Sécurité des noyaux de systèmes d'exploitation

Eric Lacombe

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par l'Institut National des Sciences Appliquées de Toulouse
Discipline ou spécialité : Systèmes Informatiques

Présentée et soutenue par Éric Lacombe
Le 15 décembre 2009

Sécurité des noyaux de systèmes d'exploitation

JURY

Président :	Daniel Hagimont
Rapporteurs :	Jean-Louis Lanet Jean-Jacques Quisquater
Examinateurs :	Cédric Blancher Loïc Duflot Frédéric Raynal
Directeurs de Thèse :	Yves Deswarte Vincent Nicomette

École doctorale : EDSys
Unité de recherche : LAAS - CNRS
Directeurs de Thèse : Yves Deswarte et Vincent Nicomette

Remerciements

Les travaux présentés dans ce mémoire ont été effectués au Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS) du Centre National de la Recherche Scientifique (CNRS). Je remercie Messieurs Malik Ghallab et Raja Chatila, qui ont assuré la direction du LAAS-CNRS depuis mon entrée, pour m'avoir accueilli au sein de ce laboratoire. Je remercie également Monsieur Jean Arlat et Madame Karama Kanoun, Directeurs de Recherche CNRS, responsables successifs du groupe de recherche Tolérance aux fautes et Sûreté de Fonctionnement informatique (TSF), pour m'avoir permis de réaliser ces travaux dans ce groupe.

Je remercie mes directeurs de thèse Yves Deswarte et Vincent Nicomette pour m'avoir guidé durant ces trois années de thèse : Yves, pour sa rigueur, sa grande culture en sécurité informatique, et son sens de l'écoute ; Vincent, pour sa disponibilité malgré un emploi du temps très chargé, son écoute et ses compétences qu'il cache toujours derrière son énorme modestie ;) Travailler avec eux pendant ces trois années fut très appréciable tout au long de cette thèse, mais également une grande chance.

Merci aux membres du jury qui ont accepté de juger mon travail, c'est pour moi un grand honneur :

- Cédric Blancher, Ingénieur de recherche à EADS IW, Suresnes ;
- Yves Deswarte, Directeur de recherche au LAAS-CNRS, Toulouse ;
- Loïc Duflot, Ingénieur de recherche à l'ANSSI, Paris ;
- Daniel Hagimont, Professeur à l'INP/ENSEEIHT, Toulouse ;
- Jean-Louis Lanet, Professeur à l'université de Limoges ;
- Vincent Nicomette, Maître de conférence à l'INSA de Toulouse ;
- Jean-Jacques Quisquater, Professeur à l'université catholique de Louvain ;
- Frédéric Raynal, Ingénieur de recherche à Sogeti, Paris.

Je suis ravi et honoré que Jean-Jacques Quisquater et Jean-Louis Lanet aient accepté d'être les rapporteurs de cette thèse, je les en remercie profondément. Je leur suis très reconnaissant pour tout l'intérêt qu'ils ont porté à mes travaux.

J'adresse également mes sincères remerciements à Éric Filiol, directeur du laboratoire de cryptologie et de virologie opérationnelle de Laval, pour ses commentaires constructifs concernant la première version de mon manuscrit.

Ces quelques lignes supplémentaires de remerciements vont à Frédéric Raynal, rédac' chef de MISC Magazine, mon ancien mentor à EADS Innovation Works (durant mon stage ingé). Il m'a initié à la rigueur de la recherche, convaincu de me lancer dans une thèse, soutenu, mais il m'a également permis de rencontrer des gens très talentueux (tout comme il l'est ;) ce qui a été pour moi une source d'inspiration et de motivation inestimable.

J'en profite pour remercier l'équipe SSI de EADS IW sans qui cette thèse n'aurait pas été possible et notamment Cédric Blancher pour son soutien indéflectible lors de sa mise en place.

Ma thèse n'aurait pas vu le jour si Denis Bodor, rédac' chef de GNU/Linux Magazine, ne m'avais pas permis de connaître Frédéric Raynal. L'histoire remonte à 2002, lorsqu'il a accepté

de publier mon tout premier article. Depuis lors je ne cesse d'écrire pour son magazine ;) Un grand merci à lui !

Mes remerciements à tous les membres du groupe TSF et notamment les doctorants pour toutes les discussions et pauses café que l'on a eu ;) Vous avez su créer une ambiance détendue. Il est toujours très appréciable de savoir que l'on est pas tout seul dans cette barque qui navigue durant trois ans et qui nous fait vivre un voyage si tumultueux.

Je souhaite écrire encore quelques mots de remerciements à ma famille, et mes parents plus particulièrement, pour m'avoir soutenu dans cette entreprise. Je leur dois mon envie d'aller toujours de l'avant, leur soutien inconditionnel dans toutes les épreuves, mais bien d'autres choses encore ...

Enfin, je voulais remercier tous mes amis rencontrés en prépa, pour leur soutien, leur humour et tous ces moments inoubliables passés ensemble, toutes ces soirées, fêtes, festivals, vacances, mais également pour toutes les discussions jusqu'à pas d'heure, toutes les plaisanteries, farces, bêtises, conneries qu'on a pu faire. À tous un grand Merci !

« I believe in intuition and inspiration. Imagination is more important than knowledge. For knowledge is limited, whereas imagination embraces the entire world, stimulating progress, giving birth to evolution. »

— Albert Einstein

Table des matières

Table des figures

Introduction

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. Pour réduire les coûts de développement, les systèmes informatiques utilisent de plus en plus de composants sur étagère (ou COTS pour *Components Off-The-Shelf*), même pour des applications plus ou moins critiques. Ces composants peuvent être des matériels (microprocesseurs, *chipsets*), des logiciels de base (systèmes d'exploitation, outils de développement ou de configuration), des applications génériques (base de données, serveurs Web) ou dédiées à des fonctions plus spécifiques (pare-feux, commande et contrôle). Ces composants étant conçus pour être génériques, ils sont souvent beaucoup plus complexes que ce qui est vraiment utile pour un système particulier. Cette complexité les fragilise vis-à-vis de la sécurité. En effet, il est pratiquement impossible de les vérifier parfaitement, c'est-à-dire de garantir d'une part l'absence de bogues, et d'autre part l'absence de fonctions cachées. Par conséquent, des pirates peuvent profiter de cette situation pour réussir à pénétrer dans ces systèmes, et d'en utiliser les ressources associées. Ces pirates 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. Cela peut même aller jusqu'à la prise de contrôle complète du système par l'attaquant. Le noyau d'un système d'exploitation doit donc 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.

Dans nos travaux nous nous focalisons sur la sécurité *en vie opérationnelle* des systèmes d'exploitation de type COTS. Nous partons du principe que ces systèmes contiennent toujours des vulnérabilités. L'étude publiée dans [Chou *et al.*, 2001] tend à confirmer cette hypothèse[1]. Notamment, elle conclut sur la présence de failles de sécurité bien plus nombreuses dans les

[1] Cette étude a été menée sur le noyau Linux pour les versions courant de 1994 à 2001.

pilotes de périphériques que dans le reste du noyau[2], et qu'avant d'être corrigées, ces failles restent en moyenne dans le noyau 1,8 années. En ce qui concerne les logiciels applicatifs, il est possible d'implémenter des mécanismes de sécurité efficaces dans la mesure où ces mécanismes sont intégrés 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 semble évident de le faire depuis un mode plus privilégié que le noyau et de façon incontournable (par le noyau lui-même, les applications ou encore les périphériques). Il nous paraît alors nécessaire de reposer sur les fonctionnalités du matériel. D'une part, le matériel constitue la base essentielle au fonctionnement de tout système informatique : les logiciels reposent intégralement sur ces bases pour leur exécution. D'autre part, les composants matériels sont plus difficiles à corrompre que les composants logiciels. Leur corruption implique le plus souvent un accès physique au système informatique, dont le problème relève de la sécurité physique et organisationnelle. C'est pourquoi, dans notre thèse, nous ne considérons que les attaques logiques.

2 Présentation de nos travaux

Nous commençons ce manuscrit par un premier chapitre rappelant la terminologie et les concepts de base associés à la sécurité des systèmes d'information. Ceci nous permet alors de proposer une caractérisation des attaques sur les systèmes informatiques, afin de mieux cerner dans quel contexte s'articule la sécurité des systèmes d'exploitation. Nous nous focalisons sur les attaques dont la réalisation implique l'altération du système d'exploitation, et plus particulièrement de son noyau. Ces attaques reposent sur la combinaison de différentes actions dont certaines sont malveillantes. Nous présentons également la classification de ces actions que nous avons établie suite à une étude des différentes sources à l'origine de la corruption d'un noyau.

Dans le deuxième chapitre, afin de mieux appréhender la sécurité de ces noyaux, nous adoptons le point de vue de l'attaquant. Nous étudions alors un type de "logiciel malveillant" dont les actions reflètent en partie notre classification : les *rootkits* "noyau". Après un tour d'horizon des différentes techniques employées par ces parasites informatiques, nous les confrontons aux infections informatiques classiques afin d'en montrer la singularité. Nous proposons ensuite un modèle de ces *rootkits* et identifions trois de leurs attributs essentiels (invisibilité, robustesse, pouvoir de nuisance) afin d'en évaluer les performances. Enfin, fort de ces connaissances, nous présentons notre propre *rootkit* (pour une cible Linux) qui met en œuvre les méthodes originales de détournement du noyau que nous avons élaborées.

Après nous être mis dans la peau de l'attaquant, nous examinons dans un troisième chapitre les parades existantes relativement à notre classification. Bien qu'à la frontière de notre sujet, nous abordons tout d'abord brièvement les moyens actuels mis en œuvre pour éliminer les vulnérabilités en phase de développement. Nous présentons ensuite les approches de prévention, de détection et restauration, qui s'appliquent à nos différentes classes d'actions malveillantes.

[2] Les raisons principales pour cela sont que : (1) ce 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.

Finalement, dans notre dernier chapitre nous exposons notre approche afin de pallier les manques des solutions de protections actuelles vis-à-vis des actions malveillantes ciblant les noyaux de système d'exploitation. Notre démarche est de limiter les risques de corruption d'un noyau, en préservant des contraintes associées aux éléments essentiels dont il dépend pour assurer sa fonction. Ces éléments peuvent lui être internes (la table des interruptions, son code, etc.), et d'autres lui sont externes (les différents registres du processeur, ceux du contrôleur mémoire, etc.). Afin de mettre en œuvre notre principe de préservation de contraintes, notre solution s'appuie sur les extensions matérielles de support à la virtualisation (des processeurs Intel x86), afin d'agir à un niveau de privilège supérieur à celui du noyau. Notre démonstrateur a d'ailleurs été développé pour une cible Linux x86 64 bits.

Nous concluons par une synthèse de nos contributions et présentons quelques perspectives à notre travail. Parmi ces perspectives, est notamment proposée la poursuite de notre tentative de modélisation des systèmes informatiques. Cette modélisation n'ayant pas abouti, nous la présentons en annexe de ce manuscrit.

Chapitre 1

Les actions malveillantes qui affectent l'intégrité du noyau du système d'exploitation

La motivation associée à la caractérisation d'une attaque sur un système informatique est de fournir un modèle sur lequel on peut s'appuyer pour concevoir et mettre en place des mécanismes visant à protéger un système vis-à-vis de malveillances. L'objectif poursuivi dans cette caractérisation est d'identifier ce sur quoi repose une attaque afin de mettre en évidence les différents points d'actions "stratégiques" où le système doit intervenir pour assurer sa protection.

Les différents modèles d'attaques que nous trouvons dans la littérature comme les arbres d'attaques [Schneier, 1999], les graphes d'attaques [Sheyner, 2004], ou encore les graphes de privilèges [Dacier et Deswarte, 1994] ne mettent pas suffisamment l'accent sur les aspects que nous venons de mentionner. À cela se rajoute le fait que la sécurité d'un système n'est pas définie de façon uniforme sur tous les modèles. Mis à part les modèles traitant de types particuliers d'attaques, les modèles plus génériques se fondent sur des hypothèses que nous trouvons trop restrictives (au niveau de la définition des propriétés de sécurité), ou au contraire trop floues.

Aussi, dans notre démarche, nous repartons des concepts de base associés à la sécurité des systèmes d'information. Nous poursuivons alors sur la caractérisation des attaques logiques sur un système particulier : le système informatique. Enfin, nous restreignons notre étude aux cas des attaquants visant uniquement un sous-système de ces systèmes informatiques : les noyaux de systèmes d'exploitation. Afin d'aborder plus sereinement cette dernière partie, nous introduisons dans un premier temps la notion de *système d'exploitation* et de *noyau* ; et dans un second temps, l'architecture des ordinateurs actuels de type IA-32 et Intel 64, sur laquelle nos travaux se sont concentrés.

La caractérisation des actions malveillantes ciblant le noyau d'un système d'exploitation a abouti à une classification de ces actions, sur laquelle reposent les chapitres suivants.

1.1 La sécurité des systèmes

1.1.1 Concepts de base et terminologie des systèmes

Avant de définir la notion de sécurité de systèmes, il convient de définir au préalable ce que nous entendons par le terme *système*. Nous prenons comme référence les définitions de [Laprie, 2004].

Un *système* est une entité qui interagit avec d'autres entités, donc d'autres systèmes, y compris le matériel, le logiciel, les humains, et le monde physique avec ses phénomènes naturels. Ces autres systèmes qui interagissent avec le système considéré constituent l'*environnement* du système considéré. La *frontière* du système est la limite commune entre le système et son environnement.

La *fonction* d'un système est ce à quoi il est destiné. Elle est décrite par la *spécification fonctionnelle*, qui inclut les performances attendues du système. Le *comportement* d'un système est ce que le système fait pour accomplir sa fonction, et est décrit par une séquence d'états. L'*état total* d'un système est l'ensemble des états de traitement, de communication, de mémorisation, d'interconnexion, ainsi que des conditions physiques.

La *structure* d'un système est ce qui lui permet de générer son comportement. D'un point de vue structurel, un système est constitué d'un ensemble de composants interconnectés en vue d'interagir. Un composant est un autre système, etc.

Le *service* délivré par un système est son comportement tel que perçu par ses *utilisateurs* ; un *utilisateur* est un autre système, éventuellement humain, qui interagit avec le système considéré. La partie de la frontière du système où ont lieu les interactions avec ses utilisateurs est l'*interface de service*. La partie de l'état total qui est perceptible à l'interface de service est son *état externe* ; le reste est son *état interne*.

Enfin, fonction et service peuvent être vus comme constitués de *fonctions élémentaires* et de *services élémentaires*.

De ces concepts nous déduisons qu'un système est alors entièrement défini par sa structure et son état total.

1.1.2 La sûreté de fonctionnement

1.1.2.1 Définitions de base

La sécurité des systèmes s'inscrit dans le cadre plus large de la *sûreté de fonctionnement*. [Laprie, 2004] nous en donne la définition suivante.

La *sûreté de fonctionnement* d'un système est son aptitude à délivrer un service de *confiance justifiée*. Cette définition met l'accent sur la justification de la *confiance*, cette dernière pouvant être définie comme une *dépendance acceptée*, explicitement ou implicitement. La *dépendance* d'un système d'un autre système est l'influence, réelle ou potentielle, de la sûreté de fonctionnement de ce dernier sur la sûreté de fonctionnement du système considéré.

Selon les applications auxquelles le système est destiné, l'accent peut être mis sur différentes facettes de la sûreté de fonctionnement, ce qui signifie que la sûreté de fonctionnement peut être

vue selon des propriétés différentes mais complémentaires, qui permet de définir ses attributs : la *fiabilité*, la *sécurité-innocuité*, la *maintenabilité*, la *disponibilité*, la *confidentialité* et l'*intégrité*.

L'association, à la confidentialité, de l'intégrité et de la disponibilité vis-à-vis des actions autorisées, conduit à la sécurité-immunité (ou *security* en anglais), pour la distinguer de la sécurité-innocuité.

1.1.2.2 Entraves de la sûreté de fonctionnement

Un *service correct* est délivré par un système lorsqu'il accomplit sa fonction. Une *défaillance du service*, souvent simplement dénommé *défaillance*, est un évènement qui survient lorsque le service délivré dévie du service correct. Le service délivré étant une séquence d'états externes, une défaillance du service signifie qu'au moins un état externe dévie du service correct. La déviation est une erreur. La cause adjugée ou supposée d'une erreur est une faute. Les fautes peuvent être internes ou externes au système. La présence antérieure d'une vulnérabilité, c'est-à-dire d'une faute interne qui permet à une faute externe de causer des dommages au système, est nécessaire pour qu'une faute externe entraîne une erreur, et, éventuellement, une défaillance.

1.1.2.3 Moyens de la sûreté de fonctionnement

Les moyens de la sûreté de fonctionnement sont les méthodes et techniques permettant de fournir au système l'aptitude à délivrer un service conforme à l'accomplissement de sa fonction, et de donner confiance dans cette aptitude. Cet ensemble de méthodes et techniques peuvent être classées en :

- *prévention des fautes* : comment empêcher l'occurrence ou l'introduction de fautes ;
- *tolérance aux fautes* : comment fournir un service à même de remplir la fonction du système en dépit des fautes ;
- *élimination des fautes* : comment réduire la présence (nombre, sévérité) des fautes ;
- *prévision des fautes* : comment estimer la présence, le taux futur, et les possibles conséquences des fautes.

1.1.3 La sécurité

Dans la suite de ce mémoire, nous nous intéresserons uniquement à la sécurité-immunité que, dés à présent, nous référencerons par le seul mot sécurité. Assurer la sécurité d'un système, c'est assurer que les propriétés, confidentialité, intégrité et disponibilité sont toujours vérifiées.

Notons dès à présent que les trois propriétés de la sécurité se rapportent à l'information, et le terme d'information doit être pris ici dans son sens le plus large, couvrant non seulement les données et les programmes, mais aussi les flux d'information, les traitements et la connaissance de l'existence de données, de programmes, de traitements, de communications. Cette notion d'information doit aller jusqu'à couvrir le système informatique lui-même, dont parfois l'existence doit être tenue secrète (confidentialité).

1.1.3.1 La confidentialité

La *confidentialité* est la propriété d'une information de ne pas être révélée à des utilisateurs non autorisés à la connaître. Ceci signifie que le système informatique doit :

- empêcher que des utilisateurs puissent connaître une information confidentielle s'ils n'y sont pas autorisés ;
- empêcher que des utilisateurs autorisés à connaître une information confidentielle puissent la divulguer à des utilisateurs non autorisés.

Assurer la confidentialité d'un système est un travail bien plus colossal qu'il n'y parait à première vue. Il faut analyser tous les chemins que peut prendre une information dans le système pour s'assurer qu'ils sont sécurisés. C'est un travail fastidieux et souvent trop coûteux par rapport aux besoins réels ; aussi dans bon nombre de cas, on s'occupera de sécuriser un certain nombre des chemins que peut prendre l'information.

Les attaques contre la confidentialité consistent à essayer d'obtenir des informations malgré les moyens de protection et les règles de sécurité. Ces attaques peuvent être passives ou actives. Les attaques passives consistent à accéder aux informations qui sont générées, transmises, stockées ou affichées dans des composants vulnérables du système informatique (voies de communication, mémoires, disques, etc.). Par exemple, lorsqu'un utilisateur U se connecte sur une machine B depuis une autre machine A, une écoute passive permet de capter l'identité et le mot de passe de l'utilisateur U tapés depuis la machine A et transmis jusque la machine B. Cette capture permettra à un intrus de se connecter sur la machine B en prenant l'identité de l'utilisateur U. Un attaque active nécessite quant à elle, une action d'un ou plusieurs individus dans le système. L'utilisation des canaux cachés [Lampson, 1973] peut permettre ainsi à un individu autorisé à accéder à des informations de les transmettre à un utilisateur non autorisé à y accéder. Il peut pour cela utiliser par exemple des canaux de mémoire (réutilisation de zones mémoires non réinitialisées) ou des canaux temporels (modulation de l'utilisation des ressources communes comme l'unité centrale ou les disques).

1.1.3.2 L'intégrité

L'*intégrité* est la propriété d'une information d'être correcte. Cela signifie que le système informatique doit :

- empêcher une modification indue de l'information, c'est-à-dire une modification[1] par des utilisateurs non autorisés ou une modification incorrecte par des utilisateurs autorisés ;
- faire en sorte qu'aucun utilisateur ne puisse empêcher la modification légitime de l'information ; par exemple, empêcher la mise à jour périodique d'un compteur de temps serait une atteinte contre l'intégrité ;
- faire en sorte que l'information soit créée et vérifier qu'elle est correcte.

L'interception est un exemple d'attaque contre l'intégrité. Elle consiste à accéder à des données transmises sur des voies de communication et à les modifier (en les détruisant, en les modifiant ou en y insérant des messages). Un exemple classique d'interception concerne les messages échangés entre deux machines utilisant le protocole IP (*Internet Protocol*). Chaque paquet IP contient une adresse source (la machine émettant le message) et d'une adresse de

[1] Le terme de modification doit être entendu au sens large, comprenant à la fois la création d'une nouvelle information, la mise à jour d'une information existante et la destruction d'une information.

destination (l'adresse de la machine à qui est destiné le message). Des interceptions de ces messages ont été réalisées de telle façon à changer l'adresse source dans le message. La machine cible croit alors recevoir un paquet d'une machine alors que c'est une autre qui l'a émis.

Un *virus* est également un bon exemple d'attaque contre l'intégrité d'un système. Un virus est un programme qui, lorsqu'il s'exécute, se propage et se reproduit pour s'adjoindre à un autre programme. Le virus sera exécuté (souvent en fonction de certaines conditions) lorsque le programme sur lequel il s'est adjoint sera lancé. La propagation du virus est une attaque contre l'intégrité des programmes, l'exécution du virus pouvant par ailleurs avoir d'autres effets qui peuvent être des attaques contre la confidentialité, l'intégrité et la disponibilité.

1.1.3.3 La disponibilité

La *disponibilité* est la propriété d'une information d'être accessible lorsqu'un utilisateur autorisé en a besoin. Cela signifie que le système informatique doit :

- fournir l'accès à l'information pour que les utilisateurs autorisés puissent la lire ou la modifier ;
- faire en sorte qu'aucun utilisateur ne puisse empêcher les utilisateurs autorisés d'accéder à l'information.

La disponibilité implique l'intégrité, puisqu'il ne servirait à rien de rendre accessible une information fausse. La disponibilité implique également des contraintes plus ou moins précises sur le temps de réponse du système. La propriété de disponibilité s'applique aussi au *service* fourni par le système informatique et une atteinte contre la disponibilité est souvent appelée *déni de service*.

Le déni de service est souvent le type d'attaque le plus facile, puisqu'il suffit d'émettre des requêtes, valides ou non, en très grand nombre (on parle alors d'inondation ou de *flooding*), de façon à saturer les ressources disponibles pour un service donné. Pour multiplier le nombre de requêtes, un attaquant méthodique peut d'abord prendre le contrôle d'un très grand nombre de machine mal protégées (par exemple des ordinateurs personnels), puis d'y installer un "zombie", logiciel qui lancera automatiquement ou sur commande un grand nombre de requêtes vers une même cible.

Enfin, il est important de souligner que la disponibilité doit tenir compte à la fois des fautes intentionnelles mais aussi des fautes accidentelles. Il en est de même pour l'intégrité. Dans les techniques de conception classique de systèmes sécurisés, la disponibilité est souvent négligée et est souvent uniquement considérée du point de vue des fautes intentionnelles.

1.1.4 Politiques de sécurité

L'ensemble des propriétés de sécurité que l'on désire assurer dans un système ainsi que la façon dont on va les assurer sont définies dans la *politique de sécurité* du système [European Communities, 1991]. Elle est définie d'une part par un ensemble de propriétés de sécurité qui doivent être satisfaites par le système, et d'autre part par un schéma d'autorisation qui présente les règles permettant de modifier l'état de protection du système. Par exemple, une propriété de sécurité pourra être "une information classifiée ne doit pas être transmise à un utilisateur non habilité à la connaître", alors qu'une règle du *schéma* d'autorisation pourra être "le propriétaire

d'une information peut accorder un droit d'accès pour cette information à n'importe quel utilisateur". Si la politique d'autorisation est cohérente, il ne doit pas être possible, partant d'un état initial sûr (c'est-à-dire satisfaisant les propriétés de sécurité), d'atteindre un état d'insécurité (c'est-à-dire un état où les propriétés de sécurité ne sont pas satisfaites) en appliquant le schéma d'autorisation. Notons que les propriétés de sécurité peuvent être définies en fonction de la confidentialité, l'intégrité ou la disponibilité d'informations ou de méta-informations.

La plupart des politiques de sécurité sont basées sur les notions de sujets, d'objets et de droits d'accès[2]. Un *sujet* est une entité active, correspondant à un processus qui s'exécute pour le compte d'un utilisateur. Dans ce contexte, un *utilisateur* est une personne physique connue du système informatique et enregistrée comme utilisateur, ou un *serveur*, sorte de personne morale représentant des fonctions de service automatiques, tel que serveur d'impression, serveur de base de données, serveur de messagerie, etc. Un *objet* est une entité considérée comme "passive" qui contient ou reçoit des informations (cette notion d'objet est à prendre dans un sens large, pouvant recouvrir les notions classiques d'objet des systèmes dits orientés-objets et incluant les sujets eux-mêmes). À un instant donné, un sujet a un *droit d'accès* sur un objet si et seulement si le processus correspondant au sujet est autorisé à exécuter l'opération correspondant à ce type d'accès sur cet objet. Les droits d'accès sont généralement représentés sous la forme d'une *matrice de droits d'accès*, avec une ligne par sujet, une colonne par objet, chaque case de la matrice contenant la liste des droits que possède (à un instant donné) le sujet défini par la ligne sur l'objet défini par la colonne.

Les politiques de sécurité sont généralement décrites par un modèle formel, ce qui permet de vérifier que la politique est complète et cohérente, et que la mise en œuvre par le système de protection est conforme. Il existe deux types de modèle :

- des modèles spécifiques, développés pour représenter une politique d'autorisation particulière, comme les modèles de Bell-LaPadula [Bell et LaPadula, 1974], de Biba [Biba, 1977], de Clark & Wilson [Clark et Wilson, 1987], etc. ;
- des modèles généraux, qui sont plutôt des méthodes de description formelle, pouvant s'appliquer à toute sorte de politiques, comme le modèle HRU [Harrison *et al.*, 1976] ou le modèle Take-Grant [Jones *et al.*, 1976].

Les politiques de sécurité ou plus précisément leurs schémas d'autorisation, se classent en deux catégories : les politiques discrétionnaires (correspondant au DAC : *Discretionary Access Control*) et les politiques obligatoires (correspondant au MAC : *Mandatory Access Control*).

Dans le cas d'une politique discrétionnaire, les droits d'accès à chaque information sont manipulés librement par le responsable de l'information (généralement le propriétaire), à sa *discrétion*. Les droits peuvent être accordés par ce responsable à chaque utilisateur individuellement ou à des groupes d'utilisateurs, ou les deux. Ceci peut parfois amener le système dans un état d'insécurité (c'est-à-dire contraire à la politique de sécurité qui a été choisie).

Les politiques dites "obligatoires" imposent, par leur schéma d'autorisation, des règles incontournables qui s'ajoutent aux règles discrétionnaires. Ces règles sont bien sûr différentes selon qu'il s'agit de maintenir des propriétés liées à la confidentialité ou à l'intégrité.

[2] Certaines politiques ne considèrent que des flux d'information entre sujets, sans référence à des objets.

1.1.5 Évaluation de la sécurité

1.1.5.1 Trois méthodes d'évaluation

Il est important d'évaluer la sécurité des systèmes d'information et de communication, pour savoir si on a obtenu un niveau de sécurité satisfaisant, pour identifier les points les plus critiques à surveiller ou à corriger, et enfin pour estimer s'il est rentable de mettre en œuvre telle ou telle défense supplémentaire. La première méthode d'évaluation est l'*analyse de risques*, qui consiste à identifier les menaces auxquelles devra faire face le système (quels types d'attaquants, mais aussi quels phénomènes physiques : incendie, inondation, etc.), identifier les *vulnérabilités* du système face à ces menaces, et estimer les conséquences (surtout financières) qui résulteraient de la réalisation des menaces et de l'exploitation des vulnérabilités. Les risques sont alors évalués à partir d'une évaluation de la fréquence de réalisation des menaces et des conséquences. En France, les méthodes d'analyse de risques (dédiées aux systèmes informatiques) les plus connues sont la méthode MARION, la méthode MELISA, et enfin la méthode MEHARI, plus récente, préconisée par le CLUSIF.

La deuxième méthode d'évaluation est basée sur des critères d'évaluation. Les critères portent à la fois sur des fonctionnalités de sécurité (authentification, autorisation, etc.) et sur des niveaux d'assurance, définis en fonction des méthodes de développement et de vérification. Les premiers critères ont été définis par le DoD (*Department of Defense*) des États-Unis dans le Livre Orange [DoD, 1985]. En Europe, les ITSEC [European Communities, 1991] ont été les premiers critères reconnus par les différents pays européens et soutenus par la Commission européenne. Depuis, une coopération internationale a permis de développer les Critères Communs, qui sont maintenant une norme internationale [ISO/IEC, 1999]. En France, l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) a mis en œuvre la méthodologie EBIOS (Expression des Besoins et Identification des Objectifs de Sécurité) [ANSSI, 2004] qui s'appuie sur ces Critères Communs afin d'apprécier et de traiter les risques relatifs à la sécurité des systèmes d'information.

Enfin, une troisième méthode consiste à utiliser des indicateurs quantitatifs de la sécurité, permettant aux administrateurs de surveiller au jour le jour le niveau de sécurité et de prendre des mesures correctives en cas de baisse de ces indicateurs [Ortalo *et al.*, 1999]. Ceci permet d'évaluer la sécurité opérationnelle, c'est-à-dire correspondant à la façon d'utiliser les systèmes et ses mécanismes de protections, plutôt qu'à une vision statique comme celle donnée par l'analyse de risques ou les critères d'évaluation.

Parmi ces trois méthodes, nous donnons dans la suite quelques informations supplémentaires sur les Critères Communs.

1.1.5.2 Les Critères Communs

Les Critères Communs contiennent deux parties bien séparées : fonctionnalité et assurance. Ils définissent une *cible d'évaluation* (*Target of Evaluation* ou TOE) qui désigne le système ou le produit à évaluer, et la *cible de sécurité* (*Security Target* ou ST) qui contient l'ensemble des exigences de sécurité et de spécifications à utiliser comme base pour l'évaluation d'une TOE identifiée. La cible de sécurité pour une TOE représente la base d'entente entre développeurs et évaluateurs. Une cible de sécurité peut contenir les exigences d'un ou de plusieurs *profils de protection* (*Protection Profiles* ou PP) prédéfinis. Une des différences essentielles entre ITSEC

et CC réside dans l'existence des profils de protection, qui avaient auparavant été introduits dans les Critères Fédéraux(*Federal Criteria*) [NIST et NSA, 1992]. Un profil de protection définit un ensemble d'exigences de sécurité et d'objectifs, indépendants d'une quelconque implémentation, pour une catégorie de TOE. L'intérêt de ces profils de protection est double : un développeur peut inclure dans une cible de sécurité un ou plusieurs profils de protection ; un client désirant utiliser un système ou un produit peut également demander à ce que son système corresponde à un profils de protection particulier, ceci lui évitant de donner une liste exhaustive de fonctionnalités et assurances qu'il exige du système ou produit. Une partie importante des Critères Communs est donc consacrée à la présentation détaillée de profils de protection prédéfinis.

Cette notion de PP représente la volonté américaine qui consiste à préférer évaluer des systèmes qui entrent dans le cadre de profils connus. Le cas d'un système ne cadrant pas exactement avec un profil connu conduit simplement à l'élaboration d'un nouveau profil. La tendance européenne serait plutôt de définir systématiquement une TOE et ses exigences de sécurité et d'évaluer cette TOE sans nécessairement s'appuyer sur des profils prédéfinis. Les Critères Communs tentent de réaliser un compromis équitable entre ces deux positions.

1.2 Caractérisation d'une attaque logicielle sur un système informatique

Les propriétés de sécurité définies dans la section 1.1.1 s'appliquent à la notion générale de système. Tout acte sur un système visant à nuire à ces propriétés peut être qualifié de *malveillant*. Aussi nous définissons tout *acte de malveillance* sur un système (ou encore *attaque* sur un système) comme une combinaison d'actions qui ciblent ce système et dont l'*intention* est de nuire au moins à l'une des propriétés de disponibilité, d'intégrité ou de confidentialité, de ce système, ou d'un système ayant une quelconque relation avec lui (par exemple un (sur-)système le contenant, un système externe qui en dépend, etc.).

Dans la suite de cette section nous focalisons notre attention sur la sécurité de systèmes particuliers : les systèmes informatiques que nous définissons comme suit. Un *système informatique* est la composition de deux systèmes, l'un matériel et l'autre logiciel, vérifiant les deux propriétés suivantes :

- le système logiciel appartient à l'état du système matériel ;
- la structure du système matériel est apte à interpréter le système logiciel.

Un système informatique peut alors être un simple ordinateur, ou encore un réseau d'ordinateur. Il est également possible de voir un réseau d'ordinateur comme un système de systèmes informatiques. Les deux visions concordent avec la définition, seule la granularité de la modélisation change. Cependant pour plus de clarté, dans le reste de ce document, lorsque la distinction entre un unique ordinateur et un réseau d'ordinateurs est importante, nous employons respectivement le terme de *système informatique simple*, et de *système informatique complexe*.

De la définition que nous venons de donner, une attaque impactant un système informatique, agit :

- soit sur l'état du système matériel et par la même agit sur la structure ou l'état du système logiciel ;

– soit sur la structure du système matériel.

Si l'on se restreint aux attaques logiques, une action sur la structure du système matériel n'est généralement possible que si ce dernier est un système adaptable par logiciel (par exemple, un système à base de FPGA). Il convient toutefois de mentionner qu'une attaque logique peut entraîner indirectement une altération de la structure du système matériel. En effet, des phénomènes physiques (liés à la température, aux champs électromagnétiques, etc.) peuvent être déclenchés lors d'une attaque logique et aboutir à une dégradation du matériel. Dans nos travaux, nous considérons uniquement les attaques logiques qui n'entraînent pas de tels phénomènes physiques.

Dans la suite de cette section nous proposons une caractérisation des attaques logiques.

1.2.1 Le problème de l'attaquant

Une attaque logique sur un système peut être vue comme l'expression de la solution d'un problème particulier dans lequel figure un objectif de malveillance sur ce système ou sur un système ayant une quelconque relation avec lui. Ce problème que nous pouvons qualifier de *problème de l'attaquant* peut se décomposer : en un objectif de malveillance, et en l'ensemble des informations relatives au contexte de l'attaque.

Notons que généralement un problème de l'attaquant, suivant sa complexité, se décompose en de multiples sous-problèmes, lesquels nécessitent d'être accomplis pour que le *problème de l'attaquant* le soit. Cette hiérarchie n'est d'ailleurs pas sans faire penser aux arbres d'attaques de Schneier [Schneier, 1999].

Quelque soit la nature de l'attaquant (opportuniste, hacktiviste, mafieux, etc.), nous identifions en général six objectifs d'attaque :

1. récupérer des informations contenues dans ou transitant par le système ;
2. introduire des informations fallacieuses dans le système ;
3. bloquer l'accès au système, c'est-à-dire provoquer un *déni de service* (DoS) (la notion de système incluant ici le réseau) ;
4. détourner des services du systèmes ;
5. prendre le contrôle du système afin de l'utiliser pour espionner, pour le faire participer à un (D)DoS (*Distributed* DoS), ou pour le transformer en serveur de contenus illégaux ;
6. rebondir vers d'autres systèmes, auquel cas il sert uniquement de point de transfert vers une autre cible.

L'ensemble des informations relatives au contexte de l'attaque comprend notamment les caractéristiques du système informatique attaqué et du système d'information le contenant, ainsi que d'éventuelles contraintes sur le mode opératoire que devrait suivre l'attaquant.

Notons que les caractéristiques du système informatique attaqué peuvent être groupées suivant leur nature :

– les caractéristiques vis-à-vis de la structure du système matériel du système informatique (architecture des ordinateurs en jeu, topologie du réseau, etc.) ;
– les caractéristiques vis-à-vis de la structure du système logiciel du système informatique (type de système d'exploitation, protections logicielles déployées, etc.) ;

– les caractéristiques vis-à-vis de l'état du système logiciel du système informatique (sessions en cours, protections logicielles activées, etc.).

Quant aux éventuelles contraintes sur le mode opératoire que devrait suivre l'attaquant, elles sont de nature diverse (opérer de façon masquée, laisser volontairement de fausses traces sur le système, etc.). Elles limitent de fait le nombre de solutions possibles pour un *problème de l'attaquant*.

1.2.2 La résolution du problème de l'attaquant

Les informations relatives au contexte de l'attaque ne sont pas forcément toutes connues à l'avance par l'attaquant. En général, seul un sous-ensemble l'est. Suivant l'objectif de malveillance, le niveau de connaissance initial sera suffisant pour l'accomplir. Dans les autres cas, l'attaquant devra acquérir plus d'informations (pouvant alors aboutir à un nouveau *problème de l'attaquant*). Il faut aussi considérer la *compétence* de l'attaquant vis-à-vis du système qu'il doit attaquer, car son niveau de compétence est déterminant quant au succès de l'attaque, c'est-à-dire pour résoudre le problème. Nous retrouvons notamment cette notion de *compétence* dans les ITSEM [European Communities, 1993] qui la définit comme la connaissance nécessaire à une personne pour pouvoir attaquer un système donné.

Pour résoudre son problème, l'attaquant a également besoin de *ressources*, et éventuellement d'*opportunités* (notions également définies dans les ITSEM). Les *ressources* représentent ce qu'un attaquant doit employer pour résoudre son problème (le temps et l'équipement). Les *opportunités* recouvrent des facteurs qui peuvent généralement être considérés comme hors du contrôle de l'attaquant, tels que le cas où l'assistance d'une autre personne est nécessaire (collusion), la possibilité d'un concours de circonstances particulier (chance), et la possibilité et les conséquences de la capture d'un attaquant (détection).

Le processus de résolution du problème de l'attaquant (*cf.* figure 1.1) passe alors par une phase de préparation de l'attaque dans laquelle, suite à l'analyse du problème, l'attaquant acquiert des connaissances sur la cible de son attaque (comme la version du système d'exploitation, les protections présentes, etc.), des ressources (comme l'ordinateur depuis lequel il va mener l'attaque, des programmes malveillants, etc.), et attend éventuellement des opportunités (période des vacances estivales)[3]. Une fois que le niveau de connaissance, les ressources dont il dispose, et les opportunités qui se présentent, lui paraissent satisfaisants, l'attaquant entreprend son attaque. Au cours de l'attaque, il peut toutefois se heurter à des (sous-)problèmes qu'il n'avait pas prévus qui requièrent de nouvelles ressources, de nouvelles opportunités ou encore des informations supplémentaires.

Dans la suite nous ne considérons que la dernière phase du processus de résolution du problème de l'attaquant, à savoir la réalisation de l'attaque elle-même.

[3] Nous supposons dans cette représentation que la compétence de l'attaquant est sans limite.

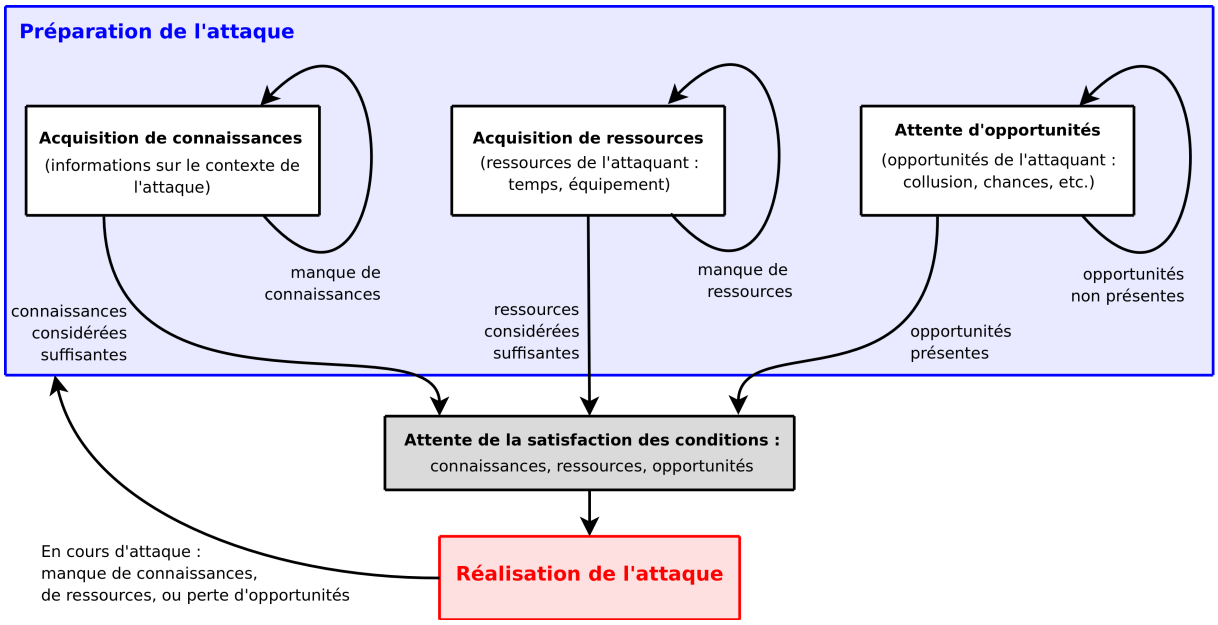

FIG. 1.1 – Processus de résolution du problème de l'attaquant

1.2.3 Modèle générique d'attaque logique sur un système informatique

Nous conjecturons que la combinaison d'actions qui composent une attaque logique peut toujours être décomposée suivant un nombre limité de blocs sémantiques. Nous avançons que pour toute attaque, il est possible de la décomposer à partir d'un nombre restreint de blocs sémantiques. Nous justifions cela par le fait que la façon de construire une solution à un problème de l'attaquant est conditionnée par l'utilisation d'un support très contraint, le système informatique, et également par le fait que les différents objectifs possibles d'un *problème de l'attaquant* sont en nombre très limités.

Nous présentons dans ce qui suit une description (sous forme de graphe) de la forme que toute attaque doit avoir, pour répondre à un quelconque *problème de l'attaquant* (*cf.* figure 1.2). Les différents nœuds du graphe correspondent aux différents blocs sémantiques que nous avons identifiés, ils constituent *les étapes fondamentales* d'une attaque. Une solution est ainsi caractérisée par le chemin qui débute par l'un des deux états initiaux et se termine à l'état final. L'attaque qui correspond à cette solution, est dite *réussie* dès lors que tous les nœuds du chemin emprunté sont franchis avec succès.

Toute attaque logique sur un système informatique peut ainsi être classée d'après ce graphe. Plus précisément, les différents chemins du graphe partant d'un des états initiaux et se terminant à l'état final constituent une partition sur les attaques.

Nous donnons tout d'abord, dans ce qui suit, la sémantique associée aux différentes *étapes fondamentales* d'une attaque. Nous détaillons par la suite différents exemples d'attaques issus de ce graphe.

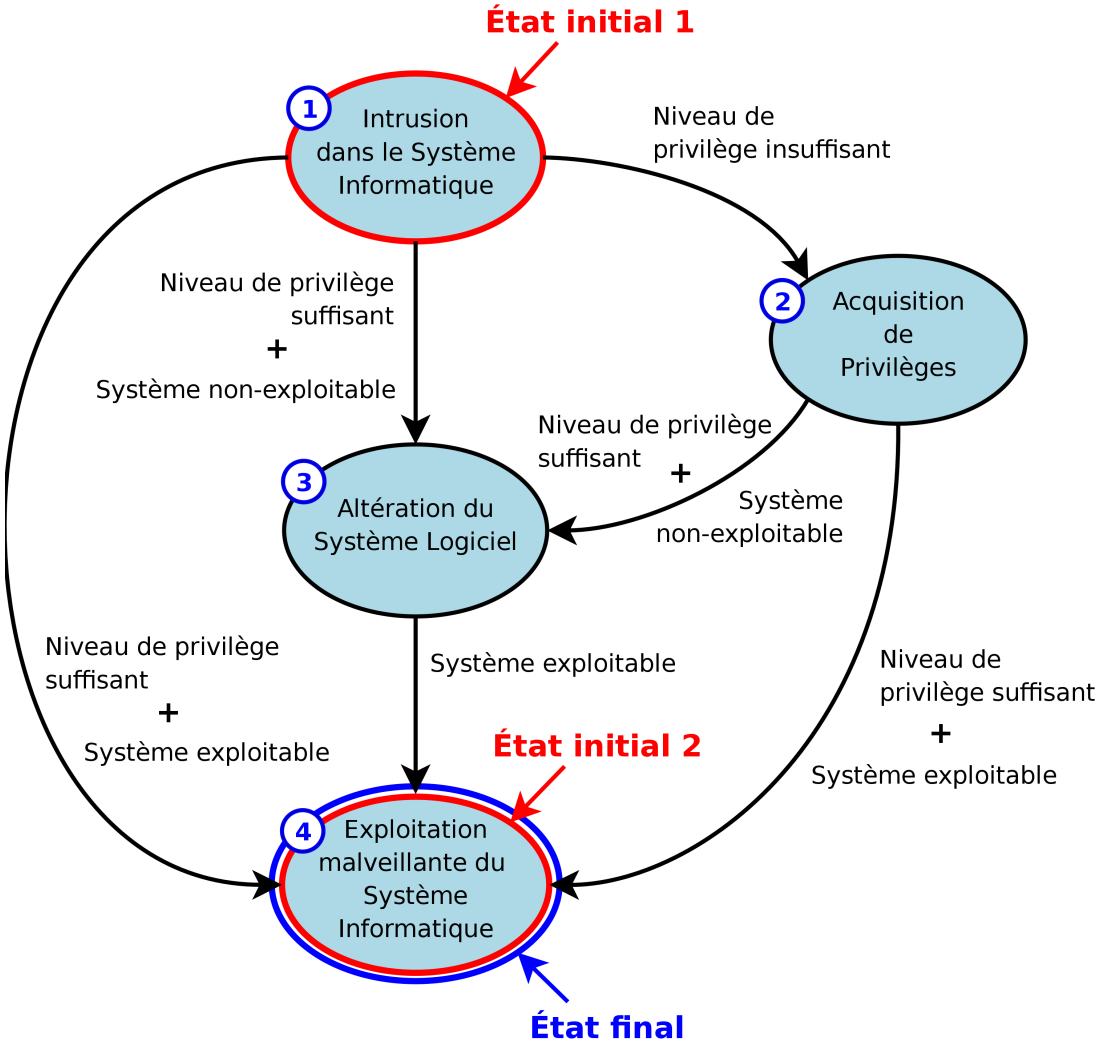

FIG. 1.2 – Modèle générique d'attaque logique sur un système informatique

1.2.3.1 Étape 1 – Intrusion dans le système informatique

Une intrusion dans un système informatique est le fait pour un système externe d'y *pénétrer* alors que sa présence n'y est pas autorisée. Il s'agit dans le cas de la sécurité, d'un système malveillant que nous appelons "attaquant"[4].

Afin de réaliser une intrusion, divers choix peuvent se présenter pour l'attaquant. Par exemple, il peut usurper l'identité d'un utilisateur légitime du système (vol d'identifiant et de mot de passe, attaque par dictionnaire, etc.), ou encore exploiter une faille du système logiciel qui lui permet d'accéder aux services du système informatique. Ces différentes possibilités d'intrusion pour l'attaquant dans le système informatique sont directement liées aux contraintes associées à ce système.

Cette étape est par exemple le point de départ d'une attaque qui installe un *sniffer* sur un

[4] Il peut aussi bien s'agir d'un humain que d'un autre système informatique.

système informatique complexe. L'attaquant souhaite alors récupérer par exemple des mots de passe d'utilisateurs légitimes du système. Alors que le travail du *sniffer* ne requiert aucune intrusion sur le système (il effectue une écoute passive), son installation nécessite une intrusion préalable.

1.2.3.2 Étape 2 – Acquisition de privilèges

Ce nœud du graphe est associé à l'état dans lequel se trouve un attaquant lorsque les privilèges dont il dispose sur un système informatique sont insuffisants pour la réalisation de son objectif. Ainsi, dans cet état, l'attaquant tente d'acquérir les privilèges qui lui sont nécessaires au travers de failles existantes dans le système informatique dans lequel il a pénétré.

Par exemple, un attaquant ayant usurpé l'identité d'un utilisateur légitime du système peut avoir besoin des privilèges d'un administrateur du système pour réaliser son objectif. Il peut également nécessiter des privilèges encore supérieurs comme ceux associés à un noyau de système d'exploitation[5].

1.2.3.3 Étape 3 – Altération du système logiciel

Ce nœud du graphe représente l'état dans lequel se retrouve l'attaquant lorsque le système dans lequel il a pénétré ne dispose pas de la structure adéquate à la réalisation de son objectif, mais que l'attaquant dispose des privilèges nécessaires pour y remédier. Il s'agit alors pour l'attaquant, de modifier la structure et/ou l'état du système logiciel afin que ce dernier soit apte à la réalisation de l'objectif de l'attaquant.

Une altération du système logiciel par un attaquant peut être vu comme l'ajout (ou encore l'installation) d'un ou de plusieurs sous-systèmes malveillants au système logiciel, soit de l'altération d'informations qui impactent la sécurité du système. À noter que tout système peut être qualifié de malveillant dès lors que l'utilisation qui en est faite a pour objectif de réaliser un acte malveillant. Certains systèmes sont cependant intrinsèquement malveillants et peuvent être classés suivant leur nature et leur comportement. Il s'agit des infections informatiques.

Infections informatiques Une infection informatique [Filiol, 2004] est le résultat de l'installation dans un système d'information, à l'insu du ou des utilisateurs, d'un programme à caractère offensif, en vue de porter atteinte à la sécurité de ce système. Nous présentons dans ce qui suit les principales infections informatiques :

Bombe logique Une *bombe logique* est une infection qui attend un évènement (date, action, données particulières, etc.) appelé en général "gâchette" pour exécuter sa fonction offensive.

Cheval de Troie Un *Cheval de Troie* est directement issue de la mythologie grecque, plus précisément de la guerre de Troie[6]. Un cheval de Troie, depuis lors désigne

[5] Dans le cas d'un système matériel d'architecture Intel, le niveau de privilège auquel nous faisons référence ici est le *ring* 0.

[6] Dans sa version la plus connue, après dix ans de siège, les grecs auraient construit une statue gigantesque de cheval dans lequel une troupe de soldats d'élite aurait été dissimulée. Les grecs auraient fait semblant de lever le siège et de repartir en mer. Les troyens auraient alors récupérés la statue, symbole de leur victoire. Cependant, la nuit tombée, la troupe dissimulée dans le "cheval" serait allée ouvrir les portes de la cité au gros de l'armée grecque, lui livrant ainsi la ville.

un artifice quelconque qui entraîne une cible à "inviter" un ennemi dans un endroit sécurisé.

Dans le contexte de la sécurité informatique, un cheval de Troie, désigne un système réalisant une fonction a priori souhaitée par un utilisateur mais qui réalise une fonction pour l'attaquant. Il vise de part sa nature à leurrer sa victime.

Porte dérobée Une *porte dérobée* (*Backdoor/Trapdoor*) est un moyen de contourner les mécanismes de contrôle d'accès. Il s'agit d'une faille du système de sécurité qui provient soit d'une faute de conception (volontaire ou accidentelle[7]), soit d'une altération du système[8] durant sa phase opérationnelle.

Spyware Un *spyware* est une infection dont l'objectif est de divulguer à un attaquant des informations issues du système infecté.

Adware Un *adware* est une infection dont l'objectif est d'envoyer à l'utilisateur du système infecté de la publicité ciblée. Pour cela, il espionne les habitudes de l'utilisateur.

Programmes autoreproducteurs Pour référence nous mentionnons la définition originelle et plus formelle d'un virus informatique, issue des travaux de Fred Cohen et Leonard Adleman [Cohen, 1986; Filiol, 2004] :

"Un *virus* est une séquence de symboles qui, interprétée dans un environnement donné (adéquat), modifie d'autres séquences de symboles dans cet environnement, de manière à y inclure une copie de lui-même, cette copie ayant éventuellement évolué."

Cette définition est assez générale pour regrouper l'ensemble des programmes autoreproducteurs. Néanmoins, il est de coutume dans la littérature de différencier, par analogie avec la biologie, deux classes de programmes autoreproducteurs : les virus et les vers. Cette différentiation tend à s'atténuer avec l'évolution des programmes autoreproducteurs[9] [Filiol, 2004]. Nous donnons dans ce qui suit les définitions usuelles d'un virus et d'un ver.

Virus Un virus est un segment de programme qui, lorsqu'il s'exécute, se reproduit en s'adjoignant à un autre programme. L'analogie avec les virus biologiques tient à leur comportement : un virus biologique ne peut se reproduire par lui-même ; pour se reproduire, il doit modifier le code génétique des cellules qu'il infecte pour que les cellules ainsi modifiées produisent des copies du virus ; de même un virus informatique ne peut être activé (et donc se reproduire) que par l'exécution d'un programme *porteur* du virus.

Vers Un vers (*worm* en anglais) est un programme *autonome* qui se propage sur les réseaux, se reproduit et s'exécute à l'insu des utilisateurs normaux.

Rootkit Un *rootkit* est un système parasite permettant à un attaquant de maintenir dans le temps un contrôle frauduleux sur un système informatique.

[7] À noter que la faute accidentelle ne peut être malveillante.

[8] Il s'agit le plus souvent dans ce cas d'une modification intentionnelle.

[9] En réalité les vers informatiques actuels ne sont que des virus un peu particuliers, capables d'exploiter les fonctionnalités réseau que les autres catégories de virus ignorent.

L'accent est mis sur le *maintien* du contrôle du système infecté. Ainsi, un rootkit est caractérisé notamment par son invisibilité, sa robustesse et son pouvoir de nuisance (*cf.* chapitre 2).

Altération d'informations qui impactent la sécurité du système Ces altérations peuvent s'appliquer à la fois à la structure du système et à son état. Nous donnons ci-dessous une liste non exhaustive de telles actions :

- suppression de mécanismes de protection ;
- inhibition des mécanismes de l'audit ;
- suppression des journaux d'activités du système ;
- etc.

1.2.3.4 Étape 4 – Exploitation malveillante du système informatique

Ce nœud du graphe représente l'état final dans lequel se retrouve l'attaquant lorsque toutes les conditions pour accomplir son objectif (espionnage, dénis de service, etc.) depuis ou sur le système informatique attaqué sont réunies. La satisfaction de ces conditions est atteinte dès lors que le système attaqué dispose d'une structure adéquate pour l'accomplissement de l'objectif et l'attaquant des privilèges suffisants.

Ce nœud peut également être un état initial. Les attaques considérées sont alors généralement de type *déni de service* pour lesquelles aucune intrusion sur le système n'est nécessaire, si le système fournit des services réseau directement au(x) système(s) source(s) de l'attaque, et si, de plus, le système est vulnérable à ce type d'attaque.

Nous remarquons l'importance de la première étape d'intrusion (pour les attaques qui en ont besoin), car la façon dont elle est accomplie (qui dépend également de l'état de vulnérabilité du système attaqué), contraint fortement le cheminement de l'attaque pour arriver à l'étape finale.

Nous donnons dans la section suivante des exemples d'attaques qui correspondent à différents chemins de notre graphe.

1.2.4 Exemples d'attaques issus du modèle

Avant d'aborder ces différents exemples, nous introduisons le terme d'attaque élémentaire pour qualifier les combinaisons d'actions qui réalisent les étapes fondamentales d'une d'attaque. La réutilisation du terme "attaque" se justifie par le fait que toute étape fondamentale d'une attaque est intrinsèquement un acte malveillant.

Premier exemple Dans notre premier exemple, l'attaquant enchaîne dans l'ordre les étapes 1, 2, 3 et 4. Il a alors besoin :

- d'agir de l'intérieur du système ;
- d'utiliser des services du système pour lesquels *il ne dispose pas des privilèges adéquats*, après avoir pénétré dans le système ;
- et d'utiliser des services dont le système ne dispose pas, et donc d'ajouter de nouveaux services au système (à noter que la désactivation d'un mécanisme de protection peut être vue comme un service).

Nous considérons dans cet exemple le cas d'un attaquant qui travaille pour une entreprise donnée et qui souhaite espionner un système informatique complexe d'une entreprise concurrente. Nous décrivons ci-dessous les grandes lignes de l'attaque. La première étape de l'attaquant est d'usurper l'identité d'un utilisateur légitime du système-cible au travers d'une attaque élémentaire par dictionnaire[10]. Il s'agit de l'*étape d'intrusion dans le système informatique*. Une fois connecté à une machine du système informatique au travers du réseau sous l'identité d'un utilisateur légitime, l'attaquant exploite une faille de sécurité d'un processus privilégié du système (qui s'exécute sous l'identité *root*[11]) afin d'exécuter un *shell root* (c'est-à-dire un interpréteur de commandes ayant les privilèges root), depuis lequel il peut employer toutes les ressources du système. Il s'agit de l'*étape d'acquisition de privilèges*. Une fois ces privilèges acquis, l'attaquant installe un *rootkit* dans le noyau du système d'exploitation de la machine attaquée[12], afin de pouvoir espionner le système informatique dont la machine fait partie. Il s'agit de l'*étape d'altération du système logiciel*. Deux jours plus tard l'attaquant communique avec le rootkit qu'il a mis en place afin de recueillir des informations ayant transité par la machine compromise du système informatique, et récupère ainsi les données confidentielles relatives à son objectif. Il s'agit de l'*étape d'exploitation malveillante du système informatique*.

Deuxième exemple Dans ce deuxième exemple, l'attaquant enchaîne dans l'ordre les étapes 1, 3 et 4. Il a alors besoin :
- d'agir de l'intérieur du système ;
- d'utiliser des services du système pour lesquels *il dispose des privilèges adéquats*, après avoir pénétré dans le système ;
- et d'utiliser des services dont le système ne dispose pas, et donc d'ajouter de nouveaux services au système.

Nous reprenons l'exemple précédent auquel nous changeons l'étape d'intrusion sur le système informatique. Au lieu d'effectuer une attaque élémentaire par dictionnaire, l'attaquant exploite une faille de sécurité du noyau affectant la couche réseau, et parvient à exécuter un code dans ce contexte privilégié, lui permettant d'installer directement le rootkit "noyau". L'étape d'acquisition de privilège n'est alors pas nécessaire car l'intrusion place l'attaquant directement dans un contexte privilégié.

Troisième exemple Dans ce troisième exemple, l'attaquant enchaîne dans l'ordre les étapes 1, 2 et 4. Il a alors besoin :
- d'agir de l'intérieur du système ;
- et d'utiliser des services du système pour lesquels *il ne dispose pas des privilèges adéquats*, après avoir pénétré dans le système.

Nous reprenons le premier exemple que nous modifions quelque peu. La machine attaquée dispose à présent d'un service de transfert de fichiers vers l'extérieur (via un client *ssh* par exemple) auquel a accès seulement l'utilisateur dont l'identifiant est employé par l'attaquant (et non l'utilisateur *root*). En outre, l'attaquant souhaite récupérer les fichiers

[10] Dans cette attaque élémentaire, il s'agit de tester tous les couples (identifiant, mot de passe) issus de *dictionnaires* particuliers

[11] Il s'agit de l'identifiant de l'administrateur principal dans les systèmes d'exploitation de type Unix

[12] Pour installer un *rootkit*, l'attaquant doit tout d'abord le récupérer. Il s'agit d'une acquisition de ressources en cours d'attaque, *cf.* figure 1.1.

d'un utilisateur particulier différent de celui dont il a usurpé l'identité. Dans cette nouvelle version de l'exemple, l'attaquant, après avoir acquis les privilèges *root* accède aux fichiers de l'utilisateur visé et les transfère sur sa propre machine au travers du réseau via une connexion *ssh*.

Quatrième exemple Dans ce quatrième exemple, l'attaquant enchaîne dans l'ordre les étapes 1 et 4. Il a alors besoin :
- d'agir de l'intérieur du système ;
- et d'utiliser des services du système pour lesquels *il dispose des privilèges adéquats*, après avoir pénétré dans le système.

Nous pourrions prendre le cas où le système informatique attaqué a déjà été compromis d'une façon adéquate lors d'une précédente attaque. Pour plus de clarté nous reprenons l'exemple précédent auquel nous apportons une ultime modification, à savoir, le compte de l'utilisateur sur lequel se connecte l'attaquant est directement le compte de celui dont il souhaite récupérer des fichiers confidentiels. L'attaquant n'a alors plus besoin d'acquérir des privilèges, suite à l'intrusion dans le système, pour parvenir jusqu'à l'étape finale de son attaque.

Cinquième exemple Nous considérons dans cet exemple que l'attaquant n'a pas besoin d'agir de l'intérieur du système. L'attaque est donc composée uniquement de l'étape finale (étape 4). Il s'agit alors d'un cas de déni de service (distribué ou non) sur un système informatique directement vulnérable depuis l'extérieur à ce type d'attaque.

Notons que pour le cas d'un déni de service distribué (DDoS), il faut d'abord réussir de multiples attaques sur les machines qui participent au DDoS. Ces attaques sont d'ailleurs la plupart du temps issues de l'installation d'un système parasite (pour effectuer du DDoS) par un système infectieux de type vers ou virus informatique. Les attaques préliminaires sont alors issues d'un attaquant non humain (le vers ou le virus).

Dans la suite de ce manuscrit, nous nous intéressons uniquement aux attaques logiques qui impactent l'intégrité d'un noyau de système d'exploitation en cours d'exécution, et plus particulièrement aux actions qui composent ces attaques. Nous désignons par la suite l'ensemble de ces attaques, par $K|i$[13].

1.3 Introduction aux systèmes d'exploitation

Un *système d'exploitation* [Tanenbaum, 2003] peut être vu comme un gestionnaire de ressources matérielles (processeurs, mémoire, périphériques d'entrée/sortie, etc.) qui doivent être partagées entre les différents programmes qui les sollicitent.

Le rôle d'un système d'exploitation est aussi de fournir à l'utilisateur l'équivalent d'une machine étendue, plus simple à programmer que la machine réelle (c'est-à-dire les composants matériels qui la constituent). Le système d'exploitation définit pour cela un certain nombre de services, que l'on nomme appels-système.

Le *noyau* d'un système d'exploitation est sa partie la plus essentielle, celle qui nécessite le plus de privilèges sur le matériel sur lequel elle s'exécute, afin d'accéder sans restriction aux

[13] La lettre "K" désignant le noyau (*kernel*) et la lettre "i" la propriété d'intégrité.

fonctions de ce matériel[14]. Le mode d'opération dans lequel s'exécute le *noyau* est souvent appelé *mode "noyau"*.

Dans la suite de ce chapitre nous adjoignons à certains termes, le nom "noyau". Nous faisons alors référence au fait que l'objet ou le concept désigné par le terme en question est lié au contexte matériel dans lequel s'exécute le noyau. Dans le cas des architectures x86, que nous abordons dans la section suivante (section 1.4), il s'agit du *ring 0*. Quelques exemples significatifs sont données ci-dessous :

- mode "noyau" : il s'agit du mode dans lequel s'exécute le noyau ;
- espace "noyau" : il désigne l'espace d'adressage de la mémoire, accessible depuis le mode d'exécution du noyau ;
- pointeur "noyau" : il s'agit d'un pointeur dont la signification n'a de sens que lorsque le processeur s'exécute dans le mode d'exécution du noyau ;
- mémoire "noyau" : à ne pas confondre avec la *mémoire du noyau*. Il s'agit ici de la mémoire accessible depuis le mode d'exécution du noyau ;
- processus/*thread* "noyau" : processus/*thread* qui s'exécute dans le mode d'exécution du noyau.

Nous employons également ces raccourcis de langage avec le nom "utilisateur" (espace "utilisateur", mode "utilisateur", etc.). Dans ce cas, nous faisons référence au fait que l'objet ou le concept désigné par le terme en question est lié au contexte matériel dans lequel s'exécute les applications ou programmes employés par l'utilisateur du système.

Suivant la structure du système d'exploitation, son noyau revêt un rôle différent. Parmi les systèmes d'exploitation les plus répandus, nous trouvons les systèmes monolithiques et les systèmes client-serveur. Ces derniers tentent de réduire au minimum la partie fonctionnant en mode "noyau" laissant un micronoyau minimal. L'approche la plus répandue consiste à implémenter la plus grande partie du système d'exploitation sous forme de serveurs (serveurs de fichiers, serveurs de processus, etc.) qui sont des processus "utilisateur" ce qui permet un cloisonnement des différents services du système d'exploitation. La demande d'un service (par exemple, la lecture d'un bloc de fichier) passe par l'envoi par le processus demandeur (appelé processus client) d'une requête à un processus serveur, qui fournit le service et renvoie le résultat au client. Un des rôles majeurs du noyau d'un système de ce type est de gérer les communications entre clients et serveurs.

Au contraire, dans les systèmes monolithiques, la majorité des services jugés critiques réside dans le "noyau". Parmi ces services, citons la gestion du matériel (interruptions matérielles, entrées/sorties, etc.), la gestion de la mémoire, l'ordonnancement des tâches et les appels-système fournis à l'utilisateur.

Nous focalisons notre attention dans le reste du manuscrit sur les systèmes d'exploitation de type Linux, lesquels sont actuellement fortement à tendance monolithique, même si toutes les fonctions du système d'exploitation ne sont pas exclusivement implémentées dans le noyau (comme par exemple le sous-système `udev` qui crée les fichiers nécessaires à l'espace "utilisateur" pour que les applications accèdent aux différents périphériques du système, les scripts dans l'`initramfs` qui continuent l'initialisation du système qui a été débutée par le noyau,

[14] Précisons que dans le cas des systèmes virtualisés, le noyau a un accès total au matériel *virtuel* et non au matériel physique.

le programme */bin/init* sur le disque dur qui termine cette initialisation, etc.). Notons également que le noyau Linux est modulaire, car il est possible de charger dynamiquement (c'est-à-dire au cours de son exécution) des modules ayant des fonctionnalités supplémentaires, étendant ou modifiant ainsi sa panoplie de services. Enfin, ce noyau essaie de fournir des services faciles à utiliser. Ainsi, la logique de développement est de repousser dans l'espace "utilisateur" les parties du système d'exploitation qui nécessitent des prises de décision complexes, ou sinon qui dépendent de choix dont seul l'utilisateur est apte à juger correctement.

1.4 Introduction à l'architecture x86

Ces considérations techniques concernent les systèmes informatiques d'architecture IA-32 (*Intel Architecture for 32 bit system*) ou Intel 64 [Intel Corporation, 2008d,e], lesquels sont basés sur des processeurs x86 (32 bits) ou x86-64 (64 bits)[15]. 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. Par souci de clarté, nous illustrons notre propos avec des exemples, lesquels s'appuient toujours sur la mise en œuvre de cette architecture dans le noyau Linux.

Un système IA-32 est généralement basé sur deux composants principaux, un *chipset* et un processeur (ou CPU pour *Computer Processing Unit*). Les principaux composants logiciels (BIOS, système d'exploitation, applications) s'exécutent sur le processeur. Le chipset est quant à lui responsable de la gestion des périphériques. Il est généralement composé d'une partie nord (*Northbridge*) connectée à la mémoire principale (RAM) et à la carte graphique, et d'une partie sud (*Southbridge*) connectée aux autres périphériques de la machine via différents bus de communication (*cf.* figure 1.3).

1.4.1 Modes d'opération du processeur

L'architecture IA-32 supporte trois modes d'opération et un mode quasi assimilable à un mode d'opération [Intel Corporation, 2008d] :

- Mode protégé : c'est le mode nominal du processeur. Il fournit un ensemble riche de fonctionnalités comme notamment : les *anneaux* qui répartissent les fonctions du processeur sur différents niveaux de privilège (*cf.* section 1.4.2), les unités de segmentation et de pagination qui assure la protection de la mémoire (*cf.* section 1.4.5), etc.
- Mode réel : ce mode d'opération fournit l'environnement des processeurs Intel 8086, avec quelques extensions (telle que la possibilité de permuter en *mode protégé* ou en *mode de gestion système*). Les processeurs x86 s'exécutent initialement dans ce mode pour des raisons de compatibilité. Il est à la charge du système d'exploitation de passer ensuite dans le mode protégé ou IA-32e.
- Mode de gestion système (*System Management Mode* — SMM) : ce mode est une fonctionnalité architecturale standard de tous les processeurs IA-32 (depuis le processeur In-

[15] Notons que les familles x86 et x86-64 regroupent la plupart des processeurs PC classiques (Pentium®, Xeon®, Core Duo^{TM}, Athlon^{TM}, Turion^{TM}).

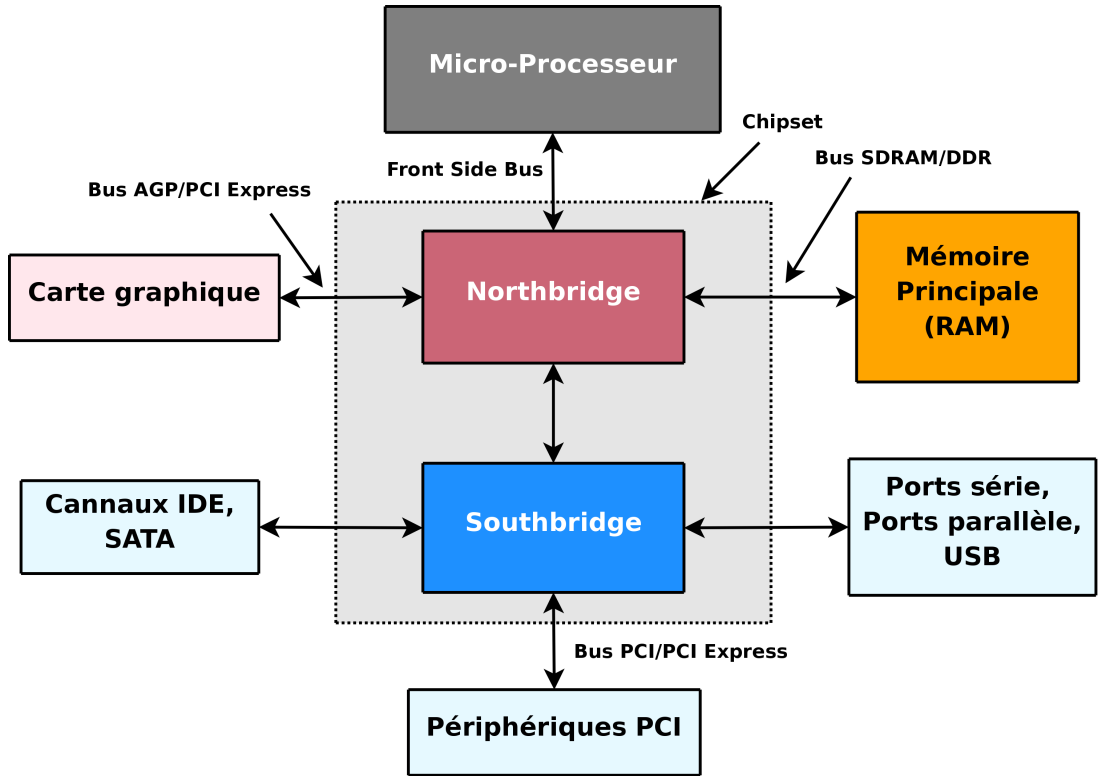

FIG. 1.3 – Architecture simplifiée des ordinateurs x86

tel 386 SL). Le SMM fournit un mode d'opération ou d'exécution avec un mécanisme transparent pour l'implémentation de la gestion d'énergie et de fonctionnalités OEM (*Original Equipment Manufacturer*, comme la gestion des erreurs physiques de la mémoire, le contrôle des différents ventilateurs, etc.). Le processeur entre en mode SMM dès l'activation de la broche d'interruption système externe (SMI#), ou la réception d'une SMI (*System Management Interrupt*) depuis l'APIC (*Advance Programmable Interrupt Controler*). Lorsque le processeur passe en mode SMM, il entre dans un environnement d'opération distinct (contenu dans la SMRAM — *System Management RAM*) dans lequel est aussi sauvegardé le contexte de la tâche qui s'exécutait jusqu'alors. Du code spécifique au SMM peut alors être exécuté de façon transparente, qui plus est, sans restriction : ce code a accès à l'ensemble de la mémoire physique de la machine (dans une limite de 4 Go) et à l'ensemble des registres de configuration ou de l'espace de mémoire propre des périphériques.

– Mode 8086 virtuel (*virtual-8086 mode*) : en *mode protégé*, le processeur peut passer dans ce mode d'opération qui rend possible l'exécution d'anciens logiciels (prévus pour des CPU 8086 qui ne disposent d'aucune notion de protection) dans un environnement multi-tâches *protégé* (où chaque tâche dispose d'un espace d'adressage isolé).

L'architecture Intel 64 supporte tous les modes de l'architecture IA-32 et un mode supplémentaire, le mode IA-32e :

– Mode IA-32e : dans ce mode, le processeur supporte deux sous modes : le *mode de*

compatibilité et le *mode 64 bits*. Ce dernier mode permet l'adressage linéaire sur 64 bits et supporte un espace d'adressage physique supérieur à 64 Go. Il étend également le nombre de registres généraux du processeur (de 8 à 16), et augmente leur taille à 64 bits. Le *mode de compatibilité* permet quant à lui d'exécuter sans changement la plupart des applications en mode protégé.

1.4.2 Niveaux de privilèges du processeur

Le mécanisme de sécurité principal fourni par les processeurs x86 et x86-64 est le mécanisme d'anneaux [Intel Corporation, 2008a] (encore appelé *rings*, ou CPL pour *Current Privilege Level*) qui définit quatre niveaux de privilège différents. Ce mécanisme est disponible dans les modes nominaux de fonctionnement des processeurs (*mode protégé* pour les processeurs 32 bits et mode *IA-32e* pour les processeurs 64 bits). Il consiste à associer un niveau de privilège à la tâche qui s'exécute à un moment donné sur le processeur. Si la tâche s'exécute en *ring* 0 (c'est généralement le cas du noyau d'un système d'exploitation), elle possède des privilèges maximaux qui lui permettent d'utiliser l'ensemble des instructions et d'accéder aux registres de configuration du processeur. En revanche une tâche qui s'exécute en *ring* 3 (une application par exemple) ne possède que des privilèges restreints, et les instructions jugées critiques sur le plan de la sécurité lui sont interdites. La plupart des registres de configuration du processeur ne sont d'ailleurs accessibles qu'au code s'exécutant en *ring* 0.

1.4.3 Environnement basique d'exécution

Chaque programme ou tâche s'exécutant sur un processeur IA-32 ou Intel 64 dispose d'un ensemble de ressources pour exécuter des instructions et pour stocker du code, des données et des informations d'état. Ces ressources forment l'environnement basique d'exécution pour de tels processeurs.

Cet environnement basique d'exécution est employé conjointement par les applications et le système d'exploitation s'exécutant sur le processeur. Nous donnons ci-dessous, les éléments principaux de cet environnement :

- L'espace d'adressage : chaque tâche ou programme s'exécutant sur un processeur IA-32 peut employer un espace d'adressage linéaire allant jusqu'à 2^{32} octets et un espace d'adressage physique allant jusqu'à 64 Go. Pour un processeur Intel 64, l'espace d'adressage linéaire s'étend jusqu'à 2^{64} octets (toutefois restreint par l'utilisation d'un adressage canonique), et l'espace adressage physique jusqu'à 2^{48} octets.
- Les registres basiques d'exécution de programme : les huit registres généraux[16], les six registres de segments (dont les valeurs sont appelées, sélecteurs de segments), le registre d'état (`EFLAGS`), et le registre de compteur d'instructions constituent un environnement basique d'exécution pour les processeurs IA-32 dans lequel un ensemble d'instructions génériques peut être exécuté. Ces instructions accomplissent de l'arithmétique basique sur les entiers, contrôlent le flux d'exécution des tâches, opèrent sur des chaînes de bits ou d'octets, et adressent la mémoire. Pour les processeurs Intel 64, le nombre de registres généraux passe à seize, et tous les registres sont étendus à 64 bits.

[16] Ce sont des registres n'ayant pas un rôle particulier, si ce n'est celui de stocker temporairement des données.

– La pile : pour supporter l'appel à des procédures ou sous-routines et le passage de paramètres entre procédures ou sous-routines, une pile et des ressources de gestion de pile (les deux registres de base et de haut de pile) sont incluses dans l'environnement d'exécution. La pile est localisée dans la mémoire.

En plus de cet environnement basique d'exécution, les architectures IA-32 et Intel 64 fournissent des ressources supplémentaires, qui apportent un support étendu pour les systèmes d'exploitation.

– Les ports d'entrée/sortie : ces architectures supportent le transfert de données au travers de canaux de communication d'entrée/sortie.
– Les registres de contrôle : les registres de contrôle (cinq registres sur IA-32 et six sur Intel 64) déterminent le mode d'opération du processeur et les caractéristiques d'exécution de la tâche courante.
– Les registres de gestion mémoire : `GDTR`, `IDTR`, `TR`, et `LDTR` spécifient la localisation de structures de données employées pour la gestion de la mémoire en *mode protégé*.
– Les `MTRR` (*Memory Type Range Registers*) : ces registres sont utilisés pour assigner à différentes régions de mémoire des types de fonctionnement particulier[17]. Onze de ces registres s'appliquent à des plages d'adressage fixes en mémoire (64 régions sur le premier méga-octet de mémoire) et huit autres à des plages d'adressage variables qu'il faut renseigner.
– Les `MSR` (*Model-Specific Registers*) : les processeurs IA-32 et Intel 64 fournissent une variété de registres spécifiques au modèle du processeur. Ils sont employés pour contrôler différentes fonctionnalités architecturales ou spécifiques au processeur.

1.4.4 Les interruptions matérielles

Le passage de l'exécution des programmes de l'espace "utilisateur" à celle du noyau (c'est-à-dire le basculement depuis l'anneau 3 vers l'anneau 0 et inversement) peut provenir de différents événements. Les interruptions sont parmi les plus fréquentes[18]. Elles sont constituées d'exceptions (interruptions levées par le processeur quand se produit une 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[19]).

Sur l'architecture IA-32, 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

[17] Ces différents types sont : *write-through* (le cache de la CPU et la mémoire sont toujours synchronisés lors d'une écriture), *write-back* (la mémoire n'est mise à jour que lors d'une invalidation de lignes du cache), *write-combining* (plusieurs commandes d'écriture peuvent être émises d'une traite) et *uncacheable* (les données en mémoire ne se trouvent jamais dans le cache de la CPU). Pour plus de détails, se reporter à [Drepper, 2007].

[18] Notons également comme évènement, l'exécution des instructions `sysenter/sysexit` qui sont employées bien souvent pour l'implémentation des appels-système.

[19] Bien que cette méthode soit délaissée, pour des raisons de performance, au profit des instructions `sysenter/sysexit` ou `syscall/sysret`.

cette table avec notamment les adresses des routines et charge ensuite l'adresse de la table dans le processeur via l'intruction `lidt`.

Une interruption matérielle ou une exception du processeur stoppe l'exécution en cours 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 traite l'interruption ou l'exception, puis rend la main à l'exécution interrompue. 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.

1.4.5 Accès à la mémoire principale

Pour adresser les données en mémoire, l'architecture x86 propose deux mécanismes évolués, la segmentation et la pagination (tout deux constituant la MMU — *Memory Management Unit*). Alors que la segmentation est obligatoire, la pagination reste optionnelle (*cf.* figure 1.4). Contrairement à l'unité de segmentation, celle de pagination est présente sur la plupart des types d'architecture. Comme Linux est un noyau multi-plateformes, l'unité de segmentation est seulement utilisée dans son mode basique (le mode *flat*[20]). Ceci permet de se passer facilement de cette unité pour n'utiliser que le seul mécanisme de pagination (*cf.* figure 1.5). Néanmoins, nous proposons ici une rapide explication de la façon dont l'unité de segmentation est utilisée.

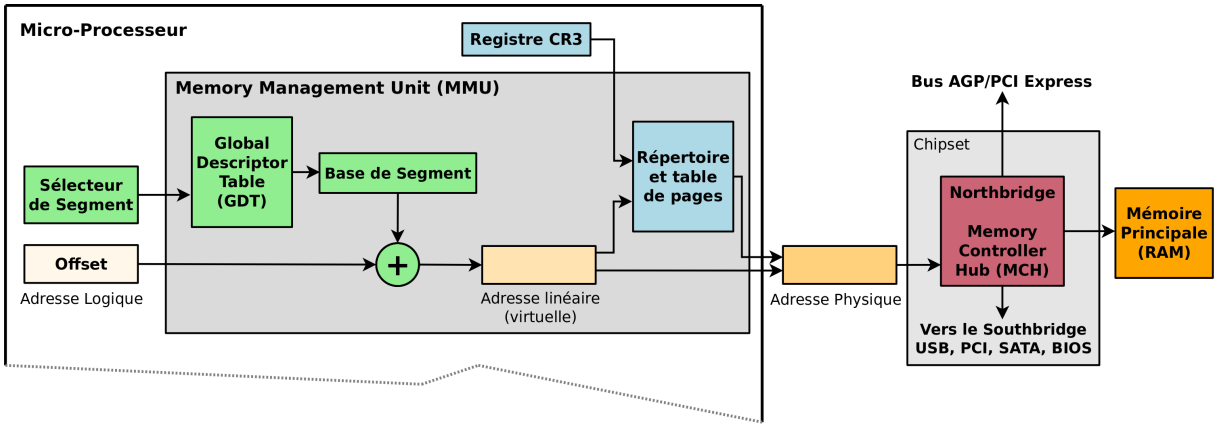

FIG. 1.4 – MMU : unités de segmentation et de pagination

1.4.5.1 Segmentation

Le noyau définit 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. Pour fonctionner, le processeur a besoin d'au moins un segment de code dans lequel il va trouver les instructions à exécuter (`CS`

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

– *Code Segment*), d'un segment de données (`DS` – *Data Segment*), et d'un segment de pile (`SS` – *Stack Segment*). Pour accéder à ces segments, le processeur se fonde sur des registres de segment qui sont chargés avec des valeurs appelées des sélecteurs et qui correspondent à des index dans la `GDT` (pour être exact, seuls les 13 bits de poids fort des sélecteurs correspondent à l'index, les trois bits de poids faible de ces sélecteurs apportent un complément d'information sur lequel nous ne nous attardons pas). Ainsi le processeur dispose des registres `cs` (pour l'accès au segment `CS`), `ds` (pour l'accès au segment `DS`), `ss` (pour l'accès au segment de pile) et de registres de segments à usage divers `es`, `fs` et `gs`. Lorsque un registre de segment est chargé avec un sélecteur, une partie cachée (c'est-à-dire qui n'est pas modifiable directement par l'utilisateur) de ce registre est automatiquement chargée par le processeur avec les informations contenues dans le descripteur de segment correspondant de la GDT. Il s'agit d'un mécanisme permettant d'optimiser les accès à la mémoire, car seule la partie cachée des registres de segments est vérifiée lors d'un accès.

Revenons aux descripteurs de segments que l'on place dans la `GDT`. Un descripteur consiste en une adresse de base, une taille limite, et des informations, notamment sur les droits d'accès au segment décrit. Parmi ces informations, le DPL (*Descriptor Privilege Level*) reflète le niveau de privilège du segment. Pour pouvoir y accéder, le processeur doit se trouver dans un niveau de privilège (*ring*) au moins égal à celui-ci.

Afin de reposer le moins possible sur l'unité de segmentation, Linux emploie le mode *flat* de cette unité qui consiste à définir des segments dont la base est égale à l'adresse linéaire 0 et la taille limite égale à celle de l'espace linéaire (sous x86 en 32 bits, 2^{32}). Lorsque le processeur récupère des instructions en mémoire pour les exécuter (segment CS), il dispose alors d'un accès sur la totalité de l'espace d'adressage. Il en est de même pour les données. La `GDT` mise en place par Linux dispose cependant de plusieurs entrées, notamment parce qu'à chaque segment est associé un DPL. Linux a ainsi besoin d'établir dans la `GDT` des descripteurs de segments différents pour le mode "noyau" (*ring* 0) et le mode "utilisateur" (*ring* 3). De plus, dans le mode IA-32e (mode 64 bits), il est possible d'établir des segments de code (CS) dans lesquels les instructions sont exécutées en mode 32 bits. C'est pourquoi un noyau Linux x86 64 bits établit également des descripteurs en 32 bits, afin de pouvoir tout de même exécuter des applications 32 bits. Chaque entrée dans la `GDT` prend 8 octets (la taille d'un descripteur), excepté pour deux types de segment, le `TSS` (*Task Switch Segment*) et la `LDT` (*Local Descriptor Table*) que nous ne détaillons pas.

1.4.5.2 Pagination

Quand le processeur opère en mode protégé, l'architecture IA-32 prévoit que l'espace d'adressage linéaire soit projeté directement dans un vaste espace de mémoire physique (comme 4 Go de RAM) ou indirectement (via l'utilisation de la pagination) dans un plus petit espace de mémoire physique et de mémoire de masse (comme un disque dur). Quand la pagination [Intel Corporation, 2008d] est employée, le processeur divise l'espace d'adressage linéaire en des pages de taille fixe (4 Ko, 2 Mo, ou 4 Mo) qui peuvent être projetées dans la mémoire physique et/ou la mémoire de masse. Quand une tâche référence une adresse logique en mémoire, le processeur la traduit en une adresse linéaire et utilise ensuite son mécanisme de pagination pour traduire l'adresse linéaire en l'adresse physique correspondante.

Si la page contenant l'adresse linéaire n'est pas actuellement en mémoire physique, le pro-

cesseur génère une exception, appelée faute de page. Le gestionnaire de cette exception s'occupe généralement de charger la page depuis le disque en mémoire physique (il se peut que ce gestionnaire ait à libérer de la mémoire en effectuant l'opération inverse, c'est-à-dire en déplaçant une autre page de la mémoire vers le disque). Quand la page est chargée dans la mémoire physique, le retour du gestionnaire d'exception provoque une nouvelle exécution de l'instruction fautive, qui peut maintenant s'exécuter normalement. Les informations que le processeur emploie pour traduire une adresse linéaire en une adresse physique ainsi que pour générer des fautes de pages (si besoin), sont contenues dans des répertoires de page (appelés PGD sous Linux, pour *Page Global Directory*) et des tables de pages stockées en mémoire. La figure 1.5 illustre la traduction d'une adresse linéaire 32 bits en une adresse physique sur une architecture IA-32 avec des pages de 4 Ko. Le registre de contrôle `CR3` référence le répertoire de pages à partir duquel est effectué la traduction. Notons enfin, que les informations qui renseignent sur l'état des pages (comme la présence ou non d'une page en mémoire physique), appelées *attributs de pages*, sont stockées dans les bits de poids faibles des entrées des différentes tables, ces bits ne servant pas à la traduction d'adresses. Quelques exemples d'attributs de pages sont donnés ci-dessous :

- bit `U/S` (*User/Supervisor*) : le niveau de privilège requis pour accéder à la page, indépendant du RPL de la segmentation ; il n'y a là que deux niveaux : *user* (bit à 1) ou *supervisor* (bit à 0), qui correspondent respectivement au *ring* 3 ou *ring* 0.
- bit `R/W` (*Read/Write*) : si ce bit est à 0, la page est accessible en lecture seule, sinon en lecture/écriture.
- bit `NX` (*Non eXecutable*) : la page est exécutable seulement si ce bit est à 0 (cet attribut n'est disponible qu'en mode IA32e ou bien en mode protégé 32 bits avec l'extension PAE activée).
- bit `D` (*Dirty*) : ce bit est mis à 1 pour signifier que la page a été modifiée depuis qu'elle est en mémoire.
- bit `P` (*Present*) : ce bit est mis à 1 pour signifier que la page est présente en mémoire, sinon une faute de page est émise.

La pagination sous Linux est utilisée comme base de sa gestion mémoire car la plupart des architectures qu'il supporte fournissent un mécanisme de pagination. Cependant la mise en œuvre de cette pagination varie suivant l'architecture. Linux a mis en place un *framework* générique pour gérer ce mécanisme. Ainsi, Linux part du principe que quatre tables d'indirection (ce qui est le cas pour les architectures Intel 64) sont employées par la pagination. Pour les architectures n'utilisant pas autant de niveaux (comme l'IA-32 qui n'utilise que deux niveaux), les primitives s'occupant de ces niveaux se développent alors en des procédures qui relaient les traitements au niveau d'indirection suivant. Ces quatre niveaux d'indirection sont appelés : *Page Global Directory* (PGD), *Page Upper Directory* (PUD), *Page Middle Directory* (PMD) et enfin, *Page Table* (PT).

1.4.6 Accès aux périphériques

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

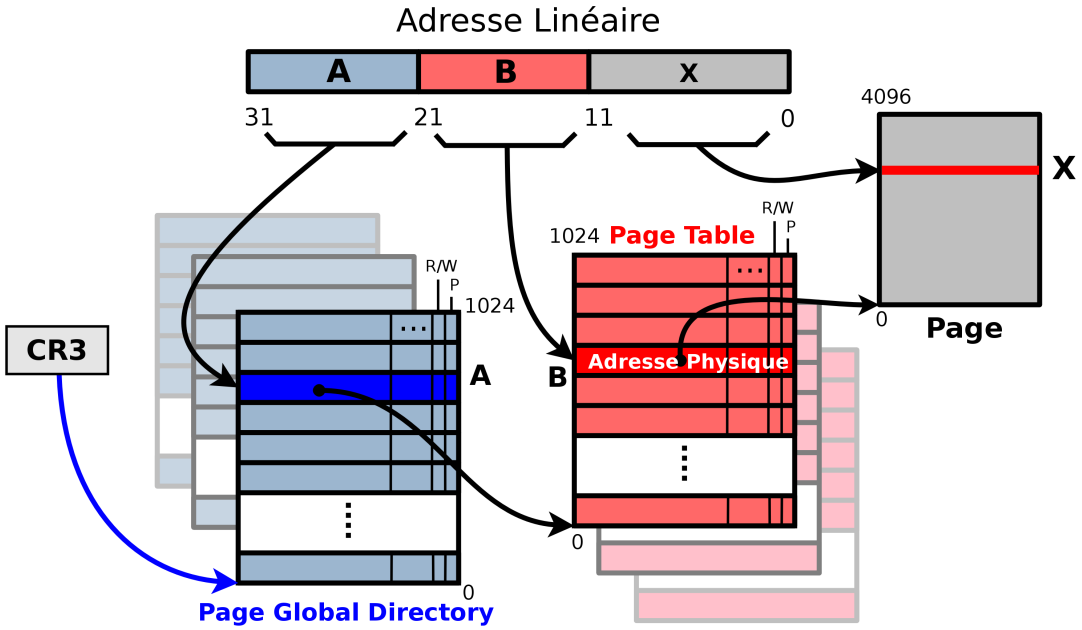

FIG. 1.5 – Mécanisme de pagination

– les registres de configuration accessibles en MMIO (*Memory-mapped I/O*) sont des registres projetés par le *chipset*[21] en mémoire à une adresse donnée. Ils peuvent donc être lus et écrits via l'instruction assembleur `mov` [Intel Corporation, 2008c] comme la mémoire physique ;
– les registres de configuration PIO (*Programmed I/O*) sont eux projetés dans un espace d'adressage 16 bits indépendant. Ils peuvent être lus et modifiés à l'aide des instructions assembleur `in` [Intel Corporation, 2008b] et `out` [Intel Corporation, 2008c] ;
– les registres de configuration PCI [PCI SIG, 1998] sont situés dans un troisième espace d'adressage. On y accède par le mécanisme de configuration PCI, en spécifiant dans le registre PIO d'adresse `0xcf8` l'adresse du registre PCI auquel on souhaite accéder. Le *chipset* met alors automatiquement à jour le registre PIO d'adresse `0xcfc` avec la valeur du registre recherché, qui peut alors être lue ou modifiée à l'aide d'un accès PIO.

1.4.7 Transfert direct de données entre périphériques et mémoire principale

L'accès direct à la mémoire ou DMA (*Direct Memory Access*) est un mécanisme informatique où les transferts de données depuis la mémoire principale de la machine vers un périphérique, ou inversement, sont effectués par un contrôleur adapté, sans intervention du processeur si ce n'est pour initier et conclure le transfert. La conclusion du transfert ou la disponibilité du périphérique peuvent être signalées par interruption. On l'oppose ainsi à des techniques de

[21] Plus exactement, le *chipset* fait en sorte que l'espace d'adressage physique soit mis en correspondance avec ces registres.

polling où le processeur doit attendre chaque donnée.

On parle de DMA maître du bus (*DMA bus-master*) quand le périphérique qui fait du DMA peut adresser lui-même les données sur le bus système, et de *scatter-gather* quand un même transfert peut porter sur une succession de plusieurs zones mémoires.

Pendant le fonctionnement du DMA, celui-ci entre parfois en conflit avec le processeur, les circuits de la mémoire ne pouvant effectuer qu'un accès par cycle. Le DMA ne pouvant pas forcément attendre aussi longtemps que le processeur (qui ne risque pas, lui, de perdre de l'information au vol), il a la priorité d'accès, technique qui se nomme le vol de cycle.

1.5 Les attaques sur les noyaux de systèmes d'exploitation

Dans cette section, nous nous intéressons aux attaques de $K|i$, et plus précisément aux actions qui les composent. Toutes ces actions ne sont pas nécessairement malveillantes, mais il est nécessaire qu'au moins une action de chaque attaque élémentaire le soit car comme nous l'avons dit une attaque élémentaire est un acte *malveillant*.

Pour chacune des étapes fondamentales du graphe d'attaque présentées dans la section précédente, on peut trouver des attaques dans $K|i$ qui entraînent une perte d'intégrité du noyau. Dans la suite nous nous proposons de caractériser les actions responsables de cette perte d'intégrité.

La perte d'intégrité d'un noyau en cours d'exécution provient d'une perte d'intégrité soit (1) du composant matériel qui le supporte (c'est-à-dire de la structure ou de l'état du composant matériel qui contient le noyau), qui est la mémoire principale, soit (2) des composants matériels dont il dépend pour son exécution (le processeur, le *chipset*, et le BIOS), soit enfin (3) des composants matériels avec qui il communique (qui sont les périphériques).

Nous incluons la perte d'intégrité des périphériques dans les causes possibles à la perte d'intégrité du noyau car le noyau est censé permettre à l'utilisateur d'accéder aux services offerts par les périphériques. Si la perte de leur intégrité provoque une impossibilité pour le noyau d'assurer à l'utilisateur l'accès à ces services, alors l'intégrité du noyau est également affectée.

Dans notre travail, nous ne considérons que les attaques logiques (à opposer aux attaques physiques) et nous faisons les hypothèses suivantes :

Hypothèse 1. *La structure du matériel*[22] *dont le noyau a besoin pour s'exécuter est considérée inaltérable, sauf le cas échéant, au travers de fonctions prévues à cet effet (comme par exemple l'infrastructure de mise à jour du microcode des processeurs Intel [Intel Corporation, 2008d]).*

Hypothèse 2. *Les composants matériels dont le noyau a besoin pour s'exécuter sont exempts de failles de sécurité (c'est-à-dire exempts de bogues, de* backdoors *ou de fonctions non documentées [Duflot, 2008] dont l'exploitation peut compromettre la sécurité du système).*

De la première hypothèse, nous pouvons considérer que la partie de la structure matérielle qui peut être altérée, au travers de fonctions prévues à cet effet, est incluse dans l'état du matériel. Ainsi, concernant les composants matériels dont le noyau a besoin pour s'exécuter, nous supposons que seul leur *état* peut être la cible d'une perte d'intégrité.

[22] Rappelons qu'un système, dans ce cas un système matériel, est fait d'une structure qui lui permet de générer son comportement et de supporter son état.

Il s'ensuit qu'une perte d'intégrité d'un noyau en cours d'exécution provient de l'altération (c'est-à-dire d'une modification inappropriée) soit (1) des régions de mémoire du noyau, soit (2) de l'état d'au moins un composant matériel dont dépend le noyau pour son exécution (pour le processeur, il s'agit notamment de l'état de ses registres), soit (3) d'au moins un composant matériel, avec qui il communique mais dont il ne dépend pas directement pour son exécution (c'est-à-dire les périphériques qui sont connectés pour la plupart au reste du système par le *southbridge*).

Ainsi nous pouvons classer à un premier niveau les actions responsables de cette perte d'intégrité, suivant la nature de ce qu'elles modifient :

- les actions qui altèrent les régions de mémoire du noyau forment la *Classe 1* ;
- les actions qui altèrent l'état d'au moins un composant matériel dont dépend le noyau pour son exécution, forment la *Classe 2* ;
- les actions qui altèrent au moins un composant matériel avec qui le noyau communique mais dont il ne dépend pas directement pour son exécution, forment la *Classe 3*.

Nous désignons par la suite l'état des composants matériels dont dépend le noyau pour son exécution comme la *mémoire du support de contrôle*[23], et les composants matériels avec qui le noyau communique mais dont il ne dépend pas directement pour son exécution comme les *périphériques*.

Nous analysons dans un premier temps les vecteurs d'accès à la mémoire du noyau, puis à la mémoire du support de contrôle, et enfin aux périphériques. Dans un second temps, fort de notre analyse, nous proposons un raffinement de la classification des actions malveillantes responsables de la perte d'intégrité d'un noyau.

1.5.1 Vecteurs d'accès à la mémoire du noyau

La première façon d'accéder à la mémoire du noyau se fait au travers de la CPU. Cet accès fait intervenir nécessairement dans un premier temps la MMU (*Memory Management Unit*, *cf.* section 1.4.5) de la CPU et dans un second temps le MCH[24] (*Memory Controller Hub*, *cf.* figure 1.4). Une modification inappropriée de la mémoire du noyau peut donc provenir :

- d'une fonctionnalité du système qui fournit directement le moyen de modifier n'importe quelle région de l'espace de mémoire du noyau. Il s'agit soit :
  - d'une fonctionnalité logicielle, telle que, dans le cas de Linux, le chargeur de modules "noyau" [Truff, 2003], ou encore les périphériques virtuels `/dev/kmem` et `/dev/mem` [Sd et Devik, 2001; c0de, 2003]) ;
  - soit d'une fonctionnalité matérielle, telle que le mode SMM (*System Management Mode*) du CPU [BSDaemon *et al.*, 2008; Duflot *et al.*, 2006a]).
- d'une fonctionnalité du système vulnérable qui fournit le moyen de corrompre l'espace "noyau" au travers de l'exploitation de sa vulnérabilité, par exemple :
  - le débordement d'un tampon [Hoglund et McGraw, 2004] ;
  - l'utilisation d'une chaîne de format malveillante [Hoglund et McGraw, 2004] ;
  - l'utilisation de données incorrectes (déréférencement de pointeur "noyau" ou de valeur

[23] Ces composants matériels formant le support de contrôle du noyau.

[24] Ce dernier composant fait partie du support de contrôle en tant que maillon nécessaire entre la CPU et la mémoire principale.

nulle [Sqrkkyu et Twzi, 2007]) ou périmées (*cf.* par exemple la vulnérabilité qui a affecté les noyaux Linux intégrant la solution de sécurité *PaX* [Lacombe, 2006, section 2]) ;
- etc.

Notons, cependant que l'accès à la mémoire du noyau peut se produire également dans le cache du CPU, si les portions auxquelles on accède y sont disponibles. Ainsi des lectures ou des écritures à ces portions de mémoire peuvent être accomplies sans l'intervention de la MCH. Nous revenons sur ce sujet dans la section 1.5.2, au travers d'un exemple concret propre à la mémoire du support de contrôle.

La seconde façon d'accéder à la mémoire du noyau se fait via les périphériques au travers d'un bus d'entrée/sortie (E/S) supportant le DMA (*Direct Memory Access*, *cf.* section 1.4.7) et engage par conséquent le MCH. Ces vecteurs d'accès de type DMA peuvent être classés en deux catégories suivant que l'accès est initié par le périphérique ou commandé par la CPU[25].

- Lorsque l'accès est à l'initiative du périphérique, celui-ci intègre nécessairement un contrôleur DMA et est connecté sur un bus d'entrée/sortie qui accepte que ses périphériques puissent être maîtres du bus (comme les bus PCI et PCI Express des architectures IA-32 et Intel 64). Ces périphériques capables de *DMA bus-mastering* sont aptes à prendre le contrôle du bus et à effectuer un transfert de données dans la mémoire principale sans implication de la CPU. Par exemple, le bus Firewire peut être utilisé pour lire ou écrire des données dans la mémoire physique sans le contrôle de la CPU et donc du système d'exploitation [Piegdon et Pimenidis, 2007; Dornseif *et al.*, 2005; Boileau, 2006]. Un autre exemple d'accès DMA malveillant à l'initiative d'un périphérique a été publié dans [Devine et Vissian, 2009]. La preuve de concept met en jeu une carte PCI à base de FPGA qui a été reprogrammée afin de compromettre la mémoire du noyau au travers d'un canal DMA.
- Lorsque l'accès à la mémoire par le périphérique est commandé par la CPU, la mise en œuvre d'une modification de la mémoire du noyau provient d'un sous-système logiciel, c'est-à-dire d'un code s'exécutant par l'intermédiaire du système d'exploitation.

Sur les plateformes matérielles récentes, il est possible de contrôler ces accès au travers du *Northbridge* par un composant appelé IOMMU (*Input/Output Memory Management Unit*, *cf.* chapitre 4, section 4.1.1.3) [Rutkowska, 2007] qui agit en tant que "routeur" et filtre des accès en provenance des périphériques et à destination de la mémoire principale.

1.5.2 Vecteurs d'accès à la mémoire du support de contrôle

La mémoire du support de contrôle est composée des registres et de la mémoire interne de la CPU (dont ses différents niveaux de cache[26]), des registres du MCH, et enfin de certaines régions du BIOS. L'exécution du noyau repose en effet sur certaines fonctionnalités du BIOS. Mis à part celles qui sont employées lors de l'initialisation du noyau, deux autres sont fréquemment utilisées au cours de l'exécution du noyau. Il s'agit d'une part des tables ACPI qui

[25] Dans le cas où un périphérique commande à un autre périphérique d'effectuer du DMA, nous considérons ce dernier comme l'initiateur du DMA.

[26] Pour des précisions sur les caches des CPU, se référer au document [Drepper, 2007].

définissent notamment des méthodes de gestion de l'énergie et de la configuration de la plateforme matérielle (ces méthodes sont employées par le système d'exploitation). D'autre part, le BIOS contient également la routine de traitement de l'interruption SMI qui s'occupe de la gestion de certaines spécificités matérielles de la plateforme. Elle fait alors partie du support de contrôle. Notons que le BIOS est copié dans la mémoire principale lorsque la machine est démarrée. Ainsi cette partie de la mémoire du support de contrôle se situe à deux niveaux.

Les registres de la CPU sont accessibles uniquement depuis la CPU, et par conséquent depuis le logiciel qui est exécuté dessus. Notons que pour le logiciel s'exécutant dans le mode nominal des CPU x86 (*cf.* section 1.3) en *ring* 0, tous les registres sont accessibles exceptés des registres spécifiques au mode SMM. Dans les anneaux moins privilégiés comme le *ring* 3, le logiciel est sujet à des restrictions sur les registres accessibles et le jeu des instructions utilisables[27]. Dans le mode SMM, tous les registres sont accessibles. Il reste néanmoins des registres ou régions de mémoire privées de la CPU qui ne sont pas accessibles et dont l'usage est propre à la CPU (comme par exemple, la partie cachée des registres de segments). Certains registres privés de la CPU sont cependant accessibles indirectement depuis la mémoire principale sous certaines conditions. Par exemple le registre `SMBASE` (stockant l'adresse mémoire où se trouve la SMRAM), qui n'est pas accessible depuis le jeu d'instructions de la CPU, est sauvegardé en mémoire lorsque la CPU passe en mode SMM et exécute le gestionnaire de l'interruption SMI. Lorsque le gestionnaire se termine, la CPU charge son registre `SMBASE` avec la valeur correspondante en mémoire. Ainsi, `SMBASE` peut être modifié suite à l'exécution du gestionnaire si une écriture à l'adresse mémoire correspondant à la sauvegarde de `SMBASE` a été effectuée. La modification de `SMBASE` peut donc être réalisée depuis la routine de traitement de la SMI, mais ne peut normalement pas être réalisée depuis un autre mode de la CPU, car le MCH implémente un contrôle d'accès à la région de mémoire de la `SMRAM`. Cependant, il reste possible de modifier `SMBASE` en accédant directement aux caches de la CPU, sans passer par la mémoire et donc en évitant le contrôle d'accès mis en place par le MCH [Duflot et Levillain, 2009]. L'exemple précédent révèle des failles sur le contrôle d'accès à la mémoire effectué par le MCH. À partir du moment où la région de mémoire à laquelle on souhaite accéder peut être mise en cache (c'est-à-dire placée dans une mémoire interne spécifique de la CPU)[28], les accès en lecture ou écriture à cette région sont effectués directement dans le cache de la CPU et ne sollicitent donc pas le MCH pour l'accès à la mémoire principale, évitant par là-même de se confronter au contrôle d'accès de ce composant matériel.

Les registres du MCH sont accessibles uniquement au travers de la CPU et ainsi depuis le logiciel qui s'exécute dessus. En effet, ces registres sont accessibles uniquement au travers du mécanisme de configuration PCI (*cf.* section 1.4.6) car ils sont dans l'espace de configuration du PCI [Shanley et Anderson, 1999]. Par conséquent, les périphériques ne peuvent en aucun cas accéder aux registres du MCH.

Vis-à-vis de la copie du BIOS en mémoire, les accès peuvent provenir à la fois de la CPU ou de périphérique (DMA). Le MCH peut aussi être configuré par le BIOS de façon à interdire ou à limiter certains accès (par exemple, l'accès à la routine de la SMI peut être restreint au mode SMM de la CPU). Vis-à-vis de l'image originale du BIOS située dans une mémoire de type

[27] Les instructions `hlt`, `lgdt` et `invd` ne sont par exemple pas disponibles. Ainsi leur exécution génère l'exception *General Protection* (vecteur n° 13).

[28] La politique de mise en cache peut être positionnée depuis les registres `MTRR`, *cf.* section 1.4.3.

ROM, si celle-ci est *flashable*, alors il faut considérer qu'elle fait aussi partie de la mémoire du support de contrôle ; sinon nous considérons qu'elle appartient à la structure du matériel car aucune action logique ne peut être effectuée dessus. Lorsque les accès logiques à cette ROM sont possibles, ils sont effectués depuis la CPU.

1.5.3 Vecteurs d'accès aux périphériques

La CPU est le composant matériel principal qui accède aux périphériques d'un ordinateur. Il est cependant possible pour un périphérique d'accéder à certaines régions de mémoire d'autres périphériques. Nous commençons par présenter le cas de la CPU, avant de brièvement exposer le cas des accès entre périphériques.

La CPU accède aux périphériques pour d'une part les configurer et d'autre part pour employer leurs fonctions (ces accès sont généralement regroupés sous la désignation d'*accès d'entrée/sortie*). Trois principales méthodes d'accès sont fournies par les architectures IA-32 et Intel 64 (*cf.* section 1.4.6) mais dépendent du périphérique auquel on accède :

- le mécanisme MMIO (*Memory-Mapped I/O*), qui accomplit la projection des registres dans l'espace d'adressage physique ;
- le mécanisme PIO (*Programmed I/O*), qui accomplit la projection des registres dans un espace d'adressage séparé de 16 bits ;
- le mécanisme PCI [PCI SIG, 1998], qui est employé pour accéder aux registres de configuration PCI, situés dans un troisième espace d'adressage et qui emploie le mécanisme PIO.

Les accès d'entrée/sortie de type MMIO et PIO sont limités généralement au *ring* 0 (dans lequel s'exécute le noyau)[29]. Le système d'exploitation peut cependant déléguer certains de ces accès à des applications privilégiées de l'espace "utilisateur" (pour des systèmes d'exploitation de type Unix, il s'agit généralement des applications s'exécutant avec les privilèges *root*) au travers des appels-système `iopl()` (pour l'accès total aux PIO) et `ioperm()` (pour l'accès à des PIO spécifiques). Notons également, que la routine SMI qui s'exécute dans le mode SMM de la CPU peut effectuer également des accès d'entrée/sortie à l'insu du noyau.

Au sujet des accès entre périphériques, la spécification des bus PCI [PCI SIG, 1998] prévoit qu'un périphérique PCI branché sur le même pont qu'un autre périphérique PCI puisse accéder à certaines de ses régions de mémoire (à l'exclusion de la mémoire dédiée à l'espace de configuration PCI). Certaines plateformes matérielles supportent également les transactions PCI entre périphériques connectés sur des bus PCI différents (et donc qui traversent plusieurs ponts PCI). Toutefois, la spécification du bus PCI précise que ce comportement n'est pas obligatoire.

1.5.4 Classes d'actions malveillantes visant le noyau

Nous examinons à présent les différents types d'actions malveillantes qui altèrent le comportement d'un noyau. L'analyse que nous avons effectué a mené à une classification plus détaillée sur ces actions malveillantes.

[29] Nous ne considérons pas ici le mode réel de la CPU car il n'est plus employé de façon nominale par les systèmes d'exploitation d'aujourd'hui.

Nous avons expliqué dans les sections précédentes que les actions malveillantes visant un noyau sont soit externes au noyau (comme les actions qui proviennent de l'espace "utilisateur" ou d'accès DMA), soit internes (les actions effectuées depuis le niveau d'exécution du noyau). Ces dernières actions sont souvent rendues possibles par une action préalable qui est externe au noyau[30].

Enfin, que ces actions malveillantes soient internes ou externes au noyau, les cibles sur lesquelles elles s'appliquent n'en dépendent pas. C'est pourquoi la classification regroupe sans distinction les actions malveillantes internes et externes.

1.5.4.1 Classe 1 – Altération de la mémoire du noyau

L'organisation de cette classe découle directement de la vision d'un système comme la combinaison de sa structure et de son état total (*cf.* section 1.1). Nous employons cependant une terminologie plus spécifique au logiciel en appelant *chemins d'exécution* la structure d'un système logiciel[31], et *variables* son état total.

- **Classe 1.1 – Modification inappropriée des chemins d'exécution du noyau**
  Cette classe est caractérisée par les actions malveillantes qui nécessite d'injecter du code afin d'accomplir leur objectif. Ces injections de code peuvent être classées suivant le type de région mémoire dans laquelle s'effectue l'injection. Notons également que le code injecté doit être atteignable, c'est-à-dire qu'il doit être inscrit dans les chemins d'exécution, sinon l'action d'injection elle-même n'est pas une action dangereuse pour la sécurité[32].

  - *Classe 1.1.1 – Ajout d'une région de code "noyau" malveillant atteignable*
    Cette classe est caractérisée par les actions malveillantes qui injectent une région de code dans l'espace de mémoire du noyau. Certaines de ces actions malveillantes profitent par exemple d'une fonctionnalité du noyau telle qu'un chargeur de module "noyau" [Pragmatic et THC, 1999].

  - *Classe 1.1.2 – 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, soit 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 [Cesare, 1999a]).

[30] Sinon le noyau agit directement de façon malveillante, ce qui suppose une faute de conception délibérée ou un bogue dans le noyau lui même.

[31] La structure d'un système logiciel est définie par son code et l'agencement en mémoire de ses différents segments.

[32] L'injection de code non atteignable peut cependant se présenter au cours d'un acte malveillant, lorsqu'une deuxième action succède à cette injection dans l'optique de rendre le code atteignable. Dans ce cas, seule la deuxième action est dangereuse pour la sécurité. Nous revenons sur ce cas dans notre analyse de la classe 1.2.

– *Classe 1.1.3 – Injection de code malveillant atteignable dans une région de données "noyau"*
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 [Hoglund et McGraw, 2004] appartiennent à cette classe. Celles qui injectent du code dans des emplacements vides (*padding*) de pages de données appartiennent également à cette classe.

– *Classe 1.1.4 – Injection de code malveillant atteignable au sein d'une région n'appartenant pas au noyau (typiquement, une région de l'espace "utilisateur")*
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"[33]). Cela provient, par exemple, de bogues du noyau qui peuvent être exploités afin de copier 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"[34], 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 [Corbet, 2008c,b] (*cf.* [Sqrkkyu et Twzi, 2007] pour une explication détaillée sur la façon d'exploiter un déréférencement de pointeur "noyau" de valeur nulle).

– **Classe 1.2 – Modification inappropriée des variables du noyau**
Cette classe est caractérisée par les actions malveillantes qui n'injectent pas du code dans le noyau, mais provoquent une modification inappropriée du comportement du noyau en modifiant ses variables.

– *Classe 1.2.1 – Altération des variables d'état de l'exécution*
Les actions de cette classe altèrent le comportement du noyau en modifiant des variables sur lesquelles repose son exécution[35].
Des exemples de telles variables sont les données de contrôle du flux d'exécution (particulièrement le compteur d'instruction) qui résident dans la pile, les données employées dans les conditions de branchement du code "noyau", les attributs des tables de pages (comme les drapeaux `Present`, `Read/Write`, `No eXecution`, etc. — *cf.* section 1.4.5).
Les actions malveillantes qui altèrent les données de contrôle du flux d'exécution

[33] L'hyperviseur ou "moniteur de machines virtuelles" est employé dans le cadre de la virtualisation de système, *cf.* chapitre 4 section 4.1.1.
[34] La limite de l'espace "utilisateur" sous Linux est représentée par la constante `TASK_SIZE`.
[35] Notons qu'une action de cette classe peut être précédée d'une injection de code arbitraire atteignable ou non atteignable. Dans le premier cas, il s'agit alors d'une combinaison de deux actions dangereuses pour la sécurité, alors que dans le second cas, seule l'action de la classe 1.2.1 l'est.

dans la pile peuvent être employées afin d'exécuter du code "noyau" existant mais dans un mauvais ordre [Nergal, 2001; Solar Designer, 1997]. 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 inapproprié vis-à-vis de l'exécution courante[36].
D'autres actions malveillantes écrasent des attributs de pages afin de passer outre la protection par le drapeau `No eXecution` sur une page particulière.
En outre, les débordements d'entier et plus particulièrement les débordements de compteurs de références sont des actions malveillantes qui appartiennent à cette classe [Pol, 2004].
De même, toutes les actions malveillantes qui désactivent des protections de sécurité en écrasant seulement des données du noyau (sans exécution de code supplémentaire) font parties de cette classe.

– *Classe 1.2.2 – Altération des variables auxiliaires*
Les actions de cette classe altèrent le comportement du noyau en modifiant certaines de ses variables qui n'affectent pas le flux d'exécution, et que nous appelons les variables auxiliaires.
De telles actions peuvent être employées pour empêcher l'affichage de messages d'erreur, de messages d'alerte, etc. (en mettant à zéro par exemple des chaînes de caractères utilisées par la primitive `printk()` dans une section de code "noyau" particulière).
D'autres actions peuvent seulement modifier des variables auxiliaires qui seront envoyées au travers du réseau vers un autre ordinateur, depuis lequel elles seront employées comme des variables d'état de l'exécution.

1.5.4.2 Classe 2 – Altération de la mémoire du support de contrôle

– **Classe 2.1 – Altération des registres de la CPU ou de sa mémoire interne**
Cette classe est caractérisée par les actions malveillantes qui modifient de façon inappropriée :
  – des registres critiques de la CPU, tels que les registres de segments (`cs`, `ds`, `ss`, etc.), le registre `idtr`, le registre `gdtr`, les `MTRR` (*Memory Type Range Registers*), les `MSR` (*Model-Specific Registers*), etc. ;
  – des parties de la mémoire interne de la CPU, telles que la région de microcode (si elle est accessible) employée pour modifier le comportement de la CPU [Intel Corporation, 2008d] ; ou encore ses différents niveaux de caches.

  Par exemple, afin d'installer un rootkit "noyau" [Lacombe *et al.*, 2008], des attaquants copient l'IDT (*Interrupt Descriptor Table*), puis modifient cette copie pour finalement charger son adresse dans le registre `idtr` du processeur (remplaçant ainsi la précédente table) [Kad, 2002]. Cette dernière action est malveillante et fait partie de cette classe.

[36] Cette approche peut être généralisée afin d'exécuter en séquence plusieurs parties de code "noyau" existant. Une action de ce type peut alors être qualifiée d'exécution ordonnée de façon malveillante.

Un autre exemple d'action malveillante issue de cette classe se trouve dans l'attaque présentée par Loïc Duflot, sur la modification du gestionnaire de la SMI (qui est la routine exécutée en mode SMM par la CPU, en réponse à une *System Management Interrupt* — *cf.* section 1.4.1) [Duflot *et al.*, 2009; Duflot et Levillain, 2009]. Sa preuve de concept implique la modification du registre privé `SMBASE` de la CPU, ainsi que de l'un de ses registres `MTRR`.

– **Classe 2.2 – Altération des registres du MCH**
Cette classe est caractérisée par les actions malveillantes qui modifient des registres du MCH afin d'altérer le comportement du noyau. De telles actions malveillantes sont illustrées encore par des preuves de concepts (d'attaques) de Loïc Duflot dans [Duflot *et al.*, 2009] où la preuve de concept modifie le registre `SMRAMC` du MCH[37], et dans [Duflot *et al.*, 2006b] où elle exige la modification du registre `AGPM` du MCH (afin de permettre l'accès à "l'ouverture graphique").

– **Classe 2.3 – Altération du BIOS**
Cette classe est caractérisée par les actions qui altèrent le BIOS de la machine. Certaines de ces actions altèrent directement la ROM qui la contient [Sacco et Ortega, 2009b,a], et d'autres sa copie en mémoire principale qui est employée au cours de l'exécution du système. Il s'agit alors notamment des tables ACPI ou encore de la routine de traitement de la SMI [Duflot et Levillain, 2009].

1.5.4.3 Classe 3 – Altération des périphériques

Cette classe est caractérisée par les actions malveillantes qui altèrent la valeur de certains registres d'un périphérique[38], ou de sa mémoire interne (s'il en dispose), ou encore qui altèrent la structure d'un périphérique s'il est adaptable (comme les périphériques qui sont construits autour de FPGA). Parmi les mémoires d'un périphérique, des ROM *flashable* d'extension du BIOS, appelées *expansion ROM*, sont employés fréquemment sur les périphériques PCI. Ces ROM contiennent du code que le BIOS exécute afin notamment d'initialiser les périphériques en question.

Comme exemple d'action de cette classe, imaginons le scénario suivant dans lequel un périphérique de type contrôleur réseau serait construit à partir de circuits reprogrammables. Une action malveillante pourrait alors reprogrammer le périphérique de façon à ce que l'envoi de paquets "réseau" soit désormais impossible. Ainsi, il en découle une perte d'intégrité du noyau, car il lui est désormais impossible d'assurer le bon fonctionnement de ce périphérique réseau. Un problème de ce type est d'ailleurs apparu sous Linux de façon accidentelle à cause d'un bogue dans sa version 2.6.27. Ce bogue a conduit dans certains cas à la corruption de la ROM de contrôleurs réseau de la marque Intel de type `e1000e`, rendant alors ces périphériques inopérants [Corbet, 2008a].

[37] De plus amples informations sur ce sujet sont disponibles dans [Embleton *et al.*, 2008], qui identifie une nouvelle classe de rootkits fondés sur le mode SMM.

[38] Rappelons, que nous appelons "*périphériques*" les composants matériels avec lesquels le noyau communique mais dont il ne dépend pas directement pour son exécution.

Encore plus insidieux, la reprogrammation du périphérique pourraient faire en sorte d'injecter des charges malveillantes (par exemple des vers) dans les paquets "réseau" que le noyau souhaite envoyer. À notre connaissance, aucune publication ne fait état de telles actions malveillantes, bien que les résultats sur la corruption de ROM (d'expansion du BIOS) de périphérique [Heasman, 2006a], ou des tables ACPI dans le BIOS [Heasman, 2006b] pourraient être employés à ces fins. Les travaux publiés considèrent uniquement l'implémentation, dans ces mémoires de périphériques, d'un rootkit qui modifierait la mémoire du noyau, c'est-à-dire qui effectuerait uniquement des actions de la classe 1.

1.6 Conclusion

Dans ce chapitre, nous avons introduit une partie des connaissances nécessaires à la compréhension de nos travaux. Nous avons également introduit les concepts de base et la terminologie associée à la sécurité des systèmes. Nous nous sommes alors concentrés sur la caractérisation des attaques logiques sur un système informatique. Suite à cette analyse, nous nous sommes focalisés sur le sujet de notre étude, c'est-à-dire les attaques qui s'appuient sur la corruption d'un noyau de système d'exploitation. Ce travail nous a servi à identifier les différents vecteurs d'accès pour altérer le comportement d'un noyau. Suite à cette activité, nous avons pu caractériser les actions malveillantes nuisant à l'intégrité des noyaux (composantes nécessaires de toute attaque reposant sur la corruption d'un noyau). Enfin nous avons proposé une classification de ces actions [Lacombe *et al.*, 2009b], élément essentiel à la mise en œuvre de mécanismes visant à assurer l'intégrité d'un noyau, sujet des chapitre 3 et 4.

Dans le prochain chapitre, nous exposons le résultat de notre travail sur l'étude des rootkits "noyau", parasites informatiques s'appuyant sur tout ou partie des actions malveillantes de notre classification. Ce travail nous a permis de mieux comprendre les problèmes de sécurité qui touchent les noyaux de système d'exploitation, et nous a permis d'élaborer des solutions visant à en garantir l'intégrité (sujet que nous développons dans le chapitre 4).

Chapitre 2

Étude d'un type de parasite informatique : les *rootkits* "noyau"

Contrairement à une idée répandue, il n'est nullement besoin d'*exploit*[1] pour s'introduire frauduleusement sur un système d'information[2]. En revanche, une fois le système compromis, l'intrus entreprend des actions profitant de vulnérabilités afin d'utiliser le système, et dans la majorité des cas, de pérenniser son accès à l'insu des utilisateurs légitimes : il s'appuie pour cela sur un *rootkit* (ensemble de modifications permettant à un attaquant de maintenir dans le temps un contrôle frauduleux sur un système d'information).

Ces *rootkits* sont susceptibles d'employer n'importe quelle action malveillante parmi celles que nous avons identifiées dans le chapitre 1 en section 1.5.4. Ils font ainsi l'objet d'un intérêt tout particulier dans notre travail. Dans ce chapitre, nous développons les principes fondamentaux caractérisant les *rootkits*. Nous nous plaçons résolument du point de vue de l'attaquant, concepteur et utilisateur du *rootkit*, cherchant ainsi à identifier tant les diverses stratégies envisageables que les contraintes techniques.

Nous présentons tout d'abord en section 2.1 la logique qui a conduit à l'évolution actuelle des *rootkits*. Nous rappelons à cette occasion les mécanismes d'injection et de détournement employés par les *rootkits* "noyau" sous Linux. La section 2.2 s'attache aux principes qui dirigent l'élaboration d'un *rootkit*. Pour cela, nous nous plaçons dans la peau de l'attaquant, et voyons comment en fonction de ses objectifs et de ses contraintes, la nature du *rootkit* est susceptible de changer. Nous proposons une analogie avec la *dissimulation d'information* afin d'évaluer la qualité d'un *rootkit*. Nous présentons alors en section 2.3 les méthodes que nous avons élaborées (et implémentées en partie dans un démonstrateur) pour corrompre un noyau Linux tout en étant le plus discret possible. Finalement, la section 2.4 clôt le chapitre avec une synthèse sur les apports de notre approche.

[1] Un *exploit* est un petit programme tirant parti d'une faille dans un logiciel pour pénétrer le système hôte.

[2] Par exemple, un attaquant peut se connecter à un système en employant le compte d'un utilisateur légitime dont il aurait volé le mot de passe.

2.1 État de l'art sur les *rootkits* "noyau" sous Linux

Nous présentons tout d'abord un bref historique des *rootkits*, en montrant comment ils ont évolué de façon à devenir de plus en plus efficaces.

2.1.1 Évolution des *rootkits*

Le premier réflexe d'un administrateur lorsqu'un événement éveille son attention sur son système est de consulter ses logs, d'appeler la commande `last` pour vérifier qui s'est connecté en dernier, d'appeler `netstat` pour examiner les connexions réseau, et ainsi de suite. Un pirate, connaissant la réaction de l'administrateur, va tenter de dissimuler sa présence sur le système en remplaçant les commandes usuelles par des versions modifiées : c'est la première forme des *rootkits*. Ainsi, lorsque l'administrateur en appelle à ces outils, ceux-ci cachent certaines informations sur l'intrus, affichant les résultats qui apparaîtraient si l'intrus n'était pas là et, rassurant par la même l'administrateur peu averti.

Cependant, cette approche n'est pas fiable pour le pirate :

- sur les premiers Unix, il était assez fréquent de recompiler les sources car la bande passante ne permettait pas de télécharger des giga-octets de binaire, et l'intrus devait donc substituer ses programmes aux originaux à chaque fois que ceux-ci étaient recompilés ;
- il est aisé d'obtenir la même information par le biais de plusieurs commandes (exemple : lister des fichiers avec `ls`, `find`, `grep -r`, ...), et le risque pour l'attaquant est d'en oublier une, risquant ainsi que sa présence soit révélée ;
- bien souvent, et surtout dans les environnements sécurisés, des *empreintes* des binaires sont calculées à l'aide d'une fonction de hachage, afin de détecter la modification de ces programmes, par exemple lors d'une analyse post-intrusion.

Afin de résoudre en partie ces problèmes, et en particulier les deux premiers, les *rootkits* ont évolué, cherchant à corrompre un maximum de programmes avec un minimum d'effort. Avec l'apparition des bibliothèques dynamiques, partagées par une majorité de binaires, les intrus ont vu là un bon moyen de régler leurs soucis : modifier une fonction dans une bibliothèque affecte tous les programmes qui emploient cette bibliothèque. Néanmoins, les problèmes restent les mêmes, dans une moindre mesure.

Cette démarche de factorisation (modifier moins, corrompre plus) s'est donc naturellement poursuivie vers la dernière ressource partagée par tous les éléments : le noyau. En charge tant de la gestion du matériel que de l'ordonnancement des tâches ou de la gestion de la mémoire, le noyau est le point de passage obligatoire pour tous les éléments du système d'exploitation. Nous détaillons respectivement dans les sections 2.1.2 et 2.1.3 les méthodes d'injection et de détournement en espace "noyau".

D'autres types de *rootkit*, allant encore au delà des approches de type *rootkit* "noyau", ont été mis en œuvre. Nous les mentionnons dans la suite de cette section par souci d'exhaustivité.

Les *rootkits* de *boot* (aussi appelés *bootkits*) sont installés dans le secteur de démarrage de la machine. Ainsi, ils prennent la main avant le système d'exploitation. Cette idée est mise en œuvre par exemple dans les *rootkits* BootRoot [Soeder et Permeh, 2005] ou encore Boot Kit [Kumar et Kumar, 2006].

Une autre variété de *rootkits* exploitent les technologies de virtualisation de système. Ces

technologies ont été conçues dans l'optique d'implémenter des logiciels (appelés hyperviseurs) capables de gérer plusieurs machines virtuelles (sur lesquelles s'exécutent des systèmes d'exploitation appelés "systèmes invités",) sur une unique machine physique. Le détournement de ces technologies octroie aux rootkits un contrôle plus efficace des systèmes infectés, et leur fournit les moyens de se dissimuler plus efficacement du système d'exploitation qu'il souhaite contrôler (celui-ci étant un "système invité"). Parmi ces *rootkits*, deux catégories sont à distinguer. Les *rootkits* "hyperviseur" parasitent des systèmes qui exécutent un hyperviseur [King *et al.*, 2006]. Les *rootkits* HVM (*Hardware-assisted Virtual Machine*), quant à eux, se fondent sur les technologies matérielles d'aide à la virtualisation (comme Intel `VT` ou AMD `SVM`) et embarquent en leur sein un hyperviseur léger[3]. Blue Pill [Rutkowska, 2006] est un exemple de *rootkit* HVM qui utilise le support matériel des processeurs AMD pour la virtualisation [AMD, 2005]. Il installe tout d'abord un hyperviseur léger à l'insu du système d'exploitation et place ensuite ce dernier dans une machine virtuelle. Il en obtient alors le contrôle sans pour autant le modifier.

Un nouveau type de *rootkits* est apparu à la suite des *rootkits* "hyperviseur" et HVM. Il s'agit des *rootkits* SMM dont les concepts ont été présentés dans [Embleton *et al.*, 2008]. Ces *rootkits* opèrent depuis le mode SMM des processeurs x86 (*cf.* section 1.4.1) en s'installant à la place de la routine de gestion de l'interruption SMI. Ils s'exécutent alors avec les privilèges les plus élevés sur la machine et cela à l'insu du système d'exploitation. Une preuve de concept a été mise en œuvre dans [Wecherowski, 2009].

D'autres *rootkits* s'implantent, quant à eux, au niveau du BIOS d'une machine [Sacco et Ortega, 2009b,a]. Une approche affectant les tables ACPI (*Advanced Configuration and Power Interface*) du BIOS a été présentée dans [Heasman, 2006b]. Le principe est d'installer des fonctions malveillantes dans ces tables[4]. Le *rootkit* serait alors déployé suite à leur utilisation par le système d'exploitation. Bien que cette technique permette d'améliorer la persistance du *rootkit* sur le système tout en lui octroyant de grands privilèges. Plusieurs technologies permettent la détection de ces *rootkits*. Par exemple, les cartes mère qui embarquent les technologies Intel SecureFlash et Phoenix TrustedCore empêchent l'installation de mises à jour du BIOS qui ne sont pas signées.

Enfin, la dernière catégorie de *rootkits* dont nous ayons connaissance à ce jour s'implantent au niveau de périphériques PCI disposant de ROM *flashable* d'extension du BIOS, appelées *expansion ROM*[5] [Heasman, 2006a]. Dans ce cas, les technologies aptes à détecter ce genre de *rootkits*, s'appuieraient sur des composants matériels de type TPM (*Trusted Platform Module*) [ISO/IEC, 2009; Trusted Computing Group, 2004].

Le tableau suivant présente pour chacune de ces catégories de *rootkits*, les classes d'actions malveillantes qui les caractérisent.

[3] Le support que le matériel apporte pour la virtualisation facilite la réalisation d'un hyperviseur. Cet aspect est très important dans notre contexte car il est préférable pour un *rootkit* d'avoir une faible empreinte mémoire, son transfert de la machine de l'attaquant vers la machine à compromettre pouvant alors s'effectuer de façon plus discrète.

[4] Ces fonctions sont écrites en AML (*ACPI Machine Language*). Elles ont accès à la mémoire du système et aux différents périphériques.

[5] Ces ROM contiennent du code que le BIOS exécute afin notamment d'initialiser les périphériques en question.

Type de rootkit	**Classe 1** (Image noyau)	**Classe 2** (Supp. contrôle)	**Classe 3** (Périphériques)
rootkit "noyau"	x		
rootkit de boot	x		x
rootkit "hyperviseur"		x	
rootkit HVM		x	
rootkit ACPI		x	
rootkit SMM		x	
rootkit PCI			x

Les *rootkits* "noyau" se caractérisent par des modifications de l'image du noyau. Ils entreprennent alors des actions de la classe 1[6]. Les *rootkits* de boot, avant de modifier le noyau, altère le secteur de boot du disque. Ils effectuent alors des actions de la classe 3. Les *rootkits* "hyperviseur" et HVM, quant à eux, modifient le support de contrôle (classe 2). Pour les premiers, il s'agit d'un support de contrôle virtuel (celui qui est créé par l'hyperviseur parasité), alors que pour les seconds c'est le support de contrôle de la machine physique qui est altéré. Les *rootkits* ACPI modifient les tables ACPI du BIOS, et les *rootkits* SMM modifie la SMRAM. Ils altèrent donc des éléments du support de contrôle (classe 2). Enfin les *rootkits* PCI s'installent au sein des périphériques et opèrent notamment par des actions de la classe 3.

Dans la suite du chapitre nous focalisons notre attention uniquement sur les *rootkits* "noyau".

2.1.2 Méthodes d'injection en espace "noyau"

Nous avons identifiés six méthodes pour injecter du code ou des données dans le noyau Linux. La première passe par l'ajout dynamique de modules au noyau [Truff, 2003]. Cette méthode n'est possible que si le support par le noyau des LKM (*Linux Kernel Module*) est activé. Dans ce cas, des contre-mesures existent. Pour détecter des modules "noyau" malveillants, il est possible de mettre en place une infrastructure de vérification de l'intégrité des modules via l'utilisation de signatures cryptographiques [Microsoft Corporation, 2006]. Une autre approche prend le problème par l'autre bout. Elle est fondée sur une analyse comportementale des modules "noyau". Elle est effectuée avant ajout du module, afin de vérifier s'il s'agit ou non d'un *rootkit*. Cette approche s'articule sur de l'analyse statique de fichiers binaires. Un prototype fonctionnel existe [Kruegel *et al.*, 2004].

La deuxième méthode d'injection consiste à modifier l'image du noyau sur disque [Jbtzhm, 2002]. Lorsqu'une machine démarre, l'une des premières tâches effectuée par le système est de charger le noyau en mémoire. Ce noyau est décrit par un fichier binaire présent sur un périphérique de stockage. L'idée est d'insérer un module "noyau" malveillant dans ce fichier afin qu'au démarrage du système, le module soit chargé en même temps. Les contre-mesures mentionnées pour la première méthode s'appliquent également dans ce cas.

La troisième méthode consiste à corrompre le noyau en accédant à sa mémoire via le périphérique virtuel `/dev/kmem` [Sd et Devik, 2001; c0de, 2003]. Toutefois, des protections

[6] Évidemment, un *rootkit* "noyau" peut éventuellement effectuer des actions des classes 2 ou 3. Toutefois, ces actions ne sont pas celles qui le distinguent des autres *rootkits*. Cette constatation s'applique également aux autres types de *rootkit*.

comme *Grsecurity* peuvent être configurées afin d'interdire l'écriture ou la lecture sur ce périphérique. Malheureusement, une configuration de ce type bloque l'exécution du serveur graphique (le serveur X), à moins de désactiver la politique obligatoire de *Grsecurity*. Ce genre de protection est alors généralement désactivé sur un poste client.

La quatrième correspond aux exploitations de failles du noyau [Sqrkkyu et Twzi, 2007]. Certaines permettent l'injection de code alors que d'autres ont un périmètre beaucoup plus restreint. La difficulté avec cette approche est que l'exploitation n'est possible que sur les versions du noyau affectées par la faille.

La cinquième exploite les particularités des périphériques pouvant accéder au contrôleur de la mémoire physique sans intervention du processeur (c'est-à-dire accès DMA - *Direct Memory Access*). Ainsi, l'utilisation par exemple du bus Firewire permet de lire ou d'injecter des données en mémoire physique sans intervention du système d'exploitation [Dornseif *et al.*, 2005; Boileau, 2006][7]. Dans ce cas, l'accès a été commandé par un attaquant depuis l'extérieur de la machine. Il est également possible d'envoyer des requêtes DMA, en programmant les périphériques depuis le système d'exploitation. Il faut pour cela accéder aux ports d'E/S des périphériques. Sous Linux, deux façons sont envisageables. La première est d'utiliser `/dev/port`. Comme pour `/dev/kmem`, il suffit de lire et d'écrire à la bonne adresse correspondant au port voulu. La deuxième est d'utiliser les instructions de la CPU, `in` et `out` [Intel Corporation, 2008c] qui permettent l'accès aux ports d'E/S.

La dernière méthode identifiée est de corrompre le BIOS [Scythale, 2007]. Le BIOS est la première entité qui s'exécute sur le système. En d'autres termes, il permet d'exécuter du code avant que le système d'exploitation ne démarre. L'idée serait de modifier le BIOS pour qu'il charge un code de notre choix au démarrage du système et non le *bootloader* original. Ce code pourrait alors exécuter le noyau original après lui avoir appliqué certaines modifications.

Notre démonstrateur s'appuie sur de l'injection de code via `/dev/kmem`. Cette approche nous semble être un bon compromis[8]. En effet, un système dans lequel les LKM sont désactivés reste tout à fait utilisable sur un poste client, alors que la désactivation de `/dev/kmem` empêche le fonctionnement de certaines applications typiques sur ce genre de machine. Finalement, l'exploitation de failles du noyau n'est pas suffisamment fiable dans le temps.

2.1.3 Détournement des fonctions du noyau

Les premières techniques de détournement se fondent sur la modification de la table des appels-système [Sd et Devik, 2001]. L'attaquant redirige certains services du noyau vers des fonctions malveillantes qu'il a préalablement injectées en mémoire. Des méthodes de détection simples peuvent être toutefois mises en place. Certaines opèrent en comparant les adresses dans la table des appels-système avec une sauvegarde effectuée lors de l'installation du système. D'autres vérifient les emplacements du code des appels-système les uns par rapport aux autres.

Pour pallier ce problème, l'attaquant peut remonter un peu plus haut dans la chaîne et s'attaquer au gestionnaire des appels-système [Cesare, 1999b]. Dans une première phase, l'attaquant

[7] J. Rutkowska explique que la lecture en mémoire physique via un accès DMA peut aussi être mis en défaut [Rutkowska, 2007] (si la technologie matérielle de type IOMMU est présente sur la plateforme).

[8] Le dernier type d'injection mentionné n'a pas été étudié lors de la création de notre démonstrateur.

duplique la table des appels-système en mémoire et la modifie afin d'y inclure ses fonctions malveillantes. Ensuite, il modifie le pointeur vers cette table dans le gestionnaire des appels-système et le tour est joué. Les méthodes de détection précédentes ne sont plus efficaces car la table des appels-système originale est toujours présente en mémoire et n'est pas modifiée. Pour déjouer cette approche, la défense vérifie à présent, en plus, l'adresse de la table que le gestionnaire des appels-système utilise.

Pour effectuer un appel-système, il faut lever en espace "utilisateur" une interruption logicielle. Elle déclenche le passage en mode "noyau". Cette interruption peut être interceptée [Kad, 2002] en modifiant l'adresse du gestionnaire d'interruption correspondant à la procédure de traitement des appels-système. Pour détecter cette opération frauduleuse, il suffit de comparer l'adresse correspondante dans l'IDT à une sauvegarde effectuée lors de l'installation du système. Cependant, si le noyau n'autorise que l'utilisation de l'instruction `sysenter` cette approche de détection ne peut être mise en œuvre, car alors l'IDT n'est plus employée.

La table des interruptions comporte également l'adresse du gestionnaire de faute de page. Il s'agit d'une exception qui est levée par le processeur lorsqu'un accès est effectué à une page avec des privilèges non suffisants ou lorsque cette page n'est pas présente en mémoire. L'interception de cette interruption via la modification de l'IDT sert par exemple à injecter dans n'importe quel processus du code malveillant [Buffer, 2003]. De façon générale n'importe quelle interruption peut être détournée de sa fonction initiale.

En remontant encore dans le chemin d'exécution, l'attaquant peut, cette fois, dupliquer en mémoire l'IDT du système, la modifier et ensuite charger son adresse auprès du processeur à la place de l'actuelle [Kad, 2002]. Cependant il est facile de récupérer l'adresse de l'IDT que connaît le processeur via l'instruction `sidt`. Ainsi, il suffit de nouveau de comparer cette adresse avec une sauvegarde effectuée lors de l'installation du système, pour contrer cette technique.

L'attaquant peut aussi descendre dans les arcanes du noyau. Notamment, les fonctions du VFS (*Virtual File System*) peuvent être détournées par *hooking* (c'est-à-dire modification des pointeurs de fonctions), comme dans le *rootkit adore-ng* [Stealth, 2003]. Ainsi, l'attaquant cache les fichiers et processus qu'il désire. Cependant, le même type de détection que précédemment peut encore être employé.

Afin de s'affranchir de ce problème, l'attaquant peut, dans un premier temps, injecter en mémoire le code effectuant ses opérations malveillantes. Il l'appelle ensuite au sein du gestionnaire des appels-système juste avant l'endroit où sont exécutés les appels-système. Pour appeler son code dans le gestionnaire, l'attaquant peut employer par exemple les techniques de *hijacking* de Silvio Cesare [Cesare, 1999a]. Les solutions mises en place pour contrer ces techniques sont plus lourdes que les précédentes : il faut contrôler l'intégrité du code du gestionnaire.

Les solutions de détection, expliquées brièvement dans les paragraphes précédents, et d'autres plus évoluées [Rutkowski, 2002] sont mises à mal par certains *rootkits*. En effet, ce qui est lu par un programme de détection peut être filtré par le *rootkit* (celui-ci ayant la main sur les appels-système `read` et `write`, il peut contrer un programme de détection en espace "utilisateur"). Par conséquent il pourra renvoyer de fausses informations cachant sa présence.

Pour obtenir une détection efficace, des mécanismes peuvent être mis en œuvre en espace "noyau" où la lecture de la mémoire ne peut pas être interceptée aussi facilement. Nous trouvons des solutions réalisées sous forme de modules "noyau" comme *Saint Jude* [Lawless, 2002].

Des techniques plus sophistiquées sont alors nécessaires pour passer outre cette détection.

Par exemple, le *rootkit* Shadow Walker [Sparks et Butler, 2005] parvient à cacher ses données à l'intégralité du système. Qu'il s'agisse de l'espace "utilisateur" ou bien du noyau, les données ne sont visibles que par lui. Ce *rootkit* utilise une méthode analogue à celle mise en œuvre dans le *patch* de protection mémoire `PaX`. Nous revenons sur la technique employée dans la section 2.3.4.2.

Enfin, mentionnons des techniques de détournement d'exécution qui s'appuient sur l'installation de *callgate* dans la GDT (*cf.* section 1.4.5). Les *callgate* sont une spécificité architecturale des plateformes x86. Il s'agit d'un mécanisme qui autorise un processus à exécuter des fonctions à un niveau de privilège supérieur au sien. Typiquement, ce mécanisme peut être mis à profit par des processus de l'espace "utilisateur" (qui s'exécutent en *ring 3*) afin d'exécuter des fonctions en mode "noyau" (*ring 0*). La création d'une *callgate* requiert l'ajout d'une entrée dans la GDT, dans laquelle doivent être spécifiés notamment le *ring* depuis lequel la fonction doit s'exécuter, l'adresse de cette fonction et le nombre d'arguments qui doivent être passés à cette fonction. Des preuves de concept existent actuellement pour les systèmes Microsoft Windows [Crazylord, 2002; Kasslin, 2006].

Nous résumons ces différentes techniques au travers de la figure 2.1 qui illustre les principaux points du noyau que les *rootkits* sous Linux altèrent pour détourner le flux d'exécution.

Nous ne développons pas dans cet état de l'art tous les services que fournissent habituellement les *rootkits*. Cependant, la section 2.3 explique de façon détaillée les approches que nous avons élaborées afin de satisfaire certains de ces services. Avant d'entamer cette partie, nous développons, dans la section suivante, une réflexion sur l'élaboration de *rootkits*.

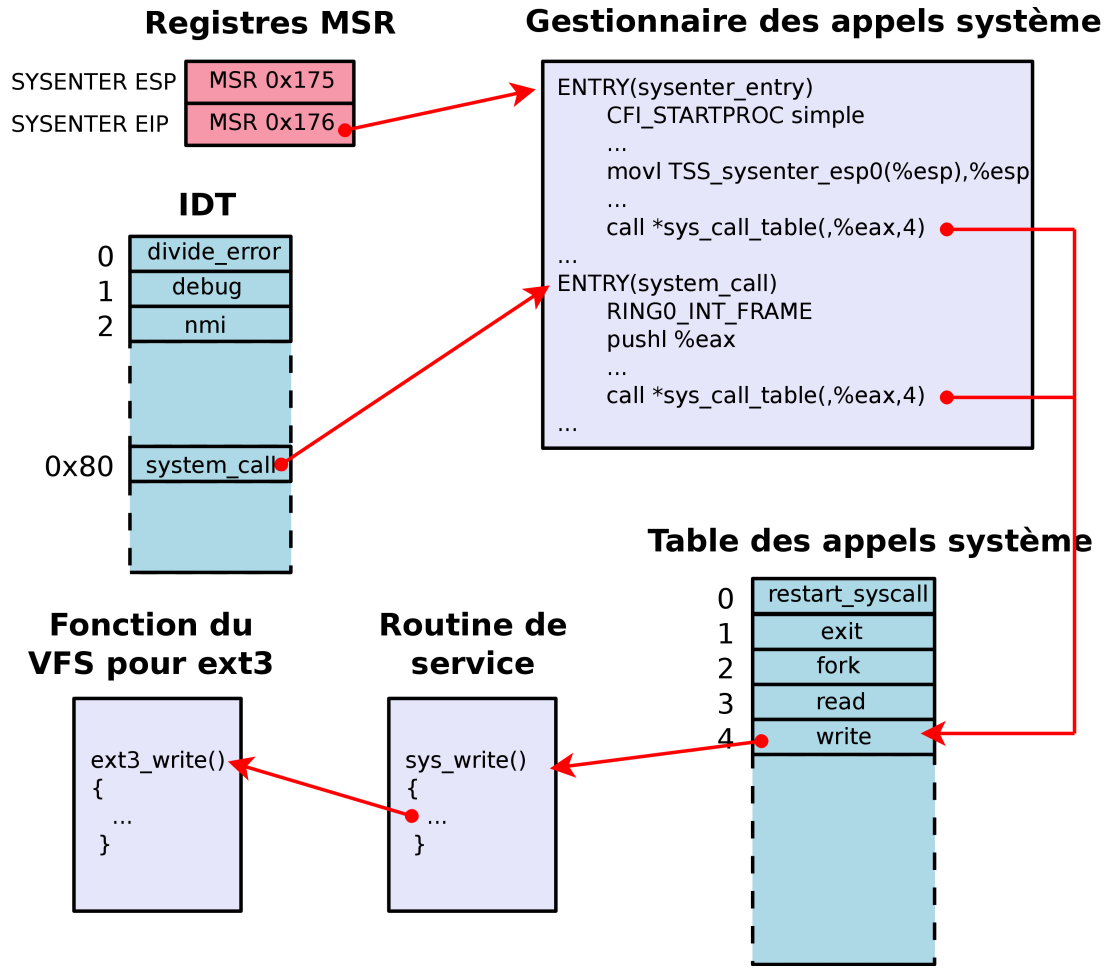

FIG. 2.1 – Principales cibles du noyau détournées par les *rootkits*

2.2 Caractérisation des *rootkits*

Cette partie aborde les éléments fondamentaux qu'un attaquant doit prendre en compte lorsqu'il construit un *rootkit*. Tout d'abord, nous analysons la définition d'un *rootkit*, afin de mieux saisir où se situent ces codes malveillants par rapport aux virus et autres chevaux de Troie. Ensuite, l'architecture d'un *rootkit* nécessite un certain nombre de composants qui doivent être développés ou réutilisés. Ces composants remplissent un certain nombre de fonctions essentielles que nous décrivons.

Un *rootkit* permettant à un attaquant de conserver le contrôle d'un système informatique *dans le temps*, il est impératif que l'attaquant puisse communiquer, interagir avec le *rootkit*. Nous développons donc dans une deuxième partie une réflexion sur ces besoins de communications entre l'attaquant et son *rootkit*.

Enfin, un attaquant doit choisir le *rootkit* le plus approprié en fonction de ses objectifs. Pour cela, il est important de disposer de critères objectifs, qui, à notre connaissance, font actuellement défaut. Nous introduisons, dans la dernière partie, trois attributs essentiels d'un *rootkit* afin de les évaluer. Notons que l'utilisation de ces attributs est bien évidemment intéressante du

point de vue des défenseurs qui tentent de se protéger des *rootkits* : mieux parvenir à caractériser ces derniers permet d'aider à leur détection mais aussi à leur éradication.

2.2.1 Analyse de la définition d'un *rootkit*

Nous avons défini un *rootkit* dans le chapitre 1 (section 1.2) comme un ensemble de modifications permettant à un attaquant de maintenir dans le temps un contrôle frauduleux sur un système informatique.

Plusieurs éléments dans cette définition sont caractéristiques d'un *rootkit* :

- *ensemble de modifications* : une première différence flagrante apparaît ici par rapport aux autres codes malveillants. D'une part, un *rootkit* est rarement un unique programme, mais est souvent constitué de plusieurs éléments. D'autre part, les multiples éléments d'un *rootkit* sont rarement des programmes autonomes, mais plutôt des modifications effectuées dans différents composants du système (programmes en espace "utilisateur", partie du noyau, ou autres). Ce type de modifications nous fait penser à la notion de *parasitisme* associée aux codes malveillants qui propagent leur charge utile en fonction de leurs cibles.
- *maintien dans le temps* : les autres codes malveillants n'ont pas réellement de relation au temps, sauf dans quelques cas pour les bombes logiques si le déclencheur est lié à ce facteur. Dans le cas d'un *rootkit*, un attaquant prend le contrôle du système pour y effectuer certaines opérations (vol d'information, rebond, déni de service, etc.) et doit fiabiliser son accès aussi longtemps que son objectif n'est pas atteint.
- *contrôle frauduleux sur un système d'information* : cela signifie que l'attaquant dispose des privilèges dont il a besoin pour effectuer ses opérations alors qu'il ne devrait pas être en mesure d'utiliser le système. La plupart du temps, l'attaquant cherche à maintenir son contrôle à l'insu des utilisateurs légitimes du système, mais ce n'est pas une nécessité. Par ailleurs, cela suppose des interactions, et donc une communication, entre le système et le détenteur du *rootkit*.

2.2.2 L'architecture d'un *rootkit*

Nous décrivons dans cette section l'architecture des *rootkits*. Afin d'identifier les différents éléments qui la compose, nous déterminons les fonctions essentielles que doit remplir le *rootkit*. Avant d'être utilisé, le *rootkit* doit être installé, d'où le besoin d'une fonction d'injection. Une fois en place, l'attaquant souhaite alors pérenniser son accès au système compromis par l'intermédiaire du *rootkit*. Ce dernier doit donc disposer d'une fonction de protection, mais également d'un moyen assurant la communication entre l'attaquant et le système (c'est-à-dire une porte dérobée ou *backdoor*). Enfin, l'attaquant souhaite effectuer les opérations nécessaires à l'accomplissement de ses objectifs. Le *rootkit* doit alors fournir des services qui répondent à ces besoins.

Nous décrivons dans la suite ces différents composants. La figure 2.2 expose les interactions d'une part, entre l'attaquant et le *rootkit*, et d'autre part, entre les composants du *rootkit*. Nous illustrons essentiellement le cas des *rootkits* "noyau". Ainsi tous ces composants se situent dans l'espace "noyau" du système compromis, ou du moins sont en partie implémentés dans cet espace. Certaines approches, comme celle que nous avons conçue (*cf.* section 2.3), emploie un

processus complice dans l'espace "utilisateur" pour la mise en œuvre de certaines des sous-fonctions des composants.

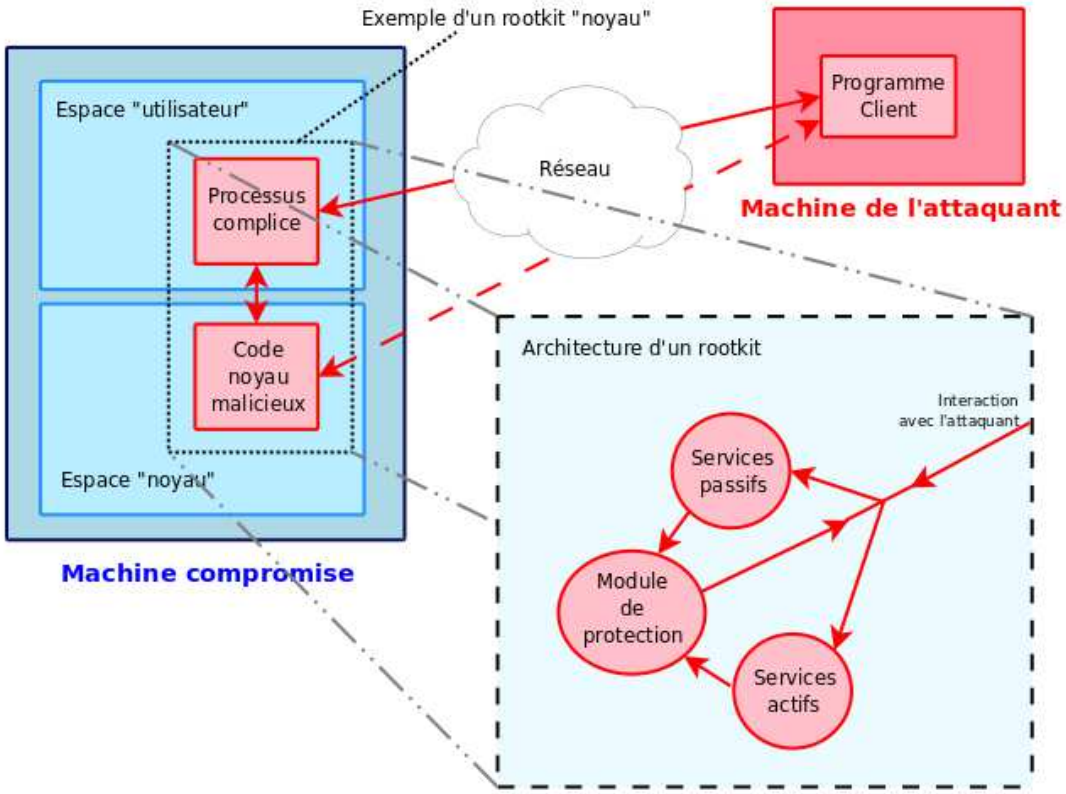

FIG. 2.2 – L'architecture d'un rootkit

2.2.2.1 Un injecteur

On appelle injecteur le moyen que l'attaquant emploie afin d'implanter le *rootkit* dans le système (injection dans des modules "noyau", injection de code via `/dev/kmem`, etc.). En effet, que l'attaquant parvienne à rentrer sur le système en exploitant une faille logicielle ou un mot de passe faible, pour s'installer sur le système, il doit y placer son *rootkit*. La nature de l'injecteur reste constante quelque soit le *rootkit*. En effet, qu'il s'agisse d'un *rootkit* en espace "utilisateur" ou "noyau", ou même "hyperviseur", son installation requiert l'accès puis la modification à certaines structures du système, une seule et unique fois[9].

2.2.2.2 Un module de protection

Le module de protection a pour but de permettre au *rootkit* de survivre tout le temps nécessaire à l'attaquant. Plusieurs stratégies sont envisageables et combinables. Citons à titre

[9] Comme dans le cas de notre *rootkit* (*cf.* section 2.3), l'injecteur peut utiliser un mécanisme de *bootstrap* qui sert d'amorçage à des modifications plus profondes et complexes.

d'exemples :

- *Dissimuler le rootkit :*
  Deux cas sont à distinguer. D'une part, il s'agit de le dissimuler sur le système lorsque celui-ci est en cours d'exécution. D'autre part, si le *rootkit* est persistant, la portion de code résidant sur le système compromis y est dissimulée.
- *Rendre résistant le rootkit :*
  Nous supposons ici que la présence du *rootkit* a été détectée. Il s'agit alors de munir le *rootkit* de capacités lui permettant de résister aux tentatives de suppression. Par exemple, s'il détecte qu'il a été détecté, il peut menacer l'utilisateur légitime de "détruire" le BIOS de la carte mère si cet utilisateur tente d'y accéder. Cela repose en fait sur la théorie du jeu : le *rootkit* cherche à placer l'utilisateur légitime dans une position où il a plus à perdre à réinstaller complètement le système qu'à vivre avec le *rootkit*.
- *Rendre persistant le rootkit :*
  Il s'agit de mettre en œuvre des mécanismes qui permettent au *rootkit* de "survivre" au redémarrage de la machine. Par exemple, un code de *bootstrap* peut être injecté dans des éléments non volatiles du système.
- *Camoufler l'activité de l'attaquant :*
  Le camouflage se révèle, d'une part, dans les fonctionnalités ayant pour but la dissimulation de processus, de *sockets* "réseau" ou de fichiers utilisés par l'attaquant. D'autre part, il consiste à filtrer des journaux d'évènements du système compromis.

2.2.2.3 La porte dérobée

La porte dérobée permet d'une part à l'intrus de conserver le contrôle (niveau de contrôle dépendant de ce qu'il cherche à faire sur le système[10]), et d'autre part d'accéder aux services. La porte dérobée est le point central du *rootkit*, un peu comme un noyau qui joue le rôle d'interface entre l'extérieur (l'intrus) et les services fournis par le *rootkit*.

La porte dérobée se caractérise par un vecteur d'interaction avec le système, que nous découpons en deux parties distinctes et indépendantes l'une de l'autre :

- *Du système de l'attaquant vers le système compromis :*
  Il s'agit du moyen de communication que l'attaquant utilise afin de piloter le *rootkit* (connexion de l'attaquant via un compte existant sur le système compromis, communication au travers d'un canal caché, etc.). Nous développons cette problématique dans la section 2.2.3.
- *Du système compromis vers les services du rootkit :*
  Cette partie caractérise le chemin qu'emprunte le *rootkit* lorsqu'il répond aux requêtes de l'attaquant. (ex : *hooking* de la table des appels-système, *hooking* du *Virtual File System*, *hijacking* de fonctions, etc.) afin de satisfaire les exigences associées à ses objectifs.

La porte dérobée est également associée à un mécanisme qui l'active. Nous l'appelons *actionneur de la porte dérobée* (*backdoor-actuator*).

[10] Dans certains cas, l'attaquant n'a pas besoin du maximum de privilèges sur le système, comme quand il cherche par exemple à espionner un utilisateur donné.

2.2.2.4 Les services

Un *rootkit* fournit plusieurs services grâce auxquels l'attaquant effectue les opérations dont il a besoin sur le système compromis. Nous distinguons deux catégories de services :

- *Les services passifs ou l'espionnage :*
  Au travers des fonctionnalités d'espionnage, l'attaquant peut obtenir des informations sensibles transitant sur le système compromis. Un exemple typique d'un tel service est le *keylogger* qui intercepte les caractères saisis sur le clavier de système.
- *Les services actifs :*
  Il s'agit généralement des services que l'attaquant emploie afin d'effectuer des attaques sur d'autres systèmes (dénis de service, suppression d'information ou de "logiciels", etc.). Ils correspondent également aux services intermédiaires qui permettent à l'attaquant de rebondir sur d'autres systèmes afin de poursuivre son intrusion plus avant.

2.2.3 La communication avec le *rootkit*

Lors d'une intrusion informatique, nous distinguons trois phases pendant lesquelles les communications jouent un rôle principal :

1. lors de l'intrusion à proprement parler, qui permet à l'intrus de compromettre la sécurité du système cible ;
2. une fois le système sous contrôle, il s'agit d'y transférer le *rootkit* puis de l'installer ;
3. enfin, lors de l'utilisation du système compromis, l'intrus envoie ses instructions et récupère éventuellement les résultats.

Cette section détaille les deux dernières phases puisque nous nous concentrons sur les *rootkits*, et non les techniques d'intrusion. Il est à noter que les problèmes liés à la réalisation de ces deux phases ne se posent que dans certaines conditions. La phase de récupération du *rootkit* fait nécessairement suite à une attaque à distance. Lorsque l'attaque est locale, nous imaginons bien que l'intrus transporte avec lui non seulement ses outils offensifs, mais également de quoi maintenir son accès au système. Quant à la troisième phase, l'utilisation à distance du *rootkit* suppose que l'attaquant n'y ait pas accès localement, et que de plus il souhaite maintenir son accès dans le temps.

Ces communications engendrent nécessairement un transit d'informations à l'extérieur du système compromis. Quand bien même l'attaquant serait invisible sur le réseau, il va laisser des traces sur l'hôte compromis comme l'utilisation de *sockets* ou les statistiques d'envoi/réception de paquets sur les interfaces réseau. La dernière partie de cette section traite de ce problème.

2.2.3.1 La récupération du *rootkit*

Une fois qu'il a le contrôle du système cible, une des premières tâches entreprise par l'attaquant est de pérenniser son accès au système. Nous distinguons généralement deux approches :

- classique : il rapatrie d'un site tiers son *rootkit*, et l'installe (un préambule nécessaire peut également être l'exécution d'un exploit local pour acquérir les privilèges nécessaires à l'installation du *rootkit*) ;

– *tout en mémoire* : aucun octet n'est écrit sur le disque à aucun moment de l'intrusion, tout se passant dans la mémoire du processus compromis, ou d'autres processus du système [Caceres, 2002; Grugq, 2004; Pluf et Ripe, 2005; Dralet et Gaspard, 2006].

Dans les deux cas, il s'agit pour l'attaquant de se connecter à un site externe, puis de transférer des données. Que les octets téléchargés soient stockés dans un fichier ou en mémoire ne change pas la problématique par rapport au réseau : il y a une connexion vers une base de l'intrus, il cherche donc à la protéger. Ainsi, la sécurité de l'attaquant se trouve confrontée à plusieurs risques :

– la base, c'est-à-dire l'endroit d'où un intrus télécharge son outil, est potentiellement connue après une utilisation ;
– si le flux est intercepté, le défenseur est susceptible de reconstruire le *rootkit*, et par conséquent peut connaître avec exactitude les actions entreprises par le pirate sur le système.

Les auteurs de [Raynal *et al.*, 2004] ont montré que ces risques sont réels. À partir d'une capture effectuée sur un pot à miel, il a été possible de retrouver plusieurs bases de l'intrus, de s'y connecter et de récupérer de multiples outils. En outre, l'analyse du *rootkit* a fourni une explication détaillée de la compromission.

Pour protéger sa base, l'attaquant peut soit choisir de passer par des relais et autres méthodes d'anonymisation, mais l'effort nécessaire pour les mettre en œuvre semble disproportionné par rapport à l'autre choix : utiliser l'immensité des services d'Internet. Il existe en effet de nombreux endroits où stocker des données pour ensuite les récupérer : *newsgroups*, sites *ftp* de partage, réseaux P2P, etc. Un attaquant consciencieux prendra soin d'y placer son *rootkit* juste avant son attaque, en le chiffrant comme nous verrons ci-après, avec une *clé environnementale*.

Face au risque de décodage et reconstruction du flux en cas d'interception, la parade est connue : le chiffrement. En effet, si le flux est correctement chiffré, même une interception empêche toute reconstruction ... sauf si la clé est également récupérée. Or, elle va devoir transiter par le réseau pour finalement être stockée sur le système avant de déchiffrer le *rootkit*.

Une solution à ce problème est mise en œuvre par le virus Bradley, analysé par Éric Filiol dans [Filiol, 2007b, 2005]. Ce virus comporte plusieurs couches de chiffrement successives, la clé pour déchiffrer une couche étant calculée par le niveau précédent. De cette manière, il est impossible d'analyser le virus tant que la première couche n'a pas calculé la bonne clé. De plus, le virus ne se retrouve jamais complètement en clair en mémoire, mais seulement par morceau, qui s'effacent lorsqu'ils ont terminé leurs actions. Le protocole cryptographique employé pour obtenir ce résultat repose sur la notion de *clés environnementales* [Riordan et Schneier, 1998]. Pour les auteurs, un code mobile chiffré évolue en milieu hostile, et doit donc protéger sa clé de déchiffrement. Pour cela, il ne doit donc pas la transporter en son sein, mais la reconstruire à partir de différentes informations issues de son environnement. Le virus Bradley utilise plusieurs méthodes de reconstruction de clés. Dans le cas d'un *rootkit*, il s'agit de rendre la clé dépendante à la fois d'une information propre à la cible, et d'un secret externe, connu seulement de l'attaquant.

La figure 2.3 illustre ce mécanisme. Les blocs EVP_i sont chiffrés avec une clé calculée par le niveau antérieur. La première clé est calculée par exemple avec une information dépendant de la cible (son adresse ou un nom d'utilisateur) et un secret sous le contrôle de l'attaquant, comme l'empreinte d'une page web ou encore le champ RR d'une réponse DNS, etc.

Ainsi, même si le *rootkit* est capturé, son analyse est impossible sans la connaissance de tous les éléments pour le déchiffrer. Il a été prouvé dans [Filiol, 2005, 2007b] que le problème

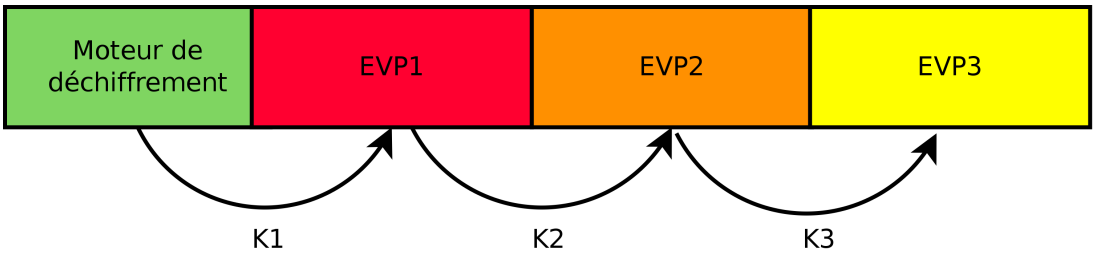

FIG. 2.3 – Structure du virus Bradley

général de l'analyse du code du virus Bradley est équivalent à la cryptanalyse d'un algorithme cryptographique sûr, dans le sens où sa complexité est exponentielle.

2.2.3.2 Le canal de commande

Dans le canal de commande, il y a deux choses à protéger. Pour le contenu de la communication lui-même, de nombreux protocoles mettent déjà en place une protection sous forme de chiffrement (*ssl*, *ssh*, ou encore *ipsec*), mais cela n'est pas suffisant. En effet, un administrateur consciencieux pourrait remarquer des anomalies dans le comportement réseau de son système, par exemple, si les instructions parviennent au système à des heures nocturnes supposées calmes. Là encore, des solutions connues et efficaces existent : les canaux cachés [Girling, 1987; Wolf, 1989; Rowland, 1996; Raynal, 2003], ou des techniques de "simulabilité" statistique [Filiol, 2007b; Filiol et Josse, 2007].

En mode "noyau", la pile réseau est accessible, ce qui laisse un choix assez large pour l'installation d'un canal de communication. Le niveau du protocole choisi pour placer un canal caché influence son utilisation. Si le choix se porte sur IP ou TCP par exemple, des équipements en coupure sont susceptibles de modifier les en-têtes des paquets : *load balancer*, *traffic shaper*, ou même des *proxies*. Inversement, de plus en plus d'outils matériels tentent d'intercepter les flux réseau pour les analyser et en vérifier la conformité aux normes. Heureusement pour l'attaquant, ces méthodes sont encore assez peu fiables. Par exemple, au niveau TCP, très peu d'équipements font du suivi précis de numéro de session[11], ce qui permet de désynchroniser la vue du flux, selon que nous nous plaçons sur l'équipement de surveillance ou le destinataire.

Mais il est également possible d'agir via des protocoles plus hauts, comme DNS ou HTTP(S) qui sont souvent autorisés à sortir du réseau local. En outre, l'intérêt de passer par de tels protocoles, mais en mode "noyau", est que les outils d'analyse présents sur le système compromis ne verront rien. En effet, les données sont construites en espace "utilisateur", avant de passer en espace "noyau" pour être envoyées par le pilote de périphérique à destination. Inversement, lors de la réception, les données arrivent sur le pilote, passent par la pile réseau située en espace "noyau", puis sont enfin transmises à l'application en charge de les traiter. Ainsi, un code qui agit en espace "noyau" peut éviter tous les mécanismes d'analyse[12] puisque la majorité se contentent d'agir en espace "utilisateur". Ainsi, un *rootkit* "noyau" peut parasiter les communi-

[11] Vérifier la cohérence des numéros de séquence et d'acquittement de tous les paquets, en plus d'autres vérifications, conduirait inexorablement à un ralentissement voire un engorgement du réseau au niveau de l'appareil vérificateur.
[12] Nous pensons aux anti-virus ou autres HIPS.

cations :

- quand des données sont émises : il ajoute ses propres octets au paquet juste avant de le passer au pilote ;
- quand des données sont reçues via le pilote : le *rootkit* les retire puis les retransmet, assainies, à l'application légitime qui les attend.

À noter que dans le cas de notre *rootkit*, celui-ci fonctionnant en mode "noyau", il agit avant toute règle de filtrage. Ainsi, tant qu'une interface réseau est active, et même si le pare-feu interne de la machine bloque les connexions entrantes et sortantes, notre *rootkit* peut continuer à communiquer avec l'extérieur.

2.2.3.3 Les traces d'activité "réseau" dans le système

Comme un *rootkit* "noyau" ne met pas explicitement un port en écoute puisqu'il agit en mode "noyau", des commandes agissant en espace "utilisateur", comme `netstat`, ne révéleront pas sa présence. Néanmoins, l'exemple de Sebek [Honeynet Project, 2003] montre qu'il n'est pas si facile de cacher la présence d'un *rootkit* "noyau" traitant des connexions réseau [Bioforge, 2003] lorsqu'il agit sur le système. Par exemple, dans les premières versions de Sebek, les statistiques d'envoi et réception des paquets augmentaient, même si un *sniffer* branché directement sur l'interface ne capturait aucun paquet.

Nous ne détaillons pas plus ici cet aspect. Néanmoins, il est important de le conserver à l'esprit.

2.2.4 Vers l'évaluation d'un *rootkit*

Les stratégies envisageables pour le module de protection d'un *rootkit* (*cf.* section 2.2.2), nous aident à identifier ses attributs essentiels. Les stratégies de dissimulation du *rootkit* révèlent un attribut d'*invisibilité* (notion sur laquelle nous revenons ci-dessous). Celles qui cherchent à rendre résistant ou persistant le *rootkit* mettent en évidence sa *robustesse*. Enfin, quelque soit le niveau d'invisibilité ou de robustesse associé au *rootkit*, ce dernier a pour but de modifier le système sur lequel il se trouve pour permettre à l'attaquant de maintenir de façon durable le contrôle sur ce système. Un troisième attribut permettant d'exprimer cette modification que fait un *rootkit* sur le système compromis nous semble également pertinent pour évaluer un *rootkit*.

L'analogie avec le tatouage (ou filigrane, en anglais *watermarking*) peut nous aider dans l'expression de ces attributs [Filiol, 2007a]. En effet, la problématique de la conception d'un *rootkit*, du point de vue de l'attaquant, est proche de celle du tatouage : il s'agit de modifier un support sain, appelé *stégano-médium* de façon à y dissimuler un message secret.

Dans notre analogie, le stégano-medium est un système informatique, et le message secret à dissimuler est un ensemble de données et d'actions liées à un *rootkit*. Les critères habituellement utilisés en stéganographie sont l'*invisibilité* (le message secret doit bien sûr être dissimulé, mais l'existence même de la communication ne doit pas être visible), la *robustesse* (une transformation du stégano-médium ne doit pas altérer le message secret) et la *capacité* (qui exprime la quantité d'information dissimulée dans le stégano-médium).

Nous proposons d'utiliser les deux premières propriétés et d'en donner une définition adaptée dans le cadre des *rootkits*. En effet, comme nous l'avons déjà abordé ci-dessus, les notions

d'invisibilité et de robustesse sont naturellement transposables au monde des *rootkits*. En revanche, l'analogie avec la stéganographie s'arrête ici car il nous semble inapproprié de transposer directement la notion de capacité au *rootkit*. En effet, les informations dissimulées dans un stégano-médium sont passives alors que celles qui sont dissimulées dans un système informatique par un *rootkit* sont à la fois passives et actives : elles concernent bien sûr des modifications de fichiers mais aussi de données en mémoire ainsi que d'activités telles que des processus. Ces modifications se produisent lors de l'insertion elle-même du *rootkit* mais aussi lors de l'exécution de toutes les activités malveillantes réalisées à partir du *rootkit*. Aussi, même si un *rootkit* dissimule effectivement des informations dans le système, la nature de ces informations ne nous permet pas d'utiliser le critère de capacité tel quel.

Nous proposons donc d'introduire comme troisième attribut essentiel d'un *rootkit* une grandeur permettant en quelque sorte de mesurer la malveillance du *rootkit*, c'est-à-dire la façon dont il corrompt le système. Cet attribut nous semble être proche de la notion de *virulence* qui est définie par Éric Filiol [Filiol, 2004] et qui s'applique habituellement aux virus. Cette notion est définie ainsi :

Définition 1.

$$virulence = I_v^0 * I_v^1$$

où :

- I_v^0 *représente le rapport entre le nombre de fichiers infectables par le virus* v *et le nombre total de fichiers sur le système.*

- I_v^1 *représente le rapport entre le nombre de fichiers infectés (lorsque l'infection n'évolue plus) par le virus* v *et le nombre de fichiers infectables par ce virus.*

Cette notion de virulence est plus adaptée que la notion de capacité car elle nous indique non seulement la quantité d'information introduite dans le système mais aussi à quel point le système est infecté. Cependant, cette notion fait intervenir des fichiers comme support des modifications, or les modifications réalisées par un *rootkit* ne se limitent pas uniquement aux fichiers. De plus, à la notion de virulence manque un des aspects les plus importants pour les *rootkits* que nous essayons de décrire. Il s'agit de mesurer l'importance des éléments affectés par le *rootkit* vis-à-vis des possibilités qu'ils lui donnent sur le contrôle du système[13]. Il s'agit en quelque sorte de mesurer la *gravité* de l'action du *rootkit*.

Définissons donc les trois attributs ainsi :

Définition 2. *L'*invisibilité *d'un* rootkit *exprime la difficulté pour l'utilisateur du système de détecter le* rootkit *lui-même ainsi que les activités malveillantes exécutées à l'aide de ce* rootkit.

Définition 3. *La* robustesse *d'un* rootkit *exprime la difficulté de retirer un* rootkit *d'un système sur lequel il est installé.*

Définition 4. *Le* pouvoir de nuisance *d'un* rootkit *exprime le degré d'ingérence du* rootkit *dans le système, c'est-à-dire la quantité d'éléments du système qui sont affectés par le* rootkit *et l'importance de ces éléments dans l'utilisation du système.*

[13] Notons que la virulence en termes médicaux est l'aptitude des microbes à se développer dans l'organisme.

Soulignons que la notion d'invisibilité englobe le *rootkit* lui-même ainsi que les différentes activités que l'attaquant parvient à exécuter sur le système compromis à travers le *rootkit*. De plus, elle regroupe à la fois la dissimulation d'objets passifs (typiquement des fichiers)[14], mais aussi la dissimulation d'activités (typiquement des processus). Éric Filiol dans son ouvrage [Filiol, 2007b] propose des définitions pour le *camouflage* (dissimulation d'objets passifs) et pour la *furtivité* (dissimulation d'activités). La notion d'invisibilité telle que nous l'avons définie englobe donc ces deux notions de camouflage et de furtivité.

La robustesse concerne la difficulté d'éradiquer le *rootkit* lorsque le système est en cours d'exécution mais aussi lorsque le système est arrêté puis éventuellement redémarré. Ces deux notions sont déjà classiquement abordées pour l'éradication de maliciels en général, en particulier de virus ou de vers. Nous utilisons donc ici les termes qui leur sont habituellement associés, à savoir la *résistance* et la *persistance*. Ces notions ont été présentées dans la section 2.2.2 au travers de l'explication des stratégies envisageables pour le module de protection d'un *rootkit*. La robustesse englobe à la fois ces deux notions de résistance et persistance.

Le pouvoir de nuisance d'un *rootkit* doit nous permettre d'évaluer à quel point est compromis le système sur lequel le *rootkit* est inséré. Il indique jusqu'où le *rootkit* s'est propagé dans le système et donne donc une évaluation de l'étendue des dégâts causés.

Nous voyons déjà poindre différentes stratégies possibles pour la création de *rootkits* : un attaquant peut par exemple opter pour de petites modifications, difficilement identifiables mais faciles à réparer ; ou bien il peut opter pour un *rootkit* qui s'insinue partout, rendant le système pratiquement impossible à assainir (et donc mettre l'accent sur la robustesse).

Ces trois propriétés étant définies, il faut ensuite leur associer une mesure. En effet, si nous sommes capables de mesurer l'invisibilité, la robustesse et le pouvoir de nuisance d'un *rootkit*, nous sommes alors capables d'évaluer de façon objective chaque *rootkit* et de le comparer à d'autres.

Concernant la mesure de l'invisibilité, nous pouvons établir un parallèle avec les travaux de Christian Cachin [Cachin, 1998] concernant la formalisation de systèmes stéganographiques. Cachin propose d'utiliser la théorie de l'information et les tests statistiques. Il introduit ainsi une définition formelle d'un système stéganographique et propose de mesurer la fiabilité de ce système par un calcul probabiliste. Cette approche a été employée afin de définir ce qu'est "l'invisibilité" pour des programmes et permet également de traiter du problème de la détection de cette "invisibilité" [Filiol, 2007a].

À notre connaissance, il n'existe pas de travaux en cours permettant d'associer une mesure à la notion de robustesse telle que nous la définissons pour un *rootkit*. Il existe cependant le concept de *survivabilité* de réseau [Chen *et al.*, 2002], qui pourrait être un bon point de départ dans la définition d'une mesure adéquate de la robustesse des *rootkits*. En effet, la survivabilité de réseau reflète l'aptitude d'un réseau à continuer de fonctionner durant et après l'occurrence de défaillances. Nous faisons ici directement le parallèle avec la résistance qu'un *rootkit* oppose à toute action de retrait (analogues aux *défaillances dans le réseau*) du système compromis.

Enfin, pour mesurer le pouvoir de nuisance d'un *rootkit*, nous pourrions penser, dans un premier temps, nous reposer sur des tests d'intégrité de fichiers. Néanmoins, les fonctions de ha-

[14] Les activités exécutées par l'attaquant peuvent engendrer une production de données, qui doivent être également dissimulées (si elles sont utiles au déroulement de l'activité) ou bien supprimées (par exemple, les traces dans les différents journaux systèmes).

chage classiques, puisque respectant le principe d'avalanche[15], sont trop sensibles à la moindre modification apportée au système[16]. Pour cela, nous préférons alors des outils comme la distance de Levenshtein qui permet de mesurer la similarité entre deux chaînes de caractères[17] (par exemple des exécutables). Mais le pouvoir de nuisance d'un *rootkit* ne se mesure pas uniquement sur les modifications de fichiers dans le système. Il faut également être capable d'évaluer les modifications apportées à des activités dans le système (processus), ce qui est plus délicat puisque ces modifications s'appliquent à la fois à la structure des processus mais également à leur état.

Dans la suite de ce chapitre, nous nous focalisons principalement sur l'attribut d'invisibilité.

2.3 Exemple de construction d'un *rootkit* "invisible"

Dans cette section, nous présentons le *rootkit* "invisible" que nous avons réalisé. Nous présentons tout d'abord notre technique de détournement, mise en œuvre sur un noyau Linux 2.6 sur architecture `x86` (technique qui constitue notre vecteur d'interaction avec le système). Nous abordons ensuite la problématique de la dissimulation du *rootkit* et de l'installation de sa porte dérobée. Nous finissons sur les aspects de furtivité des activités de l'attaquant[18], qui font parties du module de protection de l'architecture fonctionnelle d'un *rootkit* présentée dans la section 2.2.2.

2.3.1 Principe original de notre *rootkit* : sa porte dérobée

Notre approche consiste à ne corrompre qu'un seul processus ou un seul fil d'exécution (ou *thread* en anglais, terme que nous employons dans la suite de ce manuscrit) dans le système. L'originalité réside donc ici dans le fait que les autres processus s'exécutant sur le système ne voient aucune modification de leur environnement, ce qui n'est pas le cas des approches que nous avons présentées précédemment.

La technique de *hooking* d'appels-système par processus [Pluf, 2006] n'est pas comparable à la notre car elle agit uniquement au niveau de l'espace "utilisateur"[19] : aucune exécution de code "noyau" malveillant n'est possible. Aussi, les modifications effectuées sur le *thread* du processus infecté affectent l'ensemble des *threads* composant le processus, alors que la granularité de notre approche est le *thread*.

Nous utilisons dans notre approche l'appel-système 0 de façon détournée. Il est normalement employé par le noyau pour relancer certains appels-système interrompus avec de nouveaux

[15] Le principe d'avalanche implique pour une bonne fonction de hachage qu'un bit modifié en entrée change en moyenne la moitié des bits de sortie.

[16] Les fonctions de hachage ne sont pas adaptées pour évaluer un *rootkit*. Cependant, du point de vue de l'administrateur, cela est suffisant pour déterminer que le système a été compromis s'il détecte qu'un fichier a été modifié alors qu'il n'aurait pas dû l'être.

[17] La distance de Levenshtein est égale au nombre minimal de caractères qu'il faut supprimer, insérer ou remplacer pour passer d'une chaîne à l'autre. Elle peut être considérée comme une généralisation de la distance de Hamming.

[18] Nous ne traitons pas ici de la dissimulation de fichiers.

[19] La technique consiste à remplacer en mémoire l'instruction `int 0x80` par `int 3` dans les *wrappers* d'appels-système de la *glibc*. Ainsi, le signal `sigtrap` est envoyé au processus lorsqu'il effectue un appel-système. Ce signal est alors intercepté par un gestionnaire malveillant.

paramètres de façon transparente pour l'espace "utilisateur". Ce cas se présente par exemple lorsqu'un processus endormi (via `sys_nanosleep`) doit être réveillé afin d'exécuter un gestionnaire de signal. Après avoir traité le signal, le processus doit être endormi de nouveau (si nécessaire) durant une période plus courte : la durée initiale moins la durée d'exécution du gestionnaire. Pour cela, la fonction `sys_nanosleep` est relancée via l'appel-système 0 avec cette nouvelle durée. Étant donné que l'appel-système à relancer est propre à chaque processus (ou *thread*) et peut varier en fonction du temps, une référence à cet appel est conservée pour chaque *thread*. Ainsi, au moment où un appel-système (parmi ceux qui doivent être relancés) est exécuté, son adresse est stockée temporairement dans le descripteur du processus l'ayant appelé. Plus précisément, elle se trouve dans une structure `thread_info` liée au descripteur (fig. 2.4).

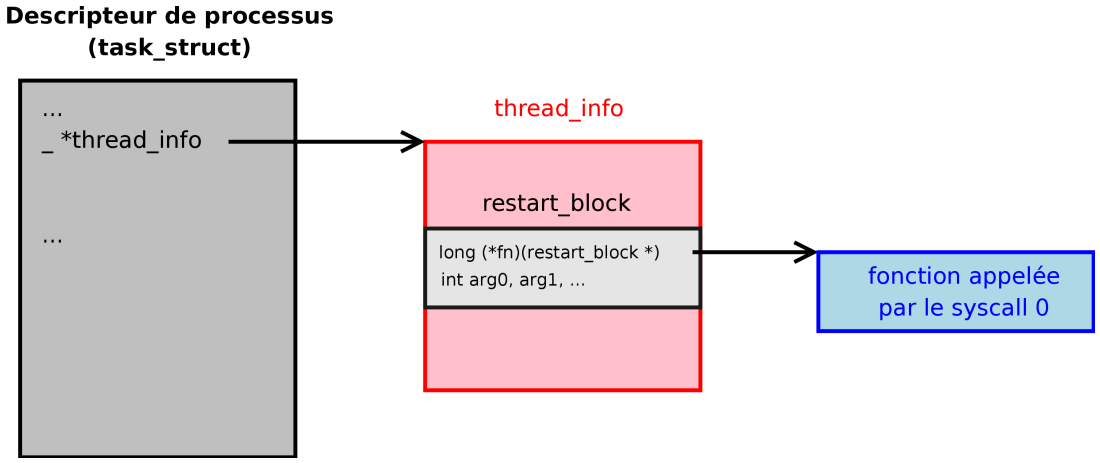

FIG. 2.4 – Le descripteur de processus et l'appel-système 0

Notre technique de détournement consiste à modifier cette adresse. Ainsi, nous pouvons exécuter en *ring 0* n'importe quelle fonction de l'espace "noyau" (ou code arbitraire préalablement injecté dans l'espace "noyau") depuis l'espace "utilisateur". Et cela est visible uniquement depuis le processus modifié.

2.3.2 Architecture de notre *rootkit*

Vis-à-vis de l'architecture introduite dans la section 2.2.2 (figure 2.2), nous présentons la façon dont est conçu notre *rootkit*. Il consiste tout d'abord en une porte dérobée en espace "noyau" qui est protégée grâce à une stratégie de camouflage (*cf.* section 2.3.4). Cette partie de la porte dérobée comprend :

1. le code qui modifie la sémantique de l'appel-système 0 afin d'appeler n'importe quelle fonction du noyau (*cf.* section 2.3.1) et ;
2. le code qui autorise l'attaquant à injecter de nouvelles fonctionnalités si elles ne sont pas fournies par le noyau.

L'attaquant construit ensuite ses opérations en combinant l'utilisation de fonctions du noyau et de celles du code qu'il a injecté. Il agit alors soit depuis le système compromis, soit depuis

un autre ordinateur qui n'appartient pas au réseau de la machine compromise. Dans le premier cas, ses opérations (qui consistent en des séquences d'appels-système 0) sont effectuées depuis sa session active sur la machine compromise. Ainsi, le processus qui est le support de cette session a besoin d'être activé par notre actionneur de porte dérobée. Cela est présenté dans la section 2.3.5. Dans le dernier cas, un processus complice est mis en œuvre sur le système compromis afin de relayer les commandes (c'est-à-dire les appels-système 0) qui ont été émises par l'attaquant, à la partie de la porte dérobée en espace "noyau".

Nous décrivons dans la section suivante, le mode d'installation basique de notre *rootkit* à partir du processus de l'attaquant. Un mode d'installation plus évolué permet de dissimuler ce processus avant d'effectuer l'installation dudit *rootkit*. Cependant, nous n'expliquons pas cette dissimulation ici mais dans la partie 2.3.7.1, traitant de l'invisibilité.

2.3.3 Installation préliminaire : la partie "noyau" de la porte dérobée

Tout d'abord depuis notre processus (celui de l'attaquant), nous ouvrons le périphérique virtuel `/dev/kmem` pour accéder à l'espace d'adressage du noyau. Il nous faut pour cela avoir les bons privilèges (acquis avant l'installation du *rootkit*). Nous recherchons ensuite au travers de `/dev/kmem`, la primitive `get_zeroed_page()` (plus exactement son adresse) par "appariement de formes" (*pattern matching*) sur le début de la fonction. Cette primitive est l'appel de plus bas niveau de l'allocateur mémoire du noyau. Elle réserve une page de mémoire physique et renvoie son adresse. Par la suite, nous recherchons l'emplacement de la pile de notre processus (ainsi que l'emplacement de son descripteur)[20].

L'étape suivante consiste à injecter un code de *bootstrap* au fond de la pile "noyau" de notre processus. Ce code est uniquement constitué d'un appel à la primitive précédente (`get_zeroed_page()`) et renvoie l'adresse de la page allouée. Son exécution est lancée depuis le processus de l'attaquant en détournant l'appel-système 0. Pour cela, nous remplaçons tout d'abord l'adresse de la fonction exécutée par l'appel-système 0 par celle du code que nous venons d'injecter. Ensuite, nous exécutons, depuis notre processus, l'appel-système 0 sans aucun paramètre. Le code que nous avons injecté s'exécute ainsi en *ring 0* et nous récupérons l'adresse de la page mémoire que le noyau nous a allouée. Nous injectons alors dans cette page (via l'écriture sur `/dev/kmem`) un code "tremplin" servant à exécuter n'importe quelle fonction du noyau depuis l'espace "utilisateur" (il s'agit alors d'un mécanisme de relais). Nous modifions alors l'adresse employée par l'appel-système 0 (qui référençait notre code dans la pile), par l'adresse de ce nouveau code.

À présent, l'appel-système 0 a changé de sémantique. Lors de son appel nous passons respectivement en paramètres : l'adresse de la primitive du noyau (ou du code arbitraire injecté en espace "noyau") que nous souhaitons exécuter en *ring 0*, puis les paramètres à passer à cette primitive, s'il y en a. Le code tremplin récupère alors les paramètres passés par l'appel-système 0, dans la pile "noyau" du processus appelant. Enfin, il appelle la fonction demandée avec les arguments adéquats.

Ce détournement mis en place, il nous est alors possible de déployer la logique de notre *rootkit* dans l'espace "utilisateur". Nous pouvons par exemple créer de nouveaux *threads* "noyau" (c'est-à-dire exécution en *ring 0*) exécutant le code de notre choix. Toutefois, avant d'utiliser les services du *rootkit* (fournis par le programme client), nous le dissimulons. C'est ce que nous

[20] La technique employée étant assez complexe, nous ne la détaillons pas.

allons voir dans ce qui suit.

2.3.4 Camouflage de la partie en espace "noyau" de la porte dérobée

Le camouflage des données et du code du *rootkit* est à considérer au cours de son fonctionnement, mais également lorsque le système est hors tension. Nous développons dans cet chapitre uniquement le premier cas. Tout d'abord nous exposons l'approche spécifique à Linux que nous avons conçue, et abordons ensuite brièvement la démarche originale employée par le *rootkit* Shadow Walker [Sparks et Butler, 2005], indépendante du système d'exploitation.

2.3.4.1 Détournement de `vmalloc()`

Cette section explique une de nos méthodes de camouflage en mémoire physique reposant sur le mécanisme de pagination de la MMU (Memory Management Unit) et sa mise en œuvre sous Linux.

À chaque processus est associée un PGD (*Page Global Directory*, qui référence les tables de pages du processus) dont l'adresse est chargée dans la MMU à chaque changement de contexte de processus. Ce mécanisme permet de cloisonner les processus entre eux. Chacun possède son propre espace d'adressage. Sur `x86`, l'intervalle de 3 Go à 4 Go de l'espace d'adressage correspond à l'espace "noyau", lequel n'est accessible qu'en *ring 0*. Les adresses virtuelles de cet espace correspondent aux mêmes adresses physiques quelque soit le processus.

Nous utilisons dans notre approche l'allocateur de mémoire non-contiguë `vmalloc()` pour allouer une page mémoire en espace "noyau". Elle est alors utilisée pour conserver le code malveillant. Son adresse linéaire se situe dans la zone de l'espace d'adressage réservée à cet allocateur. Dans cette zone, des pages successives de l'espace d'adressage linéaire (toutes de 4 Ko) correspondent à des pages physiques qui ne sont pas forcément contiguës. Il n'y a donc aucune contrainte quant à l'association entre une adresse linéaire et une adresse physique dans cette zone. Cela n'est pas le cas dans reste de l'espace d'adressage du noyau (pages de 4 Mo dont les adresses correspondent à celles des pages physiques à une constante près).

L'utilisation de l'allocateur `vmalloc()` provoque uniquement la modification des tables de pages primaires[21] lors d'une allocation de mémoire. Il s'agit d'un mécanisme qui est *paresseux* afin d'améliorer les performances du système. Ainsi, après une allocation de mémoire avec `vmalloc()` par un processus, le noyau ne met pas à jour les tables de pages de ce processus, mais seulement les primaires. Néanmoins, il renvoie au processus appelant l'adresse linéaire du début de la région de mémoire allouée. Ainsi, quand le processus accède pour la première fois aux adresses linéaires de cette région, une faute de page est levée et le gestionnaire correspondant est exécuté, lequel met à jour les tables de pages du processus en les synchronisant avec les tables de pages primaires.

Notre approche (figure 2.5) exploite le comportement paresseux de `vmalloc()`. Elle consiste à réserver deux pages de mémoire au travers de `vmalloc()` : une qui contient le code malveillant et une autre qui est vide. Nommons l'adresse linéaire de la page malveillante L_1 et son adresse physique P_1. De même, nommons respectivement L_2 et P_2 les adresses linéaire et physique de la page vide. Le processus complice (qui a alloué ces pages) cherche dans les tables de

[21] Ces tables de pages sont accessibles depuis la structure `init_mm`, il s'agit des tables de pages qui décrivent l'espace "noyau" (*cf.* figure 1.5 de la section 1.4.5).

pages primaires l'adresse physique P_1 qui correspond à l'adresse linéaire L_1. Il effectue cela de nouveau pour l'adresse L_2. Ensuite, il remplace dans les entrées des tables de pages primaires l'adresse P_1 par P_2. Il modifie enfin ses propres tables de pages avec l'association $L_1 \leftrightarrow P_1$. De cette façon, il peut accéder à la page malveillante alors que les autres processus du système ne le peuvent pas.

En effet, quand ces processus tentent d'accéder pour la première fois à la région de mémoire allouée par `vmalloc()` à l'adresse L_1, leurs tables de pages sont mises à jour avec l'association $L_1 \leftrightarrow P_2$, et ainsi ils accèdent à la page vide. Notons que l'adresse linéaire de la région allouée par `vmalloc()` peut être associée à n'importe quelle adresse physique sans restriction. Ainsi, quand un processus cherche la page physique que nous employons, il doit parcourir l'ensemble de la mémoire physique.

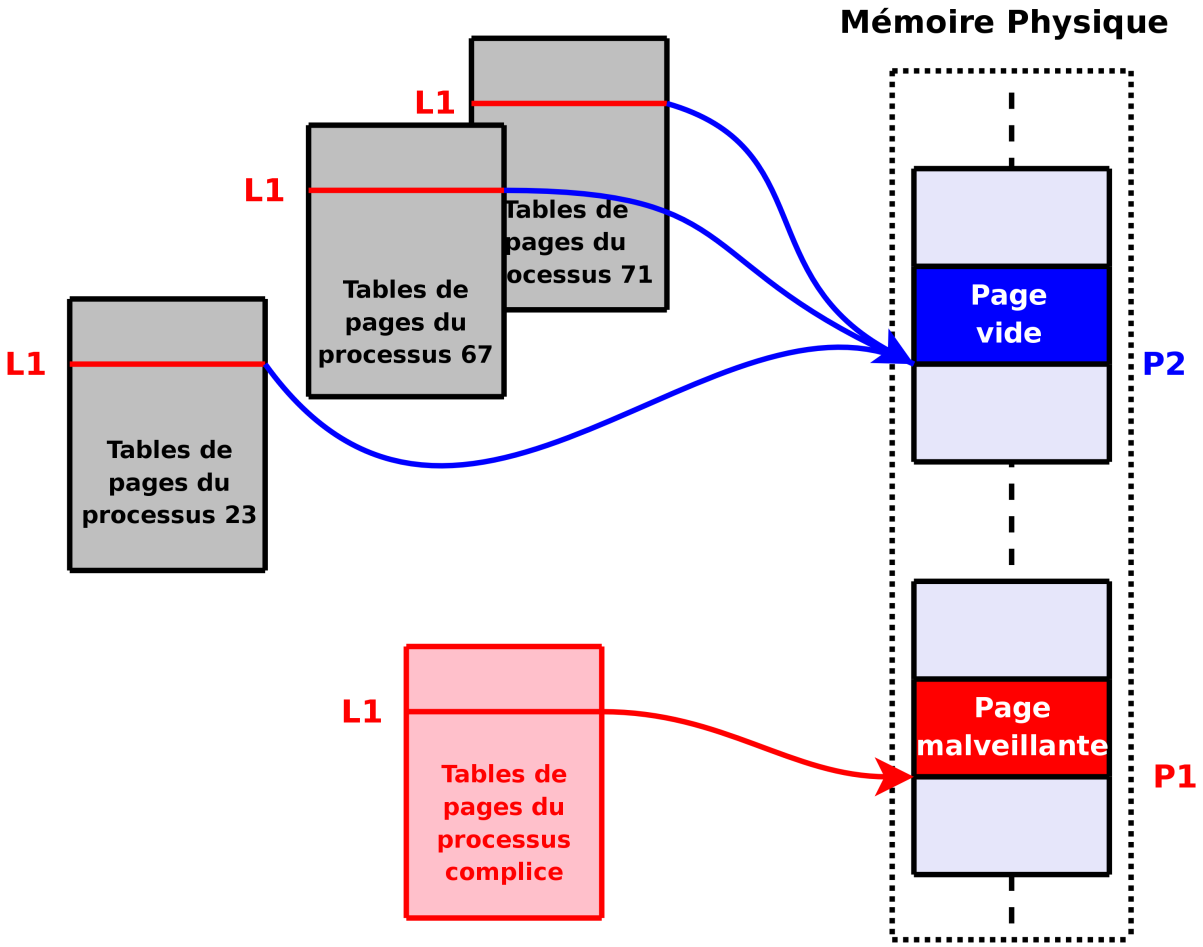

FIG. 2.5 – Exploitation du comportement paresseux de `vmalloc()`

2.3.4.2 Modification des attributs de pages

Nous citons ici l'exemple du *rootkit* Shadow Walker [Sparks et Butler, 2005] car il s'agit d'une alternative à notre approche, très pertinente (cependant elle dépend totalement d'une spécificité matérielle qui se trouve dans la majeure partie des processeurs de type `x86`). La technique employée profite de la division du TLB (*Translation Lookaside Buffer* : ITLB pour les instructions et DTLB pour les données) afin de cacher ses données à l'ensemble du système. Nous supposons ces données inscrites dans une page mémoire. Shadow Walker marque alors cette page non-présente (dans la table des pages correspondante) et l'entrée correspondante du TLB est vidée, entraînant ainsi une faute de page lors du premier accès. Le *rootkit* va alors vérifier s'il s'agit d'un accès en exécution ou d'un accès en lecture/écriture. Dans le cas où il s'agit d'un accès en exécution il va charger l'ITLB avec la page malveillante. Dans l'autre cas il peut à sa guise charger une page vide ou bien une autre page de son choix dans le DTLB. Ainsi, la lecture à l'adresse correspondant à la page malveillante entraîne la lecture d'une page banale, alors que l'exécution à partir de cette adresse déclenche le code malveillant.

2.3.5 Les actionneurs de la porte dérobée pour un processus

Il est important de remarquer qu'après avoir installé un *rootkit* "noyau", l'attaquant ne reste pas forcément connecté sur le système compromis. Il a ainsi besoin d'un moyen de communication pour rétablir sa connexion avec le *rootkit*. Nous présentons, au travers de notre méthode, comment un processus avec des privilèges d'utilisateur classique peut interagir avec l'espace "noyau" et tout particulièrement, comment il peut activer la porte dérobée. Un avantage de notre technique est que nous pouvons exécuter n'importe quelle primitive du noyau depuis un processus ordinaire de l'espace "utilisateur" (c'est-à-dire qui n'a pas les privilèges *root*).

Nous présentons dans ce qui suit les deux types d'actionneurs de portes dérobées que nous avons mis en œuvre. Ils reposent sur la création de *threads* "noyau". Nous supposons dans ces deux approches que l'attaquant se connecte au système compromis afin d'en récupérer le contrôle.

2.3.5.1 Premier type d'actionneur de portes dérobées

Dans cette première approche, l'actionneur est un *thread* "noyau" qui doit être partiellement caché. Par cela, nous entendons le délier uniquement de la liste qui regroupe l'ensemble des *threads* pour le cacher du système de fichier `/proc` ; et ainsi le cacher des utilitaires de contrôle de l'activité du système qui reposent sur `/proc` comme `ps` ou `top`. Notre *thread* n'est ici que partiellement caché, car il est également référencé dans une table de hachage, utilisée pour l'envoi des signaux entre processus via l'appel-système `sys_kill`, ou lorsqu'un processus est tracé par un autre (via l'appel-système `sys_ptrace`).

L'idée est que notre *thread* "noyau" installe un gestionnaire de signal pour que le processus de l'attaquant puisse s'authentifier auprès de lui[22]. Cette authentification se fait au travers de l'émission d'un signal particulier depuis le processus de l'attaquant vers notre *thread* "noyau".

[22] Pour que l'envoi d'un signal depuis le processus de l'attaquant non-root à notre *thread* "noyau" soit accepté, il est possible de modifier seulement l'UID du *thread* "noyau" lors de sa création et de le faire correspondre à celui de l'attaquant.

Ce dernier parcourt alors la liste des processus de l'espace "utilisateur" jusqu'à trouver celui qui a émis le signal[23].

Toutefois, il est aisé de mettre à mal le camouflage du *thread* "noyau", en lui envoyant un signal de terminaison non-blocable et non-interceptable (`SIGKILL`) (par exemple en balayant l'espace des identifiants de processus). Des techniques plus sophistiquées qui mettraient en œuvre plusieurs *threads* qui se recréent mutuellement empêcheraient cette défense active de fonctionner, en renforçant l'attribut de robustesse du *rootkit*.

2.3.5.2 Second type d'actionneur de portes dérobées

Dans cette alternative, nous délions de surcroît le *thread* "noyau" créé de la table de hachage et coupons par conséquent toute communication avec lui (les signaux ne peuvent plus fonctionner, pas plus que les IPC System V par exemple). Nous ne pouvons cependant pas le rendre totalement invisible. En effet, pour s'exécuter il doit toujours se trouver dans les listes de l'ordonnanceur. Nous limitons tout de même son séjour dans ces listes en l'endormant temporairement et périodiquement. Ces précautions d'invisibilité étant prises, aucune interaction depuis l'espace "utilisateur" n'est alors possible. Le *thread* doit alors passer en revue périodiquement l'ensemble des descripteurs de processus[24] à la recherche de celui correspondant à l'identifiant de l'attaquant (c'est-à-dire l'UID utilisé par l'attaquant). Le descripteur de processus trouvé, nous remplaçons l'adresse utilisée par l'appel-système 0, par l'adresse de notre code tremplin.

2.3.6 Partie en espace "utilisateur" de la porte dérobée

2.3.6.1 Préambule

La modification nécessaire au détournement de l'appel-système 0 (c'est-à-dire le changement d'une adresse), exposée dans la section 2.3.5, est effectuée dans le processus de l'attaquant sur le système compromis. La partie en espace "utilisateur" de la porte dérobée est employée quand l'attaquant agit à distance sur le système compromis sans s'y connecter avec un compte d'utilisateur. Nous présentons dans la suite de cette section deux procédés qui peuvent être mis en œuvre. Notons que pour ce scénario, l'actionneur de la porte dérobée consiste seulement en un moyen d'authentification (s'il existe) entre l'attaquant et la partie de la porte dérobée en espace "utilisateur" (le processus complice sur la machine compromise étant déjà détourné). Nous ne développons pas davantage ce sujet.

2.3.6.2 Premier procédé pour la porte dérobée en espace "utilisateur"

Dans ce procédé, la plupart de la logique du *rootkit* vis-à-vis des opérations malveillantes s'effectue au niveau du programme client, localisé sur la machine de l'attaquant. Les commandes (c'est-à-dire des appels-système 0) que l'attaquant souhaite exécuter sur sa machine sont relayées depuis son ordinateur au travers d'un mécanisme de relais [Caceres, 2002] qui est injecté dans un processus local pour lequel l'appel-système 0 a été détourné. Étant donné que

[23] Le processus émetteur du signal est connu du récepteur par son *Process IDentifier* (PID).

[24] Dans le reste du document nous employons le terme descripteur de processus pour nommer également un descripteur de *thread* car sous Linux il s'agit de la même structure.

seul les appels-système 0 doivent être relayés, ce mécanisme est vraiment commode. Toutefois, il est également possible d'adopter des approches de type *remote userland-execve* [Dralet et Gaspard, 2006] (qui permettent d'exécuter des programmes transférés depuis la machine de l'attaquant dans l'espace d'adressage d'un processus de la machine compromise, et cela sans écrire sur le disque), si des programmes qui ne sont pas présents sur la machine compromise doivent être exécutés (comme par exemple *nmap*).

2.3.6.3 Procédé alternatif pour la porte dérobée en espace "utilisateur"

Dans ce procédé, le mécanisme de relais d'appels-système est maintenant exécuté grâce à notre technique de parasitisme mobile sur les processus qui s'exécutent sur le système. Nous expliquons succinctement notre algorithme de parasitisme mobile dans la section 2.3.7.3. De même, la modification de l'adresse employée par l'appel-système 0 suit cette stratégie. Pour comprendre l'intérêt d'employer une technique de parasitisme mobile vis-à-vis de la dissimulation de l'activité d'un *rootkit*, nous renvoyons le lecteur à la section 2.3.7. Notons qu'à présent, l'actionneur de portes dérobées consiste en une communication spécifique entre l'attaquant et le processus infesté.

2.3.6.4 Épilogue

Une fois que la partie en espace "noyau" de la porte dérobée a été camouflée et que sa partie en espace "utilisateur" a été installée, il est alors possible pour l'attaquant de se reconnecter sur le système compromis et d'en garder le contrôle. Dans ce qui suit, nous développons les services que nous avons mis en œuvre dans notre *rootkit* afin qu'il dissimule l'activité de l'attaquant (*cf.* section 2.3.7).

2.3.7 Dissimulation de l'activité système

Dans cette section, nous proposons trois méthodes pour dissimuler l'activité système de l'attaquant, autrement dit le code qu'il exécute. Nous commençons par développer la méthode que nous avons mentionnée dans la section 2.3.5.1.

2.3.7.1 Cacher un processus

Pour dissimuler l'activité de l'attaquant, nous devons rendre invisibles aux administrateurs les tâches qu'il effectue. Ainsi il nous faut cacher les processus lancés par l'attaquant (s'il en exécute). Nous avons parlé dans la section 2.3.5.1 d'une méthode pour cacher un processus en l'effaçant de la liste doublement chaînée qui les relie et de la table de hachage, et en ne le laissant que de manière temporaire dans les listes de l'ordonnanceur.

Cette méthode est employée par notre démonstrateur afin de se cacher en mémoire. Cependant l'attaquant a besoin de lancer des programmes sur la machine et ces derniers doivent rester cachés sur le système.

Une méthode naïve est de supprimer les liaisons, après l'appel de la fonction `sys_fork` (créant un nouveau processus) par notre processus, et ensuite de charger le programme à exécuter via l'appel-système `sys_execve`. Cependant, entre le moment de la création du processus

et la suppression de ses liens au reste du système, un autre processus peut prendre la main et donc le repérer.

Pour pallier ce problème, nous dupliquons en mémoire le code de l'appel-système `sys_fork` et de la fonction `copy_process` (appelée par `sys_fork`) qui effectue les opérations de liaison du nouveau processus créé. Il suffit alors de supprimer ces opérations de liaison. Notre nouvelle fonction `sys_fork` modifiée peut être utilisée à la place de l'appel-système réel via le détournement de l'appel-système 0, comme expliqué précédemment. En procédant ainsi, le processus créé n'est plus lié après sa création.

Une alternative à cette dernière approche est de créer un autre processus chargé de la suppression des liens et dont la priorité est maximale. De ce fait, lorsque nous créons un processus avec notre démonstrateur via l'appel-système `sys_fork` le processus tiers prend la main juste après sa création et parcourt la liste des processus à la recherche du nouveau créé afin de le délier. Pour effectuer cette recherche, nous vérifions sa parenté avec le processus du démonstrateur. Nous utilisons le même procédé de recherche dans la section suivante afin de dissimuler la descendance d'un processus.

2.3.7.2 Cacher la descendance d'un processus

Les programmes exécutés par l'attaquant, qui ont été cachés, peuvent créer eux-mêmes des processus. Nous expliquons dans cette section une façon de cacher dynamiquement la descendance d'un processus quelconque.

L'idée est de créer un *thread* caché dont le rôle est de parcourir périodiquement la liste des processus du système, et pour chacun d'eux de remonter les liens de parentés afin de vérifier s'il s'agit d'un descendant du processus cible. Si tel est le cas le processus est alors caché, sinon nous passons au processus suivant. L'algorithme arrête de remonter les liens de parenté pour un processus donné à partir du moment où il tombe sur l'*idle task* de PID 0 qui est la première tâche créée (c'est-à-dire parente de tous les processus).

Afin d'améliorer la furtivité de cette tâche nous la mettons en sommeil en l'enlevant temporairement des listes de l'ordonnanceur. Nous évitons ainsi de monopoliser le processeur et donc d'alerter l'administrateur.

2.3.7.3 Parasitisme mobile de *threads* "noyau"

Les solutions d'exécution furtive proposées jusqu'alors posent un problème de détection du fait des liaisons multiples qui existent entre les structures constituant le descripteur d'un processus. C'est de cette constatation que nous est venue l'idée de nous débarrasser de ce descripteur pourtant nécessaire à l'exécution d'un programme. Ainsi, nous avons élaboré un algorithme de parasitisme *mobile* de processus ou de *threads* "noyau" s'exécutant sur le système compromis[25]. Dans la suite, nous expliquons le principe de l'algorithme sans entrer dans les détails.

Notre méthode consiste à voler des cycles d'exécution à deux processus. Le principe est le suivant :

1. nous insérons préalablement du code dans la mémoire du noyau ;

[25] L'implémentation de l'algorithme est plus facile dans le cas des *threads* "noyau" que dans celui des processus. En effet, l'espace d'adressage du noyau est commun pour tous ses *threads*.

2. nous déroutons l'exécution d'un *thread* afin qu'il exécute ce code, pour que
3. finalement le code boucle sur lui-même en faisant des va-et-vient sur les processus parasités[26].

Cet algorithme a été conçu pour rester le plus discret possible. Ainsi, nous ne souhaitons pas altérer le travail initial des processus parasités. Dans ce qui suit nous expliquons très brièvement comment cela est accompli pour le cas de deux processus.

Notre code est composé de deux blocs, chacun associé à un des processus parasités (représentés par leurs descripteurs sur la figure 2.6). Ces blocs sont similaires et se divisent en trois parties : le prologue (qui restaure l'état initial du processus qui exécute l'autre bloc), le code malveillant et l'épilogue (qui initie l'exécution de l'autre bloc au sein du processus qui lui est associé).

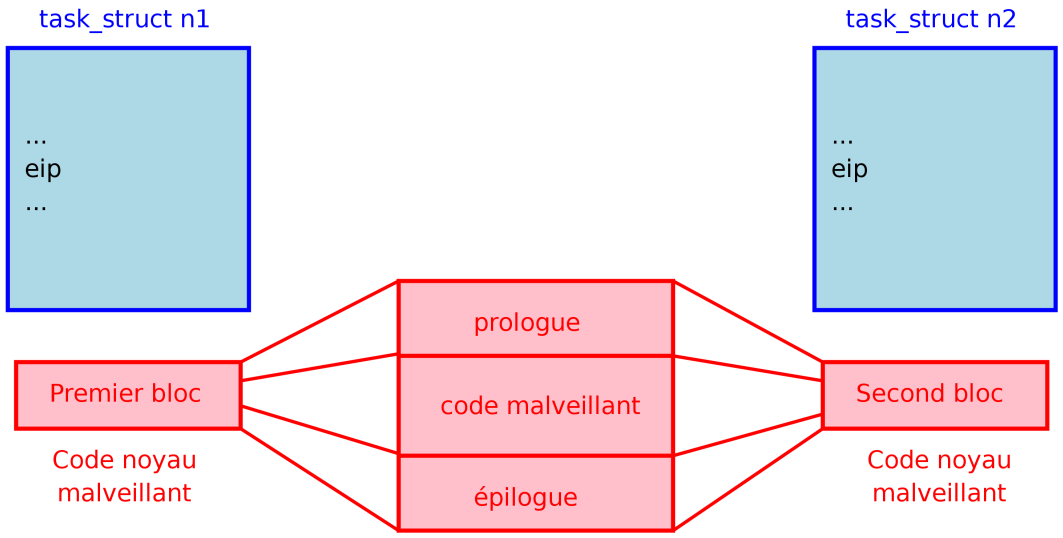

FIG. 2.6 – Vol de cycles d'exécution 1/2

Nous montrons succinctement son fonctionnement sur la figure 2.7. Le premier bloc s'exécute sur le processus 1, passe ensuite la main au processus 2 qui exécute le second bloc. Ce dernier repasse la main au processus 1 pour l'exécution du premier bloc et ainsi de suite. Ainsi, nous obtenons un parasitisme mobile transitant d'un processus à l'autre. Ces derniers sont parasités temporairement afin qu'ils puissent continuer d'effectuer le travail pour lequel ils ont été créés, limitant ainsi la détection de l'activité malveillante.

[26] Le compteur d'instructions d'un processus en attente d'exécution est sauvegardé dans son descripteur. Par conséquent il est possible de le remplacer par l'adresse de notre code parasite.

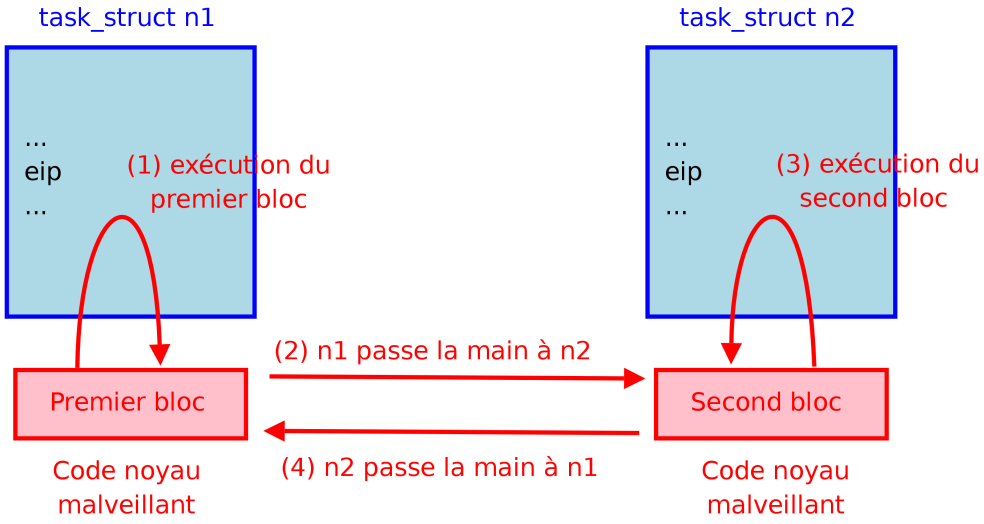

FIG. 2.7 – Vol de cycles d'exécution 2/2

2.4 Synthèse de l'expérimentation

Nous commençons par rapprocher les actions malveillantes effectuées par notre *rootkit* avec la classification établie dans le chapitre 1 en section 1.5. Nous discutons ensuite brièvement de l'efficacité de notre approche vis-à-vis des attributs des *rootkits* (*cf.* section 2.2.4). Enfin, nous faisons état des contributions et des limites de notre approche par rapport aux méthodes employées par les *rootkits* "noyau" existants.

2.4.1 Les actions malveillantes effectuées par notre *rootkit*

Les actions malveillantes de notre *rootkit* font toutes parties de la classe 1 (altération de la mémoire du noyau) dans la classification que nous avons établie dans le chapitre 1 en section 1.5.

Plus précisément, au sujet de l'installation de la partie en espace "noyau" de notre porte dérobée : l'action de modification de l'adresse employée par l'appel-système 0 (*cf.* section 2.3.3) fait partie de la classe 1.2.1 (altération des variables d'état de l'exécution) ; l'injection du code de *bootstrap* dans la pile "noyau" du processus complice fait partie de la classe 1.1.3 (injection de code malveillant atteignable dans une région de données du noyau), tout comme l'injection dans une page de données du noyau, du mécanisme de relais (le code "tremplin") d'appels de primitives du noyau.

L'injection de codes additionnels dans la zone VMALLOC (*cf.* section 2.3.7.1) relève également de la classe 1.1.3. Ces codes additionnels (les services du *rootkit*) effectuent principalement, pour ceux présentés dans la section 2.3.7.1, des actions de la classe 1.2.2 (altération des variables auxiliaires). En effet, ces services dissimulent des processus en supprimant leurs liens vers les différents sous-systèmes du noyau, sauf ceux qui les relient à l'ordonnanceur.

Concernant nos actionneurs de portes dérobées pour un processus (*cf.* section 2.3.5), ils

font appels à des actions de la classe 1.2.1 (lors de la modification de l'adresse employée par l'appel-système 0) et 1.2.2 (pour les mêmes raisons que celles mentionnées pour les services de dissimulation de processus). Notons que pour la partie en espace "utilisateur" de la porte dérobée (*cf.* section 2.3.6), il n'y a pas d'actions malveillantes qui relèvent de notre classification. En effet, il ne s'agit que de mécanismes de relais vers notre *rootkit* "noyau" qui, lui, effectue les actions malveillantes.

Au sujet du camouflage de données au travers des modifications des tables de pages de la zone VMALLOC (*cf.* section 2.3.4.1), notre approche emploie des actions qui relèvent de la classe 1.2.2. Enfin, notre approche de parasitisme mobile de *threads* "noyau" (*cf.* section 2.3.7.3) s'appuie sur des actions de la classe 1.2.1 pour ce qui est du détournement de l'exécution des *threads*, et sur des actions de la classe 1.1.1 (ajout d'une région de code "noyau" malveillant atteignable) ou 1.1.3 pour l'injection du code malveillant à exécuter.

2.4.2 Quelques considérations sur l'efficacité de notre *rootkit*

Afin que notre *rootkit* soit compatible au maximum avec les différentes version du noyau, il est préférable de dépendre de ses points stables. Ils correspondent aux portions de code qui ont été éprouvées et ne sont que très rarement modifiées.

Par ailleurs, les points critiques du noyau vont également nous intéresser. Ils correspondent aux portions de code (c'est-à-dire les éléments essentiels) qui ont un impact important sur l'efficacité globale du système. Ainsi, ces points sont implémentés avec une grande attention et ne sont que rarement modifiés par la suite. La plupart sont alors également des points stables et cela peut s'avérer profitable à notre *rootkit*.

Pour favoriser la dissimulation de notre *rootkit* ou bien de ses activités (c'est-à-dire favoriser l'attribut d'invisibilité), il est pertinent d'agir au niveau de l'environnement des points critiques du noyau (c'est-à-dire les données qu'ils manipulent ou qu'ils utilisent telles quelles), sans avoir à les modifier. En effet, comme nous venons de l'expliquer, ajouter du code en ces points ou les changer peut être catastrophique pour les performances du système. Ainsi, la défense est confronté à un choix douloureux. Si elle veut mettre en œuvre des mécanismes de détection ou de prévention comme du filtrage de données en ces points, elle risque de rendre le système inutilisable[27]. Illustrons cela par notre approche originale d'interaction avec le noyau du système compromis. Nous détournons un point critique du noyau (l'appel-système 0) sans pour autant le modifier (*cf.* section 2.3.1). Nous n'influons que sur les données qu'il utilise.

En passant par les points critiques du noyau, nous y rendons difficile la réalisation de mécanismes de protection et ainsi favorisons la *robustesse* du *rootkit*. En outre, ce dernier a tout intérêt à profiter de ces points dans lesquels ses actions ne vont pas compromettre significativement son *invisibilité*.

2.4.3 Contributions

Dans cette section nous reprenons point par point les apports de notre travail en les comparant avec les approches existantes.

[27] L'impact sur le système dépend tout de même du point critique modifié. Mais c'est la modification de plusieurs de ces points qui peuvent rendre le système inutilisable.

2.4.3.1 Partie en espace "noyau" de la porte dérobée

Dans notre méthode, les étapes nécessaires à l'installation du *rootkit* se limitent à un accès en lecture et écriture sur le périphérique `/dev/kmem` pour les opérations préliminaires de mise en place d'un *bootstrap*. Par la suite, tout le reste de l'installation s'effectue au travers de l'appel-système 0, les futures injections dans l'espace "noyau" comprises. Ainsi, l'activité de l'attaquant est masquée dès l'installation du *rootkit*.

Détaillons les différences entre cette approche et les techniques usuellement employées dans les autres *rootkits* :

– *Visibilité locale des modifications :*
Notre approche ne modifie le comportement du système que pour le processus à partir duquel nous agissons. Le fonctionnement de l'appel-système 0 est inchangé pour tous les autres processus du système. Ainsi, nous parlons de détournement *local* à comparer aux méthodes de *hooking* et de *hijacking* employées par les *rootkits* qui affectent de manière globale le système.
Rappelons que la technique de *per-process syscall hooking* [Pluf, 2006] n'est pas comparable à notre méthode (*cf.* section 2.3.1), car elle ne permet pas l'exécution de code arbitraire en ring 0.

– *Corruption des données et non du code :*
De nombreux *rootkits* altèrent le code du noyau. Notre approche agit uniquement sur l'environnement de ce code, c'est-à-dire sur les données qu'il manipule. De plus, nous ne modifions qu'une seule variable : un pointeur de fonction dans le descripteur d'un *thread*. Ensuite quelques autres portions de codes sont ajoutées au noyau dans des emplacements que nous allouons via les mécanismes du noyau.

– *Exécution de code arbitraire en ring 0 :*
Le détournement de l'appel-système 0 permet l'exécution du code de notre choix en ring 0. Mais c'est ensuite grâce à notre tremplin (*cf.* section 2.3.3) que nous pouvons exécuter n'importe quelle primitive du noyau ou du code que nous avons injecté.

– *Utilisation des mécanismes fournis par le noyau du système compromis :*
Les approches courantes à notre connaissance (lors de la création de notre *rootkit*) ne mettent pas en place un tremplin permettant de profiter de tous les mécanismes du noyau. En effet, la majorité des opérations malveillantes d'un *rootkit* sont écrites par l'attaquant entièrement, alors que le noyau fournit bon nombre de services qui lui faciliterait sa tâche. L'utilisation des services du noyau dans notre *rootkit* nous permet de déployer la majeure partie de sa logique dans un programme client situé sur une machine distante. Ainsi, l'apport de notre approche par rapport aux techniques d'attaque "tout en mémoire" actuelles [Grugq, 2004; Pluf et Ripe, 2005; Dralet et Gaspard, 2006] est que la mémoire du système compromis ne contient à aucun moment des parties complètes de notre *rootkit*. Seules les actions ne pouvant pas être effectuées via les services du noyau sont codées pour être exécutées directement dans la mémoire du système compromis.
Nous entravons ainsi les mécanismes d'analyse *forensics online* en ne leur laissant que

des éléments incomplets pour comprendre ou reconstruire les activités malveillantes entreprises par l'attaquant[28].

2.4.3.2 Dissimulation de la porte dérobée "noyau"

Afin de camoufler le code et les données de notre *rootkit*, nous avons proposé une méthode profitant des caractéristiques de l'allocateur de mémoire non-contiguë du noyau Linux (*cf.* section 2.3.4.1).

– *Utilisation des mécanismes du noyau :*
Pour dissimuler un *rootkit*, nous ne créons pas de nouveaux mécanismes, nous exploitons les caractéristiques de l'allocateur `vmalloc()`. Au travers de son utilisation, notre *rootkit* peut être camouflé. Encore une fois, nous essayons de profiter au maximum des fonctionnalités standard fournies par le noyau du système, comme le ferait n'importe quel sous-système.

– *Efficacité :*
La méthode de Shadow Walker [Sparks et Butler, 2005] (*cf.* section 2.3.4.2) impose d'utiliser directement le matériel pour dissimuler le *rootkit*. Ainsi, en agissant à un niveau plus bas, il est techniquement plus fiable que notre technique. Le prix à payer cependant est la complexité de l'implémentation, ainsi qu'une dépendance forte vis-à-vis de l'architecture matérielle, au contraire de notre approche.

2.4.3.3 Partie en espace "utilisateur" de la porte dérobée

Ces avantages ne sont pertinents que dans le cas où un attaquant opère à distance sans se connecter au système avec un vrai compte d'utilisateur. Nous avons proposé deux approches qui emploient toutes les deux un mécanisme par procuration (*cf.* section 2.3.6).

– *Utilisation d'un mécanisme de relais :*
Nos porte dérobées se servent d'un mécanisme d'exécution d'appels-système par procuration. Ainsi, elles sont déclenchées depuis l'espace "utilisateur". Cela semble moins intéressant que les approches qui interagissent avec l'attaquant directement depuis le noyau. Toutefois, de cette façon, nous sommes moins dépendants des spécificités internes au noyau (qui évoluent régulièrement de version en version).

– *Procédé alternatif :*
Le procédé alternatif envisagé pour la partie en espace "utilisateur" de la porte dérobée inclut une nouveauté vis-à-vis des approches de parasitisme fixe. En effet, contrairement aux approches qui parasitent un processus, la notre se déplace d'un processus à un autre[29]. L'exécution de la porte dérobée n'altère que temporairement le travail des processus parasités. En cela, nous pouvons dire que nous améliorons l'*invisibilité* face aux approches de parasitisme fixe. Cependant nous touchons plusieurs processus à la place d'un seul.

[28] Nous ne relayons que des appels-système 0, entravant de surcroît la reconstruction de l'attaque.
[29] Actuellement, l'ensemble des processus infectés doit être établi avant l'exécution de l'algorithme.

Ainsi, suivant le type de détection mis en place sur le système compromis, l'efficacité du camouflage de notre porte dérobée peut varier.

2.4.3.4 Services du *rootkit*

Parmi les services que doit fournir un *rootkit*, nous focalisons notre attention sur ceux qui assurent la dissimulation des activités de l'attaquant.

– *Camouflage a posteriori :*
La plupart des *rootkits* cachent les processus qu'ils créent *a posteriori* (c'est-à-dire une fois qu'ils sont lancés) en supprimant leurs liens avec le système. Nous avons présenté une démarche inverse (*cf.* section 2.3.7.1), en empêchant l'ajout de ces liens lors de la création même du processus (duplication et "simplification" de `sys_fork`). De cette manière, nos processus n'ont que peu d'interaction avec les structures internes du noyau (uniquement avec l'ordonnanceur).

– *Camouflage dynamique :*
Nous proposons une technique pour camoufler un processus ainsi que toute sa descendance au fur et à mesure de sa constitution (*cf.* section 2.3.7.2). Les approches qui camouflent systématiquement tous les processus ayant un UID spécifique parviennent à un résultat proche.

– *Parasitisme mobile :*
Notre algorithme transite sur tout ou partie des processus ou des *threads* "noyau" du système (*cf.* section 2.3.7.3 pour une brève présentation du principe). Les techniques de parasitisme courantes se contentent généralement d'une seule cible à infecter.
L'apport majeur de notre technique découle de l'objectif que nous nous sommes fixés lors de sa conception : le travail des processus parasités ne doit être altéré que temporairement. Ainsi, nous favorisons la furtivité de l'exécution malveillante en rendant temporaire la corruption des processus.

2.4.4 Limites de notre *rootkit*

2.4.4.1 Partie en espace "noyau" de la porte dérobée

La principale limite de cette partie de la porte dérobée est que si le défenseur parvient à nous tracer, il peut constater que nous appelons l'appel-système 0 depuis l'espace "utilisateur". Ce type de comportement étant *a priori* suspect, la défense aura de quoi se poser des questions quant à la compromission de son système.

Un autre problème se situe au niveau de la compatibilité interne des différentes versions du noyau. En effet, les primitives que nous exécutons au travers de l'appel-système 0 sont susceptibles d'être modifiées d'une version à l'autre du noyau. Bien que ces changements soient peu courants pour des primitives critiques ou stables, ils restent plus probables que des modifications de l'API de la *libc* ou de l'ABI (*Application Binary Interface*) du noyau. Ainsi, les attaques de

type *remote userland execve* [Dralet et Gaspard, 2006] possèdent cet atout indéniable qu'est la compatibilité garantie quelque soit la version du noyau du système compromis.

2.4.4.2 Dissimulation de la porte dérobée "noyau"

Nous présentons ici les limites de notre technique de dissimulation du code d'un *rootkit*.

– *Lecture complète de la mémoire physique :*
Notre méthode fondée sur l'allocateur de mémoire non-contiguë du noyau Linux ne résiste pas à une lecture complète de la mémoire physique. Cependant, une action de ce type prend beaucoup de temps tout en monopolisant grandement le processeur. C'est pourquoi une solution de détection de ce type n'est en général pas satisfaisante[30].

– *Compromis entre dissimulation et sûreté :*
Ce qui peut également trahir notre dissimulation est le descripteur de zone que crée `vmalloc()`. Nous pouvons bien entendu le supprimer, mais alors nous n'avons plus l'assurance que notre code ne soit pas écrasé lors de l'insertion d'un module "noyau" par exemple.

2.4.4.3 Actionneur de la porte dérobée

Les deux actionneurs de portes dérobées (*cf.* section 2.3.5) que nous avons proposés se servent d'un *thread* "noyau" qui est dissimulé (par la suppression de la majorité de ses liens avec le système compromis). Ces actionneurs ne sont pertinents que dans le cas où un attaquant se connecte au système avec un vrai compte d'utilisateur. Dans le cas d'un attaquant qui ne se reconnecte pas au système avec un vrai compte, ces actionneurs ne fonctionnent pas.

2.4.4.4 Partie en espace "utilisateur" de la porte dérobée

Ces limitations ne sont pertinentes que dans le cas où un attaquant opère à distance sans se connecter au système avec un vrai compte d'utilisateur.

– *Utilisation d'un mécanisme de relais :*
Comme la porte dérobée dépend d'un processus de l'espace "utilisateur", elle dépend également de sa capacité à survivre dans le système.
– *Procédé alternatif :*
Notre procédé alternatif souffre des mêmes limites que notre technique de parasitisme mobile. Nous les abordons dans la suite de cette section.

2.4.4.5 Services du *rootkit*

Nous donnons ici les limites de nos techniques de dissimulation des activités de l'attaquant.

[30] La discrétion est aussi l'apanage de la défense car un *rootkit* peut détecter les activités déclenchées à son encontre et ainsi réagir en conséquence.

Le camouflage d'un processus est faillible, dès lors que son descripteur reste présent et visible en mémoire. En effet, les relations multiples entre les structures `task_struct` et `thread_info` constituant en partie le descripteur, sont exploitables pour les démasquer. Ainsi, parcourir la mémoire physique à la recherche de processus cachés de la sorte est tout à fait réalisable[31].

Le constat de ces limites nous a poussé à développer notre parasitisme mobile. Cette approche ne reste cependant pas sans défaut.

– *Robustesse limitée :*
  Au travers de notre parasitisme mobile, lorsque le code malveillant s'exécute à un moment donné sur un certain processus, sa survie (c'est-à-dire le fait qu'il puisse continuer à s'exécuter ou à se déplacer vers un autre processus) dépend de celle du processus parasité. Ainsi, si le processus meurt, notre code malveillant meurt avec lui.

– *Difficultés d'implémentation de la charge malveillante :*
  La structure de la charge malveillante doit être pensée spécifiquement pour fonctionner avec notre algorithme. Ainsi, par exemple, pour le cas de deux processus, le code malveillant doit être découpé en deux blocs indépendants.

– *Risque de détection :*
  La corruption d'un nombre important de processus est une limite en soit. En effet, plus ce nombre est grand, plus le camouflage du code malveillant est faillible. Certains des processus parasités pourraient être des leurres mis en place par la défense afin de détecter un parasitisme mobile.

2.5 Conclusion

Nous avons mis en évidence dans ce chapitre : d'une part les principes nous permettant de modéliser les *rootkits* et d'autre part l'approche du *rootkit* "invisible" au travers du détournement malveillant de sous-systèmes du noyau. Nous avons également développé une réflexion sur l'un des objectifs primordiaux des *rootkits*, à savoir dissimuler l'activité de l'attaquant. Afin de mieux évaluer ces parasites informatiques, et ainsi concevoir des solutions visant à s'en prémunir, nous avons tenté de les caractériser. Tout d'abord nous avons proposé une architecture des *rootkits*. La problématique de la communication entre l'attaquant et ce type de parasite a ensuite était exposée. Enfin, nous avons dégagé trois attributs essentiels qui les qualifient.

L'étude expérimentale que nous avons menée dans ce chapitre sur ces parasites informatiques (qui procurent à l'attaquant un outil de "travail" pertinent), nous a fourni des éléments de réponse quant à la façon dont le *problème de l'attaquant* (*cf.* section 1.2) peut être appréhendé. Nous pensons que si les activités menées par l'attaquant débute par une succession d'actions dont la malveillance ne peut être déduite[32], alors la difficulté de la détection de ces activités

[31] Nous avons d'ailleurs mis en œuvre cela dans notre démonstrateur. Les faux positifs, dus aux descripteurs de processus morts, toujours présents en mémoire, sont supprimés après une étape vérifiant si les descripteurs sont figés ou non.

[32] Les actions menées semblant tout à fait légitime.

frauduleuses croît en fonction du temps pendant lequel il opère sous couvert de la normalité. En effet, l'attaquant prend alors le temps de préparer son environnement afin que son activité frauduleuse se passe dans les meilleures conditions (le plus rapidement possible, en laissant peu d'indices d'intrusion sur la machine, etc.). Nous pensons donc que la solution que doit choisir l'attaquant dans la réponse à son problème doit emprunter le chemin par lequel le moins d'actions malveillantes doivent être accomplies, et dont la première à l'être ne l'est que le plus tard possible.

Après avoir étudié un certain nombre d'actions malveillantes ciblant un noyau de système d'exploitation au travers de l'analyse et de la conception de *rootkits*, nous examinons dans le prochain chapitre les différentes approches existantes pour protéger ces noyaux contre ces actions.

Chapitre 3

État de l'art sur la protection du noyau vis-à-vis des actions malveillantes

En fonction de l'architecture des noyaux, les mécanismes de protection peuvent être différents. L'objectif poursuivi dans nos travaux est d'assurer la protection des systèmes d'exploitation de type COTS (*Components Off-The-Shelf*[1]) qui sont pour la plus grande majorité fondés sur des noyaux monolithiques. C'est pourquoi, nous nous limitons dans cet état de l'art aux protections applicables aux noyaux monolithiques.

Ce chapitre est structuré suivant la classification établie dans le chapitre 1 en section 1.5. Les mécanismes de protection examinés recouvrent les différents moyens de la sûreté de fonctionnement. Ils sont plus communément appelés dans le cadre de la sécurité opérationnelle : moyens de prévention, et moyens de détection et recouvrement[2]. Pour ce qui est de la validation de la sécurité il s'agit des moyens appliqués lors de la phase de développement. Il est alors question, dans le cadre de la sécurité, d'éliminer les vulnérabilités. Nous présentons brièvement ces derniers moyens par souci d'exhaustivité, car notre travail se concentre exclusivement sur la sécurité en vie opérationnelle du système. Enfin, il faut distinguer parmi les approches existantes lorsque cela est pertinent, celles qui ont été prévues pour traiter les actions incorrectes *externes* au noyau (qui proviennent de l'espace "utilisateur", de périphériques ou encore de la routine SMI), de celles, beaucoup moins nombreuses, qui traitent des actions *internes* au noyau (ces solutions se fondent alors généralement sur la virtualisation, *cf.* section 4.1.1 pour une introduction sur la virtualisation).

3.1 À propos de l'élimination de vulnérabilités

L'élimination de vulnérabilités en phase de développement consiste notamment en de la vérification statique et de la vérification dynamique [Ernst, 2003]. Dans le cadre de la sécurité elle est effectuée pour s'assurer de l'absence de vulnérabilités dans un système.

[1] En français, "composants sur étagère".

[2] La détection d'erreur et le rétablissement du système sont les moyens mis en œuvre pour la tolérance aux fautes.

3.1.1 Vérification statique

La vérification statique est conduite sans exécution du système. Il s'agit alors le plus souvent d'analyse statique effectuée sur une description du système (logiciel dans notre cas).

La plupart des analyseurs statiques ne sont pas *corrects* et ne font que signaler des bogues potentiels à l'utilisateur. Il n'est pas possible d'en conclure que le programme analysé est sûr. En pratique, des états atteignables sont oubliés au cours de l'analyse, souvent pour améliorer les performances en temps ou en mémoire de l'analyseur. Les méthodes employées sont notamment : la reconnaissance syntaxique de motifs [GNU grep], avec heuristiques [Wheeler, 2007], et plus généralement la propagation de propriétés (interactions de pointeurs, numériques, etc.) mais sans assurance de correction [Coverity].

D'autres analyseurs statiques reposent sur des méthodes formelles *correctes*. Citons par exemple : le *model checking* [Clarke *et al.*, 1986; Clarke, 1997] dont le principe de base est d'explorer l'ensemble des états atteignables du système, le *theorem proving* [Filliatre, 2003], qui traduit la sémantique du programme et les propriétés à vérifier en formules logiques puis tente de les prouver à l'aide d'un assistant de preuve, ou encore l'interprétation abstraite [Cousot et Cousot, 1977] qui permet de sur-approximer l'ensemble de tous les comportements possibles d'un programme et par là-même de montrer formellement l'absence d'erreurs dans un logiciel.

Vis-à-vis de la sécurité, des travaux ont été menés pour appliquer l'analyse statique par interprétation abstraite sur la détection dans les logiciels de débordement de tampon [Allamigeon et Hymans, 2007], ou encore de débordement dans le tas (*heap overflow*) [Allamigeon et Hymans, 2008]. Notons enfin l'application de cette théorie pour la vérification automatique de propriétés de sûreté sur les circuits matériels de type FPGA (*Field-Programmable Gate Array*) ou ASIC (*Application Specific Integrated Circuit*) [Hymans, 2002]. L'analyse statique s'applique alors à la description comportementale de ces circuits, écrite dans le langage VHDL.

3.1.2 Vérification dynamique

La vérification dynamique est basée sur des exécutions du système. Les moyens mis en œuvre sont alors notamment l'injection de fautes ou l'exécution symbolique.

Parmi les méthodes d'injection de fautes, on retrouve notamment le *fuzzing*. L'idée de cette approche est d'injecter des données aléatoires ou hors domaine, dans les entrées d'un programme, afin de le faire défaillir et ainsi repérer des vulnérabilités. Différentes approches sont alors possibles, dont notamment : le *fuzzing* classique où la structure du logiciel n'est pas connue (approche de type *black box*) [Oehlert, 2005; Thompson *et al.*, 2002], le *fuzzing* intelligent où une analyse statique préliminaire est menée sur le logiciel, afin d'adapter la génération de données [Lanzi *et al.*, 2007] (approche de type *grey box*), ou encore des approches de *fuzzing* où une connaissance du système est nécessaire (approche de type *white box*) [Godefroid *et al.*, 2008]. Notons que ces méthodes ont été appliquées afin de trouver des vulnérabilités au sein de pilotes de périphérique (modules "noyau") [Butti et Tinnes, 2007]. Elles ont également été appliquées pour tester la robustesse de noyau de système d'exploitation où l'injection de fautes est effectuée à différents endroits comme au niveau de l'interface entre le noyau et les applications [Kalakech *et al.*, 2004] ou encore entre le noyau et ces pilotes de périphériques [Albinet *et al.*, 2004]. D'autres techniques d'injection de fautes ont également été appliquées au test de la robustesse de noyau, comme la mutation de code d'applicatif en cours d'exécution.

En ce qui concerne l'exécution symbolique [King, 1976], l'idée est d'évaluer les instructions d'un programme avec des données d'entrée symboliques, le long d'un chemin sélectionné dans le graphe de flot de contrôle. Ce procédé entraîne le calcul de conditions de chemin qui tendent à être simplifiées ou résolues dans le but de trouver des données de test qui sensibilisent le chemin sélectionné ou de démontrer la non-exécutabilité de ce chemin. Cette technique a par exemple été appliquée à la détection de débordement d'entiers sur des binaires x86 [Wang *et al.*, 2009].

3.2 Protection contre les actions de la classe 1

3.2.1 Contrôle des vecteurs d'accès à la mémoire du noyau

Nous avons déterminé deux vecteurs d'accès à la mémoire pour les actions malveillantes dans la section 1.5.1. 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 d'autres actions malveillantes. Aussi remarquons que le contrôle d'accès à la mémoire du noyau n'a de sens que pour les actions externes au noyau.

3.2.1.1 Contrôle des vecteurs d'accès de type CPU

Comme expliqué au chapitre 1 en section 1.5.1, les fonctionnalités du 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" [Lacombe *et al.*, 2008]. Ces fonctionnalités doivent évidemment être contrôlées. Par exemple, les périphériques `/dev/kmem` et `/dev/mem` peuvent être désactivés (comme le fait par exemple grsecurity [Spengler *et al.*, 2009]) ou filtrés afin d'autoriser uniquement les accès aux entrées/sorties projetées en mémoire (comme le font les noyaux Linux actuels s'ils sont correctement configurés). Aussi, afin de détecter les modules "noyau" malveillants, une solution est d'établir une vérification des modules avant chargement via l'utilisation de signatures cryptographiques [Microsoft Corporation, 2006] (classe 1.1.1). Cependant, de cette façon nous n'empêchons pas l'exploitation de bogues 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 du 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 bogués) y étant ajoutés est important. En fait, la grande majorité des failles du noyau proviennent des pilotes de périphériques (*cf.* chapitre d'introduction, annotation 2). La solution de sécurité *PaX* [Spengler *et al.*, 2003], 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.

3.2.1.2 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'entrée/sortie, et cela requiert la modification des pilotes de périphérique 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 entrées/sorties brutes ainsi que du périphérique virtuel `/dev/port` (comme le fait par exemple grsecurity [Spengler *et al.*, 2009]) empêche la mise en place de transferts DMA à partir de l'espace utilisateur.

Finalement, une meilleure approche nécessite un système informatique qui inclut une IOMMU (*Input/Output Memory Management Unit*, implémentée par exemple par la technologie Intel VT-d, *cf.* section 4.1.1.3). Avec cette unité matérielle, il est alors possible de protéger la mémoire principale contre les périphériques malveillants [Rutkowska, 2007]. Une IOMMU est une unité de gestion mémoire (MMU) qui connecte un bus d'entrée/sortie (supportant le DMA) à la mémoire principale. Tout comme une MMU traditionnelle, la IOMMU s'occupe de la correspondance entre les adresses d'entrée/sortie 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.

Le projet SecVisor [Seshadri et Qu, 2007] emploie une IOMMU (limitée au niveau de ces fonctionnalités) au travers de la technologie *Device Exclusion Vector* (DEV) des extensions matérielles SVM de AMD [AMD, 2005].

3.2.2 Mécanismes de prévention

Avant d'aborder les différentes solutions de prévention existante, remarquons que les techniques de randomisation de l'agencement de l'espace d'adressage (ASLR — *Address Space Layout Randomization*), telles que proposées par *PaX* [Spengler *et al.*, 2003], ne sont pas efficaces pour protéger le noyau d'actions malveillantes. Non seulement l'ASLR doit être effectuée sur une architecture 64 bits [Shacham *et al.*, 2004] afin d'obtenir une protection efficace, mais elle n'est de plus envisageable que pour l'espace utilisateur. En effet, des structures du 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. Dès lors l'installation d'une *callgate* devient possible (*cf.* chapitre 2 section 2.1.3 et les travaux menés dans [Crazylord, 2002; Kasslin, 2006]) et l'attaquant peut alors prendre le contrôle de la machine.

Nous présentons dans cette section tout d'abord les mécanismes actuels de prévention relatifs aux actions malveillantes externes au noyau, pour ensuite examiner ceux qui traitent également les actions internes.

3.2.2.1 Vis-à-vis des actions externes au noyau

Au sujet de la classe 1.1 En ce qui concerne la classe 1.1.2, les régions de code peuvent être rendues exécutables et non modifiables. De même, afin de se protéger de la classe 1.1.3 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. Un *patch* du noyau Linux (proposé par Siarhei Liakh [Liakh, 2009]) répond justement à cette problématique, et place dans des pages séparées le code et les données du noyau[3].

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 deviennent 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* de 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 le chapitre 4, 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 en partie de la classe 1.1.4, les solutions génériques de protection contre l'exploitation de débordements de tampon 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. Ces approches couvrent alors les actions de la classe 1.1.4 qui requièrent au préalable une action de la classe 1.2.1 (car elles agissent fondamentalement à ce dernier niveau). Elles sont toutefois inefficace si le noyau souhaite exécuter sciemment du code dans une région de l'espace "utilisateur" par exemple. Une autre approche qui couvre de façon plus complète cette classe est d'empêcher les pointeurs de l'espace "utilisateur" d'être déréférencés en mode "noyau"[4]. Ce principe est d'ailleurs suivi par la solution de sécurité *PaX* [Spengler *et al.*, 2003] avec son mécanisme *UDEREF* [Spengler, 2007].

Au sujet de la classe 1.2 Pour se protéger contre la classe 1.2, les approches adoptées pour la classe 1.1 ne sont pas satisfaisantes parce qu'il n'y a pas d'injection de code, seulement des variables du noyau qui sont modifiées.

[3] L'intégration du *patch* devrait être effectuée dans une version postérieure à la version 2.6.31 du noyau.

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

Afin d'empêcher les actions malveillantes de la classe 1.2.1 (qui provoque l'altération des variables de l'état d'exécution), il est primordial de protéger les données de contrôle de flux (par exemple : protéger les informations de contrôle de flux dans la pile, empêcher les pointeurs "noyau" d'être altérés, etc.), mais ce n'est pas suffisant.

Les variables de l'état d'exécution sont nombreuses, des exemples ont été donnés dans la section 1.5. Il n'y a pas de solution générique pour protéger le noyau contre des modifications inappropriées de ces variables. Mais des solutions pour certains types de variable existent. Nous donnons dans la suite les principales approches qui on été publiées à notre connaissance.

Au sujet des données de contrôle de flux, nous pouvons considérer à un premier niveau les mécanismes de protection de détournement du flux d'exécution vis-à-vis des débordements de pile, comme StackGuard [Cowan *et al.*, 1998] ou Propolice/SSP (*Stack-Smashing Protection*) qui emploient des "canaris" (ces canaris sont des valeurs connues qui sont placées entre les données de contrôle et les variables locales dans la pile, afin de détecter des débordements éventuels).

Mais ces mécanismes ne protègent pas contre les débordements de tampon qui écrasent les pointeurs de fonction [Bulba et Kil3r, 2000] (comme les débordements dans le tas[5] [Anonymous, 2001]). Ainsi, à un deuxième niveau nous pouvons employer une approche générique de protection contre les débordements de tampon qui écrase les pointeurs, comme PointGuard [Cowan *et al.*, 2003], solution fondée sur le chiffrement automatique des pointeurs lorsqu'ils se trouvent en mémoire. Cette dernière solution est vraiment intrusive et dépend de la confidentialité de la clé de chiffrement. À ce niveau, l'approche que nous proposons dans le chapitre 4 est complémentaire, elle limite les mauvais comportements du noyau.

Concernant la protection contre les débordements dans le *tas* du noyau Linux (c'est-à-dire au niveau de l'allocateur dynamique de mémoire de type SLAB) qui résulte en des modifications de pointeurs de fonction (classe 1.2.1), le mécanisme *KERNHEAP* [H., 2009] propose une solution complète qui déploie des moyens à différents niveaux dans le code du noyau. Ce mécanisme sert également à empêcher l'exploitation des vulnérabilités associées, comme par exemple la libération d'une région de mémoire à deux reprises sans qu'il n'y ait eu de réallocation entre (vulnérabilité de type *double-free*). Ce dernier type de vulnérabilité peut être exploité dans des conditions très spécifiques qui dépendent de l'allocateur (le comportement que cet allocateur doit adopter lors de la seconde libération de mémoire n'étant spécifié par aucun standard) [Dobrovitski, 2003].

Afin de protéger le noyau contre des actions malveillantes plus insidieuses comme les débordements de compteurs de référence [Pol, 2004], la solution de sécurité *PaX* [Spengler *et al.*, 2003] fournit une protection générique avec son mécanisme *REFCOUNT*.

Pour bloquer les actions malveillantes de la classe 1.2.2 (qui provoque l'altération des variables auxiliaires), il n'existe actuellement, à notre connaissance, aucune approche. Le chapitre suivant présente notre démarche fondée sur la préservation de contraintes au travers des mécanismes matériels de support à la virtualisation. Elle fournit une solution pour couvrir partiellement cette classe.

[5] Le tas est la région de mémoire employée pour les allocations dynamique de mémoire. Dans le noyau, le tas est souvent géré par un allocateur de type SLAB.

3.2.2.2 Vis-à-vis des actions internes au noyau

Remarquons que les mécanismes présentés précédemment sont inefficaces, dès lors que l'on considère des actions malveillantes agissant depuis le noyau. Par exemple la mise à zéro de l'attribut NX de pages de données (les rendant alors exécutables), ne peut être empêchée par les méthodes précédentes.

Au sujet de la classe 1.1 Le projet SecVisor [Seshadri et Qu, 2007] a mis en œuvre notamment une approche similaire à la notre (bien qu'implémentée de façon différente) concernant l'agencement en mémoire de l'image du noyau. Il s'appuie sur les extensions de virtualisation SVM des processeurs AMD, pour virtualiser la mémoire et empêcher toute modification des régions de code du noyau et toute exécution de ces régions de données. Cette approche couvre alors essentiellement les actions de la classe 1.1.2 et 1.1.3 (un mécanisme est aussi mis en œuvre pour couvrir la classe 1.1.4).

Le projet NICKLE [Riley *et al.*, 2008], quant à lui, s'appuie sur de la virtualisation logicielle afin d'empêcher que du code "noyau" non autorisé ne puisse être exécuté. Cette approche n'est pas intrusive car elle ne requiert pas la modification du système d'exploitation qu'elle protège. Elle emploie une technique dite de *memory shadowing*. L'idée est de copier initialement le code du noyau dans une région sous le contrôle de l'hyperviseur et de faire en sorte que l'exécution des instructions du code "noyau" (lors de la phase de *fetch* de la CPU) soit redirigée vers la copie. Par contre l'exécution de code "utilisateur" n'est alors pas redirigée comme tous les autres types d'accès à la mémoire. Cette approche couvre alors les actions de la classe 1.1.2 et 1.1.3. Par contre, elle n'est d'aucun secours contre les actions de la classe 1.1.4, car fondamentalement l'approche n'empêche aucune exécution de code de l'espace "utilisateur" depuis le noyau. De plus l'approche employée pour la redirection de l'exécution n'est envisageable que dans un contexte d'émulation de CPU[6]. C'est d'ailleurs pourquoi elle n'a été mise en œuvre qu'avec les émulateurs *qemu* [Bellard, 2005] et VirtualBox. L'approche nous semble alors trop restrictive.

En ce qui concerne les actions de la classe 1.1.1, SecVisor et NICKLE proposent d'adopter une démarche de chargement de module soumise à une politique d'autorisation. Ils ne s'étendent toutefois que très peu sur ce sujet.

Au sujet de la classe 1.2 Le projet *Secure Virtual Architecture* (SVA) [Criswell *et al.*, 2007], propose un environnement virtuel complet (actuellement implémenté pour un ordinateur x86) sur lequel peut être porté un système d'exploitation et ses applications. L'environnement proposé dispose de son propre jeu d'instructions pour lequel les sources du noyau et des différents programmes doivent être compilés. Ce jeu d'instructions permet de mettre en œuvre des propriétés de sûreté de fonctionnement vis-à-vis de l'utilisation de la mémoire et de l'intégrité du flot de contrôle, ce qui est effectué par un compilateur dédié. Toutefois, le *bytecode* résultant est également vérifié vis-à-vis de ces propriétés lors de son chargement par SVA. Cette approche ne considère toutefois pas les actions notamment de la classe 1.2.1 qui rendent possibles les actions de la classe 1.1.4, sauf pour les cas simples de type déréférencement de pointeurs non-initialisés. En outre, les propriétés vérifiées ne garantissent que l'innocuité du code, mais ne concluent pas en l'absence de vulnérabilités. Dès lors, certaines actions malveillantes résultant

[6] Elle modifie l'émulation de la CPU au niveau de la phase de *fetch* des instructions.

de l'exploitation de ces éventuelles vulnérabilités ne peuvent être capturées. Les auteurs expliquent d'ailleurs clairement (contrairement au nom de leur architecture) qu'ils adoptent dans leur approche le point de vue de la sécurité-innocuité car il ne considère pas les actions malveillantes. Enfin, l'approche ne permet pas de capturer les actions de la classe 1.2.2, car assurer l'intégrité des variables auxiliaires sort du cadre de la sécurité-innocuité.

3.2.3 Mécanismes de détection et de restauration

Pour les mécanismes de détection, il n'y a pas à différencier les actions internes des actions externes, car la détection n'a lieu qu'après que les actions aient été effectuées. Par contre, il est important de considérer différents types de détection suivant qu'elle s'effectue depuis le noyau (voire l'espace "utilisateur" pour les solutions les plus rudimentaires) ou depuis un mode d'exécution plus privilégié (au travers de la virtualisation par exemple) ou du moins depuis une zone qui n'est pas sous le contrôle du noyau (comme la routine de traitement de la SMI). Vis-à-vis des mécanismes de restauration en ligne, à notre connaissance ils n'ont été étudiés que dans le cadre de la sécurité-innocuité. Notons par exemple, le projet Nooks [Swift *et al.*, 2005, 2006] qui vise à améliorer la robustesse d'un système d'exploitation en ne considérant que les fautes accidentelles. Il couple pour cela un mécanisme de détection de fautes à un mécanisme de restauration.

3.2.3.1 Agissant depuis le noyau ou l'espace "utilisateur"

Le problème majeur de ces approches est qu'elles agissent au même niveau de privilège que ce qu'elles souhaitent contrôler, voire depuis un niveau inférieur (l'espace "utilisateur").

Parmi les solutions existantes, on trouve le plus souvent des outils qui ont été développés dans l'optique de détecter la présence de *rootkit* "noyau". Les premières approches s'appuyait notamment sur la vérification de points particuliers dans le noyau, dont l'altération révélait souvent la présence de *rootkits* [Butler et Hoglund, 2004]. En soit, il s'agissait de détecter une sorte de signature des *rootkits*. Cependant ces approches se sont révélées infructueuses avec l'évolution des *rootkits* (*cf.* chapitre 2 section 2.1.3). Néanmoins, elles restent efficaces pour contrer la majorité des infestations actuelles, car les *rootkits* plus évolués ne sont pas vraiment répandus. Ces mécanismes détectent alors des modifications inappropriées qui font suite à l'occurrence d'actions de la classe 1.1.2 (lorsque du code a été modifié) ou de la classe 1.2.1 (lorsque les modifications touchent à des adresses de fonctions).

Les approches de détection de *rootkits*, les plus en vogue actuellement, recoupent des informations sur l'état du noyau qui ont été récupérées à différents niveaux d'abstraction pour détecter d'éventuelles incohérences (*cross-view based detection*). Typiquement, une analyse de l'état est effectuée depuis l'API du système, et une autre à partir d'une lecture brute de la mémoire (détection de processus cachés), ou du disque (détection de fichiers cachés) selon le cas. Il s'agit alors principalement de détecter la présence d'altérations suite à l'occurrence d'actions de la classe 1.2.2. Dans le cas de la détection de fichiers cachés, il s'agit par exemple de parcourir le système de fichiers et de comparer avec les résultats d'une reconstruction du système de fichiers à partir d'une lecture directe du disque (soit en appelant directement le pilote du disque, soit en envoyant directement les commandes d'entrée/sortie au disque). Bien que l'idée semble alléchante, à partir du moment où le résultat des lectures peut être sous le contrôle du

rootkit, la détection peut être mise en échec. C'est le cas des projets RootkitRevealer [Cogswell et Russinovich, 2006], [Black Light] et [Wang *et al.*, 2005] (tous développés pour le système d'exploitation Windows), qui implémentent leur approche au même niveau de privilège que le *rootkit*.

3.2.3.2 Agissant depuis une zone qui n'est pas sous le contrôle du noyau

Approche fondé sur l'emploi d'un co-processeur Le projet Copilot [Petroni *et al.*, 2004], tout comme les travaux menés dans [Zhang *et al.*, 2002], détectent des altérations de la mémoire dans des régions spécifiques du noyau, afin de détecter la présence de certains *rootkits*. Ces approches se fondent sur une lecture de la mémoire depuis une carte PCI (via du DMA) qui intègre un processeur. La détection repose sur la vérification de la correspondance de deux empreintes de mémoire, l'une correspondant à l'empreinte de référence (stockée sur la carte PCI lors d'une préalable initialisation) et l'autre effectuée en cours d'exécution. Dès lors cette technique ne permet que de détecter les altérations de régions statiques du noyau. Elle détecte alors uniquement l'occurrence des actions de la classe 1.1.2, et certaines actions de la classe 1.2.1 (modification des tables statiques qui contiennent des pointeurs de fonctions, comme la table des appels-système).

Les auteurs de [Petroni Jr *et al.*, 2006] proposent une approche très intéressante se préoccupant de la détection de violation de contraintes vis-à-vis de données dynamiques du noyau (classe 1.2.1 et 1.2.2). Les travaux menés ont pour objet notamment la détection de *rootkits* et se reposent sur le fait qu'ils cachent habituellement des processus en cassant les liens qui les relient au noyau, et donc altèrent la valeur de certains pointeurs de données. La détection se fonde sur la constatation de violations de contraintes, lesquelles ont été préalablement spécifiées dans un langage haut niveau. Un générateur de code est également proposé pour produire automatiquement le code de vérification. La détection se repose sur des lectures périodiques de la mémoire du système au travers d'une carte PCI qui effectue des accès DMA, similaire à l'approche qu'ils ont proposé dans le projet Copilot [Petroni *et al.*, 2004]. On retrouve dès lors le même problème constaté auparavant, à savoir que la lecture de la mémoire via ce canal peut être biaisée par un *rootkit* [Rutkowska, 2007]. Toutefois cette approche pourrait être déployée au sein d'un hyperviseur afin de pallier ces problèmes. Aussi, la façon dont ils modélisent les contraintes pour en vérifier *a posteriori* leur violation, pourraît être un point de départ à une automatisation de notre approche de *préservation* de contraintes que nous développons dans le prochain chapitre.

Approche fondé sur la virtualisation Suite à leur travaux dans [Petroni Jr *et al.*, 2006], les auteurs ont proposé une approche de détection de violation de l'intégrité des flots de contrôle [Petroni Jr et Hicks, 2007], toujours dans l'optique de détecter la présence de *rootkit*. Cette fois, l'approche est mise en œuvre au sein d'un hyperviseur (preuves de concept proposées pour Xen et Vmware Workstation). L'analyse de la mémoire est effectuée périodiquement et en deux étapes. La première passe vérifie que le code du noyau n'a pas été modifié, et la seconde vérifie que "tous" les pointeurs de fonction ciblent du code valide. Vis-à-vis de cette dernière étape, une génération automatique du code de l'analyseur est effectuée à partir des sources du noyau (la mise en œuvre a été effectuée pour Linux). Cependant, leurs travaux ne traitent pas les régions de données variables telles que celles dans la pile (de l'espace "noyau"), par laquelle les flots

de contrôle peuvent être détournés. Notre approche de porte dérobée "noyau" (*cf.* chapitre 2 section 2.3) ne serait par exemple pas détectée. Leur ambition est tout de même affichée : ils ne s'occupent que de la détection de *rootkits* effectuant des redirections permanentes.

Le projet Livewire [Garfinkel et Rosenblum, 2003] est un mécanisme de détection d'intrusion construit au sein d'un hyperviseur. Il détecte des intrusions en observant l'état du noyau qui est exécuté dans une machine virtuelle. Notons que ce projet propose tout comme SecVisor et NICKLE, une protection du code du noyau mais de façon plus limité car la démarche suivie ne couvre pas les classes 1.1.1, 1.1.3 et 1.1.4.

Le projet [Jiang *et al.*, 2007] met en œuvre la technique de recoupement d'information (*cross-view based detection*) similaire à celle des approches précédentes, mais cette fois-ci au sein d'un hyperviseur. Il échappe dès lors au contrôle du *rootkit*, ce lui permet de détecter efficacement les altérations effectuées par des actions de la classe 1.2.2. Un problème cependant affecte toujours ces approches si l'on considère que le *rootkit* emploie une technique de parasitisme mobile (approche que nous avons proposée dans le chapitre 2 section 2.3.7.3) qui lui suffit pour lancer les commandes de l'attaquant. Dès lors la détection d'objets cachés devient caduque.

Notons enfin l'absence de publication sur des approches de détection implantées au sein de la routine de gestion de la SMI.

3.3 Protection contre les actions de la classe 2 et de la classe 3

Les protections existantes à ce niveau se limitent bien souvent aux fonctionnalités offertes par le noyau du système d'exploitation.

3.3.1 Contrôle des vecteurs d'accès à la mémoire du support de contrôle et aux périphériques

Nous avons déterminé un unique vecteur d'accès à la mémoire du support de contrôle et aux périphériques dans les sections 1.5.2 et 1.5.3 (mis à part les accès possibles entre périphériques, que nous n'avons pas étudiés dans nos travaux). Nous examinons maintenant les mesures de sécurité qui s'appliquent à ce niveau.

À notre connaissance, les actions de ces classes ne sont couvertes que partiellement. Elles ne traitent au mieux que les actions externes qui proviennent de l'espace "utilisateur", lesquelles agissent en *ring* 3 et sont ainsi contrôlées par le noyau ou un autre mécanisme de sécurité dans l'espace "noyau". Ces derniers octroient ou enlèvent des privilèges aux processus de l'espace "utilisateur" afin qu'ils accèdent ou non aux périphériques critiques ou à la mémoire du support de contrôle (sous les systèmes de type Unix il s'agit des appels-système `iopl()` et `ioperm()`, *cf.* section 1.5.3). Toutefois, deux problèmes sont à percevoir. D'une part, ce contrôle des privilèges est effectué en autorisant ou non l'accès à certains *ports* d'entrée/sortie, dont les conséquences en terme de sécurité peuvent être impossible à prévoir. Par exemple, l'autorisation d'accès aux deux *ports* PIO de configuration PCI, donne libre accès aux périphériques PCI et à la mémoire du support de contrôle. D'autre part, ces approches comme nous l'avons dit ne contrôlent pas (et d'ailleurs ne le peuvent pas) les actions internes au noyau ni celles

provenant du mode SMM (*cf.* section 1.4.1). Ainsi, ces protections peuvent être abusées. Par exemple, un attaquant pourrait exploiter en premier lieu une éventuelle vulnérabilité du noyau ou de la routine de traitement de la SMI (c'est-à-dire effectuer une action malveillante *externe* au noyau) qui lui permette d'exécuter du code en *ring* 0 (ou en mode SMM), et ainsi avoir un accès total sur les périphériques et la mémoire du support de contrôle.

3.3.2 Mécanismes de prévention

Les systèmes qui gèrent de multiples machines virtuelles, proposent des mécanismes d'isolation entre ces machines deux à deux, mais également entre ces machines et la machine physique. Nous discutons plus longuement de la virtualisation dans la section 4.1.1.

Dans notre travail nous proposons une solution de protection de la mémoire du support de contrôle et des périphériques au travers de l'utilisation de mécanismes matériels de support à la virtualisation (*cf.* chapitre 4).

3.3.3 Mécanismes de détection et de restauration

L'approche du *Trusted Computing Group* [ISO/IEC, 2009] (sur laquelle nous revenons brièvement dans le chapitre 4), prévoit l'utilisation d'un module matériel appelé TPM (*Trusted Platform Module*), afin de garantir l'intégrité de composants matériels et logiciels lors du démarrage d'un ordinateur. Ainsi, l'approche du TCG prévoit la vérification par le TPM de l'intégrité du BIOS de la machine ainsi que des ROM d'extension du BIOS qui se trouvent sur les différents périphériques PCI[7].

Les travaux menés dans [Desnos *et al.*, 2009] rendent possible la détection de *rootkits* "hyperviseur" (plus précisément des *rootkits* HVM ou *Hardware-assisted Virtual Machine*, comme *Blue Pill* [Rutkowska, 2006]). Une des caractéristiques de ces *rootkits* est qu'ils modifient le support de contrôle. Les auteurs proposent une méthode de détection de cette modification. L'idée repose sur la modélisation du comportement d'une machine infestée par un *rootkit* HVM, et du comportement de la même machine non compromise. Leur modélisation est statistique. La difficulté revient ainsi à choisir un estimateur qui révèle une différence de comportement statistiquement significative. La construction de l'estimateur qu'ils ont mis en œuvre, se fonde sur l'acquisition de la valeur d'un compteur qui mesure un temps relatif. Ils exécutent sur un des cœurs du processeur de la machine cible un programme incrémentant un compteur, et sur un autre cœur, un programme exécutant une opération que tout *rootkit* HVM à l'obligation d'intercepter, augmentant par là-même la durée d'exécution. Cette approche suppose toutefois que la machine dispose d'un processeur multicœurs (à défaut, elle peut être étendue pour employer le GPU d'une carte graphique), mais également que le *rootkit* HVM a activé la virtualisation sur tous les cœurs (et intercepte donc des instructions sur tous ces cœurs). Or, un *rootkit* HVM pourrait se contenter d'activer la virtualisation sur un seul cœur, si le contrôle total de la machine compromise n'est pas requis par l'attaquant. Encore plus insidieux, ce *rootkit* pourrait

[7] Certains périphériques disposent de ROM qui contiennent un code d'initialisation. Ce code est appelé lors du démarrage d'une machine par le BIOS. C'est pourquoi, il est nécessaire de s'assurer de l'intégrité de ces ROM si l'on souhaite s'assurer de l'intégrité du démarrage d'une machine.

également se déplacer de cœur en cœur. Toutefois cette dernière approche semble bien plus complexe a mettre en œuvre.

Notons enfin un travail de synthèse sur la détection de *rooktits* HVM publié dans [Fritsch, 2008].

3.4 Conclusion

Dans ce chapitre, nous avons analysé les moyens de protection existants contre les actions malveillantes ciblant un noyau de système d'exploitation, qu'il s'agisse de moyens de prévention ou de détection et restauration. Nous avons également présenté brièvement les moyens usuels d'élimination de vulnérabilités bien que ne faisant pas partie de notre problématique, dans un but d'exhaustivité. Enfin, nous avons effectué cette analyse à chaque niveau de notre classification, et suivant que les actions incorrectes soient externes ou internes au noyau. Les manques vis-à-vis de la protection contre les actions incorrectes internes au noyau, et les manques sur la couverture notamment des actions de la classe 1.2 en matière de prévention, nous ont amenés à élaborer une nouvelle approche.

Concernant le positionnement de notre démarche vis-à-vis des moyens de la sûreté de fonctionnement, nous nous sommes concentrés dès le début de notre travail de manière privilégiée sur l'élaboration de mécanismes de prévention. Cet axe de recherche a été privilégié par rapport aux mécanismes de détection et restauration de par la connaissance de résultats fondamentaux quant à ces derniers. En effet, il a été démontré par Fred Cohen que la détection des maliciels est un problème indécidable [Filiol, 2004, Chap. 3]. Les mécanismes de restauration nécessitant une détection préalable de l'infection, tombent également sous le joug de l'indécidabilité (bien qu'indirectement). En outre, détecter qu'une action a effectué un comportement incorrect, pour ensuite le signaler ou restaurer le système est une approche intrinsèquement faillible. En effet, il serait juste de se demander pourquoi le logiciel malveillant ne pourrait pas, suite à ces actions, fausser les résultats de lecture de ces "moniteurs d'intégrité", d'autant plus au vu des complexités architecturales des plateformes x86. L'exemple est frappant avec le cas dont nous avons discuté en section 3.2.3 au sujet de [Petroni Jr *et al.*, 2006].

Nous développons dans le prochain et dernier chapitre, notre approche de protection des noyaux, fondée sur la préservation de contraintes pour laquelle nous nous appuyons sur les extensions matérielles d'aide à la virtualisation. Notre démarche de préservation de contrainte se démarque des solutions actuelles fondées essentiellement sur la détection *a posteriori*. Elle nous permet également de traiter le problème de l'intégrité d'un noyau de façon plus globale. En effet, contrairement aux approches que nous avons vues, nous considérons l'intégrité d'un noyau dans son ensemble, c'est-à-dire en couvrant notamment également le support d'exécution du noyau (classe 2).

Chapitre 4

Hytux : assurer l'intégrité d'un noyau au travers de la préservation de contraintes

L'intégrité du noyau d'un système d'exploitation, revient à considérer deux facettes comme nous l'avons vu dans le chapitre 1. D'une part il faut assurer l'intégrité de la structure du noyau et d'autre part celle de son état. Alors que l'intégrité de la structure du noyau renvoie à des solutions existantes ou dont la mise en œuvre est possible (cf. chapitre 3), l'intégrité de son état est un problème bien plus complexe étant donné sa nature variable. Garantir l'intégrité de l'état d'un noyau revient à garantir l'intégrité des éléments dont il dépend pour son fonctionnement (et qui forme l'état du support de contrôle), ainsi qu'à garantir l'absence de vulnérabilités dans sa structure (leur exploitation conduisant inévitablement à une perte d'intégrité au moins de l'état du noyau) ou sinon de pallier ces vulnérabilités au travers de mécanismes externes au noyau. Nous justifions le caractère externe de ces mécanismes dans les paragraphes suivants.

Les mesures traditionnelles de sécurité (analysées dans le chapitre 3) font face à des problèmes insolubles en ce qui concerne les actions malveillantes qui s'exécutent dans l'espace "noyau". Les approches fondées sur la virtualisation que nous avons également examinées dans le chapitre 3, font partie des travaux qui rendent envisageable la protection des noyaux. Il s'agit principalement de solutions de détection de compromission de noyau. Mais quelques rares projets ont une approche relativement similaire à la notre, à savoir la mise en œuvre de mécanismes de préservation de l'intégrité de noyau au sein d'un hyperviseur. Ces mécanismes s'exécutent ainsi à un niveau de privilège supérieur à celui du noyau. Le projet SecVisor [Seshadri et Qu, 2007] a mis en œuvre notamment une approche très similaire à la notre concernant la protection de la mémoire du noyau[1]. Toutefois, notre travail traite le problème de l'intégrité d'un noyau de façon plus globale, en essayant de préserver l'intégrité des différentes éléments nécessaires au bon fonctionnement du noyau, et non seulement de sa mémoire.

Dans notre approche, nous essayons de couvrir les problèmes évoqués dans la classification que nous avons mise en place dans le chapitre 1 en limitant les dommages que le noyau peut causer au système. Afin de rendre ces mesures de sécurité possibles, nous implémentons un hyperviseur qui contrôle certaines des actions qu'un noyau peut effectuer. L'idée principale de notre démarche est de préserver les contraintes qui sont censées exister entre le système informatique et le noyau. Cette approche est possible grâce aux technologies matérielles de

[1] Nos travaux et ceux du projets SecVisor ont été menés de façon indépendante.

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, notre hyperviseur est de taille très réduite. D'une part il n'est conçu que pour contrôler une seule machine virtuelle, à savoir le système que l'on souhaite protéger, et d'autre part il profite du support matériel pour la virtualisation, ce qui facilite grandement son implémentation. Ainsi il serait envisageable d'effectuer une analyse statique du code de notre hyperviseur pour en garantir sa correction vis-à-vis de la sécurité (absence de vulnérabilités).

Nous commençons ce chapitre par une introduction sur la virtualisation (section 4.1.1) afin de mieux comprendre dans quel contexte notre approche se situe. Nous parcourons rapidement les principes de la virtualisation, dressons la liste des différentes formes de virtualisation qui existent, et examinons le support matériel fournit par les architectures de type x86. Nous continuons alors sur des rappels techniques sur la façon dont est structuré l'espace d'adressage du noyau Linux (section 4.1.2). Dans notre travail, nous tentons d'assurer l'intégrité des noyaux de système d'exploitation et plus particulièrement celle du noyau Linux. C'est pourquoi, un des points primordiaux à connaître est la façon dont le noyau Linux s'articule en mémoire.

Nous présentons alors succinctement notre démarche pour la protection de noyaux (section 4.2), qui repose sur l'élaboration d'un hyperviseur léger dont la charge est de contrôler un noyau Linux. Nous poursuivons sur une explication détaillée de notre approche de préservation de contraintes pour l'assurance de l'intégrité d'un noyau avec une application sur le noyau Linux (section 4.3). Il s'ensuit une discussion sur les mécanismes que déploie notre hyperviseur pour assurer son intégrité (section 4.4). Nous terminons par une synthèse sur notre approche de préservation de contraintes (section 4.5), à propos de sa couverture des différentes classes d'actions malveillantes sur un noyau (établie dans le chapitre 1), et au sujet de l'impact qu'elle occasionne sur les performances du système.

4.1 Concepts préliminaires

4.1.1 Introduction à la virtualisation

Les systèmes informatiques sont construits selon une hiérarchie d'interfaces qui séparent différents niveaux d'abstractions. Chaque couche d'abstraction propose au niveau supérieur une interface qui permet de simplifier l'interaction avec le niveau inférieur. Par exemple, au niveau du système d'exploitation, les fichiers constituent une abstraction qui rend possible une interaction simplifiée avec la mémoire de masse pour les applications.

Définissons à présent le concept de *virtualisation* [Smith et Nair, 2005]. On virtualise un composant (comme un processeur, la mémoire ou un périphérique d'E/S) à un niveau d'abstraction donné lorsqu'on propose une interface et des ressources, construites à partir des siennes, aux composants se situant au dessus de ce niveau d'abstraction. Le composant virtualisé est alors vu comme un ou plusieurs composants virtuels. Ces derniers proposent soit une interface et des ressources différentes du composant virtualisé (on parle dans ce cas d'émulation), ou bien proposent une interface et des ressources identiques. Il est important de noter que la virtualisation se distingue de la notion d'abstraction (ou couche d'abstraction) car son rôle n'est pas de

simplifier l'interaction avec le système sous-jacent, mais d'en proposer une vision différente. Un exemple de virtualisation très répandu dans la plupart des OS actuels est celui de la mémoire. Cette virtualisation s'opère via la mise à disposition d'un espace d'adressage identique pour chaque processus (grâce à l'unité matérielle de pagination que l'on retrouve dans la plupart des architectures). Enfin, dans le cas où le composant virtualisé est un système complet, on appelle ce composant virtuel, une Machine Virtuelle (VM — Virtual Machine).

Notons enfin que les premiers modèles formels sur la virtualisation sont issus des travaux de Goldberg [Goldberg, 1973], lequel a ensuite mis en évidence avec Popek, ce qu'une architecture informatique requiert pour être virtualisable [Popek et Goldberg, 1974].

4.1.1.1 Les deux classes principales de virtualisation

La virtualisation se découpe généralement en deux classes suivant qu'elle s'effectue au niveau applicatif ou au niveau système. Dans le premier cas, il s'agit de virtualiser uniquement l'environnement des applications par le biais d'un virtualiseur nommé *runtime software* comme la JVM (*Java Virtual Machine*), laquelle exporte de surcroît sa propre ISA (*Instruction Set Architecture*). Les applications qui s'exécutent dans cet environnement disposent alors chacune d'un ensemble privé de ressources, toujours agencées de la même façon mais dont les quantités peuvent varier. Dans le cas de la virtualisation de système, il s'agit de virtualiser complètement l'environnement matériel au sein d'une machine virtuelle pour qu'elle puisse accueillir un système d'exploitation au complet (ou au minimum son espace "utilisateur" comme avec la solution *Linux-Vserver* [des Ligneris, 2005]). Les virtualiseurs mis en jeu dans ce type de virtualisation sont couramment appelés *Virtual Machine Monitor* (VMM) ou encore *hyperviseur*. Dans la suite de cette section, nous rappelons les différents éléments se rapportant à la virtualisation de systèmes. Ce rappel sert à mieux cerner notre travail sur la protection de noyaux de systèmes d'exploitation, qui emprunte certaines techniques issues de la virtualisation de systèmes.

4.1.1.2 La virtualisation de systèmes

Nous venons de voir qu'un hyperviseur met en place une ou plusieurs machines virtuelles dans lesquelles sont exécutés des systèmes d'exploitation. Ces systèmes sont alors appelés systèmes invités (*guest system*).

Rappelons qu'une machine virtuelle doit proposer une virtualisation des différents éléments matériels nécessaires à l'exécution d'un système d'exploitation. Les différents composants à virtualiser sont la CPU, la mémoire, et les périphériques d'E/S.

La virtualisation de systèmes est mise en œuvre au travers de la gestion de machines virtuelles. Celle-ci est orchestrée par l'un des deux types d'hyperviseurs suivants [King *et al.*, 2003].

– **Hyperviseur de type I :**
  Un hyperviseur de type I, aussi appelé *bare-metal hypervisor*, s'exécute directement sur le matériel et propose des machines virtuelles aux systèmes invités en s'appuyant uniquement sur l'interface et les ressources du matériel sous-jacentes. Il doit ainsi implémenter, entre autres, la gestion mémoire complète des machines virtuelles et leur ordonnance-

ment. En d'autres termes, il doit implémenter la plupart des services que fournissent les noyaux de systèmes d'exploitation courants. Certains hyperviseurs de ce type réduisent toutefois leur complexité, en employant une machine virtuelle privilégiée qui abrite les pilotes des périphériques du système (et autorise également le contrôle de l'hyperviseur). Ces types d'hyperviseurs sont parfois qualifiés d'hybride. Ils profitent des avantages du type I et du type II, tout en limitant les inconvénients des deux approches [Abramson *et al.*, 2006].

– **Hyperviseur de type II :**
Un hyperviseur de type II, aussi appelé *host-based hypervisor*, s'installe sur un système d'exploitation (lequel est alors appelé système hôte) et profite de ses abstractions et fonctionnalités pour créer les environnements virtuels. Pour la plupart, l'abstraction des processus des OS sert de support aux machines virtuelles et ainsi l'ordonnancement de ces machines est effectué directement par l'ordonnanceur de l'OS. La gestion mémoire s'appuie également sur les services de l'OS, simplifiant par conséquent l'implémentation d'un hyperviseur de ce type. Un inconvénient d'un hyperviseur de type II est que sa sûreté de fonctionnement dépend directement de celle du système hôte.

Quelque soit le type de l'hyperviseur, il existe différentes approches pour celui-ci de présenter le système physique aux systèmes invités, c'est-à-dire qu'il existe différentes approches pour concevoir une machine virtuelle.

– **La virtualisation complète :**
La virtualisation complète ou virtualisation native fournit un environnement virtuel (interface et ressources) censé représenter une architecture réelle et cela sans aucune simplification (ou plus généralement sans aucune modification) de l'interface et des ressources associées à cette architecture. Ainsi, un système d'exploitation développé pour être exécuté sur une architecture particulière, s'exécute de façon native (c'est-à-dire sans changement) dans une machine virtuelle qui virtualise complètement cette architecture. Lorsque l'architecture virtualisée est différente de l'architecture matérielle sous-jacente on qualifie la virtualisation d'émulation matérielle car il s'agit alors d'une imitation d'architecture construite à partir d'une autre architecture.
Comme solutions de virtualisation complète on remarque par exemple les solutions :
  – KVM (*Kernel Based Virtual Machine*) qui est un hyperviseur de type II fonctionnant sous Linux et qui propose une virtualisation complète de l'architecture IA-32 et IA-64 [Kivity *et al.*, 2007] ;
  – qemu qui virtualise complètement de multiples architectures [Bellard, 2005] ;
  – VMware ESX et VMware Workstation [VMware, Inc., 2006] respectivement de type I et II.

– **La paravirtualisation :**
La paravirtualisation ne cherche pas à présenter une architecture matérielle complète aux systèmes invités. À la différence de la virtualisation complète où le système invité croit idéalement s'exécuter à même le matériel, elle instaure une collaboration entre l'hyperviseur et ses systèmes invités, afin d'en faciliter leur gestion et ainsi d'obtenir de très

bonnes performances d'exécution. Cela entraîne bien évidemment la modification des systèmes invités pour qu'ils fassent appel directement à l'hyperviseur pour effectuer des opérations privilégiées sur le matériel, évitant ainsi que l'hyperviseur implémente un mécanisme d'interception de ses opérations privilégiées. Deux interfaces génériques d'appel à un hyperviseur ont notamment été proposées : l'infrastructure `paravirt_ops` [Russell, 2007] et VMI [Amsden *et al.*, 2006]. L'objectif est de remplacer les opérations privilégiées d'un OS par des appels à l'hyperviseur. L'hyperviseur n'a ainsi plus besoin d'implémenter de mécanismes d'interception de ces opérations, et améliore ainsi ses performances globales.
Cette approche permet d'obtenir de bonnes performances avec des architectures matérielles qui ne sont pas virtualisables, comme le x86 d'origine (avant l'arrivée des extensions matérielles de virtualisation). Par exemple, le x86 ne définit pas `sidt` (instruction qui écrit en mémoire la valeur du registre `idtr`) comme une instruction privilégiée (c'est-à-dire qu'elle peut être exécutée en *ring 3*), et empêche ainsi son interception par l'hyperviseur. Cette interception est cependant nécessaire à une virtualisation complète, car l'architecture x86 ne dispose que d'un seul registre `idtr`, et virtualiser ce registre suppose d'intercepter les différents accès qui peuvent s'y produire. Pour pallier les déficiences de ce type d'architecture en matière de virtualisation, des traitements logiciels lourds sont nécessaires si l'on ne souhaite pas mettre en place de la paravirtualisation (parce que l'on ne souhaite pas faire confiance à l'OS invité par exemple). Certaines instructions du code du système invité peuvent être par exemple remplacées en mémoire avant exécution afin que l'hyperviseur puisse les intercepter.
On trouve, par exemple, comme hyperviseurs appartenant à cette classe :
– Xen [Barham *et al.*, 2003] et lguest [Russell, 2007] qui emploient `paravirt_ops` ;
– User Mode Linux qui relocalise le noyau Linux sur x86 en *ring 1* [Dike, 2001] ;
– VMware Workstation associé aux VMware tools [VMware, Inc., 2006].
Notons que les *VMware tools* s'installent sur le système invité et remplacent les opérations privilégiées de ce système par des appels directs à l'hyperviseur. Ainsi, *VMware Workstation* associé au *VMware tools* fournit la paravitualisation.

– **La virtualisation au niveau du système d'exploitation :**
Également appelée virtualisation par *containers*, la virtualisation au niveau du système d'exploitation autorise l'instanciation de multiples espaces "utilisateur" sur un même noyau, en les isolant les uns des autres par le biais de différents espaces de noms (chaque *container* dispose de son propre espace de `PID` par exemple). Dans ce type de virtualisation, il s'agit de rendre la plupart des structures du noyau instanciables afin de renforcer l'illusion de l'existence de multiples machines. Les *Jails* de *BSD* [Kamp et Watson, 2000] peuvent être vu comme un précurseur des *containers*.
Des solutions de ce type de virtualisation sont par exemple :
– les Zones de Solaris [Price et Tucker, 2004] ;
– Linux Vserver [des Ligneris, 2005] ou OpenVZ/Virtuozzo [Kolyshkin, 2006], lesquels peuvent faire tourner différentes distributions GNU/Linux sur un même noyau.
À noter toutefois que l'isolation entre les "machines virtuelles" de ce type de virtualisation dépend du niveau d'"instantiabilité du noyau" (afin qu'il y ait le moins d'interactions possibles au sein du noyau entre les différentes machines virtuelles).

4.1.1.3 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 les architectures IA-32 et Intel 64. Cette extension, nommée VT-x [Intel Corporation, 2008e], 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.

Afin de rendre une machine x86 complètement virtualisable, Intel a mis également en œuvre la technologie VT-d (*Virtualization Technology for Directed I/O*) [Intel Corporation, 2007; Abramson *et al.*, 2006]. Cette extension matérielle rend possible la virtualisation des périphériques d'entrées/sorties en agissant au niveau du *northbridge* (cf. section 1.4).

La technologie Intel VT-x 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 logiciels invités. 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 déclenchées par la CPU lorsque le système invité effectue certaines actions (que l'hyperviseur souhaite intercepter) ou encore lors d'évènements extérieurs (comme les interruptions). 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`[2]. La Figure 4.1 résume la façon d'utiliser ces instructions.

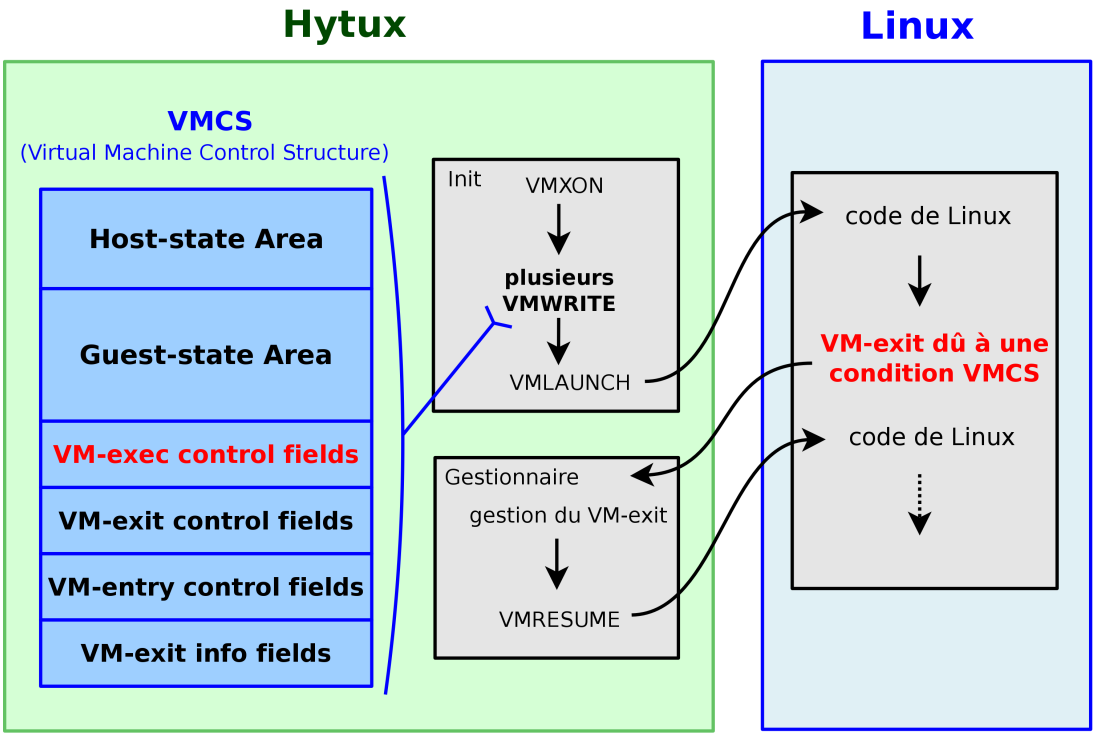

FIG. 4.1 – Bref aperçu de Intel VT-x

La technologie Intel VT-d Une *Input Output Memory Management Unit* (IOMMU) est une unité de gestion mémoire placée dans le *northbridge* du *chipset* de la carte mère (cf. figure 4.2). Comme la MMU traditionnelle (cf. section 1.4.5), l'IOMMU permet de virtualiser la mémoire physique en apportant un mécanisme similaire à la pagination dans le *chipset*[3]. Des tables de traduction d'adresses, associés à chaque périphérique, initialisées par le système d'exploitation et stockées en mémoire principale, permettent à l'IOMMU de faire correspondre aux adresses mémoire virtuelles (*DMA Virtual Address* ou DVA) du périphérique des adresses dans la mémoire physique (*Host Physical Address* ou HPA). Chaque périphérique possède ainsi sa propre vision de l'espace des adresses physiques (cf. figure 4.3).

L'IOMMU propose des fonctionnalités qui s'avèrent très utiles pour empêcher la corruption de la mémoire du système. Elle rend possible notamment le contrôle de l'accès des périphériques à la mémoire principale en s'assurant qu'il soit impossible à tout périphérique d'effectuer une opération DMA vers une adresse mémoire non autorisée.

[2] Notons que ces instructions ne peuvent être employées qu'en mode VMX root.
[3] Elle virtualise également les interruptions de type MSI (*Message Signaled Interrupt* employées par le protocole PCI Express).

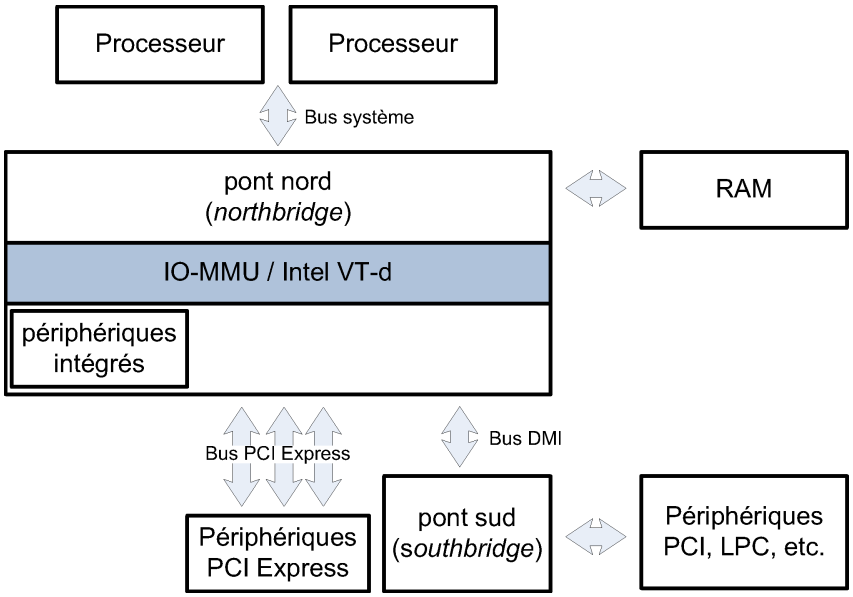

FIG. 4.2 – Architecture simplifiée d'un système avec la technologie Intel® VT-d

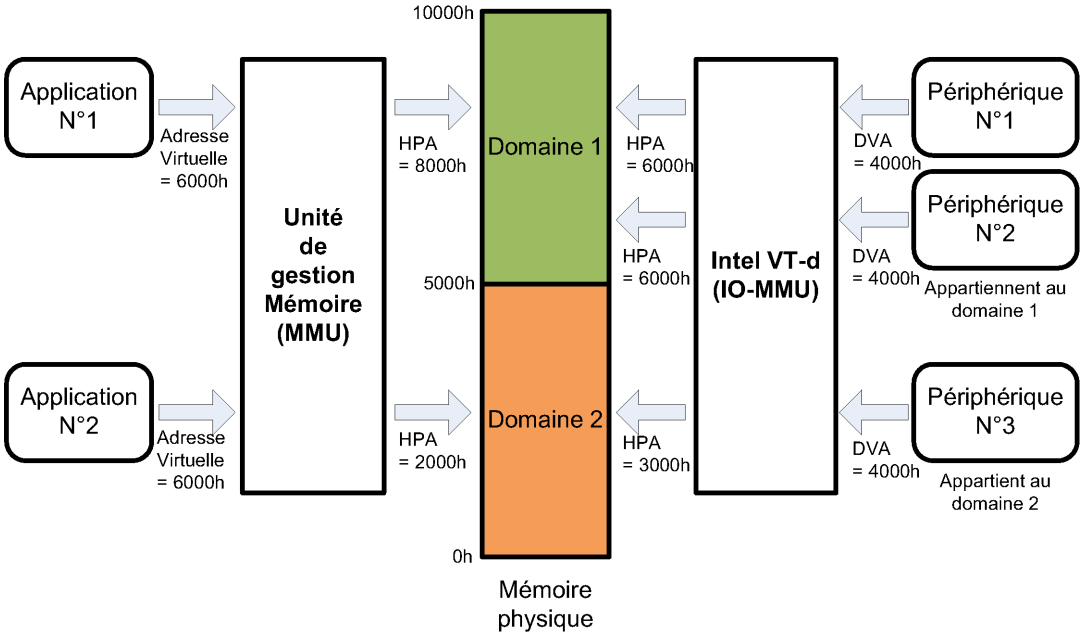

FIG. 4.3 – Virtualisation de la mémoire physique

Dans la pratique, chaque périphérique est affecté à un domaine logique, c'est-à-dire un ensemble d'adresses mémoire auquel il a accès. Ces domaines sont définis par des tables de traduction d'adresses. Lorsqu'un périphérique souhaite accéder à une adresse en mémoire, il génère une requête DMA (lecture ou écriture) sur le bus PCI. L'IOMMU, placée en coupure entre les périphériques et la mémoire principale, intercepte alors cette requête et utilise les tables de traduction d'adresses associées au périphérique et les informations qu'elles contiennent (existence de l'adresse demandée, accessibilité en lecture/écriture, etc.) pour évaluer la requête. Si la requête est correcte, l'adresse DVA contenue dans la requête DMA est alors traduite en l'adresse physique HPA correspondante. Dans le cas contraire, l'IOMMU bloque simplement la requête, prévient le périphérique que la requête a été refusée, et génère une interruption pour avertir le système d'exploitation.

4.1.2 L'espace d'adressage linéaire des processus sous Linux

Les processus disposent d'un espace d'adressage linéaire (aussi appelé virtuel) découpé en deux parties : un espace d'adressage pour leur propre usage, nommé espace "utilisateur" et un espace commun à tous les processus, nommé espace "noyau" (cf. figure 4.4) [Gorman, 2007; Bovet et Cesati, 2005]. Ce dernier est un espace pour lequel l'accès est restreint (l'attribut S/U des pages de mémoire qui constituent cet espace est positionné à 0 — cf. section 1.4.5.2). Il faut que le CPL (*Current Privilege Level*, encore nommé *ring*) du processeur soit à 0 pour que l'accès soit autorisé par la MMU. Ainsi seul le noyau peut y accéder. Si toutefois un processus qui s'exécute en *ring 3* essaie d'accéder à l'espace "noyau", une exception (faute de page) est levée par la MMU, le processeur passe alors en *ring 0* et donne la main au noyau afin qu'il gère la situation (terminaison du processus ayant causé cette faute).

La première partie de l'espace d'adressage des processus qui leur est spécifique, s'étend de l'adresse 0 jusqu'à l'adresse `PAGE_OFFSET` (`PAGE_OFFSET` dépend de l'architecture, sur x86 il s'agit de 3 Go, c'est-à-dire de l'adresse linéaire `0xc0000000`). Quant à l'espace "noyau", il débute à `PAGE_OFFSET` et se termine à la fin de l'espace d'adressage.

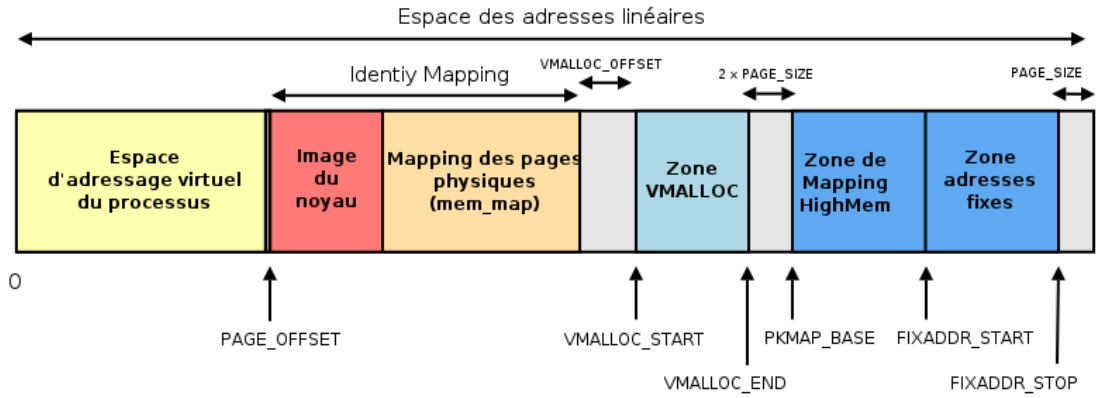

FIG. 4.4 – Espace des adresses linéaires sous Linux x86 32 bits

Revenons à présent sur l'espace d'adressage spécifique des processus.

4.1.2.1 L'espace "utilisateur"

Un programme est associé à un fichier binaire, lequel contient plusieurs parties (appelés *segments*), notamment le code du programme, ses données statiques, etc. Lorsque le noyau souhaite exécuter un programme, il crée un processus, charge en mémoire les différents segments du programme et modifie l'espace d'adressage virtuel du processus (au travers de la modification des tables de pages) pour que le processus accède aux segments du programme. La projection (appelée *mapping*) de ces segments en mémoire virtuelle forme ce que l'on appelle les régions virtuelles du processus ou encore VMA (*Virtual Memory Area*).

Pour visualiser ces régions de mémoire depuis l'espace "utilisateur", le noyau exporte ces informations dans `/proc/PID/maps`. Nous détaillons dans la suite les régions d'un processus typique, en commentant la sortie fournie par `cat /proc/self/maps`[4] sur une machine x86 32 bits.

```
08048000-0804c000 r-xp 00000000 03:06 7185885    /bin/cat
0804c000-0804e000 rw-p 00004000 03:06 7185885    /bin/cat
0804e000-0806f000 rw-p 0804e000 00:00 0          [heap]
...
b7e17000-b7f41000 r-xp 00000000 03:06 5735198    /lib/libc-2.6.1.so
b7f41000-b7f43000 r--p 0012a000 03:06 5735198    /lib/libc-2.6.1.so
b7f43000-b7f44000 rw-p 0012c000 03:06 5735198    /lib/libc-2.6.1.so
...
b7f78000-b7f79000 r-xp b7f78000 00:00 0          [vdso]
b7f79000-b7f93000 r-xp 00000000 03:06 4958218    /lib/ld-2.6.1.so
b7f93000-b7f94000 r--p 00019000 03:06 4958218    /lib/ld-2.6.1.so
b7f94000-b7f95000 rw-p 0001a000 03:06 4958218    /lib/ld-2.6.1.so
bf801000-bf817000 rw-p bffea000 00:00 0          [stack]
```

Chaque ligne identifie une région de mémoire. Ainsi la première région qui s'étend de l'adresse linéaire `0x08048000` à `0x0804c000`, comprend deux pages de mémoire et contient le code du programme `/bin/cat` (segment `.text`), identifié par l'*inode* `7185885` sur le périphérique bloc `03 :06`. L'écriture dans cette région n'est pas autorisée (`r-xp` ; le `p` signifie qu'il s'agit d'une projection privée (c'est-à-dire qu'elle n'est pas partagée par les fils du processus) et entraîne une faute de page si le processus essaie d'y écrire. La région suivante correspond au segment des données globales et initialisées d'un programme (segment `.data`). Si le programme contient des données globales non initialisées, alors une autre région est créée juste après `.text`, que l'on nomme `.bss`. La région débutant à `0x0804e000` et identifiée par le nom `[heap]` (le *tas* du processus), est la région où sont stockées les données dynamiquement allouées. N'étant pas reliée à un *inode*, la valeur 0 apparaît au niveau de la colonne correspondante (cette région est alors dite *anonyme*). La taille de cette région ne pouvant être déterminée à l'avance, elle peut augmenter au cours de la vie du processus en fonction des besoins. C'est la *bibliothèque C* qui s'occupe de cela. On trouve à la fin de l'espace d'adressage de l'utilisateur, la région anonyme identifiée par `[stack]` (la pile du processus) et débutant dans notre exemple à l'adresse `0xbf801000`.

[4] L'entrée `self` correspond au processus en cours.

Les bibliothèques partagées sont projetées à la suite du programme dans l'espace d'adressage du processus. Dans notre exemple il s'agit seulement de la bibliothèque C. On retrouve la section de code, la section des données initialisées et une section entre ces deux qui correspond non pas à la section `.bss` mais aux données en lecture seule (segment `.rodata` dont les permissions sont `r-p`). Vient ensuite la région anonyme VDSO (*Virtual Dynamic Shared Object*) par laquelle passe le déclenchement des appels-système. Finalement, les trois régions qui débutent à partir de `0xb7f79000` correspondent à l'éditeur de liens (appelé *linker*) *ld* qui s'occupe de la résolution dynamique des symboles non encore résolus et du chargement dynamique des bibliothèques partagées.

Notons que l'adresse de début des régions associées aux bibliothèques partagées (`libc`, etc.), à la pile et au VDSO change à chaque exécution dans les versions actuelles du noyau. Ces adresses sont calculées avec un certain aléa afin de renforcer la sécurité du système (en rendant plus difficile l'exploitation de failles où la connaissance de la position de ces régions est primordiale).

4.1.2.2 L'espace "noyau"

L'espace "noyau" est commun à tous les processus par soucis de performance. En effet, lorsque un processus s'exécute en mode "noyau" lors de l'exécution d'un appel-système par exemple, l'espace d'adressage du processus appelant est conservé (c'est-à-dire que le registre `cr3` n'est pas modifié) d'où le gain de temps lors du passage en mode "noyau". Aussi, le fait que le registre `cr3` ne soit pas chargé avec une nouvelle valeur implique la conservation du TLB[5] (lorsque le registre `cr3` est chargé, le TLB est automatiquement vidé), et donc un gain potentiel de performance (dans le cas où un autre processus ne s'est pas exécuté entre temps) pour le processus. Les *threads* noyau, n'accédant qu'à l'espace "noyau", utilisent l'espace d'adressage du processus s'étant exécuté avant eux, pour les mêmes raisons de performance qui viennent d'être expliquées.

Décrivons brièvement la composition de l'espace "noyau" (*cf.* figure 4.4). Le code du noyau et ses données sont stockés en tête de l'espace "noyau", c'est-à-dire à partir de `PAGE_OFFSET`. Ensuite, prend place la zone d'*identity mapping*. Il s'agit d'une zone associant les adresses physiques et les adresses linéaires de façon continue. Ainsi une adresse linéaire du type `PAGE_OFFSET + x` dans cette zone correspond à l'adresse physique `x`. Pour les architectures de type x86 32 bits, cette zone fait au plus 896 Mo, car l'espace d'adressage du noyau est limité à 1 Go du fait de l'adressage sur seulement 32 bits. Ainsi lorsque le système dispose de plus de RAM que 896 Mo, une région supplémentaire est créée, la *mémoire haute* (*Highmem*). Elle débute à `PKMAP_BASE` et consiste en une "fenêtre glissante" sur les adresses physiques allant au delà des 896 Mo. Cette "fenêtre" étant évidemment de taille limitée, les projections effectuées sont temporaires. La zone des adresses fixes, correspond à des projections préétablies qui sont nécessaires au fonctionnement de certains périphériques (comme l'ACPI qui emploie toujours les mêmes adresses de l'espace linéaire). Finalement la zone `VMALLOC` est la zone employée par l'allocateur de mémoire non-contiguë du noyau. À la différence de la primitive `kmalloc()` qui effectue des allocations de mémoire dont les adresses physiques se suivent, `vmalloc()` effectue des allocations dont seules les adresses linéaires se suivent (les adresses physiques ne

[5] Le *Translation Lookaside Buffer*), est un tampon de mémoire très rapide qui est situé sur la puce du processeur et qui est employé par ce dernier pour conserver le résultat des traductions d'adresses linéaires en adresses physiques.

sont pas forcément contiguës). Ainsi, de plus gros tronçons de mémoire peuvent être alloués. Cet allocateur est ainsi employé par exemple pour le chargement des modules noyaux en mémoire.

Après cette brève description de l'espace d'adressage du noyau, nous présentons une autre vue (simplifiée) de cet espace avec la figure 4.5. Nous y faisons ressortir les modes de pagination employés, suivant les types de régions (code et données). Rappelons que cet espace est mis en correspondance avec l'espace de mémoire physique grâce à l'unité matérielle de pagination de la MMU (cf. section 1.4.5.2). Les attributs de page permettent à l'unité de pagination de gérer des droits d'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)[6]. 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 aussi être utilisées directement comme des entrées de pages de 4 Mo, si leur attribut de taille de page (*Page size*) est positionné à 1.

C'est d'ailleurs des pages de 4 Mo qui sont employées par le noyau Linux sur architecture x86 32 bits (et de 2 Mo sur x86 64 bits, dans le mode IA32e) pour des raisons de performances. En effet, les traductions d'adresses linéaires en adresses physiques sont coûteuses en temps CPU, car les tables de traductions sont stockées en mémoires. C'est pourquoi le *TLB* est employé par la CPU pour conserver le résultat de ces traductions. Cette mémoire étant de taille très réduite, elle ne peut conserver l'intégralité des traductions. L'utilisation de pages de 4 Mo par le noyau Linux permet ainsi de contourner ce problème, car une entrée de TLB pour stocker la traduction d'une adresse de page de 4 Mo est l'équivalent de mille entrées dans la TLB relativement à des pages de 4 Ko.

Dans le cas d'un noyau x86 64 bits, les pages employées pour le code et les données du noyau sont de 4 Ko, alors que pour la zone d'*identity mapping* il s'agit principalement de pages de 2 Mo (cette zone utilise aussi des pages de 4 Ko pour la plage des adresses correspondant au code et aux données du noyau).

La figure 4.5 présente une vue quelque peu incorrecte de l'espace d'adressage du noyau. En fait les pages "noyau" sont mixtes[7], c'est-à-dire qu'elles contiennent à la fois du code et des données. Néanmoins un travail proposé par Siarhei Liakh dans une série de deux *patchs* "noyau" modifie cette situation[8].

[6] Les attributs `R/W` (Read/Write) et `NX` (No eXecution) sont mis en valeur dans la figure, car il font l'objet d'une attention toute particulière dans la suite de ce chapitre.

[7] C'est le cas actuellement pour toutes les versions du noyau 2.6 jusqu'à la 2.6.31.

[8] Ces modifications devraient être disponibles dans une version future du noyau et activée au travers de l'option de configuration du noyau `CONFIG_DEBUG_RODATA`.

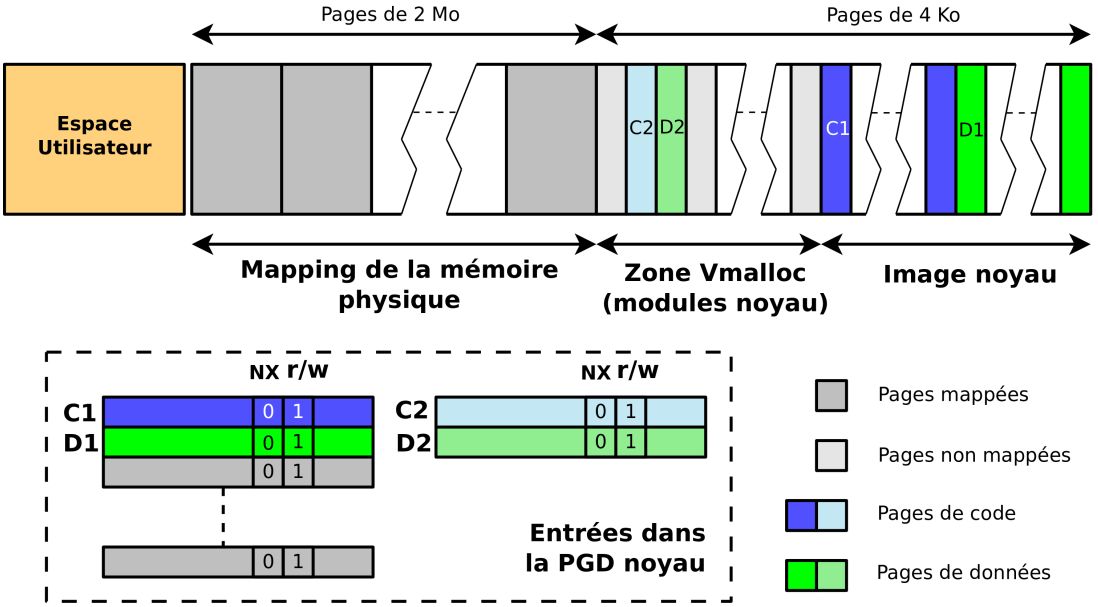

FIG. 4.5 – Organisation (simplifiée) de l'espace d'adressage du noyau sous Linux x86 64 bits

4.2 Le chargement de Hytux

Nous avons développé une preuve de concept[9] pour une cible Linux sur architecture Intel 64 supportant la technologie Intel VT-x [Intel Corporation, 2008e] et facultativement Intel VT-d [Intel Corporation, 2007]. Notre preuve de concept est baptisée *Hytux* et consiste en un *hyperviseur léger* qui dépend de ces technologies de virtualisation. Hytux emprunte l'approche de l'*hyperviseur léger* au projet *bluepill* [Rutkowska, 2006]. Ce dernier est un logiciel malveillant qui s'installe en tant qu'hyperviseur (sur un processeur AMD possédant le support matériel pour la virtualisation, fourni par la technologie AMD Pacifica [AMD, 2005]) et place un système d'exploitation Microsoft Windows dans une machine virtuelle afin de l'espionner. Dans notre cas, avec Hytux, il s'agit d'assurer la protection du noyau Linux au travers des fonctionnalités offertes notamment par la technologie Intel VT-x.

Hytux se présente sous la forme d'un module noyau qui, une fois ajouté au noyau Linux et exécuté, s'installe en tant qu'hyperviseur et fait en sorte que l'exécution du système d'exploitation se déroule au sein d'une machine virtuelle qu'il a préalablement configurée (*cf.* figure 4.6). Cette machine virtuelle est alors surveillée et contrôlée par l'hyperviseur afin de garantir certaines propriétés qui visent à assurer l'intégrité du noyau Linux. Ce contrôle est rendu possible par la configuration adéquate de la structure VMCS[10] afin d'intercepter certaines opérations effectuées par le système Linux.

Comme expliqué en introduction de ce chapitre, nous développons ici uniquement les mécanismes mis en œuvre pour assurer l'intégrité du noyau Linux, en considérant que Hytux commence à s'exécuter sur un système sûr. Cette prérogative peut être assurée par des mécanismes

[9] Elle n'est pas encore intégralement terminée.

[10] La création d'une telle structure en mémoire principale est spécifiée par la technologie Intel VT-x (*cf.* section 4.1.1.3).

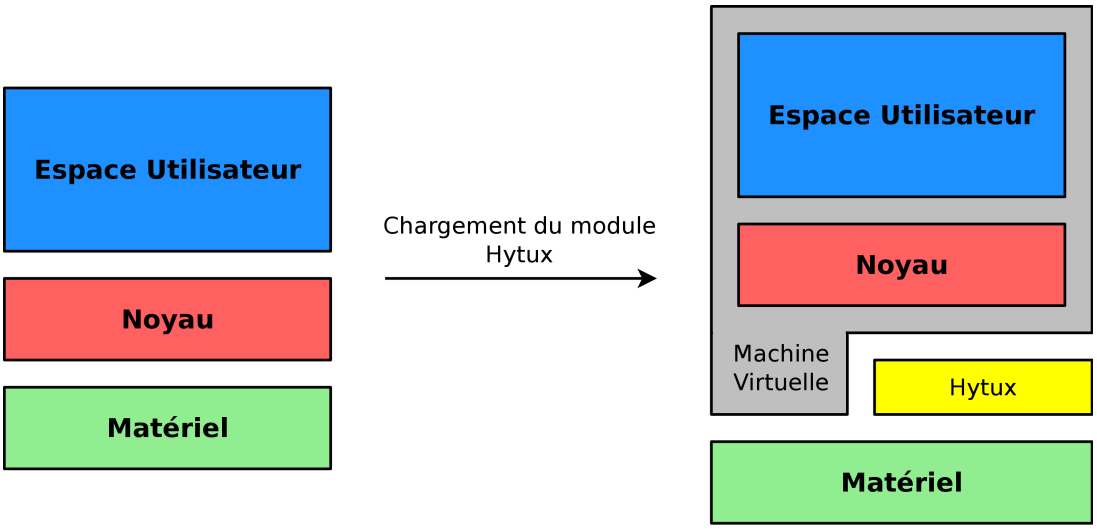

FIG. 4.6 – Hytux — un hyperviseur léger

mettant en œuvre le principe de chaîne de confiance. Ce principe est fondé d'une part sur la définition d'un socle de confiance (appelé le CRTM pour *Core Root of Trust Measurement*, *cf.* figure 4.7) qui est le plus petit ensemble matériel et logiciel auquel l'utilisateur du système fait initialement confiance et d'autre part sur la définition d'une opération transitive qui permet d'élargir le champ de confiance au reste du système.

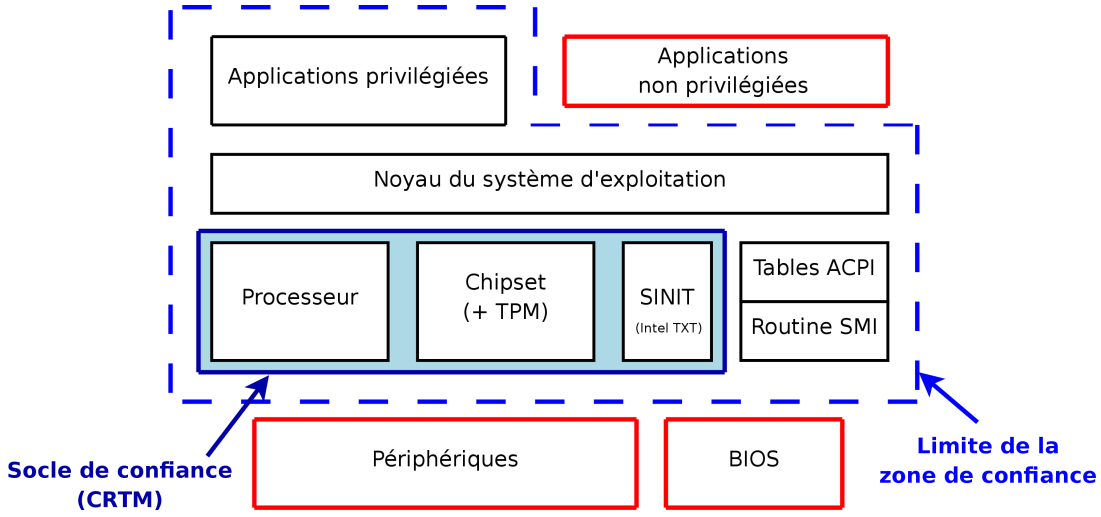

FIG. 4.7 – Exemple de socle de confiance d'une architecture x86 pourvue de Intel TXT

Cette opération est exécutée initialement par un composant du socle de confiance et, de proche en proche, par tous les composants qui forment alors la chaîne de confiance. Elle consiste habituellement en la mesure (le plus souvent il s'agit d'une empreinte cryptographique) d'un composant qui doit être exécuté, puis en la vérification de la correction de cette mesure et fina-

lement, le cas échéant, en l'exécution de ce composant (sinon le système interrompt son exécution) qui est alors considéré être de confiance. Cette approche a été standardisée par le TCG (le *Trusted Computing Group* est un consortium d'industriels qui s'intéresse à cette problématique) puis normalisée par l'ISO dans [ISO/IEC, 2009]. Elle a vu une mise en œuvre récente dans la technologie Intel TXT [Intel Corporation, 2008f], qui propose un support matériel à la définition d'une chaîne de confiance dynamique[11], appelée *Dynamic Root of Trust Measurement* (DRTM). Elle est qualifiée de dynamique car la chaîne de confiance peut être modifiée en cours d'exécution. L'objectif de cette technologie est alors de fournir à l'utilisateur la possibilité de passer en cours d'exécution d'un environnement non sûr (sur lequel s'exécute des logiciels potentiellement malveillants) à un environnement de confiance (où seul les logiciels qui s'exécute sont certifiés de confiance), appelé alors MLE (*Measured Launched Environment*), sur un système informatique pourvu de la technologie TXT[12]. Notamment, au travers de cette technologie, Intel souhaite enlever du socle de confiance, le BIOS de la machine. Intel a d'ailleurs implémenter le démonstrateur *tboot* [Cihula, 2007] qui exploite cette technologie et rend possible le chargement de l'environnement de virtualisation Xen en suivant le principe de DRTM. Ce démonstrateur fait l'objet d'une étude menée en complément de nos travaux de thèse, afin de l'adapter au chargement d'un environnement Linux muni de notre module Hytux. La figure 4.8 illustre le principe à mettre en place.

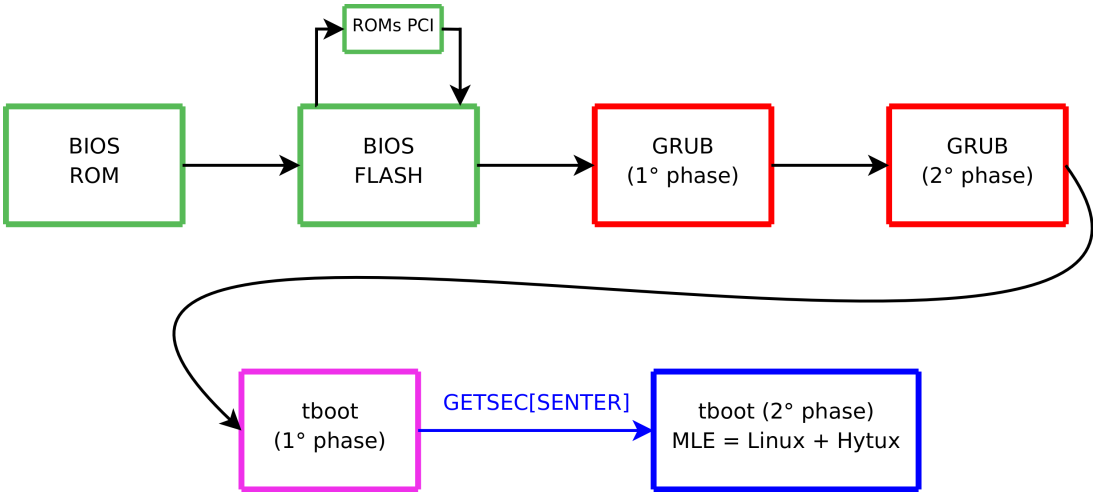

FIG. 4.8 – Le chargement d'un environnement sous le contrôle de Hytux

Au travers de cette approche que nous avons brièvement évoquée, il est alors possible de s'assurer que lorsque Hytux s'installe en tant qu'hyperviseur, il l'effectue toujours dans l'environnement pour lequel il a été prévu. Il est ainsi possible de s'assurer de l'innocuité de l'environnement dans lequel Hytux est chargé. Évidemment s'assurer de l'innocuité de l'environnement, ne garantit pas que cet environnement est exempt de vulnérabilités, et que l'exploitation de ses

[11] Cette technologie repose sur l'utilisation de la puce matérielle TPM (*Trusted Platform Module*) définie par le TCG.
[12] Un jeu d'instructions processeurs spécifiques, appelé SMX implémente la technologie TXT, notamment l'instruction `GETSEC[SENTER]`. Cette instruction est censée exécuter dans sa dernière étape un code signé appelé SINIT (qui est projeté en mémoire depuis un fichier fourni par Intel). C'est alors ce code qui mesure puis exécute le MLE.

failles ne puisse rendre l'environnement malveillant. C'est à ce niveau qu'entre alors en jeu Hytux. Son rôle est d'empêcher l'exploitation de potentielles vulnérabilités de cet environnement en tentant de maintenir certaines propriétés.

4.3 Protection des objets contraints par le noyau

Garantir l'intégrité du noyau au cours de son exécution, en partant de l'hypothèse qu'il est exempt de vulnérabilités peut se concevoir. En prenant l'hypothèse inverse, il devient nettement plus difficile de concevoir une solution qui *garantisse* cette intégrité.

Notre approche n'a pas été de trouver une solution complète à ce problème, car nous nous interrogeons toujours sur son existence. Notre approche est de fournir à l'utilisateur une certaine confiance sur l'intégrité d'un noyau en cours d'exécution, en assurant l'intégrité de certains éléments clés dont il dépend et sur lesquels nous pouvons agir. Il s'agit donc de limiter les atteintes possibles à l'intégrité de ce noyau au travers de la garantie de l'intégrité de certains des éléments dont il dépend et que nous pouvons contrôler (au travers de la virtualisation). Cette approche nous permet de considérer également que le noyau peut se mettre à agir de façon incorrecte dès lors que nos mécanismes de protection sont en activité. Ainsi, nous couvrons également la problématique de l'attaquant qui exploite une éventuelle vulnérabilité du noyau.

Les éléments dont dépend le noyau pour assurer sa fonction sont de différentes natures, ils peuvent être physiques (la CPU, ses différents registres, etc.) comme logique (son code, les tables de pages résidant en mémoire, la table des interruptions, etc.). Certains lui sont internes (la table des interruptions, son code, etc.), et d'autres lui sont externes (les différents registres du processeur, ceux du contrôleur mémoire, etc.). Ces éléments peuvent en général être regroupés par *fonction*, comme par exemple les éléments impliqués dans la gestion des interruptions (le registre `idtr`, la table des interruptions en mémoire principale, le code des gestionnaires d'interruptions). Lorsqu'ils ne le sont pas, c'est qu'ils interviennent dans l'accomplissement de plusieurs de ces fonctions ou de toutes ces fonctions (comme l'ensemble du code du noyau) et font généralement partis de la structure du noyau.

L'identification des fonctions essentielles assurées par le noyau nous permet de déterminer quels sont les éléments dont l'intégrité a le plus d'importance. Nous entendons par le terme "importance", qu'une atteinte à l'intégrité de l'un de ces éléments provoque une *importante* nuisance dans l'accomplissement des fonctions du noyau.

Le problème est alors de garantir l'intégrité de ces éléments. De par la variété de ces éléments, ce problème est à résoudre au cas par cas, au mieux en fonction du type des éléments. Toutefois, à un niveau d'abstraction supérieur, tous ces éléments peuvent être considérés comme des objets répondant à des contraintes. Ainsi, notre approche a été d'identifier les contraintes s'appliquant à ces éléments, et plus particulièrement d'identifier celles qui importent au noyau.

Autrement dit, l'idée est de préserver les éléments qui sont considérées contraints par le noyau. Nous appelons ces éléments particuliers que nous souhaitons protéger, des *Objets Contraints par le Noyau* (*Kernel-Constrained Object*). Nous les définissons comme suit.

Définition 5. *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, 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 boguée ou qu'une faille de conception existe.

D'après le chapitre 1, un KCO peut dès lors être un objet appartenant à la mémoire principale (dans les régions du noyau ou à l'extérieur), ou bien à l'état des composants matériels dont dépend le noyau pour son exécution (que nous avons appelé mémoire du support de contrôle), ou enfin à un composant matériel, avec qui le noyau communique mais dont il ne dépend pas directement pour son exécution (c'est-à-dire un périphérique).

Dans la suite de cette section nous expliquons notre façon de procéder pour préserver l'intégrité de différents KCO que nous avons identifiés. Notons que l'état dans lequel se trouve les KCO lorsque notre hyperviseur (Hytux) est initialisé est supposé être sûr[13]. À partir de cet instant, Hytux essaie d'empêcher que les KCO ne soient altérés.

4.3.1 Identification des différentes fonctions d'un noyau

Un système d'exploitation permet à l'utilisateur de se servir du matériel plus simplement que s'il devait programmer la machine réelle. Il fournit pour cela un certain nombre d'abstractions (processus, fichiers, etc.) qui sont les éléments de construction des applications. À cela se rajoute une interface de dialogue entre les applications et le système d'exploitation (cf. section 1.3). Afin que la fourniture de ce service soit possible, un certain nombre de fonctions doivent être remplies par le système d'exploitation et notamment par son noyau. Les différentes fonctions que le noyau doit accomplir se retrouvent dans ces trois catégories :

- *Assurer la configuration et le bon fonctionnement du matériel (catégorie 1) :*
  Dans cette catégorie se trouve notamment le traitement des interruptions matérielles, la configuration de la CPU, du *chipset*, de la mémoire principale, et des différents périphériques.
- *Fournir des abstractions qui facilitent la programmation des applications (catégorie 2) :*
  Cette catégorie regroupe essentiellement les deux fonctions suivantes : la définition de services pour l'utilisateur (les appels-système), et la mise en œuvre d'une interface d'accès à ces services.
- *Assurer le partage des ressources matérielles pour toutes les applications qui s'y exécutent (catégorie 3) :*
  Dans cette catégorie se retrouvent les fonctions de gestion des ressources d'exécution (CPU, GPU, etc.), des ressources de mémoire (RAM, disque dur, etc.) et des ressources de communication (affichage, réseau, etc.).

Dans la suite nous traitons le cas de la préservation de l'intégrité de certaines fonctions de la catégorie 1 et des fonctions de la catégorie 2. La préservation de l'intégrité de ces fonctions est rendue possible par la préservation de contraintes propres à ces fonctions[14] (section 4.3.2) mais également par la préservation de contraintes fortes sur la structure du noyau (section 4.3.3).

La garantie de l'intégrité par la préservation de contraintes est rendue possible sur un certaines des fonctions essentielles du noyau. Il découle de la mise en œuvre de ces mécanismes,

[13] Cette hypothèse est justifiée par les mécanismes qui peuvent être mis en place et dont nous discutons dans la section 4.2.

[14] Ces contraintes sont issues d'objets qui appartiennent soit à l'état, soit à la structure du noyau.

une assurance *partielle* de l'intégrité du noyau au cours de son exécution. L'évaluation de la couverture de notre solution vis-à-vis des classes d'actions malveillantes identifiées dans la section 1.5 est développée à la fin de ce chapitre dans la section 4.5.

4.3.2 La préservation de l'intégrité du noyau : préservation de contraintes liées à ses fonctions

4.3.2.1 La chaîne de traitement des interruptions, composée de trois KCO

Pour bien comprendre ce concept, prenons l'exemple de la préservation du traitement des interruptions par le noyau. Pour préserver ce traitement, il est nécessaire de préserver l'intégrité de tous les éléments participant au traitement, et cela peut être accompli par la préservation de contraintes simples. Les objets associés au traitement des interruptions sont : le registre `idtr`, la table des interruptions (IDT) pointée par ce registre, et les différents gestionnaires d'interruption (c'est-à-dire des sections de code) référencés par cette table. Ces objets peuvent être considérés comme des KCO. Nous justifions cela par la suite.

Commençons par traiter le cas du registre `idtr`. Ce registre est un KCO car 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* (c'est-à-dire 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 registre `idtr` ne peut plus être modifié. L'émulation de l'instruction `lidt` est aisément accomplie via l'utilisation de Intel VT-x. En effet, si on positionne à 1 le champ `Descriptor-table_exiting` de la VMCS, tout accès au registre `idtr` est interrompu par le processeur qui déclenche alors une VM-exit[15]. Le contrôle de l'exécution est alors donné à l'hyperviseur, lequel peut alors récupérer le contexte de cet accès (via la lecture de la région `VM-exit_info_fields` de la VMCS) et décider des modifications à entreprendre sur la machine virtuelle et sur la machine physique[16] avant de rendre la main au noyau.

La préservation de ce que nous appelons le KCO `idtr` n'est cependant pas encore accomplie. En effet, ce registre a pour nature d'être la référence employée par le processeur pour accéder à la table des interruptions (IDT), et l'adresse qu'il contient est une adresse linéaire. Ainsi, la préservation de ce KCO nécessite la préservation de la traduction de cette adresse linéaire en l'adresse physique correspondant à l'emplacement de l'IDT. Il est donc nécessaire de vérifier qu'à chaque modification des tables de pages, les entrées qui correspondent à l'adresse

[15] Cette fonctionnalité n'est cependant disponible que pour une partie des processeurs qui supportent Intel VT-x (à partir de l'architecture Nehalem).

[16] Dans notre cas, il s'agit uniquement de la mise à jour du compteur d'instruction de la machine virtuelle afin de tout simplement sauter l'instruction qui a causé la VM-exit.

linéaire stockée dans le registre `idtr` ne sont pas modifiées. Pour effectuer cette vérification, la technique mise en œuvre est similaire à celle qui est présentée dans la section 4.3.3. Cette dernière traite de la préservation de l'agencement de l'espace de mémoire du noyau.

Pour préserver le traitement des interruptions il est également nécessaire de protéger l'IDT comme nous l'avons expliqué. Cette table n'a pas de raisons d'être modifiée en cours d'exécution. Ainsi tout accès en écriture à cette table doit être empêché. La section 4.3.3 explique justement comment parvenir à ce résultat. De même, les gestionnaires d'interruptions, qui correspondent à des sections de code référencées dans cette table, n'ont pas à être modifiés durant l'exécution du système. Ainsi, ces gestionnaires doivent être protégés d'une quelconque modification, ce qui est également couvert par la section 4.3.3.

4.3.2.2 La configuration de la CPU et du *chipset*

Le noyau doit assurer un bon fonctionnement du matériel. Pour cela il configure notamment la CPU et le *chipset*, au travers de l'initialisation de certains de leurs registres. Une partie de ces registres n'a alors plus aucune raison d'être modifiée (comme nous l'avons vu pour le cas du registre `idtr`), si ce n'est pour l'accomplissement d'actions malveillantes. La valeur que contient ces derniers registres peut alors être asservie à une contrainte. Afin de préserver cette contrainte nous employons les mécanismes déployés par la technologie matérielle de virtualisation VT-x. Nous traitons ci-dessous quelques exemples de contraintes à préserver sur des registres, pour éviter que le noyau ne puisse effectuer certaines actions incorrectes (c'est-à-dire dont les conséquences pour le système seraient mauvaises) en cours d'exécution, actions pouvant être notamment le résultat d'actions malveillantes.

– **Les registres de contrôle :**
  Les registres de contrôle de la CPU sont le moyen de configurer des aspects très importants de la CPU. Certains bits des registres `cr0` et `cr4` doivent rester dans un état spécifique si l'on souhaite garantir le bon fonctionnement du système, mais également protéger notre hyperviseur d'éventuelles actions incorrectes du noyau. Il est important de se rappeler que notre hyperviseur s'exécute dans le même environnement (CPU, chipset, etc.) que le noyau. Il faut donc, au delà de sa fonction de protection du noyau, qu'il se protège lui-même de ce noyau. Cette problématique inclut également la protection de la mémoire de l'hyperviseur. Nous traitons cela dans la section 4.4.
  Afin de protéger ces registres de contrôle, VT-x fournit des champs de configuration spécifique dans la VMCS. Deux champs de "masque" (`CR0_guest/host_mask` et `CR4_guest/host_mask`) permettent de spécifier les bits qui sont sous la maîtrise du système hôte (c'est-à-dire de l'hyperviseur). Le système exécuté dans la machine virtuelle (l'invité) ne peut alors plus les modifier. Lorsque l'invité tente de lire la valeur d'un de ces registres (`cr0` ou `cr4`) le processeur renvoie des valeurs de bits qui proviennent de deux sources différentes. Si les bits sont "masqués" la valeur renvoyée provient d'un champ de la VMCS que l'hyperviseur a pris soin de remplir (`CR0_read_shadow` et `CR4_read_shadow`) et si les bits ne sont pas masqués leur valeur provient directement du registre de contrôle. Dans l'éventualité où l'invité tente de modifier un bit appartenant à l'hyperviseur (c'est-à-dire un bit masqué), le processeur déclenche une VM-exit. L'hyperviseur prend alors la main pour gérer la situation avant de rendre l'exécution à la machine virtuelle. Un hyperviseur classique doit faire en sorte d'émuler le comporte-

ment souhaité par les actions issues des systèmes invités qu'il contrôle. Les mécanismes d'interceptions fournit par VT-x sont justement élaborés pour rendre cette émulation possible et facile à réaliser. Il faut comprendre que l'objectif d'un hyperviseur classique est de gérer plusieurs machines virtuelles qui nécessitent potentiellement des états de configurations différents et donc des valeurs de `cr0` et `cr4` différentes. Dans notre cas, ce n'est absolument pas la raison qui nous motive à intercepter les accès à ces registres de contrôle. Nous souhaitons juste préserver des contraintes sur ces registres, pour empêcher les conséquences désastreuses qu'impliqueraient des comportements incorrects (notamment malveillants) du système durant son exécution.

Ainsi, nous protégeons notamment de cette façon les bits suivants :

- Les bits `PG` (*PaGing*) et `PE` (*Protection Enable*) du registre `cr0` sont conservés à la valeur 1, afin que l'unité de pagination de la MMU soit active. Cette condition est nécessaire à l'utilisation de VT-x, et sans elle, il ne serait d'ailleurs pas possible d'assurer la virtualisation de la mémoire.
- Les bits `VMXE` (*VMX-Enable*) et `SMXE` (*SMX-Enable*) du registre `cr4` sont conservés à la valeur 1, afin que respectivement soit active la technologie VT-x[17] et la technologie TXT[18].

– **Le registre des fonctionnalités étendues de la CPU :**
Le registre `MSR_EFER` (MSR *Extended Feature Enable*) est un MSR (*Model-Specific Register*) de la CPU qui permet d'activer des fonctionnalités optionnelles. Notre hyperviseur emploie notamment la protection des pages de mémoire contre l'exécution de leur contenu[19]. Cette fonctionnalité est contrôlée par le bit NXE *Non-eXecution Enable* de ce MSR. Afin d'empêcher la désactivation de cette fonctionnalité, notre hyperviseur intercepte tout accès à ce MSR et ignore toute demande de modification de ce bit. De même, les bits `LMA` et `LME` servent à l'activation du mode IA32e (*cf.* section 1.4.1) du processeur (pour les architectures Intel 64). En empêchant toute modification de ces bits par le système contrôlé, nous évitons le passage de la CPU notamment dans le mode réel (pour lequel les protections de la mémoire ne sont plus disponibles), ou encore le mode protégé 32 bits. Sur une plateforme Intel 64 qui exécute un Linux 64 bits, ces deux modes n'ont pas besoin d'être employés[20]. De plus, la préservation de la CPU dans ce mode[21] évite un contrôle spécifique des autres modes par notre hyperviseur, le rendant ainsi plus simple.

– **Les tables de l'ACPI :**
La gestion de l'alimentation et de la configuration d'une plate-forme x86 nécessite la manipulation de registres situés dans le *chipset* ou au sein de périphériques (accessibles pour certains en MMIO, pour d'autres en PIO, et dans les autres cas via le mécanisme de confi-

[17] Il s'agit ici de protéger notre hyperviseur contre une malencontreuses désactivation de la part du système contrôlé.

[18] Dans notre cas cette technologie n'est pas strictement nécessaire en cours d'exécution du système, car nous souhaitons l'employer que lors de l'initialisation du système. Néanmoins il reste nécessaire d'intercepter toute tentative d'utilisation de cette technologie en dehors de notre hyperviseur (lorsque VMXE est à 1, l'exécution de `getsec` l'instruction de TXT, déclenche automatiquement une VM-exit en mode VMX non-root). Un attaquant pourrait éventuellement l'utiliser de façon à s'évader de notre environnement contrôlé.

[19] Cette protection est nécessaire pour assurer l'intégrité de la structure du noyau qui est contrôlé. La section 4.3.3 explique la façon dont est employée cette fonctionnalité.

[20] À l'exception du mode réel à l'initialisation du système.

[21] En fait, au cours de l'exécution du système la CPU passe sporadiquement dans le mode SMM. Un contrôle peut être également mis en place pour ce mode grâce à Intel VT-x. Nous revenons sur cela par la suite.

guration PCI). L'ACPI (*Advanced Configuration and Power Interface*) est un standard qui a été créé afin de permettre au système d'exploitation de gérer notamment la gestion de l'alimentation de la plate-forme indépendamment de ses spécificités matérielles. Pour cela, des méthodes "virtuelles" ont été définies dont l'implémentation (qui est spécifique à la plate-forme) incombe aux développeurs du BIOS de la plate-forme. Ces méthodes sont alors fournies aux systèmes d'exploitation par le BIOS qui lors de son initialisation, les charge en mémoire dans une région prévue à cet effet. L'accès à ces méthodes par le système d'exploitation s'effectue dans cette région, au travers de tables dites tables ACPI. Parmi ces tables se trouvent notamment la RSDT (*Root System Description Table*, table racine qui référence toutes les autres tables) et la DSDT (*Differentiated System Description Table*, qui contient les méthodes de gestion de l'alimentation, définit les registres ACPI et décrit comment les modifier). Ces tables contiennent des données sensibles dont la modification peut altérer l'intégrité du noyau. Loïc Duflot a d'ailleurs mis en œuvre une attaque fondée sur l'ajout d'une méthode malveillante dans la DSDT entraînant l'exécution d'un *shell root* depuis un compte utilisateur simple [Duflot et Levillain, 2009]. Cette région de mémoire associée à l'ACPI est alors considérée par notre approche comme un KCO. Il faut donc que notre hyperviseur interdise toute modification des pages de mémoire associées à cette région. La technique mis en œuvre pour cela est expliquée dans la section 4.3.3 qui traite de la préservation de l'agencement de l'espace de mémoire du noyau.

– **La configuration du cache de la CPU vis-à-vis de l'espace d'adressage :**
La SMRAM est la région de mémoire mise en place par le BIOS, qui contient la routine de traitement de la SMI (*cf.* section 1.4.1). L'accès à cette région est contrôlé par le MCH au travers de la configuration de certains bits des registres `SMRAMC` (*SMRAM Control*) et `ESMRAMC` (*Extended SMRAM Control*) du MCH[22]. Si le bit `D_OPEN` est à 0, alors l'accès à la SMRAM n'est possible que depuis le mode SMM. S'il est à 1, la SMRAM est accessible depuis n'importe quel mode de la CPU. Le BIOS est censé, après avoir écrit la routine SMI dans cette région, positionner le bit `D_LCK` à 1, ce qui provoque la mise à 0 de `D_OPEN` et empêche toute modification des registres `SMRAMC` et `ESMRAMC` avant le prochain redémarrage de la machine[23].
Cela est censé assurer l'intégrité de la routine SMI. Cependant, comme l'a montré Loïc Duflot [Duflot et Levillain, 2009], si cette routine se retrouve dans le cache de la CPU, il est possible de la modifier. Il faut tout d'abord savoir que les registres MTRR (*Memory Type Range Registers*) du processeur servent à configurer le comportement de son cache en fonction des plages de l'espace d'adressage de la mémoire auxquelles il accède (*cf.* section 1.4.3). Normalement, la plage comportant cette SMRAM est configurée au travers de ces registres de façon à ne pas être "cachée" par le processeur. Cependant, il est possible depuis le mode "noyau" de configurer cette région en *write-back*. Ainsi lorsque le processeur accède à la routine SMI, celle-ci se retrouve chargée dans son cache. Il est alors possible d'y accéder en lecture comme en écriture depuis le noyau, car le contrôle d'accès n'est assuré que par le MCH et que l'accès n'atteint ici que le cache du proces-

[22] Ces registres sont accessibles depuis le mécanisme de configuration PCI (*cf.* section 1.5.3).
[23] Hytux vérifie tout de même que le bit `D_LCK` est à 1 lors de son chargement, sinon il le positionne à 1.

seur[24].
Il est donc important d'empêcher que la région de la SMRAM ne soit mise en cache. Pour cela notre hyperviseur intercepte les accès aux MTRR[25].

4.3.2.3 La chaîne de traitement des appels-système

Les appels-système sont employés par les applications (plus généralement par l'espace "utilisateur") pour accéder aux services du noyau. Pour effectuer un appel-système, un processus utilisateur charge les registres généraux de la CPU pour préciser l'identifiant de l'appel-système qu'il souhaite exécuter ainsi que ces paramètres. Ensuite, il effectue l'appel au système au travers d'un mécanisme standard. Suivant les processeurs de la famille x86, trois mécanismes sont supportés. Le plus ancien et le moins performant est l'utilisation de l'interruption logicielle `0x80`. Le déclenchement depuis le processus de cette interruption (via l'instruction `int 0x80`) provoque le passage de la CPU en `ring 0`, le chargement des segments `CS`, `DS` et `SS` du noyau[26], et l'exécution du gestionnaire de l'interruption `0x80`. Ce gestionnaire transfère alors le contrôle au gestionnaire des appels-système qui exécute alors le service demandé.

Sur les processeurs x86 actuels des instructions spécifiques sont mises en œuvre pour répondre aux besoins des appels-système (et sont plus rapides que l'utilisation des interruptions). Pour les architectures 32 bits, il s'agit uniquement des instructions `sysenter` et `sysexit`. La première déclenche le passage en `ring 0` à un point spécifique du noyau alors que la seconde rend la main au processus appelant (qui s'exécute en `ring 3`). Le comportement de ces instructions doit être configuré au travers des MSR `SYSENTER_EIP`, `SYSENTER_ESP` et `SYSENTER_CS`. Afin d'employer ce mécanisme, le noyau met en place dans l'espace d'adressage des processus, une région de code, appelée VDSO (*Virtual Dynamic Shared Object*), laquelle contient notamment l'instruction `sysenter`[27]. Un processus qui souhaite effectuer un appel-système doit alors transférer le contrôle à cette région (via l'instruction `call __kernel_vsyscall`) après l'étape de chargement des registres généraux. Une fois l'instruction `sysenter` exécutée le contrôle est donné au gestionnaire des appels-système, lequel lance le service et retourne dans l'espace "utilisateur" via `sysexit` une fois le traitement accompli. Le retour se fait dans la région de la VDSO qui transfère ensuite le contrôle au code du processus qui avait effectué le `call __kernel_vsyscall`.

Pour les architectures 64 bits, le mécanisme précédent est supporté mais il existe un mécanisme similaire encore plus performant fondé sur l'utilisation des instructions `syscall` et `sysret`. Globalement, il s'agit d'employer ces instructions en lieu et place de leur homologue `sysenter` et `sysexit`. Pour que ces instructions soient disponibles sur un processeur com-

[24] La valeur du registre interne `SMBASE` de la CPU (stockant l'adresse mémoire où se trouve la SMRAM) est sauvegardé dans la SMRAM lorsque la CPU passe en mode SMM, pour ensuite être restaurée à la sortie de ce mode. Il est ainsi possible de positionner de façon permanente (jusqu'au prochain redémarrage de la machine) une nouvelle routine SMI dans une région de mémoire choisie par l'attaquant et n'étant pas sous le contrôle du MCH.

[25] La VMCS permet de spécifier une liste de MSR dont l'accès depuis le mode VMX non-root, déclenche une VM-exit.

[26] Pour plus de précision sur la segmentation se reporter à la section 1.4.5.

[27] Contrairement au mécanisme des interruptions qui range sur la pile l'adresse de retour lors de l'appel à l'instruction `int`, `sysenter` n'effectue pas cela. Pour retourner dans l'espace "utilisateur" une autre méthode est alors employée, `sysexit` utilise comme adresse de retour celle spécifiée dans le registre `edx`. C'est pourquoi le noyau utilise une région unique d'appel (la VDSO) dont il connaît l'emplacement.

patible, le noyau doit cependant activer leur support au travers du registre MSR `EFER` (le bit `SCE` *SysCall Enable* doit être positionné à 1). La configuration de leur comportement se fait alors au travers des registres MSR `STAR`, `LSTAR` et `FMASK`.

Pour préserver l'intégrité de cette chaîne de traitement des appels-système, il faut alors préserver l'intégrité des différents KCO qui la composent. Dans le cas du mécanisme fondé sur l'interruption `0x80`, l'intégrité est assurée par la technique mis en place dans la section 4.3.2.1. Pour les deux autres mécanismes, il s'agit d'agir de la même manière que pour la préservation de l'intégrité de la chaîne de traitement des interruptions. Ainsi, en tête de la chaîne à la place du KCO `idtr`, nous avons les KCO associé aux MSR `SYSENTER_EIP`, `SYSENTER_ESP`, `SYSENTER_CS`, `STAR`, `LSTAR` et `FMASK`. Les adresses qu'ils contiennent sont des adresses linéaires, il faut donc user des mêmes techniques que pour le KCO `idtr`, afin d'éviter une manipulation malveillante des tables de pages visant à modifier le point d'entrée du noyau pour la gestion des appels-système. Enfin, un dernier KCO est à préserver avec ces deux mécanismes : la VDSO. La prévention de toute modification de cette région de mémoire est analogue au cas des régions de code du noyau, traité dans la section 4.3.3[28].

4.3.3 La préservation de l'intégrité de la structure du noyau : un espace d'adressage contraint

La préservation de l'intégrité de la structure du noyau répond au actions malveillantes de la classe 1.1. Pour traiter plus particulièrement les classes 1.1.2 et 1.1.3, il convient de faire respecter les contraintes suivantes sur l'espace d'adressage du noyau. Les régions de code doivent être exécutables mais non modifiables, et les régions de données ne doivent pas être exécutables. Il est important de noter également que les régions de données n'ayant pas besoin d'être modifiées en cours d'exécution doivent également être protégées de toute modification. Il s'agit alors de préserver dans une certaine mesure l'état du noyau en empêchant le succès de certaines des actions de la classe 1.2. L'identification de ces régions est effectuée au travers de l'analyse des fonctions du noyau, telle que nous l'avons menée au cours de la section 4.3.2 pour quelques unes d'entre elles. Bien qu'il ne s'agisse pas de contraintes liées à la structure du noyau, nous les développons dans cette section, car les techniques de préservation de l'intégrité employées y sont étroitement liées[29].

Avant de pouvoir préserver ces contraintes, il est tout d'abord nécessaire que le code et les données du noyau se situe en mémoire dans des pages différentes. Or, par défaut le noyau Linux emploie des pages de 2 Mo (sur une architecture x86 64 bits) qui contiennent à la fois son code et ses données. Afin de changer ce comportement, des modifications dans le noyau sont nécessaires. Un *patch* (proposé par Siarhei Liakh [Liakh, 2009]) répond justement à cette problématique, et place dans des pages séparées le code et les données du noyau[30].

Ces conditions satisfaites, il faut alors effectuer, pour préserver ces contraintes, une modification des attributs des entrées des tables de pages (cf. section 1.4.5.2). Notons que l'interdiction

[28] En réalité, une option du noyau permet de placer cette VDSO de façon aléatoire dans l'espace d'adressage pour chaque processus. Dans ce cas, quelques précautions sont alors nécessaires pour garantir son intégrité.

[29] Toutes les techniques développées dans cette section mettent en œuvre une certaine virtualisation de la mémoire.

[30] Il reste toutefois nécessaire de placer séparément des pages de données, celles qui ne doivent pas être modifiées.

d'exécution des pages de données associées aux piles "noyau" des processus ne pose pas de problèmes contrairement aux piles "utilisateur". En effet, du code est parfois légitimement injectée 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. Heureusement, dans le cas de Linux, ces problèmes ne concernent pas la pile "noyau"[31].

Les modifications effectuées sur les attributs des tables de pages du noyau, ne sont cependant pas suffisantes pour préserver les contraintes. En effet, une action malveillante en mode "noyau" pourrait désactiver cette protection en changeant tout d'abord les attributs des pages d'une région de données qui contiendrait du code malveillant et ensuite exécuter cette région. Il est donc nécessaire d'empêcher tout changement des attributs de pages qui résulterait en la modification des contraintes que l'on souhaite préserver.

Pour cela, notre hyperviseur virtualise en quelque sorte la mémoire du noyau, afin de préserver ces contraintes vis-à-vis d'actions incorrectes issues par le noyau (ou de tout autre code s'exécutant en `ring 0`). Le contrôle de notre hyperviseur est effectué notamment sur les tables de pages du noyau, c'est-à-dire au niveau de la pagination. Mais il est également nécessaire d'effectuer un contrôle au niveau de la segmentation car la structure du noyau en dépend.

Dans la suite de cette section, nous commençons par examiner le contrôle que nous apportons sur la pagination en prenant en compte le cas du chargement dynamique de module "noyau"[32], pour ensuite traiter le cas de la préservation de l'intégrité des segments du noyau.

4.3.3.1 La préservation des contraintes au niveau de la pagination

Afin de se protéger contre les actions malveillantes des classes 1.1.1 et 1.1.2, nous avons vu en introduction de cette section 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, l'attribut `R/W` (Read/Write) n'est pas positionné ; pour une page qui contient des données qui peuvent être modifiées, les attributs `NX` (No eXecution) et `R/W` sont positionnés ; enfin, pour une page de données en lecture seule l'attribut `NX` est positionné mais pas l'attribut `R/W`. Comme présenté dans la section 4.1.2, la première partie de l'espace d'adressage du noyau est remplie de pages de 2 Mo ou 4 Mo (suivant que l'on soit en mode 64 bits ou 32 bits) 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 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 de cette zone n'est valide sauf celles qui contiennent les modules déjà chargés. L'infrastructure de chargement des modules noyau doit alors être quelque peu modifiée. La modification que nous proposons agit

[31] 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.

[32] Notons que la préservation de la structure du noyau vis-à-vis des actions de la classe 1.1.1 exige un contrôle des modules ajoutés dynamiquement au noyau. Nous revenons sur ce point à la fin de cette section.

notamment sur la primitive noyau `vmalloc()` qui est employée pour l'allocation de mémoire des LKM. Nous expliquons l'approche à mettre en œuvre dans la section 4.3.3.2. Au final, l'espace d'adressage est contraint tel que l'illustre la figure 4.9.

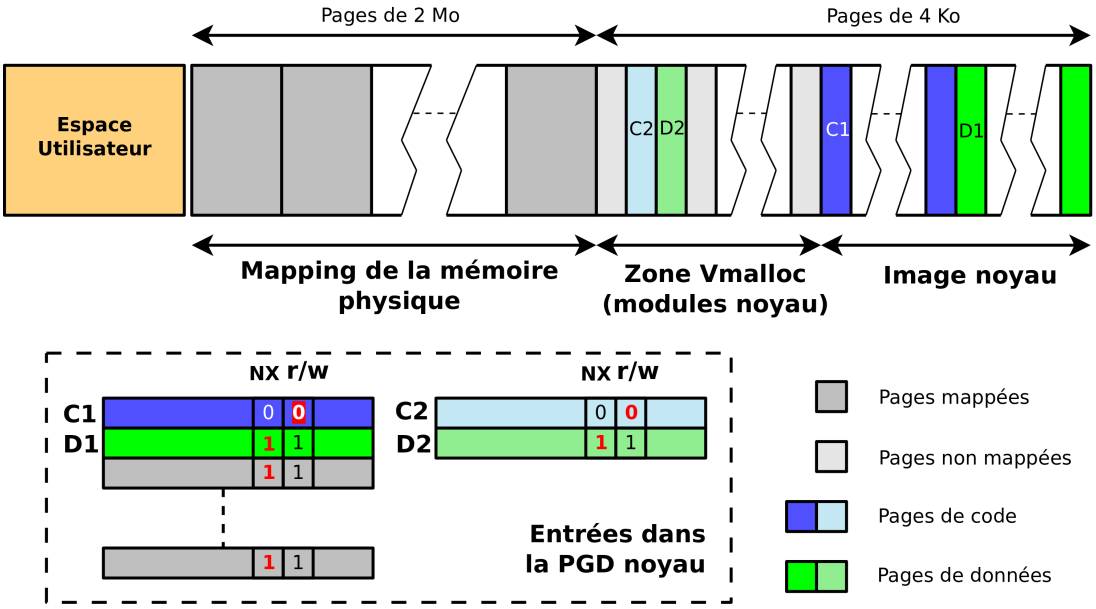

FIG. 4.9 – L'agencement de l'espace d'adressage noyau (première modification)

Cependant, comme nous l'avons expliqué en introduction de cette section, une action du 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` doit être positionné à la valeur 0 sur l'ensemble des pages qui contiennent les tables de pages "noyau", afin d'éviter toute modification d'attributs. La figure 4.10 illustre cette situation.

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 ces contraintes, l'hyperviseur a besoin de conserver une copie de l'agencement initial de l'espace du noyau en ce qui concerne les pages en exécution-seule, lecture/écriture-seule et lecture-seule (c'est-à-dire qu'il conserve une copie des tables de pages "noyau"), afin de valider ou non les futures modifications des entrées dans les tables de pages. Normalement, les entrées dans ces tables ne sont pas changées après l'initialisation du système sauf pour la zone

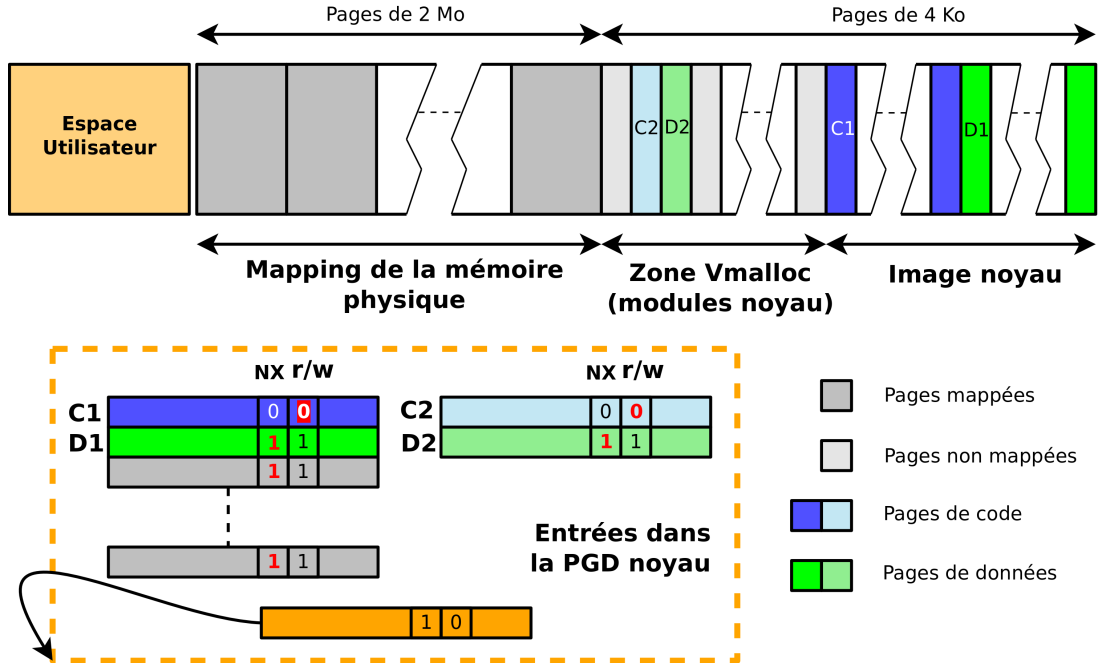

FIG. 4.10 – L'agencement de l'espace d'adressage noyau (seconde modification)

`VMALLOC`[33]. 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 qu'elle informe l'hyperviseur de l'allocation de nouvelles pages. Cette notification est réalisée au travers d'un *hypercall* (c'est-à-dire d'un appel à l'hyperviseur) fondé sur l'instruction `VMCALL` qui, lors de son exécution, déclenche une VM-exit (sans effectuer d'autres actions). Lors d'une VM-exit engendrée par l'appel à `vmalloc()`[34], notre hyperviseur ajoute alors les entrées nécessaires dans les tables de pages du noyau (après vérification que les adresses physiques à projeter appartiennent à la zone `VMALLOC`), en positionnant les attributs relatifs au type de la région allouée (pour des raisons d'implémentation que nous expliquons dans la section 4.3.3.2, le type de ces régions est toujours celui de données modifiables mais non exécutables). Il s'ensuit également la mise à jour des tables de pages qui servent de référence à notre hyperviseur (afin qu'il puisse alors préserver l'intégrité de ces nouvelles contraintes).

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

[33] 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` et ne s'applique que pour les noyaux 32 bits.

[34] Afin de s'assurer de la provenance de l'appel, nous avons élaboré une technique que nous expliquons dans la section 4.3.3.3.

d'une VM-entry[35]). 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 en comparant avec les tables qu'il a initialement sauvegardés et qu'il utilise comme référence[36]. 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 "noyau" que lors de l'appel à la fonction `vmalloc()` (*cf.* annotation 33), 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 autre problème. Considérons qu'un attaquant 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 d'exécution). Par la suite, l'attaquant 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 (le registre `cr3` est un KCO impliqué dans la préservation de la structure du noyau). 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.

Il reste encore un dernier point à traiter afin d'assurer la protection de la structure du noyau. Il s'agit de protéger les tables de pages qui décrivent l'espace d'adressage de l'utilisateur pour chaque processus.

D'une part, Hytux doit empêcher toute exécution de code situé dans l'espace "utilisateur" lorsque que le noyau s'exécute. De telles actions sont toujours illégitimes : du code qui réside dans l'espace "utilisateur" ne devrait pas s'exécuter dans le mode le plus privilégié du processeur. Ces actions ont été identifiées comme appartenant à la classe 1.1.4 (*cf.* chapitre 1 section 1.5). Une solution à ce problème est de positionner à `1` l'attribut `NX` sur les entrées des tables de pages qui correspondent à l'espace "utilisateur", lors de chaque entrée dans le noyau, et ensuite de rétablir les tables initiales lors de chaque sortie. Ce travail peut être effectué directement au niveau du noyau. Une autre solution, plus performante et n'entraînant pas (ou peu) de modifications du noyau, est de profiter de la technologie Intel EPT (*Extended Page Table*) qui n'est toutefois disponible que pour une partie des processeurs qui supportent Intel VT-x (à par-

[35] 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 au noyau.

[36] 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.

tir de l'architecture Nehalem). Cette technologie ajoute un niveau supplémentaire de traduction d'adresses lors d'un accès à la mémoire physique quand le processeur s'exécute dans le mode VMX *non-root* (c'est-à-dire dans notre cas le mode d'exécution du système invité Linux et non celui de Hytux). Ainsi, une adresse physique dans le système invité ne correspond plus à une adresse physique de la machine réelle. Elle doit subir une étape de traduction auparavant. Cette traduction supplémentaire est contrôlée par le biais de tables similaires aux tables de pages. On les nomme "tables EPT". Ainsi, pour empêcher toute exécution de code "utilisateur" depuis le noyau, Hytux doit préparer deux jeux de tables EPT, l'une où seul le code du noyau est autorisé à s'exécuter et l'autre qui permet l'exécution du code de l'espace utilisateur. Ensuite, lors d'un passage en mode noyau, Hytux doit mettre en place le premier jeu de tables, et à la sortie le second jeu. Notons que Hytux doit notamment intercepter les interruptions afin de prendre la main lors d'un passage en mode "noyau". Toutefois ces interceptions ne sont pas suffisantes, car le mode "noyau" est également atteint dans d'autres situations particulières, comme lors de l'exécution de l'instruction `sysenter` (appels-système). Pour que Hytux prenne la main dans ces cas, une solution est d'insérer l'instruction `vmcall` (qui produit une VM-exit) à chaque point d'entrée du noyau qui est atteint lors de ces situations particulières. Toutefois, le cas de `sysenter` peut être traité de façon plus élégante, sans avoir à modifier le code du noyau. Il suffit que Hytux modifie le MSR `SYSENTER_EIP` lors de son chargement afin qu'il pointe vers une fonction particulière qu'il fournit. Cette fonction exécute simplement l'instruction `vmcall` et effectue ensuite un saut vers l'adresse qui était présente dans `SYSENTER_EIP` avant le chargement de Hytux.

D'autre part, il faut empêcher toute tentative d'accès à l'espace "noyau" depuis l'espace "utilisateur". Ainsi, Hytux doit s'assurer qu'aucune entrée de tables de pages qui décrivent l'agencement de la mémoire de l'espace "utilisateur", ne contient d'adresses physiques de pages "noyau". La solution que nous proposons modifie le noyau pour que ce dernier informe Hytux (au travers d'un *hypercall*) des tables de pages qu'il crée[37]. Hytux protège alors ces tables de la même manière qu'il protège les tables "noyau", et vérifie à chaque chargement de `cr3` que les entrées de l'espace "utilisateur" de la table de plus haut niveau sont valides (c'est-à-dire qu'elles correspondent aux adresses des tables qu'il connaît). Une autre solution est de s'appuyer sur la technologie Intel EPT, et d'employer deux jeux de tables EPT comme précédemment. La seule différence est que le second jeu de tables EPT (celui qui est employé pour la traduction lorsque le système s'exécute dans l'espace "utilisateur") est configuré afin d'interdire toute modification du noyau[38]. Cette dernière solution n'est toutefois pas complète, car elle n'empêche pas la modification des tables de pages de l'espace "utilisateur". Un attaquant qui opère *depuis l'espace "noyau"*, peut modifier ces tables afin qu'elles pointent vers la mémoire du noyau, et par la suite accéder sans restriction au noyau.

4.3.3.2 La préservation des contraintes sur l'agencement des modules en mémoire

Les modules "noyau" sont des binaires au format ELF (*Executable and Linking Format*) [T.I. Standard, 1995]. Depuis le noyau 2.6 (seule version que nous considérons dans ce chapitre), la

[37] La création de ces tables s'effectue lors de la création d'un processus. Il s'agit donc de placer un *hypercall* à cet endroit.

[38] Il suffit de positionner à `0` l'attribut `Write access` des tables EPT pour toutes les pages du noyau.

résolution des symboles des modules "noyau" est effectuée directement au sein du noyau[39]. Ainsi il est nécessaire de modifier le code du module une fois chargé en mémoire afin que les symboles qu'il emploie dans son code soient remplacés par les adresses des symboles "noyau" correspondants.

Détaillons quelque peu le fonctionnement du sous-système noyau qui s'occupe du chargement des modules, implémenté dans la primitive `load_module()` (appelé par `init_module()` et présent dans le fichier `kernel/module.c` des sources du noyau). Bien qu'un module soit un binaire ELF, et révèle ainsi des sections de codes et de données, Linux n'alloue que deux régions de mémoire dans la zone `VMALLOC` (via l'appel à `vmalloc()`[40]) pour ajouter le module à l'espace d'adressage du noyau. Ces deux régions référencées pour chaque module par les variables `module_init` et `module_core`, contiennent respectivement le code et les données de la procédure d'initialisation du module, et le code et les données du module qui ne sont pas relatifs à son initialisation. Notre approche nécessite alors une modification de `load_module()`, afin que cette primitive alloue une région de mémoire pour la section de code principale du binaire ELF et une autre pour sa section de données, voire une troisième pour une éventuelle section de données non modifiables[41].

Nous envisageons alors une procédure exceptionnelle qui serait appelée par la primitive `load_module()` (responsable du chargement des modules), et qui informerait notre hyperviseur qu'une région de mémoire préalablement employée comme une région de données doit être transformée en une région de code. L'approche envisagée est de mettre en œuvre un *hypercall* (c'est-à-dire un appel à notre hyperviseur) via l'exécution de l'instruction `VMCALL` (qui déclenche une VM-exit), et le chargement de deux registres généraux afin de spécifier l'opération que nous souhaitons que l'hyperviseur effectue, ainsi que l'adresse de la page en mémoire qui doit subir la transformation[42]. Afin de préserver l'unicité du point d'appel de cette procédure, notre hyperviseur vérifie que cet appel provient du sous-système de chargement des modules (*cf.* section 4.3.3.3). Il vérifie également qu'aucune modification demandée par la primitive sur les tables de pages de la zone VMALLOC ne fait intervenir une adresse physique de l'espace de l'hyperviseur, ou encore de l'espace du noyau en dehors de la zone VMALLOC.

Enfin, concernant la couverture de la classe 1.1.1, qui implique notamment les actions de chargement de modules "noyau" malveillant, notre approche de préservation de contraintes n'est pas applicable. Une approche complémentaire doit être employée afin de ne charger que

[39] C'est pourquoi, les modules contiennent un certain nombres de sections supplémentaires par rapport à un binaire classique de programme.

[40] Plus précisément, les allocations de mémoires pour les modules sont effectuées par `load_module()` au travers de la fonction `module_alloc_update_bounds()`) qui exécute `module_alloc()`. Cette dernière fonction fait appel à la primitive `vmalloc()` afin d'allouer de la mémoire dans la zone `VMALLOC`.

[41] Ces régions seraient en surplus de la région contenant le code et les données d'initialisation du module, cette dernière étant recyclée (libérée) par le noyau après le chargement du module. Un *patch* noyau, proposé par Siarhei Liakh, implémente justement ce découpage en mémoire du module.

[42] Notons que la mise en œuvre d'un *hypercall* n'est pas obligatoire mais permet d'améliorer les performances. En effet, sans l'utilisation de l'*hypercall*, il est possible d'intercepter au travers d'une VM-exit l'accès aux tables de pages de la zone VMALLOC, et ensuite de déterminer si la modification à effectuer est valide en se fondant sur son origine, et le cas échéant de l'effectuer. Cependant, il est plus efficace en temps de calcul, de passer par un *hypercall*, car l'hyperviseur peut alors effectuer toutes les modifications d'une seule traite sans que de multiples VM-exit ne soient déclenchées (une pour chaque instruction `mov`) qui accède aux tables de pages).

des modules pour lesquels une certaine confiance peut être établie. Cette approche peut être mise en œuvre au travers d'un mécanisme de signature de modules par un tiers de confiance, analogue au mécanisme mis en œuvre dans [Microsoft Corporation, 2006]. La signature du module peut alors être ajoutée dans une nouvelle section du module "noyau", et sa vérification effectuée au niveau de la primitive `load_module()`.

Il est important de remarquer que les modules qui peuvent être chargés sous le contrôle de notre hyperviseur n'entraînent pas de modification du code du noyau ou de données spécifiées non modifiables. Cela est une conséquence directe de la mise en œuvre des mécanismes de préservation de la structure du noyau qui garantissent son intégrité vis-à-vis des actions des classes 1.1.2 et 1.1.3. De même, l'intégrité des fonctions du noyau dont la préservation est traitée dans la section 4.3.2 ne saurait être remise en cause par le chargement de module "noyau". La préservation de l'intégrité des KCO nous permet de restreindre l'étendue des dégâts que pourraient commettre un module "noyau", qu'il soit réputé de confiance ou non. En effet, le chargement d'un module signé par un tiers de confiance ne garantit pas forcément l'innocuité du module ou encore l'absence de vulnérabilités dans le module. Dans le cas où une analyse statique complète menée sur le module pourrait attester de l'absence de vulnérabilités, le module pourrait alors réellement être réputé de confiance. Cependant cette confiance dépendrait toujours du tiers ayant effectué l'analyse. Ainsi, adopter une démarche de préservation de l'intégrité d'un système en limitant au maximum les dépendances de ce système à des systèmes tiers, reste donc une approche louable.

4.3.3.3 Comment assurer l'unicité du point d'invocation d'un *hypercall*

Reprenons l'exemple de l'*hypercall* appelé par la fonction `load_module()`. Le fait que ce soit `load_module()` qui invoque l'*hypercall* et que cet *hypercall* ne puisse être appelé que de `load_module()` rend la tâche de l'attaquant plus difficile s'il souhaite employer l'*hypercall*, car il devra passer par `load_module()`, qui est une fonction de haut niveau ayant une utilisation bien précise, et dont la réutilisation dans un autre contexte est difficile, voire impossible, contrairement aux primitives ayant une porté générique comme `vmalloc()` et pouvant être employées directement par l'attaquant[43]. Il est donc plus sûr de placer le point d'invocation de l'*hypercall* dans les plus hautes couches d'abstraction afin de rendre son détournement plus difficile.

Cette approche est envisageable à partir du moment où l'hyperviseur peut effectuer deux types de vérification. D'une part, il doit empêcher l'exécution de l'*hypercall* depuis des adresses de mémoire non autorisées[44]. Cela est réalisable via la lecture du compteur d'instruction (`rip`) de la VMCS lors de l'exécution de l'*hypercall* (faisant suite à l'exécution de l'instruction `vmcall` qui déclenche la VM-exit). D'autre part, il doit empêcher l'exécution d'instructions de branchement vers l'adresse du point d'invocation de l'*hypercall* (adresse pouvant être trouvée via une technique de *pattern matching* en mémoire), comme `jmp` ou `call`. L'hyperviseur devrait alors connaître l'instruction qui s'est exécutée juste avant l'instruction `vmcall`, et s'il s'agit d'un `jmp` ou d'un `call`, l'interdire.

[43] Nous renvoyons notamment le lecteur au chapitre 2 section 2.3 traitant du *rootkit* que nous avons conçu et qui emploie justement directement la primitive `vmalloc()`.

[44] Les adresses autorisées sont celles des points d'invocation de l'*hypercall* dans le code initial du noyau, qui a été chargé en mémoire de façon sûre.

Cependant, le problème est plus complexe qu'il n'y paraît. En effet, le branchement (via `jmp` ou `call`) pourrait être effectuée un peu avant le point d'invocation à l'*hypercall*. Notre idée est alors de forcer à ce que tout branchement soit effectué au début de la fonction qui doit exécuter l'*hypercall* (dans notre exemple il s'agit de `load_module()`) si l'on souhaite que l'*hypercall* fonctionne. Nous mettons en œuvre cela, en invoquant au tout début de la fonction un "*hypercall de préambule*" qui autorise l'hyperviseur à exécuter le prochain *hypercall* classique, et avant la fin de la fonction (instruction `ret`), un "*hypercall de conclusion*" qui ordonne à l'hyperviseur de ne plus considérer aucun *hypercall*[45]. Enfin, la dernière vérification à effectuer pour notre hyperviseur, et de s'assurer que ces deux hypercalls sont invoqués depuis des adresses autorisées[46].

De cette analyse, et afin de restreindre les actions malveillantes ou incorrectes d'un module "noyau", notre approche serait de placer les primitives critiques (comme `vmalloc()` et toutes les primitives non "exportées" par Linux[47]) du noyau dans l'espace "hyperviseur" et de les exécuter au travers d'un *hypercall* dont l'unicité du point d'invocation serait assurée par notre approche.

4.3.3.4 La préservation des contraintes au niveau de la segmentation

Il s'agit, à ce niveau, de préserver l'intégrité les segments de code et de données mis en place par le noyau (*cf.* section 1.4.5) et d'empêcher tout ajout incorrect de nouveaux segments.

Comme expliqué dans la section 1.4.5, la segmentation repose sur deux éléments : la table des descripteurs de segments, appelée GDT (*Global Descriptor Table*) et le registre `gdtr` du processeur qui contient l'adresse linéaire de cette table. Ces deux éléments sont considérés dans notre approche comme étant des KCO. La préservation de l'intégrité du KCO `gdtr` est analogue à celle du KCO `idtr` traitée dans la section 4.3.2.1. Quant à la `GDT`, quelques spécificités sont à prendre en compte. En effet, certaines entrées sont employées dans la mise en œuvre du mécanisme TLS (*Thread Local Storage*). Ces entrées sont au nombre de trois et servent, si nécessaire, à stocker les descripteurs de segments relatifs aux zones de mémoire privées des *threads* (établies via l'appel-système `sys_set_thread_area()`) des processus de l'espace "utilisateur". Ces descripteurs de segments "TLS", sont conservés pour chaque *thread* qui emploie ce mécanisme. Lors de l'ordonnancement d'un de ces threads, le noyau prend soin de modifier la GDT afin d'y écrire les entrées "TLS" correspondantes.

La GDT n'est donc pas une structure de données figée comme l'est l'IDT. Il convient alors d'autoriser la modification des entrées relatives au mécanisme TLS. Cela est effectué sous le contrôle de notre hyperviseur. Il s'agit alors de vérifier que la tentative de modification de la GDT provient du code relatif au mécanisme TLS (*cf.* section 4.3.3.3), que les descripteurs de

[45] Bien entendu, l'exécution de l'instruction `vmcall` provoque toujours une VM-exit. L'hyperviseur peut déterminer s'il s'agit d'un *hypercall* classique, d'un *hypercall de préambule*, ou d'un *hypercall de conclusion*, via l'utilisation d'un registre particulier que le code du noyau renseigne avec une valeur spécifiant sa nature avant l'exécution de `vmcall`.

[46] *cf.* annotation 44.

[47] Il s'agit de fonctions ou de variables qui ne doivent pas être utilisées par des modules. Elle ne figurent donc pas dans la table des symboles du noyau et sont alors invisibles lors de la phase de résolution des symboles d'un module. Mais cela n'empêche en rien le module de parcourir la mémoire à la recherche de ces fonctions ou variables "interdites".

segments à écrire dans la GDT ont un format adapté au TLS (vérification notamment du `DPL`), et que l'écriture doit avoir lieu dans les entrées de la GDT prévues pour le TLS.

Notons enfin, que des traitements similaires doivent être effectués sur la GDT à propos des descripteurs de segment LDT et TSS.

4.3.4 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 de 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.

Il serait ainsi possible de préserver des contraintes sur des structures sensibles du noyau comme les descripteurs de processus (les structures `task_struct` sous Linux). Ces descripteurs contenant un nombre important d'informations, le suivi des modifications par notre hyperviseur serait cependant trop coûteux en temps d'exécution. De plus, les données contenues dans ces descripteurs ne sont pas toutes dans un état prédictible pour notre hyperviseur [48]. Toutefois, pour les données sensibles des processus et que nous pouvons considérer comme des KCO, il conviendrait de les conserver dans des structures de données indépendantes des descripteurs de processus mais qui resteraient accessibles depuis ces derniers[49]. Cette modification permettrait alors de placer ces KCO dans des régions contrôlées par notre hyperviseur, sans avoir à intercepter la majorité des modifications intervenant sur les descripteurs de processus au cours de l'exécution du système.

4.3.5 Prévention de la corruption des KCO depuis les périphériques ou le mode SMM de la CPU

D'après les considérations du chapitre 1, les KCO concernés sont ceux qui résident uniquement dans la mémoire principale. Les solutions à mettre en œuvre pour prévenir la corruption de ces KCO depuis les périphériques ou le mode SMM de la CPU sont des solutions génériques de protection de la mémoire principale vis-à-vis de ces vecteurs d'accès. Nous les développons dans la section 4.4 qui traite de la protection de notre hyperviseur.

4.4 Préservation de l'intégrité de Hytux

Afin de prévenir toute corruption de l'espace de mémoire de notre hyperviseur, il est nécessaire de considérer les différents vecteurs d'accès à la mémoire principale du système, lesquels

[48] En effet, notre hyperviseur ne peut capturer l'essence de toutes les manipulations des données du noyau. Ces capacités d'observation sont directement liées aux interceptions qu'il effectue. Et ces interceptions sont limitées le plus souvent possible afin d'éviter une surcharge sur la CPU trop importante.

[49] Il serait alors nécessaire d'effectuer une modification envahissante du noyau Linux, bien qu'elle puisse être accomplie de façon automatisée.

sont identifiés dans le chapitre 1 (section 1.5.1)[50].

4.4.1 Au travers du contrôle des vecteurs d'accès de type CPU

Afin de prévenir la corruption de l'espace "mémoire" de Hytux, 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[51].

Une autre solution pour empêcher toute altération de Hytux depuis un accès mémoire paginé est d'employer la technologie matérielle Intel EPT[52]. Pour que Hytux protège son espace mémoire, il lui suffit alors de ne pas le référencer dans les tables EPT, et de ne pas référencer ces tables elles-mêmes. Il n'est alors possible de modifier l'espace "mémoire" de Hytux que depuis Hytux lui-même.

Hytux doit également filtrer certains ports d'E/S[53] (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 [BSDaemon *et al.*, 2008; Duflot *et al.*, 2006a].

De façon plus poussée et afin de contrevenir à l'exploitation d'éventuelles vulnérabilités de la routine de gestion de la SMI (fournie par le BIOS) qui pourrait amener à la corruption du système contrôlé par notre hyperviseur ou encore de notre hyperviseur lui-même, nous proposons de virtualiser le mode SMM au travers de la technologie Intel VT-x. La virtualisation du mode SMM, permet de contrôler le mode SMM dans lequel s'exécute la routine SMI et pour lequel aucun mécanisme matériel de protection du processeur n'est habituellement activé.

Pour mettre en œuvre la virtualisation du mode SMM, la technologie VT-x propose la fonctionnalité *Dual-Monitor Treatment*. Avant d'être utilisée, elle doit tout d'abord être activée. Pour cela, il faut agir depuis le mode SMM qui permet l'écriture dans le MSR `SMM_MONITOR_CTL` lequel sert à activer la fonctionnalité. Ensuite, depuis le mode SMM, il faut placer le code appelé *SMM monitor* qui agit comme un hyperviseur de routines SMI[54].

C'est le BIOS qui doit effectuer ces étapes à l'initialisation de la machine, car la SMRAM n'est accessible que lors de l'exécution du BIOS après quoi ce dernier a la charge d'en protéger l'accès en configurant notamment le registre SMRAMC du MCH (*cf.* section 4.3.2). Il reste néanmoins possible d'accéder à la SMRAM en employant la technique de Loïc Duflot présentée dans [Duflot et Levillain, 2009]. Ainsi, par ce moyen il est possible de modifier la SMRAM lors de l'initialisation de Hytux et d'installer alors un *SMM monitor* (*cf.* annotation 24) afin que la

[50] Rappelons que nous ne considérons pas dans notre travail les cas de corruption issus d'éventuels piégeages matériels, bogues matériels ou encore fonctions matérielles non documentées du support de contrôle.

[51] 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*.

[52] Une explication très succincte de cette technologie est donnée en section 4.3.3.1.

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

[54] Ce code doit être situé dans le segment MSEG de la SMRAM.

routine SMI s'exécute dans un environnement contrôlé. On fournit alors une solution générique de contrôle de la routine SMI puisque nous n'avons pas besoin de la coopération du BIOS.

Nous concluons cette section en expliquant comment s'effectue le contrôle du mode SMM après l'avoir activé et configuré de façon adéquate dans notre hyperviseur. Dans cette situation, lorsqu'une SMI se produit, il se déclenche une SMM VM-exit qui charge une VMCS spécifique (configurée initialement par notre hyperviseur), la *SMM-transfert VMCS* (référencé par le registre interne `SMM-transfer VMCS pointer` de la CPU), laquelle fournit l'environnement virtuel sous lequel le SMM monitor s'exécute. Celui-ci effectue alors une VM-entry avec une VMCS qu'il configure préalablement pour exécuter la routine SMI.[55]. Lorsque la routine termine son exécution et exécute l'instruction `rsm`, au lieu de sortir du mode SMM (ce qu'effectue l'instruction habituellement) cela déclenche une VM-exit qui rend la main au *SMM monitor*, lequel rend la main à son tour soit à l'hyperviseur, soit au système contrôlé (suivant que la SMI a été générée en mode *VMX root* ou *VMX non-root*).

4.4.2 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 53) 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 [Duflot et Absil, 2007]. 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 *DMA busmaster* (*cf.* section 1.4.7), lesquels prennent le contrôle d'un bus (tel que le bus Firewire [Piegdon et Pimenidis, 2007]) sans implication de la CPU. Afin de se protéger contre ce type de problème, un système qui contient une IOMMU est nécessaire.

Notre approche profite de la technologie Intel VT-d qui met en œuvre le principe d'une IOMMU pour les architectures de type x86 (au sein du *northbridge*) comme expliqué dans la section 4.1.1.3. Le noyau linux dispose déjà d'un pilote pour ce composant matériel. Lors de l'initialisation du pilote les tables sont établies de façon à ce qu'aucun périphérique ne puisse accéder à la mémoire principale. Les primitives de l'infrastructure de gestion du DMA sont alors responsables de la modification de ces tables afin d'autoriser l'accès à la mémoire aux périphériques, mais seulement à une zone prévue par le noyau pour les accès DMA.

Les tables associées à l'IOMMU doivent cependant être protégées contre toute modification de la même façon que les tables relatives à la pagination, telle que nous l'avons vu dans la section 4.3.3 (à savoir au travers de la mise à 0 du bit `R/W` des attributs des pages contenant les tables de l'IOMMU). Notre démarche est alors de définir un nouvel *hypercall* dans notre hyperviseur (toujours au travers de l'instruction `VMCALL`) afin d'autoriser des modifications

[55] Cet environnement doit être configuré afin de notamment protéger l'accès à la mémoire du noyau et de l'hyperviseur.

sur ces tables. Les primitives du noyau de gestion du DMA sont alors modifiées afin de se servir de cet *hypercall*[56].

Lorsque notre hyperviseur prend la main suite à l'exécution de l'*hypercall*, il effectue deux types de vérification. Tout d'abord il s'assure de l'origine de l'appel (*cf.* section 4.3.3.3), ce qui permet de n'autoriser que le code du noyau implémentant la gestion du DMA à employer cet *hypercall* et ainsi de faire un premier tri sur des utilisations incorrectes. Notons toutefois que cela n'empêche pas l'utilisation des primitives de gestion du DMA afin de modifier de façon malveillante les tables de l'IOMMU. Ces modifications peuvent alors provenir d'actions incorrectes d'un module "noyau" (issues par exemple de l'exploitation d'une éventuelle vulnérabilité l'affectant), ou encore d'actions incorrectes du noyau lui-même. C'est pour cela qu'intervient une autre vérification afin d'empêcher toute modification des tables de l'IOMMU entraînant une autorisation d'accès depuis les périphériques à l'espace de l'hyperviseur, mais également à l'espace du noyau (en dehors de la zone prévue pour le DMA).

Notons finalement, que la technologie VT-d prévoit l'utilisation de *Device-IOTLB* [Abramson *et al.*, 2006] implémentées au niveau des périphériques, qui sont des IOTLB[57] supplémentaires qui s'ajoutent à celle du *chipset* compatible VT-d. Il s'agit alors de faciliter le passage à l'échelle de l'IOTLB. Cependant, cette fonctionnalité peut alors servir à des périphériques malveillants pour contourner le contrôle d'accès mis en place au travers des tables de traduction de l'IOMMU. Heureusement, cette fonctionnalité peut être désactivée au niveau du *chipset*.

4.5 Synthèse sur notre approche de préservation de contraintes

Notre approche fondée sur la préservation de contraintes est applicable aux trois classes principales regroupant les actions malveillantes ciblant un noyau (la classification est établie au chapitre 1 dans la section 1.5).

Au sujet des classes 1.1.2, 1.1.3 et 1.1.4, notre approche fournit une solution complète au travers de la préservation de l'intégrité de la structure du noyau (section 4.3.3). Notre approche ne s'applique cependant pas à la classe 1.1.1 (comme cela est expliqué dans la section 4.3.3.2). Par contre, les éventuelles actions malveillantes qui en découlent, appartiennent généralement aux autres classes. C'est à ce niveau que notre approche peut agir.

En ce qui concerne les classes 1.2, 2 et 3, notre approche possède une aptitude unique à contrôler le mode *ring 0* (c'est-à-dire le mode "noyau") et ainsi peut partiellement enrayer les actions malveillantes de ces classes. Nous adoptons pour cela, une démarche qui vise à préserver les contraintes liées aux fonctions essentielles du noyau, et ainsi d'assurer un certain degré de confiance sur son intégrité (section 4.3.2). Cette préservation de contraintes se réalise toutefois d'une façon spécifique pour contrevenir aux actions de la classe 3. Elle s'effectue au travers de l'utilisation de technologie de type IOMMU (dans notre cas il s'agit de Intel VT-d).

Enfin, il est nécessaire d'assurer l'intégrité de notre hyperviseur à qui incombe la charge de préservation des contraintes. La démarche adoptée vise à empêcher tout accès à l'espace "mémoire" de l'hyperviseur (section 4.4). Nous faisons alors l'hypothèse que notre hyperviseur

[56] Plus précisément, ces primitives emploient les fonctions du pilote de l'IOMMU. L'*hypercall* est effectué au sein de ces fonctions.

[57] Une IOTLB joue le même rôle qu'une TLB (*cf.* section 4.1.2) pour l'unité de pagination de la MMU, mais s'applique à l'IOMMU.

ne contient pas de vulnérabilités. Cette hypothèse se justifie par le fait que notre hyperviseur est de taille très réduite. D'une part il n'est conçu que pour contrôler une seule machine virtuelle, à savoir le système que l'on souhaite protéger, et d'autre part il profite du support matériel pour la virtualisation, ce qui facilite grandement son implémentation. Il est donc envisageable d'effectuer une analyse statique du code pour en garantir sa correction vis-à-vis de la sécurité (absence de vulnérabilités).

Nous tenons à souligner le fait que tous les éléments essentiels au fonctionnement du noyau ne peuvent être facilement capturés en tant que KCO. Tout d'abord, ces éléments ne sont pas forcément toujours dans un état prédictible. Ensuite pour ceux qui le sont, le contrôle sur les contraintes qui en découle est limité par les capacités d'observation de notre hyperviseur. Ces capacités sont directement liées aux types d'évènements interceptés. Plus le nombre d'interceptions est élevé, plus la capacité d'observation de notre hyperviseur est importante. Cependant, les performances du système sont également impactées à hauteur du nombre de ces interceptions.

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. Le nombre et le type des interceptions effectuées par notre "hyperviseur de protection" sont comparables à ceux des hyperviseurs classiques (section 4.1.1) dont la problématique est la gestion de multiples machines virtuelles. Ensuite, la charge des traitements effectués par notre hyperviseur au cours de chaque interception est faible et constante (vérification de contraintes simples). En outre, 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 nous baser sur les hyperviseurs existants qui mettent à profit la virtualisation matérielle (tel que KVM — Kernel Based Virtual Machine [Kivity *et al.*, 2007]). Ces solutions ne causent pas de ralentissements majeurs du système et par conséquent des résultats similaires sont attendus avec notre approche.

4.6 Conclusion

Dans ce chapitre nous avons expliqué notre démarche quant au problème de la garantie de l'intégrité d'un noyau de système d'exploitation. Afin d'aborder sereinement notre approche, nous avons tout d'abord présenté les principes de la virtualisation, contexte dans lequel s'articule notre logique. Ensuite, un développement sur l'organisation en mémoire du noyau a été exposé afin de mieux appréhender l'objet de notre étude. C'est alors que nous avons dépeint notre approche fondée sur la préservation de contraintes afin d'assurer l'intégrité d'une part de la structure d'un noyau et d'autre part de ses fonctions essentielles. Cette préservation exige un contrôle à la fois de la mémoire principale, de l'état des composants matériels dont dépend le noyau pour son exécution, mais également des composants matériels avec qui le noyau communique. Pour cela, nous avons également proposé une approche supplémentaire visant à restreindre les actions malveillantes ou incorrectes d'un module "noyau" (*cf.* section 4.3.3.3). Enfin, nous avons positionné notre approche vis-à-vis de la classification des actions malveillantes ciblant un noyau (établie dans la chapitre 1), et évalué de façon qualitative l'impact de notre approche sur les performances du système.

L'objectif qui a été poursuivi dans notre travail fût d'assurer la protection d'un système informatique contre les attaques évoluées qui requièrent des actions parmi les plus privilégiées sur le système. Il s'agit notamment des attaques qui se servent de *rootkits* noyau (que nous avons étudiées dans le chapitre 2). Nous nous sommes alors penchés sur la question de la protection de l'intégrité d'un noyau afin d'enrayer ces attaques. L'étude de ce problème nous a mené à employer certaines technologies (Intel VT-x, VT-d, TXT) et à élaborer certains principes (les KCO), que nous pensons être applicables à la couche moins privilégiée d'un système informatique, à savoir l'espace "utilisateur". De nombreuses solutions existantes mettent en œuvre à ce niveau des mécanismes au sein du noyau du système. Ces approches sont tout à fait louables car elles s'appuient sur le fait que le noyau s'exécute à un niveau de privilège matériel supérieur à celui de l'espace utilisateur. Néanmoins les technologies matérielles de virtualisation autorisent un contrôle plus fin des actions effectuées sur le système que celui permis par le mode "noyau". Il serait alors intéressant d'étudier l'apport que fourniraient ces technologies aux solutions de protection de l'espace "utilisateur".

Enfin, afin de valider notre approche fondée sur la préservation de l'intégrité des objets contraints par le noyau (KCO), nous travaillons actuellement sur la mise en œuvre d'un formalisme qui représente 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 d'une part à vérifier la correction de notre approche de façon formelle, et d'autre part à rendre possible la représentation d'objets contraints par le noyau dès l'étape de spécification d'un noyau.

Conclusion

Dans nos travaux, nous nous sommes focalisés sur la sécurité *en vie opérationnelle* des systèmes d'exploitation de type COTS, en supposant qu'ils contiennent des vulnérabilités de par leur complexité. De plus, nous avons supposé que le problème lié aux attaques physiques était résolu, nous poussant dans notre étude à ne considérer que les attaques logiques.

Afin de proposer des solutions à ce problème complexe, nous avons tout d'abord essayé de cerner le contexte des attaques informatiques en en proposant une caractérisation. Nous nous sommes alors intéressés principalement aux attaques requérant la corruption du noyau d'un système d'exploitation. Cette démarche s'est justifiée d'une part, parce que la sécurité d'un système d'exploitation dépend de façon considérable de l'intégrité de son noyau. D'autre part, le domaine s'intéressant à la protection des noyaux en vie opérationnelle est encore relativement peu exploré, notamment en ce qui concerne les moyens de prévention, moyens qui nous semblent être d'une importance capitale pour assurer la sécurité d'un système d'exploitation. Après avoir notamment analysé les parasites informatiques de type *rootkits* "noyau" et en avoir élaboré un, les moyens de "détection *a posteriori*" nous sont apparus clairement insuffisants.

Afin de proposer des solutions valables contre les attaques requérant la corruption d'un noyau, nous avons analysé les différentes actions malveillantes qui entraînent la perte d'intégrité d'un noyau. Cette étude nous a permis d'élaborer une approche de protection de noyau, résolument orientée vers la prévention. La démarche que nous avons suivie a été tout d'abord d'identifier les éléments essentiels sur lesquels repose en partie le fonctionnement d'un noyau. Nous avons alors analysé les limites d'utilisation normale de ces éléments par le noyau afin de déterminer les invariants nécessaires à un fonctionnement *correct* du noyau. Ces invariants ont été exprimés sous forme de contraintes sur ces éléments. Notre approche a alors été de garantir le respect de ces contraintes. Nous avons mis en œuvre cette approche sur un noyau Linux x86 64 bits (version 2.6.27 et antérieures), en développant un hyperviseur léger (*Hytux*) s'appuyant sur les extensions matérielles de virtualisation Intel VT-x et VT-d pour *empêcher* toute violation de ces contraintes.

1 Nos contributions

Nous avons dans un premier temps (premier chapitre) proposé une caractérisation des attaques sur un système informatique. Suite à cette analyse, nous avons identifié différentes classes d'attaques suivant leur mode opératoire, et ainsi proposé une classification des attaques sur un système informatique. Cette analyse a été menée dans un souci d'identification des points critiques où la protection d'un système informatique est en jeu. Nous nous sommes alors focalisés sur les attaques qui s'appuient sur la corruption d'un noyau de système d'exploitation, en

analysant leurs composantes (c'est-à-dire leurs actions) malveillantes. Ce travail nous a servi à identifier les différents vecteurs d'accès pour altérer le comportement d'un noyau. Suite à cette activité, nous avons pu caractériser les actions malveillantes qui nuisent à l'intégrité des noyaux. Enfin nous avons proposé une classification de ces actions [Lacombe *et al.*, 2009b], élément essentiel à la mise en œuvre de mécanismes visant à assurer l'intégrité d'un noyau.

Nous avons élaboré ensuite (deuxième chapitre) d'une part les principes nous permettant de modéliser les *rootkits*, et d'autre part une approche originale de *rootkit* "invisible" pour un noyau Linux. Nous avons ainsi développé dans un premier temps une réflexion sur l'un des objectifs primordiaux des *rootkits*, à savoir dissimuler l'activité de l'attaquant. Ensuite, afin de mieux évaluer ces parasites informatiques, et ainsi concevoir des solutions visant à s'en prémunir, nous en avons proposé une caractérisation (architecture et attributs essentiels d'un *rootkit*). Enfin, nous avons développé un *rootkit* original pour lequel nous avons conçu et mis en œuvre des techniques de dissimulation inédites [Lacombe *et al.*, 2008, 2007; Lacombe, 2007; Lacombe et Gaspard, 2007].

Finalement, nous avons (dernier chapitre) conçu une approche fondée sur la préservation de contraintes afin d'assurer l'intégrité d'une part de la structure d'un noyau [Lacombe *et al.*, 2009b,c,a] et d'autre part de ses fonctions essentielles, vis-à-vis d'actions incorrectes qui sont externes au noyau mais également *internes* au noyau. Cette préservation exige un contrôle à la fois de la mémoire principale, de l'état des composants matériels dont dépend le noyau pour son exécution, mais également des composants matériels avec qui le noyau communique. Notre approche s'appuie sur les technologies matérielles de support à la virtualisation, car elles nous permettent de restreindre les capacités d'un noyau, et par là-même d'intercepter toute tentative de violation de nos contraintes. En outre, dans cette démarche de restrictions des déviations possibles du comportement correct d'un noyau, nous avons proposé une approche complémentaire pour assurer l'unicité d'invocation (dans le code du noyau) des *hypercalls* que nous avons définis. Nous avons alors exposé la façon dont pourrait être mise à profit cette démarche pour restreindre encore plus les déviations du noyau qui pourraient provenir notamment du chargement d'un module "noyau" malveillant. Nous avons mis en œuvre nos approches dans un prototype que nous appelons Hytux. Il s'agit d'un hyperviseur léger (reposant sur les technologies Intel VT-x et VT-d) qui implémente nos idées pour protéger un noyau Linux x86 64 bits (à partir de la version 2.6.27).

2 Travaux futurs

2.1 L'évaluation des *rootkits*

Nous avons proposé dans le chapitre 2 une architecture fonctionnelle des *rootkits*, et mis en évidence trois de leurs attributs fondamentaux, à savoir leur *invisibilité*, leur *robustesse* et leur *pouvoir de nuisance*. Après avoir défini ces trois attributs, nous avons proposé des pistes pour les mesurer. La détermination de métriques sur ces attributs doit permettre d'une part d'évaluer de façon objective chaque *rootkit* qui affecte un même hôte. D'autre part, ces métriques seraient utiles pour évaluer la résilience d'un système[1] face aux *rootkits*. En effet, l'évaluation de cette

[1] Nous entendons par résilience d'un système, sa protection intrinsèques, c'est-à-dire sa capacité à survivre aux agressions.

résilience doit nécessairement prendre en compte les différents attributs des *rootkits*. De plus, l'évaluation pourrait être effectuée suivant notre classification des actions malveillantes qui affectent l'intégrité d'un noyau (*cf.* section 1.5), car la résilience d'un système face aux *rootkits* est intimement liée aux *types* d'actions que ces derniers effectuent.

L'établissement de ces métriques rendrait alors possible l'évaluation des protections anti-*rootkits*, mais aussi des solutions de protection de noyaux.

2.2 Empêcher l'utilisation inappropriée de fonctionnalités critiques du noyau

Afin d'empêcher une utilisation inappropriée de fonctionnalités critiques du noyau par un attaquant (par exemple au travers d'un module "noyau") notre approche serait de les placer dans l'espace "hyperviseur" et que leur exécution s'effectue au travers d'un *hypercall* dont l'unicité du point d'invocation serait assurée par l'approche que nous avons proposée (*cf.* section 4.3.3.3). Ces *hypercalls* seraient alors uniquement employés par les fonctions de haut niveau du noyau qui en auraient besoin. Pour qu'un attaquant se serve de ces primitives critiques il devrait alors le faire au travers de l'exécution des fonctions y faisant appel.

Nous voyons ici poindre des considérations architecturales pour les noyaux. Il serait intéressant de définir un noyau essentiellement à base de fonctions de haut niveau (par exemple "le chargement d'un module") qui feraient appel à des fonctions critiques par le bais d'*hypercalls*. Cette démarche, permettrait de rendre l'emploi des primitives critiques difficile voire impossible par l'attaquant. En effet, si ces primitives ne peuvent être exécutées que par les fonctions dédiées de haut niveau, l'attaquant devrait alors utiliser ces dernières pour obtenir réellement le service qu'il souhaite. Par exemple il devrait utiliser la fonction de "chargement d'un module" pour allouer de la mémoire via la primitive critique `vmalloc()` (*cf.* section 4.3.3.3). Nous pensons que cette démarche limiterait le potentiel d'un attaquant notamment pour deux raisons. D'une part, les paramètres à fournir pour l'emploi d'une fonction de haut niveau (dans notre exemple il s'agit du fichier binaire qui décrit un module) peuvent être une condition d'échec pour l'attaquant suivant la difficulté qu'il aurait à fournir des paramètres valides. D'autre part, l'emploi d'une primitive critique (convoitée par l'attaquant) au sein d'une fonction de haut niveau peut ne pas être relié uniquement à ses paramètres, mais à d'autres variables du noyau (y compris le compteur d'instruction aux points d'appel) que l'attaquant devrait alors contrôler pour maîtriser complètement l'utilisation de la primitive critique.

La démarche que nous proposons peut être vue comme une diminution des degrés de liberté d'un noyau afin qu'il se rapproche d'un noyau idéal qui ne pourrait effectuer que des actions appropriées. Un problème se pose alors sur la définition des fonctions de haut niveau, mais également sur la façon de mesurer la difficulté de l'emploi d'une primitive critique par un attaquant. Ce dernier point est important pour évaluer l'efficacité de cette démarche.

2.3 Modélisation des noyaux

2.3.1 La motivation

Finalement, afin de valider notre approche fondée sur la préservation de l'intégrité des objets contraints par le noyau (KCO), nous travaillons actuellement sur la mise en œuvre d'un forma-

lisme qui représente 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")[2]. Nous espérons que cette formalisation nous aidera d'une part à vérifier la correction de notre approche (car elle nous offre un moyen d'expression non-ambiguë sur lequel notre raisonnement peut s'appuyer), et d'autre part à rendre possible la représentation d'objets contraints par le noyau dès l'étape de spécification d'un noyau. Ce dernier axe de recherche est directement associé à l'architecture des noyaux, et dépasse ainsi largement le cadre de la sécurité des systèmes d'exploitation de type COTS.

2.3.2 La démarche

Les problèmes de sécurité que nous souhaitons résoudre pour les noyaux de système d'exploitation se situent dans le cadre du traitement, de l'échange et du stockage de l'information. Ainsi, nous avons réalisé une première ébauche d'un *modèle formel* qui représente ces trois fonctions essentielles, lesquelles sont directement reliées à la notion de système d'information. En effet, ce dernier consiste notamment en un ensemble d'*acteurs* qui *manipulent* des *informations* et *interagissent* entre eux, au cours du *temps*. C'est pourquoi, nous avons défini un méta-modèle qui intègre un temps logique dans lequel évolue un ou plusieurs agents d'exécution (par exemple un processeur) qui interprètent chacun un ou plusieurs agents logiciels (par exemple un noyau, un hyperviseur ou encore une application). La description des agents d'exécution peut alors être effectuée de façon partielle suivant les problèmes à traiter. Toutefois cette description doit couvrir (1) le support (ensemble de cases de mémoire) et l'adressage des données, (2) les opérations sur ces données que peuvent effectuer les agents logiciels, ainsi que (3) les mécanismes de l'agent d'exécution qui ont une incidence sur l'interaction avec l'agent logiciel (par exemple le mécanisme des interruptions).

L'ensemble de ces travaux devraient permettre de mieux combattre les maliciels futurs que les attaquants pourraient tenter de développer, y compris sur des architectures nouvelles de processeurs (avec de nombreux cœurs, par exemple)

[2] Il n'existe à ce jour que peu de publications concernant la modélisation formelle de noyaux. Notons toutefois l'ouvrage [Craig, 2006] de Iain D. Craig qui établit un modèle formel en Object Z d'un noyau assez minimal de type monolithique.

Annexe A

Une tentative de modélisation des systèmes informatiques

A.1 Introduction

Nous présentons dans cette annexe notre tentative de modélisation formelle des systèmes informatiques, matériels et logiciels. Notre premier objectif était de modéliser les noyaux de système d'exploitation afin de confronter le résultat de la modélisation aux différentes classes d'actions malveillantes visant ces noyaux, et déterminer ainsi des critères permettant de juger de l'efficacité d'une architecture de noyau pour contrer ou inhiber les différentes classes d'actions malveillantes[1]. Il est dès lors souhaitable d'avoir un modèle général et évolutif : il doit être possible d'ajouter des concepts supplémentaires pour faire apparaître les critères adaptés au traitement d'un problème particulier, un peu à la façon de calques qui s'appliqueraient au modèle et en donneraient un aspect différent. Suite à l'identification de ces critères, l'étape suivante serait alors de déterminer des propriétés qui devraient être vérifiées par un noyau afin de le rendre plus robuste aux différentes classes d'actions malveillantes.

Notre idée de départ pour cette modélisation de noyau, pour faciliter son adaptabilité, était de représenter en premier lieu les concepts essentiels qui sont nécessaires à l'étude de tout problème de sécurité, une sorte de dénominateur commun. Cette approche nous permettrait alors d'orienter plus facilement notre modèle pour l'étude d'un problème particulier. Pour identifier ces concepts essentiels nous avons eu l'idée de faire une analogie entre les noyaux et les systèmes d'information, lesquels ont des objectifs similaires mais à des niveaux d'abstractions différents, objectifs qui sont de définir les moyens nécessaires aux opérations de traitement, de communication et de stockage de l'information.

Au début de nos travaux sur la modélisation, nous avons étudié l'ouvrage [Craig, 2006] de Iain D. Craig, détaillant pour la première fois une modélisation formelle de noyaux de système d'exploitation. Il y présente une modélisation de noyaux monolithiques simples (en partant d'un noyau minimaliste auquel sont ajoutés des fonctionnalités de plus en plus élaborées). Les travaux menés par l'auteur s'appuient principalement sur le langage de spécification Object-Z [Smith, 2000] qui est une surcouche de la notation Z [Spivey et Abrial, 1992], elle-même

[1] Par exemple, le critère de "similarité" entre sous-systèmes d'un noyau pourrait être relié aux actions malveillantes de détournement de sous-système.

reposant sur la théorie des ensembles [Cori et Lascar, 2003]. En outre, cette modélisation fait également intervenir l'algèbre de processus [Fokkink, 2000] CCS (*Communication and Concurrency*) pour les aspects de communication. D'autre part, l'approche de Iain D. Craig consiste à partir d'un niveau abstrait qui est raffiné successivement pour atteindre un niveau proche du matériel. Il nous semble préférable de partir d'une représentation fine du matériel et d'y ajouter successivement les "calques" pour représenter les aspects intéressants vis-à-vis des attaques, en particulier par *rootkits*.

Un résultat important de la modélisation de Iain D. Craig est la représentation du comportement du matériel. Cette considération nous a orientés tout au long de notre tentative de modélisation. Nous avons alors souhaité définir un modèle dans lequel pouvait être spécifié le comportement du matériel vis-à-vis de différents niveaux de logiciels. Ainsi, pour représenter les concepts d'unités de traitement (processeurs) et de mémoire, nous avons été amené à définir une entité particulière (nommée dans la suite *agent d'exécution*) qui est centrale dans notre modèle. Les multiples cœurs d'un processeur moderne sont alors autant d'agents d'exécution.

Toujours dans ce souci de rendre notre modèle flexible pour la représentation de différents concepts, nous avons orienté notre modélisation de façon à pouvoir représenter différents types de logiciels (noyau, hyperviseur, application). En outre, des logiciels doivent pouvoir s'exécuter en concurrence sur ces agents d'exécution. L'objectif était alors de pouvoir représenter les interactions entre ces logiciels afin d'identifier celles qui pourraient mettre en péril la sécurité d'un noyau, et ensuite d'en déduire des concepts architecturaux de noyau qui permettraient d'éliminer ces interactions néfastes. Enfin, un autre constat a confirmé l'importance de modéliser correctement le comportement du matériel : les interactions entre ces différents types de logiciel sont étroitement liées aux mécanismes matériels sous-jacents.

Dans la première section nous présentons les concepts fondamentaux de notre modélisation (les agents d'exécution, les agents logiciels et les systèmes d'agents). Nous décrivons alors (partiellement) dans la deuxième section, un exemple particulier de système d'agents. Enfin, nous exposons une extension à notre modèle, censée représenter la notion d'interaction entre agents logiciels et agents d'exécution.

A.2 Modélisation de systèmes d'agents

Notre modélisation repose sur deux notions principales : les agents d'exécution et les agents logiciels. Un agent d'exécution est composé d'une mémoire (éventuellement partagée avec d'autres agents d'exécution) et d'une unité de contrôle. Il peut exécuter ou interpréter des agents logiciels (comme un noyau, une application, un hyperviseur, etc.). Pour être exécuté par un agent d'exécution, ces derniers doivent être présents dans la mémoire de cet agent. La notion d'exécution d'un agent logiciel sous-entend l'existence d'un temps logique sur l'agent d'exécution. En effet, une exécution peut être vue comme une succession d'actions, ce qui définit implicitement une relation de précédence causale entre ces actions, et donc définit un temps logique [Lamport, 1978].

Dans notre modélisation, nous considérons des systèmes constitués d'un ou plusieurs agents d'exécution. Chacun de ces agents peut être soumis à des temps logiques différents. De plus ces agents sont asservis à un certain nombre de propriétés, qui sont les lois qui régissent les évolutions possibles du système. Nous définissons à présent les notions d'agents d'exécution et

d'agents logiciels.

A.2.1 Agent d'exécution

Un *agent d'exécution* est défini par un quadruplet :

$$X = (T^X, \Phi^X, M^X, \Delta^X)$$

où :

- T^X est le temps logique dans lequel évolue l'agent d'exécution. Il est représenté par un ensemble d'instants muni d'un bon ordre (c'est-à-dire muni d'une relation d'ordre total, et qui pour toutes ses parties, admet un minimum[2]) : $(T^X, <)$.
- Φ^X est un ensemble de propriétés (ou "lois") d'évolution de l'agent d'exécution X (c'est-à-dire des règles qui sont respectées à tout moment par X).
- M^X est l'ensemble des cases de mémoire de l'agent d'exécution. M^X est constitué de deux sous-ensemble : M^X_{priv}, l'ensemble des cases de mémoire privées, et M^X_{pub} l'ensemble des cases de mémoire publiques. On muni M^X :
  - d'une fonction totale $\eta^X : M^X \times T^X \rightarrow D^X$ qui pour une case $c \in M^X$ et une date $t \in T^X$, retourne le contenu de c à t. L'ensemble des *contenus* (ou "données") possibles des cases de mémoire est désigné par D^X ;
  - d'une fonction bijective $\alpha^X : M^X \rightarrow P^X$ (fonction d'adressage) qui pour une case c retourne une "adresse" (ou "position"). P^X est un ensemble de mots de D^{X*} (D^{X*} est la fermeture itérative ou fermeture de Kleene[3] de D^X) de longueur $W^X_{size} \in \mathbb{N}$[4].
  - d'une fonction totale $\tau^X_r : M^X \rightarrow T^X$ qui pour une case $c \in M^X$ retourne l'instant du dernier accès en lecture à la case c.
  - d'une fonction totale $\tau^X_w : M^X \rightarrow T^X$ qui pour une case $c \in M^X$ retourne l'instant du dernier accès en écriture à la case c.

[2] Pour plus d'informations, se reporter à l'ouvrage [Cori et Lascar, 2003].
[3] Pour plus d'informations, se reporter aux ouvrages [Wolper, 2006] ou [Autebert, 1994].
[4] Pour notre modélisation, nous avons également besoin de munir P^X d'un ordre total.

Notes :
- D^X doit avoir une structure de groupe.
- D^{X*} (par exemple $\{0,1\}^*$) est muni d'une loi interne $+$, telle que $< D^{X*}, + >$ soit un monoïde[5]($+$ doit être une loi de composition interne associative et D doit admettre un élément neutre pour cette loi). De plus, on considère que D^{X*} est bien ordonné par une relation $<$.
- Le bon ordre $<$ doit vérifier la propriété suivante sur P^X :

$$\exists 1_p \in P^X \mid \forall c_1 \in P^X : (c_1 < c_1 + 1_p) \wedge (\forall c_2 \in P^X : c_1 < c_2 \Rightarrow (c_2 = c_1 + 1_p) \vee (c_1 + 1_p < c_2))$$

  Dans la suite, on utilise respectivement les notations 2_p, 3_p, ... , pour représenter les valeurs $(1_p + 1_p)$, $(1_p + 1_p + 1_p)$, ...
- Par abus de notation, nous employons également le symbole η pour désigner la fonction qui à une séquence de cases S, de longueur l, (notée, $S \in Seq(M^X, l)$) renvoie la séquence des contenus des cases ($Seq(D^X, l)$).
- La n-ième case d'une séquence S de M^x est noté $S[n]$.

– Δ^X représente l'unité de contrôle de l'exécution. Il est défini par le triplet :

$$(A^X_{pub}, A^X_{priv}, O^X)$$

où :

– A^X_{pub} est l'ensemble des actions élémentaires que fournit l'agent d'exécution pour la "manipulation" de M^X_{pub}, que l'on appelle *actions publiques*[6]. Ces actions effectuent des modifications des fonctions η^X, τ^X_r et τ^X_w. On munit A^X_{pub} :

  – d'une fonction de coût $\$^X_{pub}$ qui représente la durée d'exécution d'une action publique ;
  – d'une fonction bijective $\gamma^X : A^X_{pub} \rightarrow C^X$ qui pour une action publique a associe un "codage". C^X est un ensemble de mots de D^{X*}.

[5] Pour plus d'informations, se reporter à l'ouvrage [Autebert, 1994].

[6] À noter que ces actions peuvent modifier M^X_{priv}. Seules leurs opérandes sont dans M^X_{pub}.

Notes :
- Une séquence d'actions publiques S_{pub} est une suite d'éléments de A^X_{pub}. On note l'ensemble des séquences possibles par $Seq(A^X_{pub})$ et l'ensemble des séquences de longueur m (c'est-à-dire un m-uplet) par $Seq(A^X_{pub}, m)$.
- Un agent d'exécution doit au moins disposer d'une action publique qui n'effectue aucune opération. Nous la notons ε. Cette action doit satisfaire à la propriété suivante :

$$\forall a \in A^X_{pub} \ : \\ (a, \varepsilon) = (\varepsilon, a) = a$$

- On définit sur $Seq(A^X_{pub})$, la loi de composition interne $\bullet$ qui vérifie la propriété suivante :

$$\forall S_a, S_b \in Seq(A^X_{pub}) \ : \\ S_a \bullet S_b = (S_a, S_b)$$

- Cette loi est associative et admet ε comme élément neutre dans $Seq(A^X_{pub})$. $< Seq(A^X_{pub}), \bullet >$ est donc un monoïde.

- A^X_{priv} est l'ensemble des actions élémentaires que fournit l'agent d'exécution pour la "manipulation" de M^X_{priv}, que l'on appelle *actions privées*. Ces actions effectuent des modifications des fonctions η^X, τ^X_r et τ^X_w. On munit A^X_{priv} d'une fonction de coût $\$^X_{priv}$ qui représente la durée d'exécution d'une action privée.

- O^X représente l'ensemble des *opérations spéciales* de l'agent d'exécution. Ce sont les opérations internes de l'agent qui ont une incidence sur l'interaction avec l'agent logiciel (par exemple le mécanisme des interruptions). Ces opérations sont des séquences d'éléments de A^X_{priv}, que l'on note $Seq(A^X_{priv})$. Trois opérations de O^X sont toujours définies :

  - $\vdash^X$ est appelée l'*opération d'évolution* de l'agent d'exécution. Elle est responsable de l'exécution de l'agent logiciel. Elle lit le contenu de cases de mémoire de M^X_{pub} dont l'adresse est contenue dans la séquence de cases $PC^X \in Seq(M^X_{priv}, W^X_{size})$. Si ce contenu est le "codage" d'une action publique elle exécute cette action, sinon elle exécute $\perp^X_{code}$.
  - $\perp^X_{code}$ est appelée l'*opération d'erreur de codage*. Elle implémente l'opération qu'effectue l'agent d'exécution lorsqu'un codage d'action publique est invalide.
  - $\perp^X_{addr}$ est appelée l'*opération d'erreur d'adresse*. Elle implémente l'opération qu'effectue l'agent d'exécution lorsqu'une adresse accédée par une action publique n'existe pas.

Notes :
– Pour représenter le fait que l'agent d'exécution X effectue une action publique a à partir de l'instant t, on utilise la relation suivante :

$$(X,t) \vDash a \quad \Leftrightarrow \quad \begin{cases} X \text{ effectue } a \text{ à partir de l'instant } t. \\ \text{si } a \in A^X_{pub} \text{ on utilise aussi la notation : } X \vdash^X_t a \end{cases}$$

– On généralise la relation $\vDash$ (par abus de notation) sur $Seq(A^X_{pub})$[7]. Soit S une séquence d'actions publiques, c'est-à-dire $S \in Seq(A^X_{pub})$:

$$(X,t) \vDash S \quad \Leftrightarrow \quad \text{l'agent d'exécution } X \text{ effectue } S \in Seq(A^X_{pub}) \text{ à partir de l'instant } t.$$

– Remarquons qu'un agent d'exécution X peut être représenté à des niveaux de granularité différents qui dépendent directement de la cardinalité de D^X. Par exemple si $|D^X| = 2$, les actions de l'agent X opère sur des cases de mémoire qui peuvent représenter des bits. En revanche, si $|D^X| = 4096 \times 8$, les actions de l'agent X opèrent sur des cases de mémoire qui peuvent représenter des pages de mémoire d'une architecture x86 (pages de 4 Ko). Suivant l'utilisation que l'on souhaite faire du modèle, il est souhaitable de définir un D^X de cardinalité appropriée.

A.2.2 Agent logiciel

Un *agent logiciel* est défini par un triplet :

$$L = (X, F^L, \Lambda^L)$$

où :
– X est un agent d'exécution.

– F^L correspond à l'ensemble des fonctions de l'agent logiciel[8]. Formellement, il est un sous-ensemble de $Seq(A^X_{pub})$.

– Λ^L correspond à la fonction de localisation de l'agent logiciel sur l'agent d'exécution X. Elle retourne pour chaque séquence de F^L la case de mémoire depuis laquelle débute le codage de la séquence (le codage est défini par γ^X).

A.2.3 Système d'agents

On définit un système d'agents $\mathcal{X}$ comme un ensemble d'agents d'exécution ayant des propriétés d'évolution compatibles, auxquelles se rajoutent des propriétés d'évolution spécifiques

[7] On généralise également $\vDash$ sur $Seq(A^X_{priv})$.

[8] À noter, que contrairement aux *opérations spéciales* de X, nous considérons que les fonctions de l'agent logiciel se situent dans la mémoire de X et peuvent donc être modifiées.

relatives à la cohérence d'évolution du système.

On définit aussi la notion de type d'agents d'exécution, afin de faciliter la définition de systèmes d'agents communicants et/ou qui partagent de la mémoire (représentation de processeur multicœurs, etc.).

Soient $X, Y \in \mathcal{X}$. X a le même type que Y (et inversement) si et seulement si $X \approx_t Y$, où $\approx_t$ est la relation définie par :

$$X \approx_t Y \Leftrightarrow \begin{cases} (\Delta^X = \Delta^Y) \\ \wedge\ (D^X = D^Y) \\ \wedge\ (W^X_{size} = W^Y_{size}) \\ \wedge\ (\text{les cases de } M^X_{priv} \text{ ont la même "sémantique" que celles de } M^Y_{priv}) \end{cases}$$

A.3 Un exemple de système d'agents

Soit X un agent d'exécution tel que $X = (T^X, \Phi^X, (M^X_{priv}, M^X_{pub}), (A^X_{pub}, A^X_{priv}, O^X))$, et soit $\mathcal{X}$ un système d'agents contenant uniquement des agents d'exécution de même type que X.

Nous détaillons dans la suite (partiellement) l'agent X, et donnons quelques précisions sur le système $\mathcal{X}$.

A.3.1 Propriétés d'évolution

A.3.1.1 Propriétés d'évolution de l'agent d'exécution X

Soient $A^X = A^X_{pub} \cup A^X_{priv}$ et $\$^X = \$^X_{pub} \cup \$^X_{pub}$.

Propriété 1 (séquentialité des évènements d'un agent d'exécution)**.**
L'agent d'exécution X ne peut exécuter qu'une action publique à tout instant.

$$\begin{array}{l} \forall t \in T^X\ : \\ \exists_1\ a \in A^X_{pub} \mid (X,t) \vDash a \end{array}$$

Propriété 2 (continuité des actions publiques)**.**
À chaque instant, l'agent d'exécution X exécute une action publique ou privée.

$$\begin{array}{l} \forall t \in T^X\ : \\ (\exists a \in A^X \mid (X,t) \vDash a) \\ \vee\ (\exists t_1 \in T^X,\ \exists a \in A^X \mid (t_1 \leq t) \wedge ((X,t_1) \vDash a) \wedge (t_1 + \$^X(a) > t)) \end{array}$$

Propriété 3 (prise d'effet des modifications de l'état du système)**.**
Les modifications engendrées par l'exécution des actions publiques ne prennent effet qu'à la fin de leur exécution.

$$\begin{array}{l} \forall t_1 \in T^X,\ \forall a(c_1, ...c_n) \in A^X \mid c_i \in M^X \textit{ pour } i = 1..n\ : \\ \textit{Notons } t_2 \textit{ l'instant } t_1 + \$^X(a(c_1, ...c_n)) \\ ((X,t_1) \vDash a) \Rightarrow (\forall t \in < t_1, t_2 >, \forall i \in < 1, n > \mid t \neq t_2\ : \eta(c_i, t) = \eta(c_i, t_1)) \end{array}$$

A.3.1.2 Propriétés d'évolution du système d'agent $\mathcal{X}$

On définit $T = \bigcup_{X \in \mathcal{X}} T^X$.

Propriété 4 (cohérence de l'évolution du système)**.**
L'exécution de plusieurs actions publiques, par des agents d'exécution différents, qui se terminent à un même instant ne peut avoir pour conséquence la modification d'une même case de mémoire.

$$\begin{array}{l}\forall X \in \mathcal{X},\ \forall t^X \in T,\ \forall a(c_1, ...c_n) \in A\ : \\ \textit{Notons } t_e^X \textit{ l'instant } t^X + \$^X(a(c_1, ...c_n)) \\ ((X, t^X) \vDash a)\ \Rightarrow\ [(\exists Z \in \mathcal{X},\ \exists c_i,\ \exists b(c_1, ...c_n) \in A,\ \exists t^Z \in T\ \mid \\ \qquad (((Z, t^Z) \vDash b)\ \wedge\ (t_e^X = t^Z + \$^Z(b(c_1, ...c_n))))\ \Rightarrow\ (\tau_w(c_i) = t_e^X)\,)\ \Longrightarrow\ (X = Z)]\end{array}$$

A.3.2 Cases de mémoire de l'agent d'exécution X

A.3.2.1 Contenus (ou données) possibles des cases

On définit $D^X = \{0, 1, ..., 15\}$[9]. Dans la suite les notations suivantes sont employées :
- $false$ est synonyme du symbole $0 \in D^X$;
- $true$ est synonyme du symbole $1 \in D^X$;
- $user$ est synonyme du symbole $3 \in D^X$;
- $system$ est synonyme du symbole $0 \in D^X$;

A.3.2.2 Cases de mémoire privées

On définit plusieurs séquences de cases de mémoire privées (M^X_{priv}) qui sont employées pour stocker l'état de certaines actions publiques (*cf.* section A.3.3). Pour chaque $Y \in \mathcal{X}$ on définit les séquences :
- $E^X_Y \in Seq(M^X_{priv}, m)$ qui est une séquence (de longueur m, appelé *multiplicité d'entrées*) de cases, nommées *entrées* de l'agent d'exécution X. Les cases de cette séquence sont employées pour la communication entre les agents d'exécution de même type que l'agent X, c'est-à-dire entre les agents de $\mathcal{X}$.
- $L^X_Y \in Seq(M^X_{priv}, m)$ qui est une séquence (de longueur m) de cases, nommées *verrous* de l'agent d'exécution X. Les cases de cette séquence sont employées de concert avec les *entrées* lors de communications *synchronisées* entre les agents d'exécution de même type que l'agent X.

On définit également les séquences de cases suivantes :
- Une séquence PC^X (*Program Counter*) contenant l'adresse de la case où débute le codage de la prochaine action publique à exécuter. Cette séquence de cases privées est employée par l'opération spéciale $\vdash^X$.
- Une séquence SP^X (*Stack Pointer*) qui à chaque instant contient l'adresse du sommet de la pile (consistant en une autre séquence de cases *publiques*) employée pour la sauvegarde

[9] Cela revient à voir M^X comme un ensemble de cases de mémoire de 4 bits.

des adresses de retour de fonction.

Enfin, on définit un ensemble de cases représentant certains états de l'agent d'exécution. Il contient :

- une case CPL^X (*Current Privilege Level*) qui doit refléter le niveau de privilège courant de l'agent d'exécution. La donnée placée dans cette case ($user$ ou $system$) est modifiée par certaines actions publiques (par exemple les interruptions) ou opérations spéciales de O^X ;
- une case F_z^X qui doit être modifiée (dans notre exemple d'agent d'exécution) par l'action publique $comp$ (cette action y écrit la donnée $true$ si les données qu'elle compare sont égales, sinon la donnée $false$, *cf.* section A.3.3) ;
- une case F_s^X qui doit être modifiée par l'action publique $comp$ (cette action y écrit la donnée $true$ si la donnée de son premier opérande est inférieure à celle de son second, sinon la donnée $false$, *cf.* section A.3.3).

A.3.3 Actions publiques de l'agent d'exécution X

La définition de ces actions publiques respecte les propriétés d'évolution Φ^X définies pour l'agent d'exécution X.

On note :

- t_s : l'instant à partir duquel l'action publique est exécutée par l'agent d'exécution X ;
- $t_e = t_s + \$^X_{pub}$(l'action publique exécutée).

Nous présentons dans la suite les actions publiques de l'agent X. Nous les avons regroupées par catégorie :

Les actions publiques représentant les manipulations élémentaires que l'agent d'exécution X peut effectuer sur les cases de mémoire de M^X_{pub} :

Soient $c_1, c_2, c_3 \in M^X_{pub}$ et $d \in D^X$.

- l'initialisation d'une case de mémoire :

$$setbox(c_1, d) \equiv (\eta \oplus ((c_1, t_e) \mapsto d) \wedge (\tau_w \oplus (c_1 \mapsto t_e)))$$

> Note : notre opérateur $\oplus$ correspond à l'idée de l'opérateur de surcharge employé dans Z [Spivey et Abrial, 1992], sauf que dans notre cas il modifie la fonction à sa gauche. Soit une fonction totale $f : X \to Y$ et $(x, y) \in X \times Y$. En Z, $f \oplus (x \mapsto y)$ est la fonction qui est identique à f, à l'exception de sa valeur en x qui est y. Dans notre formalisme, $f \oplus (x \mapsto y)$ *remplace* la fonction f par la fonction qui est identique à f, à l'exception de sa valeur en x qui est y.

– l'initialisation de la case de mémoire c_1 avec d, sous condition que le contenu de c_2 soit $true$:

$$\begin{aligned} setbox_{cond}(c_1, d, c_2) \quad \equiv \quad & (\eta(c_2, t_s) = true \Rightarrow \\ & (\eta \oplus ((c_1, t_e) \mapsto d)) \\ & \wedge (\tau_w \oplus (c_1 \mapsto t_e))) \\ & \wedge (\tau_r \oplus (c_2 \mapsto t_s))) \end{aligned}$$

– l'écriture de c_1 vers c_2 :

$$\begin{aligned} move(c_1, c_2) \quad \equiv \quad & (\eta \oplus ((c_2, t_e) \mapsto \eta(c_1, t_s))) \\ & \wedge (\tau_w \oplus (c_2 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s)) \end{aligned}$$

– l'écriture de c_1 vers c_2 sous condition que le contenu de c_3 soit $true$:

$$\begin{aligned} move_{cond}(c_1, c_2, c_3) \quad \equiv \quad & (\eta(c_3, t_s) = true \Rightarrow \\ & (\eta \oplus ((c_2, t_e) \mapsto \eta(c_1, t_s))) \\ & \wedge (\tau_w \oplus (c_2 \mapsto t_e))) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s) \oplus (c_3 \mapsto t_s)) \end{aligned}$$

– l'écriture dans la case c_1 du contenu de la case dont l'adresse est précisée dans la séquence de cases S (si $\alpha^{-1}(\eta(S, t_s))$ n'est pas défini, l'opération spéciale $\perp_{addr}$ est exécutée) :

$$\begin{aligned} move^{g}_{ind}(S, c_1) \quad \equiv \quad & (\eta \oplus ((c_1, t_e) \mapsto \eta(\alpha^{-1}(\eta(S, t_s)), t_s))) \\ & \wedge (\tau_w \oplus (c_1 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (S \mapsto t_s) \oplus (\alpha^{-1}(\eta(S, t_s)) \mapsto t_s)) \end{aligned}$$

– l'écriture du contenu de la case c_1 dans la case dont l'adresse est précisée dans la séquence de cases S :

$$\begin{aligned} move^{d}_{ind}(c_1, S) \quad \equiv \quad & (\eta \oplus (\alpha^{-1}(\eta(S, t_s)), t_e) \mapsto \eta(c_1, t_s))) \\ & \wedge (\tau_w \oplus (\alpha^{-1}(\eta(S, t_s)) \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s) \oplus (S \mapsto t_s)) \end{aligned}$$

– l'addition du contenu des cases c_1 et c_2, suivie de l'écriture du résultat dans c_2 :

$$\begin{aligned} add(c_1, c_2) \quad \equiv \quad & \eta \oplus ((c_2, t_e) \mapsto \eta(c_1, t_s) + \eta(c_2, t_s)) \\ & \wedge (\tau_w \oplus (c_2 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s) \oplus (c_2 \mapsto t_s)) \end{aligned}$$

– l'addition de la donnée d au contenu de la case c_1 :

$$\begin{aligned} add_1(c_1, d) \quad \equiv \quad & \eta \oplus ((c_1, t_e) \mapsto \eta(c_1, t_s) + d) \\ & \wedge (\tau_w \oplus (c_1 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s)) \end{aligned}$$

– la soustraction du contenu de c_1 avec celui de c_2 suivie de l'écriture du résultat dans c_2 :

$$\begin{aligned} sub(c_1, c_2) \quad \equiv \quad & \eta \oplus ((c_2, t_e) \mapsto \eta(c_1, t_s) - \eta(c_2, t_s)) \\ & \wedge (\tau_w \oplus (c_2 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s) \oplus (c_2 \mapsto t_s)) \end{aligned}$$

– la soustraction de la donnée d au contenu de c_1 :

$$\begin{aligned} sub_1(c_1, d) \quad \equiv \quad & \eta \oplus ((c_1, t_e) \mapsto \eta(c_1, t_s) - d) \\ & \wedge (\tau_w \oplus (c_1 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s)) \end{aligned}$$

– le test d'égalité entre les contenus de c_1 et c_2 suivi de l'écriture du résultat (un booléen) dans c_3 :

$$\begin{aligned} equal(c_1, c_2, c_3) \quad \equiv \quad & (\eta(c_1, t_s) = \eta(c_2, t_s) \Rightarrow (\eta \oplus ((c_3, t_e) \mapsto true)) \\ & \wedge (\eta(c_1, t_s) \neq \eta(c_2, t_s) \Rightarrow (\eta \oplus ((c_3, t_e) \mapsto false) \\ & \wedge (\tau_w \oplus (c_3 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s) \oplus (c_2 \mapsto t_s)) \end{aligned}$$

– le test d'infériorité entre les contenus de c_1 et c_2 suivi de l'écriture du résultat (un booléen) dans c_3 :

$$\begin{aligned} lesser(c_1, c_2, c_3) \quad \equiv \quad & (\eta(c_1, t_s) < \eta(c_2, t_s) \Rightarrow (\eta \oplus ((c_3, t_e) \mapsto true)) \\ & \wedge (\eta(c_1, t_s) \geq \eta(c_2, t_s) \Rightarrow (\eta \oplus ((c_3, t_e) \mapsto false) \\ & \wedge (\tau_w \oplus (c_3 \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s) \oplus (c_2 \mapsto t_s)) \end{aligned}$$

– la comparaison entre les contenus de c_1 et c_2 suivi de l'écriture du résultat dans les case F_s^X et F_z^X :

$$\begin{aligned} comp(c_1, c_2) \quad \equiv \quad & (\eta(c_1, t_s) = \eta(c_2, t_s) \Rightarrow (\eta \oplus ((F_z^X, t_e) \mapsto true) \oplus ((F_s^X, t_e) \mapsto false)) \\ & \quad \wedge (\eta(c_1, t_s) < \eta(c_2, t_s) \Rightarrow (\eta \oplus ((F_z^X, t_e) \mapsto false) \oplus ((F_s^X, t_e) \mapsto true)) \\ & \quad \wedge (\eta(c_1, t_s) > \eta(c_2, t_s) \Rightarrow (\eta \oplus ((F_z^X, t_e) \mapsto false) \oplus ((F_s^X, t_e) \mapsto false)) \\ & \wedge (\tau_w \oplus (F_z^X \mapsto t_e) \oplus (F_s^X \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c_1 \mapsto t_s) \oplus (c_2 \mapsto t_s)) \end{aligned}$$

Il est également possible de définir des actions publiques travaillant sur des séquences de cases de mémoire[10] :

Soient $c_1, c_2 \in M_{pub}^X$ et soit la séquence de cases $S \in Seq(M_{pub}^X)$ de longueur W_{size}^X.

– écriture de l'adresse de la case c_1 dans la séquence de cases S :

$$\begin{aligned} S\_setaddr(S, c_1) \quad \equiv \quad & (\eta \oplus ((S, t_e) \mapsto \alpha(c_1)) \\ & \wedge (\tau_w \oplus (S \mapsto t_e)) \end{aligned}$$

[10] Il faut également prendre en compte cela au niveau des propriétés d'évolution du système.

Note : la notation $\eta \oplus ((S, t_e) \mapsto \alpha(c_1))$ correspond à :

$\eta \oplus ((S[1], t_e) \mapsto \alpha(c_1)[1]) \oplus ((S[2], t_e) \mapsto \alpha(c_1)[2]) \oplus ... \oplus ((S[W^X_{size}], t_e) \mapsto \alpha(c_1)[W^X_{size}])$

– écriture de l'adresse de la case c_1 dans la séquence de cases S, sous condition que le contenu de c_2 soit $true$:

$$\begin{aligned} setbox_{cond}(S, c_1, c_2) \quad \equiv \quad & (\eta(c_2, t_s) = true \Rightarrow \\ & (\eta \oplus ((S, t_e) \mapsto \alpha(c_1))) \\ & \wedge (\tau_w \oplus (S \mapsto t_e))) \\ & \wedge (\tau_r \oplus (c_2 \mapsto t_s)) \end{aligned}$$

Les actions publiques représentant les manipulations élémentaires que l'agent d'exécution X peut effectuer sur les *entrées* des agents du même type que lui :

Soit c une case de mémoire de l'agent d'exécution X, soit Y un autre agent d'exécution du système, et soit $i \in \mathbb{N} \mid i < m$ (m étant la multiplicité d'entrées de X).

– l'envoi par X du contenu de sa case c à l'agent d'exécution Y dans sa i-ème entrée associée à X :

$$\begin{aligned} out_Y(c, i) \quad \equiv \quad & (\eta \oplus ((E^Y_X[i], t_e) \mapsto \eta(c, t_s))) \\ & \wedge (\tau_w \oplus (E^Y_X[i] \mapsto t_e)) \\ & \wedge (\tau_r \oplus (c \mapsto t_s)) \end{aligned}$$

– la lecture par X de sa i-ème entrée associée à Y, suivie de l'écriture de son contenu dans la case de mémoire c :

$$\begin{aligned} in_Y(i, c) \quad \equiv \quad & (\eta \oplus ((c, t_e) \mapsto \eta(E^X_Y[i], t_s))) \\ & \wedge (\tau_w \oplus (c \mapsto t_e)) \\ & \wedge (\tau_r \oplus (E^X_Y[i] \mapsto t_s)) \end{aligned}$$

– l'envoi "synchronisé" par X du contenu de la case c à l'agent d'exécution Y dans sa i-ème entrée associée à X :

$$\begin{aligned} out^{sync}_Y(c, i) \quad \equiv \quad & (\eta(L^Y_X[i], t_s) = false \Rightarrow \\ & (\eta \oplus ((E^Y_X[i], t_e) \mapsto \eta(c, t_s)) \oplus ((L^Y_X[i], t_e) \mapsto true)) \\ & \wedge (\tau_w \oplus (E^Y_X[i] \mapsto t_e) \oplus (L^Y_X[i] \mapsto t_e))) \\ & \wedge (\tau_r \oplus (c \mapsto t_s)) \end{aligned}$$

– la lecture "synchronisée" par X de sa i-ème entrée associée à Y, suivie de l'écriture de son contenu dans la case de mémoire c :

$$\begin{aligned} in^{sync}_Y(i, c) \quad \equiv \quad & (\eta(L^X_Y[i], t_s) = true \Rightarrow \\ & (\eta \oplus ((c, t_e) \mapsto \eta(E^X_Y[i], t_s)) \oplus ((L^X_Y[i], t_e) \mapsto false)) \\ & \wedge (\tau_w \oplus (c \mapsto t_e) \oplus (L^X_Y[i] \mapsto t_e))) \\ & \wedge (\tau_r \oplus (E^X_Y[i]) \mapsto t_s)) \end{aligned}$$

Les actions publiques représentant le contrôle que l'agent d'exécution X peut avoir sur son exécution, au travers de la modification de son compteur ordinal PC^X :

Soient $c_1, c_2 \in M^X_{pub}$.

– l'exécution de l'action publique codée à partir de c_1 :

$$jump(c_1) \equiv S\_setaddr(PC^X, c_1)$$

– l'exécution de l'action publique codée à partir de c_1, sous condition que la donnée à c_2 soit $true$:

$$jump_{cond}(c_1, c_2) \equiv S\_setaddr_{cond}(PC^X, c_1, c_2)$$

– l'exécution de l'action publique codée à partir de c_1, sous condition que la donnée dans F^X_z soit $true$:

$$jump_e(c) \equiv setbox_{cond}(PC^X, \eta(c_1), F^X_z)$$

– l'exécution de l'action publique codée à partir de c_1, sous condition que la donnée dans F^X_s soit $true$:

$$jump_l(c) \equiv setbox_{cond}(PC^X, \alpha(c_1), F^X_s)$$

– l'appel d'une séquence d'actions publiques codées à partir de c_1, et sauvegarde de l'adresse de retour :

$$call(c_1) \equiv sub_1(SP^X, 1) \bullet move^d_{ind}(PC^X, SP^X)^{11} \bullet jump(c_1)$$

– le retour d'un appel de séquence d'actions publiques :

$$ret \equiv move^g_{ind}(SP^X, PC^X)^{12}$$

Nous donnons ci-dessous un exemple d'action publique qui change de comportement suivant le niveau de privilège actuel de l'agent d'exécution X (c'est-à-dire suivant le contenu de la case CPL^X) :

Nous considérons l'existence sur X d'une séquence de cases privées S_{gdtr}. L'opération $lgdt$, définie ci-dessous, inscrit l'adresse de la case passée en opérande dans la séquence S_{gdtr} si la case CPL^X a pour contenu $system$. Sinon, elle déclenche une "exception". Les modifications de M^X effectuées par $lgdt$ pour déclencher une exception sont similaires à celles de l'action publique d'interruption int_X définie dans la section A.3.4. L'action $lgdt$ n'ayant qu'une valeur d'exemple, nous simplifions sa définition en employant directement l'action int_X dans sa définition.

[11] À noter qu'il s'agit d'une variation de l'action $move^d_{ind}$ dont les deux opérandes sont des séquences.
[12] Même remarque que pour l'annotation 11.

$$
\begin{aligned}
lgdt(c) \quad \equiv \quad & (\eta(CPL^X, t_s) = system \Rightarrow \\
& \quad (\eta \oplus ((S_{gdtr}, t_e) \mapsto \alpha(c))) \\
& \quad \wedge ((\tau_w \oplus (S_{gdtr} \mapsto t_e)) \ \wedge \ (\tau_r \oplus (CPL^X \mapsto t_e)))) \\
& \wedge (\eta(CPL^X, t_s) \neq system \Rightarrow int_X \\
& \quad \wedge (\tau_r \oplus (CPL^X \mapsto t_e)))
\end{aligned}
$$

A.3.4 Opérations spéciales de l'agent d'exécution X

Dans la suite nous précisons la définition de certaines opérations spéciales de l'agent X. Ces descriptions sont informelles pour la plupart. Pour les autres nous employons des actions privées que nous ne définissons pas mais qui ont la même sémantique que les actions publiques définies dans la section A.3.3.

A.3.4.1 Opération d'évolution : $\vdash^X$

Nous donnons une description informelle de l'opération d'évolution $\vdash^X$. Cette opération effectue successivement les étapes suivantes :

1. décodage de l'action codée à partir de PC^X, puis :
   - si le code n'est pas connu : branchement sur l'opération $\perp^X_{code}$;
   - si le code est connu : exécution de l'action publique correspondante ;
2. traitement des interruptions (*cf.* le paragraphe sur le mécanisme des interruptions) ;
3. retour à l'étape 1.

A.3.4.2 Opération d'erreur de codage : $\perp^X_{code}$

Nous donnons une description informelle de l'opération d'erreur de codage $\perp^X_{code}$. Cette opération effectue successivement les étapes suivantes :

1. écriture dans PC^X de l'adresse d'une case privée à partir de laquelle est codée une opération privée qui déclenche une exception[13] ;
2. branchement sur l'opération $\vdash^X$.

A.3.4.3 Mécanisme d'interruptions

On définit dans la suite la notion d'interruption entre deux agents d'exécution du système d'agents $\mathcal{X}$.

Tout agent d'exécution $X \in \mathcal{X}$ a la possibilité d'interagir avec un autre agent d'exécution $Y \in \mathcal{X}$ par le biais d'un mécanisme d'interruption[14]. L'utilisation de l'action privée int_Y par X envoie une requête d'interruption à l'agent d'exécution Y.

[13] Nous appelons "exception", une interruption déclenchée par un agent d'exécution à destination de ce même agent d'exécution.

[14] Notons que X peut être égal à Y. Les interruptions sont alors les "exceptions" de X.

Afin de définir ce mécanisme d'interruption pour tout agent $X \in \mathcal{X}$, ses cases $E_Y^X[1]$ et $L_Y^X[1]$ sont employées (pour tout $Y \in \mathcal{X} \mid X \neq Y$), ainsi que ses cases $E_X^X[1..m]$ et $L_X^X[1..m]$[15]. Nous supposons également que ces cases sont initialisées à $false$. C'est-à-dire qu'on a :

$$\forall X, Y \in \mathcal{X} \quad \text{avec } t_0 \in T = \bigcup_{X \in \mathcal{X}} T^X \mid (\nexists t \in T \mid t < t_0) \ :$$

$$\eta(E_Y^X[1], t_0) = false \ \wedge \ \eta(L_Y^X[1], t_0) = false \\ \wedge \eta(E_X^X[2..m], t_0) = (false)^{m-1} \ \wedge \ \eta(L_X^X[2..m], t_0) = (false)^{m-1}$$

On définit également des cases privées de X dédiées à l'opération d'interruption : la case privée I_{true}^X initialisée à $true$, et pour tout agent $Y \in \mathcal{X}$, les cases $ItRequest_Y^X$ initialisée à $false$.

On représente alors l'émission d'une interruption par un agent d'exécution $X \in \mathcal{X}$ à un agent d'exécution $Y \in \mathcal{X}$ par l'écriture de la donnée $true$ dans l'entrée $E_X^Y[1]$ via l'action privée out_Y^{sync}[16]. On définit pour cela l'action publique suivante :

$$int_Y \quad \equiv \quad out_Y^{sync}(I_{true}^X, 1)$$

La prise en compte par Y de l'interruption émise par X passe alors par la lecture de son entrée $E_X^Y[1]$. Nous utilisons pour cela l'opération in_X^{sync} de Y et sa case $ItRequest_X^Y$.

Nous donnons dans la suite la séquence d'actions privées effectuées par Y[17] pour vérifier l'occurrence d'une interruption par X, et exécuter le cas échéant, le gestionnaire d'interruption approprié, représenté par une séquence d'actions publiques dont le codage débute à partir de la case $Handler_X^Y$[18] :

$$setbox(false, ItRequest_X^Y) \circ in_X^{sync}(1, ItRequest_X^Y) \circ comp(ItRequest_X^Y, I_{true}^Y) \circ jump_e(Handler_X^Y)$$[19]

Les "verrous", qui régissent le mode opératoire des actions publiques in_X^{sync} et out_Y^{sync}, empêchent les modifications indues par ces actions des cases $ItRequest_X$. L'exécution de in_X^{sync} par Y ne modifie $ItRequest_X$ que si l'action out_Y^{sync} a d'abord été exécuté par X. De même, l'exécution de out_Y^{sync} par X n'entraînera aucune modification de la case $ItRequest_X$ de Y, si Y n'a pas exécuté in_X^{sync} depuis la dernière exécution d'un out_Y^{sync} par X. Ce mécanisme de verrous garantit ainsi un entrelacement de out_Y^{sync} (production) et de in_X^{sync} (consommation) pour tout couple d'agents d'exécution de $\mathcal{X}$.

Notons toutefois que la solution proposée n'est pas complète. En effet des traitements supplémentaires sont à effectuer. Il s'agit, dans le cas où l'interruption est à traiter (par Y), du

[15] Ces cases sont employées pour les "exceptions" de l'agent d'exécution X.

[16] Remarquons que le mécanisme de synchronisation via les "verrous" employés par les actions out_Y^{sync} de X et in_X^{sync} de Y, est adéquat à la représentation des interruptions.

[17] Cette séquence d'actions est employée par l'opération d'évolution $\vdash^X$.

[18] Pour chaque agent $X \in \mathcal{X}$ nous définissons sur Y des séquences de cases dont la première est désignée par $Handler_X^Y$ et dont l'objet est de contenir le codage de séquences d'actions publiques qui assurent le traitement des interruptions.

[19] Notons que la loi de composition $\circ$ sur les actions privées est définie de la même façon que la loi $\bullet$ sur les actions publiques.

passage à un CPL^Y de niveau $system$ et de la sauvegarde du contexte courant (PC^Y, F_z^Y, F_s^Y) dans la pile à la position pointée par SP^Y. De plus, il est nécessaire de définir une autre action publique ($iret$) qui doit être présente à la fin d'un gestionnaire d'interruption et dont l'exécution permet de recharger le contexte sauvegardé dans la pile et de passer le CPL^Y au niveau $user$.

A.4 Extension du modèle : les calques

Un calque est une extension au modèle qui apporte un ou plusieurs concepts nécessaires au traitement d'un problème particulier.

A.4.1 Le calque des interactions entre actions publiques

Soit $\mathcal{X}$ un système d'agents, et soient X_1 et X_2 deux agents d'exécution de ce système (qui répondent aux propriétés d'évolution définies pour notre exemple X, *cf.* section A.3.1), qui peuvent être éventuellement les mêmes.

La relation suivante, représente l'interaction entre X_1 et X_2, lesquels exécutent respectivement les séquences d'actions publiques S_1 à t_1 et S_2 à t_2.

Soient $T = T^{X_1} \cup T^{X_2}$, $M = M^{X_1} \cup M^{X_2}$, et $A_{pub} = A_{pub}^{X_1} \cup A_{pub}^{X_2}$.

$$(X_1, t_1, S_1)\ \mathcal{I}(t)\ (X_2, t_2, S_2) \quad \Leftrightarrow \quad \begin{cases} (\exists t_1, t_2 \in T \mid (t \leq t_1 \ \wedge\ t \leq t_2) \\ \quad \wedge ((X_1, t_1) \vDash S_1) \ \wedge\ ((X_2, t_2) \vDash S_2)) \\ \wedge\ (\exists\ a \in S_1, b \in S_2 \mid a|_t b \vee b|_t a) \end{cases}$$

$a|_t b$ représente le fait que l'action publique a "communique" avec l'action publique b à partir de t. La définition de $a|_t b$ est donnée ci-dessous :

$$a\ |_t\ b \quad \Leftrightarrow \quad \begin{cases} (1)\ \exists X, Y \in \mathcal{X},\ t_2 \in T \mid (t_2 \geq \underbrace{t + \$_{pub}^X(a)}_{t_1}) \wedge\ (X, t, a), (Y, t_2, b) \in \vDash \\ \wedge\ (2)\ \exists c \in M \mid ((X, t) \vDash a) \Rightarrow \tau_w(c) = t_1 \quad \text{(c'est-à-dire "}c\text{ a été accédée en écriture")} \\ \qquad (Y, t_2)\ \vDash\ b \Rightarrow \tau_r(p) = t_2 \quad \text{(c'est-à-dire "}c\text{ a été accédée en lecture")} \\ \wedge\ (3)\ \forall t_i \in < t_1, t_2 > : \eta(c, t_i) = \eta(c, t_1) \end{cases}$$

On peut également exprimer (3) de la façon suivante :

$$\forall a \in A, \forall t_0 \in T, \nexists Z \in \mathcal{X} \mid (t_1 < \underbrace{t_0 + \$_{pub}^Z(a)}_{t_i} < t_2) \wedge\ (((Z, t_i) \vDash a) \Rightarrow \eta(c, t_i) \neq \eta(c, t_1))$$

Une caractérisation de $a|_t b$ dans le cas où a et b sont l'action publique $move$ (de sémantique équivalente à celle donnée pour notre exemple d'agent X, *cf.* section A.3.3) est donnée ci-dessous :

$$\underbrace{move(c_1,c_2)}_{m_1} \mid_t \underbrace{move(c_3,c_4)}_{m_2} \Leftrightarrow \begin{cases} c_2 = c_3 \text{ ($\mid_t$ est non commutatif)} \\ \wedge\, \exists X, Y \in \mathcal{X},\ t_2 \in T \mid \\ \quad (t_2 \geq \underbrace{t + \$^X_{pub}(m_1)}_{t_1}) \wedge\ (X,t,m_1),(Y,t_2,m_2) \in \vDash \\ \wedge\, \forall a \in A_{pub}, \forall t_0 \in T, \nexists Z \in \mathcal{X} \mid [(t_1 < \underbrace{t_0 + \$^Z_{pub}(a)}_{t_i} < t_2) \\ \qquad \wedge\, (((Z,t_i) \vDash a)\ \Rightarrow \eta(c_1,t_i) \neq \eta(c_1,t_1))] \end{cases}$$

Via la relation $\mid_t$, on peut définir pour un intervalle de temps $< t_s, t_e > \subset \bigcup_{X \in \mathcal{X}} T^X$, le graphe non-ordonné et valué $G =< V, E, f >$, où les sommets (V) correspondent à des séquences d'actions publiques exécutées par les agents d'exécution, les arêtes (E) relient les sommets ayant des interactions, et où la fonction de valuation (f) indique pour chaque arête les instants où il y a eu des interactions entre les deux sommets reliés.

On a alors :

$$V = \bigcup_{X \in \mathcal{X}} \vdash^X_{<t_s,t_e>} \qquad V \subset (\mathcal{X} \times < t_s, t_e > \times Seq(A_{pub}))$$

$$E = \bigcup_{t \in <t_s,t_e>} \mathcal{I}(t)$$

$$\begin{array}{rccl} f: & E & \longrightarrow & \mathcal{P}(T) \\ & x & \longmapsto & \{t \mid x \in \mathcal{I}(t)\} \end{array}$$

Ce graphe pourrait aider à déterminer les interactions qui peuvent être qualifiées de dangereuse pour l'évolution du système, au sens où elles peuvent nuire à l'intégrité du système.

A.5 Conclusion

Nous avons présenté dans cette annexe notre tentative de modélisation de systèmes informatiques. Tout d'abord, nous nous sommes focalisés sur les concepts centraux de notre modélisation, à savoir : les agents d'exécution, les agents logiciels, et les systèmes d'agents. Ensuite, nous avons décrit partiellement un exemple simple de système d'agents. Enfin, nous avons abordé le concept d'interaction entre agents et nous avons défini une extension à notre modèle pour représenter ce concept.

Nos travaux n'ont malheureusement pas été menés à terme. Nous avons jugé au cours de notre modélisation qu'il n'était pas envisageable d'obtenir des résultats satisfaisants dans un temps raisonnable. Nous essayons à présent d'en analyser les raisons.

Tout d'abord la définition d'un agent d'exécution utile à l'analyse de problèmes de sécurité de noyaux est un travail conséquent, d'autant plus si l'on considère les aspects matériels que nous souhaitions étudier (les technologies de virtualisation). Nous aurions alors dû modéliser dans notre formalisme, entres autres :

- le concept de mémoire virtuelle et par conséquent l'unité matérielle de gestion de la mémoire (MMU) ;
- les opérations VMX (issue de la technologie Intel VT que nous avons étudiée) ;
- etc.

Suite à ce travail, il apparaissait alors nécessaire de concevoir un logiciel qui réponde à différents besoins : la vérification de la cohérence de la définition d'un quelconque agent d'exécution, la simulation de l'exécution d'un tel agent et/ou l'analyse des exécutions possibles afin de vérifier certaines propriétés sur ces exécutions (respect d'invariants, de contraintes, etc.).

Enfin, il serait souhaitable d'étendre un tel logiciel pour vérifier des propriétés relatives aux interactions. En effet, nous souhaitions qu'il puisse générer un graphe d'interactions (comme spécifié dans la section A.4) entre agents logiciels et agents d'exécution, sur une tranche de temps donnée. Ce graphe devrait alors être exploité pour la vérification de ces propriétés (par exemple, vérifier le respect de contraintes sur certains objets du système[20]). Toutefois la génération de ce graphe nous est apparue comme problématique, en raison de l'explosion combinatoire probable du traitement. Nous aurions pu, éventuellement, générer un tel graphe pour un agent d'exécution modélisé à un niveau d'abstraction élevé. Toutefois, nous aurions encouru le risque que le graphe ne permette plus de vérifier les propriétés qui nous auraient intéressés (par son manque de finesse). Ainsi nous aurions dépensé un temps précieux dans la définition d'un agent d'exécution qui au final n'aurait pu répondre à nos attentes. Une autre approche aurait été de conserver une définition d'agent d'exécution relativement fine, mais de construire des graphes de façon "intelligente" en éliminant *a priori* des sommets et/ou des arêtes, suivant leur pertinence pour la vérification des propriétés considérées. Toutefois, l'automatisation d'une telle approche reste encore un problème délicat.

Pour le moment cette modélisation n'atteint donc pas les objectifs que nous nous étions fixés. En particulier, il semble difficile d'exprimer des propriétés d'assez haut niveau telles que l'intégrité vis-à-vis d'attaques, mais également de pouvoir dériver d'un tel modèle des contraintes à imposer à certains objets (typiquement des KCO).

[20] Ces objets étant représentés par des séquences de cases de mémoire dans notre modélisation.

Bibliographie

D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu et J. Wiegert : Intel virtualization technology for directed I/O. *Intel technology journal*, 10(3):179–191, 2006.

A. Albinet, J. Arlat et J. Fabre : Characterization of the Impact of Faulty Drivers on the Robustness of the Linux Kernel. *In Proceedings of the International Conference on Dependable Systems and Networks*, pages 867–876, Florence, Italie, 2004.

X. Allamigeon et C. Hymans : Analyse Statique par Interprétation Abstraite : Application à la Détection de Dépassement de Tampon. *In* Eric Filiol, éditeur : *Actes du 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'07)*, pages 347–384, Rennes, France, juin 2007. École Supérieure et d'Application des Transmissions.

X. Allamigeon et C. Hymans : Static Analysis by Abstract Interpretation : Application to the Detection of Heap Overflows. *Journal in Computer Virology*, 4(1):5–23, 2008.

AMD : AMD64 Virtualization Codenamed "Pacifica" Technology - Secure Virtual Machine Architecture Reference Manual. Rapport technique, Advanced Micro Devices, Inc., 2005.

Z. Amsden, D. Arai, D. Hecht, A. Holler, P. Subrahmanyam et I. VMware : VMI : An interface for paravirtualization. *In Proceedings of the 2006 Linux Symposium*, Ottawa, Canada, juillet 2006.

Anonymous : Once upon a free()... *Phrack*, 57, 2001.

ANSSI : Expression des Besoins et Identification des Objectifs de Sécurité (EBIOS). Rapport technique, Agence Nationale de la Sécurité des Systèmes d'Information, 51 boulevard de La Tour - Maubourg - 75700 PARIS 07 SP, 2004.

J.-M. Autebert : *Théorie des langages et des automates*. MASSON, 1994. ISBN 2-225-84001-6.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt et A. Warfield : Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

D. Bell et L. LaPadula : Secure Computer Systems : Mathematical Foundations. Rapport technique M74-244, MITRE Co., octobre 1974.

F. Bellard : QEMU, a fast and portable dynamic translator. *In Proceedings of the 2005 USENIX Annual Technical Conference - FREENIX Track*, pages 41–46, Anaheim, CA, USA, avril 2005.

K. Biba : Integrity Considerations for Secure Computer Systems. Rapport technique ESD-TR 76-372, MITRE Co., avril 1977.

Bioforge : Hacking the Linux Kernel Network Stack. *Phrack*, 61, 2003.

Black Light : F-secure, Inc. `http://www.f-secure.com/blacklight`.

A. Boileau : Hit by a Bus : Physical Access Attacks with Firewire. *In Ruxcon 2006*, Sydney, Australie, 2006. `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf`.

D. Bovet et M. Cesati : *Understanding the Linux kernel*. O'Reilly & Associates, 3ème édition, 2005.

BSDaemon, Coideloko et D0nAnd0n : System Management Mode Hacks. *Phrack*, 65, 2008.

Buffer : Hijacking Linux Page Fault Handler. *Phrack*, 61, 2003.

Bulba et Kil3r : Bypassing StackGuard and StackShield. *Phrack*, 56, 2000.

J. Butler et G. Hoglund : VICE–catch the hookers. *In Black Hat USA*, Las Vegas, USA, 2004.

L. Butti et J. Tinnes : Recherche de vulnerabilites dans les drivers 802.11 par techniques de fuzzing. *In* Eric Filiol, éditeur : *Actes du 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'07)*, pages 85–106, Rennes, France, juin 2007. École Supérieure et d'Application des Transmissions.

c0de : Reverse symbol lookup in Linux kernel. *Phrack*, 61, 2003.

M. Caceres : Syscall Proxying - Simulating remote execution. Rapport technique, Core Security Technologies, 2002. `http://www.coresecurity.com/files/attachments/SyscallProxying.pdf`.

C. Cachin : An information-theoretic model for steganography. *In Proceedings of the 2nd International Workshop*, volume 1525, pages 306–318, 1998.

S. Cesare : Kernel Function Hijacking, 1999a. `http://vx.netlux.org/lib/vsc08.html`.

S. Cesare : Syscall redirection without modifying the syscall table, 1999b. `http://vx.netlux.org/lib/vsc05.html`.

D. Chen, S. Garg et K. Trivedi : Network survivability performance evaluation : : a quantitative approach with applications in wireless ad-hoc networks. *In Proceedings of the 5th ACM international workshop on Modeling analysis and simulation of wireless and mobile systems*, pages 61–68. ACM New York, NY, USA, 2002.

A. Chou, J. Yang, B. Chelf, S. Hallem et D. Engler : An empirical study of operating systems errors. *In Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88. ACM New York, NY, USA, 2001.

J. Cihula : Trusted Boot : Verifying the Xen Launch. *In Xen Summit*, Santa Clara, CA, USA, novembre 2007.

D. Clark et D. Wilson : A Comparison of Commercial and Military Computer Security Policies. *In Proceedings of the 1987 Symposium on Research in Security and Privacy, IEEE society Press*, pages 184–194, Oakland, Californie, mai 1987.

E. Clarke : Model checking. *In Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56, London, UK, 1997. Springer-Verlag.

E. Clarke, E. Emerson et A. Sistla : Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

B. Cogswell et M. Russinovich : RootkitRevealer 1.71, novembre 2006. `http://technet.microsoft.com/en-us/sysinternals/bb897445.aspx`.

F. Cohen : *Computer Viruses*. Thèse de doctorat, University of Southern California, janvier 1986.

J. Corbet : e1000e and the joy of development kernels. *LWN.net*, septembre 2008a. `http://lwn.net/Articles/300202/`.

J. Corbet : The rest of the vmsplice() exploit story, mars 2008b. `http://lwn.net/Articles/271688/`.

J. Corbet : vmsplice() : the making of a local root exploit, février 2008c. `http://lwn.net/Articles/268783/`.

R. Cori et D. Lascar : *Logique mathématique*, volume 2. DUNOD, 2003. ISBN 2-10-005453-8.

P. Cousot et R. Cousot : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM New York, NY, USA, 1977.

Coverity : Coverity, Inc. `http://www.coverity.com`.

C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang et H. Hinton : StackGuard : Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. *In Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, janvier 1998.

C. Cowan, S. Beattie, J. Johansen et P. Wagle : PointGuard : Protecting Pointers From Buffer Overflow Vulnerabilities. *In Proceedings of the 12th USENIX Security Symposium - Technical Sessions*, pages 91–104, Washington DC, USA, août 2003.

I. D. Craig : *Formal Models of Operating System Kernels*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 1-84628-375-2.

Crazylord : Playing with Windows /dev/(k)mem. *Phrack*, 59, juillet 2002.

J. Criswell, A. Lenharth, D. Dhurjati et V. Adve : Secure virtual architecture : A safe execution environment for commodity operating systems. *ACM SIGOPS Operating Systems Review*, 41(6):351–366, 2007.

M. Dacier et Y. Deswarte : Privilege Graph : An Extension to the Typed Access Matrix Model. *In Proceedings of the third European Symposium on Research in Computer Security (ESORICS'94)*, pages 317–334, Brighton, United Kingdom, novembre 1994.

B. des Ligneris : Virtualization of Linux based computers : the Linux-VServer project. *In Proceedings of the 19th International Symposium on High Performance Computing Systems and Applications (HPCS 2005)*, pages 340–346, Guelph, Canada, 2005.

A. Desnos, E. Filiol et I. Lefou : Detecting (and creating !) a HVM rootkit (aka BluePill-like). *Journal in Computer Virology*, 2009. `http://www.springerlink.com/content/k7717483424n1h41/`.

C. Devine et G. Vissian : Compromission physique par le bus PCI. *In* Eric Filiol, éditeur : *Actes du 7ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'09)*, pages 168–192, Rennes, France, juin 2009. École Supérieure et d'Application des Transmissions.

J. Dike : User-mode Linux. *In Proceedings of the 5th annual Linux Showcase & Conference*, pages 3–14, Oakland, CA, USA, novembre 2001. USENIX Association Berkeley, CA, USA.

I. Dobrovitski : Exploit for CVS double free() for Linux pserver, février 2003. `http://seclists.org/bugtraq/2003/Feb/42`.

DoD : Trusted Computer Security Evaluation Criteria (TCSEC). Rapport technique, Departement of Defense, USA, décembre 1985.

M. Dornseif *et al.* : FireWire - all your memory are belong to us. *In CanSecWest Applied Security Conference*, Vancouver, Canada, 2005. `http://md.hudora.de/presentations/#firewire-cansecwest`.

S. Dralet et F. Gaspard : Corruption de la Mémoire lors de l'Exploitation. *In* Eric Filiol, éditeur : *Actes du 4ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'06)*, pages 362–399, Rennes, France, juin 2006. École Supérieure et d'Application des Transmissions.

U. Drepper : What every programmer should know about memory. *LWN.net*, 2007. `http://people.redhat.com/drepper/cpumemory.pdf`.

L. Duflot : CPU Bugs, CPU Backdoors and Consequences on Security. *In Proceedings of the 13th European Symposium on Research in Computer Security : Computer Security*, pages 580–599. Springer-Verlag Berlin, Heidelberg, 2008.

L. Duflot et L. Absil : Programmed I/O accesses : a threat to Virtual Machine Monitors ? *In PacSec Applied Security Conference*, 2007.

L. Duflot, D. Etiemble et O. Grumelard : Using CPU System Management Mode to Circumvent Operating System Security Functions. *In CanSecWest Applied Security Conference*, Vancouver, Canada, 2006a.

L. Duflot, D. Etiemble et O. Grumelard : Utiliser les fonctionnalités des cartes mères ou des processeurs pour contourner les mécanismes de sécurité des systèmes d'exploitation. *In* Eric Filiol, éditeur : *Actes du 4ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'06)*, Rennes, France, juin 2006b. École Supérieure et d'Application des Transmissions.

L. Duflot et O. Levillain : ACPI et routine de traitement de la SMI : des limites à l'informatique de confiance ? *In* Eric Filiol, éditeur : *Actes du 7ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'09)*, pages 131–167, Rennes, France, juin 2009. École Supérieure et d'Application des Transmissions.

L. Duflot, O. Levillain, B. Morin et O. Grumelard : Getting into the SMRAM : SMM reloaded. *In CanSecWest Applied Security Conference*, Vancouver, Canada, 2009.

S. Embleton, S. Sparks et C. Zou : SMM rootkits : a new breed of OS independent malware. *In Proceedings of the 4th international conference on Security and privacy in communication netowrks table of contents*. ACM New York, NY, USA, 2008.

M. Ernst : Static and dynamic analysis : Synergy and duality. *In Proceedings of the WODA 2003 : ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, USA, 2003.

European Communities : *Information Technology Security Evaluation Criteria*. Office for Official Publications of the European Communities, juin 1991.

European Communities : *Manuel d'Évaluation de la Sécurité des Technologies de l'Information*. Commission des Communautés Européennes, septembre 1993.

E. Filiol : Strong cryptography armoured computer viruses forbidding code analysis. *In Proceedings of the 14th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, pages 216–227, StJuliens/Valletta - Malta, 2005.

E. Filiol et S. Josse : A statistical model for undecidable viral detection. *Journal in Computer Virology*, EICAR 2007 Special Issue, V. Broucek ed., 3(2):65–74, 2007.

E. Filiol : *Les virus informatiques : théorie, pratique et application*. Collection IRIS. Springer Verlag France, 2004.

E. Filiol : Formal Model Proposal for (Malware) Program Stealth. *In Proceedings of the 17th Virus Bulletin Conference*, Vienne, Autriche, septembre 2007a.

E. Filiol : *Techniques virales avancées*. Collection IRIS. Springer Verlag France, 2007b.

J. Filliatre : Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(04):709–745, 2003.

W. Fokkink : *Introduction to process alegebra*. Texts in Theoretical Computer Science. Springer, 2000. ISBN 3-540-66579-X.

H. Fritsch : Analysis and detection of virtualization-based rootkits. Mémoire de master, Institut für Informatik der Ludwig-Maximilians-Universität München, août 2008. `http://www.nm.ifi.lmu.de/pub/Fopras/frit08/PDF-Version/frit08.pdf`.

T. Garfinkel et M. Rosenblum : A Virtual Machine Introspection Based Architecture for Intrusion Detection. *In Proceedings of the Network and Distributed Systems Security Symposium (NDSS'03)*, San Diego, CA, USA, 2003.

C. G. Girling : Covert Channels in LAN's. *IEEE Transaction on Software Engineering*, février 1987.

GNU grep : Free Software Foundation, Inc. `http://www.gnu.org/software/grep`.

P. Godefroid, M. Levin et D. Molnar : Automated whitebox fuzz testing. *In Proceedings of the Network and Distributed System Security Symposium*, 2008.

R. Goldberg : Architecture of virtual machines. *In Proceedings of the workshop on virtual computer systems table of contents*, pages 74–112. ACM New York, NY, USA, 1973.

M. Gorman : *Understanding the Linux virtual memory manager*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2007.

Grugq : Remote Exec. *Phrack*, 62, 2004.

L. H. : Linux Kernel Heap Tampering Detection. *Phrack*, 66, juin 2009.

M. Harrison, W. Ruzzo et J. Ullman : Protection in Operating Systems. *Communications of the ACM*, 19(8):461–471, août 1976.

J. Heasman : Implementing and detecting a PCI rootkit. White paper. novembre 2006a. `http://packetstorm.linuxsecurity.com/papers/general/Implementing_And_Detecting_A_PCI_Rootkit.pdf`.

J. Heasman : Implementing and detecting an ACPI BIOS rootkit. *In Black Hat Federal*, Washington DC, USA, novembre 2006b. `http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf`.

G. Hoglund et G. McGraw : *Exploiting Software - How to break code*. Pearson Education. Addison-Wesley, 2004.

Honeynet Project : Know Your Enemy : Sebek - A kernel based data capture tool, 2003. `http://www.honeynet.org/papers/sebek.pdf`.

C. Hymans : Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation. *In* M. V. Hermenegildo et G. Puebla, éditeurs : *Proceedings of the Static Analysis, 9th International Symposium*, volume 2477 de *Lecture Notes in Computer Science*, pages 444–460, Madrid, Espagne, septembre 2002. Springer. ISBN 3-540-44235-9.

Intel Corporation : *Intel Virtualization Technology for Directed I/O - Architecture Specification*, 2007.

Intel Corporation : *IA-32 Intel Architecture Software Developer's Manual Volume 1 : Basic Architecture*, 2008a.

Intel Corporation : *IA-32 Intel Architecture Software Developer's Manual Volume 2a : Instruction Set Reference, a-m*, 2008b.

Intel Corporation : *IA-32 Intel Architecture Software Developer's Manual Volume 2b : Instruction Set Reference, n-z*, 2008c.

Intel Corporation : *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3A : System Programming Guide, Part 1*, 2008d.

Intel Corporation : *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B : System Programming Guide, Part 2*, 2008e.

Intel Corporation : *Intel Trusted Execution Technology - Measured Launched Environment Developer's Guide*, 2008f.

ISO/IEC : Common Criteria for Information Technology Security Evaluation. Rapport technique 2.1, ISO/IEC - 15408, août 1999.

ISO/IEC : Information technology – Trusted Platform Module – Part 1 : Overview. Rapport technique 3.1R1, ISO/IEC - 11889-1, mai 2009.

Jbtzhm : Static Kernel Patching. *Phrack*, 60, décembre 2002.

X. Jiang, X. Wang et D. Xu : Stealthy malware detection through vmm-based out-of-the-box semantic view reconstruction. *In Proceedings of the 14th ACM conference on Computer and communications security*, pages 128–138. ACM New York, NY, USA, 2007.

A. Jones, R. Lipton et L. Snyder : A Linear time algorithm for deciding security. *In Proceedings of the 17th Annual Symposium on Foundations of Computer Science*, pages 33–41, Houston, USA, 1976. IEEE Computer Society.

Kad : Handling Interrupt Descriptor Table for fun and profit. *Phrack*, 59, 2002.

A. Kalakech, T. Jarboui, J. Arlat, Y. Crouzet et K. Kanoun : Benchmarking operating system dependability : Windows 2000 as a case study. *In Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 261–270, 2004.

P. Kamp et R. Watson : Jails : Confining the omnipotent root. *In Proceedings of the 2nd International SANE Conference*, Maastricht, Pays-Bas, mai 2000.

K. Kasslin : Kernel malware : The attack from within. *In Proceedings of the 9th Annual AVAR (Association of anti-Virus Asia Researchers) International Conference*, Auckland, Nouvelle-Zélande, décembre 2006. `http://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf`.

J. King : Symbolic execution and program testing. *Communications of the ACM*, 19(7):394, 1976.

S. T. King *et al.* : SubVirt : Implementing malware with virtual machines. *In Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 314–327, Oakland, California, mai 2006.

S. T. King, G. W. Dunlap et P. M. Chen : Operating system support for virtual machines. *In Proceedings of the 2003 USENIX Annual Technical Conference*, pages 71–84, San Antonio, TX, USA, juin 2003.

A. Kivity *et al.* : KVM : the Linux Virtual Machine Monitor. *In Proceedings of the 2007 Linux Symposium*, Ottawa, Canada, juin 2007.

K. Kolyshkin : Virtualization in Linux, 2006. `http://wiki.openvz.org/Main_Page`.

C. Kruegel, W. Robertson et G. Vigna : Detecting Kernel-Level Rootkits Through Binary Analysis, 2004.

N. Kumar et V. Kumar : Boot Kit, 2006. `http://www.rootkit.com/newsread.php?newsid=614`.

E. 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`.

E. Lacombe : May the rootkit be with you. *In Hack.Lu*, Kirchberg, Luxembourg, 2007.

E. Lacombe et F. Gaspard : Les rootkits sous linux : passé, présent et futur. *MISC magazine*, 34:38–49, novembre 2007.

E. Lacombe, V. Nicomette et Y. Deswarte : A Hardware-Assisted Virtualization Based Approach on How to Protect the Kernel Space from Malicious Actions. *In Proceedings of the 18th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, Berlin, Allemagne, 2009a.

E. Lacombe, V. Nicomette et Y. Deswarte : Enforcing Kernel Constraints by Hardware-Assisted Virtualization. *Journal in Computer Virology*, 2009b. http://www.springerlink.com/content/v0w56774150764vr/.

E. Lacombe, V. Nicomette et Y. Deswarte : Une approche de virtualisation assistée par le matériel pour protéger l'espace noyau. *In* Eric Filiol, éditeur : *Actes du 7ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'09)*, Rennes, France, juin 2009c. École Supérieure et d'Application des Transmissions.

E. Lacombe, F. Raynal et V. Nicomette : De l'invisibilité des rootkits : application sous Linux. *In* Eric Filiol, éditeur : *Actes du 5ème Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC'07)*, Rennes, France, juin 2007. École Supérieure et d'Application des Transmissions.

E. Lacombe, F. Raynal et V. Nicomette : Rootkit modeling and experiments under Linux. *Journal in Computer Virology*, 4:137–157(21), mai 2008. http://www.ingentaconnect.com/content/klu/11416/2008/00000004/00000002/00000069.

L. Lamport : Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

B. Lampson : A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, octobre 1973.

A. Lanzi, L. Martignoni, M. Monga et R. Paleari : A smart fuzzer for x86 executables. *In Proceedings of the Third International Workshop on Software Engineering for Secure Systems (SESS'07)*, 2007.

J.-C. Laprie : Sûreté de Fonctionnement des Systèmes : Concepts de Base et Terminologie. *Revue de l'Electricite et de l'Electronique*, (11), décembre 2004.

T. Lawless : On Intrusion Resiliency, 2002.

S. Liakh : [V4] x86 : NX protection for kernel data, septembre 2009. http://patchwork.kernel.org/patch/46762/.

Microsoft Corporation : Digital Signatures for Kernel Modules on Systems Running Windows Vista. Rapport technique, Microsoft Corporation, 2006.

Nergal : The advanced return-into-lib(c) exploits : PaX case study. *Phrack*, 58, 2001.

NIST et NSA : Federal Criteria for Information Technology Security. Rapport technique Volume I et II, National Institute of Standards and Technology (NIST) and National Security Agency (NSA), 1992.

P. Oehlert : Violating assumptions with fuzzing. *IEEE Security & Privacy*, pages 58–62, 2005.

R. Ortalo, Y. Deswarte et M. Kaâniche : Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security. *IEEE Transactions on Software Engineering*, 25(5):633–650, 1999. ISSN 0098-5589.

PCI SIG : PCI Local Bus Specification. Rapport technique revision 2.2, PCI Special Interest Group, 1998.

N. Petroni, T. Fraser, J. Molina et W. Arbaugh : Copilot-a coprocessor-based kernel runtime integrity monitor. *In Proceedings of the 13th USENIX Security Symposium*, pages 179–194, San Diego, CA, USA, 2004. Citeseer.

N. Petroni Jr, T. Fraser, A. Walters et W. Arbaugh : An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. *In Proceedings of the 15th USENIX Security Symposium*, pages 289–304, Vancouver, Canada, 2006.

N. Petroni Jr et M. Hicks : Automated detection of persistent kernel control-flow attacks. *In Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 103–115, Alexendria, VA, USA, octobre 2007.

D. Piegdon et L. Pimenidis : Hacking in physically addressable memory - a proof of concept. *In Proceedings of the 4th International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA'07)*, 2007.

Pluf : Linux Per-Process Syscall Hooking. *7a69ezine*, 2006. `http://www.governmentsecurity.org/forum/index.php?showtopic=23072`.

Pluf et Ripe : Advanced Antiforensics : SELF. *Phrack*, 63, 2005.

J. Pol : [PINE-CERT-20040201] reference count overflow in shmat(), 2004. `http://seclists.org/bugtraq/2004/Feb/0140.html`.

G. Popek et R. Goldberg : Formal requirements for virtualizable third generation architectures. *Trans. on Computers C*, 22:644–656, 1974.

Pragmatic et THC : (nearly) Complete Linux Loadable Kernel Modules. The definitive guide for hackers, virus coders and system administrators, 1999. `http://newdata.box.sk/raven/lkm.html`.

D. Price et A. Tucker : Solaris Zones : Operating System Support for Consolidating Commercial Workloads. *In Proceedings of the 18th Large Installation System Administration Conference (LISA'04)*, pages 241–254, Atlanta, GA, USA, novembre 2004. USENIX Association Berkeley, CA, USA.

F. Raynal : Les canaux cachés. *Techniques de l'ingénieur - Sécurité des systèmes d'information*, SI1(H5860):1–10, décembre 2003.

F. Raynal, Y. Berthier, P. Biondi et D. Kaminsky : Honeypot forensics : analyzing system and files. *IEEE Security & Privacy Journal*, août 2004.

R. Riley, X. Jiang et D. Xu : Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. *In Proceedings of Recent Advances in Intrusion Detection (RAID)*, pages 1–20. Springer, 2008.

J. Riordan et B. Schneier : Environmental Key Generation Towards Clueless Agents. *Lecture Notes in Computer Science*, 1419:15–24, 1998. citeseer.ist.psu.edu/639981.html.

C. H. Rowland : Covert Channels in the TCP/IP protocol suite. *First Monday*, mars 1996. http://www.firstmonday.dk/issues/issue2_5/rowlad/.

R. Russell : lguest : Implementing the little Linux hypervisor. *In Proceedings of the 2007 Linux Symposium*, Ottawa, Canada, juin 2007.

J. Rutkowska : Subverting Vista Kernel For Fun And Profit. *In Black Hat USA*, Las Vegas, USA, 2006. http://invisiblethings.org/papers.html.

J. Rutkowska : Beyond The CPU : Defeating Hardware Based RAM Acquisition Tools (Part I : AMD case). *In Black Hat DC*, Washington DC, USA, 2007. http://invisiblethings.org/papers.html.

J. K. Rutkowski : Execution path analysis : finding kernel based rootkits. *Phrack*, 59, 2002.

A. Sacco et A. Ortega : Deactivate the Rootkit : Attacks on BIOS anti-theft technologies. *In Black Hat USA*, Las Vegas, USA, juillet 2009a. http://www.blackhat.com/presentations/bh-usa-09/ORTEGA/BHUSA09-Ortega-DeactivateRootkit-PAPER.pdf.

A. Sacco et A. Ortega : Persistent BIOS Infection. *Phrack*, 66, juin 2009b.

B. Schneier : Attack trees. *Dr. Dobb's journal*, 24(12):21–29, 1999.

Scythale : Hacking deeper in the system. *Phrack*, 64, mai 2007.

Sd et Devik : Linux on-the-fly kernel patching without LKM. *Phrack*, 58, 2001.

A. Seshadri et N. Qu : SecVisor : a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. *ACM SIGOPS Operating Systems Review*, 41(6):335–350, 2007.

H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu et D. Boneh : On the effectiveness of address-space randomization. *In Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, pages 298–307, Washington DC, USA, 2004. ACM. ISBN 1-58113-961-6.

T. Shanley et D. Anderson : *PCI system architecture (4th ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-30974-2.

O. M. Sheyner : *Scenario graphs and attack graphs*. Thèse de doctorat, University of Wisconsin, 2004.

G. Smith : *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000. ISBN 0-7923-8684-1.

J. Smith et R. Nair : The architecture of virtual machines. *Computer*, pages 32–38, 2005.

D. Soeder et R. Permeh : eEye BootRoot : A Basis for Bootstrap-based Windows Kernel Code, 2005. `http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf`.

Solar Designer : Getting around non-executable stack, 1997. `http://seclists.org/bugtraq/1997/Aug/0063.html`.

S. Sparks et J. Butler : Raising The Bar For Windows Rootkit Detection. *Phrack*, 63, 2005.

B. Spengler : PaX's UDEREF - Technical Description and Benchmarks, 2007. `http://www.grsecurity.net/~spender/uderef.txt`.

B. Spengler *et al.* : PaX documentation, 2003. `http://pax.grsecurity.net/docs`.

B. Spengler *et al.* : Grsecurity features, 2009. `http://www.grsecurity.net/features.php`.

J. M. Spivey et J. R. Abrial : *The Z Notation : A Reference Manual*. Prentice Hall International, 1992.

Sqrkkyu et Twzi : Attacking the Core : Kernel Exploiting Notes. *Phrack*, 64, 2007.

Stealth : Kernel Rootkit Experience. *Phrack*, 61, 2003.

M. Swift, M. Annamalai, B. Bershad et H. Levy : Recovering Device Drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, 2006.

M. Swift, B. Bershad et H. Levy : Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

A. Tanenbaum : *Systèmes d'exploitation*. Pearson Education, 2003.

H. Thompson, J. Whittaker et F. Mottay : Software security vulnerability testing in hostile environments. *In Proceedings of the 2002 ACM symposium on Applied computing*, pages 260–264. ACM New York, NY, USA, 2002.

T.I. Standard : Tool Interface Standard (TIS) - Executable and Linking Format (ELF) Specification - Version 1.2. *TIS Committee, May*, 1995.

Truff : Infecting loadable kernel modules. *Phrack*, 61, 2003.

Trusted Computing Group : Trusted Platform Module Specification v1.2 Enhances Security, 2004.

VMware, Inc. : Virtualization Overview, 2006. `http://www.vmware.com/solutions/whitepapers/virtualization.html`.

T. Wang, T. Wei, Z. Lin et W. Zou : Intscope : Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. *In Proceedings of the Network Distributed Security Symposium (NDSS)*, 2009.

Y. Wang, D. Beck, B. Vo, R. Roussev et C. Verbowski : Detecting stealth software with Strider GhostBuster. *In Proceedings of the International Conference on Dependable Systems and Networks (DSN 2005)*, pages 368–377, février 2005.

F. Wecherowski : A Real SMM Rootkit : Reversing and Hooking BIOS SMI Handlers. *Phrack*, 66, juin 2009.

D. A. Wheeler : Flawfinder, janvier 2007. `http://www.dwheeler.com/flawfinder/`.

M. Wolf : Covert Channels in LAN Protocols. *In Proceedings on the Workshop for European Institute for System Security on Local Area Network Security (LANSEC'89)*, pages 91–101, London, UK, 1989. Springer-Verlag. ISBN 3-540-51754-5.

P. Wolper : *Introduction à la calculabilité*. DUNOD, 3ème édition, 2006. ISBN 2-10-049981-5.

X. Zhang, T. Jaeger, R. Perez et R. Sailer : Secure coprocessor-based intrusion detection. *In Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 239–242, Saint-Emilion, France, juillet 2002.

Résumé

Cette thèse traite de la préservation de l'intégrité des systèmes d'exploitation courants. L'objectif est de répondre aux menaces actuelles et futures que représentent les logiciels malveillants qui s'implantent dans le noyau de ces systèmes (comme les *rootkits* "noyau") ou du moins en altèrent l'intégrité (comme les *rootkits* "hyperviseur"). La première partie de ce document se focalise sur ces logiciels malveillants. Tout d'abord, les attaques logiques sur les systèmes informatiques sont présentées dans leur globalité. Ensuite est proposée une classification des actions malveillantes qui provoquent la perte de l'intégrité d'un noyau. Enfin, les résultats d'une étude sur les *rootkits* "noyau" sont donnés et la création d'un *rootkit* original est expliquée. La seconde partie s'intéresse à la protection des noyaux. Après une description de l'état de l'art, une approche originale est proposée, fondée sur le concept de préservation de contraintes. En premier lieu, les éléments essentiels sur lesquels reposent un noyau sont identifiés et les contraintes sur ces éléments, nécessaires à un fonctionnement correct du noyau, sont exposées. Un hyperviseur léger (Hytux) a été conçu pour empêcher la violation de ces contraintes en interceptant certaines des actions du noyau. Sa mise en œuvre est décrite pour un noyau Linux 64 bits sur architecture x86 qui dispose des technologies Intel VT-x et VT-d.

Mots-clés: sécurité informatique, intégrité des systèmes d'exploitation, protection des noyaux, préservation de contraintes, virtualisation matérielle, codes malveillants, rootkits

Abstract

This Ph.D thesis addresses the integrity preservation of current operating systems. The main goal is to counter current and future threats coming from malware that infects the kernel of these systems (as kernel rootkits) or at least that provoke a loss of their integrity (as hypervisor rootkits). The first part of this document focuses on such malware. First, logical attacks are presented globally. Then, a classification of malicious actions that lead to the loss of kernel integrity is proposed. Finally, the outcomes of a study on kernel rootkits are given and the creation of an original rootkit is explained. The second part deals with kernel protection. After describing the state of the art, an original approach is proposed, based on the concept of constraint preservation. First, the essential elements which a kernel rests on are identified and the required constraints on these elements for correct kernel operation are exhibited. A lightweight hypervisor (Hytux) has been elaborated to prevent any violation of these constraints by intercepting some actions of the kernel. Its implementation is described for a 64-bit Linux kernel on an x86 architecture that supports the Intel VT-x and VT-d technologies.

Keywords: computer security, operating system integrity, kernel protection, constraint preservation, hardware-assisted virtualization, malicious code, rootkits