Linux Kernel Rootkit

Castets Nathan & Huge Olivier

Université de Bordeaux

21 Février 2019

Overview

Définitions

Rootkit

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'un malware classique, le rootkit se veut discret et dissimule au maximum ses actions à l'utilisateur et aux programmes de surveillance.

Définitions

Il existe 2 types de rootkit :

- Espace utilisateur
  Remplace des fonctions utilisées par un programme
  Injection de librarie dynamique via *LD_PRELOAD*
- Espace noyau
  Remplace des appels systèmes
  Module noyau qui écrase la table des appels systèmes

Définitions

Table des appels systèmes

Tableau contenant les adresses mémoires des fonctions associées aux appels systèmes. Ces appels systèmes permettent aux programmes de l'espace utilisateur de communiquer avec le noyau.

Les appels systèmes sont indispensables pour les programmes de l'espace utilisateur pour utiliser des fonctions que seul le noyau peut exécuter.
On appelle aussi la table des appels systèmes la *sys_call_table*.

Définitions

KASLR

La KASLR (Kernel Address Space Layout Randomization) est une sécurité du noyau qui charge aléatoirement les données dans la mémoire.

Cela implique qu'à chaque démarrage du système une structure de donnée n'est généralement pas à la même adresse.
C'est la sécurité principale qui empêche les rootkits de s'installer dans le système.

Pré Linux Kernel 4.17

/fs/open.c :

```
/* *** */

EXPORT_SYMBOL(sys_close);

/* *** */
```

La fonction associée à l'appel système *sys_close* est accessible par n'importe quel programme présent dans le noyau.
Cet export est présent car le module *mount* a besoin de *sys_close*.

Un brute-force de la mémoire noyau à la recherche des occurences de l'adresse de *sys_close* nous donne la *sys_call_table*.

Post Linux Kernel 4.17

- Suppression de la majorité des appels systèmes dans le code noyau
  L'export de la fonction *sys_close* n'existe plus
- Rajout de fonction avec un comportement similaire *ksys_xyzxyz()*
  Le but étant de dissocier au maximum les appels venants de l'espace utilisateur et noyau

Cela implique :

- Qu'il n'est plus possible de brute-force la *sys_call_table* à l'aide de l'adresse d'un appel système
- Qu'il n'est plus possible d'altérer le comportement de programme présent dans le noyau

Déterminer l'adresse de la table des appels systèmes

L'idée est de s'intéresser au fonctionnement des appels systèmes et plus précisément au code exécuté en préambule pour préparer l'appel système.

Retracer ce code dans la mémoire noyau jusqu'à retrouver un offset vers la *sys_call_table*.

Nous nous concentrerons sur les version 4.17 à 4.20 du noyau Linux dans la suite de cette présentation.

Déterminer l'adresse de la table des appels systèmes

Dès qu'un appel système est levé, le processeur doit exécuter du code pour préparer cet appel système. L'adresse de ce code se trouve dans le registre *MSR_LSTAR*. Voyons à l'initialisation ce que contient ce registre.

/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);
```

Déterminer l'adresse de la table des appels systèmes

/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
```

Déterminer l'adresse de la table des appels systèmes

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

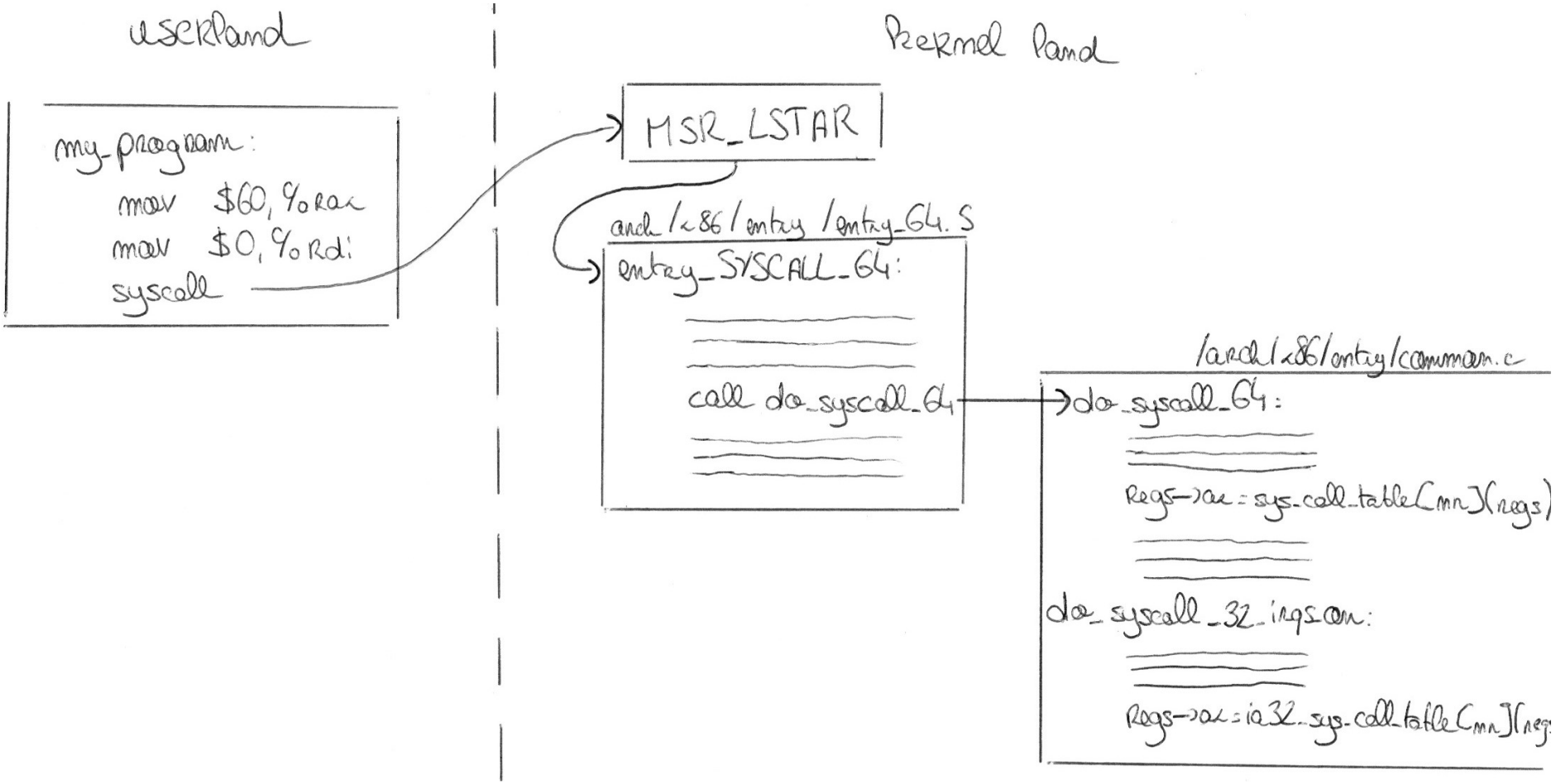
```
__visible void do_syscall_64(unsigned long nr, struct pt_regs
    *regs)
{
  /* *** */
  nr &= __SYSCALL_MASK;
  if (likely(nr < NR_syscalls)) {
    nr = array_index_nospec(nr, NR_syscalls);
    regs->ax = sys_call_table[nr](regs);
  }
  /* *** */
}
```

Déterminer l'adresse de la table des appels systèmes

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

```
static __always_inline void do_syscall_32_irqs_on(struct
    pt_regs *regs)
{
  /* *** */
  if (likely(nr < IA32_NR_syscalls)) {
    nr = array_index_nospec(nr, IA32_NR_syscalls);
  regs->ax = ia32_sys_call_table[nr](regs);
  /* *** */
}
```

Déterminer l'adresse de la table des appels systèmes



Patterns pour retrouver l'offset de *sys_call_table*

Tout d'abord il nous faut l'adresse de la fonction *entry_SYSCALL_64* :

- En version 4.20 il nous suffit de lire le registre *MSR_LSTAR*
- Dans les versions 4.17 - 4.19, on pourrait aussi lire le registre *MSR_LSTAR* et suivre le code exécuté jusqu'à atteindre *entry_SYSCALL_64*

Astuce

La fonction *native_load_gs_index* qui se trouve juste en dessous de *entry_SYSCALL_64* dans le code est exportée via un *EXPORT_SYMBOL*.

Patterns pour retrouver l'offset de *sys_call_table*

Dans *entry_SYSCALL_64* on cherche l'appel à *do_syscall_64* :

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

Il est précédé par les instructions suivantes :

```
4.17 - 4.20
41 57                   push %r15
45 31 ff                xor %r15d, %r15d
48 89 c7                mov %rax, %rdi
48 89 e6                mov %rsp, %rsi
```

Patterns pour retrouver l'offset de *sys_call_table*

Dans *do_syscall_64* on cherche l'appel à *sys_call_table* :

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

Il est précédé par les instructions suivantes :

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

Patterns pour retrouver l'offset de *sys_call_table*

Dans *do_syscall_32_irqs_on* on cherche l'appel à *ia32_sys_call_table* :

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

Il est précédé par les instructions suivantes :

```
4.17
48 19 d2                sbb %rdx, %rdx
21 d0                   and %edx, %eax
4.18 - 4.20
48 19 d2                sbb %rdx, %rdx
21 d0                   and %edx, %eax
48 89 ef                mov %rbp, %rdi
```

Cacher des fichiers à l'utilisateur

Appel système *getdents* :

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

structure *linux_dirent* :

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

Définition

Virtuelle

La mémoire virtuelle est la mémoire que les processus utilisent pour adresser leurs objets.

Physique

La mémoire physique est la mémoire réellement adressée par le système.

La mémoire est en plus de cela découpée en Page de 4KB.

Définition

MMU

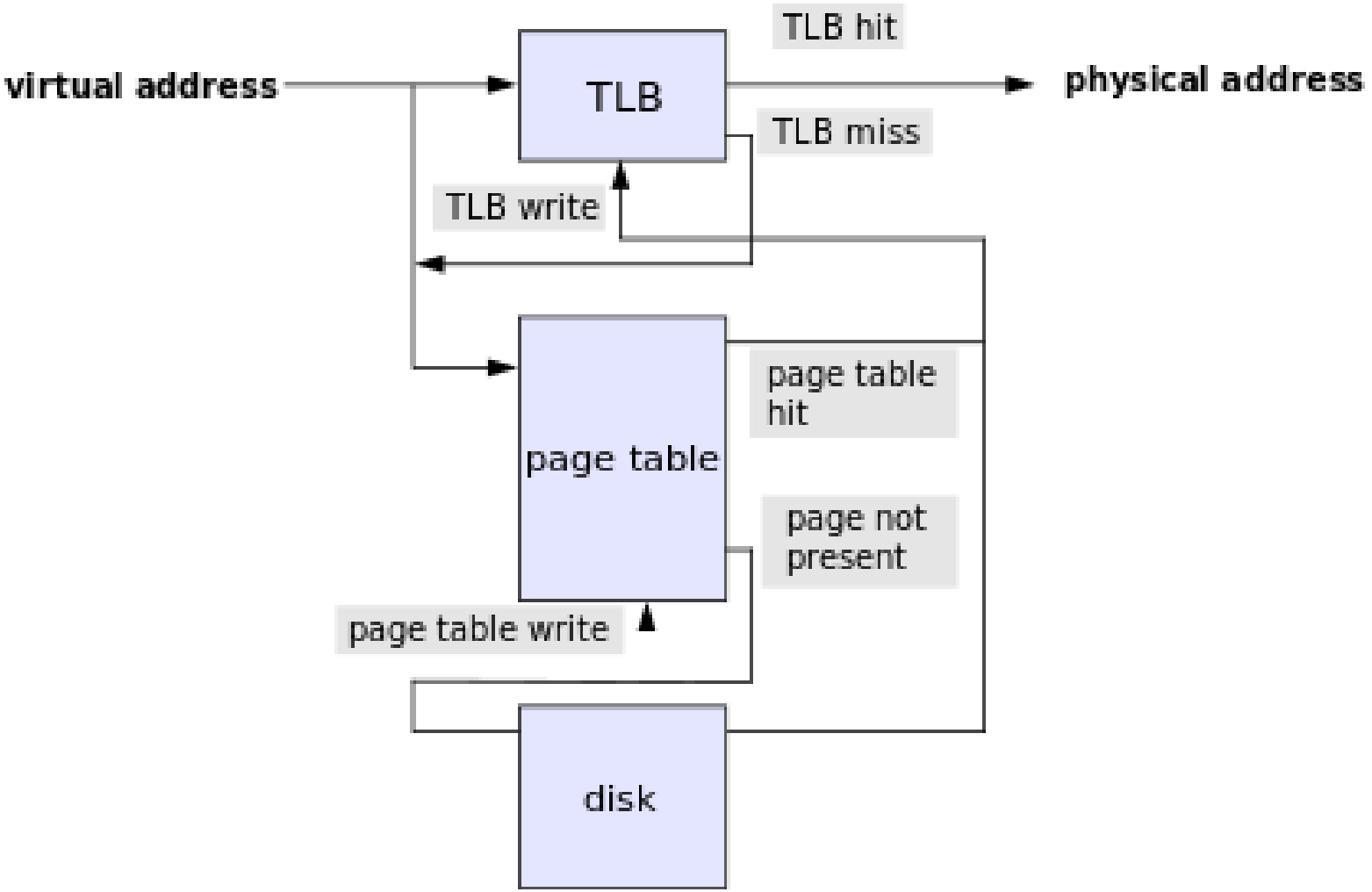
Le MMU(memory management unit) est un composant hardware qui gère les translations adresse virtuelle/physique.

TLB

Cache permettant d'accélérer la translation d'adresse, appartient au MMU.

Fonctionnement

Le MMU fonctionne de la façon suivante :



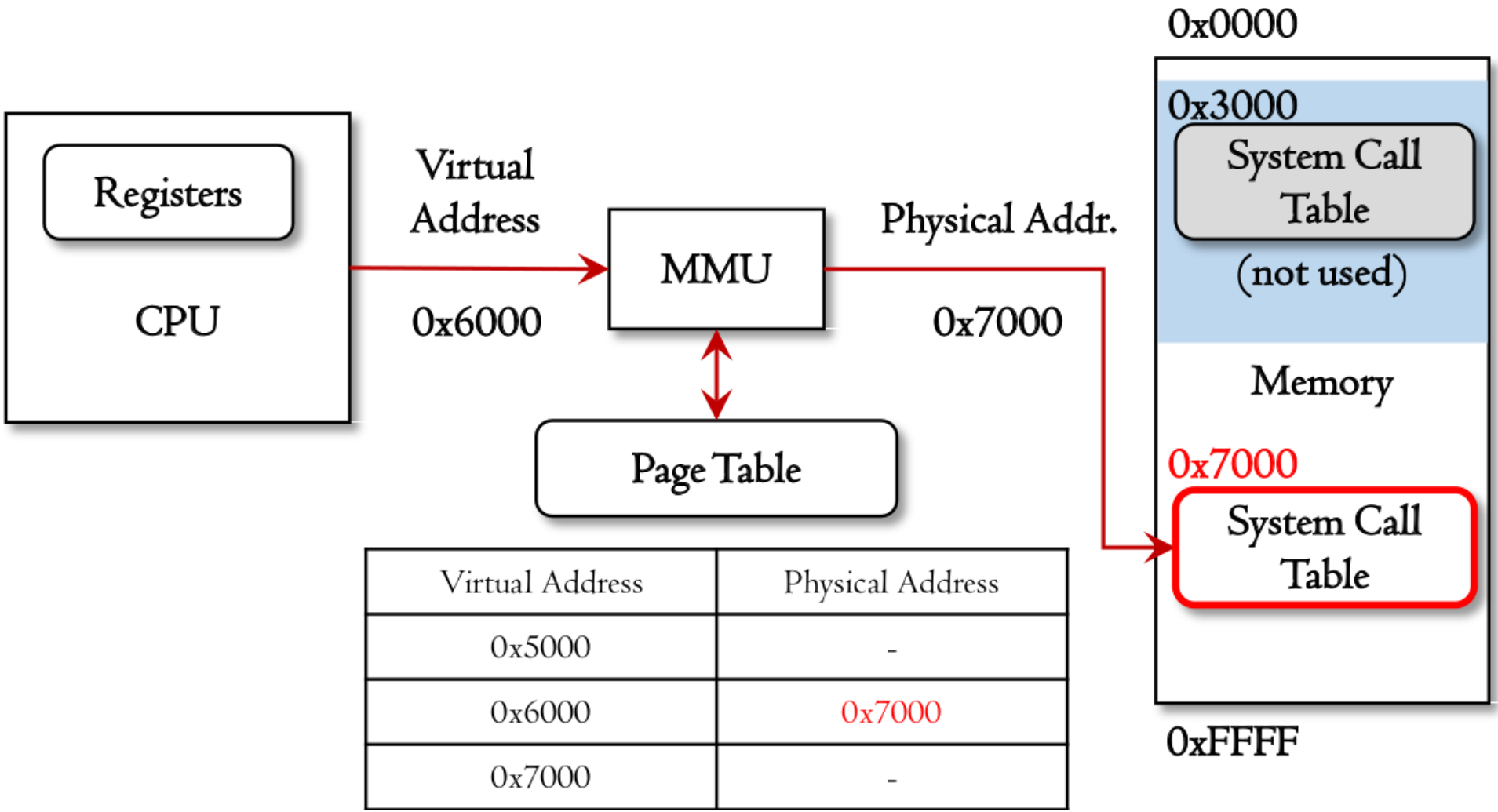
Fonctionnement

Pour savoir dans quel contexte mémoire on se trouve, chaque processeur dispose d'un registre CR3, celui-ci contient l'adresse du PGD pour le processus courant. Il contient aussi les différents droits et modes des pages.

ATRA

Cette attaque ne permet pas de récupérer l'adresse de la syscall table, mais on peut une fois celle-ci trouvée rendre notre rootkit plus furtif. En effet, cette attaque cherche à dupliquer la syscall table, et de faire utiliser la copie par le système.

ATRA schéma



Virtual Address	Physical Address
0x5000	-
0x6000	0x7000
0x7000	-

CR3

Nous savons que les structures kernel sont partagées dans tous les processus, donc dans la mémoire de chaque processus il y a un pointeur sur la syscall table.

Le but de notre attaque est de retrouver la syscall en scannant les page table de notre processus et en cherchant ce pointeur.

Conclusion

- Technique courante utilisant l'export de la fonction *sys_close*
- Version 4.17 du noyau rendant cette technique obselète
- Développement d'une technique alternative basée sur la recherche d'un appel à la *sys_call_table* dans le code en mémoire
- Exemple de hook de *getdents* pour cacher des fichiers à l'utilisateur

Références I


Sources du projet
github.com/naka53/prime


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


Linux Kernel Sources
github.com/torvalds/linux


Memory and Page table
kernel.org/doc/gorman/html/understand


ATRA
cysec.kaist.ac.kr/publications/ACM_CCS_2014_ATRA.pdf