



# Práctica 2

---

## Reversing de apps Android

## Tabla de contenido

<b>1. Objetivos .....</b>	<b>3</b>
<b>2. Proceso de generación de un apk.....</b>	<b>4</b>
2.1. Generación de un apk: depuración vs distribución .....	4
2.2. Firma manual de una app.....	6
<b>3. Desensamblado, modificación y reensamblado de apks.....</b>	<b>7</b>
3.1. Desensamblado de apks (unpacking).....	8
3.2. Análisis de código smali .....	9
3.3. Modificación del apk.....	12
3.4. Reensamblado del apk (repacking).....	13
3.5. Cuidado con los heurísticos de evasión .....	13
<b>4. ¿Y qué ocurre cuando se utilizan ofuscadores de código? .....</b>	<b>14</b>
<b>5. Reseñas finales .....</b>	<b>15</b>
<b>6. Ejercicio propuesto .....</b>	<b>16</b>

## 1. Objetivos

En esta práctica abordaremos el desensamblado (unpacking) y reensamblado (repacking o build) de apps Android. Veremos cuales son los riesgos inherentes a estos procesos y las posibilidades que éstos ofrecen a los atacantes para inyectar código malicioso en apps que son, a priori al menos, legítimas. Además, veremos cómo la ofuscación permite ponerles las cosas más difíciles a estos atacantes, aunque hay que comprender que no impide este tipo de ataques.

## 2. Proceso de generación de un apk

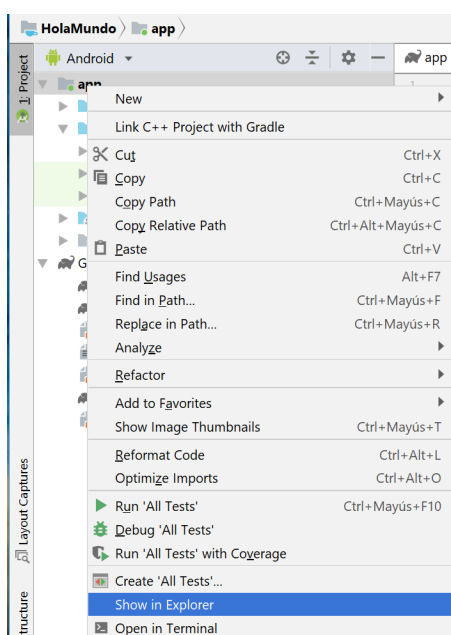
Hasta ahora nos hemos limitado a implementar apps Android que el entorno de desarrollo acababa finalmente desplegando en un emulador o en un dispositivo físico. Sin embargo, esto no genera un apk, es decir, un archivo Android Application Package. Los archivos apk son, por tanto, archivos ejecutables y cada app tiene uno asociado. En esencia, un apk no es otra cosa que un paquete para el sistema operativo Android. Su formato es una variante del formato JAR de Java y se usa para distribuir e instalar componentes empaquetados en smartphones y tablets Android. La ventaja, y también el riesgo, con este tipo de archivos es que nos permiten su instalación en cualquier dispositivo Android, sin necesidad de utilizar la tienda oficial de aplicaciones de la plataforma, el Google Play.

Un proyecto de Android Studio puede producir dos tipos de apk, los utilizados para depurar (*debug*), que son los que se generan por defecto cuando ejecutamos las apps en desarrollo en un emulador o dispositivo conectado al entorno, y los producidos para distribuir (*release*), que son los que se preparan para ser subidos al Google Play y que deben firmarse convenientemente. Pasemos a ver cómo generamos estos ficheros y dónde los podemos localizar.

Cabe señalar que Google detalla minuciosamente el proceso a seguir para distribuir apps a través de Google Play. Este proceso es complejo y comporta la preparación de mucho material (capturas de la app, iconos, logos, videos promocionales, texto descriptivo, etc.) y la realización de ciertas actividades de monetización y marketing que no vamos a abordar en esta práctica. El alumno interesado encontrará información al respecto en <https://developer.android.com/distribute/marketing-tools/alternative-distribution?hl=es-419>.

### 2.1. Generación de un apk: depuración vs distribución

Los directorios de depuración (*debug*) y distribución (*release*) en los que Android Studio deposita las apks generadas forman parte de la estructura de directorios del proyecto. Las siguientes imágenes muestran cómo ir al directorio asociado a una *app* y como localizar en el mismo los directorios de depuración y distribución que acabamos de mencionar. Se está considerando que partimos de la app *HolaMundo* que Android Studio genera automáticamente cuando se genera un proyecto con una actividad vacía.



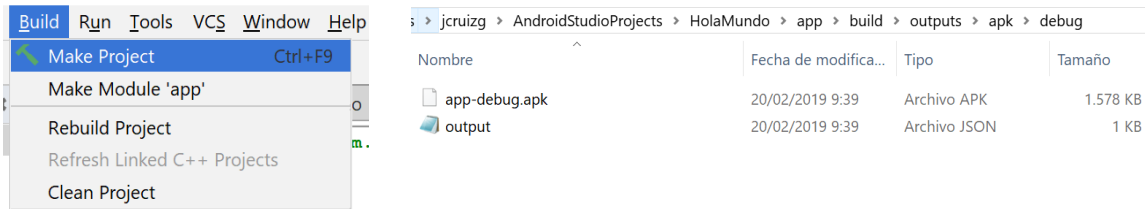
(C:) > Usuarios > jcrui zg > AndroidStudioProjects > HolaMundo > app			
Nombre	Fecha de modifica...	Tipo	Tamaño
build	18/02/2019 18:25	Carpeta de archivos	
libs	18/02/2019 18:16	Carpeta de archivos	
release	20/02/2019 9:27	Carpeta de archivos	
src	18/02/2019 18:16	Carpeta de archivos	

(C:) > Usuarios > jcrui zg > AndroidStudioProjects > HolaMundo > app > build > outputs > apk			
Nombre	Fecha de modifica...	Tipo	Tamaño
debug	20/02/2019 9:22	Carpeta de archivos	

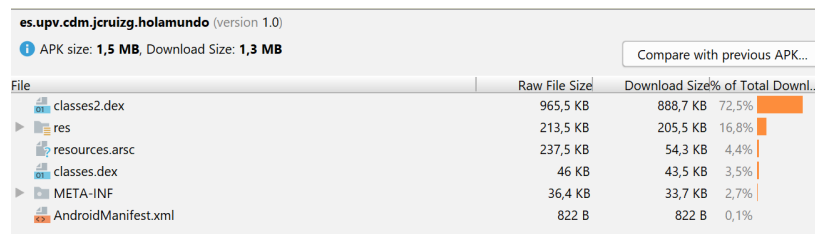
Estos directorios están normalmente vacíos, aunque hayamos ejecutado la app varias veces en nuestro emulador. Para poblarlos, procederemos como se muestra a continuación.

Utilizaremos la opción Build→Make Project, o Ctrl+F9, para conseguir que Android Studio genere una apk de depuración (*app-debug.apk*). Esta apk será útil para hacer pruebas, pero no es una app firmada y, por tanto, no está preparada para ser distribuida.

HolaMundo] - app - Android Studio



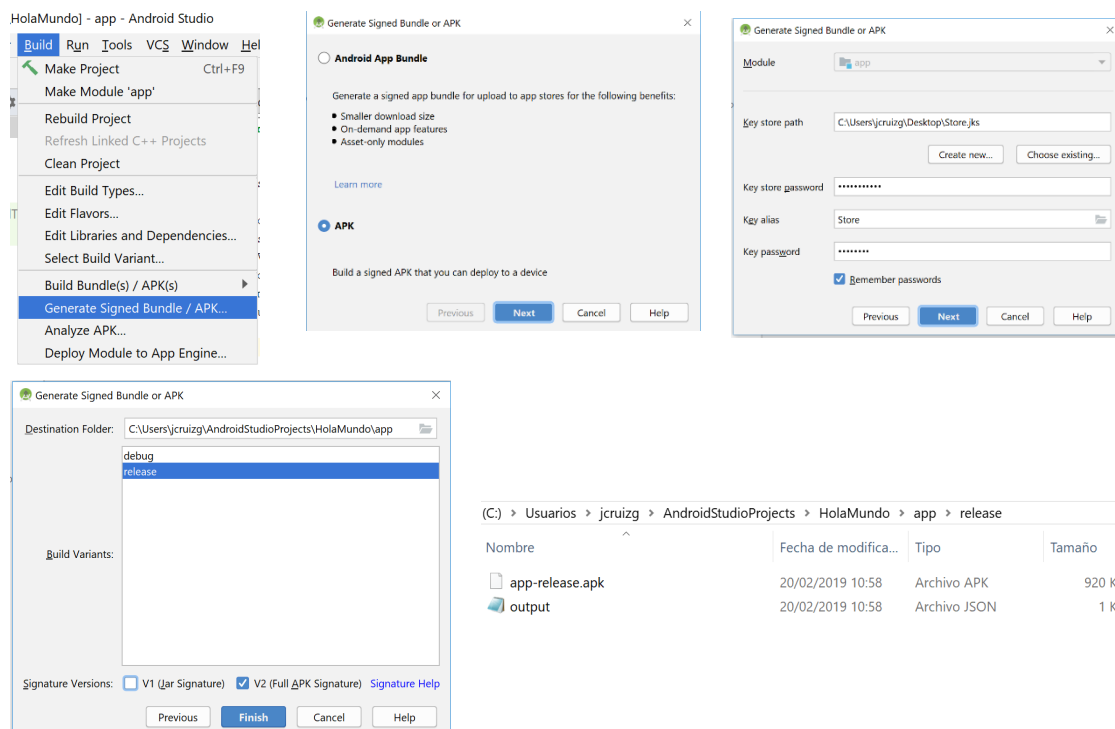
Una vez generada la apk podremos analizar su contenido mediante la opción Build→Analyze APK... que nos ofrece, entre otras cosas, información relativa al tamaño del APK generado y de la aportación al mismo de cada uno de los ficheros, clases y recursos incluidos en la app. La siguiente imagen da una idea de cómo se muestra esta información:



es.upv.cdm.jrcruizg.holamundo (version 1.0)  
APK size: 1,5 MB, Download Size: 1,3 MB

File	Raw File Size	Download Size	% of Total Downl...
classes2.dex	965,5 KB	888,7 KB	72,5%
res	213,5 KB	205,5 KB	16,8%
resources.arsc	237,5 KB	54,3 KB	4,4%
classes.dex	46 KB	43,5 KB	3,5%
META-INF	36,4 KB	33,7 KB	2,7%
AndroidManifest.xml	822 B	822 B	0,1%

Para preparar una app para ser distribuida debemos proceder de otra manera. Utilizaremos la opción Build→Generate Signed Build/APK, a continuación APK e indicaremos dónde se encuentra el almacén en el que guardamos nuestra firma y la contraseña para poder utilizarla. Al final, indicaremos que lo que deseamos es que se genere una *apk release*.



Si el almacén de claves no existiere deberemos crear uno y utilizarlo para firmar la app. A nivel de firma es posible generar una firma antigua (V1), con la básicamente se firman las clases que contiene el APK o una firma V2 con la que obtenemos una firma de todo el APK. Se recomienda utilizar este segundo tipo de firma.

Cabe señalar que este proceso de firma se sirve de las aplicaciones `apksigner.jar` (firma V2) o `signapk.jar` (firma V1). Con estas aplicaciones es posible firmar cualquier apk sin depender de tener su código o no y, por tanto, de si se puede o no utilizar Android Studio.

Alternativamente, si por alguna razón decidiéramos no servirnos de Android Studio para generar el almacén de claves sino hacerlo manualmente sirviéndonos de un terminal podemos hacerlo utilizando la herramienta `keytool`, que se suministra con la JDK de Java. La orden que podríamos utilizar para crear un almacén de claves es la siguiente:

```
C:\Users\jrcruiz>keytool -genkey -v -keystore nombrellave.keystore -alias usuario -keyalg RSA -keysize 2048 -validity 10000
```

Sin embargo, si utilizamos esta orden veremos que en el terminal nos aparece la siguiente advertencia:

```
Warning:  
El almacén de claves JKS utiliza un formato propietario. Se recomienda migrar a PKCS12, que es un formato estándar del s  
ector que utiliza "keytool -importkeystore -srckeystore nombrellave.keystore -destkeystore nombrellave.keystore -deststo  
retype pkcs12".
```

Por consiguiente, podemos migrar el almacén de claves a PKCS12 simplemente utilizando la orden que se nos propone.

```
C:\Users\jrcruiz>keytool -importkeystore -srckeystore nombrellave.keystore -destkeystore nombrellave.keystore -deststoretype pkcs12  
Introduzca la contraseña de almacén de claves de origen:  
La entrada del alias usuario se ha importado correctamente.  
Comando de importación completado: 1 entradas importadas correctamente, 0 entradas incorrectas o canceladas  
  
Warning:  
Se ha migrado "nombrellave.keystore" a Non JKS/JCEKS. Se ha realizado la copia de seguridad del almacén de claves JKS como "nombrellave.keystore.old".
```

Veamos a continuación como firmar manualmente una apk utilizando `apksigner.jar`.

## 2.2. Firma manual de una app

Esta aplicación forma parte de las build-tools de la SDK de Android. Para poder firmar una app necesitamos dos cosas: un almacén (archivo jks de *Java Key Store*) que contenga una clave de firma válida y, obviamente, una app no firmada.

Lo primero es alinear el apk sirviéndonos de `zipalign`. Esta aplicación también forma parte de las build-tools de la SDK Android que estamos utilizando y garantiza que todos los datos descomprimidos de la apk comiencen con una alineación de bytes en particular relacionada en relación con el comienzo del archivo, lo que reduce significativamente el consumo de memoria RAM de una aplicación. Se recomienda una alineación de 4 bytes. Para realizar la alineación procedemos como sigue:

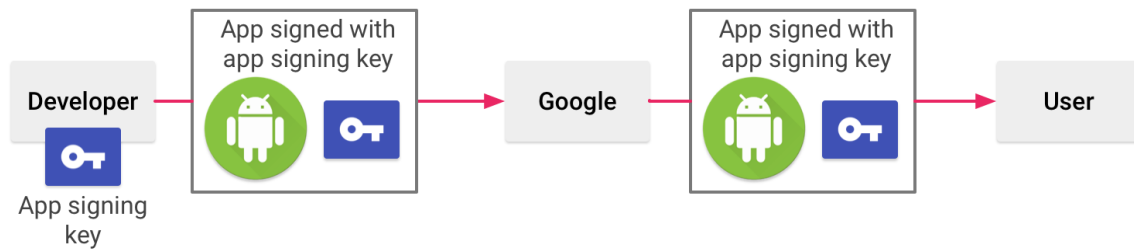
```
zipalign.exe -p 4 app.apk app-alineada.apk
```

Una vez hecho esto procedemos a firmar la versión de la apk ya alineada:

```
apksigner.bat sign --ks Store2.jks --out app-alineada-firmada.apk app-alineada.apk
```

El archivo `apksigner.bat` es un script de Windows que acaba realizando una llamada a java indicándole que ejecute la aplicación contenida en `apksigner.jar`. Si deseas profundizar en esta temática consulta la URL: <https://developer.android.com/studio/publish/app-signing?hl=es-419>. Allí podrás comprobar que tal y como muestra la siguiente imagen, una vez alineada y firmada, la

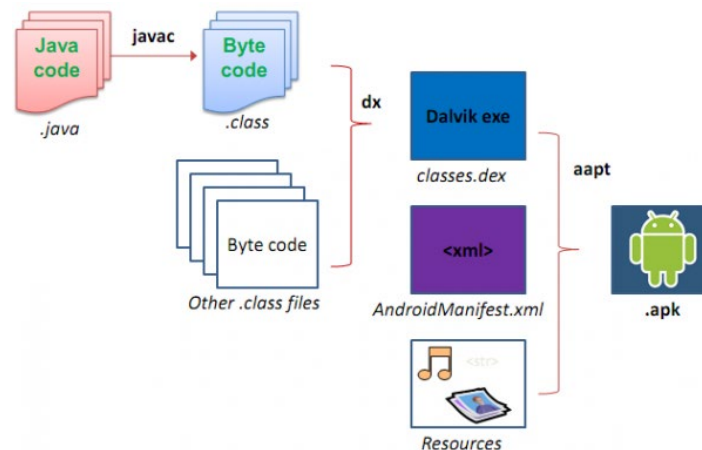
app está lista para ser distribuida a través de Google Play, tras una serie de comprobaciones que Google debe realizar.



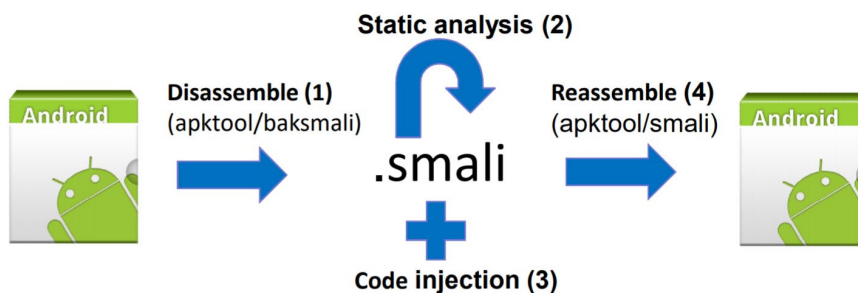
### 3. Desensamblado, modificación y reensamblado de apks

Todo apk que podamos descargarnos de Google Play o de cualquier otra tienda de aplicaciones puede ser desensamblado con el objetivo de conocer su estructura y programación, modificarla si se desea y reensamblado para producir una nueva apk funcional. Este proceso, que resulta de gran interés para la verificación y mantenimiento de apps Android, es también un proceso de riesgo si se realiza con fines malintencionados y, de hecho, constituye el proceso básico de hacking para este tipo de aplicaciones.

Tal y como muestra la siguiente figura, el proyecto original programado en Java de una app experimenta varias transformaciones hasta producir el fichero apk final que se utiliza para instalar dicha app en un dispositivo móvil.



Sin embargo, estas transformaciones pueden revertirse y volverse a realizar. En la siguiente figura observamos los 4 pasos que podemos seguir para llevar a cabo este proceso, lo que puede servir no sólo para estudiar la estructura de la app y su comportamiento, sino también para modificarlos (fraudulentamente o no).



El código smali es a la máquina virtual de Android, lo que el ensamblador es a una máquina física convencional. Como vemos, el proceso de desensamblado y ensamblado es soportado por la herramienta apktool a través de las aplicaciones baksmali y smali respectivamente. Una vez desensamblado el código de la aplicación, éste puede analizarse (paso 2) y modificarse (paso 3) convenientemente antes de desensamblarse para producir nuevamente un apk funcional. Estudiemos a continuación paso a paso este proceso.

**NOTA:** Aunque en esta práctica nos centraremos en el uso de la herramienta de desensamblado y reensamblado de apks más conocida y utilizada actualmente, la herramienta ApkTool, cabe señalar que existen multitud de alternativas a ésta, como APK Easy Tool o Advanced ApkTool, entre otras.

### 3.1. Desensamblado de apks (unpacking)

La aplicación ApkTool se distribuye desde la web <https://ibotpeaches.github.io/Apktool> y evoluciona a medida que lo hacen las distintas versiones de Android.

```
C:\Users\jcruiz\Desktop\CDM\Apktool>apktool.bat
Apktool v2.3.4 - a tool for reengineering Android apk files
with smali v2.2.2 and baksmali v2.2.2
Copyright 2014 Ryszard Wisniewski <brut.all@gmail.com>
Updated by Connor Tumbleson <connor.tumbleson@gmail.com>

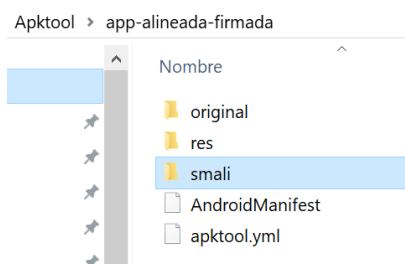
usage: apktool
  -advance,--advanced    prints advance information.
  -version,--version      prints the version then exits
usage: apktool if|install-framework [options] <framework.apk>
  -p,--frame-path <dir>  Stores framework files into <dir>.
  -t,--tag <tag>         Tag frameworks using <tag>.
usage: apktool d[ecode] [options] <file_apk>
  -f,--force             Force delete destination directory.
  -o,--output <dir>      The name of folder that gets written. Default is apk.out
  -p,--frame-path <dir>  Uses framework files located in <dir>.
  -r,--no-res            Do not decode resources.
  -s,--no-src            Do not decode sources.
  -t,--frame-tag <tag>   Uses framework files tagged by <tag>.
usage: apktool b[uild] [options] <app_path>
  -f,--force-all        Skip changes detection and build all files.
  -o,--output <dir>      The name of apk that gets written. Default is dist/name.apk
  -p,--frame-path <dir>  Uses framework files located in <dir>.

For additional info, see: http://ibotpeaches.github.io/Apktool/
For smali/baksmali info, see: https://github.com/JesusFreke/smali
```

El desensamblado de un apk con esta herramienta es de lo más sencillo y simplemente requiere del uso de la opción *d* y de la especificación del apk a desensamblar:

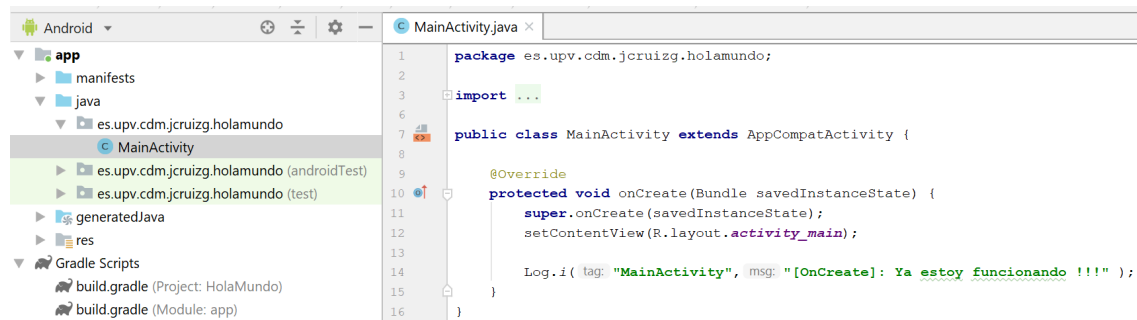
```
C:\Users\jcruiz\Desktop\CDM\Apktool>apktool.bat d apk\app-alineada-firmada.apk
I: Using Apktool 2.3.4 on app-alineada-firmada.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
S: WARNING: Could not write to (C:\Users\jcruiz\AppData\Local\apktool\framework), using C:\Users\jcruiz\AppData\Local\Temp\ instead...
S: Please be aware this is a volatile directory and frameworks could go missing, please utilize --frame-path if the default storage directory is unavailable
I: Loading resource table from file: C:\Users\jcruiz\AppData\Local\Temp\1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Se generará un directorio que recibirá el mismo nombre que el fichero apk que ha sido desensamblado y que contendrá la siguiente información:

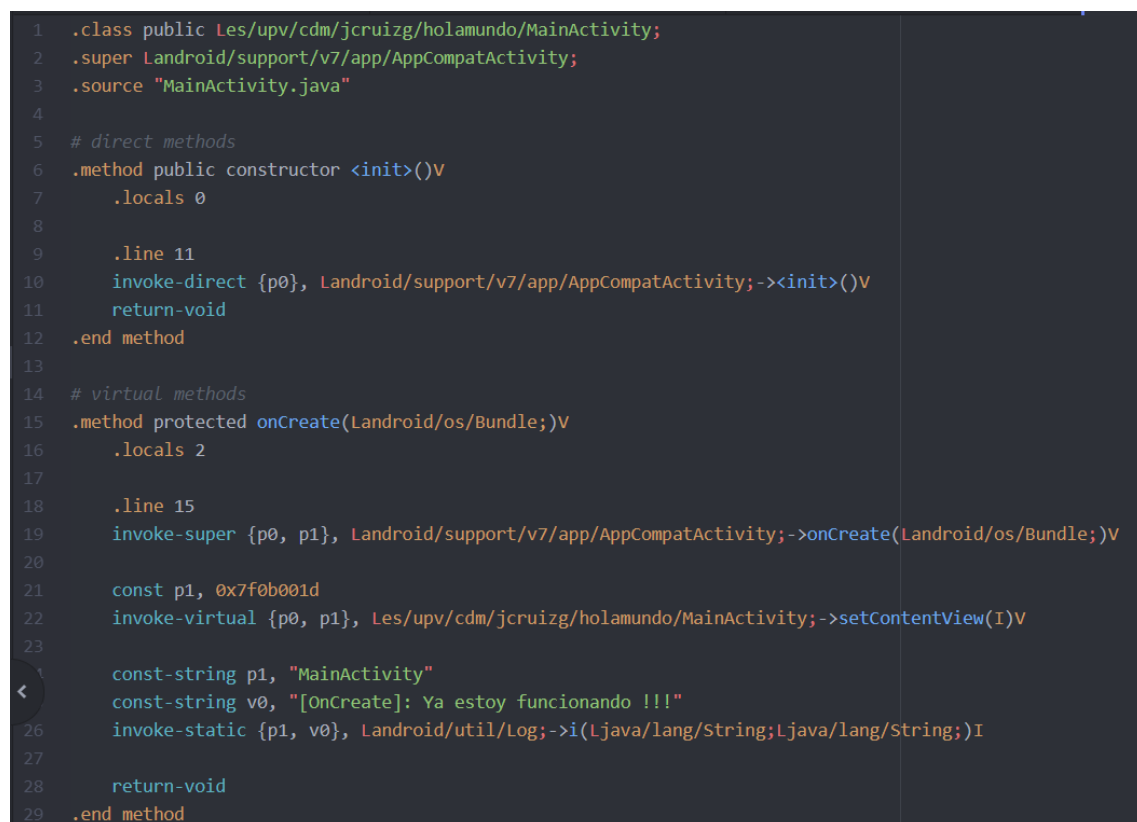




En este directorio encontraremos el Manifiesto del apk, sus recursos (directorio *res*) y el código smali de sus clases (directorio *smali*). Por ejemplo, supongamos que la actividad principal de nuestra aplicación tiene este código:



El código *Smali* generado para esta actividad se encontrará en el fichero *smali\es\upv\cdm\jcruizg\holamundo\MainActivity.smali* y será el siguiente:



### 3.2. Análisis de código smali

Existe un plug-in llamado *smalidea* que permite la depuración de código smali desde el propio Android Studio. En esta sección de la práctica no vamos a utilizar este plug-in, pero si estás interesado en utilizarlo encontrarás información al respecto en la siguiente URL: <https://crops.net/blog/software-development/mobile/android/android-reverse-engineering-debugging-smali-using-smalidea>.

El análisis que se va a realizar a continuación se basa en la información que ofrece la WIKI oficial de smali, accesible en: <https://github.com/JesusFreke/smali/wiki>.

Vemos que en el ejemplo anteriormente introducido el código:

```
Log.i( tag: "MainActivity", msg: "[OnCreate]: Ya estoy funcionando !!!" );
```

Se ha traducido en:

```
29     const-string p1, "MainActivity"
30
31     const-string v0, "[OnCreate]: Ya estoy funcionando !!!"
32
33     .line 14
34     invoke-static {p1, v0}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I
```

Observamos que siempre que se hace referencia a una clase, la referencia comienza por “L” y termina por “;” como en el caso de “*Landroid/util/Log;*”. Por otra parte, la llamada al método *Log.i* se traduce en smali en una invocación de tipo *invoke-static* a la que se pasa como argumentos los parámetros *p1* y *v0*. Tal y como se explica en <https://github.com/JesusFreke/smali/wiki/Registers> en la máquina virtual los registros son siempre de 32 bits y no están tipados, es decir que pueden recibir información de cualquier tipo. En un método la directiva “*.registers*” permite especificar el número de registros que utiliza un método, incluyendo sus parámetros y variables locales, mientras que la directiva “*.locals*” permite indicar cuantas variables locales se van a utilizar, sin contar los parámetros. Esta doble manera de llamar a las variables de un método nos lleva a que tengamos dos maneras distintas de nombrarlos. Imaginemos que un método especifica “*.registers 5*” y que recibe 2 parámetros. Esto significa que utiliza 5 registros (*v0-v4*), de los cuales 2 serán registros locales (*v0-v1*), y 3 serán registros de parámetros (*v2-v4* aunque también podrán referenciarse como *p0-p3*). Hay que señalar que el parámetro 0 (*p0*) siempre existe y hará referencia al *this* del objeto. El resto de parámetros son los que el método reciba. En resumen, podremos referenciar a los registros del método con los nombres que aparecen a continuación:

Local	Param	
v0		the first local register
v1		the second local register
v2	p0	the first parameter register
v3	p1	the second parameter register
v4	p2	the third parameter register

En el código smali proporcionado en la página anterior, vemos que el método *onCreate* recibe un parámetro de tipo *Landroid/os/Bundle;* y declara el uso de *.locals 1*, es decir, de una única variable local. Por eso el código del método se sirve de la variable *v0*, y de los parámetros *p0* (recuerda *this*) y *p1* (el *Bundle* que se recibe como argumento).

También cabe señalar que en la llamada *Log.i*, que recordemos que en smali es así:

```
33     .line 14
34     invoke-static {p1, v0}, Landroid/util/Log;->i(Ljava/lang/String;Ljava/lang/String;)I
```

Se utilizan 2 parámetros *p1* y *v0*, previamente definidos como strings. La llamada en sí termina con una *I* al final de su definición. Esto significa que el método invocado *Log.i* retorna un valor entero.

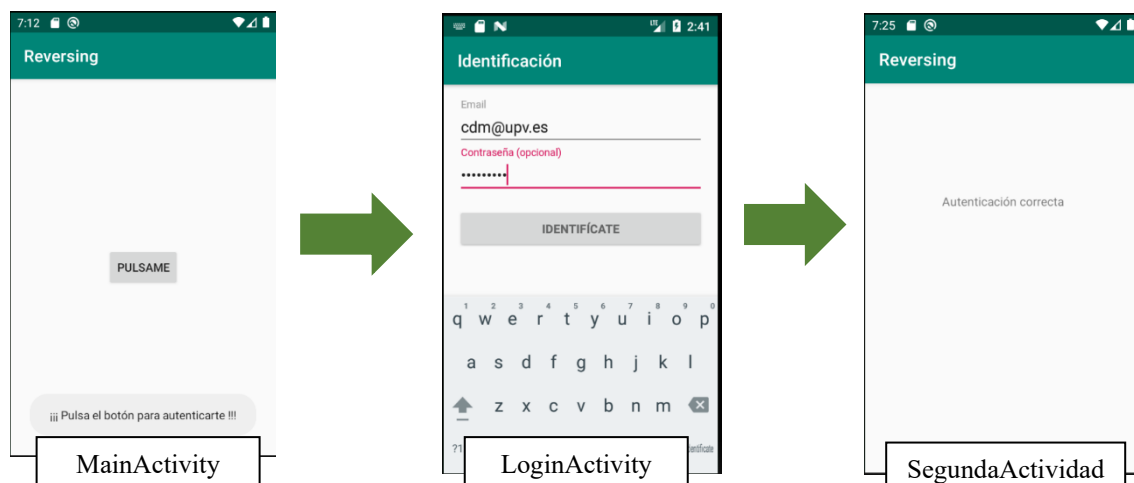
Para finalizar cabe señalar que aunque hemos dicho que los registros son de 32 bits, algunos de los tipos de datos soportados por la máquina virtual, como son los tipos *Long* y *Double* son datos de 64 bits, con lo que requieren del uso de 2 registros (consecutivos) para su representación. Los siguientes son los tipos de datos básicos soportados por una máquina virtual Android standard:

V	void - can only be used for return types
Z	boolean
B	byte
S	short
C	char
I	int
J	long (64 bits)
F	float
D	double (64 bits)

Lo ideal es generar ejemplos sencillos de código para estudiar su equivalencia en smali y así aprender.

Por ejemplo, ya hemos visto cuál es el código necesario para generar un mensaje de log de información. Esto nos puede resultar muy útil cuando estamos depurando una app que creemos maliciosa e intentamos comprender lo que está haciendo. Insertando estos sencillos logs podremos ejecutar la app en el emulador y obtener una traza de la misma, aún sin tener su código fuente.

Consideremos ahora la siguiente app formada por 3 actividades:



La app sólo autentica positivamente a dos usuarios [cdm@upv.es](#) (con contraseña *holamundo*) y [reversing@upv.es](#) (con contraseña *mundohola*).

Ahora lo que vamos a hacer es ver qué código se genera cuando mostramos un Toast en una actividad *MainActivity*. El código Java sería este:

```
Toast.makeText(getApplicationContext(), text: "Pulsa el botón para autenticarte", Toast.LENGTH_LONG).show();
```

Y su equivalente en smali será este:

```

36 .line 19
37 invoke-virtual {p0}, Landroid/content/Context;->getApplicationContext()Landroid/content/Context;
38 move-result-object p1
39 const-string v0, "Pulsa el botón para autenticarte"
40 const/4 v1, 0x1
41 invoke-static {p1, v0, v1}, Landroid/widget/Toast;->makeText(Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;
42 move-result-object p1
43 invoke-virtual {p1}, Landroid/widget/Toast;->show()V
  
```

Si ahora observamos el código del método *onClick* del botón de *MainActivity* vemos que el código Java:

```
@Override
public void onClick(View v) {
    startActivity(new Intent(getApplicationContext(), LoginActivity.class));
}
```

Se transforma en el siguiente código smali:

```
37 # virtual methods
38 .method public onClick(Landroid/view/View;)V
39     .locals 3
40
41     .line 26
42     iget-object p1, p0, Les/upv/cdm/jcruizg/holamundo/MainActivity$1;->this$0:Les/upv/cdm/jcruizg/holamundo/MainActivity;
43
44     new-instance v0, Landroid/content/Intent;
45
46     invoke-virtual {p1}, Les/upv/cdm/jcruizg/holamundo/MainActivity;->getApplicationContext()Landroid/content/Context;
47     move-result-object v1
48     const-class v2, Les/upv/cdm/jcruizg/holamundo/LoginActivity;
49     invoke-direct {v0, v1, v2}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V
50     invoke-virtual {p1, v0}, Les/upv/cdm/jcruizg/holamundo/MainActivity;->startActivity(Landroid/content/Intent;)V
51
52     return-void
53 .end method
```

En este código vemos cómo procedemos para activar la actividad de login (*LoginActivity*).

### 3.3. Modificación del apk

El análisis y comprensión del código smali del ejemplo nos posibilita realizar el *hackeo* de la app bajo estudio con la simple modificación de dicho código smali. Obviamente las modificaciones posibles son múltiples con lo estudiado. El objetivo de esta sección no es el de ser exhaustivo sino el de dar un ejemplo de cómo dicha modificación puede llevarse a cabo. Vamos a *hackear* nuestro ejemplo para que la pulsación del botón de la actividad *MainActivity* se salte el proceso de autenticación y active directamente la actividad *SegundaActividad*.

Haciendo un mínimo cambio en el código del método *onClick* del botón de la actividad *MainActivity* conseguiremos el efecto deseado:

```
37 # virtual methods
38 .method public onClick(Landroid/view/View;)V
39     .locals 3
40
41     .line 26
42     iget-object p1, p0, Les/upv/cdm/jcruizg/holamundo/MainActivity$1;->this$0:Les/upv/cdm/jcruizg/holamundo/MainActivity;
43
44     new-instance v0, Landroid/content/Intent;
45
46     invoke-virtual {p1}, Les/upv/cdm/jcruizg/holamundo/MainActivity;->getApplicationContext()Landroid/content/Context;
47     move-result-object v1
48     const-class v2, Les/upv/cdm/jcruizg/holamundo/SegundaActividad;
49     invoke-direct {v0, v1, v2}, Landroid/content/Intent;-><init>(Landroid/content/Context;Ljava/lang/Class;)V
50     invoke-virtual {p1, v0}, Les/upv/cdm/jcruizg/holamundo/MainActivity;->startActivity(Landroid/content/Intent;)V
51
52     return-void
53 .end method
```

Obviamente, la app podría modificarse de muchas otras formas maliciosas, como por ejemplo para conseguir las credenciales (login y contraseña) de los distintos usuarios que instalen y utilicen la app, o para eliminar la publicidad que la app pudiera mostrar con el fin de monetizar su desarrollo. Algunas de estas modificaciones pueden no sólo

comportar la modificación del código (smali) de la app, sino también de su manifiesto, layouts o recursos.

Cabe señalar que un usuario podría también servirse del unpacking y repacking de la app con fines completamente lícitos, como para auditar dicha app, mejorarla app o simplemente mantenerla. Y todo esto aún cuando no se disponga del código fuente de dicha app.

### 3.4. Reensamblado del apk (repacking)

Sea cual sea la modificación introducida en la app, ésta debe reensamblarse (algunos llaman a esto recompilarse) con el objetivo de volver a producir una apk funcional. Para ello volvemos a servirnos de la utilidad apktool como sigue:

```
apktool.bat b -o dist/cdm-prac3-hacked.apk cdm-prac3
```

Estamos asumiendo que la app *cdm-prac3.apk* ha sido decompilada en el directorio *cdm-prac3* y que la modificación se ha realizado en el código smali de dicho directorio. En la orden hemos solicitado a *apktool* que reensamble el proyecto y lo deposite (opción *-o*) en el directorio *dist* bajo el nombre *cdm-prac3-hacked.apk*.

Obviamente, el fichero resultante no puede utilizarse ni en el emulador ni en el dispositivo. Para ello, primero el fichero debe alinearse:

```
zipalign.exe -p 4 dist/cdm-prac3-hacked.apk dist/cdm-prac3-hacked-alineada.apk
```

Y luego firmarse:

```
apksigner.bat sign --ks Store2.jks --out dist/cdm-prac3-hacked-alineada-firmada.apk dist/cdm-prac3-hacked-alineada.apk
```

A continuación, eliminaremos la app del dispositivo:

```
C:\Users\jcruihg\Desktop\CDM\Apktool>adb uninstall es.upv.cdm.jcruihg.holamundo
Success
```

Y luego instalaremos su versión modificada, con cuidado de instalar la versión modificada, pero también alineada y firmada de la app:

```
C:\Users\jcruihg\Desktop\CDM\Apktool>adb install dist\cdm-prac3-hacked.apk
adb: failed to install dist\cdm-prac3-hacked.apk: Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES: Failed to collect certificates]
C:\Users\jcruihg\Desktop\CDM\Apktool>adb install dist\cdm-prac3-hacked-alineada-firmada.apk
Success
```

### 3.5. Cuidado con los heurísticos de evasión

Hay que tener mucho cuidado con el código que desensamblamos y analizamos ya que puede parecer benigno cuando se ejecute en el emulador y luego comportarse de manera muy peligrosa cuando la app corre en un dispositivo real. Esto se debe a que la app implementa un *heurístico de evasión*, es decir, su código detecta si la app se está ejecutando en un emulador, smartphone o tableta y especializa su ejecución en consecuencia. Esto podría programarse de múltiples formas, he aquí una de ellas:

```
String device = Build.DEVICE;
if (device.contains("generic")) {
    //Comportamiento al ejecutar el código en un emulador
    Log.i( tag: "[MainActivity]", msg: "onCreate: Ejecutas el código en un emulador");
}
else {
    //Comportamiento al ejecutar el código en un dispositivo real
    Log.i( tag: "[MainActivity]", msg: "onCreate: Ejecutas el código en un dispositivo real");
}
```

Lo que se traducirá en el siguiente código smali:

```
.line 26
#String device = Build.DEVICE; (p1 es device)
sget-object p1, Landroid/os/Build;:->DEVICE:Ljava/lang/String;
# v0 representa al string "generic"
const-string v0, "generic"
.line 27
#Comprobamos si p1 contiene a v0
invoke-virtual {p1, v0}, Ljava/lang/String;:->contains(Ljava/lang/CharSequence;)Z
move-result p1 #Movemos al registro p1 el resultado de la comprobación
#que es un Booleano (tipo Z)
if-eqz p1, :cond_0
#Si p1 es 1 (condición TRUE, por tanto Build.DEVICE contiene "generic")
const-string p1, "[MainActivity]"
const-string v0, "onCreate: Ejecutas el c\u00f3digo en un emulador"
invoke-static {p1, v0}, Landroid/util/Log;:->i(Ljava/lang/String;Ljava/lang/String;)I
return-void
:cond_0
#Saltamos a cond_0 si p1 es 0 (condición FALSE, por tanto Build.DEVICE no contiene "generic")
const-string p1, "[MainActivity]"
const-string v0, "onCreate: Ejecutas el c\u00f3digo en un dispositivo real"
invoke-static {p1, v0}, Landroid/util/Log;:->i(Ljava/lang/String;Ljava/lang/String;)I
return-void
```

Observa que en el código hay una instrucción de salto condicional, *if-eqz p1, :cond\_0*, con lo que en función del valor de ese registro saltaremos o no a la etiqueta *:cond\_0*. Como vemos el código smali no es otra cosa que el código ensamblador para la máquina virtual Dalvik. Los códigos de operación de dicha máquina virtual y su significado pueden encontrarse en: [http://pallergabor.uw.hu/androidblog/dalvik\\_opcodes.html](http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html).

Existen herramientas que se diseñan para detectar en los APKs algunos patrones que denotan comportamientos fraudulentos. Algunas de estas herramientas están disponibles online y, sin ánimo de ser exhaustivos, mencionaremos algunas de ellas, como <https://apkscan.nviso.be>, <https://andrototal.org>, <https://metadefender.opswat.com> o <https://www.virustotal.com>.

#### 4. ¿Y qué ocurre cuando se utilizan ofuscadore de código?

Es muy común que los programadores utilicen herramienta para ofuscar el código de sus aplicaciones y hacerlas menos legibles. Esto significa que, en lugar de tener nombres comprensibles, las clases y sus métodos reciben nombres generados de manera aleatoria que mezclan letras y/o números y/o otros caracteres. Lo que se persigue es que alguien de decompile el código no pueda leerlo de manera sencilla. Pero hay que señalar que esto no evita que el apk de una app pueda decompilarse y recompilarse. Y desde la perspectiva del reversing, es decir, del análisis de la app, sólo ralentiza el proceso, no lo impide.

En Android el ofuscador por defecto es ProGuard. Para habilitar el uso del ofuscador de código en un proyecto de desarrollo Android simplemente tendremos que acceder al fichero *build.gradle* de la app y cambiar dentro del *BuildTypes* de tipo *release* el campo *minifyEnabled* de *false* (su valor por defecto) a *true*.





Este cambio también impactará en el tamaño del apk que obtengamos cuando compilemos nuestro apk ya que éste se optimizará y se verá reducido en la mayor parte de los casos. Sin embargo, y tal y como ya hemos mencionado, el apk resultante podrá, a pesar de las ofuscaciones realizadas, ser desensamblado y recompilado siguiendo el mismo proceso que hemos estudiado.

Para más información al respecto de la ofuscación de código en Android Studio podéis consultar la URL: <https://developer.android.com/studio/build/shrink-code?hl=es-419>.

## 5. Reseñas finales

Como hemos visto en esta práctica, las herramientas de decompilación (desensamblado) y ensamblado (build o reensamblado) no sólo ofrecen la posibilidad de estudiar la estructura y comportamiento de una app móvil sino que también permiten modificar su código y producir un nuevo apk que puede probarse en un dispositivo o emulador.

Sin embargo, hay que entender que este proceso de deconstrucción y reensamblado tiene un coste y no siempre funciona todo lo bien que desearíamos. En efecto, no sólo en ocasiones trabajaremos con apks que no podrán decompilarse, sino que en ocasiones, cuando reensemblemos la app una vez procesada, su apk puede no funcionar exactamente como lo hacía el original. Esto ocurre, por ejemplo, cuando la app utiliza una interfaz que requiere del uso de alguna clave ligada al apk generado y que es el nuestro. En otras palabras, mientras que el apk original estaba firmado con la clave original del desarrollador de la app, el que hemos producido nosotros está firmado con una clave nuestra, y por tanto no original. En consecuencia, toda API que verifique la clave con la que se ha firmado el apk constatará que el app ha sido modificada y, por tanto, no ofrecerá su servicio. Esto es lo que pasa con las APIs de Google, como la que se utiliza para acceder a los mapas. Un ejemplo de esto se puede apreciar si descargamos el apk de la EMT de Valencia (<https://apkpure.com/es/emt-valencia/es.emtvalencia.emt>). Si la descargamos, decompilamos y reensamblamos, veremos que somos incapaces de utilizar los mapas y la geolocalización sobre los mismos. Podemos decir, por tanto, que el proceso que hemos estudiado en esta práctica es un proceso no inocuo que puede llegar a romper una app.

Otro aspecto a tener en cuenta en el proceso que hemos estudiado es la revisión de código en lenguaje smali, que no es algo precisamente fácil. Por ello existen distintos decompiladores que son capaces de interpretar el código smali para producir un código de alto nivel (java normalmente) o lo más parecido al mismo. A título de ejemplo mencionar que es posible utilizar la herramienta dex2jar (<https://github.com/pxb1988/dex2jar>) para obtener una versión en formato jar del apk de una app, que luego podrá ser leído por un decompilador como jd-gui (<http://java-decompiler.github.io>). El uso de este tipo de herramienta nos ayudará mucho a la hora de leer e interpretar el código smali bajo estudio.

## 6. Ejercicio propuesto

Junto a este enunciado se ha suministrado un apk de ejemplo con el que trabajaremos. El comportamiento del apk es el descrito en la sección 3.2 de esta práctica.

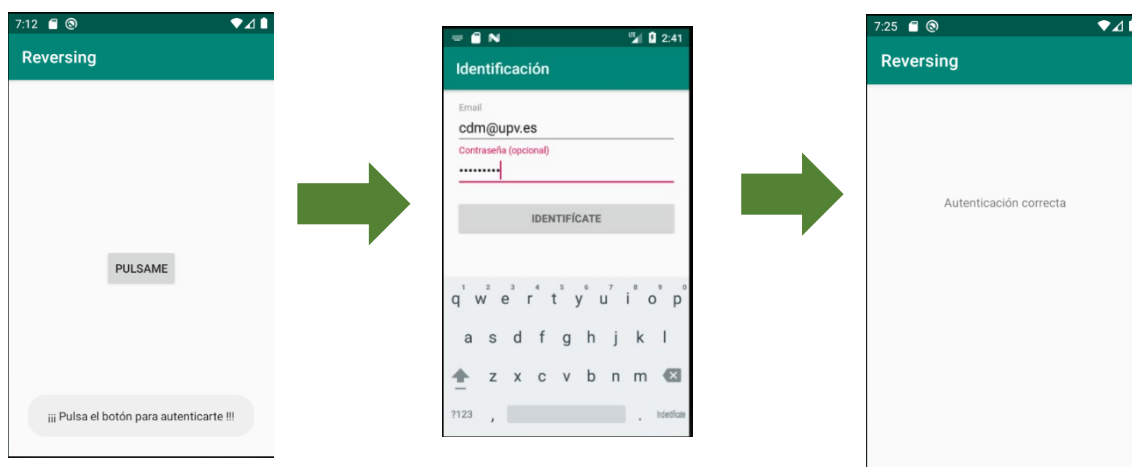
Lo que se pide es:

- Desensamblar utilizando la herramienta apktool el apk suministrado y examinar su código smali y su manifiesto. Recuerda que la herramienta está disponible en descarga desde <https://ibotpeaches.github.io/Apktool/>.

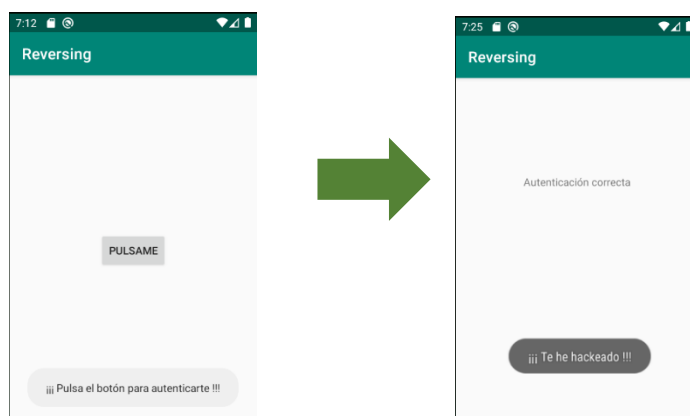
Ten en cuenta que la herramienta no es otra cosa que un fichero jar, y se suministra junto con un script que encapsula su uso. Las instrucciones a seguir para utilizar correctamente la herramienta son las siguientes y se pueden encontrar en su web (rúbrica instalación):

1. Download Linux wrapper script (Right click, Save Link As `apktool`)
2. Download apktool-2 (find newest here)
3. Rename downloaded jar to `apktool.jar`
4. Move both files (`apktool.jar` & `apktool`) to `/usr/local/bin` (root needed)
5. Make sure both files are executable (`chmod +x`)
6. Try running `apktool` via cli

- Modificar el apk para que implemente un heurístico de evasión como el visto el descrito en el apartado 3.5 de esta práctica. Por tanto, si el apk se ejecuta en un emulador su comportamiento deberá ser el exhibido por el apk original:



Recordad que el apk sólo autentica con dos tuplas *usuario/contraseña* “*cdm@upv.es / holamundo*” y “*reversing@upv.es / mundohola*”. Sin embargo, cuando el apk pase a ejecutarse en un dispositivo real su comportamiento deberá ser el siguiente:





Recuerda que para poder ser ejecutado el APK no sólo debe generarse, sino también alinearse y firmarse. Para hacer pruebas en un dispositivo real primero averiguaremos su identificador, luego desinstalaremos el APK si ya estuviera instalado y finalmente lo reinstalamos:

```
Símbolo del sistema

C:\Users\jcruizg\Desktop\CDM\Apktool\dist>adb devices
List of devices attached
ZX1G425MM8      device
emulator-5554   device

C:\Users\jcruizg\Desktop\CDM\Apktool\dist>adb -s ZX1G425MM8 uninstall es.upv.cdm.jcruizg.holamundo
Success

C:\Users\jcruizg\Desktop\CDM\Apktool\dist>adb -s ZX1G425MM8 install .\app-h-a-s.apk
Success

C:\Users\jcruizg\Desktop\CDM\Apktool\dist>
```

- Reensambla el apk e inclúyelo en el zip que entregues como memoria de esta práctica. El fichero ZIP que constituirá la memoria de la práctica, y que deberá subirse al espacio compartido de *poliformaT*, deberá incluir lo siguiente:
  - Apk generado con la solución;
  - Un documento pdf en el que se explique cuál ha sido el proceso seguido para la generación del apk que se presenta como solución del ejercicio. El documento deberá explicar todos los pasos realizados e incluir capturas de pantalla que ilustren todos esos pasos. También deberá incluirse el código smali que se genere para adaptar el comportamiento de la app al deseado para esta práctica.
- Un consejo, en lugar de hacer todo esto a mano cada vez se aconseja utilizar APK Studio, que una vez instaladas las herramientas, sistematiza el proceso de creación del apk (Build), firma (Sign) e incluso instalación (Install). La alineación de la app la realiza de manera implícita la aplicación de firma *uber-apk-sign* que APK studio utiliza, y que deberás instalarte también.

### **Pistas de ayuda a la realización de la práctica**

- Centra tus esfuerzos en modificar 2 actividades, la actividad principal de la app, que es la que presenta el botón que dice *PULSAME*, y la actividad a la que accede el usuario una vez autenticado. Utiliza el manifiesto de la app para determinar el nombre de estas actividades y la ubicación de sus archivos *smali*.
- En el caso de la actividad principal fíjate bien, porque al ser desensamblada, el código de la actividad se distribuirá entre 2 ficheros con extensión *smali*.
- Conseguirás la funcionalidad deseada si editas el código smali de la actividad principal y en respuesta a una pulsación del botón *PULSAME* insertas el código smali necesario para implementar la siguiente funcionalidad:

```
public void onClick(View v) {  
  
    String device = Build.DEVICE;  
    if (device.contains("generic")) {  
        //Comportamiento al ejecutar el código en un emulador  
        Log.i( tag: "[MainActivity]", msg: "Ejecutas el código en un emulador");  
        startActivity(new Intent(getApplicationContext(), LoginActivity.class));  
    }  
    else {  
        //Comportamiento al ejecutar el código en un dispositivo real  
        Log.i( tag: "[MainActivity]", msg: "Ejecutas el código en un dispositivo real");  
        startActivity(new Intent(getApplicationContext(), SegundaActividad.class));  
    }  
}
```

**NOTA:** No empieces de cero, porque el código casi se te da ya en el código existente, sólo tendrás que comprender su funcionamiento y adaptarlo en consecuencia. Además, tienes muchos ejemplos en la memoria de esta práctica sobre cómo debes proceder en cada caso. En caso de duda es recomendable desarrollar el heurístico en un proyecto aparte y reutilizar el código smali que resulte de la aplicación, o al menos basarse en él.

- Además del de la actividad principal, también deberás modificar el código smali de la actividad a la que acceden los usuarios autenticados. En ella simplemente deberás insertar el código necesario para mostrar un mensaje de tipo Toast que diga “*!!! Te he hackeado !!!*”. Ten en cuenta que la codificación UNICODE del carácter *¡* es *\u00a1*, aunque eso puedes constatarlo ya en el mensaje que muestra la actividad principal cuando se lanza a ejecución y que dice “*!!! Pulsa el botón para autenticarte !!!*” y que está en uno de los ficheros smali ya generados.
- Finalmente, un consejo. Lee con atención el enunciado de la práctica y recuerda que no sólo debes introducir el código que falte, sino que debes adaptar (mínimamente, pero hay que hacerlo) el código ya existente. Esto ocurre, por ejemplo, cuando utilices más registros de los que use ya el código existente en el método que modifiques. En ese caso, deberás modificar la directiva *.locals* del método en consecuencia (leer la sección 3.2 para más información).