


RESEARCH

Open Access



IOMMU protection against I/O attacks: a vulnerability and a proof of concept

Benoît Morgan^{2,1*} , Éric Alata^{2,1}, Vincent Nicomette^{2,1} and Mohamed Kaâniche¹

Abstract

Input/output (I/O) attacks have received increasing attention during the last decade. These attacks are performed by malicious peripherals that make read or write accesses to DRAM memory or to memory embedded in other peripherals, through DMA (Direct Memory Access) requests. Some protection mechanisms have been implemented in modern architectures to face these attacks. A typical example is the IOMMU (Input-Output Memory Management Unit). However, such mechanisms may not be properly configured and used by the firmware and the operating system. This paper describes a design weakness that we discovered in the configuration of an IOMMU and a possible exploitation scenario that would allow a malicious peripheral to bypass the underlying protection mechanism. The exploitation scenario is implemented for Intel architectures, with a PCI Express peripheral Field Programmable Gate Array, based on Intel specifications and Linux source code analysis. Finally, as a proof of concept, a Linux rootkit based on the attack presented in this paper is implemented.

Keywords: Security, IOMMU, Firmware, Linux, Vulnerability, Attack

Introduction

Historically, early personal computers and their peripherals were mostly designed and built by the same company. The peripherals used to be much less complex than today (microcode, firmware, etc.) and the processor manufacturers used to trust the peripherals. However, with the increasing demand for higher performance levels and hardware services, more sophisticated architectures have emerged with multiple input/output communication channels. In particular, normalized communication buses have been specified to allow tier manufacturers to complement bare architectures with complex peripherals. Then, to relieve the host processor from performing certain data copies and operations, DMA cycles have been added to these external buses, allowing peripherals to perform read/write accesses to other peripherals and RAM segments. These communication channels raise serious security concerns as they offer opportunities to attackers to corrupt the system and the hosted applications using some malicious peripherals. To cope with such attacks, also called I/O attacks, some hardware protection

components, such as the IOMMU, have been included in modern computers.

In order to really take advantage of these security components, they have to be properly configured and activated by the firmware and the kernel at boot time. The security of the boot process is crucial, as weakness at this stage may lead to a serious security flaw, despite the reliable design of these components. To the best of our knowledge, the security of the boot process has not been thoroughly investigated in the literature. This paper focuses on the security analysis of the IOMMU activation process at boot time. In particular, it is shown that even if this component has been introduced 10 years ago, some serious security concerns may be raised about its actual efficiency to prevent malicious I/O attacks. Briefly, this paper highlights a novel attack scenario that is related to the fact that the IOMMU configuration tables are initialized in a DRAM region which is *not protected from DMA at startup*. As a consequence, a malicious peripheral may modify these tables just before the activation of the IOMMU by the hardware. To illustrate the feasibility of this scenario, a proof of concept is implemented and presented in this paper.

A preliminary description of the vulnerabilities and the exploitation scenario of the IOMMU was presented in [1].

*Correspondence: benoit.morgan@laas.fr

²INSA Toulouse, 135 avenue de Rangueil, 31400 Toulouse, France

¹Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS), 7 avenue du Colonel Roche, 31400 Toulouse, France

In this paper, more technical details are provided, in particular, regarding the related work, the description of the proof of concept, and the experiment carried out to illustrate the IOMMU vulnerability and its exploitation. The potential impact of the identified vulnerability and the main limitations are also discussed.

This paper is organized as follows. The next two sections describe fundamental components of the architecture involved in the identified design weakness. The “Technical background” section presents some basic technical background about PCI Express bus and communications that are used to perform DMA, while the “IOMMU internals—DMA remapping” section presents the IOMMU component as well as some of its internals that are necessary for the reader to understand the vulnerability and its potential exploitation. The “I/O vulnerabilities and related work” section briefly presents some examples of I/O attacks. The “Bypassing IOMMU” section describes the vulnerability that we discovered in the configuration of the IOMMU and a scenario illustrating its possible exploitation. The “Countermeasures and discussion” section proposes some countermeasures to cope with this vulnerability. Finally, the “Conclusion” section concludes and discusses future work.

Technical background

This section presents basic background concepts related to the PCI Express bus and communications that are useful to understand the rest of the paper.

PCI Express bus

Several bus specifications like Industry Standard Architecture or Peripheral Component Interconnect (PCI) have been implemented to support the communications between the CPU and the peripherals. Today, the PCI Express bus is used in most personal computers and servers. There are three main types of PCI Express devices. The root of the bus hierarchy, called the root complex, is connected to the CPU, thanks to the host bridge and to the first-level PCI Express children devices. These devices can be endpoints (so-called peripherals in the paper) and bridges. A bridge connects two different logical bus domains with an upstream and a downstream port.

Communications

Each PCI Express device is identified with a PCI logical bus, a device, and a function identifier (id), noted `bus:dev.fun`. This identification is used to route PCI Express messages between devices.

The receiver of a message is either identified by its identifier or by an address. Thus, an address has two purposes. Either it corresponds to an element in the main memory and the memory controller redirects the corresponding

access to the DRAM or it corresponds to a register of another device and the memory controller redirects the corresponding access to the device. In the latter case, the registers of the device are said *memory mapped*. For instance, a memory read message contains a destination address and a device requester id: the destination of the corresponding memory read completion (response) is the associated requester id. PCI Express messages are therefore routed by address or id. To route the messages correctly, the bridges are configured by the host to know which ids and memory ranges are responsible for downstream communications. Figure 1 illustrates the routing of a memory read message targeting address `0xfxc` from device `03:00.0` and its associated completion from the host bridge to the requester device.

At boot time, the devices are not yet configured and they do not know their own identifier. Indeed, the manufacturer does not know the bus on which the device will be plugged. However, when the firmware looks for all available devices (using the assembler instruction `mov` to read the PCI Express space mapped in memory), each memory access is processed by the host bridge. The host bridge then translates this access into a PCI Express configuration request which is routed to the corresponding bus. In particular, this request contains the identifier of the contacted device. If this device is available in the system, it receives this request and then knows its identifier. This step is important to allow a device to communicate.

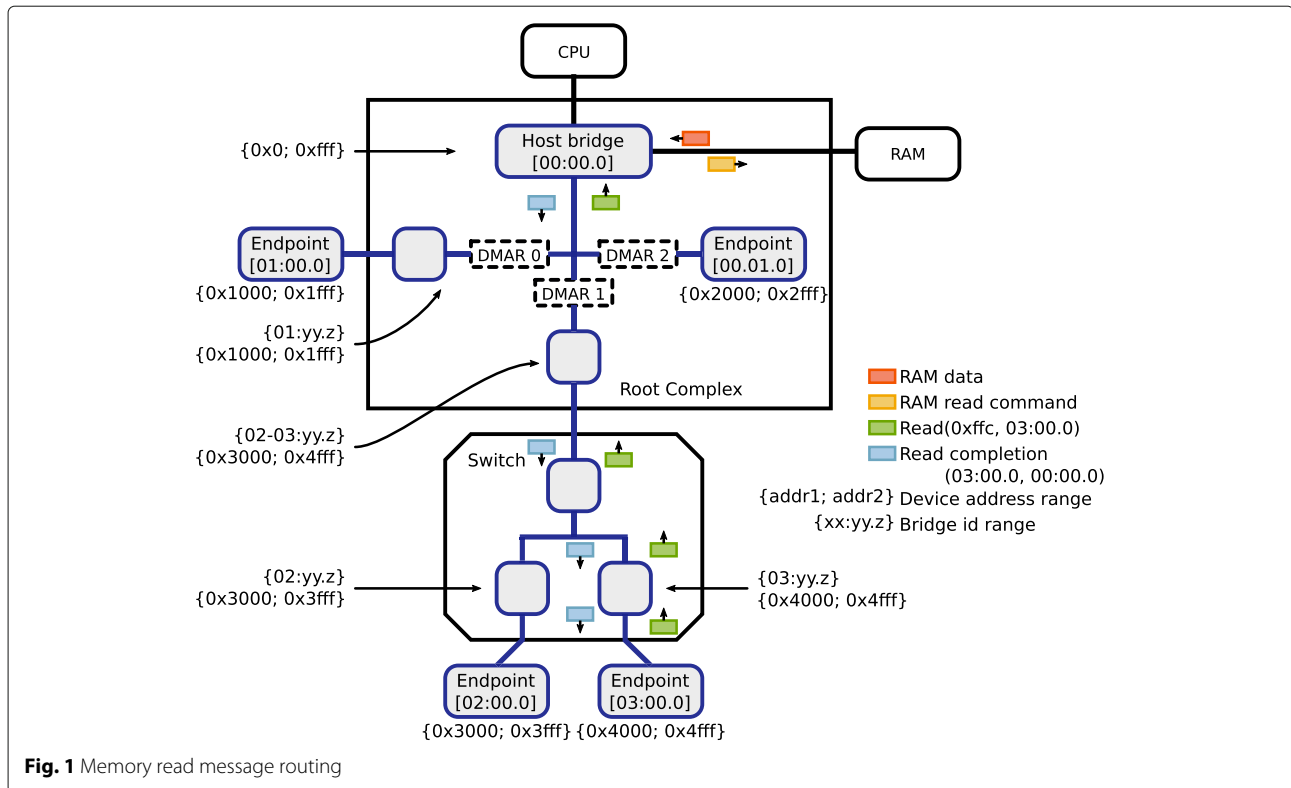
PCI Express peripherals are able, through DMA requests, to access other peripheral memory and the main RAM, even in the kernel memory regions, without any control of the processor. This raises a major security concern if the peripheral is controlled by an attacker. To mitigate this threat, Intel has integrated the IOMMU hardware component, to filter PCI Express messages. This component is presented in the next section.

IOMMU internals—DMA remapping

This section presents the services provided by the IOMMU and describes how it is configured.

Peripheral address space virtualization

IOMMUs are designed to virtualize the memory space and the interrupts of the peripherals. Memory virtualization is implemented in the so-called DMA remapping units (DMAR) of the IOMMU. The DMAR has been designed to simplify address space association and coherency between the drivers and the hardware. Indeed, device drivers use a different address space than the peripherals because they process virtual addresses configured by the kernel. When it is necessary to pinpoint a specific location in the DRAM to configure DMA, the drivers have to translate virtual addresses to the physical address space used by the device. This operation increases the



design complexity of the device drivers and microkernel-s/hypervisors. Using DMAR, the peripherals can share the same virtual address mappings as the drivers (at least for translation). As a consequence, the drivers can configure virtual addresses into peripheral DMA configuration registers. Furthermore, since DMAR has a two-level address translation mechanism, the same strategy can even be used within virtual machines and pass-through drivers. Consequently, virtualized MMU mappings can be easily matched to IOMMU mappings by the hypervisors. The IOMMU also brings similar advantages as the MMU in the cores. It is for example possible to detect faulty devices accessing unmapped physical pages because of software or hardware bugs.

From the security point of view, the filtering feature of DMAR allows to immediately cope with the largest part of legacy I/O attacks, if properly configured. The next section introduces the basic concepts of DMAR configuration which are necessary to understand our attack.

IOMMU configuration

An architecture can contain several IOMMUs, each one dedicated to a subset of buses (Fig. 1). DMAR units translate and filter requests according to the protection domain assigned to the emitter device. A protection domain is defined by a set of translation policies. The process is divided into two phases. The first one identifies the

protection domain assigned to the emitter device. This phase, called device to domain mapping, is conceptually similar to an address translation but instead, it associates PCI identifiers to address translation domains. In the second phase, called address translation, the addresses used by peripheral memory accesses are translated by DMAR, before crossing the host bridge (Fig. 1). This translation is similar to the one carried out by the cores Memory Management Unit (MMU). Access controls are applicable in the two translation phases.

Each DMAR unit can be configured separately in a configuration page and a tree structure (Fig. 2), the latter placed in DRAM. This configuration page contains the main control register called Global Command Register (GCMR) which is designed to activate the translation mechanism, thanks to the translation enable (TE) bit [2, section 10.4]. It also contains a pointer to the tree structure root (i.e., the root entry of the root table in Fig. 2), named the Root Table Address Register (RTAR). The location of the configuration page of a DMAR is identified by means of a dedicated register in the memory controller, set by the firmware. The identifier of a PCI Express message sender is used to index the first two tables of the tree structure (the root table and the context table) in the first phase. The resolved address is then used to pinpoint the structures for the second phase. These structures are indexed with the destination

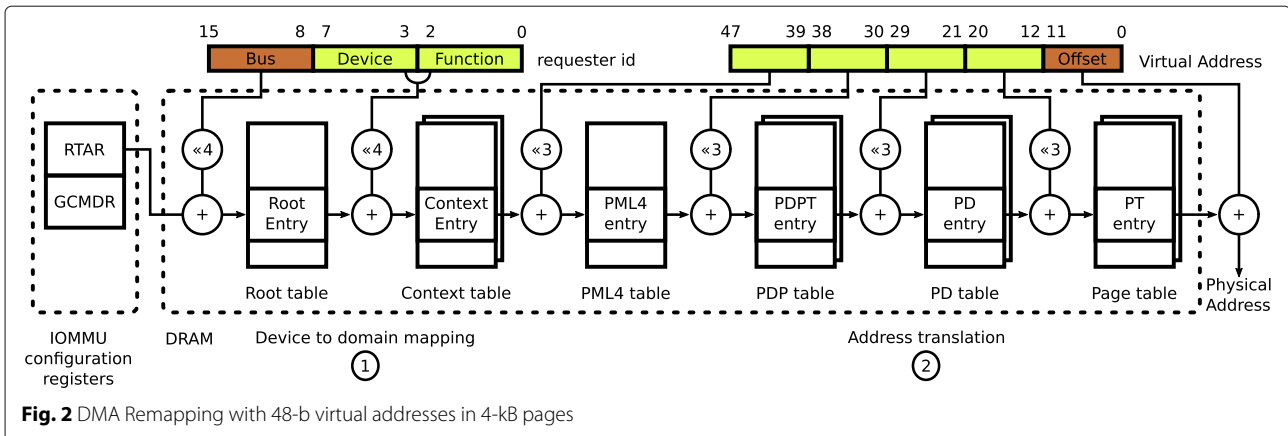


Fig. 2 DMA Remapping with 48-b virtual addresses in 4-kB pages

address of the PCI Express message. The result is the physical address of the translation. During the handling of these tables, a dedicated bit indicates if the access is granted or forbidden. Finally, it is noteworthy that the second translation phase can be deactivated (pass through mode) thanks to the translation type field of a context entry.

Unfortunately, like other hardware or software systems, these mechanisms may be inefficient if the implementation and the configuration are not correct. In the following, we briefly present some examples of such vulnerabilities before describing in detail the novel vulnerability we discovered as well as the exploitation example we implemented.

I/O vulnerabilities and related work

Many I/O attacks have been presented in the literature [3–7]. In this paper, we focus on DMA attacks. These attacks were described in several studies. In particular, [8] demonstrates that an outsider can compromise an entire system by exploiting a vulnerability in one of its network cards. Fortunately, most of these vulnerabilities have been fixed with the integration of the IOMMU.

As presented in the previous section, a well-configured DMAR unit is theoretically able to enforce an expected access control policy on a DMA accessible memory. The access control policy is given by the DMAR domain, and a domain is associated to a device through device to domain mapping. Consequently, in order to be efficient, the DMAR unit needs to identify precisely each DMA-capable device.

However, in [9], the authors take benefit of the collocation of PCI Express and PCI bus to exploit a weakness in the filtering performed by the IOMMU. Indeed, the PCI to PCI Express bridge uses its own PCI id as the requester id when translating PCI read/write cycles to PCI Express messages, acting like a proxy. Therefore, all the

devices behind the PCI Express to PCI bridge are sharing the identity of the bridge from the IOMMU point of view, and so the same domain. They exploited this vulnerability with a malicious firewire controller plugged behind the same bridge of a legitimate network card, to inject malicious Ethernet frames into an IP kernel stack. Finally, they consequently succeeded in corrupting an operating system ARP cache by injecting customized ARP reply packets.

The authors of [10] argue that, for performance reasons, kernels are unable to enforce a strict IOMMU security policy regarding DMA buffers management. Buffers shared between a driver and a peripheral for DMA are allocated and deallocated at a given frequency. Each time a DMA buffer is allocated or freed, the IOMMU configuration has to be updated. To commit configuration changes, the CPU has to flush the IOMMU translation caches called Input/Output Translation Lookaside Buffer (IOTLB) to update the security policy. However, the update cannot be committed at each change of the DMA memory map because that will severely degrade the system performances. Indeed, this operation consumes an average of 2000 CPU cycles in Intel Sandy Bridge architecture. This is why the IOTLB flush is deferred by the kernel and performed at a lower frequency than DMA mapping changes in the system. This performance scaling technique opens potential time intervals during which a buffer is still available to a peripheral despite the fact that it had been reallocated. To our knowledge, this vulnerability has not yet been exploited to date.

In 2011, Wojtczuk and Rutkowska succeeded to attack Xen hypervisor, with an active IOMMU, using malicious Message-Signaled Interrupts (MSI) [11]. The main idea is to reproduce the obsolete bouncing I/O attack in which software could use peripherals’ capability to DMA in order to modify kernel memory space. To bypass the protection, authors use the capability of devices, here an

Intel e1000 network controller, to set an arbitrary interrupt vector into an MSI, to remotely fire a hypercall. If the malicious hypercall is issued when the virtual machine is executing a specific system call, which is controlled by unprivileged malicious application code, it is possible to execute a remote buffer overflow with hypervisor execution privileges.

Even if most of these attacks are today inefficient in recent architectures, they highlight the fact that the configuration of the IOMMU by the firmware and IOMMU Linux driver in recent kernels presents new weaknesses that could be exploited by an attacker. More recently, authors of [12] presented a boot time DMA attack allowing an attacker to recover macOS FileVault2 passwords, using a thunderbolt device. This attack leverages the same vulnerability we had previously introduced in [13] and [1], which allows DMA at boot time before the activation of any protection mechanism. Also, this vulnerability has been highlighted in a developers mailing list of the coreboot x86 firmware project [14]. The next section details this vulnerability and presents an attack scenario in which one can directly bypass IOMMU protection against I/O Attacks.

Bypassing IOMMU

This section describes some weaknesses discovered in the firmware and the Linux kernel and discusses when and how these weaknesses can be exploited to bypass DMAR. The observations presented in this section are based not only on the Intel documentation, but also on the information collected empirically with the hypervisor and Field Programmable Gate Array (FPGA) malicious peripheral prototypes presented in the next section. The experiments performed in our study are based on Dell machine precision T1700, with the following technical details: firmware A06 (12/05/2013); Linux 4.3.4 (with `IOMMU_INTEL=y`); Intel i7-4770 processor; chipset Intel PCH c220; bootloader grub 2.02.beta2.

Attack assumptions

Two preconditions are needed for the attacker to be successful: the first one concerns the configuration of the target machine and the second one concerns the access to the PCI Express bus.

As regards the first precondition, we assume that the attacker has a precise knowledge of the hardware and kernel version of the victim machine, either directly, through physical access, or indirectly by fingerprinting. This assumption is realistic in many situations, e.g., in big companies or professional working environments, where, most of the time, homogeneous and standard machines are deployed, to simplify administration and management tasks.

As regards the access to the PCI Express bus, this can be achieved either by plugging a malicious peripheral or by remotely corrupting a peripheral of the victim machine using for instance the attack presented in [8]).

A firmware-induced vulnerability

The attack we present in this paper first relies on a vulnerability of the firmware used in the architecture.

Device configuration

At startup, IOMMUs are deactivated. During the boot time configuration, the firmware scans the PCI configuration space to discover and initialize vital devices, like the main video controller, loading and executing its embedded firmware. Read and write accesses to the configuration space generate PCI Express configuration messages that are sent to the targeted device. Devices know their PCI ids after the first scan, as explained in the “Communications” section. Therefore, the peripherals are able to generate valid messages at the early firmware execution stage, long before the execution of the bootloader and then of the operating system kernel. At the end of the execution of the firmware, the control is given to the bootloader and to the kernel.

PCI Express bridges early configuration

Nowadays, DMA is granted early by firmwares to all peripherals, mainly for compatibility reasons. The reason is that modern kernels, in our case Linux, do not seem to know how to deal with PCI Express bridges configuration. This is a common fact that we have verified on our target firmware architecture and which is also widely acknowledged in open source firmware projects [14]. More precisely, PCI Express bridges configuration contains the Bus Master Enable (BME) bit which controls the upstream communications, i.e., if the messages generated by child devices are allowed to go upstream and so perform DMA.

As a consequence, devices can initiate DMA requests long before IOMMU configuration and kernel loading.

Linux and Intel IOMMU driver DMA weaknesses

To fulfill our attack, we also exploited a weakness in the Intel IOMMU Linux driver `drivers/iommu/intel-iommu.c`. The latter makes the IOMMU configuration vulnerable during the boot up process. In order to understand the details, it is necessary to review the boot sequence on a Linux machine.

IOMMU configuration

After the firmware operations, the kernel is uncompressed and loaded by the bootloader (GNU GRUB in our experiment). Linux kernel modules and drivers are then loaded and initialized. DMAR is configured by one of these

drivers. The Intel IOMMU driver creates the translation structures and writes them in the main DRAM. It builds the address translation domains and device to domain mapping before copying the root table pointer into the associated register. Finally, the driver activates DMAR by setting the TE bit of the GCMD Register. Let us note that these structures are stored in memory in *areas not protected by any security mechanism*.

Cache policy

The vulnerability we discovered is exploitable also because Linux flushes cache lines (L1 / L2 and Last Level Cache) after every table entry modification, to ensure the integrity of the structure in memory. Consequently, the time period during which the entire DMAR configuration exists in memory is maximized.

Physical memory space

We noticed that as long as the machine hardware configuration is not modified, the physical address of the DMAR root table is not changed. This property simplifies the exploitation presented in this paper, preventing the attacker from searching DMAR structures into the physical memory space.

A vulnerability window

Considering the previous observations, Fig. 3 highlights the time periods during which some malicious peripherals can initiate DMA requests. Two important time windows are identified. The vulnerability window represents the time interval during which peripherals are able to perform DMA. It starts right after device address association by the firmware and ends after IOMMU activation in the IOMMU driver activation phase. Right after IOMMU activation, DMA is denied for malicious peripherals. The write window depicts the short interval of time during which the IOMMU configuration is fully placed in DRAM and is vulnerable.

Let us remind that, as stated in the “Cache policy” section, the DMAR configuration physical address does not change, as long as the hardware, the firmware, and the kernel remain unmodified.

Attacking DMAR service

This subsection presents how to exploit the vulnerability window described in the previous section, in order to bypass IOMMU protection against I/O attacks.

Prerequisites

The attack scenario requires to locate the root of IOMMU configuration in memory by reading the RTAR register. As explained in the previous sections, this address remains constant and the RTAR register can be easily read within a Linux kernel module loaded at runtime.

The attack requires a free memory page. As the physical memory map does not change between two reboots, the attacker can easily find free memory pages using the coloring technique to see if any software has modified the page during the boot process. One can color a page during the preboot phase, using an UEFI custom application and can read it after the boot process using the dd command on /dev/mem file for example.

Exploitation

Our exploitation aims at bypassing the IOMMU memory protection without altering the integrity of the kernel itself. Fulfilling this constraint makes the attack more difficult to identify.

The easiest way for a malicious peripheral to access the DRAM is to enable pass through mode in the context entry of the DMAR (the address translation part of Fig. 2 must grant every access of the malicious peripheral). The following steps describe a possible scenario to grant these accesses at the startup of the system.

First, the configuration of the system, at startup, allows all peripherals to write to DRAM. Thus, the malicious peripheral can easily produce a malicious context table in the preselected free memory page (step 0, Fig. 4), with all entries set to pass through. This step must be performed at the beginning of startup, during the vulnerability window.

When Linux begins its execution, it first writes its own root table (steps 1 and 2). The second step to be performed by the attacker is to overwrite the root table entry associated to her device. If the corrupted device is connected to

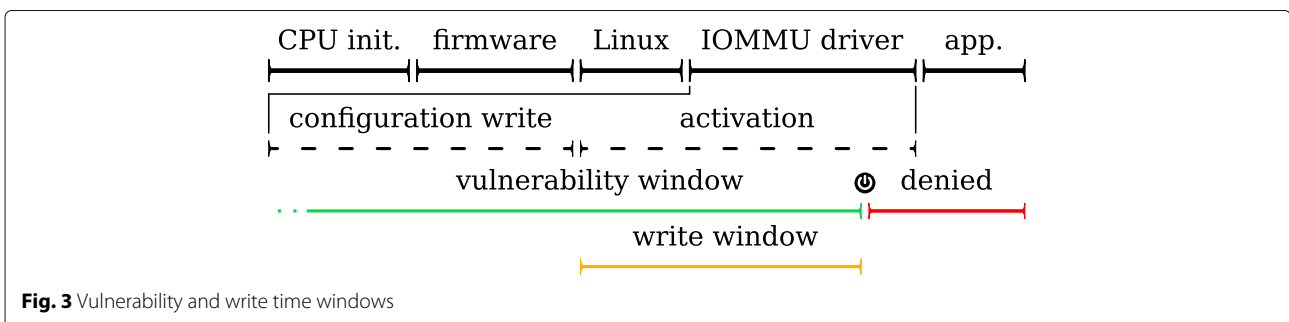


Fig. 3 Vulnerability and write time windows

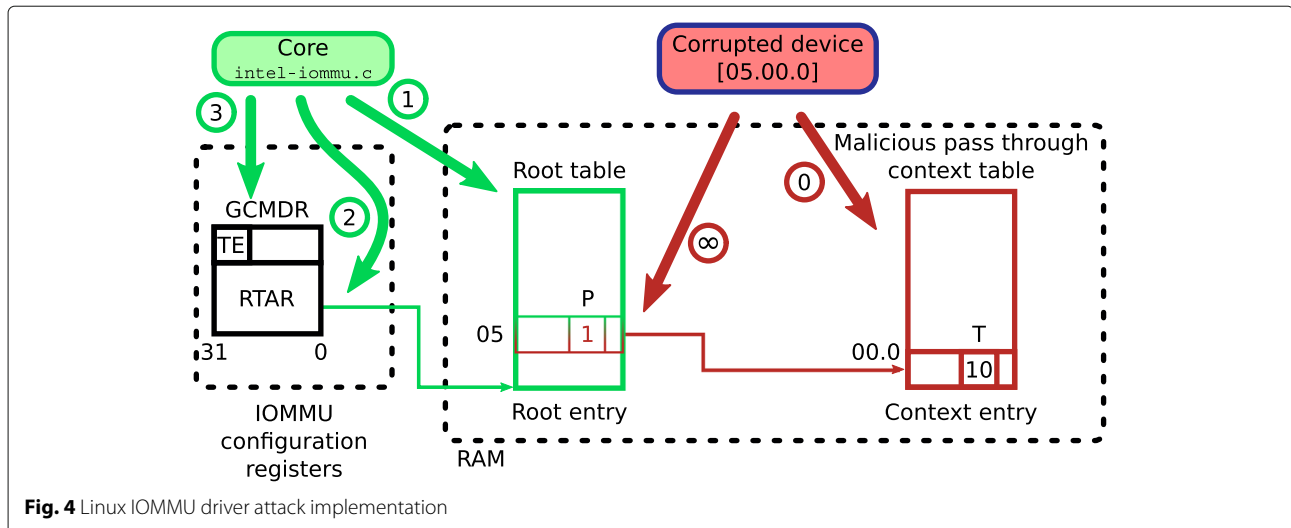


Fig. 4 Linux IOMMU driver attack implementation

logical bus 5, the sixth entry has to be overwritten before the IOMMU activation through bit TE.

To maximize the chances to actually overwrite the root table entry during the write window, the malicious peripheral floods the bus with PCI Express write requests to the sixth entry of the root table (step ∞). Finally, Linux activates the IOMMU (step 3) with compromised DMAR structures. From now, the malicious peripheral can perform any DMA.

Attack proof-of-concept implementation

In order to validate our intuition, we performed preliminary experiments and developed a proof of concept. For that purpose, we first used a tiny bare-metal recursive hypervisor library called *Abyme* that we developed. Then, we ran the real attack with a corrupted peripheral, simulated by a FPGA PCI Express peripheral.

Preliminary hypervisor-based implementation

Abyme is used as a privileged tool to monitor IOMMU configuration events. The goal here is to perform, with software, the steps (0) and (∞) of the attack implementation. Since x86 hypervisors share DRAM physical address space with PCI Express peripherals, the simulation of the preceding steps in our hypervisor is close enough to the malicious peripheral point of view.

Abyme library allows the developer to register hooks which are functions called back when selected virtual machine interruptions occur. It is also possible to reconfigure virtual machine level memory translation, remap pages, or change their access control in order to generate memory access faults and trigger the corresponding hook.

To simulate the attack, our strategy is as follows. We first configure our hypervisor at boot time, before the startup

of the Linux virtual machine. We register a hook to the virtual machine page fault event (EPT violations), which is raised when virtual machine memory accesses are denied. We also register a hook to virtual machine interruptions raised when the step-by-step execution mode is active. Then, we unmap (read, write, and execute credentials off) the DMAR unit 0 configuration page, associated to the external peripherals. Finally, we launch the virtual machine.

The goal here is to detect when the Linux IOMMU driver activates the translation setting `GCMDR.TE` bit to 1, to perform steps (0) and (∞) with the hypervisor. The Linux IOMMU driver accesses the configuration page several times. Each time an access is performed (i) a virtual machine interruption occurs; (ii) the single step execution is activated; (iii) if `GCMDR.TE` bit is to be written, we perform steps (0), (∞) and flush the caches; (iv) we remap DMAR 0 page; (v) we continue virtual machine execution to let the virtual machine execute the write access; (vi) single step virtual machine interruption occurs; (vii) we unmap again DMAR 0 page; and (viii) we resume virtual machine execution.

With this scenario, we were able to successfully simulate the attack described in Fig. 4.

In this preliminary experiment with our hypervisor being successful, we decided to implement a real proof of concept on an FPGA PCI Express peripheral. This FPGA and our implementation choices are presented in the following.

FPGA device implementation

We based the development of our malicious peripheral on Milkymist System on Chip [15], which is originally a video Djing open hardware project. Thanks to the hardware flexibility of FPGAs and the thoughtful modularity of

hardware and software developed for Milkymist, we removed unneeded functionalities and added a PCI Express endpoint stack with minimum effort. Milkymist is made for a custom board hosting a Xilinx virtex 4 FPGA [16]. Since we needed a PCI Express connector and gigabit transceivers to develop the PCI Express peripheral, we have chosen a Xilinx ML605.

Our SOC contains the original Milkymist Lattice Mico32 microprocessor (LM32), Onchip ROM, Ethernet MAC, bus bridges, caches, and controllers. In addition, we developed the malicious PCI Express peripheral, PCIE-EP. This core brings host memory access to the SOC through PCI Express memory messages. With PCIE-EP, the LM32 is able to program memory reads and memory writes (with a high rate mode). Note that this SOC is flexible and can be adapted for other purposes, e.g., for the implementation of integrity tests in the context of a hardware-assisted trusted architecture as presented in [17].

Results and proof of concept

In order to demonstrate that the exploitation of the discovered vulnerability allows a potential attacker to further take control of the host, we considered an example of a kernel rootkit that we injected in kernel memory through DMA requests performed by our malicious FPGA device, once it has successfully modified the IOMMU configuration so that it can make read/write accesses to kernel memory.

Rootkit attack

The considered rootkit is a binary code which is injected in kernel memory and modifies the behavior of the `setuid` kernel system call. According to the POSIX programmer's `setuid` manual page, the normal behavior for this system call is to modify the real user uid, effective uid, and others accordingly, if the user has the rights or enough privileges to do so. Our rootkit modifies the preceding behavior in a way that each time this functionality is called, the `uid` (effective uid) of the calling task is systematically set to 0, which gives root user effective privileges to the calling process. We developed a small C code to call `setuid` function, that we executed by a non-root user, both in the presence and in the absence of our exploitation of the IOMMU configuration. The short video at <http://homepages.laas.fr/nicomett/SSTIC2016/iommu-pwn-sstic.webm> shows the rootkit installation and use.

Implementation details

Linux system call implementation is located in the file `kernel/sys.c`. Our attack modifies the `setuid()` system call implementation. The following listing contains the most relevant parts of its code.

```
SYSCALL_DEFINE1(setuid, uid_t, uid)
{
    struct user_namespace *ns = current_user_ns();
    const struct cred *old;
    struct cred *new;
    int retval;
    kuid_t kuid;
    // (1)
    kuid = make_kuid(ns, uid);
    [...]
    new = prepare_creds();
    [...]
    old = current_cred();
    [...]
    // (2)
    new->fsuid = new->euid = kuid;
    // (3)
    retval = security_task_fix_setuid(new, old,
        LSM_SETID_ID);
    [...]
    // (4)
    return commit_creds(new);
    [...]
}
```

Our objective is to modify the `setuid` system call code to set the `euid` to 0 for the current process. If we look at the source code, we can notice that the `euid` is stored in the new structure (2) and that this modification is committed at the function return (4). Let us note that we also have to jump the function call located in (3), which invalidates our possible modifications to `new->euid` field.

Consequently, we have to inject some code to modify `new`, cancel security checks, and make sure that `new` would not be modified afterwards.

The next listing illustrates the main steps of new affectation with the required `uid` as function parameter.

```
ffffffff810868f0 <Sys_setuid>:
[.]
// (1)
ffffffff81086909: mov %rdi,%rbx
[.]
// (2)
ffffffff8108698e: mov %ebx,0x14(%r12)
ffffffff81086993: mov %ebx,0x1c(%r12)
// (3)
ffffffff81086998: mov $0x1,%edx
ffffffff8108699d: mov %r13,%rsi
ffffffff810869a0: mov %r12,%rdi
ffffffff810869a3: callq ffffffff81279be0 <
    security_task_fix_setuid>
[.]
// (4)
ffffffff810869af: mov %r12,%rdi
ffffffff810869b2: callq ffffffff81094870 <
    commit_creds>
ffffffff810869b7: movslq %eax,%r14
ffffffff810869ba: pop %rbx
ffffffff810869bb: mov %r14,%rax
ffffffff810869be: pop %r12
ffffffff810869c0: pop %r13
ffffffff810869c2: pop %r14
ffffffff810869c4: pop %rbp
ffffffff810869c5: retq
```

The parameter is copied in `%rbx` (1). The code affectation of `new` is located at (2), directly followed by the credential check function call (3). Then, (4) invokes the credential commit function and returns from the system call. Finally, let us recall that Linux core kernel

virtual address space is not randomized. Furthermore, physical addresses can be trivially deduced from virtual one with the following bitwise function: $f(a) = a \wedge \neg 0xffffffff80000000$. Also, we can note that instruction at `0xffffffff81086998` is 4 B aligned, which makes DMA easier because it is also aligned to double words.

Therefore, our rootkit payload is written at physical address `0x01086998`. The payload simply writes the double word 0 to the (2) affectation addresses. Then, it jumps over (3) function call, to (4). The rootkit payload listing is listed hereafter.

```

ffffffff81086998 <rk>:
ffffffff81086998: xor  %ebx,%ebx
ffffffff8108699a: mov  %ebx,0x14(%r12)
ffffffff8108699f: mov  %ebx,0x1c(%r12)
ffffffff810869a4: jmp  0xffffffff810869af
ffffffff810869a6: xchg %ax,%ax //padding

```

The first instruction sets b register (`%rbx`) to zero. Then, the two next instructions set to zero new structure pointed by register 12 (`%r12`). Finally, we jump over the remaining invalid bytes of credential function call to go to the commit call.

This proof of concept illustrates that in the presence of the vulnerability window highlighted in this paper, IOMMU protection can be bypassed at boot time to run old-fashioned classical I/O attacks. These consequences may be extremely serious, far beyond those of the experiment. It could be possible for an attacker to compromise confidentiality by listening to the activities on the system by adding a sniffer. It could also be possible for the attacker to simply shutdown the system and make it unavailable.

Countermeasures and discussion

Technically, the vulnerability window described in the “Bypassing IOMMU” section is present in every machine whose firmware allows DMA in the bridges because of kernel compatibility and legacy reasons. We believe that a large part of nowadays machines are impacted by this DMA vulnerability window. In particular, the firmware developers almost all the time rely on frameworks or libraries like [14] and [18] in which this option is enabled. Accordingly, it is important to implement corrective actions.

This section discusses some countermeasures to cope with the investigated attack scenario and the trade-off between the cost and the efficiency of these countermeasures. Also, we discussed the limitations and constraints associated with the considered attack.

Countermeasures

Efficient protection solutions to cope with the attack discussed in the previous sections, and more generally boot

time attacks, mainly rely on the configuration of the hardware of each computer, which is the responsibility of both the firmware and the kernel.

From our point of view, even if it is challenging for operating systems to take into account all the details of all hardware platforms, they should support at least the security-related features of the system bus, PCI Express in our case study. We previously mentioned the behavior of the firmware regarding the BME bit present in the PCI Express bridges. This bit can prevent the unnecessary pre-boot DMA capability of peripherals and so avoid the vulnerability window. This protection seems to be the best countermeasure for the attack presented in this paper. As a matter of fact, both kernel and firmware developers have to rethink the system bus pre-boot configuration. Actually, as stated in [14], this bit is set to zero because of legacy and kernel compatibility reasons. This solution could be easily included in the next generation of machine firmwares. However, there is still the question about updating currently used firmwares. This complex and expensive operation, which is under the responsibility of machine manufacturers, can take several months to be implemented.

Some alternative solutions can also be investigated to increase the effort needed by an attacker to succeed in performing the considered attack. For instance, the platform that we have studied brings additional security features: some DMA protected configurable memory segments (the DMA Protected Range specified by the processor and the Protected Memory Ranges implemented in the IOMMU) [19, Vol. 2, 2.5]. Linux does not use these memory segments, placing IOMMU structures outside the protected ranges. Devices are consequently able to read and write the IOMMU configuration before its activation. These DMA-protected memory segments are common in modern architectures and should be systematically used to set up such hardware protection components, such as the IOMMU.

Despite these protections, the system remains vulnerable to DMA attacks while the firmware is being executed, in the first phase of the boot process. This weakness is due to the fact that the firmware does not filter DMA. It can be exploited by a malicious peripheral to modify the code of either Linux or the firmware itself and so prevent the activation of the IOMMU. To address this problem, it is necessary to check the integrity of software components, e.g., by using a technology like Intel TXT.

Limitations

As discussed in the “Bypassing IOMMU” section, to be successful, the attacker must have a precise knowledge of the hardware and kernel version of the victim machine, to be able to predetermine the exact physical addresses

of IOMMU configuration and of the required free pages. While such information can be easily obtained for standard machines, it is less realistic in the case of specific or customized platforms.

Another concern that can be raised is related to the stealthiness of the attack. The attack consists in configuring the IOMMU using identity mapping¹ in the way that it deactivates the translation with pass-through mode.

In our attack scenario, either the attacker has a physical access to the victim machine and uses a malicious peripheral or a device of the victim machine is corrupted remotely. In the first case, the attack can be designed to be stealthy by construction (e.g., by hiding the malicious logic in addition to the implementation of the legitimate behavior). In the second case, two situations can be distinguished. If the corrupted device is configured by the kernel with identity mapping option, the attack will not be perceived by the victim. On the other hand, if it is not identity mapped, the corrupted driver will probably fail and the victim machine will have to be rebooted, though it would have been already permanently compromised. Nevertheless, the attacker can enforce the usage of identity mapping for the corrupted device using ACPI Reserved Memory Region Reporting Structures.

Conclusion

The IOMMU has been included several years ago in Intel processor architectures to provide better protection against low-level attacks. While this mechanism has proved to be efficient to cope with several I/O attacks, this paper shows that it can be bypassed by exploiting a design weakness in its configuration by both firmwares and the Linux IOMMU driver. The corresponding vulnerability is discussed in this paper, a proof of concept and an experiment illustrating its possible exploitation are also presented. The attack explored makes it possible for malicious peripherals to make read and write accesses in the main memory and to bypass the protection mechanisms embedded in the IOMMU. We are currently studying other operating systems (such as Windows, BSD systems) in order to check whether this vulnerability is only related to Linux kernel or not. In the same way, we also plan to investigate the boot process when the Intel TXT is activated to check that this technology does not suffer from similar weaknesses.

Endnote

¹ Basically, Linux creates two types of memory domains. In the first type of domain, the I/O Virtual Addresses (IOVAs) used by the peripheral and configured by the driver are different than the physical ones (IOPAs), translated by the IOMMU. The second type defines IOVAs as identical as IOPAs and is called identity mapping.

Abbreviations

ARP: Address Resolution Protocol; BME: Bus Master Enable; CPU: Central processing unit; DMA: Direct Memory Access; DMAR: DMA Remapping; (D)RAM: (Dynamic) Random-Access Memory; FPGA: Field Programmable Gate Array; EPT: Extended Page Tables; GCMR: Global Command Register; I/O: Input/output; IOMMU: Input-Output Memory Management Unit; IOTLB: Input-Output Translation Lookaside Buffer; LM32: LatticeMico32; L1/L2: x86 cache levels; MMU: Memory Management Unit; PCI: Peripheral Component Interconnect; PCIe: PCI Express; PCIe-EP: PCIe Endpoint; POSIX: Portable Operating System Interface (unix); RTAR: Root Table Address Register; SOC: System On Chip; TE: Translation enable; TXT: Trusted Execution Technology; USB: Universal Serial Bus

Acknowledgements

Not applicable.

Funding

This work is funded by the French laboratory *Centre National de la Recherche Scientifique* (CNRS).

Availability of data and materials

Not applicable.

Authors' contributions

This paper results from a collaborative work. The order of the authors reflects the importance of the contribution of each author concerning the identification of the IOMMU vulnerability and the design of the experimental scenario to validate its possible exploitation. All authors read and approved the final manuscript.

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Received: 12 June 2017 Accepted: 6 December 2017

Published online: 09 January 2018

References

- Morgan B, Alata E, Nicomette V, Kaënche M (2016) Bypassing IOMMU protection against i/o attacks. In: 2016 Seventh Latin-American Symposium on Dependable Computing (LADC). pp 145–150. doi:10.1109/LADC.2016.31
- Intel Corporation Intel® virtualization technology for directed I/O. <http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf>. Accessed 14 Dec 2017
- Dornseif M (2004) Owned by an iPod: Firewire/1394 issues. In: PacSec. <https://pacsec.jp/psj04/psj04-dornseif-e.ppt>. Accessed 26 Dec 2017
- Becher M, Dornseif M, Klein CN (2005) FireWire: all your memory are belong to us. In: CanSecWest. <https://cansecwest.com/core05/2005-firewire-cansecwest.pdf>. Accessed 26 Dec 2017
- Maynor D (2005) DMA: skeleton key of computing and selected soap box rants. In: CanSecWest. <https://cansecwest.com/core05/DMA.ppt>. Accessed 26 Dec 2017
- Aumaitre D, Devine C (2011) Subverting windows 7 x64 kernel with dma attacks. sogeti esec lab (July 2010). <http://esec-lab.sogeti.com/static/publications/10-hitbamsterdam-dmaattacks.pdf>. Accessed 14 Dec 2017
- Stewin P, Bystrov L (2013) Understanding DMA Malware. In: Flegel U, Markatos E, Robertson W (eds). Detection of Intrusions and Malware, and Vulnerability Assessment: 9th International Conference, DIMVA 2012, Heraklion, Crete, Greece, July 26–27, 2012, Revised Selected Papers. Springer Berlin Heidelberg, Berlin. pp 21–41. https://doi.org/10.1007/978-3-642-37300-8_2

8. Dufloy L, Perez YA, Morin B (2011) What if you can't trust your network card? In: Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection. Springer-Verlag, Berlin. pp 378–397. http://dx.doi.org/10.1007/978-3-642-23644-0_20. doi:10.1007/978-3-642-23644-0_20
9. Sang FL, Lacombe E, Nicomette V, Deswarte Y (2010) Exploiting an IOMMU vulnerability. In: 2010 5th International Conference on Malicious and Unwanted Software. IEEE. pp 7–14. doi:10.1109/MALWARE.2010.5665798
10. Markuze A, Morrison A, Tsafirir D (2016) True IOMMU protection from DMA attacks: when copy is faster than zero copy. *ACM SIGOPS Oper Syst Rev* 50(2):249–262
11. Wojtczuk R, Rutkowska J (2011) Following the white rabbit: software attacks against intel VT-d technology. I.T.L: <http://www.invisiblethingslab.com/resources/2011/Software%20Attacks%20on%20Intel%20VT-d.pdf>. Accessed 14 Dec 2017
12. Frisk U macOS FileVault2 password retrieval. <http://blog.frizk.net/2016/12/filevault-password-retrieval.html>. Accessed 14 Dec 2017
13. Morgan B, Averlant G, Alata E, Nicomette V (2016) Bypassing DMA remapping with DMA. In: Symposium sur la Sécurité des Technologies de l'Information et des Communications. p 9. https://www.sstic.org/2016/presentation/dma_bypass_with_dma/
14. Coreboot Mailing List [coreboot] DMA protection? [AMD-Vi]. <https://mail.coreboot.org/pipermail/coreboot/2016-November/082497.html>. Accessed 14 Dec 2017
15. Bourdeauducq S Milkymist system on chip. <https://github.com/m-labs/milkymist>. Accessed 14 Dec 2017
16. Bourdeauducq S Milkymist one board. https://events.ccc.de/camp/2011/Fahrplan/attachments/1874_mm_ccc2011.pdf. Accessed 14 Dec 2017
17. Morgan B, Alata E, Nicomette V, Kaaniche M, Averlant G (2015) Design and implementation of a hardware assisted security architecture for software integrity monitoring. In: Dependable Computing (PRDC), 2015 IEEE 21st Pacific Rim International Symposium On. pp 189–198. doi:10.1109/PRDC.2015.46
18. TianoCore. <http://www.tianocore.org/>. Accessed 14 Dec 2017
19. Intel Corporation Desktop 4th Gen Intel® Core™ Processor Family: Datasheet, vol. 2. <http://www.intel.com/content/www/us/en/processors/core/4th-gen-core-family-desktop-vol-2-datasheet.html>. Accessed 14 Dec 2017

Submit your manuscript to a SpringerOpen[®] journal and benefit from:

- ▶ Convenient online submission
- ▶ Rigorous peer review
- ▶ Open access: articles freely available online
- ▶ High visibility within the field
- ▶ Retaining the copyright to your article

Submit your next manuscript at ▶ springeropen.com
