Truly random notes on computer security

Rodrigo Rubira Branco (BSDaemon)
rodrigo *noSPAM* kernelhacking.com

Into modern software exploitation

- Why software can be exploited
  - Sergey Bratus (yeah, your teacher) created an amazing definition: because it contains a ‘weird machine’ [that runs attacker’s crafted inputs as if they were a program]
- Thinking about group theory, exploitation is made possible because the attacker has the ability to group together what we name ‘PRIMITIVES’ [“weird assembly instructions”]

Primitives x Techniques

- Exploiting technique
  - Very general (works for *ALL* cases that the technique applies)
  - Think about:
    - tdelete()
    - pop-pop-ret
    - jmp %esp

Mind detour
CVE-2010-0083

```
(gdb) x/i $pc
        0xff0d1ec4 <_malloc_unlocked+356>:  ld [ %l2 ], %l4
(gdb) i r $l2
        l2      0x41414141
```

Heap Structure:

```
        "\x00\x00\x00\x00".          // Size (must be zero)
        "\xff\xff\xff\xff".          // Whatever value
        $what_to_write.
        "\x00\x00\x00\x00".          // Whatever value
        "\xff\xff\xff\xff".          // -1 to force getting into the t_delete function
        "\xff\xff\xff\xff".          // Whatever value
        $pointer_to_null.            // Specific to this bug only (usually Whatever valid pointer)
        "\xff\x00\x00\x00".          // Whatever value
        $where_to_write_minus_8
```

Mind detour

```
(gdb) x/i $pc
        0xff0c766c <t_delete+52>:         st %o0, [ %o1 +8 ]
(gdb) i r $o0
        o0     0x61626364
(gdb) I r $o1
        o1     0x41424344
```

We have a write 4!

Challenges:

- How to send our shellcode within the packet
- What to overwrite

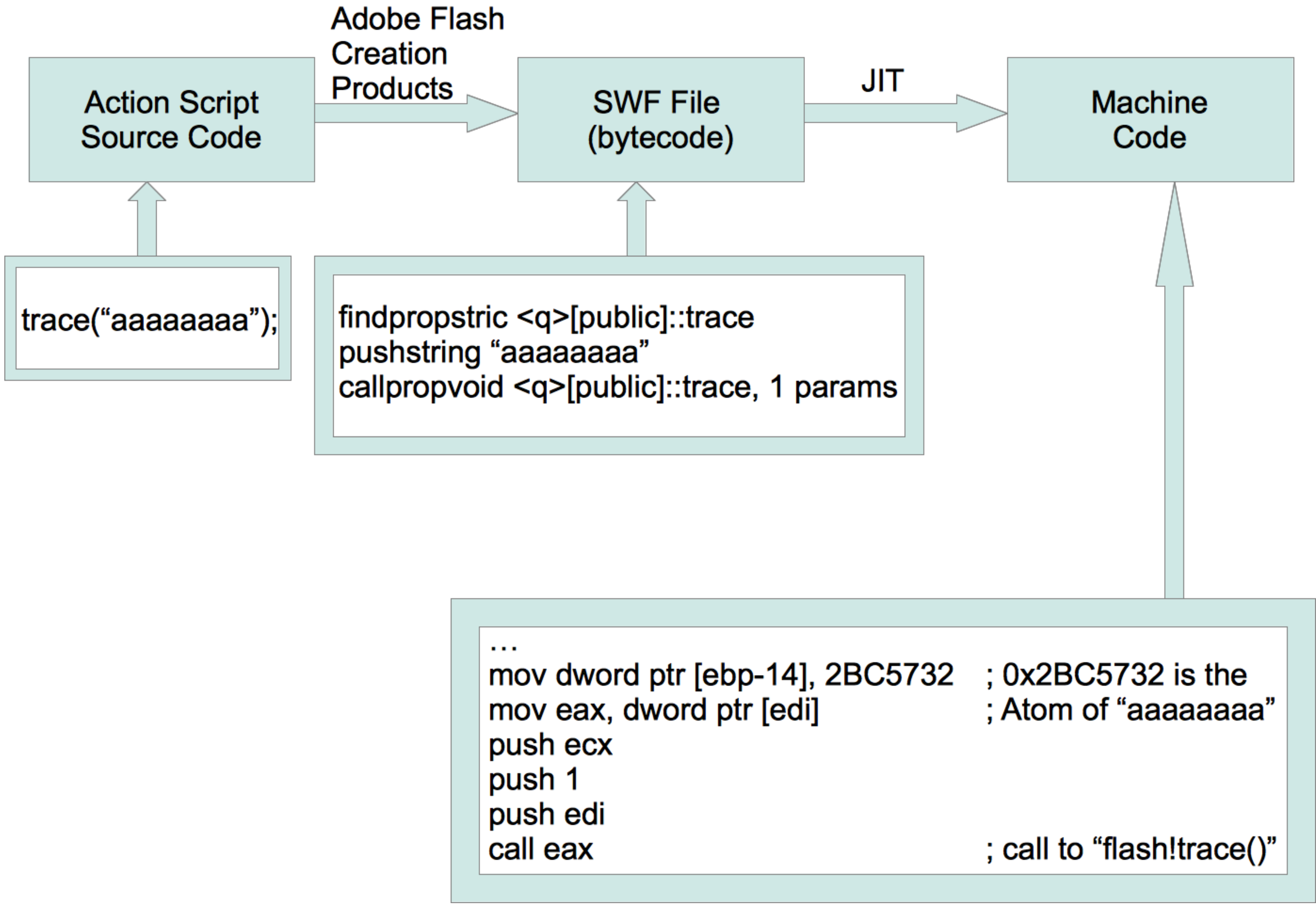
Primitives x Techniques

- Exploiting primitives
  - Grouping together what you need to create the ‘weird machine’
  - Here we talk about per-vulnerability conditions
  - Harder to generalize, harder to prevent

Mind detour CVE-2011-0609

SWF File divided in ABC Segments:

```
abcFile {
        u16 minor_version
        u16 major_version
        cpool_info constant_pool
        u30 method_count
        method_info method[method_count]
        u30 metadata_count
        metadata_info metadata[metadata_count]
        u30 class_count
        instance_info instance[class_count]
        u30 script_count
        script_info script[script_count]
        u30 method_body_count
        methody_body_info method_body[method_body_count]
}
```

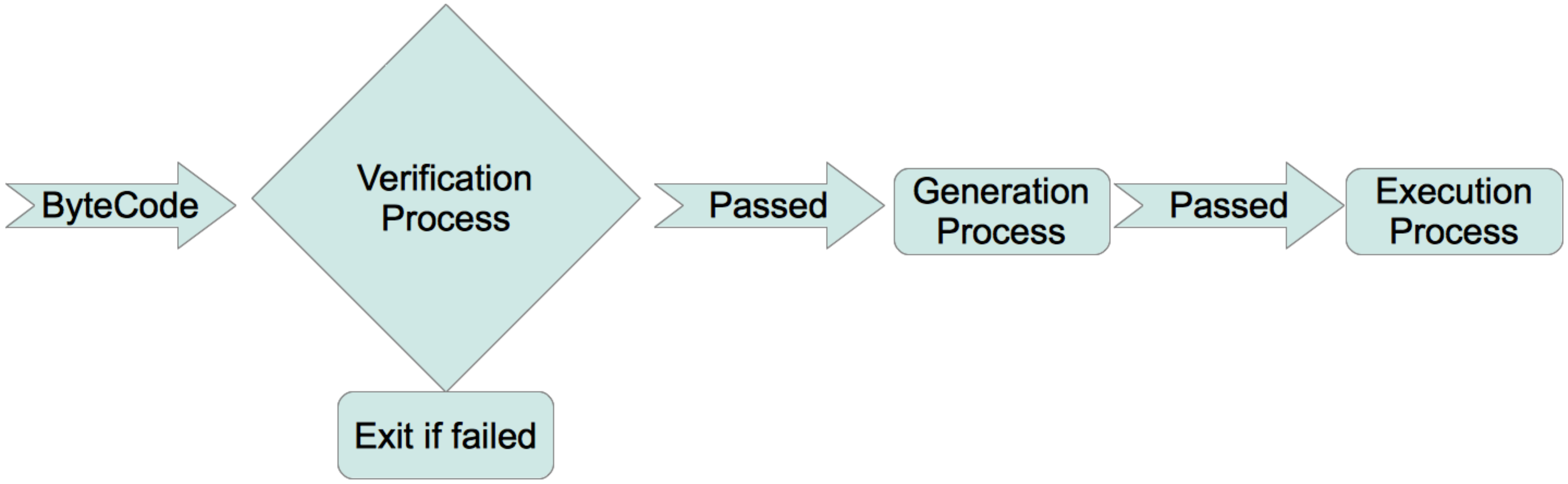


```
trace("aaaaaaaa");
```

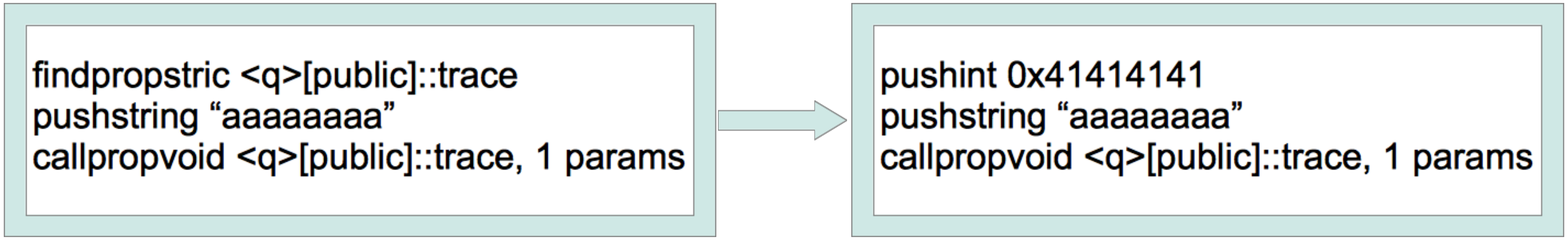
```
findpropstric <q>[public]::trace
pushstring "aaaaaaaa"
callpropvoid <q>[public]::trace, 1 params
```

```
…
mov dword ptr [ebp-14], 2BC5732    ; 0x2BC5732 is the
mov eax, dword ptr [edi]           ; Atom of "aaaaaaaa"
push ecx
push 1
push edi
call eax                           ; call to "flash!trace()"
```

Source: Hai Fei Li Presentation @Cansecwest 2011



Source: Hai Fei Li Presentation @Cansecwest 2011



```
findpropstric <q>[public]::trace
pushstring “aaaaaaaa”
callpropvoid <q>[public]::trace, 1 params
```

```
pushint 0x41414141
pushstring “aaaaaaaa”
callpropvoid <q>[public]::trace, 1 params
```

Source: Hai Fei Li Presentation @Cansecwest 2011

Inconsistent stack state after a jump to the incorrect position
Instructions write to the wrong object in the ActiveScript Stack, overwriting memory:

```
mov ecx, dword ptr ds:[edx+70] -> Program fails here
lea edx, dword ptr ss:[ebp-70]
mov dword ptr ss:[ebp-70], eax
mov eax, dword ptr ds:[ecx]
push edx
push 0
push ecx
call eax
```

Shellcode:

- Relies on a technique to get the KernelBase that does not work on Windows 7 (InInitializationOrderModuleList traversal from the PEB)
- Uses hashes for calling functions (normal)
- Finds the handle to the excel process
    - Then it reads the binary from the memory image
    - Identifies in the file the sequence: 43 2E 42 47 04 06 89
- Creates the binary a.exe and executes it
    - It inserts the header of the file from the shellcode itself (call to WriteFile('MZxx')) in order to bypass AVs detecting binaries inside excel files

Bypassing DEP:

- Need to use ROP
- Create a first stage shellcode that will:
  - Allocate Memory using NtAllocateVirtualMemory
  - Copy a second stage shellcode to that location and execute it
- PDF gadgets are widely available from other exploits:

```
dd 7004919h             pop ecx
                                  pop ecx
                                  mov dword ptr [eax+0Ch], 1
                                  pop esi
                                  pop ebx
                                  retn
dd 0CCCCCCCCh      ecx = 0xCCCCCCCC
dd 70048EFh             ecx = 0x070048EF
dd 700156Fh             esi = 0x0700156F
dd 0CCCCCCCCh      ebx = 0xCCCCCCCC
dd 7009084h             retn
dd 7009084h             retn
…
```

We control eax, do you remember?

So, what do you say when you look into a loaded library and see this instruction sequence:

```
0x070048EF     xchg esp, eax
0x070048F0    ret
```

What about ASLR?

The PDF gadgets shown are in a library that gets randomized

In Excel, some libraries are NEVER randomized and its known
how to force than to be loaded (using specific entries) -> Thus,
if attacker use gadgets from Excel, it is game over (already reported to MS)

In Acrobat they improved (a lot) that and it is much more tricky…

Fortunately, Flash leaks objects addresses and thus it is possible
to create an exploit that uses objects to get addresses of base
pointers, and then offsets to determine the library locations

Back on Track
What do we have to do with that?

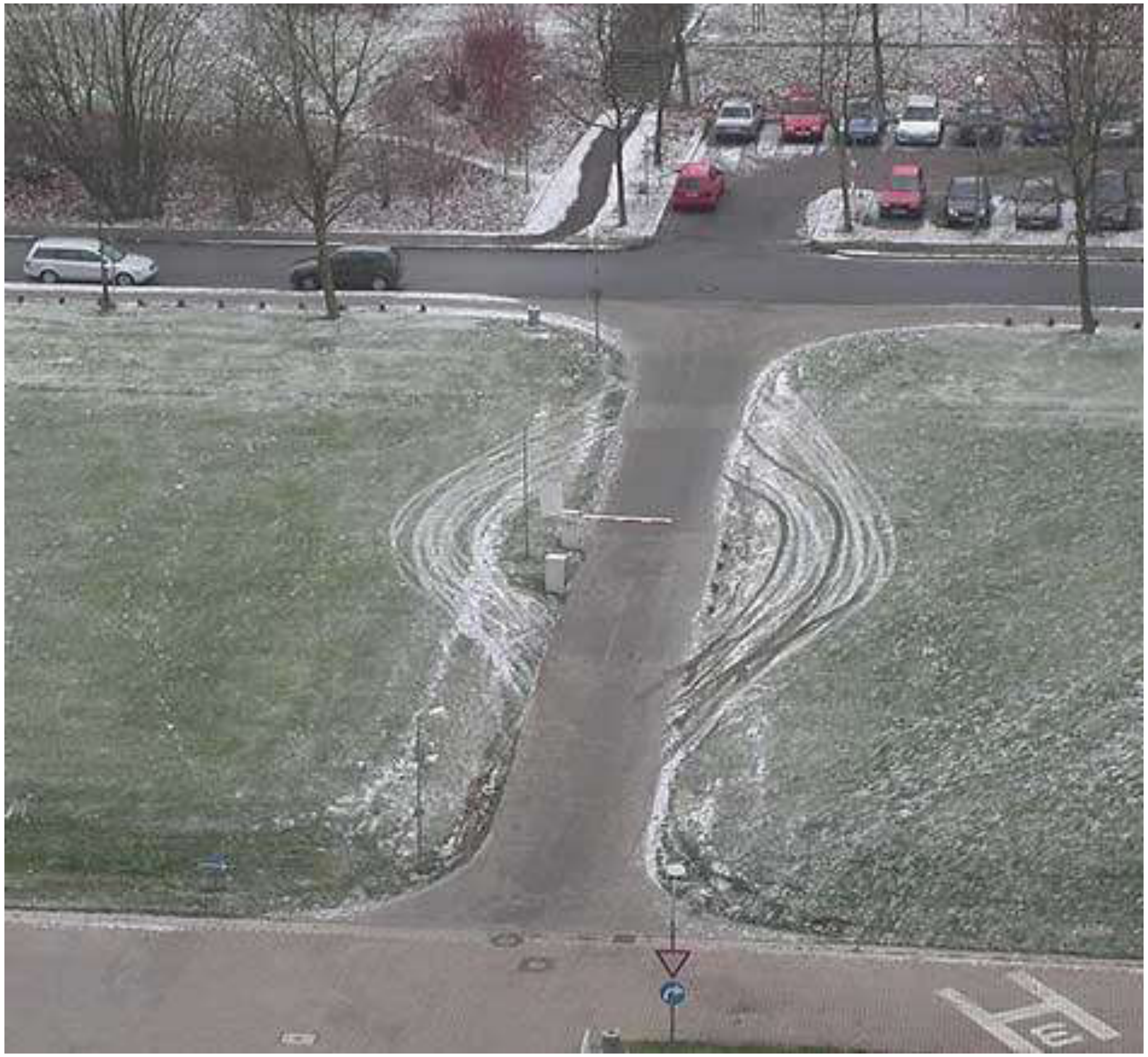
- We need to remove from the attackers the capability of complete controlling of the ‘weird’ machine, and thus force them to rely on primitives that they control
  - ASLR -> Add primitives
  - DEP -> Add primitives
  - Cookies -> Add primitives

- In the end, what we do:
  - Force new primitives, in order to create a complex weird machine for the attack
  - The more complex is the weird machine for the attacker to control, the less RELIABLE is the exploit

Kernel Land
Into the inners of our computers

- 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
- Most of modern OSs use only 2 privileges rings provided by Intel arch (4)
- These efforts in most of current security tools/methods/ politics try to block ring3 (user-mode) escalation to ring0 (kernel-mode)
- Many (a lot!) of public exploits were released for direct kernel exploitation
- Beyond 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!)

How do we all fail?



Security Hooking
Easier to the attacker than to the defender

```
ssize_t h_read(int fd, void *buf, size_t count){
        unsigned int i;
        ssize_t ret;
        char *tmp;
        pid_t pid;

        If the fd (file descriptor) contains something
        that we are looking for (kmem or mem)

        return_address();
        At this point we could check the offset being
        required. If is our backdoor addr, send
        another task_struct
        ret=o_read(fd,buf,count);
        change_address();
        return ret;
        }
```

```
int return_address()
{
return our hacks to the original
state
}
```

```
int change_address()
{
put our hacks into
the kernel
}
```

What has been happening?

- **Spender's public exploit (null pointer dereference):**

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

StMichael uses session keys to encrypt internal strings since 2003!

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 and the Kernel

- 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).

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 (forensics challenge)
- Detection on kernel-based backdoors or attacks rely on “mistakes” made by attackers - how to detect an 'unknown' rootkit?

Changing page permissions writing to PaX-protected memory areas

```
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 similar code to change kernel pages to RO
```

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

We 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)

Optimizing code

- Many efforts are needed to accomplish code optimization
- Lazy TLB:
  - When a threads executes, copy the old active mm pointer to be thread's own pointer
  - Doing so, the system does not need to flush the TLB (one of the most expensive things)
  - Because a defense system just touches kernel-level memory, it doesn't need to care about wrong resolutions
  - That's why we cannot just protect the kcrash kernel

Interrupt Handling

- Here we will try to cover two different platforms: Intel and PowerPC
- The general idea is to begin showing how our 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

System calls - Intel

- 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)

System calls - PowerPC

- 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 come at the fixed address 0x500 and are treated in a similar way to the intel IDT jump

Efforts on bypassing StMichael

- Julio Auto at H2HC III proposed an IDT hooking to bypass StMichael – in Vietnam I showed a working sample of this proposal (he just gave a theoretical idea to bypass it)
- 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 protects against module insertion not against arbitrary write (main purpose of StMichael)

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 **split TLB**s for data and code execution
- Someone can force a TLB desynchronization to hide kernel-text modifications from our reads
  - This technique relies on the page fault handler patch, since we protect the hardware debug registers (see more ahead) and we also 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 VMAs in the system, detecting the ones with execution property enabled in the dynamic memory section
  - Doing so you can spot where the StMichael code is in the kernel memory, so, just need to attack it...

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

How? SMM!

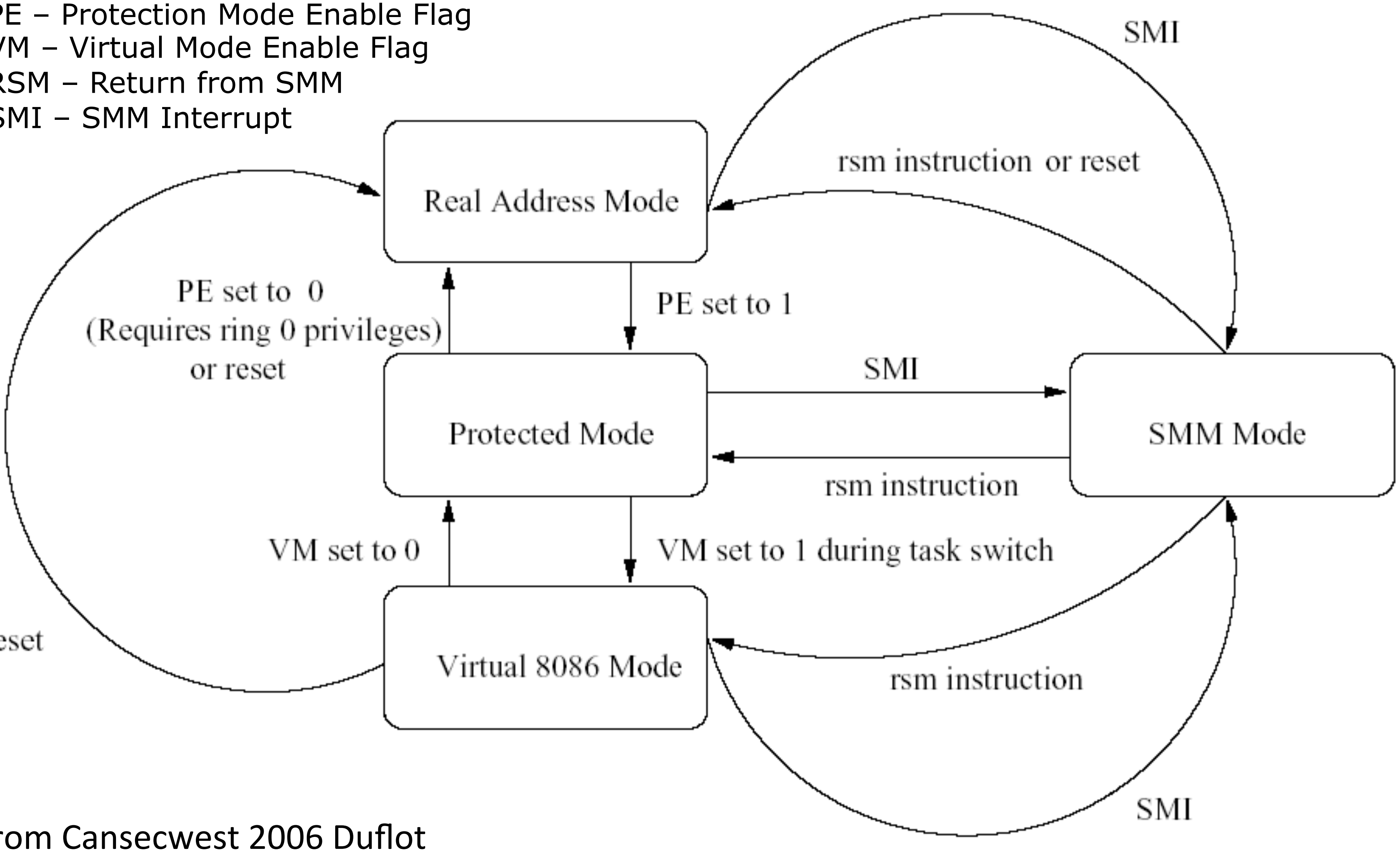
SMM – System Management Mode

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

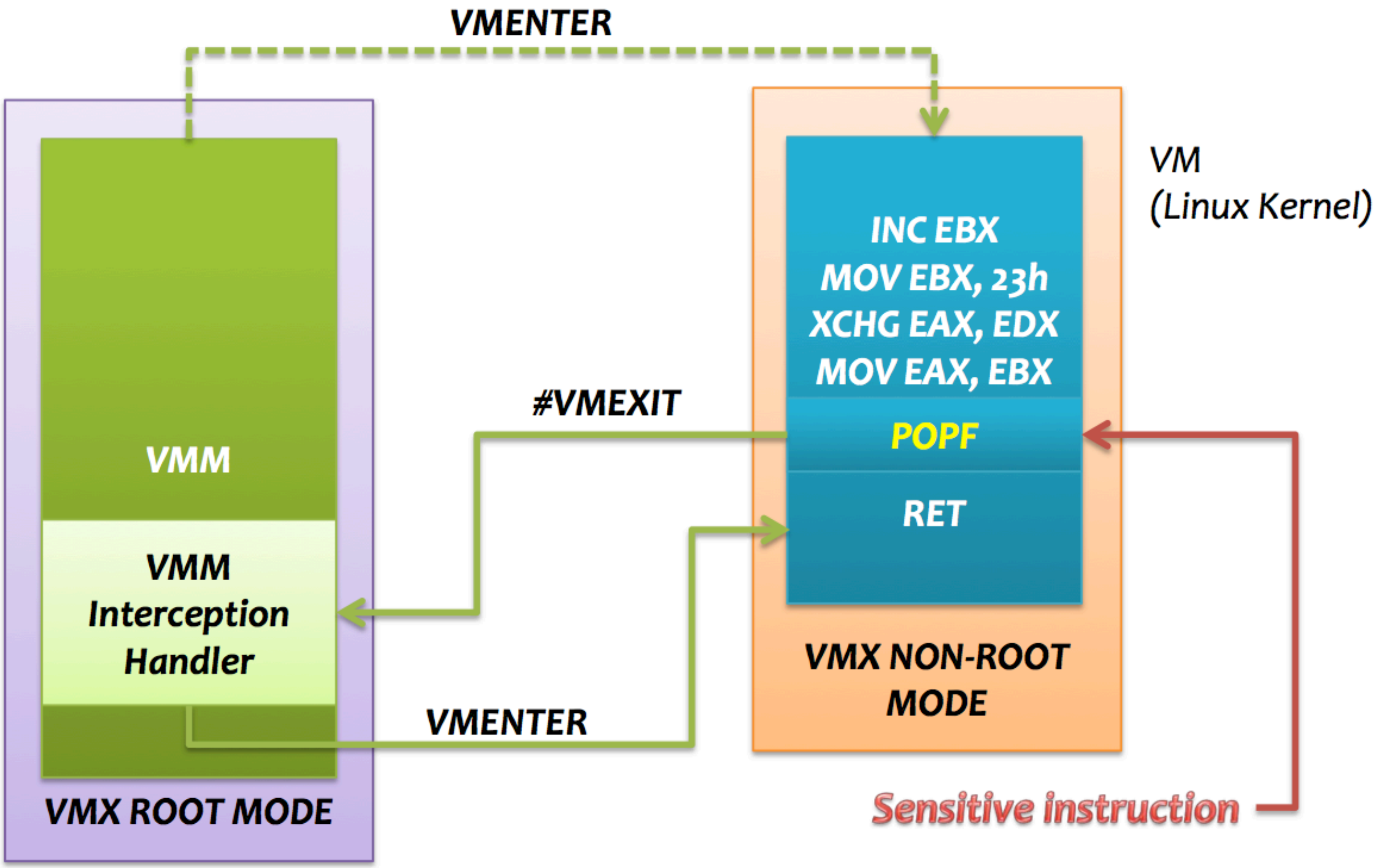
Context switches

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



From Cansecwest 2006 Duflot

Mind Detour


VMENTER
VM
(Linux Kernel)
INC EBX
MOV EBX, 23h
XCHG EAX, EDX
MOV EAX, EBX
#VMEXIT
POPF
RET
VMM
VMM
Interception
Handler
VMENTER
VMX NON-ROOT
MODE
VMX ROOT MODE
Sensitive instruction

Mind Detour

- Virtual Machine Control Structure
- Data structure used by the CPU to store all information about the Virtual Machine and the VMM.
- Divided in 5 areas:
  – Guest-state area
  – Host-state area
  – VM-execution control fields – VM-entry control fields
- – Vm-exit information fields.
- One for each VM and for each CPU
- Must not be directly accessed
  – Use the VMX instructions to access the VMCS.

Mind Detour

- The virtual machine must handle all the interception events (VMEXIT).
- • Types of events
  – CRx/DRx register access – Interrupts
  – I/O instructions
  – TSC and MSR registers – much more....

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 (thanks 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
  - 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, static 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.

Address Translation while in SMM

- The biggest difficulty
  - We need to have the ***cr3*** register value (in x86 systems)
  - 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 sound good, since *sparsemem* systems will be really difficult to protect and dynamically generated kernel structures too)
- Another approach is just to 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>
}
```

- 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
```

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 */

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 we 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
  - We 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 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).

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);
```

Useful?

- 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 maybe used to avoid SMM memory dumping for analysis

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 unducumented bits that completely modifies the CPU behavior when entering SMM (earlier ICE – In-Circuit Emulation ◊ previous of SMM)

```
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, we 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.
```

collateral effects!

Debugging theory in Intel

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

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

The meaning of having MSRs is to remember the last branches, interruptions or exceptions generated that 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 exiting 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

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
- We can just change the saved EIP to point to our task and also the privilege level, forcing the system to return to our task, with full memory access
- Since the SMRAM is protected by the hardware itself, it is really difficult to detect this kind of rootkit

Compatibility Problems

- Yeah, we have SMM just in the Intel platform... but:
  - Many platforms already support something like firmware interrupts
  - PowerPC does not have the IDT register problem, so what we can do?

PowerPC Kernel Protection

- The idea of putting the entire kernel as read-only seems good
- The attacker cannot modify the pages permissions, since we can use watchpoints to monitor that
- 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]) we can choose to monitor the effective or the real address
- We also can instrument an external debug systems, 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 just “hackish”, it's offered by major 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."

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

Who wanna test?

- Everyone uses KIDS ;)

- If someone wants to join these guy from the Carnival land:

Acknowledgments

Sergey Bratus for inviting me over to this class… and inspiring me again to look over this material.

Edmond Rogers – the cheeseee-makeeerrr – the best a friend can be for the always insightful exchange of ideas!

Spender for help with many portions of the model

PaX Team for resolving my doubts about PaX and giving many helpful explanations of the PaX implementation

XCon crew: Opportunity to go to Beijing and present part of this crazy slide-deck ;)

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

Questions?

Thank you :D

Rodrigo Rubira Branco

rodrigo@kernelhacking.com