Following the White Rabbit:
Software attacks against Intel$^{(R)}$ VT-d technology

Rafal Wojtczuk
rafal@invisiblethingslab.com

Joanna Rutkowska
joanna@invisiblethingslab.com

---===[Invisible Things Lab]===---

April 2011

Abstract

We discuss three *software* attacks that might allow for escaping from a VT-d-protected driver domain in a virtualization system. We then focus on one of those attacks, and demonstrate **practical and reliable code execution exploit against a Xen system**. Finally, we discuss how new hardware from Intel offers a potential for protection against our attacks in the form of Interrupt Remapping (for client systems available only on the very latest Sandy Bridge processors). But we also discuss how this protection could be circumvented on a Xen system under certain circumstances...

Wake up, Neo...

The Matrix has you...

Follow the white rabbit.

Table of Contents

Introduction

What is Intel VT-d?

Intel VT-d, a hardware technology for device virtualization, also known as IOMMU, is one of the several key technologies that together comprise ~~The Matrix~~ Intel Virtualization Technology (VT). VT-d complements another hardware technology, VT-x, which is used for CPU and memory virtualization. VT-d is also an important element for Intel TXT technology, which itself is Intel's key approach to Trusted Computing.

One can argue that Intel VT-d and Intel TXT are the two most important technologies for building secure operating systems, especially desktop systems [2], no matter whether one builds a system based on safe language, on formal verification of a small microkernel, or on some other approach,

Previous work on attacking Intel VT-d

We are not aware of any previous work on software attacks against Intel VT-d technology, nor against any other IOMMU technology.

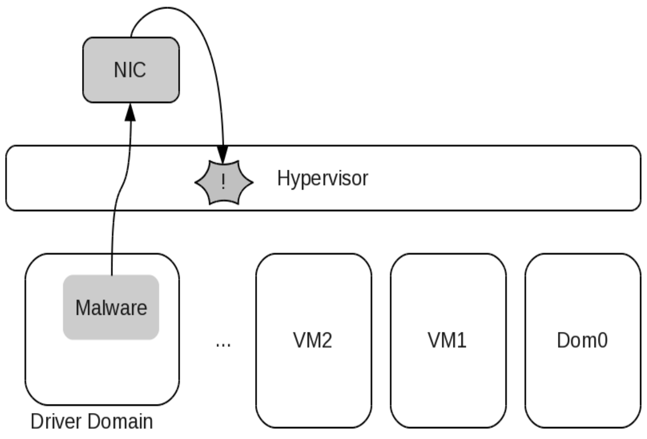
The attacks discussed in this paper abuse devices ability to send malicious interrupts, and, as discussed at the end of the paper, can be prevented with Interrupt Remapping, technology available on recent Intel platforms (in case of client systems, only on the very latest Sandy Bridge processors), but, we have found no mention in the official Intel specs, that Interrupt Remapping is security-critical element of VT-d.

To be fair, however, we have found presentation slides from Intel engineers [3] that briefly mention the security role of Interrupt Remapping, specifically saying that "Without [Interrupt Remapping], malicious guest can attack a host by [g]enerating interrupts". However, the presentation doesn't mention what type of attacks could be carried this way, how serious they could be, and generally doesn't provide any details on those attacks. This might create a wrong impression that the only attacks possible are DoS attacks – as e.g. expressed by one of the KVM developers [4].

Our paper shows how the lack of Interrupt Remapping might allow for the most critical code execution attacks on a VMM system.

The attack scenario

In this paper we assume a modern virtualization system which makes use of Intel VT-d in order to create so called *driver domains*, or *driver virtual machines (VMs)*. Such domains are similar to traditional guests, with an exception that they have been assigned direct access to some select physical devices, such as network cards, or disk controllers.



Driver domains are desirable on both server, as well as on client systems. In case of the former they could bring significant performance benefits over the traditionally virtualized devices. New hardware technologies, such as Single Root I/O [5], allow to *natively* share a single physical device among many guests. Example of virtualization systems that support driver domains are Xen [6], and VMWare ESX [7].

On client systems, driver domains could be used to improve security of the system by sandboxing select subsystems and drivers, such as the networking subsystem and networking drivers [2]. Qubes OS [8] and Xen Client [9] are examples of client systems that use VT-d-isolated network domains. Xen Client also requires VT-d to allow for direct access to the GPU to select user VMs. In this case, all the user VMs that have been granted access to the GPU could be also considered as *driver domains* for the purpose of this paper, and so could be used to trigger the attack (ironically, usually the least trusted VMs would be allowed direct access to the GPU, such as e.g. “gaming” VMs).

The attacks that we consider in this paper originate from one of such driver domains and attempt to gain full control over the whole system[1]. We assume that the attacker already somehow managed to gain full control over the driver domain – e.g. as a result of exploiting a buggy WiFi driver [10] or DHCP client [11] in the “Net VM”. Normally the properly used IOMMU should constrain the attacker from compromising the rest of the system[2].

We also assume the platform doesn't support Interrupt Remapping, which is true e.g. for all client systems before Sandy Bridge processors. All the experiments have been conducted on a 64-bit Xen 4.0.1.

1 Specifically, our exploit, presented later in this paper, succeeds at executing arbitrary shellcode in the hypervisor.

2 We should note, that without hardware IOMMU technology, such as VT-d, it's not possible to safely grant a VM access to any DMA-capable device, as the VM would always be able to attack the rest of the system via DMA attacks (see e.g. [12])

Message Signaled Interrupts (MSI)

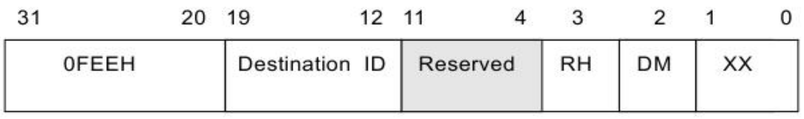
The attacks we describe later in this paper work by forcing the corresponding device[3] to generate a so called Message Signaled Interrupt (MSI). Thus, we start by first describing what MSI is, and also methods for generating MSIs.

The MSI interrupt format

Legacy systems used special out-of-band mechanisms for interrupt signaling (special pins and wires on the motherboard). All newer systems, especially those which employ PCI Express interconnect, use an in-bound mechanism for interrupt signaling that is called *Message Signaled Interrupts.*

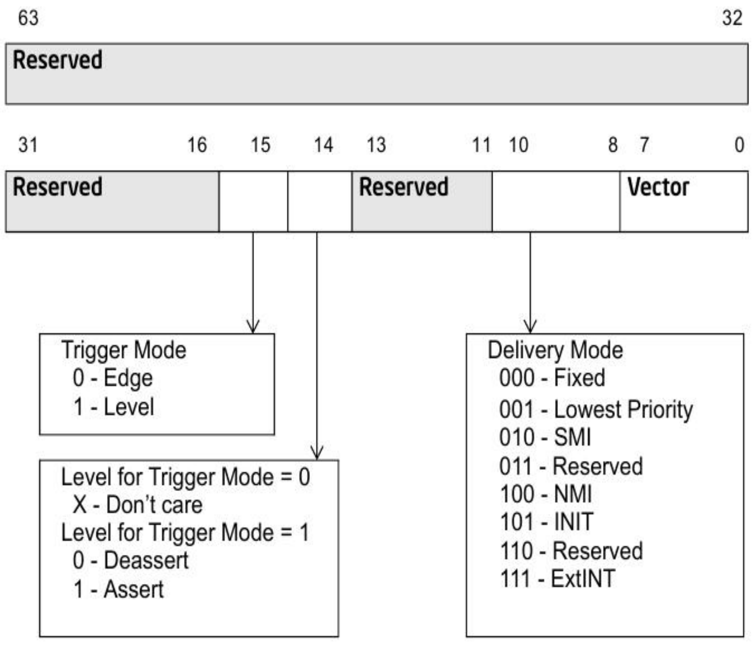
From the device's point of view, an MSI is an ordinary PCI/e Memory Write transaction, just that it is destined to a special physical address. Those special addresses for MSIs that are recognized by the processor[4] as interrupts are defined by Intel in the Software Development Manual (vol. 3A, chapter 10.10).

Specifically, the figure below shows the format of the destination address for MSI (a copy from Intel SDM):



So, any[5] PCIe write transaction to address `feeXXXXh` results in an interrupt being signaled to one of the CPUs in the system. The specific CPU(s) can be further selected via the *Destination ID, RH and DM* fields.

The data part of the PCIe write packet that signals the MSI has the following format (a copy from Intel SDM):



3 That is the device assigned to the driver domain.

4 In fact it's the Local Advanced Programmable Interrupt Controller (LAPIC) that translates MSI writes into interrupts and delivers them to the CPU. Of course on modern hardware, LAPICs are part of the processor die.

5 The additional requirement is that the data payload be exactly 4 bytes, as it is discussed later.

There are two important fields here:

- The *Vector* field, that in most cases translates to the interrupt vector that is signaled to the processor (these are the same interrupt vectors as used for IDT addressing, allowed values are in the range: 0x10-0xfe),
- The *Delivery Mode* that tells LAPIC how the interrupt should be interpreted. The most common value here would be *Fixed* or *Lowest Priority*, which would cause LAPIC to deliver "normal" interrupt with the vector specified in the *Vector* field.

Below we investigate three MSI attacks originating from driver domains: one to deliver a SIPI interrupt, one to inject a syscall or a hypercall interrupt, and finally one to inject an #AC exception to the system. All of those attacks work by filling the above fields with some special values.

Software approach to generate MSIs

What makes the MSI-based attacks especially interesting, is that in most cases it is possible to mount such an attack *without* cooperating hardware (malicious device), using entirely innocent and regular device, such as an integrated NIC[6].

This is possible, because each MSI-capable PCI/PCIe device contains a special capability registers in its PCI configuration space that are used to configure MSI signaling. Specifically, the MSI capability contains two registers that lets the system software to configure the *address* and *data* payload of packets used for MSI generation (see above). This means that system software can configure any MSI-capable device to generate *any* type of MSI – with arbitrary *vector* and *delivery mode*.

This means that all that is necessary for software to configure a device to generate arbitrary MSIs is an access to the device configuration space.

Configuration space access restrictions on Xen

We should stress, however, that some VMMs, such as Xen, implement special precautions in order to restrain driver domains from being able to fully control the assigned devices' configuration space.

In case of para virtualized driver domains (PV domains), Xen, by default, doesn't allow write-access to most of the device configuration space[7]. Because some devices might not function correctly in this case, Xen offers two workarounds. First (the default one) there is a small database of per-device "quirks", i.e. listings of configuration space registers for each specific devices, identified by DID:VID, to which the guest should be granted write-access. Another option is to set the special *permissive* flag[8], which globally causes all driver domains to gain full access to the configuration spaces of their corresponding devices.

A few months ago there have been an interesting discussion on the Xen-devel mailing list about the purpose of this very flag and reasons why users might *not* want to use it[9]. The discussion was, incidentally, started by one of the authors of this paper (back then unaware of the threats from MSI-based attacks), and involved participation of key Xen developers. The discussion generally reached a conclusion that it was *safe* to set the *permissive* flag, because VT-d should still be able to prevent any potential damage that a compromised driver domain or a malicious device could do to the rest of the system, while at the same time providing users with the most smooth operation of their devices, without the need for using per-device quirks. With a notable exception of Ian Pratt (Xen Chief Architect), who expressed concerns of some devices potentially being able to trigger SMIs in response to configuration space accesses and also to potential problems with intentionally overlapping MMIO regions, which VT-d could not prevent. None of the participants pointed to problems caused by MSIs.

The bottom line of the above is that even to a group of the most experienced Xen developers, it wasn't clear that driver domains should really be constrained from accessing their assigned devices' configuration space, and some even recommended to always allow full access to configuration space.

In the case of fully virtualized Xen guests (HVM guests), the access to the configuration space is also limited, and granted mostly to *unknown* registers, and some known registers that are considered safe, such as the

6 There are also ways to compromise VT-d using malicious devices, but in this paper we focus only on software attacks, and assume that all the hardware is intact.

7 Some of the allowed write accesses include the *command* register, so e.g. enabling/disabling of MSI, bus master enable bit, etc., but not defining MSI parameters, such as address and data payload for MSI packets.

8 The permissive flag is an argument for the Xen PCI backend.

9 See the full thread here: http://lists.xensource.com/archives/html/xen-devel/2010-07/msg00257.html

C*ommand* register. The MSI configuration registers are among the *known* registers, and accesses to them are always emulated by Xen. Thus, HVM guests cannot spawn an MSI attack by directly manipulating their devices' MSI configuration registers.

Generating MSI without access to device config space

However, we have discovered that even without access to the device configuration space[10], still in case of many devices the driver would be able to program the device to generate an MSI, and consequently could still mount a software-only attack against VT-d. This is because many devices support a so called *Scatter Gather* mechanism, that allows to split one DMA transaction into several smaller ones, each destined to a different memory location[11].

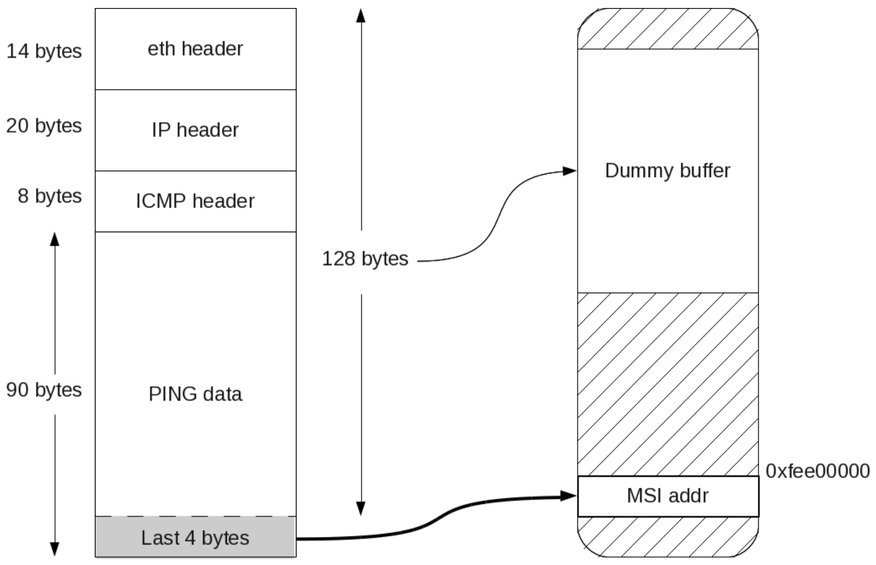
The idea is to use such a scatter gather mechanism in order to generate a 4-byte memory write transaction that will just happen to be destined to a special 0xfeeXXXXX address – in other words to generate MSI using regular DMA write.

MSI generation on Intel e1000e NIC

I know Kung Fu.

-- *Neo*

To provide a practical example that it is indeed feasible to generate MSI from an *untrusted* driver domain that doesn't have any special access to the device configuration space, we have used the Intel integrated network card, the popular *e1000e*[12]. We have programmed the card's scatter gather unit to always split the incoming packets into two parts: the first 128 bytes, which will be written to some *dummy* address, and the remaining part that will be sent to the magic 0xfeeXXXXX address. Now, if we send an ICMP echo request with exactly 90-bytes of payload this will result in the ICMP echo reply packet coming back (with the same payload) with a total size of 128+4 bytes, i.e. counting also the Ethernet frame. If we carefully chose the payload of ICMP Request, so that the last 4 bytes contain a meaningful MSI data payload, then the delivery of those last 4 bytes of the packet will cause MSI generation in the system!



In practice, the attacker will restore NICs normal operation soon after the MSI was generated. It might also be possible to use the NIC in a loopback mode, so without the need to relay on some external host to ping packets back to us[13].

10 So, e.g. when considering an attack coming from a Xen HVM domain, or from a PV driver domain run with the permissive flag cleared.

11 Most devices do not allow for generation of very small DMA writes and we need a 4-byte write to emulate MSI – no more, no less, but exactly 4 bytes of data payload. Hence the idea to use Scatter Gather mechanism.

12 Naturally the integrated NIC is a likely candidate to be assigned to the system's untrusted Net VM.

13 The key observation here is that if we want to use NIC for DMA write to the host DRAM, we must somehow deliver a packet to the NIC, so it could write it to the RX buffer in the host memory.

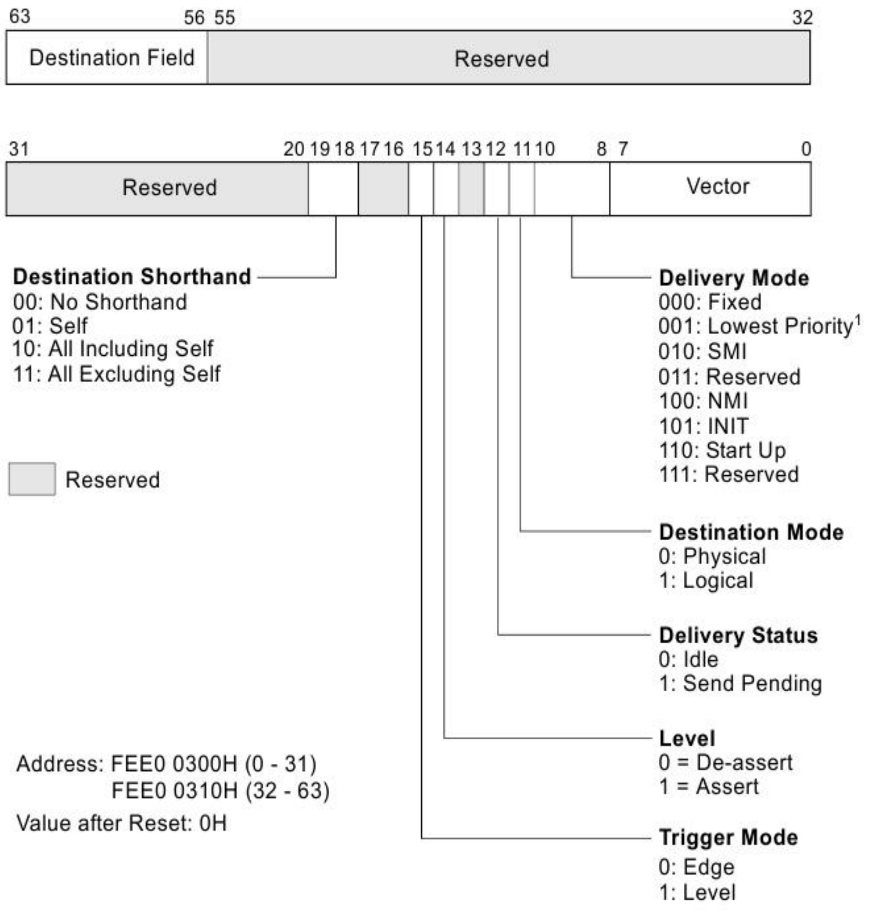
The Attack #1: The SIPI attack

About SIPI interrupt

The SIPI interrupt, which stands for *Start-up Inter Processor Interrupt*, provides a key functionality to any multi-processor (or multi-core) Intel-based system. It is the SIPI interrupt that is used by the BIOS to initialize all the processors upon boot, and to distribute work amongst them. When the platform boots, only one processor is active – it is called the Boostrap Processor or BSP – and its the job of the BSP to initialize and get other processors (called Application Processors or AP) running.

SIPI interrupt directs the destination CPU to start executing special start-up code located at address 0xVV000, where *VV* is the vector passed as part of the SIPI interrupt. In order for SIPI to have any effect on a CPU, the CPU must first be sent an INIT interrupt, which resets the CPU and puts it into a wait-for-SIPI state. In normal situation BSP sends SIPI interrupts to all other processors in the system (APs).

The only documented mechanisms for delivering a SIPI interrupt is via programming Local APIC control register called Interrupt Command Register (ICR) as described in Intel Software Developers Manual, Vol. 3A, chapter 10.6. The ICR register is depicted below (a copy from Intel SDM):



The LAPIC registers are mapped into physical address space (typically starting at base address 0xfee00000) and can be accessed only from a CPU.

However, we have found a way to generate SIPI interrupts also from PCIe devices...

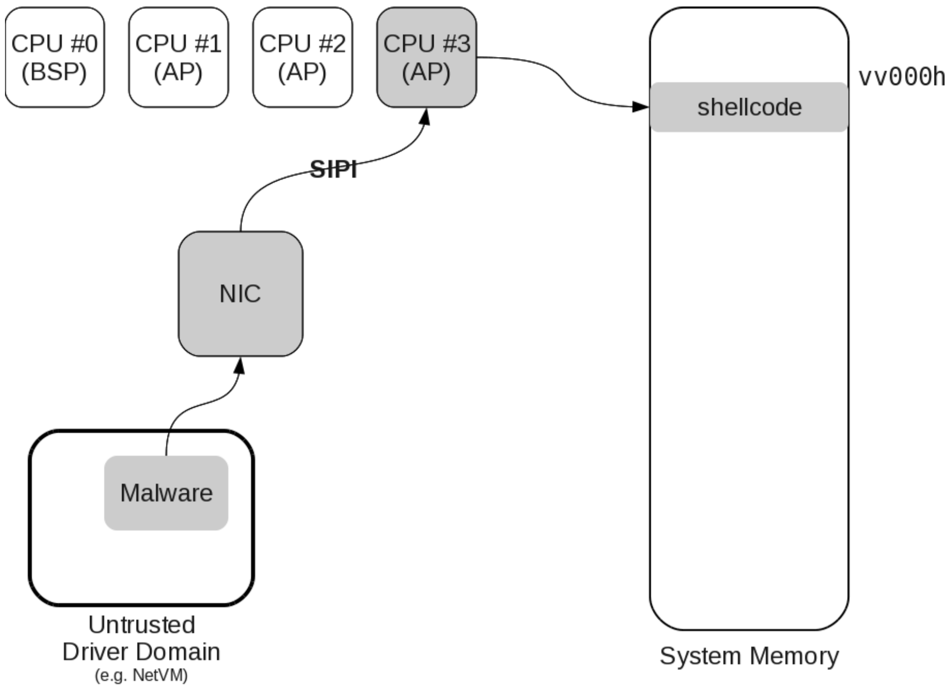
The SIPI Attack Idea

If we look closely at the MSI data payload and compare it with the ICR register described in the previous section, we can spot some common fields, such as *Vector* and *Delivery Mode*.

Interestingly, the *Start Up* Deliver Mode (so, the SIPI interrupt), which is available via LAPIC's ICR register, is not present in the MSI data format. The binary value 110b*,* which otherwise we would expect to be used for

SIPI in MSI packets, is marked as *Reserved* in the MSI packet documentation.

But apart from this, the similarities between the format of the *Delivery Mode* and *Vector* fields between ICR and MSI data packet are striking. So, we couldn't resist to see what would happen if we send an MSI packet from our device with a *Delivery Mode* set to (officially forbidden) value 110b. Just as we anticipated, this resulted in a SIPI interrupt being delivered to one of the AP processors!



This is a significant result, because in our MSI message we can additionally specify a *Vector* field, which will be interpreted as part of a physical address (0xVV000) where the receiving CPU should jump to, and from where it should start fetching instructions.

In other words, we can selectively restart one of the processors in the system and get it executing code from an almost arbitrary address (constrained to 0xVV000). This means that if the attacker managed to place (or find) a shellcode within the 0-1MB range, the shellcode will be executing unconstrained by any system isolation mechanisms, and will have access to the full system memory, e.g. to all processes or virtual machines memory.

Shellcode injection difficulty

One mitigating factor for the above attack is the potential difficulty for the attacker to place (or find) meaningful instructions (the shellcode) in the physical memory below 1MB, and more precisely starting at page boundaries (aligned to 0x1000). We consider this to be a system-software-specific challenge, but definitely not impossible.

For example, if the system software makes use of the memory below 1MB (specifically 0-640kB) and allocates this memory, like any other non-reserved memory, to various processes or Virtual Machines, then it might be easy for the attacker to control this memory and place a meaningful shellcode there, and later trigger INIT and SIPI and get it executed with system privileges[14]. Indeed, if not for our SIPI attack, there is really nothing that should prevent the system-software from using this low memory.

System-software that doesn't allocate low memory to its processes or VMs, like e.g. Xen hypervisor, might still be exploitable, although we haven't figure out yet a general method to achieve it.

VMX and INIT blocking

Intel processors offer a partial mitigation against SIPI attacks by blocking INIT interrupts when a CPU is in the VMX root[15] mode (VT-x).

14 We have found, however, that Xen hypervisor never allocates the low 1MB of memory to any of the guests. This doesn't mean other VMMs or OSes do not do this as well.

However, an INIT and SIPI interrupts sent to a CPU during time when it is in a VMX mode are remembered and delivered, perhaps hours later, when the CPU exits the VMX mode (so immediately after it executes VMXOFF instruction)[16].

This means that an attacker, who manged to send INIT and SIPI interrupts at some point when all the processors were in a VMX mode, can still hope to get the shellcode executed at a later time, specifically when the system will be going for reboot or shutdown[17]. This means the attacker's shellcode will be executed with full ring0 permissions at the stage when the system is shutting down.

This presents two potential attack possibilities for the attacker. First, if the operating system or VMM doesn't properly clean the memory, then the attackers shellcode will be able to steal some sensitive data that still remains in DRAM, and likely will also be able to leak them out, e.g. through one of the NIC interfaces.

In case the operating system or VMM is carefully designed to scrub the memory *before* disabling VMX, the attacker will not gain immediate advantage of executing the shellcode at shutdown time. However, because the attacker's shellcode executes with ring0 permissions, it opens many opportunities for the attacker to spawn additional attacks, that normally would not be possible to conduct, such as attacks on BIOS flash memory, or TXT bypassing attacks.

Indeed, as it has been demonstrated *in practice*, an attacker who had access to the Master Boot Record (i.e. can infect MBR, or some early OS loader, such as GRUB), was able to: 1) reflash secure Intel vPRO BIOS with arbitrary code, despite digital signatures were used for protecting the BIOS [15], 2) bypass Intel TXT trusted boot [14].

The attacker could also create a bridge between two NICs in the system, e.g. to leak some secrets from a confidential, otherwise isolated network, to the public network.

For all those actions the attacker's shellcode might need to delay the CPU shutdown process.

SMX and INIT blocking

When the VMM is started via SENTER instruction and so the system runs in the SMX mode, then the delivery of INIT causes immediate platform shutdown if outside of VMX (and if in VMX mode, then the shutdown happens immediately after leaving VMX mode). This means that the SIPI attack doesn't work against TXT-loaded systems (unless the attacker is interest in DoS attacks only).

Summary

The SIPI attack vector is a direct result of allowing devices to generate SIPI interrupt and having the Local APIC deliver this SIPI to one of the CPUs in the system. We cannot really find any good reason that would explain why a device might need to generate such a *disturbing* interrupt as SIPI. In fact, even the official Intel documentation explicitly excludes SIPI from available interrupt delivery methods.

We believe the fact that SIPI could nevertheless be generated from devices and delivered to a CPU is an implementation-level security bug.

The attack seems to be mitigated in practice by the difficulty to place meaningful shellcode in the physical memory below 1MB. However, in some circumstances the exploitation might still be possible, e.g. if this memory is not specially protected by the VMM (and none of the Intel spec suggests it should be protected in any way).

The attack seems to be fully prevented when the VMM is launched via a TXT-based trusted boot.

---

15 When the processor is in VMX guest mode, delivery of INIT causes a normal VMEXIT, of course.

16 We have tested this by placing a shellcode that plays a tune, and then we delivered the SIPI. After some time, when we rebooted the system, we could hear our tune being played, just before the platform reboot :)

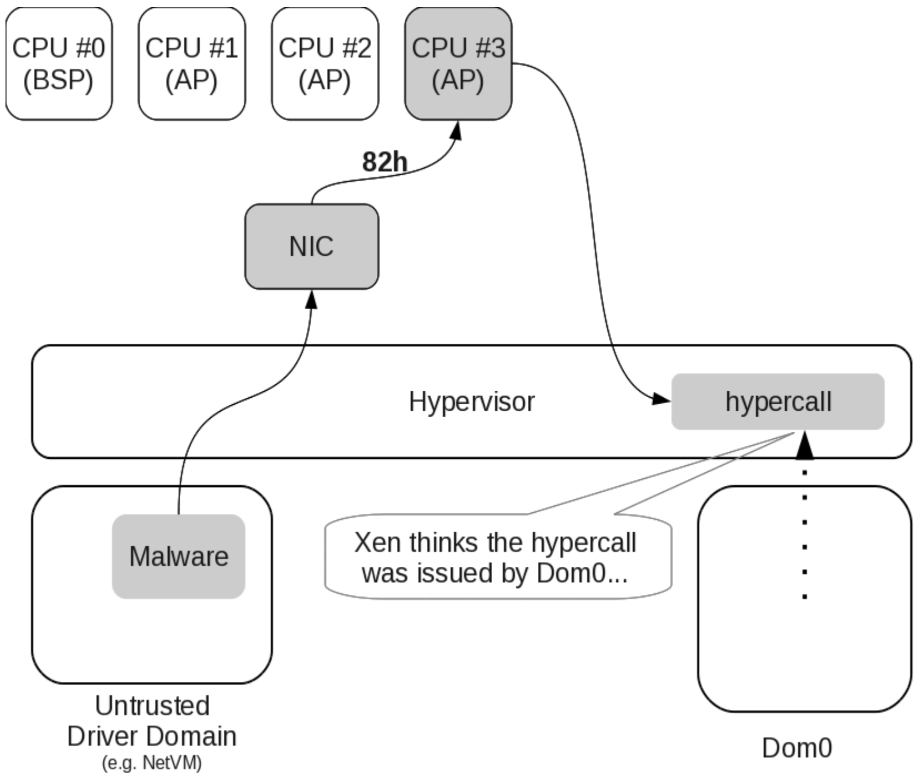
17 Theoretically the system software might not execute VMXOFF before doing a platform shutdown, but this seems unlikely and would rather be considered a software bug.

The Attack #2: The syscall injection attack

The idea of the attack

In this variant of an MSI attack we inject a well-known syscall or hypercall interrupt vector, such as 0x80 which is used as a legacy syscall on Linux, or 0x82 which is used as a legacy hypercall on Xen. Let's further focus on 0x82 injection, as it relates directly to Xen.

When the CPU gets an interrupt with such a vector, it thinks it was a hypercall issued by whatever domain was active at the time when the interrupt was delivered. Thus, if we somehow managed to deliver the interrupt at the time when Dom0 was active on the CPU, then Xen will think that Dom0 issued the hypercall and thus the permission context used to execute the hypercall will be that of Dom0 (so, full permissions).


CPU #0 (BSP)
CPU #1 (AP)
CPU #2 (AP)
CPU #3 (AP)
82h
NIC
Hypervisor
hypercall
Malware
Xen thinks the hypercall was issued by Dom0...
Untrusted Driver Domain (e.g. NetVM)
Dom0

One can expect that there are hypercalls that, when called by Dom0 with carefully chosen arguments, can provide some advantages to the attacker. In case of Xen an example of such a hypercall could be the `do_domctl` hypercall, and specifically its `XEN_DOMCTL_set_target` sub-command, that allows to grant control rights over one domain to another. Using this hypercall, the attacker can ask Xen to grant the attacker's untrusted domain (the driver domain) control over the Dom0 domain, essentially making the attacker's domain super-privileged and all-mighty.

Register values

The key challenge for an attacker in making this attack successful in practice is how to ensure proper values of some registers at the moment when the interrupt arrives?

While definitely not an easy task, we think it might be possible to find code paths in some backend processes running in Dom0 (or similar trusted entity in case of other VMMs or OSes) where the registers will be conveniently set. In the case of Xen VMM, it could perhaps be some branch in the Xenstore Daemon which is responsible for processing the Xenstore's incoming data. Timing is, of course, crucial in such an attack and it makes the whole process so difficult in practice. Especially, that, as explained below, the attacker might only have a single chance to test the attack...

The EOI register resetting

One additional (but accidental) problem that makes the exploitation even more difficult for the attacker is the need to reset the EOI register after the interrupt injection.

Normally, every *hardware* interrupt handler (also called Interrupt Service Routine) is responsible for resetting the so called *End Of Interrupt* register (EOI) in the Local APIC, which is done by performing a write-to-memory by the CPU to an address where LAPIC EOI register is mapped at.

However, the interrupt handlers for software-generated interrupts, such as 0x80 or 0x82 which are used for syscall or hypercall transitions, and are *not* expected to be used for servicing any hardware-generated interrupts originating from devices, do not reset EOI register.

The above presents a problem for the attacker, because after injecting e.g. the 0x82 interrupt, the Local APIC will be expecting system software to clear EOI, and until that happens all subsequent 0x82 interrupts (as well as all other, lower priority ones) will be blocked. In practice this means the system will become unusable.

In order to unlock the system, the attacker would have to succeed with the attack (i.e. executing the “hypercall of death”) with the first attempt. If the attacker succeeded, it would allow them to now execute some code that would clear EOI, and so to recover the system from a temporal lock down.

But, if the attacker doesn't succeed with the first attempt, the system will remain in the locked state (at least this one processor) and further exploitation attempts will not be possible anymore (at least using the same processor), as this very interrupt the attacker wishes to use for the attack will be blocked until EOI is cleared.

Unfortunately for the attacker, the delivery of other hardware interrupts in the meantime, like e.g. a clock interrupt, will not unblock the delivery of 0x82. This is because the delivery of those other interrupts would either be disabled, if their vectors were numerically smaller than 0x82, or would not affect the 0x82 interrupt mask, in case their vectors were numerically larger than 0x82, as the LAPIC logic is to always unblock interrupts starting from he highest priorities whenever a CPU executes a write-to-EOI.

Summary

In case of this attack, the problem is related to the legacy of the x86 architecture. While most (all?) other architectures have special instructions for syscall generation, on early x86 processors it had become a common practice to use interrupts for various system calls, such as int 0x16 for BIOS services, int 0x21 for DOS services, int 0x2e for Windows NT syscalls, int 0x80 for Linux system calls, etc. Even though current x86 processors do have dedicated syscall instructions, most OSes still expose the legacy, interrupt-based services, even including such new system software like Xen!

Such reuse of interrupts for syscall and signaling strikes as a bad design mistake and the described attack demonstrates why it is so. Of course in the past, long before technology such as VT-d was even considered for use on x86 architecture, and when all drivers and devices must had been trusted, it wasn't so much a security problem and instead “just” a problem of aesthetics. Unfortunately, as it often happens, inelegant designs finally often lead to very practical errors and problems later. And this is exactly what has happened here.

Note that besides int 0x80 and 0x82, Xen uses other interrupts internally as well. They are used for inter-processor communication, and are triggered via inter-processor interrupt mechanism (IPI). For example, when one CPU wants to execute code on another CPU, it sends IPI with vector CALL_FUNCTION_VECTOR (0xfb). It is unclear whether an attacker can compromise Xen system by manually sending MSI message with one of these internally used vectors; however again, it is very worrisome that a device can trigger execution of interrupt handlers meant to be used only internally by the processors.

This attack should never be possible on a well-designed architecture. There is no reason why a device might be allowed to signal a syscall or hypercall to the processor. The attack results from the messy design decisions made in the early days of x86 architecture.

Further in this paper we present how this attack could be *reliably* used in *practice* to escape from a driver domain on a Xen system.

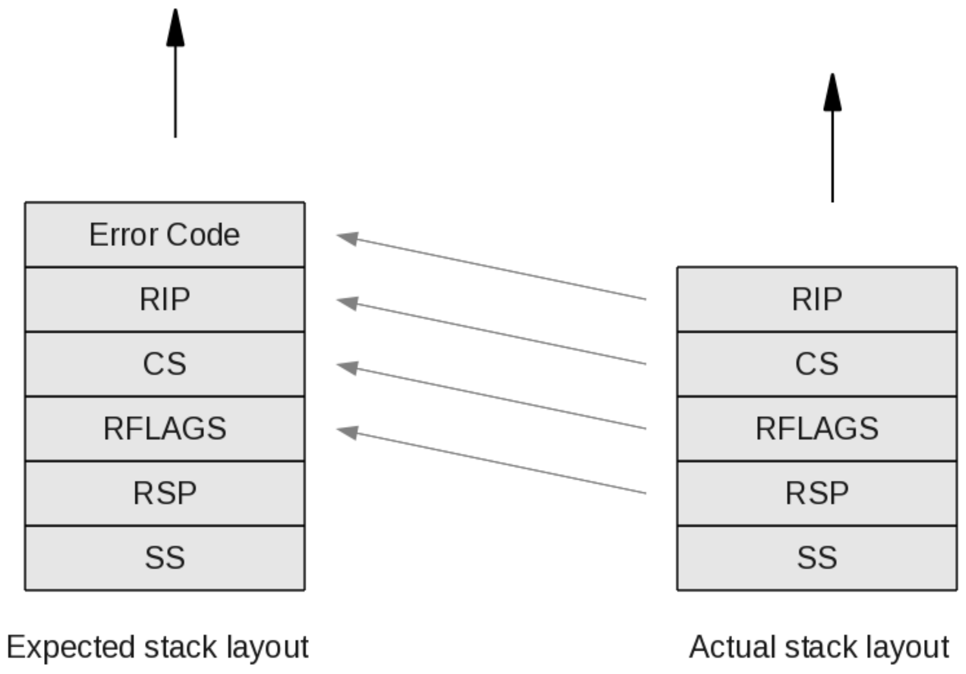
The Attack #3: The #AC injection attack

The #AC injection attack idea

The #AC injection attack exploits a similar problem in x86 architecture as the syscall injection attack described above. However, in this case we try to confuse the CPU about the stack layout that the exception handler expects to see. On the figure below we present the layout of the stack that is expected by an exception handler for all exceptions that generate error code, such as #AC, as well as the stack layout that is actually build by the processor when a hardware interrupt is delivered, such as MSI with vector = 0x11, which incidentally corresponds to #AC exception.

The #AC exception is in fact the only one exception that meets the following two requirements:

- Has a vector number greater than 15, so that we could deliver it via MSI[18]
- Is interpreted as exception that stores an error code (other exceptions, with vectors greater than 15, do not store error code on the stack, and so there will be no difference in expected and actual stack layouts seen by the handler).



If we now deliver an MSI with vector = 0x11 (#AC) from some device, it will trigger #AC handler execution on the target CPU. Because the handler expects the stack layout *with* error code placed at the top of the stack, thus it will misinterpret all other values passed on the stack. Thus, the CS placed by the CPU when the interrupt arrived will now be interpreted as RIP, RFLAGS as CS, and so on (see the figure above).

When the exception handler finishes, it executes IRET instruction that pops the previously saved register values from the stack and jumps back to CS:RIP. This means the handler will return, in fact, to RFLAGS:CS address instead!

Because we can (mostly) control RFLAGS in the guest, we can setup RFLAGS so that it looks like a valid privileged CS segment selector. In case of Xen, there is a slight complication: the default selectors have e0XXh format, and we cannot set RFLAGS (from within VM) to this value, because it means setting one of the privileged *IOPL* bits. Therefore, we need to create valid LDT entries (by legally issuing MMUEXT_SET_LDT hypercall), which would make it now possible to use low numbered selector values, that we could now “set” via RFLAGS.

We don't control the CS that was stored on the stack (and which is now interpreted as RIP by IRET), but we know that CS is a 16-but register, so it translates to a small number, when interpreted as RIP. We can easily

18 MSI cannot be used to deliver interrupts with vectors 0-15.

use mmap() to allocate some memory in our guest so that we can place arbitrary instruction at the virtual address pointed by the “CS” pointer. Because we would set RFLAGS to read as ring0 CS selector, now our shellcode in our guest would be executed with ring0 privileges, which, in case of para-virtualized guests under Xen means hypervisor privileges.

This attack will likely not work against fully virtualized guests (that use VMX guest mode), though. This is because of the very limited ability to control the address space and selectors of in the root mode.

Practical exploitation attempts on Xen

We have tried to exploit this attack on a Xen system. The Xen's #AC handler is mostly implemented by the `do_trap()` function, which calls the `search_exception_table()` function with a task to check whether the RIP that caused the exception is "special", namely is present in a special predefined table (called `exception_table`), and if so, the handler should not panic, but instead resume execution at the "error" branch of the code that triggered the exception. The same mechanism is used to implement the `copy_from_user()` primitive, that we discuss later in The practical MSI attack on Xen.

Thus, if we could force #AC handler to think that the exception was thrown from one of these special addresses, we would avoid the undesired call to `panic()`. Unfortunately, as it has been just discussed above, the #AC handler interprets the value of CS stored in the exception frame as the RIP. CS is 16bit register, which means the resulting “RIP” is a very small number. This is a problem, because all the addresses in the `exception_table` are in the hypervisor range, near the top of the address space. Thus, `search_exception_table()` will inevitably fail.

Note this is a pure lucky coincidence (for Xen) – the integrity of crucial data structures (exception frame) has been violated, it's just that the attacker cannot avoid a call to the `panic()` function.

Anyway, here is a copy of the Xen oops we were able to achieve with this attack:

```
(XEN) ----[ Xen-4.0.1  x86_64  debug=n  Tainted:    C ]----
(XEN) CPU:    0
(XEN) RIP:    0206:[<000000000000e008>] ???
(XEN) RFLAGS: ffff82c480367f28   CONTEXT: hypervisor
```

Summary

Similarly as with the previous attack, the problem we exploit here is related to the legacy of the x86 architecture, specifically the messy design of the IDT, where exceptions vectors are intermixed with hardware interrupts. Interestingly one can even see some attempts from Intel to somehow mitigate such attacks, which are expressed by requirement that MSIs cannot deliver interrupts with vectors 0-15. Unfortunately Intel must have forgot about the next 16 vectors (16-31), that are also reserved for exceptions or just marked “Intel reserved”, and specifically about #AC which is vector #17...

This attack, again, should never be possible on a well-designed architecture. There is no reason why a device might be allowed to deliver an exception, especially #AC, to the processor. The attack results from messy design decisions made in the early days of x86 architecture.

The successful exploitation in practice might be mitigated by the actual implementation of the exception handler, e.g. by the exception handler halting the system, instead of trying to resume execution, like it is in case of Xen. The attack seems to also be limited to PV guests only.

The practical MSI attack on Xen

Stop trying to hit me and hit me!

-- Morpheus

In the previous chapters we have described three different MSI-based attacks against VT-d, where each of the attacks was *theoretically* capable of executing custom code in the hypervisor context. In practice, however, and specifically when considered in the context of a Xen-based system, each of the attacks seemed somehow mitigated by various, often accidental circumstances. This might have created a wrong impression on the reader, that in practice such exotic attacks would never work, and so are not to be considered a problem in practice. We have thus decided to spend a bit more time on one of those *theoretical* attacks with the aim to turn it into a *practical* one...

The Hypercall Injection MSI Attack recap

Specifically, we have focused on the MSI-based hypercall injection attack. The attack seems very hard to exploit in practice, because, it seems, the attacker would need to trigger the MSI in the very moment when the CPU (the one which would serve the MSI) would be executing such a piece of code, so that some select CPU registers have specific values, that when interpreted as hypercall arguments would 1) make sense, and 2) bring some advantage to the attacker. Additionally, the attacker was constrained by the fact that hardware-generated hypercall leaves the Local APIC with uncleared EOI flag, effectivelly blocking any further attempts of hypercall interrupt delivery from devices for this specific LAPIC. In other words, the attacker would only be able to try to trigger the hypercall once for each processor in the system. The above description certainly suggested that the attack is a mere theoretical problem...

The Trick!

It has the same basic rules. Rules like gravity. What you must learn is that these rules are no different than the rules of a computer system. Some of them can be bent. Others can be broken. Understand?

-- Morpheus

However, after we have analyzed the actual hypercall handler implementation in Xen, we have figured out a clever exploitation approach that allowed to get around the above mentioned problems. Below is a fragment of the `copy_from_user()` function used by Xen hypercall handler for accessing the hypercall arguments[19]:

```
; rax – len of data to copy from usermode
; rsi – source buffer
; rdi – destination bufer
mov         %rax,%rcx
shr         $0x3,%rcx               ; rcx = len/8
and         $0x7,%rax
rep movsq   %ds:(%rsi),%es:(%rdi)   ; slow operation
mov         %rax,%rcx               ; rcx = len % 8
rep movsb   %ds:(%rsi),%es:(%rdi)
```

The *rep movsq* instruction is a copy operation that executes *rcx* times. Let's assume the source operand points to a slow memory, such as some MMIO of a device. A typical speed of such a slow memory could be of the order of single MB/s, which is very slow, comparing to even unbuffered reads from DRAM, which have speeds of the order of single GB/s.

Now, let's imagine that the attacker issued a hypercall from a driver domain (this is a legal action, and every domain can issue a hypercall to obtain some service from the hyperisor). Additionally the input arguments for the hypercall (buffer) was chosen to point to a virtual address in the attacker's domain that is mapped to

19 The actual implementation is written in GCC inline assembly and the instruction between the two *rep mov* instructions is written as *mov %3, %0*, which means the actual registers used in the generated code are left up to the compiler (in fact only %3, as %0 is specified to be always rcx). We were just very lucky that the default gcc in Fedora 13 used *eax* register in place of %3, making our attack possible – see below. On Fedora 14, however, we observed the compiler uses *r8* instead of *rax* for *%3*, unless the `–finline-functions` flag is explicitly used.

MMIO memory of the device that has been assigned to the domain (we assume it is NIC, specifically Intel integrated e1000e ethernet controller).

Because the MMIO memory is slow, and because we can choose a hypercall that expects some larger buffer[20], we have determined that some 80-90% of time for servicing such a hypercall will be spend in the *rep movsq* instruction mentioned above.

That means that if the attacker, in the meantime, issues an MSI, the chances of it being delivered at the time when the processor is executing the *rep movsq* instruction are very high. Even if we cannot precisely control the timing of MSI generation (which indeed is the case, when we use ICMP PING for MSI generation – see Generating MSI without access to device config space), still we can just keep issuing the hypercall in a loop, and the chances of the MSI hitting when (one of) the hypercall handler will be at *rep movsq* are very high.

Now, let's assume the MSI we're generating is also for vector 0x82 – this means that the original hypercall handler will be interrupted at the *rep movsq* instruction, and the processor will execute another instance of the hypercall handler to service the fake (MSI-generated) hypercall. Of course, the processor will save and then restore all the CPU registers, so this should have no effect on the previous instance of the hypercall...

Except, however, that the hypercall returns status in the *rax* register, which means this very register will get modified by the execution of this (unexpected) additional hypercall. As a result, when the original hypercall handler execution will be resumed, the value of *rax* will be different[21]. Specifically, it's very likely that *rax* will contain the value of *-ENOSYS*[22], which, when interpreted as unsigned integer is a very-large-number.

The changed *rax* value will not affect the *rep movsq* instruction that was interrupted by the fake, MSI-triggered hypercall. But it will affect the next instruction*,* which copies *rax to rcx*, and then the next *rep movsb* instruction that is supposed to copy the remaining bytes. The *rep movsb* will now try to copy -ENOSYS bytes, causing a big overflow past the *rdi* address. Because, however, we're not interested in crashing Xen, but instead in some code execution, we will have to stop the copy operation much sooner before the huge *rcx* gets zeroed.

The logical way to stop the copy operation seems to be to place an unmapped page, directly after our input buffer. This, however, is not enough – when the Xen's `copy_from_user()` function touches the unmapped usermode page, the #PF exception is raised, which is handled by Xen #PF handler. The #PF handler detects that the #PF was raised inside `copy_from_user()` and transfers control to a special "error branch" inside `copy_from_user()` via a mechanism called fixup tables[23], which in turn tries to complete the copy operation by writing zeros to the destination buffer (as many of them, as there are remaing bytes to copy).

This zero-writing behavior is, of course, undesirable as it can lead only to a system crash (this time the #PF handler will not return to a .fixup section). It seems like the only option for the attacker, is to overwrite the IDT table first, in order to make the #PF entry in IDT to point to the attacker's shellcode. Then, when the overflowing continues, the #PF will sooner or later be raised, and the attacker's shellcode will get executed instead of the original #PF handler. For this approach to work, the IDT must be located below the address pointed by the *rdi* register, and not too far away from it...

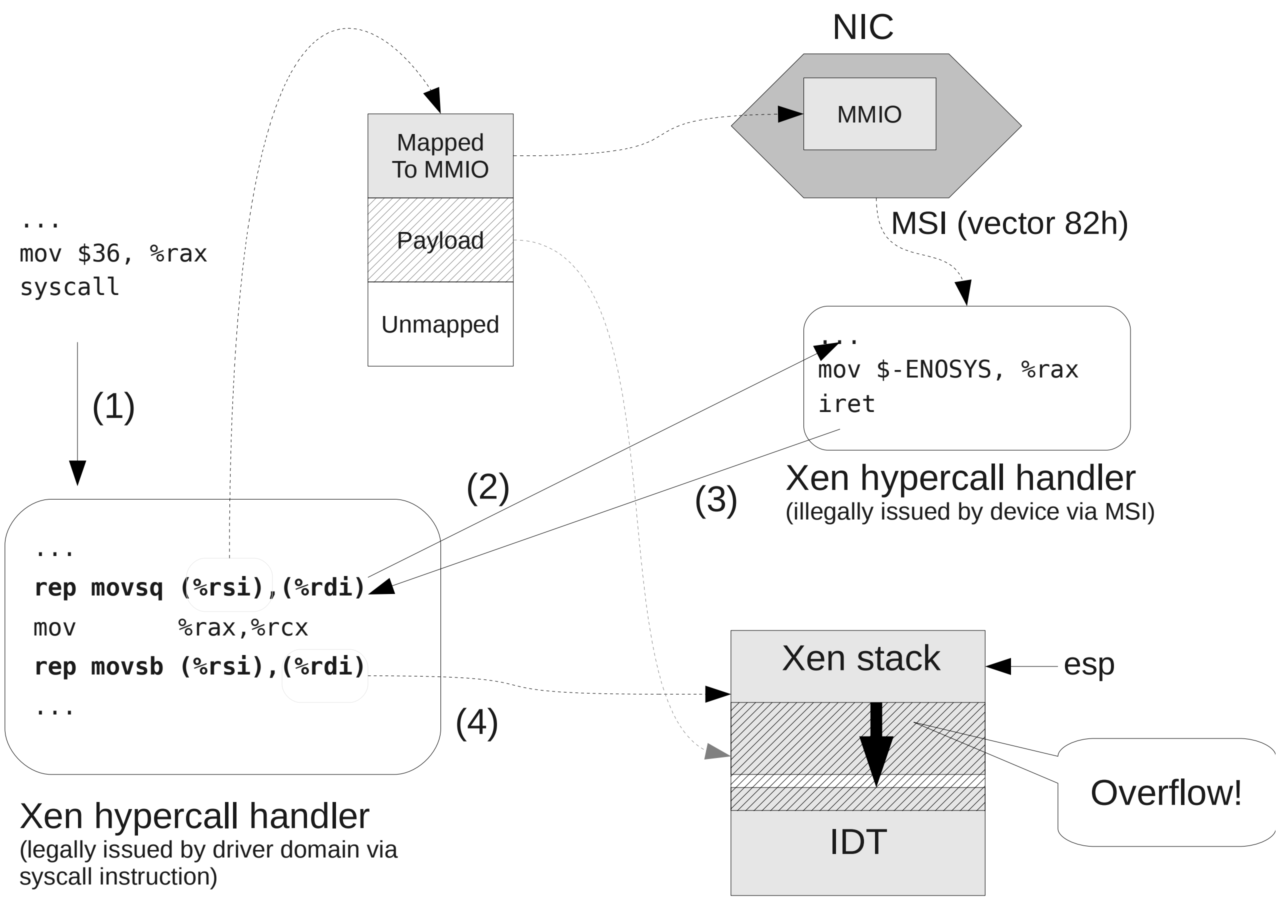
The summary of the above scenario is depicted on the figure below.

20 The maximum input buffer we found, that was expected by an unprivileged Xen hypercall was just 144 bytes – it was the input argument for the do_domctl hypercall – not really that large, but given how slow MMIO memory is, it turned out to be just enough for the attack to succeed.

21 In fact, if the nested hypercall used hypercall_create_continution() function, then its side effect would include changing other register values, too. We, however, did not find this behavior to be of any help with exploitation due to the specific assembly code used between the two rep mov instructions.

22 It's very likely that the fake, MSI-triggered hypercall will return -ENOSYS, or some other -EXXX error code, because the input arguments are effectively random.

23 This "error branch" is marked by the .fixup section in the function's assembly code.



Mastering the overflow

Using the trick described above, we are now able to *very reliably* trigger an overflow inside the Xen hypervisor, starting from the address held in the *rdi* register. The *rdi* register inside the `copy_from_user()` function points to the local hypervisor stack. Quite accidentally, it turned out that just after the Xen stack, the IDT table is *often* located, which, as described above is what we have been hoping for...

Unfortunately the exact location of the IDT table, depends on whether we're on the BSP processor, or on one of the AP processors.

In case we're on the BSP processor, the IDT table is allocated as part of the .bss section, and is always after BSP stack, more or less 48 pages below. This means that the attacker must trash ca. 48 pages of memory before reaching the IDT.

On AP processors, the IDT seems to be very often located directly after the Xen stack. However this might not always be the case, as both the stack, and the IDT tables, are allocated from the heap inside the `do_boot_cpu()` function:

```
stack_base[cpu] = alloc_xenheap_pages(STACK_ORDER, 0);
idt_tables[cpu] = xmalloc_array(idt_entry_t, IDT_ENTRIES);
```

Thus, if we're unlucky, and if the heap is somehow fragmented[24], the IDT might even be located above the stack, and the attacker won't be able to overwrite it.

24 The primary factor that might affect heap fragmentation, we believe, are the ACPI tables exposed by BIOS. We have observed that in case of Q45 and Q67 based systems, the data structures layout on the heap was such that AP processors could be exploited to yield code execution to an attacker, without making the system unstable. On the other hand, it was not so on Q57. The difference was most likely caused by different ACPI tables exposed on the latter system.

The Exploit

The proof-of-concept exploit we have coded has two modes of operation: the BSP mode, and the AP mode.

When used in the BSP mode, the exploit assumes the attacker's domain executes on CPU #0 (the BSP processor) and so delivers the MSI to LAPIC for this processor #0. Additionally the exploit uses the 48 pages of padding in the payload, because it has been determined that so long is the distance between the *rdi* (when used in the copy_from_user() called at the beginning of the hypercall handler) and the IDT table. The shellcode executes as #PF handler.

When used in the AP mode, the exploit assumes we're on the CPU #1 (AP processor)[25] and so delivers the MSI accordingly. The exploit uses just 1 page long payload padding, because it has been determined that on our test Q45 system, the IDT is always so close to the *rdi* address (this might not be the case on other systems, because of different heap allocation layout, that could be most likely influenced by the ACPI tables passed from BIOS). Because the overwrite is relatively small (ca. 4k bytes), the shellcode is able to recover the overwritten Xen stack, and so is able to resume normal execution of the system after the attack, but of course with some additional instructions executed for the advantage of the attacker (that might e.g. install a rootkit in the hypervisor...).

Improving the Exploit Reliability

One challenge for the attacker is to ensure that the MSI will be delivered to the same CPU on which the attacker's domain is also scheduled to run. This is necessary, because for the attack to work the original hypercall handler must be interrupted by the execution of another hypercall handler (triggered by MSI). Hence the two modes of the exploit discussed above.

Unfortunately it seems like it's not possible for the attacker to figure out on which physical CPU his or her domain is executing, and so there is no easy way for the attacker to choose the proper mode of the exploit...

To get around this problem, the attacker might use the following approach:

- Try to iteratively deliver MSIs to all the CPUs in the system (and at the same time to keep triggering the hypercall from the driver domain)
- Combine the “1-page” and “48-page” payloads into one universal payload.

A slight problem with this approach is that there is no guarantee that we could iteratively generate all the MSIs without being moved to another physical processor in the meantime. That would be not good, as the hypercall generation (via syscall instruction) would now be happening on a different processor then when we started the iterative delivery of MSIs. Nevertheless the vcpu migration to a different CPU should be a rare case, so it seems more of a theoretical problem, rather than a practical one.

All in all, it seems that on a system with correct heap layout (such as our Q45) we could achieve a very high success rate using the above scenario.

Summary

The presented exploit is very Xen-specific, and even distribution-specific (as the compiler configuration used to build Xen hypervisor can impact how the key assembly code is generated – see above). It shows, however, how seemingly innocent “theoretical only” problem can be turned into a reasonably reliable *practical* exploit, if only the adversary can invest substantial amount of time and energy into the process of exploit development.

Needless to say, we're quite proud that we were able to demonstrate a complete exploitation process, from generating an MSI via scatter-gather tricks, to finally achieving code execution in the hypervisor context. This is probably the most complex exploit we have ever presented.

25 We have assumed the Q45 platform which has only 2 CPUs.

Interrupt Remapping

Neo... nobody has ever done this before.

That's why it's going to work.

-- Trinity and Neo

The MSI Attacks Prevention via Interrupt Remapping

As part of the VT-d spec, Intel describes a mechanism, called Interrupt Remapping, that seems capable of protecting the system from all the previously mentioned MSI-based attacks.

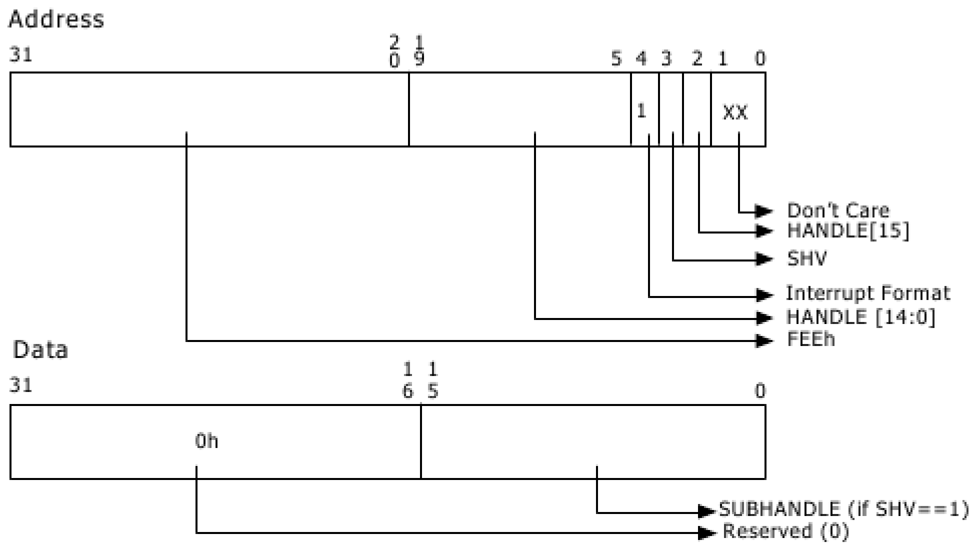
When Interrupt Remapping is enabled in the chipset, all incoming interrupts can be blocked and/or translated, depending on their originating device (just like in the case of DMA remapping). Specifically, the system-software is expected to fill in interrupt remapping tables, for each device domain, which would explicitly list all allowed interrupt vectors and formats available for those devices.

Even though the interrupt remapping seems vulnerable to the BDF spoofing attack[26] (just like DMA remapping is), still we believe it has a potential to prevent all the above-mentioned MSI attacks, because we anticipate that none of the interrupt vector used in our attacks should be allowed to *any* devices in the system. In that case, BDF spoofing would not bring any advantage to the attacker, as there should be no device at all that the attacker might want to impersonate, which could deliver those dangerous interrupts, such as SIPI, syscall interrupts, or the #AC exception.

On the client side, only the very latest Sandy Bridge processors (released in Q1 2011) support Interrupt Remapping. On the server side, however, some advanced Xeon systems have had this support for at least 2 years now.

The new MSI interrupt formats

When interrupt remapping is enabled, the devices are expected to generate MSIs in a new format, the so called *Remappable Format*, which is illustrated on the figure below:



The key difference between the new (remappable) MSI format vs. the old format (called *compatibility format* in the spec), is that the new format doesn't directly specify any properties of the generated MSI. Instead only a so called *handle* is specified, that is interpreted as an index in the *interrupt remapping table*, which is managed by the VMM and which is device- (domain-) specific. This way the device should have no way to generate MSIs with e.g. arbitrary vector.

26 BDF spoofing is a type of a *hardware* attack against VT-d, where a malicious device generates PCIe packets with spoofed BDF address. We have also thoroughly researched such attacks, but hardware attacks are out of scope of this paper, which focuses on software attacks only.

Interrupts in the compatibility format

For compatibility reasons, devices might still be allowed to generate interrupts in the compatibility format. According to the spec, this is allowed if both of the following conditions hold:

1. Extended Interrupt Mode (also called *x2APIC*) is not enabled,
2. The Compatibility Format Interrupt (CFI) field in the Global Command register is set,

In this case, interrupt remapping **does not** offer resistance from malicious interrupts - a device can generate arbitrary MSIs just as it was described in the previous chapters.

It seems like the system software should always switch to the x2APIC mode when running on an Interrupt Remapping capable system, which automatically guarantees that compatibility MSIs will be blocked.

Interrupt Remapping implementation in Xen

We have studied how the Xen hypervisor configures interrupt remapping to protect the system from untrusted driver domains. For each PCI device that is to be configured, Xen does the following:

1. Allocates an interrupt vector in the global IDT for this device,
2. Adds an entry to the Interrupt Remapping Table for this device (domain), so that it translates to the vector generated in the previous point,
3. Writes to the device's configuration space registers for MSI (MADDL, MADDH, MDAT), so that the device generates MSIs in a remapble format, with a proper index (handler).

The above procedure seems correct, because it only assigns the actual vectors that the devices need, and nothing more (such as vectors for SIPI, 0x82, or #AC). Thus, even though it seems like one device could use BDF spoofing to impersonate another device, it still would not allow to trigger any of the attacks presented in the previous chapters.

Bypassing Interrupt Remapping on Xen

Let's assume now that the attacker, as a result of controlling one of the driver domains in the system, is capable of modifying the system's Master Boot Record. This requirement might be trivial, *or not*, depending on the type of the driver domain the attacker controls:

- If the attacker controls a driver domain that has been assigned the *main disk controller*, e.g. the *untrusted storage domain,* as it is described in Qubes OS architecture[27] specification [11], then the attacker can trivially modify the system's MBR.
- If the attacker controls a driver domain that has some other device assigned, e.g. a network device, or a GPU, it might be non-trivial for the attacker to figure out a practical way to compromise the system's MBR. We can imagine some potential scenarios, that however haven't been verified in practice by us yet, such as e.g. reflashing the NIC's firmware and forcing it to do DMA early on the system boot, overwriting in-memory MBR image. Possibility to execute such an attack would likely be very device-specific.

Now, assuming the attacker managed to modify the system's MBR, it should still be impossible for the attacker to compromise the system, if the system is well designed. This is because a well designed system should be using some form of trusted boot technology, specifically to prevent such attacks[28] [2].

But, thanks to the interrupt-based attacks presented in this paper earlier, the attacker might still have a chance to attack the system. Specifically we have successfully tested the following attack against the Xen hypervisor:

27 Note that in Qubes architecture the storage domain is assumed to be untrusted, i.e. the attacker that controls it, still should not be able to compromise any other domain in the system. This might sound counter-intuitive, but nevertheless could be achieved with the help of cryptography and trusted boot technology.

28 Now it should be clear why VT-d and TXT could be thought as two elementary and most important building blocks for any modern system. VT-d allows to sandbox drivers, but is not complete without a DMA-proof trusted boot technology.

1. The attacker modifies the ACPI DMAR tables, so that Xen thinks the platform doesn't support Interrupt Remapping (this can be done if the attacker controls the MBR)
2. Now, the system boots as usual, using TXT-based trusted boot implemented via tboot.
3. Xen initializes VT-d, but figures out there is no Interrupt Remapping support, so proceeds without it.
4. Now, the driver domain that the attacker controls can perform an MSI attack as described earlier in this paper.

Details on cheating Xen via ACPI DMAR manipulation

The ACPI DMAR table [1] contains a field called *Flags* with two bits of the following meaning (quoting literary the description from the VT-d spec):

- Bit 0: INTR_REMAP - If Clear, the platform does not support interrupt remapping. If Set, the platform supports interrupt remapping.
- Bit 1: X2APIC_OPT_OUT - For firmware compatibility reasons, platform firmware may Set this field to request system software to opt out of enabling Extended xAPIC (X2APIC) mode. This field is valid only when the INTR_REMAP field (bit 0) is Set. Since firmware is permitted to hand off platform to system software in legacy xAPIC mode, system software is required to check this field as Clear as part of detecting X2APIC mode support in the platform.

So, it seems like we could just modify those bits in the in-memory copy of the DMAR table, before the TXT-launch, and force Xen not to enable Interrupt Remapping[29]. Unfortunately, it turned out that Xen doesn't interpret those fields at all.

So, we have analyzed how does Xen determines IR support, and we came up with a different cheating method, that works by removing the IOAPIC device from the *Device Scope[]* lists in DRHD tables for each remapping unit (See [1] again). Apparently this confuses Xen so much, that it doesn't enable Interrupt Remapping, as shown in the `xm dmesg` logs:

```
(XEN) [VT-D]intremap.c:147: There is not a DRHD for IOAPIC 0x0 (id: 0x0)!
(XEN) Not enable x2APIC due to iommu_supports_eim fails!
(...)
(XEN) Intel VT-d Interrupt Remapping supported.
(XEN) [VT-D]iommu.c:1883: ioapic_to_iommu: ioapic 0x0 (id: 0x0) is NULL! Will not try to
enable Interrupt Remapping.
```

This has been tested on a Core i5 2500K processor (Sandy Bridge) and on Intel DQ67 board. The boot arguments to Xen were explicitly requiring IOMMU to be always enabled, or the boot to fail:

```
iommu=pv,verbose,required vtd=1
```

Xen has been loaded via a TXT-based boot via Intel's *tboot*. SENTER succeeded without any problem:

```
Intel(r) TXT Configuration Registers:
        STS: 0x00018091
            senter_done: TRUE
```

The SINIT module for Sandy Bridge processors

It should be noted that Intel hasn't yet officially published SINIT modules for Sandy Bridge processors. SINIT modules are a required element for TXT launch, and they are platform-specific. Without appropriate SINIT modules one cannot use TXT on a given platform. This opens it up for a number of attacks – in case above the attacker wouldn't even need to cheat the ACPI table, and instead could just subvert the xen.gz image.

In order to make this paper complete we wanted to test if our IR-disabling ACPI attack would work indeed, and so we have asked Intel to provide us with a required SINIT module for Sandy Bridge processors. Intel told us that they still don't have a “production” SINIT for Sandy Bridge processors, and instead provided us with a “pre-production” SINIT module, which is what we used for the above experiment.

Because the pre-production SINIT randomizes PCR registers after SENTER, we were not able to test if our

29 It turned out that Xen doesn't explicitly clear the CFI bit (see Interrupts in the compatibility format), so disabling x2APIC mode and, clearing CFI before SENTER should be all that is required to force Xen to not enable Interrupt Remapping.

DMAR modifications do not affect the PCR registers after TXT launch, however we see no reason why they could affect them. Indeed, in one of our previous attacks against Intel TXT [14], we have also modified ACPI table before SENTER, and the PCR registers remained intact. We have also looked at the tboot sources, and it doesn't seem like it extended PCRs with any hash of the DMAR table (which would be difficult, because of the over-complex structure of ACPI tables).

On a side note, we think it is very strange that Intel still doesn't offer a production-ready SINIT modules for their flagship security technology, several months after the release of Sandy Bridge processors...

Summary

This attack that tricks Xen to not enable Interrupt Remapping support is required only on recent Intel hardware that has IR support. On all other platforms that support VT-d, but do not support Interrupt Remapping, which seems to include *all the client platforms except the latest Sandy Bridge processors*, one can just attack VT-d using the MSI attacks described earlier in this paper.

The attack described in this chapter takes advantage of Xen careless programming, i.e. ignoring the fact that Interrupt Remapping is not available and still allowing to boot the system as if nothing happened (despite the **iommu=reqiured** boot option). However, one should not blame Xen for this mistake, as it has never been made clear that Interrupt Remapping is such a security-critical element of VT-d, potentially allowing for code execution in the hypervisor.

Working with Vendors

A few weeks ago before publishing of this paper, we have provided a draft to Intel security team and a few Xen developers.

Reaction from Intel

We've believed that Intel security team should be the primary contact point for discussing this vulnerability, because the attacks presented in this paper exploit flaws in the x86 architecture, rather than specific flaws in any VMM software. Furthermore, Intel has been selling client VT-d enabled platforms since at least 2007, most of which being vulnerable to our MSI attacks, and yet none of the Intel specs or manuals made it clear that such platforms are vulnerable to potentially fatal MSI attacks.

We have thus expected that Intel would somehow officially recognize the problem we presented in the paper, issue a warning to customers, such as VMM vendors, and update their manuals and specs to indicate that IR is an absolutely required security element of any VT-d system.

Strangely, it doesn't seem Intel has done any of the above.

Two of the Intel engineers, involved with Xen.org development, have engaged, however, in preparing a patch against the attack we described in Bypassing Interrupt Remapping on Xen chapter – so not the actual MSI attacks, but rather an attack to be used on IR systems when the attacker somehow managed to compromise the MBR. The patch that they have finally proposed, and that was accepted by Xen.org [16], was unfortunately not optimal (as described below), because Intel wanted to allow TXT on non-IR platforms. This shows a rather strange logic of thinking, because on the one hand Intel seems to be ignoring the problem of the attacks presented in this paper (apparently because they now have IR-capable hardware), yet at the same time they attempt to tailor patches to allow “secure” TXT on non-IR hardware, which is hardly possible given the VT-d vulnerability to MSI attacks.

Reaction from Xen.org

We have also made a draft of this paper available to select Xen developers. We have done it because our paper specifically targets the Xen hypervisor, and also because we believed that even though the MSI attacks exploit a hardware problem, still some reasonably effective software patches are possible to come up with for Xen. We have discuss this with a few Xen developers and, as a result, an MSI-blocking patch has been created by Keir Fraser shortly afterwards [16].

The patch works by adding extra code at the beginning of 0x80 and 0x82 interrupt handlers and ensures that the interrupt has *not* been generated by a device by checking the LAPIC status registers. Additionally the patch blocks delivery of #AC exception from any device[30]. The patch seems to provide a reasonably good prevention against the attacks presented in this paper (although the attacker can still trigger a DoS attack against the whole system). Yet, there still might be other MSI-based attacks that this patch not prevents, such as those we briefly speculated about on page 13.

Another patch was to prevent a potential attack that tricks Xen into not enabling IR even if running on a hardware that supports it, as described in Bypassing Interrupt Remapping on Xen. There have been a discussion between Xen.org developers whether to always fail Xen boot if IR cannot be enabled and if `iommu=required` was passed as a Xen boot option, vs. whether to fail it only if the previous conditions hold but only on *platforms that supports IR*.

The argument for the first approach, i.e. to always abort Xen boot if `iommu=required` has been passed and if IR couldn't be enabled for *any reason*, was that when the users pass `iommu=required` their intention is to ensure IOMMU will always work and provide required security isolation. So, it should be irrelevant whether IR couldn't be enabled because somebody tampered with ACPI tables before boot, or because the platform simply doesn't support it – in any case IOMMU/VT-d would not be secure, and we should refuse to boot.

The arguments for failing the boot only on non-IR systems, that were raised by Intel engineers, were that the default TXT-based loader for Xen requires `iommu=required` option (which is reasonable) and so TXT would no longer be usable on non-IR systems (so most client systems out there today). This argument seems to ignore the fact that it has just been demonstrated that VT-d on non-IR systems is insecure.

30 The #AC could be easily blocked by setting the LAPIC priority register to 0x20, so blocking all the vectors in the range 0x0-0x1f, including #AC. The same approach couldn't, unfortunately, be used to block 0x80 and 0x82 delivery, because setting LAPIC priority so high would also block many useful vectors legitimately assigned to various devices.

Reaction from other vendors

We have *not* attempted to contact any other vendors except Intel and Xen.org. The primary reason for this was that we have naturally assumed that it should be Intel's responsibility, as the problem has been discovered in Intel's products. At the same time, we don't have resources to contact and work with every potentially affected vendor who might be using VT-d. Just to give an impression of the amount of resources such a “Samaritan” action would require, we should mention that we have exchanged around 100 emails with Xen.org developers within the last 3 weeks discussing this problem and patching approaches...

Final words

You take the blue pill, the story ends, you wake up in your bed and believe whatever you want to believe. You take the red pill, you stay in Wonderland, and I show you how deep the rabbit hole goes.

-- Morpheus

This paper demonstrates that Interrupt Remapping is a security-critical component of an IOMMU technology for x86 platforms. It shows that all the VT-d enabled systems that do not support Interrupt Remapping are prone to various security attacks, including arbitrary code execution in the hypervisor. None of the Intel specs we looked at made it clear that Interrupt Remapping is such a security-critical component of VT-d.

While the interrupt-based attacks might seem “just theoretical” and so innocent in practice, we have demonstrated that, given enough determination and skills, it is often possible to turn even the most “impossible” attack into a real and reliable exploit.

As far as we are aware, this is the first publication showing the practical security problems that could arise from lack of Interrupt Remapping.

We shall point out it's not the first time Intel is selling a half-baked security solution without informing customers about it. More than two years ago we have presented our first software attack against Intel Trusted Execution Technology [17], which exploited a problem with SMM mode being over-privileged and surviving the TXT launch sequence. Intel responded that the proper solution to our attack was to use a so called *SMM Transfer Monitor* (STM) to sandbox a potentially malicious SMM code. Yet Intel never made it clear in their specs or manuals that TXT without STM is vulnerable to software attacks as we demonstrated.

Acknowledgments

We would like to thank Keir Fraser of Xen.org, for hinting us on SIPI queuing in VMX mode, and also for a prompt creation of MSI-blocking patches for Xen.

References

[1] Intel Corp., *Intel Virtualization Technology for Direct I/O*, Feb 2011, ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf

[2] Joanna Rutkowska, *On Trusted Computing, Desktop Security, And Why This All Matters?*, 5th European Trusted Infrastructure Summer School, 2010, http://qubes-os.org/files/doc/etiss.pdf

[3] Allen Kay, Eddie Dong, *VT-d and SRIO Update,* Xen Summit 2009*,* *http://www.xen.org/files/xensummit_oracle09/VTDSRIOV.pdf*

[4] Anthony Liguori, *Re: A few KVM security questions*, KVM mailing list, 2009, http://www.mail-archive.com/kvm@vger.kernel.org/msg25807.html

[5] PCI SIG, *Single Root I/O Virtualization*, http://www.pcisig.com/specifications/iov/single_root/

[6] Xen Wiki, *Xen VT-d Howto*, http://wiki.xensource.com/xenwiki/VTdHowTo

[7] Intel blog, *Step by Step Guide on How to Enable VT-d and Perform Direct Device Assignment*, http://software.intel.com/en-us/blogs/2009/02/24/step-by-step-guide-on-how-to-enable-vt-d-and-perform-direct-device-assignment/

[8] Qubes OS, http://www.qubes-os.org

[9] XenClient, http://www.citrix.com/English/ps2/products/product.asp?contentID=2300325

[10] Karl Janmar, *FreeBSD 802.11 Remote Integer Overflow*, Black Hat Europe 2007, http://www.blackhat.com/presentations/bh-europe-07/Eriksson-Janmar/Whitepaper/bh-eu-07-eriksson-WP.pdf

[11] The H, *DHCP client allows shell command injection*, April 2011, http://www.h-online.com/security/news/item/DHCP-client-allows-shell-command-injection-1222805.html

[12] Rafal Wojtczuk, *Subverting the Xen Hypervisor*, Black Hat 2008, http://invisiblethingslab.com/resources/bh08/part1.pdf

[13] Joanna Rutkowska, Rafal Wojtczuk, *Qubes OS Architecture Specification*, 2010, http://qubes-os.org/files/doc/arch-spec-0.3.pdf

[14] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin, *Another Way to Circumvent Intel® Trusted Execution Technology*, 2009, http://invisiblethingslab.com/resources/misc09/Another%20TXT%20Attack.pdf

[15] Rafal Wojtczuk and Alexander Tereshkin, *Attacking Intel® BIOS*, Black Hat 2009, http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

[16] Ian Jackson, *Xen security advisory CVE-2011-1898 - VT-d (PCI passthrough)*, http://lists.xensource.com/archives/html/xen-devel/2011-05/msg00687.html

[17] Rafal Wojtczuk, Joanna Rutkowska, *Attacking Intel Trusted Execution Technology*, http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf, 2009