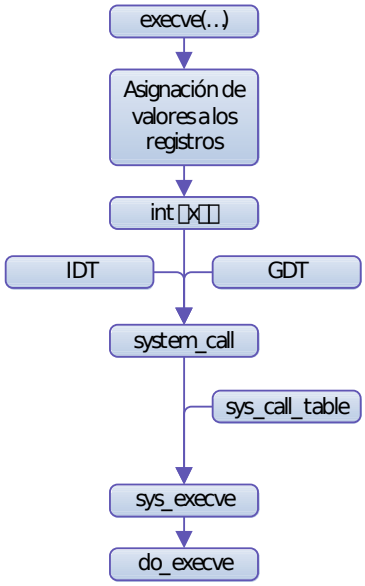
LECCIÓN 9: EXEC

9.1 Introducción

Llamada al sistema

A pesar de que la siguiente figura esté adaptada a execve, todas las llamadas al sistema tienen un flujo similar.



Hasta ahora hemos visto cómo se crean hilos de un proceso existente, así como las llamadas al sistema que permiten esperar por su terminación y notificarla. Sin embargo, aun no sabemos cómo ejecutar un programa determinado en memoria.

El lanzamiento de procesos en cualquier Sistema Operativo exige siempre situar en memoria el código ejecutable del programa, asegurando siempre que se realiza de forma correcta (seguridad, verificación de archivos ejecutables, disponibilidad de memoria…). A continuación, se debe situar algún tipo de marca para señalar la posición de inicio de las instrucciones y, por último, insertar el proceso en la cola de tareas. Esta es la función de la llamada al sistema execve.

Como podemos comprobar en la imagen lateral, execve utiliza el mismo flujo de funciones que muchas de las llamadas al sistema vistas hasta ahora como fork, wait y exit. Las tareas más importantes, como veremos posteriormente, las realiza la función do_execve.

Uso de execve

La llamada al sistema execve requiere siempre tres parámetros explícitos:

- Nombre (y ruta) del programa
- Lista de parámetros que se pasarán al programa
- Lista de variables de entorno y sus valores

La interfaz de esta llamada se encuentra declarada en el fichero de cabecera unistd.h, por tanto será necesario incluirlo en los proyectos que realicemos si deseamos usarla.

Según dicha interfaz, los tipos de los parámetros son una cadena constante de caracteres (const char *) y dos vectores del mismo tipo en los que la última posición sea nula ((const char *) NULL), respectivamente. Este último detalle permite a la función do-_exec identificar cuándo finaliza la lista correspondiente.

La llamada execve no retorna si la ejecución del programa tiene éxito. En caso contrario, retornará al proceso que la invocó comunicando el código de error en la variable errno. A pesar de que esto es lo que la especificación del kernel 2.6.11 comenta, en realidad hay un “punto de no retorno” tras el cual, aunque el proceso termine erróneamente no es capaz de informar al proceso invocador debido a que se desalojó de la memoria los datos que permitían acceder al mismo (ver Implementación para más detalle).

La familia exec

La llamada al sistema execve puede ser en ocasiones algo complicada para trabajar con ella ya que no realiza una búsqueda del fichero ejecutable y no tiene en cuenta las variables de entorno ya definidas.

La librería de C (libc) define un número de funciones alternativas que usan execve en su implementación, pero que permiten usar otras interfaces más intuitivas al usuario. A continuación se detalla en una tabla los parámetros y el uso de dichas funciones:

---

execl(...)

- Nombre y ruta del fichero ejecutable.
- Ristras separadas por comas con cada uno de los argumentos.

Ejemplo

```
extern const char* environ[];
execl("/bin/ls", "/bin/ls", "-l", (const char *)NULL);
```

---

execlp(...)

- Nombre del fichero ejecutable (implementa búsqueda del programa).
- Ristras separadas por comas con cada uno de los argumentos.

Ejemplo

```
extern const char* environ[];
execlp("ls", "ls" , "-l", (const char *)NULL);
```

---

execle(...)

- Nombre y ruta del fichero ejecutable.
- Ristras separadas por comas con cada uno de los argumentos.
- Vector de ristras de caracteres con las variables de entorno y sus valores.

Ejemplo

```
char *env[] = {"PATH=/bin", (const char*)NULL};
execle("/bin/ls", "/bin/ls", "-l", (const char *)NULL, env);
```

---

execv(...)

- Nombre y ruta del fichero ejecutable.
- Vector de ristras de caracteres con los argumentos.

Ejemplo

```
extern const char* environ[];
char *argv[] = {"-l", (const char*)NULL};
execv("/bin/ls", argv);
```

---

execvp(...)

- Nombre del fichero ejecutable (implementa búsqueda del programa).
- Vector de ristras de caracteres con los argumentos.

Ejemplo

```
extern const char* environ[];
char *argv[] = {"-l", (const char*)NULL};
execv("ls", argv);
```

Como se puede comprobar en los ejemplos anteriores, aquellas funciones que no requieran el uso de una lista de variables de entorno y sus valores, usan la variable environ, que es un vector con dichos valores.

Si se desarrollan aplicaciones que requieran migración a otro Sistema Operativo, se recomienda utilizar la llamada al sistema tal cual, y no utilizar ninguna de las funciones de la familia exec, debido a que muchos compiladores cruzados no reconocen correctamente dichas llamadas.

Nombres en la familia exec

La familia de funciones exec utiliza letras como sufijos para indicar los parámetros que requieren y su comportamiento:

p	*Buscan la ruta del programa*
l	*Los parámetros se pasan separados por comas*
v	*Los parámetros se pasan en un vector*
e	*Requieren un vector de*

9.2 Implementación del exec

La llamada al sistema sys_execve() requiere 3 parámetros.

1. Dirección donde se encuentra la string del pathname del archivo ejecutable.

2. Dirección de un array de punteros a strings. El final del array lo determinará un puntero a NULL dentro del array. Cada cadena representa un argumento de la línea de comandos.

3. Otro array de punteros a string, pero cada cadena representa una variable de entorno, representada como:

   NOMBRE=valor

Todos estos parámetros residen en el espacio de memoria del Modo Usuario.

Esta llamada al sistema primero copia el ejecutable del pathname a una nueva página. Luego invoca a do_execve() pasándole un puntero a esa nueva página, los punteros a los dos arrays antes comentados (el de variables de entorno y el de argumentos de la línea de comandos) y por último un puntero a la pila donde se encuentran los contenidos de los registros. Esta pila se encuentra en el espacio de memoria del Modo Kernel.

Principalmente esta función do_execve, se encarga de rellenar una estructura de tipo linux_binprm, con las distintas características del nuevo programa que deseamos que sustituya al anterior, chequeando que el programa sea correcto en todos sus aspectos.

Una vez hecho esto, usa esta estructura para actualizar la ranura de procesos y espacio de memoria, con lo que el programa ya estará cargado en la máquina.

```
1279 int do_execve(char * filename,
1280         char __user *__user *argv,
1281         char __user *__user *envp,
1282         struct pt_regs * regs)
1283 {
```

La función ***do_execve()*** va a realizar las siguientes acciones:

1. Genera dinámicamente una estructura ***linux_binprm***, la cual será rellenada con los datos del nuevo archivo ejecutable. El rellenado se irá haciendo progresivamente en sucesivos pasos.

```
1284        struct linux_binprm *bprm;
…
1294        bprm = kzalloc(sizeof(*bprm), GFP_KERNEL);
…
```

Antes de continuar, vamos a ver los campos que incluye la estructura ***linux_binprm***:

```
  27 struct linux_binprm{
  28        char buf[BINPRM_BUF_SIZE];
  29 #ifdef CONFIG_MMU
  30        struct vm_area_struct *vma;
  31 #else
  32 # define MAX_ARG_PAGES  32
  33        struct page *page[MAX_ARG_PAGES]; // Tabla de página del proceso
  34 #endif
  35        struct mm_struct *mm;
  36        unsigned long p; /* current top of mem */
  37        unsigned int sh_bang:1,
  38                     misc_bang:1;
  39 #ifdef __alpha__
  40        unsigned int taso:1;
  41 #endif
  42        unsigned int recursion_depth;
  43        struct file * file;
  44        int e_uid, e_gid; // Identificador que usa el sistema para controles
                             // de acceso e identificador de usuario
  45        kernel_cap_t cap_post_exec_permitted;
  46        bool cap_effective;
  47        void *security;
  48        int argc, envc;   // Variables de entorno y argumentos
  49        char * filename;        /* Name of binary as seen by procps */
  50        char * interp;          /* Name of the binary really executed. Most
  51                                   of the time same as filename, but could
  52                                   be different for binfmt_{misc,script} */
  53        unsigned interp_flags;
  54        unsigned interp_data;
  55        unsigned long loader, exec;
  56};
```

Aunque iremos viendo con qué datos se van rellenando los diferentes campos a lo largo de la ejecución del ***do_exec()***, vamos a explicar brevemente algunos de ellos:

char buf[BINPRM_BUF_SIZE]	Se almacenarán los primeros 128 bytes del archivo ejecutable
struct page *page[MAX_ARG_PAGES]	Tabla de páginas del proceso. Máximo 32 páginas
struct mm_struct *mm	Mapa de memoria del proceso
unsigned long p	Cantidad de memoria actualmente utilizada
struct file * file	Descriptor del Fichero ejecutable
int e_uid, e_gid	Permisos de ejecución del ejecutable
kernel_cap_t cap_inheritable, cap_permitted, cap_effective	Definen un conjunto de privilegios, privilegios heredados, permitidos y efectivos
void *security	Guarda información de seguridad
int argc, envc	Número de argumentos y de variables de entorno
char * filename	Nombre del ejecutable
char * interp	Nombre del binario realmente ejecutado. La mayor parte del tiempo es lo mismo que *filename*, pero puede ser diferente en ciertas situaciones.

2. Se invoca a ***open_exec()***, el cual se ocupará de invocar a las funcion ***path_lookup_open()*** para obtener los objetos ***dentry***, ***file*** e ***inode*** (objetos con información sobre el binario) asociados con el archivo ejecutable. Si ocurre algún error se devolverá el correspondiente código de error. Se chequea que el archivo es ejecutable por el proceso actual. Se chequea que no se va a escribir en el fichero ejecutable, mirando el campo ***i_writecount*** del ***inode***. Se guarda un -1 en este campo para evitar futuros accesos para escribir en este fichero. Esto lo realiza la llamada a la función ***deny_write_access***. Al final nos devolverá una estructura ***file*** con todos los datos necesarios para continuar.

```
1298        file = open_exec(filename); // Rellena información
1299        retval = PTR_ERR(file);     // Comprobar si hubo error
1300        if (IS_ERR(file))
1301                goto out_kfree;
```

3. Se invoca a ***sched_exec()*** para, si nos encontramos en un sistema multiprocesador, determinar la CPU con menos carga que pueda ejecutar el nuevo programa y migrar el proceso actual a dicha CPU.

```
1357        sched_exec();
```

4. Se invoca a *init_new_context()* para ver si el proceso actual tiene una LDT (Local Descriptor Table) personalizada. Si es así, se creará una nueva LDT y se rellenará para ser usada por el nuevo programa. La LDT es una estructura en memoria de la arquitectura x86 que contiene información sobre los segmentos. Esta función está dentro de bprm_mm_init(struct _linux_binprm *bprm)

```
356         err = init_new_context(current, mm); // Rellena LDT
357         if (err)                             // Comprueba errores
358                 goto err;
```

Se invoca a ***prepare_binprm()*** para rellenar algunos campos de la estructura ***linux_binprm*** creada en el paso 1.

```
1325        retval = prepare_binprm(bprm);
1326        if (retval < 0)
1327                goto out;
```

5. Se copian el ***pathname***, los argumentos de la línea de comandos y las variables de entorno en una o más páginas nuevas reservadas para el proceso. Estas nuevas páginas se encontraran en el espacio de direccionamiento del Modo Usuario.

```
1329    retval = copy_strings_kernel(1, &bprm->filename, bprm);
...
1333    bprm->exec = bprm->p;
1334    retval = copy_strings(bprm->envc, envp, bprm);
...
1338    retval = copy_strings(bprm->argc, argv, bprm);
```

6. Se invoca a *search_binary_handler()*. Esta función se va a encargar de recorrer la lista de formatos de ficheros ejecutables y tratará de ejecutar la función *load_binary()* de cada elemento de la lista de formatos, pasándole a esta función la estructura *linux_binprm*. Esta búsqueda termina cuando un *load_binary()* tiene éxito en la averiguación del formato del archivo ejecutable.

```
1343    retval = search_binary_handler(bprm,regs);
```

7. Si el formato del archivo ejecutable no se encuentra en la lista de formatos, se liberan todas las páginas que se han adquirido a lo largo del proceso y se devuelve el código de error:

   –ENOEXEC *: Linux no reconoce el formato del archivo ejecutable*. Es un tipo de error

   Además aprovechamos para mostrar los diferentes tratamientos para los diferentes errores que se pueden dar a lo largo de la ejecución de la función ***do_execve()***. Las diferentes etiquetas dan lugar diferencian los puntos de entrada que desharán lo hecho hasta el momento de producirse el error.

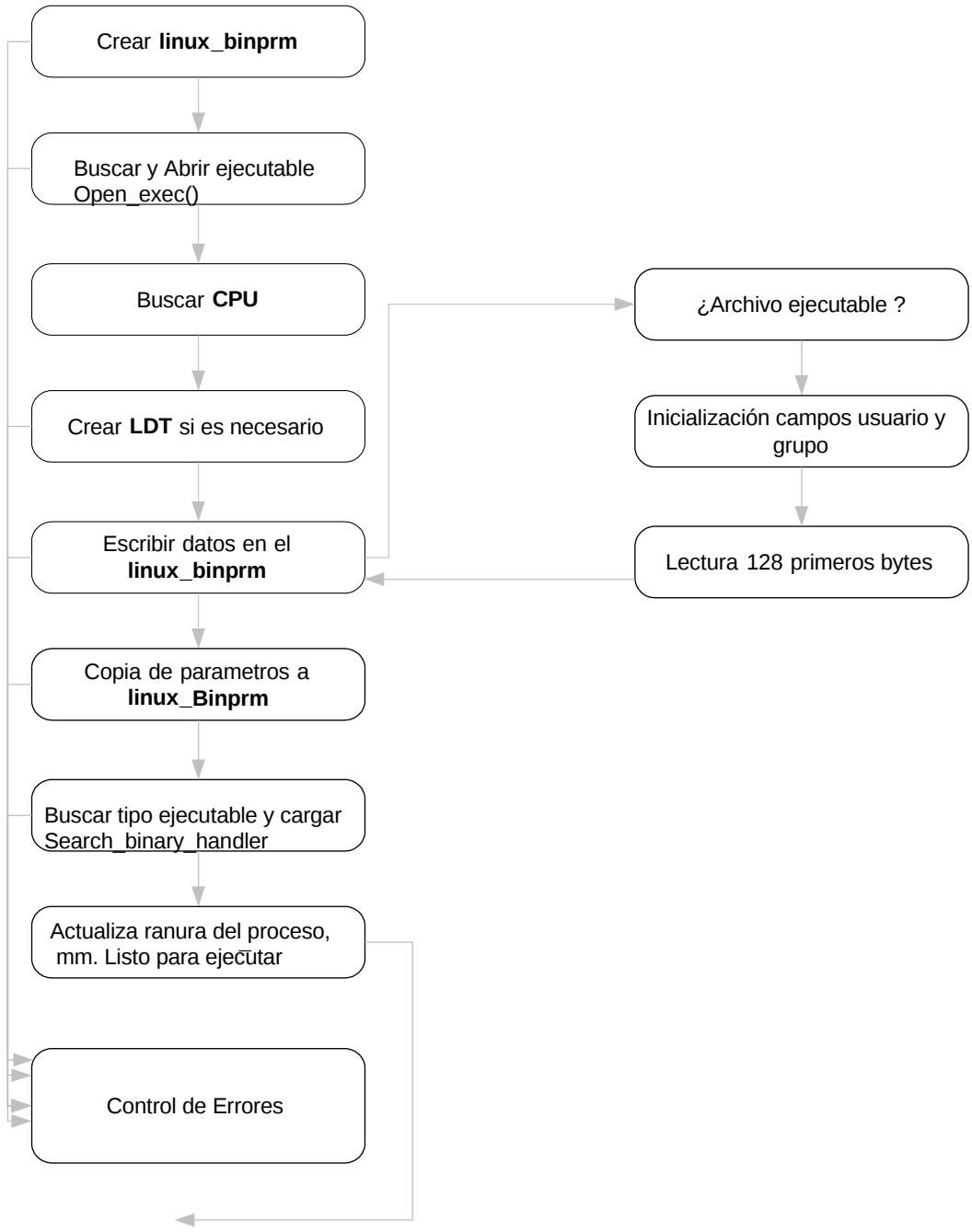
```
1354  out: // Etiqueta out para liberar memoria de la variable bprm
1355        if (bprm->security)
1356                security_bprm_free(bprm);
1357
1358  out_mm: // Etiqueta out_mm para liberar del mapa de memoria
1359        if (bprm->mm)
1360                mmput (bprm->mm);
1361
1362  out_file: // Etiqueta out_file libera el fichero
1363        if (bprm->file) {
1364                allow_write_access(bprm->file);
1365                fput(bprm->file);
1366        }
1367  out_kfree: // Etiqueta out_kfree para liberar memoria de bprm
1368        free_bprm(bprm);
1369
1370  out_files:
1371        if (displaced)
1372                reset_files_struct(displaced);
1373  out_ret: // Etiqueta out_ret para retornar el valor del error
1374        return retval;
1375  }
```

8. Si todo ha ido bien, la función libera la estructura ***linux_binprm*** y retorna el código obtenido de la función ***load_binary()*** asociada con el formato del archivo ejecutable.

```
1344     if (retval >= 0) { // Si tiene valor positivo, tiene éxito
1345             /* execve success */
1346             security_bprm_free(bprm);
1347             acct_update_integrals(current);
1348             free_bprm(bprm);
1349             if (displaced)
1350                     put_files_struct(displaced);
1351             return retval; // Retorna el valor del exec
1352     }
```

En este código no se ve bien exactamente como se actualiza la ranura de procesos. Se actualizan los campos “mm” en la mm_struct del proceso que invocó al exec. Esto significa que no se borra la task struct del proceso llamador, sino que se actualizan sus campos. Uno de los campos que se actualiza es el referente a la memoria del programa. En el caso de la memoria SÍ SE LE ASIGNA UN NUEVO ESPACIO DE MEMORIA, evidente si se considera que el nuevo programa puede tener más, o menos código que el padre.

Resumen de do_execve



9.3 Funciones auxiliares

función – count

Esta es una de las funciones auxiliares usadas en do_execv e implementadas en /fs/exec.c. Se encarga de recorrer el array apuntado por argv (que en do_execve puede tratarse de argv para revisar los argumentos, o envp para revisar variables de entorno), contando los argumentos y revisando que realmente existen y pueden ser accedidos. Al primer nulo, para la cuenta, y devuelve el número de no-nulos, que es el número de argumentos (o variables entorno, según el parámetro de entrada).

```
 375 /*
 376  * count() counts the number of strings in array ARGV.
 377  */
 378 static int count(char __user * __user * argv, int max)
 379 {
 380         int i = 0;
 381
 382         if (argv != NULL) {
 383                 for (;;) {
 384                         char __user * p;
 385                    // Se accede a los argumentos. Si hay problemas,
                       // error. Por esto se debe usar Count
 386                         if (get_user(p, argv))
 387                                 return -EFAULT;
                       // Desde que se encuentre el primer nulo, se
                       // deja sale del bucle. Fin de argumentos
 388                         if (!p)
 389                                 break;
 390                         argv++;
                       // Si los argumentos se pasan del máximo, error.
 391                         if (i++ >= max)
 392                                 return -E2BIG;
 393                         cond_resched();
 394                 }
 395         }
 396         return i;
 397 }
```

prepare_binprm

Rellena partes significativas de la estructura binprm usando información del inodo. Así, prepara los parámetros del fichero ejecutable, Revisa permisos, y después lee la cabecera los primeros 128 bytes del ejecutable.

```
1036/*
1037 * Fill the binprm structure from the inode.
1038 * Check permissions, then read the first 128 (BINPRM_BUF_SIZE) bytes
1039 */
1040int prepare_binprm(struct linux_binprm *bprm)
1041{
1042         int mode;
1043         struct inode * inode = bprm->file->f_path.dentry->d_inode;
1044         int retval;
1045    // Se guarda el modo, que reflejará los permisos del fichero a ejecutar
```

```
1046        mode = inode->i_mode;
1047        if (bprm->file->f_op == NULL)
1048                return -EACCES;
1049
```

Se inicializan los campos e_uid y e_gid de la estructura linux_binprm, teniendo en cuenta los valores de los flags seguid y setgid del archivo ejecutable. Estos campos representan los user y grupo ID's respectivamente. También se chequean las capacidades del proceso.

```
1050        bprm->e_uid = current->euid; // euid efectivo, el del proceso actual
1051        bprm->e_gid = current->egid; // egid efectivo, el del proceso actual
1052
1053        if(!(bprm->file->f_path.mnt->mnt_flags & MNT_NOSUID)) {
1054                /* Set-uid? */
```

Si esta activado el bit de SETUID, entonces el nuevo programa tendrá euid efectivo diferente, y por ello tiene que cambiarse. El nuevo euid será el uid del propietario del fichero

```
1055                if (mode & S_ISUID) {
1056                        current->personality &= ~PER_CLEAR_ON_SETID;
1057                        bprm->e_uid = inode->i_uid;
1058                }
1059
1060                /* Set-gid? */
1061                /*
1062                 * If setgid is set but no group execute bit then this
1063                 * is a candidate for mandatory locking, not a setgid
1064                 * executable.
1065                 */
```

Similar si esta activado el bit SETGID, donde ahora se cambia el grupo efectivo al del propietario del fichero gid.

```
1066                if ((mode & (S_ISGID | S_IXGRP)) == (S_ISGID | S_IXGRP)) {
1067                        current->personality &= ~PER_CLEAR_ON_SETID;
1068                        bprm->e_gid = inode->i_gid;
1069                }
1070        }
1071
1072        /* fill in binprm security blob */
```

Rellena el campo de seguridad del bprm

```
1073        retval = security_bprm_set(bprm);
1074        if (retval)
1075                return retval;
1076
```

Rellena buf en la estructura bprm

```
1077        memset(bprm->buf,0,BINPRM_BUF_SIZE);
```

Se rellena el campo buf de la estructura linux_binprm con los primeros 128 bytes del archivo ejecutable. Estos bytes incluyen el número mágico del formato del ejecutable y otra información útil para el reconocimiento del archivo ejecutable. Devuelve menor que cero si hubo error al leer; el número de bytes leídos si no ha habido error.

```
1078        return kernel_read(bprm->file,0,bprm->buf,BINPRM_BUF_SIZE);
```

```
1079}
```

Search_Binary_Handler

Linux tiene soporte para distintos formatos de ejecutables. Esto implica que necesite de esta función, Search_binary_handler para identificar que tipo de ejecutable es el que se intenta ejecutar con el exec, y asignarle el manejador más adecuado.

Esta función, se basa en una estructura importante, que es la linux_binfmt. Por tanto, vamos a proceder a explicar esta estructura para posteriormente explicar la función en cuestión.

Estructura: linux_binfmt

Esta estructura contiene una lista enlazada con los distintos formatos que puede ejecutar Linux. Está declarada en include/linux/binfmts.h

```
  71 struct linux_binfmt {
  72     struct list_head lh;
  73     struct module *module;
  74     int (*load_binary)(struct linux_binprm *, struct  pt_regs * regs);
  75     int (*load_shlib)(struct file *);
  76     int (*core_dump)(long signr, struct pt_regs *regs, struct file *file,
unsigned long limit);
  77     unsigned long min_coredump;     /* minimal dump size */
  78     int hasvdso;
  79 };
```

Vemos que esta estructura proporciona funciones para poder acceder a dicha lista enlazada. Así, con next podemos acceder al siguiente elemento de la lista enlazada. Además de esto, presenta 3 funciones asociadas al formato que esté actualmente seleccionado de la lista enlazada, que son:

load_binary: carga ficheros, y, si no puede cargarlo, devuelve un valor negativo.
load shlib: sirve para buscar librerías compartidas. Actualmente solo se usa uselib.
core_dump: sirve para volcados de memoria.

De todas estas funciones, nos va a interesar la load_binary. El Search_Binary_Handler irá recorriendo la lista de formatos y probando con load_binary si ese formato es el adecuado para el ejecutable contenido en bprm.

Esta estructura es protegida por un cierre read-write binfmt_lock: el binfmt_lock es tomado para leer en la mayoría de las ocasiones excepto para el registro/desregistro de los formatos binarios. Binfmt_lock no es mas que un cerrojo propio de la concurrencia, para evitar problemas concurrentes.

El cierre read-write consiste en que uno puede tener múltiples lectores (de una variable, o en este caso, de la variable que es de tipo linux_binfmt) a la vez pero sólo un

escritor y no puede haber lectores mientras hay escritores - esto es, el escritor bloquea a todos los lectores, y los nuevos lectores se bloquean mientras un escritor está esperando.

Search_binary_handler.

Busca el manejador para el tipo de ejecutable del que se trate. Esto es necesario, ya que linux maneja distintos tipos de ejecutables con estructuras de fichero diferentes.

Esta búsqueda la hará mediante un recorrido de la lista enlazada de formatos del tipo antes especificado (linux_binfmt).

```
1157 /*
1158  * cycle the list of binary formats handler, until one recognizes the image
1159  */
1160 int search_binary_handler(struct linux_binprm *bprm,struct pt_regs *regs)
1161 {
1162         unsigned int depth = bprm->recursion_depth;
1163         int try,retval;
```

Crea una estructura de tipo binfmt

```
1164         struct linux_binfmt *fmt;
```

Si es un procesador Alpha

```
1165 #ifdef __alpha__
...
...
1199 #endif
```

Chequea los bits se seguridad de la estructura binprm, y en caso de que hayan problemas, salir.

```
1200         retval = security_bprm_check(bprm);
1201         if (retval)
1202                 return retval;
1203
1204         /* kernel module loader fixup */
1205         /* so we don't try to load run modprobe in kernel space. */
1206         set_fs(USER_DS); // Sistema de ficheros del usuario actual
1207
1208         retval = audit_bprm(bprm);
1209         if (retval)
1210                 return retval;
1211
1212         retval = -ENOENT;
```

Se van a hacer justo dos búsquedas. En cada una de ellas, se va a recorrer la estructura enlazada binfmt, y por tanto, se va a intentar asignar como manejador todos y cada uno de los formatos, hasta que uno coincida.

```
1213         for (try=0; try<2; try++) {
1214                 read_lock(&binfmt_lock);
1215                 list_for_each_entry(fmt, &formats, lh) {
```

Para cada formato se llama a load_binary para buscar un manejador, en ese caso sale del bucle

```
1216                         int (*fn)(struct linux_binprm *, struct pt_regs *) =
```

```
                        fmt->load_binary;
1217                    if (!fn)
1218                            continue;
```

En caso de que si se cargue se comprueba que realmente el fichero puede ser ejecutado y que el manejador buscado es el adecuado. Si no se puede traer el módulo, ir a por el siguiente formato.

```
1219                    if (!try_module_get(fmt->module))
1220                            continue;
```

Abre cerrojo en modo lectura

```
1221                    read_unlock(&binfmt_lock);
1222                    retval = fn(bprm, regs);
1223                    /*
1224                     * Restore the depth counter to its starting value
1225                     * in this call, so we don't have to rely on every
1226                     * load_binary function to restore it on return.
1227                     */
1228                    bprm->recursion_depth = depth;
```

Si se retorna un buen valor, añade información a bprm, y retorna el valor, positivo.

```
1229                    if (retval >= 0) {
1230                            if (depth == 0)
1231                                    tracehook_report_exec(fmt, bprm,
regs);
1232                            put_binfmt(fmt);
1233                            allow_write_access(bprm->file);
1234                            if (bprm->file)
1235                                    fput(bprm->file);
1236                            bprm->file = NULL;
1237                            current->did_exec = 1; // Se hizo un exec
1238                            proc_exec_connector(current);
1239                            return retval;
1240                    }
1241                            read_lock(&binfmt_lock); // Cierra cerrojo en modo
lectura
1242                    put_binfmt(fmt);
1243                    if (retval != -ENOEXEC || bprm->mm == NULL)
1244                            break;
1245                    if (!bprm->file) {
1246                            read_unlock(&binfmt_lock);
1247                            return retval;
1248                    }
1249            }
1250            read_unlock(&binfmt_lock);
```

Se contempla la posibilidad de no tener ni que dar la segunda iteración

```
1251            if (retval != -ENOEXEC || bprm->mm == NULL) {
1252
                break;
```

El modo más elegante para emplear módulos de kernel es el uso del cargador de módulos del kernel. Kmod permanece en segundo plano y se ocupa de cargar automáticamente los módulos con llamadas a modprobe cuando se necesita la correspondiente función del kernel. Para usar el Kmod se debe activar, durante la configuración del kernel, la opción ‘Kernel module loader’ (CONFIG_KMOD).

```
1253 #ifdef CONFIG_MODULES
```

```
...
...
1262 #endif
1263                 }
1264         } // De no ser así, al no encontrarse manejador, hace un 2º intento
1265         return retval; // Finalmente se retorna retval
1266}
```

LOAD_BINARY()

A continuación vamos a detallar los pasos que se dan en la función load_binary() que no solo comprobará si el fichero ejecutable es de un determinado formato, sino que también lo cargará en conjunción con la búsqueda y carga de las diferentes librerías compartidas que se requieren para la ejecución del mismo.

Hay que destacar que en realidad esta función no existe como tal, sino que existe una implementación específica de este para cada formato de ejecutable. De todos modos, vamos a comentar los pasos generales que realizarán todas las implementaciones de load_binary().

1. Cada función chequeará que el número mágico que se encuentra en los primeros 128 bytes corresponde con su formato de fichero ejecutable. Si ninguno de los números mágicos corresponde, se devolverá el anteriormente comentado –ENOEXEC.

2. Se lee la cabecera del fichero ejecutable. En esta cabecera se describen los segmentos del programa y las librerías compartidas requeridas.

3. Se obtiene del ejecutable el pathname del linkador dinámico, que es usado para localizar las librerías y cargarlas en la memoria.

4. Obtiene los objetos dentry, inode y file del linkador dinámico.

5. Chequea los permisos de ejecución del linkador dinámico.

6. Copia los primeros 128 bytes del linkador dinámico en un buffer.

7. Se realizan algunas comprobaciones de consistencia en el linkador dinámico.

8. Se invoca a flush_old_exec() para liberar y limpiar casi todos los recursos usados por la ejecución anterior (el proceso viejo que hizo la llamada al exec). Se realizan acciones como:

    a) Resetear con los valores por defecto la estructura de manejadores de señales.

    b) Se cierran todos los ficheros abiertos.

    c) Se liberan todas las regiones de memoria y páginas asignadas al proceso.

    d) Se limpian los valores de los registros en punto flotante que se encuentran guardados en el segmento TSS.

Como es de prever, a partir de este momento no hay vuelta atrás. Si ocurre cualquier error ya no se puede retornar a la llamada del execve(), ya que el código de ese proceso no existe, al igual que el valor de las variables, etc.

9. Se generan las regiones de memoria requeridas para cargar los segmentos del nuevo ejecutable.

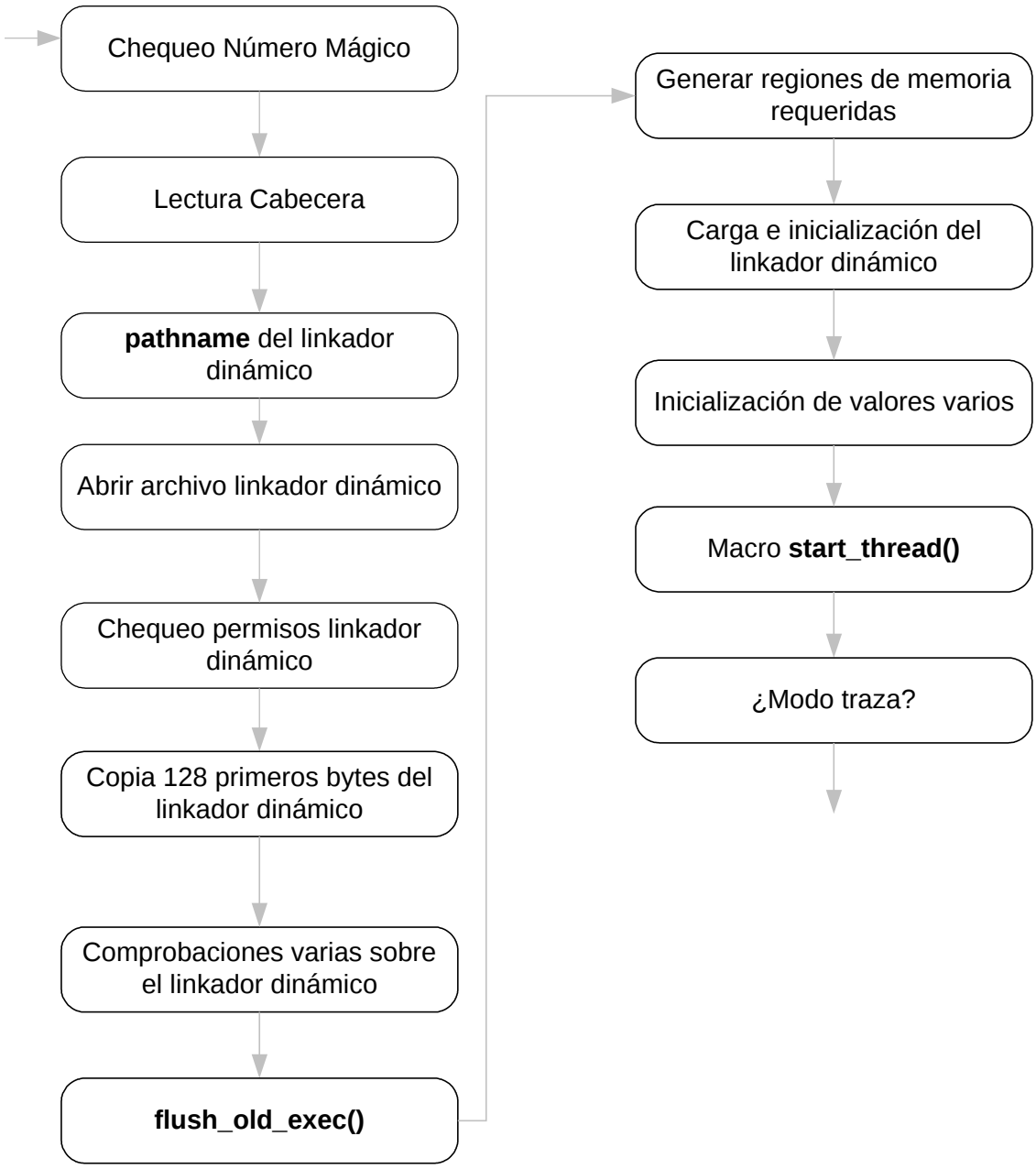
10. Se llama a la función que cargará e inicializará el enlazador dinámico. Se procura cargar el enlazador dinámico en zonas de memoria altas, para evitar colisiones con los segmentos del programa que va a ser ejecutado.

11. Se inicializan valores como start_code, end_code, start_data, end_data, start_stack, etc. para indicar donde se encuentran las zonas de código, datos, pila, etc, en la memoria.

12. Se invoca a la macro start_thread() para modificar los valores de los registros eip y esp, guardados en la pila del Modo Kernel, para que apunten al punto de entrada del linkador y a la base de la pila del Modo de Usuario, respectivamente.

13. Si estamos en modo traza, se notificará al depurador que la llamada al sistema execve() se ha completado con éxito.

Resumen


Chequeo Número Mágico
Lectura Cabecera
pathname del linkador dinámico
Abrir archivo linkador dinámico
Chequeo permisos linkador dinámico
Copia 128 primeros bytes del linkador dinámico
Comprobaciones varias sobre el linkador dinámico
flush_old_exec()
Generar regiones de memoria requeridas
Carga e inicialización del linkador dinámico
Inicialización de valores varios
Macro **start_thread()**
¿Modo traza?

En este punto, se sale de la función load_binary(), a su vez se retorna de la función search_binary_handler() y se sale de la llamada del sistema. Con lo que pasamos de nuevo al modo Usuario y se reanudará la ejecución del proceso llamador. Pero en este punto, el escenario de trabajo del proceso ha cambiado drásticamente, estando ahora otro código, en otro lugar, etc. Por esto se suele decir que la llamada al sistema execve() no retorna nunca si todo ha ido bien. En su lugar tenemos un nuevo programa en el espacio de direccionamiento del proceso.

Sin embargo, el nuevo programa no puede comenzar su ejecución todavía, porque el enlazador dinámico debe cargar las librerías compartidas.

Si el programa fuese enlazado estáticamente, la llamada a load_binary() correspondiente se simplificaría mucho, limitándose a cargar los diferentes segmentos y preparar los diferentes registros.

El enlazador dinámico tendrá ahora la misión de alojar en memoria las diferentes librerías que requiere el programa. Luego deberá buscar en el programa todos los símbolos que se refieren a estas librerías y deberá cambiarlas por las direcciones de memoria donde se encuentran alojadas.

Una vez concluido este proceso, ya está listo el programa para lanzarse a ejecutar, por lo que el enlazador saltará al punto de entrada del programa, comenzando entonces la ejecución real del programa.

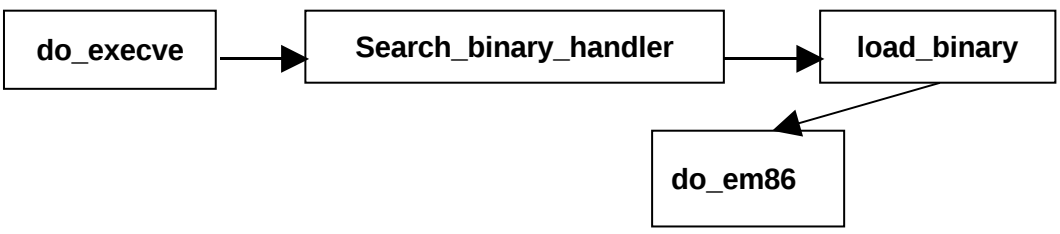
9.4 Ejemplo: em86

A continuación vamos a ver, a modo de ejemplo, uno de los manejadores que posee Linux, que es el que da soporte al formato EM86. Se encuentra en el fichero fs/binfmt_em86.c.

De esta forma, este fichero contiene las funciones encargadas de manejar los ficheros de ejecución binarios de Intel Linux en una máquina Alpha, como si fuera un fichero nativo binario Alpha.

Al igual que sucedía con el exec, este fichero tiene una función principal (y por tanto que hace las veces del manejador de ficheros de formato em86), y después un conjunto de funciones auxiliares que dan suporte a esta función principal. En este caso, la función principal es load_em86.

Para enlazar con lo ya explicado sobre el exec, vamos a ver la secuencia que activa a este manejador:



Así, observamos que es invocado por el load_binary en el proceso de recorrido de la lista enlazada de formatos en busca del formato adecuado al programa, en la función Search_Binary_Handler anteriormente explicada.

```
98 struct linux_binfmt em86_format = { //Define un linux_binfmt donde el load
binary
99 .module = THIS_MODULE, // es la función manejadora del em86.
100 .load_binary = load_em86,
101 };
102
103 static int __init init_em86_binfmt(void) //procedimiento inicializador para
104 { //la estructura binfmt creada.
105 return register_binfmt(&em86_format);
106 }
107
108 static void __exit exit_em86_binfmt(void) // procedimiento finalizador.
109 {
110 unregister_binfmt(&em86_format);
111 }
112
113 core_initcall(init_em86_binfmt);
114 module_exit(exit_em86_binfmt);
115 MODULE_LICENSE("GPL");
```

Como podemos observar en este extracto de código del fichero binfmt_em86, se agrega en la lista enlazada al formato em86, (como em86_format), asignándole al load_binary de este elemento de la lista enlazada la función load_em86 (línea 100), lo cual quiere decir que, cuando en Search_Binary_Handler se vaya recorriendo la lista enlazada de formatos, y se llegue al formato em86_format, al hacer la llamada a load_binary éste invocará a la función load_em86, que es la función manejadora y que veremos a continuación:

Do_em86

Es la función que maneja a los ejecutables con formato em86, y consiste en lo siguiente:

```
27 static int load_em86(struct linux_binprm *bprm,struct pt_regs *regs)
28 {
29 char *interp, *i_name, *i_arg;
30 struct file * file;
31 int retval;
32 struct elfhdr elf_ex;
33
34 /* Comprueba que realmente es un ejecutable de este formato... */
35 elf_ex = *((struct elfhdr *)bprm->buf);
37 if (memcmp(elf_ex.e_ident, ELFMAG, SELFMAG) != 0) //Para ello revisa el número
mágico.
38 return -ENOEXEC;
39
40 /* Chequea la consistencia del fichero, buscando posibles errores */
41 if ((elf_ex.e_type != ET_EXEC && elf_ex.e_type != ET_DYN) ||
42 (!((elf_ex.e_machine == EM_386) || (elf_ex.e_machine == EM_486))) ||
43 (!bprm->file->f_op || !bprm->file->f_op->mmap)) {
44 return -ENOEXEC;
}
//rellena algunos parámetros de bprm, donde se configura información de un proceso.
```

```
47 bprm->sh_bang++;
48 allow_write_access(bprm->file);
49 fput(bprm->file);
50 bprm->file = NULL;

52 // a menos que estemos en el caso de un script, no tenemos que complicarnos cn tareas de búsqueda para encontrar nuestro interprete.
55 interp = EM86_INTERP;
56 i_name = EM86_I_NAME;
i_arg = NULL; //Se reserva la posibilidad de añadir un argumento en el futuro.

//Ahora se obtiene el nombre que el interprete da a argv[0], los argumentos del interprete (opcionales), y finalmente el directorio y nombre del fichero emulado (que reemplaza a arv[0].
//Esto se hacer en orden inverso.
67 remove_arg_zero(bprm);
68 retval = copy_strings_kernel(1, &bprm->filename, bprm);
69 if (retval < 0) return retval; // se tratan errores
70 bprm->argc++;
71 if (i_arg) {
72 retval = copy_strings_kernel(1, &i_arg, bprm);
73 if (retval < 0) return retval; // se tratan errores
74 bprm->argc++;
75 }
76 retval = copy_strings_kernel(1, &i_name, bprm);
77 if (retval < 0) return retval; // se tratan errores
78 bprm->argc++;

80
// ya en este punto, reejecutamos el proceso, pero ahora con el inodo del interprete. Fíjate que usamos open_exec tal y como está ahora el kernel, y no necesitamos hacer copias.
Como volver a ejecutar el exec, pero ahora sobre el interprete
85 file = open_exec(interp);
86 if (IS_ERR(file))
87 return PTR_ERR(file);
88
89 bprm->file = file;
90
91 retval = prepare_binprm(bprm);
92 if (retval < 0)
93 return retval;
94
95 return search_binary_handler(bprm, regs);
96 }
97
```

Como anotación, mencinar que en la línea 55 y 56 se dan las macros EM86_INTERP, y EM86_I_NAME (referentes a directorio en el que se encuentra el interprete de em86) que se definen en la cabecera del fichero, de la forma:

```
24 #define EM86_INTERP "/usr/bin/em86"
25 #define EM86_I_NAME "em86"
```

9.5 Formatos ejecutables

Linux soporta de forma estándar múltiples formatos de ficheros ejecutables. A pesar de que la adopción del tipo ELF está ampliamente extendida, la compatibilidad hacia atrás obliga a permitir la ejecución del resto de formatos. Uno de los problemas que acarrea este hecho es el diseño de las funciones de detección de qué ficheros contienen instrucciones ejecutables y cuáles no.

La mayor parte de los sistemas operativos dispone de un mecanismo de detección para averiguar si un fichero es ejecutable o no. Este mecanismo consiste en situar en los primeros 128 bytes del fichero ejecutable un código denominado "número mágico" que el sistema operativo es capaz de identificar. Linux ha permanecido leal a este principio en todos los formatos de archivos ejecutables que soporta; sin embargo, ha utilizado una aproximación distinta para detectar cómo se deben emplazar las páginas del proceso en memoria: en lugar de almacenar cada uno de los procedimientos a nivel interno al kernel, utiliza unos módulos externos al kernel donde se ubican los manejadores que se ocupan de preparar el archivo ejecutable en memoria listo para ejecutarse.

Esta aproximación es muy interesante, debido a que el sistema también provee de procedimientos que permiten definir nuevos formatos ejecutables siempre que se le asigne un manejador que pueda usarse para alojar las páginas en memoria.
Los formatos ejecutables por el kernel de linux son los siguientes:

a.out: El antiguo y clásico formato de objeto. Usa una cabecera corta y compacta con un número mágico en el inicio que es usado para caracterizar el formato. Contiene tres segmentos: .text, .data y .bss más una tabla de símbolos y una tabla de cadenas de caracteres.
elf: (formato lincable y ejecutable. Es el formato binario estándar de linux, reemplazando al a.out, debido a su portabilidad y a su capacidad para cargar bibliotecas compartidas durante la ejecución.
som: Es un formato binario ejecutable heredado de HP/UX.

Otros manejadores contenidos en el kernel son:

Binfmt_Em86: Ejecuta binarios Linux/Intel ELF como un archivo binario nativo de Alpha en una máquina Alpha.
Binfmt_FLAT: Soporta formatos binarios FLAT.
Binfmt_FLATZ: Soporta formatos binarios comprimidos FLAT.
Binfmt_Misc: es un manejador utilizado para proveer a linux de la posibilidad de agregar formatos binarios adicionales.

El manejador binfmt_misc es un manejador que provee la posibilidad de agregar formatos binarios adicionales al kernel sin compilar, como un módulo kernel adicional.
Para ello, el manejador binfmt_misc necesitará un número mágico en el comienzo del fichero o una extensión para reconocer el tipo de fichero binario.

El binfmt_misc es un manejador que no trae definido un formato binario por defecto. Este manejador mantendrá una lista enlazada de estructuras, que contendrán una descripción de un formato binario, incluyendo el número mágico con su tamaño (o una extensión), un offset y una máscara, y el nombre de un intérprete.

En una petición de ejecución de un formato binario definido, el manejador invocará al intérprete especificado. Para las versiones, a partir de la versión 2.4.13, necesitaremos montar binfmt_misc antes. Para montarlo usaremos la sentencia:

mount -t binfmt_misc none /proc/sys/fs/binfmt_misc.

Nos situaremos en la carpeta "/proc/sys/fs/binfmt_misc" mediante el comando cd.
Podemos encontrar los archivos y directorios siguientes relacionados con el binfmt_misc en este directorio:

register: Este fichero se utiliza para registrar un nuevo formato binario.

Podemos registrarlo con el siguiente comando:

echo :name:type:offset:magic:mask:interpreter: > register

Status: Mediante la sentencia "cat status", obtenemos el estado actual (activado/desactivado) de binfmt_misc. Se puede cambiar el estado de binfmt_misc escribiendo mediante el comando "echo" un 0 para deshabilitar, o un 1 habilitar.
Si en lugar de escribir un 0 o un 1 escribimos un -1, se eliminarán los formatos binarios que hayamos creado, además de desactivar el binfmt_misc.

name: (donde name es el nombre del registro). Este fichero hace exactamente lo mismo que el fichero status, excepto por que su acción está limitada al actual formato binario. Haciendo un "cat name" se obtendrá información sobre el interprete, el número mágico, etc.

Para registrar un nuevo tipo de binarios, debe crear una cadena de caracteres :nombre:tipo:posición:magic:máscara:intérprete:
(donde puede escoger el ':' según sus necesidades) y escribirla en /proc/sys/fs/binfmt_misc/register.

Esto es lo que significan los campos:

- '*nombre*' es una cadena de identificación. Un nuevo fichero /proc se creará con ese nombre bajo /proc/sys/fs/binfmt_misc.
- *'tipo'* es el tipo de reconocimiento. Ponga 'M' para magic y 'E' para extensión.
- *'posición'* es la posición de la máscara magic en el fichero, contado en bytes. Por defecto es 0 si lo omite (p.e. si escribe ':nombre:tipo::magic...').
- *'magic'* es la secuencia de bytes que buscará binfmt_misc. La cadena mágica puede contener caracteres codificados en hexadecimal como \x0a o \xA4. En un entorno de shell deberá escribir \\x0a para evitar que el shell elimine su \. Si escoge una identificación por extensión de fichero, esta es la extensión que se reconocerá (sin el '.', los caracteres especiales \x0a no se permiten). ¡El ajuste por extensión es sensible a las mayúsculas!
- *'máscara'* es una máscara opcional (es 0xff por defecto). Usted puede enmascarar algunos bits mediante ajuste si proporciona una cadena del mismo tipo que la de magic y de su misma longitud. Con la máscara y la secuencia del fichero se realiza una función lógica "and".
- *'intérprete'* es el programa que debería ser llamado con el ejecutable como primer argumento (especifique la ruta completa).

Hay algunas **restricciones**:

- la cadena de registro completa no puede superar los 255 caracteres.
- la cadena magic debe residir en los primeros 128 bytes del fichero, p.e. situación +tamaño(magic) debe ser menor de 128.
- la cadena del intérprete no puede superar los 127 caracteres.

Un ejemplo de uso de binfmt_misc (emulando binfmt_java)

```
cd /proc/sys/fs/binfmt_misc
echo ':Java:M::\xca\xfe\xba\xbe::/usr/local/java/bin/javawrapper:' > register
echo ':HTML:E::html::/usr/local/java/bin/appletviewer:' > register
echo ':Applet:M::<!--applet::/usr/local/java/bin/appletviewer:' > register
echo ':DEXE:M::\x0eDEX::/usr/bin/dosexec:' > register
```

Estas tres líneas añaden soporte para ejecutables Java y applets Java (como hacía binfmt_java, y además reconociendo la extensión .html para que no haga falta poner <!--applet> por cada applet). Tiene que instalar el JDK y el shell-script /usr/local/java/bin/javawrapper también. Éste proporciona un arreglo de los problemas de Java del manejo de nombres de fichero. Para añadir un binario Java, simplemente cree un enlace al fichero clase en algún directorio accesible normalmente.

Dado que este sistema es totalmente ajeno al cometido de la llamada al sistema descrita en este documento, se deja a cargo del lector que investigue este tipo de diseños así como el proceso correcto para usarlo si está interesado.

9.6 Bibliografía

Linux cross reference

http://lxr.linux.no/

Versión del núcleo 2.6.28.7

Understandind the Linux Kernel (3ª Edición)

Daniel P. Bovet, Marco Cesati

Ed. O'Reilly

2005