Another Way to Circumvent Intel® Trusted Execution Technology

Tricking SENTER into misconfiguring VT-d via SINIT bug exploitation

Rafal Wojtczuk
rafal@invisiblethingslab.com

Joanna Rutkowska
joanna@invisiblethingslab.com

Alexander Tereshkin
alex@invisiblethingslab.com

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

December 2009

Abstract

Earlier this year our team has presented an attack against Intel® TXT that exploited a design problem with SMM mode being over privileged on PC platforms and able to interfere with the SENTER instruction. This time we present a different attack that allows an attacker to trick the SENTER instruction into misconfiguring the VT-d engine, so that it doesn't protect the newly loaded MLE. This attack exploits implementation flaws in a so called SINIT module.

keywords: Intel TXT, Intel VT-d, SINIT, SENTER, Trusted Boot, Attack, Circumvention

1. Introduction

For the basic introduction about Intel® TXT, the reader is referenced to our previous paper on this topic [1], or alternatively, for a much more complete and in-depth introduction, to the updated book by David Grawrock [2].

The attack presented below assumes the attacker can execute his or her code before the TXT's SENTER instruction is executed, e.g. by infecting the boot loader. The attacker code, as we will show below, can then misconfigure the chipset in such a way that the SENTER instruction would be unable to properly setup VT-d protections for the newly loaded MLE (e.g. hypervisor). As a result, the attacker would be able to compromise the securely loaded hypervisor using a classic DMA attack.

The Intel® TXT technology has been designed exactly to prevent scenarios like the above. In other words, Intel® TXT secure launch process assumes that the system might be compromised before the SENTER instruction is executed, and yet the SENTER instruction is expected to securely load and start the hypervisor. The attack described in this paper demonstrates this assumption doesn't hold in practice, because of certain implementation errors.

2. Intel VT-d background information

For the attack described in this paper, it is important to understand certain internals of how Intel VT-d technology is implemented. The reader can find much more details about VT-d internals in the Intel official specification [3].

As illustrated on Figure 1, Intel VT-d logic is implemented in the Memory Controller Hub (MCH, also called the Northbridge). System software, such as the OS or the hypervisor, can configure each VT-d remapping unit so that all devices connected under the particular unit are allowed DMA access to only certain regions of the system physical memory[1].

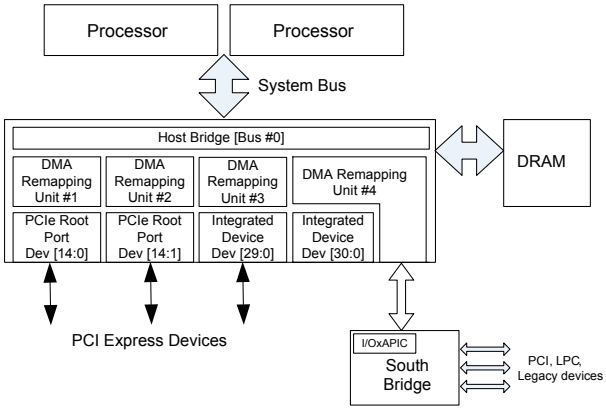

Figure 1. VT-d remapping units located in the Memory Controller Hub (MCH). Source: `intel.com`.

In particular, the hypervisor memory should never be accessible to any DMA device. Otherwise the attacker can perform a DMA attack, e.g. from a driver domain, and subvert the hypervisor, see e.g. [4].

1 Each of the remapping units can not only grant/deny accesses to certain physical memory, but also perform arbitrary translations of the DMA addresses.

It is important to stress that a typical system has more than just one DMA remapping unit. In the example in the figure above there are four independent remapping units in the system. Separate remapping units could be used, for instance, for graphics and audio cards, another for Intel Management Engine (which is used for AMT), and yet another one for all the other PCI and PCI Express devices, like network cards, SATA controllers, and all the legacy PCI devices connected via the Southbridge.

An important question we should answer, is how the system software knows how many DMA remapping units are present in the system, which devices are connected to each unit, and where their configuration registers are located?

The answer to the above questions is provided by the so called DMAR ACPI table.

3. The DMAR ACPI table

The Intel VT-d specification [3] defines the so called DMAR ACPI table, whose sole purpose is to provide detailed description about each DMA remapping unit in the system. Those tables are being constructed in RAM, and exposed to the system software by the BIOS. Here, the assumption is that the OEM, which provides the BIOS, knows exactly the hardware used by the platform (in that case the specifics of the Memory Controller Hub), as well as where its configuration registers has been mapped into system memory.

On Figure 2, a structure used to describe properties of one DMA Remapping Unit is presented. Two fields are of special importance for us: the *Register Base Address* field, that tells the system software where the memory-mapped configuration registers are located in the system physical memory address space, and also the *Device Scope[]*, which identifies devices connected to this DMA Remapping Unit. The devices are identified by specifying their BDF addresses, or ranges of BDF addresses.

Field	Byte Length	Byte Offset	Description
Type	2	0	0 - DMA Remapping Hardware Unit Definition (DRHD) structure
Length	2	2	Varies (16 + size of Device Scope Structure)
Flags	1	4	Bit 0: INCLUDE_PCI_ALL • If Set, this remapping hardware unit has under its scope all PCI compatible devices in the specified Segment, except devices reported under the scope of other remapping hardware units for the same Segment. If a DRHD structure with INCLUDE_PCI_ALL flag Set is reported for a Segment, it must be enumerated by BIOS after all other DRHD structures for the same Segment[1]. A DRHD structure with INCLUDE_PCI_ALL flag Set may use the 'Device Scope' field to enumerate I/OxAPIC and HPET devices under its scope. • If Clear, this remapping hardware unit has under its scope only devices in the specified Segment that are explicitly identified through the 'Device Scope' field. Bits 1-7: Reserved.
Reserved	1	5	Reserved (0).
Segment Number	2	6	The PCI Segment associated with this unit.
Register Base Address	8	8	Base address of remapping hardware register-set for this unit.
Device Scope []	-	16	The Device Scope structure contains zero or more Device Scope Entries that identify devices in the specified segment and under the scope of this remapping hardware unit. The Device Scope structure is described below.

Figure 2. A DMA-remapping hardware unit definition (DRHD) structure. DMAR ACPI table contains one such structure for each DMA remapping unit in the system. Source: `intel.com`

4. Problem with the DMAR ACPI table

An alert reader should have already spotted a problem related to the use of DMAR ACPI table for reporting of VT-d hardware configuration: ACPI tables are not digitally signed by the BIOS, nor they are protected in any way. In fact it is easy for the attacker to modify DMAR table by simply overwriting system memory.[2]

Indeed, our first idea for the attack was to subvert the DMAR ACPI table in such a way that the SENTER instruction (as well as the rest of the system software executed afterwards, e.g. tboot and Xen) got a wrong picture of VT-d hardware. More precisely, we wanted to cheat that there is only one DMA remapping unit in the system, and we would choose a unit which is "innocent" from our attacker's perspective.

E.g. we observed that vPro systems have a separate DMA remapping unit dedicated for the Management Engine (ME) only. Because our DMA attack would not come from the ME, but rather from some ordinary device like the network card or disk controller, it would be useful to pretend to the system that this ME-specific remapping unit is the only one remapping unit in the system and that it covers all the devices. In that case, we anticipated, the SENTER instruction and later the MLE (e.g. tboot, or Xen) would only initialize and setup this one remapping unit, thinking this would protect against DMA attacks from all the devices in the system.

2 It is indeed very easy, because we assume a scenario where the attacker executes code before SENTER, which in practice means the attacker subverted e.g. the bootloader. In that situation there is no OS that could apply any protections on the memory where BIOS stored the ACPI tables, including the DMAR table.

After all, they should have no way of knowing there are other remapping units for other devices.

Interestingly, after implementing a simple DMAR subverting exploit, it turned out that the SENTER instruction didn't execute successfully. In fact, it returned an error about the VT-d configuration reported in the DMAR table[3]...

5. The mysterious SINIT module

A curious reader would already be asking a question: so how the SENTER instruction knew that the VT-d configuration reported in DMAR was a fake one?

It turned out that this is one of the roles of the so called SINIT Authenticated Code modules, that are distributed by Intel for all TXT-capable chipsets. Currently the modules are being made available as part of the Intel's reference implementation of TXT-based trusted boot project [5], but in the future the specific modules might be bundled together with the BIOS firmware.

The internals (exact algorithms and code) of SINIT modules are not documented by Intel. All that is officially known is that the SENTER instruction loads[4] and executes the SINIT module specific to the chipset on which it's running, and that the code the SINIT module contains is tasked with enforcing appropriate platform configuration needed for secure launch process.

It is important to notice that the SINIT modules are digitally signed and that the code inside SINIT will get executed by SENTER only if the signature is intact.

We took a closer look at the SINIT module for our platform (based on Q45 chipset), suspecting that it has something to do with ability to detect fake VT-d configuration. It turned out the module contains regular x86 code, so it was easy for us to disassemble and further analyze it.

6. Inside the SINIT module

Because we knew the error codes that SENTER was returning when we tried to cheat about DMAR table, it was rather easy to locate the pieces of code that likely were responsible for detecting the VT-d misconfiguration.

In particular the following code, in the SINIT module for Q45 chipsets, turned out to be responsible for verifying device scopes and BAR registers in the DMAR table:

```
mov     edi, es:MCHBAR
mov     esi, es:DMAR
add     esi, size ACPI_TABLE_DMAR
check DRHD_0:
mov     eax, [edi+0D40h]
and     eax, 0FFFFFFFEh
mov     edx, eax
pusha
mov     eax, 0D800h ; PCI device 0:1b.0
call    pci_read_word
cmp     bx, 0FFFFh
popa
jz      short check_DRHD1
cmp     eax, [esi+DMAR_DRHD.BAR]
jnz     BARs_mismatch
...
// perform other checks in this DRHD
...
check_DRHD_1:
mov     eax, [edi+0D00h]
and     eax, 0FFFFFFFEh
mov     ecx, eax
pusha
mov     eax, 1000h ; PCI device 0:2.0
call    pci_read_word
cmp     bx, 0FFFFh
popa
jz      short check_DRHD_2
cmp     eax, [esi+DMAR_DRHD.BAR]
jnz     BARs_mismatch
...
check_DRHD_2:
mov     eax, [edi+0D80h]
...
check_DRHD_3:
mov     eax, [edi+0DC0h]
and     eax, 0FFFFFFFEh
cmp     eax, [esi+DMAR_DRHD.BAR]
jnz     short BARs_mismatch
...
```

We see here that if certain PCI devices are present (their configuration register at offset `0x0` doesn't read as `0xffff`), then it is assumed that certain DMA remapping unit is present, and the corresponding DRHD structure for this remapping unit

3 We have also tried more subtle attacks, that e.g. preserved the original number of DRHD structures, but only modified the BAR field or the Device Scope fields. In any case SENTER was throwing various errors, about BAR mismatches or invalid device scopes.

4 It seems that the SINIT code is never loaded into DRAM, but rather only to the so called AC Execution Area, which seems to be located in the CPU cache.

should be verified. In the fragments above, one can see e.g. that it is verified whether the Register Base Address field (BAR) reported in DRHD structure matches the one reported by the chipset in some (undocumented) fields of the MCHBAR region[5].

In the case of a Q45-based system we used for testing the following PCI devices were checked in order to establish if given DMA remapping unit is present and if given DRHD structure in the ACPI DMAR table should be verified:

DRHD	PCI BDF address	Device Name
0	0:1b.0	Intel Integrated Audio
1	0:02.0	Intel Integrated Graphics
2	0:03.0	Intel AMT/ME
3	*	All other devices

We see there are 4 DMA remapping units in this system: one exclusive remapping unit for audio and for graphics, yet another separate unit for Intel AMT/ME device, and finally the last one for all the other devices in the system.

7. The role of the ACPI DMAR table

One could ask why do we need an ACPI DMAR table, if the SINIT module is so smart that it could figure out all the VT-d hardware configuration by just by querying the chipset?

That's true indeed that the SINIT code, and consequently the SENTER instruction, does not need any help from the DMAR table. However, all the rest of the system software, in particular the MLE that is about to be started by SENTER, is not expected to know all the specifics of the platform, and consequently it's convenient to present information about VT-d hardware configuration in some unified, chipset-independent way. And this is the role of the DMAR ACPI table.

A careful reader would, however, point out that it's insecure for MLE to relay on the ACPI tables. As we pointed out earlier, the ACPI tables are neither signed, nor protected by BIOS in any way against tampering.

To protect against tampering with DMAR table and misleading MLE into wrong usage of VT-d, the SINIT module, after it verifies the integrity of DMAR table, copies it onto the so called TXT heap, which is a special region of memory that is protected against tampering during and after secure launch.

It's important, however, for the developers of the system software to realize this security problem and to consciously use the copy of DMAR ACPI table from TXT heap, rather then from BIOS memory.

As an example, one of the developers of tboot, that is an Intel's open source implementation of trusted boot based on TXT, and that is part of the popular Xen hypervisor, has patched this problem only at the beginning of this year [6].

However, back to our attack, the situation looks pretty secure today. We have a smart SINIT code that can detect any potential tampering of DMAR table introduced before SENTER execution, and also we assume the system software is properly written and uses a copy of DMAR table from the TXT heap[6].

Is there a chance to still bypass TXT in that case?

8. The bug in the SINIT module

It turned out there is (obviously, otherwise, the authors would not be boring our dear readers with this little paper).

Let's have a look at the actual code used by the SINIT module that is used to read the base of the MCHBAR region (which, as we saw before, is extensively used for various checks of DMAR table):

```
pusha
mov     eax, 0x48 ; MCHBAR address
call    pci_get_long
and     ebx, 0xfffffffe
mov     DWORD PTR es:MCHBAR, ebx
cmp     ebx, 0xfec04000
ja      continue
mov     al, 0x4
mov     ah, 0xc
call    sinit_error
continue:
or      ebx, 0x1
call    pci_write_long
popa
ret
```

5 These are the fields at offsets 0xd40, 0d00, 0xd80 and 0xdc0. Interestingly those fields are not documented in the public chipset specification.

6 Obviously, we have also verified that it was not possible to modify the mentioned above undocumented chipset registers in the MCHBAR at offsets 0xd40, etc. Most likely thay have been locked down by the BIOS.

The problem with the code above will become clear if we look at the chipset specification for MCHBAR register definition:

Bit	Access	Default Value	RST/PWR	Description
63:36	RO	0000000h	Core	Reserved
35:14	R/W/L	000000h	Core	**(G)MCH Memory Mapped Base Address (MCHBAR):** This field corresponds to bits 35:14 of the base address (G)MCH Memory Mapped configuration space. BIOS will program this register resulting in a base address for a 16 KB block of contiguous memory address space. This register ensures that a naturally aligned 16 KB space is allocated within the first 64GB of addressable memory space. System Software uses this base address to program the (G)MCH Memory Mapped register set.
13:1	RO	0000h	Core	Reserved
0	R/W/L	0b	Core	**MCHBAR Enable (MCHBAREN):** 0 = MCHBAR is disabled and does not claim any memory 1 = MCHBAR memory mapped accesses are claimed and decoded appropriately

It shows that the MCHBAR register is a 64-bit register. However, the code accessing this register used by the SINIT module, is treating MCHBAR as if it was 32-bit long.

While this looks innocent, because on most systems the MCHBAR region is mapped below 4GB address, it might be abused by the attacker.

The attacker can, indeed, modify the value of the MCHBAR register before executing SENTER instruction. It can be done via a standard write to the PCI configuration space of the chipset (BDF 0:0.0, register offset `0x48`).

9. The attack sketch

In particular, the attacker might overwrite the MCHBAR register with the following value:

```
Y = (1 << 32) + X
```

where X is a 32-bit number. This way the SINIT code will believe that the MCHBAR region is mapped at the memory starting at the address `X`, while the real MCHBAR region will be mapped by the chipset at the address `Y`.

Now, let's assume the attacker managed to map some memory at address `X`, located between `0xfec04000` and `0xffffffff` (see the condition check in the code above). In that case the attacker can prepare a fake MCHBAR region at this lower address.

The fake region will be almost identical to the original MCHBAR region (we can just copy a block of memory[7] mapped at the address `Y`), with one exception, however. Namely, the value of the (undocumented) register in the MCHBAR region, that holds the value of Register Base Address (BAR) for the last DMA remapping unit #3 that covers most of the PCI devices in our system, will be modified to point to the BAR field of either of the first three remapping units, that are used to filter Audio, Graphics, or ME devices respectively[8].

Additionally, we need to change the corresponding BAR field in the ACPI DMAR table. This way, the SINIT code will not be able to realize that there is something wrong with the BAR field for the last remapping unit[9].

As a result, both SENTER and the system software (e.g. Xen) will be tricked into believing that the last remapping unit has its configuration registers mapped at some other memory location (in our case where are the registers of the unit #0, #1, or #2). Consequently, it won't be possible for the SENTER instruction, and later for system software, to properly setup VT-d permissions for most of the PCI devices in the system, as most of the devices are being handled by this last remapping unit.

This will result in the SENTER completing successfully, and the MLE initializing successfully, but their memory will not be VT-d protected[10]. Now, the attacker can use a DMA attack, e.g. from an unprivileged VM that has at least one PCI device assigned to it (PCI pass-through) via VT-d, and compromise the hypervisor, or other system software.

The attacker can even program one of the PCI devices *before* executing SENTER instruction in order to schedule a malicious DMA, so that it automatically subverted the newly loaded hypervisor just a moment after it gets loaded. In that case no further access to a driver domain will be needed.

10. Some details

For the attack to work, the attacker must map some usable memory above `0xfec04000` address (but still below 4G).

We should observe that even if the target system has 4GB, or more, of DRAM memory installed, still no physical DRAM is mapped in the window be-

7 We copy 16kB of memory, starting at MCHBAR address.

8 In the case of our proof of concept attack described later, we chose to point to the BAR of the DRHD[0], although the DRHD[1], and DRHD[2] would be just as good. Our DMA attack was using HDD controller, so neither of the first three DMA remapping units could be used to stop such an attack.

9 Perhaps the SINIT code could figure out that there are two different DMA remapping units in the system that use the same base addresses for mapping their registers. But perhaps this is not something forbidden by the specification. Even if SINIT could detect such a situation as suspicious, we could point the BAR register for the last remapping unit to some custom memory region.

10 Of course the values in the PCR registers 17 and 18 are not affected by the attack.

tween the address specified in the TOLUD register and 4GB (see Figure 3 below).

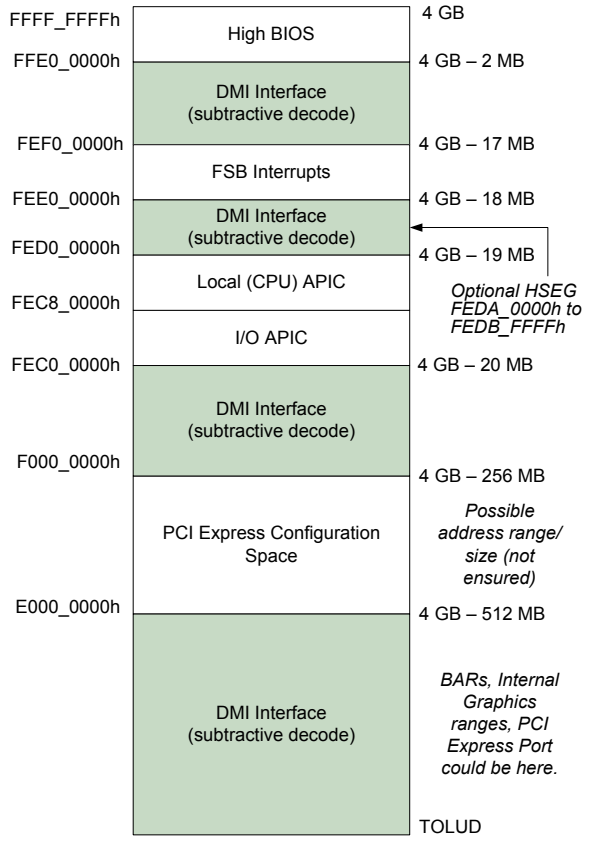

Figure 3. Physical memory mapping above TOLUD on Q45-based systems. Source: `intel.com`

However, there is a window of addresses between `0xfef00000` and `0xffe00000` that are mapped to the ICH (southbridge). Having a memory mapped in this range would satisfy the requirement for our attack, as those addresses lay above `0xfec04000`.

We have decided to use a network card device (based on Intel integrated e1000e chipset, that is part of the standard Intel ICH) in order to map some memory. It turned out that by writing to certain configuration register of the network card (MBARA), we could make a substantial amount of memory (64kB) mapped at the desired address range.

11. The Proof Of Concept

Below we present a pseudo-code for a working exploit we have written and tested on a Q45-based system.

The code below should be executed before the secure launch is started, e.g. as part of the MBR or boot loader, e.g. GRUB.

```
#define ETH_MEM 0xfef00000
#define FAKE_MCHBAR (ETH_MEM+0x10000)
#define NEW_MCHBAR ((1ULL<<32) +
                    FAKE_MCHBAR)

// map some memory above 0xfec00000...
pci_write_long (ETH_DEV, MBARA_REG,
ETH_MEM)

// copy original MCHBAR region there...
MCHBAR = pci_read_long (MCH_DEV,
MCHBAR_REG);
memcpy (FAKE_MCHBAR, MCHBAR&~1, 0x4000);

// do the trick!
DMAR = get_acpi_dmar_base();
DMAR.DRHD[3].BAR = DMAR.DRHD[0].BAR
*(FAKE_MCHBAR + 0xdc0) = DMAR.DRHD[3].BAR

// update MCHBAR...
pci_write_longlong (MCH_DEV, MCHBAR_REG,
NEW_MCHBAR | 0x1)
```

We have tested our exploit code on both a Q35-based and a Q45-based systems. We used Xen 3.4.2 hypervisor, with the most recent release of tboot [5], (tboot-20090330), together with the latest SINIT modules, and also latest versions of Intel BIOS[11].

12. Affected products

Intel® TXT is a very new technology. The first desktop machines supporting TXT started appearing only at the end of 2007, while laptops with TXT support have not been available before 2009. Consequently very few products use TXT. One example is the open source Xen hypervisor [7], that uses the earlier mentioned tboot for implementing TXT-supported boot.

11 A word of caution for those brave users who would like to try TXT at home (even without exploiting anything). If you execute SENTER on a system which has a too-old BIOS, that is not TXT-compatible, you risk... bricking your box (i.e. making it unbootable). This, in fact, happened to us a few times. We determined that this happens when we reboot the machine without doing a proper SEXIT (e.g. Xen crashes as a result of us performing some attack, or perhaps because of the power failure). In that case the Memory Controller Hub would assume that “secrets are still in memory” and would not let anybody talk to DRAM, even the BIOS, before a special SCLEAN module is loaded and executed. Of course, a non-TXT-aware BIOS doesn’t know it should execute some SCLEAN module. As a result BIOS cannot access DRAM, which obviously doesn’t allow to boot the platform. The only solution in this case seem to be desolder the SPI-flash from the motherboard, and re-flash it with original image using an external flash programmer. Obviously, not the most entertaining activity one could wish for.

There is also at least one upcoming commercial product, Citrix's XenClient, that is said to make extensive use of Intel VT-d and TXT technologies [8].

We should stress, however, that if the virtualization system is well designed, a buggy TXT doesn't automatically render the system useless and vulnerable. Rather, in normal circumstances, the attacker would still need to bypass the system-imposed protections first (e.g. exploit a potential bug in the hypervisor). The difference that the TXT makes, is that in the event of such an attack, if the attacker tried to introduce any permanent changes to the system, e.g. subvert the hypervisor or Dom0 binaries on disk, the TXT would be able to prevent such a subverted system from booting the next time.

However, there exist scenarios where TXT attack can be fatal and no other bug in the system software is required to undermine security of the platform.

One such case is when the system's goal is to contain a potentially malicious user. As an example we can consider a virtualization system, where one of the virtual machines is treated as "corporate" VM, while other VMs might be the user's private machines. The corporate IT department might want to grant this "corporate" VM an access to corporate internal network and servers (e.g. only the machine that positively passes Remote Attestation, can log into the corporate intranet), but at the same time might want to forbid the user to connect any USB device to the "corporate" VM, in order to ensure the employee will not be able to make copies of the company's internal documents. This is, in fact, one of the goals of the earlier mentioned Citrix's upcoming XenClient product [9].

For the above scenario TXT is absolutely necessary, and if the attacker can bypass TXT secure launch, e.g. using the exploit presented in this paper, the attacker can bypass such restrictions at will. And this all using an extremely cheap software-only attack[12].

Another scenario where TXT circumvention might be fatal is full disk encryption, especially in case of laptops. It's widely known that a full disk encryption scheme that is not based on a trusted boot scheme is subject to trivial attack where the attacker can subvert e.g. the boot loader in order to capture the user passphrase. Later the attacker can steal the encrypted laptop and will know the passphrase needed to decrypt it. Such attacks have been demonstrated in practice, see e.g. Evil Maid Attack [10].

It is thus very important to provide some form of trusted boot when full disk encryption is in use. Intel® TXT is a good candidate, as it doesn't require maintaing a long chain of trust throughout the boot process. However, an attack on TXT like the one presented in this paper, can be used to circumvent such a trusted boot protection of full disk encryption program, consequently leading to a successful "Evil Maid"-like attack, with potentially fatal consequences.

13. Status of the SINIT vulnerability

We have informed Intel about our discovery of the SINIT implementation error, together with description how it could be exploited by an attacker to circumvent TXT secure launch, on September 30, 2009. We then agreed to withhold the publication of this paper until Intel fixes the problem and publishes updated SINIT modules and the appropriate security advisory.

Intel has patched the SINIT bug and published updated SINIT modules on December 21, 2009 [11].

14. Summary

In this paper we have presented another attack that could be used to fully circumvent Intel Trusted Execution Technology, specifically its core mechanism for secure late launch. This is the second attack on Intel® TXT our team has disclosed this year, with the previous attack presented in February [1].

Our research in this area demonstrates that hardware-aided security is not a silver bullet and that such hardware technologies can still be sometimes attacked using software-only attacks. The mere fact that some mechanism is implemented in the CPU or in the chipset, doesn't automatically make it secure (see also our recent work on Intel AMT rootkits [12]).

We (still) believe, however, that hardware technologies, such a Intel VT-d and Intel® TXT are crucial in building secure systems.

---

12 Of course, the person that is in a physical possession of the machine, e.g. laptop, can theoretically, always gain full control over the software executing on this machine. In particular, such an attacker can e.g. replace the processor with a malicious processor that would allow for certain backdoors (e.g. ring3 to ring0 escalation), or can retrieve the secrets stored in the TPM using electron microscope, or can perform active attacks on the LPC bug in order to reset the PCR17 and PCR18 registers without executing the SENTER instruction, or can replace the DRAM chips with ones that would record the contents of the memory onto external device. Such physical attacks are however considered very expensive to perform, often much more expensive than the data that are supposed to be protected by such systems.

It is unavoidable for such complex technologies such Intel® TXT (and we also think Intel VT-d) to contain bugs. But there is a lot that vendors, like Intel, could do to improve security of their products. For instance, it seems to be a rather unfortunate decision to keep certain things closed source and undocumented, e.g. the SINIT module internals. Publishing the SINIT source code, together with more complete information about the chipset, and perhaps even then microcode used by the SENTER instruction, would likely allow more people (besides Intel employees) to better review the security properties of those new technologies.

Acknowledgments

Authors would like to thank Joseph Cihula from Intel for reviewing the paper.

References

[1] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel® Trusted Execution Technology. Presented at Black Hat DC 2009, Washington, DC, USA, February 2009.

[2] David Grawrock. Dynamics of a Trusted Platform: A Building Block Approach. Intel Press, 2009-04-15.

[3] Intel Corp. Intel® Virtualization Technology for Directed I/O. http://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf,

[4] Rafal Wojtczuk, Joanna Rutkowska, and Alexander Tereshkin. Xen 0wning Trilogy: code and demos. http://invisiblethingslab.com/resources/bh08/, August, 2008.

[5] Intel Corp. Trusted Boot (tboot). http://sourceforge.net/projects/tboot, 2007-2009.

[6] Joseph Cihula. [PATCH] txt: 3/6 - use TXT's DMA-protected DMAR table to setup VT-d. http://lists.xensource.com/archives/html/xense-devel/2009-01/msg00004.html,

[7] Citrix. Xen Hypervisor. http://xen.org/,

[8] Citrix and Intel Corp. Citrix & Intel Working Together to Deliver Local Virtual Desktops. http://community.citrix.com/download/attachments/100303689/CitrixXenClient_SolutionBrief.pdf?version=1,

[9] Patrick Gelsinger and Ian Pratt. Citrix Synergy 2009 conference keynote. http://www.citrix.com/tv/#videos/423,

[10] Alexande Tereshkin and Joanna Rutkowska. Evil Maid goes after TrueCrypt! http://theinvisiblethings.blogspot.com/2009/10/evil-maid-goes-after-truecrypt.html, October, 2009.

[11] Intel Corp. SINIT misconfiguration allows for Privilege Escalation. http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00021&languageid=en-fr, December 2009.

[12] Alexander Tereshkin and Rafal Wojtczuk. Introducing Ring -3 Rootkits. Presented at Black Hat USA 2009, Las Vegas, NV, USA, July 2009.