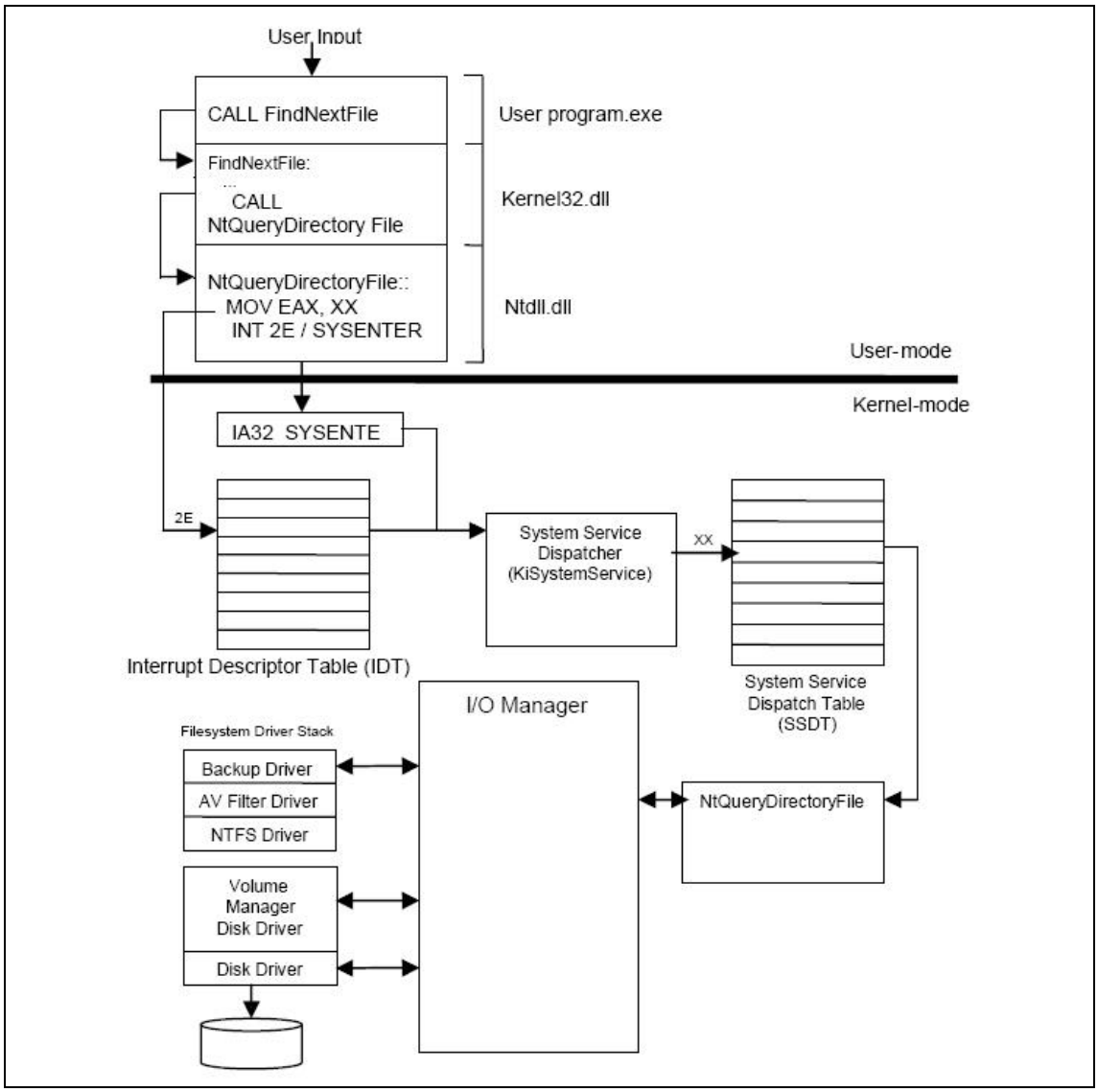


그림 9. FindFirstFile() And FindNextFile(), -Inside Windows Rootkits에서 발췌-

본 문서에서 다루고자 하는 SSDT Hooking의 목적은 바로 아래 그림으로써 표현될 수 있습니다.

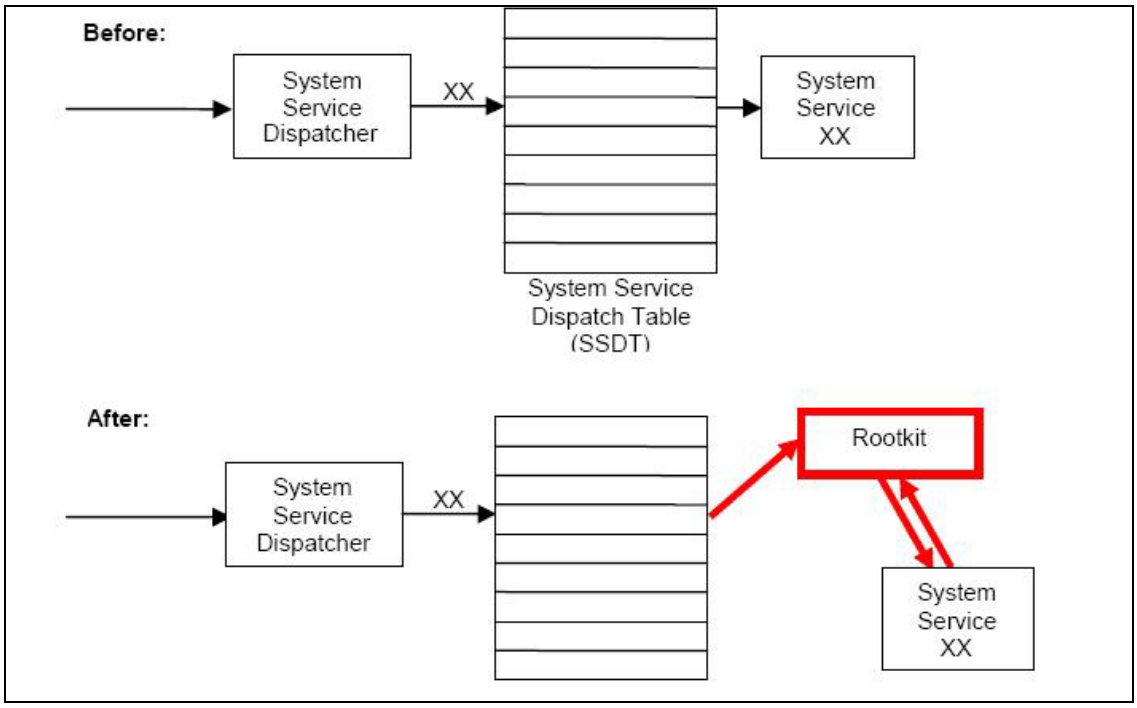


그림 10. SSDT Hooking, -Inside Windows Rootkits에서 발췌-

위의 그림 10에서 볼 수 있듯이 Rootkit을 중간에 삽입시키는 것이 본 문서의 목적이라고 할 수 있겠습니다.

자 그럼 어떻게 Rootkit을 삽입시키는가, 아니 그전에 어떻게 SSDT를 후킹하여 우리가 만든 Rootkit으로 향하게 하는가, 그것이 해결해야 할 가장 중요한 문제이며 그 답은 디바이스 드라이버에 있습니다.

4. Device Driver 기초 및 활용

자 디바이스 드라이버 시간입니다.
기존의 SDK 프로그래밍에 익숙한 사용자라면 약간 어색할 듯 하고 프로그래밍에 자신이 없는 분이라면 매우 어려울 듯 합니다.
몇 년 전만 해도 윈도우 드라이버 프로그래밍은 윈도우 프로그래밍의 꽃이다 라고 생각되었었습니다.(저만의 견해인가요? -_-;). 자료라고는 그저 MSDN 밖에 없던 시절, 극악무도 또는 진짜 독한 사람들만이 하는 분야라고 취급되었었습니다. 그만큼 어렵다는 이야기입니다. -_-;
그러던 것이 시간이 흘러 한글로 된 윈도우 디바이스 드라이버 책도 나올만큼 대중화(?)되었습니다만 여전히 그 난해함은 있습니다.

이번 절은 정말정말 필요한 부분만을 모아놓았으니 심화학습을 원하시는 분들은 드라이버 관련 전문 서적을 읽으시고 정말 당장 써먹을 수 있는게 필요해~ 하시는 분들은 RootKits[1]를 읽으시길 바랍니다.

자 이제 시작해보겠습니다.
먼저 RING에 대한 설명으로 시작해야 할 것 같습니다.
앞 절에서 유저모드와 커널모드에 대한 설명이 있었습니다. 이것은 운영체제에 의한 분류(?)법이라고 할 수 있습니다. INTEL CPU에서는(이하 x86) 이를 Ring이라고 부르며 우리가 잘 알고 있는 유저모드는 Ring 3, 커널모드는 Ring 0를 나타내게 됩니다.

x86에서 지원하는 Ring 모델의 구조는 아래 그림 11과 같습니다.

[1] Rootkits: Subverting the Windows Kernel

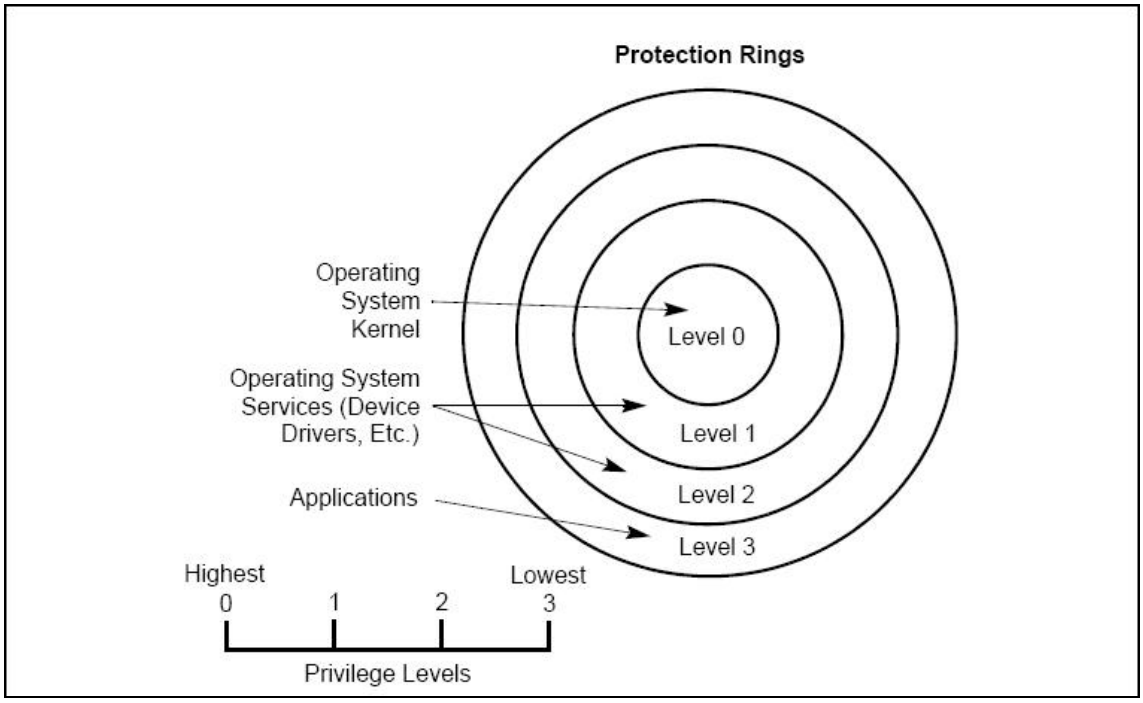


그림 11. Protection Rings

그림 11에서 볼 수 있듯이 x86은 총 4개의 Ring을 이용하여 Access Control을 지원합니다.
실제 이 Ring의 구분은 숫자 0~3으로 구분되며 물리적으로 Ring 모양과는 전혀 상관이 없습니다.
네 개의 Ring 중에서 Ring 0가 가장 큰 권한을, Ring 3이 가장 낮은 권한을 가지게 됩니다.
4개의 Ring 중 Windows는 Ring 0 와 Ring 3의 두 개만을 사용하며 Ring 0를 커널모드, Ring 3을 유저모드라고 부릅니다. 즉 Windows의 모든 커널 코드들은 Ring 0의 권한을 가지고 실행되는 것입니다.

Ring 0와 Ring 3에서 실행되는 프로그램들의 접근 제한 규칙은 아래와 같습니다.

> • 특권 수준이 낮은 코드 세그먼트가 특권 수준이 높은 데이터세그먼트로 액세스하는 것은 불가능
> • 특권 수준이 낮은 코드 세그먼트에서 특권 수준이 높은 코드 세그먼트로의 제어 이행은 특별한 방법을 사용해서만이 가능
> • 특권 수준이 높은 코드세그먼트에서 특권 수준이 낮은 코드 세그먼트로의 제어 이행은 불가능
>
> - Windows 구조와 원리 그리고 Codes, 정덕영 저-

앞에서도 얘기했지만 실제 Ring 을 구분하는 것은 0~3 까지의 숫자입니다. 현재 프로그램을 실행하는 특권 레벨을 결정하는 것은 **코드 세그먼트**의 특권 레벨입니다. X86 보호모드에서 실행되는 프로그램의 각 코드 세그먼트는 **세그먼트 디스크립터**라고 불리는 8바이트의 데이터 구조체에 의해 기술됩니다.

세그먼트 디스크립터는 아래와 같은 구조를 가지고 있습니다.

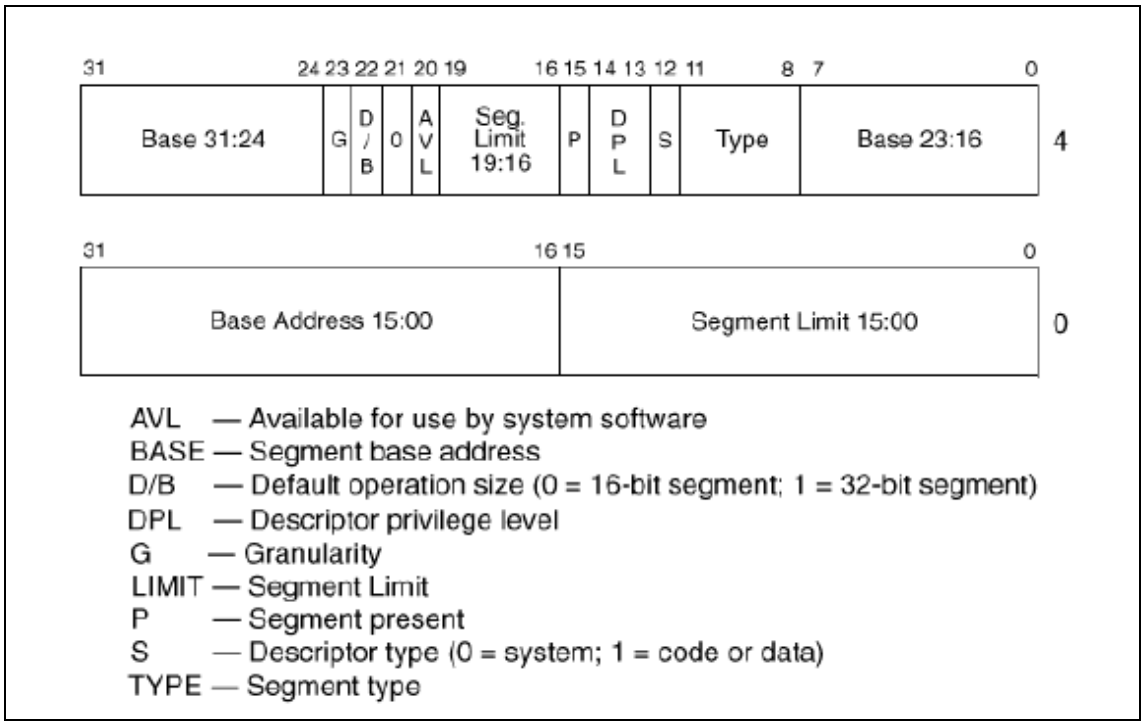


그림 12. Segment Descriptor

-출처 : University of Nebraska-Lincoin Computer Science & Engineering CSCE 351-

그림 12에서 DPL(Descriptor Privilege Level) 부분이 바로 특권 레벨을 나타내는 부분입니다. 운영체제는 해당 필드를 참고하여 커널모드, 유저모드를 구분하게 됩니다. DPL 중 현재 실행되고 있는 코드의 DPL 값을 CPL(Current Privilege Level)이라고 부릅니다.

세그먼트 디스크립터는 메인 메모리에 저장되며 이 디스크럽터들을 디스크럽터 테이블이 관리하게 됩니다. 메인 메모리에는 두 개의 중요한 테이블이 있는데 그것이 바로 앞 절에서 살펴본 GDT 와 LDT d입니다. 즉 GDT와 LDT에 세그먼트 디스크럽터에 대한 정보가 있습니다. GDT, LDT 의 두 개의 테이블이 있기 때문에 어떤 테이블에서 찾아야 할 것인지를 알 필요가 있습니다. 이를 위해 세그먼트 셀렉터(Segment Selector)라고 하는 것이 존재합니다만 이에 대한 설명은 넘어가도록 하겠습니다.

실제 GDT는 어떤 값을 가지고 있는지 WinDBG에서 확인을 해보도록 합시다.

Dump를 위해 ProtMode[1] 라는 WinDBG 확장 dll 파일을 이용했습니다.

ProtMode.dll 파일을 C:₩Program Files₩Debugging Tools for Windows₩winxp 에 복사합니다.(Debugging Tools for Windows 가 설치되어 있어야 합니다.)

WinDBG에서 먼저 ntsdexts.dll 파일을 load 한 후 ProtMode.dll을 load 합니다.

그리고 !ProtMode.Descriptor GDT 1 를 입력하면 GDT 를 확인할 수 있습니다.

[1] ProtMode : http://www.codeguru.com/dbfiles/get_file/ProtMode.zip

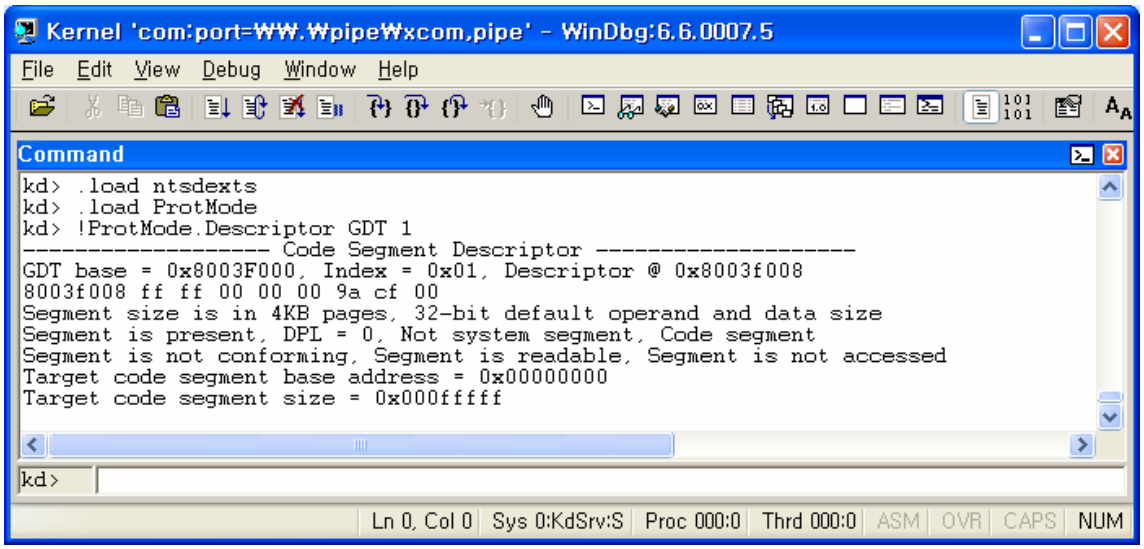


그림 13. Dump GDT

그림 13은 GDT를 WinDBG에서 확인한 모습입니다. 커널모드에서 작동하는 프로그램의 경우에는 DPL 부분이 0, 유저모드에서 작동하는 프로그램의 경우에는 DPL 부분이 3으로 표시됩니다. 저의 경우는 VMWARE를 이용한 커널 디버깅 모드로 동작 중이기 때문에 DPL이 0으로 되어 있습니다.

Ring에 의한 메모리 접근 제한과 더불어 x86에서 제공하는 다른 하나의 보안 장치가 있습니다. 몇몇 명령어들의 경우 특권을 가지고 있는 것으로 간주되어 오직 Ring 0에서만 실행됩니다. 이 명령들은 일반적으로 CPU의 행동을 수정하거나 직접 하드웨어에 접근하기 위해 사용됩니다. 그 명령어들 중 일부는 아래와 같습니다.

- cli : 현재 CPU에서 실행 중인 인터럽트를 중지시킵니다.
- sti : 현재 CPU에서 인터럽트를 실행시킵니다.
- in : 하드웨어 포트로부터 데이터를 읽어들입니다.
- out : 하드웨어 포트로 데이터를 씁니다.

위와 같은 명령어들이 있다 정도만 알아두시면 될 듯합니다.
디바이스 드라이버를 설명하기 위해서 필수적인 Ring Model에 대한 설명이 여기까지입니다.

디바이스 드라이버란 위에서 설명한 유저모드 - 커널모드 스위칭을 하지 않고 직접적으로 하드웨어에 접근하는 소프트웨어라고 말할 수 있습니다. 즉 커널모드에서 동작하며 하드웨어에 직접 접근할 수 있는 프로그램입니다.
윈도우는 이런 디바이스 드라이버를 설계하기 쉽도록 하나의 모델을 제시하고 있는데 이것이 WDM(Windows Driver Model)이라고 합니다. 다만 모든 운영체제가 WDM 방식을 지원하는 것은 아니

며 모든 드라이버가 WDM 방식을 사용해서 개발할 수 있는 것도 아닙니다.
WDM 의 역사, 문제, 특징 등과 같은 자세한 내용은 디바이스 드라이버 서적을 참고하시기 바랍니다.

디바이스 드라이버 프로그램은 실행 프로그램이 아닙니다. 즉 확장자가 exe나 dll이 아니라 sys입니다.
개발된 디바이스 드라이버 프로그램을 사용하기 위해서는 해당 드라이버를 LOAD 시켜줘야만 합니다.
리눅스에서의 모듈이라고 생각하시면 훨씬 더 이해가 쉬울 듯 합니다.

디바이스 드라이버의 기본 골격은 아래의 4가지 엔트리로 구성됩니다.

- DriverEntry Routine
- AddDevice Routine
- IRP Dispatch Routine
- DriverUnload Routine

- 디바이스 드라이버 구조와 원리 그리고 제작 노하우, 이봉석 -

간단히 정말 간단히 필요한 것들만 설명하겠습니다. 디바이스 드라이버 전문 서적 참고를 권고합니다. :)

* 디바이스 드라이버 코드를 예시함에 있어서 커널모드 드라이버 형태 또는 WDM 형태로 구현될 수 있으며 이 차이점에 대해서는 상세히 기술하지 않았습니다. 표현의 차이만 있을 뿐 원리는 동일합니다. 디바이스 드라이버 분야는 넓고도 방대해서 일일이 설명하려면 무지막지한 시간이 걸리는 관계로 (본인도 잘 모른다는 게 큰 이유입니다 -_-;) 대충 대충 넘어가겠습니다.

- **DriverEntry Routine**

일반 C 프로그램의 시작점은 main(), 윈도우 응용 프로그램은 WinMain(), dll 프로그램은 DllMain() 그리고 디바이스 드라이버의 시작점은 DriverEntry입니다. 즉 드라이버가 로드된 후 처음으로 실행되는 곳이 바로 이 부분입니다. 이 부분은 처음 드라이버가 로드될 때 단 한번만 수행하는 부분이므로 주로 초기화 과정에 많이 사용됩니다.

- **AddDevice Routine**

자신이 만든 새로운 디바이스를 추가하고자 할 때 사용되는 부분입니다. 이 부분에서 Device Object를 생성하게 됩니다.

- **IRP Dispatch Routine**

IRP는 디바이스와 IO Manager 사이에서 명령을 전달하는 역할을 하는 구조체이며 그 실체는 하나의 버퍼에 불과합니다. 유저모드 프로그램에서 파일을 열고 쓰는 동작 같은 것이 IRP를 통해서 이루어지게 됩니다.

• DriverUnload Routine

이름 그대로 드라이버를 언로드 할 때 수행되는 부분이며 이것저것 수행한 것들을 깨끗하게 정리하는 부분으로 사용됩니다.

먼저 만인의 연인 “Hello World”를 작성해 보도록 하겠습니다.

빌드를 위해서는 소스 파일 외에도 Makefile 과 Source 파일이 필요하게 됩니다.

역시 구체적인 것은 전문 서적을 참고하시고 아래의 화면대로 입력하시기 바랍니다.

```
1  TARGETNAME=HelloWorld # 컴파일 된 파일의 이름
2  TARGETPATH=obj # 현재 디렉토리 하위에 obj 디렉토리를 생성 후 파일 생성
3  TARGETTYPE=DRIVER # 디바이스 드라이버 타입 중 Driver로 설정
4
5  # 필요한 라이브러리 파일 include 시킴
6  #TARGETLIBS= ₩
7  # $(DDK_LIB_PATH)₩wdmsec.lib₩
8  # $(DDK_LIB_PATH)₩csq.lib
9
10 # 빌드 할 소스 파일의 이름
11 SOURCES=HelloWorld.c
12
```

그림 14. Sources

```
1
2 !INCLUDE $(NTMAKEENV)₩makefile.def
3
```

그림 15. Makefile

```
1  #include "ntddk.h"
2
3  VOID OnUnload(IN PDRIVER_OBJECT DriverOjbect)
4  {
5    DbgPrint("OnUnload called!₩n");
6  }
7
8  NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING theRegistryPath)
9  {
10   DbgPrint("Hello World!₩n");
11   theDriverObject->DriverUnload = OnUnload;
12   return STATUS_SUCCESS;
13 }
14
```

그림 16. HelloWorld.c

위의 세 개 파일을 작성한 다음 DDK 를 설치하면 생기는 Windows XP Checked Build Environment 를 실행한 다음 해당 파일들이 있는 디렉토리로 이동합니다. 그리고 아래 그림 17처럼 컴파일을 하고 나면 HelloWorld.sys 파일이 생성됩니다.

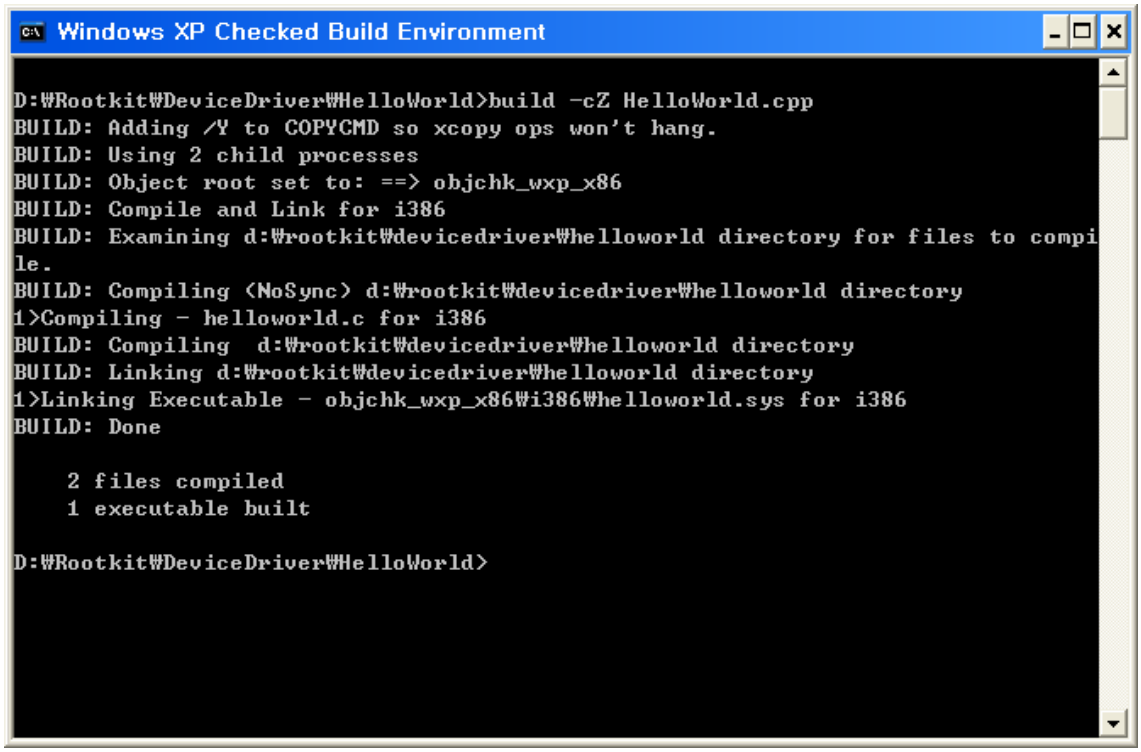


그림 17. build

그림 18. build 결과

이제 작성한 프로그램을 로드하고 메시지를 볼 수 있는 프로그램이 필요합니다. 드라이버를 로드하는 방법은 두 가지가 있는데 그 방법은 역시 서적을 참고하시길 바랍니다.

본 문서에서 드라이버를 로드하는 툴은 Rootkit.com [1] 의 InstDrv를, 메시지를 보기 위해서는 sysinternals[2]의 Debugview 프로그램을 사용했습니다.

[1] http://www.rootkit.com
[2] http://www.sysinternals.com

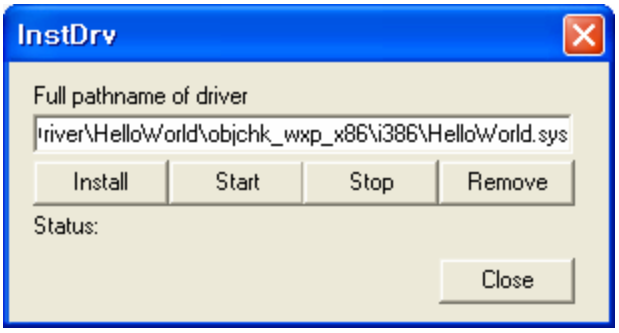


그림 19. 드라이버 Load

Install -> Start -> Stop -> Remove 버튼을 차례로 누르면 DebugView 화면에 HelloWorld.c 파일에서 작성한 DbgPrint 함수가 출력하는 메시지를 확인할 수 있습니다.

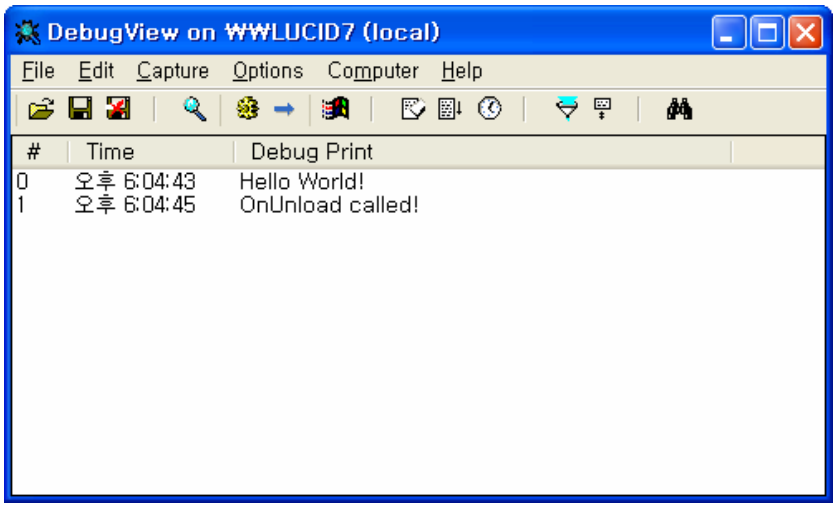


그림 20. Debug Message

HelloWorld.c 파일을 보시면 알겠지만 위에서 구현된 것은 DriverEntry에 진입했을 때와 Unload 시에 대한 것 뿐입니다. 이제 여기에 조금 더 살을 붙여보도록 하겠습니다.

```
TARGETNAME=Minimal # 컴파일 된 파일의 이름
TARGETPATH=obj # 현재 디렉토리 하위에 obj 디렉토리를 생성 후 파일 생성
TARGETTYPE=DRIVER # 디바이스 드라이버 타입 중 Driver로 설정

# 필요한 라이브러리 파일 include 시킴
#TARGETLIBS= ₩
# $(DDK_LIB_PATH)₩wdmsec.lib₩
# $(DDK_LIB_PATH)₩csq.lib

# 빌드 할 소스 파일의 이름
SOURCES=Minimal.cpp
```

그림 21. Source

Source 파일은 TARGETNAME과 SOURCES 부분만 변경되었습니다.

```
#pragma once

extern "C" {
#include <NTDDK.h>
}
```

그림 22. Minimal.h

Minimal.cpp는 조금 더 많은 부분이 추가되었습니다. 하나씩 보겠습니다.

```
#include "Minimal.h"

static NTSTATUS CreateDevice (IN PDRIVER_OBJECT pDriverObject);
static VOID DriverUnload (IN PDRIVER_OBJECT pDriverObject);

extern "C" NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject, IN PUNICODE_STRING pRegistryPath)
{
    NTSTATUS status;
    pDriverObject->DriverUnload = DriverUnload;
    DbgPrint("Driver Entry 진입성공₩n");
    status = CreateDevice(pDriverObject);

    return status;
}
```

그림 23. Minimal.cpp #1

CreateDevice 함수가 하나 더 추가된 것을 볼 수 있습니다. DriverEntry 부분은 CreateDevice를 호출하는 부분이 추가되었습니다.

아래는 CreateDevice 함수입니다.

```
NTSTATUS CreateDevice(IN PDRIVER_OBJECT pDriverObject)
{
    NTSTATUS status;
    PDEVICE_OBJECT pDevObj;

    WCHAR DeviceName[] = L"\\Device\\MINIMAL"; // Device 이름
    WCHAR DeviceLinkName[] = L"\\DosDevices\\MIN"; // Device의 Symbolic Link 이름

    UNICODE_STRING DeviceNameUnicodeString;
    UNICODE_STRING DeviceLinkUnicodeString;

    // 유니코드로 변환
    RtlInitUnicodeString(&DeviceNameUnicodeString, DeviceName);
    RtlInitUnicodeString(&DeviceLinkUnicodeString, DeviceLinkName);

    // Device 생성
    status = IoCreateDevice(pDriverObject, sizeof(DEVICE_EXTENSION), &DeviceNameUnicodeString,
        FILE_DEVICE_UNKNOWN, 0, TRUE, &pDevObj);

    if (!NT_SUCCESS(status)) { // Device 생성에 실패하면
        DbgPrint("CreateDevice Failed!!\n");
        return status;
    }
    else // Device 생성에 성공하면
        DbgPrint("CreateDevice Sucess!\n");

    // Device의 Symbolic Link 생성
    status = IoCreateSymbolicLink(&DeviceLinkUnicodeString, &DeviceNameUnicodeString);

    if (!NT_SUCCESS(status)) { // Symbolic Link 생성에 실패하면
        DbgPrint("IoCreateSymbolicLink Failed!!\n");
        IoDeleteDevice(pDevObj); // 생성한 Device 삭제
        return status;
    }
    else // Symbolic Link 생성에 성공하면
        DbgPrint("IoCreateSymbolicLink Success!!\n");

    return STATUS_SUCCESS;
}
```

그림 24. Minimal.cpp #2

이 부분은 WDM에서는 AddDevice 함수가 처리합니다. 디바이스 드라이버는, 즉 커널 내부에서는 모두 유니코드를 사용합니다. 생성할 디바이스의 이름을 유니코드로 변환하는 부분이 CreateDevice함수의 초반 부분입니다.

중간쯤에 가면 IoCreateDevice 함수를 이용해서 디바이스를 생성하게 됩니다. 이 API의 자세한 설명은 전문서적을 참고하시기 바랍니다.

디바이스 생성에 성공하면 해당 디바이스를 유저모드 어플리케이션에서 이용하기 위해 전역변수를 만들어줘야 합니다. 이를 위해 Symbolic Link를 만들게 되며 이를 IoCreateSymbolicLink API에서 담당하게 됩니다.

만약 Symbolic Link 생성에 실패하게 되면 IoDeleteDevice API를 이용해 생성한 디바이스를 삭제합니다. 이 부분은 왜 이렇게 해야 하는지에 대한 명확한 답변을 해드리기 어려우나 아마도 일종의 안전장치로써 사용되는 것이 아닌가 짐작됩니다.

Symbolic Link를 생성하는 방법에 있어서 WDM 을 이용하게 되면 DriverExtension->AddDevice에서 처리를 하게 됩니다만 해당 내용은 생략하도록 하겠습니다.

CreateDevice 함수가 완료되면 디바이스 드라이버 객체가 생성되고 어플리케이션에서 이를 이용할 수

있는 Symbolic Link 까지 생성되게 됩니다.

```
void DriverUnload (IN PDRIVER_OBJECT pDriverObject) {
    WCHAR DeviceLinkName[] = L"\\??\\MIN"; // Unload 할 Device의 Symbolic Link 이름을 설정
    UNICODE_STRING DeviceLinkUnicodeString;
    RtlInitUnicodeString(&DeviceLinkUnicodeString, DeviceLinkName);

    IoDeleteSymbolicLink(&DeviceLinkUnicodeString); // Device의 Symbolic Link 삭제
    DbgPrint("IoDeleteSymbolicLink Success!!\n");

    IoDeleteDevice(pDriverObject->DeviceObject); // Device 삭제
}
```

그림 25. Minimal.cpp #3

위의 그림 25는 DriverUnload 부분입니다. 드라이버를 Unload할 때 Symbolic Link를 삭제한 후 디바이스를 삭제합니다.

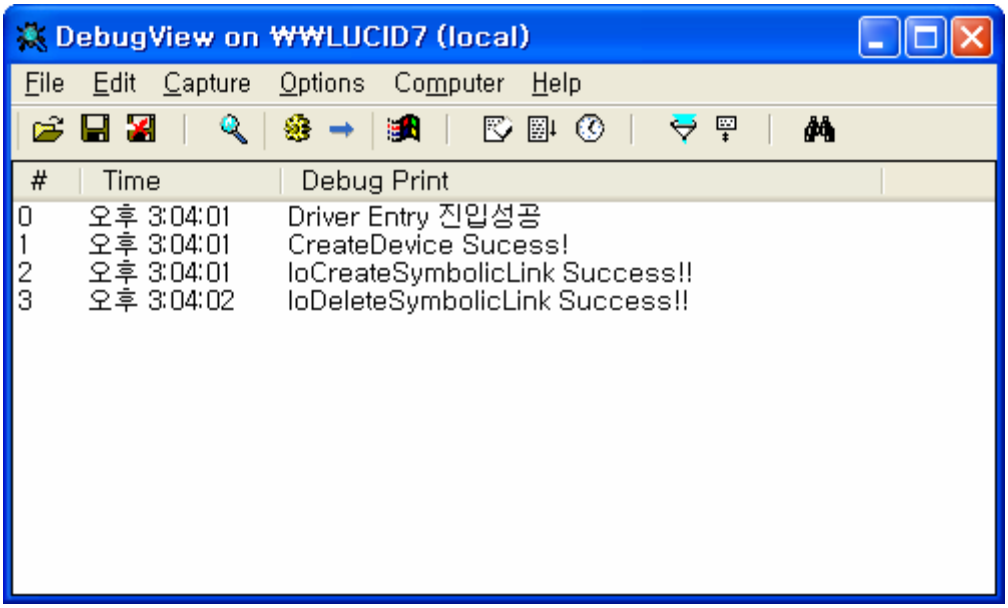


그림 26. Debug Message

작성한 디바이스 드라이버를 설치하면 위의 그림 26과 같은 화면을 볼 수 있습니다.
자 여기까지 오면 이제 우리는 Windows System에 커널모드에서 작동하는 디바이스를 생성할 수 있게 된 것입니다.

다음 예제는 디바이스 익스텐션(Device Extension) 구조체가 추가된 것입니다. 디바이스 익스텐션은 nonpaged 풀에 존재하며 디바이스의 상태정보 등을 기록하는데 쓰입니다. 즉, 디바이스 전용이므로 디바이스 익스텐션의 구조체는 헤더 파일에 정의되어 있어야 합니다.

아래는 디바이스 익스텐션 구조체가 추가된 헤더파일 입니다.

```
#pragma once

extern "C" {
#include <NTDDK.h>
}

typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT pDevice;
    ULONG DeviceNumber;
    UNICODE_STRING ustrDeviceName;  // internal name
    UNICODE_STRING ustrSymLinkName; // external name
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;
```

그림 27. NewMinimal.h

그림 27에서 디바이스 익스텐션 구조체 변수 중 PDEVICE_OBJECT 부분은 반드시 있어야 하는 부분입니다. Linked List에서 Next 포인터의 역할을 한다고 생각하시면 됩니다. 나머지는 드라이버 개발자가 필요한 전용 변수들을 선언하면 됩니다.

```
#include "NewMinimal.h"

static NTSTATUS CreateDevice (IN PDRIVER_OBJECT pDriverObject, IN ULONG ulDeviceNumber);
static VOID DriverUnload (IN PDRIVER_OBJECT pDriverObject);

extern "C" NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
                                 IN PUNICODE_STRING pRegistryPath)
{
    NTSTATUS status;
    ULONG ulDeviceNumber = 0;
    pDriverObject->DriverUnload = DriverUnload;
    DbgPrint("Driver Entry 진입성공₩n");
    status = CreateDevice(pDriverObject, ulDeviceNumber);

    return status;
}
```

그림 28. NewMinimal.cpp #1

NewMinimal.cpp 의 초반부입니다. CreateDevice 함수에 전달되는 변수가 하나 늘었습니다. 바로 ulDeviceNumber 입니다.

하나의 드라이버는 하나 이상의 디바이스를 다룰 수 있습니다. 그러므로 ulDeviceNumber 변수는 드라이버에서 다룰 디바이스를 차례로 참조할 수 있는 매개체로써의 역할을 하게 됩니다.

```
NTSTATUS CreateDevice(IN PDRIVER_OBJECT pDriverObject,
                      IN ULONG ulDeviceNumber)
{
    NTSTATUS status;
    PDEVICE_OBJECT pDevObj;
    PDEVICE_EXTENSION pDevExt;

    UNICODE_STRING DeviceNameUnicodeString;
    UNICODE_STRING DeviceLinkUnicodeString;

    // 유니코드로 변환
    RtlInitUnicodeString(&DeviceNameUnicodeString, L"₩₩Device₩₩NewMINIMAL");
    RtlInitUnicodeString(&DeviceLinkUnicodeString, L"₩₩DosDevices₩₩NewMIN");

    // Device 생성
    status = IoCreateDevice(pDriverObject, sizeof(DEVICE_EXTENSION), &DeviceNameUnicodeString,
        FILE_DEVICE_UNKNOWN, 0, TRUE, &pDevObj);

    if (!NT_SUCCESS(status)) { // Device 생성에 실패하면
        DbgPrint("CreateDevice Failed!!₩n");
        return status;
    }
    else // Device 생성에 성공하면
        DbgPrint("CreateDevice Sucess!₩n");

    // Device Extension 초기화
    pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;

    pDevExt->pDevice = pDevObj; // back pointer
    pDevExt->DeviceNumber = ulDeviceNumber;
    pDevExt->ustrDeviceName = DeviceNameUnicodeString;
    pDevExt->ustrSymLinkName = DeviceLinkUnicodeString;

    // Device의 Symbolic Link 생성
    status = IoCreateSymbolicLink(&DeviceLinkUnicodeString, &DeviceNameUnicodeString);

    if (!NT_SUCCESS(status)) { // Symbolic Link 생성에 실패하면
        if (status == STATUS_OBJECT_NAME_COLLISION) // 만약 이름이 중복되는 경우
            DbgPrint("같은 이름을 가진 SymbolicLink 존재!₩n");
        else
            DbgPrint("IoCreateSymbolicLink Failed!!₩n");
        IoDeleteDevice(pDevObj); // 생성한 Device 삭제
        return status;
    }
    else // Symbolic Link 생성에 성공하면
        DbgPrint("IoCreateSymbolicLink Success!!₩n");

    return STATUS_SUCCESS;
}
```

그림 29. NewMinimal.cpp #2

CreateDevice 에서는 우선 PDEVICE_EXTENSION형 pDevExt 변수가 생긴 것이 눈에 들어옵니다. 중간쯤에 디바이스 익스텐션 구조체를 초기화합니다. 디바이스의 정보 중 Device Name, Symbolic Link Name 등을 저장하고 있습니다.

추가로 status 의 값에 따른 조건문 하나를 넣어봤습니다. 동일한 이름의 Symbolic Link가 존재할 경우 메시지를 뿌립니다. NTSTATUS형이 반환하는 값은 여러 가지가 정의되어 있는데 자세한 것은 C:₩WINDDK₩3790.1830₩inc₩ddk₩wxp₩ntstatus.h 파일을 참고하시기 바랍니다.

```
void DriverUnload (IN PDRIVER_OBJECT pDriverObject) {
    PDEVICE_OBJECT  pNextObj;
    pNextObj = pDriverObject->DeviceObject;

    while (pNextObj != NULL) {
        PDEVICE_EXTENSION pDevExt = (PDEVICE_EXTENSION)pNextObj->DeviceExtension;

        // DevExt also holds the symbolic link name
        UNICODE_STRING pLinkName =  pDevExt->ustrSymLinkName;
        IoDeleteSymbolicLink(&pLinkName);
        DbgPrint("IoDeleteSymbolicLink Success!!₩n");

        pNextObj = pNextObj->NextDevice;
        IoDeleteDevice( pDevExt->pDevice );
        DbgPrint("IoDeleteDevice Success!!₩n");
    }
}
```

그림 30. NewMinimal.cpp #3

DriverUnload 부분은 형태가 조금 틀려졌습니다. pNextObj가 마지막 노드에 도달할 때까지, 즉 생성된 모든 디바이스에 대하여 차례로 Symbolic Link와 Device를 삭제합니다.(Linked List라고 생각하시면 됩니다.) Symbolic Link Name은 디바이스 익스텐션에 저장되어 있는 값을 사용하고 있습니다.

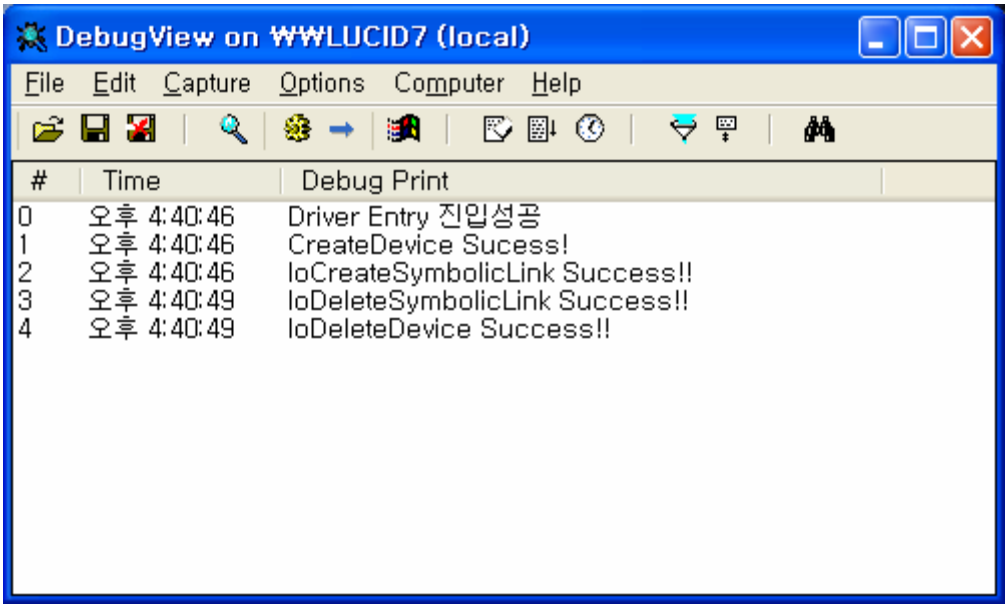


그림 31. Debug Message

DbgView 화면입니다.

이번에는 마지막으로 디스패치 루틴을 넣어보겠습니다. 앞에서 이야기했던 디바이스 드라이버의 기본 골격은 이 디스패치 루틴이 들어감으로써 완성되게 됩니다.

```
#pragma once

extern "C" {
#include <NTDDK.h>
}

typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT pDevice;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

#define DeviceName          L"\\Device\\LastMINIMAL"
#define DeviceSymbolicName  L"\\DosDevices\\LastMIN"

static VOID DriverUnload(IN PDRIVER_OBJECT  pDriverObject);
NTSTATUS DispatchDummy(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);
```

그림 32. LastMinimal.h

헤더파일 입니다. 앞에서 지저분한 소스를 좀 고칠 겸 이쪽으로 몇 개 뺐습니다.

그림 32에서 볼 수 있듯이 디바이스 익스텐션 구조체를 간단하게 수정했고 DispatchDummy라는 함수가 추가되었습니다.

```
#include "LastMinimal.h"

extern "C" NTSTATUS DriverEntry (IN PDRIVER_OBJECT pDriverObject,
                                 IN PUNICODE_STRING pRegistryPath)
{
    NTSTATUS status;
    PDEVICE_OBJECT pDevObj; // Driver Object
    PDEVICE_EXTENSION pDevExt; // Device Extension

    UNICODE_STRING DeviceNameUnicodeString;
    UNICODE_STRING DeviceLinkUnicodeString;

    // 유니코드로 변환
    RtlInitUnicodeString(&DeviceNameUnicodeString, DeviceName);
    RtlInitUnicodeString(&DeviceLinkUnicodeString, DeviceSymbolicName);


    status = IoCreateDevice(pDriverObject, sizeof(DEVICE_EXTENSION), &DeviceNameUnicodeString,
        FILE_DEVICE_UNKNOWN, 0, TRUE, &pDevObj);

    if (!NT_SUCCESS(status)) { // Device 생성에 실패하면
        DbgPrint("CreateDevice Failed!!₩n");
        return status;
    }

    // Device Extension 초기화
    pDevExt = (PDEVICE_EXTENSION)pDevObj->DeviceExtension;
    RtlZeroMemory(pDevExt, sizeof(DEVICE_EXTENSION));
    pDevExt->pDevice = pDevObj;

    // Device의 Symbolic Link 생성
    status = IoCreateSymbolicLink(&DeviceLinkUnicodeString, &DeviceNameUnicodeString);

    if (!NT_SUCCESS(status)) { // Symbolic Link 생성에 실패하면
        DbgPrint("IoCreateSymbolicLink Failed!!₩n");
        IoDeleteDevice(pDevObj); // 생성한 Device 삭제
        return status;
    }

    // Dispatch 초기화
    for (int i = 0; i <= IRP_MJ_MAXIMUM_FUNCTION; i++) {
        pDriverObject->MajorFunction[i] = DispatchDummy;
    }
    pDriverObject->DriverUnload = DriverUnload;

    return status;
}
```

그림 33. LastMinimal.cpp #1

그림 33은 DriverEntry 부분입니다. 기존 소스에서 따로 분리해 놓았던 CreateDevice를 합쳐놓았습니다.

디바이스 익스텐션 초기화가 간략하게 된 것을 중간 쯤에서 확인할 수 있습니다.

마지막 부분에 Dispatch 초기화 부분이 새로이 추가된 것이 보입니다.

MajorFunction 테이블의 모든 디스패치 루틴에 대해 DispatchDummy 함수를 지정하고 있습니다.

MajorFunction 테이블은 특정 I/O 요청에 대한 디스패치 루틴의 함수 포인터들을 저장하고 있으며 이런 I/O 요청은 IRP_MJ_XXX 형태의 심볼로서 정의되어 있습니다. 실제로는 상수 값을 가지고 있는 심볼입니다. Ntddk.h 또는 WDM.h 파일을 살펴보시면 상세히 기술되어 있습니다.

```
#define IRP_MJ_CREATE                   0x00
#define IRP_MJ_CREATE_NAMED_PIPE        0x01
#define IRP_MJ_CLOSE                    0x02
#define IRP_MJ_READ                     0x03
#define IRP_MJ_WRITE                    0x04
#define IRP_MJ_QUERY_INFORMATION        0x05
#define IRP_MJ_SET_INFORMATION          0x06
#define IRP_MJ_QUERY_EA                 0x07
#define IRP_MJ_SET_EA                   0x08
#define IRP_MJ_FLUSH_BUFFERS            0x09
#define IRP_MJ_QUERY_VOLUME_INFORMATION 0x0a
#define IRP_MJ_SET_VOLUME_INFORMATION   0x0b
#define IRP_MJ_DIRECTORY_CONTROL        0x0c
#define IRP_MJ_FILE_SYSTEM_CONTROL      0x0d
#define IRP_MJ_DEVICE_CONTROL           0x0e
#define IRP_MJ_INTERNAL_DEVICE_CONTROL  0x0f
#define IRP_MJ_SHUTDOWN                 0x10
#define IRP_MJ_LOCK_CONTROL             0x11
#define IRP_MJ_CLEANUP                  0x12
#define IRP_MJ_CREATE_MAILSLOT          0x13
#define IRP_MJ_QUERY_SECURITY           0x14
#define IRP_MJ_SET_SECURITY             0x15
#define IRP_MJ_POWER                    0x16
#define IRP_MJ_SYSTEM_CONTROL           0x17
#define IRP_MJ_DEVICE_CHANGE            0x18
#define IRP_MJ_QUERY_QUOTA              0x19
#define IRP_MJ_SET_QUOTA                0x1a
#define IRP_MJ_PNP                      0x1b
#define IRP_MJ_PNP_POWER                IRP_MJ_PNP      // Obsolete....
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
```

그림 34. ntddk.h

IRP 함수 코드 값 중 몇 가지만 나열해보면 아래와 같습니다.

- IRP_MJ_CREATE : 핸들을 요청합니다. CreateFile에 대응됩니다.
- IRP_MJ_CLENUP : 핸들을 닫을 때 지연된 IRP를 취소시킵니다. CloseHandle에 대응됩니다.
- IRP_MJ_CLOSE : 핸들을 닫습니다. CloseHandle에 대응됩니다.
- IRP_MJ_READ : 디바이스로부터 데이터를 받습니다. ReadFile에 대응됩니다.
- IRP_MJ_WRITE : 디바이스로 데이터를 보냅니다. WriteFile에 대응됩니다.
- IRP_MJ_SHUTDOWN : 시스템이 셧다운될 때 사용됩니다. InitateSystemShutdown에 대응됩니다.

- 출처 : 원리와 예제로 배워보는 Windows 2000 디바이스 드라이버 -

```
// 아무 일도 하지 않는 Dummy용 함수
NTSTATUS DispatchDummy(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp) {
    Irp->IoStatus.Status = STATUS_SUCCESS;
    IoCompleteRequest(Irp,IO_NO_INCREMENT);

    return Irp->IoStatus.Status;
}

void DriverUnload(IN PDRIVER_OBJECT pDriverObject) {

    UNICODE_STRING  DeviceSymbolName;

    RtlInitUnicodeString(&DeviceSymbolName , DeviceSymbolicName);
    IoDeleteSymbolicLink(&DeviceSymbolName);
    IoDeleteDevice(pDriverObject -> DeviceObject);
}
```

그림 35. LastMinimal.cpp #2

그림 35의 주석에도 달려있지만 실제 DispatchDummy 함수는 아무 일도 하지 않는 함수입니다. 해당 함수가 호출되면 status를 STATUS_SUCCESS로 대입하고 IoCompleteRequest API를 호출한 뒤 status를 return 합니다. IoCommpleteRequest API는 이름에서도 알 수 있듯이 해당 요청이 완료되었음을 알리는 역할을 합니다.

마지막으로 DriverUnload 부분은 Symbolic Link 이름을 지정하는 부분만 추가되었습니다.

이것으로써 디바이스 드라이버의 기본 골격은 다 다루어보았습니다. 디바이스 드라이버 항목의 마지막 예제로써 디바이스 하나를 생성하여 읽고 쓰는 작업을 하는 드라이버를 소개할까 했으나 실제 우리가 할 SSDT Hooking 시에 디스패치 루틴은 사용되지 않으므로 패~쓰 하도록 하겠습니다.

5. SSDT Hook

멀고 먼 길을 돌아 드디어 SSDT Hook으로 다시 진입하게 되었습니다. 자 이제 실제 코드를 작성하기 전에 관련 내용들을 알아보도록 하겠습니다.

System Service Dispatcher, 즉 KiSystemService는 SSDT(System Service Dispatch Table)을 참고하여 적절한 System Service를 실행하게 됩니다. 하지만 이 SSDT, 즉 KiServiceTable은 커널에 의해 익스포트 되지 않은 구조체이므로 원칙적으로는 내부 구조를 알 수 없습니다.

하지만 KeServiceDescriptorTable이라는 ntoskrnl.exe에 의해 익스포트 된 구조체(Service Descriptor Table) 내에 KiServiceTable에 대한 정보가 들어 있습니다.

KeServiceDescriptorTable 구조체는 아래와 같습니다.

```
typedef struct ServiceDescriptorTable {
        SDE ServiceDescriptor[4];
} SDT;
```

SDT 구조체는 아래와 같습니다.

```
typedef struct ServiceDescriptorEntry {
        PDWORD KiServiceTable;
        PDWORD CounterTableBase;
        DWORD ServiceLimit;
        PBYTE ArgumentTable;
} SDE;
```

KiServiceTable 에는 서비스 테이블의 시작 주소가 들어있습니다. 이 주소로부터 4바이트 단위로 각 System Service들의 주소가 들어가있습니다.

세 번째 필드 ServiceLimit는 SDT에 등록되어 있는 System Service의 개수가 들어있습니다. 마지막 필드인 ArgumentTable에는 각 System Service마다 전달되는 파라미터가 몇 바이트인지 기록되어 있습니다. 이 주소로부터 1바이트 단위로 전달되는 파라미터의 바이트 수가 기록되어 있습니다.

(이를 SSPT-System Service Parameter Table라고 합니다.)

아래는 KeServiceDescriptorTable을 WinDBG에서 Dump한 화면입니다.

```
kd> d KeServiceDescriptorTable
80554180  30 30 50 80 00 00 00 00-1c 01 00 00 a4 34 50 80  00P.........4P.
80554190  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
805541a0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
805541b0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
805541c0  10 27 00 00 87 db 80 bf-00 00 00 00 00 00 00 00  .'..............
805541d0  80 4a 1b f8 38 f5 bd 81-90 1a bd 81 40 1f 6e 80  .J..8.......@.n.
805541e0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
805541f0  c0 89 35 12 92 13 c7 01-00 00 00 00 00 00 00 00  ..5.............
```

그림 36. KeServiceDescriptorTable Dump(d)

그림 36에서 4바이트마다 SDE 구조체의 구성 요소를 나타냅니다. 처음 4바이트 0x80503030이 바로 ServiceDescriptor[0]의 내용을 나타내며 이 주소는 KiServiceTable의 주소를 가리키고 있습니다.
다음 화면부터는 편의를 위해 바이트 순서가 읽기 쉽게 표시되도록 하였습니다.(dd 명령어를 사용하면 됩니다. 다시 원래의 화면으로 보시길 원한다면 db 명령어를 사용하면 됩니다.)

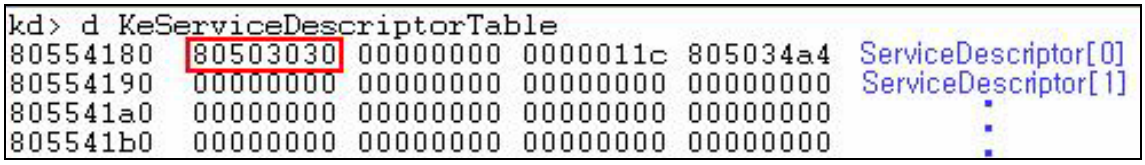


그림 37. KeServiceDescriptorTable Dump(dd)

그림 37에서 각 줄은 SDE 구조체 배열의 구성요소들입니다. 그리고 빨간색으로 표시된 부분이 바로 KiServiceTable의 주소입니다.
WinDBG로 해당 주소의 메모리를 덤프하면 아래와 같은 화면이 나옵니다

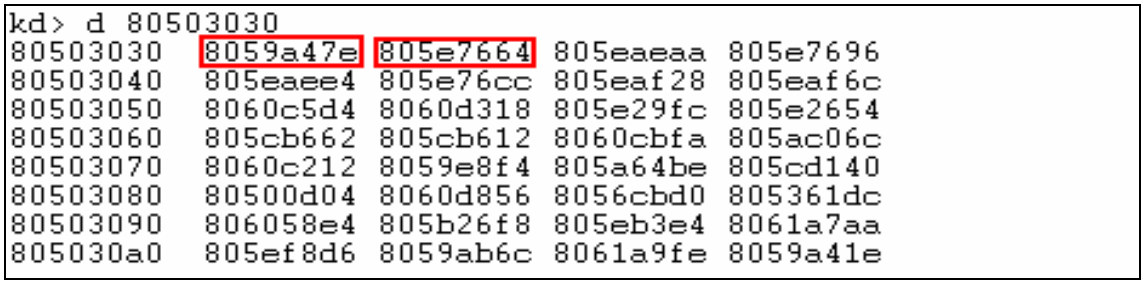


그림 38. Dump Address of KiServiceTable

그림 38에서 빨간색으로 표시한 부분처럼 각 부분은 KiServiceTable에 저장된 System Service들의 주소를 나타냅니다. 0x8059a47e 주소의 메모리를 덤프하면 아래와 같습니다.

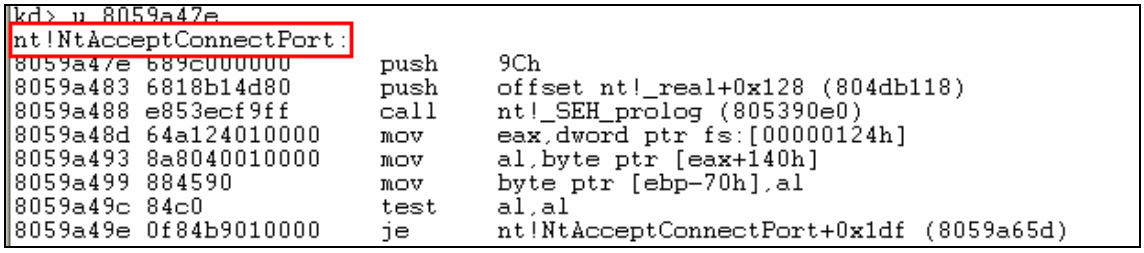


그림 39. Dump Address of NtAcceptConnectPort

첫 번째 줄을 살펴보면 NtAcceptConnectPort 라는 API 임을 확인할 수 있습니다. 동일한 방법을 이용하여 ServiceLimit 개의 System Service를 확인할 수 있습니다.

자 다시 처음으로 돌아가서 위에서 확인한 NtAcceptConnectPort Api로 넘겨지는 파라미터의 크기는 아래의 그림 40에서 빨간색 상자로 표시된 주소에 저장되어 있습니다.

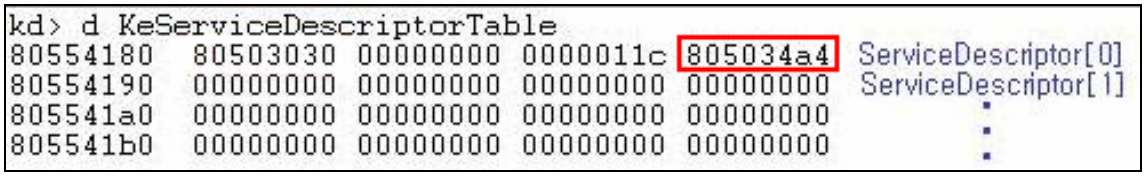


그림 40. KeServiceDescriptorTable Dump(dd)

해당 주소를 덤프하면 그림 41과 같은 화면이 보이는데 그 중 가장 첫 번째 바이트가 파라미터의 크기를 나타내게 됩니다. 여기에서 NtAcceptConnectPort API의 파라미터 크기는 0x18 바이트가 됩니다.

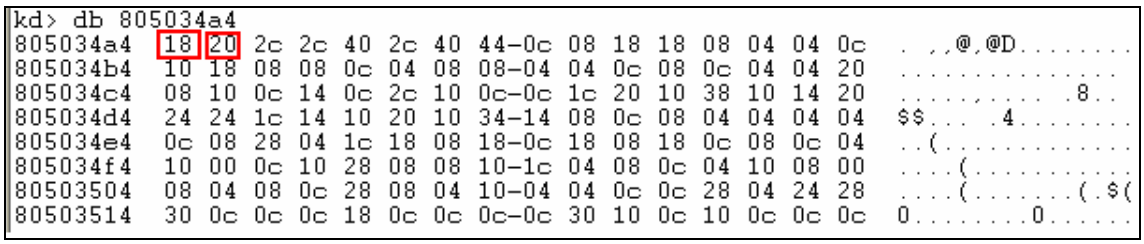


그림 41. Dump Address of SSPT

보통 하나의 파라미터는 4바이트를 차지하기 때문에 NtAcceptConnectPort API는 총 6개의 파라미터를 가짐을 추측할 수 있습니다. 위의 그림에서 두 번째 빨간색 상자 안의 숫자 0x20은 NtAcceptConnectPort 다음에 SDT에 저장되어 있는 API의 파라미터 크기를 나타냅니다. 즉 그림 5-3에서 두 번째 4바이트 805e7664 주소에 저장되어 있는 API의 파라미터 크기를 나타냅니다.

이와 같은 방법을 이용하여 KeServiceDescriptorTable 구조체를 통해 KiServiceTable에 설정되어 있는 모든 API를 살펴볼 수 있습니다.

만약 특정 API를 찾고 싶은 경우 일일이 서비스 테이블 주소를 찍어보는 것은 매우 많은 시간이 요구됩니다. 2장 Native API에서 살펴보았듯이 각 Zw로 시작하는 함수들은 eax 레지스터로 인덱스 번호를 넘기게 됩니다. 그러므로 해당 API가 위치하는 주소는 KeServiceDescriptorTable의 Base 주소(여기서는 0x80503030 입니다)에 인덱스*4 를 더한 값이 됩니다.

아래의 그림 42는 NtCreateFile API를 확인하는 화면입니다.

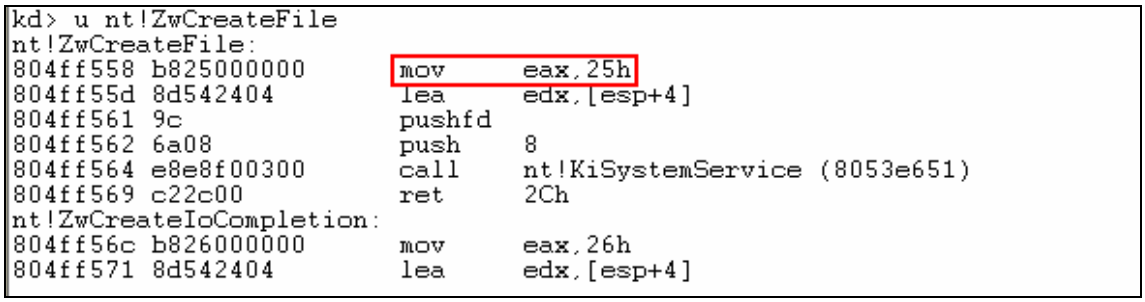


그림 42. 인덱스로 0x25를 넘김

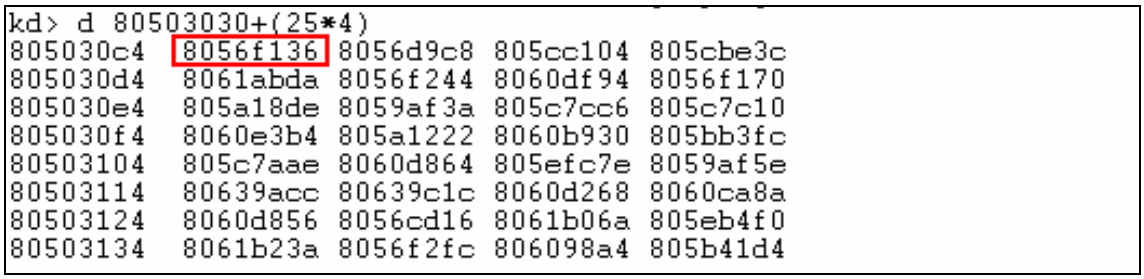


그림 43. 0x8056f136이 NtCreateFile API의 주소

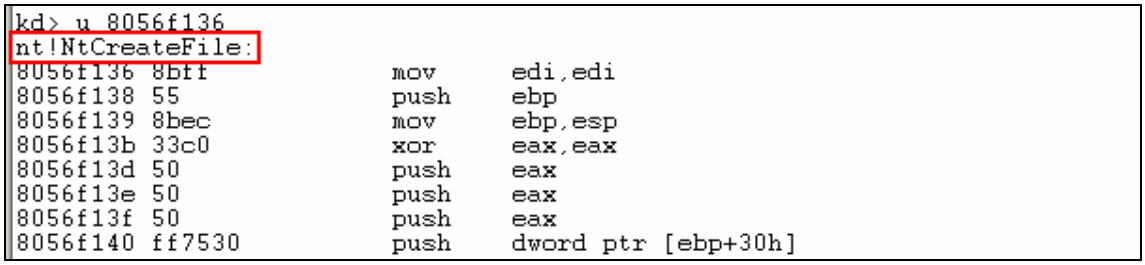


그림 44. NtCreateFile API의 주소임을 확인

추가 사항으로 SDT에는 KeServiceDescriptorTable 외에도 KeServiceDescriptorTableShaodw 라는 배열이 하나 더 포함됩니다. KeServiceDescriptorTable은 ntosrknl에서 구현된 서비스들을, KeServiceDescriptorTableShadow는 Win32k.sys에서 구현된 서비스들(Windows USER, GDI)을 포함하고 있습니다.

이를 그림으로 나타내면 아래와 같습니다.

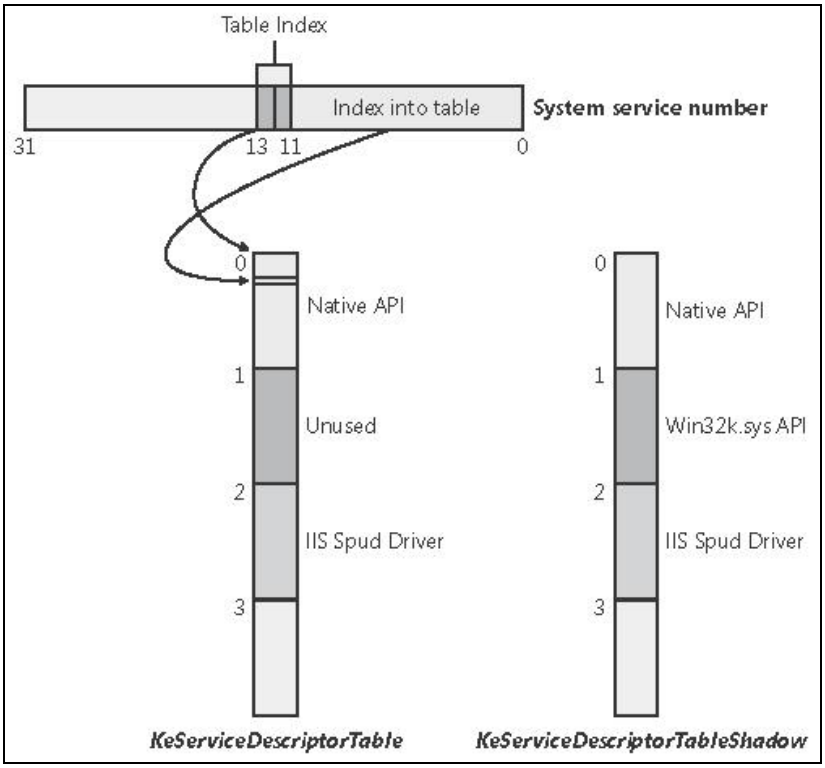


그림 45. System Service Number to system service translation

출처 : Microsoft Windows Internals 4th Edition

자 이제 본격적으로 소스코드를 보도록 하겠습니다.

해당 소스는 somma[1]님 블로그에서 제공된 bkdp3 소스와 HackerDefender[2] SDT Hooking 예제소스를 기반으로 사알짝 수정만 가해진 것임을 미리 알려드립니다.

먼저 디바이스 드라이버 프로그램의 골격을 만듭니다. 위에서 만들었던 LastMinimal을 통째로 복사해 오면 되겠습니다. 드라이버가 로드되고 가장 처음 해야할 일이 무엇일까요? 우리의 목적은 SSDT에 저장되어 있는 API의 주소를 우리가 만든 함수로 바꿔치기하는 것입니다. 즉 먼저 SSDT 에 접근할 수 있어야만 합니다. 그래야 테이블 내 우리가 후킹하기 원하는 API의 주소를 바꿔쓸 수 있기 때문입니다.

5장 처음에서 살펴보았듯이 Kernel에 의해 export 된 KeServiceDescriptorTable 변수를 통해 SSDT에 접근할 수 있으므로 다음과 같이 선언하면 됩니다.

```
#include "SSDTHook.h"

__declspec(dllimport)  SERVICE_DESCRIPTOR_ENTRY KeServiceDescriptorTable;
```

그림 46. KeServiceDescriptorTable 선언

[1] http://somma.egloos.com
[2] http://hxdef.org/

그리고 역시 앞에서 살펴보았던 SSDT 구조체의 원형을 선언해줍니다.

```
#pragma once

extern "C" {
#include <NTDDK.h>
}

typedef struct _DEVICE_EXTENSION {
    PDEVICE_OBJECT pDevice;
} DEVICE_EXTENSION, *PDEVICE_EXTENSION;

#define DeviceName          L"\\Device\\SSDTHook"
#define DeviceSymbolicName  L"\\DosDevices\\SSDT"

static VOID DriverUnload(IN PDRIVER_OBJECT  pDriverObject);
NTSTATUS DispatchDummy(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp);

// Service Descriptor Table
#pragma pack(1) // 1byte alignment 를 위해 pack(1) 지정

typedef struct SERVICE_DESCRIPTOR_ENTRY
{
    unsigned int *ServiceTableBase;
    unsigned int *ServiceCounterTableBase;
    unsigned int NumberOfServices;
    unsigned char *ParamTableBase;
} SERVICE_DESCRIPTOR_ENTRY,*PSERVICE_DESCRIPTOR_ENTRY;

#pragma pack( ) // default alignment로 복귀
```

그림 47. SSDT 구조체 추가

여기까지 오면 우리는 KeServiceDescriptorTable 이라는 변수를 통해 SSDT 구조체까지 접근할 수 있게 됩니다.

어떤 API를 후킹할지는 이제 입맛에 따라 고르기만 하면 됩니다. 쉽게 테스트해 볼 수 있도록 ZwWriteFile을 후킹해 보도록 하겠습니다. 짐작하시다시피 파일 쓰기를 하면 에러 메세지를 뱉어내도록 할 것입니다.

ZwWriteFile이 SSDT에서 몇 번째 테이블에 저장되어 있는지를 먼저 알아봅니다.

위의 그림 43, 44에서 살펴본 것과 동일한 방법으로 찾아보면 NtWriteFile의 주소는 0x805730a0 임을 알 수 있습니다.

이렇게... 무식하게 하는 방법이 하나 있고 좀 더 코드가 우아하게 보이도록 하려면 아래와 같이 정의하여 사용할 수 있습니다.

```
#define SYSTEMSERVICE(_func) \
    KeServiceDescriptorTable.ServiceTableBase[*(PULONG)((PUCHAR)_func+1)]
#define SYSCALL_INDEX(_Function) *(PULONG)((PUCHAR)_Function+1)
```

그림 48. SYSTEMSERVICE 매크로

자 위의 매크로를 한번 살짝 뜯어보겠습니다.

먼저 SYSTEMSERVICE 매크로는 함수 이름을 받아서 해당 함수가 위치하는 메모리의 주소를 반환합니다. 앞에서도 살펴보았듯이 KeServiceDescriptorTable.ServiceTableBase[함수의 인덱스 번호], 이렇게 함수의 주소로 접근할 수 있습니다.(위에서 넘겨주는 함수는 ZwXXX, 리턴값은 NtXXX입니다)
_func로 ZwCreateFile을 넘겨주면 해당 API가 저장되어 있는 주소가 넘겨집니다. 4장에서 작성한 디바이스 드라이버 예제에서 ZwCreateFile을 DbgPrint로 찍어보시면 확인할 수 있습니다.
아래 그림 49는 의심많은 분들을 위한 확인화면 입니다 ㅎㅎ;

그림 49. ZwCreateFile의 주소

[DbgPrint("ZwCreateFile Address : [%x]₩n₩n", ZwCreateFile)]

ZwCreateFile을 찍어보면 0x804ff558이 나옵니다. 해당 메모리의 내용은 아래와 같습니다.

```
kd> u 0x804ff558
nt!ZwCreateFile:
804ff558 b825000000      mov     eax,25h
804ff55d 8d542404        lea     edx,[esp+4]
804ff561 9c              pushfd
804ff562 6a08            push    8
804ff564 e8e8f00300      call    nt!KiSystemService (8053e651)
804ff569 c22c00          ret     2Ch
nt!ZwCreateIoCompletion:
804ff56c b826000000      mov     eax,26h
804ff571 8d542404        lea     edx,[esp+4]
```

그림 50. 0x804ff558 메모리의 내용

위의 그림 50에서 ZwCreateFile이 시작되는 메모리는 move eax, 25h 라는 내용을 포함하고 있는 것을 알 수 있습니다. 여기에서 규칙적인 패턴을 발견할 수 있는데 모든 ZwXXX의 시작은 move eax, index가 된다는 것입니다. 그러므로 매크로에서 ((PUCHAR)ZwWriteFile+1) 이 부분은 ZwWriteFile의 메모리 시작주소의 다음 4바이트, 즉 Index 값을 가리키게 됩니다.(PUCHAR은 unsigned char형입니다)
결국 SYSSTEMSERVICE 매크로는
KeServiceDescriptorTable.ServiceTableBase[*(PULONG)(함수의 인덱스 4바이트)]가 되며 NtXXX 함수가 저장되어 있는 주소를 반환하게 됩니다.
자 그럼 이렇게 가정한 이론이 맞는지 실제로 값을 출력해보겠습니다.

```
DbgPrint("ZwCreateFile Address : [%x]₩n₩n", ZwCreateFile);
DbgPrint("(PUCHAR)ZwCreateFile + 1 Address : [%x]₩n₩n", (PUCHAR)ZwCreateFile+1);
DbgPrint("*(PULONG)(PUCHAR)ZwCreateFile + 1 Address : [%x]₩n₩n",
        *(PULONG)((PUCHAR)ZwCreateFile+1));
```

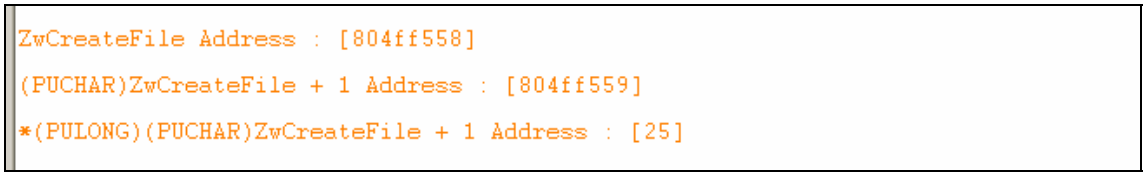


그림 51. 매크로 결과 확인

그림 51의 두 번째 결과 0x804ff559 주소의 내용은 인덱스 번호(0x25)임을 알 수 있습니다.

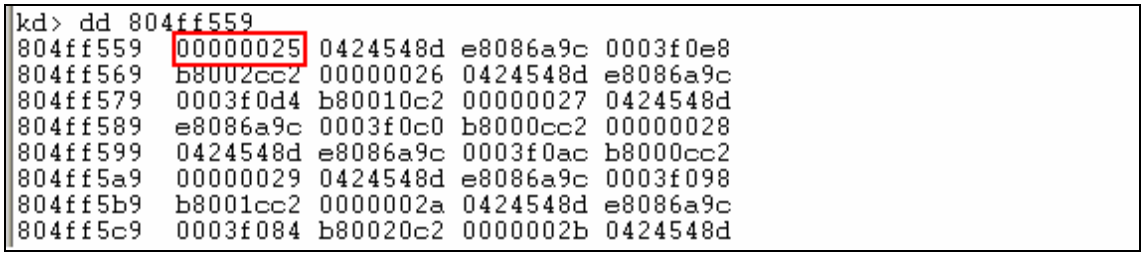


그림 52. 804ff559 메모리 덤프

SYSTEMSERVICE 매크로는 ZwXXXX 함수 주소를 입력받아 NtXXXX 함수의 주소를 반환한다는 것이 증명되었습니다.

다음으로 SYSCALL_INDEX 매크로는 함수 이름을 받아서 해당 함수의 SSDT내 인덱스 번호를 리턴합니다. 그러므로 NtXXXX 함수의 주소를 얻기 위해 아래 그림 53처럼 사용하는 것이 가능합니다.

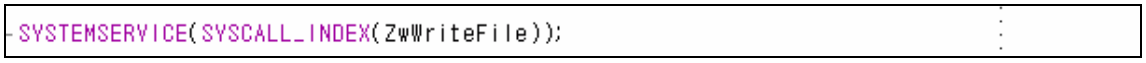


그림 53. SYSTEMSERVICE 매크로 사용법

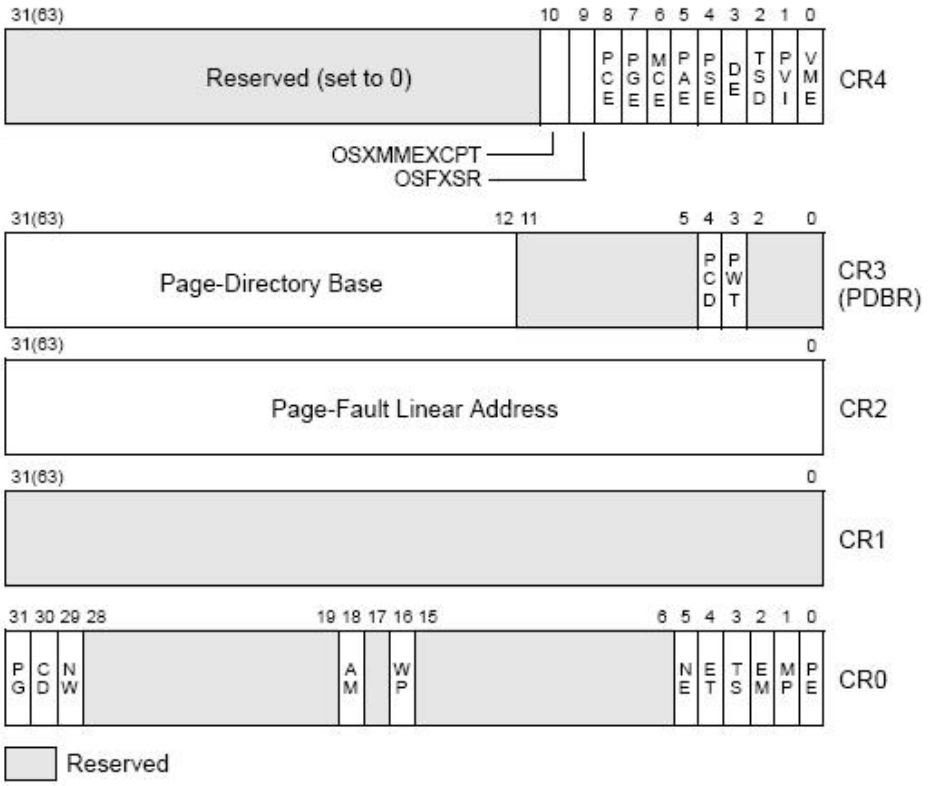
자 이제 우리는 SSDT 안에 저장되어 있는 ZwWrite 의 주소에 접근할 수 있게 되었습니다. 그 주소를 우리가 작성한 API가 저장된 주소로 바꿔치기만 하면 되는데 XP이상에서 SSDT는 Read Only로 되어 있습니다. 이것은 Windows의 메모리 보호를 위해 사용된 기법으로써 Write Protection이라고 합니다. 이 부분을 제거해야만 SSDT에 바꿔칠 주소를 쓰는 것이 가능해집니다.

Write Protection 기법을 우회하는 방법은 여러 가지가 있는데 이 문서에서는 CR0 레지스터를 이용하는 방법과 MDL(Memory Descriptor List)을 이용하는 방법에 대해서만 설명하도록 하겠습니다.

먼저 CR0 레지스터를 이용하는 방법입니다.

CR0에서 Write Protection을 제거하는 방법과 관련한 설명은 somma 님 블로그에 자세히 설명되어 있습니다. 저의 설명이 들어가면 원래의 뜻을 제대로 전달하지 못할 수도 있다는 생각에 해당 부분을 염치없이 죽 긁어왔습니다. somma 님의 양해를 구합니다.

cr0 레지스터를 이용한 Write Protection 제거



[그림 x86 의 Control register]

컨트롤 레지스터는 현재 수행중인 태스크의 특성과 프로세스의 동작모드를 결정 짓는 특별한 레지스터이다. 32 비트와 32 비트 호환 아키텍쳐에서 이 레지스터들은 32 비트이고, 64 비트에서는 64 비트다. mov CRn 인스트럭션으로 이 레지스터들을 건드릴수 있고..

CR3 레지스터는 페이지 디렉토리를 찾아가기 위한 레지스터이고.. 나머지는.. RTFM!
CR0 는 CPU 의 operating mode 와 상태를 제어하는 플래그를 포함하고 있다. 오홋.. ^^
PG, CD 등...중요한 플래그들이 많지만 일단 관심대상은 아니고.. ^^

WP

Write Protect (bit 16 of CR0) — Inhibits supervisor-level procedures from writing into user-level read-only pages when set; allows supervisor-level procedures to write into user-level read-only pages when clear. This flag facilitates implementation of the copy-on-write method of creating a new process (forking) used by operating systems such as UNIX*.

Copy-On-Write 매커니즘과 관련있는 넘이었군..
결국 이 플래그를 조작하면 write protection 속성을 바꿔치기 할 수 있다는 거다.

```
//
// 콘트롤 레지스터 관련 (IA-32 manual vol3, ch 2.5
// CR0 (Control Register Zero) 레지스터의 WP 비트(16)는 쓰기 속성제어에 사용됨
//
#define    CR0_WP_MASK       0x0FFFEFFFF

VOID   ClearWriteProtect(VOID)
{
  __asm
  {
       push   eax;
       mov    eax, cr0;
       and    eax, CR0_WP_MASK;   // WP 클리어
       mov    cr0, eax;
       pop    eax;
   }
}

VOID   SetWriteProtect(VOID)
{
  __asm
  {
       push   eax;
       mov    eax, cr0;
       or     eax, not CR0_WP_MASK;   // WP 비트 세팅
       mov    cr0, eax;
       pop    eax;
   }
}
```

더 자세한 내용은 IA-32 메뉴얼의 4.1 섹션을 참고하면 된다.

요는 페이지 레벨의 프로텍션을 en/disable 하기 위해서는 PDE, PTE 의 플래그와 CR0 의 WP 비트를 조작한단 거다.

- Clear the WP flag in control register CR0.

- Set the read/write (R/W) and user/supervisor (U/S) flags for each page-directory and pagetable entry.

-출처 : 윈도우 쪼물딱거리기 (http://somma.egloos.com/2131561)-

두 번째 방법으로 MDL을 이용하는 방법입니다.

MDL은 Direct I/O를 이용해 buffer에 접근할 때에 사용되는 구조체입니다. 일반적으로 사용자 buffer와 디바이스 간의 데이터를 교환하는 방법에는 Neither I/O, Buffered I/O, Direct I/O의 세가지 방법이 있습니다. Neither I/O는 직접 찾아보시고 Buffered I/O는 I/O Manager가 사용자 buffer와 동일한 buffer를 nonpaged pool에 할당한 후 데이터를 Swap 하는 방법이고 Direct I/O는 중간 단계의 buffer 생성이 생략되는 것이라고 할 수 있습니다. 즉 Buffered I/O에서는 메모리 복사가 일어나고(buffer에 데이터를 swap), Direct I/O에서는 일어나지 않습니다.

드라이버는 사용자 buffer가 page 되지 않도록 lock을 건 후 MDL에 기술된 정보(사용자 buffer의 물리적 주소)를 참고해서 사용자 buffer에 접근하게 됩니다. 그림 54와 55는 두 방식의 차이를 그림으로 표현한 것입니다.

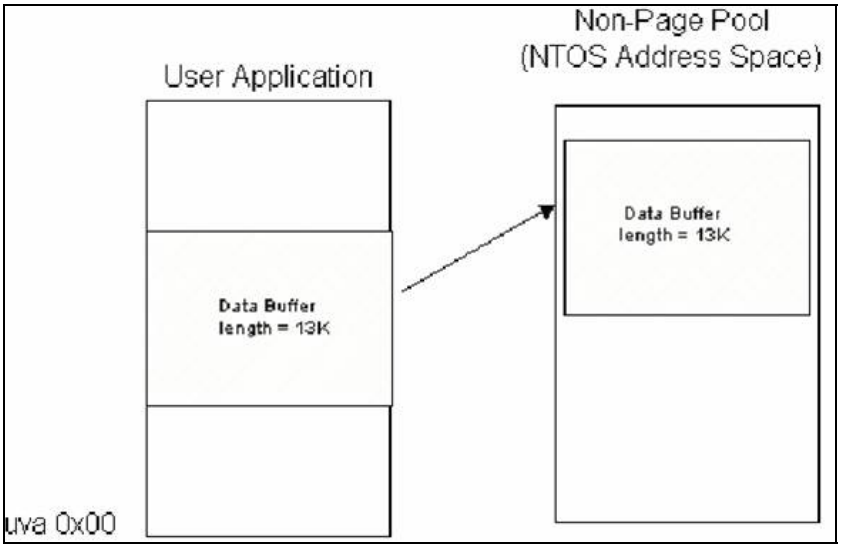


그림 54. Buffered I/O

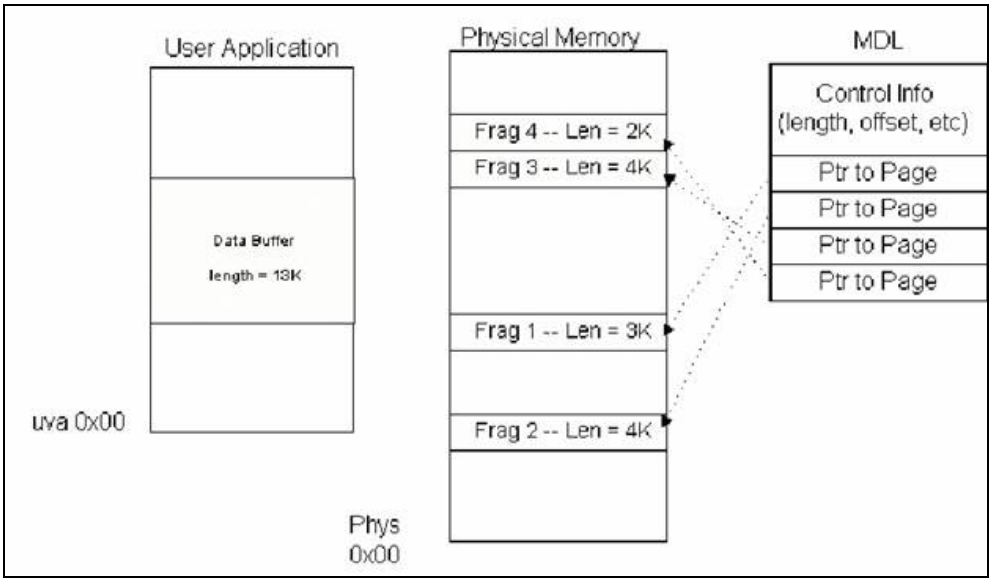


그림 55. Direct I/O

우리가 할 일은 nonpaged pool에 MDL을 생성하여 그것이 SSDT를 가리키게 하는 것입니다. 물론 생성된 MDL은 쓰기 가능하게 설정되어 있으므로 해당 MDL이 가리키는 실제 물리적 메모리 영역, 즉 원래 SSDT에도 쓰는 것이 가능하게 되는 것입니다.

MDL의 구조는 아래 그림 56과 같습니다.

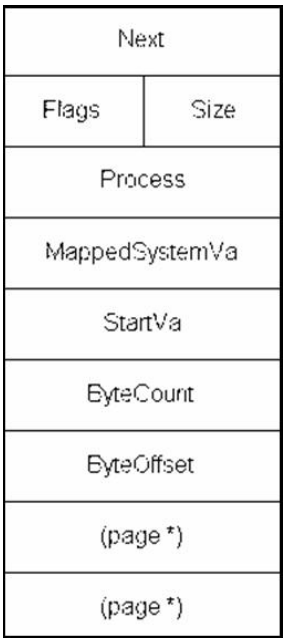


그림 56. MDL 구조

MDL의 구조체는 아래와 같으며 ntddk.h 에 정의되어 있습니다.

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

그림 57. MDL 구조체의 원형

MDL을 이용하여 SSDT 의 Write-Protection을 제거하는 코드는 아래와 같습니다.

```
PMDL g_pmdlSystemCall;
PVOID *MappedSystemCallTable;

// Map the memory into our domain so we can change the permissions on the MDL
g_pmdlSystemCall = MmCreateMdl(NULL, KeServiceDescriptorTable.ServiceTableBase,
                               KeServiceDescriptorTable.NumberOfServices*4);

if(!g_pmdlSystemCall)
    return STATUS_UNSUCCESSFUL;
MmBuildMdlForNonPagedPool(g_pmdlSystemCall);

// Change the flags of the MDL
g_pmdlSystemCall->MdlFlags = g_pmdlSystemCall->MdlFlags | MDL_MAPPED_TO_SYSTEM_VA;

MappedSystemCallTable = MmMapLockedPages(g_pmdlSystemCall, KernelMode);
```

그림 58. MDL을 이용한 Write Protection 제거

위의 소스를 살펴보면 먼저 MDL 구조체 포인터 변수를 선언한 뒤(g_pmdSystemCall) 새로운 MDL을 만들어서 가리키게 합니다.

MmCreateMdl[1] 이라는 API는 이름에서도 알 수 있듯이 MDL을 생성하는 기능을 합니다.

MSDN에서 살펴보면 해당 API는 없어지고 기존에 제작된 Driver 바이너리를 위해 지원 용도로만 사용되고 있으므로 IoAllocateMdl[2] 라는 API를 사용할 것을 권장하고 있습니다.

그러나! 귀찮기 때문에 소스에서는 그냥 MmCreateMdl을 사용하도록 하겠습니다. IoAllocateMdl의 사용 예제는 여기[3]에서 확인하실 수 있습니다.

다시 소스로 돌아가서 MmCreateMdl API로 만들어진 새로운 MDL을 nonpaged pool 메모리 영역에 생성하기 위해 MmBuildMdlForNonPagedPool[4] API를 사용합니다.

여기까지 오면 nonpaged pool 메모리 영역에 SSDT를 가리키는, 마치 심볼릭 링크 같은 것이 하나 생성되게 됩니다. 이제 MdlFlags에 MDL_MAPPED_TO_SYSTEM을 추가함으로써 쓰기가 가능하게 되었

[1] MmCreateMdl : http://msdn2.microsoft.com/en-gb/library/ms801972.aspx
[2] IoAllocateMdl : http://msdn2.microsoft.com/en-gb/library/aa490866.aspx
[3] 여기 : http://bbs.driverdevelop.com/htm_data/16/0105/710.html
[4] MmBuildMdlForNonPagedPool : http://msdn2.microsoft.com/en-gb/library/ms801996.aspx

습니다. MDL_MAPPED_TO_SYSTEM 은 구글님께 물어봐도 속 시원한 대답이 없어서 그냥 쓰기 가능하게 해주는 변수인가보다..라고 생각하고 넘어가겠습니다. -_-;

마지막으로 MmLockedPages[1] API로 생성된 MDL의 물리 메모리 주소를 얻어옵니다. 두 번째 인수로는 KernelMode, UserMode가 있는데 MSDN에도 자세한 설명이 없고 그냥 "대부분의 드라이버에서는 KernelMode를 사용한다"라고만 되어 있습니다.

두 번째 인수 AccessMode가 KernelMode일 경우 MDL이 가리키고 있는 메모리의 유저모드 공간 주소를 구할 때, UserMode는 그 반대의 경우일 때 사용됩니다.

여기까지 오면 이제 우리는 MDL을 통해 시스템의 SSDT를 읽고 쓰기 가능하게 됩니다. 이제 후킹하기 원하는 임의의 API를 우리가 만든 API의 주소로 바꿔치기 하는 과정만이 남았습니다.

```
// Change Function
oldZwWriteFile = (ZWWRITEFILE)InterlockedExchange((PLONG)&SYSTEMSERVICE(ZwWriteFile),(LONG)NewZwWriteFile);
```

그림 59. InterlockedExchange

그림 59에서 볼 수 있듯이 InterlockedExchange[2] 라는 API가 사용되었습니다. 원래 InterlockedXXX 함수들은 멀티 스레드 환경에서 동기화를 위해 사용되는 함수들인데 InterlockedExchange API의 경우 인수로 전달받은 두 값을 비교하여 같지 않다면 첫 번째 인수 Target의 값을 두 번째 인수 value로 바꿉니다. 그리고 리턴 값으로 Target의 값을 반환합니다.

그림 59는 ZwWriteFile라는 원래의 API를 NewZwWriteFile, 즉 우리가 작성한 API로 바꿔치기 하는 코드입니다.

마지막으로 NewZwWriteFile을 살펴보겠습니다.

[1] MmLockedPages : http://msdn2.microsoft.com/en-gb/library/ms801970.aspx
[2] InterlockedExchange : http://msdn2.microsoft.com/en-us/library/ms683590.aspx

```
NTSTATUS NewZwWriteFile(
    IN HANDLE FileHandle,
    IN HANDLE  Event OPTIONAL,
    IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
    IN PVOID ApcContext OPTIONAL,
    OUT PIO_STATUS_BLOCK IoStatusBlock,
    IN PVOID Buffer,
    IN ULONG Length,
    IN PLARGE_INTEGER ByteOffset OPTIONAL,
    IN PULONG Key OPTIONAL
)
{
    NTSTATUS status;
    PUCHAR szProcessName;
    PEPROCESS CProcess;
    CProcess = (PEPROCESS)PsGetCurrentProcess();

//  KdPrint(("ProcessName : [%s]\n", curproc->ImageFileName));
    szProcessName = CProcess->ImageFileName;

    status = oldZwWriteFile( //원래 함수를 호출해서 정상적으로 처리되게함
        FileHandle,
        Event,
        ApcRoutine,
        ApcContext,
        IoStatusBlock,
        Buffer,
        Length,
        ByteOffset,
        Key);

    if(!strncmp((PCHAR)szProcessName, "FDel.exe", 7))
        KdPrint(("ZwWriteFile was Called by [%s]\n", szProcessName));

    return status;
}
```

그림 60. NewZwWriteFile

NewZwWriteFile의 기능은 단순합니다.

원래의 ZwWriteFile를 호출하여 정상적으로 기능이 작동하도록 하는데 만약 FDel.exe라는 프로그램이 ZwWriteFile을 발생시켰다면 특정 디버그 메시지를 출력하도록 하였습니다.

ZwWriteFile을 발생시킨 프로세스의 이름 확인을 위해 PsGetCurrentProcess[1] API를 이용하여 현재 프로세스의 _EPROCESS 구조체를 얻어온 후 ImageFileName 필드를 참조하여 프로세스의 이름을 알아냅니다.

_EPROCESS 구조체는 프로세스를 나타내는 구조체인데, 설명하려면 약간 복잡합니다. 따로 찾아보시길 바랍니다. 해당 구조체를 포함시키기 위해 IFS에서 사용되는 ntifs.h 헤더 파일을 포함시켰습니다. 해당 헤더 파일에 이미 SSDT에 대한 구조체가 정의되어 있어 컴파일 시 중복 에러를 발생시킵니다. ntifs.h 헤더 파일에서 해당 부분을 주석 처리하면 이상 없이 컴파일이 됩니다.

아래는 Fdel.exe 프로그램을 실행시켰을 때 디버그 메시지가 찍히는 것을 캡쳐한 것입니다. Fdel.exe 프로그램은 CreateFile, WriteFile, DeleteFile 을 이용해서 단순히 파일을 생성, 삭제하는 예제 프로그램입니다.

[1] PsGetCurrentProcess : http://msdn2.microsoft.com/en-us/library/ms802957.aspx

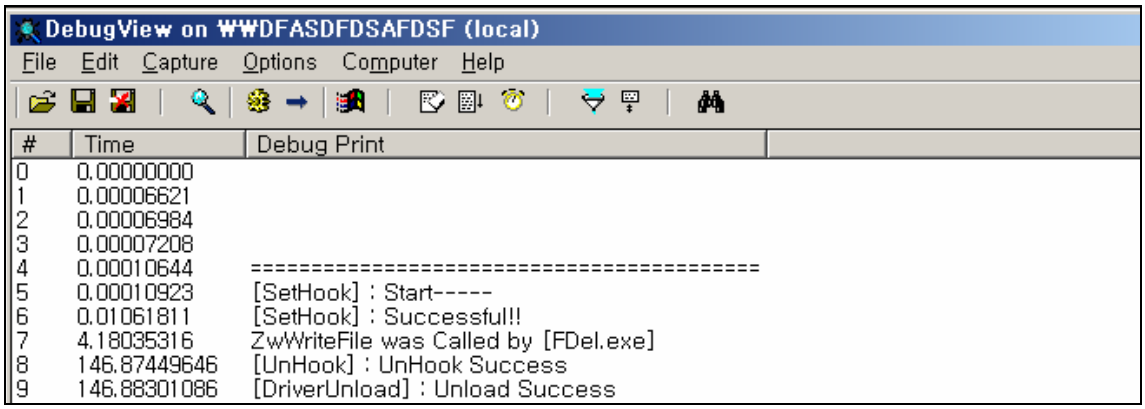


그림 61. ZwWriteFile Hooking Message

CR0 레지스터를 이용한 SSDT Hooking 은 somma님의 bkdp3 소스를 참고하시기 바랍니다.

작성된 모든 프로그램은 아래 링크에서 다운 받으실 수 있습니다.

코드 : http://pds3.egloos.com/pds/200701/04/51/code.zip

(코드 중 FileDeleteSample은 바이너리 파일 이름을 FDel.exe로 변경하시기 바랍니다)

6. 추가

SSDT Hooking에 관한 흥미로운 주제가 몇 가지 있습니다.

1. SSDT Hooking이 모든 스레드를 Hooking 할 수 있는 것은 아닙니다.
   각 스레드는 자신의 서비스 테이블에 대한 포인터를 가집니다. 그러므로 새로운 서비스 테이블을 생성하여 그것을 가리키게 할 경우 SSDT Hooking은 더 이상 쓸모가 없어지게 됩니다.
   선경렬 님의 ksthb[1] 프로그램 또는 Dual 님의 saruengang[2] 프로그램으로 실제 확인해 볼 수 있습니다. 소스가 궁금하시다면 위의 프로그램 외 Dual 님이 공개하신 HyperOlly[3] 과 Dual's ksthb[4]를 보시면 많은 도움이 될 것입니다.

2. SDT Restore
   SSDT의 상태를 검사해서 원래대로 복원시켜 버리는 프로그램이 있습니다. SIG[5] 라는 회사에서 공개한 프로그램인데요 해당 프로그램을 사용하면 역시 SSDT Hooking 은 무용지물이 됩니다.
   해당 프로그램에 대한 내용은 아래의 링크에서 확인하시기 바랍니다.
   SDT Restore : http://www.security.org.sg/code/sdtrestore.html

3. Microsft는 2007년 1월 Windows Vista를 출시하였습니다. Vista에서는 기존의 방법대로 후킹을 하는 것이 불가능해집니다. 현재 수 많은 해커들에 의해 분석이 시도되고 있으며 많지는 않지만 그 결과물로 Vista의 Unexported Kernel Symbol이 발표되었습니다.
   아래의 링크에서 확인하시기 바랍니다.
http://www.msuiche.net/2007/01/01/windows-vista-64-bits-and-unexported-kernel-symbols/

현재 마이크로소프트는 Windows Driver Kit이라는 새로운 형태의 Framework를 제공하고 있으며 커널 모드 Driver Framework(KMDF), User Mode Driver Framework(UDMF)로서 구현하고 있습니다.

[1] Ksthb : http://www.zap.pe.kr/index.php?page=pages/researches/winternals_kr.php
[2] Saruen Gang : http://dual.inxzone.net/blog/entry/Saruen-Gang寫輪眼-100-Official-Release
[3] HyperOlly : http://dual.inxzone.net/blog/attachment/2571194572.zip
[4] Dual's ksthb : http://dual.inxzone.net/blog/attachment/731037.zip
[5] Security and Information InteGrity : http://www.security.org.sg

7. 참고 자료

Documents)

1. Rootkits: Subverting the Windows Kernel, Greg Hoglund

1. Inside Windows Rootkits

2. Windows Driver Model2, Microsoft Press

3. Attack Native API, Devguru

4. Rootkit을 이용하는 악성코드, 사이버시큐리티 2005년 11월호

5. ROOTKITS, Greg Hoglund, James Butler

6. Defeating Kernel Native API Hookers by Direct Service Dispatch Table Restoration, Chew Keong TAN

7. 원리와 예제로 배우는 Windows 2000 디바이스 드라이버, 인포북

8. Windows Internals, Microsoft Press

9. 64비트 윈도우 커널분석 AMD64, Micro Software 2005년 10월호

Web)

1. Hooking Windows NT System Services

http://www.windowsitlibrary.com/Content/356/06/2.html

2. Windows Global API Hooking

http://dual.inxzone.net/backup/blog/index.php?pl=125&ct1=4

3. Window 쪼물딱거리기

http://somma.egloos.com/2731001

4. How Do Windows NT System Calls REALLY Work?

http://www.codeguru.com/Cpp/W-P/system/devicedriverdevelopment/article.php/c8035

5. System Call Optimization with the SYSENTER Instruction

http://www.codeguru.com/cpp/w-p/system/devicedriverdevelopment/article.php/c8223

6. Hooking the kernel directly

http://www.codeproject.com/system/soviet_direct_hooking.asp

7. That's Just the Way It Is - How NT Describes I/O Requests

http://www.osronline.com/custom.cfm?name=articlePrint.cfm&id=74

8. MSDN

http://msdn.microsoft.com/library/default.asp

9. 드라이버온라인

http://www.driveronline.org/