

---

---

# ROOTKIT ET NOYAU LINUX 2.6.28

Etat de l'art et détournement d'appels système

---

---

MARTIN Benjamin

PAULIAT Romain

PELLETIER Alexandre

*"The greatest trick the devil ever pulled off was convincing the world he didn't exist."*

Keyser Söze, *Usual Suspects*

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Rappels</b>	<b>4</b>
2.1	Généralités . . . . .	4
2.1.1	Notion de Ring . . . . .	4
2.1.2	Notion d'appel système . . . . .	4
2.1.3	Les registres de debuggage . . . . .	5
2.2	Rootkits . . . . .	6
2.2.1	Cycle de vie d'un rootkit . . . . .	7
2.2.2	Historique . . . . .	7
2.2.3	Exemples . . . . .	8
<b>3</b>	<b>Moyens de corruption d'un système</b>	<b>9</b>
3.1	Ring 3 . . . . .	9
3.2	Ring 0 . . . . .	10
3.2.1	Exploration de la mémoire noyau . . . . .	10
3.2.2	Corrompre la table des appels systèmes . . . . .	11
3.2.3	Subtil inline Hooking . . . . .	11
3.2.4	Debug Register . . . . .	13
3.3	Ring -1 . . . . .	14
3.3.1	Systèmes virtualisés . . . . .	14
3.3.2	Blue Pill . . . . .	15
3.3.3	SubVirt . . . . .	16
<b>4</b>	<b>Elaboration d'un rootkit</b>	<b>17</b>
4.1	Généralité . . . . .	17
4.2	Conception . . . . .	17
4.3	Implémentations . . . . .	18
4.3.1	Recherche d'informations générales . . . . .	18
4.3.2	Recherche complémentaire pretty hook . . . . .	20
4.3.3	Mise en place dirty hook . . . . .	20
4.3.4	Mise en place pretty hook . . . . .	21
<b>5</b>	<b>Conclusion</b>	<b>23</b>

# Chapitre 1

## Introduction

Année 2005. Sony-BMG met sur le marché des millions de CDs contenant la mesure technique de protection contre la copie, nommée "XCP", ce programme a principalement pour fonction d'empêcher le CD d'être lu avec un autre logiciel que celui fourni par Sony et d'empêcher le CD d'être copié en plus de trois exemplaires ou converti au format « mp3 ».

Cependant, une fois inséré dans le lecteur, le CD procédait automatiquement à l'installation, à l'insu de l'utilisateur, d'un programme espion indétectable et non désinstallable. Celui-ci était destiné à recueillir les identifiants des CD écoutés et à les communiquer à Sony-BMG par le biais de son site internet. Après avoir pendant quelques jours nié l'évidence, la réaction du PDG de Sony-BMG, Thomas Hesse, a été d'expliquer que « la plupart des gens ne savent pas ce qu'est un "rootkit", alors pourquoi devraient-ils s'en préoccuper ? ».

"On nomme rootkit un programme ou ensemble de programmes permettant à un tiers (un pirate informatique, par exemple, mais pas nécessairement) de maintenir - dans le temps - un accès frauduleux à un système informatique. La fonction principale du « rootkit » est de camoufler la mise en place d'une ou plusieurs « portes dérobées ». Ces portes dérobées (utilisables en local ou à distance) permettent au pirate de s'introduire à nouveau au cœur de la machine sans pour autant exploiter une nouvelle fois la faille avec laquelle il a pu obtenir l'accès frauduleux initial." <sup>1</sup>

Ils ont été conçus à la base pour des plateformes Linux mais existent maintenant aussi bien sous Solaris, \*BSD ou Windows et des utilisations industrielles sont donc déjà connues. Difficile de déterminer la date de création du premier d'entre eux mais des outils de détection existent depuis 1997. Ils ont donc eu le loisir de s'améliorer avec le temps, de se multiplier, prenant diverses formes et s'attaquant à des endroits de plus en plus sensibles pour le système. Nous allons donc prendre le temps d'étudier plus en détails l'apparition de ces rootkits, de leur complexification au cours du temps ainsi que les moyens mis en place pour arriver à corrompre une machine.

Avec la mise à disposition du noyau linux 2.6.28, fin décembre 2008, l'intérêt pour les rootkits a été relancé. En effet, cette nouvelle version du noyau, utilisant de nouvelles structures ou champs associés, rend impossible l'utilisation directe de rootkits comme EnyeLKM, DR rootkit ou autres moins connus. Nous allons essayer d'effectuer les modifications nécessaires afin de pouvoir mettre en place un rootkit s'apparentant à ces derniers et pour ce type de noyau.

Les différentes notions évoquées au cours des chapitres suivants nécessitent, si ce n'est une connaissance approfondie, tout du moins une certaine aisance avec le fonctionnement du système d'exploitation utilisé. L'intérêt du chapitre qui suit sera d'évoquer les aspects nous concernant avant de plonger au cœur du sujet.

---

<sup>1</sup>Source : Wikipedia

# Chapitre 2

## Rappels

### 2.1 Généralités

Afin de comprendre comment intervenir sur les mécanismes des appels systèmes pour les détourner, il est nécessaire de comprendre un certain nombre de concepts au niveau du système d'exploitation. Nous allons étudier ces concepts dans cette section.

#### 2.1.1 Notion de Ring

Pour leur fonctionnement, les logiciels applicatifs ont besoin d'utiliser les fonctions du système d'exploitation qui dépendent en partie de l'architecture des processeurs. Cet ensemble complexe se situe à différents échelons dont chacun correspond à un mode de fonctionnement et un « anneau de protection » ou « ring ».

Les processeurs les plus courants à ce jour, de type x86, permettent quatre modes dont deux seulement sont utilisés par les systèmes Linux et Windows.

- Les programmes des utilisateurs et les outils communs (processus) fonctionnent en « mode utilisateur » (user mode), au niveau « ring3 », dans un espace mémoire particulier qui est géré par le système d'exploitation.
- Les fonctions fondamentales du système (gestion du processeur, de la mémoire, des entrées-sorties, des communications ...), s'exécutent en « mode superviseur » (supervisor mode) aussi appelé « mode noyau » (« kernel » mode), au niveau « ring0 », un échelon privilégié où toutes les ressources matérielles et logicielles sont accessibles.
- Les autres modes ou niveaux de protection ne sont pas utilisés.

Les processeurs les plus récents permettent la « virtualisation » qui correspond à un nouveau « mode hyperviseur » (hypervisor mode) au niveau 'ring -1'<sup>1</sup>. Le plus haut degré de privilèges est accordé à ce niveau où il est possible d'exercer un contrôle complet tant des applications que du(des) système(s) d'exploitation qui fonctionne(nt) sur le PC.

#### 2.1.2 Notion d'appel système

Un appel système est une routine exécutée par le noyau. Chaque application peut demander un travail à réaliser au système d'exploitation. Il peut s'agir d'une lecture en mémoire, ou d'accès à un périphérique ou autre. Sauf exception prévue par les développeurs pour des liens privilégiés entre deux logiciels, il n'y a pas de relation directe entre leurs processus ou entre un processus et les ressources du système.

C'est par l'intermédiaire d'appels systèmes ('system calls') que les logiciels communs peuvent solliciter les fonctions du noyau. Les relations entre les différents éléments se font grâce aux interfaces de programmation ('API') et à une série de tables en mémoire qui permettent de situer chaque composant

---

<sup>1</sup>Certains considèrent que l'anneau de protection de niveau le plus privilégié devrait être le 'ring0' qui dans ce cas de processeur, deviendrait celui du mode hyperviseur. Celui du noyau serait alors le 'ring1' et celui des logiciels applicatifs le 'ring4'.

ou de lister les opérations à effectuer.

On peut voir tous les appels systèmes demandés par une application `myApp` à l'aide de la commande :

```
strace ./myApp
```

L'application informe le noyau de sa demande d'appel système en invoquant la commande assembleur `int 80h` qui correspond à l'exécution de l'interruption logicielle numéro 80.

Pour s'interfacer avec le noyau, l'application utilise les registres du processeur. Elle stocke le numéro de l'appel système demandé dans le registre `eax`, les paramètres dans d'autres registres. Chaque appel système est caractérisé par son numéro d'appel. Les adresses de chaque appel sont stockés dans la table des appels système (`syscall_table` ou SCT). Cette table est indexée par chaque numero d'appel.

Le noyau dispose de l'adresse de l' Interrupts Description Table (IDT) dans ses registres, c'est-à-dire constamment. Il suffit qu'il exécute la routine localisée à l'adresse écrite dans la 80ème case du tableau IDT, `system_call`. Lors du déroulement de cette routine, le noyau va appelé le bon appel système demandé, à l'aide du registre `eax`. L'ensemble des appels systèmes sont stockés dans la system call table (SCT). De la même façon que les interruptions, `system_call` va ensuite appelé la Nième case du tableau SCT, dont le noyau connaît l'adresse ou l'offset.

La figure 2.1 décrit le déroulement d'un appel système standard. Un détournement d'appel système consiste à repérer certaines adresses dans ce cheminement (étape 1) et à réécrire certaines instructions directement en mémoire afin de modifier le cheminement du flux d'exécution (étape 2).

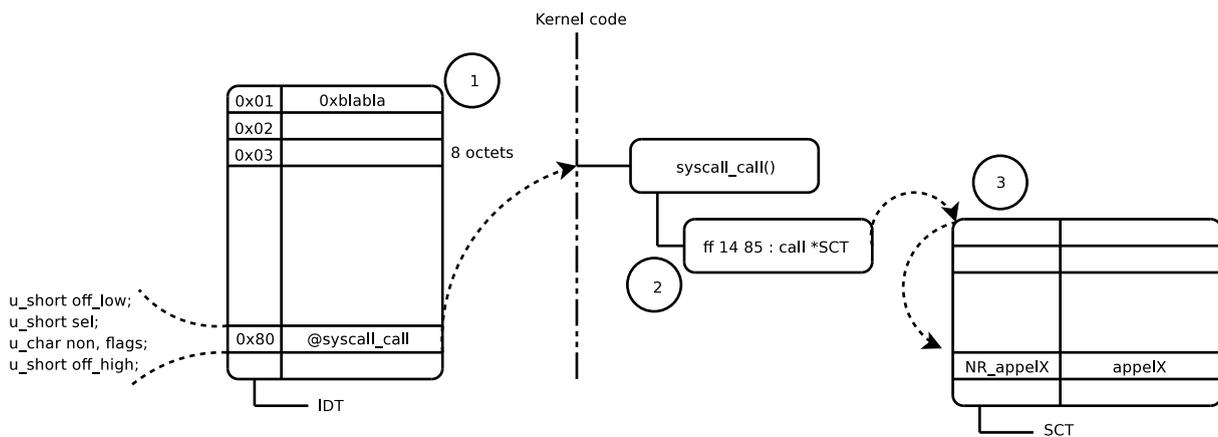


FIG. 2.1 – Déroulement d'un appel système apres `int80h`

Le problème qui se pose à présent est de pouvoir lire la mémoire du noyau. Selon les versions du noyau linux, divers stratagèmes sont à mettre en oeuvre, selon les protections de la mémoire noyau.

### 2.1.3 Les registres de debugage

Sur l'architecture x86, les debug registers sont aux nombres de 8, numéroté de DR0 à DR7. Ils ne sont accessibles qu'à partir du ring 0 et permettent au processeur de gérer le mécanisme de debugage. On les utilise en se servant de l'instruction MOV.

L'utilité de ces registres est de poser 4 breakpoints (notés 0 à 3) sur des adresses linéaires et de piloter ces derniers à l'aide des registres spéciaux DR6 et DR7. Dans chaque breakpoint, on peut stocker les informations suivantes :

- L'adresse à laquelle le breakpoint est placé
- La longueur de l'emplacement du breakpoint. (1, 2 ou 4 octets)
- L'opération qui sera effectuée à l'adresse par l'exception générée
- Si le breakpoint est actif

- Si la condition du breakpoint était présente lorsque l'exception a été générée.

Les registres DR0 à DR3 contiennent chacun une adresse 32 bit du breakpoint qui lui est associé. DR4 et DR5 étant en quelque sorte des alias de DR6 et DR7, c'est pourquoi dans les deux parties suivantes nous ne parlerons que de ces deux registres.

### Le registre DR6

On l'utilise pour déterminer quelle condition de debug est apparue. On peut également noter que le processeur ne vide jamais ce registre. Il contient les informations suivantes :

- **B0 à B3** : Ce sont les bits 0 à 3, ils servent à signaler que la condition de breakpoint a été détectée. Ces flags sont actifs si les flags **LENn** et **R/Wn** de DR7 le sont. Et ils le restent même si on n'allume pas les flags **Gn** et **Ln** de DR7.
- **BD** : C'est le bit 13, il indique si la prochaine instruction dans le flot d'exécution veut accéder aux debug registers. Il est activé si le flag **GD** de DR7 est mis.
- **BS** : C'est le bit 14, il nous dit, si on l'active, que l'exception de debug a été attrapée par le mode pas à pas.
- **BT** : C'est le bit 15, s'il est actif et que le flag debug trap de TSS de la tâche est mis, il permet de révéler que l'exception est le fait d'un changement de tâche.

### le registre DR7

Il sert à piloter les breakpoints : à les activer ou non et à poser leurs conditions. Les flags de DR7 s'utilisent de la manière suivante :

- **L0 .. L3** : Ce sont les bits 0,2,4,6, ils activent les breakpoints de façon local. Quand on allume un de ces bits, la condition posé pour ce breakpoint concerne seulement la tâche courante. Le processeur vide automatiquement ces flags, pour qu'ils n'empiètent pas sur la tâche suivante.
- **G0 .. G3** : Ce sont les bits 1,3,5,7, eux activent les breakpoints de façon global. Quand ils sont à 1, la condition de breakpoint reste quelque soit la tâche. Si on voit une condition de breakpoint, et que son drapeaux est levé, on génère une exception de debug. Le processeur ne vide jamais ces flags, ils peuvent donc être utilisé pour toute les tâches.
- **LE et GE** : Ce sont les bits 8 et 9, ils gèrent l'activation exact local ou global des breakpoints. Ils repèrent quelle est l'instruction exacte qui a causé la condition du breakpoint.
- **GD** : C'est le bit 13, il s'occupe de la détection générale. Il lance une exception de debug des qu'une instructuin MOC tente d'accéder à un debug register. Quand ceci se produit, le flag **BD** de DR6 est activé avant de généré l'exception.
- **R/W0 .. R/W3** : Ce sont les bits 16 et 17 (DR0), 20 et 21 (DR1), 24 et 25 (DR2), 28 et 29 (DR3), ils servent à commander la condition de breakpoint pour un breakpoint donné. On les utilise ainsi : si les deux bits sont à zéro, on arrête sur l'exécution, s'ils valent 01 la condition est que l'on essaie d'écrire des donnée et enfin ils sont égaux à 11, si on essaie de lire ou écrire sur une certaine donnée.
- **LEN0 .. LEN3** : Ce sont les bits 18 et 19 (DR0), 22 et 23 (DR1), 26 et 27 (DR2), 30 et 31 (DR3), ils indiquent quelle longueur de mémoire est surveillée par les breakpoints. On associe à chacun des breakpoints deux bits dont la valeur 00 représente un octet de mémoire surveillé, 01 deux octets, 10 quatre octets et enfin 11 huit octets.

## 2.2 Rootkits

L'intérêt de cette partie sera d'effectuer une première présentation des rootkits nottament dans leur principe de fonctionnement mais aussi de leurs apparitions au cours des dernières années. Nous reviendrons plus précisément, dans le chapitre suivant, sur les méthodes qu'ils utilisent pour parvenir à leurs fins.

## 2.2.1 Cycle de vie d'un rootkit

Comme dirai Alfred de Musset, « Qu'importe le flacon, pourvu qu'on ait l'ivresse ».

Ainsi, quel que soit le rootkit étudié, et peu importe les fonctionnalités qu'il permet, son principe est toujours le même, se composant essentiellement de quatre grandes étapes :

- **Collecte d'information**

Avant de pouvoir commencer à corrompre le système, le rootkit doit tout d'abord récupérer les informations essentielles à son exécution, le plus souvent, c'est l'offset de la table des appels systèmes qui est primordiale, afin de pouvoir les détourner par la suite aux moyens des routines corrompues.

- **Déploiement des composants**

Celui-ci consiste essentiellement à l'installation à proprement parler du rootkit, par chargement du module dans le noyau ou injection de code dans la mémoire volatile, puis grâce à un mécanisme de hook des appels systèmes, il dissimule sa présence (omission du module dans le lsmod pour un LKM rootkit par exemple).

- **Mise à disposition de services**

De la même manière que lors de son déploiement, le crochetage des appels systèmes permet de mettre à disposition des services qui seront dissimuler à la pauvre victime. Il n'y a pas tellement de limite aux services proposés si ce n'est l'ingéniosité de l'attaquant. On peut malgré tout les décomposer en deux types : passif (espionnage, keylogger ...) et actif (suppression, modification de données sensible, DOS ...).

- **Survie au sein du système compromis**

Le but d'un rootkit est avant tout de pérenniser un accès à une machine corrompue. Pour cela, il tentera de se protéger des logiciels de détection tel que chkrootkit ou autre mais aussi de survivre à un redémarrage par le biais d'un script qui sera caché au système.

Les rootkits ne sont pas en eux mêmes des « exploits ». Même s'ils cachent souvent des maliciels, leurs techniques peuvent aussi être utilisées par des logiciels sains, à commencer par les utilitaires de défense.

La méthode pour émettre un appel système ou le réceptionner peut varier. Néanmoins un rootkit consiste à se positionner entre l'appelant et l'appelé. Ainsi il sera possible de modifier le contenu de la réponse.

Nous verrons plus en détails dans le chapitre suivant les diverses manières que peut utiliser celui-ci.

## 2.2.2 Historique

Au fur et à mesure de leurs apparitions, les rootkits ont entraîné une réponse de la part des développeurs informatique sous forme de nouveaux moyens de protection du système d'exploitation. Le jeu du chat et de la souris a alors commencé, les pirates tentant de contourner ces nouvelles règles en complexifiant de plus en plus leurs programmes malicieux. En l'espace d'une dizaine d'années, les rootkits sont donc passés de l'espace utilisateur (Ring 3) à l'espace noyau (Ring 0) et enfin à la virtualisation (Ring -1). Dans chacun de ces « anneaux », un nombre considérable de variantes d'un même principe ont été développées, comme le résume la liste non exhaustive de la figure 2.2.

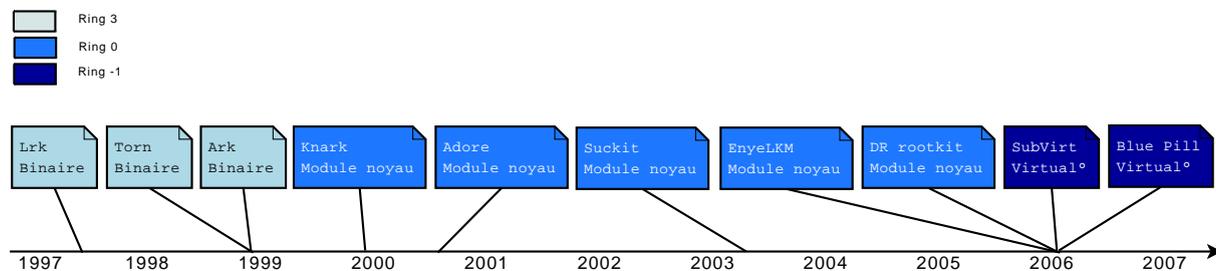


FIG. 2.2 – Chronologie et complexification des rootkits sous Linux

Il est difficile de lister l'intégralité des rootkits existant ou ayant existés, et ce, principalement pour trois causes.

Premièrement, une majeure partie des rootkits cités a évolué au cours du temps pour donner lieu à des versions 2.0, 3.0 ou autres appellations semblables. Il faut donc essayer de les voir en tant que famille plutôt que dans leur individualité.

Deuxièmement, les rootkits sont des structures souvent organisées de façon modulaire (cœur + fonctions annexes) et donc facilement modifiables. Les concepteurs ont donc eu le loisir de s'appuyer sur des modèles préexistants, créant ainsi des dérivés - probablement non connus - des rootkits cités précédemment.

Et troisièmement, il faut tenir compte du fait que certains rootkits n'ont pas encore été rendus « public », tant au niveau de la preuve de concept que de l'implémentation, car constituant une trop grande menace du point de vue de la « défense » ou trop utile pour être dévoilé si l'on se place au niveau de « l'attaquant ».

Les quelques exemples de la partie qui suit n'auraient probablement pas été connus si un utilisateur n'avait pas découvert, de manière relativement inopportune, leur présence au sein d'un système supposé sain.

### 2.2.3 Exemples

Il existe, en effet, des exemples d'utilisation industrielle de rootkits. Comme évoqué lors de l'introduction, en 2005, un de ces programmes a été découvert dans des CD audio de marque Sony-BMG, l'installation se faisant, lors de l'écoute, comme composant de gestion numérique des droits (DRM). Ce rootkit permettait, une fois chargé, de cacher tous les fichiers dont le nom commençait par \$sys\$. Cette fonctionnalité a été exploitée par des virus pour cacher leur code malveillant et échapper ainsi aux programmes anti-virus.

Durant l'année 2006, c'est au tour de Symantec Corporation d'être démasqué pour avoir tenté d'insérer un rootkit dans son anti-virus Norton. Celui-ci cachait le répertoire NProtect et son contenu, permettant ainsi de récupérer des données sensées avoir été supprimées par l'utilisateur. Le scandale provoqué par l'affaire Sony-BMG un an plus tôt les incitera à publier un patch correctif afin d'éviter une action juridique de la part des clients concernés.

On aurait alors pu croire que ces révélations auraient dissuadé d'éventuels successeurs. Il n'en fut rien et c'est à nouveau Sony-BMG qui se retrouva au centre de la polémique, courant 2007, après la mise en circulation de ses clés USB biométrique recélant elles aussi un rootkit. Pour sa défense, Sony-BMG rejettera la faute sur un sous-traitant taiwanais qui l'aurait incorporé à leur insu et ce malgré le devoir de vérification dont aurait dû faire preuve Sony-BMG.

L'intégration de rootkit aux logiciels n'est donc plus simplement réservée aux pirates informatiques en quête d'exploits mais se démocratise au point d'être utilisée directement par les producteurs de ces logiciels ou équipements commerciaux.

L'attaque par DOS subie par la Géorgie lors de son conflit avec la Russie (perte de plus de 80% de ses services informatiques), lors de l'année 2008, repose elle aussi sur l'utilisation de rootkits. Ceux-ci permettant la prise de contrôle de milliers de machines, il fut alors facile à l'attaquant de synchroniser l'ensemble des requêtes effectuées afin de saturer les serveurs ciblés.

## Chapitre 3

# Moyens de corruption d'un système

Il existe différents vecteurs d'attaque pour infecter un système, s'appuyant sur chacun des rings décrits lors du chapitre précédent. Le but de cette partie sera donc de faire un état de l'art des techniques de corruption mises à disposition.

### 3.1 Ring 3

Les rootkits visant le ring 3 - ou espace utilisateur - sont les premiers à avoir été développés massivement. Ce sont les plus faciles à mettre en place mais aussi à détecter. Il en existe essentiellement deux familles :

#### 1. Les rootkits "Applicatifs"

Ils modifient ou remplacent des logiciels applicatifs sains par leur "code" qui inclut généralement un cheval de Troie. Sous Linux, au moment de leur installation, ils remplacent souvent les utilitaires de système suivants : `find`, `ifconfig`, `login`, `ls`, `netstat`, `ps`, etc ... afin de masquer leur présence et un programme permettant de mettre la machine "en écoute" est démarré, dans le but de pouvoir "parler" au rootkit. Ceux-ci sont donc simples à concevoir mais en contrepartie une simple méthode d'empreinte tel MD5 permet de s'en prémunir.

#### 2. Les rootkits "Bibliothèque"

Cette technique de rootkit en « mode utilisateur », qui n'altère pas le noyau, est connue sous le nom de "API hooking". Ils modifient ou remplacent les appels ("system calls") aux ressources du système d'exploitation pour les orienter vers eux. Cette méthode a pour but de toucher un maximum d'applications grâce à un minimum de modification. De même que les rootkits applicatifs, il est facile de détecter et de contrer ce genre d'attaque via une compilation statique des programmes.

Comme rootkit en mode utilisateur, on trouve `lrk`, `t0rn`, `ark` et autres. Prenons `tr0n` comme exemple. Celui-ci exécute une série d'actions pour cacher sa présence dans le système au moment de son installation :

1. stoppe `syslogd-demon`,
2. remplace les utilitaires de systèmes : `du`, `find`, `ifconfig`, `login`, `ls`, `netstat`, `ps`, `top`, `sz`,
3. une version trojan de `syslogd-demon` est rajoutée dans le système,
4. un sniffeur est démarré en tâche de fond,
5. le lancement des daemons `telnetd`, `rsh`, `finger` est rajouté dans `inetd.conf`.

D'ordinaire, `tr0n` se situe dans `/usr/src/.puta` mais grâce à la commande `ls` déjà installée, ce dossier est invisible.

Des contre-mesures ayant été trouvées et les possibilités offertes étant restreintes, les concepteurs de rootkits se sont donc naturellement tournés vers un espace plus privilégié du système, à savoir l'espace noyau.

## 3.2 Ring 0

Au contraire de l'espace utilisateur, le ring 0 correspond à un niveau privilégié du système. En effet, celui-ci rend possible un accès à des ressources que les rootkits décrits ci-dessus ne permettraient pas, car ne bénéficiant pas des droits suffisants.

### 3.2.1 Exploration de la mémoire noyau

Afin de lire la mémoire noyau il est possible d'utiliser, si le noyau le permet, le périphérique `/dev/kmem`. C'est la technique utilisée par le rootkit `suckIT`, dont la dernière version est estimée à 2003. On ouvre ce périphérique à la manière de n'importe quel fichier :

```
kmem = open(KMEM_FILE, O_RDONLY, 0);
```

Deux routines sont implémentées pour accéder à la mémoire noyau via `/dev/kmem` :

- `rkm()` permet de lire la mémoire à une adresse spécifiée et récupérer le motif dans un buffer.
- `wkm()` permet d'écrire dans la mémoire à une adresse spécifiée en paramètre.

Comme le montre l'implémentation décrite dans la figure 3.1, afin de lire la mémoire on se place à l'adresse `offset` et on copie `size` octets dans `buf` et pour écrire, il suffit de se placer au bon offset mémoire, puis d'utiliser les fonctions standards.

```
static inline int rkm(int fd, int offset, void *buf, int size){
    if (lseek(fd, offset, 0) != offset) return 0;
    if (read(fd, buf, size) != size) return 0;
    return size;

static inline int wkm(int fd, int offset, void *buf, int size){
    if (lseek(fd, offset, 0) != offset) return 0;
    if (write(fd, buf, size) != size) return 0;
    return size;
```

FIG. 3.1 – Implémentation des routines de lecture/écriture en mémoire en espace noyau via `/dev/kmem`

La première étape de repérage d'adresse mémoire est réalisée via la première fonction `rkm()`. Dans notre exemple, ceci est implémenté par le module `extract.c` de `suckIT`. Ce module est donc responsable de récupérer l'adresse de la `syscall_table`.

```
ulong get_sct(ulong *i80){
struct idtr idtr;
struct idt idt;
int kmem;
ulong sys_call_off;
char *p;
char sc_asm[INT80_LEN];
asm("sidt %0" : "=m" (idtr)); // Recuperation de IDT
// Ouverture du peripherique /dev/kmem
kmem = open(KMEM_FILE, O_RDONLY, 0);
// Lecture de la 80eme case de IDT (routine syscall_call)
if (!rkm(kmem, &idt, idtr.base +
    sizeof(idt)*SYSCALL_INTERRUPT, sizeof(idt))) return 0;
// mise en forme de l'adresse syscall_call
sys_call_off = (idt.off2 << 16) | idt.off1;
// recuperation de l'ensemble de la routine syscall_call
if (!rkm(kmem, &sc_asm, sys_call_off, INT80_LEN)) return 0;
close(kmem);
// Recherche en local par pattern de l'adresse de la table dans le code
p = memmem(sc_asm, INT80_LEN, "\xff\x14\x85", 3) + 3;
// mise en forme de l'adresse de la syscall_table
```

```

if (p){
    *i80 = (ulong) (p - sc_asm + sys_call_off);
    return *(ulong *)p;}
return 0;

```

Cette étape nous a permis d'effectuer des repérages mémoires. La deuxième étape consiste à poser des hooks pour détourner le flux d'exécution. Dans notre exemple, les adresses de la `syscall_table` sont modifiées. On écrit directement dans cette table l'adresse de notre propre fonction, cette action efface donc la véritable adresse de l'appel système, il est donc nécessaire de la sauvegarder, pour pouvoir l'utiliser potentiellement par la suite lors de l'implémentation de notre propre fonction (troisième étape).

### 3.2.2 Corrompre la table des appels systèmes

Comme le montre la figure 3.2, la `syscall_table` est modifiée directement. `rkm()` est utilisée pour sauvegarder l'adresse de l'appel système à détourner, et `wkm()` est utilisée pour réécrire l'adresse de la fonction. L'adresse à écrire est celle de notre propre fonction. Dans notre exemple, `suckIT` tient à jour une structure `handlers` contenant chaque numéro d'appel système, permettant de trouver l'adresse à réécrire à partir de l'adresse de la `syscall_table` et le pointeur de fonction corrompue, à écrire en mémoire lors du hook. Ceci permet de hooker tous les appels systèmes en parcourant la structure `handlers`.

```

//Mise en place du hook par écrasement des adresses
while (handlers->nr) {
    if ((ulong) handlers->handler){
        // Sauvegarde de l'adresse
        rkm(kmem, sct + (handlers->nr * 4),
            handlers->old_handler, 4);
        // Ecrasement de l'adresse
        wkm(kmem, sct + (handlers->nr * 4),
            handlers->handler, 4);
    }
    handlers++;
}

```

FIG. 3.2 – Hook par altération de la `syscall_table`

### 3.2.3 Subtil inline Hooking

Il existe un autre moyen de détourner les appels systèmes de manière plus subtile. Cela nécessite de comprendre en détail la façon dont fonctionnent les routines bas-niveaux du noyau. La figure 3.3 est issue d'un dump du noyau 2.6.28. La colonne de gauche représente les adresses mémoire, au milieu l'opcode et à droite l'instruction assembleur traduite.

```

c0403c88 <system_call>:
c0403c88:    50                push   %eax
c0403c89:    fc                cld
    --- snip --- snip --- snip --- snip --- snip --- snip ---
c0403cbd:    0f 83 4d 01 00 00  jae   c0403e10 <syscall_badsys>

c0403cc3 <syscall_call>:
c0403cc3:    ff 14 85 00 3b 65 c0  call  *-0x3f9ac500(,%eax,4)
c0403cca:    89 44 24 18       mov   %eax,0x18(%esp)

c0403cce <syscall_exit>:

```

FIG. 3.3 – Désassemblage du noyau dans la zone des appels systèmes issus de `int 80h`

- La première étape consiste à récupérer trois pointeurs précis :
- adresse d’avant l’exécution de l’appel, noté *before\_call* (BC).
  - adresse d’après l’exécution de l’appel, noté *after\_call* (AC).
  - adresse de la fin de l’appel (*syscall\_Exit*), note *exit*

Ces adresses récupérées, une pseudo table des appels systèmes va être implémentée. Une fonction *new\_sct* va effectuer un switch sur le registre *eax* pour savoir quel appel est invoqué. Si l’appel ne concerne pas le rootkit, on saute vers *before\_call*. Si l’appel est concerné, on exécute la routine hookée, puis on saute vers *after\_call*.

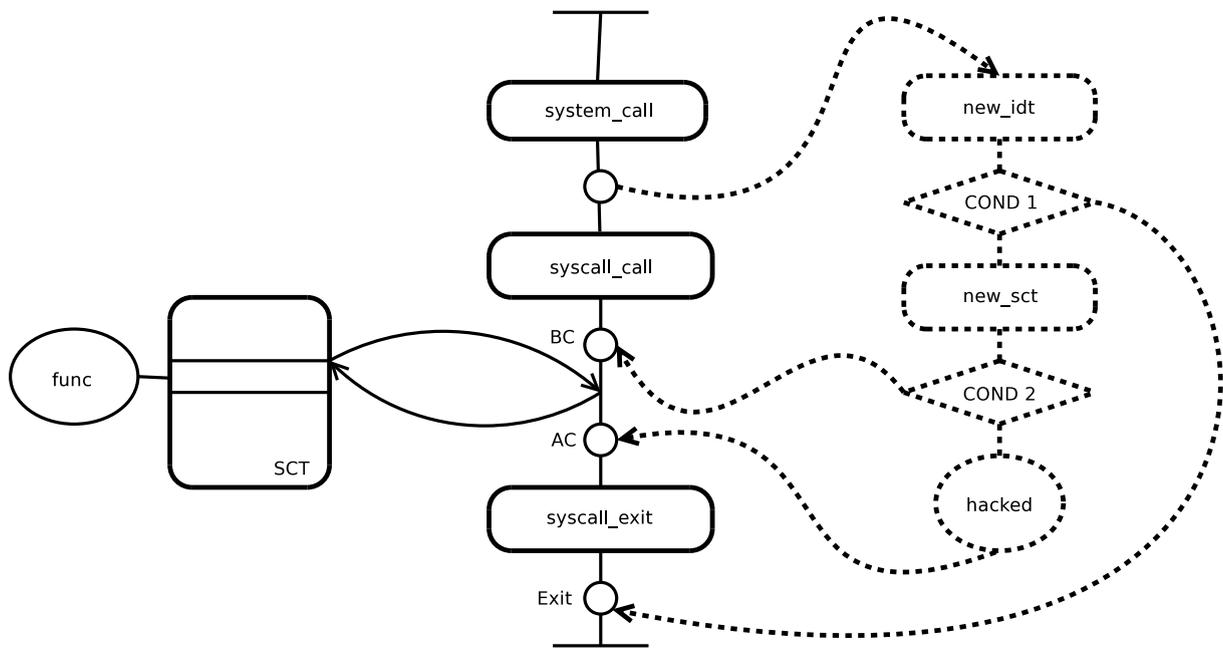


FIG. 3.4 – Inline hooking sans altération de la SCT

La figure 3.4 résume les différents endroits où vont se greffer les hooks. On peut aussi distinguer sur la figure les trajectoires du flux d’exécution dans le cas :

- d’un appel système normal non concerné par le détournement (COND2).
- d’une erreur « normale » dans le système (COND1).
- d’un appel système normal concerné par le détournement.

La façon d’accrocher les routines s’effectue en assembleur directement dans le code du noyau. Deux cas sont possibles :

- Patch du noyau : il existe des contraintes de taille, pour ne pas réécrire du code noyau important.
- Assembleur dans notre routine : aucune contrainte dans ce cas.

Il existe plusieurs techniques pour accrocher une fonction :

- on peut écrire un *jump* en assembleur.
- on peut faire un *push* puis un *ret*.
- on peut faire un *call*.

La figure 3.4 montre bien qu’aucune structure de la mémoire noyau n’est altérée, à l’exception de quelques octets pour le premier hook. En ce sens, cette technique est plus subtile, car elle ne touche pas la tables des appels systèmes.

### 3.2.4 Debug Register

Le mécanisme de hook utilisant les registres de debuggages unix offre une ultime furtivité. La figure 3.5 nous montre le déroulement de détournement d'un appel système :

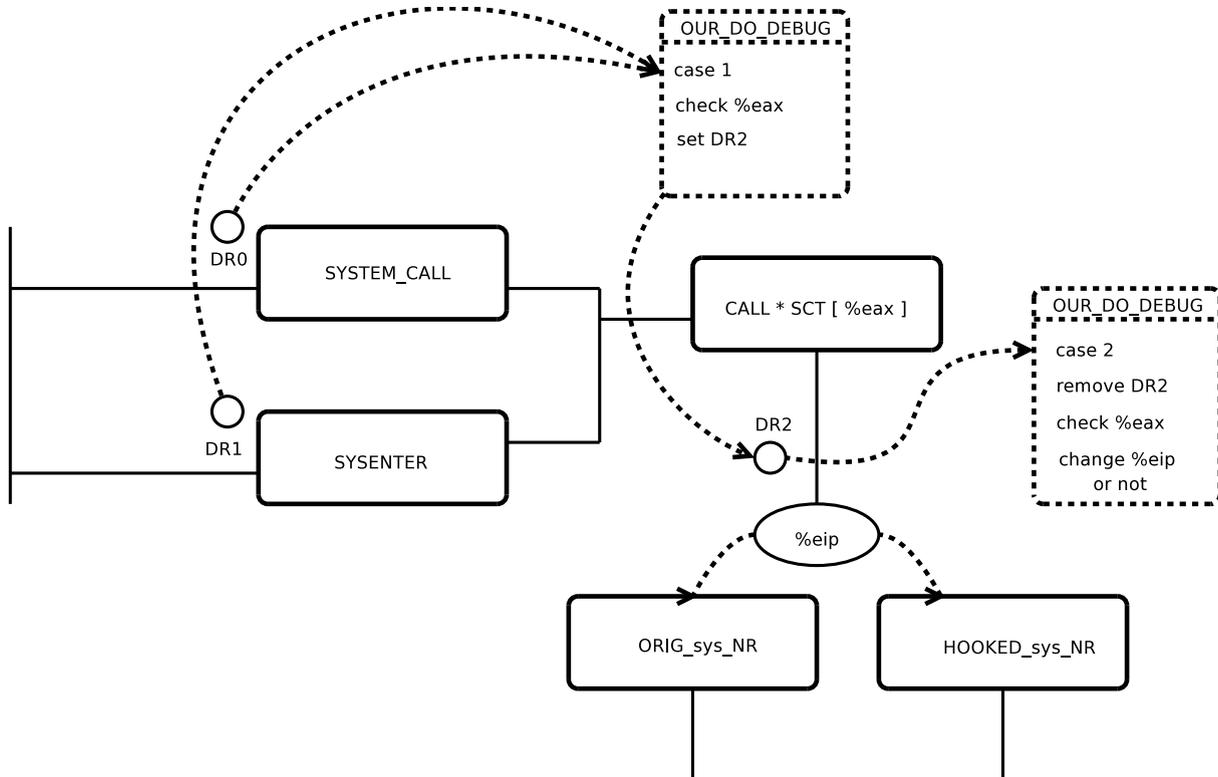


FIG. 3.5 – Mise en place du mécanisme de hook via les registres de debug

On peut découper quatre phases dans le processus de hook via les debugs :

- hook de la routine de debuggage unix.
- positionnement de breakpoints fixés sur le chemin de tout appel système.
- positionnement d'un breakpoint temporaire et sous condition.
- détournement du flux via le registre %eip.

En suivant la chronologie d'un appel système étudié dans une section précédente, il existe deux endroits par lesquels le flux d'exécution passera obligatoirement. Les breakpoints sont positionnés sur ces deux endroits, sur la routine d'interruption 80 `system_call` et sur l'instruction `sysenter`. Ainsi, quel que soit l'appel système déclenché, le flux d'exécution va se retrouver sur le breakpoint fixe et, par conséquent dans la routine de debuggage qui a été préalablement hookée.

Une fois dans notre routine de debuggage, nous avons accès au registre du processeur et nous pouvons donc savoir si l'appel système est concerné par notre hook via le registre %eax. S'il n'est pas concerné, rien ne se passe. S'il est concerné, un breakpoint est placé précisément à l'adresse de l'appel système. Le flux d'exécution reprend son cours, passe par la table des appels système qui demeure inchangée, puis arrive sur l'adresse de l'appel système. Le breakpoint est donc levé et la routine de debuggage s'exécute à nouveau. Ce breakpoint est détruit, d'où l'appellation breakpoint temporaire, puis le registre %eip contenant l'adresse de la prochaine instruction à exécuter, qui doit être égale à l'adresse de l'appel système, est effacé puis remplacé par une autre de notre choix.

C'est ici que se situe la force du rootkit. En effet, la table des appels système n'est pas altérée, le système de debuggage est un système standard sous unix. Les effets de bord se limitent donc à un seul hook et à deux breakpoints. Le breakpoint temporaire n'existe qu'entre l'arrivée sur `system_call` et l'arrivée sur l'adresse de l'appel système original i.e. quelques instructions assembleurs. Nous allons dans la partie implémentation nous baser sur ce mécanisme, en adaptant le code de DR rootkit au noyau 2.6.28.

## 3.3 Ring -1

### 3.3.1 Systèmes virtualisés

La virtualisation est un ensemble de procédés logiciels et/ou matériels permettant le fonctionnement de plusieurs systèmes d'exploitation sur une même machine. Ceux-ci se comportent alors comme s'ils étaient des machines physiques distinctes. Bien que ces techniques datent de 1960, la popularisation des machines virtuelles a eu lieu au début des années 2000 avec VMware et donna naissance à tout un panel de logiciels libres ou propriétaires.

Ensuite, les constructeurs ont équipé leurs processeurs d'un nouveau jeu d'instructions, ainsi, avec l'apparition des derniers processeurs AMD64 et Intel Dual Core, des mécanismes facilitant la virtualisation totale ou à défaut la para-virtualisation permettent de réaliser le passage entre les différentes machines virtuelles directement au niveau du processeur.

Il existe donc différentes catégories de virtualisation listées et schématisées ci-dessous :

- émulation,
- virtualisation totale,
- para-virtualisation,
- virtualisation matérielle.

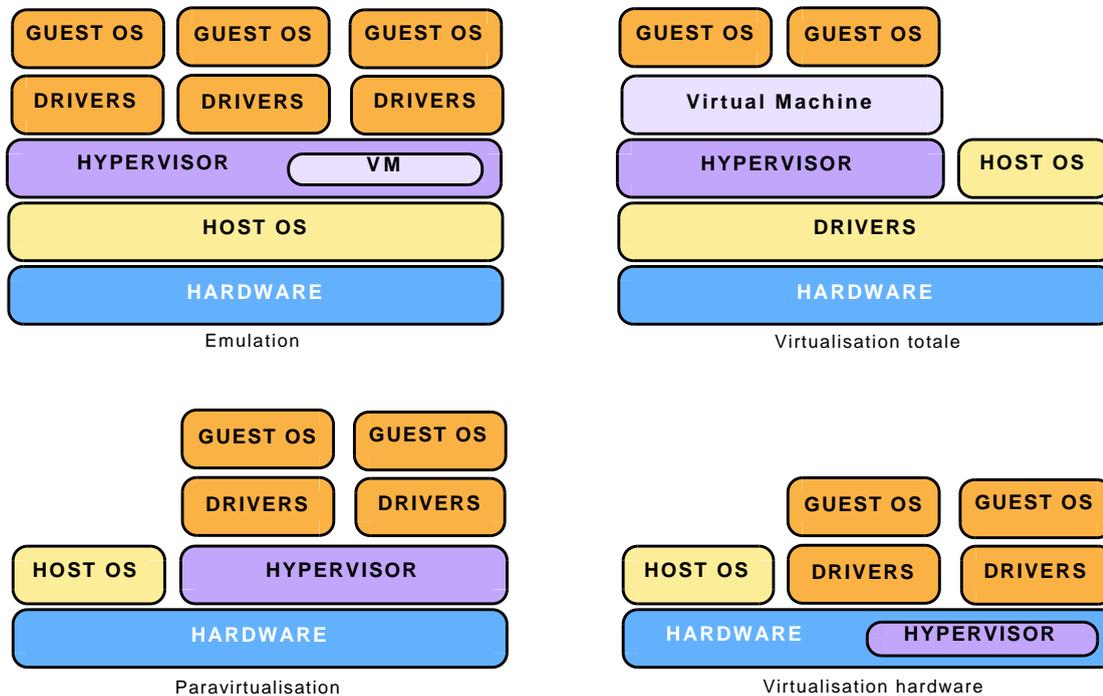


FIG. 3.6 – Les différents modes de virtualisation

L'émulation, qui s'oppose à la simulation ou modélisation, est la reproduction du comportement d'un système matériel, logiciel ou autre. Les deux points suivants sont les techniques utilisées par les logiciels comme VMware ou Virtualbox et leurs machines virtuelles. Quant au dernier point, il correspond à cette nouvelle génération de virtualisation liée au processeur.

Le problème posé par les rootkits de type HVM (Hardware Virtual Machine) est qu'ils n'installent aucun hook en mémoire, se satisfaisant du rôle d'hyperviseur leur permettant de gérer entièrement la machine et ainsi de brouiller toutes pistes.

Nous allons en évoquer deux, Blue Pill, fondé sur ce principe de virtualisation matérielle et SubVirt, reposant sur le principe de virtualisation totale.

### 3.3.2 Blue Pill

BluePill est le premier rootkit HVM public, réalisé par Joanna Rutkowska en 2006, et a fait l'objet de nombreuses publications. La première version publique (0.11) de BluePill ne supportait que la virtualisation sur les processeurs AMD. La dernière version (0.32) permet une plus grande souplesse, en supportant les processeurs AMD et Intel, et également en écrivant directement dans les logs systèmes.

Son principe est simple, il permet de faire passer un système d'exploitation en tant que système invité et ce, au niveau du processeur. Du fait qu'il n'effectue aucun hook, on le prétend indétectable mais il n'a malheureusement pas que des avantages. En effet, il ne peut contenir aucune fonction active. Soit il hook des fonctions ou structures pour réaliser le comportement de rootkits dits classique et permet des mécanismes de détection bien connus, soit il surveille les entrées/sorties. Cette dernière solution est envisageable dans la théorie mais le coût en temps de traitement serait alors largement augmenté.

Les deux schémas ci-dessous décrivent la méthode d'infection de Blue Pill :

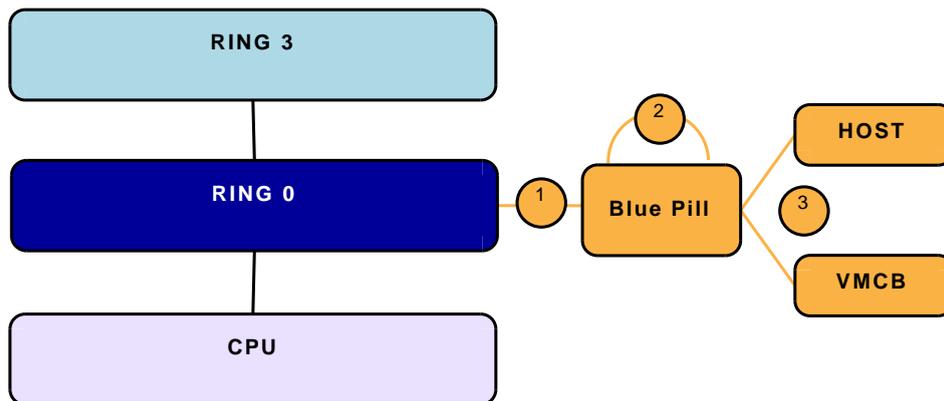


FIG. 3.7 – Avant infection par Blue Pill

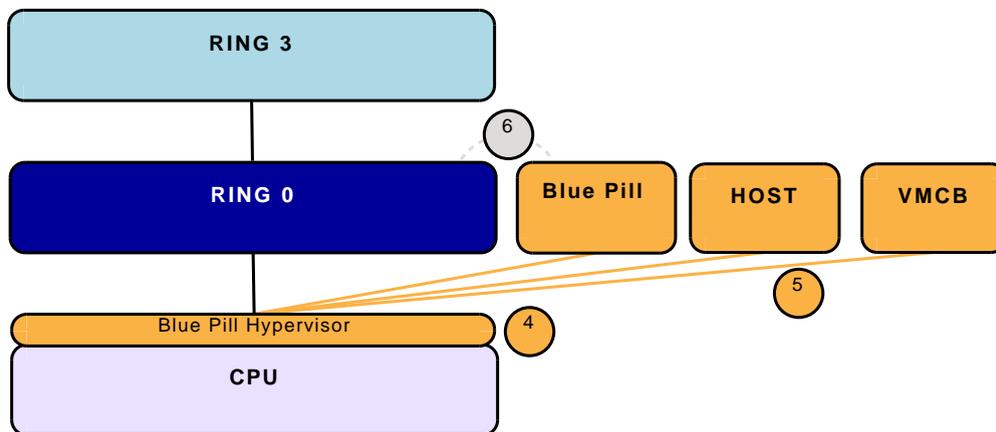


FIG. 3.8 – Après infection par Blue Pill

1. Chargement du driver,
2. Vérification/Activation la virtualisation matérielle,
3. Allocation des espaces mémoire nécessaire et initialisation des structures,
4. Transfert de l'exécution au code de l'hyperviseur,
5. Appel de l'instruction qui lance la machine virtuelle,
6. Déchargement du driver.

Ce genre de rootkit est donc pratiquement indétectable dans la théorie mais en pratique, les ressources et le temps consommé pour masquer sa présence pourraient le trahir. On pourrait vouloir se pencher plus amplement sur ce sujet, malheureusement, la partie la plus intéressante du code de Blue Pill n'est disponible que contre la modique somme de 200 000\$ payable à son auteur. On peut donc se demander ce que justifie ce prix, soit protéger l'innovation qu'apporte Blue Pill, soit dissuader la communauté de découvrir l'intox qu'il constituerai.

### 3.3.3 SubVirt

SubVirt est aussi un rootkit prenant appui sur la virtualisation mais il fonctionne sur un principe différent.

Il modifie la séquence de boot de l'OS pour être lancé avant lui et démarre à son tour le système d'exploitation en tant que machine virtuelle. Il peut alors intercepter les communications de son invité<sup>1</sup> avec les composants matériels.

La figure 3.9 résume le principe d'installation et de fonctionnement de SubVirt :

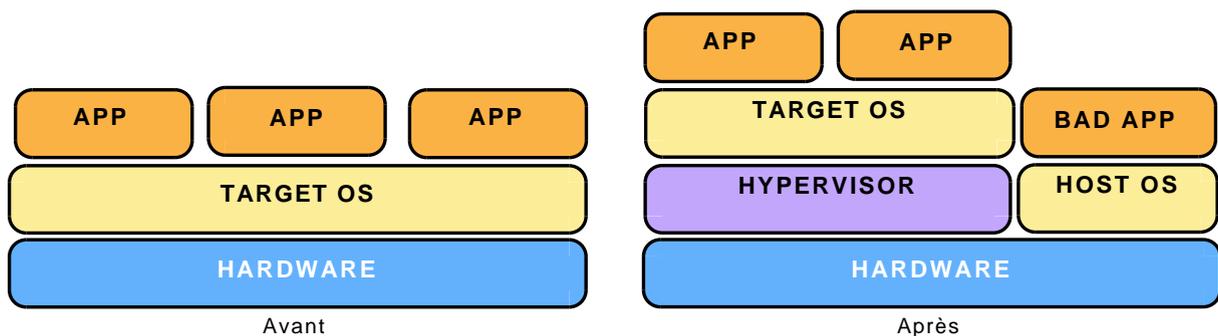


FIG. 3.9 – Avant/après infection par SubVirt

Ce rootkit permet de faire passer un système d'exploitation cible dans une machine virtuelle tout en contrôlant le système hôte. Il s'assure ainsi de l'incapacité de détection par la cible, celle-ci étant emprisonnée dans la machine virtuelle et n'ayant pas connaissance du système hôte corrompu.

Comme tout rootkit s'appuyant sur la virtualisation, il est difficilement détectable mais, le fait qu'il modifie la séquence de boot de l'OS afin de survivre au redémarrage le rend moins furtif. Au contraire de Blue Pill, il existe quelque moyens de détections mais ceux-ci restent relativement inefficaces.

---

<sup>1</sup>guest OS

# Chapitre 4

## Elaboration d'un rootkit

Nous allons décrire dans cette section les modules noyaux que nous avons écrit et qui fonctionnent. Ceux-ci ont été complètement réécrits mais largement inspirés des codes sources des rootkits `enye1km` et `dr rootkit`.

### 4.1 Généralité

Nous avons tenté au cours de ce projet d'élaborer plusieurs implémentations de détournement d'appels système en utilisant plusieurs techniques - actuelles - développées dans les sections précédentes. Cette implémentation a pour but d'étayer nos propos et d'effectuer une preuve de concept de détournement d'un appel système sur un noyau linux 2.6.28.

Les contraintes prises en compte peuvent se ramener aux propositions suivantes :

- compilation et exécution stable sur un noyau linux 2.6.28.
- modularité relative aux fonctions hookées (voir section suivante).
- pouvoir décrire et tester plusieurs principes de hook et montrer leurs limites.

Nous allons étudier dans la section suivante la conception des modules :

### 4.2 Conception

L'idée générale de notre projet est de permettre facilement l'ajout d'un hook d'un appel système de façon simple. Notre rootkit va se composer de deux modules, un module `général` et un module `client`.

Le module général s'occupe de mettre en place le hook. Nous avons pour l'instant autant de modules généraux que de mécanismes de hook. Idéalement, ces mécanismes pourraient être fusionnés dans un seul module qui proposerait le choix de la méthode de hook.

Le module client contient l'ensemble des fonctions hookées. On dispose dans ce module de variables globales pointant sur des adresses précises en mémoire<sup>1</sup>. Pour réécrire un appel système, aucune information spéciale n'est nécessaire. Un ensemble de macros, défini dans `arch/x86/include/asm/unistd_32.h`, définissent l'index de chaque appel système dans la table des appels système en fonction de leur nom.

Par exemple, l'appel système `kill` est dans la 37ème case de la `syscall_table`, une macro est définie pour permettre d'utiliser ce nombre sans le connaître :

```
#define __NR_kill 37
```

Ainsi lors de l'écriture de notre routine, il est très simple de savoir l'adresse de la fonction originale que l'on réécrit, elle se situe à l'adresse `sct_glb[__NR_kill]`, avec `sct_glb` l'adresse de la table des appels système, qui est une variable globale.

---

<sup>1</sup>cf la section sur les variables globales.

La figure 4.1 montre comment les modules interagissent entre eux. Après avoir choisi, avant compilation, la méthode de hook utilisée, bleu ou rouge sur la figure, et après avoir écrit et lié l'ensemble des fonctions hookées, le module noyau sera fonctionnel et détournera les appels systèmes demandés et écrits dans les fichiers préfixés par `hook_`, avec la méthode sélectionnée.

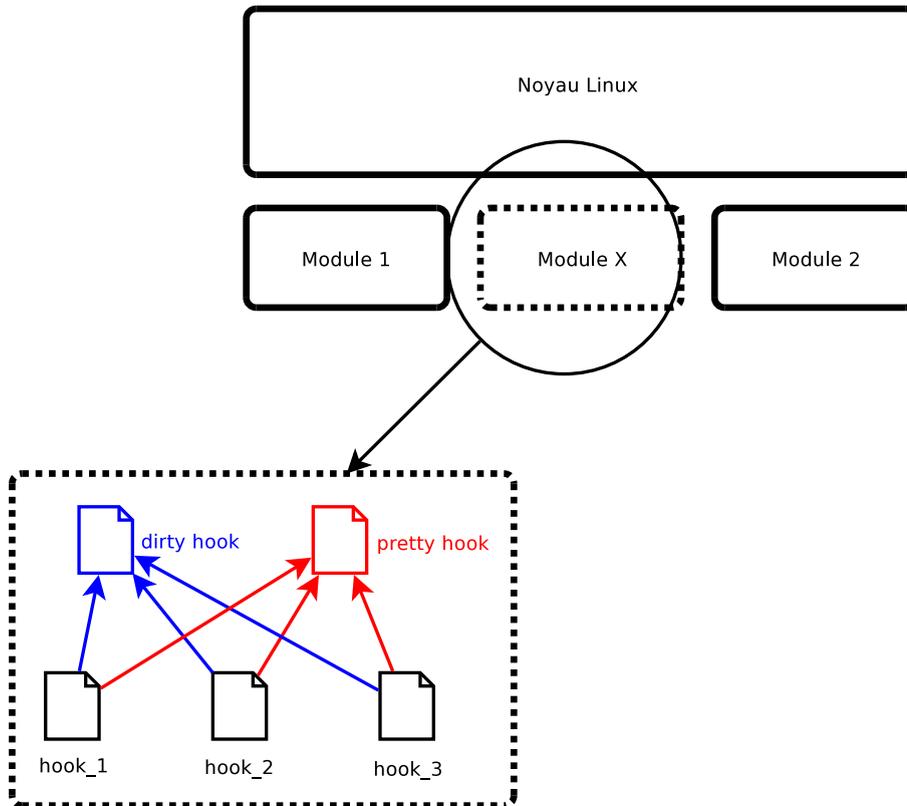


FIG. 4.1 – Découpage de nos implémentations

## 4.3 Implémentations

Deux techniques ont été implémentées dans ce projet.

- altération de la table des appels système (dirty hook).
- utilisation des breakpoints unix sans altération de la table des appels (pretty hook).

Les deux techniques abordées nécessitent une phase de recherche d'informations ou d'adresses. Nous allons traiter dans une première sous-section les différentes implémentations réalisées dans le but de rechercher ces adresses, qui sont communes à nos deux implémentations. Ensuite nous étudierons le dirty hook, et enfin nous décrivons notre implémentation du hook utilisant les registres de debug.

### 4.3.1 Recherche d'informations générales

Nous avons décrit dans la première section, sur les rappels, le cheminement d'un appel système. Toutes les implémentations de rootkits LKM doivent, obligatoirement, connaître un certain nombre d'adresses :

- adresse de l'interruption 80 `system_call`.
- adresse de l'entrée `sysenter` `ia32_sysenter_target`.
- adresse de la table d'appel système `sys_call_table`.

De façon générale, un hook va remplacer l'adresse d'une fonction *d'origine* par une fonction *hookée*. S'il est facile d'accéder à l'adresse de la fonction hookée, car nous la codons dans notre module, il est moins

facile de trouver celle d'origine. Cela devient très simple avec l'adresse de la table des appels système. En effet, pour fixer les idées, le numéro de l'appel système `kill` est 37. L'adresse originale s'obtient donc en sommant l'adresse de la table avec  $37 * 4$  octets, car la table contient des cases de 4 octets, qui correspondent aux adresses des appels. Plus simplement, en copiant l'adresse de la table directement dans un pointeur de tableau, on peut accéder à la fonction `kill` originale en prenant la 37ème case. Nous utilisons la deuxième méthode comme le montre la figure 4.2.

```
asmlinkage static int hacked_kill(int pid, int sig)
{
    void **sys_p = (void **)sct_glb; /* prealablement calculee */
    asmlinkage long (*original_sys_kill)(int pid, int sig) = sys_p[37];
```

FIG. 4.2 – Récupération de la fonction d'origine

### Interrupt Descriptor Table

C'est la première adresse à trouver, et aussi la plus simple. Cette table contient toutes les routines d'interruption. Le processeur stocke son adresse dans un registre, il est donc possible de la récupérer via une commande assembleur :

```
asm ("sidt %0" : "=m" (idtr));
base = *((uint32_t *) &idtr[2]);
```

Comme décrit dans une section précédente, chaque case de l'IDT est composée de 8 octets. On trouve par conséquent les interruptions 1 et 80 avec :

```
sys = (uint32_t *)(base + (0x80*8));
deb = (uint32_t *)(base + (0x01*8));
```

Les premiers octets représentent les bits de poids forts, les derniers octets ceux de poids faibles, on choisit de mettre toute la case dans une structure, et de calculer l'adresse :

```
memcpy(&desc, (void *) sys, sizeof(desc));
sum_up = (uint16_t)(desc.off_high) << 16 ;
sum_down = (uint16_t) desc.off_low ;
int80_glb = sum_down + sum_up;
```

### Récupération de la table des appels

Il suffit d'observer la figure 3.3 pour découvrir qu'il suffit de partir de la routine d'interruption 80 `system_call`, puis rechercher le motif `ff 14 85` pour avoir l'adresse de la table des appels `sys_call_table`. Le code suivant réalise cette tâche.

```
uint8_t *p = (uint8_t *) int80_glb;
while (!(p[0] == 0xff) && (p[1] == 0x14) && (p[2] == 0x85))
    p ++;

sct_glb = (uint32_t **)*(uint32_t *) (p+3);
```

### Variables Globales

Nous avons décidé dans un premier temps, de globaliser les variables-clés précédemment décrites :

```
extern uint32_t int1_glb;      /* Adresse interruption 1 */
extern uint32_t ** sct_glb;   /* Adresse table des appels */
extern uint32_t int80_glb;    /* Adresse interruption 80 */
extern uint32_t do_deb_glb;   /* Adresse routine de debuggage unix */
extern uint8_t *pointer_offset; /* Adresse de base pour le calcul d'un offset */
```

### 4.3.2 Recherche complémentaire pretty hook

Dans le cas du pretty hook, il nous faut des informations supplémentaires, sur le déroulement des routines de debug unix :

- adresse de la routine d'interruption 1 `debug`.
- adresse de la routine de debug unix `do_debug`

L'interruption 1 se calcule de la même façon que l'interruption 80.

#### récupération de `do_debug`

Après avoir observé le fichier `entry.S` ou bien objdumpé le noyau et regardé à l'adresse de l'interruption 1, on constate que cette routine appelle la routine de debug unix `do_debug`.

On recherche donc l'opcode `e8` responsable du `call` afin de récupérer l'offset de la fonction. Comme le montre la figure 4.3, lorsque la routine `debug` est exécutée, l'appel de la sous-routine `do_debug` est calculée à l'adresse `c064e602`, en sommant l'adresse de base `c064e607` avec l'offset, entre crochets sur la figure. Cet offset est `6dd`. Ainsi, l'adresse de `do_debug` est récupérable via le résultat de l'addition. En effet,  $0xc064e607 + 0x6dd = 0xc064ece4$ .

```
c064e5c0 <debug>:
c064e5c0: 81 3c 24 64 3b 40 c0    cmpl   $0xc0403b64, (%esp)
c064e5c7: 75 17                    jne    c064e5e0 <debug_stack_correct>
c064e5c9: 66 83 7c 24 04 60      cmpw   $0x60, 0x4(%esp)
c064e5cf: 75 0f                    jne    c064e5e0 <debug_stack_correct>
--- snip ---
c064e602: e8 [dd 06 00 00]       call   c064ece4 <do_debug>
c064e607: e9 14 55 db ff        jmp    c0403b20 <ret_from_exception>
--- snip ---
c064ece4 <do_debug>:
c064ece4: 55                      push   %ebp
c064ece5: 89 e5                   mov    %esp,%ebp
```

FIG. 4.3 – Dump du noyau dans la plage d'adresse relative au debug

### 4.3.3 Mise en place dirty hook

Le premier hook consiste à altérer la table des appels système. La figure 4.4 décrit notre implémentation, basique, du hook. Il s'agit d'écrire brutalement l'adresse de notre fonction là où le noyau appelle la fonction originale. Notons qu'un module noyau calculant un checksum de la table des appels systèmes a intervalles réguliers peut repérer ce hook de façon immédiate.

```
void set_dirty_hook(int num, uint32_t hook)
{
    old_sys_glb = (uint32_t)sct_glb[num];
    sct_glb[num] = (uint32_t)hook;
}

void unset_dirty_hook(int num)
{
    sct_glb[num] = old_sys_glb;
}
```

FIG. 4.4 – API dirty hook

En effet, l'adresse de la fonction hookée persiste longtemps en mémoire. Afin de passer outre ce problème, nous pouvons penser à deux solutions :

- changer cette adresse uniquement dans le laps de temps où s'effectue l'appel. Quelques microsecondes rendent le diagnostic plus complexe. C'est une solution décrite dans *Phrack*.
- créer une autre table des appels. C'est la solution adoptée par Enyelkm.
- modifier en temps réel la valeur du registre `eip` afin que pile au moment où la fonction est appelée, nous mettons l'adresse de notre fonction dans ce registre. C'est la solution que nous pensons plus subtile, développée dans le rootkit DR `rootkit`. Nous avons adapté cette solution au noyau 2.6.28.

## 4.3.4 Mise en place pretty hook

### Utilisation des breakpoints

Nous avons construit pour l'utilisation des breakpoints, une API en langage C, décrite par la figure 4.5.

```
uint32_t get_dr0();
uint32_t get_dr1();
uint32_t get_dr2();
uint32_t get_stat();
uint32_t get_ctrl();

void set_dr0(uint32_t addr);
void set_dr1(uint32_t addr);
void set_dr2(uint32_t addr);
void set_ctrl(uint32_t ctrl);
void set_stat(uint32_t stat);

uint32_t build_ctrl_glb_dr0();
uint32_t build_ctrl_glb_dr1();
uint32_t build_ctrl_glb(void);
uint32_t build_ctrl_dr2(void);
```

FIG. 4.5 – API permettant de piloter les breakpoints unix

Elle est constituée de trois sous-ensembles de fonctions :

- fonctions permettant de positionner les breakpoints en écrivant dans les registres de debuggage unix : `set_*`
- fonctions permettant de lire les registres de debuggage unix : `get_*`
- fonctions permettant de construire le flag responsable du pilotage des breakpoints<sup>2</sup> `build_*`

Le positionnement des breakpoints, fixes ou variables peut ainsi se faire facilement. On positionne ainsi facilement un breakpoint sur l'interruption 80 :

```
set_dr0(int80_glb);
set_ctrl(build_ctrl_glb_dr0());
```

### Hook de `do_debug`

Le hook de `do_debug` est réalisé par la fonction `hack_offset`. La figure 4.3 décrit le déclenchement de la routine de debuggage lorsqu'un breakpoint a été relevé. Le hook de cette fonction va consister à modifier l'offset, encadré dans la figure, pour qu'il pointe sur notre propre fonction.

---

<sup>2</sup>cf : section 2.1.3

Les informations nécessaires pour mener à bien ce hook sont :

- adresse de l'offset (adresse de base).
- adresse de notre fonction `debug_hooked`.

L'adresse de base nous a été donnée par la phase de repérage. L'adresse de `debug_hooked` est immédiate. Une fois l'offset calculé, il faut le mettre en forme, comme le montre la figure 4.6. Les huit premiers bits sont récupérés avec l'opérateur `&` (ET bit-à-bit). Ce nombre de 32 bit est ensuite casté en nombre de 8 bits, (les seuls non nuls).

De la même façon, on récupère ensuite les bits 8-16, on décale à droite pour former un nombre de 32 bits dont seul les 8 premiers sont non-nuls et on est ramené au cas précédent, et ainsi de suite.

On commence par les bits de poids faibles car le concept intel (little endian) place les bits de poids faible d'abord, dans le sens de lecture.

```
pi[1] = (offset & 0x000000ff);  
pi[2] = (offset & 0x0000ff00) >> 8;  
pi[3] = (offset & 0x00ff0000) >> 16;  
pi[4] = (offset & 0xff000000) >> 24;
```

FIG. 4.6 – Mise en forme d'une adresse en little endian

# Chapitre 5

## Conclusion

Au cours de ce projet, nous avons pu acquérir une vision plus précise des mécanismes de hook et des fonctionnements bas-niveaux des appels système. La lecture de plusieurs codes, nous a permis d'appréhender de façon globale le problème des détournement.

Nous avons recueilli pour ce mémoire un ensemble d'informations préexistantes dans le but de les catégoriser selon certains critères, comme leur localisation ou leur furtivité. L'intérêt était de donner au lecteur une vue globale et actuelle des rootkits.

Après avoir étudié les mécanismes de hook avec des exemples concrets d'implémentation, nous avons dans ce projet pu effectuer une preuve de concept pour détourner un appel système sur les noyaux 2.6.28. En nous inspirant de `DR rootkit` et `Enyelkm`, nous avons construit notre module noyau, dont la conception permettra de greffer assez aisément d'autres fonctionnalités.

Techniquement et *a fortiori*, nous avons pu constater que l'installation de ce type de rootkit ne constitue pas une grande difficulté, malgré les modifications mises en place dans les dernières versions du noyau. Notons que nous avons travaillé sur un noyau dont la protection d'écriture sur les pages mémoires a été désactivée lors de la compilation. Il est néanmoins possible d'outrepasser cette protection à l'aide des mêmes mécanismes qui la réalisent.

On peut supposer, du fait de l'apparition exponentielle des rootkits lors de ces dernières années, que de nombreuses nouvelles techniques seront - ou sont déjà - développées, notamment celles liées à la virtualisation.

# Table des figures

2.1	Déroulement d'un appel système apres <code>int80h</code> . . . . .	5
2.2	Chronologie et complexification des rootkits sous Linux . . . . .	7
3.1	Implémentation des routines de lecture/écriture en mémoire en espace noyau via <code>/dev/kmem</code> . . . . .	10
3.2	Hook par altération de la <code>syscall_table</code> . . . . .	11
3.3	Désassemblage du noyau dans la zone des appels systèmes issus de <code>int 80h</code> . . . . .	11
3.4	Inline hooking sans altération de la SCT . . . . .	12
3.5	Mise en place du mécanisme de hook via les registres de debug . . . . .	13
3.6	Les différents modes de virtualisation . . . . .	14
3.7	Avant infection par Blue Pill . . . . .	15
3.8	Après infection par Blue Pill . . . . .	15
3.9	Avant/après infection par SubVirt . . . . .	16
4.1	Découpage de nos implémentations . . . . .	18
4.2	Récupération de la fonction d'origine . . . . .	19
4.3	Dump du noyau dans la plage d'adresse relative au debug . . . . .	20
4.4	API dirty hook . . . . .	20
4.5	API permettant de piloter les breakpoints unix . . . . .	21
4.6	Mise en forme d'une adresse en little endian . . . . .	22

# Bibliographie

- [1] Eric Filiol Anthony Desnos. Détection opérationnelle des rootkits hvm ou quand la recherche remplace le buzz. *MISC*, 42, 2009.
- [2] D. Bovet, M. Cesati, and A. Oram. *Understanding the Linux kernel*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2002.
- [3] Eddy Deligne. Rootkit lkm : Enyelkm. 2009.
- [4] Halfdead. Mystifying the debugger for ultimate stealthness. *Phrack*, 61(0x08), 2006.
- [5] J. Kong. *Designing BSD Rootkits*. No Starch Press San Francisco, CA, USA, 2007.
- [6] E. Lacombe, F. Raynal, and V. Nicomette. De l'invisibilité des rootkits : application sous Linux.