KIDS – Kernel Intrusion Detection System

Troopers 2008

Rodrigo Rubira Branco

(BSDaemon)

<rodrigo *noSPAM* kernelhacking . com>

<rodrigo *noSPAM* risesecurity . org>

Munich - Germany, 04/23/2008

Disclaimer

This presentation is just about issues I have worked on in my own time, and is NOT related to the company ideas, opinions or works.

I'm just a security guy who work for a big company and in my spare time I do security research.

My main research efforts are in going inside the System Internals and trying to create new problems to be solved

Agenda

- Motivation – Kernel Protection Challenges
- Tools that try to act on this issues and their vulnerabilities
- Differences between protection levels (software / hardware)
- StMichael – what it actually does
- The Proposal – SMM Internals
- Comments on efforts of breaking the ideas
- Intel and PowerPC Protection Resources
- Questions and Astalavista baby :D

Motivation

- Linux is not secure by default (I know, many *secure* linux distributions exist...)
- Most of efforts till now on OS protection don't really protect the kernel itself
- Many (a lot!) of public exploits were released for direct kernel exploitation
- Beyond of the fact above, it is possible to bypass the system's protectors (such as SELinux)
- After a kernel compromise, life is not the same (never ever!)

Motivation

- Intel platform (not talking about virtualization) supports 4 different privilege leves: from ring0 to ring3
- Most of current security systems try to protect ring3 (user-land) jump to ring0 (kernel-land). Eg: PatchGuard, PaX
- Security systems running on ring0 and malicious code running on ring0 are always fighting for “who arrives first” - Inside ring0 everything is a mess
- Few efforts have been done to protect the kernel itself against other malicious code that is running on the kernel

Userland protections

I loved this picture from Julie Tinnes presentation on Windows HIPS evaluation with Slipfest

Breaking into security systems – SELinux & LSM

Spender's public exploit (null pointer dereference is a sample):

- get_current
- disable_selinux & lsm
- change gids/uids of the current
- chmod /bin/bash to be suid

Disabling SELinux & LSM

disable_selinux

- find_selinux_ctxid_to_string()

/* find string, then find the reference to it, then work backwards to find a call to selinux_ctxid_to_string */

What string? "audit_rate_limit=%d old=%d by auid=%u subj=%s"

- /* look for cmp [addr], 0x0 */
then set selinux_enable to zero

- find_unregister_security();

What string? "<6>%s: trying to unregister a"
Than set the security_ops to dummy_sec_ops ;)

PaX Details – Kernel Protections

- KERNEXEC
* Introduces non-exec data into the kernel level
* Read-only kernel internal structures

- RANDKSTACK
* Introduce randomness into the kernel stack address of a task
* Not really useful when many tasks are involved nor when a task is ptraced (some tools use ptraced childs)

- UDEREF
* Protects agains usermode null pointer dereferences, mapping guard pages and putting different user DS

The PaX KERNEXEC improves the kernel security because it turns many parts of the kernel read-only. To get around of this an attacker need a bug that gives arbitrary write ability (to modify page entries directly).

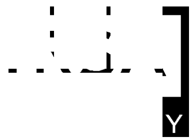
Changing page permissions (writing in a pax protected kernel)

```
static int change_perm(unsigned int *addr)
{
    struct page *pg;
    pgprot_t prot;
    /* Change kernel Page Permissions  */
    pg = virt_to_page(addr); /* We may experience some problems in RHEL 5 because it
   uses sparse mem */
    prot.pgprot = VM_READ | VM_WRITE | VM_EXEC;  /* 0x7 - R-W-X */
    change_page_attr(pg, 1, prot);
    global_flush_tlb();  /* We need to flush the tlb, it's done reloading the value in cr3 */
    return 0;
}//
```

StMichael uses this code to change kernel pages to RO

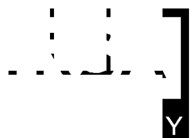
Changing page permissions (writing in a pax protected kernel)

```
void disable_write_protection( void );
asm("        .text                                  ");
asm("        .type disable_write_protection, @function        ");
asm("cli");                          // disable interrupts
asm("mov %cr0, %eax");
asm("mov $0x10000, %ebx");
asm("notl %ebx");
asm("andl %ebx, %eax"); // disable WP bit in cr0
asm("mov %eax, %cr0");
)
```



Actual Problems

- Security normally runs on ring0, but usually on kernel bugs attacker has ring0 privilleges
- Almost impossible to prevent (Joanna said we need a new hardware-help, really?)
- Lots of kernel-based detection bypassing (forensic challenge)
- Detection on kernel-based backdoors or attacks rely on “mistakes” made by attackers

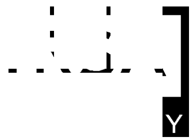


Introducing StMichael

- Generates and checks MD5 and, optionally, SHA1 checksum of several kernel data structures, such as the system call table, and filesystem call out structures;
- Checksums (MD5 only) the base kernel, and detect modifications to the kernel text such as would occur during a silvo-type attack;
- Backups a copy of the kernel, storing it in on an encrypted form, for resto- ring later if a catastrophic kernel compromise is detected;
- Detects the presence of simplistic kernel rootkits upon loading;
- Modifies the Linux kernel to protect immutable files from having their immutable attribute removed;
- Disables write-access to kernel memory through the /dev/{k}mem device;
- Conceals StMichael module and its symbols;
- Monitors kernel modules being loaded and unloaded to detect attempts to conceal the module and its symbols and attempt to "reveal" the hidden module.
- Uses encrypted messages to avoid signature detection of its code
- Random keys
- MBR Protection

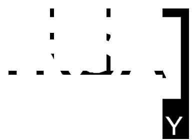
Optimization

- **Many efforts are needed to accomplish code optimization**
- **I already do Lazy TLB:**
  - **When my threads executes, I copy the old active mm pointer to be my own pointer**
  - **Doing so, the system does not need to flush the TLB (one of the most expensive things)**
  - **Because the system just touch kernel-level memory, I don't need to care about wrong resolutions**
  - **That's why I cannot just protect the user-mode memory**



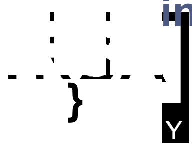
Efforts on bypassing StMichael

- Julio Auto at H2HC III proposed an IDT hooking to bypass StMichael
- Also, he has proposed a way to protect it hooking the init_module and checking the opcodes of the new-inserted module
- It has two main problems:
  - Can be easily defeated using polymorphic shellcodes
  - Just protect against module insertion not against arbitrary write (main purpose of stmichael)



Hooking IDT

```
/* To load the new value */
void load_myidt( void *value )
{
        asm("     lidtl  %0 " : : "m" (*(unsigned short*)value) );
}


/* To handle the interrupts */
asmlinkage void our_handler( unsigned long *interrupt_info )
{
        struct task_struct  *p = current;
        int    cpu = task_cpu( p )&1;  /* identify the processor
        int    i = interrupt_info[ 10 ];         /* identify the interrupt */
        interrupt_info[ 10 ] = old_table[ i ];      /* setup the original handler */
}
```



Hooking IDT

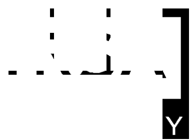
```
void our_entry( void );
asm("      .text                        ");
asm("      .type our_entry, @function   ");
asm("      .align 16                    ");
asm("our_entry:                         ");
asm("      i = 0;                       ");
asm("      .rept  256                   ");
asm("      pushl $i                     ");
asm("      jmp   ahead                  ");
asm("      i = i+1                      ");
asm("      .align 16                    ");
asm("      .endr                        ");
asm("ahead:                             ");
asm("      ret                          ");
```

```
asm("      pushal                ");
asm("      pushl     %ds         ");
asm("      pushl     %es         ");
asm("      mov       %ss, %eax");
asm("      mov       %eax, %ds");
asm("      mov       %eax, %es");
asm("      push      %esp        ");
asm("      call      our_handler");
asm("      addl      $4, %esp    ");
asm("      popl      %es         ");
asm("      popl      %ds         ");
asm("      popal                 ");
asm("      ret                   ");
```

Proposed solutions against it

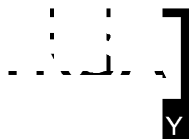
- **Julio Auto proposed statical memory analysis as solution – but, what about polymorphic code? :**

```
asm("jmp label3          \n\
label1:                  \n\
popl %%eax               \n\
movl %%eax, %0           \n\
jmp label2               \n\
label3:                  \n\
call label1              \n\
label2:" : "=m" (address));
```



Memory cloaking

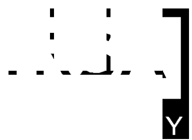
- As exposed by Sherri Sparks and Jamie Butler in the Shadow Walker talk at Blackhat and already used by PaX project, the Intel architecture has splitted TLB's for data and code execution
- Someone can force a TLB desynchronization to hide kernel-text modifications from our reads (I explained more about that in HITB Malaysia talk)
  - This technique relies in the page fault handler patch, since I protect the hardware debug registers and also I check the default handler, it cannot be used to bypass StMichael.



Efforts on bypassing StMichael

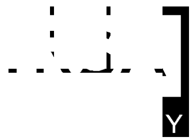
- The best approach (and easy?) way to bypass StMichael is:
  - Read the list of VMA's in the system, detecting the ones with execution property enabled in the dynamic memory section
  - Doing so you can spot where is the StMichael code in the kernel memory, so, just need to attack it...

That's the motivation in the Joanna's comment about we need new hardware helping us... but...



Where do I want to go? My Proposal

- StMichael must be a SW independent of other set of programs that try to defend the system
- I will put another layer of protection between the system's auditors/protectors/verifiers and the hardware
- Are the researchers wrong about the impossibility of protecting the O.S. without a hw-based solution?

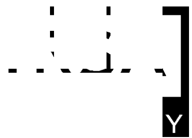


How? SMM!

SMM – System Management Mode

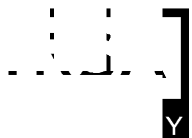
The Intel System Management Mode (SMM) is typically used to execute specific routines for power management. After entering SMM, various parts of a system can be shut down or disabled to minimize power consumption. SMM operates independently of other system software, and can be used for other purposes too.

From the Intel386tm Product Overview – intel.com

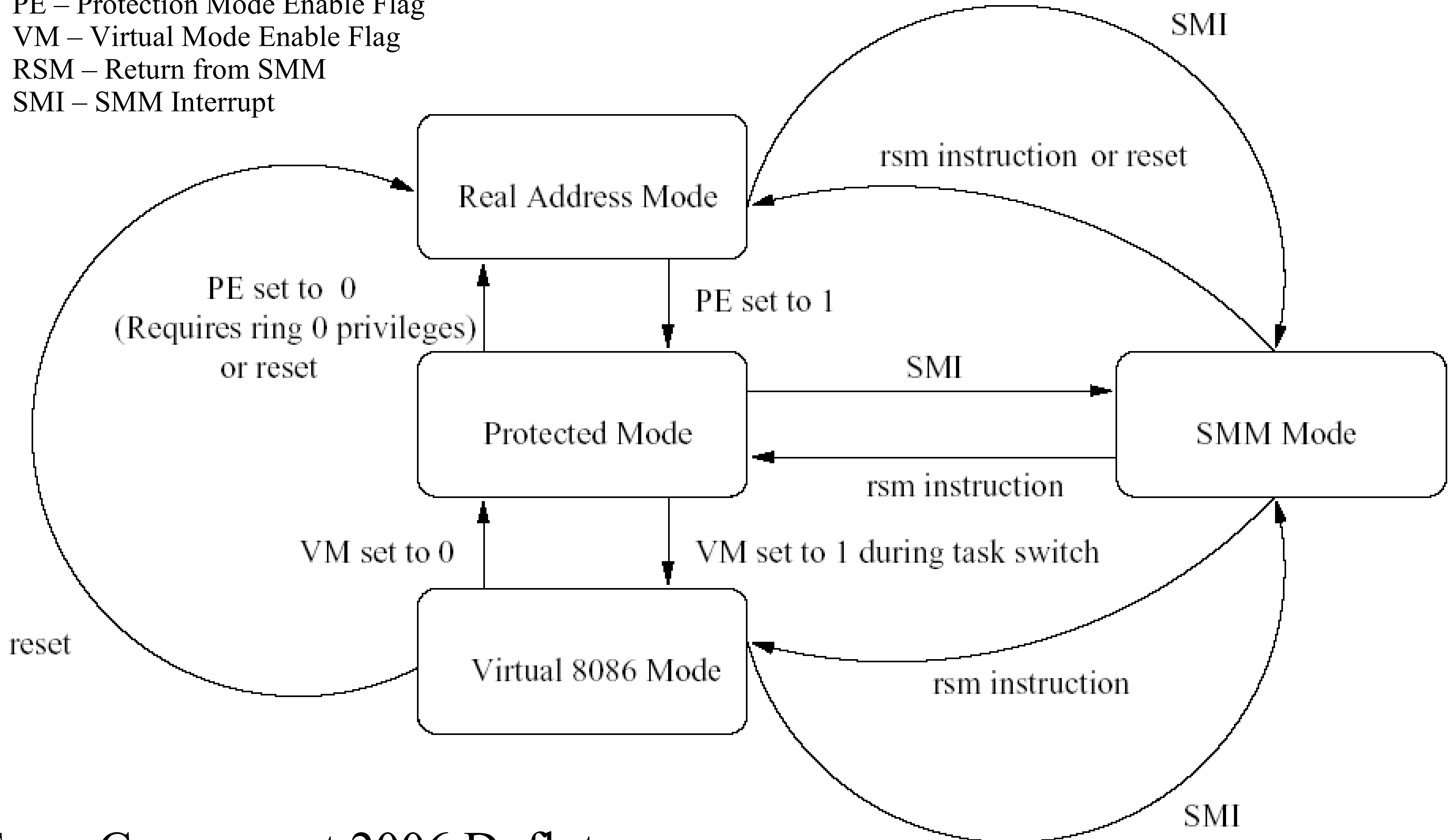


How does it work?

- Chip is programmed to grab and recognize many type of events and timeouts
- When this type of event happens, the chipset gets the SMI (System Management Interrupt)
- In the next instruction set, the processor saves it owns state and enters SMM
- When it receives the SMIACT, redirects all next memory cycles to a protected area of memory (specially reserved for SMM)
- Received SMI and Asserted the SMIAct output? -> save internal state to protected memory
- When contents of the processor state are fully in protected memory area, the SMI handler begins to execute (processor is in real-mode with 4gb segments limit)
- SMM Code executed? Go back to the previous enviroment using the RSM instruction



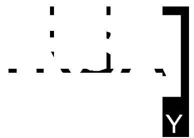
Context switches



From Cansecwest 2006 Duflot

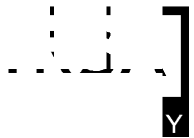
SMM Resources

- No paging – 16 bits addressing mode, but all memory accessible using memory extension addressing
- To enter SMM, need an SMI
- To leave the SMM, need the RSM instruction
- **When entering in SMM, the processor will save the actual context** – so, can leave it in any portion of the address space wanted – see more ahead
- SMM runs in a protected memory, at SMBASE and called SMRAM



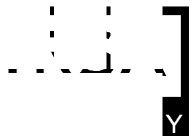
SMM Details

- SMM registers can be locked setting the D_LCK flag (bit 4 in the MCH SMM register)
- SMI_STS contains the device who generated the SMI (write-reset register)
- In the NorthBridge, the memory controller hub contains the SMM control register – the bit 6, D_OPEN, specifies that access to the memory range SMRAM will go to SMM and not for the I/O port
- The BIOS may set the D_LCK register, if so, we need to patch the BIOS too (tks to the LinuxBIOS project, it's pretty easy)



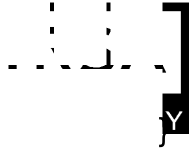
PCI Configuration

- **Two I/O Port ranges:**
  - Data Port : 0xCFC-0xCFF
  - Address Port: 0xCF8-0xCFB
- **Write the device and register you want to acces in the address port and read/write the data to/from data port**
- **The addressing are:**
  - PCI bus
  - Device
  - Function



Setting the D_OPEN bit

```
int open_smram(void)  {
        struct pci_access *pacc; struct pci_dev *smram_dev; u8 current_value;
        pacc = pci_alloc(); pci_init(pacc);
        smram_dev = pci_get_dev(pacc, 0, 0, 0, 0);
        current_value = pci_read_byte(smram_dev, SMRAM_OFFSET);
        if(current_value & D_OPEN_BIT) {  /* D_OPEN_BIT was set */
                pci_cleanup(pacc); /* close everything */
                return 0;
        } else {  /* D_OPEN_BIT is not set, we must set it */
                pci_write_byte(smram_dev, SMRAM_OFFSET, (current_value | D_OPEN_BIT));
                current_value =  pci_read_byte(smram_dev, SMRAM_OFFSET);
                if(current_value & D_OPEN_BIT) {   /* D_OPEN_BIT was set */
                     pci_cleanup(pacc); /* close everything */
                     return 1;
                } else {  /* it was not able not set D_OPEN */
                     pci_cleanup(pacc);
                     return -1;
                }
        }
        return -1;
}
```



The SMM Handler

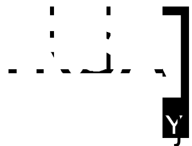
```
asm (            ".data"                                );
asm (            ".code16"                       );
asm (            ".globl handler, endhandler"    );
asm (            "\n" "handler:"                 );
asm (            " addr32 mov $stmichael, %eax"  ); /* Where to return */
asm (            " mov %eax, %cs:0xfff0"                ); /* Writing it in the save EIP  */

/* Check the integrity of the called code and save the current state */

asm (            " rsm"                                 ); /* Switch back to protected mode */
asm (            "endhandler:"                          );
asm (            ".text"                                );
asm (            ".code32"                       );
```

Copying the handler to the SMRAM

```
int fd;

unsigned char *vidmem;

fd = open(MEMDEVICE, O_RDWR);

vidmem = mmap(NULL, MAPPEDAREASIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
    SMIINSTADDRESS);

close(fd);

if(vidmem != memcpy(vidmem, handler, endhandler-handler)) {
        printf("Could not copy asm to memory...\n");
        exit(EXIT_FAILURE);

}

if(munmap(vidmem, MAPPEDAREASIZE) < 0) {
        printf("Could not release mapped area, errno: %d\n", errno);
        exit(EXIT_FAILURE);

}

if(close_smram() < 0) {
        printf("Could not close SMRAM. Abort.\n");
        exit(EXIT_FAILURE);

}
```

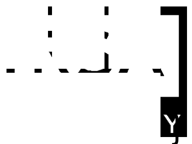
Enabling the #SMI globally

```
int enable_smi_gbl(u16 smi_en_iop)
{
        u32 smi_en_value;
        iopl(3);
        smi_en_value = inl(smi_en_iop);
        if(smi_en_value & GBL_SMI_EN_BIT) {   /* gbl_smi_en was set */
                return 0;
        } else {
                outl(smi_en_value | GBL_SMI_EN_BIT, smi_en_iop);
                smi_en_value = inl(smi_en_iop);
                if(smi_en_value & GBL_SMI_EN_BIT) {    /* gbl_smi_en is set */
                      return 1;
                } else {      /* gbl_smi_en cannot be set */
                      return -1;
                }
        }
        /* we should never reach this */
        return -1;
}
```



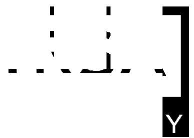
Enabling #SMI when writing to APM_CNT

```
int enable_smi_on_apm(u16 smi_en_iop)
{
        u32 smi_en_value;
        iopl(3);
        smi_en_value = inl(smi_en_iop);
        if(smi_en_value & APMC_EN_BIT) {    /* apmc_en was set */
                return 0;
        } else {
                outl(smi_en_value | APMC_EN_BIT, smi_en_iop);
                smi_en_value = inl(smi_en_iop);
                if(smi_en_value & APMC_EN_BIT) {    /* apmc_en is set */
                    return 1;
                } else {    /* apmc_en cannot be set */
                    return -1;
                }
        }
        /* we should never reach this */
        return -1;
}
```



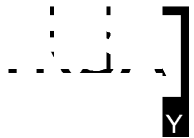
Generating #SMI

```
//0xb2 is the APM_CNT I/O
asm(
        "inb $0xb2,%al\n"
        "movb $0xff, %al\n"
        "outb %al, $0xb2\n"
);
```



SMM locking

- As said SMM registers can be locked setting the D_LCK flag (bit 4 in the MCH SMM register). After that, control registers are locked and cannot be changed, lacking of a reboot for that
- I need also to lock the SMI_EN (otherwise, someone can just disable the #SMI)



SMM locking

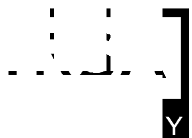
```
int lock_smram(void) {
    struct pci_access *pacc;    struct pci_dev *smram_dev;    u8 current_value, orig_value;
    pacc = pci_alloc();    pci_init(pacc);
    smram_dev = pci_get_dev(pacc, 0, 0, 0, 0);
    current_value = pci_read_byte(smram_dev, SMRAM_OFFSET);
    orig_value = current_value;
    /* lock it if not locked */
    if(!(current_value & D_LCK_BIT))    pci_write_byte(smram_dev, SMRAM_OFFSET, (current_value | D_LCK_BIT));
    /* then try to unlock it */
    pci_write_byte(smram_dev, SMRAM_OFFSET, (current_value & ~D_LCK_BIT));
    current_value = pci_read_byte(smram_dev, SMRAM_OFFSET);
    /* is locked and cannot be unlocked */
    if(current_value & D_LCK_BIT)    return 0;
    /* D_LCK_BIT is not set and could be not set  */
    return -1;
}
```

Studying the SMM

```
u8 show_smram(struct pci_dev* smram_dev, u8 bits_to_show)
{
        struct pci_access *pacc;
        struct pci_dev *smram_dev_default;
        u8 current_value;
        if (! smram_dev) {
                pacc = pci_alloc();
                pci_init(pacc);
                smram_dev_default = pci_get_dev(pacc, 0, 0, 0, 0);
        } else smram_dev_default=smram_dev;
        current_value = pci_read_byte(smram_dev_default, SMRAM_OFFSET);
        printf("Current value in SMRAM: 0x%04x\n", current_value);
        if(RESERVED0_BIT & bits_to_show)
                printf("RESERVED0_BIT: %d\n", (current_value & RESERVED0_BIT) ? 1 : 0);
        if(D_OPEN_BIT & bits_to_show)
```

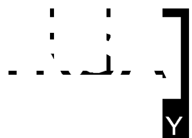
Compability Problems

- **Yeah, there is SMM just in the Intel platform... but:**
  - **Many platforms already supports something like firmware interrupts**
  - **Although any platform have some way to instrument it to debug against hardware problems -> I covered some difficulties for Power platforms in the Xcon/China (next slides)**



How interrupts are handled

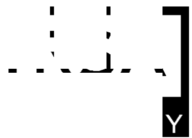
- Here I will try to cover two different platforms: Intel and PowerPC
- The general idea is to begin showing how my model can be expanded to other architectures (Like Power, which does not have System Management Mode in the same way as the Intel arch)
- Interruptions are handled in different ways by different platforms



Intel Platform – system calls

- Two different ways:
  - Software interrupt 0x80
  - Vsyscalls (newer PIV+ processors – calls to user space memory (vsyscall page) and using sysenter and sysexit functions

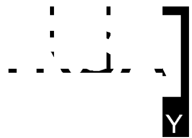
- To create the system call handler, the system does: set_system_gate(SYSCALL_VECTOR,&system_call)
  - This is done in entry.S and creates a user privilege descriptor at entry 128 (the syscall_vector) pointing to the address of the syscall handler (in that case, system_call)



Power Platform – system calls

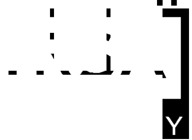
- PPC interrupt routines are anchored to fixed memory locations
- In head.S the system does:

```
. = 0xc00
SystemCall:
EXCEPTION_PROLOG
EXC_XFER_EE_LITE(0xc00, DoSyscall)
```



Intel Platform – Time interrupts

- Historically used a cascaded pair of Intel 8259 interrupt controllers
- Now, most of the system uses APIC, which can emulate the old behavior
- Each interrupt on x86 is assigned a unique number, known as vector.
- At the interrupt time, this vector is used as index to the Interrupt Descriptor Table (IDT)
- Uses the Intel 8254 timer with a Programmable Interval Timer (PIT) – 16-bit down counter – activate an interrupt in the IRQ0 of the 8259 controller

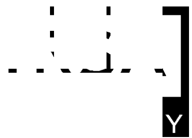


Power Platform – Time interrupts

- Power uses a 32 bit decrementer, built-in in the CPU (running in the same clock)
- The timer handler is located at the fixed address 0x900:
  - In head.S:

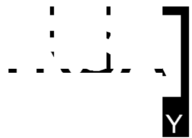
  EXCEPTION(0x900, Decrementer, timer_interrupt, EXC_XFER_LITE)

- External interrupts comes in the fixed address 0x500 and are treated in a similar way to the intel IDT jump



PowerPC Kernel Protection

- The idea of putting the entire kernel as read-only seems good
- The attacker cannot modify the pages permissions, since I can use watchpoints to monitor that
- There is no IDT, so if the attacker cannot touch the memory, everything is protected??
- But... life cannot be perfect...



PowerPC Protection Problems

- **From the manual:**

"The optional data address breakpoint facility is controlled by an optional SPR, the DABR. The data address breakpoint facility is optional to the PowerPC architecture. However, if the data address breakpoint facility is implemented, it is recommended, but not required, that it be implemented as described in this section."

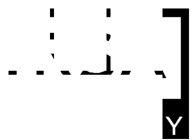
The architecture does not include execution breakpoints too.

PowerPC 32 Debugging...

DAB		BT	DW	DR
0	28	29	30	31

0–28 DAB Data address breakpoint
29 BT Breakpoint translation enable
30 DW Data write enable
31 DR Data read enable

A match will generate a DSI Exception, which you can check in the DSISR register bit 9 (set if it is a DABR match)



PowerPC 4xx Study

- Debug Control Registers: DBCR 0-2
- Data Address Compare Registers: DAC 1-2
- Instruction Address Compare Registers: IAC 1-4
- Data Value Compare Registers: DVC 1-2

Detail: A patch has been sent to the linux kernel to include the DAC support. In anyway, it can be used directly just using the mtspr instruction to load the specified address in the register

Detail2: Cache management instructions are treated as 'loads', so will trigger the watchpoints

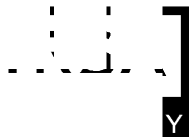
Detail3: Platform also supports Watchdogs, but if the interrupts are disabled, they will not trigger in anyway

PPC 4xx Study

- Supports different conditions:
  - DBCR0[RET]=1 – Return exception
  - DBCR0[ICMP]=1 – Instruction completion
  - DBCR0[IRPT]=1 – Interruption
  - DBCR0[BRT]=1 – Branch
  - DBCR0[FT]=1 – Freeze the decrementer timers
  - Others...
- To enable debug interrupts:
  - MSR[DE] = 1 and DBCR0[IDM]=1
- Using the IAC (DBCR1[IAC1ER, IAC2ER, IAC3ER, IAC4ER]) I can choose to monitor the effective or the real address
- I can also instrument an external debug system, setting DBCR0[EDM] to 1 and using a JTAG interface

PPC 405EP and Firmware instrumentation

- I2C interface between the real system and the embedded processor
- PowerPC Initialization Boot Software (PIBS). Source code is provided.
- Embedded PowerPC Operating System (EPOS). Source code is provided.
- Not a hackish, it's offered by the companies ;)
- cpc925_read addr numbytes and cpc925_read_vfy addr numbytes mask0[.mask1] data0[.data1] commands



PPC 405EP and Firmware instrumentation

- From the manual:

“Synopsis

Read and display memory in the PPC970FX address space using the PPC405EP service processor. The service processor accesses the CPC925 processor interface via its connection to the CPC925 I2C slave.

Command Type

PIBS shell command or initialization script command.

Syntax

cpc925_read addr numbytes

Parameters

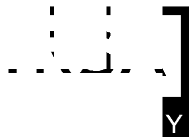
addr The least significant 32 bits of the 36 bit PPC970FX physicaladdress to read. The 4 most significant physical address bits are

assumed to be zero.

numbytes The number of bytes to read and display.“

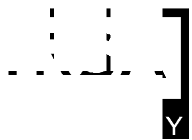
SMM and Anti-Forensics?

- Duflot paper released a way to turn off BSD protections using SMM
- A better approach can be done using SMM, just changing the privilege level of a common task to RING 0
- The segment-descriptor cache registers are stored in reserved fields of the saved state map and can be manipulated inside the SMM handler
- Someone can just change the saved EIP to point to his task and also the privilege level, forcing the system to return to his task, with full memory access
- Since the SMRAM is protected by the hardware itself, it is really difficult to detect this kind of rootkit



Descriptor Cache

- From the Intel Manual: "Every segment register has a "visible" part and a "hidden" part. (The hidden part is sometimes referred to as a "descriptor cache" or a "shadow register.") When a segment selector is loaded into the visible part of a segment register, the processor also loads the hidden part of the segment register with the base address, segment limit, and access control information from the segment descriptor pointed to by the segment selector. "
- RPL – Request Privilege Level
- CPL – Current Privilege Level
- DPL – Descriptor Privilege Level

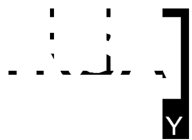


Descriptor Cache

- **In the saved state map (inside SMM – this values differ from Intel Manual just because I tested in an old machine):**
- TSS Descriptor Cache (12-bytes) - Offset: 7FA4
- IDT Descriptor Cache (12-bytes) - Offset: 7F98
- GDT Descriptor Cache (12-bytes) - Offset: 7F8C
- LDT Descriptor Cache (12-bytes) - Offset: 7F80
- GS Descriptor Cache (12-bytes) - Offset: 7F74
- FS Descriptor Cache (12-bytes) - Offset: 7F68
- DS Descriptor Cache (12-bytes) - Offset: 7F5C
- SS Descriptor Cache (12-bytes) - Offset: 7F50
- CS Descriptor Cache (12-bytes) - Offset: 7F44
- ES Descriptor Cache (12-bytes) - Offset: 7F38

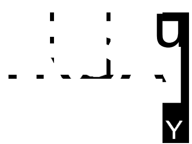
Future

- Some advanced hardware, like pSeries support firmware services to abstract portions of the hardware of the operating system
- pSeries for example has the RTAS (run-time abstraction service) to easily access NVRAM and heartbeat mechanics
- This operating system running in the firmware maybe modified to offer integrity verification



Other approaches

- PaX KernSeal – compiler modifications – not released yet
- Maryland Info-Security Labs Co-pilot and others (firewire, tribble, etc) – PCI Card to analyze the system integrity – cache/relocation attacks, Joanna ideas, hardware based
- Intel System Integrity Services – SMM-based implementation – depends on external hardware (also uses client/server signed heartbeats)
- Microsoft PatchGuard – Self-encryption and kernel instrumentation – many problems spotted by uninformed.org articles



REFERENCES

Spender public exploit:
http://seclists.org/dailydave/2007/q1/0227.html

Pax Project:
http://pax.grsecurity.net

Joanna Rutkowska:
http://www.invisiblethings.org

Julio Auto @ H2HC – Hackers 2 Hackers Conference:
http://www.h2hc.org.br

A Tamper-Resistant, Platform-Based, Bilateral - INTEL
Approach to Worm Containment

Runtime Integrity and Presence Verification for
Software Agents - INTEL

BIOS and Kernel Developer´s Guide for AMD Athlon 64 and AMD Opteron
Processors - AMD

Intel Architecture Software Developer´s Manual
Volume 3: System Programming

Security Issues Related to Pentium System Management Mode
Loïc Duflot

Acknowledges

Spender for help into many portions of the model

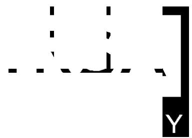
PaX Team for solving doubts about PaX and giving many help point directly to the pax implementation code

All conferences that trusted me as a speaker

Special tks to Troopers organizers, for give me a chance to show this research results!

Your patience!

Let's stop this bullshit and drink ;D



Just kidding, I know it's morning!

End! Really is?

Questions?

Thank you :D

Rodrigo Rubira Branco

(BSDaemon)

<rodrigo *noSPAM* kernelhacking . com>

<rodrigo *noSPAM* risesecurity . org>