

# Evolución de los esquemas de seguridad extendidos en Unix

Gunnar Eyal Wolf Iszaevich  
gwolf@gwolf.cx

UNAM FES Iztacala

Departamento de Seguridad en Cómputo (DGSCA-UNAM)

Congreso Nacional de Software Libre CONSOL 2003

17 de febrero de 2003

## Resumen

Los sistemas Unix son descendientes del proyecto Multics, un sistema con magníficas ideas pero demasiado complejo para su época. Unix fue en un principio tan sólo una simplificación de Multics, y esto fue lo que lo hizo tan exitoso.

La seguridad en un sistema Unix tradicional está completamente basada en la separación entre *kernel* y *userland*, y en los tradicionales permisos `rw-rw-rw-`.

Sin embargo, con el paso de los años fue haciéndose necesario ampliar este esquema para cubrir las necesidades actuales del mundo real.

En esta plática cubriré:

- Los esquemas de MAC (Mandatory Access Control) existentes en diferentes Unixes (Trusted-\*)
- Las capacidades POSIX 1003.1e, su implementación en Linux
- Systrace, sus implementaciones en BSD y en Linux
- Ventajas/desventajas de estos métodos

# Índice

<b>1. Esquemas de seguridad – ¿Qué? ¿Para qué?</b>	<b>3</b>
<b>2. Modelo de Multics</b>	<b>3</b>
2.1. ¿Qué es Multics? ¿Por qué iniciamos con él? . . . . .	3
2.2. Un sistema de anillos . . . . .	4
2.3. ACLs completos . . . . .	4
<b>3. Unix clásico</b>	<b>5</b>
3.1. Espacios e interfaces . . . . .	5
3.1.1. Espacio de kernel . . . . .	5
3.1.2. Espacio de usuario . . . . .	6
3.1.3. Llamadas al sistema . . . . .	6
3.2. Permisos de uso . . . . .	6
3.2.1. Uso de dispositivos . . . . .	7
3.3. El superusuario . . . . .	7
3.4. La problemática con el esquema tradicional de Unix . . . . .	7
<b>4. Sistemas confiables (<i>Trusted-*</i>)</b>	<b>8</b>
4.1. Trusted Solaris . . . . .	8
4.2. TrustedBSD . . . . .	9
4.2.1. Listas de control de acceso (ACLs) . . . . .	9
4.2.2. Eventos auditables . . . . .	9
4.2.3. Atributos extendidos . . . . .	10
4.2.4. Capacidades granulares . . . . .	10
4.2.5. Control de acceso mandatorio (MAC) . . . . .	10
<b>5. Capacidades POSIX.1e</b>	<b>10</b>
5.1. ¿Qué es el estándar POSIX.1e? . . . . .	10
5.1.1. Su estado actual . . . . .	11
5.2. Implementaciones existentes de las capacidades POSIX.1e . . . . .	11
5.2.1. Las capacidades POSIX en Linux . . . . .	12
<b>6. Systrace</b>	<b>14</b>

## 1. Esquemas de seguridad – ¿Qué? ¿Para qué?

Parte esencial al conocer o comparar cualquier sistema operativo hoy en día debe ser el esquema de seguridad que nos ofrece. Ya sea para sistemas diseñados como servidores o como clientes, conectados o no conectados a red, hoy en día vemos que casi la totalidad de los sistemas operativos manejan a algún grado, aún en el mundo de las PDAs, los conceptos de multitarea, redes y usuarios.

En este texto estudiaremos los esquemas de seguridad implementados por la familia de sistemas operativos probablemente más longeva y exitosa de la historia, los Unixes, así como las nuevas direcciones que nos marcan los recientes desarrollos en los sistemas Unix libres.

## 2. Modelo de Multics

El primer esquema que analizaremos es el implementado por Multics, el cual pese a tener ya cerca de 40 años de existencia aún es tomado como referencia para diseño de nuevos esquemas.

### 2.1. ¿Qué es Multics? ¿Por qué iniciamos con él?

El diseño de Multics (*Multiplexed Information and Computing Service*) fue descrito en 1965 [1] y desarrollado desde entonces y hasta 1973. En su momento fue un sistema operativo muy revolucionario y ambicioso – Planteaba soportar brindar servicios de cómputo como se brindan los servicios telefónico o eléctrico: Los subscriptores tendrían una toma a través de la cual conectarían su terminal al sistema central. Esto exigía un sistema que hoy sería catalogado como *de alta disponibilidad* a todo nivel, y con muy altos estándares de seguridad. Es uno de los primeros sistemas que expresamente buscaba dar soporte a múltiples ambientes de programación e interfaces al usuario, un amplio rango de aplicaciones, y tener la habilidad de evolucionar conforme cambie la tecnología. Multics fue también uno de los primeros sistemas operativos escrito en un lenguaje de alto nivel (en PL/1 [2]), pensando en permitir una gran portabilidad y facilidad en la depuración.

El basar el sistema en PL/1 fue, además, una muy acertada decisión. En más de treinta años prácticamente no se detectaron *buffer overflows* en Multics, uno de los mayores dolores de cabeza de los programadores de C, dado que PL/1 maneja nativamente los límites máximos de las cadenas.

Las ambiciones de Multics fueron, sin embargo, demasiado elevadas para la época. El sistema operativo tardó mucho en estar listo. Era ridículamente grande y lento para las capacidades de memoria y procesamiento aún de las computadoras más grandes de su época. Esto hizo que varias de las empresas que inicialmente lo impulsaron abandonaran su desarrollo. Una de estas fue Bell Labs – Tras abandonar el proyecto Multics, pero partiendo de varias interesantes ideas que éste planteó por primera vez, sus hoy famosos empleados Kernighan, Thompson y Ritchie se avocaron al diseño de un Multics recortado, llamado

—como broma— UNICS y posteriormente Unix, y un lenguaje de relativamente alto nivel, el actual C.

Pese a las demoras en su desarrollo, Multics sí fue un sistema exitoso. Hubo una buena cantidad de sistemas corriendo Multics, el último de los cuales fue *jubilado* en el 2000 en el ejército canadiense. El primer sistema en obtener la clasificación A1 del Departamento de Defensa de los EUA [3] —cosa que definitivamente no ocurre todos los días— fue SCOMP, un desarrollo basado en Multics. Multics mismo, en 1985, recibió la certificación B2. Hay actualmente varios proyectos de construir emuladores de Multics [4]. Emuladores, sí, no adecuaciones, pues Multics requiere características de hardware no presentes en las computadoras actuales.

Resulta ya obvio que la influencia de Multics es de grandísima importancia e innegable. Mencionaremos brevemente sus principales características de seguridad. Para quien busque información más a detalle respecto a Multics, recomiendo fuertemente el sitio [multicians.org](http://multicians.org) [5, 6, 7, 8].

## 2.2. Un sistema de anillos

La seguridad en Multics parte de un diseño conceptualmente conformado de ocho *anillos concéntricos* de privilegio, implementados en hardware (que, por software, podían dar la ilusión de ser en realidad 64 anillos). Entre más cerca del centro (0) está un anillo, mayores privilegios tiene. Los procesos que requieren privilegios de sistema por regla general corren en el anillo 2 o inferiores, las bibliotecas compartidas en anillo 3, y los programas de usuario en anillos superiores.

Buena parte de esto está implementado desde hardware, lo cual hace que explotar la seguridad del sistema sea mucho más difícil que en sistemas como los actuales, en los que prácticamente todos los mecanismos de seguridad —a excepción de los más básicos para sistemas multitarea, como la separación de segmentos de memoria— están implementados en el sistema operativo.

## 2.3. ACLs completos

Con el nombre ACL nos referimos genéricamente a las listas de control de acceso — La lista de quién tiene derecho de hacer qué con cada objeto en el sistema. Probablemente el primer sistema en incluir verdaderas listas de control de acceso fue Multics.

Un ACL en Multics puede verse así:

```
rew VanVleck.SysAdmin.a r Backup.SysDaemon.z rw *.SysAdmin.*
```

Este ACL tiene tres entidades, cada una de ellas especificando permisos, el juego de usuarios al que se aplican estos permisos<sup>1</sup>, y las instancias de estos usuarios.

---

<sup>1</sup>Además de los permisos para archivos (**rew**, lectura, ejecución y escritura) están definidos los permisos para directorio (**sma**, status, modificación, agregar). Al reimplementar la idea en Unix, se estandarizó en **rwX** (lectura, escritura y ejecución) para todo objeto, considerando al directorio como un tipo especial de archivo.

Los ACLs se construyen del elemento más específico hacia el más general, y la primer ocurrencia encontrada es la que determina acceso, con una política de denegación por default. En este caso, indica que el usuario VanVleck en el rol o instancia SysAdmin tienen derecho de lectura, escritura y ejecución (**rew**) cuando fueron autenticados en una sesión interactiva (**a**), el usuario Backup en la instancia SysDaemon tienen lectura cuando son demonios (**z**), y cualquier usuario en el rol de SysAdmin con cualquier forma de autenticación tiene lectura y escritura.

Los ACLs de Multics son muy eficientes comparados con muchos desarrollados posteriormente, incluso los de Posix y los de Windows NT, por la simplicidad del esquema y la ausencia de líneas de negación.

El que Multics haya logrado una certificación B2 es, en buena parte, consecuencia de su diseño de ACLs.

### 3. Unix clásico

Como mencionamos en 2.1, Unix puede considerarse hijo directo de Multics. Analicemos ahora pues el esquema de seguridad existente en los sistemas Unix y derivados.

#### 3.1. Espacios e interfaces

El sistema de anillos de Unics fue simplificado reduciéndolo a únicamente dos modos: El espacio de kernel y el espacio de usuario.

##### 3.1.1. Espacio de kernel

El código que se ejecuta en espacio de kernel no tiene restricción alguna en el sistema: Tiene acceso directo a todo el hardware, puede realizar cualquier operación sin acudir a las capas de abstracción que el kernel ofrece (y de hecho, es justamente su función — Por ejemplo, los módulos que permiten al kernel entender diferentes sistemas de archivos y ofrecer a los programas usuario un modelo abstracto uniforme se ejecutan en espacio de kernel). El código que se ejecuta en espacio de kernel, además, tiene acceso a toda la memoria de la computadora, es capaz de manipular la tabla de procesos e inclusive de modificar datos de los diferentes procesos. En la mayoría de los sistemas Unix, hay únicamente un proceso corriendo en espacio de kernel — Precisamente, el kernel. Varias implementaciones de Unix<sup>2</sup>

, tienen una implementación de *microkernel*, lo que significa que el kernel incluye únicamente una fracción del código privilegiado, y las demás facilidades de bajo nivel (por ejemplo, manejo de memoria, manejo de sistemas de archivos, manejo de red) son implementadas por procesos privilegiados adicionales. Esto nos da una mucho mayor separación funcional, lo que lleva a una mayor limpieza en el código, e incluso a veces nos permite substituir el código en ejecución

---

<sup>2</sup>Como NeXTstep/OpenStep, Darwin, Minix, y varios más

para funciones fundamentales del sistema, pero al mismo tiempo lleva a una implementación más compleja, que tiende a ser más lenta.

### 3.1.2. Espacio de usuario

Todos los programas a excepción del kernel y —en su caso— estos programas privilegiados adicionales trabajan en espacio de usuario. Esto significa que tienen ciertas restricciones, entre ellas:

- No pueden utilizar directamente ningún dispositivo
- No pueden leer espacios de memoria que pertenezcan a otro proceso y no estén explícitamente marcados como memoria compartida
- No pueden tocar ninguna de las tablas internas del kernel (de procesos, de asignación de memoria, etc.)

Para llevar a cabo cualquiera de estas acciones tienen que solicitarlo al kernel.

### 3.1.3. Llamadas al sistema

Cuando un programa no privilegiado requiere efectuar una operación para la cual no tiene privilegios, hace una llamada al sistema. Esta es básicamente como una llamada a cualquier función proporcionada por la biblioteca estándar de C, con la única particularidad que el encargado de atenderla no será el mismo proceso, sino que el kernel.

Cuando estamos depurando un programa, nos puede ser de gran utilidad emplear el comando `strace` en Linux [9], `ktrace` y `kdump` en los sistemas BSD, y `truss` en Solaris y en otros Unixes. Este comando nos muestra cada llamada al sistema efectuada por el programa en cuestión. Como puede observarse tras correr este comando, hay una gran cantidad de llamadas al sistema, las cuales pueden ser clasificadas en diferentes categorías (archivos, procesos, red, señales, IPC).

## 3.2. Permisos de uso

Ahora bien, ¿cómo puede determinar el kernel si un proceso determinado tiene derecho de llevar a cabo una determinada llamada al sistema? Para la mayor parte de ellos

En Unix tenemos definido un sistema simple y efectivo de permisos para cada objeto existente en nuestro árbol de directorio. Al solicitar una operación sobre de él, el kernel verifica si el usuario bajo el cual está corriendo dicho proceso, o alguno de los grupos a los que pertenece, tiene derecho de llevar a cabo la operación indicada sobre el objeto. Para esto, cada objeto tiene definido un usuario dueño y un grupo, y permisos de lectura, escritura y ejecución. Este simple esquema típicamente lo vemos representado como los tres juegos de `rxw` — Permiso de lectura, escritura y ejecución para usuario, grupo y resto del mundo.

Para comunicación simple entre procesos, y para que el kernel notifique a los procesos de determinados eventos, contamos con diferentes señales predefinidas (ver [10]), algunas de ellas *atrapables* (esto es, que el proceso puede tener una rutina diseñada para atender a esta señal, y en el caso de no existir dicha rutina, el sistema operativo utilizará un comportamiento predefinido dependiendo de la señal), algunas de ellas no (el proceso no puede decidir qué hacer con ella - el sistema operativo es quien lo decide y lo impone). Las reglas de comunicación por señales mandan que un proceso sólo puede mandar un proceso a otro si el proceso que envía la señal corre como root, o si ambos procesos tienen el mismo UID real o efectivo (ver [11]).

Respecto a la comunicación en red, la restricción es básicamente que ningún usuario que no sea root puede abrir un socket TCP o UDP por puertos menores al 1024, y que sólo root puede enviar paquetes *crudos* (que no fueron procesados de la manera estándar por la pila).

### 3.2.1. Uso de dispositivos

Una computadora tiene una gran cantidad de dispositivos de muy diferentes tipos. En Unix se hace una muy práctica abstracción, al afirmar que *todo es un archivo*. Todos los dispositivos tienen una entrada en el árbol de directorio del sistema, que por medio de un inodo especial (de caracteres o de bloques) apunta al manejador indicado en el kernel.

Los dispositivos por bloques (por ejemplo, los discos) nos permiten acceso aleatorio, transfiriendo bloques de longitud fija (típicamente 512 bytes). Los dispositivos por caracteres (por ejemplo, un puerto serial) están más bien orientados a un flujo de datos, permitiéndonos transferencias de caracteres individuales, pero con forma de acceso secuencial, sin proporcionarnos la posibilidad de retroceder o adelantar nuestra posición.

Los inodos que describen a un dispositivo lo hacen por medio de dos números, el mayor y el menor. Con ellos indican al kernel qué manejador utilizar para este dispositivo y con qué parámetros. Estos inodos llevan el mismo sistema de permisos que cualquier otro archivo del sistema.

## 3.3. El superusuario

En Unix todo está regulado por los permisos que tiene cada usuario. Hay un usuario especial en cada sistema: *root*, o el superusuario. Para el superusuario, los permisos de cualquier archivo siempre le permitirán lectura, escritura y ejecución — Esto significa que el superusuario puede hacer cualquier cosa en el sistema, nada le será negado.

## 3.4. La problemática con el esquema tradicional de Unix

Si bien para muchas aplicaciones el esquema de Unix es adecuado, dista mucho de ser perfecto. El tener un usuario capaz de *absolutamente* todo, el tener a un administrador con capacidad de leer, eliminar o modificar cualquier

dato en el sistema hacen que un sistema Unix no sea suficiente para ciertas aplicaciones donde hacen falta garantías de confidencialidad e integridad — En muchos casos, aplicaciones gubernamentales o militares<sup>3</sup>.

Yéndonos un poco hacia la terminología técnica, el problema es que el sistema de permisos en Unix está basado en DAC (*control de acceso discrecional*), del cual el administrador está exento, mientras que muchas aplicaciones del mundo real nos demandan MAC (*control de acceso mandatorio u obligatorio*, 4.2.5).

En muchos casos es también necesaria la separación de roles de administración — Tener un administrador diferente para los diferentes servicios, manteniendo la auditabilidad de cambios<sup>4</sup>.

Estas limitaciones, ciertamente, son comprensibles si recordamos que Unix no es sino una simplificación de Multics. Sin embargo, dada la popularidad que con el paso de los años adquirió Unix, fue haciéndose necesario encontrar cómo paliar estas carencias.

## 4. Sistemas confiables (*Trusted-\**)

En 1987, el National Computer Security Center de los Estados Unidos comisionó al grupo Trusted Unix Working Group (Trustix) para emitir una serie de recomendaciones que elevaran la seguridad en los sistemas Unix ([12]). Varios sistemas operativos han implementado estas recomendaciones (junto con varias más, como niveles de confidencialidad y roles de usuarios), aunque siempre tienen que mantenerse como una rama del sistema especializada y aparte, dadas las diferencias que estas recomendaciones implican para usuarios, administradores y desarrolladores — Los sistemas que implementan esto dejan de ser estrictamente Unixes para convertirse en sistemas basados en Unix. Los más conocidos en esta categoría son Trusted Solaris y Trusted DG/UX.

### 4.1. Trusted Solaris

Si bien el foco de nuestra atención son los sistemas operativos libres, vale la pena hacer un análisis breve de lo que provee Trusted Solaris [13], probablemente el sistema confiable más difundido y con mayor historia.

Trusted Solaris implementa muchas características enfocadas a permitir un control granular de permisos y a proveer un registro completo de eventos [14]. La primer gran diferencia que brinca a la vista al utilizar TrustedBSD es que ya no existe una cuenta de superusuario (`root`) [15]. La administración se realiza a través de diferentes usuarios con roles administrativos, y hay mecanismos

---

<sup>3</sup>Cabe mencionar que con la aparición de sistemas de criptografía fuerte, como GnuPG, cada usuario puede tener garantía de la confidencialidad e integridad de su información. El uso de sistemas de criptografía, sin embargo, requiere de intervención manual, y no puede ser considerado parte integral del sistema operativo.

<sup>4</sup>Esto, nuevamente, es implementable a través de programas adicionales que han sido desarrollados a lo largo de los años de existencia de Unix, pero tomó en algunos casos varias décadas encontrar los mecanismos adecuados.

de registro de sucesos que hacen tan rastreables las actividades de un administrador como las de cualquier otro usuario. Además, a diferencia de los Unixes estándar, para entrar con un rol administrativo al sistema es indispensable registrarse primero como usuario, y posteriormente desde su cuenta asumir un rol administrativo. Este proceso, además de corregir malas prácticas de administración, permite rastrear a un usuario cualquier mal uso de las funciones de administración.

Los objetos del sistema de archivos, aparte de tener información y mecanismos para llevar a cabo MAC, tienen *etiquetas* asociadas describiendo su nivel de sensibilidad — Un archivo puede estar marcado como *confidencial*, *público*, etc. Estas, además de ser informativas y ser desplegadas notoriamente en cada ventana en que se trabaje, permiten definir relaciones y colaborativas jerárquicas entre usuarios.

Pese a las grandes diferencias que estos puntos suponen, el entorno general sigue teniendo un fuerte *sabor* a Unix. Según la documentación de Sun, cualquier administrador familiarizado con Solaris podrá administrar Trusted Solaris sin mucho problema.

También para los usuarios hay también fuertes diferencias en la operación del sistema. En todo momento, el usuario ve en pantalla en el entorno CDE información respecto al nivel de sensibilidad de la información que está trabajando. Para mayor información acerca del uso diario de Trusted Solaris, sugiero referirse a [16].

## 4.2. TrustedBSD

El proyecto TrustedBSD [17] implementó buena parte de las recomendaciones de Trustix, así como varias extensiones adicionales [18], de manera mucho menos intrusiva sobre un sistema FreeBSD. La recientemente liberada versión 5.0 de FreeBSD incorpora ya buena parte del código desarrollado por éste proyecto [19], y, aunque en menor grado, también OpenBSD y Darwin han importado subsistemas importantes.

Los principales subsistemas desarrollados por TrustedBSD son:

### 4.2.1. Listas de control de acceso (ACLs)

Como mencionamos en 2.3, las listas de control de acceso nos permiten un control mucho más granular de qué usuarios o grupos gozan de qué privilegios de acceso a cada objeto en el sistema.

### 4.2.2. Eventos auditables

Para que un sistema sea considerado confiable, es fundamental tener la posibilidad de analizar las bitácoras y determinar inequívocamente cuál fue la causa de los eventos registrados.

### 4.2.3. Atributos extendidos

El manejo de atributos extendidos permiten asociar a cada archivo la *metainformación* que el usuario o el administrador juzguen importante para las diferentes extensiones de módulos de seguridad que existan (u otros fines cualquiera). Por ejemplo, la información de ACLs y de MACs es guardada ya de esta manera, y la información de las capacidades también lo será una vez que su desarrollo llegue a un nivel adecuado de estabilidad.

### 4.2.4. Capacidades granulares

TrustedBSD implementa un subconjunto de las capacidades que veremos en la sección 5.

### 4.2.5. Control de acceso mandatorio (MAC)

Es fundamental para considerar a un sistema seguro el no tener usuarios todopoderosos. A diferencia de los sistemas Unix estándar, que nos proporcionan control de acceso discrecional (DAC), esto permitirá restringir a las operaciones efectuadas aún por superusuarios.

Además de esto, es importante poder controlar el acceso que tienen procesos u otros objetos a determinados recursos del sistema — por ejemplo, uso de sockets de red, nodos de `sysctl`<sup>5</sup>, objetos del sistema de archivos, etc.

## 5. Capacidades POSIX.1e

Buscando estandarizar las diferentes respuestas a las limitaciones en materia de seguridad de las diferentes implementaciones de Unix, el grupo POSIX inició a mediados de los 80s a definir el estándar POSIX.1e. Otros nombres por los cuales este estándar es conocido son IEEE 1003.1e, P1003.1e y POSIX6.

### 5.1. ¿Qué es el estándar POSIX.1e?

El trabajo sobre este estándar parte de que es muy importante tener definido un consenso respecto a las implementaciones de seguridad entre los diferentes sistemas operativos, puesto que sólo así los desarrolladores se verán motivados a utilizarlas, aprovechando sus ventajas sin perder portabilidad. POSIX.1e busca corregir varias situaciones no deseables en los sistemas Unix tradicionales, notablemente el agregar ACLs y el poder asignar privilegios o limitaciones específicas no sólo a la relación de objetos en el sistema de archivos contra usuarios, sino que a cada proceso de manera independiente, limitando la cantidad de acciones que requieren asumir la identidad de root o de algún otro usuario, y limitando el daño que puede causar un atacante una vez que comprometió a un proceso [20].

---

<sup>5</sup>La interfaz estándar en Unix a las banderas del kernel en ejecución. En Linux, el pseudo-sistema de archivos `/proc` es en buena medida una implementación de llamadas a `sysctl`.

Hay otro estándar relacionado con POSIX.1e — POSIX.2c. Mientras que el primero busca definir las interfaces de seguridad para sistemas abiertos para ACLs, separación de privilegios, control de acceso mandatorio y mecanismos de etiquetas de metainformación, POSIX.2c busca definir utilidades de seguridad para implementar lo definido por POSIX.1e.

#### 5.1.1. Su estado actual

Tristemente, en 1999 POSIX decidió retirar la propuesta de este estándar, por lo cual POSIX.1e nunca será un estándar aprobado. Las razones para retirar el trabajo sobre este estándar fueron principalmente [21]:

**Falta de consenso por prácticas divergentes** Durante la etapa de desarrollo de este estándar, diferentes grupos de desarrollo implementaron las características que estaban a discusión. Cada grupo, claro, lo hizo de manera independiente y descoordinada, lo cual llevó a interfaces completamente diferentes, con grupos de usuarios con mayor interés en que no cambie lo ya establecido que en estandarizarse.

**Falta de implementaciones** Algunos elementos en el estándar sufrieron de lo opuesto: Nunca fueron implementados, y de la manera en que estaban siendo descritos, simplemente no eran implementables — El estándar cubría una interfaz ideal teórica, alejada por completo de cualquier desarrollo similar, alejada de las necesidades reales.

**Cambios en las necesidades** El mundo del cómputo avanza a muy gran velocidad. Cuando el trabajo en este estándar inició, la guía principal para juzgar un sistema como seguro eran los criterios de TCSEC (Trusted Computer Security Evaluation Criteria)

. El mundo ha ido evolucionando rápidamente, y estos criterios simplemente ya no son válidos hoy en día.

## 5.2. Implementaciones existentes de las capacidades POSIX.1e

Dentro del mundo del software propietario, podemos hablar básicamente de la implementación de los ACLs en Solaris y de los mecanismos de seguridad presentes en Irix 6.5.

En el mundo del software libre, ha habido bastante más avance hacia la implementación de las porciones más relevantes de este estándar.

El proyecto FreeBSD reporta avance en la implementación de las extensiones de auditoría, MAC, ACLs y atributos extendidos desde marzo de 1999 y hasta abril del 2000 [22]. Para estructurar mejor el foco de trabajo y desarrollo del sistema y facilitar compartir el código con los otros proyectos BSD, a partir de esa fecha el desarrollo en esta materia fue delegado al grupo de trabajo del proyecto TrustedBSD [17].

### 5.2.1. Las capacidades POSIX en Linux

En Linux ha habido también un muy interesante trabajo para la implementación de las capacidades granulares POSIX [23]. La mayor parte de este esfuerzo se ha centrado en las capacidades POSIX. Esta es una implementación de granularización de privilegios, nuevamente haciendo menos necesario el uso de la cuenta de superusuario. Para este modelo, cada proceso tiene tres juegos de banderas, o *bitmaps*: Capacidades heredables (*inheritable, I*), permitidas (*permitted, P*) y efectivas (*effective, E*). Cuando un proceso intenta llevar a cabo una operación que requiere privilegios, el sistema operativo revisa en el bitmap E, en vez de verificar, como tradicionalmente se hace, que el UID efectivo del proceso sea 0.

Un proceso puede tener capacidades declaradas como permitidas sin que estas sean efectivas. Esto es porque los procesos pueden desactivar temporalmente alguna de sus capacidades, para disminuir los riesgos en caso de ataque. Este proceso puede solicitar nuevamente al kernel que le asigne las capacidades inactivas únicamente si estas figuran en el bitmap P. El bitmap I son las capacidades que heredará un nuevo proceso que éste invoca a través de la llamada `exec`<sup>6</sup>.

En Linux, no sólo los procesos pueden tener definidas capacidades. También los archivos — aunque para evitar confusiones, a estos tres mismos bitmaps se les llama permitidos (*allowed, A*), forzados (*forced, F*) y efectivo (*effective, E*). El bitmap A indica qué capacidades puede heredar del bitmap I del proceso que lo invoque un ejecutable a través de un `exec`. El bitmap F indica las capacidades que los procesos creados a partir de este ejecutable *siempre* recibirán, independientemente de lo que indique el proceso que lo llame — un tanto al estilo del SUID en Unix. Por último, el bitmap E indica cuáles bits del bitmap del proceso nuevo deben ser heredados a procesos hijos suyos. Todos los bits de este último bitmap serán sólo 1 cuando el programa no está diseñado pensando en capacidades, o sólo 0 cuando sí implemente control interno de capacidades, y por ello puede ser implementado utilizando un solo bit en el sistema de archivos. El bitmap E que será asignado a un nuevo proceso al iniciar su ejecución es:

$$P = F | (A \& I)$$

Lo cual nos indica que hay que ser *muy* cuidadosos al activar bits en el bitmap F — Este, repito, es el equivalente granular del SUID de Unix, y hay que tratarlo con el mismo cuidado..

Hay 29 capacidades definidas en el kernel de Linux, a la versión 2.4.20. Para mayor detalle, sugiero consultar [24]. Estas son:

- CAP\_CHOWN
- CAP\_DAC\_OVERRIDE
- CAP\_DAC\_READ\_SEARCH

---

<sup>6</sup>No así con las llamadas `fork` o `clone` — En estas, el proceso hijo es una copia exacta del proceso padre.

- CAP\_FOWNER
- CAP\_FSETID
- CAP\_FS\_MASK
- CAP\_KILL
- CAP\_SETGID
- CAP\_SETUID
- CAP\_SETPCAP
- CAP\_LINUX\_IMMUTABLE
- CAP\_NET\_BIND\_SERVICE
- CAP\_NET\_BROADCAST
- CAP\_NET\_ADMIN
- CAP\_NET\_RAW
- CAP\_IPC\_LOCK
- CAP\_IPC\_OWNER
- CAP\_SYS\_MODULE
- CAP\_SYS\_RAWIO
- CAP\_SYS\_CHROOT
- CAP\_SYS\_PTRACE
- CAP\_SYS\_PACCT
- CAP\_SYS\_ADMIN
- CAP\_SYS\_BOOT
- CAP\_SYS\_NICE
- CAP\_SYS\_RESOURCE
- CAP\_SYS\_TIME
- CAP\_SYS\_TTY\_CONFIG
- CAP\_MKNOD
- CAP\_LEASES

A través de la capacidad `CAP_SETPCAP`, un proceso puede alterar (para agregar o para retirar capacidades) los bitmaps de otros procesos en ejecución en el sistema. Además de esto, utilizando el programa `execcap` podemos ejecutar un programa especificándole un juego de capacidades:

```
execcap 'cap_sys_admin=eip' update
```

ejecutará el programa `update` con `CAP_SYS_ADMIN` en sus tres bitmaps.

## 6. Systrace

Niels Provos, del equipo de desarrollo de OpenBSD, desarrolló y presentó en el 2002 una interesante propuesta [25]: Un mecanismo, parcialmente implementado en el kernel y parcialmente en espacio de usuario, que permite dejar de depender en privilegios SUID/SGID y, en general, en los roles de un superusuario, y a la vez permite restringir el comportamiento de un programa para evitar que, de ser éste atacado, lleve a cabo operaciones no deseables, limitando las llamadas al sistema que puede ejecutar (e inclusive permitiendo especificar diferentes comportamientos para llamadas con diferentes argumentos). Las tres metas principales de diseño de Systrace son controlar la escalación de privilegios a través de políticas, facilitar la detección de intrusos a nivel host, y aumentar las capacidades de monitoreo y auditoría de un sistema sin salir del esquema de un Unix genérico. Este mecanismo, llamado Systrace, permite confinamiento granular de procesos, detección de intrusos, auditoría y elevación de privilegios, y facilita el tedioso proceso de generación de políticas, al contar con un generador de políticas interactivo. Además de todo esto, systrace permite que cada usuario genere juegos de políticas para cada uno de los binarios del sistema sin intervención de root (y, claro, sin exceder sus propios privilegios). Además de todo esto, systrace es portable a otros sistemas operativos, y (a diferencia de otros sistemas similares) representa muy poca carga adicional al rendimiento del sistema — Sin embargo, esto es aún un punto en el que está trabajando el equipo de desarrollo, pues sí hay un impacto sensible con ciertas llamadas al sistema relacionadas con el sistema de archivos.

La decisión de implementar a systrace de manera híbrida, en espacio de kernel y en espacio de usuario, fue tomada pensando en eficiencia (el implementar en espacio de usuario hubiera hecho que para toda llamada al sistema hubiera una gran cantidad de cambios de contexto, haciendo sensiblemente más lenta la operación general), seguridad (al tener al monitor de privilegios como un proceso de usuario es mucho más fácil detenerlo, engañarlo o aprovechar la ventana de oportunidad entre la creación de un proceso y la asignación de sus privilegios — Implementando en kernel, un proceso al ser creado ya tiene la política asignada, heredada de su proceso padre) y portabilidad (si todo fuera implementado en espacio de kernel, adecuar systrace para otros sistemas operativos hubiera básicamente consistido en reescribirlo).

Claro está, systrace fue diseñado con la consigna de que fuera prácticamente inviolable. Algunas ideas ingeniosas fueron implementadas — Por ejemplo, cuan-

do un proceso hace una llamada al sistema, el kernel copia los argumentos de esta llamada, los verifica y ejecuta la llamada *desde la copia*, no desde el espacio original. Esto evita que los argumentos puedan ser modificados (por ejemplo, con un proceso que comparta memoria con el primero) después de solicitar autorización pero antes de ser realmente ejecutados. Esto permitió además simplificar varias acciones tradicionalmente complejas en los esquemas de ACLs y políticas implementados en otros sistemas — Por ejemplo, el kernel reescribe los argumentos de las llamadas que involucran archivos, de modo que cuando verifica los privilegios sobre una liga simbólica, antes de continuar con la operación resuelve la liga, y trabaja ya directamente con el objeto real en el sistema de archivos. Otra aplicación de la reescritura de llamadas a función es para tener un mejor control sobre las aplicaciones que requieran tener acceso a red.

La generación de políticas puede llevarse a cabo (ejecutando binarios que presuponemos confiables) ya sea de forma automática o manual. Si lo hacemos de forma automática, ejecutamos nuestra aplicación de todas las maneras que normalmente esperamos que sea llamada, registrando las llamadas al sistema que ésta realiza y etiquetándolas como válidas, y una vez generada y activada la política, detectará cualquier uso fuera del patrón definido. Si lo hacemos de forma manual, cada llamada al sistema que el proceso realice será suspendida hasta que el usuario (a través de una aplicación independiente) marque a dicha llamada como válida o inválida, opcionalmente especificando ciertos parámetros. Además de esto, las políticas son definidas por medio de un lenguaje muy claro y simple, por lo que ajustarlas a mano es una labor bastante sencilla.

Hay un proyecto muy interesante relacionado con Systrace llamado Hairy Eyeball [27], que ha generado políticas estándar para ya 160 binarios, y probablemente el resultado del trabajo de este proyecto sea pronto integrado al sistema base de OpenBSD. Hoy en día systrace es ya parte del árbol estable de OpenBSD, y existen paquetes no oficiales para brindarle soporte en Linux, inclusive integrado ya a la rama inestable de Debian como parche no oficial al núcleo y como un par de paquetes con los programas en espacio de usuario [28].

## Referencias

- [1] Corbató, F. J., y Vyssotsky, V. A., Introduction and overview of the Multics system <http://www.multicians.org/fjcc1.html>, AFIPS Conf Proc 27, 185-196, 1965.
- [2] Multics PL/1 <http://www.multicians.org/pl1.html>
- [3] Department of Defense Trusted System Evaluation Criteria <http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html>, Dec. 1985
- [4] SIMTICS Isn't MULTICS <http://simtics.sourceforge.net/design.html>
- [5] Multicians <http://www.multicians.org>

- [6] FAQ e información general de Multics <http://www.multicians.org/general.html>
- [7] Características (software y hardware) de Multics <http://www.multicians.org/features.html>
- [8] Historia de Multics <http://www.multicians.org/history.html>
- [9] Página de manual de strace(1) (en sistemas Linux)
- [10] Página de manual de signal(7)
- [11] Página de manual de kill(2)
- [12] TRUSTIX: Access Control Lists <http://www.cotse.com/texts/rainbow/silver.htm>
- [13] Trusted Solaris[tm] 8 Operating Environment Features and Benefits <http://www.sun.com/software/solaris/trustedsolaris/features.html>
- [14] Trusted Solaris[tm] Operating Environment [www.sun.com/software/solaris/trustedsolaris/ds-ts8/index.html](http://www.sun.com/software/solaris/trustedsolaris/ds-ts8/index.html)
- [15] FAQ: Why can't I log in as root after installing Trusted Solaris? [www.sun.com/software/solaris/trustedsolaris/ts\\_tech\\_faqs/faqs/root\\_log.html](http://www.sun.com/software/solaris/trustedsolaris/ts_tech_faqs/faqs/root_log.html)
- [16] Trusted Solaris User's Guide: Introduction to Trusted Solaris [docs.sun.com/db/doc/805-8115-10/6j7klujb8](http://docs.sun.com/db/doc/805-8115-10/6j7klujb8)
- [17] The TrustedBSD Project <http://www.trustedbsd.org/>
- [18] TrustedBSD - Components <http://www.trustedbsd.org/components>
- [19] The TrustedBSD Project - Daemonnews <http://ezine.daemonnews.org/200110/trustedbsd.html>
- [20] Summary about Posix.1e <http://wt.xpilot.org/publications/posix.1e/index.html>
- [21] Fwd: Security extensions to Posix what would have been Posix.1e/2c) <http://lists.nas.nasa.gov/archives/ext/linux-security-audit/1999/06/msg00057.html>
- [22] POSIX.1e implementation for FreeBSD <http://www.watson.org/fbsd-hardening/posix1e/index.html>
- [23] Linux Capabilities FAQ 0.2 <ftp://ftp.guardian.no/pub/free/linux/capabilities/capfaq.txt>
- [24] Archivo del código fuente del kernel de Linux `include/linux/capability.h`, Andrew G. Morgan, Alexander Kjeldaas

- [25] Improving Host Security with System Call Policies <http://www.citi.umich.edu/techreports/reports/citi-tr-02-3.pdf>, Niels Provos
- [26] Systrace - Interactive Policy Generation for System Calls <http://www.citi.umich.edu/u/provos/systrace/index.html>, Niels Provos
- [27] Proyecto Hairy Eyeball <http://blafase1.org/~floh/he/index.html>
- [28] Paquete kernel-patch-systrace en Debian <http://packages.debian.org/unstable/devel/kernel-patch-systrace.html> Paquete de Systrace en Debian <http://packages.debian.org/unstable/admin/systrace.html>, Paquete de xsystrace en Debian <http://packages.debian.org/unstable/x11/xsystrace.html>
- [29] Systrace <http://www.openbsd.org.mx/~alex/gulev2002/Systrace/index.html>, Alejandro Juárez Robles, Congreso GULEV 2002