

KIDS – Kernel Intrusion Detection System

VNSecurity 2007 – Vietnam

Rodrigo Rubira Branco

<rodrigo@kernelhacking.com>

Domingo Montanaro

<conferences@montanaro.org>

Ho Chi Minh City - Vietnam, 03/08/2007

Disclaimer

This presentation is just about issues we have worked on in our own time, and is NOT related to the companies ideas, opinions or works.

We are just security guys who work for big companies and in our spare time we do security research and we are organizers of the Brazilian's largest hacking conference called H2HC – Hackers 2 Hackers Conference (Hope 2 c u there!)

Montanaro main research efforts are in Forensics and Anti-Forensics technics and backdoor detection/reversing

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

Agenda

- Motivation – Actual Issues to be solved
- Tools that try to act on this issues and their vulnerabilities
- Differences between protection levels (software / hardware)
- StMichael – what it actually does
- Our Proposal
- Comments on efforts of breaking our ideas
- Improvements on StMichael – Technical Stuff
- Questions and Astalavista baby :D

Motivation

- Linux is not secure by default (we 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 levels: 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



We 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"
Then set the security_ops to dummy_sec_ops ;)

Why this code is so important?

- **It is a public way to bypass security mechanisms if you have kernel code execution**
- **WE HAVE USED IT TO PROTECT SELINUX AND LSM in StMichael ;)**

Kernel Backdoor

- As we already showed in another conference, write an efficient kernel backdoor is pretty easy, including for not so experienced kernel hackers.
- Anti-forensics are a real trend, so we need to improve our systems security.
- * UDP Client/Server -> You can use it to receive and respond to backdoor commands
- * LSM registered functions (or hooks) -> Can intercept commands, hide things, and do interesting things
- * Execution from the kernel mode -> Can execute commands requested by the user
- * Schedule tasks -> Permits scheduling the backdoor to run again (maybe to begin a new connection - connback), after a period of time

Yeah, only using public available sources!!

PaX Details

- **“The Guaranteed End of Arbitrary Code Execution”**
- **Address Space Layout Randomization (ASLR)**
- **Provides non-executable memory pages**

Seems good, but

- Various methods of by-passing some PAX resources were successful demonstrated (H2HC 2005)
- Any kind of bug that leads to arbitrary kernel write/execute could be used to re-mark the page-protection mechanism (PaX KernSeal will try to prevent it)
- PAX needs complementary solutions (grsecurity)
- Most ppl think that PAX+SELinux is a perfect world, but it isn't since SELinux doesn't provide lsm modules that implements some protection that PAX needs

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 against 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).

Actual Problems

- Security normally runs on ring0, but usually on kernel bugs attacker has ring0 privileges
- 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

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
```

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 restoring 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.

Introducing StMichael

continuing..

- loctl() hooking
- Call flags test (and sets it again) -> and returns the original call
- Self protection : Removes itself from the module list
- Uses encrypted messages to avoid signature detection of its code
- Random keys
- MBR Protection
- Modules syscalls hooked (create_module,init_module,etc)

StMichael known problems

```
#ifdef ROKMEM

void sm_kmem_ro ( void ) {

#ifdef SYM_KMEM_FOPS

struct file_operations * fops = (struct file_operations *) SYM_KMEM_FOPS;

#endif

#ifdef !defined SYM_KMEM_FOPS

    int fd = -1;

    char * us_name;

    int size;

    size = sjp_l_strlen("/dev/kmem")+1;

    return;

}

us_name = sjp_l_malloc(size);
```

StMichael known problems

```
if ( ! copy_to_user(us_name, "/dev/kmem", size) ) { <handle the error> }  
fd = sm_open(us_name, O_RDONLY, 0);  
sjp_l_free(us_name, size);  
if (fd < 0) { <handle the error> }  
sj_write_kmem = current->files->fd_array[fd]->f_op->write;  
current->files->fd_array[fd]->f_op->write = sj_deny_write;  
sm_close(fd); fd = -1;  
  
#else // if we have the fops for the device  
    sj_write_kmem = fops->write;  
    fops->write = sj_deny_write;  
#endif
```

Why is it a problem?

- **It can be bypassed mmap'ing the device and writing directly in memory**
- **Solving is trivial, just need to hook the mmap interface and protect it there too**
- **The issue here is that StMichael is a hackish, it has not been developed to be used in real production systems**
- **Nowadays it's just a collection of security ideas to be used in other systems**

Optimization

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

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...

Hooking IDT

/* To load the new value */

void load_myidt(void *value)

{

asm(" lidt%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                ");
```

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));
```

Where do we want to go?

Our Proposal

- StMichael must be a SW independent of other set of programs that try to defend the system
- We 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.

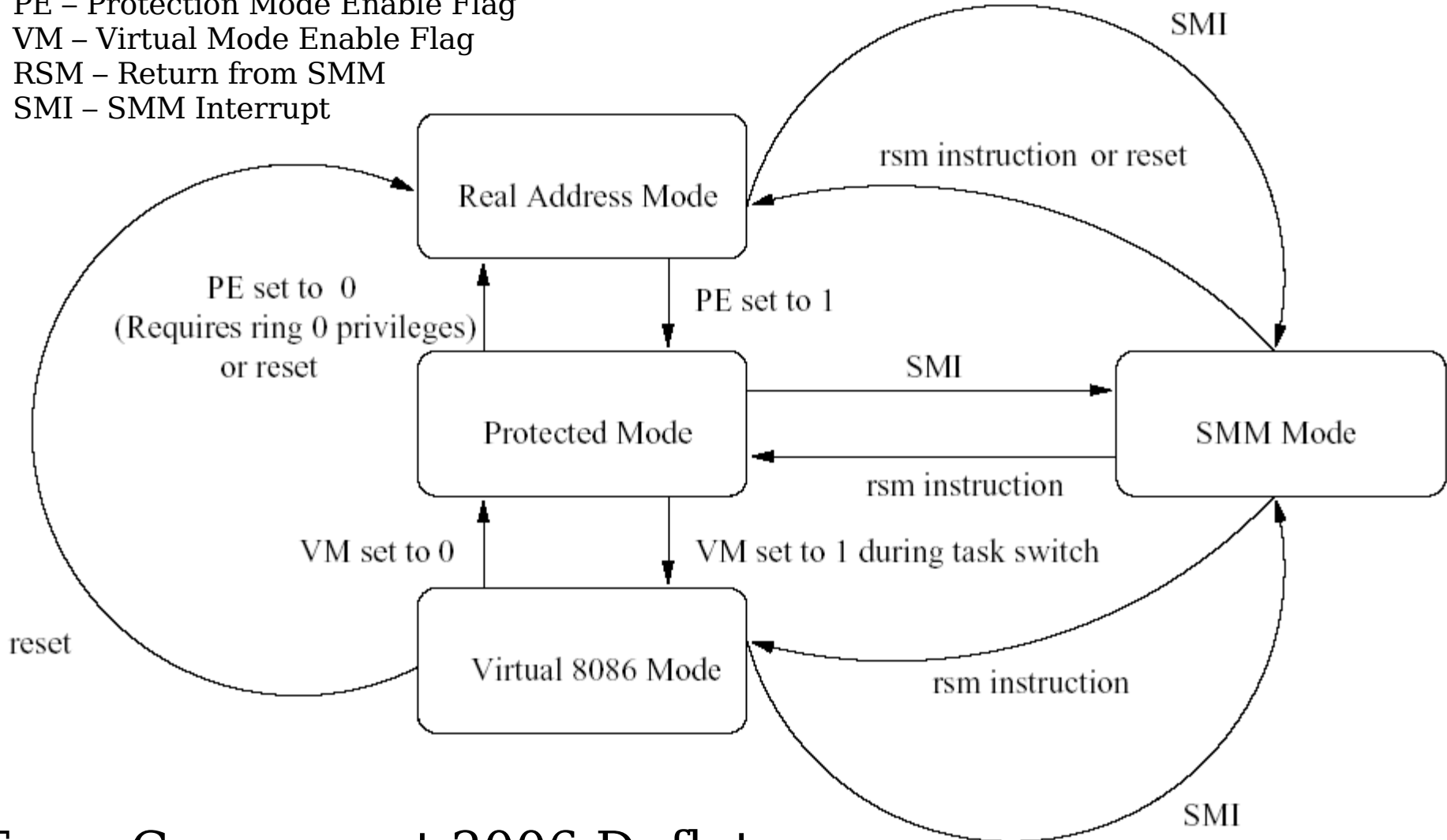
From the Intel386™ Product Overview – intel.com

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 Dufлот

SMM Resources

- **No paging – 16 bits addressing mode, but all memory accessible using memory extension addressing**
- **To enter SMM, we need an SMI**
- **To leave the SMM, we need the RSM instruction**
- **When entering in SMM, the processor will save the actual context – so, we can leave it in any portion of the address space we want – 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

- **We have many possibilities:**
 - **Using ACPI events (do you remember hibernation and sleep)**
 - **Using an external #SMI generator in the bus (well, we have discovered it is patent pending...)**
 - **Some systems (AMD Geode?) are always generating this kind of interrupt**
 - **Writing to a specific I/O port also generates an #SMI**

Address Translation while in SMM

- **The biggest difficulty**
 - We need to have the cr3 register value (in x86 systems... for more info in other platforms see our presentation at Xcon)
 - We must parse the page tables used by the processor (used by the OS)
 - Using DMA we can read the page tables (do you remember the PGD, PMD and PTE?)
- **Maybe we can just read the physical pages used by the kernel and compare it against a 'trusted' version (it doesn't sounds 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 our handler in main memory (that's what we are using now)**

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>
}
```

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:0xffff0"    ); /* Writing it in the save EIP */
asm (      " rsm"                   ); /* Switch back to protected mode */
asm (      "endhandler:"            );
asm (      ".text"                 );
asm (      ".code32"               );
```

Dangerous

- **Someone can use SMRAM cached memory to attack our SMI handler**
- **To protect against it we must flush the cache lines before execute the RSM instruction**

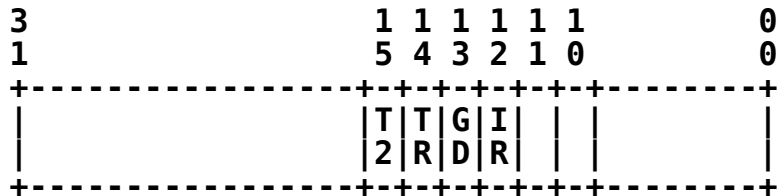
SMM locking

- **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. We don't want these to be executed, so we set SMISPCYCDIS and the RSMSPCYCDIS to 1 (prevents the input and output cycle respectively).**

Protecting missing portions

- Where will be our handler? In the memory, so someone can attack it?
- 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!

The Debug Register 7 (DR7) has few undocumented bits that completely modifies the CPU behavior when entering SMM (earlier ICE – In-Circuit Emulation → previous of SMM)



collateral effects!

- +-- IceBp 1=INT01 causes emulator to break emulation
0=CPU handles INT01
- +---- General Detect = Yeah, we can spot CHANGES in the Registers
- +----- Trace1 1=Generate special address cycles after code discontinuities. On Pentium, these cycles are called Branch Trace Messages.
- +----- Trace2 1=Unknown.

Debugging theory in Intel

In Intel platform we have dr0-7 and 2 MSR (model-specific registers)

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

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

Also, we 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

We 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. We need to set it again after exit our 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 we modify (setting it to 1) the general detect flag in the dr7**
- **We 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 we trigger the #DB we need to know (taken from mood-nt):**
 - **Why it occurred.**
 - **If someone is touching the #DB registers, the dr7 bit 13 (set to 1) will generate this exception to us, before executing the instruction.**
 - **So, our handler must parse what is the next instruction (pointed by EIP – the debugger exception handler receives a struct regs)**
 - **Also, we need to set the dr7 again – here we can use some randomization in what point of code we will protect with this registers**
 - **Then, if the instruction is touching the debug registers we 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 we 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 we set the RWn of the dr7 to 00 (instruction only) we must have the len() to "00", otherwise we 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, we have 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 -> We will cover some difficulties for Power platforms in the Xcon (China)**

Future? Who wanna test?

- We are looking for a secure OS that wants to try our proposal
- If someone wants to join these guys from the Carnival land:



Come on over :-)

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

VNSecurity crew: Awesome for us to be with so many 133t friends. Let's stop this bullshit and drink ;D

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

Thanks!

Questions?

Thank you :D

Rodrigo Rubira Branco
<rodrigo@kernelhacking.com>
Domingo Montanaro
<conferences@montanaro.org>