ANDROID : REVERSE ENGINEERING

Benjamin Zores [Directeur technique @ Alcatel-Lucent Enterprise]

Si le système d'exploitation Android (AOSP pour les intimes), tel que proposé par Google, se veut libre et Open Source, ce n'est pas forcément le cas pour l'écosystème applicatif le constituant. Pour autant, voyons comment il est possible de jouer les curieux ;-)

Mots-clés : Android, Rétro-Ingénierie, Compilation, Mobile, Applications, Java

Résumé

Si les applications Android sont écrites en Java, elles sont néanmoins compilées sous la forme d'un *bytecode* Dalvik, exécuté par une machine virtuelle dédiée. Les capacités d'introspection du langage Java rendent cependant aisée sa rétro-ingénierie, permettant ainsi de remonter à tout ou partie des sources Java originelles à partir d'une application Android du PlayStore. Au travers de cet article, nous reviendrons donc sur la façon dont les applications sont compilées, et verrons ensuite par le biais d'un exemple pratique (le jeu libre Frozen Bubble) comment il est possible de le décompiler, pour en modifier le contenu et ainsi démarrer le jeu à un niveau avancé. Vous apprendrez ainsi comment retrouver le code Java à partir d'une archive APK et à modifier directement le code assembleur Smali généré.

Avant toute chose, sachez que le principe de rétro-ingénierie constitue une pratique interdite en France. Cet article n'a donc qu'un but pédagogique. Loin de moi donc l'idée (où les capacités d'ailleurs) de vous expliquer comment « hacker » le système ou encore ses applications. Il est de bon ton cependant, pour un développeur d'applications Android de savoir ce que font les outils qui sont mis à sa disposition et ce qu'il est possible de faire pour quelqu'un de mal intentionné. Java n'est clairement pas mon langage de prédilection. Pour autant, sa capacité d'introspection rend sa décompilation très facile et fort pratique. Cela tombe bien, c'est ainsi que sont écrites les applications Android. Alors, pourquoi vouloir décompiler une application ?

Il peut s'agir de votre propre application. Vous voulez ainsi comprendre ce que quelqu'un d'autre peut y découvrir. Vous pensiez avoir protégé votre API ou votre code ? Ce n'est hélas pas si sûr. Vous pouvez aussi décider de décompiler une application tierce, qu'elle soit libre ou propriétaire. Plusieurs raisons me viennent en tête. La curiosité en premier lieu. Android a beau afficher explicitement l'ensemble des permissions requises par une application à son installation, cela ne vous dira pas exactement ce que le code va faire. Vous seriez surpris de voir le nombre de paquetages liés à de la publicité Facebook ou GooglePlus qui peuvent faire partie des applications du PlayStore. Vous pouvez également vouloir modifier une application pour votre propre usage : ajouter ou modifier des traductions, changer le contenu de l'application (chaînes de caractères, logos, images, ressources audio ...) ou encore modifier le code source pour rendre vos jeux plus simples ... Sans parler à proprement parler de « hacking » (ou plutôt « cracking »), cela s'apparente davantage à du « modding », l'art de

personnaliser une application. Vous le verrez dans cet article, nous allons illustrer tout cela autour d'un jeu Français, très connu et tout ce qu'il y a de plus libre, à savoir le bien nommé **Frozen Bubble [1]**. L'idée toute simple sera de faire en sorte que le jeu commence au niveau 5 plutôt qu'au premier niveau. Rien de bien méchant ni de révolutionnaire donc ;-)

1 Java, DEX et APK

Pour les moins familiers d'entre vous, revenons quelques instants sur les bases d'une application Android. Une application se compose d'un binaire, d'éventuelles librairies partagées, natives à la plate-forme sur laquelle elle est supposée tournée et de données (textuelles, multimédia ...), généralement appelées « ressources ». Il n'y a pas à proprement parler d'exécutable. Android exécute une instance de machine virtuelle pseudo-Java (appelée **Dalvik** et **ART** depuis Android 5.0) qui exécute du code-machine. La transformation de l'état de sources à celle d'exécutable Dalvik se passe conformément à la figure 1, que nous allons détailler tout de suite.

Le code source d'une application est écrit en Java. Il s'agit donc d'un ensemble de fichiers `.java` qui peuvent utiliser les paquetages du projet **Apache Harmony** (une implémentation libre de Java, pour contrer Oracle) ainsi que les paquetages du SDK Android. Ces fichiers sources Java sont ensuite compilés en *bytecode* Java au moyen du compilateur `javac` pour devenir des objets et fichiers `.class`. Android dispose cependant de sa propre JVM. Les `.class`, habituellement concaténés dans une archive JAR, ne le sont donc pas sous Android. Au contraire, au moyen de l'exécutable « `dx` », les `.class` de votre application ainsi que les classes des librairies tierces que vous intégrez seront ainsi concaténées, fusionnées et optimisées pour devenir du *bytecode* Dalvik, conforme au système Android et seront archivées sous la forme d'un fichier appelé `classes.dex`. Ce *bytecode* Dalvik répond au doux nom de `DEX` (pour *Dalvik EXecutable*), un pseudo-binaire exécuté par la JVM Dalvik. De la même façon, les ressources (ensemble de fichiers textes, de chaînes de caractères, d'images, de vidéos, de musiques ou encore de traductions) qui composent votre application, ainsi que le fichier `AndroidManifest.xml`, une méta-description de votre application, de ses propriétés, ses capacités, ses privilèges et ses pré-requis sera empaqueté au moyen du pré-processeur `aapt` pour devenir le fichier de ressources. Les références aux ressources sont également mises à disposition dans un fichier Java dit « `R` » qui permettra au code applicatif de pointer sur ces dernières. Enfin, notre fichier `classes.dex` et notre fichier de ressources seront empaquetés dans un fichier `APK` (pour *Android PacKage*), qui facilitera la distribution de votre application, au moyen du programme `apkbuilder`. Si vous êtes curieux et essayez la commande `file` sur un APK, vous vous rendrez très vite compte qu'il ne s'agit ni plus ni moins que d'une archive ZIP. Notez enfin que l'APK généré est brut, ou non-signé. Afin d'être distribué sur le **PlayStore** ou même d'être installé sur votre téléphone, ce dernier devra être signé numériquement, afin de garantir sa provenance et son intégrité. Au moyen de sa clé privée, le développeur signera donc son APK via le programme `jarsigner` qui générera une archive APK équivalente à laquelle aura été adjointe l'authenticité du développeur. L'archive APK résultante pourra alors ensuite être mise à disposition sur le Store ou déployée sur votre téléphone.

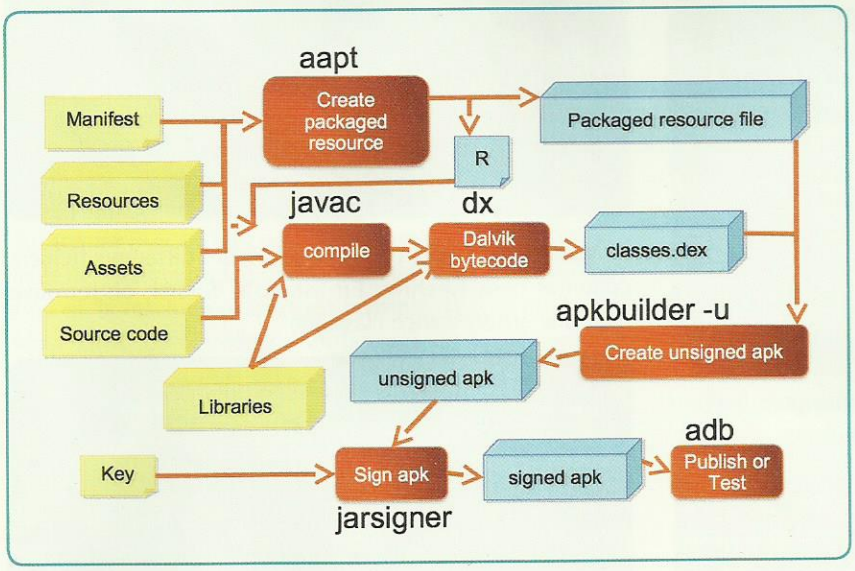

Figure 1 : Cheminement de la construction d'une archive APK.

Lors de l'installation sur votre téléphone, le système Android, via le `PackageManager`, se chargera de dépaqueter l'archive APK, d'en vérifier la signature et l'intégrité, d'en extraire et installer les ressources et enfin d'en extraire le *bytecode* `DEX`. Ce *bytecode* sera ensuite optimisé par le système au travers du processus `DexOpt` pour devenir un fichier de type `ODEX` (ou

Optimized DEX), qui sera installé dans le cache de la JVM. Le but de cette optimisation est de s'adapter aux contraintes matérielles sur laquelle l'application va effectivement être exécutée afin de garantir de meilleures performances. Avant Android 5.0, cette étape se faisait directement à la compilation (approche **JIT** ou *Just-In-Time*). Depuis Lollipop, cette étape se fait maintenant directement à l'installation (approche **AOT** ou *Ahead-Of-Time*).

C'est un peu compliqué mais vous l'aurez normalement compris, le *bytecode* qui sera donc réellement exécuté sur votre téléphone se veut bien différent de ce que vous aurez écrit en Java (le passage de source Java vers du *bytecode* Java, vers du *bytecode* **DEX** puis vers du *bytecode* **ODEX** spécifique à votre plate-forme pouvant laisser quelques traces). Pour autant, ce processus s'avère assez simplement réversible et il est possible de revenir (à quelques exceptions près) vers les sources originales à partir d'un fichier APK.

2 Mise en Pratique

Commençons donc sans plus attendre. Pour se faire, vous aurez besoin de Java 7 (il fallait s'y attendre), dans sa version JDK, soit propriétaire en provenance d'Oracle, soit dans sa version **OpenJDK**. Je passerai volontiers sur l'installation du JDK si vous me le permettez. Pour vous interfacer avec votre téléphone (par USB), il vous faudra également l'utilitaire **adb** (ou *Android Debug Bridge*). Ce dernier est disponible avec le SDK Android **[2]**. Le cas échéant, sous Ubuntu vous pouvez également le récupérer sous la forme de paquets via PPA :

```
$ sudo add-apt-repository ppa:phablet-team/tools
$ sudo apt-get update
$ sudo apt-get install android-tools-adb android-tools-adbd
```

Il nous est maintenant nécessaire de récupérer notre application de mise en pratique, à savoir l'APK de Frozen Bubble. Si le PlayStore vous permet facilement de choisir et d'installer vos applications sur téléphone ou tablette, rien n'est cependant prévu pour récupérer directement l'APK. Il existe cependant des moyens pour contourner cela. Allez sur le PlayStore et recherchez donc notre jeu. Vous devriez être redirigé vers la page suivante : https://play.google.com/store/apps/details?id=org.jfedor.frozenbubble.

Notez bien l'identifiant de notre jeu. Il s'agit du nom unique de notre paquetage Java, à savoir **org.jfedor.frozenbubble**. Enfin, à titre de référence, notez qu'au moment de cet article, le jeu en est à sa version 3.4. Vous pourrez donc éventuellement voir quelques ajustements si la version venait à changer. Pour télécharger simplement un APK, il vous suffit d'utiliser la plateforme Web **APK Downloader [3]**. Saisissez alors l'identifiant de votre application et le site vous renverra un lien de téléchargement direct de l'APK, rien de plus simple. Notez enfin qu'en plus du site Web, une extension pour Chrome et Firefox est disponible, ajoutant un bouton de téléchargement direct (moyennant configuration de votre compte Google) sur la page du PlayStore.

3 Place au débogage

Nous pouvons maintenant installer notre application via **adb**, directement en USB. Autant vous dire qu'hélas, si cela fonctionne très bien, cela ne va pas nous permettre de débogguer efficacement l'application. En effet, nous allons avoir besoin des symboles de debug activés pour lancer pas à pas notre application et voir ce qui s'y passe (pour mieux la modifier ensuite). Si toutes les applications en provenance du PlayStore sont signées, elles sont malheureusement très souvent aussi compilées en mode *release* et donc sans symboles de debug. Cela ne va pas nous aider. Qu'à cela ne tienne, rendons notre application débogable. Commencez donc par extraire l'application. Souvenez-vous, l'APK est en fait une archive ZIP. Qu'à cela ne tienne :

```
$ cd /tmp
$ unzip FrozenBubble-3.4.apk
Archive: FrozenBubble-3.4.apk
  inflating: assets/levels.txt
 extracting: res/drawable/app_frozen_bubble.png
 extracting: res/drawable/background.jpg
 extracting: res/drawable/background2.jpg
[...]
 extracting: res/drawable-xhdpi/app_frozen_bubble.png
  inflating: classes.dex
  inflating: lib/armeabi/libmodplug-1.0.so
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA
```

Vous pouvez ainsi voir l'extraction des différents fichiers contenus dans l'archive. En listant le dossier, on retrouve bien une arborescence classique :

```
$ ls -l
-rw-rw-r--  1 ben ben   3956 avril 14 10:09 AndroidManifest.xml
drwxrwxr-x  2 ben ben   4096 mai    6 12:49 assets/
-rw-rw-r--  1 ben ben 254048 avril 14 10:09 classes.dex
drwxrwxr-x  3 ben ben   4096 mai    6 12:49 lib/
drwxrwxr-x  2 ben ben   4096 mai    6 12:49 META-INF/
drwxrwxr-x 10 ben ben   4096 mai    6 12:49 res/
-rw-rw-r--  1 ben ben  30840 avril 14 10:09 resources.arsc
```

Comme vu plus tôt, le fichier **classes.dex** contient notre exécutable **DEX** et les répertoires **assets** et **res** contiennent l'ensemble des ressources, référencées dans le fichier **resources.arsc**. Pour être débogguable, une application Android doit disposer du paramètre en question au sein de son fichier **AndroidManifest.xml** :

```
<android xmlns:android="http://schemas.android.com/apk/res/
android">
    <manifest>
        <application android:debuggable="true" />
    </manifest>
</android>
```

Regardez donc le contenu du fichier et, en cas d'absence, ajoutez simplement le paramètre. Vous n'avez pas réussi, n'est ce pas ? En effet, le fichier XML n'est que pur binaire, comme vous pouvez le constater :

```
$ file AndroidManifest.xml
AndroidManifest.xml: data
```

Aux grands maux les grands remèdes, il va falloir utiliser d'autres outils.

4 ApkTool

Laissez-moi vous présenter **APKTool [4]**, un fantastique outil de manipulation d'archives APK et d'aide à la rétro-ingénierie. L'outil permet en effet de décoder les ressources contenues dans une archive APK pour les restituer comme à l'original (le plus souvent) mais aussi de reconstituer une archive APK après lui avoir opéré quelques modifications. L'outil permet également de générer du code **Smali** à partir du *bytecode* **DEX** contenu dans le fichier **classes.dex**, permettant ainsi du débogage pas à pas.

Qu'est ce donc que le Smali **[5]** me direz-vous ? C'est très simple, c'est la version Assembleur du *bytecode* **DEX**. Au même titre qu'en bon développeur C, il vous est possible de coder certaines instructions en Assembleur en fonction de la machine, il vous est possible d'écrire en Assembleur DEX, conformément au langage de la machine (virtuelle) Dalvik. La JVM Dalvik tire son nom d'un village de pêche Islandais. Par respect, les termes d'assembleurs et de désassembleurs islandais ont été retenus pour l'opération de passage vers et depuis le format **DEX**, ce qui donne « naturellement » les termes de *smali* et *baksmali*.

Pour les plus curieux d'entre vous, voici un exemple de *Hello World* directement écrit en Smali :

```
.class public LHelloWorld;

.super Ljava/lang/Object;

.method public static main([Ljava/lang/String;)V
    .registers 2
    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
    const-string        v1, "Hello World!"
    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/
lang/String;)V
    return-void
.end method
```

Typiquement, le code ci-dessus déclare une classe **HelloWorld**, disposant d'une fonction **main()**, qui appelle la commande **System.out.println()** depuis le paquetage **java.io.PrintStream**, avec comme unique paramètre la chaîne de caractère « Hello World! ».

Si vous cherchez à compiler cet exemple, à le convertir en **DEX**, puis à l'exécuter sur votre téléphone, rien de plus simple. Le résultat est conforme à nos attentes :

```
$ wget https://bitbucket.org/JesusFreke/smali/downloads/
smali-2.0.6.jar
$ java -Xmx512m -jar smali-2.0.6.jar -o classes.dex HelloWorld.
smali
$ zip HelloWorld.zip classes.dex
$ adb push HelloWorld.zip /data/local
$ adb shell dalvikvm -cp /data/local/HelloWorld.zip HelloWorld
"Hello World!"
```

Notez enfin que le *bytecode* généré est bien conforme au standard **[6]**, de même que le format **DEX** de l'exécutable Dalvik résultant **[7]**.

Mais revenons à nos moutons et procédons à l'installation d'ApkTool au moyen de la procédure suivante :

```
$ wget https://bitbucket.org/iBotPeaches/apktool/downloads/
apktool_2.0.0.jar
$ wget https://raw.githubusercontent.com/iBotPeaches/Apktool/
master/scripts/linux/apktool
$ mkdir ~/bin
$ mv apktool apktool_2.0.0.jar ~/bin
$ chmod a+x ~/bin/apktool
$ export PATH=~/bin:$PATH
```

Ceci étant fait, extrayons (à nouveau) notre APK :

```
$ mkdir fb
$ cd fb
$ apktool d ../FrozenBubble-3.4.apk
I: Using Apktool 2.0.0 on FrozenBubble-3.4.apk
I: Loading resource table...
```

```
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file: /home/ben/apktool/framework/
1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Comme vous pouvez le voir, les ressources ont bien été décodées et le fichier **classes.dex** a bien été converti en Smali par l'opération de *baksmaling*. Vous pouvez désormais lister le contenu du dossier, ce qui devrait ressembler aux lignes ci-dessous :

```
$ ls -l FrozenBubble-3.4
-rw-rw-r--  1 ben ben 1975 mai    6 12:58 AndroidManifest.xml
-rw-rw-r--  1 ben ben  291 mai    6 12:58 apktool.yml
drwxrwxr-x  2 ben ben 4096 mai    6 12:58 assets/
drwxrwxr-x  3 ben ben 4096 mai    6 12:58 lib/
drwxrwxr-x  3 ben ben 4096 mai    6 12:58 original/
drwxrwxr-x 16 ben ben 4096 mai    6 12:58 res/
drwxrwxr-x  5 ben ben 4096 mai    6 12:58 smali/
```

Bonne nouvelle, tous les fichiers sont désormais lisibles, et l'on retrouve le code Smali au sein du dossier du même nom. Reste donc maintenant à modifier le fichier **AndroidManifest.xml**, ce qui devrait résulter dans le contenu suivant :

```
<application android:allowBackup="true" android:icon="@
drawable/app_frozen_bubble" android:label="@string/app_name"
android:debuggable="true">
[...]
</application>
```

Et voilà, il ne reste plus qu'à « recompiler » notre application au moyen d'**ApkTool**. On dispose en effet des sources assembleur Smali et de toutes les ressources nécessaires.

```
$ apktool b FrozenBubble-3.4
I: Using Apktool 2.0.0
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Copying libs...
I: Building apk file...
```

Une nouvelle archive APK a désormais été produite, dans le fichier **FrozenBubble-3.4/dist/FrozenBubble-3.4.apk**.

Vous êtes prêts à installer l'application avec vos symboles de débogage ? Hélas non, pas tout à fait. N'oubliez pas que seules les applications signées peuvent être installées.

5 SignAPK

Notre APK n'étant pas signé, il est nécessaire de le faire avant installation (sous peine de rejet par **adb**). Android utilise un certificat pour authentifier l'auteur de l'application et établir une relation de confiance avec les utilisateurs. Les applications sont théoriquement signées par le programme **jarsigner** que nous avons découvert plus tôt. Il existe cependant différentes alternatives ne nécessitant pas tout le SDK et **SignAPK** en est une. Récupérez donc le programme depuis la source (hautement fiable) suivante :

```
$ wget http://www.learn2crack.com/download/SignApk.zip
$ unzip SignApk.zip && cd SignApk
$ ls -l
-rw-r--r-- 1 ben ben  325 nov.   6  2008 README
-rw-r--r-- 1 ben ben 7369 nov.   5  2008 signapk.jar
-rw-r--r-- 1 ben ben 1217 nov.   5  2008 testkey.pk8
-rw-r--r-- 1 ben ben 1675 nov.   5  2008 testkey.x509.pem
```

Comme vous pouvez le voir il s'agit juste d'un **.jar** et d'un ensemble de clés. SignAPK nécessite une clé privée et une clé publique pour fonctionner. Vous pouvez générer les vôtres ou utiliser celles fournies, qui correspondent aux clés officielles de Google utilisées pour l'image de *recovery* (mais pas pour le vrai système).

Dans le cas où vous voudriez (légitimement) utiliser vos propres clés, il vous suffit de les générer (et surtout de les conserver à l'abri), au moyen des commandes suivantes :

```
$ openssl genrsa -out key.pem 1024
$ openssl req -new -key key.pem -out request.pem
$ openssl x509 -req -days 9999 -in request.pem -signkey key.pem
-out certificate.pem
$ openssl pkcs8 -topk8 -outform DER -in key.pem -inform PEM -out
key.pk8 -nocrypt
```

Reste ensuite à signer votre APK pour (enfin) générer notre APK « installable » :

```
$ java -jar signapk.jar key.x509.pem key.pk8 FrozenBubble-3.4.apk
FrozenBubble-3.4-signed.apk
```

6 Installation de notre APK

L'application signée, il n'y a plus qu'à l'installer, de manière très simple au travers d'**adb** :

```
$ adb install FrozenBubble-3.4-signed.apk
3471 KB/s (7819342 bytes in 2.199s)
        pkg: /data/local/tmp/FrozenBubble-3.4-signed.apk
Success
```

En parallèle de cette étape, lancez un deuxième shell pour analyser les logs d'installation :

```
D/AndroidRuntime( 4969): >>>>>> START com.android.internal.
os.RuntimeInit uid 2000 <<<<<<
D/AndroidRuntime( 4969): Calling main entry com.android.commands.
pm.Pm
D/Finsky  ( 1727): [1] PackageVerificationReceiver.onReceive:
Verification requested, id = 73
D/Finsky  ( 1727): [1] WorkerTask.onPreExecute: Verification
Requested for id = 73, data=file:///data/local/tmp/FrozenBubble-
3.4-signed.apk flags=112 fromVerificationActivity=false
D/DefContainer( 4555): Copying /data/local/tmp/FrozenBubble-3.4-
signed.apk to base.apk
D/PackageManager( 1909): Renaming /data/app/vmdl1441765231.tmp to
/data/app/org.jfedor.frozenbubble-1
I/PackageManager( 1909): Running dexopt on: /data/app/org.jfedor.
frozenbubble-1/base.apk pkg=org.jfedor.frozenbubble isa=arm
vmSafeMode=false
I/dex2oat ( 4998): /system/bin/dex2oat --zip-fd=6 --zip-location=/
data/app/org.jfedor.frozenbubble-1/base.apk --oat-fd=7 --oat-
location=/data/dalvik-cache/arm/data@app@org.jfedor.frozenbubble-1@
base.apk@classes.dex --instruction-set=arm --instruction-set-
features=div --runtime-arg -Xms64m --runtime-arg -Xmx512m --swap-
fd=8
D/PackageBroadcastService(32425): Received broadcast
action=android.intent.action.PACKAGE_ADDED and uri=org.jfedor.
frozenbubble
```

Vous pouvez le voir, le **PackageManager** a bien reçu une nouvelle application qu'il a vérifié. Il a ensuite copié l'APK pour optimiser le code **DEX** via **DexOpt** et ainsi générer le code **ODEX** correspondant au sein du répertoire **/data/dalvik-cache**.

7 Débogage Java

Voyons maintenant ce que fait notre application, grâce au protocole JDWP (pour *Java Debug Wire Protocol*). C'est l'utilisation de ce dernier qui nécessitait le *flag* de debug au sein de notre **Manifest** (tout cela pour ça ...). Dans un shell, lancez donc **logcat** pour analyser les logs et en parallèle, démarrez le jeu puis cliquez sur le bouton ***Puzzle*** du jeu, qui permet de lancer la partie.

```
$ adb logcat
V/ActivityManager( 1909): Display changed displayId=0
I/ActivityManager( 1909): START u0 {act=android.intent.action.
MAIN cat=[android.intent.category.LAUNCHER] flg=0x10200000 cmp=org.
jfedor.frozenbubble/com.efortin.frozenbubble.HomeScreen (has
extras)} from uid 10020 on display 0
I/ActivityManager( 1909): Start proc 5734:org.jfedor.frozenbubble/
u0a154 for activity org.jfedor.frozenbubble/com.efortin.
frozenbubble.HomeScreen
I/ActivityManager( 1909): Displayed org.jfedor.frozenbubble/com.
efortin.frozenbubble.HomeScreen: +341ms

I/ActivityManager( 1909): START u0 {cmp=org.jfedor.frozenbubble/.
FrozenBubble (has extras)} from uid 10154 on display 0
I/ActivityManager( 1909): Displayed org.jfedor.frozenbubble/.
FrozenBubble: +888ms
```

Comme vous pouvez le voir, le démarrage du jeu depuis l'application **LAUNCHER** a démarré l'activité **org.jfedor.frozenbubble/com.efortin.frozenbubble.HomeScreen**, à savoir le menu de démarrage. En cliquant sur le bouton ***Puzzle***, c'est l'activité **org.jfedor.frozenbubble/.FrozenBubble** qui a été démarrée. C'est cette dernière qui nous intéresse. Vous pouvez maintenant quitter le jeu, nous le relancerons manuellement.

Voici donc comment lancer exactement cette activité manuellement, en mode debug activé, comme si la demande provenait du LAUNCHER.

```
$ adb wait-for-device
$ adb shell am start -e debug true -a android.intent.action.MAIN
-c android.intent.category.LAUNCHER -n org.jfedor.frozenbubble/.
FrozenBubble
Starting: Intent { act=android.intent.action.MAIN cat=[android.
intent.category.LAUNCHER] cmp=org.jfedor.frozenbubble/.FrozenBubble
(has extras) }
```

La commande **jdwp** d'**adb** devrait maintenant nous indiquer sur quel port il est possible d'écouter l'application :

```
$ adb jdwp (Java Debug Wire Protocol)
9083
```

Reste alors à transférer ce port JDWP vers une socket TCP :

```
$ adb forward tcp:29882 jdwp:9083
```

Puis de s'y connecter via **jdb**, un équivalent de **gdb** pour Java :

```
$ jdb -J-Duser.home=. -connect com.sun.jdi.SocketAttach:hostname=
localhost,port=29882
Initializing jdb ...
```

Au sein de **jdb**, vous pouvez lister les *threads* existants (de nombreuses autres applications tournent), pour se restreindre à celui qui nous intéresse plus particulièrement :

```
> threads
 [...]
Group main:
 [...]
  (org.jfedor.frozenbubble.GameView$GameThread)0xe47 Thread-46698
running
```

Il s'agit ici du thread **46698**, à l'adresse mémoire **0xe47**. Mettons ce dernier en suspens et affichons la pile au sein de la directive **where** :

```
> suspend 0xe47
> thread 0xe47
Thread-46698[1] where
  [1] android.graphics.Canvas.native_drawBitmap (native method)
  [2] android.graphics.Canvas.drawBitmap (Canvas.java:1,338)
  [3] org.jfedor.frozenbubble.Sprite.drawImage (Sprite.java:129)
  [4] org.jfedor.frozenbubble.GameView$GameThread.drawBackground
(GameView.java:1,354)
  [5] org.jfedor.frozenbubble.GameView$GameThread.doDraw
(GameView.java:1,149)
  [6] org.jfedor.frozenbubble.GameView$GameThread.run
(GameView.java:1,900)
```

Dans l'activité en question (qui correspond globalement au lancement du jeu), on découvre donc que le jeu se lance via la méthode **run()** de la classe **GameThread**, elle-même extraite du fichier **GameView.java**, en provenance du paquetage **org.jfedor.frozenbubble**. Cela représente déjà beaucoup d'informations et maintenant, on sait à partir d'où chercher pour modifier notre niveau de démarrage.

8 Dex2Jar

Revenons donc maintenant aux « sources » de notre application. Nous disposons du fichier **classes.dex**, reste à en extraire les sources Java, ou au moins une archive JAR. Ceci peut se faire très facilement au moyen du programme **dex2jar [8]**. Ce dernier permet de reproduire les différentes classes Java au format **.class** depuis une archive DEX.

Téléchargez et installez donc le programme :

```
$ wget https://bitbucket.org/pxb1988/dex2jar/downloads/
dex2jar-2.0.zip
$ unzip dex2jar-2.0.zip
$ cd dex2jar-2.0
$ rm *.bat
$ chmod a+x *.sh
$ mv * ~/bin
```

Puis exécutez-le sur le fichier **classes.dex** de notre APK :

```
$ cd fb/FrozenBubble-3.4/build/apk
$ d2j-dex2jar.sh classes.dex
dex2jar classes.dex -> ./classes-dex2jar.jar
```

Nous disposons désormais d'une archive JAR tout ce qu'il y a de plus « lisible ».

9 Java Decompiler

Nous allons maintenant utiliser **JD-GUI** (pour *Java Decompiler* **[9]**), qui est une interface graphique permettant de décompiler, lire et analyser une archive JAR et toutes les classes Java qui la composent. Le programme principal s'appelle **JD-GUI**, une IHM indépendante. Notez que le projet dispose également de **JD-Eclipse** et **JD-IntelliJ**, des *plugins* pour les IDE du même nom.

Vous pouvez donc maintenant télécharger l'application et l'exécuter sur notre archive JAR générée au moyen de **Dex2Jar** :

```
$ wget https://github.com/java-decompiler/jd-gui/releases/download/
v1.0.0/jd-gui-1.0.0.jar
$ java -jar jd-gui-1.0.0.jar classes-dex2jar.jar
```

Vous pouvez désormais parcourir l'ensemble des classes Java de l'application, ce qui devrait vous amener à quelque chose de sensiblement semblable à la figure 2, page suivante.

Si vous avez bien suivi le résultat précédent avec **jdb**, vous devriez vous être rués sur la méthode **run()** de la classe **GameThread** du fichier **GameView.java**, au sein du paquetage **org.jfedor.frozenbubble**. Mais quitte à avoir les sources, plutôt que la méthode **run()**, jetons un œil du côté de l'initialisation de la classe **GameThread** en elle-même.

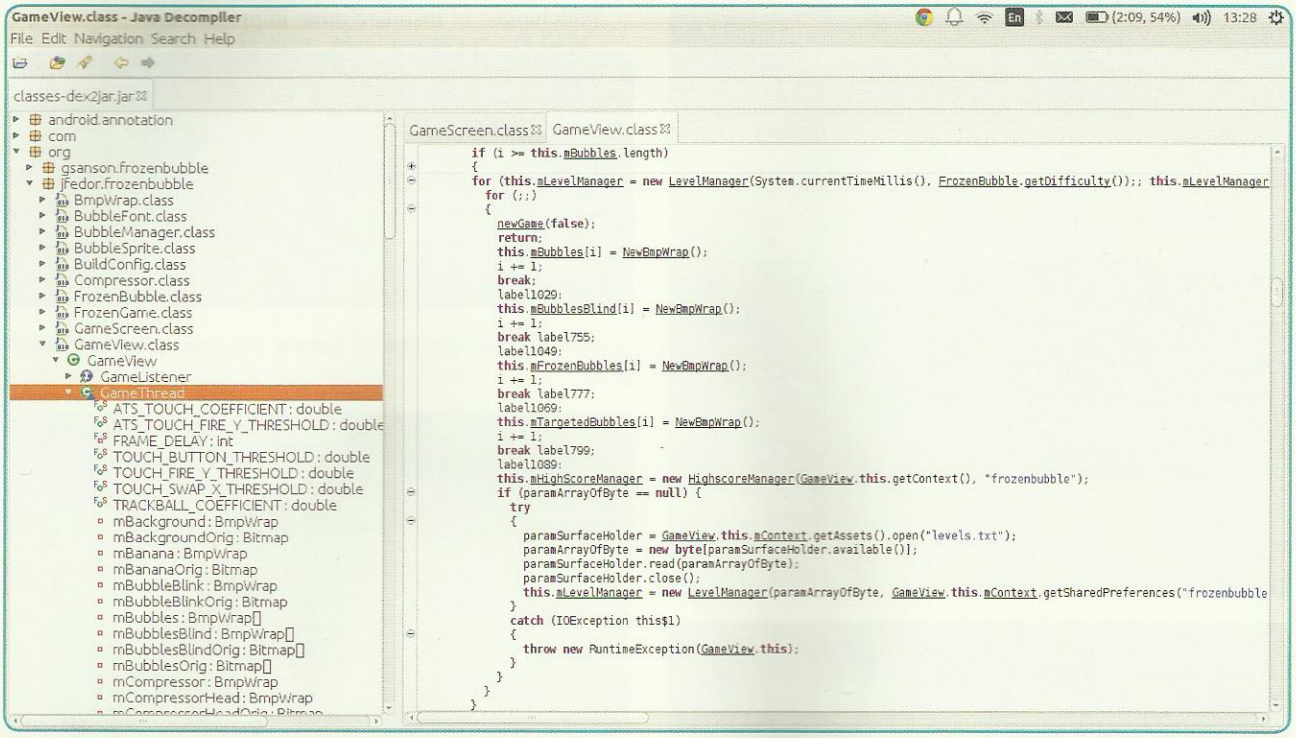

Figure 2 : Interface de Java Decompiler GUI (JD-GUI).

On retrouve donc la méthode Java suivante :

```
public GameThread(SurfaceHolder paramSurfaceHolder, byte[]
paramArrayOfByte, int paramInt, boolean paramBoolean)
   { [...] }
```

Plus spécifiquement, en regardant les sources, le code ci-dessous semble parfaitement correspondre à l'initialisation du jeu dans son mode classique (i.e. ***Puzzle***) :

```
if (paramArrayOfByte == null) {
  paramSurfaceHolder = GameView.this.mContext.getAssets().
open("levels.txt");
  paramArrayOfByte = new byte[paramSurfaceHolder.available()];
  paramSurfaceHolder.read(paramArrayOfByte);
  paramSurfaceHolder.close();
  this.mLevelManager = new LevelManager(paramArrayOfByte,
GameView.this.mContext.getSharedPreferences("frozenbubble",
0).getInt("level", 0));
}
```

La dernière ligne du code semble instancier une classe de type **LevelManager**, dont vous pourrez retrouver l'implémentation via JD-GUI. Le nom de la classe me plaît assez, cela semble converger vers la bonne direction. En regardant le code de cette classe, le dernier paramètre d'initialisation permet en effet de positionner le niveau (ou *level*). Pour ce faire, la valeur correspond à la clé **level** qui sera lue dans les préférences de l'application, sa base de données interne. Par défaut, la valeur **0** sera utilisée, permettant donc, en toute logique, d'initialiser le jeu au premier niveau. On en déduit donc (ou suppose en tout cas pour le moment) que pour démarrer le jeu au niveau **5** par exemple, il suffirait de remplacer cette ligne de code par la suivante :

```
this.mLevelManager = new LevelManager(paramArrayOfByte,
GameView.this.mContext.getSharedPreferences("frozenbubble",
0).getInt("level", 4));
```

10 Smali

Reste donc à faire cette modification. Pour boucler la boucle, nous allons le faire directement un assembleur Smali et ce, pour deux raisons. La première c'est que ça fait plus « hardcore », la deuxième, c'est que l'utilitaire ApkTool va utiliser directement le code Smali pour régénérer le *bytecode* DEX.

Nous savons désormais où se trouve la ligne à modifier dans le code Java. Reste à trouver son équivalent dans le code Smali. En toute logique, au vu du nom de fichier,

cela devrait se trouver dans le fichier **FrozenBubble-3.4/smali/org/jfedor/frozenbubble/GameView$GameThread.smali**. Éditez donc ce dernier et recherchez le constructeur de classe ci-dessous :

```
.method public constructor <init>(Lorg/jfedor/frozenbubble/GameView;Landroid/view/
SurfaceHolder;[BIZ)V
```

Un peu plus bas dans le code, vous trouverez le code Smali suivant :

```
    .line 967
    # getter for: Lorg/jfedor/frozenbubble/GameView;->mContext:Landroid/content/Context;
    invoke-static/range {p1 .. p1}, Lorg/jfedor/frozenbubble/GameView;->access$1(Lorg/
jfedor/frozenbubble/GameView;)Landroid/content/Context;

    move-result-object v11

    .line 968
    const-string v12, "frozenbubble"

    const/4 v13, 0x0

    .line 967
    invoke-virtual {v11, v12, v13}, Landroid/content/Context;-
>getSharedPreferences(Ljava/lang/String;I)Landroid/content/SharedPreferences;

    move-result-object v10

    .line 969
    .local v10, "sp":Landroid/content/SharedPreferences;
    const-string v11, "level"

    const/4 v12, 0x0

    invoke-interface {v10, v11, v12}, Landroid/content/SharedPreferences;->getInt(Ljava/
lang/String;I)I

    move-result p4
```

qui correspond au code Java précédent, à savoir :

```
mContext.getSharedPreferences("frozenbubble", 0).getInt("level", 4)
```

Si vous avez tout saisi jusqu'ici, il vous reste donc à modifier la ligne suivante :

```
const/4 v12, 0x0
```

par cette dernière :

```
const/4 v12, 0x4
```

Ca y est, le tour est joué. Il ne vous reste plus qu'à recompiler et ré-empaqueter le tout via ApkTool (comme vu précédemment) et à re-signer votre APK via SignAPK.

Procédez ensuite à la désinstallation du jeu et à la réinstallation de notre nouvel APK via les commandes suivantes :

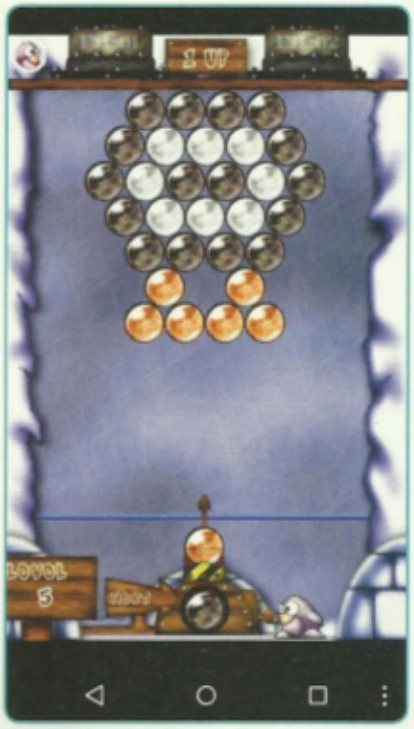

Figure 3 : Démarrage de Frozen Bubble directement au niveau 5.

```
$ adb uninstall org.jfedor.frozenbubble
$ adb install FrozenBubble-3.4-signed.apk
```

Si tout s'est bien passé, vous pouvez relancer le jeu, démarrer une partie et vous devriez voir la capture d'écran correspondant à la figure 3, preuve s'il en fallait qu'en une heure d'exercice de rétro-ingénierie, nous avons réussi à passer directement au niveau 5, ce qui aurait nécessité deux minutes de jeu, mais l'idée est là ;-)

11 Prévention et Obfuscation

Finalement, c'était plutôt simple de retrouver les sources de cette application. Tellement simple que cela semble à la portée de tous, ce qui peut poser des problèmes. Pour une application libre et/ou OpenSource, qu'importe

EN COUVERTURE

après tout. Pour une application propriétaire, tant pis pour eux je dirais, mais malgré tout il doit y avoir un moyen de compliquer un petit peu la tâche. Différents programmes d'obfuscation existent pour répondre à ce besoin. Le plus connu des développeurs Android est très certainement **ProGuard** (libre **[10]**) et son équivalent propriétaire **DexGuard [11]**, qui y ajoute des capacités de chiffrement des chaînes de caractères, de chiffrement des classes Java etc. ...

L'obfuscation de code est un procédé qui rend sa lecture plus difficile. Dans les faits, des programmes tels que ProGuard permettent de réduire la taille du code, de minimiser l'empreinte mémoire, d'obfusquer bien sûr, mais également d'optimiser le code généré. Ils agissent donc sous la forme de post-processeurs. ProGuard s'exécute à la compilation des sources Java en fichier `.class`, au travers de Ant, et via une myriade d'options. Notez cependant que, en fonction des options utilisées, il peut en résulter des optimisations telles qu'elles rendent l'application non-fonctionnelle avec ART, la nouvelle machine virtuelle d'Android, remplaçante de Dalvik. À utiliser avec précaution donc.

Le *bytecode* Java est beaucoup plus compact que les sources. C'est d'autant plus vrai avec le *bytecode* Dalvik. Ceci étant, aucune analyse n'est faite sur le code « mort », c'est à dire le code non-atteignable. ProGuard permet ainsi d'analyser le *bytecode* pour en supprimer les classes, champs et autres méthodes non utilisées (dans un contexte donné). L'exemple suivant est probant :

```
if (Config.LOGGING){
  TestClass test = new TestClass();
  Log.d(TAG, "[onCreate] testClass=" + test);
}
```

Lorsque compilé en mode *release*, il y a fort à parier que le booléen `Config.LOGGING` sera toujours faux. Le code n'a donc aucune chance d'être exécuté et peut donc être supprimé.

Le principe d'obfuscation en lui-même, est vicieux. Le *bytecode* compilé contient de nombreuses informations de débogage (que nous avons utilisé précédemment), comme le nom de fichiers sources, les numéros de lignes, le nom des champs, le nom des méthodes, le nom des arguments, des variables et j'en passe. Grâce à ces informations, nous avons pu lire le code source complet via JD-GUI. Mais il est possible de supprimer ces informations ou au moins de remplacer tous les noms par des séquences de caractères aléatoires, rendant le *reverse-engineering* beaucoup plus compliqué. Ce n'est pas impossible pour autant à qui voudra y consacrer le temps nécessaire, mais cela découragera 99% des amateurs. En effet, une fonction telle que `put(ma_clé, ma_valeur)` est assez intuitive à la lecture. La version obfusquée qui serait, par exemple, `rzurhfjhf(hhefh, ma_valeur)` l'est beaucoup moins.

Conclusion

Sans s'étendre davantage sur le sujet, vous en savez désormais un peu plus sur le fonctionnement des applications Android et la relative facilité de décompilation et de rétro-ingénierie de ces dernières. Comme vous avez pu le voir, il est relativement « facile » de désassembler une application pour voir ce que fait cette dernière. Sans parler de « hacking », il est toujours intéressant de voir que vos applications de la vie de tous les jours disposent en leurs sources d'un grand nombre de dépendances « nuisibles » dont vous vous passeriez très probablement. De même, cet article aura peut être aussi réussi à vous sensibiliser sur la sécurité de vos propres applications. S'il y a des choses qui doivent rester cachées, faites en sorte qu'elles le soient vraiment et n'hésitez pas à décompiler vos propres applications pour voir de quoi il en retourne ;-) ■

Références

[1] Frozen Bubble : http://www.frozen-bubble.org/

[2] SDK Android : https://developer.android.com/sdk/index.html

[3] APK Downloader : http://apps.evozi.com/apk-downloader/

[4] APKTool : https://ibotpeaches.github.io/Apktool/

[5] Smali : https://code.google.com/p/smali/

[6] ByteCode Dalvik : https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html

[7] Format DEX : https://source.android.com/devices/tech/dalvik/dex-format.html

[8] Dex2Jar : http://tools.kali.org/reverse-engineering/dex2jar

[9] Java Decompiler : http://jd.benow.ca/

[10] ProGuard : http://proguard.sourceforge.net/

[11] DexGuard : http://www.saikoa.com/dexguard