UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Android: Instrumentalización del Sistema y Teoría de Grafos Aplicada

Proyecto Final de Carrera

Ingeniería Informática

Autor: Andrea Villa

Director: Ismael Ripoll

6 de septiembre de 2013

Resumen

Android se presenta como una plataforma eficiente y eficaz, con un diseño complejo e innovador, que centra sus esfuerzos en la optimización y compartición de los recursos del sistema. Dadas estas peculiares características, resulta considerablemente complejo poder rastrear y restringir los recursos que un determinado proceso o aplicación tiene derecho a utilizar. Surge la necesidad de extender el modelo tradicional de seguridad del kernel de Android (Linux sobre ARM), con un mecanismo capaz de ejercer mayor control sobre el sistema. Partiendo del soporte que ofrece TOMOYO para monitorizar todas las operaciones que cada aplicación realiza, se ha extendido su funcionalidad para incorporar los servicios de alto nivel que utilizan las aplicaciones Android. TOMOYO es una implementación concreta de lo que se conoce como MAC (Mandatory Access Control).

Entre otros, los nuevos recursos que se pueden administrar son: geolocalización GPS, reproducción y grabación de audio, y captura de eventos por pantalla táctil. La gestión consiste tanto en monitorizar los recursos que cada aplicación emplea, como en aplicar una política de protección activa sobre los mismos.

Este sistema se ha empleado como herramienta para analizar la interacción entre las diferentes componentes de la plataforma. Los datos extraídos se han representado como elementos de grafos, sobre los cuales se han aplicado diferentes algoritmos.

Palabras clave: android, tomoyo, linux, binder, sandboxing, security, mandatory-access-control.

Índice general

Capítulo 1

Introducción

1.1. Motivación

Android es un sistema operativo basado en kernel Linux, principalmente orientado a plataformas móviles, con un diseño enfocado a la optimización en la utilización de recursos. Las optimizaciones incluyen: extenso uso de mapas de memoria compartidos entre diferentes aplicaciones a través de una máquina virtual Java rediseñada; mecanismos ad-hoc de RPC[1]; y un sistema de gestión de seguridad híbrido entre el tradicional DAC[2] y listas de permisos empotrados en los manifiestos de las aplicaciones — este mecanismo se ve detallado en la sección **Seguridad** que se encuentra a partir de la página 13 —. El sistema DAC utilizado en Android es igual a cualquier otro sistema implementado sobre Linux: usuarios y grupos a los que se delega la administración de ficheros y carpetas por medio de asignación de proprietarios y permisos. Por encima de éste, en el caso de Android, los permisos sobre los servicios sensibles del dispositivo — GPS, grabación de audio, acceso a contactos, mensajes y registro de llamadas entre otros — se ven gestionados a nivel de aplicación por medio de manifiestos incrustados en los paquetes instalables de cada aplicación. Todos estos servicios son ofrecidos por medio de un mecanismo IPC[3]. Dada esta fuerte orientación al IPC, resulta muy complejo poder administrar los servicios y recursos que una aplicación tiene finalmente derecho a utilizar. Es posible, de hecho, que algunas aplicaciones tengan indirectamente acceso a servicios a los que no se ha explícitamente concedido el derecho en fase de instalación [9]. Como mostrado por la figura 1.1, una aplicación a la que se ha concedido solamente el derecho a utilizar

[1]Remote Procedure Call
[2]Discretionary Access Control
[3]Inter-Process Communication

la cámara, podrá indirectamente acceder a las coordinadas GPS del dispositivo simplemente consultando los metadatos guardados en el fichero imagen resultante.

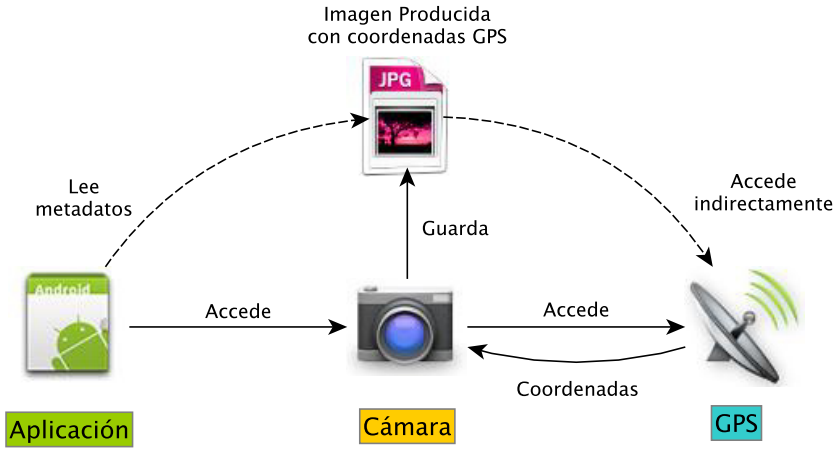


Figura 1.1: Ejemplo de *Confused Deputy*

Los mecanismos de seguridad actualmente utilizados por el sistema han resultado ineficaces contra numerosos ataques [32, 7, 30, 6], siendo la capa de middleware el principal objetivo de explotación. Es importante subrayar que varios estudios e implementaciones [3, 12, 4, 13, 11, 22, 20] ya han sido desarrollados para intentar solucionar este importante aspecto, aunque ninguno ha encontrado especial favor fuera del ámbito estrictamente académico. Dadas estas lagunas de control y rastreo en la actual implementación de la plataforma, se ha planteado la extensión del modelo de seguridad utilizado por Android introduciendo el porting de un mecanismo MAC[4]. El término MAC se refiere a todos aquéllos mecanismos de control de recursos que permiten al sistema operativo rastrear y restringir los accesos a los mismos con una granularidad más fina que el tradicional mecanismo de usuarios, grupos y permisos UNIX. Normalmente estos sistemas utilizan conjuntos de reglas ACL[5] — es decir, listas de control de acceso — sobre los recursos individuales a proteger, para poder así definir con gran precisión los derechos que las componentes del sistema pueden ejercer sobre los mismos. Importante característica de todo sistema MAC es la centralización de los conjuntos de ACL en ficheros de política a los cuales solo el administrador del equipo tendrá acceso. De este modo, contrariamente a mecanismos DAC, los usuarios no

[4]Mandatory Access Control
[5]Access Control List

tienen control sobre los permisos de los recursos que poseen, sino que éstos son bien definidos en los ficheros de política. El kernel Linux dispone de muchas implementaciones MAC, entre las cuales están *AppArmor*, *SELinux*, *grsecurity* y *TOMOYO*. Se ha decidido utilizar y portar esta última: TOMOYO Linux. Las razones de esta elección se basan en el diseño del mecanismo. TOMOYO, contrariamente a otras implementaciones de sistemas MAC como SELinux — que ha sido ideado exclusivamente como mecanismo de protección —, ha sido inicialmente diseñado y desarrollado haciendo hincapié en la instrumentalización[6] y análisis comportamental del sistema, haciendo de éste una herramienta ideal para el estudio a realizar. Solo en fases posteriores TOMOYO ha sido modificado para ser utilizado como mecanismo de protección. TOMOYO será así capaz de analizar el sistema almacenando, en modo aprendizaje, datos sobre el comportamiento de las componentes individuales de la plataforma. De este modo será más sencillo tener una visión global de la interacción entre aplicaciones y servicios para el sucesivo diseño de una política de seguridad. Además TOMOYO, contrariamente a SELinux, divide las componentes del sistema en dominios basándose en *pathnames* en lugar de *inodes* [18]. Para no entrar en detalle de lo que esto supone, solamente es útil remarcar que esta característica permite reducir considerablemente las modificaciones necesarias a la arquitectura de la plataforma Android para su adaptación, y simplifica notablemente el desarrollo de políticas.

Desafortunadamente TOMOYO no es capaz de interpretar y formalizar correctamente algunos importantes mecanismos internos de la plataforma Android. Este estudio pretende enfrentarse a la adaptación de TOMOYO a los mecanismos internos de Android mediante la modificación del código fuente relativo, para así poder disponer de un sistema de protección MAC capaz de monitorizar y restringir el acceso a recursos a lo largo de la plataforma. Es evidente que podría ser de gran interés poder analizar y eventualmente restringir solo a las aplicaciones de confianza el acceso a recursos sensibles como GPS, captura de audio y otros. Además, esta acción se aplicaría a nivel kernel, no dependiendo de ningún proceso usuario.

Cuando se instala una aplicación Android, ésta solicita al usuario permiso para utilizar un conjunto de servicios. Durante el proceso de instalación no es posible seleccionar solo aquellos permisos que se consideren oportunos, sino que todo el bloque de permisos tiene que ser aceptado o rechazado. En el caso de rechazar los permisos especificados por el manifiesto del paquete, la plataforma se negará a proceder con la instalación de la aplicación. Una vez que los permisos se han aceptado y la aplicación resulta instalada en

[6]Proceso a través del cual se insertan instrucciones en puntos críticos del sistema, para poder monitorizar componentes y comportamientos específicos del mismo

el dispositivo, ya no hay ninguna manera de averiguar si ésta, durante su uso, abusa de los permisos concedidos; por ejemplo registrando datos sobre la localización GPS del usuario y enviando periódicamente esta información a sus servidores.

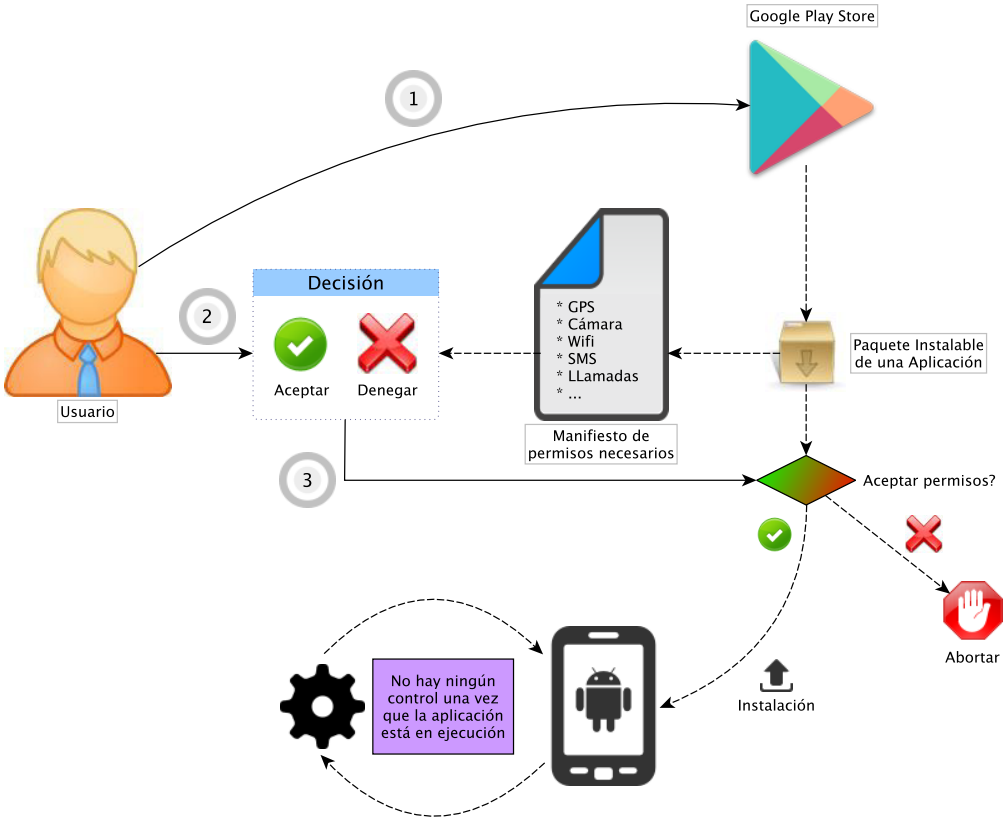


Figura 1.2: Los permisos se aceptan en bloque antes de la instalación

Surge entonces la necesidad de controlar en última instancia el acceso a servicios y recursos sensibles, introduciendo un mecanismo de control durante la ejecución de las aplicaciones. De este modo sería posible analizar y estudiar la interacción entre las varias componentes del sistema, diseñar una política de seguridad apropiada para las aplicaciones críticas, y restringir su comportamiento a lo anteriormente estudiado. Como es evidente, las implicaciones pueden tener enorme interés tanto en los aspectos de estudio y optimización del sistema operativo Android, como en la protección de la privacidad del usuario final ante comportamientos anómalos o delictivos de las aplicaciones.

1.2. Objetivos

El principal objetivo es el porting y adaptación del mecanismo TOMOYO Linux para la plataforma Android. Este objetivo supone una serie de subobjetivos:

- Realizar el porting de TOMOYO sin ninguna modificación.
- Demostrar que el mecanismo requiere de algunas modificaciones estructurales para adaptarse eficazmente a la plataforma Android.
- Estudiar detenidamente los mecanismos de seguridad ya existentes en la plataforma, para poder consecuentemente adaptar y extender la estructura de TOMOYO.
- Modificar el código fuente de los módulos de kernel Linux correspondientes para introducir las siguientes variaciones:
  - Instrumentalizar, añadiendo los controles de seguridad MAC, el driver Android RPC Binder.
  - Extender las funciones de monitorización de TOMOYO para soportar una nueva funcionalidad y sintaxis.
  - Modificar los mecanismos internos de reconocimiento de dominios de TOMOYO para el efectivo reconocimiento de los procesos de las aplicaciones usuario.
- Adaptar las herramientas usuario contenidas en el paquete TOMOYO para interactuar con la correspondiente funcionalidad kernel.

1.3. Organización de la Memoria

El Capítulo 2 (*Estructura del Sistema Android*) se centrará en ofrecer una panorámica de la plataforma Android, centrándose especialmente en aquellas componentes y mecanismos internos claves para la sucesiva implementación y adaptación de TOMOYO Linux.

Una vez descritas todas las características relevantes de la plataforma, se pasará a introducir en el Capítulo 3 (*Instrumentalización del Sistema*) Sección 3.1 (*TOMOYO*) el funcionamiento del mecanismo TOMOYO, explicando todos aquellos detalles que se verán luego involucrados en el proceso de adaptación propiamente dicho.

Sucesivamente, en la Sección 3.2 (*Android y TOMOYO*) del mismo Capítulo, se pasará a introducir dentro de un sistema Android recién compilado

en un ordenador host, las componentes necesarias para el funcionamiento del mecanismo TOMOYO. Estas abarcan tanto ejecutables usuario, como una recompilación del kernel subyacente previsto de todos los parches y configuraciones oportunas. Seguidamente se detallarán las limitaciones prácticas que el sistema TOMOYO, sin ninguna alteración, presenta en el entorno de la plataforma Android.

En la Sección 3.3 (*Adaptación de TOMOYO*) del mismo Capítulo se pasará entonces a la etapa de adaptación propiamente dicha, introduciendo y explicando todas las modificaciones aportadas a TOMOYO, tanto a los módulos del kernel como a las herramientas usuario, para adaptarse a los mecanismos propios y únicos de Android. Consecuentemente a estas adaptaciones, se volverá a repetir un ejemplo de uso del mecanismo modificado, enseñando en práctica todas las nuevas funcionalidades y comportamientos introducidos.

El Capítulo 4 (*Visualización Mediante Grafos*) abarcará la estructuración en forma de grafos de la enorme cantidad de datos recopilados con el nuevo mecanismo, para poder apreciar mejor los resultados conseguidos. Observando finalmente los grafos obtenidos gracias a esta herramienta eficazmente adaptada a la plataforma, será posible formular algunas observaciones sobre el sistema Android.

Por último, el Capítulo 5 (*Conclusiones y Trabajo Futuro*) ofrecerá un resumen de los resultados obtenidos por la implementación modificada de TOMOYO.

La memoria se concluye con cuatro apéndices. El Apéndice A (*Política Inicial*) contiene las políticas TOMOYO utilizadas durante el estudio. El Apéndice B (*Parches Kernel Linux*) contiene los parches diferenciales de los cambios aportados a la parte kernel de TOMOYO. El Apéndice C (*Parche Herramientas User-space*) contiene los parches diferenciales de los cambios aportados a las herramientas usuario de TOMOYO que respaldan las funcionalidades kernel. El Apéndice D (*Grafos*) contiene las imágenes en alta resolución de los grafos modelados a partir de la información proporcionada por el mecanismo TOMOYO.

Capítulo 2

Estructura del Sistema Android

2.1. Panorámica

Android es un sistema operativo basado en kernel Linux, especialmente orientado a dispositivos móviles con procesadores *ARM* y *x86*, se compone de librerías y API escritas en C y C++, una capa de middleware con funcionalidad RPC, una maquina virtual Java con API basada en el proyecto Apache Harmony por su licencia no privativa, y de algunas aplicaciones que desempeñan funciones básicas del sistema. La plataforma está fuertemente orientada a la optimización de recursos, siendo esta característica fundamental para permitir a un sistema tan complejo de desempeñar tanta variedad de funciones en dispositivos que abarcan una vasta gama de capacidades. Componente clave de la plataforma es la Dalvik Virtual Machine, maquina virtual Java especialmente diseñada para el sistema Android. La Dalvik VM es la componente que se ocupa de la ejecución de las aplicaciones usuario poniendo especial hincapié en la compartición de memoria, compilación JIT[1] y sandboxing[2]. Otra fundamental componente del sistema Android es la capa de middleware. Android incorpora una especial modificación del proyecto *OpenBinder*, sistema de comunicación ínter-proceso inicialmente desarrollado por *Be Inc.* y luego parte de los sistemas *PalmOS*. Este se encuentra implementado como driver Linux, y su interfaz es un normal dispositivo de caracteres. El componente Java que facilita y abstrae las llamadas RPC, por medio del Binder, entre las diferentes aplicaciones, es la componente de sistema *Activity Manager*, conjunto de proceduras que implementa varios mecanismos ICC[3]. Esta capa de middleware permite a las distintas componentes del sistema

[1] Just in Time

[2] Aislamiento de procesos

[3] Inter-Component Communication

ofrecer servicios y funciones a lo largo de la plataforma, permitiendo una eficaz y eficiente reutilización de código y recursos.

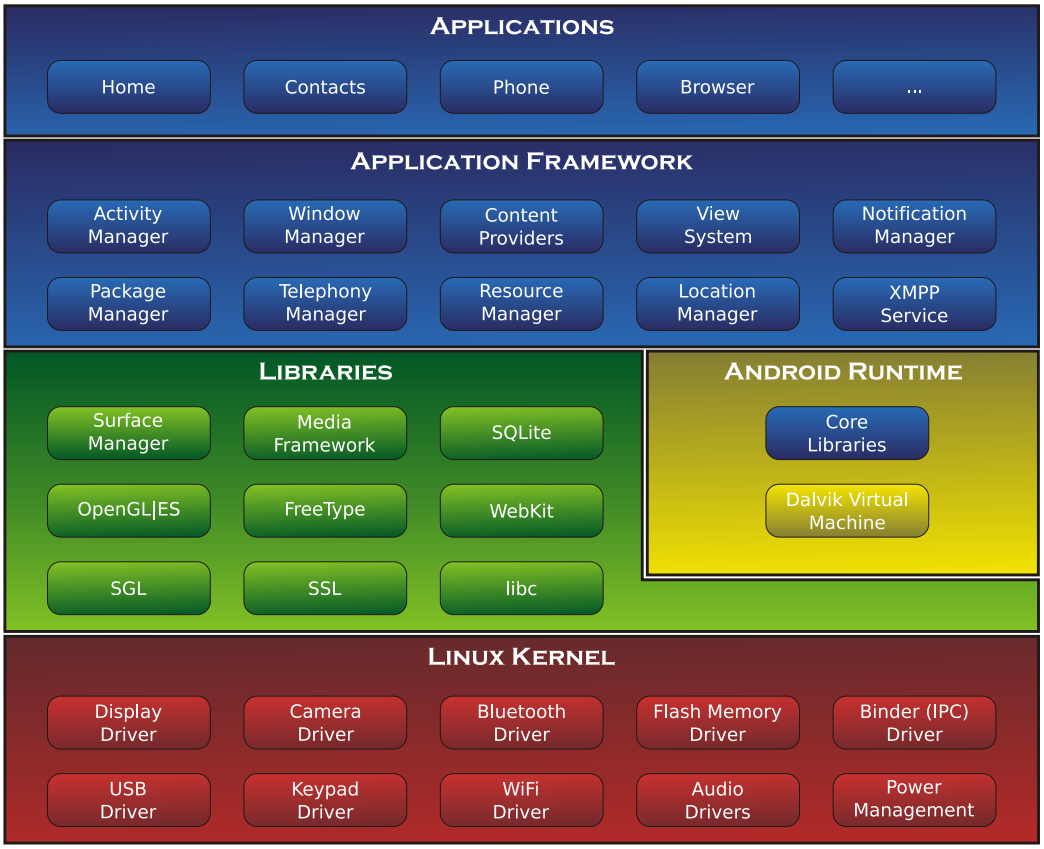


Figura 2.1: Esquema de la plataforma Android

Por encima de las componentes básicas como el Kernel, las librerías y la maquina virtual Java, se encuentra el framework. Este es un conjunto de servicios en forma de métodos ofrecidos a través de interfaces RPC registradas en un proceso que se denomina *System Server*, cuyas llamadas son gestionadas por el servicio Binder. Las aplicaciones que quieran ofrecer servicios o funciones podrán registrarlos en el System Server por medio de llamadas a la API. Estos servicios serán entonces disponibles a las otras componentes del sistema por medio de mecanismos como los *Intents* y los *Content Providers*. Un Intent es un mecanismo abstracto que contiene la descripción de una operación que necesita ser ejecutada, representando un enlace dinámico en tiempo de ejecución entre el código de aplicaciones distintas. Por otro lado un Content Provider es una clase que representa una aplicación que está ofreciendo contenido de forma distribuida. Son aplicaciones que acceden a

contenido estructurado — normalmente una base da datos en formato SQLite — y, encapsulándolo, se ocupan de tramitar esta información a otras aplicaciones, proporcionando, al mismo tiempo, mecanismos de seguridad. Todas estas interfaces abstractas, implementadas en lenguaje de alto nivel, dependen, en última instancia, del dispositivo Binder para el encapsulamiento y transmisión final de los datos entre las diferentes aplicaciones. Resulta entonces extremadamente interesante poder estudiar la interacción, por medio del Binder, de las varias componentes del sistema. A continuación se detallarán las componentes sobre las que se centrará el trabajo realizado.

2.2. Dalvik Virtual Machine

La maquina virtual Java Dalvik fue especialmente diseñada para el proyecto Android por parte de *Google Co.*. La maquina virtual se ha diseñado para ejecutarse en sistemas dotados de procesadores con capacidades de procesamiento limitadas, optimizando la RAM utilizada por medio de compartición de recursos y contando con un mecanismo rediseñado de Garbage Collection. A continuación se detallan algunas de las características de optimización presentes en la cadena de ejecución e interpretación de un programa bajo la maquina virtual Dalvik.

Espacio en Disco

Las clases Java, al igual que una maquina virtual tradicional, son compiladas en un formato intermedio definido como *bytecode*. Sin embargo, las clases que serán cargadas en la Dalvik VM son procesadas con la herramienta *dx* y empaquetadas en formato .dex[4], como se haría de forma análoga a un .jar de una maquina virtual Oracle. Los ficheros .jar se componen simplemente de un formato .zip estándar que incluye, a parte de algún metadato sobre el paquete, todas las clases compiladas que son necesarias a la ejecución de un dado programa. Cada fichero de clase contendrá entonces sus propias constantes y símbolos definidos en el encabezado, sin preocuparse de la redundancia de estas definiciones con otras clases. El formato .dex optimiza la redundancia ofrecida por las diferentes clases. El encabezado del archivo define, en un único fichero, tablas de símbolos, constantes y campos que serán utilizadas a lo largo del programa final, sin dar lugar a alguna repetición innecesaria. De este modo un formato .dex sin comprimir suele ser la mitad de extenso que su equivalente .jar [10].

[4]Dalvik Executable

Memoria

La Dalvik VM es capaz de etiquetar los mapas de memoria utilizados por diferentes aplicaciones para distinguir entre secciones "limpias" y "sucias". La secciones limpias son aquellas que corresponden a un *mmap()* de ficheros (librerías) .dex que no hayan sufrido operaciones de escritura. Estas secciones pueden ser compartidas ágilmente entre todas las instancias de la maquina virtual para permitir una utilización óptima de los recursos cargados en memoria.

Una de las características más importantes del mecanismo de Runtime de la maquina Dalvik es el proceso *Zygote*. Ya que cada aplicación necesita ser contenida en una diferente instancia de Dalvik, el proceso Zygote constituye una instancia especial que arranca con el sistema principal y precarga todas las librerías necesarias para la posterior ejecución de las aplicaciones usuario. De este modo, cuando el sistema necesita arrancar una aplicación usuario, no será necesario recargar todas las librerías básicas de la maquina virtual, sino se contactará con Zygote por medio de un socket UNIX indicándole la aplicación a cargar. Zygote, recibido el comando, hará un *fork()*, el proceso hijo cargará dinámicamente las clases de la aplicación que se necesita ejecutar sin necesidad de arrancar una nueva maquina virtual desde zero, y el proceso padre se pondrá otra vez a la espera en el socket de comandos. A pesar de estas características de compartición de memoria, Zygote implementa un mecanismo *copy-on-write*. Hasta que los mapas de memorias siguen sin alterar, la memoria queda compartida entre diferentes instancias de la maquina virtual. Cuando los mapas limpios se convierten en sucios debido a una escritura por parte de una aplicación, Zygote procede a copiar las paginas compartidas desde su instancia en el mapa de memoria de la maquina virtual que ha alterado dichas secciones. De esta forma se garantiza que la copia compartida siga coherente entre las VM que están compartiendo determinadas páginas, sin impedir a las aplicaciones aportar modificaciones a su proprio espacio de memoria sin que los cambios se repercutan en las otras instancias. El interfaz para estas fuertes características de compartición de memoria es el dispositivo */dev/ashmem*[5]. Este dispositivo, exclusivo de la plataforma Android, permite, a través de operaciones ioctl[6] sobre el mismo, alocar secciones de memoria para uso compartido. Un proceso que haya deseado alocar memoria compartida, tendrá que pasar sucesivamente el descriptor de fichero correspondiente al proceso con el que desea com-

[5]Android SHared MEMory

[6]Llamada de sistema para efectuar operaciones de input/output sobre dispositivos

partir dicha memoria. El paso del descriptor se efectúa normalmente a través del Binder. Estos son los mecanismos empleados internamente por Zygote para gestionar grandes secciones compartidas de memoria entre diferentes instancias de la máquina virtual Dalvik. Dadas estas fuertes características de reutilización de recursos, el mecanismo GC se ha rediseñado para tener en cuenta las componentes utilizadas de forma compartida por medio de un mapa de bits paralelo que cruza todas las instancias Dalvik.

CPU

El mecanismo que emplea Zygote para arrancar aplicaciones en una copia de su mismo proceso para utilizar eficientemente la memoria, también tiene una importante repercusión en el tiempo de carga de las nuevas aplicaciones. La plataforma no necesita arrancar una nueva maquina virtual, simplemente necesita "clonar" una maquina virtual ya completamente cargada en memoria, y, resulta evidente, que el sistema reacciona mucho más ágilmente a la hora de ejecutar una nueva aplicación. Como efecto secundario de este proceso, la carga sobre la CPU se ve considerablemente disminuida. Es interesante notar como la Dalvik VM se ha optimizado para procesadores RISC[7] — como lo son procesadores ARM, montados en la gran mayoría de dispositivos móviles — habiéndose implementado una ABI[8] basada en registros, contrariamente a las maquinas virtuales para PC que son basadas en stack. Estudios [17] han demostrado que una arquitectura basada en registros ejecuta un 47 % menos de instrucciones que una arquitectura basada en stack. Por otro lado, los mismos estudios, han demostrado que el código compilado para una ABI basada en registros suele ser un 25 % más extenso que su correspondiente, sin embargo el sobrecoste derivado de la búsqueda de estas instrucciones demás implica un despreciable incremento del 1.07 % de sobrecoste computacional. Por otro lado, el 25 % de incremento en el tamaño del código final es ampliamente compensado por el formato .dex, siendo esta, en última análisis, la arquitectura ideal para dispositivos que no destacan por su capacidad computacional.

Seguridad

Cada aplicación en el sistema es gestionado por una instancia totalmente diferente de la maquina virtual Dalvik. Esto permite a los di-

[7]Reduced Instruction Set Computer
[8]Application Binary Interface

ferentes procesos de tener espacios de memoria compartidos RELRO[9] y espacios de memoria propios en los que se permite escritura. Por otro lado Android implementa un modelo de sandboxing aprovechando las características DAC del subyacente kernel Linux: cada aplicación instalable en el dispositivo es un paquete .apk firmado por medio de una clave privada que solo el autor original debe poseer. Sin embargo, muy recientemente se ha comprobado que este mecanismo no ha sido suficientemente depurado, permitiendo a un atacante modificar el código ejecutable contenido en un paquete .apk sin alterar la firma digital que debería garantizar por la integridad del paquete original [29]. El sistema, cuando se procede a instalar una aplicación, registra la firma contenida en el paquete en un directorio dedicado y asigna a esta un usuario y grupo UNIX únicos. Todas las instancias de la aplicación serán un proceso hijo de Zygote llamado *app_process* cuyos EUID[10] y EGID[11] corresponderán a aquellos registrados durante la fase de instalación. Todas las carpetas y ficheros dedicados al almacenamiento no volátil de la aplicación son protegidos con el correspondiente usuario y grupo, impidiendo a aplicaciones diferentes que provengan de autores diferentes — es decir, no firmadas con la misma clave privada — acceder y potencialmente alterar los datos de otras aplicaciones. Es interesante especificar como a todas las aplicaciones de sistema se le asigne un UID y GID de 1 a 9999, mientras a las aplicaciones usuario los UIDs serán ≥ 10000. Si este mecanismo, por un lado, es completamente apropiado para proteger los ficheros de datos de las aplicaciones, por otro, muchos recursos son accedidos, a lo largo del sistema, por la capa de middleware representada por el driver Binder. Sobre éste, el sistema DAC ya no resulta suficiente para defenderse de ataques como *Confused Deputy* [9] y *Privilege Escalation* [7].

A partir de este resumen de características, los mecanismos internos de la Dalvik VM crean un entorno de ejecución ideal en sistemas móviles. Sin embargo, muchas de las características descritas no son detalladamente documentadas por ninguna fuente oficial, y han sido derivadas de una recopilación de publicaciones académicas y análisis directa y autónoma sobre el código fuente de las componentes involucradas. Los aspectos descritos en el apartado de **Seguridad** son aquellos que se estudiarán de forma más detallada, una vez presentadas las otras componenentes esenciales, a lo largo de la evolución de este trabajo.

[9] RELocation Read-Only
[10] Effective User IDentifier
[11] Effective Group IDentifier

2.3. Binder IPC

La implementación actual del driver Binder procede del proyecto OpenBinder, originariamente desarrollado como componente del sistema operativo BeOS y luego introducida a la base de los sistemas Cobalt. El driver Binder es un mecanismo IPC [19, 28] que intenta introducir un sistema de paso de mensaje con orientación a objetos por encima de kernel tradicionales. La interacción con la componente Binder se realiza principalmente por medio de un aplicación de sistema denominada Activity Manager, esta es la responsable de gestionar los Intents lanzados por las otras actividades en ejecución en el dispositivo, actuando como una especie de binding[12] a lo largo de todo el sistema entre la capa Java y el dispositivo Binder ofrecido por el driver. Esta característica se hace extremadamente importante en el sistema Android, ya que permite una abstracción de las proceduras remotas por medio de objetos Binder de Java, dando la habilidad a las aplicaciones del sistema de ofrecer servicios locales registrándose previamente en el directorio del driver. Muchas componentes esenciales del sistema Android se comunican haciendo uso del Binder, permitiendo concurrencia de operaciones con un sistema orientado a objetos por medio de interfaces y stubs como los más comunes sistemas *CORBA*. Uno de los ejemplos más interesantes es el servicio *SurfaceFlinger* que, registrado en el Binder, permite a las diferentes aplicaciones de dibujar sus elementos concurrentemente en la pantalla del sistema, dejando la responsabilidad del compositing[13] al servicio registrado.

Un proceso de sistema concreto conglomera casi la totalidad de los servicios fundamentales del sistema. Este proceso se denomina *System Server*. Como se puede incluso observar en la figura 2.2 muchas componentes de la API Android son realmente peticiones al System Server por medio del Binder. El Binder es entonces una presencia transversal en el sistema, que ofrece y coordina todos los servicios registrados en la plataforma. Las aplicaciones usuario pueden, a parte de invocar los servicios ofrecidos por las aplicaciones de sistema o bien por medio de la API o bien directamente, registrar sus propias proceduras en el Binder utilizando tres posibles alternativas: extendiendo la clase Binder, utilizando una clase de mensajería para compartir una instancia de Binder con el cliente, o utilizar una variante del lenguaje IDL[14] llamada *AIDL* para definir previamente los objetos a distribuir y dejar que un compilador genere las componentes necesarias (Proxy y Stubs). Estos procedimientos son necesarios para generar y distribuir interfaces que sean

[12] Adaptación de una biblioteca o componente para ser usada en un lenguaje de programación distinto de aquél en el que ha sido escrita

[13] Combinación de diferentes elementos para generar la imagen de salida final

[14] Interactive Data Language: lenguaje diseñado para la definición de objetos distribuidos

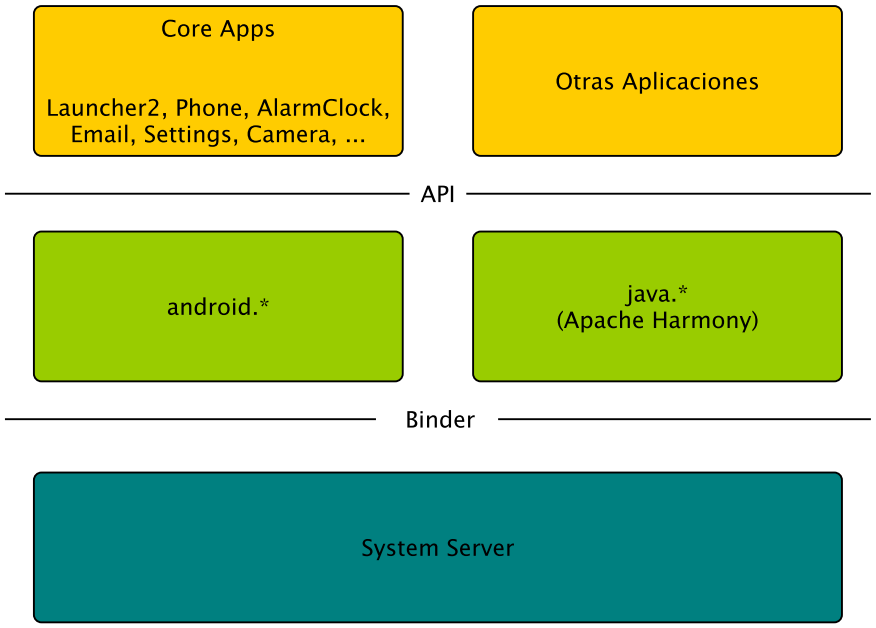


Figura 2.2: Capas System Server y Binder

extensiones de objetos Binder, que están especialmente programados para ofrecer de forma transparente mecanismos de marshalling[15] y comunicación entre otros.

Una de las componentes más interesantes y más críticas [5, 26] del sistema Binder es la gestión de la seguridad. Uno de los ataques [9] más comunes y más alarmantes sobre esta plataforma tiene como objetivo este componente. El Binder tiene de hecho dificultad en recurrir la cadena de llamadas que ha derivado en una determinada invocación de una procedura RPC, no pudiendo determinar si el proceso invocante tiene finalmente los permisos apropiados para utilizar un determinado servicio. A cada aplicación instalable en formato .apk se le asocia un manifiesto, el cual deberá contener todos los permisos a servicios que la aplicación demanda para operar correctamente. Todos estos deberán ser comprobados, validados y aceptados por parte del usuario en el momento de la instalación. A partir de ese momento, la aplicación podrá hacer uso solamente de los servicios oportunamente declarados en su manifiesto, no permitiéndose algún cambio excepto por una actualización de la aplicación, durante la cual el usuario tendrá que validar de nuevo los permisos de la misma.

De todos modos este mecanismo de permisos junto con el sandboxing im-

[15]Proceso de transformación de los objetos en un formato adecuado para la transmisión

pulsado por la Dalvik VM no han resultado suficientes contra mucha cantidad y tipos de ataques [32, 6, 30].

Como se puede notar, este es probablemente el componente que desempeña el rol más importante a la hora de gestionar los derechos que las aplicaciones deben tener en acceder a los servicios del sistema. Por estas principales razones se ha realizado el trabajo sobre esta componente, que nos permitirá rastrear y analizar las interacciones que se efectúan entre las aplicaciones durante el uso de la plataforma por parte de un normal usuario.

Capítulo 3

Instrumentalización del Sistema

3.1. TOMOYO

Las herramientas de *Mandatory Access Control* se basan en la introducción, en determinados puntos del código fuente del sistema, de especiales llamadas que verifican los permisos para ejecutar una determinada acción por parte de una aplicación. Estas bifurcaciones de control se definen *hooks*. Las herramientas MAC despliegan sus hooks en todos aquellos puntos críticos del sistema en los que se podrá controlar el accesso a recursos, como pueden ser abertura de ficheros, creación de sockets, uso de dispositivos y demás. Estas herramientas cuentan con una especial sintaxis para definir las políticas asociadas a cada recurso, permitiendo o denegando el acceso a los mismos.

TOMOYO es una implementación MAC para sistemas Linux, desarrollada por la compañía japonesa NTT Data Corporation. El modelo MAC utilizado por TOMOYO divide las aplicaciones en dominios según las cadenas de invocaciones que han llegado a la ejecución de un dado proceso, como representado por la figura 3.1.

De este modo todos los dominios tienen como raíz el dominio *<kernel>*, para luego especializarse según la cadena de invocaciones correspondiente. Es importante especificar como un dominio incluya por defecto a todos sus descendientes que no hayan sido especificados. De este modo el dominio *<kernel>/sbin/init* incluye por defecto a todos aquéllos dominios que deriven de las invocaciones por parte del proceso */sbin/init*. Así pues una regla de política declarada en el dominio *<kernel>* afectará a todo el sistema, excepto a aquéllos dominios que se hayan explícitamente declarado con reglas diferentes. A diferencia del más común y más complejo *SELinux*, TOMOYO define los dominios basándose en *pathnames* en lugar de *inodes* [18]. Este

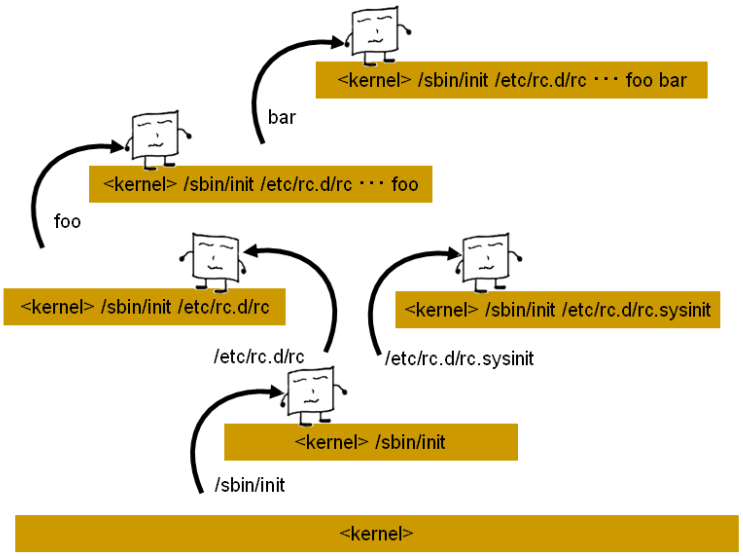


Figura 3.1: Transición de dominios TOMOYO

diseño se hace más apropiado para sistemas de ficheros agnósticos a mecanismos de acceso alternativos a DAC y que no soportan atributos extendidos *xattr*, siendo, estos últimos, componente esencial para que SELinux pueda manejar las informaciones de seguridad relacionadas a un fichero dado. A cada dominio se le puede asociar un perfil, que define la acción a tomar cuando una aplicación contenida en ese dominio intente hacer uso de un recurso que no esté definido en la política de seguridad cargada por el sistema. TOMOYO fue inicialmente lanzado como conjunto de parches para el kernel Linux, conjunto completo de todas las funcionalidades MAC que venían desarrolladas en las versiones publicadas. Empezando por la versión 2.6.30 del kernel Linux, TOMOYO ha empezado a formar parte de las versiones oficiales del código fuente del núcleo, basando su implementación en el sistema de hooks ofrecidos por los módulos *Linux Security Modules* (LSM). Los LSM han sido inicialmente desarrollados y diseñados siguiendo los requerimientos de SELinux, lo cual no ha inicialmente permitido a TOMOYO aprovechar de sus interfaces. Gracias a la introducción en los LSM de algunos hooks adicionales, TOMOYO ha sido parcialmente adaptado a este nuevo sistema. Desde entonces el desarrollo de TOMOYO se ha dividido en dos grandes ramas: la versión 1.x, ofrecida como parche externo al núcleo, que implementa sus propios hooks y no hace uso del framework ofrecido por los LSM, y la versión 2.x, ofrecida como parte del código fuente del núcleo, que, haciendo uso de LSM, no puede ofrecer toda la gama de funciones implementadas en la versión 1.x.

Se ha elegido utilizar la versión 1.x de la herramienta para poder tener el mayor rango posible de funciones y para poder adaptar y extender los hooks implementados para que fueran apropiados para la plataforma Android, sin tener repercusiones en toda la infraestructura LSM.

TOMOYO es en práctica una herramienta de control de acceso que permite reforzar el control que ya aplican los procesos en la capa usuario, centrando sus esfuerzos en los sujetos de las acciones — los procesos — en lugar de centrarse en los objetos — los ficheros —. Es esencial que este tipo de herramientas residan en la capa núcleo para evitar que procesos maliciosos puedan saltarse los controles de protección mediante varios mecanismos, mientras, de este modo, el control se hace inevitable para toda la capa usuario. TOMOYO distingue los procesos no solamente por su nombre, sino por todo el historial de invocaciones que han llegado a la ejecución de un dado ejecutable. Cada vez que un ejecutable invoca a otro, se produce una concatenación en el dominio al que pertenece el ejecutable actual para generar uno más especializado en el que incluir el nuevo proceso. Este mecanismo es así capaz de dividir finemente todos los procesos que hayan llegado a existir en el sistema, incluyéndolos en unos dominios que representen, en forma de árbol, el historial de invocaciones de los procesos en cuestión. Podemos así ver los dominios como unas cajas que incluyen, o han llegado a incluir, determinados procesos activos basándose en su historial de ejecución. Estas cajas pueden contener entradas ACL que representan derechos de acceso a recursos, distinguiendo su tipología y entidad, siguiendo una sintaxis previamente definida [21]. Cada ACL representa entonces un determinado acceso a recurso, incluyendo el tipo de recurso accedido, el tipo de acceso a dicho recurso y todo los parámetros relacionados. El objetivo del mecanismo es registrar constantemente y permanentemente los accesos a los recursos durante la ejecución normal del sistema, incluyendo las entradas ACL correspondientes en el dominio que llegó a producir las peticiones. El conjunto de todas las ACL contenidas en los diferentes dominios se define *política*. Una vez almacenadas todas las entradas necesarias para el correcto funcionamiento de una aplicación, es posible restringir los permisos de acceso a recursos de dicha aplicación solo a aquellas entradas precedentemente registradas en el dominio correspondiente, siguiendo el principio de mínimos requerimientos. Se asegura, de este modo, que serán denegadas todas aquellas peticiones que se desvíen del comportamiento normal precedentemente estudiado.

3.1.1. TOMOYO 1.x

Como se ha descrito en precedencia, la versión de TOMOYO utilizada por esta implementación es de la familia 1.x, concretamente la versión 1.8.x.

De ahora en adelante "TOMOYO" se referirá siempre a la versión 1.8.x. Esta familia, implementada gracias a la inserción de hooks específicos, cuenta con toda la funcionalidad MAC de la plataforma, y se distribuye en forma de parche a aplicar sobre un kernel Linux vanilla[1]. Al conjunto de funcionalidades así insertadas dentro del kernel, que no hacen uso de los LSM, se les denomina *CCSecurity*.

Esta versión, al igual que las otras, separa las aplicaciones en dominios siguiendo el historial de invocaciones como representado por la figura 3.1

A cada dominio se le puede llegar a asociar un perfil. Los perfiles especifican la reacción del sistema ante la petición por parte de un proceso a acceder a un determinado recurso que no esté especificado en forma de ACL en su correspondiente dominio. A un perfil se le pueden incluso asociar parámetros para controlar tanto el logging de operaciones así como el tamaño de los buffers internos utilizados por el módulo, con el fin de limitar la utilización de memoria por parte de el conjunto de políticas. Sin embargo el parámetro más significativo de un perfil es el modo de operación. Existen cuatro modos de operación exclusivos:

Disabled No se aplica ningún mecanismo de control. El kernel reacciona normalmente con controles DAC.

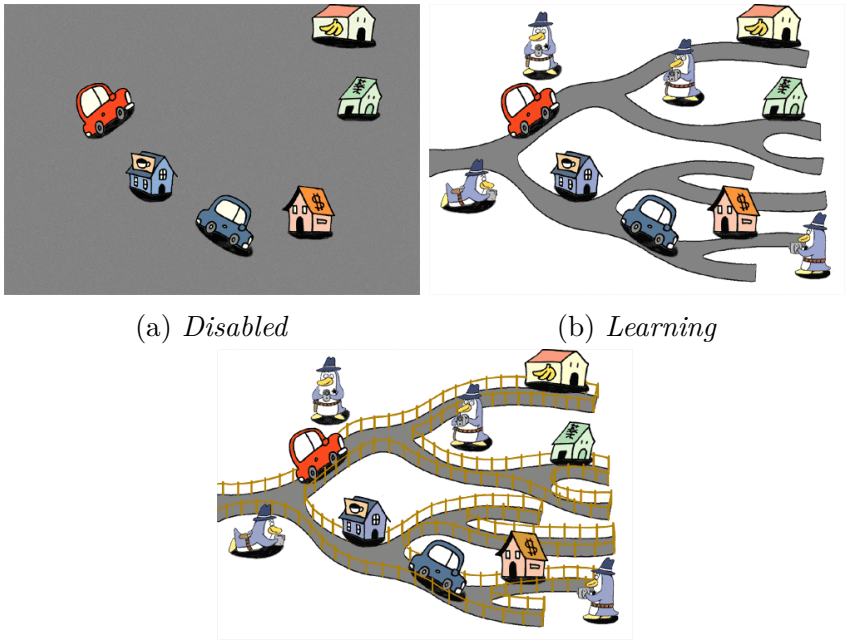
Learning Cualquier petición que viole la política de seguridad será permitida. La entrada ACL correspondiente al acceso deseado será añadida a la política.

Permissive Cualquier petición que viole la política de seguridad será permitida y será registrada en un fichero de log determinado. No se añade ninguna entrada a la política.

Enforcing Se verá denegada cualquier petición de acceso a recurso que no cumpla con la política de seguridad establecida.

Se pueden ver estos modos como las correctas fases del desarrollo de una política de mínimos requerimientos. En una primera fase (3.2a) el sistema no está realmente instrumentalizado, y los procesos acceden a los recursos sin ningún control y análisis. En la fase de aprendizaje (3.2b) TOMOYO es capaz de estudiar y tener traza de todos los accesos a recursos por parte de los diferentes dominios presentes en el sistema, definiendo y rellenando poco a poco la política de seguridad correspondiente a cada ejecución. La siguiente fase del proceso de desarrollo de la política de seguridad es comprobar que esta sea la adecuada para la correcta ejecución de los procesos críticos en

[1]Oficialmente sacado por Linus Torvalds y colaboradores, sin ningún parche adicional



(a) *Disabled* (b) *Learning*

(c) *Enforcing*

Figura 3.2: Fases de desarrollo de política

el sistema. Para ello se ponen los dominios interesados en modo permisivo, monitorizando los logs de fallos de la política hasta ahora rellenada. Una vez consolidada y asegurada la política de seguridad definida en el sistema, se pasa finalmente a restringir el comportamiento de los dominios solamente a las entradas especificadas (3.2c).

Las políticas en uso por el sistema TOMOYO son dinámicas, y pueden ser alteradas sin necesidad de volver a arrancar el kernel. TOMOYO emplea el subsistema *procfs* para ofrecer las interfaces necesarias a interactuar con el mecanismo. Para interactuar dinámicamente con TOMOYO de forma más conveniente, se ofrecen un conjunto de herramientas llamadas *ccs-tools*. Las herramientas más interesantes para este estudio, contenidas en el conjunto ccs-tools, son las siguientes:

ccs-editpolicy Esta herramienta, por medio de una interfaz ncurses[2], permite visualizar los dominios detectados por TOMOYO en forma de árbol, analizar sus transiciones, editar las políticas asociadas a cada uno, editar los perfiles y gestionar cada aspecto de la plataforma por medio de diferentes pantallas. Esta es seguramente la herramienta más esencial y completa para el control del mecanismo.

[2]Librería gráfica para terminales de caracteres

ccs-auditd Esta herramienta con función de demonio permite la redirección de los buffers de log de los diferentes perfiles de TOMOYO en ficheros apropiadamente especificados en su fichero de configuración.

ccs-editpolicy-agent Este demonio permite la redirección de las interfaces de control registradas en el subsistema procfs a través de un socket de red. Este componente es especialmente importante en plataformas embedded, ya que con estas podría no ser posible interactuar directamente a través de ccs-editpolicy, necesitando entonces redirigir el contenido de las interfaces TOMOYO a través de un socket para que pueda comunicarse con una máquina host.

Como se puede observar por la captura de pantalla 3.3 de la herramienta ccs-editpolicy, el conjunto de estas herramientas nos permite observar y modificar en vivo la evolución del mecanismo. Desde esta herramienta podemos incluso verificar que determinados procesos han acabado en un determinado dominio.

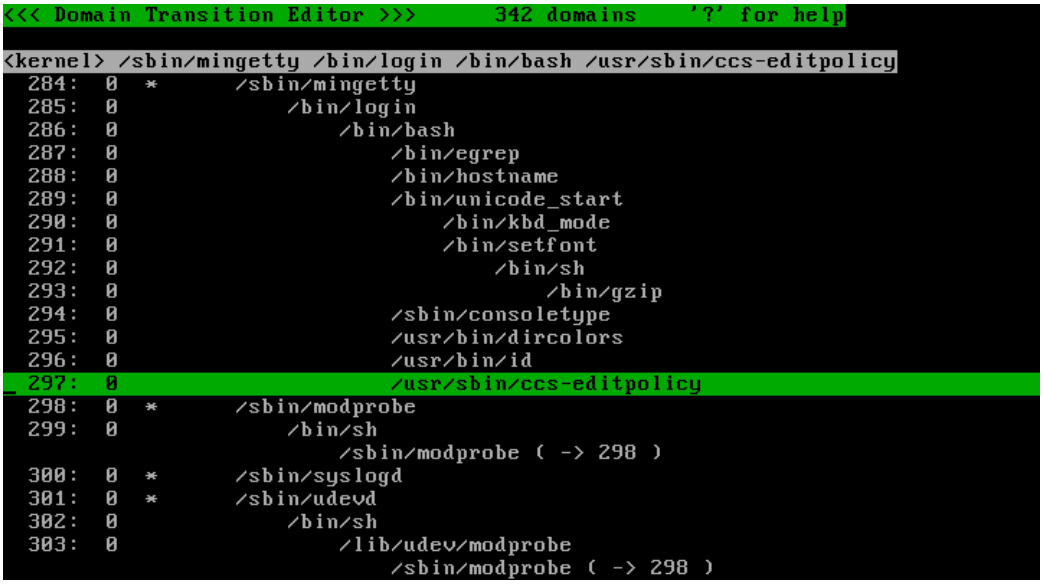


Figura 3.3: ccs-editpolicy, transacciones entre dominios

3.2. Android y TOMOYO

Aplicar TOMOYO 1.8.x a la plataforma Android no es una tarea sencilla, especialmente porque no existe documentación actualizada al respecto. El

proceso se puede resumir en dos principales bloques: la primera fase tratará de compilar enteramente Android, para poder introducir, en la imagen de arranque initramfs[3] que será cargada junto al kernel Linux por el emulador qemu *Goldfish*, la herramienta ccs-editpolicy-agent para poder interactuar con TOMOYO desde la maquina host; la segunda fase tratará de recompilar el kernel utilizado por la maquina virtual con la introducción de los parches para introducir las funcionalidades MAC CCSecurity.

3.2.1. Compilación de Android con ccs-editpolicy-agent

Este proyecto se había inicialmente centrado en la versión de Android 4.0.4, mejor conocida por su nombre en código *IceCream Sandwich*. En esta versión, el árbol de desarrollo no contaba con herramientas de compilación diseñadas adrede para la generación inicial del proyecto, y resultaba entonces imprescindible formar previamente un ambiente adecuado para la compilación, disponiendo y acondicionando el entorno con versiones especificas de las componentes esenciales para una correcta generación de la plataforma. A pesar de haberse conseguido la compilación y el funcionamiento bajo esta versión, se decidió actualizar la plataforma a la versión 4.2.2, la última disponible, conocida con el nombre de *Jelly Bean*. Esta versión, contrariamente a las anteriores, viene con una versión modificada del compiladar GNU *gcc 4.6*, lo cual hace considerablemente más sencillo el proceso de compilación en distribuciones no oficialmente soportadas por la guía de desarrollo.

La primera fase del proceso de compilación es la obtención de todos los fuentes que componen la plataforma final de Android. Para ello se dispone de una herramienta python creada adrede para la plataforma. La herramienta se compone de un script llamado *repo*, a través del cual es posible, sencillamente especificando la etiqueta de la revisión de la plataforma que se quiere adquirir, contactar y clonar directamente todos los repositorios de los proyectos que concurrirán a la correcta generación de Android y de sus componentes de emulación y gestión. De ahora en adelante la variable bash *ANDROID* representa la raíz del directorio de fuentes apenas descargado. Después de haber descargado los fuentes, es oportuno predisponer el ambiente de trabajo con una carpeta destinada al almacenamiento, por parte de *ccache*[4], de los ficheros cache necesarios para eventuales recompilaciones. Android es, de hecho, una plataforma extremadamente extensa, y su compilación puede tardar varias horas, incluso con numerosos hilos de compilación paralelos. Antes de lanzar el proceso de compilación, Android necesita que se le especifique para

[3]Initial Ram File System

[4]Herramienta que optimiza la recompilación por medio de cache

que plataforma concreta estamos generando el sistema. Para ello lanzamos en la carpeta raíz del fuente Android:

```
$ cd $ANDROID
$ source build/envsetup.sh
$ lunch full-eng
```

El primer comando es necesario para importar todos los aliases y variables de entornos esenciales para el proceso de compilación. El segundo comando especifíca que queremos una versión completa de la plataforma Android, orientada a la ejecución bajo la máquina virtual basada en qemu Goldfish y con todos los símbolos de depuración posibles. De esta forma tendremos una maquina virtual qemu con la cual será posible depurar la plataforma de forma más sencilla y eficiente.

Una vez finalizado el proceso de compilación, tendremos que introducir la herramienta ccs-editpolicy-agent dentro las carpetas de sistema de la plataforma recién compilada. Para ello necesitamos primero compilar el fuente de la herramienta con el toolchain[5] con el cual se ha compilado la plataforma Android para permitir su ejecución dentro del emulador. En este caso el toolchain toma prefijo *arm-linux-androideabi-*. Sin embargo este paso no es sencillo ya que, entre otras cosas, todos los ejecutables de Android están enlazados con una librería C — *Bionic* — diseñada adrede para el sistema, y cuentan con muchos encabezados dispersos en el árbol de fuentes de la plataforma. Andrew Ross diseñó en 2010 un script perl llamado *agcc* [27] que actuaba como wrapper[6] al toolchain de Android, permitiendo una compilación muy sencilla de programas nativos. A fecha de Agosto 2013, la herramienta no ha sido correctamente adaptada a la nuevas versiones de Android, que, como se ha comentado anteriormente, hacen uso de un toolchain modificado por la propia compañía Google. Para obviar a este inconveniente se ha entonces monitorizado el proceso de compilación de un ejecutable nativo estáticamente enlazado ya presente en la plataforma Android y se ha utilizado la misma invocación para compilar el fuente de la herramienta ccs-editpolicy-agent. La invocación final resultante es la siguiente:

```
$ prebuilts/gcc/linux-x86/arm/
   arm-linux-androideabi-4.6/bin/
   arm-linux-androideabi-gcc -nostdlib -Bstatic
   -Wl,--gc-sections -o ccs-editpolicy-agent -Isystem/
   core/include -Ihardware/libhardware/include
```

[5]Conjunto de programas utilizados en el desarrollo software de un producto específico
[6]Componente que actúa encapsulando a otro

```
   -Ihardware/ril/include -Idalvik/libnativehelper/
   include -Iframeworks/base/include -Iexternal/skia/
   include -Iout/target/product/generic/obj/include
   -Ibionic/libc/arch-arm/include -Ibionic/libc/
   include -Ibionic/libc/kernel/common -Ibionic/libc/
   kernel/arch-arm -Ibionic/libm/include -Ibionic/libm
   /include/arch/arm -Ibionic/libthread_db/include
   -Ibionic/libm/arm -Ibionic/libm -Iout/target/
   product/generic/obj/SHARED_LIBRARIES/
   libm_intermediates -Lout/target/product/generic/obj
   /lib out/target/product/generic/obj/lib/
   crtbegin_static.o  -Wl,-z,noexecstack -Wl,-z,relro
   -Wl,-z,now -Wl,--warn-shared-textrel -Wl,--icf=safe
    -Wl,--fix-cortex-a8   -Wl,--no-undefined
   ccs-editpolicy-agent.c -Wl,--whole-archive
   -Wl,--no-whole-archive -Wl,--start-group  out/
   target/product/generic/obj/STATIC_LIBRARIES/
   libc_intermediates/libc.a    prebuilts/gcc/
   linux-x86/arm/arm-linux-androideabi-4.6/bin/../lib/
   gcc/arm-linux-androideabi/4.6.x-google/armv7-a/
   libgcc.a -Wl,--end-group out/target/product/generic
   /obj/lib/crtend_android.o
$ file ccs-editpolicy-agent
ccs-editpolicy-agent: ELF 32-bit LSB  executable, ARM,
    EABI5 version 1 (SYSV), statically linked, not
   stripped
```

Como se puede observar, se ha producido un ejecutable que Android puede ejecutar nativamente bajo el emulador de arquitectura ARM qemu. La siguiente etapa es la introducción del ejecutable recién compilado en el directorio */sbin* de la plataforma, para que el proceso *init* pueda ejecutarlo durante el arranque. Para ello se procede a individuar y desempaquetar la imagen initramfs que se carga junto con el kernel en las primeras fases de boot. La carpeta *out/target/product/generic* contiene todos los ficheros imagen necesarios al arranque de la plataforma emulada. De ahora en adelante la variable de entorno *GENIMG* hará referencia a esta carpeta.

```
$ cd out/target/product/generic
$ mkdir initramfs
$ cd initramfs
$ zcat ../ramdisk.img | cpio -id
```

En *out/target/product/generic/initramfs/* se encuentra ahora entonces la raíz del árbol de directorios de la imagen. Se procede así a copiar el ejecutable del demonio CCS a la sub-carpeta *sbin*. Para que el proceso init arranque el ejecutable durante el mecanismo de boot, se han introducido las siguientes lineas en el correspondiente fichero de configuración *init.rc*, situado en la raíz del ramdisk.

```
service ccs_agent /sbin/ccs-editpolicy-agent 0.0.0.0:7000
  class core
  oneshot
```

De este modo, el demonio se asociará al puerto *7000* de la interfaz software ofrecida por qemu, permitiendo a la máquina host de interactuar con TOMOYO por medio de ccs-editpolicy.

Una vez realizados estos cambios, se produce una nueva imagen ramdisk:

```
$ find . -print0 | cpio -o0 -H newc | gzip -9 >
   ../ramdisk.img
```

La plataforma Android es así preparada para interactuar con el host para la gestión de los dominios y políticas. La siguiente fase abarca la introducción de las funcionalidades de TOMOYO dentro del kernel utilizado por la plataforma Goldfish.

3.2.2. Parchear el Kernel Goldfish

El proyecto Android dispone de su proprio árbol de desarrollo de kernel Linux. De esta forma se produce un kernel optimizado para la plataforma de emulación qemu Goldfish. TOMOYO, a fecha de Agosto 2013, soporta oficialmente la versión *2.6.29* de kernel Linux Goldfish, y distribuye un parche adaptado a esta versión. Primeros pasos para la aplicación del parche al kernel Linux Goldfish son la obtención del código fuente desde los repositorios git oficiales, individuación del branch[7] que marca la referencia a la versión 2.6.29 y hacer checkout[8] al mismo.

```
$ git clone https://android.googlesource.com/kernel/
   goldfish
[ ... ]
$ cd goldfish
$ git branch -a
```

[7]Rama del árbol de desarrollo

[8]Hacer que el árbol de desarrollo refleje el estado de una referencia especifica

```
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/android-goldfish-2.6.29
  remotes/origin/android-goldfish-3.4
  remotes/origin/linux-goldfish-3.0-wip
  remotes/origin/master
$ git checkout -b pfc origin/android-goldfish-2.6.29
Branch pfc set up to track remote branch android-
   goldfish-2.6.29 from origin.
Switched to a new branch 'pfc'
```

De ahora en adelante la variable de entorno *GOLDFISH* representará la raíz del fuente. Los siguientes pasos requieren una atención especial, en cuanto necesitamos parchear y configurar el kernel adecuadamente para el emulador Goldfish dotado de un procesador virtual *ARMv7*, activando al mismo tiempo toda la funcionalidad perteneciente a CCSecurity. Tendremos así que definir, para la configuración y compilación, dos variables de entorno. La primera variable, *ARCH*, sirve a indicar la arquitectura objetivo para los scripts que mecanizan la compilación del núcleo. La segunda, *CROSS_COMPILE*, sirve para forzar el proceso de compilado, bajo condiciones de cross-compiling[9], a utilizar un toolchain específico. Obviamente se utilizará el toolchain precedentemente empleado para la compilación del ejecutable ccs-editpolicy-agent. De este modo realizamos un cross-compiling adecuado, utilizando el toolchain de la plataforma. Por sencillez definimos un alias *kmake* de bash.

```
alias kmake='ARCH=arm CROSS_COMPILE=$ANDROID/prebuilts
   /gcc/linux-x86/arm/arm-linux-androideabi-4.6/bin/
   arm-linux-androideabi- make'
```

Tendremos además que definir una política inicial adecuada para el correcto funcionamiento del sistema. TOMOYO permite dos estrategias a la hora de almacenar y cargar las políticas de seguridad. Se puede configurar CCSecurity en el kernel para que, conforme se ejecute un determinado ejecutable por parte del sistema — normalmente */sbin/init* en cuanto primer proceso ejecutado durante el arranque — invoque otro ejecutable usuario para cargar las políticas en el sistema. Otra estrategia es la de empotrar una política base directamente en el imagen binaria del kernel, haciendo que CCSecurity no dependa de ningún ejecutable usuario. Evidentemente esta configuración tiene una desventaja importante: los cambios en tiempo de ejecución en la

[9]Técnica por la cual se compila un código fuente para una arquitectura diferente a la del ordenador que está ejecutando la compilación

política de seguridad serán efectivos solamente hasta el siguiente arranque del sistema, durante el cual se volverá a cargar la política predefinida contenida en la imagen. Sin embargo, para evitar introducir otro ejecutable ajeno en la imagen initramfs de la plataforma, y por su sencillez, se ha elegido esta última estrategia. Así pues a continuación configuramos un kernel básico para la plataforma Goldfish, lo parcheamos con el último conjunto de parches descargables de la página web oficial del proyecto TOMOYO, insertamos la configuración adecuada para la activación de CCSecurity, y finalmente creamos la carpeta que hospedará los ficheros de la política a empotrar en la imagen final del kernel compilado.

```
$ tar -zxf ccs-patch-1.8.3-20130707.tar.gz
$ patch -s -p1 < patches/ccs-patch-2.6.29-android-
   goldfish.diff
$ kmake -s goldfish_armv7_defconfig
$ cat <<EOI >>.config
CONFIG_CCSECURITY=y
# CONFIG_CCSECURITY_LKM is not set
# CONFIG_CCSECURITY_DISABLE_BY_DEFAULT is not set
CONFIG_CCSECURITY_USE_EXTERNAL_TASK_SECURITY=y
CONFIG_CCSECURITY_MAX_ACCEPT_ENTRY=2048
CONFIG_CCSECURITY_MAX_AUDIT_LOG=1024
CONFIG_CCSECURITY_OMIT_USERSPACE_LOADER=y
CONFIG_CCSECURITY_FILE_READDIR=y
CONFIG_CCSECURITY_FILE_GETATTR=y
CONFIG_CCSECURITY_NETWORK=y
CONFIG_CCSECURITY_NETWORK_RECVMSG=y
CONFIG_CCSECURITY_CAPABILITY=y
CONFIG_CCSECURITY_IPC=y
CONFIG_CCSECURITY_MISC=y
CONFIG_CCSECURITY_TASK_EXECUTE_HANDLER=y
CONFIG_CCSECURITY_TASK_DOMAIN_TRANSITION=y
CONFIG_CCSECURITY_PORTRESERVE=y
EOI
$ mkdir -p security/ccsecurity/policy/
```

La carpeta *security/ccsecurity/policy/* contendrá los ficheros de política, listados en el apéndice A. Como se puede observar en el fichero *profile.conf* (A.4), hemos definido por simplicidad solamente dos perfiles en modo learning. Esto hará que el sistema CCSecurity almacene todos los acceso a recursos no explícitamente definidos en la política. El perfil número 2, a parte de añadir dinámicamente entradas a la política, nos permitirá logear la in-

formación de forma masiva para sucesivas análisis. El perfil número 3 define un simple modo *Enforcing*.

Es posible, además, especificar un conjunto de excepciones a la política. Este se ve definido en el fichero *exception_policy.conf* (A.2). Este fichero conglomera todos aquellos accesos a recursos que el mecanismo deberá permitir, en caso de fallo se recurrirá al comportamiento definido por el modo de operación del perfil asociado al dominio que ha causado el fallo en la política. En este fichero definimos un grupo ACL 0 en el cual permitimos toda una serie de operaciones básicas, que irían contrariamente a inundar los logs con información poco interesante sobre el uso del sistema. Nótese que se ha comentado fuera la definición de excepción de ioctl sobre todo fichero o dispositivo. Esto será útil para poder logear todos los intentos de ioctl por parte de la plataforma. A bajo nivel, de hecho, las transacciones RPC efectuadas a través del binder por parte de las aplicaciones, son efectivamente operaciones ioctl sobre el dispositivo de caracteres */dev/binder*.

El fichero *domain_policy.conf* (A.1) finalmente asocia dominios a perfiles y grupos ACL. Aquí se define que los dominios hijos de <kernel>— es decir, todos — son asociados al perfil 1 en modo aprendizaje simple. Asociamos este dominio al grupo ACL 0 para evitar el logeo de los accesos básicos. El dominio *<kernel>/init /system/bin/app_process* será aquél en el cual serán contenidas todas las aplicaciones usuario bajo la máquina virtual Java Dalvik, como detallado en el apartado **Memoria** de la página 12. A este asignamos el perfil numero 2, que nos permitirá extraer muchísimos datos en forma de logs a través de la herramienta ccs-auditd. Además, asignamos a este un grupo ACL totalmente vacío, para poder tener la mayor cantidad de datos posibles. Finaliza así el proceso de configuración del kernel Linux con funcionalidad MAC CCSecurity.

Para la compilación del núcleo volvemos a hacer uso del alias kmake. Una vez finalizado el procedimiento, la imagen final del kernel se encuentra en *$GOLDFISH/arch/arm/boot/zImage*, lista para ser ejecutada en el emulador Goldfish. Lanzamos entonces el emulador, utilizando el kernel recién compilado y la imagen initramfs que contiene el demonio ccs-editpolicy-agent.

```
$ source $ANDROID/build/envsetup.sh
$ emulator -kernel $GOLDFISH/arch/arm/boot/zImage -
   ramdisk $GENIMG/ramdisk.img
```

3.2.3. Resultados y Limitaciones

Con el emulador corriendo, tendremos la plataforma ejecutándose con funcionalidad CCSecurity. El demonio ccs-editpolicy-agent estará esperando

por conexiones en el puerto 7000 de la interfaz virtual del emulador. Para interactuar con el sistema será necesario redirigir el tráfico de un puerto en la interfaz loopback[10] del ordenador host al puerto 7000 de la interfaz virtual del emulador. Al ser interfaces diferentes, utilizaremos el mismo número de puerto en la interfaz local. El emulador Goldfish dispone de una consola, a la cual es posible acceder simplemente con nc[11] o telnet conectándose a un puerto específico — en este caso el 5554 —, a través de la cual es posible, entre otras cosas, efectuar la redirección necesaria. Consecuentemente conectamos la herramienta ccs-editpolicy al puerto apenas redirigido.

```
$ echo "redir add tcp:7000:7000" | nc 127.0.0.1 5554 &
    PID=$!; sleep 1; kill $PID
[ ... ]
$ ccs-editpolicy 127.0.0.1:7000
```

Como se puede observar en la captura 3.4 todos los procesos pertenecientes a las aplicaciones usuario se ven insertados en un único dominio *<kernel>/init /system/bin/app_process*. Esto se debe al mecanismo con el cual las aplicaciones son ejecutadas por la plataforma Android. TOMOYO es capaz de reconocer las invocaciones que conllevan a una especialización de dominio solo a través de llamadas al sistema de tipo *execve*. Esta llamada es la responsable de sustituir explícitamente la imagen de un ejecutable, contenida en el proceso invocante, por otra especificada en los parámetros de la llamada. Android emplea un mecanismo muy diferente a la hora de ejecutar las aplicaciones usuario. Como anteriormente detallado, existe una instancia especial de la máquina virtual Java Dalvik denominada Zygote. Esta instancia es la responsable de recibir las órdenes de ejecución de las otras aplicaciones. Cuando Zygote recibe la orden de ejecución de una determinada aplicación, hace un fork y procede a cargar dinámicamente el conjunto de clases Java de la aplicación indicada. Con este mecanismo se generan procesos hijos directamente del proceso Zygote, cada uno representando una aplicación diferente, y, en última instancia, ninguno de ellos emplea la llamada de sistema execve. Por esta razón, TOMOYO puede seguir la cadena de invocaciones solo hasta el proceso app_process, en cuanto éste es realmente el que representa la maquina virtual Java. Como resulta evidente, este aspecto representa una enorme limitación, en cuanto un solo dominio contiene las entradas ACL de los recursos accedidos por todas las aplicaciones juntas, y no permite diferenciar y estudiar el comportamiento de cada una, repartiéndolas en dominios diferentes.

[10] Interfaz virtual que permite conexiones en el contexto de la máquina local

[11] Herramienta textual para leer y escribir a través de sockets TCP o UDP

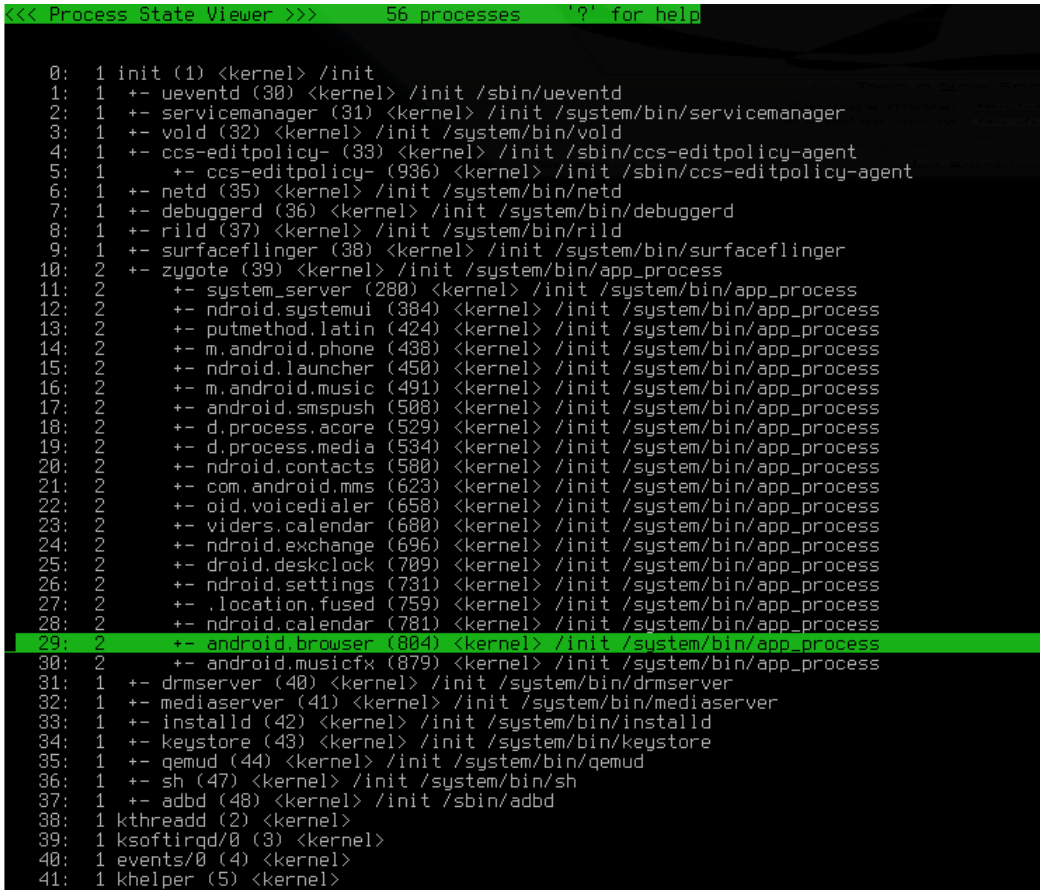


Figura 3.4: ccs-editpolicy, aplicaciones usuario colapsadas en un dominio

Además, como muestra la captura 3.5, TOMOYO no es evidentemente capaz de disectar las transacciones efectuadas a través del Binder. Esto supone un gran problema a la hora de analizar el sistema, en cuanto una enorme cantidad de servicios de la plataforma es ofrecido por medio de invocaciones RPC, gestionadas en forma de transacciones sobre el dispositivo /dev/binder — a bajo nivel, operaciones ioctl —.

Por estas razones surge la necesidad de extender el mecanismo CCSecurity para adaptarse a estas características. Por una parte las aplicaciones deberán resultar divididas en los correspondientes dominios, y, por otra, será necesario implementar algunos principios de disección sobre las transacciones a través del Binder, para así poder tener una imagen completa de la interacción entre las varias componentes de la plataforma.

```
<kernel> /init /system/bin/app_process
 1045: file ioctl    /dev/ashmem 0x40047703
 1046: file ioctl    /dev/ashmem 0x40047705
 1047: file ioctl    /dev/ashmem 0x40087707
 1048: file ioctl    /dev/ashmem 0x40087708
 1049: file ioctl    /dev/ashmem 0x41007701
 1050: file ioctl    /dev/ashmem 0x7704
 1051: file ioctl    /dev/binder 0x40046205
 1052: file ioctl    /dev/binder 0x40046208
 1053: file ioctl    /dev/binder 0xC0046209
 1054: file ioctl    /dev/binder 0xC0186201
```

Figura 3.5: ccs-editpolicy, ioctl en /dev/binder

3.3. Adaptación de TOMOYO

3.3.1. Procedimientos y Herramientas

Para el análisis y desarrollo se ha utilizado la distribución *Slackware Linux* versión *14.0*. Esta distribución es la más antigua que se sigue desarrollando activamente, nacida en 1993. Slackware Linux se considera como una de las distribuciones más avanzadas, que intenta respetar la mayor cantidad de estándares y convenciones UNIX, aportando las mínimas modificaciones posibles a los programas incluidos en ella, manteniéndose fiel a las intenciones de los autores de los mismos. Estas características hacen de Slackware Linux una de las distribuciones más avanzadas y flexibles para el hacking software.

Enfrentarse al hacking y manipulación del kernel Linux requiere configuraciones, conocimientos y procedimientos concretos y estructurados. Para enfrentarse a la depuración del kernel es necesario habilitar algunas características de compilación que permitirán producir un código ejecutable enriquecido de símbolos que ayuden durante las operaciones de tracing[12]. Para ello se habilitan, en la configuración del kernel Goldfish, unas opciones que produzcan, durante el compilado, los símbolos necesarios.

```
CONFIG_DEBUG_KERNEL=y
CONFIG_DEBUG_INFO=y
CONFIG_FRAME_POINTER=y
```

La opción *CONFIG_FRAME_POINTER* cobra especial importancia en cuanto instruye el compilador a producir código que retenga siempre, en un

[12]Operaciones finalizadas a analizar y registrar el flujo de ejecución de una aplicación

registro del procesador, un puntero coherente al último stack frame. Esto es de vital importancia en las operaciones de depuración, en cuanto, en coincidencia de un breakpoint, será posible observar el backtrace completo de la pila de llamadas.

El proceso de compilación del kernel Linux emplea fuertes características de optimización del código ejecutable. Éstas, si por un lado benefician grandemente las prestaciones en tiempo de ejecución, por otro lado dificultan la depuración, en cuanto el compilador reestructura, optimizándolos, algunos bucles y funciones, haciéndolos así no alineados con el código original, complicando consecuentemente el proceso de stepping[13] en los depuradores. Tendremos así que deshabilitar los parámetros de optimización en la producción de aquellos ficheros objeto en los que se centrará el proceso de depuración. El trabajo de adaptación se centrará en dos componentes principales: los mecanismos CCSecurity y el driver del dispositivo Binder. Para desactivar las optimizaciones de los ficheros objeto de éstas dos componentes, será necesario añadir en los *Makefile*, contenidos en las carpetas de los mismos, una línea que especifique la opción de no optimización por cada fichero ejecutable a producir. Se deberá modificar la variable *CFLAGS*, que especifica los parámetros a pasar al compilador gcc, incluyendo la opción *-O0* para indicar que se requiere un nivel de optimización nulo. En el fichero Makefile contenido en la carpeta de CCSecurity *$GOLDFISH/security/ccsecurity* se añaden las siguientes lineas:

```
CFLAGS_permission.o = -O0
CFLAGS_gc.o = -O0
CFLAGS_memory.o = -O0
CFLAGS_policy_io.o = -O0
CFLAGS_realpath.o = -O0
CFLAGS_load_policy.o = -O0
CFLAGS_ccsecurity.o = -O0
```

En el Makefile del driver Binder se añaden de forma análoga. La carpeta que contiene el Makefile es *$GOLDFISH/drivers/staging/android*.

```
CFLAGS_binder.o = -O0
CFLAGS_logger.o = -O0
```

El proceso de depuración de un kernel tiene dificultades añadidas con respecto a la depuración de un normal ejecutable. En primer lugar siempre

[13]Proceso de depuración por el cual es posible ejecutar el código línea por línea

se necesita una máquina de respaldo con la cual efectuar la depuración propiamente dicha, comunicándose con el código ejecutable por medio de un mecanismo de comunicación, más comúnmente un puerto serial. Sería de hecho imposible pretender depurar un kernel en la misma máquina en la cual éste se está ejecutando, en cuanto resulta evidente que, conforme el código en ejecución reciba una señal de trap[14] causada por un breakpoint u otra condición, el kernel y consecuentemente todos los procesos incluyendo él depurador, se bloquearían inmediatamente esperando una señal que le permitan seguir su ejecución. En el caso de la plataforma Android, será posible operar la depuración en un solo equipo gracias a la máquina virtual ofrecida por el emulador Goldfish. El kernel Linux incluye una funcionalidad denominada *KGDB* [8] que permite, con la introducción de un pequeño stub[15] en el código del kernel, comunicarse con una instancia de GDB puesta al otro lado de un puerto serial estándar. De esta forma es posible conectar el depurador GDB a un kernel ejecutándose en otra máquina. Desafortunadamente esta funcionalidad no es del todo estable, en cuanto es responsabilidad del kernel que está siendo depurado gestionar, durante su propia ejecución, la interacción con el GDB puesto en remoto. Otra dificultad es representada por la redirección del puerto serial, dentro del contexto de emulación, a la máquina host. Afortunadamente, otra posibilidad de depuración es ofrecida por el emulador de la plataforma Android. El emulador es, en buena sustancia, una extensión de la máquina virtual y emulador de arquitectura qemu. Qemu, durante la ejecución del sistema, puede hacer que una instancia de GDBServer, que se puede comunicar incluso a través de TCP/IP, arranque por encima del kernel, permitiendo de esta forma un control completo y preciso por parte del GDB en la máquina host [25, 24]. Cada versión del depurador GDB está configurada y compilada para entender un conjunto de instrucciones y ABI de una arquitectura específica. Será entonces necesario utilizar el depurador parte del toolchain precompilado en la plataforma Android. Este ha sido configurado con *--host=x86_64-linux-gnu --target=arm-linux-android*. Esta compilación concreta de GDB se ve enlazada con la librería *python 2.6*, que se ha procedido a instalar en el sistema. La variable de entorno *PY26LIBS* representará de ahora en adelante el recorrido en el cual se encuentra la librería dinámica de python 2.6. En la linea de comandos será necesario definir la variable de entorno *LD_LIBRARY_PATH* para especificar recorridos adicionales de búsqueda de librerías dinámicas. El ejecutable del cual GDB deberá cargar los símbolos de depuración es la imagen del kernel Linux sin

[14]Interrupción síncrona normalmente derivada de una condición excepcional ocurrida durante la ejecución del código

[15]Pequeño trozo de código utilizado como sustituto de otra funcionalidad

comprimir *vmlinux*. El depurador GDB se lanzará entonces de la siguiente forma:

```
$ LD_LIBRARY_PATH=$PY26LIBS $ANDROID/prebuilts/gcc/
   linux-x86/arm/arm-linux-androideabi-4.6/bin/arm-
   linux-androideabi-gdb $GOLDFISH/vmlinux
```

Se dispone, de este modo, de todas las herramientas necesarias para el análisis y modificación de las componentes involucradas en la adaptación planteada.

3.3.2. Análisis

Se ejecuta el emulador con los parámetros necesarios para la conexión de una instancia de GDB.

```
$ emulator -kernel $GOLDFISH/arch/arm/boot/zImage -
   ramdisk $GENIMG/ramdisk.img -qemu -s -S
```

El parámetro *-S* especifica, además, que la emulación no tiene que comenzar hasta que un monitor — es decir, el GDB conectado a la máquina virtual — no indique de continuar con la ejecución. Consecuentemente se conecta la instancia local de GDB a la máquina virtual. La opción *-s* asocia por defecto el GDBServer al puerto *1234* de la interfaz local.

```
(gdb) target remote :1234
```

El proceso de análisis se centrará alrededor de las operaciones ioctl sobre el dispositivo /dev/binder. La disección de las transiciones al Binder, de hecho, puede ocurrir solamente en el contexto de estas operaciones.

En primer lugar se ha detectado que las aplicaciones recién ejecutadas, después del fork de Zygote, emplean una operación ioctl sobre el dispositivo binder. Esta será la ocasión ideal para especializar el dominio al cual pertenecerán las nuevas aplicaciones en ejecución en el sistema. Para la especialización del dominio *<kernel>/init /system/bin/app_process* en uno que represente la aplicación realmente en ejecución bajo el contexto de la máquina virtual Java Dalvik, será necesario acceder a la memoria virtual en el espacio de direcciones usuario del proceso, para extraer el nombre del paquete que contiene el conjunto de clases de la aplicación en ejecución.

Segunda etapa del análisis y desarrollo es establecer una paralelismo entre el actual hook de operaciones ioctl en CCSecurity y la entrada en el driver Binder. Para ello se han añadido dos breakpoints en la funciones *___ccs_ioctl_permission*, punto de control de permisos para las operaciones ioctl en el mecanismo CCSecurity, y *binder_ioctl*, punto de entrada, en el

driver Binder, de las operaciones ioctl sobre el dispositivo, como especificado por la estructura *file_operations*[16] registrada por el código del mismo.

```
Breakpoint 1, __ccs_ioctl_permission (filp=0xdfdec0e0,
    cmd=3222823425, arg=1249250848) at security/
   ccsecurity/permission.c:2581
2581    {
(gdb) bt
#0  __ccs_ioctl_permission (filp=0xdfdec0e0, cmd
   =3222823425, arg=1249250848) at security/ccsecurity
   /permission.c:2581
#1  0xc00a39f0 in ccs_ioctl_permission (arg
   =1249250848, cmd=3222823425, filp=0xdfdec0e0) at
   include/linux/ccsecurity.h:272
#2  sys_ioctl (fd=9, cmd=3222823425, arg=1249250848)
   at fs/ioctl.c:557
#3  0xc002da00 in ?? ()
(gdb) c
Continuing.

Breakpoint 2, binder_ioctl (filp=0xdfdec0e0, cmd
   =3222823425, arg=1249250848) at drivers/staging/
   android/binder.c:2550
2550    {
(gdb) bt
#0  binder_ioctl (filp=0xdfdec0e0, cmd=3222823425, arg
   =1249250848) at drivers/staging/android/binder.c
   :2550
#1  0xc00a30c8 in vfs_ioctl (filp=0xdfdec0e0, cmd
   =3222823425, arg=<optimized out>) at fs/ioctl.c:45
#2  0xc00a3938 in do_vfs_ioctl (filp=0xdfdec0e0, fd=<
   optimized out>, cmd=<optimized out>, arg
   =1249250848) at fs/ioctl.c:539
#3  0xc00a3a0c in sys_ioctl (fd=9, cmd=3222823425, arg
   =1249250848) at fs/ioctl.c:561
#4  0xc002da00 in ?? ()
```

Como es evidente observando el flujo de ejecución, el breakpoint sobre ___ccs_ioctl_permission ocurre primero, en cuanto esta función es la res-

[16]Estructura que contiene varios punteros a función que indican las proceduras a ejecutar cuando se ejecuta una determinada operación de input/output sobre un dispositivo registrado

ponsable de conceder o denegar el acceso a la sucesiva operación ioctl sobre el dispositivo. Como era de esperar, retomada la ejecución, esta se vuelve a bloquear al encontrar el segundo breakpoint puesto a la entrada de la función binder_ioctl. Los argumentos *filp, cmd* y *arg* a las funciones, representan respectivamente el fichero, el comando y el argumento de la operación ioctl. Como se puede observar, hay un paralelismo perfecto entre los contenidos de los parámetros a estás funciones, en cuanto ambas invocaciones son efectuadas por la llamada de sistema *sys_ioctl*.

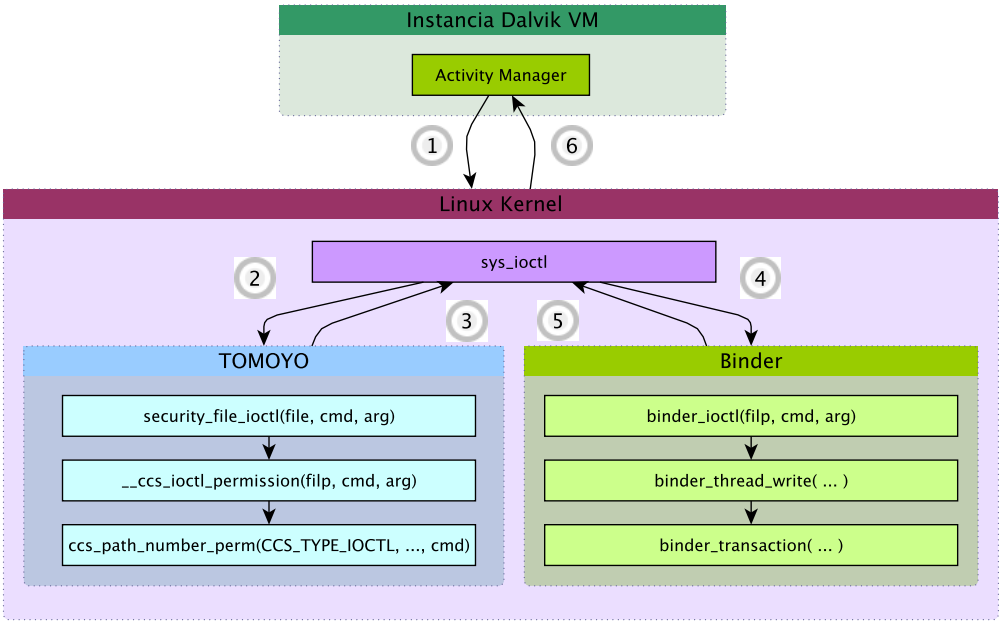


Figura 3.6: Esquema de flujo de operaciones ioctl sobre el dispositivo Binder

La llamada binder_ioctl se ocupará de invocar las funciones que conllevarán al de-marshalling de los datos de la transacción y a la ejecución de la misma, como representado por el esquema 3.6. Se concluye entonces que será posible disectar la transacción en el código de CCSecurity antes de que sea tramitada por el driver del dispositivo Binder. Para ello será oportuno añadir un hook adrede en el código del driver Binder, que hará posible el registro, aceptación o denegación de una transacción con una granularidad más fina que una simple operación genérica de ioctl sobre el dispositivo /dev/binder.

3.3.3. Especializar app_process

La primera fase de la modificación del mecanismo TOMOYO CCSecurity es la especialización de los dominios actualmente representados por el

proceso app_process. Para ello, solamente será necesario modificar las componentes propias de CCSecurity, sin añadir ningún otro elemento en otras secciones del código fuente. Antes de aportar alguna modificación al código del núcleo, definimos una variable de configuración del kernel Linux, como indicado por el parche B.1.1, para poder así habilitar o deshabilitar todas las futuras aportaciones al código fuente. Como especificado por el parche, para la activación de todas las adaptaciones se utilizará la variable de configuración *CCSECURITY_ANDROID*. Como ya comentado en la sección anterior, la especialización del dominio será efectuada durante una operación ioctl. Se ha comprobado, de hecho, que las aplicaciones recién ejecutadas en la plataforma, acceden muy pronto, por medio de operaciones ioctl, a los dispositivos /dev/binder y /dev/ashmem. El Binder es empleado por las nuevas aplicaciones para registrarse y para inicializar su entorno. Secundariamente, Zygote, al lanzar una nueva aplicación usuario, necesita hacer extenso uso del mecanismo de memoria compartida para minimizar el uso de esta por parte de la nueva máquina virtual recién clonada.

El parche B.1.2 representa los cambios necesarios para efectuar la especialización del dominio app_process. En primer lugar se inserta una llamada a una nueva función en el mecanismo de control sobre operaciones ioctl representado por la función ___ccs_ioctl_permission. Sucesivamente, la función *ccs_transit_android_app_process* se ocupa de verificar si el dominio de la aplicación que está ejecutando la operación ioctl puede ser especializado. La primera aplicación en ser asignada al dominio app_process será el proceso de Zygote. Sus hijos tendrán que ser especializados en las aplicaciones que éste lance. La especialización del dominio se realiza mediante concatenación al dominio actual de la línea de comandos o paquete Java extraído por medio de la función *ccs_current_cmdline*. Es evidente que el proceso Zygote no debe ser especializado recursivamente, y tampoco tiene que ser especializada una aplicación que todavía solo se encuentra en estado de inicialización — línea de comando *<pre-initialized>* —. El control en la línea 56 resuelve estos dos aspectos. Una vez comprobado que la aplicación puede ser especializada correctamente, se procede a ello por medio de la llamada a la función *ccs_android_specialize_app*. Esta función simplemente hace uso de la procedura precedentemente citada para acceder al espacio de direcciones del proceso involucrado y utiliza los mecanismos internos de CCSecurity para efectuar el cambio de dominio del proceso al nuevo recién creado.

3.3.4. Hook en Binder IPC

Para empezar con el control sobre las transacciones del Binder IPC, necesitamos poner una llamada a una nueva función, en lugar de hook,

ccs_android_binder_permission en el código del driver Binder de Android. Como mostrado por el parche B.2, insertamos la llamada en el código de la función *binder_transaction*. Este es efectivamente el código, en el Binder, que se ocupa de extraer los datos de la estructura de transacción, llegando finalmente a ejecutarla. Será así posible pasar al mecanismo de CCSecurity todos los datos necesarios sobre la transacción, para registrarla y evaluarla completamente. Es interesante observar como solamente se ha insertado el mecanismo de control en transacciones que no son respuestas a otras. Se ha considerado, de hecho, que si una aplicación tiene derecho a ejecutar una transacción inicial, tendrá entonces derecho a recibir la respuesta correspondiente por parte de la otra aplicación. Como se puede observar por el parche, si la llamada a la función de CCSecurity no es satisfactoria, hacemos que el Binder no efectúe la transacción, imprimiendo un mensaje de log y registrando dicha transacción como denegada. Para hacer visible esta función a lo largo del kernel se ha modificado el encabezado *ccsecurity.h* puesto en *include/linux*. Las modificaciones a este fichero (B.1.4) emplean el mecanismo de registro de funciones CCSecurity. Existe una estructura global *ccsecurity_opas* de tipo *ccsecurity_exports* que contiene varios punteros a función. Cada puntero guarda la dirección de una función responsable de la verificación de permisos sobre una determinada operación de acceso a recurso. La estructura, aunque global y exportada, no es accedida directamente por otras componentes del kernel, sino es accedida localmente por algunos stubs globalmente exportados que empiezan con el prefijo *ccs_*. Se declara entonces un stub, en el encabezado, que accederá al campo relevante de la estructura, y finalmente llamará a la función guardada en dicho puntero. Se puede observar el rellenado del puntero correspondiente en la estructura ccsecurity_opas en el parche B.1.3 en las líneas 123–125. La función ccs_android_binder_permission resulta ser entonces uno stub que pasa integralmente todos sus parámetros a la función almacenada en la estructura, la cual será la real responsable del procesamiento. Esto se ha hecho para respetar el estilo de programación de la implementación CCSecurity.

La función ccs_android_binder_permission recibe dos parámetros fundamentales: una variable de tipo *pid_t* que contiene el PID del proceso al cual es destinada la transacción, y un puntero a una estructura de tipo *binder_transaction_data*. Este último es el tipo de datos utilizado internamente por el Binder para guardar toda la información relevante sobre una determinada transacción, así como los datos a tramitar en la transferencia en sí. La estructura guarda la clave para disectar el protocolo interno de las transacciones Binder IPC.

3.3.5. Disección de Transacciones Binder

Para la disección de transacciones, las modificaciones al código fuente del mecanismo CCSecurity son más numerosas y extensas. Cuando el flujo de ejecución del kernel entra en el código del mecanismo CCSecurity para la comprobación de un acceso a recurso, se forma una estructura ACL, acompañada de una determinada sintaxis, que define y especifíca la naturaleza de dicho acceso. Ya que nos encontramos con un tipo de acceso a recurso hasta ahora desconocido por el mecanismo, es necesario establecer unas nuevas estructuras y sintaxis para una ACL que represente una transacción RPC de Android a través del Binder. Se empieza con el diseño de lo que vendrá a ser un nuevo tipo de entrada en los ficheros de política. Tomando inspiración de las entradas dedicadas a otros mecanismos de IPC, se pretende añadir el soporte para una nueva sintaxis que permita el control sobre las transacciones efectuadas a través del Binder de la plataforma Android. La sintaxis deseada es la siguiente:

```
ipc android_binder_transaction SERVICIO DOMINIODESTINO
```

Donde *SERVICIO* indica la función o interfaz remota que se pretende invocar en el contexto de la transacción. *DOMINIODESTINO* toma el valor del dominio — es decir, la aplicación — al cual dicha transacción está destinada. De este modo, CCSecurity no solamente tendrá la sintaxis adecuada para permitir o denegar la invocación de determinados servicios remotos a las aplicaciones teniendo en cuenta el dominio destino, sino que también podrá permitirnos un estudio detallado del sistema. Se pretende, de hecho, que CCSecurity pueda utilizar esta nueva sintaxis, al igual que las definidas en la implementación original, en el modo aprendizaje, para almacenar en la política activa todas las interacciones entre aplicaciones y servicios ofrecidos bajo forma RPC.

Dada la sintaxis indicada, la nueva entrada ACL deberá entonces contener dos campos: el servicio que se está invocando con la transacción y el dominio en cuyo contexto se verá procesada la petición. Para el procesamiento de los datos y para el almacenamiento de la ACL asociada, será necesario definir dos estructuras diferentes. Cada estructura utilizada para almacenar los diferentes accesos a recursos y sus respectivas entradas en los ficheros de política debe ser registrada apropiadamente en el sistema CCSecurity. Esto se debe al hecho de poder controlar la cantidad de memoria utilizada por el mecanismo, limitar su uso por medio de listas circulares enlazadas de objetos únicos utilizando la estructura *list_head*[17], y para poder emplear algoritmos

[17]Tipo definido por el kernel Linux para implementar, de forma eficiente, listas circulares enlazadas; Linux implementa código optimizado para cada arquitectura

de Garbage Collection sobre los objetos no activamente utilizados por las políticas. CCSecurity dispone, en el fichero fuente *gc.c*, de muchas funciones para el registro en memoria de objetos permanentes. Estas funciones, a parte de asignar dinámicamente la memoria en el espacio de direcciones del kernel, también son capaces de gestionar eficientemente los objetos registrados, guardando una sola copia de éstos en listas enlazadas, manteniendo contadores de uso de los mismos. El algoritmo de GC podrá así calcular el espacio consumido por dichas entradas y poder liberar la memoria adecuadamente cuando necesario. El tamaño de las nuevas estructuras tiene que ser así correctamente añadido al conjunto de registros del sistema de GC. La comprensión de las funciones y estructuras involucradas en este proceso necesita de mucho tiempo, ya que la gestión de la memoria dinámica en el contexto del kernel es uno de los aspectos más críticos de la programación en este entorno.

La primera estructura a definir, para tratar los datos en fase de comprobación y procesamiento, será una estructura que englobe los tipos básicos C necesarios: dos cadenas de caracteres. Contrariamente, para el almacenamiento de la ACL se necesita de tipos que las funciones del mecanismo GC puedan tratar y almacenar. La estructura utilizada por CCSecurity para guardar información sobre los dominios es la estructura *ccs_path_info*. Esta estructura contiene principalmente un campo cadena de caracteres y algunos otros campos necesarios al algoritmo GC. Este será el tipo elegido para almacenar tanto el dominio destino de la invocación cuanto la cadena que representa el servicio RPC pedido. A parte de estos dos campos, la estructura de la nueva ACL también necesita de un campo adicional para poder distinguirse de las otras entradas. Esto se hace necesario para que CCSecurity divida las ACL en listas diferentes según el tipo de acceso que representan, para así poder optimizar los algoritmos de búsqueda en la política. Las dos estructuras se encuentran respectivamente declaradas en las líneas 60–65 y 73–79 del parche B.1.3. La primera estructura se ha declarado como miembro de la estructura *ccs_request_info*, estructura clave para el proceso de formación de la ACL correspondiente a un acceso. En consecuencia, es necesario por un lado especificar el tamaño de la estructura que guarda permanentemente la ACL en memoria, para que CCSecurity pueda tener cuenta de la cantidad de recursos utilizados, y por otro indicar al algoritmo GC como liberar la memoria correspondiente a la misma. Estas dos características se ven reflejadas por los cambios en el fichero gc.c, líneas 9–11 y 19–28 del parche B.1.3. Se puede observar como la llamada a la función interna de CCSecurity *ccs_put_name*, contrariamente a la llamada *ccs_get_name* que observaremos más adelante, sirve para reducir el contador asociado a un determinado recurso cadena de caracteres registrado permanentemente en memoria. La función se encuentra implementada de forma que si algún contador sobre un

recurso utilizado por la política llega a ser 0, la memoria correspondiente es liberada.

El entero mecanismo de comprobación y registro de entradas ACL ha sido diseñado e implementado alrededor de la estructura *ccs_acl_param*. Esta estructura desempeña un rol absolutamente clave en todo el mecanismo de gestión de política. La estructura es, de hecho, un tipo de datos capaz de guardar en su especificación cualquier entrada ACL, haciendo uso de la mínima cantidad de memoria posible. La estructura es formada principalmente por un elemento de tipo *union*[18] que contiene las estructuras correspondientes al almacenamiento de todo tipo de entrada ACL en el sistema. Evidentemente, solo una de estas estructuras podrá ser utilizada en una variable de este tipo. Como podrá observarse, primer elemento de las estructuras ACL siempre deberá ser otra estructura de tipo *ccs_acl_info*, que guarda la información sobre el tipo concreto que ésta está representando. A pesar de la *union*, de este modo, se puede siempre acceder al primer miembro de una estructura ACL, aunque no se conozca todavía su tipo específico, para así averiguar su tipo real y aplicar las operaciones oportunas. Este detalle es clave para que CCSecurity pueda realmente utilizar la estructura ccs_acl_param como contenedor genérico de ACL y al mismo tiempo poder detectar el tipo específico de la misma para su procesamiento. Se añade el nuevo tipo de ACL como elemento adicional a este contenedor en la línea 88.

Todo tipo de entrada ACL en el sistema CCSecurity necesita de una función que se encargue de verificar que una nueva ACL, formada a partir de una petición de acceso a recurso, no sea ya presente en el conjunto de aquellas del dominio solicitante. Según el resultado de esta comprobación, una diferente decisión será tomada por el supervisor, teniendo en cuenta el modo de operación asignado al dominio. La función *ccs_check_acl* se ocupa, pasándole la ACL que representa el acceso a recurso solicitado, de iterar sobre todo el conjunto de ACL del dominio solicitante, llamando a su vez *ccs_check_entry*. Esta última se encarga de llamar la función oportuna para contrastar la ACL solicitada con aquélla sobre la cuál se está iterando, para así verificar si ésta sea ya presente en la política. Se añade la llamada a la nueva función de verificación *ccs_check_android_binder_transaction_acl* como especificado por las líneas 112–115. La función, en este caso, solamente tendrá que verificar si la ACL sobre la cuál se está iterando representa una transacción al mismo servicio que el solicitado, y su dominio destino es igual o menos específico que el de la transacción en curso. En la página 18 se ha hecho referencia al hecho

[18]Tipo de estructura de C en la cual todos los miembros son alineados a la misma posición de memoria, permitiendo así la utilización exclusiva de uno solo de éstos; el tamaño final en memoria de esta estructura es mínimo, en cuanto éste corresponderá al tamaño del miembro más grande declarado

de que todos los dominios TOMOYO, si no especificado de forma diferente, llegan también a representar sus descendientes. El cuerpo de la función, que se encuentra a partir de la línea 185, se ocupa primero de comprobar la correspondencia entre dominios, y consecuentemente de verificar si el servicio solicitado es el mismo.

El código fuente contenido en el fichero *policy_io.c* es el responsable de la manipulación de las entradas ACL en la política. En él se añaden, borran y contrastan las ACL. En este fichero se implementa también el parseado de la sintaxis contenida en los ficheros de política. Para especificar y hacer efectiva la nueva sintaxis, primero será necesario establecer esta como perteneciente al grupo *ipc*, palabra clave con el cual empieza. Seguidamente especificamos la segunda palabra clave *android_binder_transaction* como perteneciente a una nueva categoría de ACL. Ambas modificaciones se pueden apreciar en las líneas 213–232. A continuación se debe especificar cual será la función responsable de procesar los datos crudos de una línea de política y rellenar la estructura de la ACL para la nueva sintaxis, líneas 265–267. La nueva función *ccs_write_android_transaction*, definida a partir de la línea 248, se ocupa del rellenado de la estructura precedentemente declarada, y de la inserción de la ACL en la política cargada en el sistema por medio de la llamada *ccs_update_acl*. Nótese que se hace uso de las funciones definidas por la componente de Garbage Collection para almacenar permanentemente en memoria los miembros de la estructura. Una vez instruido el sistema sobre como escribir una ACL utilizando la nueva sintaxis, se deberá proceder a la definición de los mecanismos para su impresión, interpretación y evaluación. Las líneas 275–283 indican como se ha implementado la impresión, a partir de la estructura rellenada de ACL, de la nueva sintaxis. Para la efectiva evaluación de la nueva ACL es necesario insertar una llamada a la función *ccs_supervisor*. Esta función es la que se encarga de la decisión final de un determinado acceso a recursos. Una vez formada una ACL correspondiente al acceso a recurso deseado, se invoca el supervisor para que decida, tomando en cuenta el modo de operación, si autorizar o denegar la operación. Esta función, además, es la que se encarga de añadir entradas a la política activa en el caso de que el modo *Learning* sea especificado por el perfil desde el cual origina el acceso a recurso. El conjunto de llamadas a la función ccs_supervisor se efectúa en la función responsable de generar los logs del mecanismo CCSecurity. En esta última función se añade apropiadamente la llamada al supervisor, líneas 291–296.

Última y más importante etapa del proceso, es la implementación de la función ___ccs_android_binder_permission, que se encuentra implementada a partir de la línea 133. Esta función es directamente llamada, como hook, por el código del Binder, y es la responsable de juntar todos los ante-

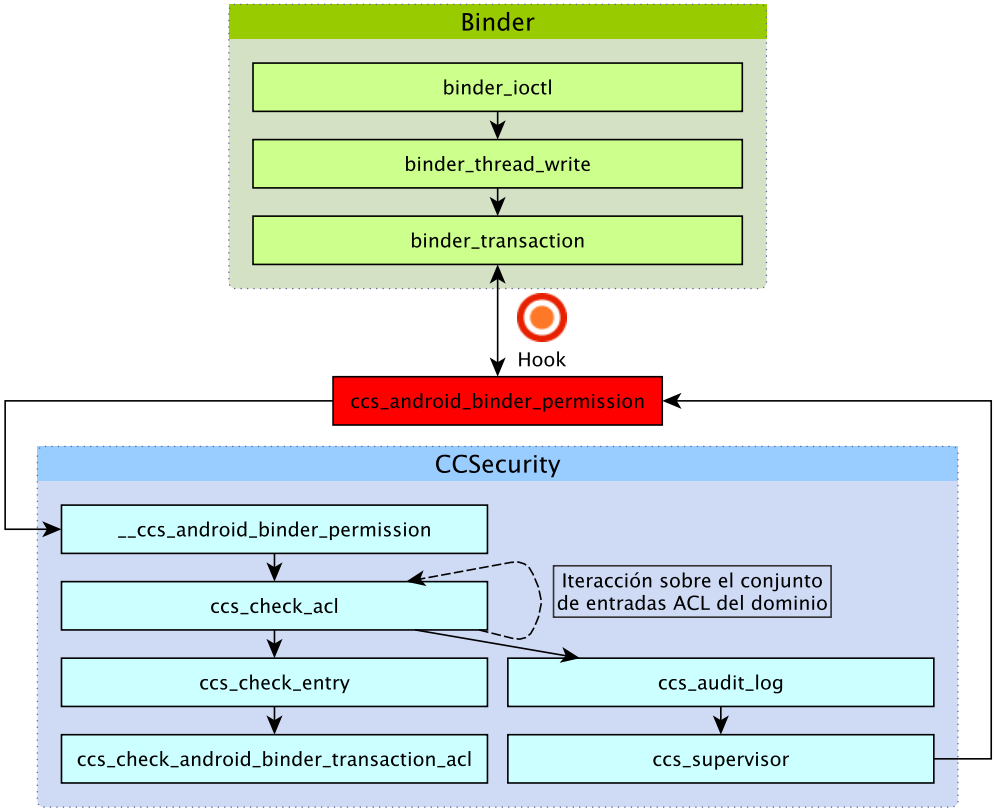


Figura 3.7: Flujo de ejecución a partir del hook en Binder IPC

riores elementos. Objetivo final de esta función es la llamada al mecanismo ccs_check_acl, el cual, después de haber comprobado la ACL, será el que finalmente llame al supervisor y tomará la decisión sobre al acceso que se está procesando. Para la llamada, es necesario primero rellenar una estructura que represente el acceso pedido, para así podérsela pasar a las funciones de comprobación ccs_check_acl. La función ccs_check_acl recibe, como parámetro, un puntero a una estructura de tipo *ccs_request_info*. Esta estructura es la responsable de transportar los datos relacionados con la petición de acceso a recurso para la consecuente formación de la ACL resultante.

Primer paso para el rellenado de la estructura es la obtención del dominio en el cual se encuentra el proceso destino de la transacción. Para la conversión del PID a una estructura *task_struct* se emplean algunas funciones del kernel Linux. La obtención del dominio se puede ver implementada en las líneas 147–164. Los PIDs menores o iguales que 1 no se consideran válidos: el PID 1 representa siempre el proceso *init*, el cual, evidentemente, no pue-

de aceptar transacciones Binder; el PID 0 cobra significado especial según el contexto y no puede así ser considerado un proceso concreto; los PIDs negativos especifican grupos de procesos, comportamiento que no se va a soportar en cuanto incompatible con el propósito del Binder. Como se puede observar en la implementación, la API ha sufrido algunos cambios después de la versión *2.6.24*, y algunas funciones ya no se encuentran exportadas. Para obviar a esta divergencia, se ponen algunas instrucciones de preprocesado para la correcta selección del código fuente adecuado para la versión para la cual se está compilando. Una vez extraído el dominio del proceso destino a través de la llamada *ccs_task_domain*, se procede a rellenar el miembro de la estructura, línea 167.

Segundo paso para el rellanado de la estructura ccs_request_info es averiguar como extraer el nombre del servicio invocado por la transacción desde la estructura binder_transaction_data. Es posible observar que la cadena representante el servicio se puede hallar con un desplazamiento de 8 bytes con respecto al elemento *data.ptr.buffer*. Se pueden incluso apreciar unos bytes nulos intercalados en la cadena.

```
Breakpoint 1, __ccs_android_binder_permission (pid=31,
    tr=0xdf33dc00) at security/ccsecurity/
   permission.c:2635
2635    static int __ccs_android_binder_permission(
   pid_t pid, struct binder_transaction_data *tr) {
(gdb) x/64cb tr->data.ptr.buffer
0x2a00e858:     0 '\000'        1 '\001'        0 '\
   000'        0 '\000'        26 '\032'       0 '\000
   '       0 '\000'        0 '\000'
0x2a00e860:     97 'a'  0 '\000'        110 'n' 0 '\
   000'        100 'd' 0 '\000'        114 'r' 0 '\000
   '
0x2a00e868:     111 'o' 0 '\000'        105 'i' 0 '\
   000'        100 'd' 0 '\000'        46 '.'  0 '\000
   '
0x2a00e870:     111 'o' 0 '\000'        115 's' 0 '\
   000'        46 '.'  0 '\000'        73 'I'  0 '\000
   '
0x2a00e878:     83 'S'  0 '\000'        101 'e' 0 '\
   000'        114 'r' 0 '\000'        118 'v' 0 '\000
   '
0x2a00e880:     105 'i' 0 '\000'        99 'c'  0 '\
   000'        101 'e' 0 '\000'        77 'M'  0 '\000
```

```
   ’
0x2a00e888:      97 ’a’  0 ’\000’         110 ’n’ 0 ’\
   000’         97 ’a’  0 ’\000’         103 ’g’ 0 ’\000
   ’
0x2a00e890:      101 ’e’ 0 ’\000’         114 ’r’ 0 ’\
   000’         0 ’\000’         0 ’\000’         0 ’\000
   ’         0 ’\000’
```

Se ha comprobado que este desplazamiento es constante entre diferentes invocaciones RPC. En la líneas 169–177 se procede a la allocación de un buffer temporal y a su rellenado, teniendo en cuenta el desplazamiento y los bytes nulos intercalados.

Teniendo rellenada, de este modo, la estructura necesaria, comprobamos que el dominio destino no sea el mismo del proceso invocante, en el cual caso se permite siempre la transacción, y finalmente se procede a llamar la función ccs_check_acl.

3.3.6. Extensión ccs-tools

Última etapa del proceso de adaptación es la modificación de las herramientas usuario para soportar e interactuar con la nueva sintaxis. De las herramientas utilizadas, solo será necesario modificar el código relacionado con ccs-editpolicy. La modificaciones que se muestran en el parche C.1 tienen muchas semejanzas con lo anterior. Algunas estructuras de las herramientas usuario son realmente copias de las utilizadas en el lado kernel, por conformidad de implementación. Como se puede observar en la línea 107, la función *ccs_write_android_transaction*, similarmente a su homónima en lado kernel, se ocupa de rellenar la estructura de la ACL que representa la transacción a través del Binder. Se puede observar el registro de esta función para tratar la nueva sintaxis en la línea 129. ccs-editpolicy, al ser una herramienta que sirve para editar la política CCSecurity en uso en el núcleo, cuenta con un conjunto de funciones que sirven para evitar que el usuario inserte una ACL que ya esté presente en la política. En el caso de las transacciones Binder, la función responsable de esta comprobación es *ccs_same_android_transaction_acl* que puede verse implementada en las líneas 92–98. Como se puede observar, la comprobación se efectúa solamente comparando el valor de los punteros miembros de las respectivas estructuras. Es posible hacer esto porqué las herramientas usuario, aunque no cuenten con el algoritmo de Garbage Collection implementado en el lado kernel, siguen el mismo patrón de allocación que seguido por el kernel. Hay funciones que se dedican a alocar memoria, según el tipo y el caso, y a insertar estos enlaces

en listas enlazadas circulares, y, con una función de hash, se aseguran que no hayan copias repetidas del mismo elemento. Así pues se puede comprobar la igualdad de dos cadenas alocadas por medio de las funciones internas simplemente comprobando el puntero devuelto por esas funciones, ya que serán estas a devolver un puntero al mismo elemento en caso de alocaciónes repetidas.

Los cambios efectuados en la función *ccs_tokenize* tienen especial importancia, líneas 149–167. Esta función se encarga de separar y validar las distintas componentes que forman una entrada de política, la cual, evidentemente, representa una ACL. Según el tipo de línea la función deberá validar ciertos campos como dominios, señales, etc. En el caso de la sintaxis a soporte de las transacciones Binder, se tienen solo dos campos de datos, que representan respectivamente el servicio invocado, una simple cadena de caracteres sin espacios, y el dominio destino. La entrada *ipc signal* tiene una estructura similar, así pues se extiende consecuentemente la función para soportar la sintaxis de transacciones Android.

Finalmente, las modificaciones a la función *ccs_editpolicy_optimize* permiten la optimización de las entradas de la nueva sintaxis, líneas 172–183. Esta función, de hecho, pretende optimizar la política intentando unificar entradas redundantes. Para ello, cada sintaxis deberá especificar la forma por la cual una entrada puede realmente ser asociada e incluida en otra. En el caso de las transacciones Android se deberá comprobar primero el servicio y en segundo lugar el dominio.

3.3.7. Resultados y Limitaciones

Recompilando el kernel y las herramientas usuario incluyendo todas las anteriores modificaciones se pueden apreciar los resultados de las mismas. A diferencia de la plataforma original, las aplicaciones ya no se encuentran colapsadas en un solo dominio, sino que se ven repartidas adecuadamente según el nombre de sus paquetes Java. Al lanzar la herramienta ccs-editpolicy con el nuevo kernel, se puede observar enseguida que las aplicaciones se encuentran correctamente repartidas en sus correspondientes dominios, captura 3.8.

La implicación evidente de este hecho es la correcta repartición de los accesos. Contrariamente a la implementación original, que no era capaz de dividir las aplicaciones usuario en sus correspondientes dominios, el estado actual de TOMOYO CCSecurity es así capaz de asignar las entradas ACL consecuentemente a la aplicación que ha efectuado el acceso a recurso.

El segundo gran efecto de las modificaciones es la correcta interpretación de las transacciones efectuadas a través del Binder, como muestra la captura 3.9. Este hecho, evidentemente, permite un análisis del sistema mucho más

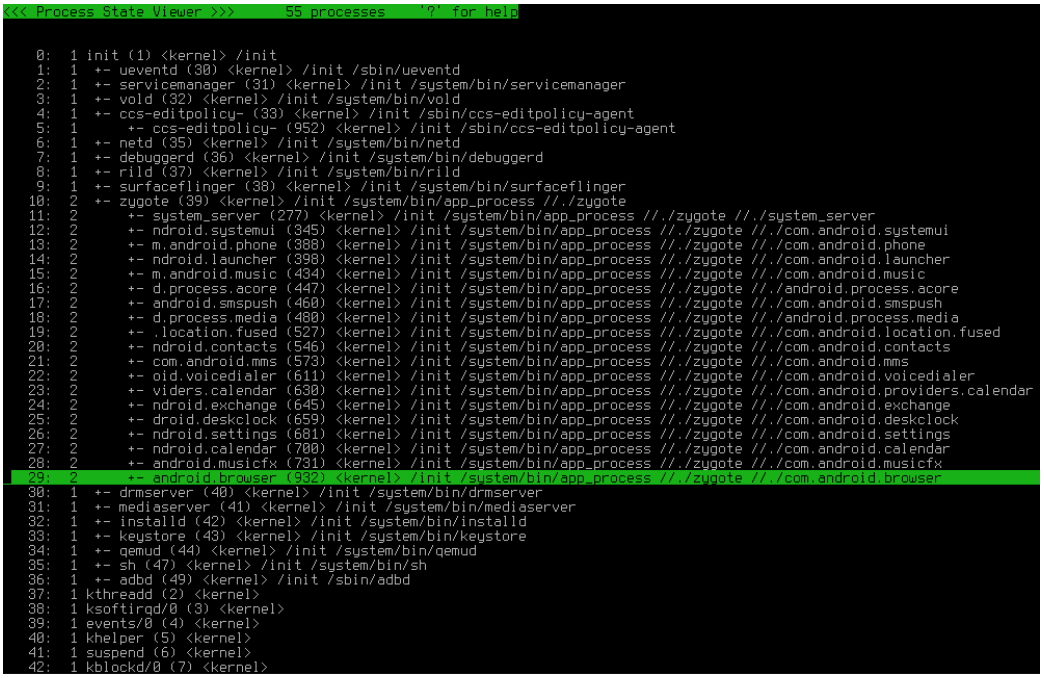


Figura 3.8: ccs-editpolicy, aplicaciones usuario repartidas correctamente

concreta y precisa.

Al igual que cualquier otra ACL definida en el mecanismo CCSecurity, la nueva sintaxis puede ser también utilizada en modo *Enforcing* (ver página 21), para así poder permitir solamente determinadas invocaciones RPC. Como ejemplo académico, se quiere denegar a la aplicación del teclado virtual la posibilidad de enviar las teclas pulsadas al navegador Android. Para ello se ejecuta la aplicación del navegador de Android y se empieza componiendo la cadena “hola ” en uno de los campos de texto disponibles. Ya que los dominios de las aplicaciones usuario se encuentran en modo *Learning*,

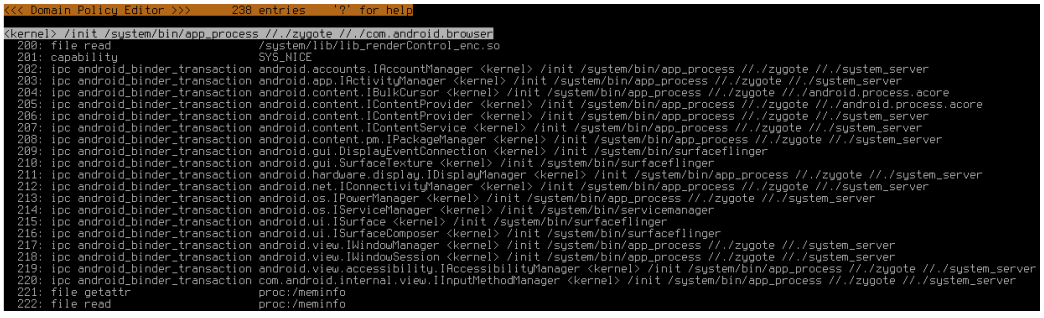


Figura 3.9: ccs-editpolicy, entradas ACL de las transacciones Binder

la entrada ACL correspondiente al envío de teclas se verá creada en el dominio de la aplicación teclado. Para interrumpir el flujo de teclas se decide entonces borrar la entrada relativa. Para ello se accede, utilizando la herramienta ccs-editpolicy, al dominio *<kernel> /init /system/bin/app_process //./zygote //./com.android.inputmethod.latin*, que contiene la aplicación del teclado virtual, y se verifica la presencia de la siguiente ACL:

```
ipc android_binder_transaction com.android.internal.
   view.IInputContext <kernel> /init /system/bin/
   app_process //./zygote //./com.android.browser
```

Como especificado por los JavaDoc de la plataforma Android, este servicio es la interfaz de un método de input a otra aplicación, permitiendo modificaciones del campo de texto activo y otras interacciones. En este caso el servicio se invoca en el contexto de la aplicación navegador. Se asigna el dominio a un perfil en modo Enforcing y se borra la entrada ACL. Como era de esperar, ahora, al pulsar las teclas en el teclado virtual, no se tiene ningún efecto en el campo de texto del navegador, y no es posible completar la cadena anterior con "mundo".

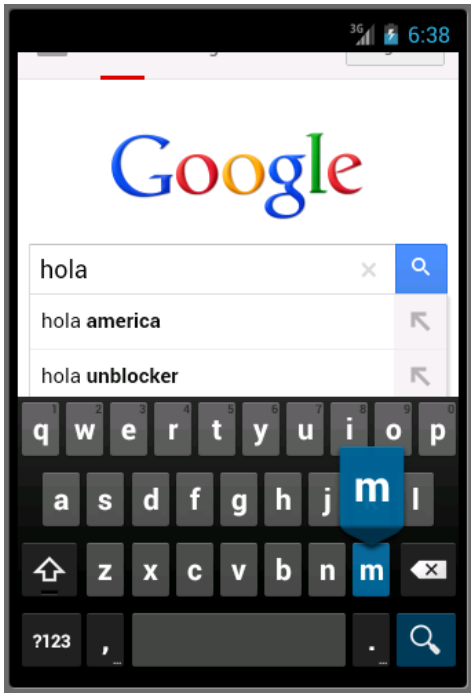


Figura 3.10: el navegador ya no recibe el input del teclado virtual

El kernel, consecuentemente a la implementación de la modificación, dispara el siguiente error:

```
binder: 381:381 BR_TRANSACTION -> 770 - node 12890 was
    denied by ccs policy
binder: 381:381 transaction failed 29201, size88-0
```

Efectivamente, el PID 381 representa correctamente la instancia de la máquina virtual Dalvik que está ejecutando la aplicación de teclado, y el PID 770 representa la aplicación navegador.

Desafortunadamente, Android no está diseñado para soportar un mecanismo de acceso MAC, y muchas aplicaciones tienen comportamientos indefinidos cuando se les ve denegado el acceso a algún servicio. Sin embargo, la nueva versión *4.3* de la plataforma Android empezará a soportar el mecanismo MAC SELinux [1]. Se considera que será posible, aprovechando los cambios de diseño aportados a la plataforma para el soporte de SELinux, sustituir este mecanismo por TOMOYO, y así usufructuar de todas las características únicas que este sistema ofrece. Aunque no puedan ofrecer tanta funcionalidad, existen, en cambio, aplicaciones usuario capaces de restringir y hasta falsificar el acceso a recursos. La más popular se llama *XPrivacy* [34, 15]. Esta herramienta funciona por encima de un framework llamado *Xposed* [33, 14] que, aportando una modificación al ejecutable app_process, permite la carga de un archivo jar en toda máquina virtual arrancada por el sistema. Este jar contiene un conjunto de funciones que, aprovechándose de la funcionalidad Java JNI[19], permite sustituir las interfaces de los servicios ofrecidos por las aplicaciones. Esta funcionalidad es utilizada por la aplicación XPrivacy para tener control sobre los servicios a los que las aplicaciones tienen derecho a acceder. XPrivacy es hasta capaz de emular algunos servicios, restituyendo determinadas respuestas a las aplicaciones — como coordenadas GPS fijas o aleatorias —. Sin embargo, una aplicación usuario no puede ofrecer las funcionalidades y la flexibilidad de un completo sistema MAC implementado a nivel kernel.

Otra limitación de la actual implementación del mecanismo CCSecurity es la disección solo parcial de las transacciones Binder IPC, extrayendo solo el servicio invocado. Sin embargo, será sin duda posible extender más la implementación para llegar a disectar los parámetros de las invocaciones, y poder así tener un control todavía más fino sobre las transacciones entre aplicaciones y servicios RPC.

A pesar de las actuales limitaciones, estas modificaciones hacen de TOMOYO una herramienta insustituible para el estudio de la plataforma Android.

[19] Java Native Interface: framework que permite al código Java de interactuar con código nativo, sea este C, C++ o Assembly

Capítulo 4

Visualización Mediante Grafos

4.1. Procedimientos

Uno de los posibles ejemplos de uso de esta potente herramienta de estudio de la plataforma Android es la creación de grafos que representen las interacciones entre las diferentes componentes RPC ofrecidas por las aplicaciones. Este estudio habría sido imposible utilizando otras herramientas, en cuanto ninguna hubiera sido capaz de registrar con precisión la multitud de peticiones que el sistema genera. Para la obtención de un cuadro completo de la plataforma, se decide asignar todo dominio a un perfil en modo *Learning*. Se han activado, además, todos los parámetros necesarios para el registro de todos los accesos ocurridos a través del Binder en ficheros de log.

Para la representación de la enorme cantidad de datos se decide utilizar una librería Java llamada *GraphStream* [16]. Esta librería permite la representación de grafos dinámicos en espacios tanto 2D como 3D, ofreciendo incluso la posibilidad de aplicar una multitud de algoritmos diferentes, representando los resultados de éstos por medio de alteraciones sobre la apariencia de los elementos del grafo, definida a través de ficheros CSS[1]. Sin embargo, Java no es el lenguaje más apropiado para el procesamiento de las grandes cantidades de texto que los ficheros de log ofrecen. Por lo tanto se ha decidido dividir el procesamiento de los grafos en dos grandes componentes. Por un lado se ejecutará el programa Java que utiliza la librería GraphStream y que será encargado del procesamiento gráfico de los elementos del grafo. Por el otro estarán: el emulador Android que ejecutará la plataforma con funcionalidad CCSecurity, la herramienta ccs-auditd en ejecución para la obtención de los ficheros de log, y un script *Ruby* que se ocupará de procesarlos adecuadamente e instruir finalmente el programa Java encargado de la renderización de

[1]Cascading Style Sheets, hojas de estilo

los grafos. El programa Java, de hecho, ofrecerá una interfaz HTTP para la interacción con el grafo que se está generando. La interfaz HTTP permitirá al script Ruby añadir nodos y aristas al grafo dinámico, y ejecutar algoritmos a través de simples peticiones HTTP. De este modo la renderización del grafo y ejecución de los algoritmos pueden tener lugar en una máquina diferente a la cual se está realmente ejecutando la plataforma Android. La carga computacional puede así ser repartida según las necesidades.

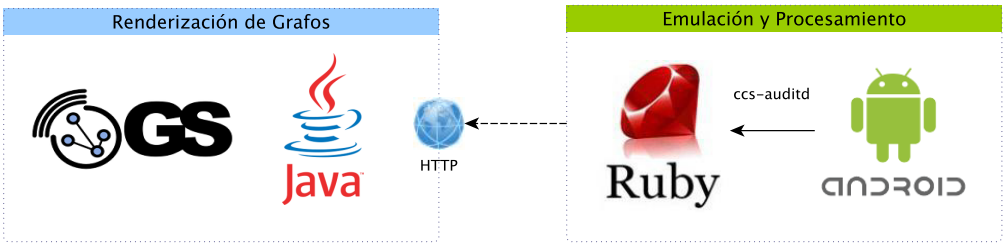


Figura 4.1: Flujo para la representación de los datos en grafo

La librería GraphStream ofrece ya una clase para la obtención de los datos para la generación de grafos por medio de HTTP. La clase se llama *HTTPSource*. Desafortunadamente, esta librería es considerablemente joven, y sufre todavía la presencia de muchas funcionalidades experimentales. Éste es el caso de HTTPSource. La clase implementada, de hecho, es capaz de gestionar solo algunas peticiones, no implementando por ejemplo la asignación de etiquetas en los elementos presentes en el grafo. Para solucionar estas graves limitaciones se ha decidido extender la clase HTTPSource para adaptarse a los objetivos prefijados. La nueva clase implementada *HTTPSourceExtended* extiende el comportamiento de la original para soportar la asignación de cualquier tipo de atributo a los elementos del grafo, y para invocar métodos que desencadenan la computación de determinados algoritmos sobre el mismo.

Por el otro lado, en la máquina que hospeda la emulación de la plataforma Android, el script Ruby se dedica al parseado de los ficheros de log para obtener las líneas que representen transacciones a través del dispositivo Binder IPC, y para instruir consecuentemente la renderización del grafo correspondiente. Se ha decidido representar las aplicaciones Android como nodos; los servicios que las aplicaciones ofrecen también se representan como nodos, que son conectados a éstas por medio de aristas dirigidas. De este modo, los servicios ofrecidos por las aplicaciones serán siempre las hojas del grafo resultante. Las transacciones entre una aplicación y un servicio que otra ofrece se representan con una arista entre la aplicación invocante y la invocada. Para evidenciar y subrayar la frecuencia y cantidad de los accesos,

las aristas se han dotado de un atributo y de una etiqueta que representan el peso de la misma. Por cada acceso a un servicio por parte de una aplicación, se incrementa, en el caso la arista ya sea presente, el peso de la arista que conecta la aplicación invocante a la invocada; consecuentemente, se incrementa el peso de la arista que conecta la aplicación invocada con el servicio interno que ofrece. Este atributo será de gran importancia para la computación de determinados algoritmos. Se podrán detectar los puntos más críticos y centrales de la plataforma Android, pudiendo así individuar las componentes y servicios en los que se concentra mayormente la carga de transacciones del sistema.

4.2. Ejemplo

Para la obtención de un grafo básico, siguiendo el procedimiento anteriormente descrito, se ha emulado la plataforma Android durante algunos minutos simulando un comportamiento normal por parte de un usuario, lanzando e interactuando con diferentes aplicaciones del sistema. El resultado del procesamiento de los datos obtenidos se puede apreciar en la figura contenida en el apéndice D.1. El grafo representa con nodos verdes las diferentes aplicaciones que han llegado a ejecutarse en el sistema. Desde ellas se extienden algunos nodos amarillos que representan los servicios para los que se ha recibido al menos una transacción por parte de otra aplicación. A todas la aristas presentes en el grafo se ve asociada una etiqueta naranja que indica la cantidad de veces que ésta se ha recorrido. La disposición de los elementos del grafo se ha formado con las siguientes reglas: cada nodo en el grafo ejerce una fuerza de repulsión contra cualquier otro nodo inversamente proporcional a su distancia con el mismo; esta fuerza se ve contrastada por las aristas que, a cambio, ejercen una fuerza de atracción entre los nodos conectados. Dadas estas reglas, se concluye que los nodos altamente conectados tenderán a situarse en el centro del grafo, en cuanto las fuerzas de las aristas provenientes de los muchos otros nodos, a los que se conectan, los atraerán cerca de ellos. La posición en la cuál las fuerzas de las aristas se ven más satisfechas es justamente en el centro del grafo, siendo esta también el centro de una circunferencia ideal formada por los otros nodos.

Como se puede claramente observar por la disposición tomada por las aplicaciones, los nodos en los que se centra la mayoría de conexiones son las aplicaciones de sistema *servicemanager* y *system_server*. Ambas, efectivamente, ofrecen una multitud de servicios fundamentales para la correcta ejecución de las otras aplicaciones.

A pesar de la cantidad de conexiones que inciden en un determinado no-

do, es interesante individuar cuáles son los nodos que ocupan un rol central en el juego de interconexiones del entero grafo. Se ha aplicado un algoritmo de centralidad al grafo anterior, y el resultado se ve representado en la figura contenida en el apéndice D.2. El algoritmo utilizado se denomina *Betweenness Centrality* [2]. El algoritmo cuenta cuantos caminos mínimos entre cada par de nodos pasan por un determinado nodo. Este proceso se repite por cada nodo presente en el grafo. Los nodos que representan la aplicaciones se han representado con un gradiente que va del verde al rojo. Cuanto más un nodo es central tanto más su color difuminará tendiendo al rojo. Contrariamente a la cantidad de conexiones analizada anteriormente, el nodo servicemanager no es un nodo central en la plataforma. Esto se debe a la escasez de servicios RPC que esta aplicación ofrece. De hecho, solamente ofrece el servicio *IServiceManager*. A pesar de las numerosas transacciones a este servicio efectuadas por muchas aplicaciones, el algoritmo tiene en cuenta la cantidad de aristas salientes de un nodo — es decir, las transacciones efectuadas y la cantidad de servicios ofrecidos — sin prestar atención al peso de las mismas. A cambio, la aplicación system_server sí que se detecta como central, gracias a la multitud de servicios que ofrece. Se puede también observar que otras aplicaciones como *phone*, *surfaceflinger* (rendering de vídeo), *latin* (teclado) y *mediaserver* (servidor audio), a pesar del numero limitado de servicios que ofrecen, interactúan de forma muy extensa con muchas otras componentes, y, por ello, son detectadas como centrales por parte del algoritmo. Dadas las funciones que estas aplicaciones desempeñan, no sorprende que tengan un rol importante para todas las otras aplicaciones, y en consecuencia sean detectadas como centrales.

A continuación se decide resaltar, a partir del grafo anterior, el peso de las aristas. Como detallado anteriormente, el peso de una arista representa la cantidad de veces que ésta se ha visto recorrida. Esto es, para los servicios, la cantidad de veces que éstos han sido invocados por cualquier otra aplicación durante el tiempo de emulación de la plataforma. Se considera importante, de este modo, evidenciar aquellas transacciones o servicios que sufren mayor estrés en la plataforma. Se ha decidido alterar la apariencia gráfica de las aristas modificando su color y grosor dependiendo de su peso. Cuanto más el peso de una arista es elevado, tanto más aumentará su grosor, y su color, similarmente, tenderá a rojo. El grafo, enteramente procesado, se puede observar en el apéndice D.3. Como se puede apreciar por el agrandamiento de la figura 4.2, las aristas de mayor peso se concentran alrededor de los nodos system_server y surfaceflinger.

Se puede notar la gran utilización que la plataforma hace de los servicios ofrecidos por surfaceflinger. Estos servicios son de hecho los responsables del compositing de la salida vídeo de la plataforma, así como de la captura

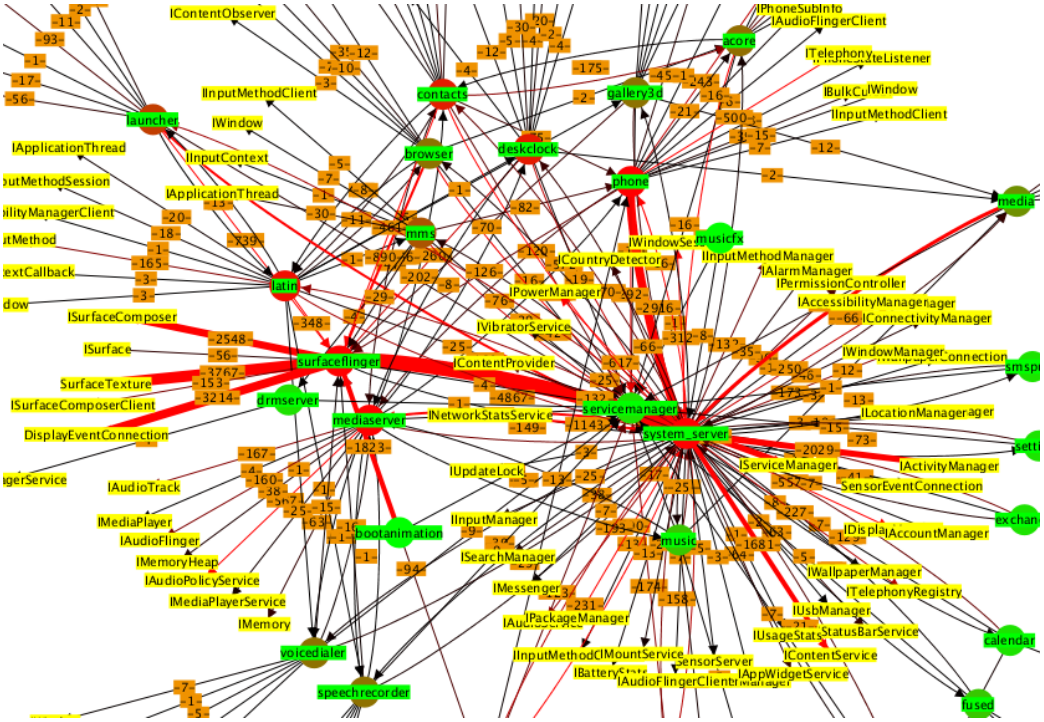


Figura 4.2: Detalle de aristas ponderadas

de los eventos de la pantalla táctil. Todas las aplicaciones van a necesitar masivamente de estas funcionalidades. Se puede notar, además, que existe una gran interacción entre esta componente y la otra aplicación de sistema system_server. Esto se debe probablemente a que muchos servicios de la plataforma, que system_server maneja y coordina, requieren el acceso a las funcionalidades de la pantalla. En la ampliación se puede observar también una fuerte interacción entre el system_server y la aplicación de teléfono phone. La interacción es debida a que esta aplicación tiene mucha prioridad en la plataforma, y la aplicación coordinadora system_server necesita comprobar a menudo el estado de la misma. En el conjunto de servicios ofrecidos por system_server se puede apreciar una carga mayor en los servicios *IActivityManager*, responsable del manejo y comunicación entre las actividades de las diferentes aplicaciones, *IServiceManager*, encargado de gestionar los servicios ofrecidos a lo largo de la plataforma tanto a nivel de transacciones como a nivel de permisos, y *IContentService*, interfaz dedicada a ofrecer el intercambio de diferentes tipos de contenidos a las aplicaciones.

Para enfatizar aún más el estrés sobre las componentes de la plataforma, se ha decidido aplicar un algoritmo *Spanning Tree* de *Prim* [23] sobre el grafo anterior. Se han utilizado los pesos con significado invertido, es decir que el

algoritmo considerará preferibles caminos con pesos mayores. Las aristas no pertenecientes al árbol resultante se difuminan al color gris. El resultado se puede apreciar en el apéndice D.4. De este modo se ven resaltados los caminos en los que se expresa la mayor carga de transacciones a través de la capa de middleware de la plataforma Android.

Capítulo 5

Conclusiones y Trabajo Futuro

Gracias a la adaptación de TOMOYO a la plataforma Android, se dispone de un framework para el control de los servicios utilizados por las aplicaciones durante su ejecución. Se puede apreciar el contraste con la ejecución de la figura 1.2 en la página 6.

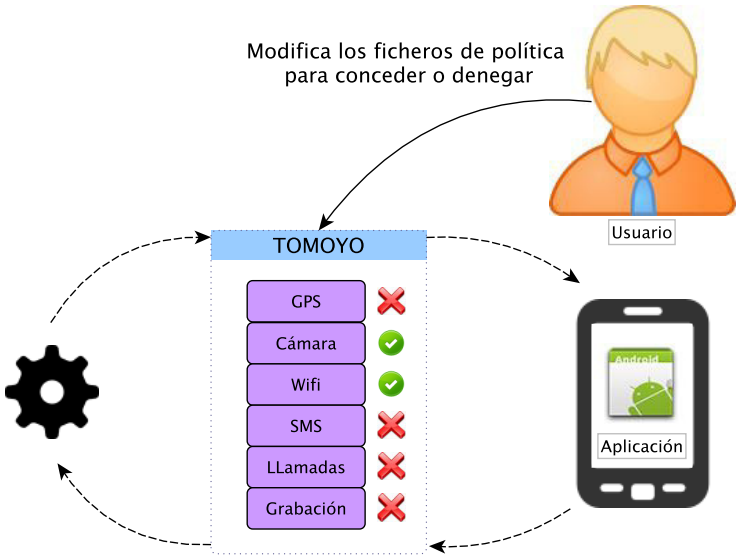


Figura 5.1: El usuario tiene el control de los servicios durante la ejecución

El usuario, gracias a TOMOYO, puede llegar a administrar los servicios y recursos que una aplicación tiene derecho a utilizar. Cualquier aplicación puede así ser instalada con todos los permisos que requiere en su manifiesto. Solo en fase de ejecución TOMOYO permitirá estudiar y denegar el acceso a servicios sensibles. TOMOYO actúa como una capa que se interpone en las

interacciones entre los diferentes elementos que forman el sistema. Esta capa permite observar y filtrar estas interacciones, denegando el acceso a servicios y datos.

El mecanismo TOMOYO Linux, así adaptado para la plataforma Android, constituye una herramienta de gran utilidad para muchos proyectos, ofreciendo una cantidad muy extensa de posibles enfoques. Se ha visto que el mecanismo es capaz de analizar detalladamente el sistema, dando la oportunidad de tener una visión diferente de la plataforma. Cualquier otro mecanismo sería insuficiente para obtener una visión similar. Cualquier aspecto de interacción entre el dispositivo y el sistema operativo puede ser analizado y restringido con profundidad; no es casual que la última versión de Android, la 4.3, haya empezado a soportar un sistema con objetivos similares: SELinux. SELinux difiere todavía en muchos aspectos de TOMOYO. SELinux es una herramienta más compleja, pero no necesariamente más potente y flexible. TOMOYO, de hecho, ha sido concebido principalmente como herramienta de estudio, y solo posteriormente refinada para acabar como mecanismo de protección MAC. En contraste con otros mecanismos MAC, entre los cuales SELinux, TOMOYO no dispone de perfiles predefinidos, sino que es capaz de adaptarse finemente al sistema en el cual se encuentra en ejecución ya que dispone de un modo de aprendizaje. Gracias a éste, TOMOYO, es probablemente el mecanismo más apropiado para plataformas embedded, en cuanto facilita de forma substancial el análisis previo a la implantación de una política de seguridad. Dadas estas características de diseño, TOMOYO es una herramienta de análisis de sistema operativo Linux que no conoce rivales. Se considera así que las aportaciones de esta adaptación puedan ser de gran interés.

A pesar de las limitaciones que el porting, con la adaptación realizada, presenta a la hora de denegar acceso a los servicios ofrecidos por medio de la capa de middleware representada por el componente Binder, se considera que será posible aprovechar los cambios introducidos por la nueva versión de la plataforma Android. La versión 4.3 de Android, empezando oficialmente a soportar el mecanismo SELinux, ofrece las modificaciones estructurales necesarias para la completa y correcta integración del mecanismo TOMOYO, de forma similar a como se ha hecho para SELinux. TOMOYO ofrecerá así una potente alternativa a SELinux para todo tipo de dispositivo que funcione con la plataforma Android.

En cuanto a los aspectos técnicos de la actual implementación de la adaptación de TOMOYO a Android, se podría notablemente mejorar la disección de las transacciones Binder. La estructura interna utilizada por el driver del Binder guarda, en su interior, toda la información perteneciente a la transacción. Podría así ser objetivo de futuras ampliaciones la implementación de un

disector de parámetros en colaboración con componentes Java y C. Se podría además, de este modo, ofrecer al usuario final el control sobre los permisos, y, por medio de *Java Native Interface*, dejarlo interactuar cómodamente, gracias a una aplicación usuario, con el entero sistema TOMOYO que se esté ejecutando a nivel kernel.

En el futuro, se plantea además refinar y ampliar oportunamente las funcionalidades de la adaptación. Una vez que el código esté lo suficientemente maduro y testeado, se piensa contribuir oficialmente al proyecto TOMOYO Linux [31] por medio de una propuesta de inclusión en el árbol oficial de desarrollo, ofreciendo una base para un soporte completo de la plataforma Android.

Apéndice A

Política Inicial

A.1. domain_policy.conf

```
<kernel>
use_profile 1
use_group 0

<kernel> /init /system/bin/app_process
use_profile 2
use_group 1
```

A.2. exception_policy.conf

```
path_group ANY_PATHNAME /
path_group ANY_PATHNAME /\{\*\}/
path_group ANY_PATHNAME /\{\*\}/\*
path_group ANY_PATHNAME /\*
path_group ANY_PATHNAME \*:/
path_group ANY_PATHNAME \*:/\{\*\}/
path_group ANY_PATHNAME \*:/\{\*\}/\*
path_group ANY_PATHNAME \*:/\*
path_group ANY_PATHNAME \*:[\$]
path_group ANY_PATHNAME socket:[family=\$:type=\$:protocol=\$]
#acl_group 0 file ioctl @ANY_PATHNAME 0-0xFFFFFFFF
acl_group 0 file getattr @ANY_PATHNAME
acl_group 0 file read /dev/urandom
acl_group 0 file read /system/bin/linker
acl_group 0 file read /system/lib/lib\*.so
acl_group 0 misc env _
acl_group 0 misc env ANDROID_ASSETS
acl_group 0 misc env ANDROID_BOOTLOGO
acl_group 0 misc env ANDROID_DATA
acl_group 0 misc env ANDROID_DNS_MODE
acl_group 0 misc env ANDROID_PROPERTY_WORKSPACE
acl_group 0 misc env ANDROID_ROOT
acl_group 0 misc env ANDROID_SOCKET_\*
acl_group 0 misc env ASEC_MOUNTPOINT
acl_group 0 misc env BOOTCLASSPATH
acl_group 0 misc env EXTERNAL_STORAGE
acl_group 0 misc env HOME
```

```
acl_group 0 misc env LD_LIBRARY_PATH
acl_group 0 misc env LOOP_MOUNTPOINT
acl_group 0 misc env ndns
acl_group 0 misc env PATH
acl_group 0 misc env qemu
acl_group 0 misc env RANDOM
acl_group 0 misc env SHELL
acl_group 0 misc env TERM
```

A.3. manager.conf

```
/sbin/ccs-editpolicy-agent
```

A.4. profile.conf

```
1-COMMENT=-----Learning Mode-----
1-PREFERENCE={ max_audit_log=1024 max_learning_entry=2048 enforcing_penalty=0 }
1-CONFIG={ mode=learning grant_log=no reject_log=no }
2-COMMENT=-----Learning Mode con mucho log-----
2-PREFERENCE={ max_audit_log=20480 max_learning_entry=20480 enforcing_penalty=0 }
2-CONFIG={ mode=learning grant_log=yes reject_log=yes }
3-COMMENT=-----Enforcing Mode-----
3-PREFERENCE={ max_audit_log=1024 max_learning_entry=2048 enforcing_penalty=0 }
3-CONFIG={ mode=enforcing grant_log=no reject_log=no }
```

A.5. stat.conf

Apéndice B

Parches Kernel Linux

B.1. CCSecurity

B.1.1. Kconfig

```
diff --git a/security/ccsecurity/Kconfig b/security/ccsecurity/Kconfig
index 45857ab..9cde316 100644
--- a/security/ccsecurity/Kconfig
+++ b/security/ccsecurity/Kconfig
@@ -154,6 +154,15 @@ config CCSECURITY_IPC
          Say Y here if you want to enable analysis/restriction of sending
          signals.

+config CCSECURITY_ANDROID
+       bool "Enable Android specific extensions."
+       default y
+       depends on ANDROID && CCSECURITY && CCSECURITY_IPC && ANDROID_BINDER_IPC
+       ---help---
+         Say Y here if you want to enable analysis/restriction of Android
+         specific behaviour.
+         This option also extends control over the binder driver middleware.
+
 config CCSECURITY_MISC
        bool "Enable environment variable names restriction."
        default y
```

B.1.2. Especialización de dominio app_process

```
diff --git a/security/ccsecurity/permission.c b/security/ccsecurity/permission.c
index ff4ad0d..932afca 100644
--- a/security/ccsecurity/permission.c
+++ b/security/ccsecurity/permission.c
@@ -2567,6 +2567,61 @@ out:
        return error;
 }

+#ifdef CONFIG_CCSECURITY_ANDROID
+#define CCS_ANDROID_APP_PROCESS_DOM "<kernel> /init /system/bin/app_process"
+#define CCS_ANDROID_ZYGOTE_DOM CCS_ANDROID_APP_PROCESS_DOM " //./zygote"
+
+static int ccs_current_cmdline(char **buf) {
```

```
+	struct mm_struct *mm;
+	int len = 0;
+	mm = get_task_mm(current);
+	if (mm) {
+		len = mm->arg_end - mm->arg_start;
+		if (len > PAGE_SIZE)
+			len = PAGE_SIZE;
+		*buf = kmalloc(len, CCS_GFP_FLAGS);
+		access_process_vm(current, mm->arg_start, *buf, len, 0);
+	}
+	return len;
+}
+
+static bool ccs_android_specialize_app(const struct ccs_path_info *domname) {
+	char *buf, *cmdline = NULL;
+	int len;
+	bool success = false;
+	len = ccs_current_cmdline(&cmdline);
+	if (!len)
+		return success;
+	len += domname->total_len;
+	if (len > PAGE_SIZE)
+		len = PAGE_SIZE;
+	buf = kmalloc(len, CCS_GFP_FLAGS);
+	snprintf(buf, len, "%s //./%s", domname->name, cmdline);
+	if (!ccs_assign_domain(buf, true))
+		ccs_transition_failed(buf);
+	else
+		success = true;
+	kfree(cmdline);
+	kfree(buf);
+	return success;
+}
+
+static void ccs_transit_android_app_process(void) {
+	char *cmdline = NULL;
+	const struct ccs_path_info *domname;
+	domname = ccs_current_domain()->domainname;
+	if (!strcmp(domname->name, CCS_ANDROID_APP_PROCESS_DOM))
+		ccs_android_specialize_app(domname);
+	else if (!strcmp(domname->name, CCS_ANDROID_ZYGOTE_DOM))
+		if (ccs_current_cmdline(&cmdline)) {
+			if (strcmp(cmdline, "zygote") && strcmp(cmdline, "<pre-
    initialized>"))
+				ccs_android_specialize_app(domname);
+			kfree(cmdline);
+		}
+}
+
+#endif
+
 /**
  * __ccs_ioctl_permission - Check permission for "ioctl".
  *
@@ -2579,6 +2634,14 @@ out:
 static int __ccs_ioctl_permission(struct file *filp, unsigned int cmd,
 				  unsigned long arg)
 {
+#ifdef CONFIG_CCSECURITY_ANDROID
+	/**
+	 * Android forked apps from zygote process make an early
+	 * ioctl to binder device. This is the time for assigning
```

```
+         * a new specialized domain for the process.
+         **/
+       ccs_transit_android_app_process();
+#endif
        return ccs_path_number_perm(CCS_TYPE_IOCTL, filp->f_dentry,
                                    filp->f_vfsmnt, cmd);
 }
```

B.1.3. Dissección de transacciónes

```
diff --git a/security/ccsecurity/gc.c b/security/ccsecurity/gc.c
index d609420..1762e61 100644
--- a/security/ccsecurity/gc.c
+++ b/security/ccsecurity/gc.c
@@ -141,6 +141,9 @@ static void ccs_memory_free(const void *ptr, const enum
    ccs_policy_id type)
 #endif
 #ifdef CONFIG_CCSECURITY_IPC
                [CCS_TYPE_SIGNAL_ACL] = sizeof(struct ccs_signal_acl),
+#ifdef CONFIG_CCSECURITY_ANDROID
+               [CCS_TYPE_ANDROID_TRANSACTION_ACL] = sizeof(struct
    ccs_android_transaction_acl),
+#endif
 #endif
 #ifdef CONFIG_CCSECURITY_TASK_EXECUTE_HANDLER
                [CCS_TYPE_AUTO_EXECUTE_HANDLER]
@@ -466,6 +469,16 @@ void ccs_del_acl(struct list_head *element)
                        ccs_put_name(entry->domainname);
                }
                break;
+#ifdef CONFIG_CCSECURITY_ANDROID
+       case CCS_TYPE_ANDROID_TRANSACTION_ACL:
+               {
+                       struct ccs_android_transaction_acl *entry =
+                               container_of(acl, typeof(*entry), head);
+                       ccs_put_name(entry->service);
+                       ccs_put_name(entry->domainname);
+               }
+               break;
+#endif
 #endif
 #ifdef CONFIG_CCSECURITY_TASK_EXECUTE_HANDLER
        case CCS_TYPE_AUTO_EXECUTE_HANDLER:
diff --git a/security/ccsecurity/internal.h b/security/ccsecurity/internal.h
index aa5f055..d6352b7 100644
--- a/security/ccsecurity/internal.h
+++ b/security/ccsecurity/internal.h
@@ -445,6 +445,9 @@ enum ccs_acl_entry_type_index {
 #endif
 #ifdef CONFIG_CCSECURITY_IPC
        CCS_TYPE_SIGNAL_ACL,
+#ifdef CONFIG_CCSECURITY_ANDROID
+       CCS_TYPE_ANDROID_TRANSACTION_ACL,
+#endif
 #endif
 #ifdef CONFIG_CCSECURITY_TASK_EXECUTE_HANDLER
        CCS_TYPE_AUTO_EXECUTE_HANDLER,
@@ -641,6 +644,9 @@ enum ccs_mac_index {
 #ifdef CONFIG_CCSECURITY_IPC
        CCS_MAC_SIGNAL,
 #endif
```

```
+#ifdef CONFIG_CCSECURITY_ANDROID
+       CCS_MAC_ANDROID_BINDER_TRANSACTION,
+#endif
 #ifdef CONFIG_CCSECURITY_CAPABILITY
        CCS_MAC_CAPABILITY_USE_ROUTE_SOCKET,
        CCS_MAC_CAPABILITY_USE_PACKET_SOCKET,
@@ -1149,6 +1155,12 @@ struct ccs_request_info {
                        const char *dest_pattern;
                        int sig;
                } signal;
+#ifdef CONFIG_CCSECURITY_ANDROID
+               struct {
+                       const char *service;
+                       const char *dest_pattern;
+               } android_transaction;
+#endif
 #endif
                struct {
                        const struct ccs_path_info *type;
@@ -1385,6 +1397,14 @@ struct ccs_signal_acl {
        const struct ccs_path_info *domainname;
 };

+/* Structure for "ipc android_binder_transaction" directive. */
+struct ccs_android_transaction_acl {
+       struct ccs_acl_info head; /* type = CCS_TYPE_ANDROID_TRANSACTION_ACL */
+       /* Pointer to destination pattern. */
+       const struct ccs_path_info *service;
+       const struct ccs_path_info *domainname;
+};
+
 /* Structure for "network inet" directive. */
 struct ccs_inet_acl {
        struct ccs_acl_info head; /* type = CCS_TYPE_INET_ACL */
@@ -1425,6 +1445,7 @@ struct ccs_acl_param {
                struct ccs_env_acl env_acl;
                struct ccs_capability_acl capability_acl;
                struct ccs_signal_acl signal_acl;
+               struct ccs_android_transaction_acl android_transaction_acl;
                struct ccs_inet_acl inet_acl;
                struct ccs_unix_acl unix_acl;
                /**/
diff --git a/security/ccsecurity/permission.c b/security/ccsecurity/permission.c
index 932afca..7b79a33 100644
--- a/security/ccsecurity/permission.c
+++ b/security/ccsecurity/permission.c
@@ -409,6 +409,11 @@ static int ccs_signal_acl0(pid_t tgid, pid_t pid, int sig);
 static int ccs_signal_acl2(const int sig, const int pid);
 #endif

+#ifdef CONFIG_CCSECURITY_ANDROID
+static int __ccs_android_binder_permission(pid_t pid, struct
    binder_transaction_data *tr);
+static bool ccs_check_android_binder_transaction_acl(struct ccs_request_info *r,
     const struct ccs_acl_info *ptr);
+#endif
+
 #ifdef CONFIG_CCSECURITY_FILE_GETATTR
 static int __ccs_getattr_permission(struct vfsmount *mnt,
                                     struct dentry *dentry);
@@ -749,6 +754,10 @@ static bool ccs_check_entry(struct ccs_request_info *r,
        case CCS_TYPE_SIGNAL_ACL:
```

```
110                 return ccs_check_signal_acl(r, ptr);
111  #endif
112 +#ifdef CONFIG_CCSECURITY_ANDROID
113 +       case CCS_TYPE_ANDROID_TRANSACTION_ACL:
114 +               return ccs_check_android_binder_transaction_acl(r, ptr);
115 +#endif
116  #ifdef CONFIG_CCSECURITY_TASK_DOMAIN_TRANSITION
117         case CCS_TYPE_MANUAL_TASK_ACL:
118                 return ccs_check_task_acl(r, ptr);
119 @@ -1705,6 +1714,9 @@ void __init ccs_permission_init(void)
120         ccsecurity_ops.sigqueue_permission = ccs_signal_acl;
121         ccsecurity_ops.tgsigqueue_permission = ccs_signal_acl0;
122  #endif
123 +#ifdef CONFIG_CCSECURITY_ANDROID
124 +       ccsecurity_ops.android_binder_permission =
        __ccs_android_binder_permission;
125 +#endif
126  #ifdef CONFIG_CCSECURITY_CAPABILITY
127         ccsecurity_ops.capable = __ccs_capable;
128         ccsecurity_ops.ptrace_permission = __ccs_ptrace_permission;
129 @@ -2620,6 +2632,79 @@ static void ccs_transit_android_app_process(void) {
130                 }
131  }
132
133 +static int __ccs_android_binder_permission(pid_t pid, struct
        binder_transaction_data *tr) {
134 +       int i;
135 +       char *buf;
136 +       const char *ptr;
137 +       struct ccs_request_info r;
138 +       struct ccs_domain_info *dest = NULL;
139 +       const struct ccs_domain_info * const domain = ccs_current_domain();
140 +       if (ccs_init_request_info(&r, CCS_MAC_ANDROID_BINDER_TRANSACTION) ==
        CCS_CONFIG_DISABLED)
141 +               return 0;
142 +       if (tr->data_size < 8 || tr->data.ptr.buffer == NULL)
143 +               return 0;
144 +       r.param_type = CCS_TYPE_ANDROID_TRANSACTION_ACL;
145 +       r.param.android_transaction.dest_pattern = domain->domainname->name;
146 +       r.granted = true;
147 +       /*
148 +        * We do something very similar to the signal IPC checks.
149 +        * */
150 +       {
151 +               struct task_struct *p = NULL;
152 +               ccs_tasklist_lock();
153 +               if (pid > 1)
154 +#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 6, 24)
155 +                       p = ccsecurity_exports.find_task_by_vpid(pid);
156 +#else
157 +                       p = find_task_by_pid(pid);
158 +#endif
159 +               else
160 +                       return -1;
161 +               if (p)
162 +                       dest = ccs_task_domain(p);
163 +               ccs_tasklist_unlock();
164 +       }
165 +       if (!dest)
166 +               return 0; /* I can't find destination... this shouldn't happen
        for binder */
167 +       r.param.android_transaction.dest_pattern = dest->domainname->name;
```

```
168 +
169 +       /* Now we extract the requested service */
170 +       buf = kmalloc(tr->data_size, CCS_GFP_FLAGS);
171 +       for (ptr = tr->data.ptr.buffer + 8, i = 0; i < tr->data_size - 8; i++,
        ptr += 2) {
172 +               buf[i] = *ptr;
173 +               if (*ptr == '\0' && *(ptr + 2) == '\0')
174 +                       break;
175 +       }
176 +       r.param.android_transaction.service = ccs_get_name(buf)->name;
177 +       kfree(buf);
178 +       if (domain == dest) {
179 +               ccs_audit_log(&r);
180 +               return 0;               /* No check for self domain. */
181 +       }
182 +       return ccs_check_acl(&r);
183 +}
184 +
185 +static bool ccs_check_android_binder_transaction_acl(struct ccs_request_info *r,
         const struct ccs_acl_info *ptr) {
186 +       const struct ccs_android_transaction_acl *acl =
187 +               container_of(ptr, typeof(*acl), head);
188 +       const int len = acl->domainname->total_len;
189 +       if (!strncmp(acl->domainname->name,
190 +                    r->param.android_transaction.dest_pattern, len)) {
191 +               switch (r->param.android_transaction.dest_pattern[len]) {
192 +               case ' ':
193 +               case '\0':
194 +                       /*
195 +                        * This means: if the remaining part of the requested
196 +                        * domain is NULL (the domains are equal) or more
197 +                        * specialized (' ') than the matched one, let's match
198 +                        * the ACL.
199 +                        * */
200 +                       if (!strncmp(acl->service->name, r->param.
        android_transaction.service, acl->service->total_len))
201 +                               return true;
202 +               }
203 +       }
204 +       return false;
205 +}
206  #endif
207
208  /**
209 diff --git a/security/ccsecurity/policy_io.c b/security/ccsecurity/policy_io.c
210 index ebe7012..0b36249 100644
211 --- a/security/ccsecurity/policy_io.c
212 +++ b/security/ccsecurity/policy_io.c
213 @@ -84,6 +84,9 @@ static const u8 ccs_index2category[CCS_MAX_MAC_INDEX] = {
214  #ifdef CONFIG_CCSECURITY_IPC
215         /* CONFIG::ipc group */
216         [CCS_MAC_SIGNAL]          = CCS_MAC_CATEGORY_IPC,
217 +#ifdef CONFIG_CCSECURITY_ANDROID
218 +       [CCS_MAC_ANDROID_BINDER_TRANSACTION] = CCS_MAC_CATEGORY_IPC,
219 +#endif
220  #endif
221  #ifdef CONFIG_CCSECURITY_CAPABILITY
222         /* CONFIG::capability group */
223 @@ -174,6 +177,9 @@ static const char * const ccs_mac_keywords[CCS_MAX_MAC_INDEX
224  #ifdef CONFIG_CCSECURITY_IPC
225         /* CONFIG::ipc group */
226         [CCS_MAC_SIGNAL] = "signal",
```

```
227 +#ifdef CONFIG_CCSECURITY_ANDROID
228 +       [CCS_MAC_ANDROID_BINDER_TRANSACTION] = "android_binder_transaction",
229 +#endif
230  #endif
231  #ifdef CONFIG_CCSECURITY_CAPABILITY
232         /* CONFIG::capability group */
233 @@ -582,6 +588,9 @@ static int ccs_write_misc(struct ccs_acl_param *param);
234
235  #ifdef CONFIG_CCSECURITY_IPC
236  static int ccs_write_ipc(struct ccs_acl_param *param);
237 +#ifdef CONFIG_CCSECURITY_ANDROID
238 +static int ccs_write_android_transaction(struct ccs_acl_param *param);
239 +#endif
240  #endif
241
242  #ifdef CONFIG_CCSECURITY_TASK_DOMAIN_TRANSITION
243 @@ -3391,6 +3400,17 @@ static int ccs_write_ipc(struct ccs_acl_param *param)
244         return ccs_update_acl(sizeof(*e), param);
245  }
246
247 +#ifdef CONFIG_CCSECURITY_ANDROID
248 +static int ccs_write_android_transaction(struct ccs_acl_param *param) {
249 +       struct ccs_android_transaction_acl *e = &param->e.android_transaction_acl
    ;
250 +       e->head.type = CCS_TYPE_ANDROID_TRANSACTION_ACL;
251 +       e->service = ccs_get_name(ccs_read_token(param));
252 +       e->domainname = ccs_get_domainname(param);
253 +       if (!e->domainname || !e->service)
254 +               return -EINVAL;
255 +       return ccs_update_acl(sizeof(*e), param);
256 +}
257 +#endif
258  #endif
259
260  #ifdef CONFIG_CCSECURITY_CAPABILITY
261 @@ -3460,6 +3480,9 @@ static int ccs_write_acl(struct ccs_policy_namespace *ns,
262  #endif
263  #ifdef CONFIG_CCSECURITY_IPC
264                 { "ipc signal ", ccs_write_ipc },
265 +#ifdef CONFIG_CCSECURITY_ANDROID
266 +               { "ipc android_binder_transaction ",
    ccs_write_android_transaction },
267 +#endif
268  #endif
269                 { "task ", ccs_write_task },
270         };
271 @@ -4012,6 +4035,15 @@ static bool ccs_print_entry(struct ccs_io_buffer *head,
272                 ccs_print_number_union_nospace(head, &ptr->sig);
273                 ccs_set_space(head);
274                 ccs_set_string(head, ptr->domainname->name);
275 +#ifdef CONFIG_CCSECURITY_ANDROID
276 +       } else if (acl_type == CCS_TYPE_ANDROID_TRANSACTION_ACL) {
277 +               struct ccs_android_transaction_acl *ptr =
278 +                       container_of(acl, typeof(*ptr), head);
279 +               ccs_set_group(head, "ipc android_binder_transaction ");
280 +               ccs_set_string(head, ptr->service->name);
281 +               ccs_set_space(head);
282 +               ccs_set_string(head, ptr->domainname->name);
283 +#endif
284  #endif
285         } else if (acl_type == CCS_TYPE_MOUNT_ACL) {
286                 struct ccs_mount_acl *ptr =
```

```
287 @@ -4943,6 +4975,12 @@ int ccs_audit_log(struct ccs_request_info *r)
288                 return ccs_supervisor(r, "ipc signal %d %s\n",
289                                       r->param.signal.sig,
290                                       r->param.signal.dest_pattern);
291 +#ifdef CONFIG_CCSECURITY_ANDROID
292 +       case CCS_TYPE_ANDROID_TRANSACTION_ACL:
293 +               return ccs_supervisor(r, "ipc android_binder_transaction %s %s\n
        ",
294 +                                     r->param.android_transaction.service,
295 +                                     r->param.android_transaction.dest_pattern);
296 +#endif
297  #endif
298         }
299         return 0;
```

B.1.4. ccsecurity.h

```
1  diff --git a/include/linux/ccsecurity.h b/include/linux/ccsecurity.h
2  index a33fbaa..8f6b2b8 100644
3  --- a/include/linux/ccsecurity.h
4  +++ b/include/linux/ccsecurity.h
5  @@ -46,6 +46,10 @@ int search_binary_handler(struct linux_binprm *bprm, struct
       pt_regs *regs);
6   #include <linux/uidgid.h>
7   #endif
8
9  +#ifdef CONFIG_CCSECURITY_ANDROID
10 +#include "../../drivers/staging/android/binder.h"
11 +#endif
12 +
13  /* For exporting variables and functions. */
14  struct ccsecurity_exports {
15         void (*load_policy) (const char *filename);
16 @@ -123,6 +127,9 @@ struct ccsecurity_operations {
17         int (*kill_permission) (pid_t pid, int sig);
18         int (*tgkill_permission) (pid_t tgid, pid_t pid, int sig);
19         int (*tkill_permission) (pid_t pid, int sig);
20 +#ifdef CONFIG_CCSECURITY_ANDROID
21 +       int (*android_binder_permission) (pid_t pid, struct
       binder_transaction_data *tr);
22 +#endif
23         int (*socket_create_permission) (int family, int type, int protocol);
24         int (*socket_listen_permission) (struct socket *sock);
25         int (*socket_connect_permission) (struct socket *sock,
26 @@ -891,6 +898,13 @@ static inline int ccs_tgsigqueue_permission(pid_t tgid,
       pid_t pid, int sig)
27
28  #endif
29
30 +#ifdef CONFIG_CCSECURITY_ANDROID
31 +static inline int ccs_android_binder_permission(pid_t pid, struct
       binder_transaction_data *tr) {
32 +       int (*func) (pid_t, struct binder_transaction_data *) = ccsecurity_ops.
       android_binder_permission;
33 +       return func ? func(pid, tr) : 0;
34 +}
35 +#endif
36 +
37  /* Index numbers for Capability Controls. */
38  enum ccs_capability_acl_index {
39         /* socket(PF_ROUTE, *, *)                                    */
```

B.2. Binder

```
diff --git a/drivers/staging/android/binder.c b/drivers/staging/android/binder.c
index 3479d75..15c18b5 100644
--- a/drivers/staging/android/binder.c
+++ b/drivers/staging/android/binder.c
@@ -33,6 +33,10 @@
 #include <linux/vmalloc.h>
 #include "binder.h"

+#ifdef CONFIG_CCSECURITY_ANDROID
+#include <linux/ccsecurity.h>
+#endif
+
 static DEFINE_MUTEX(binder_lock);
 static HLIST_HEAD(binder_procs);
 static struct binder_node *binder_context_mgr_node;
@@ -1349,6 +1353,24 @@ binder_transaction(struct binder_proc *proc, struct
    binder_thread *thread,
                        return_error = BR_DEAD_REPLY;
                        goto err_dead_binder;
                }
+#ifdef CONFIG_CCSECURITY_ANDROID
+               /*
+                * We only check for TOMOYO permissions here since
+                * we always permit a reply to an existing transaction
+                * that was previously authorized.
+                * Putting the check here we are quite sure that we
+                * have a valid target handle and node.
+                * */
+               if (ccs_android_binder_permission(target_proc->pid, tr)) {
+                       binder_user_error("binder: %d:%d BR_TRANSACTION -> "
+                                         "%d - node %d was denied by ccs policy\n
    ",
+                                         proc->pid, thread->pid,
+                                         target_proc->pid, target_node->debug_id);
+                       return_error = BR_FAILED_REPLY;
+                       goto err_ccs_denied;
+               }
+
+#endif
                if (!(tr->flags & TF_ONE_WAY) && thread->transaction_stack) {
                        struct binder_transaction *tmp;
                        tmp = thread->transaction_stack;
@@ -1641,6 +1663,9 @@ err_empty_call_stack:
 err_dead_binder:
 err_invalid_target_handle:
 err_no_context_mgr_node:
+#ifdef CONFIG_CCSECURITY_ANDROID
+err_ccs_denied:
+#endif
        if (binder_debug_mask & BINDER_DEBUG_FAILED_TRANSACTION)
                printk(KERN_INFO "binder: %d:%d transaction failed %d, size"
                                 "%zd-%zd\n",
```

Apéndice C

Parche Herramientas User-space

C.1. ccs-tools

```
diff --git a/usr_sbin/ccs-checkpolicy.c b/usr_sbin/ccs-checkpolicy.c
index 05b1c51..ba1d190 100644
--- a/usr_sbin/ccs-checkpolicy.c
+++ b/usr_sbin/ccs-checkpolicy.c
@@ -630,6 +630,7 @@ static _Bool ccs_check_domain_policy2(char *policy)
                { "capability ", ccs_check_capability },
                { "file ", ccs_check_file },
                { "ipc signal ", ccs_check_u16, ccs_check_domain },
+               { "ipc android_binder_transaction ", ccs_check_domain },
                { "misc env ", ccs_check_path },
                { "network ", ccs_check_network },
                { "task auto_domain_transition ", ccs_check_domain },
diff --git a/usr_sbin/editpolicy.h b/usr_sbin/editpolicy.h
index 1283044..45efc9a 100644
--- a/usr_sbin/editpolicy.h
+++ b/usr_sbin/editpolicy.h
@@ -335,6 +335,7 @@ enum ccs_editpolicy_directives {
        CCS_DIRECTIVE_FILE_WRITE,
        CCS_DIRECTIVE_INITIALIZE_DOMAIN,
        CCS_DIRECTIVE_IPC_SIGNAL,
+       CCS_DIRECTIVE_ANDROID_BINDER_TRANSACTION,
        CCS_DIRECTIVE_KEEP_DOMAIN,
        CCS_DIRECTIVE_MISC_ENV,
        CCS_DIRECTIVE_NETWORK_INET,
diff --git a/usr_sbin/editpolicy_keyword.c b/usr_sbin/editpolicy_keyword.c
index aa58313..4bfc161 100644
--- a/usr_sbin/editpolicy_keyword.c
+++ b/usr_sbin/editpolicy_keyword.c
@@ -313,6 +313,7 @@ struct ccs_editpolicy_directive ccs_directives[
    CCS_MAX_DIRECTIVE_INDEX] = {
        [CCS_DIRECTIVE_INITIALIZE_DOMAIN]
        = { "initialize_domain", NULL, 0, 0 },
        [CCS_DIRECTIVE_IPC_SIGNAL]    = { "ipc signal", NULL, 0, 0 },
+       [CCS_DIRECTIVE_ANDROID_BINDER_TRANSACTION] = {"ipc
    android_binder_transaction", NULL, 0, 0 },
        [CCS_DIRECTIVE_KEEP_DOMAIN]   = { "keep_domain", NULL, 0, 0 },
```

```
35          [CCS_DIRECTIVE_MISC_ENV]      = { "misc env", NULL, 0, 0 },
36          [CCS_DIRECTIVE_NETWORK_INET]  = { "network inet", NULL, 0, 0 },
37  diff --git a/usr_sbin/editpolicy_offline.c b/usr_sbin/editpolicy_offline.c
38  index d54e175..9655d7e 100644
39  --- a/usr_sbin/editpolicy_offline.c
40  +++ b/usr_sbin/editpolicy_offline.c
41  @@ -93,6 +93,7 @@ enum ccs_acl_entry_type_index {
42          CCS_TYPE_INET_ACL,
43          CCS_TYPE_UNIX_ACL,
44          CCS_TYPE_SIGNAL_ACL,
45  +       CCS_TYPE_ANDROID_TRANSACTION_ACL,
46          CCS_TYPE_AUTO_EXECUTE_HANDLER,
47          CCS_TYPE_DENIED_EXECUTE_HANDLER,
48          CCS_TYPE_AUTO_TASK_ACL,
49  @@ -285,6 +286,7 @@ enum ccs_mac_index {
50          CCS_MAC_NETWORK_UNIX_SEQPACKET_ACCEPT,
51          CCS_MAC_ENVIRON,
52          CCS_MAC_SIGNAL,
53  +       CCS_MAC_ANDROID_BINDER_TRANSACTION,
54          CCS_MAC_CAPABILITY_USE_ROUTE_SOCKET,
55          CCS_MAC_CAPABILITY_USE_PACKET_SOCKET,
56          CCS_MAC_CAPABILITY_SYS_REBOOT,
57  @@ -731,6 +733,14 @@ struct ccs_signal_acl {
58          const struct ccs_path_info *domainname;
59   };
60
61  +/* Structure for "ipc android_binder_transaction" directive. */
62  +struct ccs_android_transaction_acl {
63  +       struct ccs_acl_info head; /* type = CCS_TYPE_ANDROID_TRANSACTION_ACL */
64  +       /* Pointer to destination pattern. */
65  +       const struct ccs_path_info *service;
66  +       const struct ccs_path_info *domainname;
67  +};
68  +
69   /* Structure for "network inet" directive. */
70   struct ccs_inet_acl {
71          struct ccs_acl_info head; /* type = CCS_TYPE_INET_ACL */
72  @@ -881,6 +891,7 @@ static const char * const ccs_mac_keywords[CCS_MAX_MAC_INDEX
73          [CCS_MAC_NETWORK_UNIX_SEQPACKET_ACCEPT]  = "unix_seqpacket_accept",
74          /* CONFIG::ipc group */
75          [CCS_MAC_SIGNAL] = "signal",
76  +       [CCS_MAC_ANDROID_BINDER_TRANSACTION] = "android_binder_transaction",
77          /* CONFIG::capability group */
78          [CCS_MAC_CAPABILITY_USE_ROUTE_SOCKET]  = "use_route",
79          [CCS_MAC_CAPABILITY_USE_PACKET_SOCKET] = "use_packet",
80  @@ -1203,6 +1214,7 @@ static const u8 ccs_index2category[CCS_MAX_MAC_INDEX] = {
81          [CCS_MAC_NETWORK_UNIX_SEQPACKET_ACCEPT]  = CCS_MAC_CATEGORY_NETWORK,
82          /* CONFIG::ipc group */
83          [CCS_MAC_SIGNAL]          = CCS_MAC_CATEGORY_IPC,
84  +       [CCS_MAC_ANDROID_BINDER_TRANSACTION] = CCS_MAC_CATEGORY_IPC,
85          /* CONFIG::capability group */
86          [CCS_MAC_CAPABILITY_USE_ROUTE_SOCKET]  = CCS_MAC_CATEGORY_CAPABILITY,
87          [CCS_MAC_CAPABILITY_USE_PACKET_SOCKET] = CCS_MAC_CATEGORY_CAPABILITY,
88  @@ -3327,6 +3339,14 @@ static bool ccs_same_signal_acl(const struct ccs_acl_info
      *a,
89                  p1->domainname == p2->domainname;
90   }
91
92  +static bool ccs_same_android_transaction_acl(const struct ccs_acl_info *a,
93  +                                             const struct ccs_acl_info *b)
94  +{
95  +       const struct ccs_android_transaction_acl *p1 = container_of(a, typeof(*p1
```

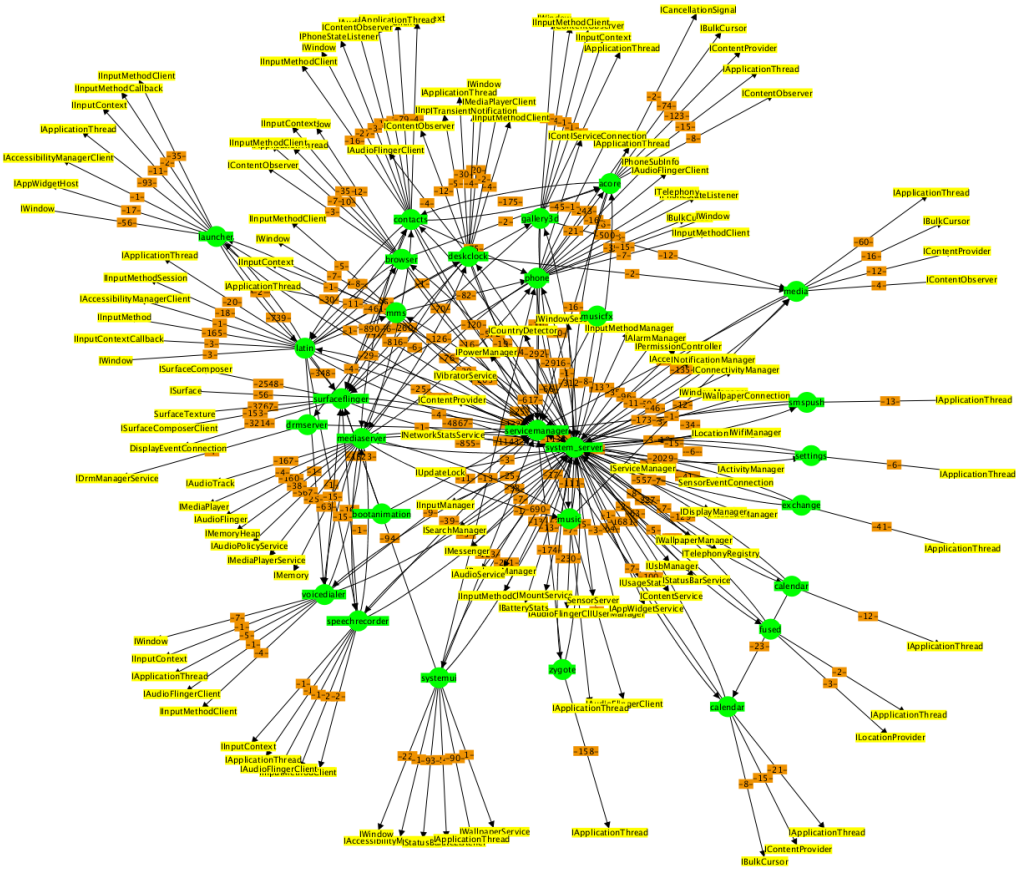
```
    ), head);
+       const struct ccs_android_transaction_acl *p2 = container_of(b, typeof(*p2
    ), head);
+       return (p1->domainname == p2->domainname && p1->service == p2->service);
+}
+
 /**
  * ccs_write_ipc - Update "struct ccs_signal_acl" list.
  *
@@ -3351,6 +3371,21 @@ static int ccs_write_ipc(struct ccs_acl_param *param)
        return error;
 }

+static int ccs_write_android_transaction(struct ccs_acl_param *param) {
+       struct ccs_android_transaction_acl e = { .head.type =
    CCS_TYPE_ANDROID_TRANSACTION_ACL };
+       int error;
+       e.service = ccs_get_name(ccs_read_token(param));
+       e.domainname = ccs_get_domainname(param);
+       if (!e.domainname || !e.service)
+               error = -EINVAL;
+       else
+               error = ccs_update_domain(&e.head, sizeof(e), param,
+                                         ccs_same_android_transaction_acl, NULL)
    ;
+       ccs_put_name(e.service);
+       ccs_put_name(e.domainname);
+       return error;
+}
+

 /**
  * ccs_same_reserved - Check for duplicated "struct ccs_reserved" entry.
@@ -3870,6 +3905,7 @@ static int ccs_write_domain2(struct ccs_policy_namespace *
    ns,
                { "misc ", ccs_write_misc },
                { "capability ", ccs_write_capability },
                { "ipc signal ", ccs_write_ipc },
+               { "ipc android_binder_transaction ",
    ccs_write_android_transaction },
                { "task ", ccs_write_task },
        };
        u8 i;
@@ -4295,6 +4331,11 @@ static void ccs_print_entry(const struct ccs_acl_info *acl
    )
                ccs_set_group("ipc signal ");
                ccs_print_number_union_nospace(&ptr->sig);
                cprintf(" %s", ptr->domainname->name);
+       } else if (acl_type == CCS_TYPE_ANDROID_TRANSACTION_ACL) {
+               struct ccs_android_transaction_acl *ptr =
+                       container_of(acl, typeof(*ptr), head);
+               ccs_set_group("ipc android_binder_transaction ");
+               cprintf("%s %s", ptr->service->name, ptr->domainname->name);
        } else if (acl_type == CCS_TYPE_MOUNT_ACL) {
                struct ccs_mount_acl *ptr =
                        container_of(acl, typeof(*ptr), head);
diff --git a/usr_sbin/editpolicy_optimizer.c b/usr_sbin/editpolicy_optimizer.c
index b875603..57a7e26 100644
--- a/usr_sbin/editpolicy_optimizer.c
+++ b/usr_sbin/editpolicy_optimizer.c
@@ -212,6 +212,7 @@ static void ccs_tokenize(char *buffer, char *w[5],
        case CCS_DIRECTIVE_FILE_RENAME:
```

```
151          case CCS_DIRECTIVE_FILE_PIVOT_ROOT:
152          case CCS_DIRECTIVE_IPC_SIGNAL:
153  +       case CCS_DIRECTIVE_ANDROID_BINDER_TRANSACTION:
154                  words = 2;
155                  break;
156          case CCS_DIRECTIVE_FILE_EXECUTE:
157  @@ -239,8 +240,8 @@ static void ccs_tokenize(char *buffer, char *w[5],
158                  w[i] = buffer;
159                  if (!cp)
160                          return;
161  -               if (index == CCS_DIRECTIVE_IPC_SIGNAL && i == 1 &&
162  -                   ccs_domain_def(buffer)) {
163  +               if ((index == CCS_DIRECTIVE_IPC_SIGNAL || index ==
         CCS_DIRECTIVE_ANDROID_BINDER_TRANSACTION) &&
164  +                   i == 1 && ccs_domain_def(buffer)) {
165                          cp = strchr(buffer, ' ');
166                          if (!cp)
167                                  return;
168  @@ -447,6 +448,18 @@ void ccs_editpolicy_optimize(const int current)
169                          if (c && c != ' ')
170                                  continue;
171                          break;
172  +               case CCS_DIRECTIVE_ANDROID_BINDER_TRANSACTION:
173  +                       /* Service name component. */
174  +                       if (strcmp(s[0], d[0]))
175  +                               continue;
176  +                       /* Domainname component. */
177  +                       len = strlen(s[1]);
178  +                       if (strncmp(s[1], d[1], len))
179  +                               continue;
180  +                       c = d[1][len];
181  +                       if (c && c != ' ')
182  +                               continue;
183  +                       break;
184                  case CCS_DIRECTIVE_NETWORK_INET:
185                          if (strcmp(s[0], d[0]) || strcmp(s[1], d[1]) ||
186                              !ccs_compare_address(s[2], d[2]) ||
```

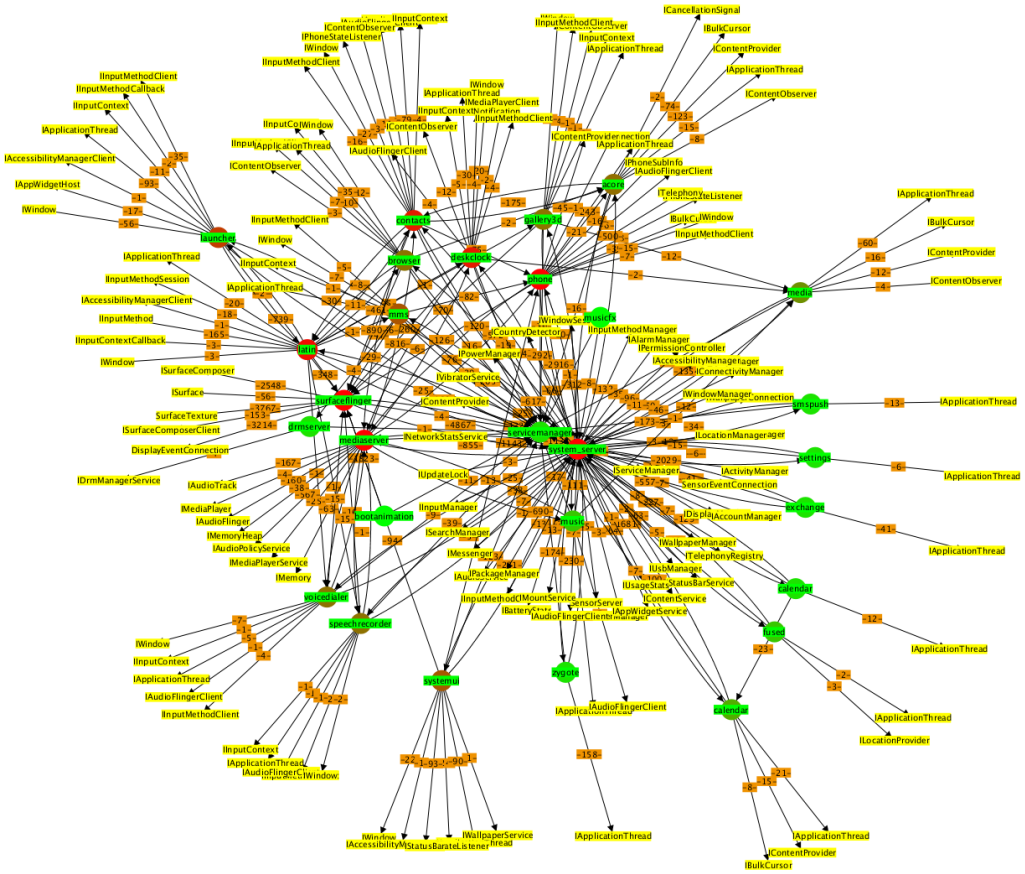
Apéndice D

Grafos

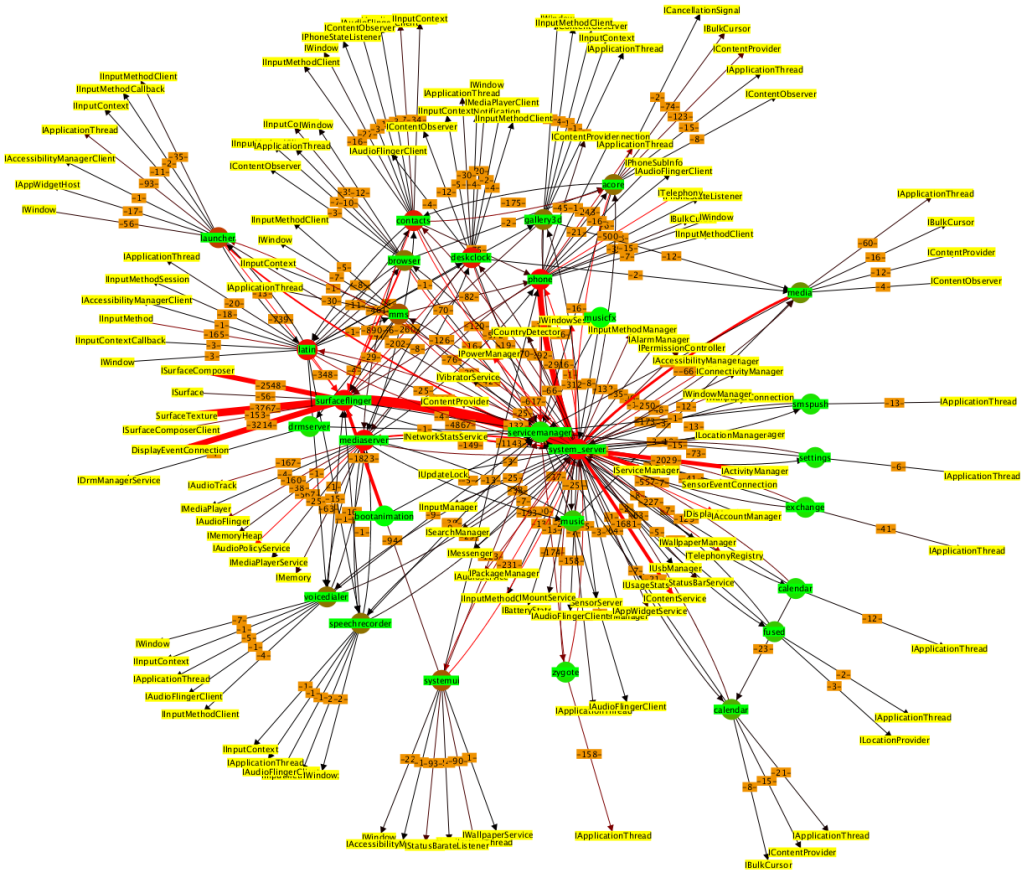
D.1. Simple



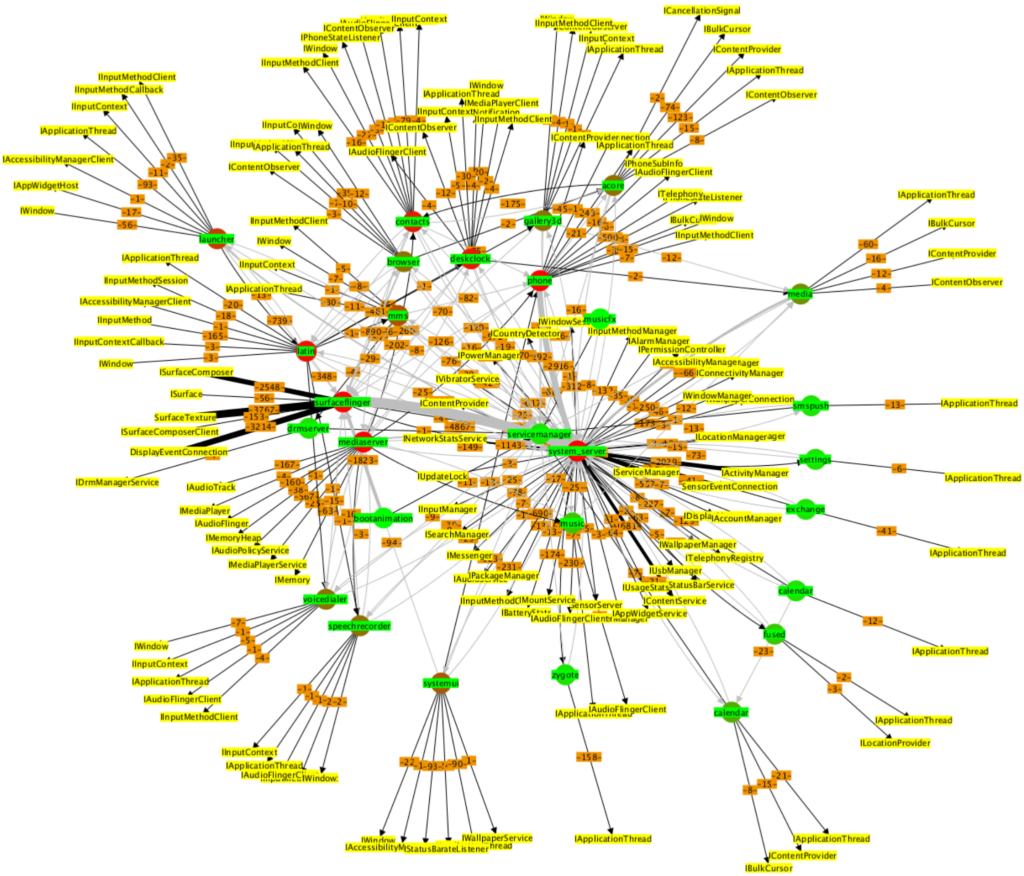
D.2. Centralidad



D.3. Arístas Ponderadas



D.4. Prim Spanning Tree



Bibliografía

[1] *Android Developers: Security-Enhanced Linux*. URL: http://source.android.com/devices/tech/security/se-linux.html (visitado 17 de ago. de 2013).

[2] Ulrik Brandes. «A Faster Algorithm for Betweenness Centrality». En: *Journal of Mathematical Sociology* 25:2 (2001), págs. 163-177. DOI: 10.1080/0022250X.2001.9990249.

[3] Sven Bugiel y col. *Towards Taming Privilege-Escalation Attacks on Android*. Inf. téc. CASED/Technische Universität Darmstadt, Germany, Ruhr-Universität Bochum, Germany y Fraunhofer SIT, Darmstadt, Germany, 2012.

[4] Sven Bugiel y col. *XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks*. Inf. téc. Technische Universität Darmstadt - Center for Advanced Security Research Darmstadt, abr. de 2011.

[5] Jesse Burns. *DEVELOPING SECURE MOBILE APPLICATIONS FOR ANDROID*. Inf. téc. ISEC Partners, oct. de 2008.

[6] Erika Chin y col. «Analyzing inter-application communication in Android». En: *MobiSys '11 Proceedings of the 9th international conference on Mobile systems, applications, and services*. 2011, págs. 239-252. ISBN: 978-1-4503-0643-0. DOI: 10.1145/1999995.2000018.

[7] Lucas Davi y col. «Privilege Escalation Attacks on Android». En: *Lecture Notes in Computer Science*. Vol. 6531. 2011, págs. 346-360.

[8] *Debugging with KGDB*. URL: http://www.stlinux.com/devel/debug/kgdb (visitado 11 de ago. de 2013).

[9] Michael Dietz y col. «Quire: Lightweight provenance for smart phone operating systems». En: *20th USENIX Security Symposium*. 2011.

[10] David Ehringer. *The Dalvik Virtual Machine Architecture*. Mar. de 2010.

[11] W. Enck, M. Ongtang y P. McDaniel. *Mitigating Android software misuse before it happens*. Inf. téc. Pennsylvania State University, 2008.

[12] W. Enck y col. «TaintDroid: An information-flow tracking system for realtime privacy monitoring on smart-phones». En: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2010.

[13] A. P. Felt y col. «Permission re-delegation: Attacks and defenses». En: *20th USENIX Security Symposium*. 2011.

[14] *GitHub: Xposed*. URL: `https://github.com/rovo89/Xposed` (visitado 17 de ago. de 2013).

[15] *GitHub: XPrivacy*. URL: `https://github.com/M66B/XPrivacy` (visitado 17 de ago. de 2013).

[16] *GraphStream: A Dynamic Graph Library*. URL: `http://graphstream-project.org/` (visitado 23 de ago. de 2013).

[17] Security Engineering Research Group. *Analysis of Dalvik Virtual Machine and Class Path Library*. Inf. téc. Institute of Management Sciences, 2009.

[18] Tetsuo Handa. «The role of "pathname based access control" in security». En: *LFJ Symposium BoF*. Jul. de 2008. URL: `http://sourceforge.jp/projects/tomoyo/docs/lfj2008-bof.pdf` (visitado 26 de jul. de 2013).

[19] Jim Huang. *Android IPC Mechanism*. Mar. de 2012. URL: `http://0xlab.org/~jserv/android-binder-ipc.odp` (visitado 26 de jul. de 2013).

[20] Jinseong Jeon y col. *Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android*. Inf. téc. University of Maryland, Department of Computer Science CS-TR-5006, dic. de 2011. URL: `http://hdl.handle.net/1903/12852`.

[21] TOMOYO Linux. *Policy specification*. URL: `http://tomoyo.sourceforge.jp/1.8/policy-specification/index.html.en` (visitado 26 de jul. de 2013).

[22] M. Ongtang y col. «Semantically rich application-centric security in Android». En: *25th Annual Computer Security Applications Conference (ACSAC) - IEEE Computer Society*. 2009.

[23] R. C. Prim. «Shortest connection networks and some generalizations». En: *Bell System Technical Journal* 36 (1957), págs. 1389-1401.

[24] *QEMU + GDB Debugging Environment*. Sep. de 2012. URL: https://www.ece.cmu.edu/~ee349/f-2012/lab2/qemu.pdf (visitado 11 de ago. de 2013).

[25] Khem Raj. *Debugging Linux systems using GDB and QEMU*. Presentación Powerpoint. URL: http://files.meetup.com/1590495/debugging-with-qemu.pdf (visitado 11 de ago. de 2013).

[26] Tomáš Rosa. *Android Binder Security Note – On >Passing Binder Through Another Binder<*. Inf. téc. 2011. URL: http://crypto.hyperlink.cz.

[27] Andrew Ross. *agcc: Script for compiling native C applications for the android*. URL: https://code.google.com/p/agcc/ (visitado 27 de jul. de 2013).

[28] Thorsten Schreiber. *Android Binder – Android Interprocess Communication*. Inf. téc. Seminarthesis. Ruhr-Universität Bochum, Germany, oct. de 2011.

[29] Bluebox Security. *UNCOVERING ANDROID MASTER KEY THAT MAKES 99 % OF DEVICES VULNERABLE*. Jul. de 2013. URL: http://bluebox.com/corporate-blog/bluebox-uncovers-android-master-key/ (visitado 26 de jul. de 2013).

[30] Asaf Shabtai y col. «Google Android: A Comprehensive Security Assessment». En: *Security & Privacy, IEEE* 8.2 (mar. de 2010), págs. 35-44. DOI: 10.1109/MSP.2010.2.

[31] *TOMOYO Linux*. URL: http://tomoyo.sourceforge.jp/ (visitado 25 de ago. de 2013).

[32] Timothy Vidas, Daniel Votipka y Nicolas Christin. «All Your Droid Are Belong to Us: A Survey of Current Android Attacks.» En: *WOOT'11 Proceedings of the 5th USENIX conference on Offensive technologies*. 2011.

[33] *XDA: Xposed Announcement*. URL: http://forum.xda-developers.com/showthread.php?t=1574401 (visitado 17 de ago. de 2013).

[34] *XDA: XPrivacy Announcement*. URL: http://forum.xda-developers.com/showthread.php?t=2320783 (visitado 17 de ago. de 2013).