Backdoors furtives et autres fourberies dans le noyau

Arnaud Ebalard[1], Pierre Lalet[2], and Olivier Matz[3]

Droids Corporation
team@droids-corp.org
http ://www.droids-corp.org/
[1]troglocan@droids-corp.org [2]pierre@droids-corp.org [3]zer0@droids-corp.org

Remarques préliminaires

Cet article expose les raisons pour lesquelles il peut être intéressant de se cacher dans le noyau d'un système d'exploitation, que l'on soit animé de bonnes ou de mauvaises intentions. Nous présenterons quelques techniques pour le faire, ainsi que quelques contre-mesures possibles.

Nos connaissances dans le domaine des noyaux portent exclusivement sur les systèmes de type Unix (Linux et ⋆BSD principalement), mais les techniques que nous allons décrire peuvent sans doute être adaptées à des systèmes d'exploitation alternatifs (BeOS, QNX, MS-Windows, ...).

Notre expérience est venue, au départ, du port de Sebek2 (voir *Know Your Enemy : Sebek*[1]) pour FreeBSD (dans le cadre d'un projet scolaire) puis pour NetBSD et OpenBSD, ainsi que de l'étude des ports *officiels* pour Linux et OpenBSD (nos travaux à propos de Sebek et des honeypots en général sont consultables sur `http://honeynet.droids-corp.org/`).

Sommairement, Sebek est un outil lié aux pots à miel. Installé sur une machine, il envoie sur le réseau le contenu de la totalité des appels à `read()`. On peut ainsi récupérer tout ce qu'a fait le pirate qui a pris la main sur le pot à miel (y compris ce qui a pu être protégé par des solutions cryptographiques comme ssh).

Pour l'implémentation de ces ports, nous nous sommes assez peu inspirés des versions existantes et nous avons préféré recommencer en respectant le protocole tel qu'il est décrit dans le document de présentation de Sebek précédemment cité. Nous avons recherché des exemples de codes similaires, c'est-à-dire, des façons de détourner le code du noyau. Nous avons rencontré beaucoup de choses liées au *côté obscur* de la bidouille informatique, et nous avons mesuré l'intérêt de se placer dans le noyau pour écrire une *backdoor*. Cela nous a donné l'idée de cette présentation.

[1] `http://www.honeynet.org/papers/sebek.pdf`

1 Introduction

1.1 De l'intérêt de se cacher dans le noyau

Le noyau est l'élément central du système d'exploitation. Cette phrase, sous les dehors d'une belle banalité digne d'un commercial, est essentielle dans les domaines de la dissimulation, de l'interception de données, et de l'action sur le système corrompu. Cela veut dire que toute tentative de détection passera par lui, que tous les actes effectués sur le système passeront par lui, et qu'il est capable de tout faire sur le système (plus encore que `root`, dont les privilèges sont susceptibles d'être limités par des patchs noyau sous Linux, ou encore par le `securelevel` sous BSD). Il est omniscient et omnipotent.

En effet, le noyau fournit une interface normalisée permettant aux applications de réaliser des tâches faisant appel au matériel (accès disque, émission/réception de trafic réseau, accès au clavier, ...). Ainsi, le noyau est en quelque sorte le point de convergence obligé des données transitant entre les processus de l'utilisateur et les périphériques du système. De plus, celui-ci contrôle et limite la façon dont les processus évoluent sur le système (identification et contrôle d'accès aux ressources).

1.2 Les choses intéressantes à faire

Il existe à partir de là deux raisons essentielles pour vouloir se cacher sur une machine : celle du pirate, qui a pris une machine et souhaite y laisser une porte dérobée, et celle de l'administrateur système, qui souhaite par exemple mettre en place un pot à miel, et veut être sûr de pouvoir accéder à tout ce qu'a fait le pirate.

Globalement, les besoins de ces deux types particuliers d'utilisateurs sont les mêmes : pouvoir effectuer certaines actions sans que l'autre ne s'en rende compte. Ceci inclut l'envoi et la réception de trafic réseau, l'espionnage des actions de l'autre, le masquage de ses propres actions, ...

En voici quelques exemples :
- récupération des frappes claviers ou de toute autre donnée ;
- lecture, modification, suppression et dissimulation de fichiers ;
- écoute du réseau ;
- émission de données sur le réseau ;
- dissimulation de trafic réseau ;
- ...

1.3 Où se placer ?

Selon les actions que l'on souhaite réaliser à l'intérieur du noyau, il existe plusieurs endroits où il peut être intéressant de venir se positionner. Le schéma qui suit présente quelques-unes des possibilités de *hook*, dont certaines seront détaillées par la suite :

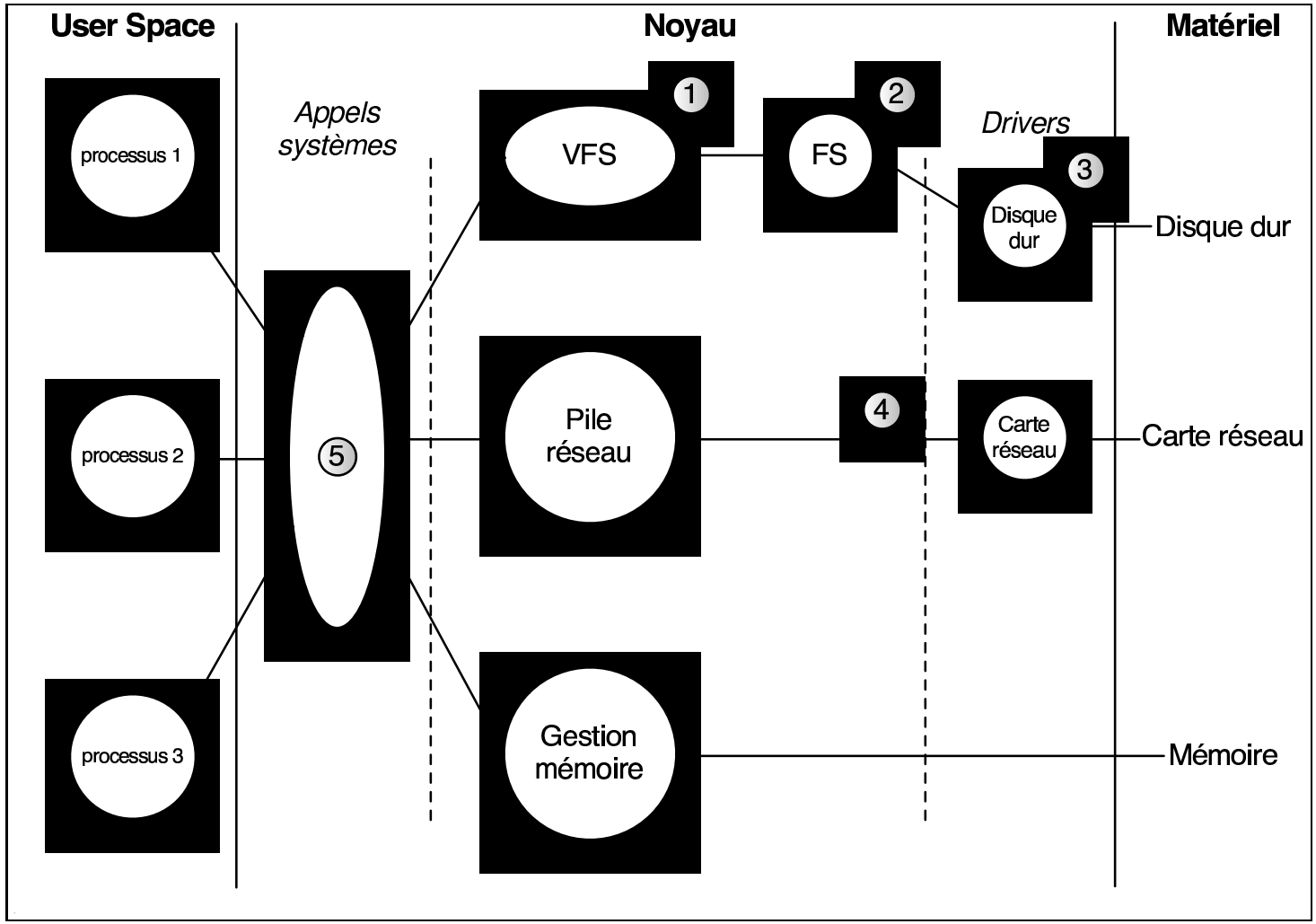

Fig. 1. Où se *hooker* dans le noyau

1. **VFS** : un hook à cet endroit permet par exemple de masquer certains répertoires ou fichiers aux utilisateurs.
2. **Système de fichier particulier** : à la différence du VFS, on est limité à des interactions avec un seul type de système de fichier (*e.g.* `ext2fs`).
3. **Driver** : les drivers sont la partie du noyau la plus proche du matériel. Ainsi, se placer à ce niveau pour un périphérique réseau permet d'envoyer et de recevoir des trames directement sur ce contrôleur particulier. On n'a par contre pas accès aux cartes réseau gérées par un autre driver.
4. **Entre la pile réseau et le driver** : dans le cas du réseau, il s'agit de la dernière couche indépendante du matériel utilisé. Elle offre donc un point d'accès global sur tous les périphériques réseaux présents sur le système.
5. **Appels système** : ils fournissent l'interface entre les processus en espace utilisateur et le noyau. Ils permettent à ces processus d'exécuter des actions qui nécessitent de travailler en mode kernel ou d'avoir d'autres types de droits particuliers (accès au matériel, communications interprocessus, ...). Ce hook est intéressant car il offre notamment un accès à l'ensemble des données transitant lors des demandes de lecture (au clavier, dans des fichiers, sur le réseau, ...).

Cette liste est loin d'être exhaustive ; il est par exemple possible de se placer au niveau du gestionnaire d'interruption pour récupérer les frappes clavier ou encore d'utiliser l'interface `bpf` pour avoir accès au trafic entrant. Nous reviendrons sur ces cas plus loin.

1.4 Comment entrer ?

Ce document n'a pas pour but d'évoquer les manières de s'introduire dans un système (dans le cas de l'administrateur, le problème ne se pose pas). Nous nous intéresserons uniquement à la manière d'injecter du code au niveau du noyau une fois cette étape d'entrée effectuée.

Selon les cas, la façon de procéder va être différente : il est en effet plus difficile de changer le noyau quand on n'est pas le propriétaire *légitime* de la machine cible. Plutôt que de modifier le noyau, il est envisageable de corrompre quelques binaires essentiels, mais ceci est facilement contournable car l'*adversaire* peut amener les siens.

Prenons un exemple simple : vous êtes un pirate et vous voulez vous assurer que vous n'êtes pas sur un pot à miel, ou bien vous êtes un administrateur, et soupçonnez que la machine soit corrompue. Vous souhaitez par exemple connaître le trafic réseau en cours. Quel que soit le programme que vous allez utiliser pour cela, il fera appel au noyau. Si celui-ci lui masque certains flux, vous ne pourrez pas y avoir accès. Les fichiers de log de votre porte dérobée peuvent également être rendus presque totalement invisibles.

Nous allons nous intéresser à quelques méthodes permettant de récupérer des informations depuis le noyau en détournant les actions de celui-ci pour qu'elles offrent les renseignements souhaités. Dans un second temps, nous traiterons de l'envoi et de la réception furtifs de données sur le réseau.

2 Insertion du code

Nous devons donc placer notre code dans le noyau, mais comment faire ? Trois solutions majeures existent. Ces solutions sont très différentes, et le choix se fera donc en fonction des possibilités offertes par la configuration de la machine, de notre niveau technique, et selon que l'on est le propriétaire de la machine ou pas.

2.1 Remplacer le fichier du noyau

On peut tout d'abord remplacer le noyau (comprendre, le fichier du noyau sur le disque), et redémarrer la machine (on peut même penser à provoquer un plantage de la machine pour contraindre l'administrateur à la redémarrer). Il s'agit de la solution quand on est le propriétaire de la machine, et quand on peut la redémarrer (et il n'y a pas vraiment de raison, *a priori*, que l'on ne puisse pas redémarrer quand on met en place un pot à miel).

2.2 LKM

On peut aussi, si le noyau le supporte, utiliser les LKM ; c'est le plus simple lorsqu'on n'est pas vraiment maître de la machine. LKM signifie *Loadable Kernel Module*, et désigne les fameux modules du noyau. Ceux-ci permettent d'ajouter dynamiquement au noyau des fonctionnalités dont on n'a habituellement pas besoin tout le temps (*e.g.*, drivers pour une Webcam).

Cela devient d'un seul coup plus facile, mais aussi plus visible. En effet, des commandes comme `lsmod` sous Linux, `kldstat` sous FreeBSD ou `modstat` sous NetBSD et OpenBSD existent pour connaître la liste des modules actuellement chargés. Il est toutefois possible de modifier cette liste de manière à masquer la présence d'un module donné. Pour cela, il suffit d'insérer un second module qui va se contenter de supprimer une entrée de la liste. Cependant, nous allons voir dans la partie suivante que ce n'est pas forcément une solution miracle.

Une autre solution (truff, Phrack 61) est de profiter des modules existants sur le système et d'*infecter* l'un d'entre eux de manière à lui ajouter les fonctionnalités qui nous intéressent. Il suffit pour cela de disposer d'un module effectuant les actions souhaitées, d'injecter le code de celui-ci dans le module présent sur le système (réalisable à l'aide de `ld`). Il ne reste alors qu'à modifier la table des symboles (section `.symtab`) de l'objet ELF (Executable and Linking Format) obtenu pour faire en sorte de remplacer, par exemple, la routine appelée à l'initialisation par une des notres. Comme de nombreux systèmes utilisent le format ELF pour leurs modules (Linux, ⋆BSD, ...), ceci est réalisable sur un grand nombre de machines. Au final, le module dans lequel se trouve le code étant l'un de ceux de l'utilisateur, il sera beaucoup plus furtif et pour peu que celui-ci fournisse un service important (comme celui d'une carte réseau, par exemple) il sera chargé de manière presque certaine.

Quand on a accès à la mémoire du noyau, on a souvent besoin de la table des appels systèmes. Quand ce symbole n'est pas exporté, on peut, comme dans Sebek-Linux, parcourir tout une plage de mémoire à la recherche de ce symbole

(on cherche un pointeur `p` tel que `p[__NR_close] == sys_close`, par exemple). Lorsqu'on a besoin de certains symboles, il est également parfois utile, sous Linux, de jeter un œil au fichier `/boot/System.map`.

Notons que le mécanisme securelevel des BSD interdit (entre autres) l'insertion de modules dans le kernel après le boot du système (si `kern.securelevel` est passé à une valeur strictement positive).

2.3 `/dev/kmem`

La troisième option est d'utiliser les fichiers spéciaux `/dev/mem` et `/dev/kmem` afin de modifier le noyau à la volée (en mémoire) sans avoir recours aux modules. En effet, ceux-ci peuvent avoir été interdits, et puis ce sera de toutes façons beaucoup plus discret. Revers de la médaille, ceci est plus difficile techniquement, et le mécanisme des securelevel des BSD précédemment évoqué rend également inefficace ce procédé en interdisant l'écriture vers ces périphériques (toujours avec la même condition que `kern.securelevel` doit être passé à une valeur strictement positive). De la même manière, des patchs noyaux de linux comme *GrSecurity* permettent de limiter l'accès à `/dev/mem` et `/dev/kmem`.

Pour cette troisième option, vous devez avoir les permissions nécessaires pour écrire dans le périphérique `/dev/kmem` (au sens du système de fichier virtuel VFS) ; mais (dans le cas de linux) ce n'est pas suffisant, car un test[2] supplémentaire est effectué au niveau de la gestion des périphériques de type `mem` (dans le fichier `drivers/char/mem.c`, fonction `open_port`), pour vérifier que l'utilisateur est autorisé à effectuer des entrées / sorties directes. Dans le cas des BSD, c'est, comme nous venons de le voir, la valeur de `securelevel`, qui peut être un obstacle, même si les permissions sont supposées permettre l'écriture.

Depuis l'espace utilisateur, on utilise les fonctions classiques `open`, `lseek`, `read`, `write`, `close`, ... pour manipuler ce périphérique.

Comme on a besoin d'obtenir certaines adresses de symboles, sous Linux, le réflexe est de se tourner vers `/boot/System.map`. Cependant, comme il arrive que ce fichier soit supprimé par un administrateur parano (et visiblement, à juste titre...), on retombe dans le même *Système D* que dans le cas des LKM, en pire, vu que l'on n'a aucun symbole exporté.

On commence donc par chercher la table des appels système, qui nous donnera beaucoup d'informations. Pour cela, une méthode classique[3] consiste à demander la table des interruptions, puis à en déduire l'adresse de la fonction de traitement de l'interruption `0x80`, qui contient un appel (`CALL`) à l'adresse de la table plus un offset qui dépend de l'appel système à traiter. On cherche alors cet appel et on en déduit l'adresse tant convoitée.

Un problème auquel on est confronté, auquel on n'avait pas à faire face dans le cas des LKM, est celui de la réservation de mémoire pour stocker notre code.

[2] (*capable(CAP_SYS_RAWIO)*)

[3] Cette méthode est entre autres décrite dans le phrack 58 (Volume 0x0b, Issue 0x3a, Phile #0x07 of 0x0e, signé sd `<sd@sf.cz>` et devik `<devik@cdi.cz>`).

Il faut pour cela localiser la fonction `kmalloc`. Une solution (nous n'en avons pas trouvé d'autre) est de se tourner vers la recherche de motifs sur le code (ou le code supposé) de la fonction. C'est assez fastidieux et dépend fortement du système et de sa version.

Un endroit où placer son code tout en évitant que celui-ci puisse être effacé de manière intempestive par le noyau (lors d'un besoin de mémoire) est proposé par Silvio Cesare dans le cas Linux. La zone mémoire permettant de servir les requêtes effectuées par appel à `kmalloc` démarre sur un début de page. La zone qui la précède en mémoire est celle réservée au noyau à la compilation. Celle-ci ne termine pas forcément une page. Un *padding* est donc nécessaire pour que le pool réservé aux appels à `kmalloc` débute sur une nouvelle page. Cette zone de *padding*, dont le noyau ne se sert pas est donc utilisable, sans risque d'être réclamée par celui-ci et ceci, sans nécessité d'allocation.

Enfin, il est possible de faire certaines choses amusantes sans avoir besoin de réserver de la mémoire. Il s'agira par exemple de supprimer le test effectué, lors de l'appel système `open`, pour vérifier que l'utilisateur qui fait cette demande dispose bien des permissions nécessaires.

Prenons l'exemple d'un noyau Linux 2.4.26 sur architecture i386. La vérification est faite après un appel à la fonction `permission` dans la fonction `open_namei`. Dans les conditions de notre expérience, l'analyse du code assembleur révèle que pour atteindre notre objectif, on peut remplacer une instruction `JNE` par une série de six `NOP` (l'instruction `JNE` et son argument occupent dans ce cas en effet six octets en mémoire). La fonction `sys_open` est appelée lorsqu'un programme en espace utilisateur effectue un appel système `open`. On cherche la troisième instruction `CALL`, ce qui nous donne l'adresse de la fonction `filp_open`. Dans cette fonction, on cherche le premier `CALL`, qui est un appel à la fonction `open_namei`. Une fois dans cette fonction, il faut chercher le troisième `CALL`, qui est celui qui appelle la fonction `permission`. En restant dans le code de `filp_open`, le saut conditionnel `JNE` qui suit (peu d'instructions après) est celui que l'on veut supprimer. On remplace alors les six octets par six `0x90`. Il n'y a plus de vérification de permissions lors d'un appel à `open` (jusqu'au prochain reboot bien entendu).

Ces exemples montrent, nous l'espérons, la relative difficulté induite par l'utilisation de `/dev/kmem` pour installer une backdoor à la volée. En contre-partie, quand ceci est bien fait, c'est beaucoup plus élégant, efficace et furtif.

3 Récupération des données depuis le noyau

Comme nous l'avons déjà évoqué précédemment, la modification des appels systèmes est une méthode privilégiée pour récupérer des données provenant des processus. Les exemples présentés ici permettent de se rendre compte dans le cas particulier du détournement des appels systèmes des techniques qui permettent l'insertion de code dans le noyau. Certaines de ces techniques ne s'appliquent d'ailleurs pas uniquement au détournement des appels systèmes mais permettent aussi de placer du code à d'autres endroits dans le noyau.

3.1 Détournement d'appels système

Si l'on a choisi d'écrire un module, on va maintenant vouloir détourner quelques appels système ou quelques fonctions clefs, que ce soit pour espionner les utilisateurs, les applications, ou les pirates dans le cas d'un honeypot.

Si l'on a choisi de modifier le noyau au moyen d'un patch, ce problème ne se pose pas, car le code est directement ajouté à l'intérieur des fonctions.

Table des appels système Lorsqu'on veut détourner un appel système (au hasard, `read`), on peut modifier l'entrée qui le concerne dans la table des appels système ; cependant, ceci peut être repéré très facilement.

Dans le cas de FreeBSD, si l'on veut détourner l'appel système `read`, on peut insérer un module contenant une fonction dont le prototype est :

```
static int false_read (struct thread *td, struct read_args *uap)
```

Notre module modifie la table des appels système dans le cas `MOD_LOAD` du `handler`, c'est à dire dans le code exécuté lors du chargement du module :

```
sysent[SYS_read].sy_call = (sy_call_t *) false_read;
```

A chaque fois que l'appel système `read` est appelé, le système interroge la table des appels systèmes, trouve l'adresse de notre fonction, et l'utilise. Notre fonction `false_read` peut même faire appel à la fonction originale, dont le code n'est pas modifié.

Mais cette méthode possède des parades simples : il suffit en effet de regarder la table des appels système, et de la comparer si possible avec une table *saine*. En effet, dans ce cas, les adresses des fonctions pointées par la tables sont proches et se suivent. Une fois la table modifiée, la fonction pointée possède une adresse différente des autres.

Utilisation d'un `JUMP` Ainsi, il peut être judicieux de ne pas modifer la table des appels systèmes, puisque ceci est facilement repérable. Une autre technique consiste alors à écraser le début du code de la fonction à remplacer par un `JUMP` vers du code qui effectue le même travail que la fonction concernée, et un petit supplément (log, analyse des données pour éventuellement donner plus de privilèges ou pour exécuter du code, envoi sur le réseau, ...).

Prenons un exemple pour FreeBSD[4]. Ces lignes doivent être placées dans le cas `MOD_LOAD` du `handler` (code d'initialisation exécuté lors du chargement du module, équivalent de `init_module` sous Linux) :

```
int *jump_address;
char *read_address = (char *) read;

jump_address = (int *) (read_address + 1);
*read_address = 0xE9;
*jump_address = (int) false_read - (int) read - 5;
```

[4] Cet exemple provient de la première version de Sebek pour FreeBSD

Nous sommes sur une architecture compatible i386 (`0xE9` signifie `JUMP`). Nous avons dans notre code une fonction dont le prototype est :

```
static int false_read (struct thread *td, struct read_args *uap)
```

Ce code va donc modifier, lors de l'insertion du module, le code de la fonction `read()` pour placer au début un `JUMP` vers une adresse relative correspondant à la différence entre les adresses en mémoire des fonctions `false_read()` et `read()` moins 5 (la taille en octets de l'instruction `JUMP` et de son argument). Cela signifie que lorsque la fonction `read()` est appelée, le `JUMP` est suivi, et le code de la routine `false_read()` est exécuté. Le reste du code de la fonction `read()` n'est jamais exécuté. Bien évidemment, notre nouvelle fonction ne peut plus réutiliser facilement la routine `read()` originale, comme c'était le cas lorsqu'on se contentait de modifier la table des appels systèmes. Il est ainsi nécessaire que notre fonction contienne tout le code de la fonction dont le début a été écrasé pour fournir les fonctionnalités désirées de celle-ci.

Pour la détection, on peut chercher à lire le début du code assembleur de quelques fonctions essentielles ; ceci se fait par exemple très simplement en insérant un module qui lit et affiche le code d'une fonction. Exemple simpliste, toujours pour FreeBSD, permettant de réaliser cela :

```
int i;
char *code = (char *) read;
for(i=0; i<10; i++)
  printf("%x ", code[i]);
printf("\n");
```

Amélioration de la furtivité Une première idée consisterait à placer un `JUMP` ou un `JUMP` avec condition (`JNE`, `JGE`, ...) plus discret, non pas au début de la fonction initiale mais au début d'une fonction légitimement appelée par cette fonction initiale. Ainsi, cette démarche oblige la personne souhaitant se rendre compte de cette modification à descendre encore d'un niveau d'appel pour réaliser ses tests (scruter le début de chaque fonction appelée par la routine initiale). Mais il faut bien se rendre compte que ceci n'est possible que lorsque la fonction initiale appelle une autre fonction avec des paramètres similaires aux siens. C'est le cas de `read` qui appelle `dofileread` avec les mêmes paramètres.

Il existe bien sûr d'autres méthodes plus compliquées permettant d'accroître la furtivité des modifications apportées.

3.2 Détournement d'interruptions

Toujours dans la famille récupération de données, un autre exemple d'endroit intéressant où se placer est le gestionnaire d'interruption. Une interruption est un événement, qui, lorsqu'il survient, a pour conséquence automatique l'exécution d'un code situé à un endroit défini. Il existe plusieurs types d'interruptions :

les interruptions synchrones au CPU, telles les exceptions et interruptions logicielles, et les interruptions asynchrones, comme celles qui sont générées par les périphériques, par exemple la souris ou la carte réseau.

Tout comme pour les appels systèmes, il existe une table qui associe à chaque numéro d'interruption un pointeur vers une fonction du gestionnaire d'interruption. C'est l'IDT (*Interrupt Descriptor Table*).

Des techniques décrites dans les articles 4 et 14 du phrack 59[5] expliquent comment il est possible de détourner l'IRQ 1 qui gère les interruptions générées par le clavier (sur les architectures de type x86).

L'idée générale est de modifier l'adresse du gestionnaire d'interruption contenue dans l'IDT. Notre propre gestionnaire est invoqué à sa place, et peut alors mémoriser les lectures effectuées sur le port d'entrée/sortie du clavier, ainsi que sur le port d'état, nous permettant d'accéder au touches enfoncées ou relachées.

En fait, on se rend compte qu'il est tout à fait possible d'utiliser le même genre de techniques que lorsqu'on veut détourner un appel système. En effet, il existe des outils permettant en userland (à partir de `/dev/kmem`) d'afficher le contenu de l'IDT. On peut de la même manière ou à l'aide d'un module, modifier soit son contenu, c'est à dire le pointeur vers le gestionnaire d'interruption (cela impose de réécrire la partie assembleur du gestionnaire d'interruption, le gestionnaire intermédiaire), mais on peut aussi modifier l'adresse du gestionnaire réel (qui lui est écrit en C), que l'on peut retrouver car le gestionnaire intermédiare l'appelle au moyen d'un `call`. La technique du `JUMP`, décrite plus haut, est certainement elle aussi utilisable.

Par contre, le détournement d'interruptions présente un inconvénient majeur, il est propre à une architecture, et il n'est donc par conséquent pas portable.

3.3 Détection & contre-mesures

Si la *backdoor* est suffisamment bien faite, elle est presque impossible à détecter localement. Si l'on est un administrateur système et que l'on constate que le noyau est corrompu, le plus sage pour remettre en place les machines est (après avoir sauvegardé les données pour l'*autopsie*) de tout réinstaller *from scratch*. Mais si l'on est un vil pirate convaincu d'être sur un *honeypot*, on voudra peut-être supprimer Sebek (par exemple), jouer un peu avec la bête, et prendre ainsi l'administrateur à son propre piège.

Dans le cas où il est permis d'insérer des LKM, rien de plus simple. On va se contenter de recoder les fonctions sensibles telles qu'elles sont supposées être, puis modifier la table des appels système et/ou utiliser la technique du `JUMP` (voir 3.1).

Si l'on ne peut pas utiliser les LKM, il va falloir se tourner vers des techniques beaucoup plus délicates, par exemple celle consistant à lire (pour détecter), et éventuellement écrire (pour mettre la *backdoor* hors d'état de nuire) directement dans `/dev/kmem`.

[5] L'article 14 du phrack 59 présente surtout d'autres endroits dans lesquels il est possible de se placer pour effectuer un *keylogger*, autres que les interruptions

Que ce soit pour les LKM ou pour le périphérique `/dev/kmem`, les techniques sont les mêmes dans ce cas que dans celui de l'insertion du code (voir la partie 2).

Pour le cas des BSD, nous avons vu que le système peut passer le `securelevel` à `1` lors du démarrage (le *super-utilisateur* peut aussi le faire à tout moment), ce qui est une action irréversible (seul `init` peut diminuer cette valeur). Celle-ci empêche, entre autres, d'insérer un module, ainsi que d'écrire dans les périphériques `/dev/mem` et `/dev/kmem` (pour les effets précis des différentes valeurs possibles de `securelevel`, voir, selon le système, `securelevel(7)`, *OpenBSD Reference Manual*, `init(8)`, *FreeBSD System Manager's Manual*, et `init(8)`, *NetBSD System Manager's Manual*).

Notons que si OpenBSD passe le `securelevel` à `1` par défaut, ce n'est pas le cas de FreeBSD ni de NetBSD. Cependant, il est possible de configurer le système pour qu'il le fasse (cela se passe dans le fichier `/etc/rc.conf` qui permet d'écraser les valeurs par défaut du fichier `/etc/defaults/rc.conf`). Cette manipulation est simple, et rend la machine nettement plus sûre ; même si cela oblige parfois à réamorcer le système, et, sauf cas exceptionnel (pot à miel, programmation LKM, système haute disponibilité, ...), il n'y a pas de raison valable pour être en `securelevel -1` (*permanently insecure*).

4 Communication depuis le noyau

Après avoir présenté dans la section précédente quelques moyens d'obtenir des informations sur le système, nous allons nous intéresser dans cette section à la manière de faire sortir ces informations mais aussi d'en recevoir de l'extérieur.

4.1 Emettre des trames

Selon la configuration dans laquelle on se trouve, on agira juste au dessous du niveau deux (ethernet ou ppp par exemple) ou trois (ip). La création d'un paquet dans le noyau est réalisée par encapsulations successives, des couches supérieures vers les couches basses ; il y a donc plusieurs couches parcourues lorsqu'une application userland souhaite émettre une trame sur le réseau.

Dans le cas de Sebek, nous agissons directement au dessous de la couche ethernet, juste au dessus du driver, sans passer par les couches supérieures (l'inconvénient est que son utilisation est limitée à une interface ethernet), ce qui permet notamment de s'abstraire des règles de filtrage qui peuvent être en place sur la machine et qui n'affectent que les trames construites par l'intermédiaire de la pile réseau.

Se placer plus haut peut simplifier la création des paquets, mais le cheminement de ceux-ci est alors grandement dépendant de la configuration réseau de la machine (table de routage, firewall, ...) et cette solution, qui a été retenue pour l'implémentation de Sebek-OpenBSD, devient par conséquent moins furtive et moins efficace.

Plus précisément, une partie de la furtivité s'obtient en forgeant soi-même les paquets contenant les données et en insérant ces trames de niveau deux valides directement dans la file d'envoi de l'interface considérée. Par exemple sous BSD, après avoir créé la trame dans la structure adaptée du noyau (une chaîne de structures `mbuf`), celle-ci est placée par la macro `IF_ENQUEUE` dans la file de l'interface. Ensuite, son émission sur le réseau est réalisée à l'aide d'une fonction spécifique au matériel (le pointeur vers cette fonction est stocké dans la structure de l'interface).

4.2 Recevoir des commandes

Nous allons voir deux façons différentes de recevoir des commandes. Il s'agit de faire remonter à notre code (dans le kernel) des données à traiter. Nous n'allons pas aborder ici ce que l'on peut faire de cette communication, mais plutôt des méthodes pour l'établir.

On souhaite pouvoir recevoir des commandes qui proviennent du réseau. On pense donc *a priori* à se placer dans des fonctions de la pile réseau du système. Plus on se place bas, plus on évite ce qui pourrait causer des ennuis (*firewall* par exemple).

Pour éviter que ce trafic soit repéré, on peut placer les commandes soit dans du trafic apparaissant légitime (cela dépend de l'utilisation qui est faite de la machine), soit masquer le trafic qui la contient (il faudra alors se tourner, en plus de la pile réseau, vers `bpf`, voir 4.3). Cependant, si l'on se place au niveau du driver, cela ne peut être intéressant que lors d'une attaque très ciblée (en contre-partie, ce sera sans doute le plus efficace).

Une solution alternative est encore de se placer dans la fonction `read`. Il faut, pour que cela fonctionne, qu'un programme dans l'espace utilisateur récupère les paquets, c'est à dire avoir un service qui les recoive, ou bien un programme qui les dumpe (de type `libpcap`). L'intérêt est que l'on peut alors cacher les commandes, en limitant les traces dans les logs. Par exemple, le démon OpenSSH ne loggue rien tant qu'aucune tentative d'authentification n'est effectuée. On peut donc envoyer la commande à la place de la ligne de version qui débute l'échange ssh.

4.3 Masquer notre trafic

La figure 2 décrit l'organisation de la pile réseau sur les BSD. Les paquets sont créés dans les couches hautes et sont passées au couches inférieures, jusqu'au driver. Ce dernier duplique les paquets entrants et sortants, pour les passer à bpf. Son rôle est de pouvoir analyser le traffic circulant sur l'interface, en appliquant des filtres qui correspondront ou pas à certains paquets que l'on désire analyser. Ce trafic est perçu par les outils utilisant, par exemple, la fameuse `libpcap`[6].

De la même manière que pour recevoir des commandes, il convient entre autres de se placer dans la fonction `bpf_filter()` pour cacher les paquets devant

[6] Une librairie pour la capture de paquets, utilisée entre autres par `tcpdump`.

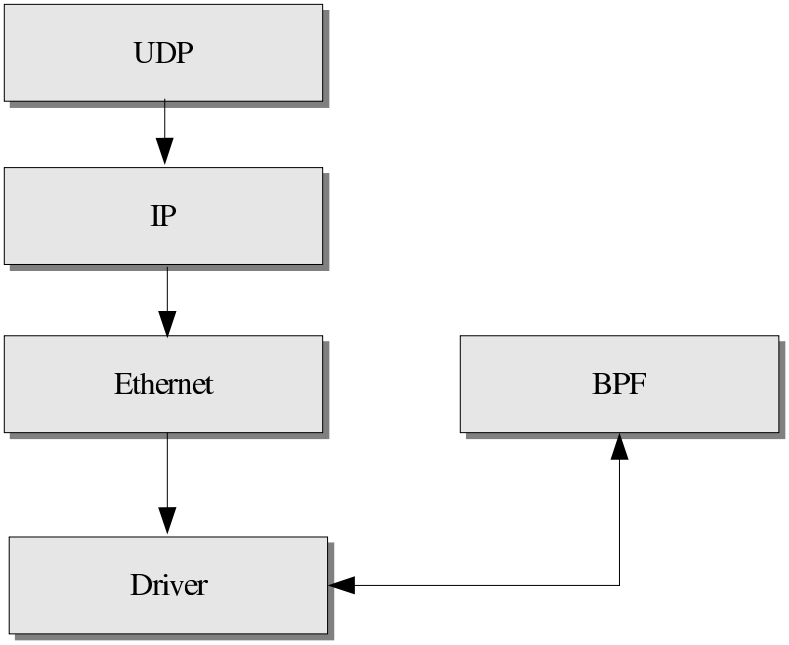

Fig. 2. Placement de bpf par rapport à la pile réseau.

être furtifs. Il suffit pour cela de faire retourner cette fonction avant qu'elle n'agisse, si le paquet doit être masqué.

Cependant, cela n'est pas vraiment suffisant pour bien faire, et il va falloir se placer plus bas. En effet, le paquet, lorsqu'il est soumis pour approbation à `bpf_filter()`, a déjà été comptabilisé (on peut voir le compteur à la fin d'une session `tcpdump` sous la forme `x packets received by filter, y packets dropped by kernel`).

Par exemple, dans les codes de Sebek-⋆BSD que nous avons développés, le filtre se trouve dans les fonctions du fichier `bpf.c` faisant appel à `bpf_filter()`, et qui comptabilisent les paquets. Juste avant d'incrémenter le compteur et d'appeler `bpf_filter()`, on vérifie si le paquet doit être filtré.

Conclusion

Que l'on soit dans la situation d'un pot à miel avec un outil de type Sebek, ou dans celle d'une machine potentiellement corrompue, la guerre entre celui qui veut altérer le noyau et celui qui souhaite un noyau *propre* (ou corrompu différemment, bien entendu) n'a aucune limite, si ce n'est le degré de paranoïa des intéressés, leur habileté technique, et leur imagination.

Si certaines techniques utilisées sont *fingerprintables*, il est difficile, voire impossible (particulièrement à distance), d'être absolument certain qu'un noyau est propre. De plus, on ne doit pas oublier que le fait que le noyau soit corrompu n'empêche pas, au contraire, que les binaires essentiels le soient aussi. Dans le cas d'une suspicion, il vaut mieux, avant de reprendre l'activité, tout réinstaller à partir de média sains.

La solution qui nous semble la plus efficace pour prévenir la mise en place de tels systèmes est l'utilisation de mécanismes semblables aux `securelevel` des BSD.

Pour terminer, nous tenons à remercier Laurent Oudot (Team RSTACK[7]), pour son encadrement dans l'écriture du port de Sebek2 pour les BSD. Merci aussi au French Honeynet Project[8].

[7] `http://www.rstack.org/`
[8] `http://www.frenchhoneynet.org/`