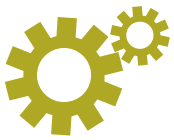


Fiche technique

Développement avancé d'un rootkit pour les modules du noyau de

Pablo Fernández 

Degré de difficulté

L'installation d'un rootkit est la phase qui permet de transformer un ordinateur « propriétaire » en ordinateur compromis. Nous allons, expliquer comment développer un rootkit sur la série 2.6 des modules du noyau de Linux. Le premier objectif consistera à présenter les techniques et les méthodes permettant de dissimuler les actions malveillantes, puis nous évoquerons les possibles détections de rootkits sur un ordinateur propriétaire.

Connaître le fonctionnement interne des rootkits se révèle très avantageux selon différents points de vue ; un pirate ne peut vraiment gérer un système qu'à partir du moment où il a trouvé le moyen adéquat de contrôler ce système intégralement. C'est la raison pour laquelle les administrateurs de système doivent connaître leur fonctionnement afin d'être capables de reconnaître un système compromis.

L'objectif du présent article vise à présenter les techniques les plus importantes utilisées par un rootkit concret et extensible, connu sous le nom de SIDE, et fonctionnant pour les modules du noyau de Linux, version 2.6. Des fonctionnalités complémentaires seront ajoutées au rootkit lors dans le cadre des prochains articles.

Dissimuler le module

Dans la mesure où le rootkit fonctionnera au sein même du système sous forme de module central, il faut bien veiller à ce que ce dernier reste dissimulé au moyen de commandes telles que *lsmod* ou via */proc/modules*. C'est ce dont se charge le code exposé dans le Listing 1. Afin de bien en comprendre le fonctionnement, il faut tout d'abord étudier l'ordre des modules dans le noyau ainsi que les principes sous-jacents à cette technique de dissimulation (voir la sous-partie intitulée *Modules*).

En règle générale, grâce à ce code, le module se détache de la liste interne double chaînée cyclique des modules chargés dans le noyau.

Cet article explique...

- Comment le système d'exploitation interagit avec les programmes de l'espace utilisateur.
- Ce que le système appelle, et comment trouver la table des appels du système.
- Comment dissimuler les modules, les processus, les connexions réseau et les fichiers.
- Comment accorder des permissions root à des utilisateurs normaux à partir du noyau.

Ce qu'il faut savoir...

- Programmation en C.
- Connaître Linux.
- Connaître les concepts de tâches, de fichiers, etc.

Dissimuler les processus

Pouvoir dissimuler les processus de chaque utilisateur du système (y compris du root) est l'une des fonctionnalités élémentaires qu'un rootkit est censé implémenter.

Les outils de l'espace utilisateur (tels que *ps(1)* ou *top(1)*) se tiennent informés des tâches (ou processus) en lisant le répertoire */proc*. Chaque tâche lancée sous le système crée une entrée sous la forme */proc/<PID>*, dans laquelle il est possible d'obtenir des informations utiles sur le processus en question. Le travail des outils de l'espace utilisateur consiste à ouvrir ce répertoire */proc* pour demander l'existence de */proc/<n>*, où 1 <= n <= *pid_max*. Si le répertoire n'existe pas, l'identificateur de processus (PID) est censé être libre. À l'inverse, si le répertoire existe bien, il est possible d'y intercepter les informations.

Grâce à ce principe et en connaissant le fonctionnement interne du système de fichiers virtuels (voir la sous-partie intitulée *Fonctionnement interne du système de fichiers virtuels*), il est possible de faire croire aux outils de l'espace utilisateur que les identificateurs de paramètres existants sont libres. Il suffit d'interrompre l'appel *readdir* dans la couche du système de fichiers virtuels. Pour ce faire, il faut modifier la table contenant l'adresse de l'appel *readdir* avec l'adresse du nouveau segment de code chargé de réimplémenter cette fonction. En effet, interrompre l'appel *readdir* permet de modifier l'argument intitulé *filldir* de manière à le faire pointer vers une implémentation différente de *filldir*, chargée de supprimer les répertoires capables d'identifier les identificateurs de processus dissimulés.

Listing 1. *Dissimuler le module*

```
lock_kernel(); /* Held the kernel lock to prevent faulting in SMP systems */
__this_module.list.prev->next = __this_module.list.next;
__this_module.list.next->prev = __this_module.list.prev;
__this_module.list.prev = LIST_POISON1; /* A common practice in kernel
                          development */
__this_module.list.next = LIST_POISON2; /* to invalidate a list that
                          shouldn't be used */
unlock_kernel();
```

Listing 2. *Extrait de la réimplémentation de l'argument filldir dans /proc*

```
if (!(process = _atoi(name, &process))); /* If this isn't a PID just call the
                          original filldir */
else if (!process_is_authed(current) && process_is_hidden(process)) /* If
                          process is hidden */
     return 0;                                        /* don't show it
                          (unless current is superroot) */

if (p_proc_filldir)
     return p_proc_filldir(buf, name, nlen, off, ino, x);
```

Détecteurs de rootkits

Les détecteurs de rootkits font appel à une technique leur permettant de trouver des processus dissimulés, en envoyant un signal *SIGCONT* vers tous les processus possibles.

Ces signaux sont envoyés aux processus au moyen de l'appel du système *kill(2)*. Lorsqu'un signal est envoyé vers un processus qui n'existe pas, *kill(2)* retourne la valeur -1 et *errno* est paramétré sur *ESRCH*. Si le processus existe bien, *kill(2)* retourne la valeur 0.

Ainsi, les détecteurs de rootkits peuvent déterminer l'existence d'un processus sans avoir recours aux données contenues dans */proc*. Une fois la manœvre achevée, la liste créée est comparée à la liste des processus affichés dans */proc*. La présence d'une différence entre les deux listes indique un processus dissimulé de l'espace utilisateur.

Côté rootkit, il est possible d'induire en erreur un détecteur de rootkit reposant sur cette technique en étendant tout simplement le

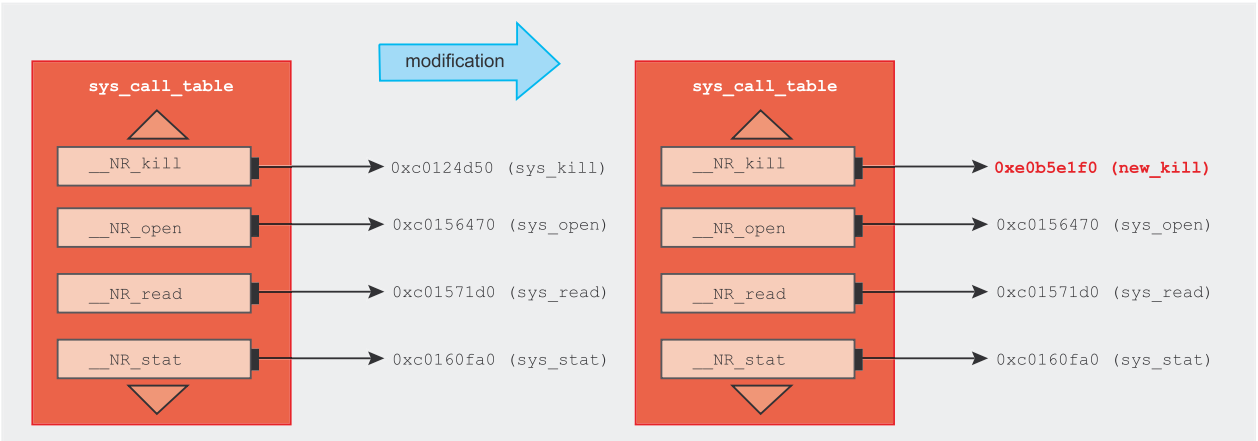


Figure 1. *Modification de la table sys_call_table*

Listing 3. *Remplacement de sys_kill()*

```
asmlinkage int new_kill(pid_t pid, int sig)
{
     struct siginfo info = { .si_signo = sig,
          .si_errno = 0, .si_code = SI_USER,
          .si_pid = current->tgid, .si_uid = current->uid };
     if (!process_is_hidden(pid) || process_is_authed(current))
          return kill_proc(pid, sig, &info);
     return -ESRCH;
}
```

code. Il suffit que le rootkit intercepte l'appel du système *kill(2)* (voir la sous-partie intitulés *Les appels système*). Si un signal est envoyé par quelqu'un d'autre que le super utilisateur (voir la sous-partie intitulée *Super utilisateur*) vers un processus dissimulé, la nouvelle fonction de rappel est censée retourner *ESRCH*. Toutefois, si tel est le cas, la fonction originale *kill(2)* est censée être appelée ainsi que la valeur de retour retournée.

Complément : technique du détecteur de rootkits

Un détecteur de rootkits dispose de plusieurs techniques pour identifier un rootkit. Dans la mesure où les outils de l'espace utilisateur sont facilement vulnérables, nul besoin d'y maintenir des détecteurs de rootkits. Une technique très simple pour détecter un rootkit consiste à contrôler les modifications de *sys_call_table* une fois dans l'espace du noyau. La détection des processus cachés est d'autant plus facilitée que ces derniers ne peuvent être détachés de la liste des processus aussi facilement que les modules. Leur présence dans une telle liste est donc la seule preuve de tranches de temps d'exécution.

Alors que certains détecteurs de rootkits ont recours à quelques unes de ces techniques, la plupart d'entre eux se cantonnent à l'espace utilisateur. Les administrateurs un peu trop paranoïaques ne doivent pas avoir une confiance absolue dans les détecteurs de rootkits.

Dissimuler des connexions réseau

Les programmes et les applications de l'espace utilisateur sont tenus informés du trafic réseau en cours grâce aux entrées */proc/net*. Cet emplacement contient plusieurs entrées (semblables à des fichiers, mais qui sont en réalité des structures de type *proc_dir_entry*), comme *tcp* et *tcp6* (si CONFIG_IPV6 est activé). Ces entrées contiennent des informations sur l'état du réseau en marche sur le système.

Grâce à une méthode de lecture différente, il est également possible d'intercepter des appels. Les informations retournées peuvent donc être modifiées pendant leur transit. Lorsque la connexion au réseau est toujours activée (ou lorsqu'une interface de connexion se trouve en mode *LISTEN*), *netstat* et les outils du même type ne pourront pas la voir.

Le rootkit SIDE propose une méthode très intéressante pour dissimuler les connexions réseau, assez semblable (quoique pas aussi puissante) à Netfilter. Une liste de conditions et de commandes est définie pendant la durée d'exécution (voir le Tableau 1). Ainsi, lorsque des informations sur les interfaces de connexion sont requises, la liste est alors comparée avec chaque interface de connexion. Si une certaine règle doit s'appliquer, la commande associée à cette condition est alors exécutée. Cette commande peut permettre, soit de *montrer* soit de *cacher* l'interface de connexion dans l'espace utilisateur. Cette liste est assez puissante. Il est possible de définir des actions par défaut grâce à la condition *all* à la fin de la liste, entièrement manipulable pendant la durée d'exécution (elle peut même exécuter des commandes par défaut lorsque le module est chargé).

La méthode utilisée pour dissimuler les connexions réseau consiste à mettre la main sur le *proc_dir_entry* du protocole à intercepter, tel que *tcp*. *proc_dir_entry* appartenant à l'espace */proc/net* et détectable grâce à la liste double chaînée cyclique dans *proc_net->subdir*. En répétant la manœuvre, cette méthode permet de contrôler le nom correct dans le membre *node->name*.

Une fois cette structure contrôlée, le pointeur *seq_show* doit être remplacé par une nouvelle implémentation de cette fonction. L'implémentation du rootkit SIDE se charge d'intercepter les données

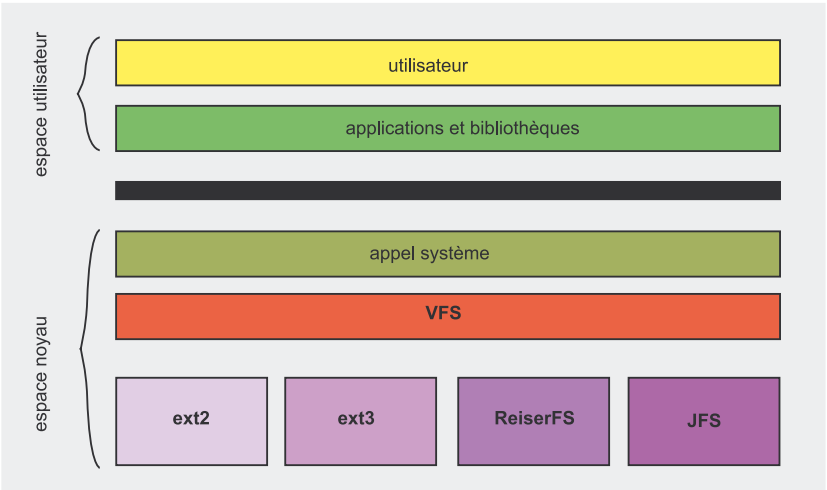


Figure 2. *Couche du système de fichiers virtuels*

correctes (au moyen de la fonction originale), puis de comparer chaque ligne de données avec les règles chargées, en appliquant des actions spécifiques dans les lignes correspondantes.

Problématique

En raison de la nature du trafic du réseau, il est quasiment impossible de dissimuler l'activité à un administrateur avisé. Il existe généralement plusieurs étapes entre le système compromis et l'autre destination d'une communication dissimulée. Ces étapes afficheront clairement ce trafic dissimulé, et il n'y a malheureusement rien à faire.

En réalité, même lorsqu'une connexion n'est pas encore établie, si une interface de connexion en mode *LISTEN* est dissimulée, *nmap* alertera l'administrateur sur la présence d'un port ouvert ignoré par *netstat*. Ce problème peut certainement être évité en ayant recours à la technique du Port Knocking (voir le numéro hakin9 du mois de juin 2005), mais une fois la connexion ouverte et le trafic lancé, il sera facile de détecter ce trafic (*Removing spiderwebs – detection of illegal connection sharing*, voir le numéro hakin9 du mois de mars 2005).

Une solution simple consisterait à n'établir aucune connexion, en échangeant des paquets d'informations dans ICMP ou des paquets UDP, ce qui permettrait de contrôler de manière intéressante un système et d'obtenir des informations sur son état. De cette façon, le pirate peut même contrôler le système compromis sans laisser aucune trace, en utilisant des paquets UDP ou ICMP piégés.

Le rootkit que nous présentons ici sera plus amplement expliqué dans les prochains articles qui présenteront ses fonctionnalités, y compris celles mentionnées dans le cadre du présent article.

Dissimuler les fichiers

Un système est souvent compromis s'il est utilisé en tant que plate forme de sécurité à partir de laquelle il est possible de lancer des attaques de type [D]DoS, ou s'il sert d'étape entre le pirate et un autre système compromis. La plupart du temps, le pirate aura sans doute besoin de charger certains fichiers vers le système afin de lancer ses attaques. Bien sûr, si l'administrateur vient à détecter de tels outils, il aura des doutes. Un rootkit est donc censé être toujours capable de dissimuler des fichiers dans le système à n'importe quel utilisateur, y compris au root.

Nous allons encore faire appel à nos connaissances du système de fichiers virtuels afin de réaliser de telles actions, en utilisant la même méthode permettant de dissimuler les processus.

Dans ce cas, l'objet *fs* va stocker une liste de noms de fichiers dissimulés. Comme cette liste est dépourvue de chemin, n'importe quel fichier dissimulé dans un répertoire exigera de cacher chaque fichier du même nom dans l'ensemble des répertoires. En effet, il est nécessaire de forcer le super utilisateur à utiliser des noms non-standardisés dans la mesure où il existe toujours d'autres méthodes non-gérées. Même si les fichiers dissimulés ne sont pas listés dans les répertoires, par exemple, ils demeurent toujours accessibles via les appels système tels que *open(2)*, *stat(2)*, etc. Le rootkit SIDE ne supporte pas, à l'heure actuelle, les fichiers dissimulés à partir de ces méthodes, bien qu'il suffise de remplacer ces appels système (et d'autres appels tels que *rename(2)*). Suggestion : il serait judicieux d'étendre ces fonctionnalités lorsque vous aurez lu le présent article.

Vous aurez sans doute remarqué que la méthodologie utilisée dépend du système de fichiers. Si vous désirez dissimuler des fichiers dans des points de montage différents, SIDE doit être modifié dans le fichier intitulé *vfs.c* afin de le cacher à ces points de montage. Il est plus sûr d'utiliser les mêmes *readdir* et *filldir* que le système de fichiers root.

Problématique

Encore une fois, cacher les fichiers soulève également des problèmes spécifiques, assez semblables aux problèmes rencontrés avec les connexions réseau.

Les fichiers sont stockés dans des disques accessibles de différentes manières, comme un CD-ROM de sauvegarde où l'utilisateur monte la partition du root. Comme le rootkit n'est pas chargé dans le noyau en pleine exécution, les fichiers dissimulés ne sont plus cachés. Différentes techniques, permettant de compliquer légèrement la détection de ces fichiers cachés, sont toutefois disponibles. La protection la plus simple et la plus fréquente demeure la sécurité par obscurité, autrement dit, stocker les fichiers dans des emplacements non-conventionnels à l'aide de noms non-descriptifs et assez confus.

Une approche bien meilleure consiste à stocker l'ensemble des fichiers dans un système de fichiers avec boucle de retour. Bien entendu, cette technique doit être surveillée de près de sorte à ne pas voir s'afficher un tel système de fichiers monté avec *mount(8)* ni via */proc/mounts*. Cette technique permet également de crypter relativement facilement le système de fichiers. Même si l'administrateur est suspicieux, il n'aura jamais l'idée de consulter le contenu du système de fichiers.

À propos de l'auteur

Originaire de Temperley, en Argentine, âgé de 21 ans, Pablo Fernández travaille comme développeur. Fort d'une expérience de plus de 6 ans en programmation GNU/Linux, et de 4 ans dans le domaine de la sécurité, Pablo a contribué au développement de nombreux logiciels libres. Il est l'auteur de GNOME mail client Cronos II, proxychain, et a participé à des projets comme Nmap (créateur de la toute dernière méthode de scan de serveur mandataire furtif), entre autres.

Sur Internet

- *http://www.littleQ.net/SIDE/* – page d'accueil du rootkit SIDE.

Utilisateur normal doté de permissions root

Le super utilisateur est identifié par un UID et un GID particulier différent de zéro. Ainsi, le super utilisateur est dépourvu de permission root, ce qui est tout à fait inacceptable. C'est la raison pour laquelle le rootkit SIDE implémente un mécanisme chargé d'établir un UID de 0 sur chaque processus exécuté par le super utilisateur.

C'est également ce qui est implémenté dans l'appel interrompu vers la fonction *lookup()* dans le répertoire */proc*. À chaque fois qu'un processus authentifié (voir la sous-partie intitulée *Super utilisateur*) accède à un quelconque élément dans ce répertoire, son UID ainsi que d'autres valeurs qui lui sont propres sont réglées sur 0 (root), et certaines fonctionnalités sont complètement activées (*cap_effective*, *cap_inheritable* et *cap_permitted*). Si le processus n'accède à aucun élément dans le répertoire */proc*, il sera exécuté au moyen de l'UID du super utilisateur. C'est la raison pour laquelle certains programmes extrêmement simples et de taille modeste ne s'identifieront pas eux-mêmes en mode root, comme par exemple la commande *whoami*.

Listing 4. *Recherche de la table sys_call_table*

```
unsigned long ptr;
extern int loops_per_jiffy;

for (ptr = (unsigned long) &loops_per_jiffy; ptr < (unsigned long)&boot_cpu_
                    data; ptr += sizeof(void *)) {
     unsigned long *p;
     p = (unsigned long *)ptr;
     if (p[__NR_close] == (u32) sys_close) /* When this condition is met p
                    points to sys_call_table */
          return  (u32 **) p;
}
```

Listing 5. *Interception d'un appel système*

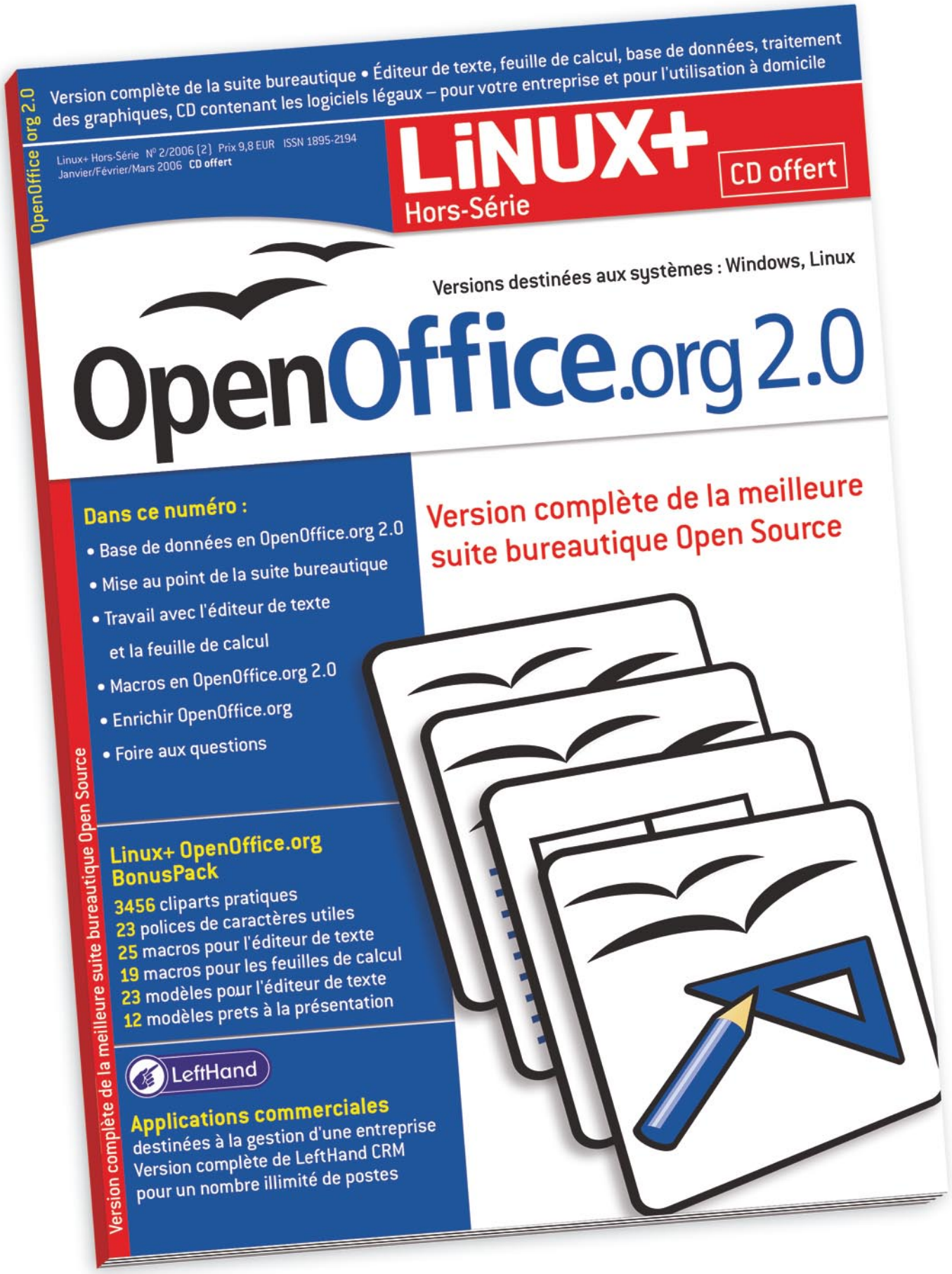
```
u32 **sys_call_table;
asmlinkage int (*old_open)(const char *, int, mode_t);

if ((sys_call_table = find_sys_call_table())) {
     old_open = (void*) sys_call_table[__NR_OPEN];
     sys_call_table[__NR_OPEN] = (u32*) new_open;
}
```

Exécution facilitée

SIDE propose une interface très conviviale pendant la durée d'exécution. Dans *vfs.c*, le rootkit se charge d'intercepter les appels *lookup()* dans le système de fichiers */proc*. De cette manière, le super utilisateur (voir la sous-partie intitulée *Super utilisateur*) peut interagir avec le rootkit. Pour ce faire, il suffit, par exemple, de dissimuler un processus ou bien d'accorder des permissions root (ou super utilisateur) à certains utilisateurs. Il est relativement facile d'envoyer des commandes au rootkit. Il suffit pour cela que tous les utilisateurs tentent d'accéder à un fichier dans le système de fichiers */proc*. Le nom du fichier sera interprété comme une commande.

Disponible sur : shop.software.com.pl/fr



www.lpmagazine.org/fr

Le rootkit SIDE organise les commandes par objets. Ainsi, selon ce que l'utilisateur souhaite faire, les commandes sont exécutées sur certains objets spécifiques.

Pour le moment, SIDE peut réorganiser trois objets différents : *net*, afin de manipuler la liste du réseau, *sys*, afin de gérer le processus ainsi que les propriétés utilisateur qui lui sont affectées, et *fs*, afin de manipuler la liste des fichiers dissimulés.

Il existe plusieurs commandes pour ces objets. Par souci de concision, nous en avons exposé quelques unes dans le Tableau 1. Les autres commandes sont disponibles dans le fichier intitulé COMMANDS.txt fourni avec le package.

Les commandes doivent être exécutées de la manière suivante :

```
echo > /proc/[object].[command[=args]]
```

Modules

Les modules chargés sont stockés au sein du noyau dans une liste double chaînée cyclique, dans laquelle chaque nœud de la liste représente un *struct module* (défini dans *include/module.h*). Il est assez facile de rassembler les modules en lisant l'entrée */proc/modules*. La liste des modules est créée par *m_show*, dans *kernel/module.c*, en traversant la liste nommée et liée précédemment.

Dans la mesure où cette liste est accessible à partir de chaque module, il est alors possible de la modifier ou de la manipuler. Cette technique, permettant de dissimuler le module, consiste à détacher le nœud du module de la liste liée, en connectant directement la valeur précédente à la valeur suivante au sein de la liste.

Mariusz Burdach a consacré un article très intéressant à cette technique dans la magazine Hakin9 de mars 2005.

Appel système

Est désignée par appel système l'interface située dans le noyau, chargée de faire communiquer l'espace utilisateur avec le noyau. Tous les échanges du noyau vers l'utilisateur ou vice versa doivent passer par un appel système. C'est la principale raison pour laquelle les rootkits se sont toujours intéressés à leur interception. En effet, contrôler ces appels système permet de contrôler ce que voit et ce que peut faire l'utilisateur.

Il existe de nombreux appels système, comme *open(2)*, *read(2)*, etc. Toutes ces fonctions sont référencées par des pointeurs dans un tableau appelé table des appels système, mieux connue sous le nom de *sys_call_table*.

Avant, modifier la table *sys_call_table* consistait à changer le pointeur désiré au moyen d'une nouvelle adresse de mémoire dotée d'une nouvelle fonction. De cette manière, il devenait très facile d'intercepter tout appel système (exception faite de l'appel *execve(2)* exigeant une manipulation plus précise).

(Mal)heureusement, depuis que Linux 2.5.41 n'exporte plus le symbole *sys_call_table*, alors que ce dernier existe toujours, l'adresse de la mémoire n'est plus disponible pour les modules.

Afin de trouver l'adresse correcte, il est possible d'avoir recours à une certaine technique : */usr/src/*

Glossaire

- LKM – Linux Kernel Module, module central de Linux
- VFS – Virtual File System ou Virtual Filesystem Switch, système de fichiers virtuels

Tableau 1. *Liste des commandes*

Commande	Exemple	Description
`net.hide.src=[IP]`	`net.hide.src=192.168.0.10`	Cache les connexions réseau où se trouve l'adresse locale [IP]
`net.show.dstport=[PORT]`	`net.show.dstport=22`	Affiche toutes les connexions réseau où se trouve le port distant [PORT]
`sys.superroot=[KEY]`	`sys.superroot=dSi2d _ q@d`	Obtient des permissions super utilisateur si la clé [KEY] est correcte
`sys.hide=[PID]`	`sys.hide=1`	Dissimule le processus avec l'identificateur de paramètres PID [PID]
`sys.show=[PID]`	`sys.show=5982`	Affiche les processus cachés avec l'identificateur de paramètres PID [PID]
`sys.guid=[UID],[GID]`	`sys.guid=1000,1000`	Supprime le super utilisateur, règle UID [UID] et GID [GID]
`fs.hide=[FILENAME]`	`fs.hide=dfdfdf-nc`	Cache les fichiers intitulés [FILENAME]
`fs.show=[FILENAME]`	`fs.show=dfdfdfdf-arpspoof`	Affiche les fichiers dissimulés intitulés [FILENAME]

linux/include/asm/unistd.h se charge d'ordonner la table *sys_call_table* au moyen de constantes __NR capables de définir le décalage dans le tableau où se trouve chaque appel système. Dans la mesure où chaque appel système de l'adresse est connu (chaque appel système est exporté), la mémoire peut alors être scannée afin de rechercher la table *sys_call_table*.

Le code chargé d'exécuter cette tâche se contente de déclarer un pointeur qui part d'une adresse de mémoire inférieure (en règle générale, l'adresse de *loops_per_jiffy* est utilisée) et effectue des boucles jusqu'à ce qu'un décalage __NR du pointeur corresponde à l'adresse correcte du même appel système. Si le pointeur atteint une adresse de mémoire supérieure (comme *boot_cpu_data*), la manœuvre a échoué (un module a peut-être déjà intercepté l'appel système utilisé pour chercher la table *sys_call_table*). En d'autres termes, il devient impossible de trouver l'appel système. Auquel cas, il sera impossible de provoquer des interruptions sur les appels système. Cette technique est complètement indépendante des interruptions du système de fichiers virtuels.

Une fois la table *sys_call_table* trouvée, il suffit de la modifier selon la vielle technique déjà mentionnée plus haut. Nous avons exposé dans le Listing 3 un exemple illustrant comment remplacer l'appel système *open(2)*.

Il ne faut surtout pas oublier que les appels système sont des éléments fondamentaux du système. Si un appel système est intercepté alors que la nouvelle fonction de rappel (dans le Listing 3, *new_open*) ne se comporte pas correctement, le système aura à son tour des ratés, et deviendra très probablement instable. La technique la plus répandue consiste à appeler l'appel système original à partir de la nouvelle fonction de rappel lorsque cette dernière décide de permettre son exécution. C'est la raison pour laquelle, le code exposé dans le Listing 3 sauvegarde un pointeur vers la fonction originale. De la même manière, lorsque le module n'est pas censé être chargé, la table *sys_call_table* doit être modifiée de sorte à pointer vers l'emplacement original.

Fonctionnement interne du système de fichiers virtuels

Est désignée par système de fichiers virtuels (Virtual File System ou Virtual Filesystem Switch) une couche située entre les appels système concernant les fichiers (comme *open(2)*) et les implémentations du système de fichiers en question (comme `ext2`, `ext3`, `reiserfs`, `jfs`, etc.). Cette couche fournit une interface conventionnelle chargée de faciliter le travail des programmes d'implémentation du système de fichiers.

Les implémentations du système de fichiers sont chargées de définir un ensemble de fonctions et de méthodes prédéfinies, puis de tenir informer la couche du système de fichiers virtuels des méthodes qu'il invoque en tant que fonctions de rappel au moyen de pointeurs de fonction. En règle générale, un système de fichiers virtuels dispose d'une structure que chaque système de fichiers doit remplir et enregistrer dans la couche du système de fichiers virtuels. Cette structure contient les informations nécessaires à la recherche d'adresses où peuvent se trouver ces fonctions de rappel.

Pendant le développement d'un rootkit, l'appel le plus intéressant est sans conteste *readdir*. Cette fonction de rappel propose un algorithme chargé d'appeler le paramètre *filldir* de la fonction, lequel sera appelé pour chaque fichier ou répertoire lu par *readdir*. Sa valeur de retour permet de préparer les informations sur la lecture du répertoire.

Nous allons utiliser tout au long de cet article une technique faisant appel à la valeur de retour 0 dans la fonction *filldir*. Cette valeur de retour permet à l'appel *readdir* de supprimer les informations sur l'élément lu.

Super utilisateur

Si n'importe quel utilisateur du système était capable de contrôler un rootkit susceptible de lui donner tous les accès au système, ce rootkit en question ne serait pas très efficace. Avant que SIDE n'exécute une commande, le rootkit doit d'abord authentifier l'utilisateur. Il utilise, pour ce faire, la commande intitulée *sys.superroot*. Afin que cette commande authentifie avec succès l'utilisateur, la clé doit être spécifiée sous forme de paramètre de la commande.

Est désignée par clé (en règle générale) une chaîne aléatoire chargée d'identifier l'installation. Le rootkit SIDE sélectionne la clé pour chaque installation lorsque le script *configure* est exécuté.

Lorsque la clé correcte obtient l'UID, le GID de l'utilisateur est remplacé par celui chargé d'identifier le super utilisateur (sélectionné également au même moment que le script *configure*).

Le super utilisateur doit alors permettre l'exécution des commandes tout en évitant de dissimuler les informations à l'utilisateur spécifique caché aux autres utilisateurs.

Résumé

Nous avons étudié, dans le cadre du présent article, différentes techniques et approches autour du développement d'un rootkit fonctionnant sur le noyau de Linux 2.6. Nous avons donc revu les méthodologies permettant de dissimuler les connexions réseau, les processus, les modules et les fichiers, puis nous avons évoqué les contre-mesures utilisées par les détecteurs de rootkits, ainsi que les contre-mesures susceptibles d'être mises en pratique par les développeurs de détecteurs de rootkits et par les administrateurs.

Le développement au sein même du noyau de Linux ouvre tout un nouveau monde d'opportunités par tout un chacun. Et ce n'est seulement que la partie immergée de l'iceberg, qui reste encore à explorer. ●