Linux Kernel Rootkit

Castets Nathan, Huge Olivier

***Abstract*—Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.**

I. Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

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 différents 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. 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 nombres 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.

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. Ceci évite que l'adresse d'un appel système soit présente dans le noyau.

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.

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* représente la KASLR). 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) :

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

On remarque un appelle 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) :

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

On a trouvé notre appel à la *sys_call_table* !

Ces différentes fonctions sont présentes sous forme de bytecode dans la mémoire noyau. Notre objectif est de partir de *MSR_LSTAR* dans la mémoire noyau et de suivre les *call* ou *jmp* jusqu'à arriver à l'appel de la *sys_call_table*. Pour les version 4.17 à 4.19, *MSR_LSTAR* ne pointant pas directement vers *entry_SYSCALL_64*, nous allons donc utiliser 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 - 4.20) :

```
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 a préalablement désactivé la KASLR pour avoir l'adresse fixe des différentes fonctions et structures. Cela nous permet de plus facilement repérer les appels dans le bytecode. On cherche une suite de bytes de la forme suivante :

```
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, voici 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
```

A ce stade là, nous avons calculé l'adresse de la fonction *do_syscall_64*. On ré-itère le procédé, on analyse la suite de byte. 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. Aprèns une analyse des suites de bytes, les patterns 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 l'adresse de la table des appels système 32bits. Elle est là pour permettre la rétro-compatibilité des binaires 32bits. On continue de regarder le fichier /arch/x86/entry/common.c on dans la fonction *do_syscall_32_irqs_on* on observe le code suivant :

```
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);
```

Cette fonction étant à la suite de *do_syscall_64*, il suffit de continuer à regarder la suite de byte. On cherche un *mov* de la forme suivante :

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

Les patterns qui précèdent 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
```

REFERENCES

[1] Linux kernel patch 4.17
https://github.com/torvalds/linux/commit/2ca2a09d6215fd9621aa3e2db7cc9428a61f2911#diff-acc7893dfa77092a11e1b53e98a34d44

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

[3] Linux kernel
https://github.com/torvalds/linux/blob/master/arch/x86/entry/entry_64.S

[4] Linux kernel
https://github.com/torvalds/linux/blob/master/arch/x86/entry/common.c

[5] Linux kernel
https://github.com/torvalds/linux/blob/master/arch/x86/kernel/cpu/common.c