Bypassing DMA Remapping with DMA

Benoît Morgan, Guillaume Averlant,
Éric Alata, Vincent Nicomette
prenom.nom@laas.fr

LAAS-CNRS, 7 av. du Colonel Roche, 31400 Toulouse
INSA Toulouse, 135 av. de Rangueil, 31400 Toulouse
Université Toulouse III - Paul Sabatier, 118 route de Narbonne, 31400 Toulouse

Résumé Afin d'illustrer les problèmes de configuration des mécanismes matériels d'isolation, nous proposons une attaque d'IOMMU. L'attaque mise en œuvre dans cet article met en avant une erreur de conception au sein du firmware et du driver d'IOMMU Intel pour linux. L'exploitation de cette vulnérabilité est réalisée uniquement à l'aide d'un périphérique PCI Express développé sur FPGA, des spécifications matérielles Intel et de la lecture du code source de linux.

1 Introduction

Les systèmes informatique deviennent de plus en plus complexes, du point de vue du logiciel comme du matériel. Il en est de même pour les périphériques qui étendent les fonctionnalités des architectures. Perez et al. démontrent [5] que ces périphériques peuvent être vulnérables et exploités pour mener des attaques par entrées / sorties donnant *in fine* un contrôle total sur la machine cible. Pour se prémunir contre ce type d'attaques, Intel a proposé d'intégrer un nouveau composant, appelé IOMMU. Ce composant, et plus précisément sa fonctionnalité de *DMA Remapping* (DMAR), permet de virtualiser l'espace mémoire des périphériques et ainsi y appliquer des contrôles d'accès. Un système peut accueillir plusieurs IOMMU et par conséquent plusieurs unités de DMAR. Pour mettre en œuvre leur service de traduction, les IOMMU s'appuient sur le matériel et des règles définies par des structures de données arborescentes placées en mémoire principale. Leur initialisation est réalisée conjointement par l'OS et le processeur à la manière des MMU des cœurs *x86*. Les IOMMU sont donc des systèmes complexes qui pourraient aussi être vulnérables de part leur conception, à d'éventuelles erreurs d'implémentation matérielle mais aussi des erreurs de configuration. Cet article présente une attaque par les entrées / sorties, en présence d'IOMMU, qui profite d'une faiblesse durant l'initialisation de cette IOMMU.

En section 2, nous commençons par présenter l'architecture matérielle sur laquelle nous avons réalisé l'attaque et apportons les connaissances requises sur les IOMMU et le DMAR. En section 3, nous continuons en décrivant les vulnérabilités historiques qui ont touché les IOMMU et les contremesures qui existent aujourd'hui. En section 4, nous présentons en détails une attaque nouvelle à notre connaissance, mettant en cause le driver d'IOMMU Intel linux. Nous terminons cet article en donnant quelques contremesures.

2 Architecture

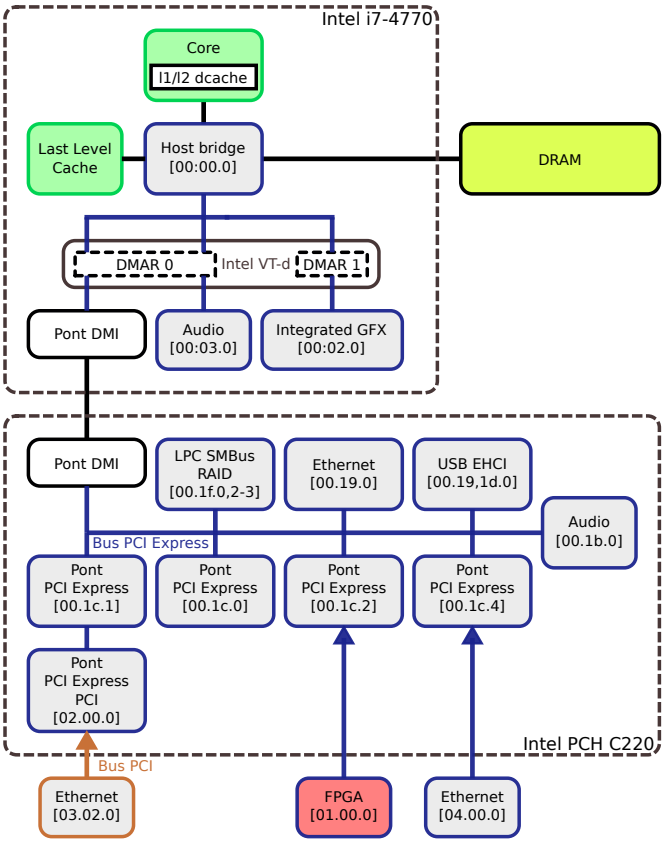

Figure 1. Vue logique du système

Afin d'illustrer une partie des menaces actuelles concernant les IOMMU, nous avons fait le choix de travailler sur une machine d'expérimentation grand public assez récente. Elle est équipée d'un processeur Intel i7-4770 de 4$^{\text{ième}}$ génération, associé au chipset Intel PCH C220 (figure 1). Afin de mieux illustrer nos propos, nous lui avons connecté plusieurs périphériques

via les slots d'extensions disponibles : un contrôleur Ethernet Broadcom PCI Express ; un second contrôleur Ethernet PCI VIA ; et enfin un prototype de périphérique PCI Express malveillant développé sur FPGA [3], [4]. Cette plateforme supporte les extensions matérielles de virtualisation des entrées / sorties Intel VT-d et notamment sa fonctionnalité de *Direct Memory Access Remapping* (DMAR). Le DMAR est mis en œuvre via des unités matérielles appellées *input / output MMU* (IOMMU) ou par abus de langage DMAR. Notre plateforme contient deux IOMMU que nous appelons respectivement DMAR0 et DMAR1, conformément à la terminologie utilisée dans les drivers linux.

Pour comprendre l'attaque présentée dans ce papier, il est nécessaire de saisir le fonctionnement du bus matériel PCI Express et du DMAR.

2.1 DMA avec le bus PCI Express

Attardons nous à présent sur la spécification des accès mémoire avec le bus PCI Express. Sur ce bus, les périphériques sont identifiés au moyen de 3 valeurs : `bus:dev:fun`.

Le standard PCI Express spécifie un type de message par opération à effectuer. En ce qui concerne l'accès à la mémoire, il existe un message pour la lecture et un message pour l'écriture. Un message de lecture valide émis sur le bus implique une réponse, appelée *completion*, du périphérique dont l'espace mémoire est interrogé. Les messages d'écriture contiennent trois informations essentielles : l'adresse mémoire de destination ; l'identifiant PCI `bus:dev:fun` source ou *requester id* ; et évidemment la donnée. Nous pouvons donc en déduire qu'il existe deux modes de routage des messages PCI Express. Les requêtes mémoire de lecture et d'écriture seront donc routées avec l'adresse de destination alors que la *completion* d'une lecture le sera par le *requester id*. Il est donc indispensable pour un périphérique de connaître son *requester id* avant d'émettre des messages nécessitant une réponse.

Les périphériques, ou *end points*, connaîssent leur *requester-id* grâce aux messages de configuration qui peuvent uniquement être émis par le *host bridge*. Les requêtes et réponses de configurations sont adressées uniquement par identifiant PCI. À chaque fois qu'un *end point* reçoit ce type de message, il doit considérer l'identifiant PCI destination comme étant son nouveau *requester id*.

De par le fait qu'un périphérique connaisse son *requester id* assez tôt dans le processus de démarrage, par exemple lorsque le *firmware* scanne les périphériques présents sur le bus pour la première fois, il est donc rapidement capable d'émettre des requêtes de lecture et d'écriture.

Enfin, notons que la spécification PCI Express permet au processeur de contrôler les émissions de messages par les éléments du bus (*end point* ou *bridges*) dans le sens montant (*upstream*) grâce au bit *Bus Master Enable* (`BME`) du registre `command` de configuration PCI. En particulier, les *bridges* ayant ce bit désactivé ne remontent pas les requêtes des périphériques *downstream*. Par contre, cette configuration est uniquement une indication pour les *end points* et ne constitue en aucun cas un moyen de réellement empêcher une émission malveillante.

2.2 Notions essentielles sur le DMA Remapping

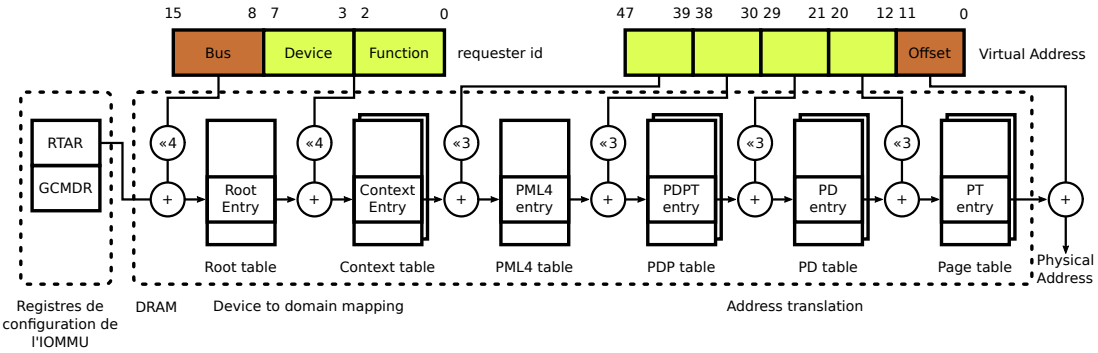

Figure 2. DMA Remapping avec 48 bits d'adressage virtuel et en pages de 4ko

Pour mieux apréhender les attaques présentées dans cet article, il est nécessaire d'introduire le fonctionnement des IOMMU ainsi que leurs mécanismes de configuration.

Les IOMMU virtualisent l'espace mémoire des périphériques. Les adresses utilisées lors des accès mémoire des *end points* seront donc traduits par les unités de DMAR avant de remonter vers le *host bridge*. Cette traduction est mise en œuvre de manière identique à celle des *Memory Management Units* (MMU) des cœurs et constitue la phase dite d'*address translation*. Le langage de traduction des adresses virtuelles vers physiques est appelé *domain*. Les IOMMU sont partagées entre plusieurs périphériques (figure 1). Afin de pouvoir appliquer un domaine de traduction différent pour chacun d'entre eux, une association préalable des périphériques vers un *domain* est nécessaire. Cette phase, nommée le *device to domain mapping*, est conceptuellement similaire à la traduction d'adresse, mais traduit plutôt des identifiants `bus:dev:fun` PCI vers le domaine adéquat. Il est possible d'appliquer un contrôle d'accès aux deux niveaux de traduction.

Du point de vue de la configuration, les deux types de traduction employés par les IOMMU s'appuyent sur des structures arborescentes placées en mémoire DRAM par le processeur au démarrage (figure 2). Pour chaque unité de DMAR existant dans la machine, est associée une page mémoire de configuration où les registres des unités de traduction sont mappés. Sont notamment présents le registre de contrôle principal, *Global Command Register* (`GCR`), permettant d'activer la traduction avec le bit *Translation Enable* (`TE`) et le registre pointant sur la racine des structures mémoire de traduction *Root Table Address Register* (`RTAR`). Les adresses des pages de configuration des IOMMU sont déterminées grâce à des *Base Address Registers* (BAR) de la page de configuration du contrôleur mémoire (`MCHBAR`). Notons que l'étape d'*address translation* peut-être désactivée (mode *pass through*) grâce au champ *translation type* d'une *context entry*.

3 Vulnérabilités matérielles identifiées

Théoriquement, une IOMMU bien configurée et exempte de vulnérabilités matérielles protège efficacement l'espace mémoire principal des requêtes de lectures / écriture malveillantes qui atteignent le *host bridge*. Certains travaux ont montré que, de par la topologie du bus et les fonctionnalités suportées par l'architecture, ce n'est pas toujours le cas [6].

Lone Sang et al. ont mis en avant une limite des IOMMU inhérente à la topologie du bus PCI Express d'une machine qu'ils ont étudiée. Profitant d'une spécificité dans la conception d'un pont PCI Express / PCI présent sur leur plateforme de test, il était possible de contourner le fonctionnement des IOMMU. En effet, ledit pont, lors de la traduction d'un cycle de lecture / écriture PCI en message PCI Express, utilisait son propre identifiant PCI comme *requester id* et non celui du périphérique réellement en cause. Un périphérique malveillant jouissait donc du domaine mémoire associé au pont lors d'accès mémoire, augmentant ainsi ses privilèges. Cette attaque est alors baptisée partage de *source-id* (*requester id*). La plateforme que nous avons étudiée dispose encore d'un pont de ce type. Il ne peut connecter par contre qu'un seul device PCI et de part cette propriété inhibe l'attaque précedente.

Les mêmes auteurs ont montré dans [2] qu'il était possible, sans IOMMU, d'acquérir en temps réel le contenu du *framebuffer* d'une carte graphique connectée au *northbridge* d'une machine cible à l'aide d'un contrôleur *Firewire* malveillant lui même connecté au *southbridge*. La lecture de la documentation du processeur actuel nous indique que la

plateforme autorisait ces accès pour des raisons fonctionnelles et de performance [1, Vol. 2, 2.15]. Il n'est aujourd'hui plus possible de lire dans ce segment mémoire, mais seulement d'y écrire.

4 Attaque d'un driver d'IOMMU

Lors de l'étude du comportement du firmware de notre machine et du driver linux pour les IOMMU Intel `drivers/iommu/intel-iommu.c`, nous avons identifié plusieurs anomalies rendant vulnérable la configuration des IOMMU pendant la phase de démarrage.

4.1 Constat

Au démarrage, les IOMMU sont désactivées. Notre *firmware* scanne l'espace de configuration des périphériques avant le chargement de linux, procurant à tous leur *requester id*. Les *root ports* où sont connectés les périphériques sous forme de carte d'extension ont le bit `BME` actif. Un périphérique peut donc émettre des requêtes de lecture / écriture valides vers des segments valides qui seront traitées par le *host bridge* avant le chargement du système opératoire et ce jusqu'à la configuration de l'IOMMU.

Comme le montre la figure 3 de (1) à (3) les structures utilisées par le DMAR sont placées en mémoire DRAM par le processeur au démarrage. Le driver IOMMU Intel constitue donc les structures des différents *domains* et du *device to domain mapping* avant de copier le pointeur de la *root table* dans le registre de configuration adéquat. Enfin, il active le DMAR avec le bit `TE` du `GCMDR` (4).

Notons qu'il existe plusieurs segments mémoire configurables où les accès DMA sont interdits. Le segment *DMA Protected Range* (DPR) spécifié par le processeur et les segment *Protected Memory Range* (PMR) spécifiés par les IOMMU [1, Vol. 2, 2.5]. Linux place les structures du DMAR en dehors de ces segments.

La documentation de notre processeur indique que la cohérence de cache des requêtes d'écriture provenant du bus DMI est respectée [1, Vol. 2, 2.15]. S'ajoute à cela le fait que le driver linux vide les lignes de cache (l1 / l2 et *Last Level Cache*, figure 1) après la modification de chaque entrée de table pour s'assurer de leur intégrité dans la DRAM au moment du démarrage de l'IOMMU. De plus, les identifiants de bus étant en général alloués aux *root ports* en commençant par 1, la *context entry* d'un périphérique est alors pointée par une *root entry* se situant plutôt au

début de la *root table*. Or, après avoir écrit et vidé la ligne de cache de notre *context entry*, le processeur doit encore renseigner le reste de la *root table* et leurs *context tables* associées. Par conséquent, nous avons donc un intervalle de temps donné entre l'écriture en mémoire de la *context entry* d'un périphérique et l'activation de l'IOMMU.

Enfin, tant que la configuration matérielle de la machine n'est pas modifée, les adresses physiques où les tables du DMAR sont mappées ne sont pas randomisées.

4.2 Exploitation

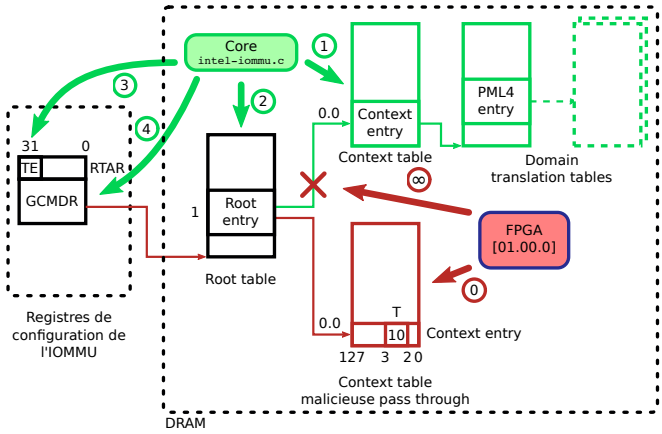

Figure 3. Mise en œuvre de l'attaque

Nous considérons le cas d'Alice et Bob travaillant dans la même entreprise et disposant tous deux de la même machine, celle que nous avons présenté dans la section 2. Alice dispose d'un exploit permettant de compromettre un périphérique PCI Express tel qu'une carte réseau comme présenté dans [5] et peut émettre des messages sur le bus PCI Express. Dans notre mise en œuvre nous avons représenté le modèle de carte réseau vulnérable par un FPGA malveillant sur lequel nous avons développé un *endpoint* PCI Express (figure 3). Alice va mettre au point l'attaque sur sa machine pour pouvoir la rejouer sur l'ordinateur de Bob. Le but suivi par Alice est de pouvoir déjouer la traduction d'adresse du DMAR pour son périphérique.

L'attaque se déroule en deux étapes. Le périphérique malveillant va d'abord créer une *context table*, dans une page mémoire libre de la DRAM, dont les entrées concernant le *device* 0 seront en mode *pass through* (1). Il va ensuite bombarder le bus d'écritures à l'adresse de la *root entry* concernant

le bus PCI correspondant au *root port* où le périphérique malveillant est connecté (symbole ∞). Grâce à l'intervalle de temps dont nous disposons (section 4.1), du fait que la cohérence de cache soit conservée et que linux ne protège pas les structures du DMAR au démarrage, nous allons pouvoir intercaler au moins une écriture valide à l'adresse désirée avant l'écriture du bit `GCR.TE` par le processeur.

Cette modification de la *root entry*, pour notre machine et dans notre version de linux (version), est durable dans le temps jusqu'au prochain redémarrage, où la même attaque fonctionnera.

Enfin, linux ne teste pas l'intégrité des pages du DMAR qu'il a configuré. Cette modification est donc transparente pour lui. Alice peut poursuivre sont attaque pour espionner l'activité de Bob en utilisant par exemple un *rootkit* DMA "pris sur étagère".

Notons que cette attaque garde le noyau intact et modifie uniquement les données utilisées par les unités de DMAR.

5 Contremesures

Pour conclure cet article nous proposons quelques contremesures. Elles peuvent être appliquées à plusieurs niveaux en fonction de l'ouverture des différents composants de la plateforme matérielle et des logiciels utilisés.

Les premières contremesures permettent de se prémunir de l'attaque présentée dans cet article, mais aussi des attaques DMA visant le code et les données placées en DRAM en général. La première contremesure consiste à utiliser le lancement sécurisé (*secure boot*) assisté par les extensions matérielles Intel TxT, à condition que la plateforme le supporte et que l'utilisateur dispose du module de code authentifié *SINIT ACM* qui est nécessaire. En effet, le *SINIT ACM* met en place un *Measured Launch Environment* (MLE) où le logiciel et les données qui y sont chargés sont protégés des attaques DMA par les PMR [1, Vol. 2, 2.5]. Le MLE peut donc configurer les structures du DMAR dans un segment contenu dans un des PMR et activer l'IOMMU sans voir ses données altérées par un périphérique malveillant.

Si l'utilisateur peut modifier le *firmware* de sa machine, celui-ci pourra aussi configurer le DPR où les PMR avant de placer son code en DRAM. Il peut ensuite donner la main aux différents logiciels chargés au fur et à mesure du démarrage dans ces segments protégés, jusqu'à ce que le système opératoire finisse par configurer le DMAR. Notre *firmware* active par défaut le bit `BME` des *root ports* aboutissant sur des slots d'extensions. Or ces périphériques externes à la machine sont potentiellement malveillants ou

peuvent posséder des vulnérabilités pouvant être exploitées. S'ils ne sont pas indispensables à la phase de démarrage, il est aussi possible de ne pas activer le bit `BME` des *root ports* où sont connectés ces périphériques afin d'inhiber leur capacité à initier des requêtes mémoire tant qu'il ne seront pas utilisés. L'OS se chargera de les réactiver s'il les considère de confiance.

Si le lancement sécurisé n'est pas disponible ou le *firmware* est non modifiable, l'utilisateur ne peut se prémunir des attaques par entrées / sorties visant à modifier le code ou les données des logiciels chargés. En outre, le système opératoire dispose des segments protégés DPR et PMR où il pourra écrire les structures du DMAR pour se prémunir contre l'attaque présentée dans cet article.

Pour conclure, il existe aujourd'hui nombre de moyens logiciels et matériels disponibles dans les architectures actuelles pour se prémunir des attaques par entrées / sorties. Force est de constater, malheureusement, que ces outils ne sont pas systématiquement mis en place par les constructeurs et les utilisateurs finaux et s'ils le sont, ne sont pas toujours accompagnés d'un lancement sécurisé. Ce qui laisse la plateforme vulnérable à certaines attaques par entrées / sorties même en présence des IOMMU.

Références

1. Intel Corporation. Mobile 4th Generation Intel ® Core TM Processor Family, Mobile Intel ® Pentium ® Processor Family, and Mobile Intel ® Celeron ® Processor Family. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/4th-gen-core-family-desktop-vol-2-datasheet.pdf`.
2. Fernand Lone Sang, Vincent Nicomette, and Yves Deswarte. Attaques dma peer-to-peer et contremesures. In *Actes du 9e Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 11)*, 2011.
3. Benoît Morgan, Éric Alata, and Vincent Nicomette. Tests d'intégrité d'hyperviseurs de machines virtuelles à distance et assisté par le matériel. In *Actes du 12ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pages 355–382, 2014.
4. Benoît Morgan, Éric Alata, Vincent Nicomette, and Guillaume Averlant. Abyme : un voyage au cœur des hyperviseurs récursifs. In *Actes du 13ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2015.
5. Yves-Alexis Perez, Loïc Duflot, Olivier Levillain, and Guillaume Valadon. Quelques éléments en matière de sécurité des cartes réseau. In *Actes du 8ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2010.
6. Fernand Lone Sang, Éric Lacombe, and Vincent Nicomette. Analyse de l'efficacité du service fourni par une i/o mmu. In *Actes du 8e Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC 10).*