Sistemas Operativos Avanzados (MUII)
Ejercicio sobre seguridad (28-abril-2021)
Control de acceso en Linux

Planteamiento del trabajo

Uno de los objetivos principales de un sistema operativo es la seguridad. Un sistema operativo debe proporcionar un entorno protegido para la ejecución de aplicaciones. El sistema operativo es un eslabón más en la cadena de componentes que trabajan de forma integrada para proporcionar el nivel de seguridad requerido en un sistema. Dentro del sistema operativo, este requisito abarca diversos aspectos. Este trabajo se centra en los mecanismos de control de acceso del sistema operativo, es decir, aquellos que, una vez que un usuario ha sido autenticado, controlan qué acciones se le permiten realizar a ese usuario sobre los distintos recursos del sistema. Concretamente, se plantea revisar algunos de los mecanismos de este tipo disponibles en Linux: desde los tradicionales de la familia UNIX, que seguramente ya conocerá el lector, a los específicos de este sistema operativo. Téngase en cuenta que no se pretende realizar un trabajo de administración de sistemas UNIX/Linux, sino hacer énfasis en los conceptos.

El control de acceso en UNIX

En esta sección se repasa brevemente, puesto que ya será conocido por la mayoría de lectores, el esquema de control de acceso original de UNIX, que se puede catalogar como un mecanismo de tipo ACL simplificado, que le hace más eficiente y sencillo, pero menos potente, que un modelo de ACL convencional.

En el control de acceso intervienen sujetos, operaciones y recursos. Revisemos cada uno de ellos.

Empecemos con los *sujetos*. Cuando se crea un nuevo usuario se le asigna un identificador de usuario (UID) y un identificador de grupo primario (GID), almacenando esa asociación en una línea del fichero de texto */etc/passwd*. Asimismo, se puede asociar al nuevo usuario con otros grupos secundarios (o suplementarios) guardando esa vinculación en el fichero */etc/group*.

Cada proceso tiene asociado un UID, un GID primario y puede tener GID secundarios. Cuando un usuario inicia una sesión, el sistema operativo crea un proceso para atender a ese usuario asignándole a ese proceso el UID (servicio *setuid*) y el GID primario (servicio *setgid*) obtenidos del fichero */etc/passwd* y los grupos secundarios (servicio *setgroups*) especificados en el fichero */etc/group*. Esas son las credenciales del proceso que serán heredadas por los procesos descendientes. Como se verá más adelante, UNIX distingue entre UID y GID reales y efectivos, pero por el momento ignoraremos esa distinción.

Sigamos con las *operaciones*. UNIX distingue tres: leer (*r*), escribir (*w*) y ejecutar (*x*). Cuando se aplican a un fichero, el significado es el que indica su nombre. Sin embargo, en el caso de usarse para un directorio, la interpretación no es tan directa: *r* indica que se puede leer el contenido del directorio, *w* especifica que se pueden crear nuevos ficheros en el directorio o borrar los existentes y *x* establece que se pueden traducir rutas (*lookup*) que pasen por ese directorio.

Concluyamos con los *objetos*. Cada fichero tiene asociado un UID y GID que identifican el dueño del fichero. Asimismo, tiene definidos 12 bits denominados *modo* que están vinculados con el control de acceso (nótese que en el inodo se empaquetan esos 12 bits en un campo de 16 bits con los 4 bits que definen el tipo del fichero). Los 9 de menor peso del modo corresponden a los permisos para realizar sobre el fichero las operaciones *r*, *w* y *x* por parte de distintos usuarios:

- Bit 8: permiso de lectura para el dueño, es decir, para un proceso cuyo UID (efectivo) sea igual al UID del fichero.
- Bit 7: permiso de escritura para el dueño.

- Bit 6: permiso de ejecución para el dueño.
- Bit 5: permiso de lectura para el grupo, es decir, para un proceso tal que su GID (efectivo) primario o alguno secundario coincide con el GID del fichero, no siendo el dueño.
- Bit 4: permiso de escritura para el grupo.
- Bit 3: permiso de ejecución para el grupo.
- Bit 2: permiso de lectura para el resto de los usuarios, es decir, para un proceso que no encaja en ninguna de las dos condiciones previas.
- Bit 1: permiso de escritura para el resto de los usuarios.
- Bit 0: permiso de ejecución para el resto de los usuarios.

Estos nueve bits se suelen especificar en octal para facilitar su interpretación. Así, por ejemplo, el valor 0751 para un determinado fichero indica que el dueño puede realizar cualquiera de las tres operaciones sobre el fichero, mientras que los usuarios del grupo pueden leerlo y ejecutarlo, y el resto de los usuarios solo pueden ejecutarlo. En los contextos donde sea posible, también se puede usar una notación simbólica con las iniciales en inglés de cada operación. En el ejemplo, los permisos se expresarían como *rwxr-x--x*.

Nótese que el usuario *root* (UID y GID iguales a 0) puede hacer cualquier operación sobre cualquier fichero, con independencia de los permisos que tenga.

Los tres bits de mayor peso del modo están también relacionados principalmente con el control de acceso.

- Bit 11: *SetUID*. Indica que, como se verá más adelante, cuando un proceso ejecute este fichero, el UID efectivo del proceso, que es el que se tiene en cuenta para el control de acceso, pasará a ser igual al del dueño del fichero.
- Bit 10: *SetGID*. Indica que, como se verá más adelante, cuando un proceso ejecute este mandato, el GID efectivo del proceso, que es el que se tiene en cuenta para el control de acceso, pasará a ser igual al del dueño del fichero. Cuando se activa para un directorio, establece que los ficheros que se creen en ese directorio tendrán como GID el del dueño del directorio. En breve, retomaremos este aspecto. Otro uso especial de este bit, no relacionado con el control de acceso, aparece cuando se activa en un fichero normal sin permiso de ejecución para indicar que los cerrojos que se establezcan sobre ese fichero (*fcntl*) sean de tipo obligatorio (*mandatory*). Observe la economía en la utilización de estos bits, lo que conlleva un uso eficiente pero confuso de los mismos.
- Bit 9: *Sticky bit*. El objetivo original de este bit era aplicarlo a un fichero ejecutable de uso frecuente para indicar al sistema que cuando terminara la ejecución del programa lo mantuviera en memoria, mejorando así el rendimiento del sistema. Con la memoria virtual este uso ha quedado obsoleto. Sin embargo, se le ha añadido una funcionalidad adicional relacionada con el control de acceso cuando se aplica a directorios. Para entender esta nueva funcionalidad hay que tener en cuenta que un usuario que tenga permiso de escritura en un directorio puede borrar los ficheros presentes en ese directorio, incluso aunque no sean suyos y no tenga ningún permiso para acceder a los mismos. Si un directorio tiene activo este bit, un usuario con permiso de escritura en el fichero solo podrá borrar aquellos ficheros que son suyos. Este bit se suele usar para directorios compartidos, como el */tmp*, que requieren este modo de operación.

Para completar la descripción del esquema de control de acceso de UNIX es necesario explicar cómo se determina la información de este tipo para un nuevo fichero:

- El UID y el GID del nuevo fichero corresponderán al UID y GID primario efectivos del proceso que crea el fichero, excepto si está activo el bit *SetGID* del directorio donde se va a crear el fichero, en cuyo caso el GID será el del directorio. Posteriormente, el usuario *root* puede cambiar la información de quién es el dueño de un fichero ya existente mediante *chown* (el servicio o el mandato que utiliza ese servicio). El dueño solo puede cambiar el GID, siempre que corresponda a uno de los GID asociados al proceso que

realiza el cambio, excepto si tiene la *capability CAP_CHOWN* (se estudiará más adelante), en cuyo caso también podrá cambiar el UID.

- El modo de un fichero lo determina el proceso que crea el fichero (por ejemplo, el programa *touch*, que crea un fichero si no existe previamente, especifica el valor 0666, dando permiso de lectura y escritura a todos los usuarios), pero filtrados por la máscara de permisos de creación de ficheros que tenga activa el proceso en ese momento (servicio y mandato *umask*, que especifica qué permisos de los especificados cuando se crea el fichero son eliminados). Así, si la máscara es 027, aunque el proceso que crea el fichero haya especificado un valor 0666, los permisos resultantes son 0640. Posteriormente, un proceso con un UID efectivo que corresponda al dueño del fichero puede cambiar cualquiera de los doce bits que corresponden al modo de un fichero mediante *chmod* (el servicio o el mandato que utiliza ese servicio). Nótese que, a diferencia de lo que ocurre con el UID y el GID asociado al fichero, no hay ninguna forma de forzar que todos los ficheros creados en un cierto directorio tengan unos determinados permisos iniciales (la máscara realiza en parte esa labor pero está asociada al proceso y no al directorio, con lo que puede cambiar a discreción del proceso). Retomaremos, y superaremos, este hándicap cuando estudiemos las ACL.

Completado el repaso al modelo de control de acceso original de UNIX, podemos plantear un primer ejercicio.

Ejercicio 1

Considere un conjunto de usuarios (*usuario1*, *usuario2, usuario3*…), cada uno con su UID y GID primario, que participan en un proyecto común. El primer usuario actúa como líder y tiene un directorio en su cuenta donde se almacenarán los ficheros compartidos del proyecto (directorio *~usuario1/proyecto*). El resto de los usuarios tiene acceso al directorio usando un enlace simbólico (en la cuenta de otro usuario: *ln -s ~usuario1/proyecto*). Se plantean los siguientes requisitos en lo que al control de acceso se refiere:

- Todos los usuarios del proyecto tienen permiso de lectura y escritura sobre todos los ficheros compartidos del proyecto y esto debe ocurrir automáticamente con todos los ficheros que se creen en ese directorio por parte de cualquier usuario del proyecto sin necesidad de cambiar sus características una vez creados.
- Un usuario puede borrar sus ficheros compartidos, pero no los de los otros usuarios, exceptuando el líder que puede borrar todos los ficheros.
- El resto de los usuarios del sistema no tendrán ningún tipo de acceso a los ficheros compartidos.

Describa qué operaciones, usando el esquema tradicional de UNIX, habrá que llevar a cabo para cumplir los requisitos planteados, especificando qué características desde el punto de vista del control de acceso tendrá el directorio compartido.

SetUID y *SetGID*

Para comprender la necesidad y el modo de operación de este mecanismo, vamos a plantear el ejemplo por antonomasia del uso del mismo en el entorno UNIX: el cambio de contraseña de una cuenta.

Para cambiar la contraseña de su cuenta, un usuario usa el mandato *passwd*, que necesita acceso de escritura al fichero */etc/shadow* donde está almacenada (realmente, en el UNIX original la contraseña se guardaba en el fichero */etc/passwd*, pero esta estrategia entrañaba problemas de seguridad ya que este fichero necesita poder ser leído por cualquier usuario, haciendo pública la contraseña cifrada lo que hacía posible un ataque de fuerza bruta para intentar descifrarla). Pero, por razones obvias, no se le puede otorgar permiso de escritura a todo el mundo en el fichero */etc/shadow*. Para resolver este tipo de problemas, se idearon los bits *SetUID* y *SetGID*, que están incluso patentados:

- Todo proceso tiene un UID real y uno efectivo, así como un GID primario real y uno efectivo. Son los identificadores efectivos los que se tienen en cuenta para todas las operaciones vinculadas con el control

de acceso. Normalmente, los identificadores reales y los efectivos coinciden, excepto cuando entran en juego los bits *SetUID* y *SetGID* asociados a un fichero ejecutable.

- Cuando un proceso ejecuta un programa con el bit *SetUID* activo, el *UID* efectivo cambia pasando a ser el UID asociado al fichero, manteniéndose el real. Evidentemente, este cambio de identidad modifica radicalmente la posición del proceso en lo que se refiere al control de acceso, especialmente si el dueño del fichero ejecutable es el usuario *root* (UID igual a 0).
- El mismo tratamiento conlleva el bit *SetGID*, pero afectando al GID primario efectivo del proceso.
- Además de los servicios *setuid* y *setgid* mencionados previamente, se dispone de los servicios *seteuid* y *setegid* que afectan, respectivamente, solo al UID y GID efectivo de un proceso.

Con esta técnica se resuelve el problema de la contraseña: no es necesario cambiar los permisos del fichero */etc/shadow*, sino que basta con activar el bit *SetUID* del mandato *passwd* cuyo dueño es el proceso *root*, de manera que cuando lo ejecutemos nos convertimos temporalmente en el usuario *root* pasando a tener un UID efectivo igual a 0.

Cuando se usa esta técnica para obtener temporalmente los poderes del usuario *root*, se está rompiendo el principio de mínimo privilegio, puesto que, aunque realmente solo necesitamos el permiso para realizar una labor concreta (en el caso planteado, modificar la línea del fichero */etc/shadow* asociada a mi cuenta), hemos adquirido plenos poderes. Cualquier vulnerabilidad en un programa del usuario *root* con el *SetUID/SetGID* activo puede permitir a un atacante adquirir el control total de un sistema.

Durante la ejecución de un programa con el *SetUID/SetGID* activo, el proceso puede retomar su identidad real (es decir, hacer que el identificador efectivo vuelva a ser igual al real), perdiendo los *superpoderes* otorgados, ya sea de forma temporal (un poco más adelante en la ejecución puede recuperarlos) o de manera definitiva. Esa última estrategia, que retomaremos en el siguiente ejercicio, es conveniente desde el punto de vista de la seguridad: una vez realizada la acción para la que se necesitaba esa promoción de permisos, vuelve a su identidad durante la ejecución del resto del programa, reduciendo la ventana de vulnerabilidad. Por lo que se refiere a la primera estrategia (perder los poderes adquiridos, pero pudiendo recuperarlos más tarde), requiere que el sistema operativo gestione un tercer identificador: el UID o GID salvado. Cuando un proceso ejecutando un programa con *SetUID* o *SetGID* hace que temporalmente el UID o GID efectivo vuelva a ser igual que el real, en el identificador salvado se guarda ese identificador efectivo sobrescrito. Una petición posterior del proceso para recuperar el identificador efectivo previo solo se satisfará si corresponde al almacenado en el identificador salvado.

Si reflexionamos sobre el rol de estos bits, podemos apreciar que en ocasiones el control de acceso no está solamente vinculado con operaciones elementales como leer, escribir o ejecutar, sino que puede estar asociado a labores de más alto nivel, como las realizadas por un programa que cambia una contraseña. El enfoque de poder establecer políticas de control de acceso asociadas a programas está detrás del modelo RBAC o de las *capabilities* de Linux, o incluso del mandato *sudo*.

"Casi todo es un fichero"

Una de las marcas distintivas del mundo UNIX es la filosofía de que "todo es un fichero", lo que proporciona una visión uniforme e integrada de todos los recursos del sistema. Sin embargo, esto no es de todo cierto (por eso, precisamente surgió el sistema operativo Plan 9: para cumplir estrictamente ese postulado) y tiene muchas excepciones, es decir, operaciones que no requieren la manipulación de ficheros para llevarse a cabo, como, por ejemplo, cambiar la hora de un equipo, reservar un puerto TCP o UDP, configurar la red o parar el sistema.

La cuestión es cómo implementar una política de control de acceso para todas estas operaciones que no están vinculadas con el mundo de los ficheros. UNIX plantea una estrategia radical contraria al principio del mínimo

privilegio: solo un proceso con UID efectivo igual a 0 puede llevarlas a cabo (en el caso de los puertos, eso solo sería necesario si se pretende reservar un puerto privilegiado).

Esta estrategia de todo o nada ha hecho posibles numerosos ataques a la seguridad en los sistemas UNIX. Así, por ejemplo, un servidor que necesite reservar un puerto privilegiado, pero ninguna otra acción privilegiada adicional, requiere ejecutar con un UID efectivo igual a 0 (ya sea porque lo ejecuta directamente el usuario *root* o porque tiene el *SetUID* activo), con lo que cualquier vulnerabilidad del programa puede permitir que un atacante tome el control total del sistema. El ataque a la seguridad denominado gusano de Morris sacó provecho del uso indebido de la función *gets* (extraído del manual de *gets*: "*Never use this function*") en el código del servidor *fingerd* para conseguir un ataque de tipo *buffer overflow* e invadir numerosas máquinas durante su ataque.

Ejercicio 2

En primer lugar, identifique un par de programas en su equipo, exceptuando los mandatos *passwd* y *sudo*, que usen *SetUID/SetGID* explicando por qué motivo es necesario el uso de esta técnica.

A continuación, vamos a usar el programa *servidor1* que reserva un puerto privilegiado. Consiga que un usuario normal pueda ejecutar satisfactoriamente este programa (puede probar si ha arrancado de forma correcta ejecutando *telnet localhost 666*). Como en el ejercicio previo, basta con explicarlo en lenguaje natural, aunque se valora positivamente si se especifican los mandatos concretos requeridos. Recuerde que cada vez que compila el programa se genera un nuevo fichero, por lo que tendrá que volver a aplicarle los mandatos requeridos para darle *superpoderes*.

Por último, el programa *servidor2* plantea la posibilidad de que, una vez que un programa ha realizado la labor que requería privilegios, los descarte (en el ejemplo, una vez realizado el *bind*, el programa ya no necesita de ningún privilegio adicional), ya sea de forma temporal, pudiendo recuperarlos posteriormente, o de manera definitiva, que sería la opción más conveniente si se quiere limitar la ventana de tiempo durante la cual el sistema es vulnerable. Explique qué servicio (*setuid* o *seteuid*) se debería usar en el punto indicado en el programa con la etiqueta *AQUÍ* si se pretende una "desescalada" temporal y cuál en caso de que se quiera una definitiva.

Delegación de la administración

En este apartado se incide nuevamente en las limitaciones de ese esquema de todo o nada que plantea UNIX para el control del uso de ciertos recursos, usando para ello un escenario habitual, al que podemos denominar como delegación de la administración.

Generalmente, en un equipo de administración de sistemas hay distintos roles que se ocupan de la gestión de diferentes tipos de recursos: administración de la red, de los sistemas de almacenamiento, de las bases de datos, del servicio web, del servicio LDAP, del servicio DNS…

Idealmente, sea por motivos de desconfianza o por satisfacer el principio de mínimo privilegio, el coordinador del equipo de administración querría asignar a cada administrador específico aquellos privilegios requeridos para llevar a cabo su labor, pero ninguno más (entre otras cosas, no le quiere proporcionar la contraseña del *root*). Como ya sabemos, eso no es posible en el mundo UNIX.

Una posible estrategia sería crear herramientas que encapsulen la funcionalidad requerida por cada tipo de administración específica y dejar accesibles esas herramientas, que son del *root* y tienen el bit *SetUID/SetGID* activo, solo para los usuarios pertinentes. Se trata de una solución laboriosa y que no satisface el principio de mínimo privilegio.

Una alternativa que se usa en numerosos sistemas de la familia UNIX es la herramienta *sudo* que facilita esta delegación de la administración. Antes de describir su modo de operación, convendría resaltar que se trata de una herramienta que no requiere soporte dentro del núcleo del sistema operativo.

El objetivo de esta herramienta es permitir que el administrador pueda configurar (fichero */etc/sudoers*) que un determinado usuario puede ejecutar en ciertas máquinas el mandato especificado usando la identidad de otro usuario. Así, por ejemplo, se puede convertir a un usuario en un administrador plenipotenciario de un sistema especificando que puede ejecutar cualquier mandato en cualquier máquina adoptando la identidad de cualquier usuario. Nótese que ese usuario omnipotente usaría su contraseña y no la del *root* a la hora de ejecutar mandatos con *sudo*. De hecho, gracias a este mecanismo, se puede prescindir de la existencia explícita de una cuenta de *root*.

Como uno puede suponer *a priori*, el mandato *sudo* pertenece al *root* y tiene activo el bit *SetUID*, lo que le permite tener todos los privilegios requeridos para llevar a cabo su labor, entre otros, realizar las llamadas a los servicios *setuid* y *setgid* necesarias para realizar el cambio de identidad requerido.

Esta herramienta cumple algunos de los requisitos necesarios para implantar un esquema de delegación de la administración, ofreciendo además una auditoría exhaustiva de las distintas operaciones realizadas bajo su cobertura. Sin embargo, sigue rompiendo el principio de mínimo privilegio: cuando un usuario ejecuta como *root* un determinado programa gracias al uso de *sudo*, la explotación de una vulnerabilidad puede permitir adquirir un control total del sistema.

Antes de abordar el tercer ejercicio, como recapitulación de este repaso del esquema original de control de acceso en UNIX, se pueden identificar dos limitaciones del mismo que serán abordadas en las siguientes secciones:

- El esquema de permisos de UNIX se puede considerar como una ACL de “baja calidad”. Para lograr establecer reglas de acceso sencillas (como, por ejemplo, que un cierto usuario puede escribir en un fichero mío), hay que realizar toda una serie de subterfugios (como el uso de grupos secundarios) que complican la gestión de la seguridad del sistema. Parece razonable incorporar un mecanismo de ACL pleno en el sistema operativo.
- Una estrategia de todo o nada rompe el principio de mínimo privilegio y facilita la explotación de las vulnerabilidades de los programas. Sería más adecuado disponer de un mecanismo que permitiera asignar a cada proceso justo los privilegios que requiere para realizar su labor.

Ejercicio 3

Explique cómo se configuraría *sudo* para delegar la administración de la red (supongamos, de forma simplificada, que para ello basta con permitir ejecutar el mandato *ip*) a un determinado usuario. Nótese que algunos de los usos del mandato *ip* no requieren privilegios y aquellos que sí los necesitan pueden causar una desconfiguración de la red. Para hacer pruebas, se sugiere usar el mandato *ip neighbour flush dev X*, donde *X* es el nombre de la interfaz de red de su equipo, que requiere privilegios, pero realiza una labor que no afecta al buen funcionamiento de la red (vaciar la caché de ARP).

Las listas de control de acceso de Linux

Como otros sistemas de la familia UNIX, Linux ha incorporado ACL en su esquema de control de acceso, usando los atributos extendidos como mecanismo para almacenarlos.

Para hacer compatible ACL con el modelo tradicional, los 9 bits de permisos se consideran como tres entradas de tipo ACL implícitas que se pueden gestionar de forma tradicional o utilizando los mandatos específicos para manipulación de ACL (*getfacl* y *setfacl*).

ls -l fichero
-rw-rw-r-- 1 usuario1 usuario1 0 jun 8 07:52 fichero
getfacl fichero
 # file: fichero
 # owner: usuario1
 # group: usuario1
user::rw-
group::rw-
other::r--
chmod g-w fichero
ls -l fichero
-rw-r--r-- 1 usuario1 usuario1 0 jun 8 07:52 fichero
getfacl fichero
 # file: fichero
 # owner: usuario1
 # group: usuario1
user::rw-
group::r--
other::r--
setfacl -m "o::-" fichero
ls -l fichero
-rw-r----- 1 usuario1 usuario1 0 jun 8 07:52 fichero
getfacl fichero
 # file: fichero
 # owner: usuario1
 # group: usuario1
user::rw-
group::r--
other::---

A partir de este punto, el dueño del fichero puede añadir nuevas entradas explícitas, que se almacenarán como atributos extendidos, para dar permisos tanto a usuarios como a grupos concretos (la notación especifica el nombre del usuario o del grupo entre los caracteres "::"). Evidentemente, esas nuevas entradas no serán mostradas por el mandato *ls*, aunque sí mostrará un carácter + para indicar que hay información adicional.

Para poder revocar de forma efectiva los permisos asignados a distintos usuarios y grupos concretos, se establece una entrada especial denominada máscara que filtra los permisos que se les ha otorgado a esos usuarios y grupos concretos. Así, si la máscara especifica solo el permiso de lectura, los permisos adicionales que se les haya concedido a determinados usuarios o grupos no serían efectivos.

Por último, el mecanismo de ACL de Linux resuelve una limitación del sistema de control de acceso de UNIX que se identificó anteriormente: la posibilidad de especificar qué permisos tendrán los ficheros que se crean en un determinado directorio. Las ACL por defecto que se pueden asignar a un directorio satisfacen este objetivo.

Ejercicio 4

Realice las siguientes actividades usando el escenario creado en el primer ejercicio y realice un pequeño informe sobre el mismo:

- Cree un fichero *F* en la cuenta de *usuario1* y asegúrese de que tiene los permisos 660 (*rw-rw----*).
- Pruebe a acceder desde *usuario2*.
- Permita acceso de lectura a todos los usuarios del proyecto.
- Compruebe mediante *getfattr* que se han creado atributos extendidos. Tenga en cuenta que, al ser atributos de sistema, por defecto no se muestran.
- Permita acceso de escritura a *usuario2*.
- Establezca una máscara de solo lectura y compruebe el acceso.
- Cree un directorio *D* de manera que todos los ficheros que se creen en ese directorio tengan permisos *rw-rw-rw-*.

Las *capabilities* de Linux

El modelo de todo o nada que sigue UNIX en el control de acceso a numerosas operaciones infringe el principio de mínimo privilegio, causando problemas de seguridad y dificultando la delegación de la administración.

Idealmente, cada proceso debería obtener estrictamente solo los privilegios requeridos para llevar a cabo su labor. Ese es el objetivo del mecanismo de las *capabilities* de Linux (recuerde del principio del documento que, desde un punto de vista teórico, se usa ese término para identificar a aquellos esquemas de control de acceso que asocian la información de control con los procesos en vez de con los recursos, como ocurre, en cambio, con las ACL).

En Linux se han definido, y se sigue haciéndolo, un conjunto de varias decenas de *capabilities*. Cada *capability* representa la capacidad de realizar un cierto tipo de operaciones, de manera que el sistema operativo solo permite realizar una operación de ese tipo si el proceso que la solicita posee la *capability* correspondiente o, como en el modelo tradicional, es un proceso del *root*. A continuación, se describen brevemente algunas de estas *capabilities* (*man capabilities*): *CAP_SYS_TIME* (cambiar la hora), *CAP_SYS_NICE* (aumentar la prioridad de procesos), *CAP_NET_BIND_SERVICE* (*bind* de un puerto privilegiado), *CAP_SYS_MODULE* (insertar módulos del núcleo), *CAP_MKNOD* (crear fichero de dispositivo), *CAP_FOWNER* (cambiar atributos de un fichero sin ser su dueño), *CAP_CHOWN* (cambiar de dueño a un fichero) y *CAP_DAC_OVERRIDE* (saltarse el control de acceso DAC).

Un proceso (o mejor dicho un *thread*) tiene asociado un conjunto de *capabilities*. Como veremos más adelante, la cosa es más compleja y hay cinco conjuntos: *permitted, effective, inheritable, bounding_set y ambient*. El trabajo se centrará en los conjuntos de *capabilities* efectivas (*effective*), que representan qué operaciones puede llevar a cabo un proceso, y permitidas (*permitted*), que son aquellas que tiene asignadas el proceso, pudiendo ser en un momento dado efectivas o no (el conjunto de efectivas es un subconjunto del conjunto de permitidas, pudiendo el propio proceso decidir en cada momento cuáles de sus *capabilities* permitidas están activas como efectivas y cuáles no).

Por defecto, un proceso del *root* posee todas las *capabilities* (pruebe *getpcaps 1* para obtener las *capabilities* poseídas por el proceso *init*) y uno de un usuario no tiene ninguna (use *getpcaps $$* para imprimir las *capabilities* de su proceso *bash*).

La llamada al sistema *capset* (se recomienda utilizar bibliotecas como *libcap*) permite manipular los distintos conjuntos de *capabilities*. Un proceso de un usuario convencional, como se analizará más adelante, está muy limitado a la hora de manipular sus *capabilities*, no pudiendo incorporar al conjunto de permitidas más de las que se le han asignado.

Falta ese aspecto clave: ¿Cómo obtiene sus *capabilities* un proceso de usuario?

De manera similar a lo que ocurre con un proceso que gana superpoderes al ejecutar un programa con *SetUID/SetGID*, un proceso obtiene sus *capabilities* cuando ejecuta un fichero que tiene asociadas *capabilities*.

Las *capabilities* de un fichero están almacenadas como atributos extendidos. El mandato *capget* permite ver las *capabilities* asociadas a un fichero y *capset*, que solo puede ser utilizado por procesos del *root* o aquellos que poseen la *capability CAP_SETFCAP*, permite gestionar las *capabilities* asociadas a un fichero.

Cada fichero tiene asociados dos conjuntos de *capabilities* (*permitted* e *inheritable*) y un bit efectivo (*effective*). Nos centramos en el conjunto de *capabilities* permitidas y en el bit efectivo. Cuando un proceso ejecuta un programa con un conjunto de *capabilities* permitidas, el proceso obtiene como *capabilities* permitidas las del fichero. En caso de que esté activo el bit efectivo, esas mismas *capabilities* pasar al conjunto de *capabilities* efectivas del proceso. Por tanto, sin el bit efectivo, el proceso ha ganado la capacidad de hacer cosas, pero no podrás hacerlas hasta que promocione las *capabilities* permitidas como efectivas. Con el bit efectivo, el proceso ya está preparado para realizar su labor.

Una vez realizadas las operaciones controladas por las *capabilities*, el proceso puede descartar esos poderes añadidos, ya sea de forma temporal, eliminando la *capability* del conjunto de efectivas, o definitiva, quitándola del conjunto de permitidas.

Dedicamos este párrafo final a comentar muy brevemente los otros conjuntos de *capabilities* no tratados:

- *Bounding set*: Conjunto asociado a un proceso que establece un límite máximo en las *capabilities* que puede obtener un proceso.
- *Inheritable*: Conjuntos asociados tanto a un proceso como a un fichero que permiten establecer qué *capabilities* se pueden heredar (en el sentido de mantenerse después del *exec*), que corresponden a la intersección de ambos conjuntos. Nótese que las *capabilities* permitidas que tiene un proceso se pierden al ejecutar un programa.
- *Ambient*: Con los conjuntos descritos hasta ahora, si un proceso con *capabilites* ejecuta un programa sin *capabilities*, se quedará sin ninguna. Esta característica limita significativamente el uso de *capabilities* en un escenario en el que hay un programa con *capabilities* que ejecuta programas convencionales (por ejemplo, un intérprete con *capabilities* las pierde al ejecutar sus programas). El conjunto de *capabilities* de tipo *ambient* resuelven este escenario porque se mantienen después del *exec*.

Ejercicio 5

En primer lugar, identifique algún programa en su equipo que use *capabilities*, especificando cuáles utiliza y mostrando cómo el fichero ejecutable tiene atributos extendidos asociados.

Como segunda actividad, vamos a *desescalar* un proceso con *SetUID*. Saque una copia de un fichero con *SetUID* (al sacar la copia ya pierde el *SetUID*) y use *capabilities* para conseguir que mantenga su funcionalidad sin ser del usuario *root*.

A continuación, consiga que *servidor1* funcione correctamente sin *SetUID* ni ser del usuario *root*.

Por último, y para sacar nota, incluya el código necesario para que *servidor3* pueda *desescalarse* de forma temporal y definitiva.

Plazo y modo de entrega

El plazo de entrega del trabajo es el 8 de junio de 2021.

La entrega se realiza en *triqui* ejecutando el mandato:

entrega.soa proteccion.2021

Este mandato realizará la recolección del directorio *~/DATSI/SOA/proteccion.2021* de los ficheros *autor.txt*, con los datos del alumno, *memoria.pdf*, que debe incluir la solución del ejercicio, así como los ficheros *servidor2.c* y *servidor3.c*.