Beist
Security Study Group

초보자를 위한
Kernel based windows rootkit
-1부-

By Beist Security Study Group
(http://beist.org)

요약: 이 문서는 윈도우2000/XP/2003 환경에서의 커널 루트킷에 대한 개요와 윈도우와 하드웨어 간의 커넥션에 대해 다룹니다. 그리고 실습을 위해 커널 레벨에서 CR0 레지스터를 변경하여 SSDT의 read-only 속성을 write 속성으로 바꾸는 프로그램을 디바이스 드라이버를 이용해서 작성할 것입니다. 이 글을 읽는 독자가 유저레벨에서의 윈도우 시스템 프로그래밍 경험이 있다는 전제하에 진행하겠습니다.

1. 소개 - Rootkit?

루트킷은 해커가 특정 시스템을 해킹한 이후에 시스템의 제어권을 획득할 목적으로 설치하는 악성 프로그램을 말합니다. 루트킷은 유저 레벨에서 구현될 수도 있고 커널 레벨에서 구현될 수도 있는데 본 강좌에서는 커널 레벨의 루트킷에 대해서 다루겠습니다. 해커는 루트킷을 설치함으로써 하드웨어의 제어권을 쥐고 있는 소프트웨어를 자신의 목적에 맞게 조작하여, 시스템을 자신이 원하는 방향으로 조작할 수 있습니다.

루트킷의 성능을 좌우하는 요소는 여러 가지가 있지만 대표적인 요소를 꼽자면,

1) 원하는 목적대로 제어권을 획득할 수 있는가?
2) detect 되지 않게 작성할 수 있는가?

이 두 가지를 꼽을 수 있습니다. 위 조건을 만족시키기 위한 가장 좋은 방법은 루트킷을 커널 레벨에서 작동하도록 제작하는 것입니다. 즉, 운영체제가 작동하는 커널 모드에서 루트킷을 작성한다면 보다 운영체제에 가까이 접근하여 운영체제만이 가질 수 있는 특권을 루트킷도 누릴 수 있다는 의미가 됩니다.

그 외, 커널 레벨에서 작동할 때의 장점을 더 꼽아보겠습니다.

- 운영체제의 코드나 데이터를 변경시킬 수 있게 됩니다. 이것을 통해 운영체제의 제어흐름을 변경시켜서 원하는 목적을 달성할 수 있습니다. 가장 커다란 결과로는 후킹을 system wide하게 적용시킬 수 있게 됩니다. 이것은 실제로 윈도우 커널 후킹 기법인 Native API Hooking과 IDT Hooking을 가능하게 합니다. 또한 강력한 기법 중 하나인 DKOM(Direct Kernel Object Manipulation)도 가능하게 합니다.

- 커널 레벨에서 동작하는 다른 모든 프로그램의 코드나 데이터도 변경시킬 수 있게 됩니다. 이러한 점을 악용한다면 대부분의 보안 프로그램을 무력화 시킬 수 있습니다. 보안 프로그램은 시스템에 대한 강력한 권한을 얻기 위하여, 커널 레벨 기반으로 작성하게 됩니다. 보안 프로그램들이 커널 레벨에서 작동되어도, 루트킷도 역시 같은 커널 레벨에서 실행되기 때문에 해커가 단순한 커널 해킹 기법만으로도 시스템 코드를 조작해서 제어권을 루트킷에게 넘어오게 할 수 있습니다. 그 결과 보안 프로그램을 무력화 시키고, 보다 조작을 가하면 보안 프로그램을 자신이 원하는 방향으로 실행할 수 있습니다.

- 마지막으로, 루트킷을 detect 하기 어렵게 만듭니다. 커널 레벨은 운영체제가 작동하는 모드인만큼 사용자나 유저 레벨 프로그램이 접근하기 어려운 영역입니다. 보안 프로그램이 커널 레벨에서 작동한다고 하더라도 위에서 말했던 것처럼 루트킷이 그 보안프로그램을 무력화시키는 코드를 작성해두었다면 detect 되지 않습니다. 루트킷과 보안프로그램은 커널 레벨에서 작동하는 만큼 시스템에 대한 권한 차이는 없기 때문에, 누가 먼저 서로를 감지하고 무력화 시키느냐에 따라 승패가 갈리게 됩니다.

2. Windows Internals

윈도우 커널 루트킷의 공격 기법을 이해하고, 루트킷을 작성하려면 윈도우의 내부 구조에 대한 이해가 요구됩니다. 프로세스와 스레드, 스케쥴링, API의 호출과정, 페이징 과정, ring 0와 ring 3, 그 밖의 윈도우 내부구조들을 이해해야 합니다. 하지만 이 내용만으로도 상당히 방대하므로 본 문서에서는 API 호출 과정에 대한 내용과 ring 0와 ring 3에 관한 내용, 그리고 몇 가지 중요한 윈도우 내부 구조만을 서술하겠습니다. 자세한 내용은 다른 책들을 참고해보시기 바랍니다.

API의 호출과정을 분석해보면 윈도우 내부 구조의 많은 개념들을 접할 수 있습니다.

- Kernel32.dll, Ntdll.dll 등과 같은 중요한 시스템 모듈들의 역할
- 유저 모드와 커널 모드
- System Call의 개념
- Native API, SSDT(System Service Descriptor Table)
- SSDT Hooking 기법에 관한 이해

이 개념들을 API 호출 과정을 분석하면서 살펴보겠습니다.

2-1. 커널 모드와 유저 모드

API 호출 과정을 커널 모드까지 진입하여 알아보려면 먼저 커널 모드가 무엇인지를 알아야 합니다.

CPU에 내릴 수 있는 명령에도 각각 권한이 있습니다. 이 권한이 높을수록 운영체제에 깊이 접근할 수 있습니다. 내릴 수 있는 명령들에 권한을 부여하여 구분하지 않았다면 갖가지 심각한 악성 프로그램들이 특별한 제한 없이 활개칠 것입니다. 이런 상황을 방지하기 위해서 CPU가 내릴 수 있는 명령에 권한을 부여했는데, 이러한 프로세서 디자인을 Multiple Ring 이라고 하고, 간단한 구조는 아래 그림과 같습니다.

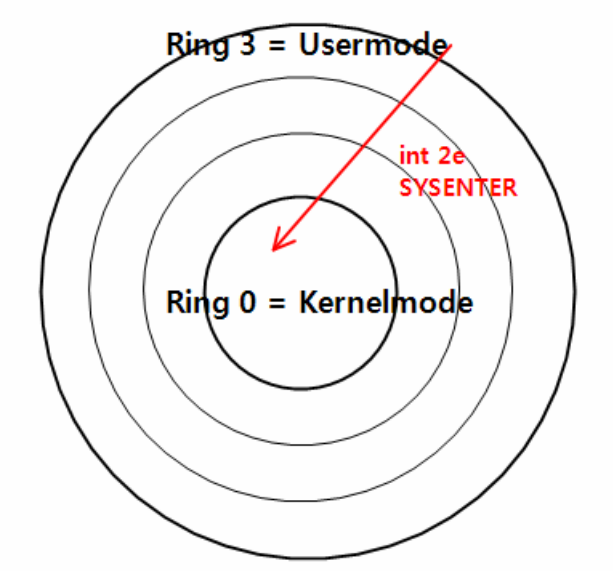

그림 1. Multiple Ring

Ring0~3까지 총 4가지 권한이 있지만 Windows에서는 Ring 0와 Ring 3만을 사용합니다. Ring 3는 유저 모드라고 하고, Ring 0는 커널 모드라고도 합니다. 그림1에서 보면 맨 바깥쪽 원의 권한이 Ring 3인 유저모드입니다. 이곳은 유저 레벨 프로그램의 코드가 실행되는 곳이며, 특별한 명령(int 2e, SYSENTER)을 통해서만 Ring 0 권한에 진입할 수 있습니다. 유저 모드에서는 하드웨어에 직접 접근하거나 커널의 가상메모리, 그 밖의 중요한 레지스터나 데이터에 접근하는 것이 제한되어있습니다.

반면에 커널 모드에서는 직접 하드웨어 컨트롤러에 명령을 보내거나 운영체제의 코드, 기타 시스템에 중요한 레지스터에도 접근이 가능합니다. 특히 유저 모드에서 접근이 불가한 0x8000000 이상인 커널의 가상메모리공간에도 접근이 가능합니다.

2-2. 유저모드에서의 API 호출과정

이제 본격적으로 API 함수의 호출 과정을 분석해보겠습니다. 여기서는 CreateFile() 함수를 호출했을 때를 가정하고 진행합니다.

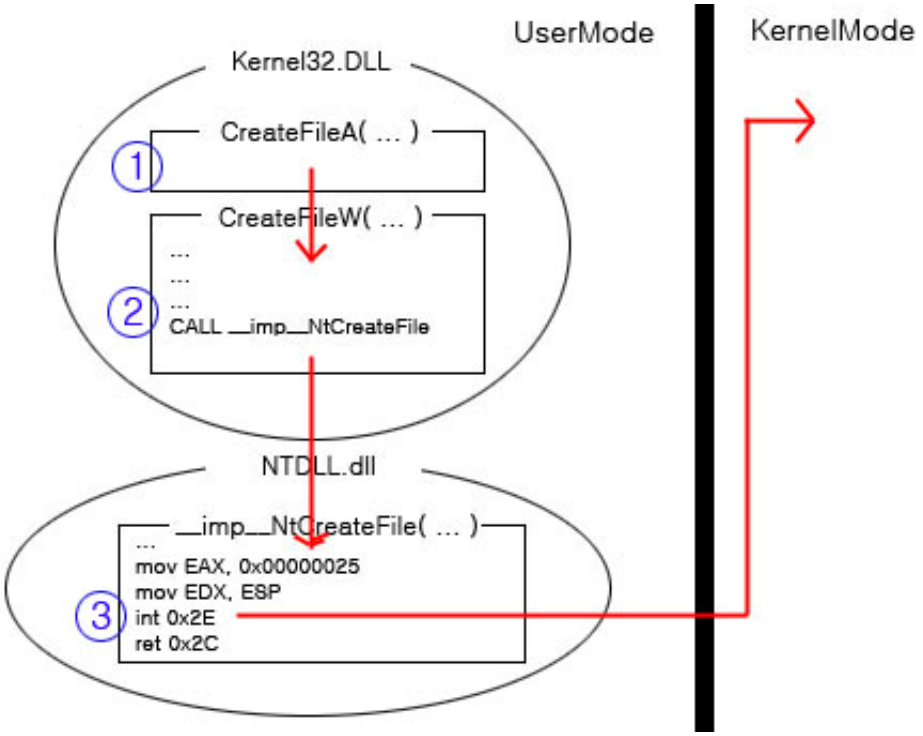

그림2. 유저레벨에서의 CreateFile() 함수호출과정

1. 일반 응용프로그램에서 CreateFile() 함수를 호출할 때, 첫 번째 인자로 파일이름에 해당하는 문자열을 전달합니다. 파일이름을 ASCII 형식으로 쓰는지 UNICODE 형식으로 쓰는지에 따라 CreateFileA() 함수 또는 CreateFileW() 함수를 호출하게 됩니다. UNICODE 형식으로 파일이름을 전달하는 경우에는 CreateFileA() 함수를 거치지 않습니다. ASCII 형식으로 파일이름을 전달할 경우엔, CreateFileA() 함수에서 ASCII 문자열을 UNICODE 형식을 위한 Wide Character로 변환시키고 나서 CreateFileW() 함수를 호출합니다. 이렇게 xxxxA() -> xxxxW() 호출되는 방식은 많은 Win32 API 함수에서 쓰이고 있습니다.

2. CreateFileW() 함수로 흐름이 건너옵니다. CreateFileW() 함수는 내부적으로 여러 가지 복잡한 과정을 거치게 되는데, 중요한 내용은 아니기 때문에 생략하겠습니다. 내부의 복잡한 과정을 거치고 나면, CALL __imp__NtCreateFile 이라는 코드가 등장하는데, 이 코드는 ntdll.dll의 __imp__NtCreateFile() 함수를 호출합니다.
   여기서 ntdll.dll은 kernel32.dll과 커널 사이의 일종의 중개 역할을 하는 모듈로서, 주로 유저 모드와 커널 모드로의 전환을 수행하는 역할을 담당합니다. 이 코드를 실행하면 이제 흐름은 __imp__NtCreateFile() 함수로 이동합니다.

3. ntdll.dll의 __imp__NtCreateFile()로 흐름이 넘어왔습니다. 그림을 보게 되면 첫 번째 코드는 mov EAX, 0x00000025인데, 이것은 EAX 레지스터에 0x25를 집어넣겠다는 코드입니다. 0x25는 커널의 약 300개의 Native API 함수 중, 어떤 함수를 호출할지 정하는 인덱스가 되어서, 나중에 커널 모드로 제어가 넘어갔을 때 참조하게 됩니다.

Native API는 반드시 알아야 하는 개념인데, 강력한 커널 후킹 기법 중 하나가 이 Native API를 후킹하는 것이기 때문입니다. Native API를 간단히 정의하면, Win32 API 혹은 이와 비슷한 다른 서브 시스템이 커널단에서의 도움이 필요한 경우에 호출하는 커널의 특별한 함수집합을 말합니다.

커널단에서 도움을 주는 개념이기 때문에 Native API 함수를 System Service Dispatcher 라고도 부릅니다.

좀더 구체적인 이해를 위해 Win32 API 함수와 이에 대응되는 Native API를 몇 개 나열해 보겠습니다.

CreateFile()->NtCreateFile(),
ReadFile()->NtReadFile(),
CreateProcess()->NtCreateProcess()
OpenProcessToken()->NtOpenProcessToken()
WriteProcessMemory()->NtWriteVirtualMemory()

위를 보시면 알 수 있듯이 중요한 시스템 API 함수들이 Native API함수를 거치게 됩니다. 이러한 Native API를 후킹한다면, 아래 그림처럼 유저 레벨에서의 모든 프로세스에 적용이 되면서 결국은 전역적으로 후킹이 됩니다. 예를 들어서 NtCreateProcess()를 후킹한다면, 모든 프로세스의 생성을 제어할 수 있고, NtWriteVirtualMemory()를 후킹하면, 모든 프로세스의 메모리 writing 시도를 제어할 수 있으므로, 특정 프로세스 메모리공간을 write하지 못하게 만들 수도 있습니다. 실제로 몇몇 보안프로그램은 이 함수를 후킹해서 프로세스 메모리 공간을 보호하기도 합니다.

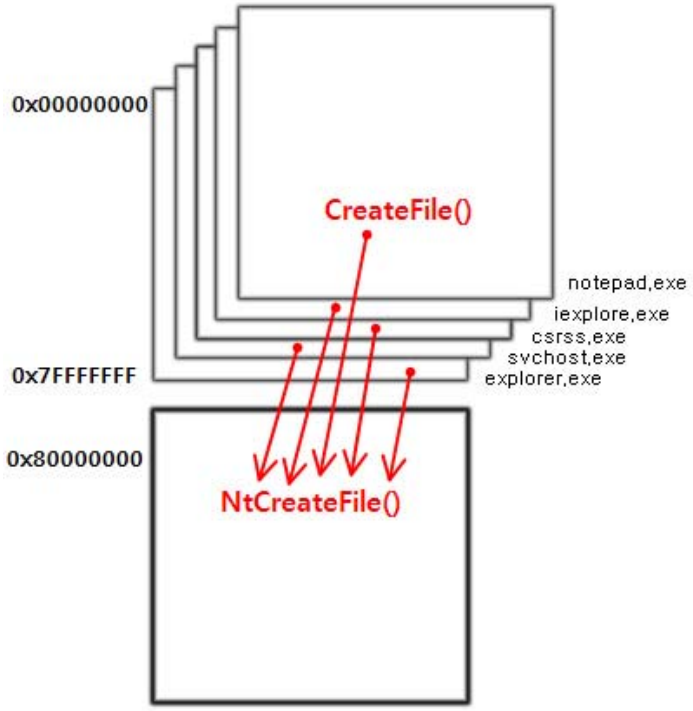

그림3. 각각의 프로세스의 CreateFile()호출은 한곳의 NtCreateFile()로 이동된다.

이제 두 번째 코드를 건너뛰고, 세 번째 코드를 살펴보면 int 0x2E 명령이 보입니다. 이 코드는 유저 모드에서 커널 모드로의 제어 이행을 수행하는 코드로써, 소프트웨어 인터럽트를 발생시키고, 유저 스택을 커널 스택으로 변경시킵니다. 그리고 int 0x2E의 인터럽트 핸들러인 KiSystemService()를 실행시킵니다. 여기서 int 2e 명령어는 Windows 2000까지만 커널로의 이행을 수행하는 게이트가 됩니다. 펜티엄2이상의 사양을 갖춘 XP 이후에서부터는 int 2e 대신 SYSENTER를 사용합니다. 동일한 기능을 수행하지만 SYSENTER는 커널로의 이행을 좀더 빠르게 수행하도록 만들어졌고, int 2e를 게이트로 쓰는 것은 이젠 구식의 방법이 되어버렸습니다. SYSENTER를 통해 커널 모드로 이동했다면, KiSystemService()를 거치기 전에 KiFastCallEntry()를 거치게 됩니다. 이것에 대해 자세한 사항은 다른 문서들을 참고하시길 바랍니다.

2-3. 커널 모드에서의 API 호출과정

이제 커널 모드로 넘어와서, API 호출과정을 분석해 보겠습니다. 전체흐름은 아래 그림과 같습니다.

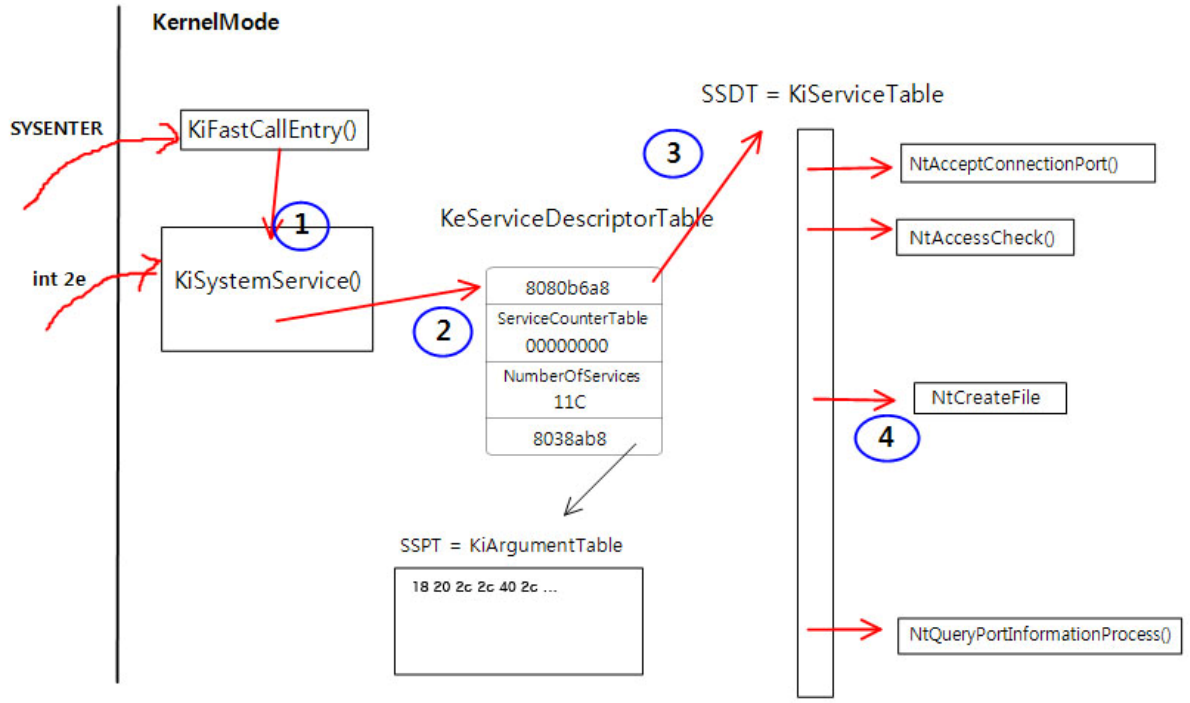

그림4. 커널에서의 API 호출과정

1. 앞에서 언급했지만 유저레벨에서 SYSENTER를 통해 커널 모드로 진입했을 경우에는 KiFastCallEntry()를 거치고, int 2e로 진입했을 경우에는 KiFastCallEntry()를 거치지 않고 바로 KiSystemService()로 넘어가게 됩니다.

2. KiSystemService()에서 하는 가장 중요한 작업은 SSDT의 주소 값을 얻어오고 어플리케이션에서 호출한 API에 맞는 Native API의 주소를 찾아내서 호출하는 것입니다. KiSystemService() 함수에서는 먼저 SSDT를 찾기 위해서 KeServiceDescriptorTable에 접근합니다. KeServiceDescriptorTable의 구성요소는 4가지로 이루어져있습니다. 여기서는 두 번째 요소를 제외하고 나머지를 살펴보겠습니다. 첫 번째 요소는 SSDT(KiServiceTable)의 주소를 담고 있고, 세 번째 요소인 NumberOfService는 뜻 그대로 풀어 쓰면 서비스의 개수가 됩니다. 여기서 말하는 서비스는 Native API를 지칭하기 때문에 결국 세 번째 요소는 Native API의 총 개수를 말합니다. 그림에 나온 11C는 Windows XP SP2에서의 실제 값으로, 십진수로는 284개가 됩니다. 네 번째 요소는 KiArgumentTable의 주소 값을 담고 있습니다. KiArgumentTable은 SSPT(System Service Parameter Table)라고도 불리는데, 이들 각각은 SSDT의 Native API와 1:1대응을 합니다. 이것들은 대응되는 Native API 함수의 파라미터 총 크기를 바이트단위로써 나타냅니다.

3. Native API를 찾기 위해서, Ntdll.dll에서 EAX 레지스터에 인덱스의 형태로 값을 저장한다는 것을 이전에 살펴봤었습니다. 이제 이것과 SSDT 주소 값을 이용해서 Native API 함수의 엔트리 주소 값을 얻어오게 됩니다. SSDT주소(KiServiceTable)+[EAX인덱스*4]를 한다면 아주 간단하게 Native API 함수주소를 얻어올 수 있는데, 이것은 실제로 KiSystemService()가 하는 코드와도 같습니다.

4. 이제 원하는 Native API함수로 진입하게 됩니다. 만약 어플리케이션에서 CreateFile()함수로 USB 메모리 디스크에 aaa.txt파일을 생성한다고 치면, 시스템 게이트를 거쳐 커널의 NtCreateFile() 함수로 진입합니다. NtCreateFile()함수에서는 커널의 구성요소인 I/O Manager를 통해 디스크 드라이버와 USB관련 드라이버들을 거치면서 일련의 작업을 진행합니다. 이 과정은 문서와는 맞지 않는 내용이기 때문에 기술하지 않겠습니다.

3. 간단한 루트킷 제작

커널 기반 루트킷은 Windows에서 디바이스 드라이버의 형태로 동작하기 때문에 루트킷을 작성하려면, WDM 디바이스 드라이버를 작성할 수 있어야 합니다. WDM을 이용하지 않아도 좋지만, Windows 2000/XP/2003 모두 호환성 있도록 작성하려면 WDM을 이용해야 합니다. (Vista까지 호환성을 유지하려면 WDF라는 새로운 드라이버 개발환경으로도 작성할 수 있어야 합니다.)

WDM 드라이버의 빌드 환경과 디버깅환경을 구축하는 방법을 간단히 설명하고, 그 다음 간단한 WDM 드라이버 제작을 해보겠습니다. 그리고 작성된 드라이버의 기본 틀에 덧붙여서 SSDT Hooking을 가능하게 하도록 CR0 레지스터를 수정하는 코드를 작성해보겠습니다.

3-1. 빌드 환경 구축하기

우선 빌드 환경을 구축하기 전에 고려해야 할 사항이 있습니다. 사용하고 있는 CPU가 32비트인가 64비트인가? 단일프로세서 환경인가 멀티프로세서 환경인가? 등을 체크해야 합니다. 이 문서에서는 IA-32 기반과 단일프로세서 환경에서 프로그래밍을 한다고 가정합니다. 64비트의 경우에는 아직 많이 연구들이 진행되지 않았고, 멀티프로세서 환경에서 CPU레지스터의 영향을 받는 루트킷의 경우에는 프로그래밍하기가 약간 까다로워지기 때문에 디바이스 드라이버를 처음 접하는 분들은 제작하기 어렵다고 느끼실 수 있습니다.

이제 본론으로 들어가서 빌드 환경 구축에 대해 살펴보겠습니다. WDM 디바이스 드라이버를 만들기 위해서는 우선 DDK를 구해야 합니다. DDK를 구하기 위해서는 Microsoft 홈페이지를 참고하세요. (http://www.microsoft.com/ddk)

DDK를 설치하셨다면, 프로그래밍 환경과 빌드 환경을 통합 할지 결정하셔야 합니다. 통합하지 않는다면 직접 빌드 프롬프트에 들어가서 build 명령어를 통해 드라이버를 컴파일 해야 합니다. 반면 프로그래밍 환경과 빌드 환경을 통합한다면 상당히 편리해지는데, 이것을 구성하는 방법은 여러 가지가 있습니다. VC6의 환경설정을 통해서 build를 기존의 메뉴를 통해 사용할 수도 있고, 혹은 DriverStudio(Softice) 3.2를 설치하면 같이 설치되는 간단한 툴바를 통해 VC6에서 버튼 하나로 빌드시킬 수도 있습니다. 저는 주로 후자의 방법을 사용하고 있습니다. 여러 가지 환경이 있으니 자신이 편하다고 생각하는 환경을 구축하시길 바랍니다.

3-2. 디버깅 환경 구축하기

디바이스 드라이버 프로그래밍은, 프로그램의 규모가 커질수록 잠재적인 버그들이 너무나 많아지고, 디버깅하기도 유저 레벨 프로그램들에 비해 상당히 까다롭습니다. 그 때문에 실무에서도 다른 프로그래밍 분야보다 안정성을 높이는데 많은 시간과 노력이 소요됩니다. 이를 위해서 커널 레벨에서의 디버거 사용법에 대해서 익숙해질 필요가 있습니다. 디바이스 드라이버를 디버깅할 수 있는 커널 디버거로는 대표적으로 WinDbg와 Softice가 있습니다.

WinDbg는 PC 2대를 연결해서 디버깅하는 구조로 되어있는데 하나는 디버기(debugee)로 다른

하나는 디버거(debugger)로 사용하게 됩니다. 두 PC를 연결하기 위해서 시리얼케이블이나 1394 케이블이 사용되는데, 시리얼케이블은 속도가 느리기 때문에 Windows XP 이상의 OS라면 1394 케이블을 쓰는 것이 좋습니다. Vista에서는 USB 2.0 디버깅 전용 케이블도 지원 된다고 하지만 아직까지는 케이블을 국내에서 구할 수 없고 약간 비싸다는 게 단점입니다.

WinDbg의 장점으로는 Microsoft가 내놓은 커널 디버거인 만큼 비교적 안정적으로 디버깅을 할 수 있습니다. Softice는 커널 디버깅을 하기 위해서 후킹 기법 같은 시스템에 불안정적인 방법을 사용하게 되지만, WinDbg는 Microsoft에서 제공하는 API를 통해 시스템에 안정적인 방법으로 디버깅을 하게 됩니다. Softice는 로컬 디버깅을 할 때 많이 사용됩니다. WinDbg도 LiveKd라는 유틸을 이용한다면 보다 쉽게 로컬 디버깅을 사용할 수 있지만, 실제적인 실시간 디버깅이 아니고 단순히 한 순간의 시스템 덤프를 이용한 디버깅이기 때문에 드라이버 디버깅은 불가능하고 그 밖에 제한 사항이 많이 있으나 간단히 Windows의 내부 구조를 둘러볼 때 쓰는 용도라면 유용합니다.

WinDbg는 Microsoft 홈페이지에서 무료로 다운로드 받을 수 있지만 Softice는 상용 프로그램입니다. Softice는 64비트 운영체제 디버깅을 지원하지 않고 현재 개발 중단된 상태이기 때문에 WinDbg를 이용해서 커널 디버깅을 시작하는 것을 추천합니다.

WinDbg나 Softice의 자세한 사용법은 홈페이지를 참고하시기 바랍니다.

3-3. 드라이버의 기본 틀

```
#include <ntddk.h>// 대부분의 드라이버가 반드시 포함해야 하는 헤더파일.

VOID OnUnload( IN PDRIVER_OBJECT DriverObject)
{
        // 드라이버 안에서 할당했던 모든 것을 해제한다.
        // OnUnload가 수행되고 나면 드라이버는 메모리상에서 제거된다.
}

NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING theRegistryPath )
{
        // 드라이버의 메인함수.
        // 이제부터 함수 안의 모든 코드는 커널 레벨에서 수행된다.

        // 드라이버 언로드 함수의 주소를 등록시킨다.
        theDriverObject->DriverUnload = OnUnload;

        // DriverEntry의 수행이 끝나면 STATUS_SUCCESS를 반환한다.
        return STATUS_SUCCESS;
}
```

KBasic.c

위의 주석에 모든 설명을 달아놓았습니다. 기억해야 할 것은 DriverEntry() 함수에서 드라이버가 시작되고, Unload() 함수에서 제거된다는 것입니다.

이제 위 소스를 빌드 시켜보겠습니다. 빌드하기 위해서는 소스파일(.c파일)뿐만 아니라 MAKEFILE, SOURCE 라는 파일도 필요합니다.

먼저 MAKEFILE 파일의 내부를 살펴보도록 하겠습니다.

```
#
# DO NOT EDIT THIS FILE!!!  Edit .₩sources. if you want to add a new source
# file to this component.  This file merely indirects to the real make file
# that is shared by all the driver components of the Windows NT DDK
#

!INCLUDE $(NTMAKEENV)₩makefile.def
```

MAKEFILE

위를 보시면 알 수 있듯이 이 파일을 수정하지 말고 그냥 이 내용 그대로 MAKEFILE 파일을 작성해서 소스파일과 같은 폴더에 넣어두면 됩니다.

```
TARGETNAME=KBasic
TARGETPATH=OBJ
TARGETTYPE=DRIVER
SOURCES=KBasic.c
```

SOURCE

각각의 항목을 살펴보면, TARGETNAME은 프로젝트의 네임을 뜻하고, TARGETPATH는 컴파일 된 파일들이 위치할 하위 폴더를 말하는데, 보통 'OBJ' 폴더를 만듭니다. TARGETTYPE는 작성하는 프로그램의 성격을 결정하는데 여기서는 DRIVER를 씁니다. 그리고 SOURCES는 작성한 소스 파일의 이름을 씁니다. 소스 파일이 여러 개일 경우에는, SOURCES=KBasic.c, add1.c, add2.c 이런 식으로 추가합니다.

이제 빌드 프롬프트를 이용해서 빌드 해보도록 하겠습니다.

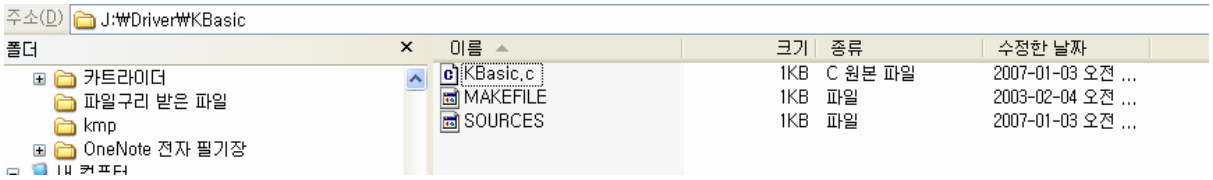

그림5. KBasic.c, MAKEFILE, SOURCES

위의 그림과 같이 세 개의 파일이 같은 폴더 내에 존재해야 합니다.

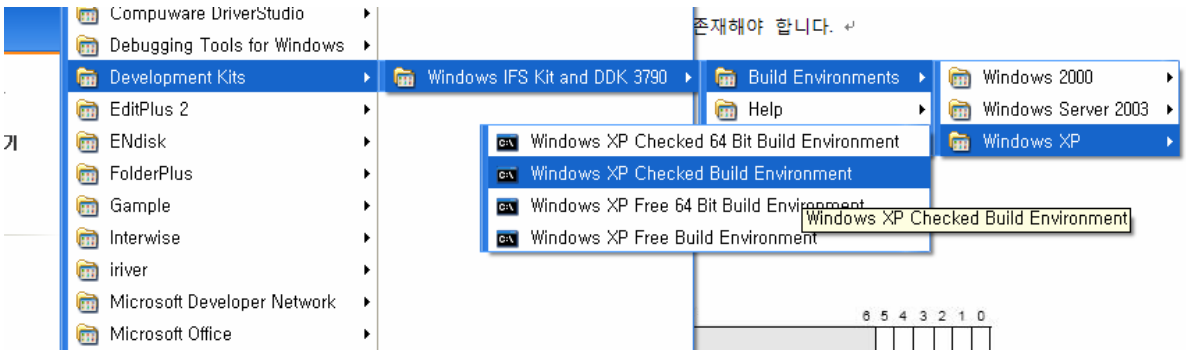

그림6. 빌드 프롬프트로 가기

드라이버를 컴파일 시키려면 빌드 프롬프트를 실행해야 합니다. 위의 그림6처럼 시작->모든프로그램->Development Kits 안에 가면 설치된 버전에 맞는 DDK 폴더 안에 Build Environments 폴더가 존재할 것입니다. 이곳에 또 하위 폴더로 Windows 2000, Windows Server 2003 그리고 Windows XP가 존재합니다. 어떤 하위OS폴더의 빌드 프롬프트를 선택하느냐에 따라서 각각의 OS환경에 특화된 드라이버 파일을 만들게 됩니다. 여기선 현재 제가 쓰고 있는 OS가 Windows XP이기 때문에 Windows XP 폴더의 Windows XP Checked Build Environment를 클릭하겠습니다. Checked Build와 Free Build의 차이점은 디버깅 정보를 포함해서 드라이버 파일을 만드느냐? 그렇지 않느냐? 의 차이입니다. Checked Build로 하는 것이 좋습니다.

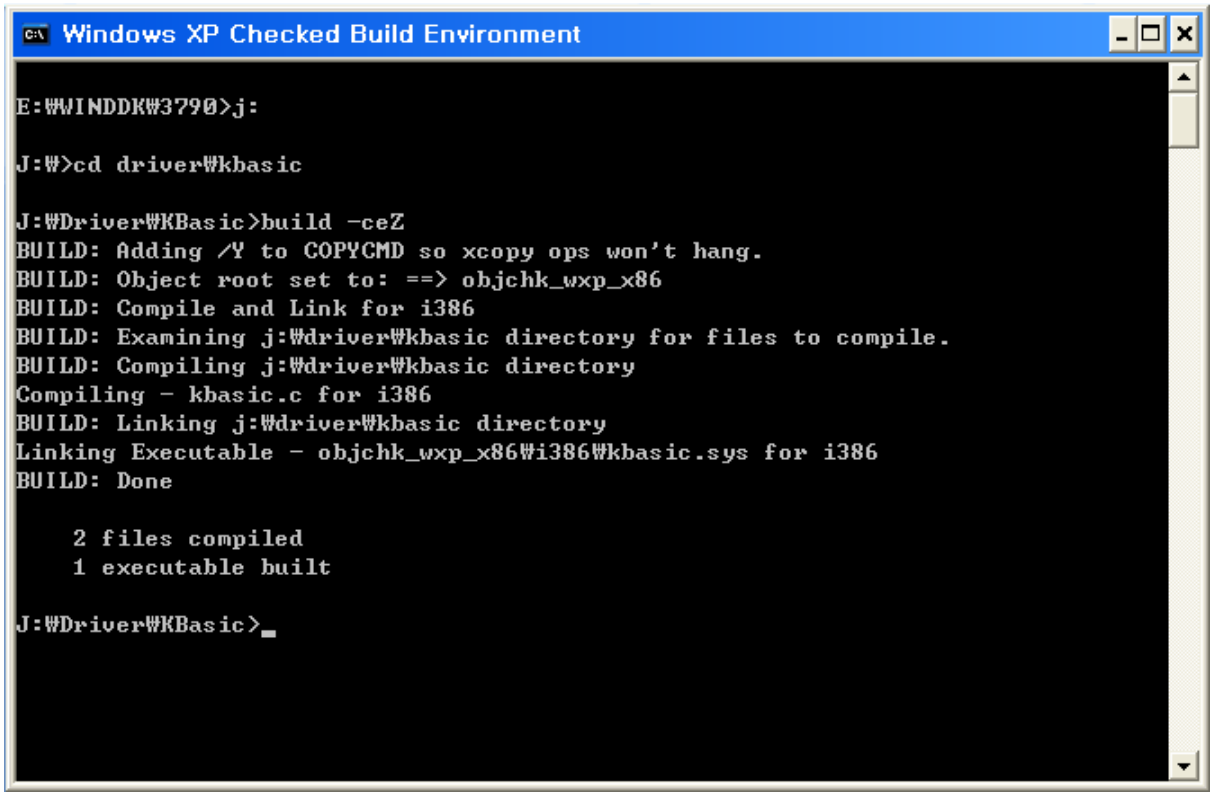

그림7. 빌드 화면

이제 빌드 프롬프트에서 소스 파일이 있는 폴더로 가서 그림7과 같이 build -ceZ라고 치면 컴파일을 하게 됩니다. 메시지를 보면 objchk_wxp_x86₩i386₩kbasic.sys로 드라이버 파일 만들기에 성공했다고 뜹니다. 이제 이 폴더에 가서 드라이버를 실행해보겠습니다.

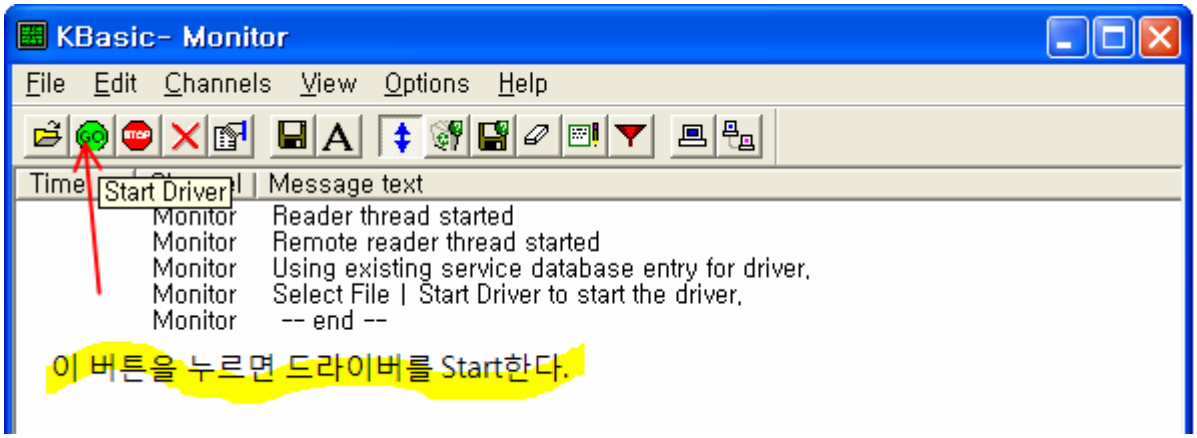

그림8. DriverStudio의 DriverMonitor 캡쳐화면

위의 그림은 DriverStudio에 구성 요소 중 하나인 DriverMonitor의 캡쳐 화면입니다. 드라이버를 로딩해서 구동시키려면 드라이버 로더 프로그램이 필요한데 이에 유용합니다. 다른 로더 프로그램도 공개된 것이 많으니 찾아보시기 바랍니다.

메뉴에서 File->Open에서 드라이버 파일인 Kbasic.sys를 선택한 다음에 위의 툴바에 나오는 Go 버튼을 누르게 되면 드라이버가 구동하게 되고, 옆의 Stop버튼을 누르게 되면 드라이버의 Unload 루틴을 실행시키고 나서 드라이버를 메모리에서 제거시킵니다.

가장 기본적인 드라이버를 작성하여 빌드하고 드라이버 로더를 통해 구동까지 해봤습니다. 이제 다음은 여기서 코드를 조금 더해서 CR0레지스터 조작을 통해 SSDT메모리의 read-only속성을 write속성으로 변경시켜보겠습니다.

3-4. CR0 레지스터 조작하기

CR0 레지스터는 제어 레지스터(Control Register)의 하나로써, 보호모드에서 프로세스의 특성과 작동 모드를 결정하는 레지스터입니다.

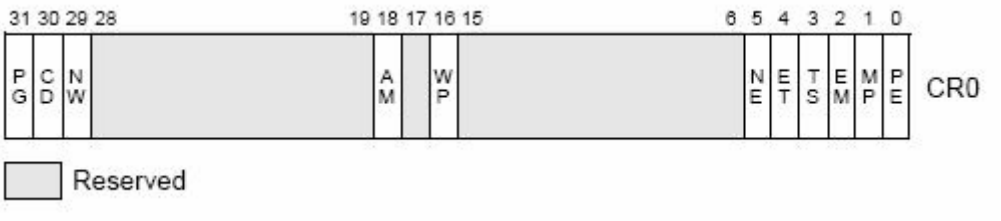

그림9. CR0 레지스터

위의 그림을 보시면 16번째 비트로 WP비트를 볼 수 있는데, Windows XP 이상의 OS에서 이 비트를 0으로 만든다면 SSDT나 IDT같은 메모리 페이지들의 read-only속성을 write속성으로 고칠 수 있고, 이것을 통해 SSDT 후킹이나 IDT 후킹 기법을 가능하게 만들 수 있습니다.

이제 이 기능을 구현한 소스 코드를 작성해 보겠습니다.

```
#include <ntddk.h>

VOID OnUnload( IN PDRIVER_OBJECT DriverObject)
{
	__asm	// CR0 레지스터 속성 되돌리기
	{
		push eax
		mov eax, CR0
		or eax, NOT 0FFFEFFFFh
		mov CR0, eax
		pop eax
	}

	DbgPrint("unload~~₩n");
}

NTSTATUS DriverEntry( IN PDRIVER_OBJECT theDriverObject, IN PUNICODE_STRING theRegistryPath )
{
	DbgPrint("DriverEntry() Start₩n");
	theDriverObject->DriverUnload = OnUnload;

	__asm	// CR0 레지스터 WP비트를 0으로 만들기~~
	{
		push eax
		mov eax, CR0
		and eax, 0FFFEFFFFh
		mov CR0, eax
		pop eax
	}

	return STATUS_SUCCESS;
}
```

KCr0.c

이전에 작성했던 KBasic.c와 다른 부분은 인라인 어셈코드가 들어가 있는 굵은 글씨의 부분밖에는 없습니다. 이것이 CR0 레지스터의 WP비트를 바꾸는 루틴입니다. 정말로 CR0 레지스터의 비트가 바뀌었는지 LiveKd를 이용해서 확인해보겠습니다. LiveKd는 지금은 Microsoft에 인수된 Sysinternals.com 홈페이지를 들어가면 다운로드 받을 수 있고, 설치법은 압축된 파일을 전부 WinDbg의 폴더에 복사시키면 됩니다. 실행법은 단순히 LiveKd.exe 파일을 실행하면 됩니다.

먼저 드라이버를 구동시키기 전의 CR0 레지스터의 값은 밑의 그림과 같습니다.

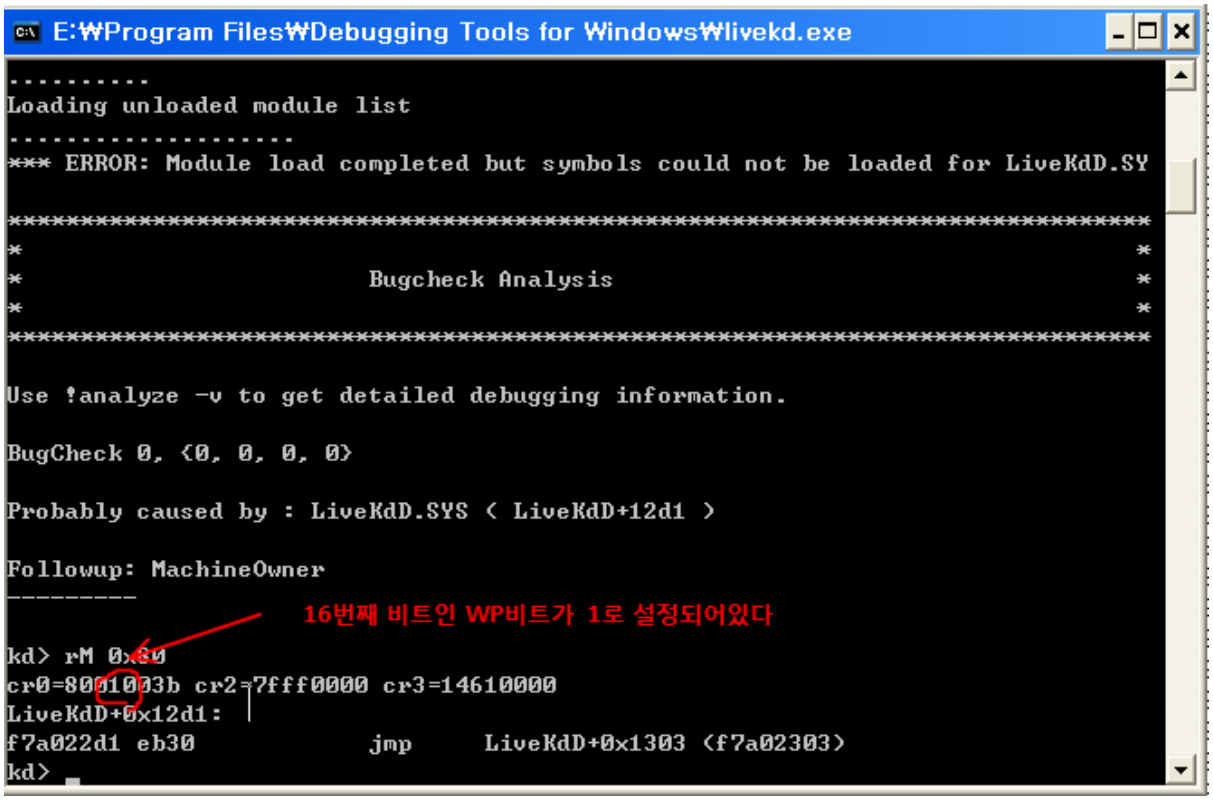

그림10. LiveKd로 CR0 레지스터값 확인

LiveKd에서 제어 레지스터(Control Register)의 값을 Windbg로 확인하는 명령어는 rM 0x80입니다. 이 명령을 사용하면 CR0, CR2, CR3 레지스터의 값이 출력됩니다. 우리가 주목해야 할 것은 CR0 레지스터이고, 이 값은 8001003b라는 값을 갖습니다. 이 값에서 중간의 1이라는 숫자가 WP비트가 포함되어있고, 이것은 WP비트가 1로 설정되어 있다는 이야기입니다. 이제 LiveKd를 종료한 후 드라이버 구동을 시키고 다시 LiveKd로 CR0 레지스터 값 확인을 통해 WP비트의 변화를 살펴보겠습니다.

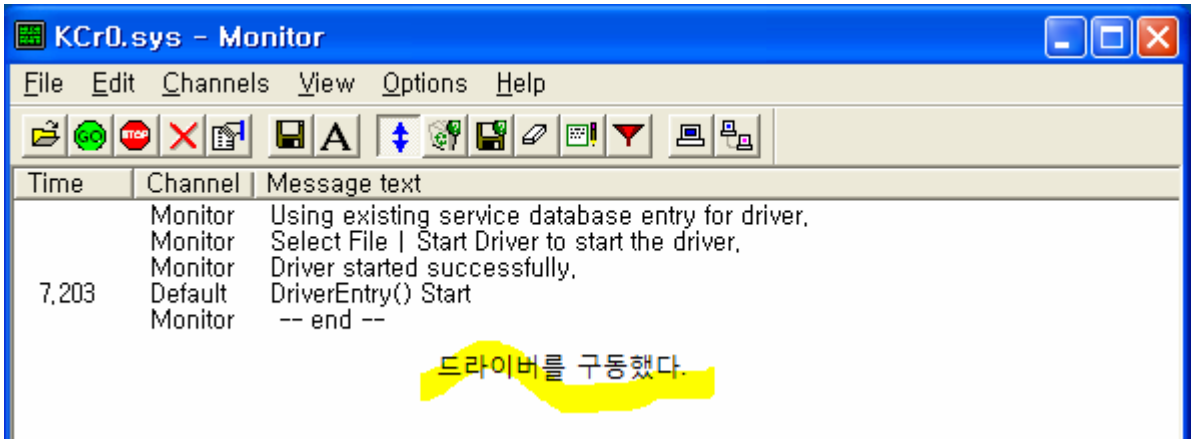

그림11. 드라이버를 구동한다.

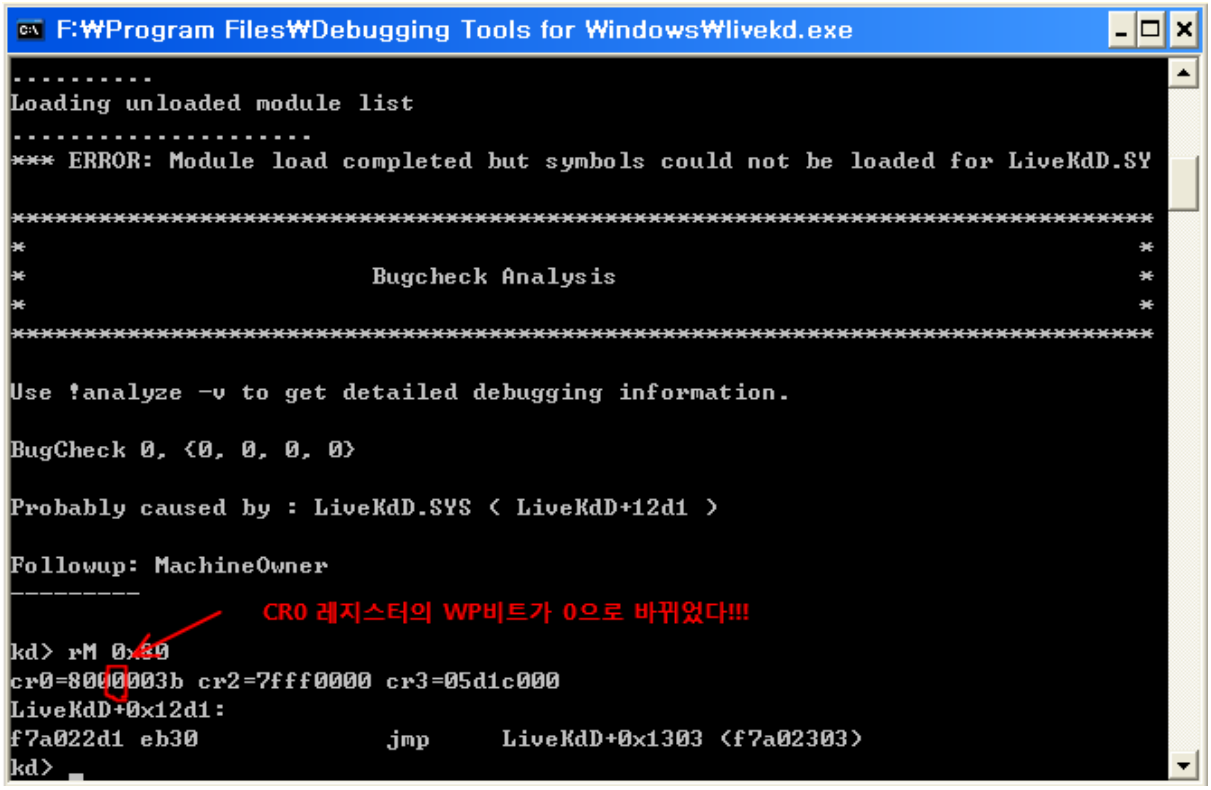

그림12. LiveKd로 바뀐 WP비트 확인

드라이버를 구동시키고, LiveKd를 다시 구동시켜서 CR0 레지스터를 확인한 결과 WP비트가 바뀌었습니다. 이제 SSDT나 IDT의 후킹을 할 수 있는 근간이 마련된 것입니다.

4. 마치는 말

지금까지 API 호출 과정을 유저 레벨과 커널 레벨에서 분석해보고, WDM 디바이스 드라이버를 이용하여 간단한 커널 기반 루트킷을 작성하는 방법을 살펴보았습니다. 아마 이 과정에서 빌드 과정이나 드라이버 프로그래밍 과정을 자세히 기술하려고 노력했기 때문에 특별히 어려운 부분은 없었을 것이라고 생각합니다. 하지만 드라이버 빌드 환경 설정하는 부분을 자세히 기술하지 않았기 때문에 그것은 뒤에 도움이 될 만한 사이트들을 올리는 것으로 대신하겠습니다.

이제 다음에 연재할 문서에는 DKOM 기법을 다뤄보면서 기본적인 드라이버의 틀에서 벗어나 드라이버 프로그래밍을 좀더 깊이 들어가겠습니다. DKOM이란 것을 간단히 소개하면, Direct Kernel Object Manipulation의 약자로서 Object Manager를 거치지 않고 커널 오브젝트를 직접적으로 수정하게 하는 커널 해킹 기법입니다.

5. References

5-1. 참고문헌

[1] Greg Hoglund, James Butler, Rootkits : Subverting the Windows Kernel (book)
[2] 이봉석, 고급개발자들만이 알고 있던 디바이스 드라이버 구조와 원리 그리고 제작 노하우

(book)
[3] Sven B.Schreiber, Undocumented Windows 2000 Secrets - A Programmer's Cookbook (book)
[4] 정덕영, Windows 구조와 원리 2판 (book)
[4] 드라이버 쪼물딱 거리기 3탄, http://somma.egloos.com/2731001
[5] rootkit.com, http://www.rootkit.com
[6] 드라이버온라인, http://www.driveronline.org

5-2. 도움이 될 만한 사이트

[1] http://www.rootkit.com - Rootkits책의 저자인 Grug Hoglund가 운영하는 곳으로 전세계의 해커들이 커널 해킹과 루트킷에 관련된 최신기술이나 정보를 교류하는 곳이다.
[2] http://www.driveronline.org - 윈도우 디바이스 드라이버 개발자 커뮤니티로서 이 분야에서는 우리나라에서 가장 규모가 큰 곳이다.
[3] http://www.kosr.org - 역시 윈도우 디바이스 드라이버 개발자 커뮤니티로서 상당히 많은 정보가 있다.
[4] http://www.osronline.org - 전세계에서 가장 큰 윈도우 디바이스 드라이버 개발자 커뮤니티로서 굉장히 많은 자료와 정보가 존재한다.
[5] http://www.microsoft.com/whdc/ - Microsoft에서 윈도우 디바이스 드라이버 개발자들에게 도움을 줄 수 있도록 많은 자료와 정보를 제공하는 사이트.
[6] http://www.microsoft.com/technet/sysinternals/ - 지금은 Microsoft에 인수된 Sysinternals 사이트. LiveKd와 같이 드라이버 개발 시에 유용한 툴과 소스가 있다.
[7] http://goedel.chonbuk.ac.kr/wcham/ - 함운철교수님의 개인 홈페이지. 커널 디버깅 관련자료들이 많다.
[8] http://www.zap.pe.kr - 여리님의 개인홈페이지. 커널 해킹에 관한 몇 가지 결과물과 아이디어가 있다.
[9] http://dualpage.muz.ro - 듀얼님의 개인 블로그.
[10] http://somma.egloos.com - somma님의 개인 블로그.