UEFI et *bootkits PCI* : le danger vient d'en bas

Pierre Chifflier
prenom.nom@ssi.gouv.fr

ANSSI

Résumé UEFI est une initiative qui vise à remplacer les BIOS traditionnels des architectures PC par un code logiciel disposant de nouvelles fonctionnalités, et pouvant être utilisé sur d'autres architectures. Les spécifications UEFI apportent de nombreuses fonctionnalités, et représentent un changement radical dans la vision du BIOS comme un composant minimal. Pourtant, il est fondamental de pouvoir avoir confiance en ce composant, ainsi que dans le matériel, pour assurer l'intégrité de tout ce qui sera ensuite exécuté.

Par ailleurs, la norme PCI a défini la possibilité d'ajouter des extensions aux périphériques connectés sur le bus PCI, pour pouvoir contenir du code logiciel qui étend les fonctionnalités du BIOS. Le support des extensions PCI a été intégré dans les spécifications UEFI, avec différents modes d'exécution.

Dans cet article, nous étudions dans quelle mesure les changements apportés par UEFI peuvent modifier les moyens disponibles pour un attaquant cherchant à élever ses privilèges. Nous verrons que, malgré les efforts de simplification et de clarification des fonctions apportées par UEFI, ces fonctions peuvent être détournées par un attaquant pour écrire un code malveillant évolué. Nous détaillons ensuite comment un attaquant pourrait concevoir une extension PCI malveillante, dont le code sera automatiquement chargé pendant la séquence de démarrage UEFI et qui résulte en la compromission du système d'exploitation. Nous étudions enfin les contre-mesures possibles permettant de prévenir de tels problèmes.

Keywords. UEFI, BIOS, PCI, ACPI, TPM, Secure Boot, sécurité, informatique de confiance.

1 Introduction

UEFI (*Unified Extensible Firmware Interface*) est une spécification qui définit l'interface entre le système d'exploitation et le matériel. Elle a pour but de remplacer le BIOS traditionnel.

Initialement développée par Intel en 1998, le projet s'appelait *Intel Boot Initiative*, puis a été renommé EFI (*Extensible Firmware Interface*). Les spécifications initiales d'EFI ont été publiées en 2003 (version 1.10). Intel en a ensuite cessé le développement en 2005, pour contribuer à l'*Unified EFI Forum* [14].

À l'heure de l'écriture de cet article les spécifications courantes sont en version 2.3.1C [15].

L'un des principaux objectifs d'UEFI est de résoudre les limitations du BIOS :

- limitation à une seule architectures (x86) ;
- restriction au mode 16-bits ;
- limitation de l'espace à 1 Mo d'espace adressable ;
- problèmes de compatibilité ;
- limitation du Master Boot Record (MBR) à 4 partitions ;
- limitation de la taille des disques à 2 To ;
- absence de vérification d'intégrité.

Du point de vue de la sécurité, la séquence de démarrage doit assurer l'intégrité et la protection des éléments depuis la mise sous tension de la machine jusqu'à l'initialisation complète du noyau du système d'exploitation.

Si les évolutions proposées par UEFI semblent nécessaires, un certain nombre de points difficiles subsistent. En particulier, le socle de confiance (*TCB, Trusted Computing Base*) doit nécessairement inclure le le système d'exploitation, mais également le matériel. La TCB devrait, dans le cas idéal, être la plus réduite possible et inclure peu d'éléments, mais l'ajout de fonctionnalités évoluées tend à augmenter considérablement la surface d'attaque comme nous allons le voir pour UEFI. De plus, le matériel, qui est un élément indispensable de la TCB, dispose de plus en plus de code logiciel permettant d'étendre les fonctionnalités de la machine hôte, comme c'est le cas pour les cartes PCI.

Dans cet article, nous nous intéressons aux évolutions apportées par UEFI et à leurs utilisation par du code malveillant, en particulier dans une carte PCI. Dans la section 2, nous rappelons les points clés de la séquence de démarrage d'un PC, du chargement du code contenu dans les cartes PCI, et des évolutions apportées par UEFI sur ces éléments. La section 3 décrit les fonctionnalités UEFI, les menaces qu'elles peuvent présenter pour la sécurité, et les techniques que pourrait utiliser un attaquant afin de compromettre la sécurité du système d'exploitation. Dans la section 4, nous montrons comment un attaquant peut adapter des mécanismes utilisés pour modifier une extension PCI avec un BIOS, et les adapter pour UEFI. Ces fonctions seront illustrées par la création d'une extension PCI pour une carte vidéo permettant de contourner les protections d'un noyau Linux d'une distribution Debian et de permettre à un processus utilisateur d'élever ses privilèges. Enfin, la section 5 décrit les mécanismes existants pour se protéger contre ces attaques.

2 UEFI

UEFI a pour but de remplacer le BIOS traditionnel. Les spécifications, assez volumineuses (plus de 2 000 pages), définissent des interfaces (appelées *protocoles*) d'accès à des fonctions du matériel, et qui sont regroupées en modules.

Dans la spécification, une implantation d'UEFI s'appelle un *Firmware*, mais on trouve également souvent l'appellation BIOS. Dans la suite du document on utilisera le terme BIOS pour désigner l'ancienne spécification, et *firmware* pour décrire une implantation d'UEFI.

Ces spécifications couvrent de nombreux aspects, qui incluent une nouvelle séquence de démarrage et le support de pilotes (*drivers*). Parmi les principales différences par rapport au BIOS traditionnel, on peut noter :

- le support de différentes architectures ;
- un *design* modulaire ;
- l'uniformisation du stockage de la configuration en utilisant de la NVRAM (*Non Volatile Random Access Memory*) pour stocker les variables de configuration ;
- le démarrage en mode 64 bits (pour les architectures correspondantes) ou 32 bits, avec la pagination et le mode protégé activés sur `x86` ;
- la programmation de haut niveau (C) ;
- les exécutables sont au format PE/COFF (format *Windows*, par exemple PE32+ pour `x86-64`) ;
- des interfaces (*protocoles*) qui remplacent le mécanisme d'interruptions ;
- un nouveau protocole graphique (GOP, *Graphics Output Protocol*) pour configurer les modes graphiques ;
- un langage (EBC, *EFI Byte Code*) exécuté par une machine virtuelle. Ce langage permet d'écrire des pilotes indépendants de l'architecture ;
- de nombreuses fonctionnalités (support du réseau, IPv4/IPv6, PXE, etc.) ;
- le support de systèmes de fichiers (FAT32 obligatoire, d'autres peuvent être ajoutés) ;
- des fonctions d'authentification de l'utilisateur (*via* une base locale ou distante, et par divers mécanismes : mot de passe, empreinte digitale) ;
- une interface utilisateur (*Human Interface Infrastructure, HII*) ;
- *Secure Boot* ;

– etc.

Les fonctionnalités décrites sont très nombreuses, mais la plupart d'entre elles sont optionnelles. Il est donc assez difficile de savoir lesquelles seront présentes sur une implémentation particulière d'un constructeur pour une machine donnée.

Une différence importante avec le BIOS est que ce dernier pouvait être considéré comme un code simple faisant peu de choses et offrant quelques services sous forme d'interruptions, alors que le code UEFI est beaucoup plus proche de celui d'un système d'exploitation, tant en fonctionnalités qu'en taille.

Les changements apportés, et le nombre de nouvelles fonctionnalités, font que les images UEFI sont assez volumineuses (il n'est pas rare de trouver des mises à jour de *firmware* d'environ 10 Mo), ce qui laisse sans doute de l'avenir à la recherche dans le domaine de la sécurité.

Malgré des spécifications assez anciennes, UEFI ne commence à être largement déployé que depuis peu de temps. On le trouve cependant désormais sur la plupart des machines récentes, avec des qualités d'implémentation assez variables.

Séquence de démarrage UEFI La séquence de démarrage UEFI est composée de différentes phases (voir figure 1) :

- SEC (*Security*) : cette phase, exécutée depuis la ROM, commence l'initialisation du CPU, applique éventuellement un micro-code, et exécute la routine principale d'initialisation. Cette phase est également chargée de vérifier l'intégrité de la phase suivante ;
- PEI (*Pre EFI Initialization*) : dans cette phase, les autres CPU et la RAM sont initialisés, et les routines de préparation à la phase suivante sont exécutées. La CRTM (*Core Root of Trust for Measurement*) et les mesures associées commencent à ce stade ;
- DXE (*Driver Execution Environment*) : dans cette phase, des pilotes sont chargés en fonction des périphériques présents et de la configuration du *firmware*. Les périphériques PCI sont découverts, et les mémoires d'extension sont projetées en mémoire vive, puis leur code est exécuté ;
- BDS (*Boot Device Select*) : dans cette phase, le *bootloader* UEFI est exécuté. Il peut soit proposer un menu avec différents choix, soit démarrer directement une image exécutable ;
- TSL (*Transient System Load*) : le code UEFI passe la main au *bootloader* du système d'exploitation. Les routines UEFI sont toujours

disponibles tant que la fonction `ExitBootServices` n'est pas appelée ;
- RT (*Run Time*) : le système d'exploitation s'exécute ;
- AL (*After Life*) : cette phase, contrôlée par le firmware, ne s'exécute qu'après l'arrêt du système d'exploitation. C'est le cas, par exemple, lorsque la machine entre en état de consommation faible (ACPI S3, S4 ou S5).

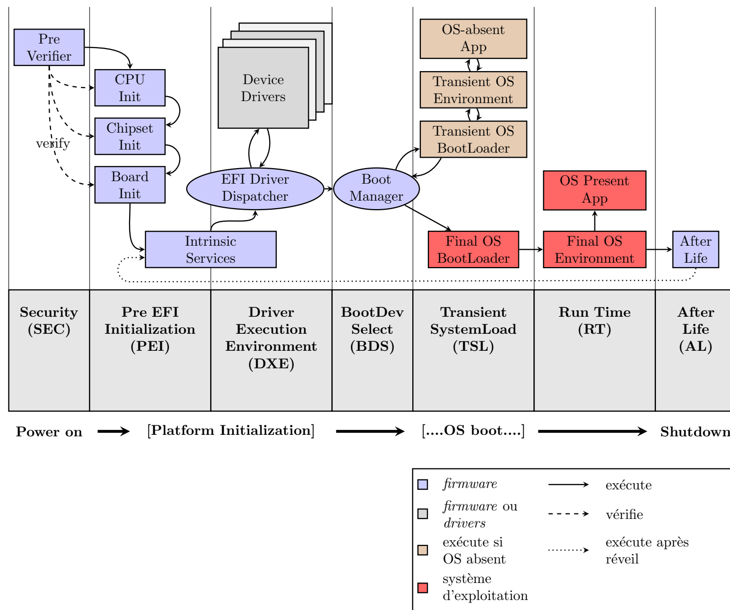

Figure 1. Séquence de démarrage UEFI.

L'ensemble des services accessibles pendant toute la phase de démarrage est appelé *Boot Services*. Ces services incluent la gestion des interruptions, un allocateur mémoire, des *timers*, des pilotes réseau, etc. Comme ils peuvent occuper beaucoup de place en mémoire, la mémoire associée à ces services est supprimée quand la fonction `ExitBootServices` est appelée par le *bootloader* ou par le système d'exploitation.

Certains services sont persistants, ils sont appelés *Runtime Services* et permettent par exemple de configurer les variables NVRAM, ou encore l'ordre de démarrage de la machine.

2.1 Secure Boot

Secure Boot est le protocole décrit dans les spécifications UEFI qui est chargé d'ajouter le support de la sécurité. Il a pour but d'assurer l'authenticité et l'intégrité de l'ensemble du code exécuté pendant la séquence de démarrage par le biais de signatures cryptographiques.

Secure Boot repose sur plusieurs points importants :

- l'ajout de modules cryptographiques comprenant des algorithmes de chiffrement, de signature et de hachage (RSA, SHA ...), et une gestion de clés avec X.509 ;
- la présence d'autorités de confiance dans l'image du *firmware* ;
- l'inclusion d'une signature cryptographique dans toutes les images (exécutables et pilotes) ;
- la vérification de la signature de tous les éléments, au moment du chargement.

Si le démarrage *Secure Boot* est activé, les fonctions UEFI de chargement de code activent la validation d'une signature, contenue dans chaque fichier. Sans la signature, ou en cas de problème de validation, l'UEFI peut bloquer le chargement de tous les programmes et de tous les pilotes, y compris les systèmes d'exploitation.

Il est également possible d'inscrire des fichiers sur des listes blanches ou des listes noires. Dans ce cas, un haché du fichier est ajouté dans la mémoire NVRAM.

Secure Boot est un module optionnel des spécifications UEFI, et n'est donc pas déployé systématiquement. On peut noter cependant qu'il est requis par les *Windows 8 Hardware Certification Requirements* [11], et que cela va probablement aboutir à une large diffusion.

Les spécifications UEFI [15] décrivent l'interface que doit proposer une implémentation de *Secure Boot*, ainsi qu'un certain nombre de procédures de gestion de clés et de formats de signatures. On peut cependant constater qu'elles restent assez floues sur les aspects pratiques de l'implémentation, et laissent beaucoup de latitude dans l'implémentation pour un éditeur de *firmware*.

C'est le cas en particulier pour le caractère obligatoire ou non des signatures sur du code provenant d'éléments extérieurs telles que des extensions PCI, qui pourraient ne pas être vérifiées si l'implémentation du *firmware* était négligente.

2.2 Rappels sur les périphériques PCI en mode BIOS conventionnel

Les extensions PCI (*PCI Expansion ROM*) sont des codes logiciels exécutables, fournis par les périphériques PCI et exécutés par le BIOS pendant la phase de démarrage (POST, *Power On Self-Test*). Ce code est généralement utilisé pour réaliser l'initialisation des périphériques, ou des opérations spécifiques : initialisation du BIOS de la carte graphique, démarrage par le réseau (PXE, *Pre-boot eXecution Environment*) ou découverte des périphériques SCSI. Ce code étant spécifique au matériel, il est présent soit sur la carte PCI, sous forme de mémoire Flash par exemple, soit dans le BIOS (généralement compressé pour économiser de la place).

Chargement de *PCI Expansion ROM* Au démarrage de la machine, le BIOS énumère les périphériques PCI et examine le registre XROMBAR (*eXpansion ROM Base Address Register*) dans le *PCI Configuration Space* de chacun des périphériques. S'il est positionné, le périphérique contient une mémoire d'extension. Le BIOS recopie alors cette mémoire en RAM à partir de l'adresse `0xC0000` pour la première mémoire (généralement celle de la carte VGA), puis toutes les mémoires des autres périphériques à la suite (avec un alignement de 2 ko).

Le format des extensions est décrit dans la figure 2. Une seule extension peut contenir plusieurs images, par exemple pour le support de différentes architectures ou de différents périphériques logiques PCI présents dans un seul élément physique.

Pendant la phase POST, le BIOS parcourt la mémoire à partir de l'adresse `0xC0000` à la recherche de marqueurs d'images (`0x55AAh`). Pour chaque image trouvée, il effectue quelques vérifications basiques :
- les identifiants PCI *Vendor ID, Device ID* doivent correspondre à ceux d'un périphérique présent sur le bus PCI. Ces identifiants sont attribués par le *PCI Special Interest Group* ;
- si l'image est au format *Plug and Play*, une somme de contrôle (sur un octet) est vérifiée.

Ces vérifications n'ont, de manière évidente, pas un objectif de sécurité, mais uniquement pour but de ne pas charger par erreur du code qui ne proviendrait pas d'une mémoire d'extension.

L'exécution du code d'extension dans un BIOS est soumise à certaines contraintes :
- le processeur est en mode réel 16 bits ;

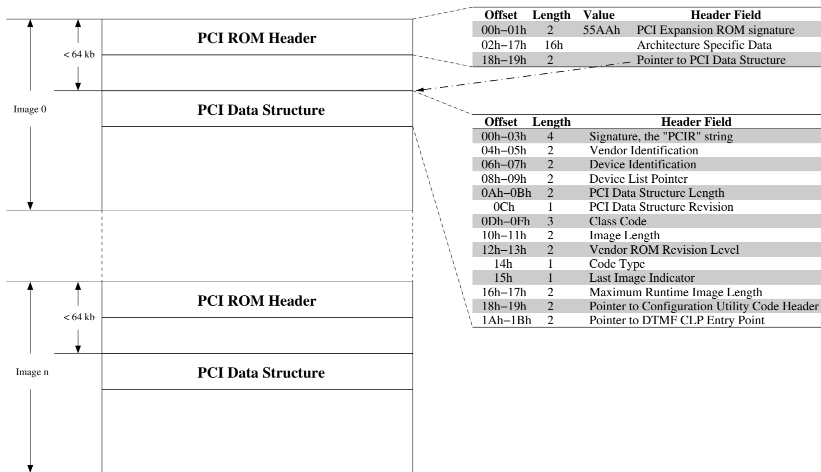

Figure 2. Format de ROM d'extension PCI pour l'architecture x86.

- la mémoire n'est pas encore entièrement initialisée, seul 1Mo est utilisable, ou jusqu'à 16 Mo en utilisant le mécanisme PMM (*POST Memory Manager*) ;
- le code doit rendre la main au BIOS en laissant les registres dans le même état qu'au départ ;
- les interruptions ne sont pas toutes accessibles ;
- le BIOS VGA n'est pas encore installé, aucun affichage n'est encore possible ;
- généralement, les composants contenant les extensions sont limités à 64 ko ou 128 ko ;
- l'extension peut contenir plusieurs images. Pour chaque image, le point d'entrée doit se situer dans les premiers 64 ko de celle ci ;
- si plusieurs images ont les mêmes identifiants, seule la première sera exécutée ;
- le code doit être entièrement indépendant de sa position mémoire ;
- le code ne doit déclencher aucune erreur ou exception non traitée.

Si le code déclenche une exception, le BIOS stoppe le démarrage et émet des signaux sonores pour diagnostiquer une erreur. Comme le démarrage est interrompu, il est évidemment impossible de reflasher la mémoire, et il faut donc trouver un autre moyen de le faire sous peine de bloquer irrémédiablement le démarrage de la machine tant que la carte fautive est présente.

2.3 *PCI Expansion ROM* : exécution en UEFI

Les extensions PCI sont chargées durant la phase DXE de la séquence de démarrage. Dans le cas d'UEFI, la norme PCI pose quelques problèmes : le code exécutable doit être du code en mode réel 16 bits, et il va mettre en place ou utiliser des interruptions. Ces deux mécanismes étant obsolètes avec UEFI, un nouveau fonctionnement est prévu dans les spécifications : un nouveau format de mémoires d'extension PCI a été défini, qui permet d'encapsuler du code UEFI. Un mode de compatibilité est également possible, pour supporter les anciens formats. Nous allons décrire les différents modes disponibles pour le support des extensions PCI.

ROM au format spécifique UEFI Un nouveau format de mémoire d'extension PCI permet, avec un marqueur spécifique `0xEF1`, d'indiquer que le format est UEFI. Dans ce cas, le code de la mémoire devra être au format PE/COFF pour l'architecture de la machine. Le format est décrit dans la figure 3.

Offset	Byte Length	Value	Description
0x00	1	0x55	ROM Signature, byte 1
0x01	1	0xAA	ROM Signature, byte 2
0x02	2	XXXX	Initialization size - size of this image in units of 512 bytes(including the header)
0x04	4	0x0EF1	Signature from EFI image header
0x08	2	XX	Subsystem value for EFI image header
0x0a	2	XX	Machine type for EFI image header
0x0c	2	XX	Compression type 0x0000 The image is uncompressed 0x0001 The image is compressed
0x0e	8	0x00	Reserved
0x16	2	XX	Offset to EFI image
0x18	2	XX	Offset to PCIR Data Structure

Figure 3. Entête d'une extension PCI au format UEFI.

L'image contient soit le code d'un *Boot Service Driver*, soit celui d'un *Runtime Service Driver*, ou encore du bytecode EBC. La création d'une image est très simple et consiste à écrire un *Boot Service Driver* puis à le convertir en utilisant l'outil `EfiRom` fourni dans le kit de développement [3], en précisant les identifiants PCI.

On peut noter que dans la figure 3, le marqueur `0xEF1` est situé en partie sur l'emplacement réservé pour indiquer le point d'entrée du code

exécutable pout les *legacy expansion ROM*. Par conséquent, si on fournit une extension au format UEFI à un ancien BIOS, celui-ci va interpréter le marqueur comme une instruction, ce qui engendre un plantage de la machine tant que la carte est présente, et nécessite une écriture de la mémoire PCI par re-programmation externe.

***Compatibility Support Module* (CSM)** Si l'UEFI est désormais assez répandu, la migration des périphériques PCI prendra du temps, et il est nécessaire de gérer la compatibilité. L'implémentation exacte est assez complexe et ne sera pas détaillée ici (plus d'informations peuvent être trouvées dans [16]). On peut cependant retenir qu'un mode de compatibilité (*CSM, Compatibility Support Module*) a été défini pour pouvoir exécuter les *legacy expansion ROM*. Une extension PCI pouvant contenir plusieurs images, on peut donc avoir, pour une seule carte, plusieurs cas possibles d'exécution pouvant se produire, suivant l'activation ou non du mode compatibilité, l'ordre de préférence choisi par le *firmware* (exécution des images UEFI en premier ou non), etc.

Extension au format UEFI + *CSM* Ce mode consiste à fournir une extension contenant deux images, une de type *legacy* et une au format UEFI. La première sera exécutée par le *CSM*, et la deuxième par le *firmware*.

Pas d'extension PCI Si le périphérique PCI est fixe le *firmware* peut directement inclure le *driver* pour gérer le matériel. L'extension PCI n'est alors plus nécessaire et peut donc être supprimée du matériel

Dans le cas d'une carte graphique pour un ordinateur portable, ce mode permet au *firmware* de démarrer intégralement en mode graphique, et de ne plus afficher le mode console du tout. On trouve par exemple ce type de configuration sur certains portables MacBook Air.

3 Utilisations malveillantes d'UEFI

Comme nous l'avons vu dans la section précédente, l'UEFI est beaucoup plus qu'une simple évolution du BIOS. Cependant, la quasi totalité des spécifications décrit l'ajout de fonctionnalités, et seul *Secure Boot* (qui est optionnel) concerne la sécurité. Les éditeurs sont donc libres sur les choix d'implémentation.

Un des avantages d'UEFI est l'ajout de nombreuses facilités de développement et d'utilisation. Cependant, ces facilités peuvent être détournées par un code malveillant pour écrire en quelques lignes un code complexe. En particulier :

- le développement se fait en C, il n'est quasiment plus nécessaire de connaître l'assembleur ! Par conséquent, la classe d'attaquants potentiels est plus grande puisque l'accès est plus simple ;
- les protocoles forment une couche d'abstraction au dessus des fonctions spécifiques du matériel. Il est donc beaucoup plus simple d'écrire un code portable ;
- la persistance est simplifiée par l'existence (obligatoire) d'une partition FAT32. Cela permet également à un code malveillant de ne plus être contraint par des problèmes d'espace disponible. Néanmoins, cela permet une détection plus simple.

Dans cette section et la suivante, nous allons montrer comment certaines de ces fonctions pourraient être utilisées pour réaliser des actions malveillantes.

3.1 Modifications des fonctions UEFI

Ce premier exemple va montrer l'immense facilité avec laquelle les fonctions UEFI peuvent être manipulées. Pour illustrer ce point, nous allons rapidement comparer les étapes initiales des *bootkits* dans le cas classique, et avec UEFI.

Avec un BIOS conventionnel, les *bootkits* doivent généralement détourner les interruptions (par exemple l'interruption `13h`) pour que le code malveillant soit appelé par le *bootloader*. Ce processus demande de recoder une interruption, qui doit également identifier la vraie cible par des signatures, car l'interruption va être appelée pour tous les accès disque. Enfin, l'ensemble doit tenir en très peu de place (généralement, le premier secteur du disque dur) et fonctionner en mode réel.

Dans le cas d'UEFI, les *Boot Services* et les *Runtime Services* UEFI sont projetés en mémoire. Ce sont des fonctions C, dont l'emplacement est identifié par deux structures, dont l'emplacement est variable (mais connu) et qui commencent chacune par un marqueur. Les pages mémoire contenant ces structures ne sont pas protégées contre l'écriture, mais cela n'aurait aucune importance puisque le code UEFI tourne avec les privilèges maximum (*ring 0*), et les programmes lancés par UEFI (exécutables ou pilotes) bénéficient des mêmes privilèges.

Il suffit donc à une application UEFI de remplacer une fonction liée au *bootloader*. La nouvelle fonction exécutera le code malveillant puis appellera la fonction d'origine. Deux services UEFI sont intéressants :

– `LoadImage` est la fonction appelée pour charger une image en mémoire. L'image peut être un pilote, une mémoire d'extension PCI, un exécutable, etc. et sera uniquement positionnée en mémoire (pas exécutée) ;
– `StartImage` est la fonction qui exécute le code d'une image chargée avec la fonction `LoadImage`.

La fonction `LoadImage` est intéressante, car elle permet de connaître le nom du fichier chargé. Si le fichier est exécutable, la fonction `StartImage` permet d'exécuter des instructions juste avant l'exécution de l'image ainsi chargée, ce qui pourrait être utilisé pour faire exécuter un code différent de celui du fichier chargé.

Le principe peut être utilisé pour intercepter n'importe quelle fonction offerte par un protocole UEFI. Il est même possible de recoder intégralement un protocole ou d'en ajouter un.

Les limitations du BIOS relatives à la taille du code utilisé, la quantité de mémoire, ou les fonctions disponibles disparaissent dans le cas d'UEFI : le processeur tourne en mode complet pour son architecture (mode 64 bits, avec mode protégé et pagination mémoire à l'identique pour l'ensemble de la mémoire), et l'exécutable est simplement un fichier sur la partition de démarrage EFI (format FAT32).

Le listing 1.1 illustre cet exemple. On pourra donc comparer ce code aux plusieurs centaines de lignes d'assembleur du *Stoned Bootkit* pour réaliser la même opération.

```
extern struct EFI_BOOT_SERVICES *gBS; // provided by UEFI

static StartImage *old_StartImage = NULL;

EFI_STATUS EFIAPI MyStartImage (
  EFI_HANDLE ImageHandle,
  UINTN      *ExitDataSize,
  CHAR16     **ExitData
  )
{
    /* do something nasty */
    // ...
    /* then really execute the image */
    return (*old_StartImage)(ImageHandle, ExitDataSize, ExitData);
}

EFI_STATUS EFIAPI UefiMain (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
```

```
  )
{
    old_StartImage = gBS->StartImage;
    gBS->StartImage = MyStartImage;
    return EFI_SUCCESS;
}
```

Listing 1.1. Exemple d'interception de chargement d'une image

Ce principe a été mis en pratique dans [1] peu de temps après la sortie de Windows 8 pour créer un *bootkit* qui désactive la vérification de signature des *drivers*, et la protection du noyau contre les modifications (*Patchguard*).

Une interception de code peut également être utilisée pour d'autres objectifs que piéger le système d'exploitation, par exemple pour tenter de récupérer le mot de passe d'un disque intégralement chiffré (*Evil Maid Attack* [12]).

Le code présenté en 1.1 fonctionne, mais est facilement détectable. En effet, pour le démarrer, il faut :

- écrire l'exécutable sur la partition EFI, en le rendant donc facilement détectable ;
- modifier les variables NVRAM du *firmware* pour démarrer sur ce code, et non plus sur le *bootloader* classique. Là encore, la modification sera facilement détectable ;
- alternativement, on pourrait modifier l'image disque d'un *bootloader* UEFI existant. La modification reste très visible.

Pour rendre le code furtif, l'attaquant peut le transformer en une extension PCI, comme nous le verrons dans la section 4.

3.2 Ajout et altération des tables ACPI

UEFI fournit différentes fonctions concernant l'ACPI :

- le protocole `AcpiTableProtocol`, qui donne accès à des fonctions de manipulation des tables ACPI ;
- des pointeurs sur les tables racines (*RSDT* ou *XSDT*), qui contiennent des pointeurs vers toutes les tables chargées en mémoire.

Ces fonctions sont normalement utilisées par les différents protocoles UEFI, qui peuvent installer des tables ACPI pendant la phase de démarrage. Cette fonctionnalité est utilisée en particulier par les cartes graphiques exploitant les spécifications ACPI récentes (version 5.0). Après le démarrage, la carte donne au système d'exploitation des pointeurs vers ses ressources dans la table *BGRT (Boot Graphics Resource Table)*, comme

le *splash screen* affiché par le *firmware*, principalement dans un objectif cosmétique.

L'ajout de tables ACPI contrôlées par l'attaquant, ou la modification de tables existantes peut également permettre de charger du code malveillant, qui sera interprété par le système d'exploitation et exécuté lorsque les événements ACPI correspondants se produiront. L'article [4] présente entre autres comment piéger un système à partir des tables ACPI.

La suppression de tables peut également conduire à la désactivation des fonctions de sécurité correspondantes, comme cela est montré dans la section 4.3.

Notons cependant que le risque n'est pas dans la possibilité de manipuler les tables ACPI (ce risque étant déjà présent dans les BIOS), mais dans l'exposition des fonctions pendant le démarrage qui simplifient ces manipulations.

3.3 Menaces relatives aux fonctions réseau

Les spécifications UEFI contiennent de nombreux protocoles et fonctionnalités réseau (présentés par ordre de regroupement dans les spécifications) :

- SNP (*Simple Network Protocol*), PXE, BIS (*Boot Integrity Services*),
- iSCSI,
- VLAN, EAP,
- ARP, DHCP,
- TCP, FTP, IPsec,
- UDP, MTFTP (client FTP pour diffusion UDP en mode *unicast* ou *multicast*),
- l'ensemble en IPv4 et en IPv6.

On notera en particulier que le démarrage d'une machine par le réseau (*PXE*) ne sera en principe plus géré par l'extension PCI de la carte réseau, mais par le *firmware* UEFI.

Ces protocoles sont optionnels, donc il est peu probable de tous les trouver dans une implémentation donnée. Il est cependant assez courant de trouver un support complet du réseau sur les *firmwares* des serveurs, pour gérer le démarrage *PXE*. Ceci, combiné avec *Secure Boot* ou *Boot Integrity Services*, présente même une fonctionnalité très intéressante de démarrage *PXE* en diffusant des codes exécutables signés, donc avec la possibilité de vérifier l'intégrité d'images provenant du réseau avant toute exécution.

La généralisation de l'ajout des fonctions réseau peut présenter plusieurs risques. Elle ajoute un volume de code important, et augmente proportionnellement la surface d'attaque si le code est utilisé. Mais elle offre également à un attaquant de nombreux moyens de communication vers d'autres éléments, ce qui pourrait être utilisé pour diffuser du code malveillant, communiquer avec un serveur de contrôle et de commande, etc.

IPsec Un protocole UEFI particulier est défini pour IPsec, pour ajouter des fonctionnalités de sécurité aux communication réseau : authenticité, intégrité et confidentialité. Le support UEFI couvre un support générique d'IPsec et permet l'utilisation de l'ensemble des fonctionnalités d'IPsec, allant d'AH (*Authentication Headers*) et ESP (*Encapsulated Security Payloads*) à IKE (*Internet Key Exchange*) et IKEv2 pour l'échange de clés.

L'implémentation d'une pile IPsec sans défauts est une tâche délicate, et on notera par exemple que l'implémentation de référence du kit de développement [3] a été réécrite intégralement, ce qui laisse craindre d'avoir des failles de sécurité inconnues dans l'implémentation.

3.4 Autres risques liées aux fonctions UEFI

Il reste d'autre éléments dont l'implémentation ou l'utilisation pourrait présenter des risques, et qui ne seront pas détaillés dans ce document. La liste des risques présentée ici n'est pas exhaustive :

- un disque dur peut être partitionné soit au format classique, soit au format de table de partitions spécifique UEFI (*GPT, GUID Partition Table*). Il est cependant possible d'avoir les deux configurations simultanément, avec des partitions différentes. Cela peut conduire à des interprétations différentes suivant les *firmwares*, et à des problèmes fonctionnels ou de sécurité ;
- l'accès à la SMM (*System Management Mode*) est rendu possible par un protocole (`EFI_SMM_ACCESS_PROTOCOL`), ce qui donne à un attaquant des outils pour manipuler cette mémoire ;
- les spécifications définissent des *Runtime drivers* qui peuvent être utilisés par le système d'exploitation. Cependant, celui-ci ne dispose d'aucun moyen de vérifier l'intégrité de ces *drivers* ;
- l'utilisation d'EBC et de la compression (dont le format est spécifique à UEFI) donne à un code malveillant des possibilités de dissimulation ou d'obscurcissement ;
- certaines variables en NVRAM peuvent être modifiées depuis le système d'exploitation, parfois sans vérifications. Ces variables peuvent

être utilisées pour attaquer le code UEFI lui-même, par exemple avec un débordement de tampon.

4 UEFI : Scénarios d'attaque PCI

Cette section décrit plusieurs utilisations possibles des fonctionnalités décrites dans les sections précédentes pour altérer la sécurité du système d'exploitation en utilisant comme vecteur d'attaque une *PCI Expansion ROM*.

Nous commençons par rappeler les méthodes existantes pour utiliser des extensions PCI pour compromettre la sécurité d'un BIOS conventionnel, puis nous décrivons les différences dans le cas d'un *firmware* UEFI. Nous décrivons ensuite des implémentations permettant de contourner la protection des allocations mémoire entre périphériques (*IOMMU*), ou encore d'élever les privilèges d'un processus utilisateur avec un système d'exploitation Linux.

L'attaquant considéré dans cet exemple doit avoir réussi à piéger une extension PCI. Il doit au préalable avoir pu lire cette mémoire et la modifier avec la méthode qui est décrite, avant de l'écrire sur la carte PCI. Ces opérations peuvent être réalisées par exemple dans un des cas suivants :

- si la machine est éteinte et que la séquence de démarrage n'est pas verrouillée dans le *firmware*, l'attaquant peut démarrer sur un périphérique USB et modifier l'extension ;
- si l'attaquant à un accès physique à une machine, il peut extraire la carte PCI pour la modifier sur une autre machine sous son contrôle ;
- si l'attaquant peut faire exécuter du code arbitraire sur le système d'exploitation de la machine, il peut écrire l'extension soit en utilisant les outils fournis par les constructeurs, soit directement par des commandes de bas niveau (entrées/sorties). Ceci demande cependant des privilèges administrateur (la manière d'obtenir ces privilèges n'est pas décrite ici).

Bien que ces attaques soient relativement proches du piégeage matériel, il est important de noter que la troisième méthode ci-dessus est entièrement logicielle et peut donc potentiellement être réalisée à distance.

Ce qui est décrit ici a été implémenté pour des architectures `x86-64`, mais reste applicable à d'autres architectures (`x86`, `ARM` par exemple).

4.1 Rappels sur les extensions PCI

La sécurité des périphériques PCI et des mémoires d'extension pour les BIOS conventionnels a déjà été le sujet de recherches par le passé, par exemple dans [7] ou [13].

La méthodologie utilisée dans ces articles est de modifier un périphérique PCI tel qu'une carte réseau pour déposer un code malveillant PCI. Le choix d'utiliser une carte réseau simplifie les choses car le code d'extension peut généralement être écrasé sans danger (il ne réalise pas d'opération indispensable au démarrage), et aussi parce que le BIOS dispose de fonctions permettant d'enregistrer une extension PCI qui est conforme à la norme *Plug and Play* [5] pour faire exécuter le code à la fin de l'initialisation du BIOS. L'exécution n'est cependant pas automatique, elle n'est déclenchée que si la machine démarre sur un *boot PXE*.

Une des premières fonctions d'un code malveillant PCI est de déposer une charge utile, et de s'assurer qu'elle sera ensuite appelée par le BIOS. En effet, pendant l'exécution du code PCI, les contraintes rendent la programmation difficile. Il faut donc trouver un moyen de faire rappeler le code avant, ou pendant le démarrage du système d'exploitation.

Pour cela, le code détourne généralement une ou plusieurs interruptions, par exemples les interruptions `10h` (modes vidéo) ou `13h` (accès au disque dur). Cependant, ces interruptions vont être appelées de nombreuses fois pendant le démarrage, par exemple pour la lecture des fichiers. Il faut donc trouver un moyen d'identifier le fichier appelant souhaité (par exemple `NTLDR` pour Windows).

Une fois le fichier identifié, il est possible d'interrompre le chargement du *bootloader* (ou du système d'exploitation) pour exécuter l'étape suivante du code malveillant. La suite du traitement n'est pas spécifique aux extensions PCI et consiste à modifier l'image du système d'exploitation. Ce processus est décrit et mis en œuvre par exemple dans le *Stoned Bootkit* [9] pour de multiples versions de Windows.

Récemment, une démonstration d'un BIOS VGA malveillant et persistant basé sur ces principes a été faite lors de la conférence EkoParty 2012 [2]. La méthodologie employée est la même, mais l'utilisation d'une carte graphique donne la garantie que le code malveillant sera exécuté a chaque démarrage.

Les avantages de l'utilisation d'une extension PCI sont les suivants :
- démarrage automatique : l'extension est chargée à chaque démarrage (suivant le type de périphérique) ;
- persistance : le code malveillant n'est pas sur le disque dur, il résistera donc même après un effacement du disque ou une réinstallation ;

- discrétion : le code malveillant n'est jamais présent sur le disque dur et aucun fichier du système d'exploitation ou du *bootloader* n'a besoin d'être modifié ;
- manque de vérifications : les mises à jour du BIOS peuvent être signées et vérifiées, alors que les mises à jour des extensions PCI ne le sont généralement pas.

Un code malveillant reste détectable en lisant la mémoire de la carte PCI. Ceci dit, la plupart des antivirus ne font que des analyses du disque dur ou des nouveaux processus exécutés, et assez rarement des périphériques PCI ou des zones mémoires précédemment allouées pour le BIOS.

4.2 Objectif ROM

Les étapes de création d'une extension PCI au format UEFI sont les suivantes :

1. transformer l'exécutable en un *Boot Service* ;
2. convertir ce *Boot Service* en une mémoire d'extension PCI à l'aide de l'outil `EfiRom` ;
3. trouver une mémoire PCI existante, avec suffisamment de place ;
4. patcher cette mémoire pour garder l'ancienne image et ajouter la mémoire malveillante dans une deuxième image d'extension ;
5. re-flasher la mémoire d'extension.

Les deux premières étapes sont triviales et décrites dans le guide de développement de *drivers* UEFI [8].

Pour les suivantes, quelques difficultés se posent. Dans le cadre de cette étude, le choix de la carte PCI s'est porté sur la carte graphique, car c'est celle donc le code est systématiquement exécuté au démarrage.

Cependant, les principales difficultés sont de ne pas altérer les fonctionnalités existantes tout en essayant de faire tenir le code dans l'espace disponible, généralement très réduit.

La première idée pourrait être de remplacer totalement le contenu de l'extension PCI, mais cela n'est pas possible car sur les machines testées, le BIOS VGA n'est pas installé et la machine refuse alors de démarrer.

Une solution consiste à créer une image au format hybride (*legacy + UEFI*) comme expliqué dans la section 2.3. Pour cela il faut extraire l'extension PCI existante d'un périphérique PCI, et que cette extension soit au format *legacy* uniquement. La modification consiste en l'ajout de l'extension PCI au format UEFI, en recopiant les mêmes identifiants PCI,

à la fin de l'image mémoire extraite. Cette étape peut facilement être automatisée. Il faut ensuite écrire l'extension modifiée sur le périphérique PCI.

Notre mémoire d'extension PCI devra contenir une *legacy expansion ROM* pour installer le BIOS VGA, et une deuxième image qui contiendra le code UEFI. Ce code sera automatiquement chargé par le *firmware* pendant la phase DXE de la séquence de démarrage.

4.3 Désactivation de l'IOMMU

Le scénario suivant consiste à intercepter la séquence de démarrage UEFI pour supprimer une table ACPI avant le démarrage du système d'exploitation. Ici, l'attaquant cherche à abaisser le niveau de sécurité en supprimant la fonctionnalité de protection des accès mémoire par les périphériques (IOMMU), qui serait utilisé par un gestionnaire de machines virtuelles utilisant les extensions matérielles. Cet exemple a été testé pour le noyau Linux, et pour Xen.

Le protocole `AcpiTableProtocol` fournit deux fonctions, `InstallAcpiTable` et `UninstallAcpiTable`, qui permettent de manipuler des tables ACPI.

Ces fonctions sont normalement prévues pour fonctionner ensembles : l'installation d'une table renvoie un objet de type `HANDLE`. Cet objet est nécessaire pour pouvoir supprimer une table ACPI (voir code 1.2).

```
EFI_STATUS MyUnloadAcpiTable(UINTN mTableKey)
{
    EFI_STATUS Status;
    EFI_ACPI_TABLE_PROTOCOL       *AcpiTableProtocol;

    Status = gBS->LocateProtocol (&gEfiAcpiTableProtocolGuid,
        NULL,
        (VOID **) &AcpiTableProtocol
        );
    if (EFI_ERROR (Status))
        return Status;

    return AcpiTableProtocol->UninstallAcpiTable (
            AcpiTableProtocol,
            mTableKey
            );
}
```

Listing 1.2. Suppression d'une table ACPI

Cependant, UEFI ne fournit pas la correspondance entre le `HANDLE` qui sert d'identifiant et la table ACPI. En pratique la plupart des implémentations fonctionnent de la même manière, et l'identifiant de la table

est en réalité un index dans un tableau contenant les tables ACPI qui est incrémenté à chaque ajout de table.

Il est donc possible de récupérer la liste des tables ACPI chargée en lisant la variable de configuration système UEFI identifiée par `EfiAcpiTableGuid`. En itérant sur les tables pointées par la RSDT et en faisant l'hypothèse que chaque pointeur est un incrément de 1, il suffit d'identifier la table désirée.

La table responsable (entre autres fonctions) de la gestion de l'IOMMU est la table `DMAR` (*DMA Remapping*). Cette table est utilisée au chargement du noyau Linux par la fonction `intel_iommu_init`.

En cherchant la table ACPI identifiée par le marqueur `DMAR`, il est trivial de la supprimer. Au démarrage, on constate que l'IOMMU est désactivée (et toutes les protections associées avec).

```
[0.000000] Intel-IOMMU: enabled
..
[0.002814] ACPI: Core revision 20120913
[0.012422] dmar: Host address width 36
[0.012424] dmar: DRHD base: 0x000000fed90000 flags: 0x0
[0.012431] dmar: IOMMU 0: reg_base_addr fed90000 ver 1:0 cap
    c0000020e60262 ecap f0101a
[0.012432] dmar: DRHD base: 0x000000fed91000 flags: 0x1
[0.012437] dmar: IOMMU 1: reg_base_addr fed91000 ver 1:0 cap
    c9008020660262 ecap f0105a
[0.012438] dmar: RMRR base: 0x000000788d5000 end: 0x000000788ebfff
[0.012440] dmar: RMRR base: 0x00000079800000 end: 0x0000007d9fffff
...
[0.012937] Enabled IRQ remapping in x2apic mode
[0.013400] ..TIMER: vector=0x30 apic1=0 pin1=2 apic2=-1 pin2=-1
```

Listing 1.3. Journal de démarrage Linux avec IOMMU

```
[0.000000] Intel-IOMMU: enabled
..
[0.002390] Freeing SMP alternatives: 16k freed
[0.002839] ACPI: Core revision 20120913
[0.012976] ..TIMER: vector=0x30 apic1=0 pin1=2 apic2=-1 pin2=-1
```

Listing 1.4. Journal de démarrage Linux **sans** IOMMU

Le *listing* 1.3 montre le démarrage de Linux en activant l'IOMMU sur la ligne de démarrage. Le *listing* 1.4 montre un démarrage avec la même ligne de commande, mais en désactivant l'IOMMU par la méthode présentée. On peut remarquer que dans les deux cas, l'IOMMU est marquée comme activée car le paramètre `intel_iommu=on` est passé au noyau.

Si elle n'a pas pu être configurée par le noyau, l'IOMMU est silencieusement désactivée. Sous Linux, cette désactivation n'empêche absolument pas le démarrage du système ou l'utilisation de machines virtuelles.

Il reste possible de le détecter en analysant les journaux du démarrage pour chercher l'absence des messages relatifs à la configuration de l'IOMMU.

La désactivation de l'IOMMU peut cependant être bloquante pour le système dans deux cas : pour Linux, si *trustedboot* est utilisé, et pour Xen, si l'option `iommu=required` est passée au démarrage. Dans chacun de ces cas, le système refusera de démarrer si l'IOMMU est désactivée.

4.4 Modification de l'image du noyau

Cette section présente un exemple complet d'utilisation d'extension PCI, illustré par la figure 4. Une carte graphique dont l'extension PCI a été modifiée est présente sur la machine. Cette carte contient l'ancien BIOS VGA (avant modification) dans l'extension au format *legacy*, et le code malveillant est ajouté dans une extension au format UEFI. Nous allons présenter les différentes étapes du démarrage de la machine et décrire comment le code malveillant s'insère dans cette exécution.

Exécution initiale Après le démarrage de la machine, le *firmware* UEFI s'exécute. Pendant la phase DXE, les extensions PCI au format *legacy* sont chargées par le module CSM. Le BIOS VGA faisant partie de ces extensions, il est exécuté normalement pendant l'étape (1).

Le *firmware* exécute ensuite le chargement des extensions au format UEFI, dont celle présente sur la carte graphique. Le code d'initialisation va intercepter les fonctions UEFI `ExitBootServices` et `StartImage` et faire l'initialisation du code malveillant (étape (2)). Ce code est également en charge de recopier le code logiciel de toutes les autres étapes dans des zones mémoires persistantes, comme décrit ci-dessous.

Persistence mémoire après le démarrage La mémoire utilisée par UEFI est normalement libérée lors de l'appel à la fonction `ExitBootServices`. Il faut donc trouver une solution pour que le code exécutable ne soit pas réalloué par le système d'exploitation.

Avec un BIOS, la tâche aurait pu être complexe, car il aurait fallu soit écraser une zone persistante, soit allouer de la mémoire et la marquer dans la table `e820` comme réservée. Là encore, l'UEFI fournit diverses fonctions

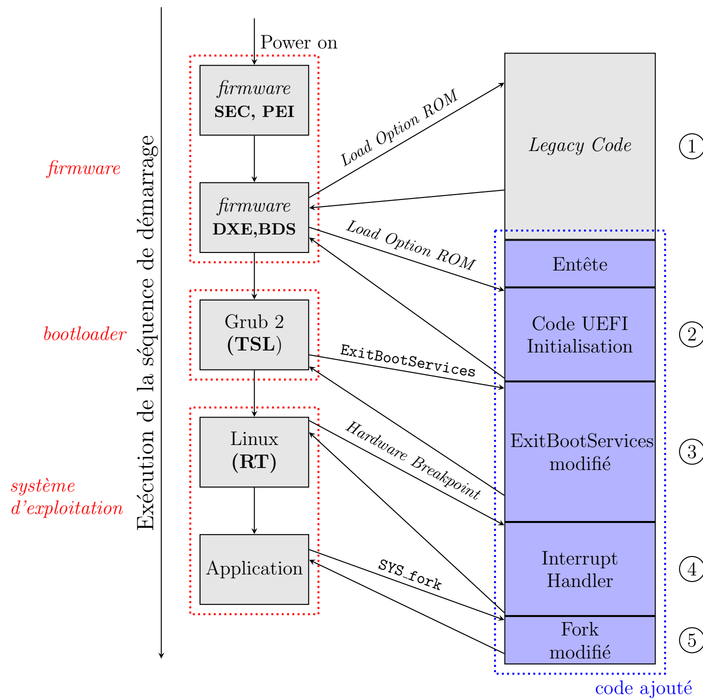

Figure 4. *Étapes du chargement de l'extension PCI modifiée*

d'allocation mémoire comme `AllocatePages` qui permet de réserver de la mémoire qui, suivant le type demandé, sera persistante après le démarrage.

Nous allons maintenant décrire comment, à partir de cet instant, la compromission du noyau du système d'exploitation n'est que l'affaire de quelques étapes et que, si ces étapes ne sont pas toutes spécifiques à UEFI, UEFI va énormément simplifier leur réalisation.

bootloader On traitera ici du cas de Grub 2, qui est le *bootloader* généralement utilisé pour démarrer le noyau Linux avec une distribution classique. On notera cependant que les principes exposés par la suite restent valables pour d'autres *bootloaders* UEFI, y compris pour le démarrage d'une machine Windows (`bootmgfw.efi`).

Pour un BIOS conventionnel, le *bootloader* réalise certaines étapes qui rendent compliquée une interception, en particulier le passage en mode protégé 32 bits, la réinitialisation des tables d'interruption etc. Un code malveillant doit donc trouver un moyen de se faire appeler, souvent en

manipulant les vecteurs d'interruption. Nous allons voir que dans le cas d'UEFI, la méthode est beaucoup plus simple.

Un *bootloader* UEFI est chargé de préparer le démarrage du système d'exploitation en recopiant (avec une décompression possible) l'image du noyau, et en effectuant toutes les routines d'initialisation requises. Pour le démarrage d'un noyau 64 bits, l'adressage n'est pas modifié par Grub, tout comme l'état général de la machine (tables d'interruptions, etc.) reste inchangé après le chargement du noyau. En effet, en mode UEFI, Grub s'appuie sur les *Boot Services* fournis par UEFI. La dernière étape avant de passer la main au noyau est d'appeler la routine UEFI `ExitBootServices` qui va libérer la mémoire utilisée par les *Boot Services*.

Cette routine ayant précédemment été détournée pendant l'étape (2), c'est donc la routine malveillante (3) qui sera appelée. Cette routine doit analyser la mémoire allouée par le *bootloader* pour trouver les structures mémoires internes, et en particulier la position du noyau.

La routine (3) doit ensuite trouver un moyen de positionner du code qui sera appelé pendant le démarrage du système. Ici encore, la mémoire allouée précédemment par l'UEFI sera tout simplement conservée par le noyau, car elle est marquée comme zone réservée.

Pendant le démarrage du système en mode UEFI sur une architecture `x86-64`, le noyau commence par décompresser son image en mémoire. Il met ensuite en place une pagination et un mode d'adressage virtuel. Il positionne enfin les structures internes du processeur telles que les tables d'interruptions et continue le démarrage.

L'ensemble de ces opérations pose problème à un code malveillant, car l'image mémoire devant être modifiée n'est pas projetée en mémoire au moment du *bootloader*.

Pour contourner ces difficultés, le code peut positionner un point d'arrêt matériel (*hardware breakpoint*), en utilisant les registres `DR0` à `DR7` des processeurs. Le code peut spécifier une adresse virtuelle qui, lorsqu'elle sera atteinte par le pointeur d'instruction, va déclencher une interruption #DB.

Le code détermine donc l'adresse de la table d'interruptions courante, et vient écraser l'interruption 1 avec un descripteur qui indique l'adresse d'une fonction, là aussi contrôlée par l'attaquant. Il doit ensuite déterminer l'adresse virtuelle sur laquelle le point d'arrêt sera positionné.

Le code malveillant doit également faire attention à fonctionner correctement avec des adresses virtuelles (qui ne sont pas encore activées à cet instant, mais qui le seront par le noyau). Dans le cas d'un noyau Linux standard, les adresses virtuelles sont connues et relativement constantes.

Il détermine ensuite l'adresse d'un code qui sera exécuté par le noyau, mais avant que l'IDT ne soit écrasée. En effet, si l'IDT est écrasée par le noyau, le point d'arrêt sera atteint et la fonction d'interruption sera appelée, mais il s'agira de la fonction du noyau et pas celle contrôlée par l'attaquant. Il faut donc trouver une adresse qui est située après l'activation de la pagination, mais avant que l'IDT ne soit écrasée. L'adresse suivant immédiatement la décompression du noyau en mémoire a été utilisée : bien qu'elle ne soit pas à un emplacement constant, en pratique sa position reste fixe pour une version compilée du noyau.

Démarrage du noyau et modification d'un appel système Une fois le point d'arrêt matériel positionné dans l'étape (3) atteint, le vecteur d'interruption 1 est exécuté. Ce vecteur contient la charge finale (4) du code malveillant.

Dans notre cas, nous avons choisi pour illustrer une exécution malveillante d'intercepter l'appel système `fork` pour le corrompre. Bien que son nom semble indiquer une opération extrêmement courante, il n'est en réalité quasiment jamais exécuté sur un système Linux actuel (depuis la glibc 2.3.3), car l'implémentation de la fonction utilisateur `fork` utilise en réalité l'appel système `clone`. L'appel système `fork` présente aussi l'avantage de ne pas prendre d'arguments, donc il n'y aura pas d'éléments à nettoyer avant de rendre la main. L'appel système peut donc être détourné sans risque de corruption du reste du système.

Dans l'exemple choisi, l'appel est modifié pour donner les privilèges administrateur (identifiant effectif d'utilisateur 0) au programme qui l'appelle, sans aucune condition. Le code va donc réaliser l'équivalent du code C listé en 1.5. Pour plus d'informations sur la construction de l'appel système modifié, voir [10].

```
void fork (void)
{
    // require Linux >= 2.6.29
    commit_creds ( prepare_kernel_cred (0) );
}
```

Listing 1.5. Appel système `fork` modifié

La difficulté ici réside dans le fait de connaître les adresses mémoire des fonctions du noyau. Dans le cas du noyau Linux, aucune randomisation n'est activée : il suffit de regarder le fichier `System.map` correspondant à la version du noyau pour connaître de manière déterministe l'emplacement des fonctions. Dans le cas d'une distribution Linux, la connaissance d'une

seule installation donne les emplacements pour toutes les autres installations partageant la même architecture CPU. Une fois les adresses connues, l'appel système pourra être écrasé par un code (5) appelant les fonctions avant de retourner, comme par exemple en listing 1.6. La charge utile de ce code, une fois compilée, tient en 17 octets.

```
xor   %rdi , %rdi                  ; %rdi := 0
mov   $0xffffffff81063fc3 , %rax   ; prepare_kernel_cred
call  %rax
mov   $0xffffffff81063d0a , %rax   ; commit_creds
call  %rax
ret
```

Listing 1.6. Appel système `fork` assembleur

Une fois l'appel système modifié, le code malveillant n'a plus de raison de rester persistant : la zone mémoire réservée par le *firmware* peut donc être écrasée, et le point d'arrêt supprimé.

Pendant le fonctionnement du système, tout programme qui appellera l'appel système `fork` se verra donc attribuer les privilèges administrateur !

Mise en œuvre et preuve de concept En pratique, les scénarios présentés ont été mis en œuvre sur différents systèmes : une machine virtuelle (*qemu*), deux portables, et deux postes fixes.

La machine virtuelle a permis de créer un prototype pour le code logiciel à écrire dans l'extension PCI, sans risquer de bloquer de machine physique pendant le temps de développement.

Sur les machines possédant une *IOMMU*, toutes ont pu être compromises par l'attaque consistant à supprimer cette fonctionnalité.

L'élévation de privilèges d'un processus a été réalisée sur l'ensemble des plateformes de test, en utilisant la même version du noyau Linux mais en installant une distribution Debian GNU/Linux en version *Wheezy* (*testing*) sur chaque machine.

Limitations La limitation la plus importante constatée est l'espace disponible dans les extensions PCI.

Même si l'utilisation de code C peut générer un code plus volumineux, la standardisation des protocoles dans UEFI permet de compenser cet aspect car le code malveillant peut s'appuyer sur un ensemble de fonctions assez riches, comparable à ceux d'un petit système d'exploitation avec une bibliothèque standard.

Dans la preuve de concept réalisée, il restait sur la carte de test (Ati FirePro V3800) moins de 20ko utilisables, alors qu'une mémoire d'extension UEFI vierge créée par les outils UEFI fait 17 ko ! Pour s'adapter proprement, nous aurions pu essayer de réduire la place utilisée par notre code en optimisant, ou en en écrivant certaines parties directement en assembleur. La solution trouvée en pratique ne fait pas vraiment preuve de finesse mais marche assez bien : la place supplémentaire a été trouvée en écrasant les polices de caractères embarquées dans la mémoire PCI de la carte vidéo. De manière surprenante, aucun dommage collatéral n'a été constaté (ni aucun artefact sur l'affichage).

4.5 Conséquences pour l'intégrité d'un système

Une extension PCI malveillante peut exécuter du code arbitraire avant l'exécution du *bootloader* et du système d'exploitation. Cette exécution permet de contourner toutes les protections.

La mise en pratique de ces scénarios a confirmé le fait que les dangers présentés pour les BIOS conventionnels sont toujours présents.

UEFI vient ajouter à ces dangers une grande portabilité, car le code malveillant devient indépendant du matériel sur lequel il fonctionne grâce aux services et aux protocoles offerts par UEFI (ce qui n'était pas le cas avec le BIOS).

Cette portabilité pourrait même être étendue à d'autres architectures, en écrivant l'extension PCI avec du *bytecode* EBC. La seconde partie du code resterait cependant spécifique à l'architecture et au système d'exploitation.

5 Contre-mesures

Les codes malveillants peuvent être introduits de deux manières différentes : soit pendant la période d'utilisation du système, soit avant. Il est donc possible de regrouper les contre-mesures possibles en fonction des cas.

Dans le premier cas, il est possible de connaître un état sain du système, qui sera pris comme un état de référence. Il est alors nécessaire d'effectuer des mesures de l'ensemble des éléments concernés, et d'utiliser des méthodes telles qu'un TPM ou *Secure Boot* pour vérifier l'intégrité de la chaîne de confiance pendant la séquence de démarrage, et être capable d'identifier un élément dont la mesure est modifiée.

Dans le second cas, le code est introduit avant la mise en fonction du système, par exemple si le matériel est piégé au préalable. Il n'est donc

pas possible de détecter un changement sur une mesure existante, et il faut appliquer des méthodes de détection de comportement suspicieux.

Dans tous les cas, un durcissement de la sécurité des éléments de la séquence de démarrage peut être appliqué, pour renforcer les protections existantes et rendre plus complexe l'exécution d'un code malveillant.

L'ensemble de ces contre-mesures est détaillé dans les sections qui suivent.

5.1 TPM et *Trusted Boot*

Le TPM (*Trusted Platform Module*) est un composant matériel, déployé dans de nombreux ordinateurs, et qui a pour but de mettre en place une informatique de confiance.

Dans le cas de la chaîne statique de confiance (*SRTM, Static Root of Trust for Measurement*), le *firmware* va fournir au TPM des valeurs (sous formes de condensats) qui seront utilisées pour mesurer tous les éléments clés de la séquence de démarrage. Les éléments mesurés sont décrits dans les spécifications fournies par le TCG [6].

Pendant le démarrage, tous les éléments chargés seront mesurés dans les différents registres attribués (PCR, *Platform Configuration Register*). Les mémoires d'extension PCI seront mesurées et placées dans les PCR 0 (pour les mémoires PCI internes au *firmware* par exemple) ou 2 (mémoires, applications et drivers chargés par les mémoires PCI).

Pour pouvoir détecter une modification d'une mémoire PCI, il faut au préalable avoir mesuré l'état d'un système dans une situation de confiance, et avoir scellé des données essentielles au démarrage (par exemple une clé de chiffrement du disque). Les PCR utilisés pour sceller les données doivent inclure tous les PCR contenant des mesures du *firmware*.

La modification (ou un ajout ou suppression) d'une des mémoires d'extension PCI va résulter en une altération des mesures des PCR correspondants. Les données scellées ne pourront pas être extraites, et le démarrage sera interrompu.

Pour que la protection soit effective, il faut respecter certains prérequis : que le TPM soit présent et activé (il est désactivé par défaut), mais aussi que le *firmware* réalise effectivement les mesures. Certains *firmwares*, pour des raisons qui ne sont pas détaillées par les constructeurs, ne semblent pas mesurer exhaustivement l'ensemble des mémoires d'extension.

Enfin, si cette solution répond aux problèmes cités et reste la protection la plus simple à déployer actuellement, il reste qu'elle peut devenir assez contraignante pour la gestion des mises à jour. En effet, toute modification du matériel ou de la configuration du *firmware* va résulter en un

changement des mesures et donc une machine qui refusera de démarrer. Il faut donc trouver une solution (qui peut être organisationnelle) pour résoudre ces problèmes.

5.2 *Secure Boot*

Secure Boot a été décrit dans la section 2.1. D'après les spécifications, il permettrait de résoudre le problème : les mémoires PCI contenant des pilotes (soit *Boot Services*, soit *Runtime Services*), la signature de ces éléments est normalement couverte par les vérifications de *Secure Boot*. Les mémoires seraient donc signées par un certificat issu d'une autorité de confiance.

Cependant, de nombreux problèmes viennent compliquer la mise en œuvre de cette solution :

- *Secure Boot* est optionnel, et il n'est pas présent sur toutes les machines UEFI ;
- même s'il est présent, il faudrait que tous les périphériques PCI disposent de mémoires au format UEFI, ce qui n'est aujourd'hui que très rarement le cas ;
- aucun périphérique au format *legacy* ne doit être présent, car il ne pourrait pas fonctionner. Ce point ne sera résolu que lorsque la transition vers UEFI sera complète, ce qui peut prendre beaucoup de temps ;
- il faudrait que tous les constructeurs / éditeurs fournissent des mémoires PCI signées par des autorités de confiance contenues dans le *firmware* UEFI. L'expérience a montré que la gestion des autorités pouvait poser de nombreux problèmes et s'avérer délicate. Par exemple, si peu d'autorités sont embarquées, il est nécessaire que les mémoires soient signées par des autorités reconnues par les constructeurs. Si un grand nombre d'autorités sont embarquées, alors le risque de compromission augmente, ainsi que la taille de l'image du *firmware*, qui est contrainte par la place disponible.

Secure Boot est donc une bonne solution, mais uniquement dans le cas d'un matériel limité et maîtrisé. Il pourrait même être envisageable, dans le cas où des mémoires au format UEFI seraient présentes, de les extraire des périphériques pour les re-signer (en remplaçant une éventuelle signature existante) pour les écrire ensuite sur les périphériques. Cette solution reste très théorique, puisqu'il n'est aujourd'hui pas acquis de pouvoir spécifier ses propres autorités dans la configuration du *firmware*, ni de pouvoir écrire dans tous les cas de nouvelles mémoires d'extension PCI car la signature prend de la place, qui n'est pas toujours disponible.

Secure Boot reste toutefois une solution intéressante car basée sur des mécanismes cryptographiques connus, et comme étant le standard prévu pour la gestion de la sécurité.

5.3 Détection

S'il est assez difficile pour le système d'exploitation de se défendre contre ce type d'attaques, la détection reste cependant possible car le code des mémoires d'extension PCI reste projeté en mémoire après le démarrage.

Cependant, le code restant en mémoire peut différer de celui ayant été exécuté. En effet, le code d'initialisation est souvent supprimé de la mémoire après son exécution pour sauver un peu d'espace. Il est donc important de vérifier le code complet, et pas seulement celui présent après le démarrage.

Le seul moyen efficace de le faire serait de *dumper* la mémoire directement depuis la carte, ce qui est assez simple soit depuis un accès direct à la mémoire Flash, soit depuis un système tiers non compromis (qui n'exécute pas les codes des mémoires d'extension, par exemple).

Cette détection resterait cependant sujette aux limites inhérentes à la détection de code malveillant, telles que la difficulté de distinguer de manière fiable un code malveillant d'un code bénin. De plus, le domaine n'est pas encore vraiment exploré, par exemple il n'existe aujourd'hui pas d'annuaire des *PCI Expansion ROM* connues.

5.4 Interdire les modifications

Puisque les cartes PCI sont des éléments qui ne sont remis à jour qu'assez rarement, il semblerait logique de vouloir interdire les modifications sur ces cartes.

En pratique, ça reste assez difficile, car la plupart des cartes PCI :
- n'ont pas de cavalier pour interdire l'écriture de la Flash
- n'imposent pas de vérification de signature pour une mise à jour

Cependant, l'OS peut bloquer l'accès en écriture à la mémoire Flash. Il est possible, par exemple, d'interdire l'écriture dans certaines zones du *PCI Configuration Space*, qui est le mécanisme généralement utilisé par les utilitaires des constructeurs pour écrire la nouvelle image.

Cette protection n'est cependant pas totale, car il reste possible d'envoyer directement les commandes d'entrée/sortie, sans passer par le *PCI Configuration Space*.

5.5 Rester en BIOS legacy ?

Devant ces menaces, il pourrait être tentant de rester en mode BIOS, et de retarder la transition à UEFI autant que possible. En effet, le BIOS conventionnel est un peu mieux connu et plus simple, et il existe déjà des options permettant de mettre en place une configuration sécurisée.

Cependant, cette solution serait fragile et non pérenne, en particulier parce que l'UEFI est désormais le standard pour toutes les machines récentes. De plus, la gestion de la compatibilité donne lieu à des résultats parfois surprenant : des tests menés sur un petit échantillon de machines ont montré que toutes ont démarrées sur un système UEFI, même quand la configuration indiquait de ne démarrer qu'en mode BIOS conventionnel.

5.6 Durcissement

Options du *firmware* Parmi les options de sécurisation classiques appliquées à un BIOS, l'une d'entre elles est de verrouiller les options de configuration, et en particulier celles concernant la séquence de démarrage.

Si le durcissement de la configuration du *firmware* est certainement une étape nécessaire, il est important de noter que ces options n'ont pas d'influence (sauf désactivation) sur les périphériques PCI et les mémoires d'extension.

Très peu de *firmwares* proposent de limiter l'exécution des mémoires d'extension, et aucune mesure d'intégrité n'est proposée. De plus, dans le cas des mémoires VGA, il est impossible d'en empêcher l'exécution sans perdre de fonctionnalités, puisque le BIOS VGA est contenu dans cette mémoire.

***bootloader* et système d'exploitation** Le principe de défense en profondeur pourrait être appliqué pour utiliser des protections classiques (randomisation des adresses mémoires, protection contre les débordements mémoire, marquage de zones mémoires en lecture seule). Ces protections pourraient être appliquées à chacun des éléments de la séquence de démarrage (UEFI, *bootloader* et noyau du système d'exploitation) pour ajouter des protections supplémentaires.

Comme il a déjà été mentionné dans la section 4.5, on peut remarquer que les systèmes d'exploitation accordent une grande confiance à la séquence de démarrage, et ne font aucune vérification de l'état dans lequel est la machine au moment ou le système d'exploitation démarre.

Pourtant, certaines mesures pourraient améliorer la sécurité du système d'exploitation en rendant beaucoup plus complexe l'exploitation depuis le démarrage. Par exemple, le noyau et les autres acteurs du démarrage pourraient remettre la machine dans un état connu et maîtrisé, en supprimant toute configuration existante telle que les registres de *debug* DR0 à DR7, les vecteurs d'interruption, etc.

Un élément important réside aussi dans le fait que les adresses mémoires utilisées par le *bootloader* et par le noyau sont relativement stables, voire même constantes.

L'utilisation systématique d'aléa dans le positionnement des fonctions et des structures pourrait également rendre plus difficile la tâche de l'attaquant. Cela ne bloquerait pas l'attaque, mais augmenterait le niveau nécessaire à sa réalisation.

6 Conclusion

Jusqu'à présent, l'écriture des extensions PCI était rendue difficile par sa complexité, et par la forte adhérence au matériel. Nous avons montré que ces paramètres sont maintenant modifiés par l'arrivée d'UEFI, qui simplifie le travail d'écriture d'un *firmware* et rend également celui d'écriture d'un code malveillant plus simple. Les mesures de protection, si elles existent, restent cependant soumises à certaines contraintes d'utilisation qui les rendent parfois difficiles à utiliser.

Les mémoires d'extension PCI peuvent être assez dangereuses, de par la difficulté d'y accéder, et la relative furtivité du code qui sera exécuté.

Ces éléments montrent que, de plus en plus, la possibilité d'écrire des *firmwares* évolués avec de nombreuses fonctionnalités rend mince la frontière entre les couches basses de l'informatique (BIOS, *firmware* et accès au matériel) et le système d'exploitation. Ces couches faisant nécessairement partie du socle de confiance du système, il devient essentiel de disposer de moyens de vérifier leur intégrité, sans quoi toutes les protections du système pourraient être mises en défaut.

Références

1. Andrea Allievi. UEFI technology : say hello to the Windows 8 bootkit. `http://www.itsec.it/2012/09/18/uefi-technology-say-hello-to-the-windows-8-bootkit/`, 2012.
2. Nicolás Economou, Diego Juarez. Ekoparty : VGA Persistent Rootkit, 2012.
3. TianoCore, Intel. EDKII : Efi Development Kit version 2. `http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EDK2`.

4. Loïc Duflot, Olivier Levillain. ACPI et routine de traitement de la SMI. https://www.sstic.org/media/SSTIC2009/SSTIC-actes/ACPI_et_routine_de_traitement_de_la_SMI_des_limite/SSTIC2009-Article-ACPI_et_routine_de_traitement_de_la_SMI_des_limites_a_l_informatique_de_confiance-duflot.pdf, 2009.
5. Compaq, Phoenix, Intel. Plug and Play BIOS specifications version 1.0A, 1994.
6. Trusted Computing Group. TCG EFI Platform Specification version 1.20, 2006.
7. John Heasman. Implementing and Detecting a PCI Rootkit. http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf, 2007.
8. Intel Corporation. UEFI Driver Writer's Guide for UEFI 2.3.1, 2012.
9. Peter Kleissner. Stoned Bootkit. http://www.stoned-vienna.com/.
10. Keegan McAllister. Writing Kernel Exploits. http://ugcs.net/~keegan/talks/kernel-exploit/talk.pdf, 2012.
11. Microsoft. Windows 8 Hardware Certification Requirements. http://msdn.microsoft.com/en-us/library/windows/hardware/hh748200.aspx, 2012.
12. Joanna Rutkowska. Evil Maid goes after TrueCrypt! http://theinvisiblethings.blogspot.fr/2009/10/evil-maid-goes-after-truecrypt.html, 2009.
13. Darmawan Salihun. Malicious Code Execution in PCI Expansion ROM. http://resources.infosecinstitute.com/pci-expansion-rom/, 2012.
14. Unified EFI Forum. UEFI. http://www.uefi.org.
15. Unified EFI Forum. Unified Extensible Firmware Interface Specification version 2.3.1, errata C. http://www.uefi.org/specs, 2012.
16. V. Zimmer, R. Hale, and M. Rothman. *Beyond BIOS*. Intel Press, 2006.