Reversing DexGuard 5.x

Note: this tutorial covers a basic understanding of DexGuard protection applied to a \*.dex file. A basic knowledge of smali syntax and Android OS are required.

Version: 0.1a

Autor: Nihilus

*Disclaimer*

This document is not a "How to crack applications protected with DexGuard", it's just a general understanding of the operations carried on by this protection. It may helps programmers who wants to improve the protections on their commercial application or may helps someone who wants to know better how a specific app works. DexGuard is often used by Malware writers to obfuscate their evil apps in conjunction with their custom protection. I do not take any responsibility on how this guide is used.

This is the first version of the paper, it offers just a basic overview of DexGuard protection mechanism.

This paper needs an adequate soundtrack to be read: Death - Symbolic.

*DexGuard*

You can find all the information on the official site: <http://www.saikoa.com/dexguard>. DexGuard is the big brother of ProGuard, take a look at the comparison.

*What is needed*

* Smali/baksmali: [h HYPERLINK "https://code.google.com/p/smali/downloads/list"ttps://code.google.com/p/smali/downloads/list](https://code.google.com/p/smali/downloads/list)
* ApkTool: <https://code.google.com/p/android-apktool/downloads/list>
* Hex editor: <http://www.handshake.de/user/chmaas/delphi/download/xvi32.zip>
* A text editor such as SublimeText or Notepad++.
* Smali Syntax Highlighter: [https://github.com/ShaneWilton/s HYPERLINK "https://github.com/ShaneWilton/sublime-smali"ublime-smali](https://github.com/ShaneWilton/sublime-smali)
* Android SDK with functioning debugger.
* A rooted Android device.
* A program that uses DexGuard as protection, I recommend TitaniumBackup 6.1 or later. This program is essential, buy the PRO version and support the programmer!
* WinZip, WinRAR or any other program that can extract \*.apk files.
* In this package you could find also some files that I'll mention in the paper.

*Warnings*

* DexGuard is the commercial counterpart of ProGuard, it really helps developers to protect their apps (or at least slow down the reversing process).
* DexGuard protection vary in every app, and it can be configured for more or less protection; it offers a variety of types of protection systems such as String/Class/Assets/etc encryption and obfuscation.
* DexGuard also offer anti-tampering checks, I'll show you an easy way to bypass this protection.
* TitaniumBackup also has a Native Protection inside libTitanium.so that helps verifying the app integrity.
* TitaniumBackup 6.1 uses DexGuard 5.3 (probably) + Native Code Checks + Custom License.
* I do not show you how to crack Titanium license protection, I've taken the app because it's one of the most interesting to reverse.

*Setup*

* Install TitaniumBackup on your device
* Go to /data/app/ on your device with a file manager or using ADB shell and copy to your computer the file com.keramidas.TitaniumBackup-\*.apk. (You could use ADB: adb pull /data/app/com.keramidas.TitaniumBackup-\*.apk)
* Extract classes.dex from the apk file and disassemble it with baksmali.jar (java –Xmx512M –jar baksmali.jar –o classes/)
* Now you can open all the \*.smali files inside the classes folder in your favourite text editor.
* Now let’s get a look to the app!

*Sublime Text 3 Smali Syntax Highlighter*

*Let’s start our adventure in the Land of Smali*

The first things to do when we start to analyze an Android app is to look what are the main classes, in this case especially the security related classes; look for interesting strings that may help us understanding the code; look at strange code construct or strange files inside the apk package such as native libs or crypted assets.

In this tutorial I don’t talk about reversing ARM code (because we don't need ARM reversing here), if you are interested there are some really well-written tutorials out in the wild such as MessyBinary’s turorial (Faceniff 2.4 reversing is a must read). If you would like to analyze native libs you need an ARM decompiler, for example IDA Pro or Disarm.

Analyzing TitaniumBackup package we find out that there is a lib called libTitanium, this lib checks the integrity of the TB package. From now until the end, I’ll call TitaniumBackup as TB; in this version there aren’t encrypted assets.

Move on the smali code. I think that executing the app and looking for interesting Toasts or Debug messages is a good way to find out where to start in the sea of smali files. Usually programmers put the strings used in the program in \*.xml files. Let’s decode the rest of the apk package. I usually use ApkTools for extracting assets, libs and resource files: java -jar apktool.jar d -f --keep-broken-res TB.apk pkg/. Let’s take a look into res folder; there are many folders (language folders/configuration folders) but we have to find two specific files: public.xml and strings.xml. They contain strings and resources IDs.



*A view of public.xml: it lists the names and the alphanumerical ID that we can find in the smali code.*



*A view of strings.xml: it lists the names with the associated string.*

Obviusly while executing and exploring the program we could see some interesting strings such as: *Free version*, *you are using the free version of TB*, or *Available only on Donate version.*

Looking first at the strings.xml file and after in the public.xml file we could find what ID is associated to a specific string: string -> name -> alphanumerical ID. Searching the IDs in the smali files we are able to locate where the strings are loaded and, looking at the code, understanding what is executed before displaying that string.

It becomes immediately evidently, looking a bit (forgive me the pun) at the smali code, that DexGuard protected apps are heavily obfuscated, in fact in the code we could notice some interesting things:

* Strange fields/methods/classes names (Unicode - Chinese characters)
* Not all the displayed strings are in the strings.xml file or incorporated in the smali files, so that means that probably exists a decryption method for displaying other strings (in this case it exists, DexGuard strings encryption)
* Really big arrays (> 1900 bytes)…what do they contains?
* Heavy use of reflection
* Dynamic Code Loading and Executing
* Heavy use of Java’s encryption classes

*Debugger: the evil friend of the programmer*

Debugging is really good while developing because it helps findings and fixings errors, but sometimes it could be disastrous to leave debugging/logging code inside a commercial application, especially if the log process involve the protection mechanism. For example is not so good to leave, in the final app, portion of code that log error messages/information about the protection, simply because it helps reverser (the evil ones) to get easily to the crucial code (license verification mechanism). TB includes (in its distributed to public version) some interesting logs. Let's see them! Fire up your Android debugger and connect the phone to the PC. The Android debugger is included in the Android SDK, you can find it in the "tools" folder, under the name "monitor.bat". Set the "info" level for debugging and start up TB on the device, this is the result:



*Some useful log info*

Don't you think these info are really useful for an evil reverser? The "Application" tab indicates the package name; the "Tags" indicates the title of the logged message, in this case it indicates the class where the message is logged; the "Text" indicates the logged message, in this case some information regarding the license verification. The program checks the presence of a valid license cache (the file named licensing-cache.xml contains the activation info).

*Dig deeper, yes, like a miner*

Now it's our interest to find those log messages, so open the file named "둆" (note the strange name, say thanks to DexGuard) inside the "o" folder (you can find it in the smali folder). But wait, I looked into strings.xml and 둆$CON.smali files (also in all the other files) and there is not a single variable that contains the above information strings. So what do we do now? Oh, we could look at the function/class used to log these information: the class is Log <http://developer.android.com/reference/android/util/Log.html> and the used methods are "i()", "e()" and so on (look the link). The "i()" method takes two strings as arguments. Let's look for the "Log;->i()" method inside the "둆" class and...we found nothing. What's happening here?! I let you guess what the problem here is! Oh, sure, DexGuard string encryption, reflection and dynamic class loading! If we take a look into the class we can notice some really big arrays and others really little; remember when I told you about these arrays at the start of the paper? Ok, these arrays contain:

* the little ones: decryption code;
* the big ones: encrypted-compressed class code.

We can also notice the heavy use of reflection (in all the classes), it's easy to spot out, take a look at this picture:



*Reflection in action*

It's easy to see that reflection is used, look at line 139, here is used the method "invoke(Object receiver, Object... args)" the method description report:

* Information: returns the result of dynamically invoking this method. Equivalent to receiver.methodName(arg1, arg2, ... , argN).
* Parameters: receiver the object on which to call this method (or null for static methods); args the arguments to the method

Take a look at this: <http://developer.android.com/reference/java/lang/reflect/Method.html>

So the "invoke" function just calls a method (contained in a class) with its parameters; we just have to look above to discover the name of the class, the name of the invoked method and it's parameters. We can identify at line 117 a method, "forName(String name)" and at line 133 a method "getMethod(String name, Class className)"; these two methods get each a String that represents the name of the Class and the name of the Method to invoke, and these strings are generated by another method named "ą(III)Ljava/lang/String" that takes 3 integers and returns a String...so strange. Let's take a look at that one!

*DexGuard: string decryption*

The "ą(III)Ljava/lang/String" method is generated by DexGuard to decrypt the encrypted strings inside the program. It receives 3 integers and returns a String. Here’s the code:



*The method (generated by DexGuard) used to decrypt the strings*

At line 14031 the method gets an array of Shorts, that array is filled with data during the initialization of the class, take a look:



*The array is filled*

You just need to know that it loads one of the "little" (not so little 649 Shorts) array that I mention before and puts it in a class variable that is loaded in the decryption method. Because this is a little paper about DexGuard I won't dig deeper into this method, but I show you how to easily get the final result (the decrypted String). The best way is to use the angelic (eheh for us) logging methods of the Log class. This is an example:



*A way to see the decrypted string*

This is an easy and powerful way to see the decrypted string; you just need to insert logging call where you need to see the output of a function, then recompile, execute and debug the new (modified) application. I’ve used the Tag "REVERSING" in the call because it is easy to find in the debugger; try to set the debugger to show only the messagges with Tag equals to "REVERSING". This method could be used with every Java type and Class (in combination with toString), so you can create your personal Log class. At this point you must take a look at "The way of the Android cracker" tutorials by Lohan+, just search it on Google. You should also take a look at his website: <http://androidcracking.blogspot.it/> it is a gold mine.

*DexGuard: encrypted assets*

In TB I haven't found traces of encrypted assets, but there are some applications that use this protection to hide sensible data. Probably the decryption method is similar to the decrypting method used with the strings. The better way to extract encrypted assets is to find where they are used in the code and try to locate the decryption method, than simply try to Log or Save on disk the decrypted data.

*DexGuard: reflection*

Let's try to understand what reflection is and how it is used as anti-reversing strategy. Start with a Wiki page to understand better what we are facing: <http://en.wikipedia.org/wiki/Reflection_(computer_programming)>. The important part is this: reflection can (and here it is) be used for observing and/or modifying program execution at runtime. DexGuard uses reflection mostly as an obfuscation method, combined with string encryption and dynamic class loading. All the important call to security / licensing method are carried on with reflection because it is difficult to undestand for a casual evil reverser. In fact DexGuard is a slightly good protection because it is able to slow down the reversing process. We have seen that it's easy to identify in the smali code the amount of calls to reflection method that are used: we can find calls to "invoke()", "getMethod()" and so on. Another important thing about reflection is that with it the programmer is able to access to private field and method in a class: reflection is powerful when it serves protection purpose. For a basic start with reflection you have to read this article: [http://mobile.tutsplus.com/tutorials/android/java-reflectio HYPERLINK "http://mobile.tutsplus.com/tutorials/android/java-reflection/"n/](http://mobile.tutsplus.com/tutorials/android/java-reflection/). But remember that reflection alone is not that scary, the scariest thing in DexGuard is probably the Dynamic Class Load process.

*DexGuard: dynamic class loading*

Printing out all the decrypted strings (using the logging method mentioned in the "string decryption" chapter) I noticed that DexGuard uses reflection on classes that apparently don't exist. In particular in TB are called some methods from a class named "둆", remember that class? It is the class that we have analyzed before, the class that contains the string decryption method and the long (> 1900 bytes) arrays. The problem is that the called methods do not exist in the previous analyzed class. So what's going on? Simple! There is another class, with the same name (it is not a coincidence), hidden and dynamically loaded at runtime, can you guess where is it hidden? In the really big arrays! The class that we are analyzing and the hidden class have the same name because they are the licensing classes.

DexGuard compress and store some important classes into arrays of bytes and at runtime it loads and execute in memory these classes.

DexGuard dynamic class store and load steps:

* DexGuard take the class that has to be hidden and executed at runtime;
* Compress it using deflate algorithm (GZIP) using java.util.zip package and store the byte representation in an array of bytes;
* At runtimes the array of bytes is read and decompressed in memory;
* Then using the methods "getClassLoader()", "getDeclaredConstructor()" and "newInstance()" create a new instance of the extracted class;
* During the runtime it could access (using reflection) at all the fields and methods of the loaded class;
* Obviously inside the loaded class there are important methods used to check license or to protect the application.

To modify how the program works you have to extract the dynamic loaded class and modify it, then re-compress (deflate) it and store the new sequence of bytes in the array; if the modified class changed size you also have to adjust the decompression routine.

TB uses two classes in the protection system, compressed and loaded with DexGuard at runtime. Analyzing the "둆" file you could find where the bytes array are loaded, here is an example:



*The array is loaded to be decompressed*

Some lines after, the array of bytes is decompressed and loaded in memory; in the archive you can find the two classes that are stored in the licensing Class in three form: compressed, decompressed and decompiled.

File list:

* classes1 (compressed), classes1.dex (decompressed), 둆$㥳.smali (decompiled)
* classes2 (compressed), classes2.dex (decompressed), 둆.smali (decompiled)

They are not explained because it isn't the goal of this paper, remember that this is not a "How to crack" tutorial.

This dynamic loading method is used often in Malware because it helps to hide payloads.

*DexGuard: anti-tampering*

Nowadays every commercial apps has an anti-tampering protection to verify if the program has been altered. The app try to detect if it has been tampered looking at some information of the classes.dex file inside the apk package. When a program has been modified something change in the sign of this file. Anti-tampering protection detects and responds to any alteration of the code in order to prevent malicious modifications from breaking license checks. Anti-tamper protection can be internal or external, for example TB uses libtitanium.so and DexGuard to verify the integrity of the app. If you try to modify TB without deactivate the anti-tampering checks the app will respond in a strange way. Here is an example, I've modified a string (in a casual smali files) and this is the result:



*Anti-tampering in action*

The protection blocked the execution of a function and displayed a nag screen to inform us that we have failed in modifying the app.

Anti-tampering checks are usually made using classes.dex infos like size, checksum or generating MD5 / SHA1 / ecc. of the file that are stored and compared during the execution to see if the application has been modified. To defeat this protection you can find where the check is made and invalidate it, or you could use another way that bypass completely the problem.

The trick is to ODEX the modified app using dexopt-wrapper generating the odex of the classes.dex, then reinstall the original application, give the execution permission to the odex file and execute the program...well done, you have defeated the anti-tampering check! The modified odex file get executed instead of the original classes.dex and if the anti-tampering protection checks the classes.dex file, it finds it not modified.

*The end*

In the future I'll try to release other versions of this paper, possibly that offers a more in-depth view of this commercial protection.

*Nihilus*