# Rootkits pour Windows, analyse et détection

Alain Tauch (tauch_a@epita.fr)

Mai 2002

# Table des matières

# Introduction

> «What is a rootkit ? A rootkit is a set of programs which *PATCH* and *TROJAN* existing exécution paths within the system. This process violates the *INTEGRITY* of the TRUSTED COMPUTING BASE (TCB). In other words, a rootkit is something which inserts backdoors into existing programs, and patches or breaks the existing security system. »
>
> Greg Hoglund, in *Phrack Magazine* [3]

## Historique

Le terme *rootkit* est habituellement associé au monde UNIX, où les privilèges 'root' ou super-utilisateur sont les plus recherchés. Un rootkit désignait alors un ensemble d'outils permettant à un utilisateur passé *root* sur une machine, de revenir plus tard ou de faire tourner ses applications sans qu'elles soient détectées par l'administrateur système. Par extension, on parle également de rootkits pour *Windows*.

On distingue deux types de rootkits :

- *user-mode rootkit* : Ce sont historiquement parlant, les premiers rootkits. Il s'agit essentiellement de versions modifiées de programmes préexistants tels que *ls*, *ps*, *top*, etc..., ainsi que des scripts shells permettant de patcher, déplacer, etc..., certains fichiers.
- *kernel-mode rootkit* : Ce sont des modules kernel qui permettent d'intercepter des appels systèmes, d'en rajouter, ou de modifier les valeurs de retour, de même pour les accès aux différent drivers. Ces derniers rootkits sont beaucoup plus difficiles à détecter du fait qu'ils n'altèrent pas les fichiers du système : ainsi on ne modifie plus *ls* pour cacher certains répertoires, mais on filtre les résultats de l'appel système *getdirentries* par exemple.

## Précisions et limitations

Il n'existe pas, à proprement parler, de *user-mode rootkits* pour Windows. Les programmes tels que *Netbus* [7] ou *Back Orifice* [8] ne sont pas considérés comme étant des rootkits mais plutôt comme des outils d'administration à distance tels que *PC-Anywhere* [1] ou *Windows Terminal Server* [2].
Nous nous intéresserons donc dans ce rapport au *NT Rootkit* [4] de Greg Hoglund, premier *kernel-mode rootkit* pour Windows (NT et 2000).

[1] Symantec PC-Anywhere : http ://www.symantec.com/pcanywhere/
[2] Microsoft Windows Terminal Server : http ://www.microsoft.com

# Chapitre 1

# Analyse

## Avant-propos

Nous nous baserons sur les travaux effectués par Greg Hoglund et son équipe dans le cadre du *NT Rootkit* [4]. Ceci nous permettra de voir le fonctionnement des *kernel-mode rootkits* pour Windows par une approche plus concrète. Nous désignerons ce rootkit spécifique par la dénomination rootkit par la suite.

## 1.1 Considérations Générales

### 1.1.1 Mode Kernel et Sécurité

Les processus utilisateurs, en *userland*, sont dépendants du kernel et interagissent avec celui ci, principalement par le biais des appels systèmes. Lors d'un appel système, il est possible de vérifier si le processus est autorisé ou non à effectuer cet appel système.
Par contre, les processus en mode kernel interagissent entre eux, mais sans aucune vérification, c'est cette faiblesse qui est utilisée par le rootkit. Ainsi, un module kernel n'est jamais soumis à un quelconque contrôle de sécurité lors de son action.

### 1.1.2 Accès aux fichiers

L'accès aux fichiers et aux informations les concernants est dépendant d'appels systèmes.

Prenons le cas d'un IDS[1], celui ci peut effectuer un contrôle de signature (type md5 par exemple) avant d'exécuter un fichier. Cependant, les informations concernant le fichier transitent par le kernel. Ainsi le kernel peut très bien rediriger la lecture d'un fichier vers le fichier original et l'exécution vers un fichier

[1] IDS : Intrusion Detection System

différent, sans que l'IDS ne s'en apperçoive. Ceci est l'une des fonctionnalités du rootkit étudié.

### 1.1.3 Accès au réseau

Ici également, le kernel est l'interlocuteur privilégié de l'interface avec le réseau : tout ce qui transite par cette interface passe par le kernel.
Ainsi il est possible au rootkit de dialoguer directement avec l'interface réseau, et ce, sans passer par la pile TCP/IP.

## 1.2 Pré-requis technique

### 1.2.1 Drivers et *device objects*

Chaque driver possède un point d'entrée (*DriverEntry()*), correspondant au *main()* en C.

Chaque driver peut créer ce que l'on appelle des *device objects* (DO).
par exemple : sur une machine possédant plusieurs disques durs, il y aura un driver de disque et autant de DO que de disques.
On retrouve également des DO pour la pile TCP/IP, le système de fichier NTFS, des systèmes de cryptage, etc...

Un driver implémente également une fonction de *callback* appelée *dispatch routine* commune à tous ses DO.

Les commandes d'entrées/sorties se font à travers des paquets, ce sont les IRP[2]. Ceux ci sont échangés entre les différents DO car plusieurs DO peuvent être impliqués lors d'un même évènement. On parle de drivers *mono-couches* quand les IRP ne communiquent qu'entre ses DO et de drivers *multi-couches* quand les IRP transitent entre des DO de drivers différents. Lorsqu'un IRP arrive à un DO, celui-ci appelle sa fonction de *callback* qui traite alors les informations apportées par l'IRP, celui ci est ensuite renvoyé au DO de la couche inférieure.
On voit ici qu'il est possible de *sniffer* et/ou d'altérer les données en transit grâce à cette fonction de *callback*.

C'est cette méthode qui est utilisée par le rootkit pour *sniffer* les données tapées sur le clavier par exemple, mais cela peut très bien s'appliquer à la souris, l'imprimante, le modem, etc...

Les DO sont visibles depuis l'espace utilisateur et cela peut être un inconvénient. Cependant, rien n'empêche de cacher la présence de certains DO en interceptant les appels systèmes appropriés.

[2]IRP : I/O Request Packet

## 1.3 Modification des appels systèmes

A chaque appel système est associé un unique numéro.
Il y a dans le kernel NT, ce que l'on appelle le *SDT*[3] qui est un tableau de correspondance entre ces numéros et les fonctions associées (*syscall routines*). A chaque entrée du tableau correspond donc un numéro d'appel système et un pointeur sur la fonction associée dans l'espace kernel.

Il est possible de changer la valeur des pointeurs de la SDT et donc de rediriger un appel système vers une fonction du rootkit. Cette fonction appelle ensuite la fonction originelle. C'est en quelque sorte du *man-in-the-middle*. La fonction du rootkit peut alors aussi bien modifier les paramétres fournis en entrée à l'appel système que les valeurs de retour.

### 1.3.1 Cacher fichiers et répertoires

Il faut pour cacher des fichiers ou répertoires intercepter l'appel système *ZwQueryDirectoryFile*.
Ici, nul besoin de changer les paramètres d'entrée, il suffit de modifier les valeurs de retour :

Cet appel système renvoie une liste de structures correspondants aux entrées présentes dans le répertoire [4]. Chaque structure pointant sur la suivante, il est aisé de supprimer une entrées en faisant pointer la structure *n*, non pas sur la structure *n+1* mais la structure *n+2* (*snipping technique*).

### 1.3.2 Cacher processus, threads, drivers

La méthode utilisée est exactement la même que celle utilisée précédemment (*snipping technique*), seul l'appel système change : il s'agit ici de *ZwQuerySystemInformation*

## 1.4 Modification des interruptions logicielles

Comme pour les appels systèmes et la SDT, on a ici une IDT[5] qui fait la correspondance entre les numéros des interruptions logicielles et leurs fonctions associées.

Pour modifier l'IDT la marche à suivre est la suivante :

- On envoie l'instruction SIDT[6] au processeur (ne peut se faire que lorsque l'on est en mode kernel). Ceci nous permet de récupérer l'adresse de l'IDT.

[3] SDT : Service Descriptor Table
[4] Cette procédure est similaire à l'appel système *getdirentries* sous BSD qui permet de recupérer une liste de structures *dirent*.
[5] IDT : Interrupt Descriptor Table.
[6] SIDT : Store Interrupt Descriptor Table.

- On envoie ensuite l'instruction CLI[7] au processeur pour désactiver les interruptions.
- On change alors les entrées de l'IDT voulues. On retrouve leur offset à l'aide de l'adresse récupérée précédemment.
- On réactive ensuite les interruptions par le biais de l'instruction STI[8].

On peut ainsi intercepter des interruptions logicielles de la même manière qu'avec les appels systèmes mais également rajouter des interruptions logicielles (il y a des interruptions non utilisées), pour dialoguer avec le rootkit par exemple.

## 1.5 Modification des interruptions matérielles

Cela est également possible, et la méthode est comparable à celle utilisée pour les interruptions logicielles.
Cependant, l'utilisation que l'on peut en faire dans le cadre d'un rootkit reste marginale car spécifique au matériel présent, et donc cette fonctionnalité n'a pas été implémentée dans le rootkit étudié.

## 1.6 *Sniffer* Réseau

Le rootkit étudié implémente un sniffer dans l'espace kernel.
Il existe une librairie dans l'espace kernel comparable à *winsocks* qui permet une utilisation aisée des interfaces réseau. Ainsi le code utilisé est complétement indépendant du type d'interface.

La mise en place de ce *sniffer* dans l'espace kernel fait appel aux fonctions[9] :

- NDISRegisterProtocol() : pour utiliser un protocole donné.
- NDISOpenAdapter() : pour utiliser une interface donnée.
- NDISRequest() : pour passer l'interface en mode promiscuous.
- On enregistre ensuite une *callback routine* destinée à la réception de tous les paquets correspondants.

Cette fonctionnalité peut être utilisée pour détecter des paquets particuliers et impliquer une action correspondante du rootkit.
Elle permet également d'utiliser des techniques de spoofing, car on est dans une situation où il est possible de recevoir (interface en mode promiscuous) et d'émettre des *raw packets* sur le réseau.

[7] CLI : Clear Interrupt bit.
[8] STI : SeT Interrupt bit.
[9] se référer au MSDN[9] pour plus de précisions

## 1.7 Insérer du code exécutable dans le kernel

Lors du boot les drivers sont chargés dans un ordre particulier dans le kernel. Cependant, il est possible de charger certains drivers par la suite. Il existe plusieurs façons de charger du code exécutable dans le kernel, et ce, de manière dynamique (sans rebooter) :

- L'appel système *ZwLoadDriver* permet de charger dynamiquement un driver dans le kernel.
- NtKernLoadDriver également.
- On peut utiliser des buffers overflows dans certains drivers : Le nombre de drivers tels que ceux pour imprimantes étant très important, il est fort probable de pouvoir trouver des buffer overflows exploitables. Cependant il faut être très vigilant car ce genre de manipulation risque de compromettre la stabilité du système. Les drivers compromis risquent fort d'avoir des fonctions de callback, certains devices risquent d'être soumis à des IRP, etc...Ainsi cela nécessite un grand travail de préparation pour pouvoir exploiter ces failles proprement et sans éveiller la méfiance de l'administrateur système.
- Cela peut passer par l'*infection* de drivers préexistants.
- Il est possible d'ècrire dans */dev/kmem* via le device */dev/PhysicalMemory*.
- Cela peut passer par l'utilisation d'un appel système utilisant *SystemLoadAndCallImage*.
- Il est possible d'insérer du code dans les sections non utilisées des *PE files* comme le font certains virus. Ou alors y insérer un appel à *LoadImage()* qui se chargera de charger le code approprié par la suite.
- On peut patcher le fichier kernel (sur le disque), cependant cette technique nécessite de patcher *NTLoader* également car ce dernier effectue des contrôles sur le fichier kernel avant de le charger au démarrage.
- etc...

Il existe sûrement beaucoup d'autres possibilités du fait que le kernel Windows n'est pas entièrement documenté, ainsi de nombreux devices ou fonctions même restent à découvrir[10]. Le rootkit utilise actuellement l'appel système *ZwSetSystemInformation* qui fait appel à *SystemLoadAndCallImage* pour se charger dans l'espace kernel.

## 1.8 Retirer le code du kernel

Il n'est pas évident de retirer des drivers de manière dynamique sans provoquer de BSOD[11] et entrainer l'arrêt du système et ce, à cause des routines de

[10] On pourra se tourner vers *Reversing NBTSTAT* [10][11] qui est l'un des exemple de reverse-engeenering des IOCTL de windows NT

[11] BSOD : Blue Screen of Death

*callback*. Si jamais le module est retiré et qu'une de ces routines est appelée, cela provoquera une erreur lors de l'accès à celle ci (*page fault*) et donc provoquera un BSOD. Sachant qu'un sniffer est implémenté au niveau kernel, imaginez le nombre de BSOD successifs provoqué par la routine de *callback* associée.

Pour permettre cette manipulation sans compromettre la stabilité du système il faut :

- Rechercher les IRP[12] et attendre la fin de celles-ci.
- Supprimer les *devices* créés et les routines associées.

Ce travail, délicat ne peut se faire sans la connaissance approfondie du module visé. Ainsi il n'est pas envisageable de retirer dynamiquement le module sans utiliser les outils développés à cet usage par les concepteurs du module.

Cette fonctionnalité est présente dans le rootkit étudié.

## 1.9 Cacher le rootkit

Trois principales précautions ont été prises pour cacher le rootkit.

### 1.9.1 L'appel système ZwQuerySystemInformation

C'est l'appel système *ZwQuerySystemInformation* qui permet la détection des nouveaux modules.

Ici, comme vu précédemment, il suffira d'intercepter l'appel système pour cacher le rootkit.
On peut très bien imaginer une requête qui analyse le buffer complet à la recherche de structures manquantes (*snipped entries*) afin de détecter la présence du rootkit. Cependant, le rootkit peut également implémenter une manière plus intelligente de modifier ledit buffer, en opérant un décalage des structures par exemple. On voit bien ici, que cela est sans fin et laisse libre court à des implémentation multiples, au gré du niveau de paranoïa des codeurs.

### 1.9.2 Lecture de la mémoire

Le rootkit peut être détecté par sa présence en mémoire : un utilisateur peut lire la mémoire à la recherche d'anomalie en utilisant les devices */dev/kmem* et */dev/PhysicalMemory*.

Pour éviter ces problèmes, une exception (*page fault*) est renvoyé lorsque l'on tente de lire sur une page mémoire allouée au rootkit.

[12] IRP : I/O Request Packet

### 1.9.3 Accès au fichiers systèmes

Les accès du rootkit aux fichiers sytèmes et à ses propres fichiers, tels _root_.sys peuvent être décelés par l'analyse des accès au système de fichier.

Comme nous l'avons vu précédemment, il est aisé de cacher l'accès à certains fichiers ou encore de rediriger les accès à ces fichiers en interceptant les appels systèmes correspondants.

# Chapitre 2

# Prévention et Détection

## Avant-propos

Il ne saurait exister de méthode globale de détection des rootkits, chaque rootkit ayant un mode d'action et une implémentation différents. Il est cependant possible de dégager quelques lignes directrices mais nous parlerons plus de prévention que de détection pour les cas généraux.

Dans un deuxième temps nous verrons quelques méthodes de détection du rootkit étudié. Ces méthodes ne sauraient être générales et reposent sur les choix d'implémentation du rootkit. Il va de soi que la plupart de ces failles d'implémentation sauraient être corrigées rapidement.

## 2.1 Cas général

Il existe de nombreuses méthodes de préventions et de détection des rootkits en *kernel-mode*. La liste des méthodes décrites dans les paragraphes suivants ne saurait donc être exhaustive.

### 2.1.1 Test d'integrité

Une méthode permettant d'empêcher l'installation de rootkits dans l'espace kernel est d'utiliser un système de signature des modules. Ainsi, seuls les modules identifiés pourront s'installer.

### 2.1.2 Contrôle par clé

Ici on distingue deux types de méthodes :

- logicielle : protection par mot de passe, nécessitant l'intervention de l'administrateur.
- matérielle : utilisation d'un *dongle USB* ou de *smartcards*.

### 2.1.3 Scanner

Deux types de scanners sont envisageables :

- *user space* : scan de la mèmoire en utilisant */dev/PhysicalMemory*.
- *kernel space* : scan de l'espace kernel.

Dans ces deux cas, les scanners sont soumis aux mêmes contraintes que tout antivirus, à savoir : nécessité d'une base de connaissances (signatures) et reconnaissance de code polymorphe ou crypté.

### 2.1.4 Audit

Il est pertinent de logguer :

- les appels à la fonction *LoadModule()*.
- les modifications de la SDT.
- les modifications de la table des fonctions de NTOSKRNL et NTDLL.

## 2.2 Cas particulier

### 2.2.1 Remarques préliminaires

Cet exemple se base sur la version 0.40 du *NT rootkit* de Greg Hoglund en environnement Windows 2000. Ce rootkit est l'oeuvre d'une équipe de développeurs réunie autour du projet de Greg Hoglund[3]. La version de ce rootkit n'est pas une version finale et est essentiellement didactique, elle offre de nombreuses possibilités aux développeurs en terme d'évolution et de personnalisation.

### 2.2.2 Description fonctionnelle

Le rootkit permet de :

- cacher des processus.
- cacher fichiers et répertoires.
- cacher des clés du registre.
- logguer les combinaisons de touches du clavier.
- provoquer un BSOD.
- rediriger l'exécution de fichiers.
- se connecter à distance sur une ip spoofée à la console d'administration du rootkit.

De plus, il peut se charger et se retirer du kernel de facon dynamique.

### 2.2.3 Méthodes de détection

#### Fichiers et répertoires cachés

Le rootkit cache tous les fichiers et répertoires commencant par _root_.
Ceux ci sont cachés en détournant les appels systèmes permettant d'afficher fichiers et répertoires. Ainsi, quel que soit l'outil utilisé pour visualiser les fichiers, ceux-ci seront toujours absents.
De plus, le code source est facilement adaptable pour accueillir un préfixe variable.

Cependant, il possible de voir les fichiers cachés par le biais du voisinage réseau, en ce connectant sur le partage c$ par exemple. Seuls les partages cachés peuvent révéler la présence de ces fichiers.

#### Processus

De même que pour les fichiers et répertoires, les processus cachés ne peuvent être révélés.

#### *Backdoor*

La connexion sur la console d'administration via une ip spoofée n'est pas détectable.
Seule l'adresse mac de :ad :be :ef :de :ad peut sembler bizarre.
Un NIDS[1] pourrait détecter la présence d'une nouvelle machine sur le réseau, lors d'une connexion.

#### Module kernel/driver

Le driver _root_ apparaît clairement danns la liste des pilotes utilisés par le noyau. La détection est ici flagrante.

#### Base de registre

Certaines clés nécessaires au rootkit sont détectables avec des outils tes *regfind.exe* mais pas *regedit*.

#### Antivirus

L'intégralité des tests a été effectué alors qu'un antivirus tournait sur la machine, ce dernier n'a rien détecté de suspicieux.

[1] Network Intrusion Detection System

### 2.2.4 Bilan

Nous voyons ici, que la détection du rootkit relève du cas particulier. Concernant la base de registre et le module kernel qui apparait dans la liste des pilotes utilisés, cela semble avoir été patché dans les versions ultérieures. Cependant il n'a pas été possible de tester les versions 0.42, 0.43 et 0.44 pour des problèmes de compilation et/ou de stabilité.

# Chapitre 3

# Conclusion

En conclusion, nous pouvons affirmer que les rootkits pour windows de type *kernel-mode* potentialisent un réel danger. Bien que le stade de développement du rootkit étudié ne soit pas encore très avancé, il offre déjà un certain nombre d'outils performants et amène toutes les bases pour la construction d'un rootkit fiable et complet. La dernière version du rootkit (0.44) permet d'avoir un shell complet en mode kernel, auquel on se connecte sur une ip spoofée, autant dire que l'avenir semble prometteur ...

Enfin ce rootkit est aussi l'occasion de faire le point sur la sécurité des systèmes, et de remettre en cause les politiques en terme de détection d'intrusion.

# Bibliographie


[1] BlackHat.com, *BlackHat Media Archives*, www.blackhat.com.

[2] Greg Hoglund, *Kernel Modules Rootkits : Stealth and subversion of trust*, BlackHat Windows security 2001, Las Vegas Feb 14 & 15.

[3] Greg Hoglund, *A *REAL* NT Rootkit, patching the NT Kernel*, Phrack Magazine 55, Sept 1999.

[4] [ www.rootkit.com ] CVS REPOSITORY, *Rootkit.com, the NT Rootkit project homepage*, www.rootkit.com.

[5] CNET, *Hackers' rootkit for NT*, builder.cnet.com.

[6] CNET, *Stop windows hackers*, builder.cnet.com.

[7] Carl-Fredrik Neikter, *Netbus*, fly.to/netbus.

[8] Cult of the Dead Cow, *Back Orifice*, www.cultdeadcow.com.

[9] MSDN Library, *Microsoft Software Developers Network Library*, msdn.microsoft.com/library/.

[10] Greg Hoglund, *Reversing NBSTAT - nstat.c*, www.rootkit.com.

[11] Greg Hoglund, *Reversing NBSTAT - nstat.h*, www.rootkit.com.

[12] Forensic Tools, *Vision v1.0*, www.foundstone.com/knowledge/forensics.html.