# FROM RING3 TO RING0: EXPLOITING THE XEN X86 INSTRUCTION EMULATOR

*Andrei Vlad Luțaș*
*Bitdefender*
*vlutas@bitdefender.com*

## ABSTRACT

While a VMM can provide a considerable level of security by isolation, it is generally true that by increasing the code-base that runs on a given host system one also increases the attack surface. Instruction emulators are a critical part of any hypervisor, as they provide the means to virtualize certain devices or handle certain types of faults (such as EPT violations or Invalid Opcode Exceptions). However, creating an emulator is not an easy task, and as with any other piece of software, any issue in the emulation of an instruction might be exploited by an attacker in various ways. If, for example, the emulator fails to properly validate an instruction as precisely as a physical CPU does, an attacker might leverage this in order to gain elevated privileges or cause denial of service. This is an important issue especially, since such problems can be successfully exploited on virtually any x86 operating system. The Xen hypervisor has several vulnerabilities involving the x86 emulator, due to the lack of validation of privilege, which enables the emulation of several sensitive instructions from ring-3: **LMSW**, **HLT**, **INT**, **LIDT**, **LGDT**. Some of them have a minor effect (**LMSW**, **HLT**), others can cause denial of service (**INT**) and some facilitate escalation of privileges (**LIDT**, **LGDT**) by leading to arbitrary code execution in ring-0. Additionally, a method to bypass Intel SMEP is presented, in the context of the discovered vulnerabilities.

## INTRODUCTION

Vulnerabilities in VMMs (Virtual Machine Monitors) are not something new. Hyper-V, Xen or VMware all had vulnerabilities at some point (and they probably still do – they're yet to be discovered). Xen Security Advisory (Xen Security Advisory, n.d.) and VMware Security Advisories (VMware, n.d.) contain a complete list of vulnerabilities identified either directly in the VMM or in other components, while a good example for Hyper-V is MS13-092 (Luft, n.d.). Some vulnerabilities may be used to cause a denial of service, while others can be used to gain elevated privileges. In this whitepaper we will describe two vulnerabilities identified in the Xen hypervisor, which allow for denial of service and elevation of privileges inside the guest. In addition, a method to bypass SMEP in the context of the presented vulnerabilities is disclosed.

## XEN X86 INSTRUCTION EMULATOR VULNERABILITIES

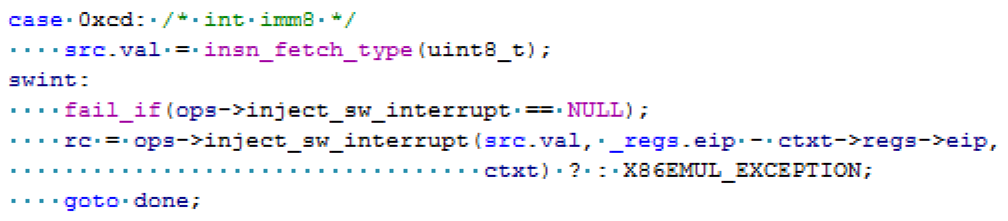
Two distinct vulnerabilities have been discovered in the Xen x86 instruction emulator, which also affect other platforms based on it, such as XenServer (tested on XenServer 6.2, build date 2013-10-15, build number 75966c), XenClient (tested on XenClient 5.1.3), XenClient XT (tested on XenClient XT 3.2.2 Trial, build 132629), Amazon and perhaps (although not tested) Oracle VM and others. Versions from at least 3.2.x onwards are vulnerable (older versions have not been tested) to:

- Logic errors in software interrupt (**INT** instruction) handling (XSA-106, CVE-2014-7156)
- Insufficient privilege-validations for **HLT**, **LMSW**, **LIDT**, **LGDT** instructions (XSA-105, CVE-2014-7155)

## LOGIC ERRORS IN SOFTWARE INTERRUPT HANDLING

### DETAILS OF THE VULNERABILITY

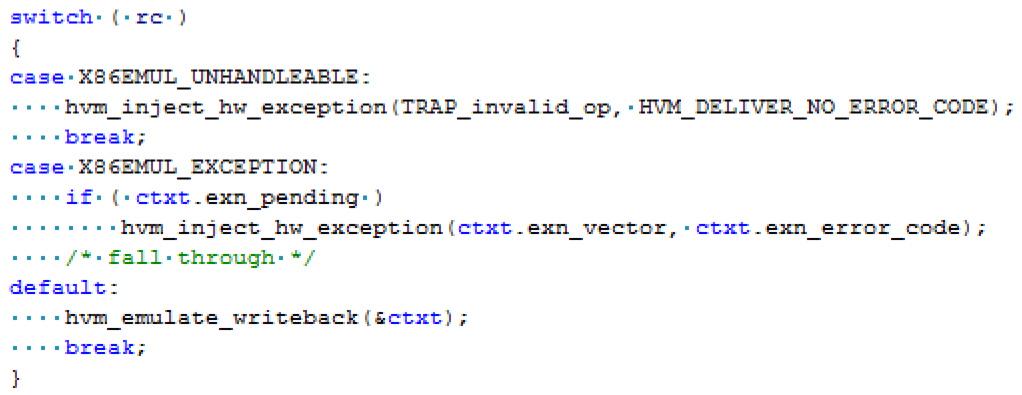
As part of x86 instruction emulation, sometimes an exception may need to be injected inside the guest (for example, a page-fault or an invalid-opcode exception). This is accomplished in Xen with the help of the *generate_exception_if* macro, which will call the *inject_hw_exception* callback and, if successful, it will return *X86EMUL_EXCEPTION*, which will be returned by *x86_emulate*. *hvm_emulate_one* function will also return *X86EMUL_EXCEPTION* if an exception of any kind has been generated by the emulator. This in turn will be handled by the caller by actually injecting the exception by calling *hvm_inject_hw_exception*. If we take a look at how **INT** is handled, we will see the following:



```
case 0xcd: /* int imm8 */
    src.val = insn_fetch_type(uint8_t);
swint:
    fail_if(ops->inject_sw_interrupt == NULL);
    rc = ops->inject_sw_interrupt(src.val, _regs.eip - ctxt->regs->eip,
                                  ctxt) ? : X86EMUL_EXCEPTION;
    goto done;
```

**Figure 1: INT instruction handling**[1]

One can see in Figure 1: INT instruction handling that although the *inject_sw_interrupt* callback is invoked in order to inject a software interrupt (not a hardware exception), *X86EMUL_EXCEPTION* is returned in case of success.



```
switch ( rc )
{
case X86EMUL_UNHANDLEABLE:
    hvm_inject_hw_exception(TRAP_invalid_op, HVM_DELIVER_NO_ERROR_CODE);
    break;
case X86EMUL_EXCEPTION:
    if ( ctxt.exn_pending )
        hvm_inject_hw_exception(ctxt.exn_vector, ctxt.exn_error_code);
    /* fall through */
default:
    hvm_emulate_writeback(&ctxt);
    break;
}
```

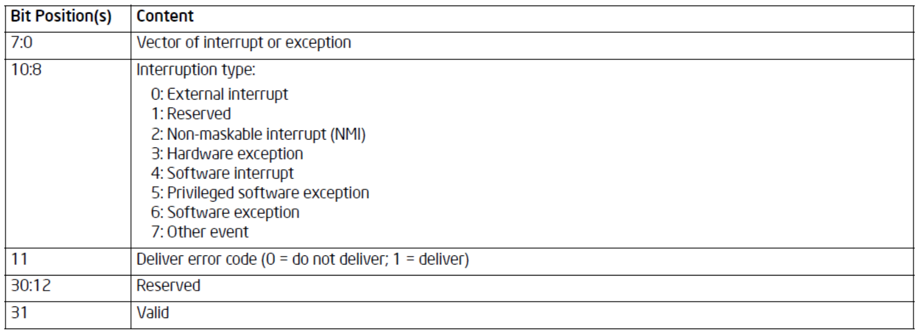
**Figure 2: Emulation return code interpretation**[2]

The handling of the *X86EMUL_EXCEPTION* is, however, the same, regardless of the type of event injected (software interrupt or hardware exception) and is handled by calling *hvm_inject_hw_exception* causing the injection to be made as a hardware exception, and not a software interrupt, as it should, as seen in Figure 2: Emulation return code interpretation.

[1] Xen/arch/x86/x86_emulate/x86_emulate.c, x86_emulate
[2] Xen/arch/x86/hvm/vmx/vmx.c, vmx_vmexit_ud_intercept

## EVENT INJECTION ON INTEL VT-X

Event injection on Intel VT-x architecture is acomplished by the means of a VMCS field. This field, dubbed *VMCS_EVENT_INJECTION*, has the layout descrbibed in Figure 3: VM-Entry Interruption Information field. The most relevant field for us is the *Interrupt type*. One can easily see that there are different types for *Hardware Exception* (3) and *Software Interrupt* (4). However, the most important difference is given in the Intel SDM, which states: "*If VM entry is injecting a software interrupt (not redirected as described above) or software exception, privilege checking is performed on the IDT descriptor being accessed as would be the case for executions of* ***INT n, INT3****, or* ***INTO*** *(the descriptor's DPL cannot be less than CPL). There is no checking of RFLAGS.IOPL, even if the guest will be in virtual-8086 mode. Failure of this check may lead to a nested exception. Injection of an event with interruption type external interrupt, NMI, hardware exception, and privileged software exception, or with interruption type software interrupt and being redirected as described above, do not perform these checks.*" This is the important difference between hardware exception injection and software interrupt injection: while descriptor checks are performed by the CPU as part of the injection for software interrupts, hardware exception injection will bypass these checks (as if the exception has been generated as part of the instruction fetch, decode or execution). If a software interrupt is injected as a hardware exception, the descriptor checks will be bypassed, and thus allowing the attacker to invoke arbitrary entries inside the IDT.



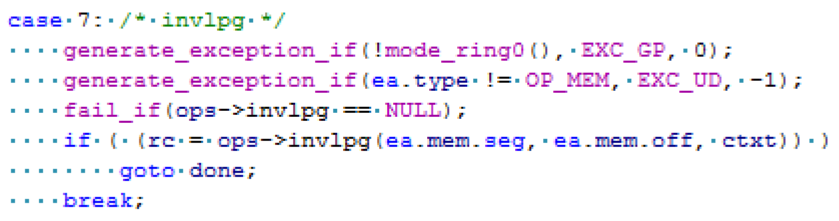
| Bit Position(s) | Content |
|---|---|
| 7:0 | Vector of interrupt or exception |
| 10:8 | Interruption type:<br>0: External interrupt<br>1: Reserved<br>2: Non-maskable interrupt (NMI)<br>3: Hardware exception<br>4: Software interrupt<br>5: Privileged software exception<br>6: Software exception<br>7: Other event |
| 11 | Deliver error code (0 = do not deliver; 1 = deliver) |
| 30:12 | Reserved |
| 31 | Valid |

**Figure 3: VM-Entry Interruption Information field**[3]

## INSSUFICIENT PRIVILEGE VALIDATIONS

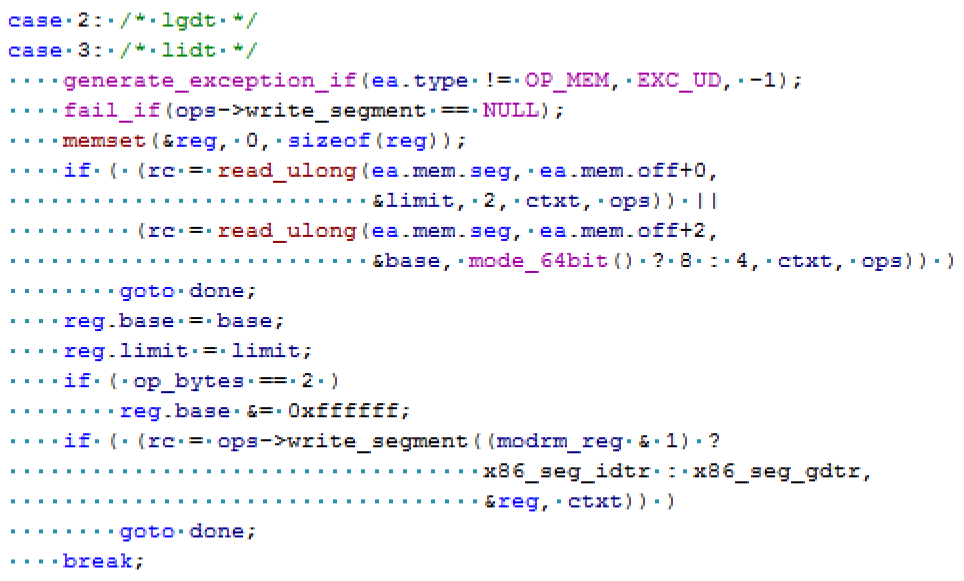
An emulator has the task to handle the emulated instruction as if it would be the CPU himself. One of the most important tasks is making sure that the instruction can be emulated, given the current context. For example, **LIDT** and **LGDT** will be executed by the CPU only from a ring0 code segment. If a ring3 code segment tries to execute them, a General-Protection Fault will be generated. The Xen emulator fails to validate the fact that the emulated **LIDT**, **LGDT** (seen in Figure 5: Lack of privilege level validation for the LIDT & LGDT instructions), **LMSW** or **HLT** instruction resides in a ring0 code segment (via the method seen in Figure 4: privilege level validation for the INVLPG instruction). This missing validation will allow an attacker to execute instructions in a lesser privileged code segment. While executing **HLT** will have no interesting effect, the **LMSW** execution will allow the attacker to change CR0 bits 1-3. On the other hand, **LIDT** and **LGDT** are the most interesting. Being able to load your own Interrupt Descriptor Table or Global Descriptor Table can easily lead to privilege escalation. Details about how an attacker can exploit these will be given in the upcoming sections.

[3] Intel SDM, 24-19, Vol. 3C, Table 24-13. Format of the VM-Entry Interruption-Information Field



```
case 7: /* invlpg */
    generate_exception_if(!mode_ring0(), EXC_GP, 0);
    generate_exception_if(ea.type != OP_MEM, EXC_UD, -1);
    fail_if(ops->invlpg == NULL);
    if ( (rc = ops->invlpg(ea.mem.seg, ea.mem.off, ctxt)) )
        goto done;
    break;
```

Figure 4: privilege level validation for the INVLPG instruction[4]

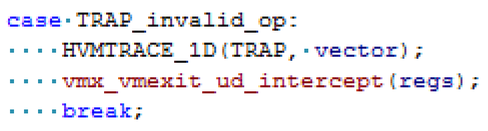


```
case 2: /* lgdt */
case 3: /* lidt */
    generate_exception_if(ea.type != OP_MEM, EXC_UD, -1);
    fail_if(ops->write_segment == NULL);
    memset(&reg, 0, sizeof(reg));
    if ( (rc = read_ulong(ea.mem.seg, ea.mem.off+0,
                          &limit, 2, ctxt, ops)) ||
         (rc = read_ulong(ea.mem.seg, ea.mem.off+2,
                          &base, mode_64bit() ? 8 : 4, ctxt, ops)) )
        goto done;
    reg.base = base;
    reg.limit = limit;
    if ( op_bytes == 2 )
        reg.base &= 0xffffff;
    if ( (rc = ops->write_segment((modrm_reg & 1) ?
                                  x86_seg_idtr : x86_seg_gdtr,
                                  &reg, ctxt)) )
        goto done;
    break;
```

Figure 5: Lack of privilege level validation for the LIDT & LGDT instructions[5]

## FORCING THE EMULATION OF ARBITRARY INSTRUCTIONS

The Xen x86 emulator gets called only in some special cases, such as MMIO accesses. Normally, instructions will not be emulated, and they will be executed natively by the CPU. However, on Xen, some instructions will always cause a VM exit and will lead to the emulator invocation. Such an instruction is **UD2**, which causes an Undefined-Opcode Exception. This will be handled by calling *vmx_vmexit_ud_intercept*, which will eventually call the emulator, as seen in Figure 6: Invalid-Opcode exception handling in Xen and Figure 7: Emulator invocation for Invalid-Opcode exception.

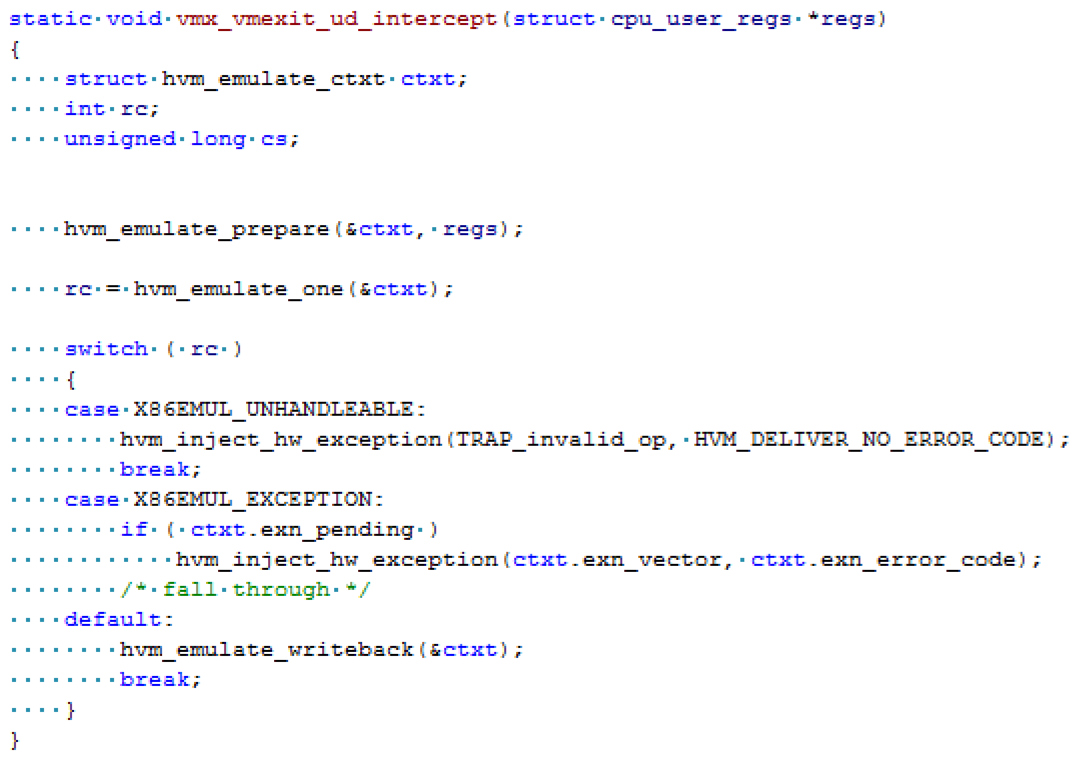


```
case TRAP_invalid_op:
    HVMTRACE_1D(TRAP, vector);
    vmx_vmexit_ud_intercept(regs);
    break;
```

Figure 6: Invalid-Opcode exception handling in Xen[6]

[4] Xen/arch/x86/x86_emulate/x86_emulate.c, x86_emulate
[5] Xen/arch/x86/x86_emulate/x86_emulate.c, x86_emulate
[6] Xen/arch/x86/hvm/vmx/vmx.c, vmx_vmexit_handler

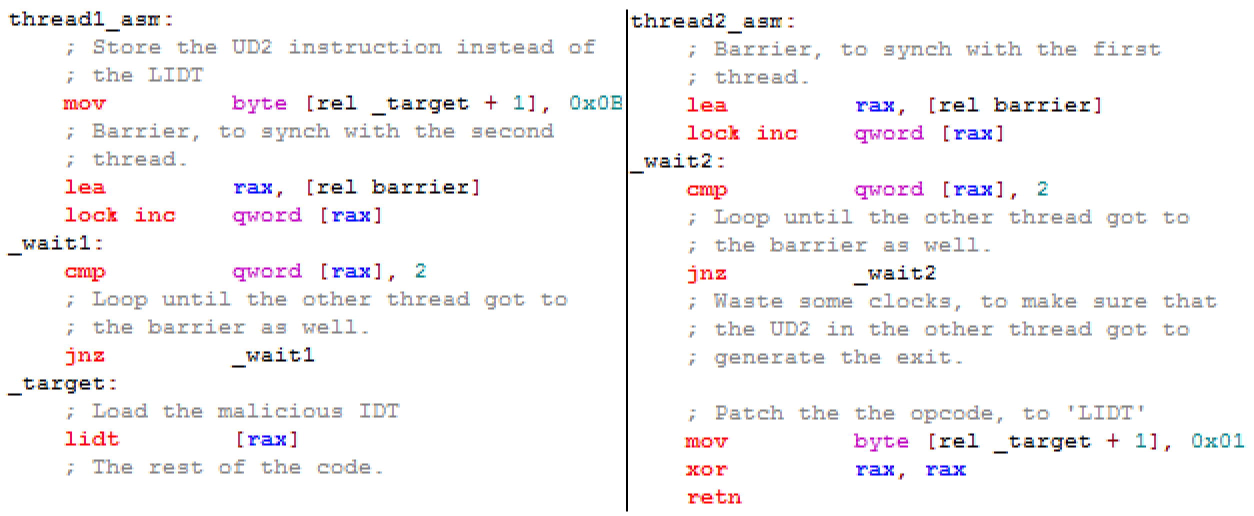
However, simply being able to force the emulation of a small subset of encodings is not enough, as we need to emulate some specific instructions. Therefore, we need a way to trick the hypervisor into thinking that a #UD took place, and in fact trigger the emulation of one of the vulnerable instructions instead. This is rather trivial, if the guest VM has at least 2 virtual CPUs. If one VCPU triggers a #UD, a VM-Exit will be generated, which will be handled by the hypervisor, and, eventually, it will call the emulator. If another VCPU maliciously overwrites the instruction that caused the #UD with another instruction exactly after the VM-Exit but before the emulator gets called, it might fool the emulator into emulating something else. The synchronization between the two VCPUs can be easily done using a barrier, and although it will fail sometimes due to the fact that interrupts might interfere with our synchronization, it will succeed enough times to pose serious problems. Example assembly code can be seen in Figure 8: Assembly code of two threads forcing the emulation of LIDT. The first thread initially overwrites byte 1 of the **LIDT** instruction with 0x0B (**LIDT** is encoded 0x0F 0x01 followed by the mod r/m while **UD2** is encoded 0x0F 0x0B) thus forcing a #UD. The second thread restores byte 1 of the original instruction, thus replacing the **UD2** with the original **LIDT [rax]**.



```
static void vmx_vmexit_ud_intercept(struct cpu_user_regs *regs)
{
    struct hvm_emulate_ctxt ctxt;
    int rc;
    unsigned long cs;


    hvm_emulate_prepare(&ctxt, regs);

    rc = hvm_emulate_one(&ctxt);

    switch ( rc )
    {
    case X86EMUL_UNHANDLEABLE:
        hvm_inject_hw_exception(TRAP_invalid_op, HVM_DELIVER_NO_ERROR_CODE);
        break;
    case X86EMUL_EXCEPTION:
        if ( ctxt.exn_pending )
            hvm_inject_hw_exception(ctxt.exn_vector, ctxt.exn_error_code);
        /* fall through */
    default:
        hvm_emulate_writeback(&ctxt);
        break;
    }
}
```

**Figure 7: Emulator invocation for Invalid-Opcode exception**[7]

[7] Xen/arch/x86/hvm/vmx/vmx.c, vmx_vmexit_ud_intercept



```
thread1_asm:
    ; Store the UD2 instruction instead of
    ; the LIDT
    mov         byte [rel _target + 1], 0x0B
    ; Barrier, to synch with the second
    ; thread.
    lea         rax, [rel barrier]
    lock inc    qword [rax]
_wait1:
    cmp         qword [rax], 2
    ; Loop until the other thread got to
    ; the barrier as well.
    jnz         _wait1
_target:
    ; Load the malicious IDT
    lidt        [rax]
    ; The rest of the code.
```

```
thread2_asm:
    ; Barrier, to synch with the first
    ; thread.
    lea         rax, [rel barrier]
    lock inc    qword [rax]
_wait2:
    cmp         qword [rax], 2
    ; Loop until the other thread got to
    ; the barrier as well.
    jnz         _wait2
    ; Waste some clocks, to make sure that
    ; the UD2 in the other thread got to
    ; generate the exit.

    ; Patch the the opcode, to 'LIDT'
    mov         byte [rel _target + 1], 0x01
    xor         rax, rax
    retn
```

**Figure 8: Assembly code of two threads forcing the emulation of LIDT**

## EXPLOITING THE VULNERABILITIES

### INT (SOFTWARE INTERRUPT)

The **INT** instruction can be executed at any privilege level. It has only one immediate operand which is always 8 bits in size and encodes the IDT entry that gets invoked (see the description of the LIDT vulnerability for more details about the IDT and IDT entries). When **INT** is executed, the CPU will automatically lookup the desired entry inside the IDT. If the entry Descriptor Privilege Level is numerically equal or greater to the privilege level at which the **INT** was executed than the handler will be called. Otherwise, a General-Protection fault will be generated.

Using the software interrupt vulnerability and the race-condition presented in the previous section, we can cause a denial of service inside the guest VM. All we need to do is force the emulation of the software interrupt instruction. If we invoke an unexpected entry (for example, 0x0F, which is reserved), we can cause the guest operating system to generate a BSOD. While this has not been tested on Linux, it is expected to cause similar effects (kernel panic).

### LMSW (LOAD MACHINE STATUS WORD)

**LMSW** (Load Machine Status Word) can be executed only when CPL (Current Privilege Level) is 0. This instruction can modify bits 0, 1, 2 and 3 which stand for PE (Protection Enabled), MP (Monitor Co-Processor), EM (Emulation) and TS (Task Switched). However, the PE bit can only be set by this instruction – it cannot be reset once it has been set.

Exploiting the emulation of **LMSW** will allow the attacker to modify bits 1, 2 and 3 (MP, EM & TS). These bits have effect on FPU instruction execution. Using certain combinations will cause a Device Not Available exception (#NM), which would be handled by the operating system.

## HLT (HALT)

**HLT** will cause the CPU to enter the halt state and cease any instruction execution. The CPU will resume normal operation once an interrupt is received. Exploiting this in our case can't do too much damage, as we basically just halt our exploit code until the next interrupt.

## LGDT (LOAD GLOBAL DESCRIPTOR TABLE)

**LGDT** is a privileged instruction (can be executed only in real-mode or in ring0) that loads the limit and the base address of the Global Descriptor Table (GDT). The GDT contains special entries, named *descriptors*. There are several types of descriptors that may be present inside a GDT:

1. Segment descriptor (either code or data)
2. Local Descriptor-Table descriptor (LDT)
3. Task-State Segment descriptor (TSS)
4. Call-Gate descriptor
5. Task-Gate descriptor

If someone can load its own GDT, it can control what kind of descriptors will be present inside this GDT. A method to exploit this in order to gain ring0 privilege would be, for example, to load a GDT with, among others, a call gate to a ring0 handler, which would be located inside the attackers program. By invoking that entry, the designated routine will get executed in ring0. Care must be taken though because certain entries inside the GDT are very important (for example, ring0 code & data descriptors, ring3 code & data descriptors, task-state segment) and the operating system may also have some descriptors that point to certain internal structures (for example, the GS register points to the KPCR – Kernel Processor Control Region on 64 bit Windows) which should be present in the malicious GDT. Otherwise, the operating system will simply crash. Exploiting the **LGDT** instruction to gain ring0 privileges is more challenging than exploiting the **LIDT** instruction.

## LIDT (LOAD INTERRUPT DESCRIPTOR TABLE)

**LIDT** is a privileged instruction (can be executed only in real-mode or in ring0) that loads the limit and the base address of the Interrupt Descriptor Table (IDT). Certain special events may take place in the system, currently running program or the CPU itself that require attention. These events will interrupt the currently executing instruction by transferring control to a special handler. Among the events that will be handled by calling an entry that resides inside the IDT are:

- Hardware interrupts (generated by hardware devices such as a network card, a mouse, etc.)
- Software interrupts (generated by software via the **INT** instruction)
- Hardware exceptions (such as a Page Fault, a General-Protection Fault, a Division by zero, etc.)
- Task switches

Whenever such an event takes place, the CPU will automatically invoke a certain entry inside the IDT (which entry will be invoked depends on the event). An IDT entry may look different, depending on the entry type. A 32 bit Interrupt Gate is depicted in Figure 9: Interrupt Gate descriptor:

- Segment Selector – contains the selector of the code-segment that will be loaded when invoking the entry.

- Offset – offset inside the previously loaded code-segment of the handler routine
- DPL – Descriptor Privilege Level required to directly invoke the entry via INT, INT3 or INTO instructions

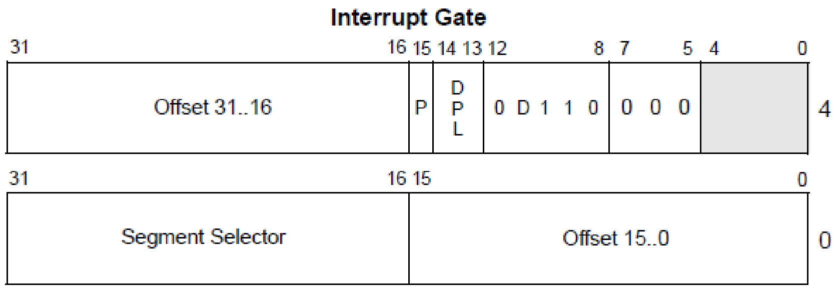


Figure 9: Interrupt Gate descriptor[8]

When a software interrupt takes place, the CPU will automatically lookup the desired entry inside the IDT. It will make sure the caller has enough privilege, it will make several other sanity checks and then it will load the segment indexed by Segment Selector into the current Code Segment and it will pass control to the routine pointed by the Offset. Usually, the Segment Selector will select a ring0 (kernel) code segment, in order to make sure that the routine will execute with maximum privilege.

If someone is able to execute **LIDT**, he can load his own IDT with his own custom malicious handlers. The exploitation of the **LIDT** emulation issue is fairly straight-forward:

1. Build a custom Interrupt Descriptor Table – each IDT handler will be specially crafted so that the segment selector points to a ring0 segment descriptor (0x08 on x86 Windows and 0x10 on x64 Windows) and the offset points to a malicious routine inside the attacker's program.
2. Force the emulation of the **LIDT** instruction, so that the malicious IDT is loaded – this can be achieved using the method described in the previous section, Forcing the emulation of arbitrary instructions.
3. Once the malicious IDT has been loaded, the attacker may either choose to wait for an external interrupt which will trigger the execution of the payload, or it can invoke the payload itself via the software interrupt instruction, **INT**. Doing this requires the invoked entry to have DPL = 3, but we control the IDT entries, so this can be easily achieved during step 1.
4. Execute the ring0 payload – this can do whatever the attacker wants to. The attack may vary from granting SYSTEM privileges to arbitrary processes to loading and executing other malicious components in ring0.
5. Restore the original IDT – the attacker must save the original IDT (which can be retrieved using the unprivileged instruction **SIDT**) before everything can resume to normal. If the event that lead to the invocation of the IDT entry was external, the attacker must also call the original handler, in order to avoid stability issues (for example, a missed clock interrupt or a missed inter-processor interrupt may cause significant stability issues).

Please note that **LIDT** can be used on both x86 and x64 operating systems in order to exploit this vulnerability. The main differences are with regards to the layout of the IDT and the IDT descriptors.

[8] Intel SDM, 6-11, Vol. 3A, Figure 6-2. IDT Gate Descriptors

## BYPASSING SMEP

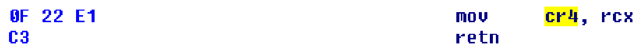
The exploitation technique presented in the previous section works well on both x86 and x64 systems, but, if present and activated, Intel Supervisory Mode Execution Prevention will block this attack from being performed successfully. However, using the **LIDT** instruction, we can bypass SMEP and successfully gain code execution in ring0.

### ABOUT SMEP

SMEP stands for Supervisory Mode Execution Prevention and was introduced in 2013 with the Intel Ivy Bridge CPUs. SMEP can be activated by setting bit 20 in CR4. Once activated, SMEP will block any attempt to execute code located inside a user-mode page from rings other than ring3. This technology is capable of blocking a wide range of exploitation attempts that rely on executing user-controlled data with ring0 privileges. Our previously described attack is blocked as well, as it relies on pointing an interrupt handler inside the user program, which lies in user-pages. When the interrupt handler is invoked, a ring0 code segment will be loaded by the CPU and as soon as it will try to fetch the first instruction from the malicious handler (located in a user page), a Page-Fault will be generated.

### CIRCUMVENTING SMEP

In order to bypass SMEP, we will use the fact that we can point the IDT handlers anywhere, including inside kernel memory. Therefore, we could completely disable SMEP by clearing bit 20 inside CR4, if we could find and adequate code sequence inside the kernel. While explicit clearing of the SMEP bit in CR4 is not done anywhere inside the kernel, there are other interesting sequences. The most suitable for our purpose is the one depicted in Figure 10: SMEP-Disable gadget. There are several other sequences identical to this one throughout the kernel image, but we only need one. What the indicated sequence does is load the content of the RCX register inside the CR4 register and then returns control to whatever lies on the top of the stack.



```
0F 22 E1                                  mov     cr4, rcx
C3                                        retn
```

Figure 10: SMEP-Disable gadget

In order to actually use this to bypass SMEP, we first have to understand two things when it comes to interrupts.

First of all, whenever an interrupt or exception takes place, the CPU will preserve all the general purpose registers (except for the stack pointer). This means that if inside our user-mode program we have RCX = 0xBDBDBDBDBDBDBDBD, when an interrupt takes place and the interrupt handler will be invoked, RCX will contain the same value. It is the handler's duty to save the registers and restore them before returning to the interrupted code.

Secondly, we have to understand what happens to the stack whenever an interrupt takes place. If the interrupt takes place while in ring3, a stack switch will occur. The newly loaded stack is the ring0 stack described inside the current Task State Segment (TSS) or a stack identified by the Interrupt Stack Table (IST) value in the IDT descriptor (in IA32e mode). After switching the stack due to the interrupt, the CPU will save (in this order) the old SS, old RSP, Flags, old CS and the interrupted RIP, as shown in Figure 11: Stack Switching. Also, since we're dealing with a software interrupt or an external interrupt, no Error Code will be pushed on the stack (which is important for us, since we want to have a valid RIP at [RSP] instead of an error code).

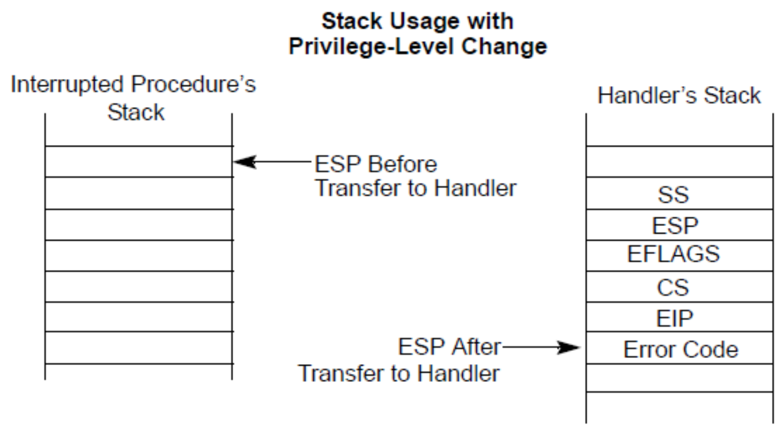

Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
Handler's Stack
ESP Before Transfer to Handler
SS
ESP
EFLAGS
CS
EIP
ESP After Transfer to Handler
Error Code

Figure 11: Stack Switching[9]

Knowing all this, it will now become clearer that a "**MOV CR4, RCX**/**RETN**" code sequence is more than enough to accomplish our task to disable SMEP. All we need to do is make sure that RCX contains a valid CR4 value (with SMEP disabled) and that we can force an interrupt to be generated. The required steps are:

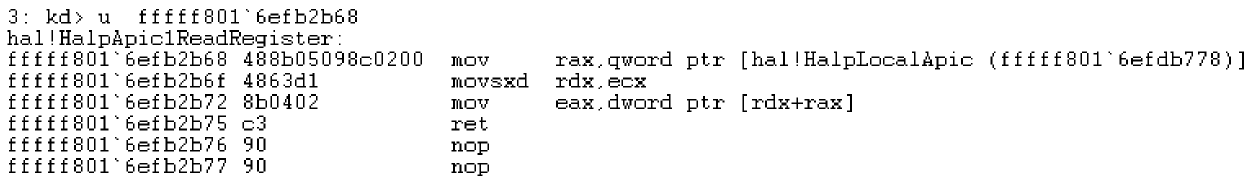
1. Store a valid CR4 value inside RCX, with SMEP disabled (CR4 contains mainly feature-enable bits and its value can be determined easily known the operating system and the capabilities of the CPU; however, we can restore CR4 once we're inside ring0, so we only need to make sure RCX contains a minimum of features enabled, such as Physical Address Extension).
2. Load our malicious IDT (using the race-condition described in Forcing the emulation of arbitrary instructions), which contains handlers that point to the "**MOV CR4, RCX**/**RETN**" gadget described in the previous section
3. Force a software interrupt, which will cause the "**MOV CR4, RCX**/**RETN**" sequence to execute; this sequence does two important things: first of all it disables SMEP by writing CR4, secondly it does a **RETN**, which will return to the RIP stored in the top of the current stack. Since the handler executes with ring0 privileges and we are doing a simple **RET** instead of an **IRET**, the ring0 code segment will be preserved. Therefore, the small code gadget will disable SMEP and it will return to the instruction following the forced software interrupt (inside our code), which will execute with ring0 privileges.

## HANDLING MISSED INTERRUPTS

Since we want a certain level of control with regard to when the payload gets invoked, we can point only one handler to the SMEP-disable gadget, and all other handlers can simply point to an **IRET** inside the kernel. This way, no matter what interrupt takes place, it will immediately return. When we are ready to launch the exploit, we will manually execute **INT** in order to trigger the vulnerability. This introduces another issue: the interrupts that were silently discarded must be handled, otherwise the system will hang (an EOI must be performed for each external interrupt). In order to call the missed interrupt handlers, we can simply parse the Local APIC In Service Register (ISR) and determine each interrupt that took place but was not handled. We have to invoke all the handlers backwards (the higher the interrupt number the higher the priority). After we're done with this, we can freely execute our payload and do whatever we want in ring0.

[9] Intel SDM, 6-11, Vol. 3A, Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

In order to parse the ISR, we have to first determine whether the VCPU is in x2APIC or in xAPIC mode. This can be achieved by checking bit 10 inside the IA32_APIC_BASE model specific register (address 0x1B). If the VCPU is in x2APIC mode, the ISR is located in model specific registers range 0x810-0x817. If the VCPU is in xAPIC mode, the task is a little more difficult, as we have to discover the mapped base address of the local APIC. One way to do this is by finding a certain code sequence inside the hal that makes use of it. An example is HalpApic1ReadRegister, illustrated in Figure 12: The HalpLocalApic variable containing the LAPIC mapped base. Once we've found this sequence, we can extract the mapped base of the local APIC from the HalpLocalApic internal variable, used in the first instruction. Another way to find the mapped base of the local APIC is by using the MmGetPhysicalAddress routine (which according to MSDN can be called at any IRQL) and do a brute force search until we find the virtual address that translates to the local APIC physical address (the local APIC physical address can be obtained from IA32_APIC_BASE MSR).



```
3: kd> u  fffff801`6efb2b68
hal!HalpApic1ReadRegister:
fffff801`6efb2b68 488b05098c0200  mov     rax,qword ptr [hal!HalpLocalApic (fffff801`6efdb778)]
fffff801`6efb2b6f 4863d1          movsxd  rdx,ecx
fffff801`6efb2b72 8b0402          mov     eax,dword ptr [rdx+rax]
fffff801`6efb2b75 c3              ret
fffff801`6efb2b76 90              nop
fffff801`6efb2b77 90              nop
```

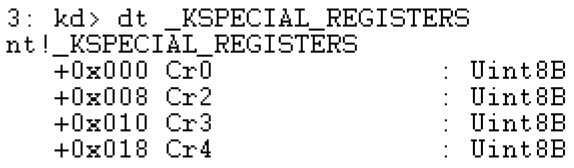
Figure 12: The HalpLocalApic variable containing the LAPIC mapped base

## RETURNING TO RING3

Once we've executed the payload, we can return to ring3. As we are still executing with a ring0 stack that already contains the old SS, RSP, FLAGS, and the ring3 CS (as they were when we triggered the software interrupt), we can simply push the address of the ring3 handler (remember that the old RIP has been used by the **RET** inside the SMEP-disable gadget in order to return control to our code) on the stack and execute the **IRET** instruction.

## PATCHGUARD AND CR4

Since we modify CR4, PatchGuard will eventually notice this and it will generate a BSOD. This is trivial to avoid by restoring the original CR4 from a small code chunk injected inside the kernel (since we restore the original CR4, SMEP will become active again and we need to make sure that our code now lays inside a ring0 page – but this is trivial to do when you have ring0 privilege). The original CR4 can be discovered using the KSPECIAL_REGISTERS structure inside the current processor control block. As these are undocumented structures, we can leverage the location of the CR4 field inside the KSPECIAL_REGISTERS structure, as illustrated in Figure 13: A part of the KSPECIAL_REGISTERS structure, and use the value of CR0 as a search tag. We will start searching from the beginning of the kernel processor control region, which is pointed by the IA32_GS_BASE MSR on x64 and IA32_FS_BASE MSR on x86. However, this method is not very reliable, as sometimes, a CR4 value that seems outdated (with several features, including SMEP, disabled) is located inside this structure. Another way to obtain the original CR4 value is by sending an IPI to another VCPU and obtain it from there (assuming the system will have the same CR4 on all CPUs, which should be the case). Also, the entire payload could be injected inside a kernel page in order to further reduce the window when PatchGuard could catch us with a modified CR4. An example of areas where the small stub that restores CR4 could be injected is the unused space at the end of the PE sections. We can easily find available space by parsing the module's PE headers. This space must be inside an executable section, but it's not necessary to be writable – we can simply clear the WP (Write Protect) bit inside CR0, which will allow us to write non-writable pages while in ring0. The injected code chunk must contain only a few instructions, the most relevant being the restoring of CR4 and the other being the **IRET**, which will return to ring3 code once our payload is done executing, as described in the previous section.



```
3: kd> dt _KSPECIAL_REGISTERS
nt!_KSPECIAL_REGISTERS
   +0x000 Cr0              : Uint8B
   +0x008 Cr2              : Uint8B
   +0x010 Cr3              : Uint8B
   +0x018 Cr4              : Uint8B
```

**Figure 13: A part of the KSPECIAL_REGISTERS structure**

## PREVENTION

Aside from the obvious solution of applying the latest patches from Xen (which fix these vulnerabilities –Xen Security Advisories XSA-105 and XSA-106), this kind of exploitation should be easily blocked via the newly introduced SMAP technology (in Intel Broadwell CPUs) – Supervisory Mode Access Prevention. This technology forbids supervisory accesses inside user pages. Implicit accesses made to the IDT or GDT are always considered supervisory accesses. Thus, if we store the IDT inside a user page, we will cause SMAP to generate a Page-Fault when an interrupt occurs. However, the Page-Fault handler can't be invoked, as SMAP will generate another Page-Fault, which will lead to a Double-Fault. Invoking the double-Fault handler is not possible and eventually a Triple-Fault will be generated. Therefore, the attack will be limited to plain denial of service instead of code execution in ring0. However, CPUs with SMAP support will probably be released in Q4 2014 and hardware support is not enough – the Operating System has to actually enable this feature in order to make use of it (most likely Windows 10 will). This means that the attack can still succeed on most of the systems running unpatched versions of Xen.

## CONCLUSIONS

Although hypervisors provide a certain level of security by isolation, sometimes they facilitate certain types of attacks. In this whitepaper we've presented a practical attack against the Xen x86 emulator, which can lead to arbitrary code execution in ring0 or denial of service by crashing the guest. In addition, in the context of the presented attack, Supervisory Mode Execution Prevention has been circumvented, making the attack functional and reliable on most of the Intel CPUs and Microsoft Windows operating systems. The proof of concept has been successfully tested on: Microsoft Windows 7 x86 (without SMEP bypass) and Microsoft Windows 8 x64, Microsoft Windows 8.1 Update 1 x64, Microsoft Windows Server 2012 x64, Microsoft Windows 10 Technical Preview build 9841 x64 (with SMEP bypass), all running on top of Citrix XenServer 6.2, with Intel® Core™ i7-4770, and on Microsoft Windows 8.1 Update 1 x64 (with SMEP bypass) running on top of Citrix XenClient 5.1.3 and Citrix XenClient XT 3.2.2 Trial, build 132629, both with Intel® Core™ i7-3770. Although the proof of concept did not work properly inside an Amazon VM running Microsoft Windows Server 2012 x64, it did crash the guest, indicating that the vulnerability is present.

## BIBLIOGRAPHY

Intel Corporation. (2014, September 23). *Intel Architectures Software Development Manual.* Retrieved September 23, 2014, from www.intel.com: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

Luft, M. (n.d.). *Exploiting Hyper-V: How We Discovered MS13-092*. Retrieved from Insinuator: http://www.insinuator.net/2014/01/exploiting-hyper-v-how-we-discovered-ms13-092/

VMware. (n.d.). *VMware Security Advisories*. Retrieved from VMware Security Advisories: http://www.vmware.com/security/advisories

Xen Security Advisory. (n.d.). *Xen Security Advisory*. Retrieved from Xen Security Advisory: http://xenbits.xen.org/xsa/