Université de Bordeaux
Faculté de Sciences et Techniques

rkduck : kernel rootkit pour Linux 4.x.x

Thomas Le Bourlot, Maxime Peterlin, Martial Puygrenier

Bordeaux, le 21 Mars 2016

Abstract

Dans la continuité des travaux réalisés sur les rootkits sur les kernel 2.6.x et 3.x.x nous avons décidé de réaliser une étude des techniques d'injection de rootkits sur les kernels 4.x.x. 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 nous ont aidés dans le développement et la conception du rootkit `rkduck`. Notre travail est disponible sur l'organisation github QuokkaLight.

Sommaire

1 Introduction

Cet article traite des rootkits. Ce sont des programmes permettant de cacher la compromission d'une machine à ses utilisateurs légitimes. Plus précisément, nous expliquerons le développement de notre rootkit nommé `rkduck`, un LKM (Linux Kernel Module) rootkit pour Linux v4.x.x.

Nos propos seront développés de la manière suivante. Dans un premier temps, nous rappelerons en quoi consiste un rootkit et comment ils ont évolué au fil du temps. Une partie sera consacrée aux différentes méthodes d'injection de code dans le noyau Linux. Suite à cela, nous expliquerons comment notre rootkit détourne le fonctionnement du système pour cacher une activité malveillante et de quelle manière il reste persistant sur la machine. Enfin, nous expliquerons les fonctionnalités mise à disposition par notre rootkit.

2 Définition et évolution des rootkits

Afin de compromettre une machine, un attaquant passe généralement par la même série d'étapes. Une fois que des vulnérabilités dans le système ont été identifiées, l'attaquant les exploite afin de prendre la main sur le système jusqu'à compromettre entièrement ce dernier afin d'avoir les droits de super-utilisateur. Cependant, l'exploitation de failles dans un système ne peut être considéré 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.

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. Ainsi, l'intrus avait le contrôle sur les sorties de se programme et pouvait cacher l'activité malicieuse qui se passait sur la machine. Cependant, il devenait fastidieux de cacher la compromission de la machine à l'administrateur légitime. 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 de programmes, ce qui peut devenir très conséquent en terme de taille et en temps de développement. Il suffit de remonter aux fonctions utilisées par tous ces programmes d'administration : les appels systèmes. 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 même niveau que ce dernier et des fonctions que l'on peut penser digne de confiance peuvent retourner des résultats falsifiés.
C'est ce type de rootkit que nous allons étudier par la suite, notamment les rootkits s'injectant dans le système sous forme de modules kernel.

3 Injection - Comment le rootkit est injecté en mémoire ?

Il existe plusieurs méthodes d'injection de code dans un noyau. Nous étudierons dans cette partie les deux méthodes les plus répandues à savoir l'utilisation du périphérique `/dev/mem` et des modules noyau. A titre informatif, il est également possible d'utiliser des vulnérabilités du noyau ou bien l'entrée `FireWire` qui permet d'avoir un accès direct à toute la mémoire de la machine, ce qui nécessite cependant un accès physique à cette dernière.

Afin de développer `rkduck` nous nous sommes, dans un premier temps, tournés vers l'injection basée sur `/dev/mem`. Cependant, un patch de sécurité empêche son utilisation sur les versions 4.x.x du noyau Linux, ce qui fait que nous nous sommes finalement tournés vers la construction d'un LKM rootkit.

3.1 Injection via /dev/mem

`/dev/mem` est un fichier qui fournit 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 sytème. `/dev/mem` est un point d'entrée pour injecter du code malicieux, 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`, comme nous le verrons 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, si on suppose qu'il est possible de charger un LKM et d'écrire dans `/dev/mem`, 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, alors que 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`. 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. La mémoire kernel possède 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 contigüe en mémoire physique, même ce dernier verra les adresses virtuelles comme contigües.

Nous avons cependant mis de côté ce type d'injection, car elle est aujourd'hui désactivée par défaut dans la plupart des distributions, notamment dans *Jessie* la version 8 de Debian sur laquelle nous développons et effectuons la plupart de nos tests.

De manière plus générale, depuis les versions 2.6.26 du kernel linux une options activé par défaut, `CONFIG_STRICT_DEVMEM` qui limite l'accès à `/dev/mem` au premier megabyte. Cela per-

met d'accéder aux périphériques PCI et certaines régions du BIOS ce qui est suffisant pour les applications qui ont besoins d'utiliser `/dev/mem` et empêche les applications d'injection du code malicieux.

3.2 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.

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 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 nécessairemment 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.

Nous nous sommes renseignés 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 de retirer 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)` 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 avant de le retirer, sauf que la fonction de suppression des modules est modifiée par le rootkit. La 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.

Il existe plusieurs contre-mesures 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 séconde 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 nous l'avons expliqué précedemment, si l'on ne souhaite pas empêcher l'insertion de LKM, alors il est possible de rajouter des étapes d'authentification et de vérification qui précèderont le chargement du module. En effet, on peut demander un mot de passe 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.

4 Détournement de l'exécution du noyau

Nous avons utilisé deux méthodes différentes 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`.

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

La solution la plus directe et la plus simple pour détourner les appels système est de modifier la table des appels système.

4.2 Détournement du `Virtual File System`

Le `Virtual File System` (VFS) est une couche de communication se situation entre le kernel et le système de fichier utilisé. Cette couche est une abstraction permettant au kernel de communiquer avec n'importe quel système de fichier (ext3, ext4, etc.), c'est le VFS qui se chargera de traduire les instructions envoyées par le kernel pour qu'elle soit compréhensible par le système de fichier.

L'intérêt de s'attaquer au VFS est de pouvoir cacher des fichiers aux yeux de l'utilisateur légitime. Ainsi, on va pouvoir dissimuler de manière relativement furtive des fichiers, des dossiers, mais aussi des processus grâce à des hooks sur des fonctions du VFS.

Nous allons, dans la suite de cette partie, illustrer nos propos en construisons le système servant à dissimuler des fichiers et des dossiers. Le même principe pourra s'appliquer à la dissimulation de processus.

Premièrement, nous allons récupérer un pointeur sur fonction `iterate` disponible dans la structure `file_operations` afin de la détourner. Étant donné que nous souhaitons ici cacher des fichiers, il faudra que ce pointeur soit relatif à la racine `/`, pour les processus on pourra se cantoner

à /proc.

```
struct file_operations {
  /* [...] */
  int (*iterate) (struct file *, struct dir_context *);
  /* [...] */
};
```

Listing 1: Fonction iterate

Pour récupérer un pointeur sur cette fonction, il faut d'abord récupérer un pointeur vers cette structure `file_operations` relative à la racine `/`, ce qui est rendu possible grâce à la fonction `filp_open`.

```
void *vfs_get_iterate(const char *path) {
    void *ret;
    struct file *file;

    if ((file = filp_open(path, O_RDONLY, 0)) == NULL)
        return NULL;

    ret = file->f_op->iterate;
    filp_close(file,0);

    return ret;
}
```

Listing 2: Fonction vfs_get_iterate dans rkduck

Le détournement de la fonction de la fonction `iterate` va s'effectuer de la manière suivante. Dans un premier temps on va garder une copie du pointeur sur cette fonction que l'on appelera `original_iterate`. Grâce à ce pointeur on va pouvoir avoir un accès direct au code de cette dernière, ce qui va nous permettre de le modifier. En effet, on va changer les premières instructions de la fonction pour que cette dernière fasse un saut vers une fonction que l'on contrôle appelée `hijacked_iterate`. Il faut également sauvegarder le code de la fonction originale, car cette dernière sera réutilisée plus tard.

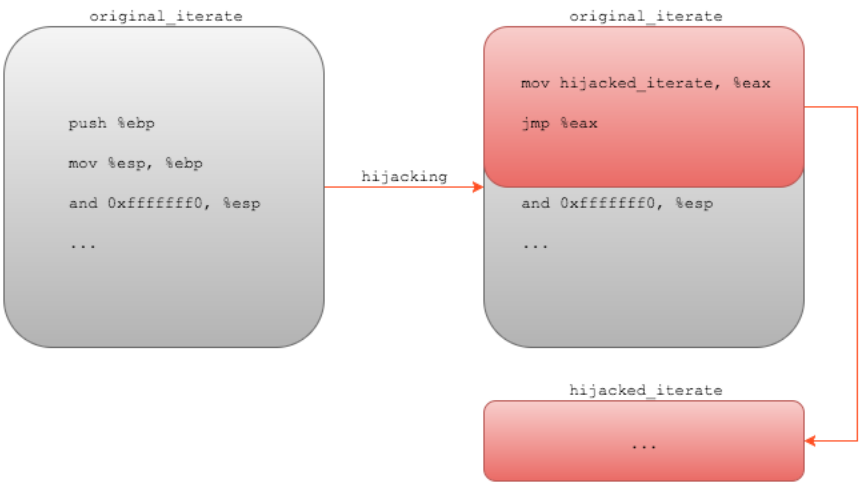

Figure 1: vfs iterate: original to hijacked

A présent, la fonction qui va nous intéresser et qui est appelée par `iterate` est la fonction `filldir`. Lorsque l'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 contenant les informations sur les éléments contenus dans ce dossier. Ainsi, l'intérêt va être de remplacer cette fonction par une fonction malveillante `filldir_hijacked` afin de contrôler quels fichiers nous souhaitons cacher.

Dans la fonction `iterate`, c'est un pointeur vers la fonction `filldir` stocké dans la structure `struct dir_context *ctx` qui permet de l'appeler. Ainsi, la fonction `hijacked_iterate` aura pour unique but de modifier le pointeur vers la fonction `filldir` pour qu'il pointe vers notre fonction `filldir_hijacked` 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 3: Version simplifiée de vfs_hijacked_iterate dans rkduck

Comme nous l'avons spécifié précédemment, le code original de la fonction `iterate` a été sauvegardé. Une fois que l'on a modifié la structure `struct dir_context *ctx` pour qu'elle contienne un pointeur vers notre fonction `hijacked_filldir`, on rétabli le code original de la fonction `iterate` grâce à la fonction `vfs_hijack_stop`. Ensuite on appelle `iterate` qui va

maintenant faire appel à notre fonction `hijacked_filldir` au lieu de l'original. Ainsi, en recodant `filldir` pour qu'elle renvoit `0` lorsqu'un fichier qu'on veut cacher a été passé en paramètre (`rkduck` dans l'exemple ci-dessous), on peut effectivement dissimuler à l'utilisateur légitime des fichiers malveillants. Il ne reste plus qu'à détourner à nouveau la fonction `iterate` pour qu'on puisse continuer la dissimulation de fichiers grâce à la fonction `vfs_hijack_start`.

```
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 4: Version simplifiée de vfs_hijacked_filldir dans rkduck

5 Persistance du rootkit

La persistance du rootkit correspond en quelque sorte à sa durée de vie une fois injecté dans le kernel. Elle est très importante, le but du rootkit étant de récupérer des informations sensibles etc... Sa persistance au sein du système est donc primordiale.

Lors d'un redémarrage les modules ajoutés manuellement ne sont pas rechargés, il faut l'indiquer manuellement au système. Nous avons donc écrit un script bash qui va écrire le nom de notre module dans un fichier spécifique `/etc/modules/` (le nom et chemin du fichier n'est pas le même suivant l'OS) et placer notre rootkit compilé dans `/lib/modules/../kernel/drivers/directory/`. Pour perdre un peu plus l'utilisateur de la machine qui pourrait chercher manuellement dans la liste des dossiers drivers un nom suspect qui pourrait le mettre sur la piste d'une activité malicieuse, le script s'occupe de placer le rooktit compilé au sein d'un dossier légitime exemple : `/kernel/drivers/tty/rkduck.ko`. Le nom du dossier légitime peut être changé par l'attaquant avant l'injection du module, il pourra pendre soin de renommer aussi le rootkit compilé en un nom qui parait légitime du style audio.ko etc.

Lorsque le rootkit est chargé pour la première fois le script s'exécute et à chaque démarrage de la machine notre rootkit sera maintenant chargé au sein du kernel, la persistance est donc acquise et nous pouvons maintenant présenter les différentes fonctionnalités de notre rootkit.

6 Fonctionnalités du rootkit

6.1 Cacher des dossiers et fichiers

6.1.1 Dossiers

lorem

6.1.2 Fichiers

lorem

6.2 Backdoor

6.2.1 Bind shell

Un bind-shell correspond dans notre cas à un serveur socket qui attend une connexion de la part de l'attaquant pour pouvoir exécuter des commandes shell et renvoyer le résultat. Dans cette optique nous avons mis en place un mini-serveur socket qui va simplement attendre les connexions entrantes sur un port spécifique fixé lors de l'injection du rootkit. Une fois la connexion établie, le serveur va attendre les commandes envoyées par l'attaquant, exécuter ces commandes, récupérer le résultat et renvoyer le tout au destinataire.

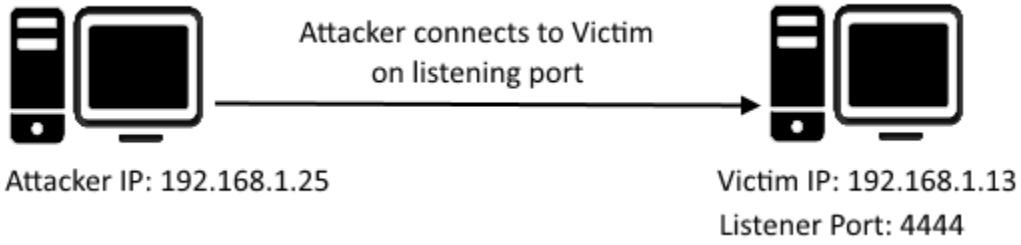

Figure 2: bind shell

L'exécution des commandes se fait via l'API `call_usermodhelper(...)`. Comme il n'est pas possible de récupérer le résultat des commandes directement, nous avons donc redirigé la sortie standard vers un fichier temporaire `/tmp/.cmd`, le résultat est lu puis renvoyé. Lorsque la connexion est interrompue avec l'attaquant le fichier qui contient le résultat de la dernière commande est supprimé.

6.2.2 Reverse shell

Le reverse shell consiste à l'opération inverse du bind-shell, ce n'est pas le rootkit qui va attendre une connexion mais l'attaquant qui va se mettre en mode "écoute". Le rootkit quand à lui va initier la connexion vers l'attaquant. (L'adresse ip doit donc être indiquer avant que le rootkit soit injecté). Lorsqu'il reçoit les commandes shell envoyées par l'attaquant le rootkit va se comporter

exactement comme le bind shell, c'est à dire copier le résultat dans un fichier temporaire, lire le fichier et renvoyer le résultat.

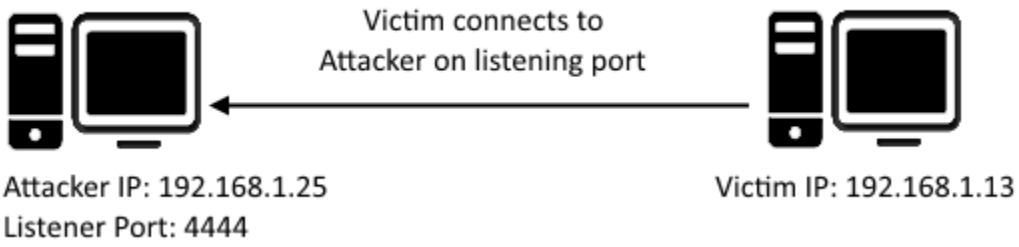

Figure 3: reverse shell

6.2.3 Activation des backdoors

Tout d'abord, l'attaquant choisit d'activer l'une des deux backdoor suivant la situation. Il peut activer et désactiver les backdoors via **crumb** comme nous l'expliquerons dans la partie (voir section **??**). Ensuite il serait inconscient de laisser un bind-shell en écoute toute la journée si on veut rester le plus discret possible. Pour le reverse-shell il faut aussi lui donner un signal pour qu'il puisse se connecter à l'attaquant, nous avons pour cela décidé que la backdoor pourrait s'activer en envoyant un paquet ICMP avec un payload spécifique. Ce payload serait seulement connu de l'attaquant et seul lui peut donc se connecter à sa backdoor.

Nous avons donc implémenté un mini firewall à l'intérieur du rootktit qui va intercepter les paquets et regarder le type et le payload contenu, si il voit par exemple passer un paquet ICMP avec un payload spécifique et une taille de 10 alors il activera la backdoor :

- pour le reverse shell le rootkit va essayé de se connecter à l'ip de l'attaquant
- pour le bin-shell le rootkit va commencer à écouter et attendre une connexion venant de l'attaquant

Enfin il est possible que les paquets soient filtrés par un firewall externe, par exemple les paquets ICMP comportant un payload pourraient êtres rejetés et l'attaquant n'aurait aucun moyen d'accéder aux backdoors. Nous avons donc mis en place un timer qui va appeler activer la backdoor toutes les 12h. Le temps est bien sur configurable par l'utilisateur à l'injection du rootkit.

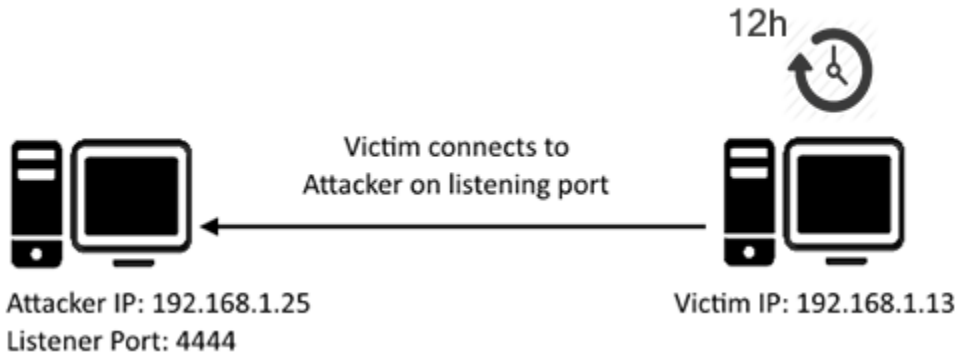

Figure 4: timer shell

6.2.4 SSH backdoor

Afin de permettre à l'attaquant d'avoir encore plus la main sur la machine corrompu et dans le cas ou le défenseur désactiverait tout les ports non conventionnels, nous avons mis en place une backdoor ssh que l'attaquant est libre d'activer ou non. Elle va tout simplement mettre sa clé publique dans le fichier `/.ssh/authorized_keys` de root, il peut ainsi tout simplement se connecter en ssh avec l'utilisateur root. Évidement cela suppose que la machine cible est accessible depuis l'extérieur et que root puisse se connecter en ssh (option qui peut être désactivé dans les fichiers de configuration ssh).

Nous considérons cette "backdoor" comme options de secours dans le cas ou les autres ne fonctionnerais plus. Elle n'est pas très robuste car il suffit au défenseur de supprimer la clé public dans le fichiers `authorized_keys` mais encore faut il qu'il regarde se fichier et détermine qu'elle clé est la mauvaise. Si nous avons rajouté cette option ssh c'est parce qu'elle est utile si on sait que le défenseur a détecté nos traces (fermeture des ports, filtrage réseaux, commandes suspects netstat etc), il y a peu de chance qu'il ferme aussi le port ssh. L'attaquant peux alors se connecter directement en ssh et supprimer le rootkit avant que la victime n'est le temps de le récupérer pour une analyse forensic. Ou si la discrétion n'est pas la priorité, l'attaquant peut décider changer les ports, le rootkit de place, supprimer le filtrage etc. Le but étant comme nous l'avons évoqué plus haut de "perdre" un maximum le défenseur ou de rester le plus discret possible...

6.2.5 Cacher les connexions réseaux

lorem

6.3 Contrôle du rootkit depuis userland

lorem

6.4 Keylogger

lorem

7 Détection du rooktit

-¿ honeypot page 12

8 Conclusion

Lorem

9 Annexes

Lorem

10 Bibliographie

Lorem