RODRIGO RUBIRA BRANCO,
FILIPE ALCARDE BALESTRA
SCANIT

Hacking du noyau & Anti-forensics

Degré de difficulté

Cet article explique, pourquoi une analyse forensic dans un système live n'est pas recommandé et pourquoi l'image de ce système peut déclencher une anti-forensic-capable rootkit avancée.

CET ARTICLE EXPLIQUE...

Avec cet article vous allez mieux comprendre le fonctionnement de l'architecture d'un ordinateur et sa dépendance vis-à-vis des systèmes d'exploitation, on se focalisera notamment sur le processus d'acquisition mémoire étant donné que les structures internes l'utilisent avec la gestion mémoire, le système de fichier et d'autres éléments seront eux aussi expliqués, en guise d'illustration nous utiliserons un système d'exploitation Linux, cependant on tentera d'avoir un certain recul afin que vos travaux soient reproduisibles sur d'autres plateformes.

CE QU'IL FAUT SAVOIR...

Pour comprendre parfaitement cet article il est recommandé que le lecteur connaisse les bases de la programmation du Kernel sous Linux(comment créer des modules, le fonctionnement basique de la programmation Kernel) mais également un peu d'assembleur et langage C. L'architecture interne sera détaillée, mais une expérience préalable en informatique ou dans l'ingénierie est indispensable pour une bonne compréhensiondes illustrations.

Un aperçu du noyau interne de la structure et du fonctionnement de l'architecture x86 sera également abordée, ainsi que les différences entre les autres architectures.

Un grand nombre d'outils [5] ont été développés pour analyser le système live afin de détecter une intrusion (comme les rootkits installés [7]).

Cet article tente d'expliquer certaines des présentations [8] ont montré les problèmes existants dans ce modèle, en expliquant les risques de cet acte et quand cela peut être toléré.

Les bases

L'architecture choisie est Intel x86, où les concepts s'appliquent également à d'autres architectures (les modifications majeures nécessaires dans les architectures sans MMU).

Pour mieux comprendre les sections suivantes, certains concepts de base sont nécessaires :

- CPL0 et il est important,
- Appels Système,
- Structures analysées à la gestion de mémoire,
- amorçage de fonctions et flux d'information.

CPL0 et son importance

L'architecture Intel a plusieurs niveaux de priorité, et les systèmes d'exploitation modernes (Linux /Windows/MacOs) utilisent cette séparation pour fournir une protection et l'isolement de chaque processus (donc un processus ne peut interférer dans l'exécution d'un autre, ni dans l'exécution du système d'exploitation lui-même).

Le système d'exploitation est exécuté dans le CPL0 (également connu comme kernel-mode ou `ring0`) parce que, dans ce mode, n'importe quelle opération privilégiée est autorisée (accès à la mémoire, gestion matériel..., etc.).

Dans cet article les systèmes d'exploitations micro-noyau sont ignorés pour faciliter le processus d'apprentissage. Il est important de comprendre que les applications utilisateur fonctionnent dans CPL3 (mode- utilisateur ou ring3).

Appels Système

Quand un programme mode utilisateur a besoin de certaines ressources privilégiées (par exemple, lire diskdata) il exécute un appel système. C'est une interruption logicielle qui transforme le système en kernelmode, exécutant le gestionnaire d'appel système pour renvoyer la commande au programme d'usermode.

La façon dont les appels système sont traités est complètement dépendante de l'architecture.

Le facteur commun est que chaque implémentation a des structures semblables, utilisant des méthodes différentes, utilisant bibliothèques et autres ressources.

Par la suite, nous examinerons la manière dont cela fonctionne dans une architecture

x86 (utilisant l'instruction int $0x80 et la nouvelle façon utilisant sysenter). Nous abordons également comment la même chose peut être appliquée dans l'architecture Power, juste pour avoir un aperçu des différences.

int $0 x80

Pour mieux comprendre cette instruction, il faut savoir qu'un outil exécutera une fonction haut-niveau aura besoin d'un appel système (par exemple, une fonction implémentée en C pour lire un fichier de données), quelqu'un peut mettre en application cela directement en assembleur, ainsi cette étape sera sautée.

La bibliothèque C (dans notre échantillon) va transformer l'appel à un appel système de la façon suivante :

- Le numéro d'appel système sera dans le registre EAX,
- Les paramètres sont transmis en utilisant les registres EBX, ECX et EDX (utilisation de la pile s'il y a plus de paramètres).

L'outil *fait* appel à int80, qui est une interruption logicielle chargée de passer le contrôle au mode-noyau (dans le gestionnaire des appels système). Le système d'exploitation durant le processus d'initialisation enregistrera une table d'interruption (IDT – table de description d'interruption) et les gestionnaires d'interruption (les fonctions qui seront exécutées quand une interruption spécifique est reçue). Dans ce cas, l'interruption int80 appelle le gestionnaire *system_call*. Pour localiser où l'IDT se situe dans la mémoire, on utilise l'instruction sidt.

Le gestionnaire d'appel système va vérifier le registre EAX et appeler le gestionnaire spécifique de cet appel système. Ce gestionnaire sera trouvé dans un vecteur appelé `sys_call_table[EAX]` (note : la valeur EAX sera utilisé comme un indice dans ce vecteur pour déterminer la fonction).

La prochaine étape est un appel à la fonction spécifique pour répondre à l'appel système. Maintenant, la fonction va exécuter ce qui est nécessaire (par exemple, copier des données du mode utilisateur en utilisant `copy_from_user()` ou à l'utilisateur en utilisant `copy_to_user()`) et ensuite on va rendre le contrôle à l'application (il y a quelques complications, comme non-blocage d'appels système et autres qui vont seront ignorées ici).

vsyscalls (sysenter)

La documentation Intel (IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference) met l'accent dans le fait que l'instruction, ainsi que sysexit, a été créé pour optimiser le

Listing 1. *cat /proc/self/maps*

```
rbranco@rrbranco:~$ cat /proc/self/maps
08048000-0804c000   r-xp   00000000   03:06 652506   /bin/cat
0804c000-0804d000   rw-p   00003000   03:06 652506   /bin/cat
0804d000-0806e000   rw-p   0804d000   00:00 0   [heap]
a7e83000-a7e84000   rw-p   a7e83000   00:00 0
a7e84000-a7fcb000   r-xp   00000000   03:06 736624   /lib/i686/cmov/libc-2.7.so
a7fcb000-a7fcc000   r--p   00147000   03:06 736624   /lib/i686/cmov/libc-2.7.so
a7fcc000-a7fce000   rw-p   00148000   03:06 736624   /lib/i686/cmov/libc-2.7.so
a7fce000-a7fd1000   rw-p   a7fce000   00:00 0
a7fe2000-a7fe4000   rw-p   a7fe2000   00:00 0
a7fe4000-a8000000   r-xp   00000000   03:06 734302   /lib/ld-2.7.so
a8000000-a8002000   rw-p   0001b000   03:06 734302   /lib/ld-2.7.so
affeb000-b0000000   rw-p   affeb000   00:00 0   [stack]
ffffe000-fffff000   p      00000000   00:00 0   [vdso]
```

Listing 2. *ldd /bin/bash*

```
rbranco@rrbranco:~$ ldd /bin/bash
   linux-gate.so.1 =>  (0xffffe000)
   libncurses.so.5 => /lib/libncurses.so.5 (0xa7f90000)
   libdl.so.2 => /lib/i686/cmov/libdl.so.2 (0xa7f8c000)
   libc.so.6 => /lib/i686/cmov/libc.so.6 (0xa7e3e000)
   /lib/ld-linux.so.2 (0xa7fe4000)
```

Listing 3. *vsyscall vidage de la mémoire*

```
rbranco@rrbranco:~$ dd if=/proc/self/mem of=rrbranco.dso bs=4096 skip=1048574 count=1
1+0 records in
1+0 records out
4096 bytes (4.1 kB) copied, 5e-05 seconds, 82 MB/s
rbranco@rrbranco:~$ objdump -d --start-address=0xffffe400 --stop-address=0xffffe414
rrbranco.dso rrbranco.dso:   file format elf32-i386
Disassembly of section .text:
ffffe400 <__kernel_vsyscall>:
ffffe400:   51   push   %ecx   -> Save %ecx in the stack
ffffe401:   52   push   %edx  -> Save %edx in the stack
ffffe402:   55   push   %ebp  -> Save %ebp in the stack
ffffe403:   89   e5  mov   %esp,%ebp  -> Save the %esp content in %ebp,
      permiting the user-mo
ffffe405:   0f 34   sysenter  -> Execute the sysenter instruction
ffffe407:   90   nop
ffffe408:   90   nop
ffffe409:   90   nop
ffffe40a:   90   nop
ffffe40b:   90   nop
ffffe40c:   90   nop
ffffe40d:   90   nop
ffffe40e:   eb f3   jmp   ffffe403 <  kernel_vsyscall+0x3>
ffffe410:   5d   pop   %ebp
ffffe411:   5a   pop   %edx
ffffe412:   59   pop   %ecx
ffffe413:   c3   ret
```

Listing 4. *Adresse ancrées*

```
. = 0xc00    -> The anchored address
SystemCall:
EXCEPTION_PROLOG
EXC_XFER_EE_LITE(0xc00, DoSyscall)
```

transfert au mode noyau (et le retour après ça). Beaucoup de valeurs de configuration sont placés par le système d'exploitation dans les MSRs (registres modèle-spécifiques) pour l'instruction de sysenter :

```
-CS (SYSENTER_CS_MSR) -EIP
(SYSEN-TER_EIP_MSR
-SS (SYSENTER_CS_MSR + 8)
-ESP (SYSENTER_ESP_MSR
```

L'instruction de sysexit transférera la commande de nouveau au mode-utilisateur et définit les registres suivants :

```
#define MSR_IA32_SYSENTER_CS 0x174
#define MSR_IA32_SYSENTER_ESP 0x175
#define MSR_IA32_SYSENTER_EIP 0x176
```

(Dans Linux elle sont définis dans : *asmmsr.h*).

Le noyau Linux TSS (*Task State Segment*) pour l'utilisation des instructions in-out en mode *utilisateur* (contrôle de permissions bitmap) dans l'architecture Intel pour passer d'usermode au *kernelmode*, on doit connaître que la pile est utilisée par le *kernelmode*.

Listing 5. *cat /proc/self/map*

```
$ cat /proc/self/maps
08048000-0804c000   r-xp   00000000   03:06   652506   /bin/cat
0804c000-0804d000   rw-p   00003000   03:06   652506   /bin/cat
0804d000-0806e000   rw-p   0804d000   00:00   0    [heap]
a7ea6000-a7ea7000   rw-p   a7ea6000   00:00   0
a7ea7000-a7fce000   r-xp   00000000   03:06   700482   /lib/tls/i686/cmov/
                                                        libc-2.3.6.so
a7fce000-a7fd3000   r—p    00127000   03:06   700482    /lib/tls/i686/cmov/
                                                        libc-2.3.6.so
a7fd3000-a7fd5000   rw-p   0012c000   03:06   700482   /lib/tls/i686/cmov/
                                                        libc-2.3.6.so
a7fd5000-a7fd8000   rw-p   a7fd5000   00:00   0
a7fe9000-a7feb000   rw-p   a7fe9000   00:00   0
a7feb000-a8000000   r-xp   00000000   03:06   733005   /lib/ld-2.3.6.so
a8000000-a8002000   rw-p   00014000   03:06   733005   /lib/ld-2.3.6.so
affeb000-b0000000   rw-p   affeb000   00:00   0    [stack]
ffffe000-fffff000   p      00000000   00:00   0    [vdso]
```

Listing 6. *vm_area_struct*

```
struct vm_area_struct {
struct mm_struct * vm_mm; /* The address space we belong to. */
unsigned long vm_start; /* Our start address within vm_mm. *
unsigned long vm_end; /* The first byte after our end address within vm_mm. */
/* linked list of VM areas per task, sorted by address */
struct vm_area_struct *vm_next;
pgprot_t vm_page_prot; /* Access permissions of this VMA. */
unsigned long vm_flags; /* Flags, listed below. */
}
```

Listing 7. *Changer la permission de mémoire*

```
static int change_perm(unsigned *addr) {
   struct page *pg;
   pgprot t_prot;
   pg = virt_to_page(addr);
   prot.pgprot = VM_READ | VM_WRITE | VM_EXEC; /* R-W-X */
   change_page_attr(pg, 1, prot);
   global_flush_tlb() ;
   return 0;
}
```

Listing 8. *Exécute le code à partir du mode-noyau*

```
static int execute(const char *string) {
   if ((ret = call_usermodehelper(argv[0], argv, envp, 1)) != 0) {
   printk(KERN_ERR "Failed to run "%s": %i\n", string, ret);
   } return ret;
}
```

Ainsi, Linux définit (dans : *archi386kern elsysenter.c*) :

- `wrmsr(MSR _ IA32 _ SYSENTER _ CS, _ _ KER-NEL _ CS, 0);` > pointant sur le segment du noyau,
- `wrmsr(MSR _ IA32 _ SYSENTER _ ESP, tss->esp1, 0);` > pointant sur la mémoire du noyau,
- `wrmsr(MSR _ IA32 _ SYSENTER _ EIP,` (unsigned long) `sysenter _ entry, 0);` > pointant sur la page défini comme le point d'entrée sysenter.

En fait, quand une instruction sysenter est reçue, le système va commencer à utiliser la pile du noyau et d'exécuter la fonction `sysenter _ entry`.

Cette page doit être liée à l'espace d'adresse de tous les processus dans le système, et Linux effectue ça (en : `archi386kernelvsyscall-sysenter.s`), utilisant un VDSO (*Virtuel Dynamic Shared Object*). Pour vérifier ça dans un système voir Listing 1.

Dans les applications où des bibliothèques partagées sont employées, la commande de ldd peut également être utilisée, voir Listing 2.

Pour copier un emplacement de la mémoire afin de vérifier ce qu'elle contient, voir Listing 3.

Le `sysenter _ entry` (definie en : `archi386kernelentry.s`) va travailler de la même façon que le gestionnaire `system _ call` montré auparavant. Utiliser la valeur `%eax` comme un index pour `sys _ call _ table`, qui détient les adresses des gestionnaires.

Power Architecture

Dans une *Power architecture* il n'existe pas de structure IDT contenant l'interruption gestionnaires d'adresses dans la mémoire. Au lieu de cela, il y a des interruptions ancrées à des adresses fixes, ou en d'autres termes, quand une interruption se produit, le contrôle sera *automatiquement* transférée à un emplacement spécifique de la mémoire. Notez que, par exemple, les interruptions *time* vont aller à l'adresse 0x900 comme on peut le voir dans le noyau de Linux `arch/ppc/kernel/head.S: EXCEPTION(0x900`, Decrementer, `timer _ interrupt, EXC _ XFER _ LITE)` où le décrémenter est défini (dans *Power architectures* le *timer* décrémenter

à la même fréquence d'horloge que le processeur, puisqu'il est interne dans le processeur), et autres interruptions externes sont ancrés à l'adresse *0x500*, et sont interprétées dans la même façon que les IDT dans l'architecture Intel. Voir Listing 1, 2.

Le gestionnaire d'appelles système est définie dans *arch/ppc/kernel/head.S* comme vous pouvez le voir dans le Listing 4.

Structures analysées pour la gestion de la mémoire

Une autre chose importante qui doit être compris est le processus de gestion de la mémoire dans les systèmes d'exploitation. Cet article montrera seulement ce qui est nécessaire pour le référencement.

Dans l'architecture Intel nous avons des pages de 4Ko (en réalité, elle peut être plus, selon le système, mais cela n'est pas important dans cette discussion). Pour un processus, la mémoire est considérée comme une adresse linéaire, de 0 à 4 Go (dans les architectures 32 bits).

Tous les pages mémoire d'un processus sont traduites en pages physique utilisant une page de table spécifique pour chaque processus. Il y a également d'autres informations dans cette structure, comme la page de protection des attributs (lecture-seule, exécutable, écriture).

Ces attributs pourraient facilement être modifiés si l'accès au cœur du système d'exploitation es possible.

Une mémoire visible pour le processus est divisé en deux grandes parties, en utilisant une constante `TASK_SIZE` (par défaut comme *0xc000000*) pour définir la plus grande adresse qui sera utilisée (après cela c'est le noyau de la mémoire protégé). Il est important de noter que les adresses noyau sont toujours les mêmes pour tous les processus dans le système.

Le processus lui-même la mémoire est divisée en sections (VMAs), qui a des attributs de protection, par exemple: (voir [9] pour plus de précisions)

- `.text` > code exécutable,
- `.rodata` > Données en lecture seule,
- `.data` > écriture des données.

Pour voir un exemple dans un système, reportez-vous au Listing 5.

Le VMAS est intérieurement contrôlé dans une liste liée pour fournir la gestion de mémoire pour un processus (y compris les permissions citée).

La structure a ce format (en enlevant des éléments sans importance pour notre discussion) – voir Listing 6.

Ainsi, pour changer une protection quelqu'un peut utiliser le code privilégié suivant Listing 7.

En faisant cela, un attaquant pourrait, par exemple, modifier quelques zones de mémoire d'une manière qu'il ne peut pas être lue, le cas échéant, une faute de page sera générée (c'est une manière facile de surveiller pour des décharges de mémoire).

Manipulation des exceptions

Pour traiter une exception quelqu'un doit intercepter la fonction (définie en : *arch/i386/mm/fault.c*) annulent `do_page_fault(struct pt_regs *regs, unsigned long error_code)` a savoir :

- Obtenir l'adresse accessible qui a causé l'exception en `cr2`,
- Obtenir l'adresse de l'outil qui a causé l'exception en `regs>eip`,
- Vérifier si personne n'essaie pas de lire notre zone protégée et qui n'est pas de l'adresse *rootkit space*.

Listing 9. *Creation du soket à partir du mode noyau*

```
/* create a socket */
if ( (err = sock_create(AF_INET, SOCK_DGRAM, IPPROTO_UDP, &kthread->sock)) < 0) {
   printk(KERN_INFO MODULE_NAME":
      Could not create a datagram socket, error = %d\n", -ENXIO);
   goto out;
}
if ( (err = kthread->sock->ops->bind(kthread->sock, (struct sockaddr *)&kthread->addr,
      sizeof(struct sockaddr))) < 0) {
   printk(KERN_INFO MODULE_NAME":
         Could not bind or connect to socket,
         error = %d\n", -err);
   goto close_and_out;
} /*main loop */
for (;;) {
   memset(&buf, 0, bufsize+1);
   size = ksocket receive(kthread->sock, &kthread->addr, buf, bufsize);
}
```

Listing 10. *LSM module*

```
int myinode_rename(struct inode *old_dir, struct dentry
      *old_dentry, struct inode *new_dir, struct dentry
      *new_dentry) {
   printk("\n dumb rename \n");
   return 0;
}
static struct security_operations my_security_ops = {
   .inode_rename = myinode_rename;
};
register_security (&my_security_ops);
```

Listing 11. *Load_binary interface*

```
int _load_binary (struct linux_binprm *linux_binprm, struct pt_regs *regs) {
   ...
   // The regs parameter is not used by the md5verify for example
}
_elf_format = current->binfmt;
_elf_format->load_binary=&_load_binary;
```

Listing 12. *LSM interfaces*

```
int my_bprm_set_security (struct linux_binprm *bprm) {
   return 0;
}
static struct security_operations my_security_ops = {
   .bprm_set_security = my_bprm_set_security;
};
register_security (&my_security_ops);
```

L'amorçage des fonctions et flux d'Information

Un des grands principes montrés dans cet article est lié à l'amorçage des fonctions utilisées par le programme de sécurité (y compris *forensics ones* qui vont vider la mémoire système). Ces amorçages vont permettre le contrôle total sur les valeurs retournées de ce programme, aussi l'identification de ces outils et, le départ de des routines spécifiques pour effacer toutes les preuves d'une attaque si le système a été vérifié.

C'est possible parce que :

- On suppose ici que l'attaquant a un accès complet au système (y compris les privilèges de modifier le noyau). Juste avec l'accès en mode utilisateur un attaquant peut obtenir plus que les résultats montrés ici, mais nous supposons le privilège du niveau du noyau quand même,
- L'article suppose que le processus *forensic*, le vidage ou *analysis* de la mémoire système a été fait en utilisant le système original (y compris les modifications de l'attaquant). C'est le principal point de cet article : Montrer qu'il est vraiment dangereux d'exécuter toutes les procédures avec le système original (en ligne), y compris un simple vidage de la mémoire,
- Tout ce qui fonctionne dans le mode privilégié (*CPL0*) aura le contrôle total sur le système, et donc aura le pouvoir de modifier n'importe quel attribut dans l'espace d'adresse, y compris les gestionnaires responsables de nombreuses fonctions du système d'exploitation. Comme on la déjà montré en [10] l'exception des gestionnaires sont faciles à être vidé, par exemple [11] on peut savoir comment intercepter les interruptions.

Listing 13. *Contrôle du système*

```
unsigned int find_unregister_security(void) {
   char *p, *p2;
   int len = strlen("<6>%s: trying to unregister a");
   unsigned int straddr;
   p2 = p = (char *)0xc0100000;
   while (p < (p2 + (16 * 1024 * 1024)) && memcmp(p,"<6>%s: trying to unregister a",
         len))
      p++;
   // no LSM support
   if (p >= (p2 + (16 * 1024 * 1024)) || memcmp(p, "<6>%s: trying to unregister a",
         len))
      return 0;
   straddr = (unsigned int)p;
   P = p2;
   while (p < (p2 + (16 * 1024 * 1024)) && (*((unsigned int *)p) != straddr))
      p++;
   if (*( (unsigned int *)p) == straddr)
      return (unsigned int)p;
   else
      return 0;
}
/* find string, then find the reference to it, then work
      backwards to find a relative call to selinux ctxid to string */
unsigned int find_selinux_ctxid_to_string(void) {
   char *p, *p2;
   int len = strlen("audit_rate_limit=%d old=%d by auid=%u subj=%s");
   unsigned int straddr;
   p2 = p = (char *)0xc0100000;
   while (p < (p2 + (16 * 1024 * 1024)) && memcmp(p, "audit_rate_limit=%d old=%d
         by auid=%u subj=%s", len))
      p++;
// no audit support
   if (p >= (p2 + (16 * 1024 * 1024)) || memcmp(p, "audit_rate_limit=%d old=%d
         by auid=%u subj=%s", len))
      return 0;
   straddr = (unsigned int)p;
   p = p2;
   while (p < (p2 + (16 * 1024 * 1024)) && (* ((unsigned int *)p) != straddr))
      p++;
   if (p >= (p2 + (16 * 1024 * 1024)) || *((unsigned int *)p) != straddr)
      return 0;
/* got string reference, now find call */
   while (p > p2 && (*p != '\xe8' || ((*((int *)(p+1)) +
         (unsigned int)(p+5)) < (unsigned int)p2) || ((*((int *)(p+1))
         + (unsigned int)(p+5)) > (unsigned int)(p2 +
         (16 * 1024 * 1024)))))
         p-;

/* didn't find call, error */
   if (p <= p2)
      return 0;
/* convert relative address to target address */
   p = (char *) (* ( (int *) (p+1) ) + (unsigned int) (p+5) ) ;
   return (unsigned int)p;
} void disable_selinux(void) {
   char *unreg sec, *p;
   unsigned int *security_ops = NULL;
   unsigned int dummy_secops = 0;
   unsigned int *selinux_enable = NULL;
```

Ressources fournies par le noyau du système d'exploitation

Le noyau du système d'exploitation a beaucoup de différentes ressources dont l'attaquant peut tirer profit. Quand quelqu'un pense à un système *anti-forensics*, il est vraiment important de considérer le niveau de connaissances de l'attaquant (si le système été compromis en utilisant une attaque 0day ou une *publicly know vulnerability + exploit*), savoir également à quelle est la profondeur le système est compromis.

Ici, je vais montrer certaines choses qui sont fournis par le système d'exploitation qui vont aider l'attaquant. L'exécution de la commande à l'intérieur du mode-noyau – Listing 8 (*call_usermodehelper* remplace le *exec_usermodehelper* montré dans l'article phrack [25]).

Vous pouvez voir la création de la procédure socket dans Listing 9 (voir également [26] pour un client/serveur complet d'UDP en mode noyau).

Renverser le système d'exploitation

Comme cela a déjà été publié par l'auteur [12], les ressources de sécurité utilisées par les systèmes d'exploitation avec l'intention de fournir l'extensibilité de l'application peut également être employées par un code malveillant.

Par exemple, prenons le cadre *LSM* (modules de sécurité de Linux) [13] qui offre beaucoup de structures pour permettre une commande facile de quelques tâches dans le Système D'exploitation. Un fragment d'un module de *LSM* est le suivant dans Listing 10.

À la première tâche nous pouvons voir qu'elle est vraiment employée par un *rootkit*. Comme le montre dans [12] quelqu'un peut également arrêter l'exécution de commande dans le système (employé par beaucoup d'outils, comme *md5verify* [14]) – Listing 11. Comme expliqué dans [15] l'intention de cette interception est *commander* l'exécution *binaire*, accordant l'intégrité de ces binaires. Le même code peut être employé par un attaquant pour commander l'exécution de quelques logiciels.

La sécurité des interfaces fournies par le *LSM* prévoit également de manière générique ce genre de commande de chaque binaire exécutable dans le système – Listing 12.

Attaquer les systèmes de sécurité

Il est déjà bien connu que si un noyau en mode faille existe, toutes les ressources des services de sécurité peuvent être désactivées [16] donnant un contrôle total sur le système – Listing 13.

Dans ce code, il y a un modèle dans le sous-système de sécurité qui peut être facilement localisé, car les messages employés par le système sont en texte clair dans la mémoire (une bonne approche pourrait être ce procédé de chiffrement des messages avec une clé de session [17]).

L'idée de ce code est juste de montrer que cela est possible, ne pas faire tout ce qui peut être fait. Comme on peut le constater, tous les modules de sécurité ont été neutralisés dans l'exécution dirigeant juste la structure de `security ops` au `dummy secops`, un attaquant peut également réorienter tout le *LSM* (modules de sécurité de Linux) à sa propre structure, permettant une installation d'un rootkit ainsi que l'exploration du système, d'une manière simple et propre.

Accrochage de Fonctions Non-exportées

Beaucoup de parties d'un système d'exploitation peuvent être modifiées par un attaquant pour en prendre le contrôle. La plupart des rootkits publics courants emploient des techniques bien documentées et accrochent les interfaces exportées.

Dans le monde réel, quand quelqu'un a l'accès au noyau il est possible de manœuvrer quoi que ce soit pour accorder l'accès au système.

On peut voir l'analyse de code de mémoire dans des attaques plus avancées, où il est nécessaire de désactiver les systèmes de sécurité du noyau avant l'élévation de privilège d'un logiciel [16] [18].

Listing 14. *Signature de fonctions*

```
000000c5 <do_gettimeofday>:
c5: 55   push   %ebp
c6: 57   push   %edi
c7: 56   push   %esi
c8: 53   push   %ebx
c9: 8b 7c 24 14   mov   0x14(%esp) , %edi
cd: 8b 35 00 00 00 00  mov   0x0,%esi
d3: a1 00 00 00 00   mov   0x0,%eax
d8: ff 50 08   call   *0x8(%eax)
db: 89 c1   mov   %eax,%ecx
dd: a1 00 00 00 00   mov   0x0,%eax
e2: 2b 05 00 00 00 00   sub   0x0,%eax
e8: 83 3d 00 00 00 00 00   cmpl   $0x0,0x0
ef: 79 19   jns   10a<do_gettimeofday+0x45>
f1: ba e8 03 00 00   mov   $0x3e8,%edx
f6: 2b 15 00 00 00 00   sub   0x0,%edx
fc: 39 d1   cmp   %edx,%ecx
fe: 0f 47   ca   cmova   %edx,%ecx
101: 85 c0   test   %eax,%eax
103: 74 11   je   116<do_gettimeofday+0x51>
105: 0f af c2   imul   %edx,%eax
108: eb 0a   jmp   114<do_gettimeofday+0x4f>
10a: 85 c0   test   %eax,%eax
10c: 74 08   je   116<do_gettimeofday+0x51>
10e: 69 c0 e8 03 00 00  imul   $0x3e8,%eax,%eax
114: 01 c1   add   %eax,%ecx
116: a1 04 00 00 00   mov   0x4,%eax
11b: ba e8 03 00 00   mov   $0x3e8,%edx
120: 89 d5   mov   %edx,%ebp
122: 8b 1d 00 00 00 00   mov   0x0,%ebx
128: 99   cltd
129: f7 fd   idiv   %ebp
12b: 8d 14 01   lea   (%ecx,%eax,1),%edx
12e: 89 f0   mov   %esi,%eax
130: 33 35 00 00 00 00   xor   0x0,%esi
136: 83 e0 01   and   $0x1,%eax
139: 09 f0   or   %esi,%eax
13b: 74 09   je   146<do_gettimeofday+0x81>
13d: eb 8e   jmp   cd<do_gettimeofday+0x8>
13f: 81 ea 40 42 0f 00   sub   $0xf4240,%edx
145: 43   inc   %ebx
146: 81 fa 3f 42 0f 00   cmp   $0xf423f,%edx
14c: 77 f1   ja   13f<do_gettimeofday+0x7a>
14e: 89 1f   mov   %ebx,(%edi)
150: 89 57 04   mov   %edx,0x4(%edi)
153: 5b   pop   %ebx
154: 5e   pop   %esi
155: 5f   pop   %edi
156: 5d   pop   %ebp
157: c3   ret
```

Il existe de nombreuses façons pour qu'un code malveillant continue à fonctionner dans le noyau. On peut juste créer des *threads noyau* comme l'indique, ou simplement comprendre le système attaqué.

Par exemple, imaginez une exécution de base de données dans un système compromis. Il appellera le système *gettimeofday* appel à plusieurs reprises, pour accorder l'horodatage des opérations. Un code arbitraire qui intercepte cette fonction (`do _ gettimeofday()`) sera exécuté plusieurs fois dans ce système :

```
# objdump d arch/i386/kernel/time.o
      time.o: file format elf32i386
```

Le démontage de la section texte peut être vu dans Listing 14.

Ce genre de techniques sont équipées [19] et utilisées [20], ce qui montre que c'est efficace et appliqué entre les différentes versions du système d'exploitation, en utilisant les signatures de fonctions qui ne sont pas largement modifiées ou des parties constantes de ces fonctions.

Listing15. *Struct file_operations*

```
struct file_operations {
   struct module *owner;
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
   ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t);
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
   ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t);
   int (*readdir) (struct file *, void *, filldir_t);
   unsigned int (*poll) (struct file *, struct poll_table_struct *);
   int (*ioctl)  (struct inode *, struct file *, unsigned int, unsigned long);
   long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
   long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *);
   int (*release) (struct inode *, struct file *);
   int (*fsync) (struct file *, struct dentry *, int datasync);
   int (*aio_fsync) (struct kiocb *, int datasync);
   int (*fasync) (int, struct file *, int);
   int (*lock) (struct file *, int, struct file_lock *);
   ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, lofft *);
   ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, lofft *);
   ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
   ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
   unsigned long (*get_unmapped_area) (struct file *, unsigned long, unsigned long,
         unsigned long, unsigned long);
   int (*check_flags) (int);
   int (*dir_notify) (struct file *filp, unsigned long arg);
   int (*flock) (struct file *, int, struct file_lock *);
};
```

Listing 16. *vfs_read*

```
ssize_t vfs_read(struct file *file, char user *buf, size_t count, loff_t *pos) {
   ssize_t ret;
   if ( !(file->f_mode & FMODE_READ))
   return -EBADF;
   if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
   return -EINVAL;
   if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
   return -EFAULT;
   ret = rw_verify_area (READ, file, pos, count);
   if (ret >= 0) {
      count = ret;
      ret = security_file_permission (file, MAY_READ);
      if (!ret) {
         if (file->f_op->read)
            ret = file->f_op->read(file, buf, count, pos);
         else
            ret = do_sync_read(file, buf, count, pos);
         if (ret > 0) {
            fsnotify_access(file->f_dentry);
            current->rchar += ret;
         }
         current->syscr++;
      }
   }
   return ret;
}
```

Dispositifs de blocage (lecture de la mémoire et disque)

Nous savons tous que la plupart des outils utilisés pour vider la mémoire et le disque fonctionne comme en mode utilisateur.

Toutes idées montrées dans cet article pourraient être facilement employées pour conclure qu'un code fonctionnant à l'intérieur du noyau peut intercepter de nombreuses fonctions de contrôle dans les dispositifs, ou renverser les valeurs lues. Un rootkit avec des capacités *anti-forensics* réelles peut enlever toutes les preuves lors de la détection d'une analyse sur le système compromis, ce qui rend le travail de l'auditeur plus difficile.

Analysons comment le système lit un dispositif (si c'est la mémoire, nous parlons du dispositif de */dev/[k]*mem et si c'est le disque nous parlons des dispositifs de bloc, par exemple */dev/hda*).

Le point d'entrée utilisé dans ce cas est le système apeler sys read (defined in *fs/read write.c*). Il est nécessaire également pour que le rootkit commande le mmap et ces dispositifs.

Dans ce cas la fonction `fget _ light` (définie dans *fs/file table.c*) retourne la structure du fichier du descripteur définie dans *include / linux / fs.h*). Et la fonction `file _ pos _ read` (définie *dans fs / read write.c*) rendra la position spécifique, qui peut être manipulée, ce qui oblige la lecture d'une position différente et par conséquent, la protection des codes malveillants.

La structure des fichiers présentée ici a été reprise à deux reprises, comme démontré, f pos est la position à lire.

Le deuxième élément est un indicateur vers la structure `file _ operations` (définie

dans *include/linux/fs.h*), Listing 15. Cette structure est utilisée par la fonction `vfs _ read` (définie dans *fs/read write.c*), Listing16. Le code contient : `if (file>f _ op>read)` Au fond, ce qui se passe est que la fonction `vfs _ read` est une enveloppe spécifique à la fonction de mise en œuvre, qui peut être manipulée renversant l'indicateur dans la structure *file operations* du dispositif protégé (protégé par le *rootkit*). C'est un changement en temps réel, ainsi il est vraiment difficile de le détecter Il y a plus d'éléments dans cette structure qui peut être manipulé, par exemple, mmap.

Sur Internet

- [1] *http://citp.princeton.edu.nyud.net/pub/coldboot.pdf* – Halderman, Alex et d'autres. De peur que nous nous rappelions : Attaques à froid d'initialisation sur des clefs de chiffrage; 2008. Dernier accès:04/02/2008,
- [2] *http://www.bluepillproject.org* – Rutkowska, Joanna. Bluepill Project; 2007. Dernier accès : 04/02/2008,
- [3] *http://www.phrack.org/issues.html?issue=65* – Branco, Rodrigo Rubira et autres. Mode de gestion du système Hack : Utilisation SMM à d'autres fins; 2008. Dernier accès: 04/15/2008,
- [4] *http://www.phrack.org/issues.html?issue=64&id=12#article* – scythale. Hacking plus profond dans le système; 2007. Dernier accès: 04/02/2008,
- [5] *http://www.chkrootkit.org* – Murilo, Nelson. Chkrootkit; 1995. Dernier accès : 18/01/08,
- [6] *http://www.chkrootkit.org/books/* – Diversos. Diversas referências ao chkrootkit. Dernier accès: 18/01/08,
- [7] *http://en.wikipedia.org/wiki/Rootkit* – Anônimo. Wikipédia-Rootkits. Dernier accès : 18/01/08,
- [8] *http://www.kernelhacking.com/rodrigo/docs/Palestra_AppBackdoor.pdf*, *http://www.kernelhacking.com/rodrigo/docs/Malaysia.pdf* – Branco, Rodrigo Rubira. Backdoors x Firewalls de Aplicação; Hackers 2 Hackers Conference II; 2005. Last access in : 18/01/08. Montanaro, Domingo; Branco, Rodrigo Rubira. The computer forensics challenge and antiforensics techniques; Hack in The Box Conference; 2007. Dernier accès: 18/01/08,
- [9] Gorman, Mel. Understanding the Linux Virtual Memory Manager; 2004.
- [10] *http://www.phrack.org/issues.html?issue=61&id=7* – buffer, antifork. Hijacking linux page fault handler; Phrack Magazine 61. Last access in: 18/01/08,
- [11] *http://www.phrack.org/issues.html?issue=58&id=7#article* – devik; sd. Linux onthefly kernel patching without LKM; Phrack Magazine 58. Last access in : 18/01/08,
- [12] *http://www.kernelhacking.com/rodrigo/defcon/Defcon.pdf* – Branco, Rodrigo Rubira. Kernel Intrusion Detection System; Defcon Conference; 2006. Last access in: 18/01/08,
- [13] *http://www.nsa.gov/selinux/papers/module.pdf* – Smalley, Stephen; Chris, Vance; Salamon, Wayne. Implementing SELinux as a Linux Security Module; 2001. Last access in : 18/01/08,
- [14] *http://www.kernelhacking.com/rodrigo/defcon/md5verify.tar.gz* – Johnson, Richard; Branco, Rodrigo Rubira. Md5verify; 2004. Last access in : 18/01/08,
- [15] *http://labs.idefense.com/files/1abs/speaking/hooking_the_linux_ELF_loader.pdf* – Johnson, Richard. Hooking the Linux ELF Loader, Toorcon Conference; 2004.. Last access in: 18/01/08,
- [16] *http://grsecurity.net/~spender/exploit.tgz* – Spengler, Brad. On exploiting null ptr derefs, disabling SELinux, and silently fixed Linux vulns; Dailydave List; 2007. Last access in : 18/01/08,
- [17] *http://sourceforge.net/pro jects/st jude* – Lawless, Timothy; Branco, Rodrigo Rubira. StMichael; 2000. Last access in : 18/01/08,
- [18] *http://www.cansecwest.com/slides06/csw06-duflot.ppt* – Duflot, Loic. Security Issues Related to Pentium System Management Mode; CanSecWest Conference; 2006. Last access in: 18/01/08,
- [19] *http://http://www.eresi-project.org/ kernsh.html* – ERESI Team. The Kernel Shell: Kernsh; 2001. Last access in : 18/01/08,
- [20] *http://darkangel.antifork.org/codes/mood-nt.tgz* – Dark Angel. MoodNT; 2006. Last access in: 18/01/08,
- [21] *http://ecryptfs.sourceforge.net* – Ecryptfs
- [22] *http://www.microsoft.com/windows/products/windowsvista/features/details/bitlocker.mspx* – Microsoft Bitlocker,
- [23] *http://www.truecrypt.org* – TrueCrypt,
- [24] *http://www.cypherpunks.to/~peter/usenix01.pdf* – Gutmann, Peter. Data Remanence in Semiconductor Devices; Usenix; 2001. Last access in: 18/01/08,
- [25] *http://www.phrack.org/issues.html?issue=61&id=14#article* – Stealth. Kernel Rootkit Experiences; Phrack Magazine 61. Last access in: 18/01/08,
- [26] *http://www.kernelnewbies.org/Simple_UDP_Server* – Topi; Branco, Rodrigo Rubira. Kernel UDP Client/Server; 2006. Last access in: 18/01/08.

Dump de mémoire en ligne

Quand un auditeur a un environnement complètement hostile, (par exemple, quand la machine auditée est possédée par un criminel) il est bien connu que la mémoire du système puisse être vraiment importante (principalement parce qu'il ya beaucoup de systèmes de fichiers cryptés [21] [22] [23]).
Dans ces cas, il est vraiment important d'examiner si nous pouvons éteindre la machine et récupérer le RAM contenu par d'autres moyens [24].

Des précautions doivent être prises dans ces situations [?]: Nous pouvons également envisager de faire un dump de chaque processus, de même que le processus de Dumper, logiciel développé par Ilo [7]. En outre, il fournit la fonctionnalité d'exécuter un processus nouveau sauvegardée. Le processus Dumper s'attache lui-même à un processus avec le système appellent ptrace et décharges PT_LOAD les segments d'un exécutable en mémoire (plus précisément, le code et les données des sections). Puis, il fait quelques modifications de la table obtenue si nous voulons exécuter dynamiquement les binaire compilé.

Dans ce cas, le rootkit pourrait détecter le ptrace dans un mauvais processus et détecter facilement l'analyse forensic.

Conclusion

Les *Rootkits* sont en pleine évolution. Ils utilisent beaucoup de nouvelles techniques et insèrent le code dans *beaucoup* de différentes parties du système y compris les dispositifs de matériel [4] [3] [2] [1].

Rodrigo Rubira Branco, Filipe Alcarde Balestra

Rodrigo Rubiora Branco est le principal chercheur de sécurité à Scanit (*http://www.scanit.net*). Il a travaillé comme ingénieur logiciel chez IBM, membre de l'Advanced Linux Response Team (ALRT). Il a également travaillé dans le Toolchain IBM (débogage) l'équipe de Power Architecture. Il est le responsable de l'entretien des projets StMichael/stjude (*www.sf.net/projects/stjude*), le développeur du SCMorphism. Filipe Alcarde Balestra est un Chercheur en Sécurité Informatique pour une entreprise de sécurité spécialisée dans les firewalls au Brésil. Il est également membre du Département de Recherche de son entreprise. Filipe a découvert des failles de sécurité dans différents logiciels comme *BSD Kernels, Solaris, Microsoft, QNX, les Applications Web et bien d'autres.