Une approche de virtualisation assistée par le matériel pour protéger l'espace noyau d'actions malveillantes

Éric Lacombe[1,2], Vincent Nicomette[1,2], Yves Deswarte[1,2]
eric.lacombe(@){laas.fr,security-labs.org},
vincent.nicomette(@)laas.fr, yves.deswarte(@)laas.fr

[1] CNRS ; LAAS ; 7 Avenue du Colonel Roche, F-31077 Toulouse, France
[2] University of Toulouse ; UPS, INSA, INP, ISAE ; LAAS ; F-31077 Toulouse, France

Résumé Cet article est consacré à la protection de la sécurité d'un noyau de système d'exploitation. Nous proposons une caractérisation des actions malveillantes pouvant corrompre ce noyau, basée sur la façon dont elles accèdent au noyau tout d'abord et dont elles corrompent l'espace d'adressage ensuite. Puis nous discutons des mesures de sécurité permettant de contrer de telles attaques. Enfin, nous exposons notre approche basée sur un hyperviseur matériel, qui est partiellement implémenté dans notre démonstrateur *Hytux*. Celui-ci est inspiré de *bluepill* [29], un logiciel malveillant qui s'installe en tant qu'hyperviseur léger – sur un CPU possédant la technique de virtualisation matérielle – et installe un système opératoire Microsoft Windows dans une machine virtuelle. À la différence de *bluepill*, Hytux est un hyperviseur léger qui implémente des mécanismes de protection dans un mode plus privilégié que le noyau Linux.

Mots-clés : actions noyau malveillantes, sécurité noyau, virtualisation matérielle

1 Introduction

1.1 Contexte et problématique

Tout le monde reconnaît maintenant que l'utilisation des ordinateurs (en particulier au travers du réseau Internet) est devenue essentielle dans la vie quotidienne. Chacun d'entre nous utilise un ordinateur pour travailler, pour échanger des informations, pour faire des achats, etc. Malheureusement, les activités malveillantes visant les ordinateurs se multiplient régulièrement et essaient d'exploiter des vulnérabilités qui sont de plus en plus nombreuses en raison de la complexité toujours croissante des logiciels d'aujourd'hui. Ces activités malveillantes peuvent s'attaquer aux applications installées sur le système mais aussi au système d'exploitation lui-même et en particulier, à son noyau. Corrompre le noyau d'un système d'exploitation est particulièrement intéressant du point de vue d'un attaquant parce que cela signifie corrompre potentiellement tous les logiciels qui s'exécutent au-dessus de ce noyau. En particulier, les *rootkits*

en mode noyau [24] sont des logiciels malveillants (appelés aussi « maliciels ») qui accomplissent ce genre de corruption. Pour opérer, ces maliciels ont besoin de failles de sécurité dans le noyau de façon à exécuter leur code malveillant dans le noyau. Ces failles de sécurité sont particulièrement répandues dans les pilotes de périphériques[3].

Comme la corruption du noyau d'un système d'exploitation provoque la corruption de l'exécution de tous les logiciels installés au-dessus de ce noyau, ce dernier doit être fortement protégé. Mais, protéger un noyau de façon efficace par des mécanismes de sécurité est particulièrement délicat car il est extrêmement difficile de rendre ces mécanismes incontournables. En ce qui concerne le logiciel qui s'exécute en espace utilisateur, il est possible d'implémenter des mécanismes de sécurité efficaces dans la mesure où ces mécanismes sont implémentés dans le noyau et s'exécutent donc dans un mode plus privilégié que les entités qu'ils protègent. Pour protéger efficacement un noyau contre des exécutions de code malveillant, il faut donc le faire depuis un mode plus privilégié que le noyau lui-même et de façon incontournable (par le noyau lui-même, l'espace utilisateur ou les périphériques).

Dans cet article, nous présentons un mécanisme qui satisfait à ces contraintes grâce aux extensions de virtualisation matérielle. Nous n'abordons pas, dans cet article, comment le système doit démarrer de façon à ce que l'hyperviseur prenne le contrôle d'un système initialement sain. Cependant cette phase peut être réalisée grâce à des techniques de type *Static Root of Trust Measurement (SRTM)*, qui vérifie le BIOS, puis le master boot record, puis le noyau, ou de techniques de type *Dynamic Root of Trust Measurement (DRTM)*, présentes dans Intel TXT [21] (*Trusted Execution Technology*) ou AMD SVM (*Secure Virtual Machine*).

1.2 Plan

La suite de cet article est organisée comme suit. Tout d'abord nous rappelons dans la section 2 quelques concepts techniques requis pour la compréhension de cet article. Nous établissons dans la section 3 une caractérisation des actions malveillantes qui peuvent provoquer une perte d'intégrité d'un noyau de système d'exploitation. La section 4 présente les mécanismes de sécurité existants qui peuvent être déployés pour partiellement couvrir les différentes classes d'actions malveillantes. La section 5 est consacrée à la présentation de notre approche, baptisée Hytux, qui implémente des mécanismes de sécurité dans un hyperviseur léger, assisté par le matériel, de façon

[3] Les raisons principales pour cela sont que : (1) le noyau est constitué en majeure partie de ces pilotes ; (2) les règles qui régulent l'intégration des pilotes dans le noyau Linux par exemple, en ce qui concerne la qualité du logiciel, sont moins sévères que celles qui sont appliquées dans les autres sous-systèmes du noyau.

à protéger le noyau Linux d'actions malveillantes. Enfin, la section 6 propose une conclusion et des perspectives.

2 Quelques considérations techniques

Ces considérations techniques concernent l'architecture IA32[4] [19,20] qui est très répandue. Bien que chaque architecture ait des caractéristiques propres, toutes les architectures partagent quelques concepts : la gestion de la mémoire, les niveaux de privilèges du processeur, la communication entre les différentes parties matérielles et le logiciel (en général au travers d'interruptions), etc.

2.1 L'architecture IA32

La gestion de la mémoire sur une architecture IA32 est réalisée par une unité de segmentation (laquelle doit être activée obligatoirement) et une unité de pagination (dont l'activation est optionnelle) (cf. Fig. 1). Contrairement à l'unité de segmentation, celle de pagination est présente sur tous les types d'architectures. Comme Linux est un noyau multi-plateformes, l'unité de segmentation est seulement utilisée dans son mode basique (le mode *flat*[5]). Ceci permet de se passer facilement de cette unité pour n'utiliser que le seul mécanisme de pagination (cf. Fig. 2). Néanmoins, nous proposons ici une rapide explication de la façon dont l'unité de segmentation est utilisée. Le noyau doit établir des segments en écrivant leurs descriptions en mémoire dans une table, appelée GDT (*Global Descriptor Table*). Ensuite, il charge l'adresse de cette table dans le registre `gdtr` de façon à indiquer au processeur où est cette table. Le processeur a besoin d'un segment de code (CS) depuis lequel il récupère les instructions à exécuter, un segment de données (DS) et un segment de pile (SS).

L'architecture IA32 est conçue à l'aide d'une structure en 4 anneaux, dont chacun d'eux représente un mode spécifique d'exécution. Un niveau de privilège est associé à chaque mode. L'anneau le plus privilégié est l'anneau 0 (le mode d'exécution du noyau) tandis que le niveau le moins privilégié est l'anneau 3 qui est reservé aux applications en mode utilisateur.

La communication entre le noyau et l'espace utilisateur (c'est-à-dire le basculement depuis l'anneau 0 vers l'anneau 3 et inversement) peut être établie par différents événements. Parmi ceux-ci, les interruptions sont les plus utilisées. Elles sont constituées d'exceptions (interruptions levées par le processeur quand se produisent une

[4] Nous ne couvrons pas le mode IA32e dans cette section car cela aurait rendu trop compliqué les explications concernant la gestion de la mémoire.

[5] Un seul segment de mémoire est utilisé et associé aux adresses linéaires 0 jusque $2^{32} - 1$ dans le mode 32 bits ou $2^{48} - 1$ dans le mode 64 bits.

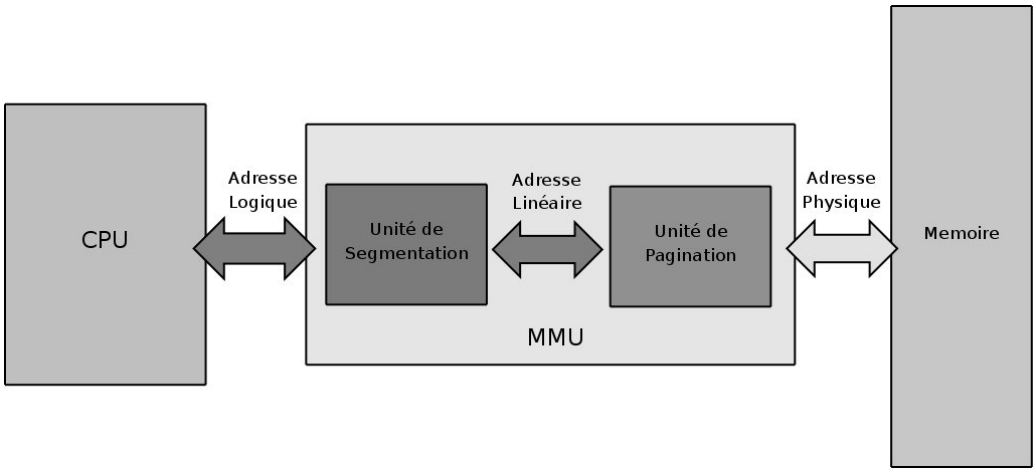

Fig. 1. MMU, unités de segmentation et de pagination

division par zéro, une faute de page, etc.), des interruptions matérielles (provoquées par des périphériques, comme l'appui sur une touche du clavier par exemple) et enfin des interruptions logicielles (interruptions qui sont levées par du code, lorsqu'une application en mode utilisateur exécute un appel système).

Sur l'architecture IA32, ces interruptions sont numérotées de 0 à 255. Chacune d'entre elles est associée à une routine si celle-ci a été positionnée par le noyau. Cette routine est une fonction qui est exécutée lorsque l'interruption est levée. Toutes ces routines sont accessibles depuis une table spécifique en mémoire : l'IDT (*Interrupt Descriptor Table*). Le noyau initialise cette table et charge son adresse dans le processeur via l'intruction `lidt`.

Une interruption matérielle ou une exception du processeur stoppe l'exécution en mode noyau ou utilisateur et exécute la routine correspondante. Les interruptions matérielles sont asynchrones alors que les exceptions du processeur sont synchrones. Le noyau gère l'interruption ou l'exception et rend la main à l'espace utilisateur. Cependant, avant cela, le noyau peut décider de se charger de tâches plus urgentes. En particulier, dans le cas de Linux, l'ordonnanceur vérifie s'il existe un processus de plus haute priorité qui a besoin d'être exécuté.

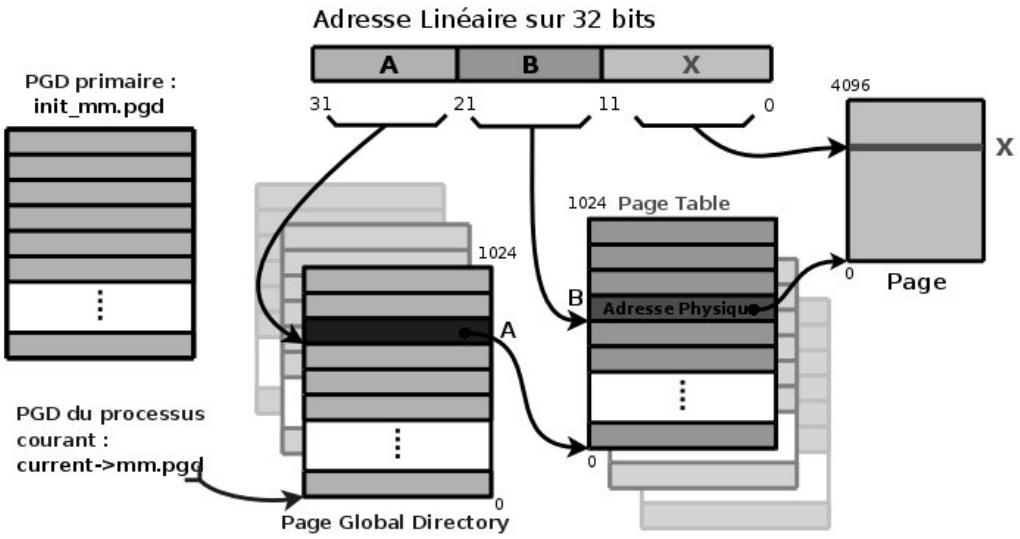

Fig. 2. Mécanisme de pagination

2.2 L'espace d'adressage du noyau Linux

La figure 3 représente une vue simplifiée de l'espace d'adressage du noyau. Concentrons-nous maintenant sur les attributs de page qui permettent à l'unité de pagination de gérer des droits accès sur chaque page. Ces attributs sont écrits (dans le mode de pagination à 4 Ko) dans les 12 bits de poids faible de chaque entrée de table de pages (du fait que ces bits d'adresse ne sont pas utilisés pour référencer des pages de 4 Ko). De façon similaire, les attributs pour les groupes de pages sont présents dans les 12 bits de poids faible de chaque entrée du répertoire de pages. Ces entrées du répertoire de pages peuvent être aussi utilisées directement comme des entrées de pages de 4 Mo, si leur attribut de taille de page (*Page size*) est positionné à 1.

Mentionnons maintenant les attributs qui ont une importance particulière dans cet article. Tout d'abord l'attribut `R/W` (Read/Write) permet un accès en mode lecture ou écriture depuis le processeur à la page concernée, s'il est positionné à 1. Sinon, la MMU n'autorise qu'un accès en lecture à la page. Le second attribut qui est important dans notre contexte est l'attribut `NX`[6] (*No eXecution*) qui, s'il est positionné, empêche d'accéder à la page pour une exécution d'instruction. Soulignons que, lorsque le

[6] À noter que le bit `NX` n'est disponible qu'en mode IA32e ou bien en mode protégé 32 bits avec l'extension PAE activée.

processeur essaie d'accéder à une page dans un mode qui est interdit, une exception, plus précisément une faute de page, est déclenchée.

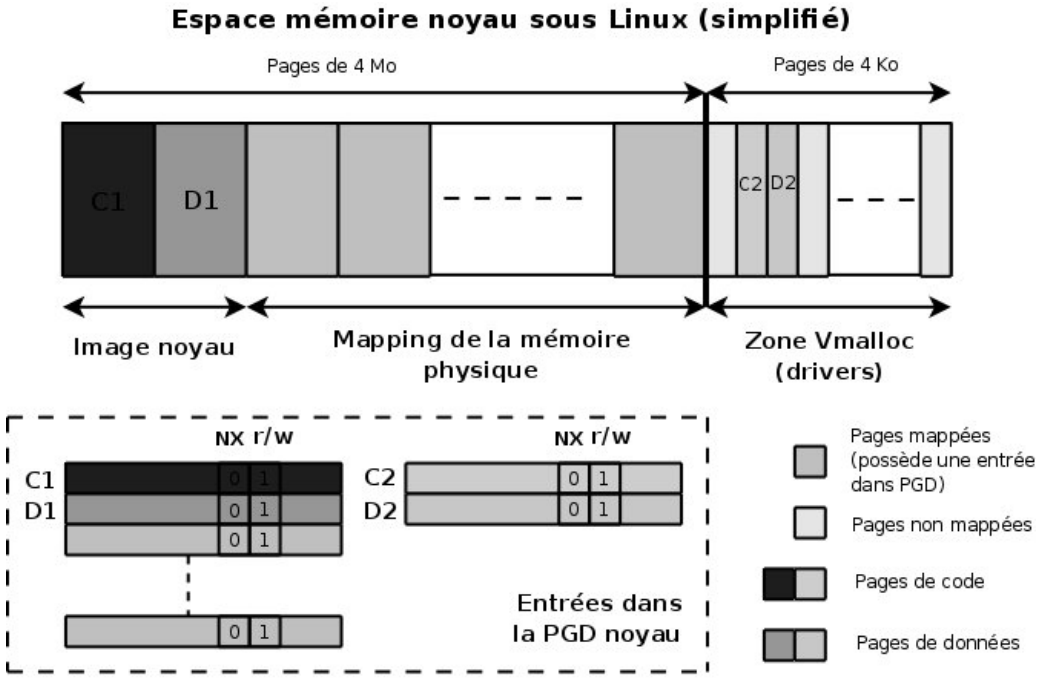

Fig. 3. Agencement de l'espace d'adressage noyau sous Linux

2.3 Support matériel pour la virtualisation — le cas d'Intel VT

Les extensions pour gérer la virtualisation sur les processeurs Intel définissent en particulier un support de virtualisation au niveau du processeur sur la série IA32. Cette extension permet de supporter deux types de logiciels : (1) le moniteur de machine virtuelle (*Virtual Machine Monitor* ou VMM, c'est-à-dire l'hyperviseur) qui se comporte comme un hôte réel est qui a le contrôle complet du processeur et des autres parties matérielles ; (2) le système invité, qui est exécuté dans une machine virtuelle (VM). Chacune des machines virtuelles s'exécute indépendamment des autres et utilise la même interface pour le processeur, la mémoire, la mémoire de masse, la carte graphique et les entrées/sorties fournies par la plateforme physique.

Le support de la virtualisation par le processeur est fourni par un ensemble d'opérations appelées opérations VMX. Il y a deux types d'opérations VMX : les opérations VMX *root*, qui sont disponibles pour l'exécution de l'hyperviseur et les opérations VMX *non-root* qui sont disponibles pour l'exécution de logiciel invité. Le comportement du processeur en mode VMX root est quasiment le même que le comportement en mode VMX non-root avec la différence qu'un jeu d'instructions supplémentaires est disponible. Le comportement du processeur en mode VMX non-root est restreint et modifié pour faciliter la virtualisation. À la place de leur comportement habituel, certaines instructions et événements causent des transitions vers la VMM ; ils sont appelés *VM-exits*. Comme ces VM-exists viennent se substituer au comportement habituel, les fonctionnalités du logiciel en mode VMX non-root sont donc limitées. Ces limitations permettent à l'hyperviseur de garder le contrôle des ressources du processeur. Comme les opérations VMX placent des restrictions même sur le logiciel qui s'exécute au niveau le plus privilégié (l'anneau 0), le logiciel en mode invité peut s'exécuter au même niveau de privilège que celui pour lequel il a été conçu à l'origine. Cette particularité peut notamment simplifier le développement d'un hyperviseur.

Le cycle de vie d'un hyperviseur peut être résumé comme suit. Tout d'abord, le logiciel passe en mode VMX en exécutant l'instruction VXMON. Ensuite, à l'aide de *VM-entries*, l'hyperviseur lance les systèmes invités dans des machines virtuelles (pour réaliser une opération de type VM-entry, l'hyperviseur exécute les instructions VMLAUNCH et VMRESUME). Il recupère ensuite le contrôle à l'aide de VM-exits qui sont automatiquement déclenchés par le logiciel invité (sans que ce dernier puisse les empêcher). Ces instructions transfèrent le contrôle à un point d'entrée spécifié par l'hyperviseur. Ce dernier peut alors prendre une décision en fonction de la cause du VM-exit courant et redonner la main à la machine virtuelle à l'aide d'une VM-entry. Facultativement, l'hyperviseur peut décider de s'arrêter et de quitter les opérations VMX (en exécutant l'instruction VMXOFF).

Les opérations VMX non-root et les transitions VMX sont contrôlées par une structure de données appelée *Virtual-Machine Control Structure (VMCS)*. L'accès à cette structure est gérée par un composant de l'état du processeur appelé « pointeur VMCS » (il contient l'adresse de la VMCS). Ce pointeur est lu et écrit à l'aide des instructions VMPTRST et VMPTRLD. L'hyperviseur configure la VMCS à l'aide des instructions VMREAD, VMWRITE et VMCLEAR. Il est important de noter que ces instructions ne déclenchent des VM-exists que si elles sont exécutées en mode VMX non-root. La Figure 4 résume la façon d'utiliser ces instructions.

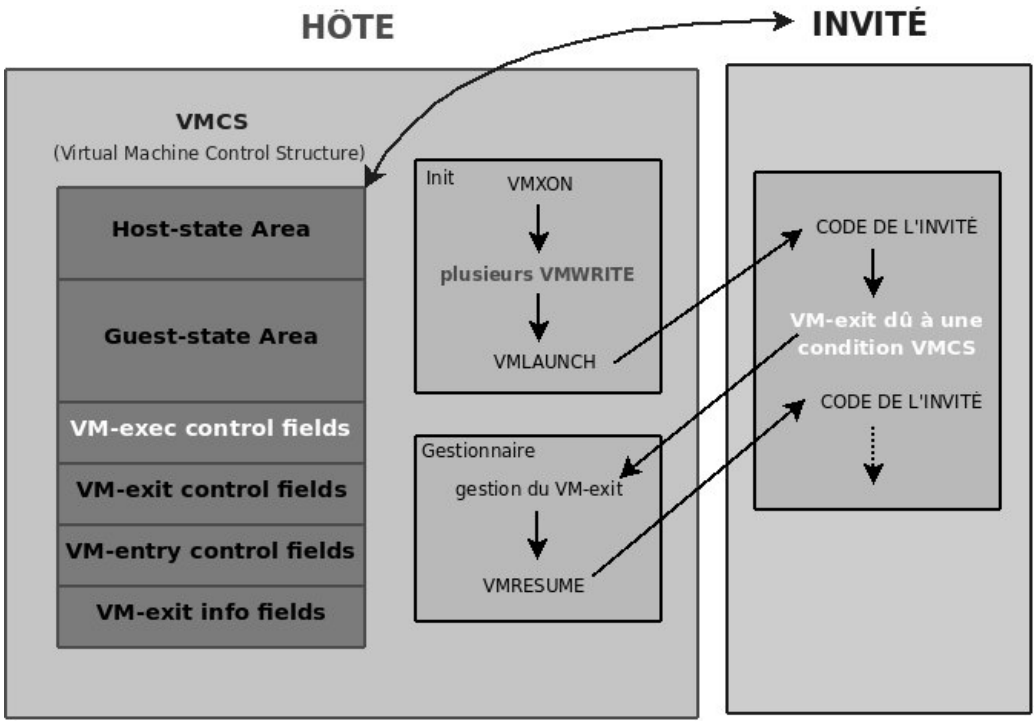

Fig. 4. Bref aperçu de Intel VT-x

3 Actions malveillantes ciblant le noyau

Dans cet article, nous considérons uniquement les actions malveillantes impliquant une perte d'intégrité d'un noyau de système d'exploitation en cours d'exécution. Cette perte d'intégrité est liée à une injection de code ou de données dans la mémoire du système. Nous commençons ainsi par analyser les vecteurs d'accès à la mémoire qu'une action malveillante peut emprunter.

3.1 Les vecteurs d'accès pour corrompre la mémoire du noyau

La première façon d'injecter des données afin de corrompre la mémoire du noyau se fait au travers de la MMU (*Memory Management Unit*) ou du MCH (*Memory Controller Hub*), ce qui implique la CPU et peut donc provenir :

– d'une fonctionnalité du système qui fournit directement le moyen de modifier n'importe quelle région de l'espace mémoire noyau. Il s'agit soit d'une fonctionnalité logicielle (tel que, dans le cas de Linux, le chargeur de modules noyau [37]

ou encore les périphériques virtuels `/dev/kmem` et `/dev/mem` [31,5]), soit d'une fonctionnalité matérielle (tel que le mode SMM — *System Management Mode* — du CPU [3,15]) ;
- d'une fonctionnalité système vulnérable qui fournit le moyen de corrompre l'espace noyau au travers de l'exploitation de sa vulnérabilité (débordement de tampon, chaînes de format, utilisation de données incorrectes — déréférencement de pointeur noyau de valeur nulle [36] — ou périmées — cf. par exemple la vulnérabilité qui a affecté les noyaux Linux intégrant la solution de sécurité *PaX* [23, Section 2]).

Un autre moyen d'injection est de passer par un bus d'E/S supportant le DMA (*Direct Memory Access*) afin d'accéder à la mémoire principale. Dans ce cas, il s'agit des périphériques de type *Bus Mastering* DMA, lesquels sont aptes à prendre le contrôle du bus et à effectuer un transfert de données dans la mémoire principale sans aucun appel à la CPU. Par exemple, le bus Firewire peut être utilisé pour lire ou écrire des données dans la mémoire physique sans le consentement du système d'exploitation [26,13,2]. Cependant, Joanna Rutkowska a montré que la lecture de la mémoire physique via un DMA peut être contrôlée au niveau logiciel [30], lorsque la CPU dispose d'une IOMMU (ou équivalent), laquelle permet au noyau de contrôler les accès DMA.

Examinons à présent, les différentes sortes d'actions malveillantes qui altèrent le comportement du noyau. Nous proposons une classification qui est découpée en trois classes principales lesquelles recouvrent la majorité des actions malveillantes que nous connaissons à l'heure actuelle.

3.2 Les classes d'actions malveillantes ciblant le noyau

Classe 1 – Injection et exécution de code malveillant Cette classe est caractérisée par les actions malveillantes qui ont besoin d'injecter du code afin d'effectuer leur activité malsaine. Suivant le type de l'action, des prérequis sont nécessaires.

- *Classe 1.1 – ajout d'une région de code noyau malveillante et détournement du flux d'exécution vers cette région :*
  Cette classe est caractérisée par les actions malveillantes qui injectent une région de code dans l'espace mémoire noyau. Certaines de ces actions malveillantes profitent par exemple d'une fonctionnalité noyau telle qu'un chargeur de module noyau [28].

- *Classe 1.2 – écrasement d'une région de code noyau existante avec du code malveillant :*

Cette classe est caractérisée par les actions malveillantes qui ont besoin d'une région de code en mémoire qui soit modifiable. Soit elles écrasent de façon permanente le code existant et empêchent ainsi son exécution future ; sinon elles recopient le code dans un autre emplacement (par exemple, dans des zones de *padding* dans des régions de code) et l'exécutent juste après leurs propres instructions (le précurseur dans ce domaine est Silvio Cesare [6]).

– *Classe 1.3 – injection de code malveillant dans une région de données noyau et détournement du flux d'exécution vers cette région :*
Cette classe est caractérisée par les actions malveillantes qui nécessitent qu'une région de données soit exécutable. Par exemple, les actions malveillantes qui injectent du code au travers de débordements de tampon [17] appartiennent à cette classe. Celles qui injectent du code dans des emplacements vides (*padding*) de pages de données appartiennent également à cette classe.

– *Classe 1.4 – injection de code malveillant au sein d'une région n'appartenant pas au noyau (typiquement, une région de l'espace utilisateur) et détournement du flux d'exécution vers cette région :*
Cette classe est caractérisée par les actions malveillantes qui ont seulement besoin que le noyau n'empêche pas le déréférencement de pointeurs invalides depuis le mode noyau. Cela signifie que l'action malveillante exploite une faille dans le noyau qui autorise l'exécution en ring 0 de code arbitraire qui se trouve à l'extérieur de l'espace noyau (en particulier, l'espace utilisateur ou l'espace hyperviseur). Cela provient, par exemple, de bugs noyau qui peuvent être exploités afin d'écrire une adresse valide de l'espace utilisateur dans un pointeur noyau, provoquant au minimum une injection de données inattendues depuis l'espace utilisateur vers l'espace noyau[7], sinon une exécution de code arbitraire de l'espace utilisateur depuis le mode noyau. Une action malveillante de ce type est illustrée par l'exploitation de la vulnérabilité de l'appel système *vmsplice* de Linux, afin d'obtenir un shell root en local [8,7] (cf. [36] pour une explication détaillée sur la façon d'exploiter un déréférencement de pointeur noyau de valeur nulle).

Classe 2 – Exécution de code existant dans un ordre arbitraire. Cette classe est caractérisée par les actions malveillantes qui n'injectent pas de code dans le noyau, mais altèrent le flux d'exécution (par exemple via l'altération de données de contrôle de flux dans la pile) afin d'exécuter du code noyau existant dans un mauvais ordre [25,12].

[7] La limite de l'espace utilisateur sous Linux est représentée par la constante `TASK_SIZE`.

Par exemple, une action malveillante peut provoquer l'exécution d'une fonction en mémoire (ou seulement du code) avec des paramètres choisis, uniquement au travers de la modification de la pile. Elle peut remplacer le compteur ordinal, sauvegardé dans la pile, avec l'adresse d'un code existant dans l'espace noyau, afin de détourner le flux d'exécution, autrement dit d'exécuter du code illégitime vis-à-vis de l'exécution courante.

Cette approche peut être généralisée afin d'exécuter en séquence plusieurs parties de code existant. Nous qualifions alors cette action d'exécution ordonnée de façon malveillante.

Classe 3 – Modification invalide de données contraintes par le noyau. Cette classe est caractérisée par les actions malveillantes qui altèrent le comportement du noyau en écrasant certaines de ses données qui sont supposées contraintes au niveau de la spécification. Par exemple, l'écrasement d'attributs d'une page mémoire afin de passer outre la protection contre l'exécution de cette page est une action malveillante qui appartient à cette classe. De même, les dépassements d'entiers et plus particulièrement les dépassements de compteurs de références font parties de cette classe [27].

De même, les actions malveillantes qui désactivent des mécanismes de sécurité seulement par l'écrasement de données noyau (sans que s'ensuive une exécution de code) font partie de cette classe.

4 La protection du noyau vis-à-vis des actions malveillantes

Dans cette section, nous analysons les moyens de protection contre les actions malveillantes agissant sur un noyau en cours d'exécution. Cette analyse nous amène à élaborer une nouvelle approche fondée sur les extensions matérielles de virtualisation que nous détaillons en section 5.

La décomposition de cette analyse suit la classification des actions malveillantes que nous avons établie dans la section précédente.

4.1 À propos des mécanismes de sécurité

Les mesures de sécurité employées pour la protection d'un système d'information sont généralement regroupées dans trois catégories majeures : prévention, détection et restauration. Il a été démontré que la détection des maliciels est un problème indécidable [16, Chap. 3]. En conséquence, puisque la restauration nécessite une détection préalable, nous privilégions dans notre approche les mécanismes de

prévention lorsque cela est possible. Dans le reste de la section, nous concentrons uniquement notre attention sur les mécanismes de prévention qui protègent l'espace noyau contre les actions malveillantes.

4.2 Le contrôle des vecteurs d'accès

Nous avons déterminé deux vecteurs d'accès à la mémoire pour les actions malveillantes dans la section 3.1. Nous examinons maintenant les mesures de sécurité qui s'appliquent à ce niveau. Notons qu'une action malveillante n'emploie qu'un seul vecteur d'accès mais elle peut à son tour activer d'autres vecteurs d'accès pour de futures actions malveillantes.

Le contrôle des vecteurs d'accès de type CPU. Comme expliqué en section 3.1, les fonctionnalités noyau qui fournissent directement le moyen d'écrire dans n'importe quelle région mémoire de l'espace noyau (tel que, dans le cas de Linux, le chargeur de modules noyau ou bien les périphériques virtuels `/dev/kmem` ou `/dev/mem`) sont couramment employées par de nombreux maliciels pour s'injecter dans l'espace noyau [24]. Ces fonctionnalités doivent évidemment être contrôlées. Par exemple, les périphériques `/dev/kmem` et `/dev/mem` peuvent être désactivés (comme le fait par exemple grsecurity [34]) ou filtrés afin d'autoriser uniquement les accès aux E/S *mappées* en mémoire (comme le font les noyaux Linux actuels s'ils sont correctement configurés). Aussi, afin de détecter les modules noyaux malveillants, une solution est d'établir une vérification des modules avant chargement via l'utilisation de signatures cryptographiques [9]. Cependant, de cette façon nous n'empêchons pas l'exploitation de bugs qui peuvent se trouver à l'intérieur des modules signés. De surcroît, nous devons garantir que la façon d'ajouter des modules est incontournable et ne peut être altérée.

L'autre vecteur d'accès employé par les maliciels afin d'altérer la mémoire du noyau est l'exploitation de failles au sein de fonctionnalités noyau qui n'ont pas pour vocation la modification de l'espace noyau. Évidemment, à la différence du vecteur d'accès précédent, celui-ci ne peut être contrôlé par les mêmes techniques. De plus, ce type de vecteur d'accès est d'autant plus facile à trouver dans le noyau que le nombre de modules (qui peuvent être potentiellement *buggués*) y étant ajoutés est important. En fait, la grande majorité des failles noyau proviennent des pilotes de périphériques (cf. Annotation 3). Une solution de sécurité, appelée *PaX* [35], développée pour Linux, contient des mécanismes (tel que *randkstack* qui implémente la randomisation de la pile noyau) fondés sur des approches génériques de protection du noyau vis-à-vis d'actions malveillantes. Cependant, ces mécanismes sont actuellement implémentés

au même niveau de privilège que le noyau et ainsi ne peuvent empêcher que l'entrée dans l'espace noyau de données malveillantes. Ils ne peuvent pas être efficaces si du code malveillant est déjà présent dans le noyau.

Le contrôle des vecteurs d'accès de type DMA. Afin d'empêcher l'accès à la mémoire par des périphériques de type *Bus Mastering* DMA, il est possible de désactiver les canaux DMA à partir du noyau, mais il est alors vraiment très pénalisant en temps CPU de transférer des données à des périphériques d'E/S, et cela requiert la modification des pilotes de périphériques afin qu'ils interrogent régulièrement les périphériques sur la disponibilité de données sans établir de transferts DMA (ce qui est inacceptable pour certains périphériques). Dans une moindre mesure, pour les noyaux Linux, la désactivation des E/S brutes ainsi que du périphérique virtuel `/dev/port` (comme le fait par exemple grsecurity [34]) empêche la mise en place de transferts DMA à partir de l'espace utilisateur.

Finalement, l'approche ultime nécessite un système informatique qui inclut une IOMMU (*Input/Output Memory Management Unit*, qui correspond à la technologie VT-d pour les plateformes Intel et fait partie de l'architecture HyperTransport dans le cas de AMD). Avec cette unité matérielle, il est alors possible de protéger la mémoire principale contre les périphériques malveillants [30]. Une IOMMU est une unité de gestion mémoire (MMU) qui connecte un bus d'E/S (supportant le DMA) à la mémoire principale. De la même façon qu'une MMU traditionnelle, la IOMMU s'occupe de la correspondance entre les adresses d'E/S et les adresses physiques de la mémoire. Les tables de traductions se trouvent dans la mémoire principale et sont sous le contrôle de la CPU, c'est-à-dire du noyau, au lieu du périphérique. Cela dit, les tables de traduction pour la IOMMU sont alors des parties critiques du système qui doivent être protégées contre de potentielles actions noyau malveillantes. Insistons de nouveau sur le fait que les mécanismes de protection ont besoin de s'exécuter dans un mode plus privilégié que le noyau s'ils ont pour vocation la protection de celui-ci.

4.3 Prévention de la corruption de l'espace noyau suivant les classes

Avant d'aborder les différentes solutions de protection suivant les classes d'actions malveillantes, remarquons que les techniques de randomisation de l'agencement de l'espace d'adressage (ASLR — *Address Space Layout Randomization*), telles que proposées par *PaX* [35], ne sont pas efficaces pour protéger le noyau d'actions malveillantes. Non seulement l'ASLR doit être effectuée sur une architecture 64 bits [32] afin d'obtenir une protection efficace, mais elle n'est de plus envisageable que pour l'espace utilisateur. En effet, des structures noyau cruciales peuvent être localisées

précisément depuis l'espace utilisateur qu'une technique d'ASLR soit employée ou non. Par exemple, la GDT peut être localisée en mémoire grâce à l'exécution de l'instruction `sgdt` qui est légale en mode utilisateur.

Comment protéger le noyau des actions de la Classe 1 En ce qui concerne la Classe 1, examinons les différentes protections possibles vis-à-vis des sous-classes d'actions malveillantes.

Afin de se protéger de la Classe 1.1, il est possible de développer des solutions qui restreignent l'utilisation des fonctionnalités du noyau capables de modifier n'importe quelle région de l'espace mémoire noyau (comme décrit dans la section 4.2). En ce qui concerne la Classe 1.2, les régions de code peuvent être rendues exécutables et non modifiables. De même, afin de se protéger de la Classe 1.3 les régions de données peuvent être rendues lisibles, modifiables et non exécutables. Cela peut être effectué par la modification des attributs des entrées des tables de pages (cf. section 2). Cependant, des problèmes pourraient survenir de la prévention de l'exécution des piles noyau. En effet, du code est parfois légitimement injecté dans la pile afin de rendre possible certaines fonctionnalités. Le projet OpenWall a du faire face à des problèmes de ce type afin d'implémenter une pile *utilisateur* non-exécutable pour Linux. C'est pourquoi l'implémentation d'une pile *noyau* non-exécutable pourrait mener à des problèmes du même acabit. Heureusement, dans le cas de Linux, ces problèmes ne concernent que la pile *utilisateur*. En effet, tout d'abord, les fonctions imbriquées ne sont pas employées au sein du noyau et ainsi *gcc* n'a pas besoin d'une pile *noyau* exécutable (nécessaire pour ses *trampolines*). Ensuite, la seule partie du noyau Linux qui dépend d'une pile exécutable (le sous-système de gestion des signaux) établit du code uniquement dans la pile *utilisateur*. Enfin, les langages fonctionnels et les programmes qui génèrent du code en cours d'exécution, dépendent d'une pile exécutable, mais ils sont exécutés en espace utilisateur et ainsi ne dépendent pas d'une pile *noyau* exécutable.

Cependant, une action malveillante pourrait désactiver cette protection en changeant tout d'abord les attributs de pages d'une région mémoire de données qui contiendrait du code malveillant et ensuite exécuter cette région. Ainsi, la modification des attributs de pages doit être impossible afin d'empêcher qu'une région de données devienne une région de code. Nous pourrions empêcher que les tables de pages soient modifiées en spécifiant que les pages qui les contiennent ne soient pas modifiables. Mais alors, il ne serait plus possible pour le noyau d'ajouter de nouveaux *mappings* mémoire (pour l'ajout de modules par exemple) étant donné que les pages contenant les tables de pages ne seraient plus modifiables. L'unique solution serait alors de créer de nouvelles tables de pages et de charger le registre `cr3` avec l'adresse physique qui les référence.

Mais cela peut aussi bien être effectué par un maliciel qui s'exécute au même niveau que le noyau. Dans notre approche, présentée dans la section 5, nous expliquons comment résoudre de tels problèmes. L'utilisation des technologies matérielles de virtualisation rend l'emploi des notions « régions de données noyau » et « régions de code noyau » applicable vis-à-vis des droits d'exécution.

Finalement, afin de se protéger de la Classe 1.4, les solutions génériques de protection contre l'exploitation de débordements de tampon (comme PointGuard [11] par exemple) sont envisageables, car elles protègent l'espace noyau vis-à-vis des modifications malveillantes de pointeurs. Ainsi, elles protègent contre le détournement du flux d'exécution vers une adresse spécifique en mémoire. Une autre approche est d'empêcher les pointeurs de l'espace utilisateur d'être déréférencés en mode noyau[8]. Ce principe est d'ailleurs suivit par la solution de sécurité *PaX* [35] avec son mécanisme *UDEREF* [33].

Comment protéger le noyau des actions de la Classe 2 Afin de se protéger de la Classe 2, l'approche relative à la Classe 1 n'est pas satisfaisante car le flux d'exécution est détourné vers une région de code qui existe déjà en mémoire. Afin de prévenir ces actions malveillantes, il est crucial de protéger les données de contrôle du flux d'exécution. Il s'agit de protéger les données de contrôle présentes dans la pile mais également d'empêcher les pointeurs noyau d'être écrasés de façon malveillante.

Une première étape est de considérer les mécanismes (comme StackGuard [10] ou Propolice/SSP — Stack-Smashing Protection) qui protègent contre les détournements de flux d'exécution, faisant intervenir des débordements de tampons dans la pile, via l'utilisation de canaris. Mais ces solutions ne protègent pas contre les débordements de tampon qui écrasent des pointeurs de fonction [4] (comme les débordements de tampon dans le *tas*, ou *heap overflow* [1]). Ainsi, une seconde étape consiste à considérer une approche générique qui protège contre l'exploitation de tout débordement de tampon, comme PointGuard [11] qui chiffre les pointeurs lorsqu'ils sont en mémoire.

Cette dernière solution est vraiment intrusive et dépend de la confidentialité de la clé de chiffrement. À ce niveau, nous proposons une approche complémentaire qui empêche que certaines actions noyau se comportent mal. En d'autres termes, nous essayons d'empêcher le noyau de se comporter de façon malveillante.

Comment protéger le noyau des actions de la Classe 3 Les protections contre les actions malveillantes de cette classe ne sont pas, à notre connaissance, très répandues. Néanmoins, la solution de sécurité *PaX* [35] fournie une protection

[8] Notons toutefois que certains de ces déréférencements sont tout à fait légitimes lorsque le noyau a besoin de récupérer des données dans l'espace utilisateurs.

générique contre les dépassements de compteurs de références [27] avec son mécanisme *REFCOUNT*.

Dans la prochaine section, notre approche (fondée sur la préservation d'objets contraints, via l'emploi des extensions matérielles de virtualisation) fournit une solution pour enrayer partiellement cette classe.

5 La virtualisation matérielle rend possible la protection contre les maliciels noyau

Les mesures traditionnelles de sécurité que nous venons d'examiner font face à des problèmes insolubles en ce qui concerne les actions malveillantes qui s'exécutent dans l'espace noyau. Dans notre approche, nous essayons de couvrir ces problèmes en limitant les dommages que le noyau peut causer au système. Afin de rendre ces mesures de sécurité possible, nous implémentons un hyperviseur qui contrôle certaines des actions qu'un noyau peut effectuer. Cette approche est possible grâce aux technologies matérielles de virtualisation qui permettent d'exécuter un hyperviseur dans un mode matériel plus privilégié que celui du noyau. Insistons encore sur le fait que nous devons agir à un niveau de privilège supérieur à celui du noyau si nous souhaitons nous protéger d'actions malveillantes s'exécutant dans le noyau. De plus, comme l'hyperviseur est léger, il est possible d'effectuer assez facilement une vérification exhaustive de son code. Dans la prochaine section, nous détaillons notre approche. Le concept majeur est d'essayer de garantir la préservation de contraintes du système.

Cette approche qui est décrite dans la suite de cette section se suffit à elle-même pour les Classes 1.1, 1.2 et 1.3. Pour les Classes 1.4 et 2, notre approche est complémentaire des solutions examinées précédemment. Enfin, en ce qui concerne la Classe 3, notre approche possède une aptitude unique à contrôler le mode ring-0 (i.e. le mode noyau) et ainsi peut partiellement enrayer les actions malveillantes de cette classe.

5.1 Aperçu général de Hytux

Nous avons partiellement développé une preuve de concept pour une cible Linux x86 qui s'exécute sur une plateforme 64 bits supportant la technologie Intel VT-x [20] et facultativement Intel VT-d [18]. Notre preuve de concept est baptisée Hytux et consiste en un hyperviseur léger qui dépend de ces technologies de virtualisation. Hytux emprunte la notion d'hyperviseur léger au projet *bluepill* [29]. Il s'installe lui-même en tant qu'hyperviseur sur un système faisant tourner Linux et place alors ce dernier dans une machine virtuelle qui est alors surveillée et contrôlée (au travers de la configuration d'une seule VMCS).

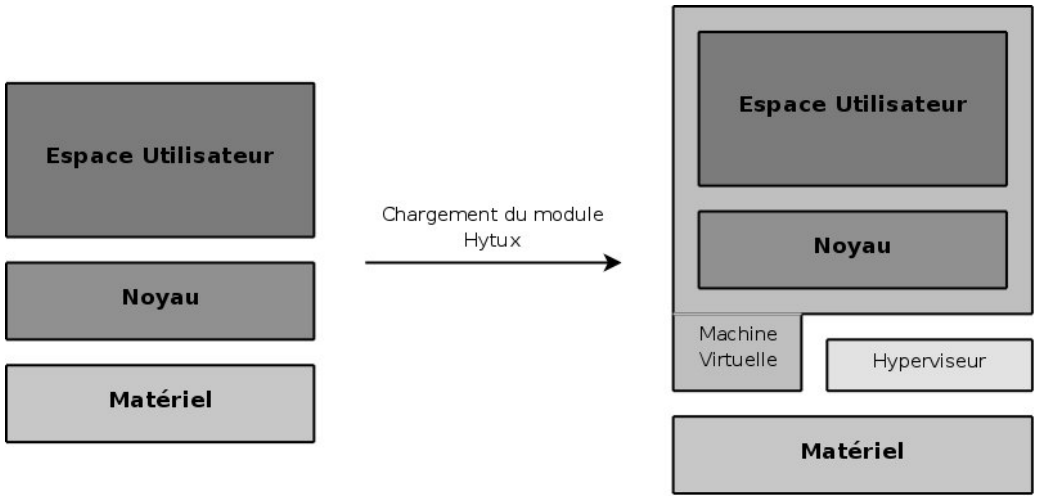

Fig. 5. Hytux – un hyperviseur léger

Dans ce qui suit, nous détaillons les différentes tâches effectuées par (ou envisagées pour) notre hyperviseur.

5.2 Protection des objets contraints par le noyau face à une altération depuis la CPU

L'idée ici est de préserver les entités qui sont considérées contraintes par le noyau. Nous définissons le concept d'Objet Contraint par le Noyau (*Kernel-Constrained Object*) dans ce qui suit.

Definition 1. *Un* Objet Contraint par le Noyau (KCO — Kernel-Constrained Object) *est une entité du système sur lequel le noyau s'exécute et qui devrait être* légitimement *dans un état fixe ou dans un état prédictible et déterministe, durant l'exécution du système.*

Nous insistons dans cette définition sur le fait que l'entité est considérée être un KCO si elle est contrainte dans sa spécification, peu importe que l'implémentation soit *bugguée* ou qu'une faille de conception existe.

La préservation des KCO expliquée au travers d'un exemple. Le principe est d'empêcher que les KCO soient altérés d'une quelconque manière. Notons que le premier état des KCO que notre hyperviseur (Hytux) voit est supposé être sûr. À

partir de cette situation, Hytux essaie d'empêcher les KCO d'être altérés. Pour bien comprendre ce concept, prenons l'exemple du registre de processeur `idtr` qui est un KCO du point de vue du noyau Linux. En effet, il est positionné lors de l'initialisation du système à l'adresse de l'IDT (*Interrupt Descriptor Table*) et n'est pas supposé être modifié plus tard. Cependant, l'instruction processeur `lidt` disponible depuis le mode ring-0 (i.e. le mode noyau) autorise le chargement d'une nouvelle adresse dans ce registre. Par conséquent, si le noyau contient un bug qui peut être exploité ou une fonctionnalité (que nous appelons dans ce contexte une faille de conception) pour exécuter cette instruction avec un paramètre arbitraire, le KCO `idtr` pourrait être altéré. C'est pourquoi nous devons dans ce cas préserver la contrainte fixe qui régie le registre `idtr`. Afin de pourvoir à ce besoin, notre approche est d'émuler l'instruction `lidt` à l'intérieur de notre hyperviseur assisté par le matériel. Ainsi, quand le noyau exécute cette instruction pour la première fois, le comportement normal est émulé par Hytux, puis ce dernier passe de façon permanente à une émulation qui ne fait rien. De cette façon, le KCO est préservé. L'émulation de l'instruction `lidt` est aisément accomplie via l'utilisation de Intel VT-x. En effet, une VM-exit est rendu obligatoire en positionnant à la valeur 1, le champ `Descriptor-table exiting` de la VMCS. Nous procédons de même pour le registre `gdtr`, qui est également étiqueté KCO. Au sujet des registres de contrôle `cr0` et `cr4`, nous agissons de la même façon mais seulement pour leurs bits qui peuvent être considérés comme des KCO[9]. Finalement, le registre de contrôle `cr3` est un cas particulier de KCO, plus compliqué, qui couvre les contraintes associées aux régions mémoire de données et de code. Nous détaillons plus longuement ce KCO dans le prochain paragraphe.

L'agencement de l'espace mémoire noyau en tant que KCO Afin de se protéger contre les actions malveillantes de la Classe 1, la section 4 a montré que les attributs de pages noyau pouvaient automatiquement être établis en fonction de l'emploi des pages. Plus précisément, pour une page qui contient du code, le drapeau `R/W` (Read/Write) n'est pas positionné ; pour une page qui contient des données qui peuvent être modifiées, les drapeaux `NX` (No eXecution) et `R/W` sont positionnés ; enfin, pour une page de données en lecture seule le drapeau `NX` est positionné mais pas le drapeau `R/W`. Comme présenté dans la section 2, la première partie de l'espace noyau est remplie de pages de 4 Mo mappées en mémoire et leurs attributs ne sont pas supposés être modifiés. Ainsi, les attributs de pages doivent être positionnés afin de faire respecter les différentes caractéristiques des pages : exécution-seule, lecture/écriture-seule, lecture-seule. De la même façon, pour la zone VMALLOC qui est composée de pages de 4 Ko, nous pouvons faire respecter ces contraintes grâce

[9] Intel VT-x fournit des masques invité/hôte pour ces registres de contrôle, ce qui simplifie la procédé.

aux attributs de pages. Cependant, la situation est ici un peu plus compliquée car cet espace mémoire est principalement utilisé pour charger des modules noyau pour Linux (LKM). Ainsi, aucune page n'est mappée en mémoire sauf celles qui contiennent les modules déjà chargés. C'est pourquoi, la primitive noyau `vmalloc` (employée pour l'allocation de mémoire des LKM) doit être modifiée. Dans notre approche, cette primitive noyau doit prendre un drapeau comme paramètre pour préciser le type de l'allocation, qui peut être : code, données ou données en lecture seule. Avec ce mécanisme en place, `vmalloc` peut alors positionner les attributs de pages en fonction des contraintes nécessaires pour les différents segments du module (code, données ou données en lecture seule), au moment ou ce dernier est chargé en mémoire et que par conséquent `vmalloc` est appelé. Ce procédé mène à la situation illustrée par la Figure 6.

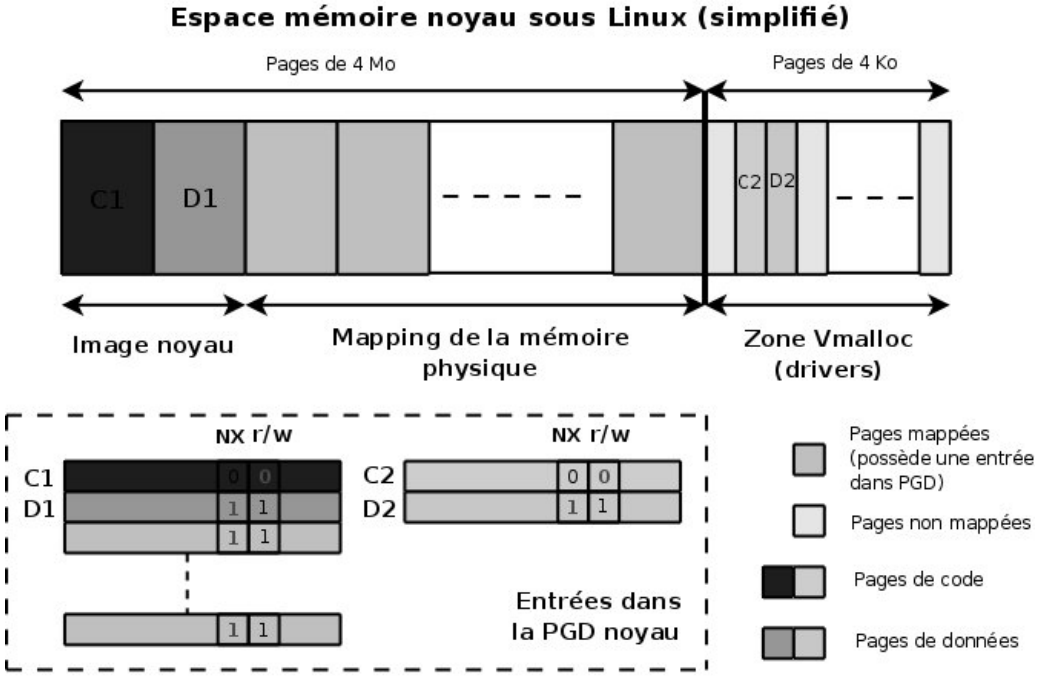

Fig. 6. L'agencement de l'espace d'adressage noyau (première modification)

Cependant, une action noyau malveillante pourrait modifier les attributs d'une page noyau afin de l'employer dans un autre but (typiquement, une page de données transformée en une page de code). Afin de résoudre ce problème, l'attribut de page

`R/W` ne doit pas être positionné sur l'ensemble des pages qui contiennent les tables de pages *noyau*, comme l'illustre la Figure 7.

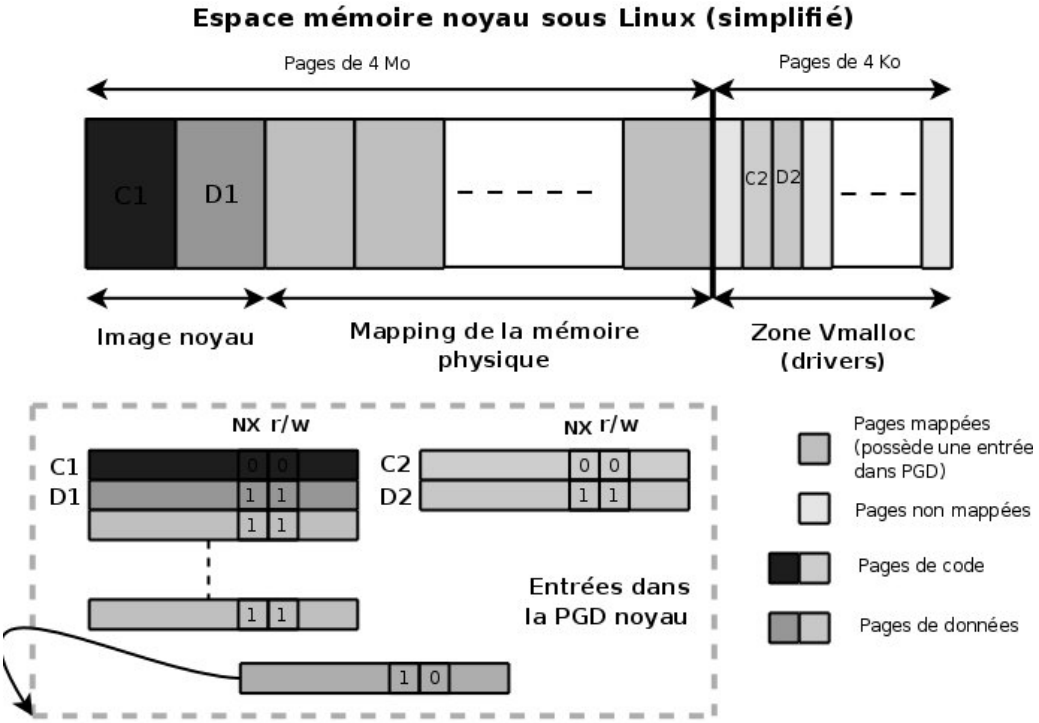

Fig. 7. L'agencement de l'espace d'adressage noyau (seconde modification)

Néanmoins, cette solution n'est pas satisfaisante car le noyau ne peut alors plus écrire de nouvelles entrées dans les tables de pages noyau lorsqu'il en a besoin, c'est-à-dire quand il charge un module, car cela produirait une faute de page qu'il ne saurait traiter. Cela n'est évidemment pas le comportement souhaité. Pour pallier ce problème, notre approche profite des extensions matérielles de virtualisation et déclenche une VM-exit quand les tables de pages noyau sont accédées. Pour parvenir à cet objectif, l'hyperviseur positionne le bit 14 dans le champ *Exception Bitmap* de la VMCS afin de déclencher une VM-exit sur les fautes de pages. Mentionnons, que notre hyperviseur dispose de ses propres tables de pages (chargées automatiquement lors d'une VM-exit) qui l'autorise à écrire dans toute la mémoire. De plus, afin de préserver le KCO, l'hyperviseur a besoin de conserver une copie de l'agencement initial de l'espace noyau en ce qui concerne les pages en exécution-seule, lecture/écriture-seule

et lecture-seule (i.e. il conserve une copie des tables de pages noyau), afin de valider ou non les futures modifications des entrées dans les tables de pages. Cependant, les entrées dans ces tables ne sont pas changées après l'initialisation du système sauf pour la zone `VMALLOC`[10]. Ainsi, l'hyperviseur n'a besoin de rester informé que vis-à-vis de l'agencement de la zone `VMALLOC`. Cela implique une modification de la fonction `vmalloc` afin d'informer l'hyperviseur de l'allocation de nouvelles pages (au travers de l'instruction `VMCALL` qui déclenche simplement une VM-exit). Tout d'abord, cela permet à l'hyperviseur de mettre à jour son KCO (l'agencement contraint de l'espace mémoire), et ensuite l'invite à écrire réellement les entrées dans les tables de pages avec des attributs qui dépendent des contraintes relatives au type des pages.

Expliquons à présent ce qu'il se passe lorsque l'hyperviseur prend le contrôle de la CPU après l'occurrence d'une faute de page. À cet instant, l'hyperviseur vérifie si la faute s'est déclenchée suite à un accès aux tables de pages noyau (via la lecture de l'adresse fautive dans le champ `exit qualification` de la VMCS). Si l'adresse ayant fait défaut n'est pas dans la plage des tables de pages noyau, alors l'hyperviseur redonne la main au noyau (au travers d'une VM-entry[11]). Autrement, si l'adresse ayant fait défaut se situe au sein des tables de pages noyau, alors l'hyperviseur rejoue l'instruction qui a causé la faute de page afin d'écrire cette fois réellement l'entrée dans les tables de pages. Ensuite, il vérifie que les contraintes sur la page affectée sont préservées suivant l'agencement de l'espace mémoire noyau qu'il connaît[12]. Si l'instruction résulte en l'invalidation de la contrainte sur une entrée existante dans une table de pages noyau, l'hyperviseur restaure la contrainte et annule donc la modification. Si l'instruction résulte en l'écriture d'une nouvelle entrée dans une table de pages noyau, l'hyperviseur efface simplement cette nouvelle entrée. Ce dernier cas se justifie par le fait que le noyau n'ajoute de nouvelles entrées dans les tables de pages de l'espace noyau que lors de l'appel à la fonction `vmalloc` (cf. Annotation 10), et que cette primitive est modifiée de façon à informer l'hyperviseur quand elle a besoin d'ajouter une entrée.

Nous devons à présent gérer un dernier problème. Considérons qu'une action malveillante construise ses propres tables de pages noyau à partir des existantes mais avec des contraintes malveillantes (par exemple une page de données avec des droits

[10] Nous laissons volontairement de côté le cas de la zone `KMAP` car l'approche déployée pour gérer ce cas est similaire à la zone `VMALLOC`.

[11] Notons que dans ce cas, l'hyperviseur a besoin d'effectuer une action supplémentaire. Il doit écrire des informations dans la VMCS à propos de la faute de page (qui vient juste de se produire) afin que la VM-entry délivre cet évènement dans l'environnement de l'invité.

[12] Notons qu'effectuer cette vérification sans rejouer l'instruction serait plus compliqué et ainsi plus pénalisant en temps CPU car il faudrait déterminer tout d'abord la nature de l'instruction et ensuite vérifier ces arguments.

d'exécution). Par la suite, cette action les injecte dans une région de données noyau et finalement déclenche le chargement du registre `cr3` avec l'adresse du début en mémoire de ces tables de pages malveillantes. Ce scénario contourne notre protection. C'est pourquoi tous les chargements de `cr3` doivent être contrôlés. C'est encore une fois aisément réalisé via les extensions matérielles de virtualisation. Dans notre approche, le champ `CR3-load exiting` de la VMCS est positionnée à 1, afin de déclencher une VM-exit à chaque chargement du registre `cr3`. À cet instant, l'hyperviseur vérifie les dernières entrées de la table de pages de plus haut niveau (connue sous le nom de Répertoire de Pages en IA32 et sous le nom de PML4 en IA32e) provenant de l'adresse qui était sur le point d'être chargée dans le registre `cr3`. Ces entrées constituent l'espace d'adressage du noyau. Ainsi, elles doivent être égales à celles que l'hyperviseur connaît. Si cela n'est pas le cas, l'hyperviseur émule l'instruction qui a généré un chargement de `cr3`, en ne faisant rien. Ensuite il redonne la main au noyau (via une VM-entry).

Finalement, il est important de constater que certaines régions noyau ont besoin d'être placées dans des pages à lecture seule. Il en est ainsi, au moins, des régions qui contiennent toutes les pages de tables noyau, la GDT et l'IDT. De même, nous n'avons pas couvert dans cette section, le cas des tables de pages qui décrivent l'espace d'adressage utilisateur pour chaque processus. Dans notre contexte, nous essayons de prévenir la corruption de l'espace noyau. Ainsi, notre hyperviseur devrait vérifier (d'une façon similaire à celle qui vient d'être expliquée dans le cas des tables de pages noyau) qu'aucune entrée de tables de pages, écrite pour décrire l'agencement mémoire de l'espace utilisateur, ne contient une adresse physique de page noyau.

Gestion générique de données simples contraintes par le noyau. Les mesures de sécurité qui viennent d'être présentées afin de préserver les tables de pages noyau peuvent être facilement employées pour n'importe quelle donnée simple en mémoire contrainte par le noyau. L'approche générique consiste à allouer une page mémoire (dans la zone VMALLOC par exemple), à y placer la donnée particulière qui est contrainte par le noyau, et enfin à positionner à 0 l'attribut de page `R/W`. Par ce procédé, l'hyperviseur est alors capable de préserver la contrainte de la même façon que cela a été décrit précédemment. Ainsi, aucun code noyau ou utilisateur ne peut altérer les contraintes des données que ce mécanisme protège.

Pour conclure sur cette activité de notre hyperviseur, il est nécessaire de remarquer que les KCO sur lesquels nous nous sommes concentrés, ne constituent pas une liste exhaustive. Nous avons seulement souhaité montrer certains KCO du noyau Linux ainsi que la façon de les protéger d'une quelconque altération. Nous tenons à souligner

le fait que tous les KCO ne peuvent être facilement capturés. Cependant, la seule préservation de certains KCO bien choisis permet la protection du noyau contre de nombreux maliciels agissant au niveau noyau (tels que ceux qui dépendent de l'écriture soit dans la GDT, ou dans l'IDT ou encore dans la table des appels système ou bien dans des registres comme `idtr`, `gdtr`, etc.), et cela d'une façon globale.

5.3 Prévention de la corruption de la mémoire de l'hyperviseur

Au travers du contrôle des vecteurs d'accès de type CPU. Afin de prévenir la corruption de l'espace mémoire de l'hyperviseur, celui-ci doit virtualiser l'unité de pagination. C'est-à-dire qu'il doit conserver le contrôle sur les mécanismes de traduction d'adresses du processeur. Dans notre cas, cela signifie que le registre `cr3` ne doit être accédé que par l'hyperviseur et qu'il doit émuler les modifications des tables de pages du système invité afin de s'assurer qu'aucune adresse physique de son espace mémoire ne s'y retrouve[13].

De même, l'hyperviseur doit filtrer certains ports d'E/S[14] (au moins les ports d'adresses PCI — `0xCF8-0xCFB`, et les ports de données PCI — `0xCFC-0xCFF`) afin de le protéger contre des attaques qui profitent du mode SMM (System Management Mode) de la CPU [3,15].

Au travers du contrôle des vecteurs d'accès de type DMA. Une première approche est de contrôler et de filtrer les accès aux ports d'E/S (cf. Annotation 14) qui ont pour origine un pilote de périphérique afin d'empêcher l'établissement d'un transfert DMA depuis le périphérique correspondant jusqu'à l'espace mémoire de l'hyperviseur. Dans ce cas, nous devons déclencher une VM-exit quand un accès aux ports spécifiques est effectué, et ensuite vérifier que les adresses physiques qui vont être écrites par le périphérique n'appartiennent pas à l'espace de l'hyperviseur. Néanmoins, cette approche semble réellement difficile à implémenter car les ports d'E/S impliqués dans l'établissement d'un transfert DMA dépendent non seulement du type de bus depuis lequel il est établi, mais également du périphérique lui-même [14]. De plus, cette approche ne prévient que la corruption de l'espace mémoire de l'hyperviseur qui provient d'intrus agissant depuis l'intérieur du système. Elle ne protège pas cet espace contre des périphériques malveillants de type BusMaster-DMA, lesquels prennent le contrôle d'un bus (tel que le bus Firewire [26]) sans implication de la CPU. Afin de

[13] Notons que l'instruction `invlpg` qui invalide une entrée dans la TLB (Translation Lookaside Buffer) n'a pas besoin d'être émulée, car notre hyperviseur n'a qu'un seul invité qui coïncide avec l'hôte. Ainsi, il n'a pas besoin de maintenir des *shadow page tables*.

[14] Notons qu'un accès à un quelconque port d'E/S peut déclencher une VM-exit si la VMCS est correctement configurée.

se protéger contre ce type de problème, un système qui contient une IOMMU est nécessaire.

5.4 Prévention de la corruption de la mémoire du noyau depuis des fonctionnalités matérielles

La section 5.3 a examiné des solutions afin de prévenir la corruption de l'espace mémoire de l'hyperviseur. Les solutions envisagées (excepté pour les mécanismes de traduction d'adresses du processeur) peuvent également prévenir la corruption de l'espace mémoire noyau provenant d'accès malveillants à des fonctionnalités matérielles.

6 Conclusion et perspectives

Dans ce papier, nous avons présenté des mécanismes de sécurité qui protègent le système de certaines classes d'actions noyau malveillantes. Cependant, ces mécanismes ont leurs limites. Afin de les rendre incontournables, ils doivent s'exécuter dans un mode plus privilégié que celui dans lequel s'exécute le noyau, et doivent donc employer du matériel dédié. C'est pourquoi nous proposons de les implémenter au sein d'un hyperviseur léger, appelé *Hytux*. Un tel hyperviseur accomplit différentes vérifications afin de prévenir la corruption d'entités cruciales contraintes par le noyau de l'invité s'exécutant au dessus de l'hyperviseur. Nous proposons une première classification des attaques possibles et pour certaines les protections correspondantes fondées sur la virtualisation matérielle. Nous proposons également une première preuve de concept pour un noyau Linux x86 sur un système 64 bits qui supporte la technologie de virtualisation d'Intel. Le démonstrateur Hytux est pour l'instant en cours de développement, et nous avons l'intention de le publier en *open source* lorsqu'il sera terminé.

Bien que nous ne puissions actuellement évaluer précisément le ralentissement du système induit par notre hyperviseur, nous pouvons tout de même l'estimer grossièrement sur la base de quelques considérations. Notre hyperviseur n'effectue que fondamentalement peu de travail : il ne fait que vérifier certaines contraintes et rend ensuite directement la main au noyau. De plus, l'impact sur les performances du système dépend également de la façon dont se comportent les extensions matérielles de virtualisation (c'est-à-dire de la rapidité d'exécution des VM-exit, VM-entry et des injections d'événements). À ce niveau, nous pouvons regarder les hyperviseurs existants qui mettent à profit la virtualisation matérielle (tel que KVM – Kernel Based Virtual Machine [22]). Ces solutions ne causent pas de ralentissements majeurs du système et par conséquent des résultats similaires sont attendus avec notre approche.

Nous travaillons également sur une solution basée hyperviseur qui protège le noyau des actions malveillantes de la Classe 1.4. De plus, afin de valider notre approche fondée sur la préservation des objets contraints par le noyau (KCO), nous développons en ce moment un formalisme afin de représenter les interactions entre la plateforme matérielle et les différentes couches logicielles (dans notre cas les couches de l'hyperviseur, du noyau et de l'espace utilisateur). Nous espérons que cette formalisation nous aidera à vérifier si notre approche est efficace pour préserver l'intégrité de l'espace noyau. Nous essayons également de rendre le modèle utile pour la représentation d'objets contraints par le noyau dès lors de l'étape de spécification du noyau.

Références

1. anonymous. Once upon a free()... *Phrack*, 57, 2001.
2. Adam Boileau. Hit by a bus : Physical access attacks with firewire. In *Ruxcon 2006*, 2006.
3. BSDaemon, coideloko, and D0nAnd0n. System management mode hacks. *Phrack*, 65, 2008.
4. Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 56, 2000.
5. c0de. Reverse symbol lookup in linux kernel. *Phrack*, 61, 2003.
6. Silvio Cesare. Kernel function hijacking. 1999.
7. Jonathan Corbet. The rest of the vmsplice() exploit story, 2008. http://lwn.net/Articles/271688/.
8. Jonathan Corbet. vmsplice() : the making of a local root exploit, 2008. http://lwn.net/Articles/268783/.
9. Microsoft Corporation. Digital signatures for kernel modules on systems running Windows Vista. Technical report, Microsoft Corporation, 2006.
10. Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. Stackguard : Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, 1998.
11. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. Pointguard : Protecting pointers from buffer overflow vulnerabilities. In *12th USENIX Security Symposium*, 2003.
12. Solar Designer. Getting around non-executable stack, 1997. http://seclists.org/bugtraq/1997/Aug/0063.html.
13. Maximillian Dornseif et al. Firewire - all your memory are belong to us. In *CanSecWest/core05*, 2005.
14. Loic Duflot and Laurent Absil. Programmed i/o accesses : a threat to virtual machine monitors ? In *PacSec 2007*, 2007.
15. Loic Duflot, Daniel Etiemble, and Olivier Grumelard. Using cpu system management mode to circumvent operating system security functions. In *CanSecWest/core06*, 2006.
16. Eric Filiol. *Computer Viruses : from theory to applications*. IRIS International Series. Springer Verlag France, 2005.
17. Greg Hoglund and Gary McGraw. *Exploiting Software - How to break code*. Pearson Education. Addison-Wesley, 2004.
18. Intel. *Intel Virtualization Technology for Directed I/O - Architecture Specification*, 2007.

19. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A : System Programming Guide, Part 1*, 2008.
20. Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B : System Programming Guide, Part 2*, 2008.
21. Intel. *Intel Trusted Execution Technology - Measured Launched Environment Developer's Guide*, 2008.
22. Avi Kivity et al. Kvm : the linux virtual machine monitor. In *Linux Symposium*, 2007.
23. Eric Lacombe. Le fonctionnement de pax : Protection against execution. *GNU/Linux Magazine France*, 79, 2006. http://www.unixgarden.com/index.php/securite/le-fonctionnement-de-pax-protection-against-execution.
24. Eric Lacombe, Fréderic Raynal, and Vincent Nicomette. Rootkit modeling and experiments under linux. *Journal in Computer Virology*, 4 :137–157(21), May 2008. http://www.ingentaconnect.com/content/klu/11416/2008/00000004/00000002/00000069.
25. Nergal. The advanced return-into-lib(c) exploits : Pax case study. *Phrack*, 58, 2001.
26. David R. Piegdon. Hacking in physically addressable memory - a proof of concept. In *Easterhegg 2008*, 2008.
27. Joost Pol. [pine-cert-20040201] reference count overflow in shmat(), 2004. http://seclists.org/bugtraq/2004/Feb/0140.html.
28. pragmatic and THC. (nearly) complete linux loadable kernel modules. the definitive guide for hackers, virus coders and system administrators, 1999.
29. Joanna Rutkowska. Subverting Vista Kernel For Fun And Profit. In *Black Hat in Las Vegas 2006*, 2006.
30. Joanna Rutkowska. Beyond the cpu : Defeating hardware based ram acquisition tools (part i : Amd case). In *Black Hat DC 2007*, 2007.
31. sd and devik. Linux on-the-fly kernel patching without lkm. *Phrack*, 58, 2001.
32. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *CCS '04 : Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, New York, NY, USA, 2004. ACM.
33. Brad Spengler. Pax's uderef - technical description and benchmarks, 2007. http://www.grsecurity.net/~spender/uderef.txt.
34. Brad Spengler et al. Grsecurity features.
35. Brad Spengler et al. Pax documentation.
36. sqrkkyu and twzi. Attacking the core : Kernel exploiting notes. *Phrack*, 64, 2007.
37. truff. Infecting loadable kernel modules. *Phrack*, 61, 2003.