Lifia Secure Linux

El diseño y su relación con el modelo formal

Lisex 0.0

Maximiliano Cristiá

Laboratorio de Investigación y Formación en Informática Avanzada

Resumen

Se describe a grandes rasgos el diseño de Lisex 0.0.

Además, se describe cómo debe interpretarse el modelo formal LSM0.0 en términos del diseño del núcleo de Linux. Cada componente del estado de LSM0.0 es proyectada a las estructuras de datos utilizadas por el núcleo; y cada llamada al sistema es interpretada en función de una o varias de las operaciones del modelo formal.

Por otro lado, al establecer la relación entre operaciones del modelo y llamadas al sistema operativo, se define la interfaz del núcleo de seguridad, el cual a su vez es la implementación del concepto de monitor de referencias. Se analizan tanto las llamadas que tienen relevancia para la seguridad del sistema como aquellas que no comprometen la información.

Esta es una versión preliminar e incompleta del documento.

Índice general

Capítulo 1

Diseño

En Lisex 0.0 tratamos de respetar el diseño del núcleo de Linux tanto como fue posible. Sin embargo, debido a las nuevas características de seguridad, fue necesario agregar y modificar ciertas estructuras y funciones del núcleo. Al hacerlo aplicamos nuestro criterio. En lo que sigue daremos una visión general sobre el diseño del núcleo de Lisex 0.0 y su relación con el diseño del núcleo de Linux.

1.1. Introducción

Linux respeta la arquitectura básica de la mayoría de los sistemas operativos: un estrato, capa o componente reune el software a nivel de usuario o software de aplicación, y otro estrato contiene el software del núcleo del sistema operativo (ver figura 1.1). Estas dos capas se comunican mediante llamada a procedimiento y memoria compartida. La capa superior (software de usuario) solicita servicios a la capa inferior (núcleo) por medio de la ejecución de *llamadas al sistema*. El núcleo ejecuta en modo protegido, es decir, el hardware impide que programas en la capa superior accedan a software o datos en la capa inferior de forma arbitraria. Cualquier interacción iniciada en la capa superior debe hacerse ejecutando llamadas al sistema. Lisex 0.0 respeta este modelo arquitectónico.

El núcleo puede dividirse en sub-sistemas (administración de memoria, administración de procesos, sistema de archivos, etc.) los que se forman dividiendo en grupos o conjuntos las llamadas al sistema. Esta división en algunos casos no es clara ni precisa (por ejemplo, algunas funciones de administración de procesos pueden accederse a través del sistema de archivos). A nivel de código fuente, estos sub-sistemas se representan por medio de uno o más archivos `.c` y `.h`, en algunas ocasiones ubicados en diferentes directorios (por ejemplo, el directorio `fs` contiene todos los archivos `.c` y `.h` relativos al sistema de archivos).

En el interior de cualquiera de estos sub-sistemas, es decir inaccesibles de forma directa desde el espacio de usuario, encontramos funciones auxiliares y estructuras de datos. Las estructuras de datos rara vez están ocultas detrás de una interfaz abstracta (en general por cuestiones de eficiencia aunque en varias ocasiones ambas cualidades, eficiencia y abstracción, podrían combinarse).

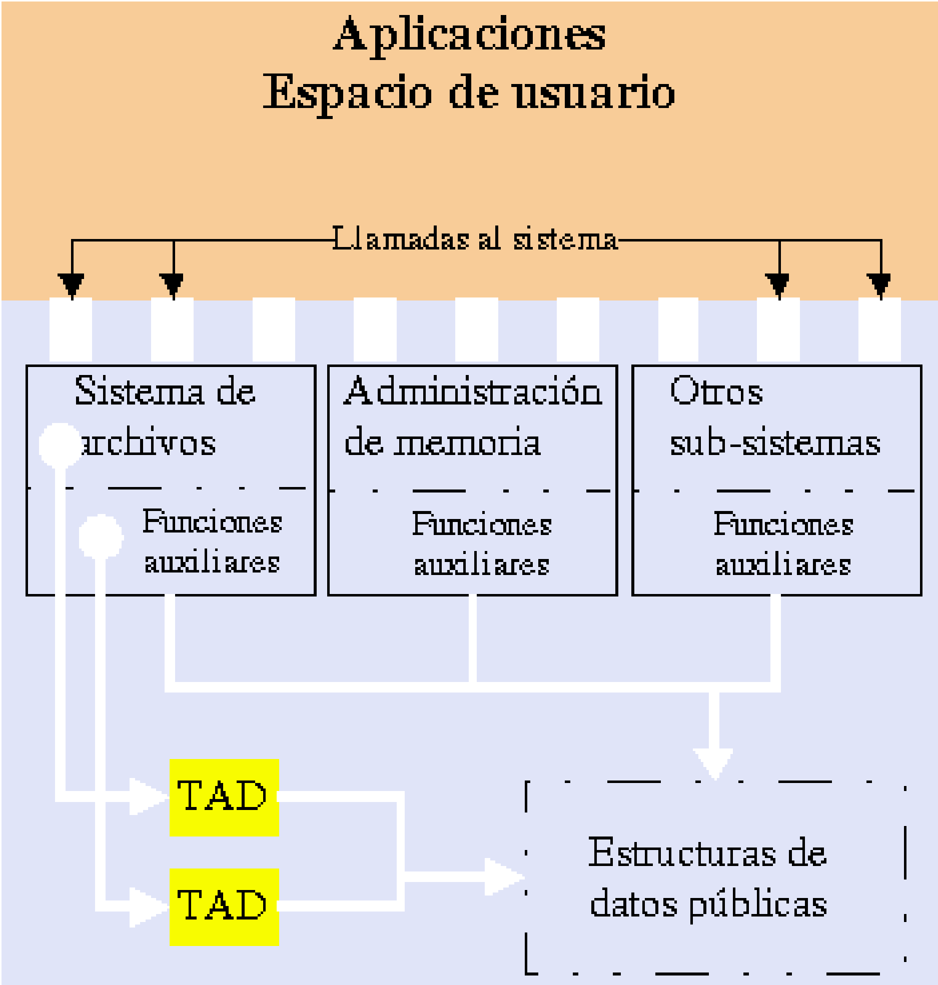

Figura 1.1: Esquema de los componentes del diseño del núcleo de Linux. Se muestran los TADs incluidos en el desarrollo de Lisex. Las flechas indican los patrones más usuales de llamadas a funciones. El esquema no pretende ser completo sino sólo mostrar los componentes de interés para el presente documento.

1.2. El sistema de archivos

Como ya hemos dicho, Lisex 0.0 es una modificación del núcleo de Linux restringida al sistema de archivos por lo que nos concentraremos es este sub-sistema específico.

El sistema de archivos de Linux puede dividirse en varios componentes: VFS[1] y un componente por cada sistema de archivos registrado ante el VFS (ver figura 1.1). Uno de los tipos principales de sistemas de archivos que se registran ante el VFS está constituido por los sistemas de archivos *reales* o *físicos*. Otros, pueden ser sistemas de archivos como `/proc` que usan el poder de VFS para organizar y hacer accesible información que no será, en general, persistente. El sistema de archivos real o físico más usando en Linux es EXT2[2].

[1] Virtual File System.
[2] Second Extended file system.

En términos generales, los sistemas de archivos físicos o reales se encargan de la administración de archivos y directorios a bajo nivel en contacto directo con los dispositivos de almacenamiento, mientras que el VFS se encarga de mostrar una representación e interfaz común de todos los sistemas de archivos que se han registrado (por ejemplo, abrir un archivo almacenado en un sistema de archivos tipo FAT es, para las aplicaciones, indistinto de abrir un archivo almacenado en EXT2 pues en ambos casos la aplicación se comunica con VFS y no con cada sistema de archivos particular).

El conector que conecta VFS con los sistemas de archivos reales es llamada a procedimiento y en menor medida memoria compartida.

Lisex 0.0 introdujo cambios a nivel de VFS y EXT2. Estos cambios incluyen adición y modificación de llamadas al sistema, funciones auxiliares, y estructuras de datos. En las secciones siguientes se hará referencia únicamente a las adiciones pues son estas las que alteran el diseño; las modificaciones a componentes existentes se describen con detalle en [GM02] (también se podrán encontrar más detalles sobre las adiciones en ese manual).

1.3. Nuevas llamadas al sistema

Lisex 0.0 expande el sistema de archivos agregando las siguientes llamadas al sistema:

— `sys_acladd()`, agrega permisos a un usuario o grupo en la ACL de un archivo o directorio.

— `sys_acldel()`, elimina permisos de un usuario o grupo en la ACL de un archivo o directorio.

— `sys_oscstat()`, retorna la clase de acceso de un archivo o directorio.

— `sys_chobjsc()`, modifica la clase de acceso de un archivo o directorio.

— `sys_aclstat()`, retorna la ACL de un archivo o directorio.

— `sys_aclfstat()`, retorna la ACL de un archivo o directorio abierto.

— `sys_chsubsc()`, cambia la clase de acceso de un usuario.

— `sys_sscstat()`, retorna la clase de acceso de un usuario.

Para más detalles ver: el capítulo 2 de este documento, [Cri02], [GM02].

1.4. Nuevas funciones auxiliares

No se agregaron funciones auxiliares a nivel de EXT2. Las funciones auxiliares a nivel de VFS que se agregaron son las siguientes:

— `may_open()`, determina si un archivo puede ser abierto por un proceso o no, usa `permission()` y `mac_permission()`; es usada por `do_execve()`, `may_delete()`, `may_create()`, `sys_setuid()` y `open_namei()`.

— `acl_chmod_ok()`, calcula si se puede realizar `sys_chmod()`; es usada por `sys_chmod()`.

— `is_open()`, determina si un archivo está abierto o no; es usada por `sys_acladd()`, `sys_acldel()`, `sys_hobjsc()`, `sys_chmod()`, `sys_chown()` y `sys_lchown()`.

— `mac_permission()`, implementa seguridad MAC y MLS; es usada por `may_open()`.

— `cp_aclstat()`, carga en una estructura los datos de la ACL de un archivo o directorio; es usada por `sys_aclstat()` y `sys_aclfstat()`.

— `logged()`, determina si un usuario está corriendo algún proceso o no; es usada por `sys_chsubsc()`.

Más detalles se pueden encontrar en [GM02]. Además se debe tener en cuenta que se agregaron otras funciones auxiliares como parte de varios tipos abstractos de datos que se detallan en la sección siguiente.

1.5. Nuevos TADs y estructuras de datos

Lisex 0.0 introduce cuatro tipos abstractos de datos que se detallan más abajo. Por TAD entendemos una estructura de datos mas un conjunto de sub-programas; la estructura sólo puede ser accedida por los sub-programas. La estructura de datos suele denominarse el *secreto* del TAD y los sub-programas la *interfaz*; se dice que este estilo de diseño impone el principio de *ocultacin de la información*[3]. Dado que el lenguaje de implementación del núcleo de Linux es C no es posible imponer el principio de ocultacin de información en tiempo de compilación, como se haría en C++ por ejemplo. En otras palabras, es posible escribir código que modificara o consultara el estado del secreto de un TAD sin usar la interfaz. La única forma de imponer el principio es con disciplina de programación, es decir, los programadores acuerdan usar la interfaz y sólo esta para acceder los secretos de los TADs.

Los TAD que se definieron son los siguientes:

— `vfs-acl`, cuyo secreto es la implementación de las ACL a nivel del VFS y provee funciones para calcular el modo UNIX a partir de una ACL, inicializar una ACL, establecer los permisos de un usuario o grupo, etc.

— `sc`, que oculta la representación de las clases de acceso y provee funciones para obtener y establecer el nivel y las categorías de una clase de acceso, comparar dos clases de acceso, etc.

[3]Information hiding.

— `acl_entry`, cuyo secreto es la representación de la terna $(U\_G\_ID, type, mode)$ que codifica el modo ($mode$) de un usuario (U_G_ID) o grupo (U_G_ID) según lo indica $type$; y provee funciones para alterar y consultar el estado de cada terna.

— `subjectsc`, que oculta la forma precisa en que se almacenan las clases de acceso de los usuarios y provee funciones para inicializar una clase de acceso, obtener y establecer la clase de acceso de un usuario, etc.

Más detalles pueden encontrarse en [GM02].

Capítulo 2

Interpretación de las operaciones

A continuación se listan todas las llamadas al sistema del núcleo de Linux que están relacionadas con el sistema de archivos. Cada una de ellas se asocia con una o varias operaciones del modelo LSM0.0 y se explica el motivo de la asociación.

2.1. Relación entre sujetos, procesos y UIDs

LSM0.0 habla exclusivamente de sujetos mientras que el núcleo de Linux habla de procesos y varios identificadores de usuarios (UIDs). Es sumamente necesario establecer una clara relación entre estos conceptos pues tanto DAC como MLS son mecanismos de control de acceso de tipo IBAC [1].

Linux utiliza los siguientes UIDs para cada usuario del sistema[2]:

UID El identificador de usuario bajo el cual el proceso está ejecutando. En principio corresponde al primer campo del archivo `/etc/passwd` y lo establece el núcleo una vez que el usuario ha ingresado al sistema para el primer proceso de usuario (usualmente el intérprete de comandos). Luego, todo proceso derivado de este hereda este identificador. Únicamente `root` (o el núcleo) puede cambiar y establecer este identificador.

EUID effective UID Normalmente EUID y UID coinciden. Pero si el proceso en cuestión ejecuta SUID a un usuario distinto al que lo lanzó, EUID coincidirá con el del usuario al cual se hizo SUID; en este caso EUID y UID serán diferentes para este proceso. Este identificador puede cambiarse mediante las llamadas `setuid`, `seteuid` o `setreuid`.

FSUID, file system UID Normalmente FSUID y EUID coinciden, aun cuando el proceso ejecute SUID a otro usuario [3]. FSUID fue introducido para ser usado en el contexto de NFS como una forma de evitar ciertos ataques. Todas las llamadas al sistema relacionadas con el sistema de archivos verifican los

[1] IBAC: Identity-Based Access Control

[2] Tomado de: Rusling, D.A., *The Linux Kernel*, pp. 34-35, julio de 1997.

[3] Por lo tanto, SUID, EUID y FSUID normalmente son idénticos

permisos de este identificador, y no de EUID, para otorgar o negar acceso a archivos y directorios (tener en cuenta que normalmente coincide con EUID). Este identificador puede cambiarse mediante la llamada `setfsuid`.

SVUID, saved UID Este identificador coincide siempre con UID. Es parte de los requisitos de compatibilidad POSIX. Se utiliza para almacenar el UID mientras el proceso efectúa un cambio en EUID (es decir mientras EUID es distinto de UID).

Por otro lado, LSM0.0 utiliza al nombre del sujeto que ejecuta la llamada como identificador para determinar los permisos de acceso de la llamada.

Entonces el modelo debe interpretarse de la siguiente forma:

— Sujetos y procesos son equivalentes. Esto significa que si el modelo establece una propiedad sobre los sujetos (por ejemplo confinamiento), esta debe imponerse, a nivel de la implementación, sobre los procesos (es decir, confinamiento debe verificarse para cada PID).

— El nombre del sujeto equivale al FSUID del proceso. Este punto debe ser explicado pues FSUID puede cambiar durante la ejecución del proceso mientras que el modelo no contempla que el nombre del sujeto cambie. Precisamente, el hecho de que el valor del FSUID pueda cambiar durante la ejecución del proceso ha sido abstraído en el modelo con la idea de respetar la autoridad del núcleo de decidir bajo qué identificador se accederá a los datos. El modelo supone que el sistema es capaz de asociar correcta y dinámicamente los permisos de cada sujeto con su nombre. En otras palabras, cada vez que el modelo haga referencia al sujeto que ejecuta cierta llamada, esto debe interpretarse como el valor de FSUID para el proceso desde el cual se invoca la llamada.

LSM0.0 identifica al sujeto que ejecuta la llamada por medio del segundo parámetro de cada operación a nivel de la especificación (llamado `u`).

2.2. handle_sys_nfsservctl

Signatura

```
handle_sys_nfsservctl
   (int cmd, void * opaque_argp, void * opaque_resp)
```

Archivo `nfsctl.c`

Corresponde a ??

Comentario No hay página de manual disponible.

2.3. sys_access

Signatura `long sys_access(const char * filename, int mode)`

Archivo `open.c`

Corresponde a `stat` y `oscstat`

Comentario Efectúa una comprobación de permisos, lo que equivale a solicitar información de control o de los atributos de seguridad por lo que se le deben aplicar las mismas restricciones que a `stat` y `oscstat`.

2.4. sys_acladd

Signatura

```
long sys_acladd(char * filename, int id, int mode, int type)
```

Archivo `acl.c`

Corresponde a `addUsrGrpToAcl`

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.5. sys_acldel

Signatura

```
long sys_acldel(char * filename, int id, int mode, int type)
```

Archivo `acl.c`

Corresponde a `delUsrGrpFromAcl`

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.6. sys_aclfstat

Signatura

```
long sys_aclfstat
   (unsigned int fd, struct acl_entry * acl_statbuf, int len)
```

Archivo `stat.c`

Corresponde a `aclstat`

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.7. sys_aclstat

Signatura

```
long sys_aclstat
   (char * filename, struct acl_entry * acl_statbuf, int len)
```

Archivo `stat.c`

Corresponde a `aclstat`

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.8. sys_bdflush

Signatura `long sys_bdflush(int func, long data`

Archivo `buffer.c`

Corresponde a `write`

Comentario Escribe datos que están en memoria al disco.

2.9. sys_chdir

Signatura `long sys_chdir(const char * filename)`

Archivo `open.c`

Corresponde a Ninguna

Comentario Si la implementación actual sólo hace que los nombres de archivo que referencia el proceso que ejecuta la llamada, sean considerados relativos a `filename`, entonces la seguridad no se ve comprometida. Si por el contrario, la posibilidad de ejecutar la llamada a un `filename` particular implica ciertos permisos sobre dicho directorio, entonces la seguridad podría verse afectada.

Verificar que el comportamiento es el esperado según el comentario anterior.

Verificación

2.10. sys_chmod

Signatura `long sys_chmod(const char * filename, mode_t mode)`

Archivo `open.c`

Corresponde a `chmod`

2.11. sys_chobjsc

Signatura

```
long sys_chobjsc
   (char * filename, int level, int * categories, int size)
```

Archivo `sc.s`

Corresponde a `chobjsc`

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.12. sys_chown

Signatura

```
long sys_chown(const char * filename, uid_t user, gid_t group)
```

Archivo `open.c`

Corresponde a `chown`

Comentario En LSM0.0 los usuarios ordinarios pueden cambiar libremente el propietario y grupo de sus archivos. La semántica actual de Linux para esta llamada restringe innecesariamente las características DAC del sistema sin aumentar su seguridad.

2.13. sys_chroot

Signatura `long sys_chroot(const char * filename)`

Archivo `open.c`

Corresponde a Ninguna

Comentario Si la implementación actual sólo hace que los nombres de archivo que referencia el proceso que ejecuta la llamada, sean considerados relativos a `filename`, entonces la seguridad no se ve comprometida. Si por el contrario, la posibilidad de ejecutar la llamada a un `filename` particular implica ciertos permisos sobre dicho directorio, entonces la seguridad podría verse afectada.

Verificar que el comportamiento es el esperado según el comentario anterior.

Verificación

2.14. sys_chsubsc

Signatura

```
long sys_chsubsc
(uid_t uid, int level, int * categories, int size)
```

Archivo `subjectsc.c`

Corresponde a `chsubsc`

2.15. sys_close

Signatura `long sys_close(unsigned int fd)`

Archivo `open.c`

Corresponde a `close`

2.16. sys_creat

Signatura `long sys_creat(const char * pathname, int mode)`

Archivo `open.c`

Corresponde a `create` seguida de `open` en modo `read-write`.

Comentario Dado que `creat` crea el archivo y luego lo abre en modo `read-write` se le deben aplicar las restricciones de `create` y luego de `open`. En caso de que falle la precondición de `create` o la de `open`, `creat` debe retornar con un error de falta de permisos.

2.17. sys_dup

Signatura `long sys_dup(unsigned int fildes)`

Archivo `fcntl.c`

Corresponde a Ninguna

Comentario Duplica el descriptor de archivo `oldfd` haciendo que el nuevo descriptor, `newfd`, utilice el mismo archivo. Como el archivo al cual terminan apuntando ambos descriptores ya fue abierto, la llamada no compromete la seguridad. Si `newfd` ya está usado, el núcleo lo cierra antes de re-asignarlo, lo que implica que no es posible que el proceso tenga acceso a otro archivo.

2.18. sys_dup2

Signatura `long sys_dup2(unsigned int oldfd, unsigned int newfd)`

Archivo `fcntl.c`

Corresponde a Ninguna

Comentario Duplica el descriptor de archivo `oldfd` haciendo que el nuevo descriptor, `newfd`, utilice el mismo archivo. Como el archivo al cual terminan apuntando ambos descriptores ya fue abierto, la llamada no compromete la seguridad. Si `newfd` ya está usado, el núcleo lo cierra antes de re-asignarlo, lo que implica que no es posible que el proceso tenga acceso a otro archivo.

2.19. sys_execve

Signatura

```
int sys_execve
   (const char * filename, const char * argv[],
    const char * envp[])
```

Archivo `exec.c`

Corresponde a `open` en modo `read`.

2.19.1. Análisis de seguridad de la llamada `sys_execve`

En lo que sigue librerías y programas deben considerarse esencialmente iguales, aunque a las primeras se las accede mediante `uselib` u otras llamadas.

Se considera que ejecutar un programa es equivalente a abrirlo en modo `read`. Por lo tanto, podría haber programas que sólo pueden ejecutar usuarios con determinadas clases de acceso (más allá de los permisos DAC que el programa pudiera exigir). Al mismo tiempo, es posible que esto restrinja (innecesariamente) el uso del sistema pues dado que, excepto el primero, todo programa se ejecuta desde otro, el núcleo impedirá la ejecución si, por ejemplo, el proceso invocante posee archivos abiertos en modo `write` con clase de acceso inferior a la del programa a ejecutar. Por otro lado, esta característica podría ser ventajosa dado que permitiría definir políticas de integridad diferentes (y complementarias) a las posibles en entornos DAC.

Analizaremos las consecuencias de considerar la ejecución de un programa como una acción que no compromete la seguridad desde el punto de vista MLS, es decir una que cualquier usuario puede realizar independientemente de las clases de acceso respectivas[4]. Entonces, podría darse el siguiente ataque:

— El UID u_1 posee clase de seguridad sc_1

— El UID u_2 y el archivo f_2 poseen clase de seguridad sc_2

— $sc_1 \prec sc_2$

— El programa p es un caballo de Troya que intenta enviar la información contenida en f_2 a u_1

— Cualquier UID puede ejecutar p y cualquier otro programa independientemente de la clase de acceso de ambos

Supongamos que u_2 ejecuta p y que este abre en modo `read` el archivo f_2. Luego p escribe el contenido de f_2 en otro archivo, p_{f_2}, en forma de instrucciones `print`, y compila tal archivo generado un programa ejecutable que únicamente escribe el contenido de f_2 en la pantalla[5].

Finalmente, u_1 ejecuta p_{f_2} pues supusimos que no se toman en cuenta las clases de acceso de los usuarios ni los programas a la hora de ejecutarlos. Al ejecutar el programa u_1 recibirá en su pantalla los datos almacenados en f_2 a los que no tenía acceso. Si, como proponemos, la ejecución de p_{f_2} es equivalente a abrirlo en modo `read` será imposible que u_1 lo ejecute ya que el programa heredará la clase de acceso de u_2 pues fue un proceso a su nombre el que lo creó.

Más allá de la posibilidad cierta de ataques como el descripto, creemos que considerar la ejecución de un programa como equivalente a abrirlo en modo `read`, está en perfecta concordancia con la filosofía de LSM0.0 y MLS en general pues:

— Cualquier programa es un archivo en el sistema de archivos

— Es imposible que un programa sea ejecutado sin que sus instrucciones sean leídas

— Podría haber programas que contuvieran, más allá de ser el producto de un ataque, información clasificada en forma de constantes a ser impresas como mensajes para los usuarios.

[4]La formulación original del modelo BLP consideraba que la ejecución de un programa no compromete la seguridad, ver Bell, D., and L. LaPadula, "Secure Computer Systems: Mathematical Foundations and Model", Vols. I-II, ESD-TR-73-278, The MITRE Corporation, Bedford, MA 1973/74.

[5]Claramente las restricciones DAC sobre la ejecución de programas no son un obstáculo serio para p (precisamente esta es la razón fundamental del proyecto Lisex) pues luego de generar el ejecutable podría ejecutar `chmod` o `addUsrGrpToAcl` para ajustar los permisos DAC de manera tal que u_1 pueda ejecutar p_{f_2}.

2.20. sys_fchdir

Signatura `long sys_fchdir(unsigned int fd)`

Archivo `open.c`

Corresponde a Ninguna

Comentario Si la implementación actual sólo hace que los nombres de archivo que referencia el proceso que ejecuta la llamada, sean considerados relativos a `filename`, entonces la seguridad no se ve comprometida. Si por el contrario, la posibilidad de ejecutar la llamada a un `filename` particular implica ciertos permisos sobre dicho directorio, entonces la seguridad podría verse afectada.

Verificar que el comportamiento es el esperado según el comentario anterior.

Verificación

2.21. sys_fchmod

Signatura `long sys_fchmod(unsigned int fd, mode_t mode)`

Archivo `open.c`

Corresponde a Esta llamada está explícitamente prohibida en LSM0.0.

Comentario En LSM0.0 está prohibido el cambio de atributos de seguridad de archivos abiertos, por tal motivo esta llamada debe retornar siempre con error y nunca debe cambiar el modo del archivo abierto. Se la preserva por compatibilidad hacia atrás.

2.22. sys_fchown

Signatura `long sys_fchown(unsigned int fd, uid_t user, gid_t group)`

Archivo `open.c`

Corresponde a Esta llamada está explícitamente prohibida en LSM0.0.

Comentario En LSM0.0 está prohibido el cambio de atributos de seguridad de archivos abiertos, por tal motivo esta llamada debe retornar siempre con error y nunca debe cambiar ni el propietario ni el grupo del archivo abierto. Se la preserva por compatibilidad hacia atrás.

2.23. sys_fcntl

Signatura

```
long sys_fcntl
   (unsigned int fd, unsigned int cmd, unsigned long arg)
```

Archivo `fcntl.c`

Corresponde a Ninguna

Comentario Realiza varias operaciones de control sobre el descriptor de archivo `fd`.

2.24. sys_fcntl64

Signatura

```
long sys_fcntl64
   (unsigned int fd, unsigned int cmd, unsigned long arg)
```

Archivo `fcntl.c`

Corresponde a Ninguna

Comentario No hay página de manual disponible. Debería ser semejante a 2.23.

2.25. sys_fdatasync

Signatura `long sys_fdatasync(unsigned int fd)`

Archivo `buffer.c`

Corresponde a `write`

Comentario Escribe datos que están en memoria al disco.

2.26. sys_flock

Signatura `long sys_flock(unsigned int fd, unsigned int cmd)`

Archivo `locks.c`

Corresponde a No compromete la seguridad.

Comentario Se utiliza para administrar candados en archivos abiertos.

2.27. sys_fsstatfs

Signatura `long sys_fsstatfs(unsigned int fd, struct statfs * buf)`

Archivo `open.c`

Corresponde a Si el archivo está abierto en modo `read` se puede ejecutar la llamada, en caso contrario se debe verificar la precondición de `stat`.

2.28. sys_fstat

Signatura

```
long sys_fstat
   (unsigned int fd, struct __old_kernel_stat * statbuf)
```

Archivo `stat.c`

Corresponde a `stat`

Comentario Si el archivo ya fue abierto en modo `read` no hace falta verificar la precondición de `stat`.

2.29. sys_fstat64

Signatura

```
long sys_fstat64
   (unsigned long fd, struct stat64 * statbuf, long flags)
```

Archivo `stat.c`

Corresponde a `stat`

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.28.

2.30. sys_fstruncate

Signatura `long sys_fstruncate(unsigned int fd, unsigned long length)`

Archivo `open.c`

Corresponde a Ninguna

Comentario Dado que la implementación actual exige que el archivo esté abierto en modo `write` ya se han efectuado en tiempo de `open` todos los controles necesarios. No compromete la seguridad.

2.31. sys_fstruncate64

Signatura `long sys_fstruncate64(unsigned int fd, loff_t length)`

Archivo `open.c`

Corresponde a Ninguna

Comentario No hay página de manual. Debería ser como 2.30.

2.32. sys_fsync

Signatura `long sys_fsync(unsigned int fd)`

Archivo `buffer.c`

Corresponde a `write`

Comentario Escribe datos que están en memoria al disco.

2.33. sys_getcwd

Signatura `long sys_getcwd(char * buf, unsigned long size)`

Archivo `dcache.c`

Corresponde a Ninguna

Comentario Obtiene el directorio de trabajo actual. No compromete la seguridad.

2.34. sys_getdents

Signatura

```
long sys_getdents
   (unsigned int fd, void * dirent, unsigned int count)
```

Archivo `readdir.c`

Corresponde a `readdir`

Comentario LSM0.0 supone que el directorio del cual se van a leer las entradas ya ha sido abierto por `open` en modo `read`.

2.35. sys_getdents64

Signatura

```
long sys_getdents64
   (unsigned int fd, void * dirent, unsigned int count)
```

Archivo `readdir.c`

Corresponde a `readdir`

Comentario No hay página de manual disponible, debería ser semejante a 2.34. LSM0.0 supone que el directorio del cual se van a leer las entradas ya ha sido abierto por `open` en modo `read`.

2.36. sys_ioctl

Signatura

```
long sys_ioctl
   (unsigned int fd, unsigned int cmd, unsigned long arg)
```

Archivo `ioctl.c`

Corresponde a Ninguna

Comentario Es una llamada que se utiliza para controlar el comportamiento de los dispositivos de E/S una vez que han sido abiertos. En ese caso, el proceso que abrió el dispositivo tiene derecho de manipular su comportamiento por lo que `sys_ioctl` puede ser invocada sin restricciones. Sin embargo, esta llamada de ser estudiada con más detenimiento.

Verificar que la implementación actual de la llamada permite leer o escribir archivos comunes, directorios o dispositivos sin antes invocar a `open`.

Verificación

2.37. sys_lchown

Signatura

```
long sys_lchown(const char * filename, uid_t user, gid_t group)
```

Archivo `open.c`

Corresponde a `chown`

Comentario En LSM0.0 los usuarios ordinarios pueden cambiar libremente el propietario y grupo de sus archivos. La semántica actual de Linux para esta llamada restringe innecesariamente las características DAC del sistema sin aumentar su seguridad.

2.38. sys_link

Signatura `long sys_link(const char * oldname, const char * newname)`

Archivo `namei.c`

Corresponde a `create` o `mkdir`

Comentario Crea una liga simbólica, lo que equivale a crear un objeto. Si bien el nuevo objeto tendrá como propietario y clase de acceso aquellos que corresponden al UID efectivo del proceso que invoca la llamada, estos no serán tenidos en cuenta al momento de abrir el objeto, el núcleo tendrá en cuenta los atributos de seguridad de `oldname`.

Verificar que el propietario y la clase de acceso de `newname` no son tenidos en cuenta al momento de abrir el objeto.

Verificar que el propietario y la clase de acceso de `newname` no son tenidos en cuenta al momento de abrir el objeto.

Verificación

2.39. sys_llseek

Signatura

```
long sys_llseek
   (unsigned int fd, unsigned long offset_high,
    unsigned long offset_low, loff_t * result,
    unsigned int origin)
```

Archivo `read_write.c`

Corresponde a Ninguna

Comentario No compromete la seguridad.

2.40. sys_lseek

Signatura

```
off_t sys_lseek
   (unsigned int fd, off_t offset, unsigned int origin)
```

Archivo `read_write.c`

Corresponde a Ninguna

Comentario No compromete la seguridad.

2.41. sys_lstat

Signatura

```
long sys_lstat
   (char * filename, struct __old_kernel_stat * statbuf)
```

Archivo `stat.c`

Corresponde a `stat`

Comentario En LSM0.0 sólo se especifica la información de seguridad que esta llamada debe devolver, la implementación debe retornar los datos especificados más todos aquellos datos que retorne la implementación actual de la llamada.

2.42. sys_lstat64

Signatura

```
long sys_lstat64
   (char * filename, struct stat64 * statbuf, long flags)
```

Archivo `stat.c`

Corresponde a `stat`

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.41.

2.43. sys_mkdir

Signatura `long sys_mkdir(const char * pathname, int mode)`

Archivo `namei.c`

Corresponde a `mkdir`

2.44. sys_mknod

Signatura `long sys_mknod(const char * filename, int mode, dev_t dev)`

Archivo `namei.c`

Corresponde a `create`

Comentario Crea nodos-i, lo que equivale a crear objetos.

2.45. sys_mount

Signatura

```
long sys_mount
   (char * dev_name, char * dir_name, char * type,
    unsigned long flags, void * data)
```

Archivo `namespace.c`

Corresponde a No tiene correspondencia con ninguna operación del modelo; puede afectar la seguridad del sistema.

Comentario Monta y desmonta sistemas de archivos; sólo `root` puede hacerlo. Si `root` monta un sistema de archivos que no impone LSM0.0, un caballo de Troya ejecutando como `root` podría violar la política de seguridad. No existe este problema si tal sistema de archivos se monta de sólo-lectura.

Verificar que las modificaciones efectuadas sobre el sistema de archivos virtual (VFS) impiden el comportamiento anterior; en caso contrario modificar la llamada para que sólo permita montar LFS (Lisex File System) en modo lecto-escritura y cualquier sistema de archivos en modo de sólo-lectura.

Verificación

2.46. sys_newfstat

Signatura `long sys_newfstat(unsigned int fd, structu stat * statbuf)`

Archivo `stat.c`

Corresponde a `stat`

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.28.

2.47. sys_newlstat

Signatura `long sys_newlstat(char * filename, struct stat * statbuf)`

Archivo `stat.c`

Corresponde a `stat`

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.41.

2.48. sys_newstat

Signatura `long sys_newstat(char * filename, struct stat * statbuf)`

Archivo `stat.c`

Corresponde a `stat`

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.66.

2.49. sys_nfsservctl

Signatura `int sys_nfsservctl(int cmd, void * argp, void * resp)`

Archivo `filesystems.c`

Corresponde a

Comentario Permite exportar un sistema de archivos. Observar que hay dos funciones con la misma signatura.

Verificar con más detalle la funcionalidad.

Verificación

2.50. sys_oldumount

Signatura `long sys_oldumount(char * name)`

Archivo `namespace.c`

Corresponde a Ninguna

Comentario No hay página de manual disponible. En principio no debería comprometer la seguridad. Debería ser semejante a 2.74.

2.51. sys_open

Signatura `long sys_open(const char * filename, int flags, int mode)`

Archivo `open.c`

Corresponde a `open`

Comentario Sin embargo debería estudiarse con más detalle la influencia del parámetro `flags` que no fue incluido en el modelo.

2.52. sys_oscstat

Signatura

```
long sys_oscstat
   (char * filename, int * level, int * categories, int len)
```

Archivo `fs_sc.c`

Corresponde a `oscstat`

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.53. sys_poll

Signatura

```
long sys_poll
   (struct pollfd * ufds, unsigned int nfds, long timeout)
```

Archivo `select.c`

Corresponde a Ninguna

Comentario No compromete la seguridad. Podría tener relevancia cuando se aseguren los mecanismos de IPC.

2.54. sys_pread

Signatura `ssize_t sys_pread(unsigned int fd, char * buf, ...)`

Archivo `read_write.c`

Corresponde a `read`

2.55. sys_pwrite

Signatura `ssize_t sys_pwrite(unsigned int fd, char * buf, ...)`

Archivo `read_write.c`

Corresponde a `write`

2.56. sys_quotactl

Signatura

```
long sys_quotactl
   (int cmd, const char * special, int id, caddr_t addr)
```

Archivo `dquot.c`

Corresponde a Ninguna

Comentario Administra cuotas de disco. No compromete la seguridad.

2.57. sys_quotactl

Signatura

```
long sys_quotactl
   (int cmd, const char * special, int id, caddr_t addr)
```

Archivo `noquot.c`

Corresponde a Ninguna

Comentario Coincide en nombre y signatura con otra llamada ubicada en el archivo `dquot.c` (ver 2.58).

Verificar las diferencias entre ambas.

Verificación

2.58. sys_quotactl

Signatura

```
long sys_quotactl
   (int cmd, const char * special, int id, caddr_t addr)
```

Archivo `dquot.c`

Corresponde a Ninguna

Comentario Administra cuotas de disco. No compromete la seguridad.

2.59. sys_read

Signatura `ssize_t sys_read(unsigned int fd, char * buf, size_t count)`

Archivo `read_write.c`

Corresponde a `read`

2.60. sys_readlink

Signatura

```
long sys_readlink(const char * path, cgar * buf, int bufsiz)
```

Archivo `stat.c`

Corresponde a `read`

2.61. sys_readv

Signatura

```
ssize_t sys_readv
   (unsigned long fd, const struct iovec * vector,
    unsigned long count)
```

Archivo `read_write.c`

Corresponde a `read`

2.62. sys_rename

Signatura `long sys_rename(const char * oldname, const char * newname)`

Archivo `namei.c`

Corresponde a Las precondiciones de `create` y `unlink`, o `mkdir` y `rmdir`

Comentario Dado que la llamada borra una entrada de directorio y crea una nueva, se requieren los mismos derechos que `create` y `unlink`, en el caso de un archivo, o que `mkdir` y `rmdir` en el caso de un directorio. No deben verificarse las postcondiciones pues no deben modificarse los atributos MAC que posee `oldname`.

2.63. sys_rmdir

Signatura `long sys_rmdir(const char * pathname)`

Archivo `namei.c`

Corresponde a `rmdir`

2.64. sys_setuid, sys_seteruid, set_setfsuid

Signatura `int sys_setuid(uid_t uid)`

Archivo `sys.c`

Corresponde a Ninguna

Comentario Existe una familia de llamadas semejantes que establecen diferentes identidades. Esta es compatible con POSIX, otras no lo son (`seteuid` es compatible con BSD). Puede encontrarse un análisis detallado del impacto de estas llamadas sobre LSM0.0 en la sección 2.64.1.

Estas llamadas deberían haber sido incluidas en el modelo.

Si un proceso ejecuta una de estas llamadas y tiene archivos abiertos, debe verificarse que el nuevo UID puede abrir esos archivos. En otras palabras, debe verificarse seguridad simple y DAC para el nuevo UID sobre todos los archivos abiertos.

Además, estas llamadas plantean problemas de seguridad en la división de la administración DAC (`root`) y la administración MAC (`secadm`).

2.64.1. Análisis de seguridad de las llamadas `set*uid`

En primer término analizaremos `setuid` y más adelante `seteuid`; consideramos que las llamadas restantes (`setreuid`, `setfsuid`, `setgid`, `setfsgid` y `setregid`) son semejantes.

Según la página de manual de `setuid`:

> [...] establece el UID efectivo del proceso en curso. Si el UID efectivo de quien llama a esta rutina es root, también se establecen los identificadores de usuario real y salvado.
>
> [...] Esto permite que un programa SUID (distinto de root) renuncie a todos sus privilegios de usuario, haga algún trabajo no privilegiado , y luego recupere su ID de usuario efectivo original de una manera segura.
>
> Si el usuario es root o si el programa es SUID a root, deben tomarse precauciones especiales. La función `setuid` comprueba el UID efectivo de quien la llama y si es el super-usuario, todos los IDs relacionados con el proceso se ponen a `uid`. Después de esto, es imposible para el programa recuperar sus privilegios de root.
>
> Así, un programa SUID a root que desea temporalmente renunciar a sus privilegios de root, asumir la identidad de un usuario no-root y luego recuperar sus privilegios de root no puede emplear `setuid`. Esto se puede hacer con la llamada (no POSIX, BSD) `seteuid`.

Entonces, asumiendo que el comportamiento de `setuid` documentado es el implementado, planteamos el siguiente escenario:

1. El UID u_1 y el archivo f_1 poseen clase de seguridad sc_1
2. El UID u_2 y el archivo f_2 poseen clase de seguridad sc_2
3. $sc_1 \prec sc_2$
4. El programa p es un caballo de Troya que intenta enviar la información contenida en f_2 a u_1
5. p puede ser ejecutado por u_2 y corre SUID a u_1

Consideramos dos alternativas:

a. El proceso puede restablecer temporalmente EUID a SUID (es decir, igual a u_2) y luego puede volver a poner EUID a u_1 (lo que implica que FSUID será igual a EUID).

b. El proceso desde su inicio hasta su finalización ejecuta con EUID igual a u_1 (a su vez igual a FSUID).

Supongamos la alternativa a. Una vez que u_2 ejecuta p, este pone EUID a u_2 y abre f_2 en modo `read`[6]. Luego, ejecuta `setuid` y EUID pasa a ser u_1. Pero u_1 no puede abrir f_2 por seguridad simple. Entonces, el núcleo del sistema deberá controlar (en tiempo de `setuid`) que u_1 puede abrir los archivos que ya están abiertos por el proceso. En este caso, `setuid` fallará por falta de permisos.

Si, por el contrario, p primero ejecutara `setuid`, luego abriera f_1, retornara a u_2 y abriera f_2 (con la intención de copiar la información de f_2 en f_1), el núcleo prohibiría esta última petición pues caso contrario se violaría la propiedad de confinamiento ($sc_1 \prec sc_2$).

Supongamos la alternativa b. En esta caso, p no sería capaz de abrir u_2 pues la clase de acceso de u_1, sc_1, está dominada por la de u_2, sc_2.

Ahora supongamos que 5 es:

5'. p puede ser ejecutado por u_1 y corre SUID a u_2

Y consideramos las alternativas simétricas a a y b:

A. El proceso puede restablecer temporalmente EUID a SUID (es decir, igual a u_1) y luego puede volver a poner EUID a u_2 (lo que implica que FSUID será igual a EUID).

B. El proceso desde su inicio hasta su finalización ejecuta con EUID igual a u_2 (a su vez igual a FSUID).

Si es A, u_1 ejecuta p el cual abrirá f_1 en modo `write`, invocará a `setuid`, y como u_2 intentará abrir f_2 pero el sistema lo negará por confinamiento. Si, en cambio, p primero invoca a `suid`, abre f_2 e intenta retornar a u_1, el sistema lo impedirá debido a que u_1 violaría seguridad simple con respecto a f_2.

Si es la alternativa B, habrá un intento de violar confinamiento cuando p quiera abrir al mismo tiempo f_1 y f_2.

Suponiendo que p ejecuta `seteuid` o `setreuid` (en lugar de `setuid`), debemos asumir que u_2 es `root` pues de lo contrario la funcionalidad actual de esas llamadas impedirá cualquier cambio significativo. Suponer que u_2 es `root` no agrega nada nuevo a las alternativas anteriores pues según LSM0.0 `root` es un usuario más con respecto a MLS.

2.65. sys_sscstat

Signatura

```
long sys_sscstat
   (uid_t uid, int level, int * categories, int size)
```

[6] Recordar que el acceso al sistema de archivos se hace en base al valor de FSUID.

Archivo `subjectsc.c`

Corresponde a `sscstat`

Comentario Llamada agregada debido al nuevo modelo de seguridad.

2.66. sys_stat

Signatura

```
long sys_stat
   (char * filename, struct __old_kernel_stat * statbuf)
```

Archivo `stat.c`

Corresponde a `stat`

Comentario En LSM0.0 sólo se especifica la información de seguridad que esta llamada debe devolver, la implementación debe retornar los datos especificados más todos aquellos datos que retorne la implementación actual de la llamada.

2.67. sys_stat64

Signatura

```
long sys_stat64
   (char * filename, struct stat64 * statbuf, long flags)
```

Archivo `stat.c`

Corresponde a `stat`

Comentario No hay página de manual disponible. Debería ser muy semejante a 2.66.

2.68. sys_statfs

Signatura `long sys_statfs(const char * path, struct statfs * buf)`

Archivo `open.c`

Corresponde a `stat`

2.69. sys_symlink

Signatura

```
long sys_symlink(const char * oldname, const char * newname)
```

Archivo `namei.c`

Corresponde a `create` o `mkdir`

Comentario Crea una liga simbólica, lo que equivale a crear un objeto. Si bien el nuevo objeto tendrá como propietario y clase de acceso aquellos que corresponden al UID efectivo del proceso que invoca la llamada, estos no serán tenidos en cuenta al momento de abrir el objeto, el núcleo tendrá en cuenta los atributos de seguridad de `oldname`.

Verificar que el propietario y la clase de acceso de `newname` no son tenidos en cuenta al momento de abrir el objeto.

Verificación

2.70. sys_sync

Signatura `long sys_sync(void)`

Archivo `buffer.c`

Corresponde a `??`

Comentario En principio no compromete la seguridad.

Verificar qué i-nodos entrega.

Verificación

2.71. sys_sysfs

Signatura

```
long sys_sysfs
   (int option, unsigned long arg1, unsigned long arg2)
```

Archivo `super.c`

Corresponde a Ninguna

Comentario Retorna información sobre el tipo de sistema de archivos. No compromete la seguridad.

2.72. sys_truncate

Signatura `long sys_truncate(const char * path, unsigned long length)`

Archivo `open.c`

Corresponde a `open` en modo `write`.

Comentario La condición anterior depende de la implementación de la llamada.

Verificar que la implementación no utiliza `read` ni abre el archivo en modo de lectura; o si lo hace y utiliza `read` verificar que el buffer donde se leen los datos no es accesible por el sujeto que ejecuta la llamada. En caso de no verificarse estas condiciones debe considerarse que la llamada es equivalente a `open` en modo `read-write`.

Verificación

2.73. sys_truncate64

Signatura `long sys_truncate64(const char * path, loff_t length)`

Archivo `open.c`

Corresponde a `open` en modo `write`.

Comentario No hay página de manual disponible. Debería ser como 2.72.

2.74. sys_umount

Signatura `long sys_umount(char * name, int flags)`

Archivo `namespace.c`

Corresponde a Ninguna

Comentario No compromete la seguridad.

2.75. sys_unlink

Signatura `long sys_unlink(const char * pathname)`

Archivo `namei.c`

Corresponde a `unlink`

2.76. sys_uselib

Signatura `long sys_uselib(const char * library)`

Archivo `exec.c`

Corresponde a `open` en modo `read`.

Comentario Selecciona la librería que usará el proceso que invoca la llamada. Por tal motivo se considera que el archivo que contiene la librería deberá ser abierto en modo de lectura, lo que implica que debe verificarse la especificación de `open` en modo `read`. Ver sección 2.19.1.

2.77. sys_ustat

Signatura `long sys_ustat(dev_t dev, struct ustat * ubuf)`

Archivo `super.c`

Corresponde a Ninguna

Comentario Retorna información sobre el tipo de sistema de archivos. No compromete la seguridad.

2.78. sys_utime

Signatura `long sys_utime(char * filename, struct utimbuf * times)`

Archivo `open.c`

Corresponde a Ninguna

Comentario Canal encubierto obvio. Los nodos-i no son considerados como objetos del sistema y por lo tanto pueden ser accedidos libremente. De todas formas se deben preservar los controles existentes en la implementación actual. No compromete la seguridad.

2.79. sys_utimes

Signatura `long sys_utimes(char * filename, struct timeval * utimes)`

Archivo `open.c`

Corresponde a Ninguna

Comentario Canal encubierto obvio. Los nodos-i no son considerados como objetos del sistema y por lo tanto pueden ser accedidos libremente. De todas formas se deben preservar los controles existentes en la implementación actual. No compromete la seguridad.

2.80. sys_vhangup

Signatura `long sys_vhangup(void)`

Archivo `open.c`

Corresponde a Ninguna

Comentario Desde el punto de vista de LSM0.0 no compromete la seguridad pues no está vinculada con el control de acceso. De todas formas, debe tenérsela en cuenta para la implementación de mecanismos de camino confiable (*trusted path*).

2.81. sys_write

Signatura

```
ssize_t sys_write(unsigned int fd, char * buf, size_t count)
```

Archivo `read_write.c`

Corresponde a `write`

2.82. sys_writev

Signatura

```
ssize_t sys_writev
   (unsigned long fd, const struct iovec * vector,
    unsigned long count)
```

Archivo `read_write.c`

Corresponde a `write`

Bibliografía

[Cri02] Maximiliano Cristi. Formal verification of an extension of a secure, compatible unix file system, 2002.

[GM02] Gisela Giusti and Felipe Manzano. Manual de referencia para el programador. lisex 0.0. *Publicacin interna del Lifia*, 2002.