KIDS – Kernel Intrusion Detection System

YSTS 2007 – Brazil

Rodrigo Rubira Branco

<rodrigo@kernelhacking.com>

<rodrigo@risesecurity.org>

São Paulo - Brazil, 10/24/2007

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
- Our Proposal – SMM Internals
- Comments on efforts of breaking our 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 kcrash kernel**

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 (see more ahead) 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.

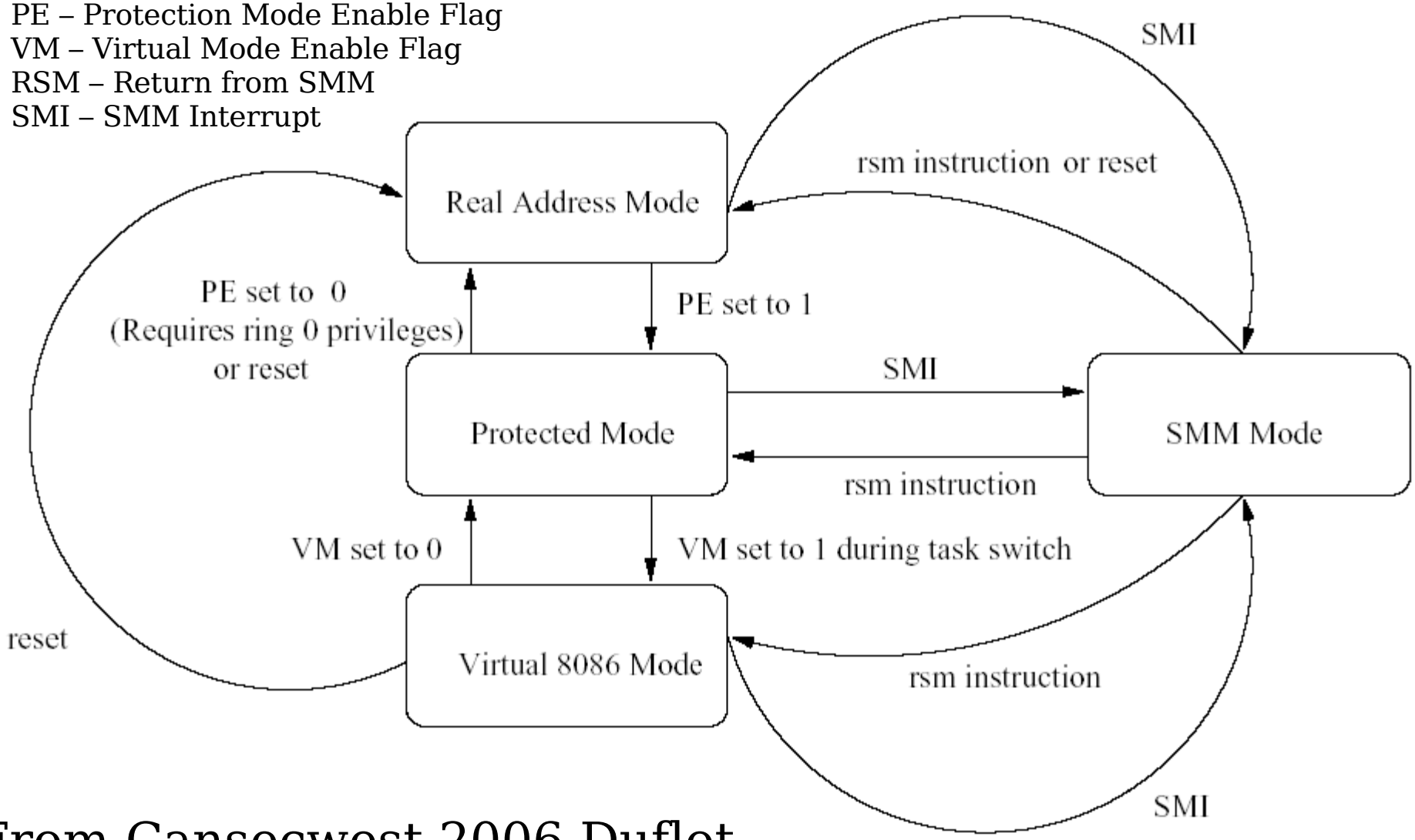
From the Intel386™ Product Overview – intel.com

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

PE – Protection Mode Enable Flag
VM – Virtual Mode Enable Flag
RSM – Return from SMM
SMI – SMM Interrupt



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)

Generating an SMI event

- **There is many possibilities:**
  - **Using ACPI events (do you remember hibernation and sleep)**
  - **Using an external #SMI generator in the bus**
  - **Some systems (AMD Geode?) are always generating this kind of interrupt**
  - **Writing to a specific I/O port also generates an #SMI**
    - **This can be used to instrument the system to generate #SMI events in some situations – compiler modifications, statical patch – need to be done yet – SystemTAP gurus wanted**

Generating an SMI event - deeper

- All memory transactions from the CPU are placed on the host bus to be consumed by some device
  - Potentially the CPU itself would decode a range such as the Local APIC range, and the transaction would be satisfied before needing to be placed on the external bus at all.
- If the CPU does not claim the transaction, then it must be sent out.
  - In a typical Intel architecture, the transaction would next be decoded by the MCH and be either claimed as an address that it owns, or determining based on decoders that the transaction is not owned and thus would be forwarded on to the next possible device in the chain.

Generating an SMI event - deeper

- If the memory controller does not find the address to be within actual DRAM, then it looks to see if it falls within one of the I/O ranges owned by itself (ISA, EISA, PCI).
  - Depending upon how old the system is, the memory controller may directly decode PCI transactions, for example.
- If the MCH determines that the transaction does not belong to it, the transaction will be forwarded on down the chain to whatever I/O bridge(s) may be present in the system. This process of decoding for ownership / response or forwarding on if not owned repeats until the system has run out of potential agents.

Generating an SMI event - deeper

- The final outcome is either an agent claims the transaction and returns whatever data is present at the address, or no one claims the address and an abort occurs to the transaction, typically resulting if 0FFFFFFFFh data being returned.
- In some situations (Duflot paper case), some addresses (sample with the 0A0000h - 0BFFFFh range) are owned by two different devices (VGA frame buffer and system memory) - This will force the Intel architecture to send a SMI signal to satisfy the transaction
- If no SMI asserted, then the transaction is ultimately passed over by the memory controller in favor of allowing a VGA controller (if present) to claim.
- If the SMI signal is asserted when the transaction is received by the memory controller, then the transaction will be forwarded to the DRAM unit for fetching the data from physical memory.

Generating #SMI's

- I explained really deeply why the system will generate #SMI in Xcon this year (past slides)
- Know, I can just instrument the kernel (in any portion of it, so turning really difficult to detect) an I/O operation to a shared address between devices (as Duflot spotted in his paper, 0xA0000h) sounds good
- This idea can be used together with a BIOS rootkit, to configure an SMI handler, lock the SMM (relocating the SMRAM) and then transfering control back to normal boot system – if someday the system triggers a SMI, it will install the backdoor, bypassing all kind of boot protections

Address Translation while in SMM

- The biggest difficulty
  - I need to have the cr3 register value (in x86 systems)
  - I must parse the page tables used by the processor (used by the OS)
  - Using DMA I can read the page tables (do you remember the PGD, PMD and PTE?)
- Maybe I can just read the physical pages used by the kernel and compare it against a 'trusted' version (it doesn't sound good, since sparsemem systems will be really difficult to protect and dynamically generated kernel structures too)
- Another approach is just transfer the control back to my handler in main memory (that's what I'm using now):
  - Need to save the current processor status inside SMM, so after the handler, I can transfer control back

Studying the SMM

```
u32 value;

struct pci_dev   *pointer = NULL;

devp = pci_find_class( 0x060000, devp ); // get a pointer to the MCH

for (i = 0; i < 256; i+=4) {
    pci_read_config_dword( pointer, i, &value );
    <print the information>

}
```

- FreeBSD systems offers to us the pciconf utility, so you can just set the D_OPEN to 1 and then dump the SMRAM memory:

```
    # pciconf -r -b pci0:0:0 0x72
    # pciconf -w -b pci0:0:0 0x72 0x4A
    # dd bs=0x1000 skip=0xA0 count=0x20 if=/dev/mem of=./foo
    # pciconf -w -b pci0:0:0 0x72 0x0A
```

- Also, in Linux you have libpci to help you solve problems when writing to an used I/O port

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"                       );
```

Dangerous

- When entering the SMM, the SMRAM may be overwritten by data in the cache if a #FLUSH occur after the SMM entrance.
- To avoid that I can shadow SMRAM over non-cacheable memory or assert #FLUSH simultaneously to #SMI events (#FLUSH will be served first) – usually BIOS mark the SMRAM range as non-cacheable for us
  - **As non-cacheable by setting the appropriate Page Table Entry to Page Cache Disable (PTE. PCD=1**
  - **Need to compare that against mark the page as non-cacheable by setting the appropriate Page Table Entry to Page Write-Through (PTE.PWT=1) - opinions?**

SMM locking

- As said SMM registers can be locked setting the D_LCK flag (bit 4 in the MCH SMM register). After that, the SMM_BASE, SMM_ADDR and others related are locked and cannot be changed, lacking of a reboot for that
- The SMM has special I/O cycles for processors synchronization. I don't want these to be executed, so I set SMISPCYCDIS and the RSMSPCYCDIS to 1 (prevents the input and output cycle respectively).
- I need also to lock the SMI_EN (otherwise, someone can just disable the #SMI)

SMM locking

- **AMD just call this lock as SMMLOCK (HWCR bit 0), and a fragment code from the LinuxBIOS project shows how simple is to set it:**

```
/* Set SMMLOCK to avoid exploits messing with
   SMM */
msr = rdmsr(HWCR_MSR);
msr.lo |= (1 << 0);
wrmsr(HWCR_MSR, msr);
```

Protecting missing portions

- Where will be my handler? In the memory, so someone can attack it?
  - No, it's checked inside SMM, but, I also want some kind of online protection
- Protection of the memory pages (already supported by PaX)
- Possibility to add watchpoints in memory pages (detect read at VMAs? At our code? Or writes against our system?)
- DR7 Register!

```
3                   1 1 1 1 1 1         0
1                   5 4 3 2 1 0         0
+-----------------+-+-+-+-+-+-+---------+
|                 |T|T|G|I| | |         |
|                 |2|R|D|R| | |         |
+-----------------+-+-+-+-+-+-+---------+
                  | | | |
                  | | | +-- IceBp  1=INT01 causes emulator
                  | | |              to break emulation
                  | | |            0=CPU handles INT01
                  | | +---- General Detect = Yeah, I can spot CHANGES in the Registers
                  | +------ Trace1 1=Generate special address
                  |                cycles after code dis-
                  |                continuities.  On Pentium,
                  |                these cycles are called
                  |                Branch Trace Messages.
                  +-------- Trace2 1=Unknown.
```

Debugging theory in Intel

In Intel platform there is dr0-7 and 2 MSRs (model-specific registers)

If one breakpoint is hit, a #DB – debug exception is generated

The meaning of having MSRs is to remember the last branchs, interruptions or exceptions generated and that have been inserted in the P6 line of Intel

Also, may have TSS T (trap) flag enabled, generating #DB in task changes

MSR contains the offset relative to the CS (code segment) of the instruction

I can also monitor I/O port using debug registers

Debugging theory in Intel

The debug registers can only be accessed by:

- SMM

- Real-address mode

- CPL0

If you try to access a debug register in other levels, it will generate a general-protection exception #GP

The comparison of a instruction address and the respective debug register occurs before the address translation, so it tries the linear address of the position

Debugging implementation

- On dr7 the 13 bit is the “general detect”
- The processor will zero the flag when entering in the debug handler. I need to set it again after exiting my handler.
- The dr6 will be used to check the BD flag (debug register access detected) - bit 13
- So, the BD flag indicates if the next instruction will access a debug register. So, it will be set when I modify (setting it to 1) the general detect flag in the dr7
- I must clean the dr6 after attending the debugging exceptions

Some code (again)

- To get/set debug register values

```
#define get_dr(regnum, val) \
        __asm__ volatile ("movl %%db" #regnum ", %0"  \
                :"=r" (val))
#define set_dr(regnum, val) \
        __asm__ volatile ("movl %0,%%db" #regnum  \
                : /* no output */ \
                :"r" (val))
```

DB registers protection

- When I trigger the #DB I need to know (taken from mood-nt):

- Why it occured.

- If someone is touching the #DB registers, the dr7 bit 13 (set to 1) will generate this exception to me, before executing the instruction.

- So, my handler must parse what is the next instruction (pointed by EIP – the debugger exception handler receives a struct regs)

- Also, I need to set the dr7 again – here I can use some randomization in what point of code I will protect with this registers

- Then, if the instruction is touching the debug registers I can just emulate it or jump to the next instruction adding bytes to EIP

More stuff … did you know?

- To monitor I/O read/write you need to set the CR4 (Control Register 4) DE (debug extensions) flag, which rules how the R/W0 to R/W3 (read/write) bits – (talking about the 16,17,20,21,24,25,28 and 29 bits of the dr7) will be interpreted – these bits rule how to conduct a breakpoint condition
- 00 - break on instruction execution only
- 01 - break on data writes only
- 10 - break on i/o reads or writes
- 11 - break on data reads or writes but not instruction fetches

More stuff ... did you know?

- The fields len0 to len3 (size) - bits 18,19,22,23,26,27,30 e 31 of the dr7 indicate the monitored memory size of the breakpoint (dr0 to 3) being:
- 00 - 1 byte
- 01 - 2 byte
- 10 – not defined
- 11 - 4 bytes
- If set the RWn of the dr7 to 00 (instruction only) must have the len() to “00”, otherwise will have an abnormal and unpredictable behavior (sounds familiar for some SOs, uhn?)
- dr4 and dr5 are reserved for us (the CR4 DE flag will be set to monitor the i/o port too) – If accessed, will generate an invalid-opcode exception (#UD)

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)**

Handling page faults

- void do_page_fault(struct pt_regs *regs, unsigned long error_code) – arch/<arch>/mm/fault.c
  - Get the unaccessible address from cr2
  - Get the address that caused the exception from regs->eip
  - Verify if someone is trying to write in a protected area

Need to care about page access violations, to provide real time detection...

When the system tries to access an invalid memory location, the MMU will generate an exception and the CPU will call the do_page_fault to search the exception table for this EIP (ELF section __ex_table)

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.“

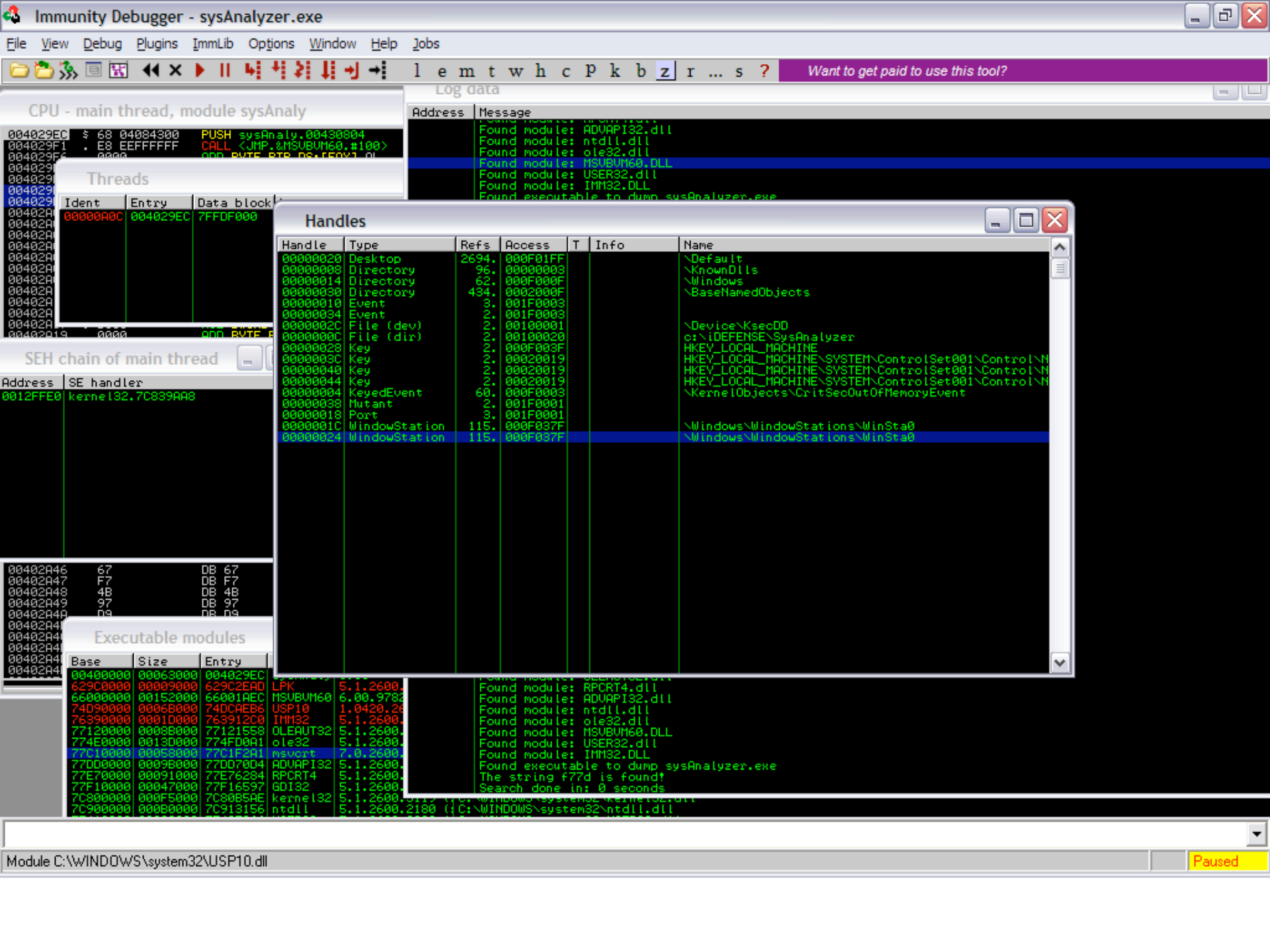
The Kernel War – A little about Anti-Forensics

- **As I showed in the beginning of the presentation, if the attacker compromised the machine and have access to the kernel, a lot of problems will appear:**
  - **He can signature detect the forensics tool:**
    - **Multiple (continuous) memory reads**
    - **Multiple (continuous) disk reads**
  - **Even deeper:**
    - **Binary program signature (like antiviruses use to detect a virus)**
    - **Program behaviour (what the program does? how they does that?)**

Looking for patterns

- **Someone can use the excelent Immunity Debugger with a simple python script to search a binary file for patterns:**

```
allmodules = imm.getAllModules()

for key in allmodules.keys():

    imm.Log("Found module: %s" %key)

usekey = ""

for key in allmodules.keys():

    if key.count(".exe"):

        imm.Log("Found executable to dump %s" %key)

        usekey = key

        break

module_to_dump = allmodules[key]

base = module_to_dump.getCodebase()

size = module_to_dump.getCodesize()

codememory = imm.readMemory(base,size)

hex_codememory = codememory.encode('hex-codec')

<Here you put your magic ;) like if you want to recognize sequences of bytes, strings unmodified between versions, etc>
```


Immunity Debugger - sysAnalyzer.exe
File View Debug Plugins ImmLib Options Window Help Jobs
Want to get paid to use this tool?
CPU - main thread, module sysAnaly
Log data
Address Message
Found module: ADVAPI32.dll
Found module: ntdll.dll
Found module: ole32.dll
Found module: MSVBVM60.DLL
Found module: USER32.dll
Found module: IMM32.DLL
Found executable to dump sysAnalyzer.exe
Threads
Ident Entry Data block
00000A0C 004029EC 7FFDF000
Handles
Handle Type Refs Access T Info Name
00000020 Desktop 2694. 000F01FF \Default
00000008 Directory 96. 00000003 \KnownDlls
00000014 Directory 62. 000F000F \Windows
00000030 Directory 434. 0002000F \BaseNamedObjects
00000010 Event 3. 001F0003
00000034 Event 2. 001F0003
0000002C File (dev) 2. 00100001 \Device\KsecDD
0000000C File (dir) 2. 00100020 c:\iDEFENSE\SysAnalyzer
00000028 Key 2. 000F003F HKEY_LOCAL_MACHINE
0000003C Key 2. 00020019 HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\N
00000040 Key 2. 00020019 HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\N
00000044 Key 2. 00020019 HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\N
00000004 KeyedEvent 60. 000F0003 \KernelObjects\CritSecOutOfMemoryEvent
00000038 Mutant 2. 001F0001
00000018 Port 3. 001F0001
0000001C WindowStation 115. 000F037F \Windows\WindowStations\WinSta0
00000024 WindowStation 115. 000F037F \Windows\WindowStations\WinSta0
SEH chain of main thread
Address SE handler
0012FFE0 kernel32.7C839AA8
Executable modules
Base Size Entry
00400000 00063000 004029EC
629C0000 00009000 629C2EAD LPK 5.1.2600.
66000000 00152000 66001AEC MSVBVM60 6.00.978
74D90000 0006B000 74DCAEB6 USP10 1.0420.2
76390000 0001D000 763912C0 IMM32 5.1.2600.
77120000 0008B000 77121558 OLEAUT32 5.1.2600.
774E0000 0013D000 774FD0A1 ole32 5.1.2600.
77C10000 00058000 77C1F2A1 msvcrt 7.0.2600.
77DD0000 0009B000 77DD70D4 ADVAPI32 5.1.2600.
77E70000 00091000 77E76284 RPCRT4 5.1.2600.
77F10000 00047000 77F16597 GDI32 5.1.2600.
7C800000 000F5000 7C80B5AE kernel32 5.1.2600.
7C900000 000B0000 7C913156 ntdll 5.1.2600.2180 (C:\WINDOWS\system32\ntdll.dll
Found module: RPCRT4.dll
Found module: ADVAPI32.dll
Found module: ntdll.dll
Found module: ole32.dll
Found module: MSVBVM60.DLL
Found module: USER32.dll
Found module: IMM32.DLL
Found executable to dump sysAnalyzer.exe
The string f77d is found!
Search done in: 0 seconds
Module C:\WINDOWS\system32\USP10.dll
Paused

Looking for patterns

- **The program behaviour is a really easy way to identify a forensic tool:**
  - **Regular reads to some directories (like configuration files, libraries and others)**
  - **Start read position in a memory dump (some systems first try to discover a backdoor manipulating the system, opening the memory devices, some others just try to load a kernel module to verify kernel violations, etc)**

Detecting forensics tool

- **Hooking system loading interfaces to easily spot a new program been runned, and them analyse the program and compare to a signature base:**
  - **ld.so, init_module, lsm, load_binary, do_execve, do_fork, ...**
- **But, how about other tools?**

Fighting against Forensics tools – The old school

- A lot of different talks about different ways to hide information from a Forensics tool – my approach is not to try to hide it, but discover a forensic tool running in the system (if someone is analysing the system, is because they already know something wrong ocurried)

Old school quick tour

- **Shadow Walker talk at Blackhat by Sherri Sparks and Jamie Butler showed the idea of use TLB desyncronization to hide your rootkit**
- **Basicly it uses:**
  - **Page fault handling patches**
  - **Pages are marked as non-present, and the page-fault system will verify if the instruction pointer is pointing to the faulted address (cr2) to differentiate between a read/write and one execution**
  - **The page fault system marks this pages as non-pageable to differentiate between 'protected' pages and the common ones (in Linux if you are just using kernel pages don't need to care about that)**

Old school quick tour

- **There are a lot of problems with this approach against a Forensic analyst (skilled one) – as spotted by the authors of this idea:**
  - **It's easy to detect IDT modifications and for sure to check the page faulting mechanics**
  - **Non present pages in non paged memory range are really not normal**

Old school quick tour

- Another approach is to hide your patches to the kernel using the debugger registers (I covered a lot about how to do that in my presentation about kernel integrity protection in the VNSecurity Conference)
- The problem is it can also be verified just using the segmentation support existent in the platform to bypass breakpoint hit or (also easy) just patching the debugging interrupt handling by yourself and trying to modify the debug registers (it will generate and exception if someone have set the general detection flag in dr7)

Anti-forensics hide rootkit

- **If you need to use disk (to transfer things to the machine and don't want to use syscall proxying-like systems) you can do that in many different ways:**
  - **Transfer your data to system memory**
  - **Force it to be loaded in a high virtual memory, and causes a page-out of this data (you also need to patch the paging system)**
  - **If it is a big machine you can use kmap to remap your addresses from ZONE_HIGH to ZONE_NORMAL when you need to manipulate it (read/write)**
  - **A simple crypting routine using a session key is enough (do you remember we are protecting the system against a memory dump) – I don't care about rootkit detection itself, the proposal is to protect the kernel, but care must be taken...**

What is needed in an anti-forensic rootkit?

- It must detect a forensic analysis and react to it (maybe removing all the evidences, including itself)
- In some way it must be 'pattern free', so it cannot be detected by common ways (to detect it will be needed a lot of knowledge from the analyst, and it is almost impossible to detect if you don't know the rootkit itself)
- Maybe the Virtualized Rootkit is dead, but what about use another hardware resource in rootkits?

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

SMM Relocation

- SMM has the ability to relocate its protected memory space. The SMBASE slot in the state save map may be modified. This value is read during the RSM instruction. When SMM is next entered, the SMRAM is located at this new address - in the saved state map offset 7EF8
  - Some problems to perform CS adjustments
- It can be used to avoid SMM memory dumping for analysis

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

Filipe Balestra and Nicolas Waisman for helping in the Immunity Debugger Stuff

HITB crew (mainly to XWings) for the nice time in Malaysia

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

XCon crew: Thanks for the good time in Beijing

VNSecurity crew: Awesome for us to be with so many l33t friends.

Your patience!

Let's stop this bullshit and drink ;D

End! Really is?

Questions?

Thank you :D

Rodrigo Rubira Branco
<rodrigo@kernelhacking.com>
<rodrigo@risesecurity.org>