De l'invisibilité des rootkits : application sous Linux

Éric Lacombe[1], Frédéric Raynal[2], and Vincent Nicomette[3]

[1] CNRS-LAAS - EADS-CCR
[2] Sogeti ESEC - Misc Magazine
[3] CNRS-LAAS

Résumé Cet article traite de la conception de rootkits. Nous montrons en quoi ces codes malicieux sont innovants par rapport aux *malwares* habituels comme les virus et chevaux de Troie, ce qui nous permet d'exposer une architecture fonctionnelle des rootkits. Nous proposons également des critères permettant de qualifier et d'évaluer les différents rootkits. Nous avons volontairement abordé le problème dans sa globalité, c'est-à-dire en ne nous restreignant pas seulement au logiciel rootkit, mais aussi à la communication entre l'attaquant et son outil ou les interactions avec le système. Naturellement, nous constatons que les problématiques rencontrées lors de la conception de rootkits sont proches de celles de la stéganographie, mais nous précisons également les limites de cette comparaison. Enfin, nous présentons notre propre rootkit, fonctionnant en mode noyau sous Linux, et plusieurs nouvelles techniques conçues afin d'en accroître la furtivité.

Mots clés : rootkit, malware, modélisation, propriétés, furtivité.

1 Introduction

Contrairement à une idée (trop ?) répandue, il n'est nullement besoin d'*exploit*[4] pour entrer sur un système d'information. En revanche, une fois compromis, l'intrus entreprend des actions 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*.

Ainsi, nous nommons *rootkit*, un ensemble de modifications permettant à un attaquant de maintenir dans le temps un contrôle frauduleux sur un système d'information.

Les méthodes employées par un rootkit pour remplir son rôle mélangent les approches des différents types de codes malicieux (bombe logique, cheval de Troie, virus, etc.). Tout en précisant la définition précédente, nous verrons qu'un rootkit constitue un nouveau type de code malicieux, même s'il emprunte aux autres (cf. Section 4).

Dans cet article, nous développons les principes fondamentaux caractérisant les rootkits. Nous nous plaçons résolument du point de vue du concepteur et utilisateur, cherchant ainsi à identifier tant les diverses stratégies envisageables que les contraintes techniques.

Concernant les propriétés en terme de sécurité du rootkit[5], l'intrus doit au minimum conserver un moyen d'envoyer des instructions au système compromis. Ensuite, tout est question de choix. Doit-il pouvoir agir en étant directement connecté sur le système ou l'envoi d'instructions puis l'exécution asynchrone suffisent-t-elles ? A-t-il besoin impérativement de rester caché aux yeux des

[4] Un *exploit* est un petit programme tirant parti d'une faille dans un logiciel pour pénétrer le système hôte.
[5] Attention, comme nous prenons l'angle de l'intrus, les propriétés de sécurité sont celles qui assurent la sécurité de l'attaquant.

utilisateurs légitimes ou peut-il entreprendre ses actions au grand jour ? La connaissance du rootkit nuit-elle à ses agissements ? Pour tenter d'apporter une réponse à ces questions d'ordre stratégique, nous détaillons les objectifs de l'intrus, et examinons ses attentes en termes de sécurité afin de les atteindre.

À partir de là, nous proposons quelques critères pour évaluer les performances d'un rootkit. Il est important de souligner que les objectifs d'un attaquant peuvent varier grandement, et par conséquent, les capacités de son rootkit également. Néanmoins, certains aspects restent immuables, comme la discrétion que procure le logiciel malicieux ou encore la possibilité d'exécuter des opérations sur la machine compromise.

Enfin, nous présentons une nouvelle méthode de détournement du noyau Linux. Nous avons axé notre méthode sur l'invisibilité au sein du système corrompu. Ainsi, nous dissimulons notre code malicieux au sein de l'espace noyau et parasitons un processus, ce qui suffit pour compromettre totalement le système.

Cet article se décompose en cinq parties. Tout d'abord, nous donnons en section 2 les bases techniques permettant d'aborder l'article plus sereinement. Ensuite, nous présentons en section 3 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 4 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, tout en nous restreignant par la suite au seul critère d'invisibilité. Nous présentons en section 5 la technique que nous avons élaborée pour corrompre un noyau Linux tout en étant le plus discret possible. Finalement, la section 6 clôt l'article avec une synthèse sur les apports de notre approche.

2 Rappels techniques

Dans cette partie, nous rappelons quelques concepts sur le fonctionnement des noyaux, ou de l'architecture x86, nécessaires à la suite.

2.1 Le noyau d'un système d'exploitation

Un noyau de système d'exploitation est un logiciel assurant la gestion du matériel d'un ordinateur (mémoires, processeurs, disques, périphériques, etc.) et offrant à l'utilisateur une interface permettant d'interagir facilement avec ce matériel.

Différents types de noyau ont été élaborés. Parmi les plus répandus nous trouvons les noyaux monolithiques et les micro-noyaux. Ces derniers contiennent uniquement ce qui nécessite une exécution en mode privilégié, le reste des services fournis est laissé en espace utilisateur. Il s'ensuit que les multiples éléments du système d'exploitation (unité de gestion mémoire, etc.) sont isolés les uns des autres et dialoguent via des messages transitant par le micro-noyau.

Au contraire, dans les noyaux monolithiques, la majorité des services jugés critiques réside en espace 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'espace utilisateur.

Nous focalisons notre attention dans le reste du document sur le noyau Linux qui est de type monolithique modulaire. Ce noyau met à disposition de l'utilisateur de nombreux services en s'abstenant, dans la mesure du possible, de lui imposer la marche à suivre.

2.2 L'architecture x86

Un des rôles du système d'exploitation est de gérer le matériel sur lequel il fonctionne, ce qui l'en rend dépendant. Cependant, une construction en couches permet de s'abstraire des spécificités du matériel aux niveaux les plus hauts. Le noyau Linux suit cette logique et réalise la plupart de ses fonctionnalités de façon indépendante du matériel. Notre étude concerne, entre autres, des parties dépendantes du matériel, comme la gestion de la mémoire, que nous présentons ci-après.

Nous nous focalisons sur l'architecture x86, diffusée très largement auprès du grand public. Bien que chaque architecture possède des particularités, elles fournissent en général toutes des fonctionnalités identiques : gestion de la mémoire (plus rarement pour l'embarqué), modes de fonctionnement associés à des privilèges différents, communication entre les différents éléments matériels et le logiciel (souvent par des interruptions), etc.

En x86, la gestion de la mémoire passe par une unité physique de segmentation (obligatoire) et une autre de pagination (optionnelle). La pagination est très courante dans les autres types d'architectures, au contraire de la segmentation. Aussi, Linux étant multiplateforme, la segmentation n'est utilisée que dans sa forme la plus épurée (i.e. le mode *flat*[6]) permettant de s'en abstraire, afin d'employer efficacement la pagination.

L'architecture x86 est conçue sur une structure en quatre anneaux (*ring*), représentant chacun un certain mode de fonctionnement du processeur. À chaque mode est associé un niveau de privilèges. L'anneau le plus privilégié est l'anneau 0 (*ring 0*), mode dans lequel le noyau Linux s'exécute, alors que l'anneau 3, le moins privilégié, est réservé à l'exécution des applications de l'espace utilisateur.

La communication entre l'espace noyau et l'espace utilisateur, autrement dit le passage du *ring 0* au *ring 3* et inversement, se produit grâce à différents mécanismes, dont le plus courant est l'interruption. Elles se divisent en exceptions (i.e. interruptions du processeur comme une division par zéro, une faute de page, etc.), interruptions matérielles (i.e. celles déclenchées par les périphériques, comme appuyer sur une touche du clavier) et enfin en interruptions logicielles (i.e. interruptions déclenchées depuis le logiciel comme une application de l'espace utilisateur exécutant un appel système).

Sur x86, ces interruptions sont numérotées de 0 à 255. Chacune est associée à un gestionnaire si celui-ci a été mis en place par le noyau. Ce gestionnaire est une fonction s'exécutant lors de la levée de l'interruption. L'ensemble de ces fonctions est accessible depuis une table appelée IDT (*Interruption Descriptor Table*) que le noyau remplit et « charge » ensuite dans le processeur via l'instruction `lidt`.

Les interruptions matérielles ou exceptions du processeur interrompent l'exécution en espace utilisateur et donnent la main au noyau. Les premières se produisent de façon asynchrone tandis que les secondes se déclenchent de manière synchrone.

Le noyau a pour rôle de servir l'interruption ou l'exception et de rendre ensuite la main à l'espace utilisateur. Cependant avant de réaliser cette dernière action, le noyau peut décider d'effectuer d'autres tâches qui lui semblent plus urgentes. Notamment, dans le cas de Linux, l'ordonnanceur vérifie si un autre processus n'a pas un besoin prioritaire de s'exécuter.

Les interruptions logicielles sont utilisées principalement dans l'implémentation des appels système. Après qu'un utilisateur a levé l'interruption logicielle définie par le noyau du système d'exploitation (`0x80` pour Linux), le processeur passe en *ring 0* et donne la main à l'entrée du noyau chargée de servir la requête de l'utilisateur. Afin de rendre plus performante cette transition, l'architecture x86 définit l'instruction `sysenter` (qui n'est pas une interruption). Son objectif est de fournir un

[6] Un seul segment est défini, allant de l'adresse physique 0 jusqu'à 4 Go.

support efficace à la réalisation de mécanismes d'appels système. Notre recherche prend d'ailleurs en considération cette technologie. L'annexe A explique le fonctionnement de cette instruction.

La section suivante reprend ces notions au travers d'un état de l'art des méthodes employées par les rootkits.

3 État de l'art

Nous présentons ici un petit historique des rootkits, en montrant comment ils ont évolué de façon à devenir de plus en plus efficaces.

3.1 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 : c'est la première forme des *rootkits*. Ainsi, lorsque l'administrateur en appelle à ses outils, ceux-ci cachent certaines informations sur l'intrus, 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 compilés à chaque fois ;
- 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, voyant ainsi sa présence révélée ;
- bien souvent, et surtout dans les environnements sécurisés, des *empreintes* des binaires sont calculées à partir de fonction de hash, 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.

La 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 3.2 et 3.3 les méthodes d'injection et de détournement en espace noyau.

De nouvelles tendances se profilent avec l'arrivée du support matériel pour la virtualisation dans les processeurs grand public (`VT-x` pour Intel et `SVM` pour AMD). Nous trouvons à présent un hyperviseur au plus bas de l'échelle qui s'occupe de gérer des machines virtuelles sur lesquelles tournent des systèmes d'exploitation.

Il est alors intéressant pour un rootkit de s'implanter au niveau de l'hyperviseur, afin de bénéficier du contrôle des systèmes invités tournant sur le système hôte sans avoir à les infecter [19]. Bien que l'hyperviseur puisse faire partie intégrante du noyau du système hôte (ex : KVM pour Linux), il dispose de moyens d'interception ou de modification reposant sur le matériel qui sont hors de porté du noyau de l'hôte. De plus, grâce au support que le matériel apporte dans la virtualisation, la réalisation d'un hyperviseur est facilitée. 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 peut s'effectuer alors de façon plus discrète.

Enfin, un effort de classification a été réalisé au sujet des infections informatiques (*malwares*) de type camouflé. En effet, J. Rutkowska propose une taxonomie [30] qui distingue les malwares suivant le type de leur corruption. Ainsi trois catégories d'invisibilité croissante sont discernées : (1) les malwares qui effectuent une corruption d'éléments figés (i.e. du code), (2) ceux qui corrompent des éléments non figés (i.e. les données) et (3) ceux qui agissent au delà du système d'exploitation ou des applications s'y exécutant, sans les modifier (par exemple les rootkits hyperviseur).

3.2 Méthodes d'injection en espace noyau employées par les rootkits sous Linux

Nous distinguons quatre voies pour injecter du code ou des données dans le noyau Linux. La première, passe par l'utilisation des modules noyaux que nous pouvons intégrer dans Linux dynamiquement [38]. Cette méthode n'est possible que si le support par le noyau des LKM (*Linux Kernel Module*) est activé. Le cas échéant, des mécanismes de protection existent. Pour détecter des modules noyau malicieux, mis à part la mise en place d'une infrastructure de vérification des modules dignes de confiance via l'utilisation de signatures cryptographiques [23], une alternative fonctionnelle prend le problème par l'autre bout. Cette méthode est fondée sur une analyse comportementale des modules noyau. Elle est effectuée avant insertion 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 [20].

La deuxième voie consiste à corrompre le noyau en accédant à sa mémoire via le périphérique virtuel `/dev/kmem` [33,7]. Cependant, des protections comme *Grsecurity* peuvent être configurées afin d'interdire l'écriture ou la lecture sur ce périphérique. Cependant, une configuration de ce type bloque l'exécution du serveur X, à moins de mettre en place une politique non mandataire mais à la discrétion de chaque binaire via le programme `rsbac`. Ainsi, ce genre de protection est généralement désactivé sur un poste client.

La troisième correspond aux exploitations de failles noyau. Certaines permettent l'injection de code alors que d'autres ont un périmètre beaucoup plus restreint. La difficulté avec cette approche est que nous restons dépendants des versions du noyau affectées par la faille exploitée.

La dernière exploite les particularités des périphériques pouvant accéder au contrôleur de la mémoire physique sans intervention du processeur (i.e. 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 [10,3]. Cependant, J. Rutkowska montre que la lecture en mémoire physique via un accès DMA peut aussi être mis en défaut au niveau logiciel [31].

Notre démonstrateur s'appuie sur de l'injection de code via `/dev/kmem` (cf. l'annexe B). Cette approche nous semble être un bon compromis[7]. 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`

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

empêche le fonctionnement de certaines applications typiques sur ce genre de machine. Finalement, l'exploitation de failles noyau n'est pas suffisamment fiable dans le temps.

3.3 Méthodes de détournement employées par les rootkits sous Linux

Les premières techniques se fondent sur la modification de la table des appels système [33]. L'attaquant redirige des services noyaux vers des fonctions malveillantes qu'il a préalablement injectées en mémoire. Cependant des méthodes de détection simples sont 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 l'emplacement du code des appels système les uns par rapport aux autres.

Pour pallier ce problème, l'attaquant est remonté un peu plus haut dans la chaîne et s'est attaqué au gestionnaire des appels systèmes [9]. Dans une première phase, l'attaquant duplique la table des appels système en mémoire et la modifie afin d'y inclure ces fonctions malveillantes. Ensuite, il modifie l'adresse qu'utilise 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 systèmes utilise.

Pour effectuer un appel système, nous levons en espace utilisateur une interruption logicielle. Elle déclenche le passage en mode noyau. Cette interruption peut être interceptée [18] 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. Aussi, si le noyau n'autorise que l'utilisation de l'instruction `sysenter` cette approche ne peut être mise en place.

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 alors que 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 [4]. De façon générale n'importe quelle interruption peut être détournée de sa fonction initiale.

En remontant encore dans le chemin d'exécution, l'attaquant, cette fois, duplique en mémoire l'IDT du système, la modifie et ensuite charge son adresse auprès du processeur à la place de l'actuelle [18]. 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. Aucune innovation flagrante n'est encore à l'œuvre.

Dans ce jeu du chat et de la souris, l'attaquant descend à présent dans les arcanes du noyau. Notamment, les fonctions du VFS (*Virtual File System*) sont détournées par *hooking* (i.e. modification des pointeurs de fonctions) dans le rootkit *adore-ng* [36]. Ainsi, l'attaquant cache les fichiers et processus qu'il désire. Cependant, le même type de détection que précédemment peut encore être déployé.

Afin de s'affranchir de ce problème, l'attaquant peut, dans un premier temps, injecter en mémoire le code effectuant ses opérations malicieuses. 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 [8]. Les solutions mises en place pour contrer ces techniques sont plus lourdes que les précédentes. Il s'agit de contrôler l'intégrité du code du gestionnaire.

Les solutions de détection, expliquées brièvement dans les paragraphes précédents, et d'autres plus évoluées [32] 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 renverra uniquement des informations ne compromettant pas sa furtivité.

Pour obtenir une détection efficace, des mécanismes sont 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* [22].

C'est maintenant au tour de l'attaquant d'apporter des innovations. Elles se révèlent dans le rootkit Shadow Walker [35]. Il 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 5.3.

Les types de détournement dont nous venons de parler, se situent tous au sein du système et plus particulièrement du noyau. Voyons à présent deux technologies relativement récentes qui sont aux frontières de ce système.

La première consiste à implanter le rootkit au niveau du secteur de démarrage de la machine. Ainsi, nous prenons la main avant le système d'exploitation. Cette idée est mise en œuvre par exemple dans les rootkits BootRoot [34] ou encore Boot Kit [21].

La seconde innovation est apportée par les *rootkits hyperviseur*. Ils mettent en œuvre en leur sein un hyperviseur frauduleux reposant sur les technologies matérielles de virtualisation. Blue Pill [29] est un exemple de rootkit hyperviseur utilisant l'extension matérielle de virtualisation des processeurs AMD. Nous supposons ici que le système d'exploitation présent sur la machine n'utilise pas la virtualisation matérielle. Le rootkit va alors se « déclarer » hyperviseur auprès du processeur et va placer le système d'exploitation actuel dans une machine virtuelle à son insu. Ainsi, il en obtient le contrôle sans pour autant le corrompre.

Nous ne développons pas dans cet état de l'art tous les services que fournissent habituellement les rootkits. Cependant, la section 5 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.

4 Architecture et élaboration d'un rootkit

Cette partie aborde les éléments fondamentaux qu'un attaquant doit prendre en compte lorsqu'il construit un rootkit.

Tout d'abord, nous proposons une définition d'un rootkit, afin de mieux saisir où se situent ces codes malicieux 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. Nous décrivons dans la première partie les éléments essentiels constituant un rootkit.

Ce dernier 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 critères pour évaluer un rootkit. Notons que l'utilisation de ces critères 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 leur éradication.

4.1 Définition d'un rootkit et comparaison avec les autres codes malicieux

Les codes malicieux sont habituellement séparés en deux grandes familles :
- les codes auto-reproducteurs, parmi lesquels nous situons les virus ou les vers, sont capables de dupliquer leur code dans un type précis de fichiers ;
- les infections simples, comme les bombes logiques et les chevaux de Troie, incapables de se répliquer.

Le problème des rootkits est qu'ils ne rentrent pas dans cette classification de *malwares*. Par conséquent, nous n'avons pas connaissance d'une caractérisation claire et précise de ce qu'est réellement un rootkit. C'est pourquoi nous proposons la définition suivante :

Definition 1.
Un rootkit *est un ensemble de modifications permettant à un attaquant de maintenir dans le temps un contrôle frauduleux sur un système d'information.*

Plusieurs éléments sont caractéristiques d'un rootkit :
- *ensemble de modifications* : une première différence flagrante apparaît ici par rapport aux autres codes malicieux. 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 sur d'autres composants du système (programmes en espace utilisateur, partie du noyau, ou autres). Ce type de modifications nous fait penser à la notion de *parasitage* associée aux codes malicieux qui propage leur charge utile en fonction de leurs cibles.
- *maintien dans le temps* : les autres codes malicieux 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 a pris le contrôle du système pour y effectuer certaines opérations (vol d'information, rebond, déni de service, etc.) et doit fiabiliser son accès tant 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.

Ces éléments rendent incompatibles le classement des rootkits dans l'organisation habituelle des codes malicieux. Nous distinguons usuellement ceux qui sont reproducteurs, comme les vers et les virus, des autres, appelés *infections simples*, comme les bombes logiques et les chevaux de Troie (cf. [12] pour une description détaillée).

Un rootkit n'ayant pas vocation à se reproduire, nous ne pouvons donc pas, *a priori* l'assimiler à un code reproducteur. Cependant, la tendance infectieuse d'un rootkit lui vient de son côté distribué. En effet, il s'insinue souvent dans plusieurs éléments du système. Si nous revenons sur les premiers rootkits, ils modifiaient plusieurs programmes, pour dissimuler des connexions réseau, l'activité sur le système, etc. Nous présentons d'ailleurs une technique (cf. section 5.5) d'infection

de processus pour maintenir le contrôle sur le système. Nous retrouvons ainsi le côté parasitaire de ces codes, qui vont se greffer à divers éléments du système, là où les virus sont souvent dédiés à un type de cible (les binaires ou les documents par exemple).

Une *bombe logique* est composée d'un déclencheur (ou gâchette) et d'une charge utile. Par exemple, la charge est activée à une heure donnée, ou quand un utilisateur effectue une action prédéterminée. Une fois installée sur un système, il n'y a plus d'interaction avec le concepteur de la bombe. Néanmoins, un rootkit peut tout à fait contenir des bombes logiques (par exemple un rootkit qui détruirait le système si nous tentions de l'analyser).

Un *cheval de Troie* est un programme souvent composé d'un serveur et d'un client. Le serveur est installé sur le système par un utilisateur, qui se fait leurrer en croyant installer autre chose. L'exemple que nous voyons le plus souvent actuellement vient des spywares où en installant un jeu sur son système, l'utilisateur installe également des programmes espions en charge de collecter puis renvoyer des informations sur l'utilisateur à des tiers. Il y a deux différences conceptuelles entre un rootkit et un cheval de Troie. Tout d'abord, le rootkit est installé sciemment par l'attaquant. Néanmoins, cette première différence n'est pas caractéristique du point de vue de l'attaquant : il doit prendre le contrôle du système, peu importe qui installe le moyen de le contrôler. En revanche, un cheval de Troie est un programme « simple » et monolithique, là où un rootkit n'est pas un programme à part entière, mais un ensemble de modifications effectuées sur plusieurs éléments du système.

Ainsi, nous aurions tendance à naturellement classer les rootkits du côté des infections simples. Nous venons néanmoins de voir qu'il ne s'agit nullement de quelque chose de simple puisqu'ils reprennent ainsi des éléments de chaque type de code malicieux connus, y compris les codes reproducteurs.

4.2 L'architecture fonctionnelle d'un rootkit

Il s'agit de décrire les éléments intervenant lors de l'utilisation d'un rootkit. La première étape pour l'attaquant est d'installer le rootkit puis d'en pérenniser l'accès au système compromis, d'où le besoin d'un module de protection. Une fois en place, l'attaquant utilise le rootkit comme intermédiaire avec le système. Il contient donc un module central servant d'interface entre le système et l'attaquant (i.e. une *backdoor*). L'intrus effectue les opérations nécessaires à l'accomplissement de ses objectifs, opérations qui caractérisent alors les services rendus par le rootkit. Le découpage fonctionnel d'un rootkit est ainsi décrit ci-après.

Un injecteur. Il s'agit du mécanisme que l'attaquant emploie afin d'insérer le rootkit dans le système (infection de 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, il est toujours besoin d'accéder puis de modifier certaines structures du système, une seule et unique fois[8].

[8] Comme avec notre rootkit (cf. section 5, nous pouvons avoir un mécanisme de bootstrap qui sert d'amorçage à des modifications plus profondes et complexes.

Un module de protection. Son objectif est de rendre le rootkit « tenace » sur le système tout le temps nécessaire à l'attaquant. Plusieurs stratégies sont envisageables et combinables. Citons à titre 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 de plus dissimulée.
- *Rendre résistant le rootkit :*
  Nous supposons ici que la présence du rootkit a été révélée. Il s'agit alors de munir le rootkit de capacités lui permettant de résister aux tentatives de suppression. Par exemple, une fois détecté, il peut menacer l'utilisateur de casser le bios de la carte mère si nous tentons 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 rendre le rootkit persistant au redémarrage de la machine. Ainsi, une portion du code en est injectée dans des éléments non volatiles du système.
- *Camoufler l'activité de l'attaquant :*
  Il se révèle, d'une part, dans les fonctionnalités de dissimulation des processus, des *sockets* réseau et des fichiers utilisés par l'attaquant. D'autre part, il consiste au filtrage des journaux d'évènements du système compromis.

La backdoor. La backdoor 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[9]), et d'autre part d'accéder aux services. La backdoor 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.

Finalement, la backdoor 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 s'entretenir avec 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 4.3.
- *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.

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 :

[9] 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.

- *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 opérations à l'encontre 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.

4.3 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 derniers points puisque nous nous concentrons sur les rootkits, et non les techniques d'évasion. Il est à noter que ces problèmes ne se posent que dans certaines conditions. Pour la récupération, cela fait suite à une attaque à distance. En effet, si 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. De même, l'utilisation à distance du rootkit suppose que l'attaquant n'y ait pas accès localement d'une part, et souhaite maintenir son accès dans le temps[10].

Ces communications sont considérées comme de l'activité externe au système compromis. Quand bien même l'attaquant serait invisible sur le réseau, il n'en est pas moins qu'il va laisser des traces sur l'hôte compromis comme utilisation de *socket* ou les statistiques d'envoi/émission de paquets des interfaces réseau. La dernière partie de cette section traite de ce problème.

La récupération du rootkit. Une fois qu'il a le contrôle du système cible, une des premières tâches entreprise par l'attaquant sera 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 [5,16,24,11].

[10] Ce maintien dans le temps constitue la différence principale entre un rootkit et une bombe logique, prévue pour agir une fois à une impulsion donnée.

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 connaître avec exactitude les actions entreprises par le pirate sur le système.

Les auteurs de [26] ont montré que ces risques sont réels. À partir de la capture effectuée sur le 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 empêchera 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 présentée dans le cadre du virus Bradley par Filiol dans [14,13]. 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* [27]. 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. Bradley propose 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 1 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 son contrôle, comme le hash 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 [14,13] que si le protocole cryptographique est correctement réalisé, la complexité de la cryptanalyse est exponentielle par rapport à la taille de la clé.

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

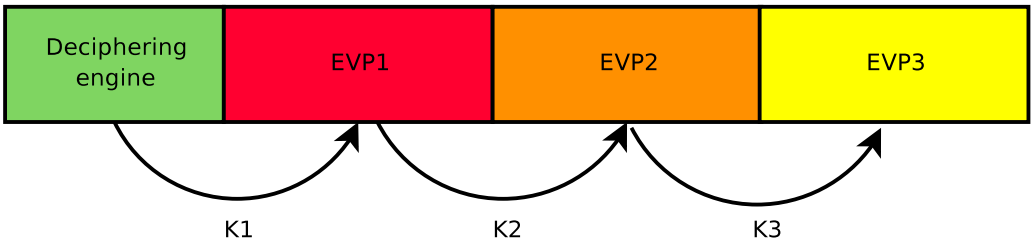

Fig. 1: Structure du virus Bradley

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 [15,39,28,25]. Il en a été développé dans de nombreux protocoles réseau.

En se positionnant en mode noyau, nous avons accès à toute la pile réseau, 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 de matériel tentent d'intercepter les flux réseau pour les analyser et en vérifier la conformité aux standards. Heureusement pour l'attaquant, ces méthodes sont encore assez peu fiables. Par exemple, au niveau TCP, très peu d'équipements font du suivi 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 driver à destination. Inversement, lors de la réception, les données arrivent sur le driver, 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 contente d'agir en espace utilisateur. Ainsi, un rootkit noyau peut parasiter les communication :

- quand des données sont émises, il ajoute ses propres octets au paquet juste avant de le passer au driver ;
- quand des données sont reçues via le driver, 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 firewall bloque les connexions entrantes et sortantes, notre rootkit peut continuer à communiquer avec l'extérieur.

[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.

Les traces d'activité réseau dans le système. Comme il 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 la présence de notre code. Néanmoins, l'exemple de Sebek [37] montre qu'il n'est pas si facile de cacher la présence d'un rootkit noyau traitant des connexions réseau [2] 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 détaillerons pas plus ici cet aspect. Néanmoins, il est important de le conserver à l'esprit.

4.4 Vers l'évaluation d'un rootkit

L'emploi d'un rootkit par un attaquant traduit sa volonté de conserver le contrôle d'un système dans le temps une fois qu'il l'a compromis. Quelque soit la nature de l'attaquant (opportuniste, hacktiviste, mafieux, etc.), nous identifions en général quatre objectifs :

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

Dans la majorité des cas, l'intrus cherche à dissimuler sa présence aux utilisateurs légitimes. Il vient donc naturellement à l'esprit qu'un des critères essentiels caractérisant un rootkit concerne son degré d'*invisibilité* (notion sur laquelle nous revenons ci-dessous). Néanmoins, il est tout à fait envisageable que l'attaquant ne souhaite pas particulièrement dissimuler son rootkit mais qu'en revanche, il fasse en sorte que le retirer du système soit très difficilement faisable sans mettre en péril le système lui-même. Cette notion fait donc intervenir un autre critère, concernant cette fois la *robustesse* du rootkit. 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 critère permettant d'exprimer cette modification que fait un rootkit sur le système compromis nous semble également pertinent pour évaluer un rootkit.

L'analogie avec la stéganographie peut nous aider dans l'expression de ces critères. En effet, la problématique de la conception d'un rootkit, du point de vue de l'attaquant, est proche de celle de la stéganographie : il s'agit de modifier un support sain, appelé *stégano-medium* 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 vérifiable), la *robustesse* (une transformation du stégano-medium ne doit pas altérer le message secret) et la *capacité* (qui exprime la quantité d'information dissimulée dans le stégano-medium).

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-medium sont des données 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 les 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 la capacité telle qu'elle.

Nous proposons donc d'introduire comme troisième critère caractérisant un rootkit une grandeur permettant en quelque sorte de mesurer la « malice » du rootkit, c'est-à-dire la façon dont il corrompt le système. Ce critère nous semble être proche de la notion de *virulence* qui est définie par Filiol [12] et qui s'applique habituellement aux virus. Cette notion est définie ainsi :

Definition 2.

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

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 par le virus v et le nombre de fichiers infectables par ce virus.*

Dans notre cas, cette notion doit encore être adaptée puisque, comme nous l'avons déjà expliqué, les modifications réalisées par un rootkit ne concernent pas uniquement des fichiers. 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 corrompu. Cependant, la virulence en termes médicaux est l'aptitude des microbes à se développer dans l'organisme. La notion d'infestation qui est définie comme l'envahissement d'un organisme par un parasite, nous paraît plus appropriée.

Définissons donc les trois critères ainsi :

Definition 3. *L'*invisibilité *d'un rootkit exprime la difficulté avec laquelle l'utilisateur du système peut détecter le rootkit lui-même ainsi que les activités malveillantes exécutées à l'aide de ce rootkit.*

Definition 4. *La* robustesse *d'un rootkit exprime la difficulté avec laquelle un rootkit peut être retiré d'un système sur lequel il est installé.*

Definition 5. *Le* pouvoir d'infestation *d'un rootkit exprime le degré d'ingérence du rootkit dans le système, c'est-à-dire la quantité d'éléments qui sont affectés par le rootkit sur le système compromis.*

Soulignons que la notion d'invisibilité englobe les différentes activités que l'attaquant parvient à exécuter sur le système compromis à travers le rootkit ainsi que le rootkit lui-même. De plus, elle regroupe à la fois la dissimulation d'objets passifs (typiquement des fichiers)[13], mais aussi la dissimulation d'activités (typiquement des processus). Nous adoptons ici la terminologie de Filiol [14]

[13] Les activités exécutées par l'attaquant peuvent engendrer une production de données, également à dissimuler (si elles sont importantes au déroulement de l'activité) ou bien supprimées (par exemple, les traces dans les différents journaux système).

à ce propos qui nomme *camouflage* la dissimulation d'objets passifs et *furtivité* la 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 malwares en général, en particulier de virus ou de vers. Nous utilisons donc ici les termes qui leur sont habituellement associés : la capacité d'un rootkit à rester implanté dans le système lorsque nous tentons de l'assainir est appelée *résistance* et la capacité d'un rootkit à survivre à un redémarrage du système est appelée *persistance*. La robustesse englobe donc à la fois ces deux notions de résistance et persistance.

Le pouvoir d'infestation d'un rootkit doit nous permettre d'évaluer à quel point est compromis le système sur lequel le rootkit est inséré. Elle indique comment le rootkit s'est « propagé » dans le système et donne donc une évaluation de l'étendue des dégâts causés. En effet, comme nous l'avons vu avec la définition d'un rootkit (cf. section 4.1), nous avons pu en constater le caractère parasitaire : un rootkit n'est pas un programme autonome mais un ensemble de modifications effectuées sur le système.

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 d'infestation 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 C. Cachin [14] concernant la formalisation de systèmes stéganographiques. Cachin [6] 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. Nous pouvons donc exploiter cette piste pour évaluer le niveau d'invisibilité d'un rootkit. Certes, ce parallèle n'est probablement pas suffisant car, comme nous l'avons souligné, les informations camouflées par un rootkit ne sont pas de la même nature que celles qui sont dissimulées dans un stégano-medium. Cependant, cette mesure est au moins un point de départ sur laquelle nous pouvons nous appuyer.

À notre connaissance, il n'existe pas de travaux en cours permettant d'associer une mesure à la notion de robustesse.

Enfin, pour mesurer le pouvoir d'infestation d'un rootkit, nous pourrions penser, dans un premier temps, se reposer sur des tests d'intégrité de fichiers. Néanmoins, les fonctions de hash classiques, puisque respectant le principe d'avalanche[14], sont trop sensibles à la moindre modification apportée au système[15]. Pour cela, nous préférons alors des outils comme la distance de Levenshtein qui permet de mesurer la similarité entre deux chaînes de caractères (par exemple des exécutables). Mais le pouvoir d'infestation 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

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

[15] Elles 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.

dans le système (processus), ce qui est plus délicat puisque ces modifications se font sur des données en mémoire.

Dans la suite de cet article, nous nous focalisons principalement sur la propriété d'invisibilité.

5 Un exemple de construction d'un rootkit « furtif »

Dans cette section, nous présentons le rootkit « furtif » 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 backdoor. Nous finissons sur les aspects de furtivité des activités de l'attaquant, correspondant aux services primordiaux d'un rootkit[16].

5.1 Principe général du vecteur d'interaction avec le noyau

Notre approche consiste à ne corrompre qu'un seul processus/thread 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 *per-process syscall hooking* [1] n'est pas comparable à la notre car elle agit uniquement au niveau de l'espace utilisateur[17] : aucune exécution de code noyau malicieux 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 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 peuvent nécessiter un redémarrage) 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).

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 (comme nous venons de le voir précédemment). Ainsi nous sommes plus discrets que les approches courantes affectant le système globalement.

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

[16] Nous ne traitons pas dans cet article de la dissimulation de fichiers.

[17] 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 malicieux.

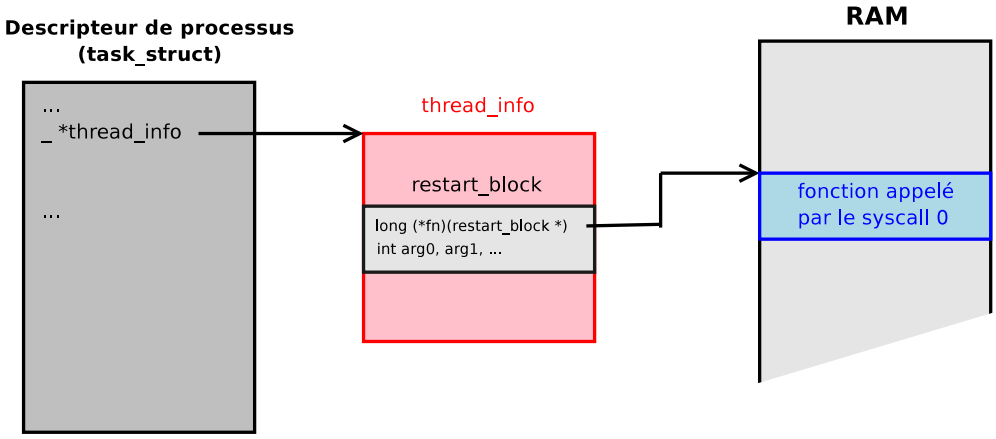

FIG. 2: Le descripteur de processus et l'appel système 0

avant d'effectuer l'installation du dit rootkit. Cependant, nous n'expliquons pas cette dissimulation ici mais dans la partie 5.5, traitant de l'invisibilité.

5.2 Installation préliminaire du cœur de notre rootkit

Tout d'abord depuis notre processus (celui de l'attaquant), nous ouvrons le périphérique virtuel `/dev/kmem` (cf. annexe B) pour accéder à l'espace d'adressage du noyau. Il nous faut pour cela avoir les bons privilèges (acquis avant l'installation d'un rootkit).

Nous recherchons ensuite au travers de `/dev/kmem`, la primitive `get_page` (plus exactement son adresse) par *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). La technique employée étant assez complexe, nous la détaillons dans l'annexe A.

L'étape suivante consiste à injecter du code au fond de la pile noyau de notre processus. Ce code est uniquement constitué d'un appel à la primitive précédente (`get_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 quelles fonctions du noyau depuis l'espace utilisateur. 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 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 (i.e. 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 allons voir dans ce qui suit.

5.3 Dissimulation du cœur du rootkit

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 article uniquement le premier cas au travers de deux méthodes.

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) 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).

Ce qui est intéressant avec l'allocateur VMALLOC, c'est que seul le PGD de référence du noyau (présent dans la structure `init_mm`) est modifié lors d'une allocation. Ainsi aucun processus ne peut accéder directement à la page mémoire allouée. Lors d'un accès à cette page, le gestionnaire de faute de page prend la main et met à jour le PGD du processus y accédant en le synchronisant avec le PGD de référence.

Ainsi, après avoir réservé une page mémoire via VMALLOC, il nous suffit de supprimer dans le PGD de référence l'adresse linéaire de la page concernée afin d'empêcher les mises à jour des processus essayant d'accéder à cette page. De plus, les adresses linéaires de la zone de VMALLOC peuvent être associées sans restriction à n'importe qu'elle adresse physique. Ainsi, pour qu'un processus retrouve la page physique que nous utilisons, il devra parcourir toute la mémoire.

Finalement nous pouvons modifier ou bien supprimer le descripteur de zone créé par l'allocateur afin d'améliorer le camouflage de la page malveillante. Toutefois, le cas de la suppression est dangereux car VMALLOC peut alors utiliser notre zone pour une future allocation.

Modification des bits de contrôle utilisés par la MMU. Nous citons ici l'exemple du rootkit Shadow Walker [35] car il s'agit d'une alternative à notre approche, très pertinente (cependant elle dépend totalement d'une spécificité matérielle que nous retrouvons tout de même 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 malicieux.

L'étape primordiale pour un *rootkit* est d'installer une *backdoor*. La section suivante présente nos différentes approches.

5.4 Fin de l'installation du cœur du rootkit : la backdoor

Nous allons présenter, avec notre méthode, comment un programme ayant les droits d'un utilisateur classique peut interagir avec l'espace noyau. Il nous sera alors possible d'acquérir les droits *root* à notre convenance. Cependant, un des atouts de notre technique est qu'il est possible d'exécuter n'importe quelle primitive du noyau à partir d'un processus utilisateur classique, ce qui favorise de surcroît l'invisibilité de l'interaction avec le noyau (en ne réveillant aucun soupçon chez l'administrateur).

Ainsi, la plupart de la logique des opérations malicieuses du rootkit s'effectue dans le programme client situé sur la machine de l'attaquant. Ensuite, les commandes (i.e. des appels système 0) qu'il souhaite faire exécuter à la machine compromise sont relayées depuis sa machine par l'intermédiaire d'un mécanisme de *syscall proxy* [5]. Seuls des appels système 0 ont besoin d'être relayés, ainsi ce mécanisme nous convient totalement. Cependant, les approches de type *remote userland execve* [11] sont également à envisager si nous souhaitons exécuter des programmes non présents sur la machine compromise (par exemple *nmap*).

Voyons à présent, les trois types de *backdoor* que nous proposons.

Les deux premiers s'articulent sur la création d'un *thread* noyau. Nous faisons l'hypothèse dans ces deux approches, que pour récupérer le contrôle de la machine compromise, l'attaquant s'y connecte en tant qu'utilisateur classique.

Finalement le dernier type de backdoor s'appuie sur notre technique de parasitage mobile, mentionnée dans la section 5.5.

Premier type de backdoor. Elle consiste à cacher partiellement un *thread* noyau. Par cela, nous entendons le délier uniquement de la liste regroupant l'ensemble des *threads* pour le cacher du système de fichier `/proc` et donc des utilitaires d'activité système s'y reposant 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`).

Dans ce mode, depuis l'espace utilisateur nous communiquons via les signaux avec notre *thread* noyau. Dans ce cas le *thread* a pour rôle de mettre en place un gestionnaire de signal et de se mettre

en attente. Le but du gestionnaire établi est de répondre à un signal venant de l'espace utilisateur[18]. Une fois le signal émis, il est ensuite géré par le *thread* noyau, lequel va alors parcourir la liste des processus utilisateurs jusqu'à trouver celui qui a émis le signal[19].

Une amélioration de cette approche consiste à utiliser une séquence de signaux (plutôt qu'un seul) avec des écarts temporels d'émission différents. Ainsi, nous assurons un minimum d'authentification pour l'attaquant.

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 mettant en œuvre plusieurs *threads* se recréant mutuellement empêcheraient notre défense active de fonctionner, en renforçant le critère de robustesse.

Deuxième type de backdoor. 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 ainsi 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[20] à la recherche de celui correspondant à l'identifiant de l'attaquant (i.e. 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. Ensuite, nous injectons dans le processus le code du *syscall proxy* et le lui faisons exécuter.

Troisième type de backdoor. Précédemment, la modification nécessaire au détournement de l'appel système 0 (i.e. le changement d'une adresse) était effectuée dans le processus de l'attaquant sur le système compromis. De même, le code du *syscall proxy* était injecté dans ce processus. Dans cette approche, il n'est plus nécessaire à l'attaquant de se connecter sur le système compromis.

En effet, le *syscall proxy* est dorénavant exécuté par parasitage mobile des processus présents dans le système. Nous expliquons notre algorithme de parasitage mobile dans la section 5.5. De même, la modification de l'adresse de l'appel système 0 suit cette stratégie. Pour comprendre les apports du parasitage en ce qui concerne la dissimulation de l'activité du *rootkit*, nous renvoyons le lecteur à la section 5.5.

Après avoir dissimulé notre rootkit et installé sa backdoor, nous pouvons revenir sur le système compromis et en garder le contrôle. Dans la suite, nous nous intéressons aux services que fourni le rootkit afin, d'une part, de dissimuler son activité système engendrée par l'attaquant (cf. section 5.5), et d'autre part, de dissimuler l'activité réseau induite par l'utilisation du rootkit par l'attaquant (cf. section 4.3).

[18] 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.

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

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

5.5 Dissimulation de l'activité système

Dans cette section, nous proposons trois méthodes pour dissimuler l'activité système de l'attaquant. Nous commençons par développer la méthode que nous avons mentionnée dans la section 5.4.

Cacher un processus. Pour dissimuler l'activité de l'attaquant, nous devons rendre invisible 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 5.4 d'une méthode pour cacher un processus en le déliant de la liste doublement chaînée les parcourant, 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 à besoin de lancer des programmes sur la machine et ces derniers doivent rester cachés sur le système.

Une méthode naïve est de supprimer, après l'appel de la fonction `sys_fork` (créant un nouveau processus) par notre processus, les liaisons 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 palier ce problème nous dupliquons en mémoire le code de l'appel système `sys_fork` et de la fonction `copy_process` (appelée par cette première) 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 pas lié à 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.

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.

Pour cela, nous créons 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 (i.e. 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. En effet, nous évitons ainsi de monopoliser le processeur et donc d'alerter l'administrateur.

Parasitage 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 parasitage *mobile* de processus ou de threads noyau s'exécutant sur le système compromis[21]. 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 en mémoire noyau, (2) nous déroutons l'exécution d'un thread pour 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[22].

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 leur descripteur sur la figure 3). 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).

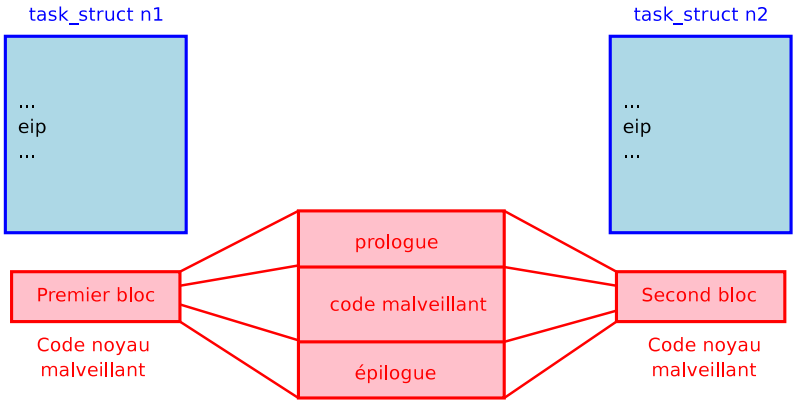

FIG. 3: Vol de cycles d'exécution 1/2

[21] 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 le même pour tous ses threads.

[22] 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.

Nous montrons grossièrement son fonctionnement sur la figure 4. 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 parasitage 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.

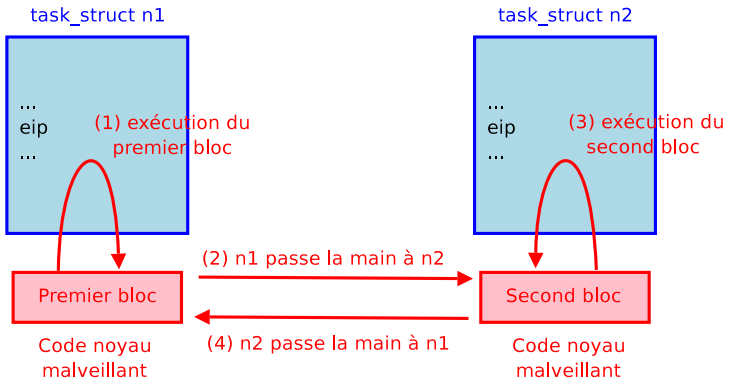

FIG. 4: Vol de cycles d'exécution 2/2

6 Synthèse de l'expérimentation

Rappelons que le système considéré est l'architecture x86 démunie d'extension matérielle de support pour la virtualisation. Ainsi nous ne considérons pas les rootkits hyperviseur dans notre synthèse.

6.1 Compatibilité et protection du rootkit

Afin que notre rootkit soit compatible au maximum avec les différentes version du noyau, il est profitable 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 codes (i.e. 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. Nous verrons que cela est profitable à notre rootkit.

Pour favoriser la dissimulation de notre rootkit ou bien de ses activités (i.e. favoriser le critère d'invisibilité), il est pertinent d'agir au niveau de l'environnement des points critiques du noyau (i.e. 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[23].

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 5.1). 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*.

6.2 Contribution

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

Vecteur d'interaction avec le noyau. 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 rootkit :

- *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* employé par les rootkits qui affectent de manière globale le système. Rappelons que la technique de *per-process syscall hooking* [1] n'est pas comparable à notre méthode (cf. section 5.1), 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.
  Suivant la taxonomie de J. Rutkowska [30], notre rootkit est de type II : la corruption du noyau est effectuée dans des zones non figées (par exemple les zones de données).

[23] 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.

Par rapport aux rootkits de type II, 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 5.2) 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 de notre connaissance ne mettent pas en place un tremplin permettant de profiter de tous les mécanismes du noyau. En effet, la majorité des opérations malicieuses 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 [16,24,11] 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 malicieuses entreprises par l'attaquant[24].

Dissimulation du rootkit. Afin de camoufler le code et les données de notre rootkit, nous avons proposé une méthode profitant des caractéristiques de l'allocateur mémoire non-contiguë du noyau Linux (cf. section 5.3).

- *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 standards fournies par le noyau du système, comme le ferait n'importe quel sous-système.

- *Efficacité :*
  La méthode de Shadow Walker [35] (cf. section 5.3) impose directement par le matériel la dissimulation du 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.

Backdoor. Nous avons proposé trois approches différentes afin de rendre le contrôle du système compromis à l'attaquant (cf. section 5.4).

[24] Nous ne relayons que des appels système 0, entravant de surcroît la reconstruction de l'attaque.

– *Utilisation d'un mécanisme de relais :*
Nos backdoors utilisent un mécanisme de relais d'appels système. Ainsi, elles se déclenchent depuis le mode utilisateur. De prime abord, cela paraît moins intéressant que les approches interagissant avec l'attaquant depuis le noyau. Cependant, les solutions courantes mettant en œuvre cela, modifient le code du noyau. Cela entraîne une baisse de la furtivité du rootkit, celui-ci étant à considérer comme un malware de type I d'après la taxonomie de J. Rutkowska [30].

– *Parasitage mobile :*
La troisième backdoor envisagée, qui n'est cependant pas encore mise en œuvre, contient une nouveauté significative face aux techniques existantes. En effet, alors que les backdoors courantes parasitent par exemple un seul processus (présent sur la machine compromise), la notre se déplace successivement de processus en processus (cet ensemble de processus parasités étant pour le moment à déterminer à l'avance).
L'exécution de la backdoor n'altère que temporairement le travail des processus parasités. En cela, nous pouvons dire que nous améliorons la furtivité face aux approches de parasitage fixe. Cependant nous touchons plusieurs processus à la place d'un seul. Ainsi, suivant le type de détection mis en place sur le système compromis, la furtivité de notre backdoor peut varier.

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 cache les processus qu'ils créent *a posteriori* (i.e. une fois qu'ils sont démarrés) en supprimant leurs liens avec le système. Nous avons présenté une démarche inverse (cf. section 5.5), 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 5.5). Les approches qui camouflent systématiquement tous les processus ayant un UID spécifique parviennent à un résultat proche.

– *Parasitage mobile :*
Notre algorithme transite sur tout ou partie des processus ou des threads noyau du système (cf. section 5.5 pour une brève présentation du principe). Les techniques de parasitage 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 malicieuse en rendant temporaire la corruption des processus.

6.3 Limites

Vecteur d'interaction avec le noyau. La principale limite de notre vecteur d'interaction et 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 courant pour des primitives critiques ou stables, ils restent plus probables que des modifications de l'API de la *libc* ou de l'ABI du noyau. Ainsi, les attaques de type *remote userland execve* [11] possèdent cet atout indéniable qu'est la compatibilité garantie quelque soit la version du noyau du système compromis.

Dissimulation du rootkit. 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 mémoire non-contiguë du noyau Linux ne résiste pas à un 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[25].

- *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.

Backdoor. Suivant la backdoor que nous utilisons, les limites différent.

- *Première et deuxième backdoor :*
  Elles créent un thread noyau qui est ensuite caché par suppression de la majorité de ses liens avec le système compromis. La limite de ces backdoors est que pour récupérer le contrôle de la machine compromise, l'attaquant a besoin de s'y connecter. Dans les situations où l'attaquant n'a pas la possibilité de se reconnecter au système compromis en tant que simple utilisateur, ces backdoors ne fonctionnent pas.

- *Troisième backdoor :*
  Notre troisième type de backdoor souffre des mêmes limites que notre technique de parasitage mobile. Nous les abordons dans la suite de cette section.

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

[25] 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 la structure `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[26].

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

- *Robustesse limitée :*
  Au travers de notre parasitage mobile, lorsque le code malicieux s'exécute à un moment donné sur un certain processus, sa survie (i.e. 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 malicieux part avec lui.

- *Difficultés d'implémentation de la charge malicieuse :*
  La structure de la charge malicieuse doit être pensée spécifiquement pour fonctionner avec notre algorithme. Ainsi, par exemple, pour le cas de deux processus, le code malicieux 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 la furtivité du code malicieux est faillible. Certains des processus parasités pourraient être des leurres mis en place par la défense afin de détecter un parasitage mobile.

7 Conclusion et perspectives

Nous avons mis en évidence dans cet article : d'une part les principes nous permettant de modéliser les rootkits et d'autre part l'approche du rootkit « furtif » au travers du détournement malicieux de sous-système noyau. Aussi, nous développons une réflexion sur l'un des objectifs primordiaux du rootkit, à savoir dissimuler l'activité de l'attaquant. Pour cela, nous poursuivons l'idée qu'agir le plus longtemps possible de façon normale avant d'opérer frauduleusement, peut nous affranchir en grande partie des problèmes liés à la détection. 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, etc.). Ainsi, en agissant au plus tard de façon frauduleuse, l'attaquant prend moins de risque d'être repéré.

Ces constatations nous encouragent dans notre démarche visant à caractériser les rootkits afin de mieux les évaluer. Nous avons ainsi dégagé trois critères essentiels les qualifiants. L'étape suivante consiste à définir une mesure pertinente sur ces critères. La finalité étant d'obtenir un comparateur objectif sur les rootkits.

Aussi, nous avons exposé l'architecture fonctionnelle d'un rootkit en nous fondant sur sa définition. Il serait alors intéressant de mettre cette architecture en correspondance avec les critères que nous avons dégagés, afin d'en cerner les points essentiels. Pour cela, la formalisation, d'une part de l'architecture et d'autre part des critères, serait d'une aide certaine.

[26] 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.

Notre étude expérimentale s'est focalisée uniquement sur le noyau Linux. Aussi, il paraît essentiel de considérer les autres noyaux existants afin d'envisager de nombreux scénarii de détournement. Cette analyse servirait de base expérimentale à la détermination des facteurs qui favorisent la potentialité de détournement d'une fonctionnalité.

Références

1. 7a69ezine Staff : Linux Per-Process Syscall Hooking, `http://www.7a69ezine.org/article/Linux_Per-Process_Syscall_Hooking` (2006).
2. Bioforge : Hacking the Linux Kernel Network Stack. *Phrack*, 61 (2003).
3. Boileau, A. : Hit by a Bus : Physical Access Attacks with Firewire. In Ruxcon 2006, `http://www.security-assessment.com/files/presentations/ab_firewire_rux2k6-final.pdf` (2006).
4. Buffer : Hijacking Linux Page Fault Handler. *Phrack*, 61 (2003).
5. Caceres, M. : Syscall Proxying - Simulating remote execution. Core Security Technologies (2002).
6. Cachin, C. : An information-theoretic model for steganography. In Proc. Int. Workshop on Information Hiding (1998).
7. C0de : Reverse symbol lookup in Linux kernel. *Phrack*, 61 (2003).
8. Cesare, S. : Kernel Function Hijacking (1999).
9. Cesare, S. : Syscall redirection without modifying the syscall table (1999).
10. Dornseif, M., and al. : FireWire - all your memory are belong to us. In CanSecWest/core05, `http://md.hudora.de/presentations/#firewire-cansecwest` (2005).
11. Dralet, S., Gaspard, F. : Corruption de la Mémoire lors de l'Exploitation. In Actes du Symposium sur la Sécurité des Technologies de l'Information et des Communications 2006, pp. 362–399, `http://actes.sstic.org/` (2006).
12. Filiol, E. : Les virus informatiques : théorie, pratique et applications. Collection IRIS, Springer Verlag France (2004).
13. Filiol, E. : Strong Cryptography Armoured Computer Viruses Forbidding Code Analysis : the Bradley Virus. In Proceedings EICAR2005 annual conference 14, StJuliens/Valletta - Malta, (2005).
14. Filiol, E. : Techniques virales avancées. Collection IRIS, Springer Verlag France (2007).
15. Girling, C. G. : Covert Channels in LAN's. *IEEE Transaction on Software Engineering*, février (1987).
16. Grugq : Remote Exec. *Phrack*, 62 (2004).
17. Intel : IA-32 Intel Architecture Software Developer's Manual Volume 2 : Instruction Set Reference (2003).
18. Kad : Handling Interrupt Descriptor Table for fun and profit. *Phrack*, 59 (2002).
19. King S. T., and al. : SubVirt : Implementing malware with virtual machines. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (2006).
20. Kruegel, C., Robertson, W., Vigna, G. : Detecting Kernel-Level Rootkits Through Binary Analysis (2004).
21. Kumar, N., Kumar, V. : Boot Kit. `http://www.rootkit.com/newsread.php?newsid=614` (2006).
22. Lawless, T. : On Intrusion Resiliency (2000).
23. Microsoft Corporation : Digital Signatures for Kernel Modules on Systems Running Windows Vista. Microsoft Corporation (2006).
24. Pluf, Ripe : Advanced Antiforensics : SELF. *Phrack*, 63 (2005).

25. Raynal, F. : Les canaux cachés. *Techniques de l'ingénieur*, décembre (2003).
26. Raynal, F., Berthier, Y., Biondi, P., Kaminsky, D. : Honeypot forensics : analyzing system and files. *IEEE Security & Privacy Journal* (2004).
27. Riordan, J., Schneier, B. : Environmental Key Generation Towards Clueless Agents. Lecture Notes in Computer Science 1419, pp. 15–24, `citeseer.ist.psu.edu/639981.html` (1998).
28. Rowland, C. H. : Covert Channels in the TCP/IP protocol suite. *First Monday*, `http://www.firstmonday.dk/issues/issue2_5/rowlad/` (1996).
29. Rutkowska, J. : Subverting Vista Kernel For Fun And Profit. In Black Hat in Las Vegas 2006, `http://invisiblethings.org/papers.html` (2006).
30. Rutkowska, J. : Stealth Malware Taxonomy. `http://invisiblethings.org/papers.html` (2006).
31. Rutkowska, J. : Beyond The CPU : Defeating Hardware Based RAM Acquisition Tools (Part I : AMD case). In Black Hat DC 2007, `http://invisiblethings.org/papers.html` (2007).
32. Rutkowski, J. K. : Execution path analysis : finding kernel based rootkits. *Phrack*, 59 (2002).
33. Sd, Devik : Linux on-the-fly kernel patching without LKM. *Phrack*, 58 (2001).
34. Soeder, D., Permeh, R. : eEye BootRoot : A Basis for Bootstrap-based Windows Kernel Code. eEye Digital Security, `http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf` (2005).
35. Sparks, S., Butler, J. : Raising The Bar For Windows Rootkit Detection. *Phrack*, 63 (2005).
36. Stealth : Kernel Rootkit Experience. *Phrack*, 61 (2003).
37. The Honeynet Project Staff : Know Your Enemy : Sebek - A kernel based data capture tool. `http://www.honeynet.org/papers/sebek.pdf` (2003).
38. Truff : Infecting loadable kernel modules. *Phrack*, 61 (2003).
39. Wolf, M. : Covert Channels in LAN Protocols. In LANSEC '89 : Proceedings on the Workshop for European Institute for System Security on Local Area Network Security, pp. 91–101, Springer-Verlag (1989).

A Recherche du descripteur du processus courant sous Linux 2.6

Pour retrouver notre descripteur de processus, nous repérons tout d'abord la pile noyau associée car une structure `thread_info` se trouve à la fin de cette pile et que c'est elle qui contient une référence à notre descripteur.

Pour retrouver l'emplacement de notre pile noyau il nous faut connaître la valeur du pointeur de pile `esp` lorsque notre processus séjourne en espace noyau. Pour cela notre approche est fondée sur la connaissance du fonctionnement des appels système dans Linux sur architecture *x86* depuis sa dernière version 2.6. Ce fonctionnement repose sur l'instruction matérielle `sysenter` [17].

Décrivons brièvement le fonctionnement (fig. 5). Depuis l'espace utilisateur, `sysenter` est exécutée, à ce moment `eip` (le registre contenant la valeur du compteur d'instruction) et `esp` (le registre contenant l'adresse de la base de la pile) sont positionnés à des valeurs prédéfinies lors de la compilation du noyau (ces valeurs sont chargés dans des registres spécifiques du processeur lors de l' initialisation du système) et le processeur passe en *ring 0* (i.e. le mode noyau).

La valeur de `esp` est donc toujours la même lors du passage en espace noyau, or les processus ont chacun leur propre pile noyau. En fait, la première instruction exécutée lors du passage en *ring 0* est le chargement de `esp` avec la valeur du `esp` du processus choisi par l'ordonnanceur. Cette valeur de `esp` est placée par l'ordonnanceur dans la structure `tss_struct` qui constitue le *Task Switch Segment* utilisé sur les architectures *x86*. Linux 2.6 n'utilise qu'une seule structure de ce type en mémoire pour chaque processeur.

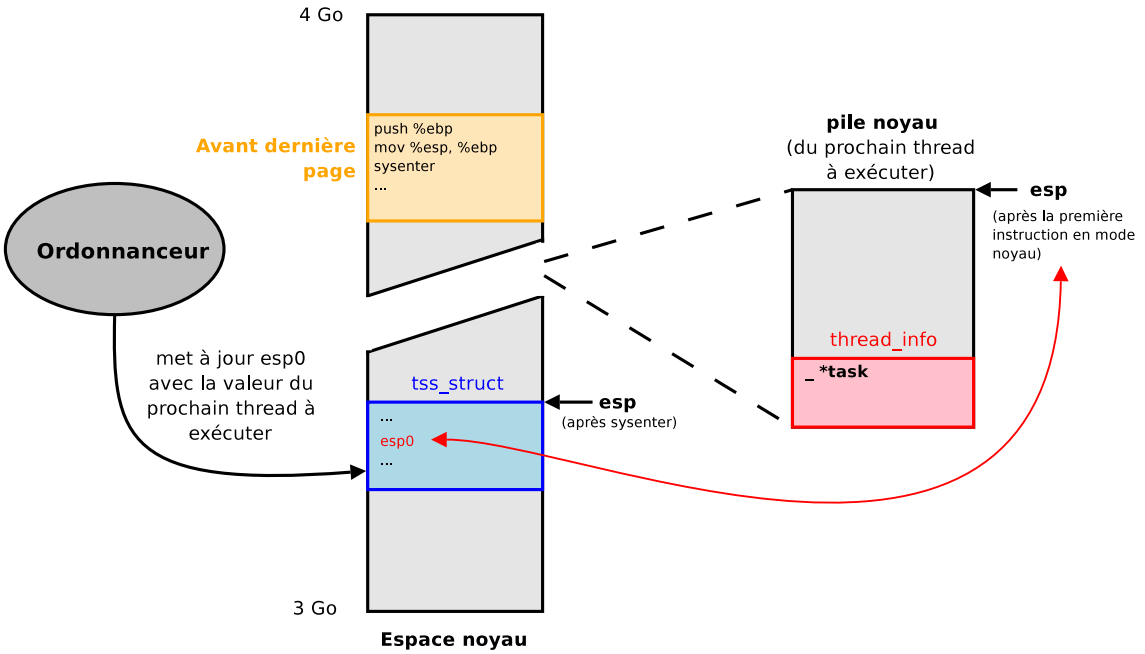

FIG. 5: Lien entre le descripteur de processus et l'instruction `sysenter`

Le registre `esp`, après l'appel à `sysenter`, est chargé avec l'adresse de cette structure. La première instruction peut donc récupérer l'adresse de la pile noyau du processus exécuté en se référant à `esp`.

Ainsi, pour récupérer l'adresse de notre pile noyau, il nous suffit d'aller lire à la position relative à `esp` lors de l'exécution de `sysenter`[27]. Le problème est de savoir comment récupérer la valeur affectée à esp lors du `sysenter`. Pour cela une instruction machine nous permet de récupérer sa valeur qui est stockée dans un registre spécifique du processeur. Cependant, cette instruction n'est exécutable qu'en *ring 0*. C'est pourquoi nous avons choisi de repérer par *pattern matching* sur `/dev/kmem` la fonction du noyau chargée de son initialisation, pour retrouver sa valeur. À partir de ce moment, nous connaissons l'emplacement de notre pile noyau et par conséquent celui de notre descripteur de processus.

B Injection au travers du périphérique virtuel /dev/kmem

Afin d'injecter du code ou de modifier le noyau au travers du périphérique virtuel `/dev/kmem`, les appels système de lecture et d'écriture suffisent. Nous proposons en illustration une partie des primitives que nous utilisons afin d'effectuer ces opérations.

Les fichiers d'en-tête nécessaires sont présentés dans ce qui suit, ainsi que la définition d'une fonction utilitaire.

[27] Lorsque notre processus lit la mémoire à cette position il s'agit bien de notre `esp0` car c'est lui qui s'exécute lorsqu'il effectue la lecture.

```
#define _LARGEFILE64_SOURCE
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

static unsigned int kernel_desc = 0;
static unsigned int ok = 0;

inline void OOPS(char *msg, int gravite){
  perror(msg);
  if(gravite)
    exit(EXIT_FAILURE);
}
```

L'ouverture du périphérique peut s'effectuer de la façon suivante.

```
int open_kmem(void)
{
  if((kernel_desc = open("/dev/kmem", O_RDWR|O_LARGEFILE)) < 0)
    OOPS("Could not open /dev/kmem", 1);

  ok = 1;

  return 0;
}
```

Voyons à présent comment réaliser des primitives de lecture et d'écriture sur `/dev/kmem` facilitant l'injection de code. Pour la lecture, nous fournissons en arguments la position à laquelle nous souhaitons lire ainsi que le nombre d'octets à récupérer. La fonction s'occupe alors de réserver la mémoire nécessaire à la copie des informations lues et retourne un pointeur sur la zone allouée.

```
void *read_kmem(off64_t offset, unsigned int len)
{
  void *buff;

  if(!ok)
    OOPS("/dev/kmem is not open", 1);

  if(!(buff = malloc(len)))
    OOPS("Allocation error", 1);

  if(lseek64(kernel_desc, offset, SEEK_SET) != offset)
```

```
    OOPS("read_kmem: lseek error", 0);

  if(read(kernel_desc, buff, len) != len)
    OOPS("read_kmem: read error", 0);

  return buff;
}
```

L'écriture n'a encore rien de bien mystérieux. Elle prend en argument la position à laquelle nous souhaitons écrire, un pointeur sur la zone mémoire contenant ce que nous souhaitons injecter, ainsi que sa taille en octets.

```
unsigned int write_kmem(off64_t offset, void *buff, unsigned int len)
{
  if(!ok)
    OOPS("/dev/kmem is not open", 1);

  if(lseek64(kernel_desc, offset, SEEK_SET) != offset)
    OOPS("write_kmem: lseek error", 1);

  if(write(kernel_desc, buff, len) != len)
    OOPS("write_kmem: write error", 1);

  return len;
}
```

La valeur de la position en mémoire qu'attend `/dev/kmem` est une valeur supérieure à 0xC0000000. Il s'agit en effet du début de l'espace d'adressage linéaire du noyau.

Finalement, afin de se prémunir d'une injection de code utilisant cette technique, il suffit de désactiver ce périphérique (qui n'est au final qu'un *proxy* donnant accès à l'espace d'adressage du noyau). Pour cela un patch noyau est nécessaire car la désactivation n'est pas permise dans le noyau officiel. *Grsecurity* propose cela dans ses fonctionnalités. Les primitives de lecture et d'écriture sur `/dev/kmem` sont alors remplacées par des coquilles vides. 2