App vs Wild

Protection d'applications en environnement hostile

Stéphane Duverger

Airbus Group Innovations, Toulouse, FRANCE
stephane.duverger@airbus.com

Résumé De nombreux mécanismes de protection d'applications ont pour but la prévention d'exécution de code arbitraire, visant majoritairement à protéger les applications d'elles-mêmes, de leurs vulnérabilités potentielles.

Mais existe-t-il pour autant des mécanismes permettant de protéger des applications de leur propre environnement ? En particulier du noyau du système d'exploitation, pièce maîtresse permettant leur exécution. Est-il possible de garantir la confidentialité du code d'une application vis-à-vis d'un noyau malveillant ?

Nous avons tenté de répondre à cette question en proposant une protection reposant sur Ramooflax, solution de virtualisation légère et libre présentée à l'occasion du SSTIC 2011.

1 Introduction

La virtualisation, qu'elle soit assistée par le matériel ou non, a principalement été utilisée dans le domaine de la sécurité des systèmes d'information (SSI) comme outil de cloisonnement, de partitionnement de systèmes d'exploitation (OS) s'exécutant de manière concurrente sur les mêmes ressources matérielles.

Que l'on mutualise l'accès aux ressources d'un serveur ou que l'on souhaite voir s'exécuter un système Windows à l'intérieur d'un système Linux sur un poste client, l'intérêt de la virtualisation du point de vue de la sécurité est qu'elle est sensée garantir l'isolation des machines virtuelles entre elles mais également vis à vis du système hôte (le cas échéant).

Cependant, depuis quelques années, différentes solutions de virtualisation couvrant d'autres aspects de la SSI ont vu le jour : rootkit, debugger, contrôleur d'intégrité, sandboxing, etc.

Ces solutions se rapprochent pour la plupart du concept de micro-virtualisation au sein duquel un hyperviseur ne gère qu'une seule et unique machine virtuelle, correspondant potentiellement au système d'exploitation préalablement installé sur une machine physique. En d'autres termes, l'hyperviseur permet l'analyse/le contrôle d'une machine physique sur

laquelle un OS est déjà installé, soit en démarrant avant lui, soit au cours de son exécution.

Les extensions de virtualisation matérielle fournies par les CPUs ne servent plus à partager efficacement des ressources mais à l'introspection de la machine virtuelle et de ses mécanismes d'accès aux ressources nécessaires à la gestion de ses applications : instructions privilégiées, MMU et autres périphériques.

Néanmoins, peu de ces solutions ont tenté d'utiliser la virtualisation afin d'isoler des applications entre elles, voire de leur propre noyau de système d'exploitation.

Cet article détaille une solution innovante de virtualisation visant à protéger, du point de vue de la confidentialité et de l'intégrité, le code d'une application s'exécutant dans un environnement potentiellement corrompu.

Cette solution repose sur un hyperviseur libre et léger, Ramooflax, et se veut indépendante du système d'exploitation virtualisé.

Nous présentons dans un premier temps le contexte et l'intérêt de la protection, mais également quelques rappels concernant Ramooflax, hyperviseur minimaliste de type *bare-metal* présenté durant la conférence SSTIC 2011 [7]. Son architecture et ses limitations seront succinctement décrites afin de bien cerner son domaine d'application.

Nous aborderons ensuite les grands principes et propriétés de notre protection, ainsi que ses détails d'implémentation.

Puis nous examinerons les divers scénarios d'attaques envisageables et leurs impacts potentiels sur la protection. Nous effectuerons également quelques comparaisons avec différentes protections, logicielles ou matérielles, proposant le même type de fonctionnalité.

Nous conclurons cet article en évoquant les limites et évolutions potentielles de la protection.

2 Contexte

Le but de notre protection est de permettre le déploiement en environnement hostile ou non maintenu d'applications dont le code constitue une information sensible à protéger.

Évidemment, notre protection ne s'applique pas aux situations pour lesquelles un attaquant aurait un accès physique à la machine. Il serait pour cela nécessaire de protéger le démarrage de l'hyperviseur et ne ferait que déplacer le problème (*trusted boot*, ROM chiffrée, TPM, obscurcissement, etc.).

Un scénario typique pourrait être celui où un service d'hébergement de type *Cloud* propose de déployer des applications en fournissant des ressources matérielles et des systèmes d'exploitation standardisés (Windows, Linux) dont le niveau de sécurité est approximatif.

En effet, même avec tous les efforts possibles, le risque d'une exploitation *via* un *zero day* est largement envisageable ; ce qui pourrait abaisser le niveau de confiance de l'utilisateur souhaitant déployer des applications sensibles.

Notre solution permettrait à un hébergeur ou à l'utilisateur de ne plus se soucier du niveau de sécurité de l'OS et de ses applications. L'hyperviseur étant la seule surface d'attaque envisageable, dans le cas d'un accès distant. Il ne constituerait que la seule brique logicielle à maintenir, évaluer, certifier, etc.

2.1 Rappels concernant Ramooflax

Ramooflax est un hyperviseur de type I, *bare-metal*, prenant le contrôle d'un poste physique avant le boot de l'OS installé sur la machine. Il fait usage des extensions matérielles de virtualisation fournies dans les CPU Intel x86 et AMD, extensions dites de seconde génération pour leur support de la virtualisation de la MMU (EPT [1] et RVI [2]).

Initialement, Ramooflax a été conçu comme un outil d'analyse distant d'OS. Il met à disposition différentes interfaces physiques pour la prise de contrôle distante (carte réseau *Intel e1000*, *USB Debug Port EHCI* ou bien port série) ainsi qu'une API Python basée sur le protocole GDB.

Cette API permet de séparer la complexité de l'analyse de l'OS du cœur de fonctionnement de l'hyperviseur, pour une plus grande souplesse d'implémentation sous forme de scripts facilement réutilisables, adaptables, redistribuables.

Notons que Ramooflax est open-source et disponible sur GitHub [3].

La figure 1 présente l'architecture de Ramooflax et ses différents chemins d'exécution.

L'hyperviseur (VMM) est constamment en attente d'évènements générés par la machine virtuelle (VM). À l'arrivée d'un événement dont on souhaite l'interception, Ramooflax prend la main est procède à son traitement avant de donner la main à un module de contrôle communiquant avec le protocole GDB avec un poste d'analyse distant.

1. Extended Page Tables
2. Rapid Virtualization Indexing
3. https://github.com/sduverger/ramooflax

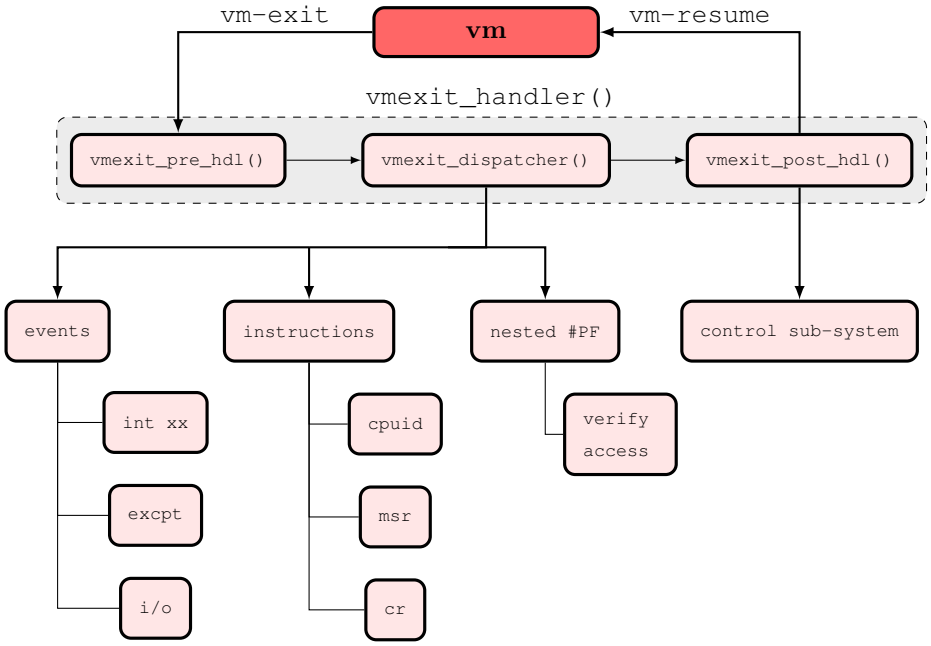

Figure 1. Chemins d'exécution de Ramooflax.

Cependant, Ramooflax propose aujourd'hui un mode *standalone* dans lequel il n'est plus nécessaire de disposer d'un accès distant à l'hyperviseur pour en prendre le contrôle. Il est dorénavant possible de programmer des plugins directement dans le cœur de l'hyperviseur et de traiter les événements interceptés grâce à son API interne.

Ce mode de fonctionnement est parfaitement approprié dans le cas de la spécialisation de l'hyperviseur à notre protection, pour laquelle il n'est plus souhaitable d'interagir à distance.

2.2 Vue d'ensemble de la protection

La protection a pour but d'empêcher l'analyse du code d'une application. D'une manière générale, elle s'applique à tous les éléments d'une application qui n'ont pas vocation à transiter par le noyau de l'OS l'exécutant.

Les données modifiables ne sont pas protégées par notre solution, en ce sens qu'elles peuvent être (et le sont généralement) manipulées par le noyau. Nous reviendrons plus en détail sur ce point.

L'idée principale de cette protection est d'offrir deux environnements mémoire distincts, disponibles à tout instant mais de manière exclusive. L'un contenant le système d'exploitation complet (noyau, bibliothèques et applications) que nous appellerons *origine*, l'autre ne contenant que ce qui est nécessaire et suffisant à la bonne exécution de l'application que nous appellerons *secret*.

L'application est déployée de manière chiffrée dans l'environnement *origine* grâce à une chaîne d'outils ne nécessitant pas le code source de l'application. Son chiffrement est opéré directement sur le binaire. Qui plus est, cela signifie également qu'il n'est pas nécessaire d'adapter l'application pour qu'elle puisse fonctionner avec la protection.

L'hyperviseur procède au déchiffrement de l'application au lancement de celle-ci, et expose les pages en clair uniquement dans l'environnement *secret*.

L'hyperviseur est responsable des transitions d'environnements mémoire durant toute la durée de vie de l'application protégée.

Un des intérêts majeur de notre approche est qu'elle est agnostique du système d'exploitation et ne repose que sur des considérations matérielles. Ainsi, il n'y a aucun filtrage ou mécanisme d'interception basé sur des paradigmes liés au système d'exploitation dans lequel s'exécute l'application à protéger.

La protection reste la même pour tout système d'exploitation fonctionnant en mode protégé 32/64 bits sur Intel x86 avec support de la virtualisation matérielle. Seule la chaîne d'outillage chiffrant le binaire de l'application est dépendante du format de l'exécutable. À ce jour, le format ELF32 est supporté dans notre preuve de concept (PoC) aussi bien pour des binaires statiques que dynamiques.

3 Conception

Nous venons d'exposer succinctement les grands principes de la protection, cette section se propose d'en présenter le fonctionnement détaillé.

3.1 Proposer deux vues de la mémoire physique

Les extensions matérielles d'aide à la virtualisation [4] des processeurs Intel permettent, depuis leur seconde génération, de virtualiser efficacement une MMU.

4. Virtualization Technology Extensions (VT-x), et leur jeu d'instructions VMX.

Si historiquement les hyperviseurs devaient implémenter des mécanismes relativement complexes, via des *Shadow Page Tables*[5], pour assurer un contrôle d'accès à la mémoire physique, ce n'est plus nécessaire de nos jours.

Cette extension, également appelée *Nested Page Tables*, ajoute un nouveau niveau d'indirection aux accès en mémoire physique effectués par une VM. Les adresses physiques de la VM sont désormais considérées comme des adresses virtuelles du point de vue de l'hyperviseur qui met en place un jeu de tables de pages les traduisant vers des adresses physiques finales dites *system physical*.

La dénomination de cette cascade d'accès devient alors :

— *guest virtual address* (traduite depuis *cr3* dans la VM) ;
— *guest physical address* (traduite depuis EPT dans le VMM) ;
— *system physical address*.

L'hyperviseur n'a plus qu'à initialiser un jeu de *Nested Page Tables* par VM avant leur lancement et le matériel effectuera la traduction de manière transparente. Outre une réduction considérable de la complexité du code de l'hyperviseur, cela réduit également drastiquement sa surface d'attaque. Le potentiel de vulnérabilités dans une implémentation de *Shadow Page Tables* n'étant vraiment pas à prendre à la légère.

Dans le cadre de notre protection, EPT nous permet de présenter à la VM des portions de la mémoire physique ; un peu à la manière d'espaces d'adressage virtuels de processus dans un OS. Plus particulièrement, Ramooflax initialise un espace d'adressage EPT (l'environnement *origine*) donnant accès à la quasi-totalité de la mémoire physique du système à la VM. L'hyperviseur conserve une partie de la RAM pour son propre usage mais également pour créer dynamiquement des espaces d'adressage EPT pour chaque application *secret* à protéger.

Chaque espace d'adressage *secret* ne contiendra que la quantité de mémoire nécessaire à l'exécution de l'application *secret*. Comme expliqué précédemment, c'est dans ces espaces d'adressage que les pages de code des applications à protéger seront respectivement déchiffrées.

La figure 2 illustre nos propos.

3.2 Propriété de sécurité

La confidentialité du code d'une application *secret* est à la fois assurée par la chaîne d'outils permettant de la déployer sur un système

5. Memory Virtualization, VMware Labs, https://labs.vmware.com/download/46/

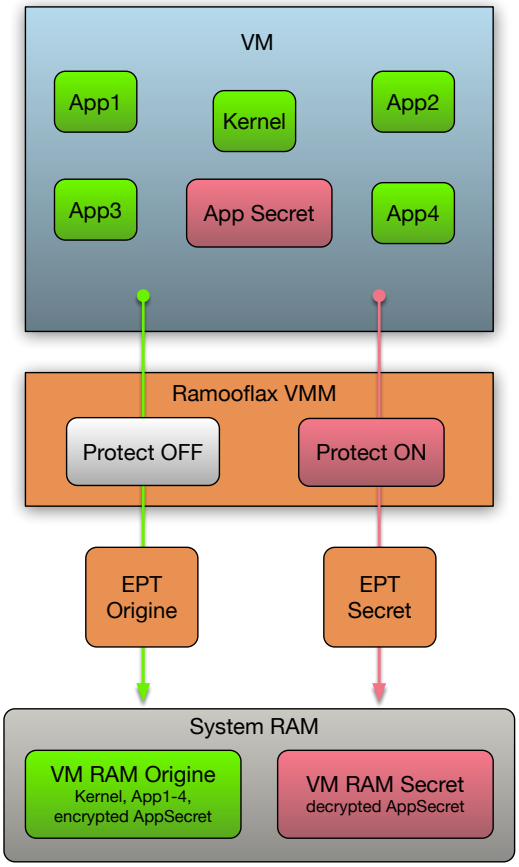

Figure 2. Deux environnements distincts de mémoire physique.

hostile (stockage sur disque par exemple), mais également par l'hyperviseur garantissant la propriété de sécurité suivante :

> Une page de mémoire exécutable ayant été déchiffrée et dont l'intégrité est vérifiée, est marquée *eXecute-only* dans l'environnement *secret* la sollicitant.

Nous parlons ici et dans le reste de cet article de mappings mémoire dans les environnement EPT, traduisant une adresse physique de la VM (*guest physical*) en adresse *system physical* finale. À aucun moment, l'hyperviseur ne modifie les mappings mémoire configurés et maintenus par l'OS. La protection agit un niveau en dessous.

Les extensions de virtualisation EPT des processeurs Intel permettent de créer des mappings mémoire de type *eXecute-only*. Grâce à ce type d'entrées, l'hyperviseur est capable de rendre illisibles les pages de mémoire déchiffrées d'une application *secret*.

Le simple fait de mettre à disposition des environnements mémoire EPT isolés les uns des autres n'est pas suffisant pour garantir que le code d'une application *secret* ne puisse être lu. Une vulnérabilité dans l'application pourrait lui faire lire son propre code et l'exposer en dehors du contrôle de l'hyperviseur (ex. zone de mémoire partagée).

Les autres pages de mémoire physique nécessaires au bon fonctionnement d'une application *secret* mais ne nécessitant pas de déchiffrement sont mappées dans l'environnement *secret* avec les mêmes droits que l'environnement *origine*. Il s'agit principalement :

— de données *read-only* et *read-write* provenant du fichier exécutable ;
— de zones de mémoire dynamique : tas, pile, bibliothèques partagées ;
— de structures noyaux : *gdt, idt, tss, pgd, ptb*

Ceci permet en outre l'exécution d'applications *secret* compilées dynamiquement et dépendantes de bibliothèques partagées non chiffrées par notre chaîne d'outils.

Les structures noyaux mentionnées sont utilisées par le CPU lors des transitions de niveaux de privilèges. Les rendre présentes dans les environnements *secret* évite de générer des exceptions inutiles et permet à l'hyperviseur, comme nous le verrons dans la section suivante, d'être plus efficace dans l'interception des transitions de niveau de privilèges utilisateur/noyau.

3.3 Intercepter les changements de privilèges

Si l'on est capable de configurer des environnements mémoire isolés les uns des autres afin de protéger nos applications, il est à présent nécessaire de trouver un moyen de les mettre en place sur le CPU uniquement quand cela est requis.

Un environnement *secret* ne doit être présent sur le CPU que lorsque son application *secret* correspondante est en cours d'exécution.

D'une manière générale, sur architecture Intel x86 et dans le cas d'OS modernes en mode protégé, les applications s'exécutent en *ring 3* et le noyau en *ring 0*, ce qui correspond respectivement au niveau de privilèges le plus bas et le plus élevé.

Ramooflax doit à la fois intercepter les transitions de niveau de privilèges mais également être capable de faire une distinction, toujours au

niveau matériel, entre toutes les applications du système qui s'exécutent au même niveau de privilèges.

En premier lieu, Ramooflax est capable d'intercepter les accès en écriture dans le registre *cr3*, correspondant à l'adresse physique de la table [6] de plus haut niveau définissant l'espace d'adressage d'une application.

Chaque application ayant un espace d'adressage unique, ce registre est tout indiqué pour identifier matériellement une application. Notons que les paradigmes de l'OS liés à la gestion des tâches nous sont ici indifférents. Nous nous attachons uniquement à des considérations matérielles.

Nous aurions également pu nous intéresser au pointeur de pile noyau de l'application, lui aussi unique. Un noyau ayant généralement une pile dédiée en *ring 0* à la gestion de chaque application. Ce pointeur est habituellement stocké dans le champs *esp0* de l'unique *tss* géré par l'OS afin de permettre au CPU d'effectuer des transitions de niveau de privilèges de manière saine. Toutefois, ce champ est attaché à une structure de données allouée par le noyau sur laquelle Ramooflax devrait positionner un *breakpoint* en écriture. Dans le cas d'un noyau malveillant changeant constamment son/ses *tss*, les performances du mécanisme d'identification d'application n'auraient pas été optimales.

L'interception des écritures dans le registre *cr3* est donc, de notre point de vue, idéale.

À présent que Ramooflax est capable d'identifier l'application courante, il ne lui reste plus qu'à détecter à quel moment le CPU commence son exécution en *ring 3* mais également à quel instant et sous quelles conditions il l'interrompt pour entrer en *ring 0*.

Les transitions de *ring 3* à *ring 0* peuvent survenir suite à l'arrivée des évènements suivants :

— une interruption matérielle (*IRQ*) ;
— une exception du processeur ;
— un appel système via les instructions *int N/sysenter/syscall*.

Inversement, une transition de *ring 0* vers *ring 3* survient en général suite à l'exécution des instructions *iret/sysexit/sysret*. Plus précisément, les instructions permettant d'effectuer des transferts d'exécution inter-segments sont susceptibles de donner lieu à des changement de niveau de privilèges (*far jump, far call, far ret*). Les OS modernes n'en font pas un usage exhaustif et ne gèrent que deux niveaux de privilèges (0 et 3) afin d'isoler le noyau des applications. Notre protection s'est principalement focalisée sur ces transitions dites *user/kernel*.

6. ex. un *Page Directory* en mode paginé 32 bits.

Nous détaillerons dans la section 5 consacrée à l'implémentation de quelle manière Ramooflax intercepte véritablement les transitions de niveau de privilèges.

Finalement, une fois l'application identifiée et les transitions de privilèges interceptées, Ramooflax n'a plus qu'à mettre en place le bon environnement mémoire. Notons que dans un soucis d'optimisation des performances, Ramooflax désactive l'interception de transition de niveau de privilèges lorsque des applications non protégées sont ordonnancées.

La figure 3 illustre le mécanisme d'interception de changement de niveau de privilèges au travers d'un exemple d'ordonnancement de processus :

- une application *secret* tente de rentrer en *ring 0* suite à l'arrivée d'une interruption ou à l'exécution d'un appel système ;
- Ramooflax intercepte le changement de niveau de privilèges et bascule immédiatement d'environnement EPT vers celui *origine* ;
- il laisse ensuite le code du noyau s'exécuter pour traiter l'évènement. À ce stade, le noyau décide d'ordonnancer un autre processus ;
- Ramooflax intercepte l'écriture dans le registre *cr3* et détecte qu'il ne s'agit pas d'une application protégée. Au moment où le noyau va tenter d'effectuer la transition de *ring 0* vers *ring 3* afin de démarrer l'exécution de cet autre processus (via un *iret/sysexit*) Ramooflax va désactiver son mécanisme d'interception jusqu'au prochain ordonnancement d'un processus *secret* ;
- lorsque le noyau décidera d'ordonnancer un processus *secret*, Ramooflax détectera encore une fois l'écriture dans *cr3* qui n'est jamais désactivée, chargera l'environnement EPT *secret* correspondant et réactivera le mécanisme d'interception de transition de niveau de privilèges.

4 Préparation d'une application *secret*

Notre chaîne d'outils permet de chiffrer les applications à déployer en environnement hostile. Elle s'applique à l'exécutable final et ne requiert pas l'accès au code source ou à l'environnement de compilation ayant permis de générer l'exécutable.

Nous supportons le format ELF 32 bits pour des binaires compilés statiquement ou dynamiquement (PIE/PIC ou non). Le support du format PE n'est pas disponible dans notre PoC.

Notre outil analyse le fichier exécutable et opère les modifications suivantes :

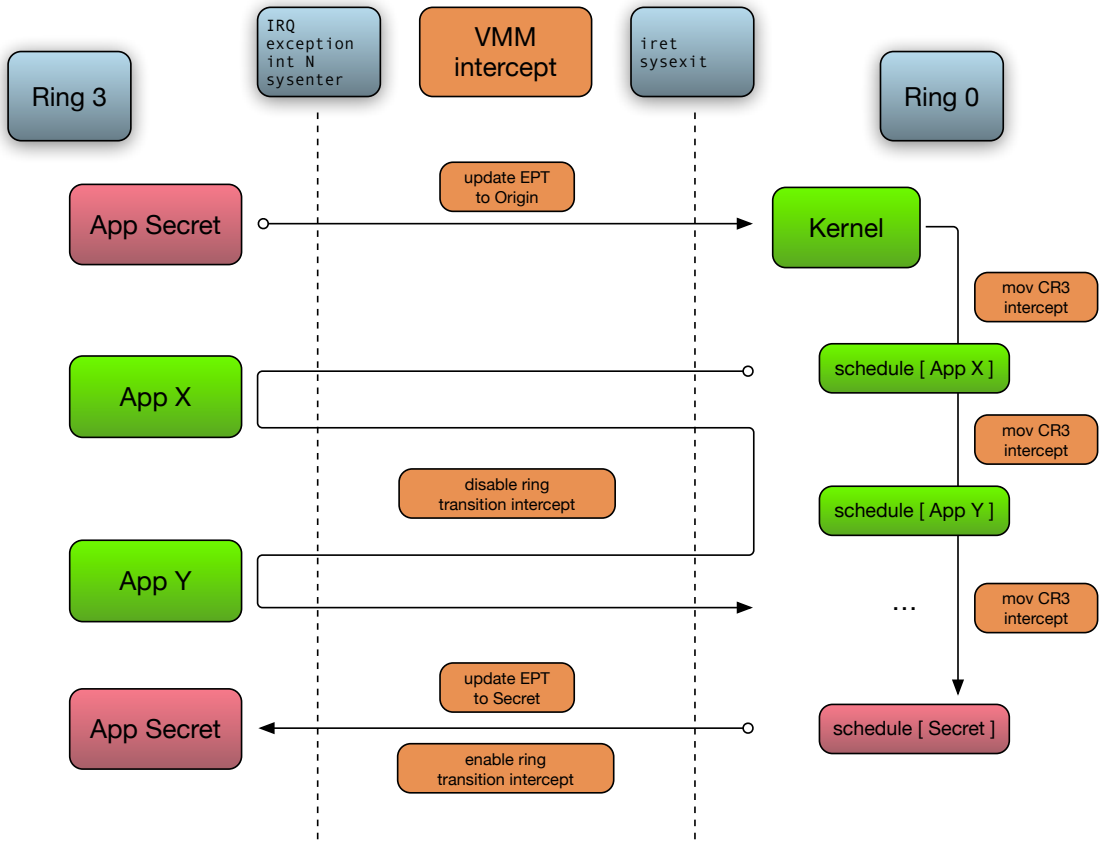

Figure 3. Interception des transitions de niveau de privilèges.

— chiffrement/signature des sections de code ;
— modification d'un *program header* ;
— ajout de méta-informations ;
— ajout d'un pré-chargeur minimaliste ;
— modification du point d'entrée.

Une vue d'ensemble d'un binaire ELF avant et après préparation est présentée dans la figure 4. Les modifications apportées apparaissent en rouge.

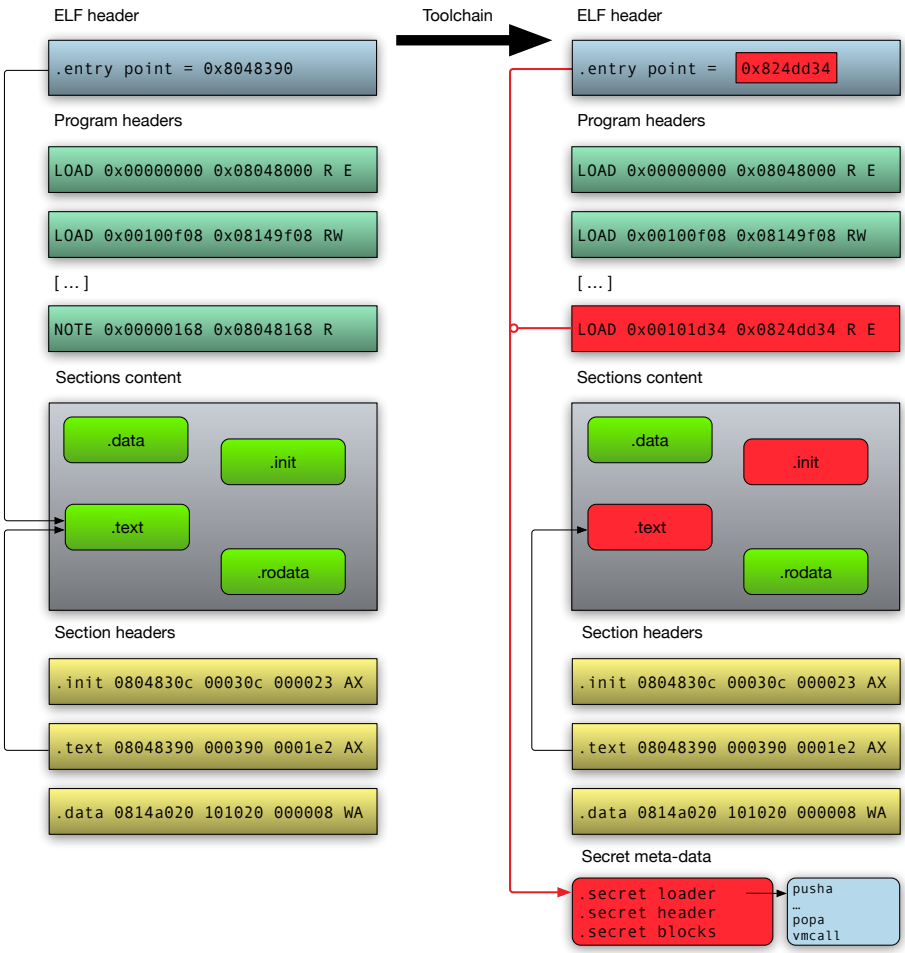

Figure 4. Modifications apportées au binaire à protéger.

4.1 Outillage

Nous fournissons en premier lieu un outil (*genkey.bin*) permettant de générer le matériel cryptographique utile à la fois à l'hyperviseur et à l'outil de chiffrement du binaire.

```
$ ./genkey.bin -c
AESK        : 6472779d30d18aa79fdce01a6ad497e6c7eb4dbff357c7899a8d22468211b4a7
AESIV       : 0ae733b7a18c727f2bd5c9b778de5200
HMACK       : 30019fef730d55589045d892bfda40e208761388c1cebce8fc336a7f2c940680
```

Cet outil génère un fichier *sec_keys.h* contenant les clefs cryptographiques qui seront compilées avec l'hyperviseur.

```
$ cat sec_keys.h
static uint8_t secret_aes_key[32] =
  "\x64\x72\x77\x9d\x30\xd1\x8a\xa7\x9f\xdc\xe0\x1a\x6a\xd4\x97\xe6"
  "\xc7\xeb\x4d\xbf\xf3\x57\xc7\x89\x9a\x8d\x22\x46\x82\x11\xb4\xa7" ;

static uint8_t secret_aes_iv[16] =
  "\x0a\xe7\x33\xb7\xa1\x8c\x72\x7f\x2b\xd5\xc9\xb7\x78\xde\x52\x00" ;

static uint8_t secret_hash_key[32] =
  "\x30\x01\x9f\xef\x73\x0d\x55\x58\x90\x45\xd8\x92\xbf\xda\x40\xe2"
  "\x08\x76\x13\x88\xc1\xce\xbc\xe8\xfc\x33\x6a\x7f\x2c\x94\x06\x80" ;
```

Ramooflax est également dépendant de fichiers d'en-tête utilisés par notre chaîne d'outils, décrivant la structure des méta-informations.

```
$ cat sec_phdr.h
typedef struct secret_process_block
{
   uint64_t size ;              /* block size */
   uint64_t offset ;            /* from load base */
   uint8_t  hash[HMAC_OUT_SZ] ; /* hash of encrypted block */

} __attribute__((packed)) sec_blk_t ;

typedef struct secret_meta_header
{
   uint64_t  aep ;        /* actual ELF entry point offset */
   uint64_t  oep ;        /* original ELF entry point offset */
   uint64_t  end ;        /* secret info ending offset */
   uint64_t  ret ;        /* sysenter caller */
   uint64_t  sys ;        /* sysexit offset */
   uint64_t  cnt ;        /* block count */
   sec_blk_t blk[0] ;     /* secret blocks */

} __attribute__((packed)) sec_mhdr_t ;
```

Une fois les clés générées, nous pouvons utiliser l'outil de chiffrement de binaires ELF en lui fournissant les paramètres suivants :

```
$ ./elfix.bin
FAIL : missing arguments
usage : ./elfix.bin <elf> <hex aes key> <hex iv> <hex hmac key>

  <elf>            ELF binary file to encrypt
  <hex aes key>    hexadecimal string of AES KEY     (32 bytes)
  <hex iv>         hexadecimal string of AES CBC IV  (16 bytes)
  <hex hmac key>   hexadecimal string of HMAC key
```

L'outil va générer un nouvel exécutable du même nom mais portant l'extension "*.sec*".

```
$ ./elfix.bin test                                                  \
 6472779d30d18aa79fdce01a6ad497e6c7eb4dbff357c7899a8d22468211b4a7  \
 0ae733b7a18c727f2bd5c9b778de5200                                  \
 30019fef730d55589045d892bfda40e208761388c1cebce8fc336a7f2c940680

--o00o-- Sections encryption
- checking section .interp              [offset 0x154 size 0x13]
- checking section .note.ABI-tag        [offset 0x168 size 0x20]
- checking section .note.gnu.build-id   [offset 0x188 size 0x24]
- checking section .gnu.hash            [offset 0x1ac size 0x20]
- checking section .dynsym              [offset 0x1cc size 0x70]
- checking section .dynstr              [offset 0x23c size 0x62]
- checking section .gnu.version         [offset 0x29e size 0xe]
- checking section .gnu.version_r       [offset 0x2ac size 0x30]
- checking section .rel.dyn             [offset 0x2dc size 0x8]
- checking section .rel.plt             [offset 0x2e4 size 0x28]
- checking section .init                [offset 0x30c size 0x23]
- checking section .plt                 [offset 0x330 size 0x60]
- checking section .text                [offset 0x1390 size 0x1d2]
 \_ new page [0x1390 - 0x2000]
  _ remain 0x000001d2 / sz 0x01d2]
  _ not full [0x1390 - 0x1562]
- checking section .fini                [offset 0x2564 size 0x14]
- checking section .rodata              [offset 0x2578 size 0x1d]
- checking section .eh_frame_hdr        [offset 0x2598 size 0x2c]
- checking section .eh_frame            [offset 0x25c4 size 0xb0]
- checking section .init_array          [offset 0x2f08 size 0x4]
- checking section .fini_array          [offset 0x2f0c size 0x4]
- checking section .jcr                 [offset 0x2f10 size 0x4]
- checking section .dynamic             [offset 0x2f14 size 0xe8]
- checking section .got                 [offset 0x2ffc size 0x4]
- checking section .got.plt             [offset 0x3000 size 0x20]
- checking section .data                [offset 0x3020 size 0x8]
- checking section .bss                 [offset 0x3028 size 0x1020]
 === encrypt 0x1390 0x01d2

--o00o-- Program Header fixup
- NOTE offset 0x168 vaddr 0x8048168 filesz 0x44 memsz 0x44
- LOAD offset 0x00003d10 vaddr 0x08051d10 memsz 0x00000110

--o00o-- Output file : selfmap.d/selfmap_32.alg.sec
- total encrypted bytes   0x1d2
- secret meta-data offset 0x0000000000003d10

--o00o-- Secret Meta Header offset 0x0000000000003dc0
  - actual   entry off  : 0x0000000000009d10
  - original entry off  : 0x0000000000001390
  - sec info end   off  : 0x0000000000009e20
  - sysexit trampo off  : 0x0000000000009da2
  - block count         : 0x0000000000000001

--o00o-- Secret Block[0] :
  - offset    : 0x0000000000001390
  - size      : 0x00000000000001d2
  - hash      : e0f4754333278fe97a0ae1d1aeaf7245c47595a1348919d828e01c729dae0f59
```

4.2 Chiffrement des sections de code

L'outil, en parcourant l'exécutable ELF, va créer des *secret_process_block* dans les méta-informations. Chaque bloc indique l'*offset* des données chiffrées (relatif à l'adresse de base de chargement du programme), leur taille ainsi qu'un HMAC du chiffré.

L'outil regroupe les sections de code par blocs de données d'une taille correspondant au plus à la granularité la plus fine d'une page de mémoire virtuelle, soit 4096 octets (4Kio). Ceci permet à l'hyperviseur de déchiffrer l'application *secret* page par page à la demande, quand celles-ci sont réellement rendues disponibles par l'OS. La section 5 détaillera le mécanisme de déchiffrement.

Chaque bloc est chiffré en AES-256-CBC lorsque cela est possible. Tous les 4Kio, un nouveau bloc est créé. Lorsque la taille d'un bloc à chiffrer est plus petite ou n'est pas un multiple de la taille d'un bloc AES (16 octets), nous chiffrons l'excédant en AES-256-CFB.

Les contraintes d'alignement des sections ainsi que leur adresse de chargement en mémoire sont respectées par l'outil. Des sections contiguës présentes en tout ou partie dans une même page de mémoire, peuvent ainsi être chiffrées dans un même *secret_process_block*.

Le HMAC des *secret_process_block* est réalisé via un SHA-256 HMAC. Ce hash porte sur le nombre exact d'octets dans le bloc chiffré. Le choix d'un HMAC permet à la fois de vérifier l'intégrité du binaire mais également d'empêcher un attaquant de préparer des binaires chiffrés invalides et de forcer l'hyperviseur à les déchiffrer. Le hash des chiffrés ne représente d'ailleurs pas une fuite d'information potentielle dans le cas où la primitive de hachage serait réversible.

La figure 5 illustre ce procédé de chiffrement.

4.3 Modification d'un *Program Header*

Afin d'injecter dans le binaire nos méta-informations, nous avons décidé de modifier un *program header* non essentiel à l'exécution du programme et systématiquement créé par les chaînes de compilation GNU standard : *PT_NOTE*.

L'ajout d'un nouveau *program header* n'était pas une solution idéale dans la mesure où ceux-ci sont situés avant le contenu des sections et aurait eu pour conséquence de fausser tous les *offsets* précalculés à l'édition de liens.

L'ABI System-V [9] mentionne que la section *NOTE* et son *program header PT_NOTE* sont optionnels. De plus le chargeur ELF du noyau

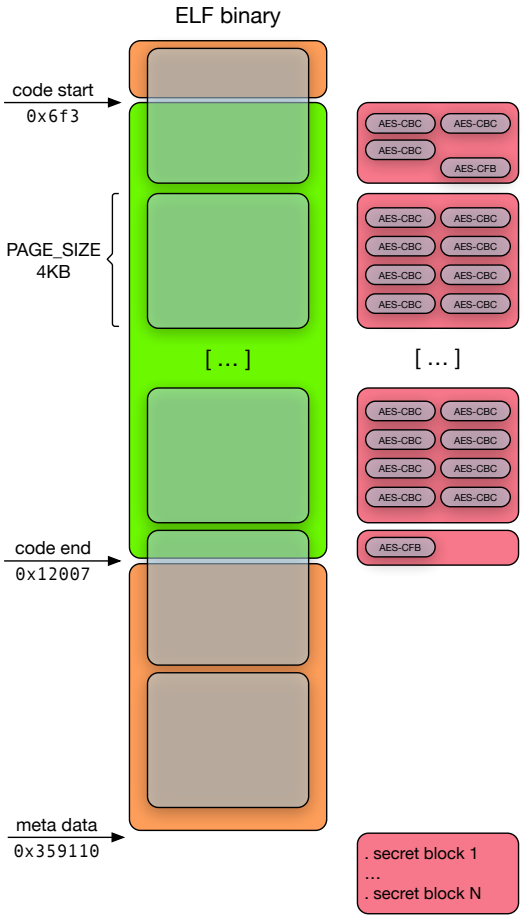

Figure 5. Chiffrement des sections par pages de mémoire.

Linux ne s'attarde que sur les *program headers* de type *PT_LOAD* afin de créer les mappings mémoire nécessaires à l'exécution du programme.

Ce *program header PT_NOTE* que nous transformons en *PT_LOAD* pour nos besoins, nous permet d'indiquer au noyau que l'application devra disposer des méta-informations dans son espace d'adressage de sorte que l'hyperviseur puisse y avoir accès au moment où il déchiffrera les sections du binaire. Sans ce *program header*, nos méta-informations seraient restées dans le fichier de l'application.

4.4 Injection d'un pré-chargeur

Lorsque le noyau Linux démarre une application, il n'alloue pas toute la mémoire dont elle pourrait potentiellement avoir besoin. Le noyau va par exemple mapper dans un premier temps la page de code référencée par le point d'entrée du programme. Si le code saute dans d'autres pages au cours de son exécution, celles-ci seront mappées à la demande.

Ce comportement peut être gênant dans certaines situations pour notre hyperviseur. En effet, si les méta-informations sont suffisamment volumineuses pour occuper plusieurs pages de mémoire, alors le noyau ne chargera au démarrage de l'application que la première page de notre *program header* dans laquelle le point d'entrée du programme se trouve redirigé.

C'est pourquoi nous injectons au début de nos méta-informations, à l'adresse du nouveau point d'entrée, un petit pré-chargeur dont le but est d'effectuer des accès mémoire sur toutes les pages occupées par nos méta-informations. Le noyau est donc contraint de les mapper en mémoire.

Parallèlement à ces accés mémoire, notre pré-chargeur alloue une page de mémoire à une adresse fixe située en dessous des 4Mio, dans laquelle il va recopier l'adresse de chargement du programme ainsi que l'addresse absolue des méta-informations. Le fait de disposer de ces informations à une adresse fixe dans l'espace d'adressage des processus *secret*, en facilite la gestion par l'hyperviseur dans certaines situations où des processus sont considérés *secret* de manière implicite. C'est à dire sans avoir explicitement demandé à l'hyperviseur de les enregistrer comme *secret*. Il s'agit de certains cas liés à l'utilisation de *fork()*, *clone()* ou durant le traitement de signaux. Nous y reviendrons dans la section 5.

Le pré-chargeur effectue finalement un *vmcall*[7] afin de prévenir l'hyperviseur que l'application est prête à être déchiffrée.

4.5 Démarrage d'une application protégée

Comme nous venons de l'expliquer, le point d'entrée du programme est modifié pour que notre pré-chargeur soit appelé avant le point d'entrée originel. Ce dernier est conservé dans les méta-informations.

Quand l'hyperviseur intercepte l'exécution de l'instruction *vmcall*, il considère qu'il doit mettre en place un nouvel environnement EPT *secret* pour l'application dont l'identité sera associée à la valeur du registre *cr3* courant.

7. Instruction spécifique à la virtualisation permettant d'appeler l'hyperviseur.

Lorsque l'application reprend son exécution suite au *vmcall*, le préchargeur s'occupe de la rediriger vers son point d'entrée d'origine. À cet instant, il se peut qu'elle déclenche l'exception *Page Fault (#PF)* pour une page non encore chargée. Le noyau s'occupera de son chargement et l'hyperviseur prendra la main à son tour pour une faute de page dite *Nested Page Fault (#NPF)* au niveau de l'environnement EPT nouvellement créé.

L'hyperviseur décidera alors ou non de déchiffrer et de mapper la page avec les privilèges adéquats.

5 Fonctionnement de l'hyperviseur

Dans sa version de base, Ramooflax dispose d'un gestionnaire d'évènements qui pour chaque *vmexit*[8] fait appel au *stub gdb* afin qu'un client distant puisse interagir avec l'hyperviseur.

Dans notre protection, le gestionnaire d'évènements fait appel à des *hooks* directement implémentés dans le code de l'hyperviseur qui serviront à traiter les évènements essentiels liés à la gestion des applications *secret* :

— écriture dans le registre *cr3* ;
— *nested page fault* (*#NPF*) ;
— exécution de *vmcall* ;
— exceptions liées au mécanisme d'interception.

5.1 Création d'un processus *secret*

Nous avons vu précédemment que lorsqu'une application *secret* démarre, elle effectue un *vmcall* afin de prévenir l'hyperviseur qu'il s'agit d'un programme dont les pages de code sont chiffrées.

Le gestionnaire de *vmcall* de l'hyperviseur va donc créer en interne quelques structures de données lui permettant de se souvenir que ce processus est une application *secret* (valeur de *cr3*, adresse de chargement en mémoire du processus, etc). Il va également lui créer un nouvel environnement EPT vierge et lui attribuer un segment de code spécifique dont nous expliquerons l'utilité dans la sous-section 5.5.

Comme l'hyperviseur sait désormais qu'une application *secret* est en cours d'exécution, il va activer son mécanisme d'interception de changement de niveau de privilèges.

8. Évènement interrompant l'exécution de la machine virtuelle afin d'être traité par l'hyperviseur

5.2 Gestion des *Nested Page Fault*

Les Nested Page Fault (*#NPF*) sont les fautes de page générées par le CPU au niveau des environnements EPT. Seul l'hyperviseur peut les voir car elles ne concernent que des fautes liées aux mécanismes implémentés par l'hyperviseur.

Le cas le plus simple de déclenchement d'un *#NPF* survient au démarrage d'une application *secret* alors que son environnement EPT est encore vierge. L'hyperviseur intercepte un *#NPF* en exécution sur une page non présente dans l'EPT *secret*. Deux solutions s'offrent à lui :

— Soit il s'agit d'une page de code chiffrée et l'hyperviseur va retrouver le *secret_process_block* associé dans les méta-données grâce à l'adresse de la faute. Puis il va marquer la page non présente dans l'environnement EPT *origine*, déchiffrer la page *in-situ* et la marquer *eXecute-only* dans l'environnement EPT *secret*.

— Soit il s'agit d'une page de code non chiffrée (ex. bibliothèque partagée), et l'hyperviseur va simplement recopier l'entrée d'EPT *origine* pour l'affecter à l'EPT *secret*.

L'application poursuit son exécution normalement après la gestion de la faute. Si une application non protégée ou le noyau de l'OS tente d'accéder à une page déchiffrée et donc protégée, l'hyperviseur le détectera grâce au mapping non présent dans *origine*.

Les entrées présentes de l'EPT *origine* sont mappées *USER—RWX*[9] afin de permettre un accès en mémoire physique sans contrainte, comme lorsque la couche de virtualisation n'est pas présente. Ajoutons que notre solution marque les pages de mémoire physique grâce à un tableau de descripteurs afin de savoir à tout instant à qui appartient une page, dans quel état elle se trouve et autres informations susceptibles d'aider l'hyperviseur.

Évidemment, gérer les cas simples n'est pas suffisant pour avoir une solution fonctionnelle. L'hyperviseur doit être en mesure de rechiffrer les pages protégées afin de les rendre à l'OS lorsque cela est nécessaire (swap, processus tué, etc.).

Nous définissons deux primitives utilisées par notre algorithme de gestion des *#NPF* : **sécuriser** et **rendre**.

Le fait de **rendre** un page protégée à la VM implique les opérations suivantes :

— chiffrement *in-situ* si page déchiffrée ;

9. Modulo quelques détails concernant la gestion du cache. Ramooflax étant open-source, le lecteur intéressé pourra y trouver toute la justification nécessaire.

— mapping *USER—RWX* dans EPT *origine* ;
— mapping non présent dans tous les EPT *secret* où elle existait.

Le fait de **sécuriser** une page dans un environnement EPT *secret* implique les opérations suivantes :

— validation du HMAC ;
— si le hash est invalide, on **rend** la page à la VM ;
— sinon marquer le mapping non présent dans l'EPT *origine* ;
— déchiffrement *in-situ* ;
— mapping *eXecute-only* dans l'EPT *secret*.

L'algorithme de gestion des *#NPF* dans l'EPT *secret* s'approche plus précisément du fonctionnement suivant :

— si la page n'est pas présente
  — accès lecture/écriture : on recopie l'entrée EPT *origine*. Si l'entrée n'est pas présente (ex. indiquant une page déchiffrée d'un autre processus *secret*), on **rend** la page à la VM
  — accès en exécution : on recherche les méta-données associées
    — si c'est une page à déchiffrer, on **sécurise** la page
    — sinon :
      — si elle est présente dans l'EPT *origine*, on recopie l'entrée dans l'EPT *secret*
      — si elle ne l'est pas, il s'agit d'une page protégée d'un autre processus *secret*. Auquel cas nous n'empêchons pas son accès. Le processus ne pourra pas la lire, cela ne viole pas la protection. Et il se peut qu'il y ait du partage de code effectué légitimement par une application protégée. Nous créons donc le mapping *eXecute-only* vers cette page de mémoire physique
— si la page est présente
  — il s'agit forcément d'une faute en lecture/écriture et on **rend** la page à la VM

Parallèlement, l'algorithme de gestion des *#NPF* dans l'EPT *origine* ne concerne que des entrées non présentes :

— En ring 0, dans toutes les situations on **rend** la page à la VM :
  — accès en lecture/écriture : la page va être utilisée pour autre chose que l'exécution d'un de nos processus *secret* ;
  — accès en exécution : on laisse l'OS tenter le coup. Cela pourrait arriver dans le cas d'un exploit ring 0 sautant dans du code

ring 3 sans changer de niveau de privilèges. Dans ce cas il va exécuter du code chiffré [10].

— En ring 3, dans un processus non secret :
  — accès en exécution : cas potentiel des *fork()*. Nous détaillerons ce point plus loin ;
  — accès en lecture/écriture : même traitement que pour le ring 0. On considère que l'on perd la page protégée et on la **rend** à la VM.

Il est indispensable de marquer non présente dans l'environnement EPT *origine* les pages déchiffrées pour des raisons évidentes de sécurité. Cela permet aussi d'intercepter de nombreux cas de réutilisation des pages protégées, légitimement ou non par le noyau, et de ne pas perturber le fonctionnement du système.

5.3 Déchiffrement des blocs

Comme expliqué précédemment, le déchiffrement des pages a lieu *in-situ*. Ceci a pour avantage de réduire à son plus strict minimum la consommation mémoire de notre solution et de ne pas contraindre à la fois la VM et l'hyperviseur en nombre d'applications protégées exécutables simultanément. Si l'OS est capable de lancer N applications protégées ou non pour une taille de RAM donnée, notre protection ne doit en aucun cas réduire ce nombre.

Pour procéder au déchiffrement, l'hyperviseur accède aux méta-données du processus, retrouve le bloc correspondant à la page, ainsi que son HMAC et effectue les opérations cryptographiques nécessaires. Si les méta-données ne sont pas accessibles, et d'une manière générale si l'hyperviseur a besoin d'accéder à des données non chiffrées d'une application protégée (ex. pile utilisateur) et que celles-ci ne sont pas disponibles, alors l'hyperviseur est capable de simuler une faute *légitime* de page de l'application afin de forcer l'OS à les lui mapper. Pour cela, il peut injecter une exception *#PF* dans la VM avec *cr2* [11] référençant la zone mémoire à laquelle l'hyperviseur a besoin d'accéder.

5.4 Détection d'un processus *secret*

D'un point de vue strictement matériel, nous identifions un changement de tâche au moment où l'OS modifie la valeur de *cr3*. Comme

10. Au passage, indirectement cela fournie une protection supplémentaire :) c'est cadeau.
11. Registre système contenant l'adresse de la faute.

Ramooflax intercepte les écritures dans ce registre, il sait si l'OS décide de *scheduler* une application *secret* ou non.

Une application *secret* se signale explicitement auprès de l'hyperviseur, grâce à un *vmcall*, afin d'être identifiée comme telle. Nous verrons néanmoins que dans certaines situations l'hyperviseur devra de lui même détecter l'ordonnancement d'une application *secret* ne s'étant pas signalée.

Néanmoins, lorsque Ramooflax détecte que l'OS souhaite ordonnancer un process *secret*, il active son mécanisme d'interception de changement de niveau de privilèges. Il laisse cependant l'environnement EPT *origine* en place pendant l'exécution du code du noyau. Le changement d'environnement EPT se fera uniquement au moment où le CPU passera du ring 0 au ring 3.

Si l'on rentre en ring 3, l'hyperviseur interdira les accès en ring 0 et autorisera la VM à exécuter du code en ring 3. Dans le cas contraire, si un processus *secret* est interrompu l'hyperviseur autorisera l'entrée en ring 0 et interdira les entrées en ring 3.

Finalement, si l'OS n'ordonnance pas un processus *secret*, l'hyperviseur désactive le mécanisme d'interception. L'interception des écritures dans le registre *cr3* n'est en fin de compte qu'une optimisation du mécanisme d'interception, car il permet de le désactiver durant l'exécution de tâches non protégées et ainsi d'éviter des *vmexit* inutiles.

5.5 Mécanisme d'interception

Le mécanisme d'interception de changement de niveau de privilèges repose principalement sur le fonctionnement de la segmentation en mode protégé. Les descripteurs de segment de code ring 0 et ring 3 des OS modernes sont souvent stockés dans la *gdt*. Dans des cas standards d'exécution, ces segments ne sont jamais modifiés ou déplacés. Ils sont configurés au démarrage une fois pour toute.

Lorsque l'hyperviseur crée une application protégée, il va parcourir (une fois seulement) la *gdt* de la VM afin de détecter où sont situés les différents segments de code ring 0 et ring 3 configurés par l'OS.

L'hyperviseur va non seulement se souvenir de leur localisation afin de pouvoir les modifier au moment opportun, mais il va également créer une nouvelle entrée pour un segment de code utilisateur identique à celui déjà présent dans la *gdt*, que nous appellerons *CS3 secret*. Nous reviendrons sur ce point, un peu plus loin.

La mécanique d'interception consiste simplement à modifier le bit de présence des différents segments de code de la *gdt* et à filtrer les

exceptions *Segment Not Present* (*#NP*) qui vont être générées lorsqu'ils seront accédés.

Quand nous souhaitons interdire une entrée en ring 0, pendant l'exécution d'une application *secret*, l'hyperviseur marque :

— CS ring 0 non présent ;
— CS ring 3 non présent ;
— CS ring 3 *secret* présent.

À l'arrivée d'un évènement provoquant une entrée en ring 0, le CPU va générer l'exception *#NP*. L'hyperviseur l'interceptera, basculera de l'environnement EPT *secret* vers *origine* et finalement modifiera les descripteurs de la *gdt* comme suit :

— CS ring 0 présent ;
— CS ring 3 non présent ;
— CS ring 3 *secret* non présent.

Ceci permet au noyau de s'exécuter et de traiter l'évènement [12]. S'il modifie le registre *cr3* et ordonnance un processus lambda, l'hyperviseur désactive le mécanisme d'interception et configure la *gdt* ainsi :

— CS ring 0 présent ;
— CS ring 3 présent ;
— CS ring 3 *secret* non présent.

Lorsque l'hyperviseur traite l'exception *#NP*, il doit distinguer le niveau de privilège auquel elle est survenue. L'entrée en ring 0 ayant été précédemment exposée. Dans le cas d'une entrée en ring 3 vers un processus *secret*, l'hyperviseur va mettre en place son environnement EPT et de nouveau marquer :

— CS ring 0 non présent ;
— CS ring 3 non présent ;
— CS ring 3 *secret* présent.

5.6 Cas particulier des *fast system calls*

Malheureusement, le mécanisme d'interception ne se résume pas uniquement aux entrées de la *gdt*. En effet, l'apparition des *fast system calls* au sein de l'architecture x86 ajoute un peu de complexité à notre mécanique. Nous parlons dans cet article uniquement des instructions *sysenter/sysexit*. Ces dernières s'appuient sur trois *Model Specific Registers (MSR)* :

— *SYSENTER_CS* indiquant l'index dans la *gdt* du CS ring 0 ;
— *SYSENTER_EIP* indiquant l'offset en ring 0 où commencer l'exécution ;
— *SYSENTER_ESP* indiquant le pointeur de pile ring 0.

12. Une interruption, une exception ou un appel système.

Le CPU calcule automatiquement la valeur du segment de pile (*SS*) en fonction de *SYSENTER_CS* ce qui impose un agencement précis de la *gdt*. De la même manière, *sysexit* récupérera les segments ring 3 à des *offsets* relatifs à l'index spécifié dans *SYSENTER_CS*. Les segments de code/données ring 0 et ring 3 doivent être contiguës dans la *gdt*.

Comme *sysenter/sysexit* ont été introduites pour accélérer les appels systèmes, il fallait bien s'attendre à des comportements câblés. Et effectivement, même si *SYSENTER_CS* indique l'index d'un segment de code ring 0, celui-ci n'est pas consulté par le CPU. Il vérifie uniquement qu'il est différent de 0 [13]. La partie cachée [14] des registres de segment CS et SS est initialisée à des valeurs fixes par le CPU durant l'exécution de *sysenter*. Par conséquent les instructions *sysenter/sysexit* sont indifférentes de nos modifications apportées à la *gdt*. Nous sommes donc contraints de mettre à zéro *SYSENTER_CS* lors de l'activation du mécanisme d'interception afin de forcer le CPU à générer l'exception *General Protection Fault* (*#GP*) durant un *sysenter/sysexit* et l'intercepter. Cela justifie également pourquoi nous filtrons les accès au registre *cr3* afin de détecter l'ordonnancement d'un processus *secret* ou non pour désactiver le filtrage des *fast syscalls* et ne pas pénaliser inutilement le système.

Le comportement de l'instruction *sysexit* aura également des conséquences qui seront détaillées dans la sous-section 5.8 consacrée aux *fork()*.

Notons finalement, que l'interception du passage ring 0 vers ring 3 au moment d'un *iret* implique de mapper la page de mémoire physique contenant l'instruction depuis l'EPT *origine* vers *secret*. Au moment où l'on effectue l'interception, nous sommes en ring 0. Suite à cela nous mettons en place l'environnement EPT *secret* pour entrer en ring 3 et permettre l'exécution de l'application protégée. Dès lors, la page de code noyau contenant l'*iret* n'est plus disponible car le processus *secret* ne l'a jamais exécutée. C'est pourquoi il est nécessaire de la rendre disponible dans cet environnement.

5.7 Terminaison d'une application

La terminaison d'une application est un paradigme dépendant du système d'exploitation et notre solution se veut indépendante de tout concept lié à l'OS. Il n'est alors pas évident de détecter la terminaison d'un processus *secret*. Lorsqu'il se termine, nous nous retrouvons devant différents cas de réutilisation de la page de mémoire physique pointée par *cr3* :

13. La première entrée de la *gdt* doit être nulle.
14. Base, limite et attributs.

— soit il se relance avec le même *cr3* et cela est complètement transparent ;
— soit il se relance avec un *cr3* différent, auquel cas on considère qu'il s'agit d'un nouveau processus *secret* ;
— soit un processus lambda réutilise son ancien *cr3*.

Ce dernier cas aurait pour conséquence d'intercepter et faire exécuter un processus non protégé dans un environnement EPT *secret* inutilement. Sans aucun impact sur notre propriété de sécurité cela étant.

Du fait qu'une application *secret* s'exécute avec un segment de code utilisateur différent des applications classiques de l'OS, nous sommes en mesure de distinguer l'exécution d'un processus non *secret* faisant usage d'un *cr3* auparavant utilisé par un processus *secret*. Lorsque Linux démarre une tâche pour la première fois, il lui crée une pile noyau initialisée avec toutes les informations nécessaires à la transition ring 0 vers ring 3 et effectue un *iret*. Dans cette pile, le noyau aura placé un segment de code ring 3 par défaut et non le *CS 3 secret* créé par l'hyperviseur. Ainsi au moment de l'*iret*, comme l'hyperviseur pense qu'il s'agit toujours d'un processus *secret*, il activera l'interception de changement de privilèges et le segment de code ring 3 par défaut ne sera pas accessible. Cela va générer une exception *#NP* en ring 0 en apparence pour un processus *secret* mais avec une référence au segment de code ring 3 par défaut. L'hyperviseur est donc en mesure de distinguer ce cas particulier et d'oublier ce *cr3* comme étant celui d'un processus *secret*.

Ce comportement est à valider sous Windows, il devrait être le même car fortement lié à des considérations matérielles.

5.8 Cas particulier des *fork()*

Uniquement présent dans le monde Unix, l'appel système *fork()* pose quelques problèmes à notre protection si l'on souhaite permettre son utilisation de manière totalement transparente et en particulier conserver l'agnosticisme de notre solution concernant l'OS virtualisé.

D'une manière générale, il s'agit de rendre l'hyperviseur capable d'identifier une application *secret* sans qu'elle le manifeste via un *vmcall*.

Si un processus *secret* P1 *fork()*, le noyau crée un nouveau processus P2 et lui alloue un nouvel espace d'adressage (*cr3*) avec les mêmes mappings que P1. Cependant l'hyperviseur ne sait pas que P2 est *secret*.

Dans le cas le plus simple, lorsque P2 démarre son exécution dans l'EPT *origine*, il va déclencher un *#NPF* en exécution sur une page protégée de P1 car marquée non présente dans *origine*. Ceci correspond à

un cas légitime de *fork()* et dans cette situation l'hyperviseur considérera que ce processus P2 (via son *cr3*) est un processus *secret* en lui créant son propre EPT *secret* qui se remplira dynamiquement.

Cependant des cas plus complexes peuvent survenir. Notamment, lorsque le processus P2 au retour du *fork()* démarre son exécution dans une page de P1 qui n'a pas encore été déchiffrée et qui ne le sera peut-être jamais. Dans cette situation, P2 crashera car il exécutera des instructions situées dans une page chiffrée de l'EPT *origine*[15].

Soit une des pages de P1 n'a jamais été déchiffrée, soit elle a été mise en *swap* par l'OS. Imaginons que P1 *fork()*, le noyau crée P2 mais décide de scheduler un processus P3 de très haute priorité. Ce P3 pourrait être très gourmand en ressources et forcer le noyau à swapper P1 (et donc P2) auquel cas l'hyperviseur rendrait les pages protégées à l'OS. Suite à cela, le noyau décide de scheduler P2 qui n'est pas *secret* aux yeux de l'hyperviseur. L'environnement EPT *origine* ayant considérablement changé depuis le *fork()* de P1, plus aucune page déchiffrée n'est disponible. Le processus P2 crashera.

Heureusement, comme pour la terminaison d'un processus, le fait d'utiliser un segment de code particulier pour les processus *secret* nous permet d'exécuter P2. En effet, à sa création P2 possède le même environnement mémoire que P1, et même s'il possède sa propre pile noyau son contenu est le même que celle de P1 car il doit reprendre son exécution au même endroit. Si P1 a effectué un *fork()* via un *int 0x80*[16] alors sa pile noyau contient le segment de code *secret* qu'il utilisait pendant son exécution. Ainsi, lorsque P2 commencera son exécution il utilisera ce segment de code particulier. La protection étant désactivée vu qu'il ne s'agit pas d'un processus *secret*, le segment de code *secret* est donc marqué non présent. Ceci va générer une exception *#NP* et permettre à l'hyperviseur de détecter ce cas de *fork()*.

Toutefois, cette astuce ne fonctionne pas avec *sysenter/sysexit* car elles n'utilisent pas les descripteurs de la *gdt* comme nous l'avons vu précédemment. Dans ce cas, notre solution consiste à créer un trampoline dans la pile utilisateur de P1 durant le *sysenter* de manière à forcer au moment du *sysexit* un retour vers une instruction *vmcall* présente dans les

15. Il peut s'agir de l'instruction suivant un *int 0x80* dans un binaire statique, du point d'entrée de *sysexit* dans un binaire dynamique ou d'un *else if(pid == 0)* situé dans une page non encore déchiffrée.

16. Ou un équivalent de saut inter-segment avec changement de niveau de privilèges responsable de l'empilement en ring 0 de SS3/ESP3/CS3/EIP3/EFLAGS.

méta-données. Comme P2 a la même pile utilisateur que P1, il exécutera le même chemin et passera par le *vmcall* avertissant l'hyperviseur.

Notons que cela induit un petit *overhead* (quelques instructions) acceptable du fait que l'interception des *fast syscalls* est essentielle à notre protection afin de permettre une exécution normale du système. Nous pourrions envisager de désactiver les *fast syscalls* pour les processus *secret* afin de nous faciliter la tâche. Le gain de performances de ces instructions étant naturellement perdu du fait de leur interception par l'hyperviseur.

La figure 6 propose une vue d'ensemble de la mémoire suite à un *fork()* de P1.

5.9 Cas particulier de *clone()* et des gestionnaires de signaux

L'appel système *clone()* est utilisé pour la création de *threads* sous Linux. Chaque *thread* dispose de sa propre pile utilisateur mais également de sa propre pile noyau. Par contre tous les *threads* d'un processus utilisent le même espace d'adressage référencé par le registre *cr3*.

Le fait que chaque *thread* dispose de sa propre pile noyau est problématique pour notre protection, du moins durant leur démarrage. En effet, pour préparer l'exécution d'un nouveau *thread*, le noyau va initialiser dans leur pile ring 0 un contexte utilisateur [17] avec le segment de code ring 3 par défaut de la *gdt*.

Imaginons que nous lancions un processus *secret* qui effectue des *clone()*. À la création du processus, l'hyperviseur prendra bien note du *cr3* de ce processus afin de l'identifier comme *secret*. Lorsqu'il va commencer à effectuer des *clone()*, chaque *thread* créé sera attaché à son espace d'adressage identifié comme *secret*. Toutefois, lorsqu'un *thread* va commencer à s'exécuter il le fera avec un segment de code ring 3 n'étant pas celui utilisé par les processus *secret*. L'hyperviseur considérera qu'il s'agit d'un cas de réutilisation de *cr3* et oubliera le processus qui finira par crasher.

La gestion des signaux sous Linux induit également une modification de la pile ring 0 du processus par le noyau. Lorsqu'un processus reçoit un signal pour lequel il dispose d'un gestionnaire, le noyau interrompt l'exécution du programme, modifie sa pile noyau pour appeler le gestionnaire de signal durant la transition ring 0 vers ring 3. Il modifie également la pile utilisateur afin de forcer un *sigreturn()* et permettre au programme de reprendre son exécution là ou celle-ci avait été interrompue.

Ces deux cas complexes nécessitent donc d'autres éléments d'identification implicite de processus *secret*. Ces indicateurs, que nous appelons

17. SS3/ESP3/CS3/EIP3/EFLAGS

hardware cookies se doivent d'être uniquement dépendant du matériel pour respecter notre philosophie, comme c'est déjà le cas avec la modification de la *gdt*.

De nos jours, la plupart des OS modernes interdisent aux applications de mapper de la mémoire à l'adresse 0 afin d'éviter les déréférencements de pointeurs *NULL*. Ceci a pour conséquence que la première entrée de la première table de pages est toujours vide ou du moins marquée non présente. Notre astuce consiste à profiter de cette entrée et y installer un *hardware cookie* qui permettra à l'hyperviseur de discerner des faux-positifs.

Nous avions mentionné dans la sous-section 4.4 que notre pré-chargeur allouait une page de mémoire à une adresse fixe située sous les 4Mio, dans laquelle il recopiait les adresses de chargement et des méta-informations du processus. Une application a le droit d'allouer de la mémoire entre 4Kio et 4Mio [18]. Le fait de forcer notre pré-chargeur à allouer une page de mémoire dans cette zone, nous garantit que le noyau aura préalablement alloué une table de pages dans la première entrée du répertoire de pages. Une fois la table allouée, l'hyperviseur pourra durant le *vmcall* du processus *secret* y installer son *hardware cookie* pour de futurs identifications.

Cette page de mémoire dispose d'un double atout. Non seulement elle permet d'installer un nouvel indicateur matériel, mais son contenu à une adresse fixe permet aussi à l'hyperviseur de retrouver les méta-informations pour des processus *secret* ne s'étant pas déclarés explicitement via un *vmcall* (cas des *fork()*).

Par ailleurs, notre *hardware cookie* situé dans la table de pages est également recopié par le noyau (*copy-on-write*) dans le nouvel espace d'adressage du processus fils créé suite au *fork()*.

6 Confronter la protection

Du point de vue de la confidentialité, la propriété de sécurité de notre solution repose uniquement sur la configuration en mode *eXecute-only* des entrées des environnements EPT *secret* et non présentes dans l'environnement EPT *origine*. Elle seule garantie que le code protégé ne sera pas lisible, ni par un noyau malveillant, ni par aucune application du système, y compris les applications *secret*.

Tous les autres mécanismes (*hardware cookies*) de l'hyperviseur : interception de changement de niveau de privilèges, gestion des cas particuliers

18. La limite basse peut être différente selon les OS. Notre pré-chargeur alloue la page à l'adresse 2Mio.

tels que *fork(), clone()* ou *sysexit*, ont été conçus et développés uniquement pour assurer le bon fonctionnement des applications protégées au sein de leur système d'exploitation.

Notre protection n'assure pas la disponibilité des applications protégées. Si l'OS malveillant décide de tuer ou altérer l'exécution d'une application *secret* l'hyperviseur ne saura l'en empêcher. Aussi, nous considérons qu'il n'est pas nécessaire à l'hyperviseur de s'assurer que tous ces mécanismes ne soient pas détournés. Dans le pire des cas, un attaquant sera capable de réaliser un déni de service. Qu'il modifie la *gdt* constamment, qu'il modifie la pile d'un processus, qu'il réutilise un *cr3* en dehors de tout fonctionnement *classique* de l'OS, cela aboutira à un plantage de l'application voir du système.

6.1 Limitations actuelles

Dans sa version actuelle, la protection ne fait pas usage de l'I/O MMU. Elle s'expose donc aux attaques AMT (ring -3), DMA et autres exécutions de code pouvant accéder à la RAM en dehors du contrôle du CPU. Cependant il suffirait de lier la configuration de l'EPT *origine* à celle de l'I/O MMU pour s'en protéger.

Le mode SMM (ring -2) reste également plus privilégié que VMX-ROOT (ring -1) utilisé par la virtualisation. À moins de disposer d'un BIOS open-source ou de réécrire notre firmware UEFI, nous sommes obligés de lui faire confiance.

Les données de nos applications protégées ne sont pas chiffrées. Cela pourrait être en pratique résolu applicativement sans l'intervention de l'hyperviseur.

Les cas de *self modifying code*, *jit compiler* ou programmes lisant leur propre code sont également problématiques pour notre protection.

Les processeurs AMD ne supportent pas le mode *eXecute-only* pour leur équivalent d'EPT. Il faudrait avoir recours à des astuces dignes de celles que l'on trouve dans PaX (SEGMEXEC [6]).

6.2 Scénarios d'attaques

En dehors des *side-channels*, ou effets de bord de gestion des caches non pris en compte dans notre protection, il reste peu de scénarios crédibles d'attaques distantes.

Nous pouvons cependant retenir celui de la prédiction d'instructions par analyse des effet produits sur les registres généraux (GPRs). Théoriquement, un attaquant pourrait avoir un nombre d'essais infini,

contrôler totalement le noyau et tenter d'échantillonner instruction par instruction l'exécution d'une application protégée en ring 3. Nous avons cependant imaginé des solutions simples permettant de considérablement complexifier une telle attaque voir de l'en empêcher.

Dans un premier temps, nous pourrions empêcher de transmettre à la VM les exceptions *#DB* et *#BP*[19]. L'attaquant ne pourrait plus *simplement* exécuter pas à pas l'application protégée. Il pourrait cependant configurer un *timer* d'horloge si finement qu'il ne permette l'exécution que d'une seule instruction en ring 3 avant de reprendre la main en ring 0. Une sorte de *single-step* du pauvre qui lui permettrait d'essayer de deviner l'instruction courante.

Pour s'en protéger l'idée de base serait évidemment d'effacer les GPRs d'un processus *secret* lorsque l'on quitte le ring 3. Bien entendu, leur contenu serait préalablement sauvegardé puis restauré par l'hyperviseur.

S'il s'agit d'une transition ring 3 vers ring 0 involontaire, suite à une exception ou une interruption matérielle, alors nous pouvons les effacer. Il conviendrait d'analyser également la pile noyau ayant sauvée EIP et d'aligner sa valeur sur la page contenant l'instruction ring 3 interrompue afin de transmettre le moins d'information possible.

Si par contre, il s'agit d'une transition ring 3 vers ring 0 volontaire, autrement dit un appel système, alors nous ne pouvons effacer les GPRs car ils contiennent les arguments de l'appel. La question qu'il convient de se poser est donc la suivante : est-il possible de deviner les instructions d'un programme en fixant des valeurs contrôlées dans les GPRs et en observant leur modification uniquement au moment d'un appel système ?

Imaginons que le noyau *backdoor* la libC et que potentiellement, tout appel de fonction externe à notre programme *secret* se transforme en appel système et donc permette au noyau de lire les GPRs. Un noyau pourrait au retour de l'appel système, fixer la pile ring 0 pour faire pointer EIP vers les instructions précédent l'appel de fonction externe, une par une. Nous ne pouvons pas sauver la frame ring 0 de retour d'appel système dans l'hyperviseur et la restaurer au moment où l'hyperviseur fait l'*iret* pour empêcher le kernel de retourner à un endroit autre que l'instruction légitime qui suit l'appel système, car il existe entre autres le cas des trampolines de gestion des signaux où le noyau interrompt l'exécution du programme pour qu'il exécute son gestionnaire de signal.

Cela étant, si le noyau redirige le programme où il veut après l'appel système et en fixant un état E1 connu des GPRs et en ayant découvert une suite d'instructions In menant à son *hook* lui permettant de lire un

19. Utilisées pour le debugging matériel.

état E2 des registres, est-il possible de deviner la première instruction exécutée juste après avoir défini E1 ?

```
on prépare E1 et on positionne EIP sur I1

insn I1 inconnue
|
| N insn connues/devinées
|
int 0x80

on récupère E2
```

Est-il possible de deviner I1 en connaissant E1, les In et E2 ? Il semble que les In peuvent perdre suffisamment d'information pour ne pas être en mesure de pouvoir déduire I1. Si par exemple *I1 = mov edx, 3* et que dans les In se trouve un *xor edx, edx* alors au moment d'analyser E2 on ne pourra pas deviner I1.

7 Autres protections

Quelques projets de recherche adressent la problématique de la protection logicielle via une solution de virtualisation. La plupart cherche à protéger l'intégrité du code du noyau de l'OS et non ses applications. C'est le cas de SecVisor [1] ou HvmHarvard [5]. Deux projets cependant se rapprochent plus de notre solution et proposent de protéger des applications contre un noyau malveillant.

HARES [4] est une preuve de concept s'appuyant sur l'hyperviseur MoRE [3] et l'implémentation d'AES-NI [20] de TRESOR [8] afin de protéger le code d'applications en le chiffrant dans l'exécutable. Il supporte Windows 7 et fonctionne en coopération avec un driver dans le noyau aidant l'hyperviseur à identifier un processus protégé et gérer sa mémoire. Leur concept semble un peu plus lourd en ce sens qu'à chaque écriture dans *cr3*, l'hyperviseur parcourt les tables de pages de l'application afin de détecter des cas de Copy-On-Write. Il ne supporte pas la mise en swap des pages de l'application et requière donc qu'elles soient allouées dans la zone *non-paged-pool*.

OverShadow [11] est une protection basée sur un hyperviseur permettant de protéger des applications (code, données et registres), sans aucune modification du système ou des applications. Le principe de la protection repose sur du *multi-shadowing* offrant plusieurs vues de la mémoire physique selon le contexte d'exécution, l'une chiffrée l'autre claire grâce à

20. Jeu d'instructions des CPUs Intel proposant une implémentation matérielle d'AES.

un protocole de *cloaking*. L'hyperviseur chiffre à la volée les pages claires avant de les rendre visibles à l'OS et permettre leur manipulation. La solution dispose de traitements spécifiques pour certains appels systèmes la rendant dépendante de l'OS. Il s'agit de la solution la plus complète à nos yeux en dehors d'Intel SGX.

En terme de protections matérielles disponibles pour architecture IA-32, Intel Software Guard Extensions (SGX [2,10]) est la dernière avancée proposée par Intel. Un article entier pourrait y être consacré tant la solution semble complète et complexe. SGX propose une protection équivalente à notre solution (confidentialité/intégrité), mais gérée matériellement et disposant de plus d'un système d'attestation distant. Le concept de base est de permettre la création d'enclaves servant à exécuter des applications de façon isolée. Ces enclaves sont initialisées chiffrées par l'OS et déchiffrées par le CPU après chargement. La protection semble très bien pensée. Malheureusement peu de détails sont donnés sur les algorithmes de chiffrement et leur implémentation dans le CPU [21]. De plus, le chargement des enclaves est conditionné par une *Launch Enclave* contrôlée et distribuée par Intel. Il est ainsi nécessaire de leur demander l'**autorisation** d'utiliser leur protection pour nos propres enclaves. *No comment* !

8 Conclusion

8.1 Disclaimer

Notre protection est encore en phase active de développement. Aussi nous souhaitons informer le lecteur que des modifications dans l'implémentation sont susceptibles d'être apportées d'ici à la tenue de la conférence, durant laquelle la version la plus récente de la protection sera présentée.

8.2 Évolutions

Le support du format d'exécutable PE et des tests sous Windows seraient une priorité. Il conviendrait de supporter également le format ELF64 ainsi que la sémantique des instructions *syscall/sysret* pour protéger des applications 64 bits.

Il serait par ailleurs souhaitable de concevoir une chaîne d'outils un peu plus collaborative avec les applications à protéger, proposant de ne chiffrer que certaines sections. D'une part pour des raisons de performances (ex. ne chiffrer que des zones qui n'effectuent pas d'appels systèmes). D'autre

21. Quid d'une *backdoor* dans le CPU ?

part car dans certains cas, il se peut que la dernière page des sections de code chiffrées contienne également le début d'une section de données contiguë (ex. *.rodata*). Dans ce cas, l'application ne pourrait fonctionner correctement car la page contenant à la fois du code et des données serait mappée *eXecute-only*. Nous pourrions fournir un script d'édition de lien utilisable par les éditeurs d'applications à protéger, garantissant que les dernières instructions de l'exécutable soient dans une page différente des premières données *read-only*.

L'utilisation des instructions Intel AES-NI pourrait également être envisagée afin d'accélérer les procédures de chiffrement/déchiffrement de l'hyperviseur.

Enfin, appliquer la protection à des bibliothèques partagées pourrait être une évolution naturelle de notre solution et représenterait peu d'efforts de développement.

Références

1. A. Seshadri, M. Luk, N. Qu, A. Perrig. SecVisor : A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. http://www.sosp2007.org/papers/sosp079-seshadri.pdf, 2007.
2. Intel Developer Zone. Intel SGX Resources. https://software.intel.com/en-us/sgx/resource-library, 2015.
3. J. Torrey. More : Measurement of running executables. https://www.blackhat.com/docs/us-14/materials/us-14-Torrey-MoRE-Shadow-\discretionary{-}{}{}Walker-The-Progression-Of-TLB-Splitting-On-x86-WP.pdf, 2014.
4. Jacob Torrey. HARES : Hardened Anti-Reverse Engineering System. http://jacobtorrey.com/HARES-WP.pdf, 2015.
5. M. Grace , Z. Wang , D. Srinivasan. Transparent Protection of Commodity OS Kernels Using Hardware Virtualization. http://www.comp.nus.edu.sg/~liangzk/papers/securecomm10.pdf, 2010.
6. PaX Team. SEGMEXEC non-executable page feature. https://pax.grsecurity.net/docs/segmexec.txt, 2004.
7. S. Duverger. Virtualisation d'un poste physique depuis le boot. https://www.sstic.org/2011/presentation/virtualisation_dun_poste_physique_depuis_le_boot/, 2016.
8. T. Müller. Tresor runs encryption securely outside ram. https://www.usenix.org/legacy/event/sec11/tech/full_papers/Muller.pdf, 2011.
9. The Santa Cruz Operation, Inc. System V Application Binary Interface, Chapter 5, Note Section. http://www.sco.com/developers/gabi/latest/ch5.pheader.html#note_section, 2013.
10. Victor Costan, Srinivas Devadas. Intel SGX Explained. https://eprint.iacr.org/2016/086.pdf, 2016.

11. X. Chen, T. Garfinkel. Overshadow : A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. http://drkp.net/drkp/papers/overshadow-asplos08.pdf, 2008.

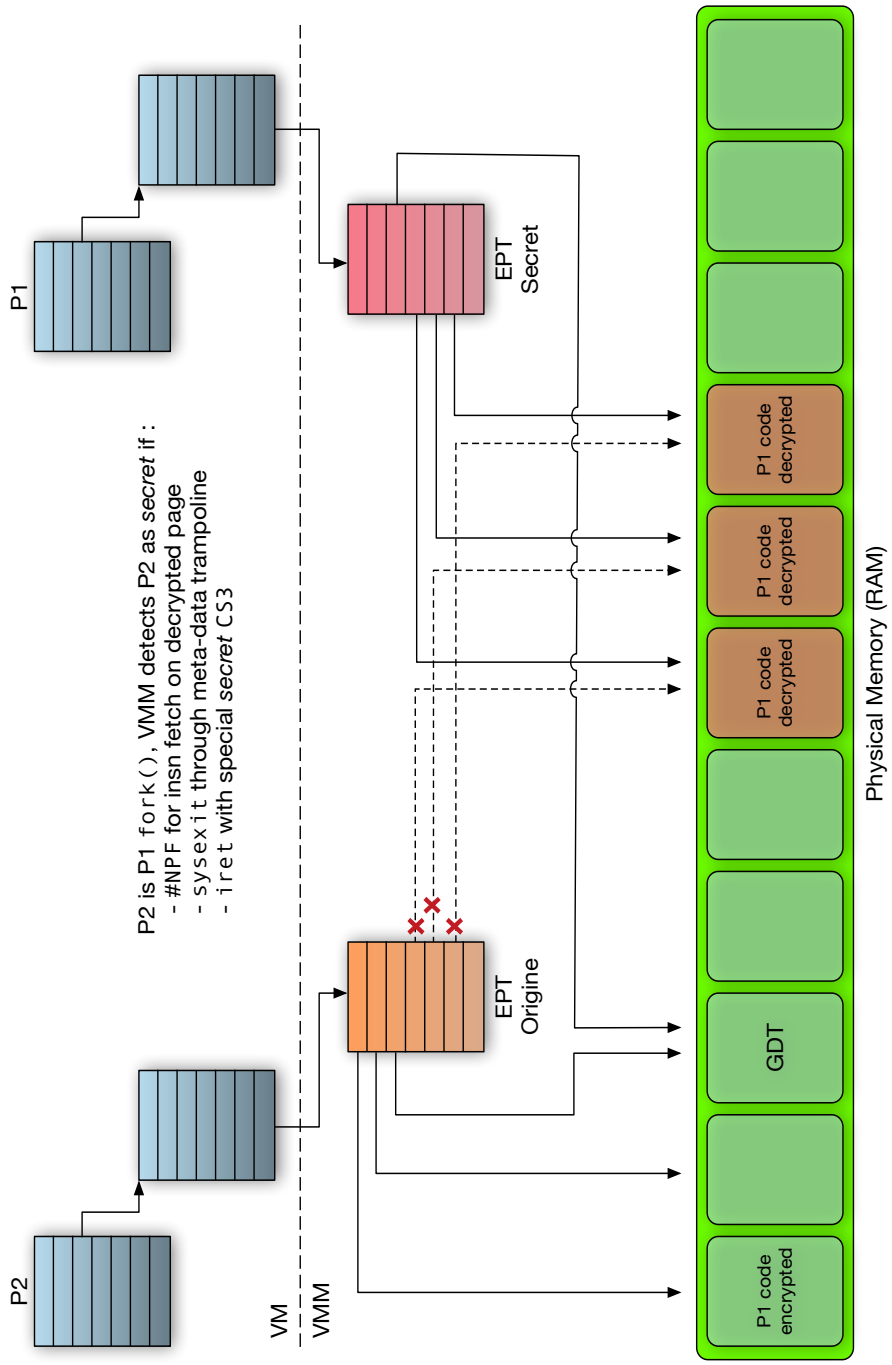

Figure 6. Détection d'un processus *secret* suite à un *fork()*.