

Information Security Basics

Self-defense capabilities for Android applications against repackaging

Mentor

Aleksandr PILGUN

Student

Davit POGOSIAN

Student id

017104145E

1. Introduction

Android is the most popular mobile operating system. In 2017, 66,74% of all smartphones were using Android as an operating system and over 3 million applications are available in Google Play for this platform. One of the top priorities of Android is openness, as open that the users can install applications from third-party markets. That is why the number of Android applications is growing rapidly. As an effect of openness, Android apps can be easily decompiled due to their structural characteristics, which mean that apps modified in this way can be repackaged.

Hackers can add or modify logics of the original apps by using some well-known tools and then release the modified app to a third-party market as new version. Researchers found out that 86% of malwares are repackaged malware, which demonstrates the popularity and severity of repackaged malware¹. Malwares are not the only problem based on repackaging. Hackers can modify original applications and stole information such as personal data, bank accounts and so on. That is why repackaging is a big problem in the Android ecosystem.

This paper is organized as follows:

- Section 1 contains the introduction,
- Section 2 takes a brief look at related work,
- Section 3 describes different ways of repackaging,
- Section 4 describes ways of protecting Android applications,
- Section 5 presents the conclusions.

2. Related work

2.1 Android

Android is primarily a mobile operating system, being designed mainly to work on mobile phone and other mobile devices.

Android has been designed for touch experience – which differentiate it from other operating systems which are not touch-centric, being created to be used with mouse and keyboard.

¹Source : <http://www.zdnet.com/article/android-malwares-dirty-secret-repackaging-of-legit-apps/>

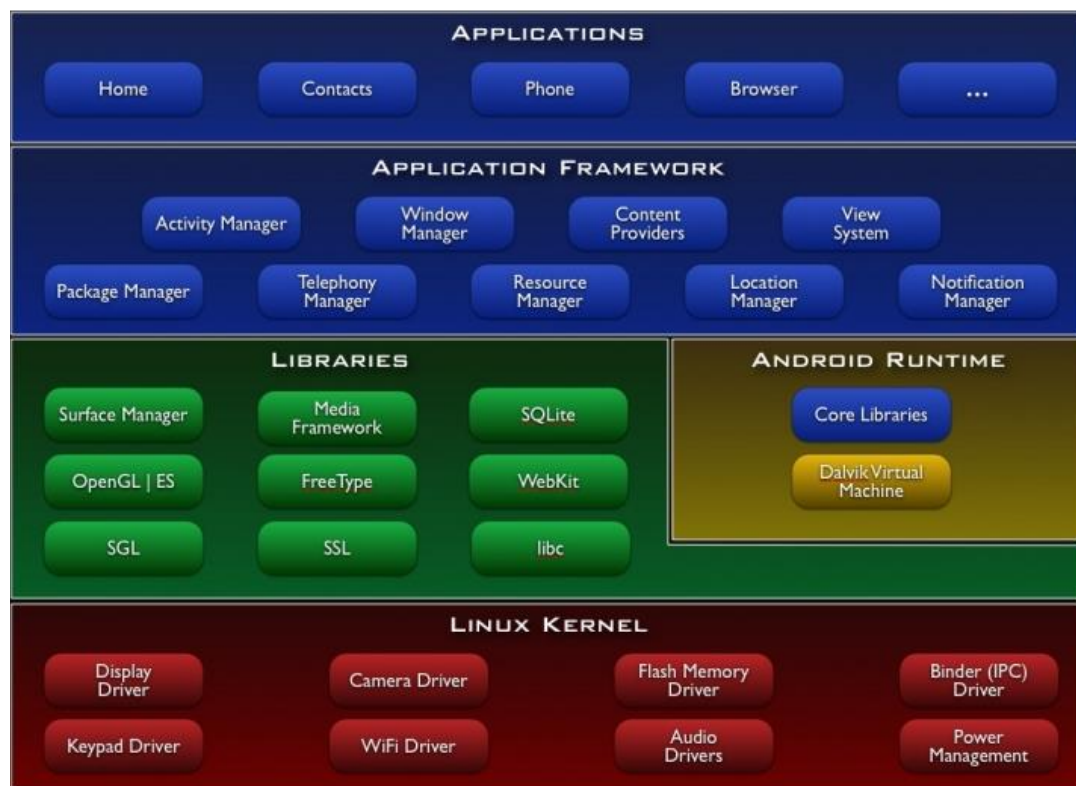
Android is also open source – one of the main reasons for its spread adoption. Open source basically means that everyone can see the code that makes up the operating system, not only seeing it, but you can also contribute to the operation system.

Android runs on all kind of hardware: phones, tablets, eBook Readers, Google TV, netbooks, smartwatches, car computers and other devices, which also makes it different from other operating systems.

Android has a pretty interesting history – it was first created by a company named Android Inc., which was funded by Google. Android Inc. was created by Andy Rubin, Rich Miner, Nick Sears, and Chris White. Google eventually ended up buying Android Inc. in 2005.

Android Architecture

The below diagram presents the architecture of the Android operating system and can be read from bottom to top:



At the base of the diagram, we can see that at its core, Android is built off Linux and the Linux kernel is the one responsible for engaging with the device hardware.

Above the core layer come low level libraries that bring some functionality which is not part of the Linux kernel, but also the Android Runtime – where the applications run.

On top of the Android Runtime and the libraries – it uses both, we have the application framework, which contains all the libraries Android developers use to write the applications.

At the top, we have applications, both the ones included in the OS and the ones written separately.

Linux kernel

Android is not very similar to the Linux operating system, but at its core, Android is based on Linux. The Linux kernel mainly handles the hardware communication, as it contains the drivers for most of the hardware for an Android device. It also controls access to that hardware and sharing it amongst the application.

A *kernel* refers to the lowest level core of an operating system – the part of the operating system that handles the very basic tasks like communicating with the resources of the device and determining what code can run on the CPU or CPUs of the device.

Low level libraries

Low level libraries are not part of the Linux kernel, but are used to provide all the services that the Android operating system provides.

The majority of the libraries are written in C or C++, as it directly communicates with the Linux kernel. Some of the libraries are hardware specific, but some of them are common libraries that are part of the core of Android:

- SQLite, which is a light database engine for storing data like contacts,
- LibWebCore, which is a web browser engine,
- SGL (Skia Graphics Engine), which is a graphics library that is able to take advantage of 2D processing capabilities of hardware that supports it and allows for graphic-intensive applications.

Android Runtime

Android Runtime contains the Dalvik virtual machine and the core Java libraries that Android supports. Android Runtime is the space where applications run and get converted from a special format (DEX files) to actual instructions that can be executed by the Android device.

The Dalvik virtual machine is optimized for mobile devices; it takes up a smaller amount of memory and it is more performant on less performant devices.

The Dalvik virtual machine runs one virtual machine per application in Android; so, if multiple applications are launched on an Android device, each one will get its own machine to run it.

Application Framework

Application developers are allowed to use the services contained in the application framework for their applications.

The way Android apps should work or should provide all the tools and components needed are contained in the set of Java libraries in the application framework. When writing an app, the code should use the application framework to display the user interface using controls that are provided inside the framework.

Applications

The top layer of the Android architecture is the apps layer, with both built-in applications and 3rd party applications.

Android contains a set of built-in applications, as the homescreen (the first application that runs when you start up an Android device), the phone, the browser, the contacts list.

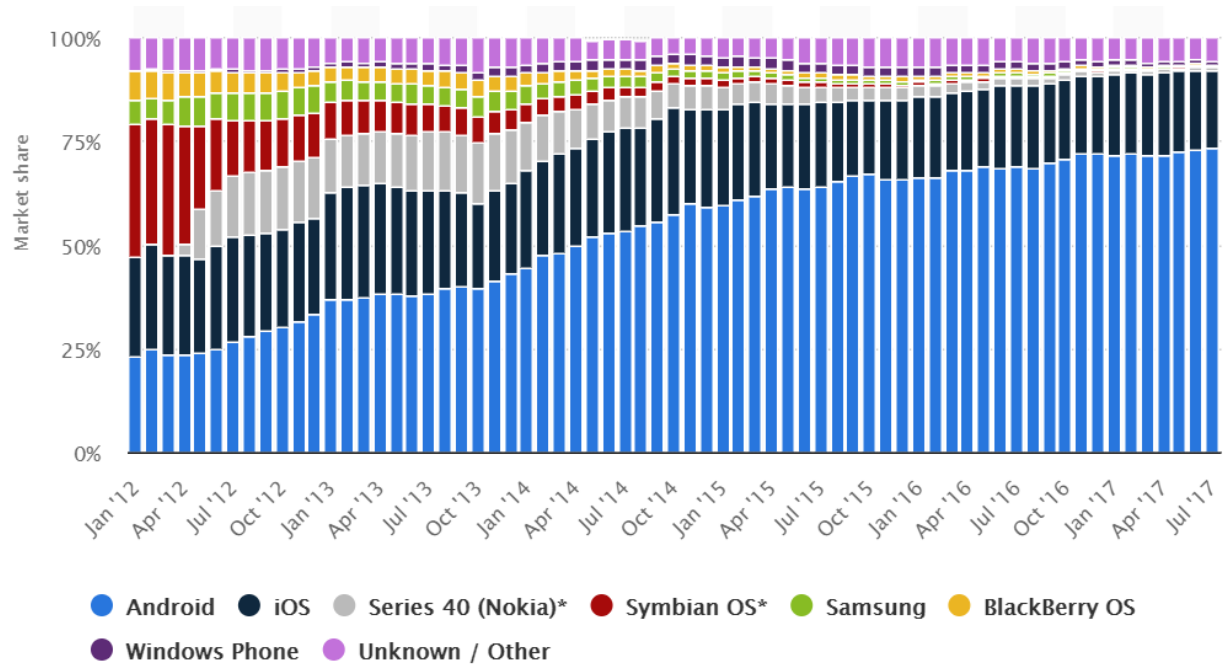
These built-in applications can be replaced by applications you developed, as anybody has access to exactly the same set of libraries and resources all the built-in apps do.

In the Google Play store, there are a lot of applications that replace built-in apps like the homescreen.

2.2 Statistics

Below there are some Android statistics taken from www.statista.com

Mobile operating systems' market share worldwide from January 2012 to July 2017²

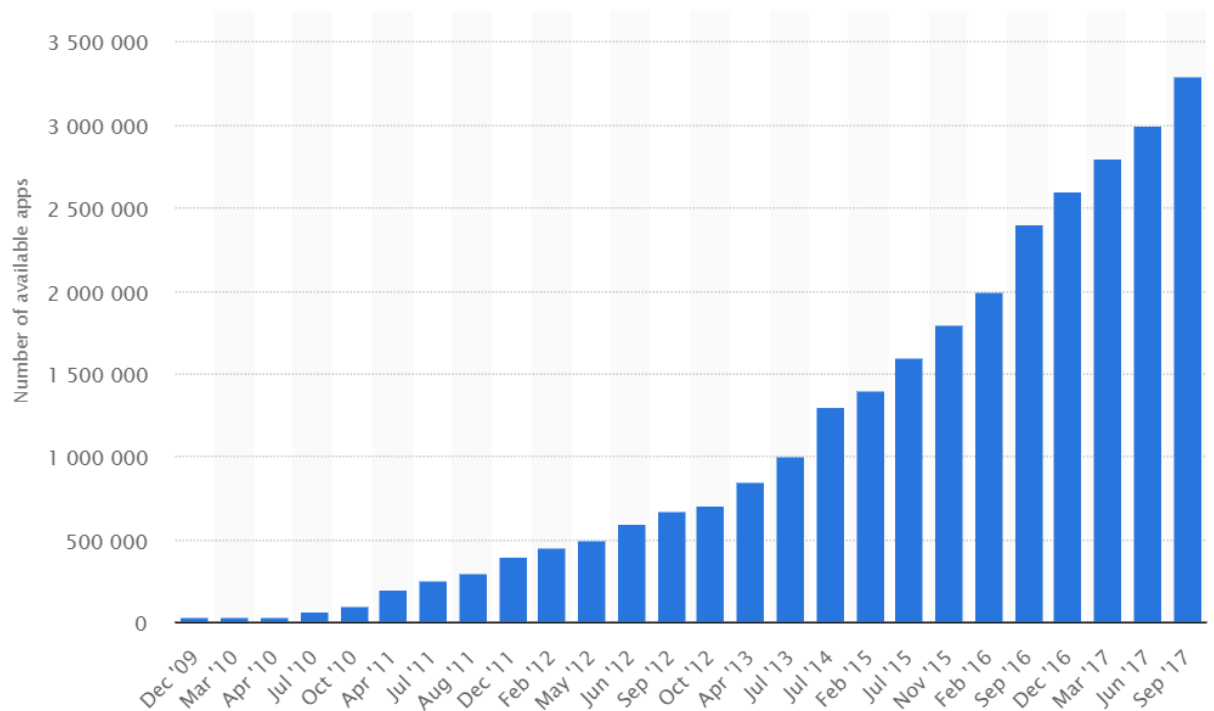


We can see that Android's market share increased quite constantly since January 2012 (23.21%), becoming the major player in the last years (73.39% in July 2017).

The evolution of Android's market share also engaged an increase in the number of applications available for download in the Google Play store (formerly known as Android Market), as we can see in the below graphic:

²Source : <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>

Number of available applications in the Google Play Store from December 2009 to September 2017³



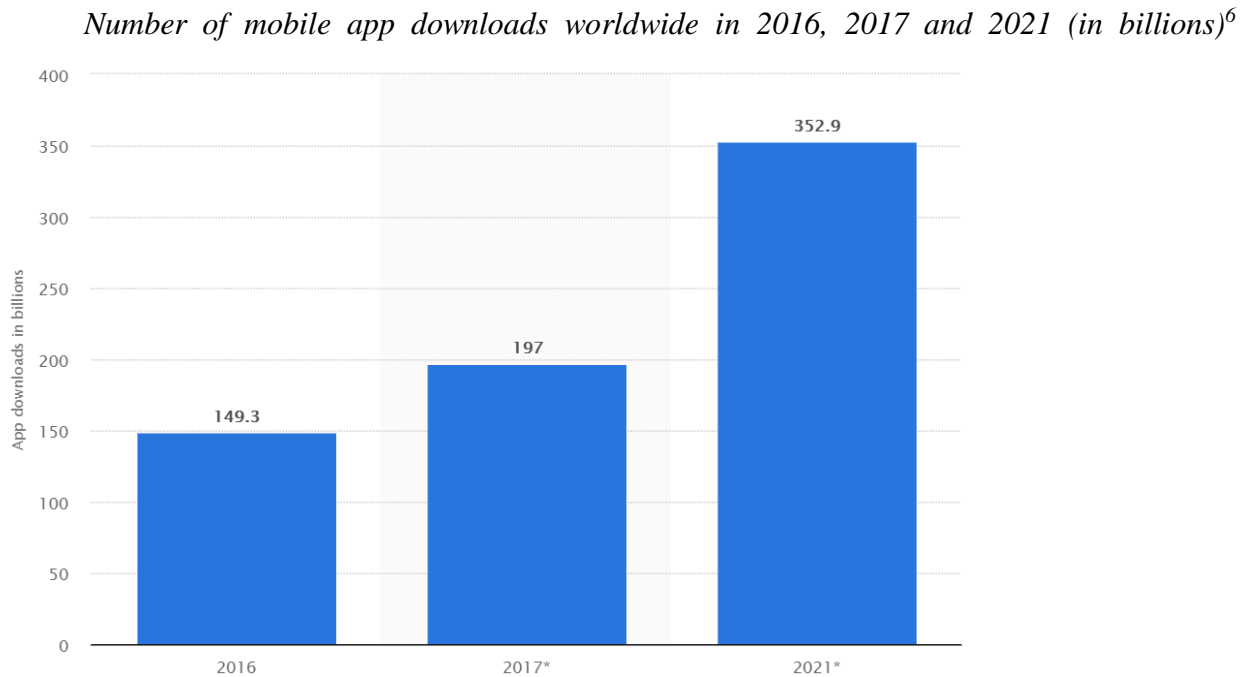
The evolution is impressive, as the number of available apps tripled since July 2013, when it reached 1 million apps; the most popular applications being the gaming apps like Candy Crush Saga, Clash of Clans, Clash Royale, Pokemon Go, the social media apps like Whatsapp, Facebook, Messenger, Instagram and others⁴⁵.

³ Source : <https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/>

⁴ Source : <https://www.statista.com/statistics/271674/top-apps-in-google-play-by-revenue/>

⁵ Source : <https://www.statista.com/chart/9834/the-top-10-android-apps-in-the-uk/>

The below statistic presents a forecast for the number of mobile app downloads worldwide in 2016, 2017 and 2021. We can see that the projection for 2021 is almost double compared with 2017.



Android Security Model

The Android application security can be described as 2 layers: the underlying Android operating system security and the security protection built into the app itself.

The Android Security Model is different from the one used in Linux, the trust boundary being a single application and not a single user.

This security model provides a sandboxed environment for applications, and explicit means of allowing interactions between the mobile hardware, the user, and the application.

The Android Security Model also introduces the concept of signing the applications to ensure trust. Starting with Android 4.4, at startup, the boot process is verified by the function `device_mapper-verify` to ensure that, based on the hash, the operating system hasn't been tampered with and it is trusted for its security capabilities.

⁶ Source : <https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/>

Android operating system is stored as Read-Only files, which provides an initial layer of protection against run-time interferences, but also, the kernel is protected by process isolation. That means that applications must use system calls to interact with kernel functions.

Android applications are primarily available from the Google Play store, a trusted source with more than 3 million applications available.

There are other sources of Android applications which can be used, including vendor-specific application repositories or email, file-sharing systems transfers or other approaches.

Developers must sign Android applications and each Android application must have a valid certificate, which can be self-signed by the developer, or signed by a certification authority, such as Verisign, Comodo, GoDaddy or another registrar service.

Android doesn't do certificate chain validation by default, it only identifies it. An application receives at installation a unique identifier (UID) and runs within its own application sandbox, to ensure that applications cannot interfere with each other.

Files created by the application are stored in the sandbox and are by default only accessible to that application.

Content providers give the ability to define read-write permissions for other applications and to share data, but also to mark the data as not exportable and be used as application-only data.

To harden the boundaries of the Android application sandbox, starting with Android 4.3, security enhanced Linux or SE Linux, has been adopted, which requires the security action to be explicitly approved. This enhancement brings mandatory access control to ensure that a program cannot arbitrarily assign permission for another app to access its data.

Developers must use policies to set up access rules, which are a key feature to protect against privileged escalation attacks.

Permissions are a key building block for security in Android. The Android system and applications own resources and data, and to access that, another application must make an explicit request by declaring the permissions they need. These include permissions like access to the camera, access to the microphone, access to the contact list, writing to external storage and so on. In general, applications should minimize the permissions they need. The set of system defined permissions cover most requirements.

Applications can, however, define their own permissions. Permissions are grouped in protection levels, with normal protection level permissions offering little risk, but dangerous protection level requiring explicit user agreement. An application declares the permissions it needs in its Android manifest file. Android uses a technique called Intents to achieve inter-process communication. An intent is a message requesting an action, and optionally providing data. An intent can require recipients to have a specific permission in order to process the intent.

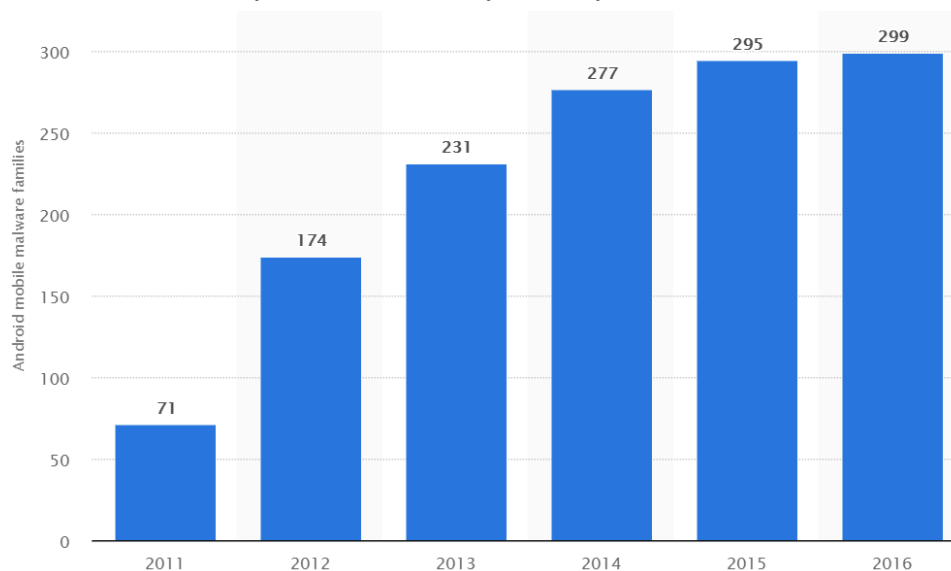
An application component can implement intent filters to specify the type of intent that an activity service or broadcast receiver can respond to.

There are three ways an intent can be used. An activity in Android represents a single screen in an application. An activity is started by sending a startActivity intent. The intent may include data, and the activity may return data. A service in Android is a component that performs actions in the background without a user interface.

The service is started by sending a startService intent. A broadcast is a message that any application can receive and is sent using one of three sendBroadcast intents. An intent can be sent explicitly to an application component, or can be implicit, in which case the operating system will find the appropriate recipient component.

Mobile devices are not used only for gaming and social media, but also for sensitive transactions like banking and email and with this evolution both on the number of the applications but also in the market share, Android operating system continues to be a target for malware and attacks:

Cumulative number of mobile malware families from 2011 to 2016⁷



⁷ Source : <https://www.statista.com/statistics/494982/android-cumulative-malware-families/>

2.3 APK

Once developing and testing an Android app are done, the final step is packaging.

Packaging an Android app creates an APK file. The APK is a ZIP archive that has compressed files and folders from which the app is made up.

The APK is then copied to Android devices – directly or through an app store.

Android apps that only use the Android SDK and don't have any native code can be installed on nearly all Android devices, including cell phones, tablets, and hybrid devices.

An Android app contains different files and folders, like:

- the **App manifest** – an XML file that describes the app's version, capabilities and permissions,
- a **signed certificate** – verifies the developer's identity,
- the Platform **code** – proper code for the version of the Android SDK targeted when compiling the app,
- the **custom Compiled classes** – the logic of the application in a special format,
- the **app's resources, graphics and other files** which can either be compiled or not compiled,
- **uncompiled assets**.

All these files and folders are compressed during the packaging process into a single APK file.

There are a few different ways to get the APK file to the users, so they can run the app on their devices:

- one approach is to use the process called Side-Loading. Google Play store and any of the other common app distribution channels are bypassed and the APK file is simply sent to the users through conventional means, using email, file-sharing systems, or letting them download it from a website or other approaches.

Contemporary Android devices block this option through a security setting which by default doesn't let the user do this, but they can turn that feature off and allow installation of apps from an unknown source. Once the setting is changed, they can simply open the APK file, and confirm that they want to use the app, and the app will be installed.

- for commercial distribution, the app stores are used: Google Play, Amazon and others.

You can package your app either from the command line, or for simplicity, through an integrated development environment like Android Studio.

2.4 Repackaging

Repackaging an application means to change its normal behavior/content and to redistribute the app, usually without the permission of original app developers. During the repackaging process, malicious payloads could be injected into the repackaged apps.

Repackaged apps, even if they are based on legitimate ones, include some changed functionalities, which can lead to many problems, like phones compromised, phone bills increased or sensitive information (stored on the phones) stolen.

Repackaging doesn't only impact end users, but also developers as they find their intellectual properties violated, the in-app revenues stolen, and their reputation impaired.

Repackaging apps has also an impact on the marketplace because mobile users and developers, as its customers, can use other marketplaces for extra security.

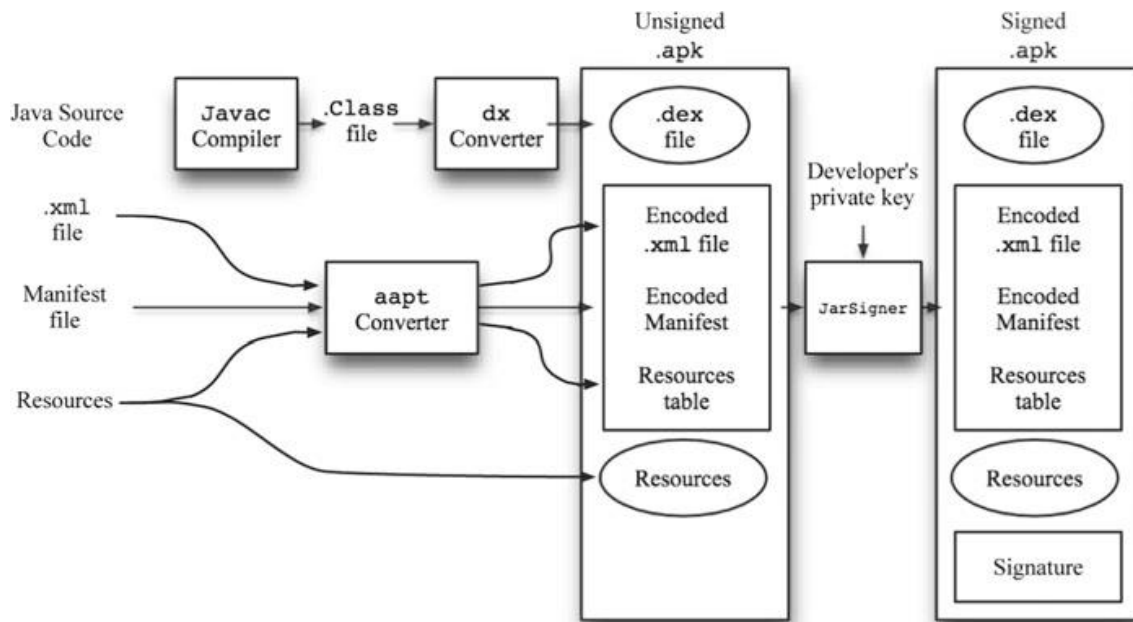
Seeing the overall impact that repackaging has, the entire smartphone ecosystem could be damaged by this practice.

Original and repackage apps have almost the same code, but, as the developers' signing keys are not known, each app is signed with a different key.

The main reasons for repackaging, as mentioned in the study "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces"⁸, are :

- injecting new in-app advertisements - new ad SDKs are added into the original app,
- usurping existing in-app advertisements - existing ad SDKs still remain, but the corresponding publisher identifiers have been replaced likely with the repackagers' identifiers
- trojanizing legitimate apps with malicious payloads - the added payloads can be used to conduct a variety of malicious activities, such as sending text messages to premium-rated numbers, downloading additional apps from the Internet, rooting the phone, and even registering the compromised phones as bots.

⁸ Source : *Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces*, Wu Zhou, Yajin Zhou, Xuxian Jiang, Peng Ning,



3. Repackaging tools and procedures

For this project I made different tests to repack my own application (Hello Romania) APK and Mobiliteit.lu APK.

Mobiliteit.lu app is a very popular application being used for information regarding bus/train schedules in the Grand Duchy of Luxembourg.

To repack Mobiliteit.lu application I use APKTool and for my own application I used dex2Jar and JD-GUI, tools which I will present below.

3.1 APKTool

APKTool is used for reverse engineering 3rd party, closed, binary Android apps, as it can decode an app almost to its original form and rebuild it after modification have been made.

APKTool features include:

- Disassembling resources to nearly original form (including resources.arsc, classes.dex, 9.png and XMLs),
- Rebuilding decoded resources back to binary APK/JAR,
- Organizing and handling APKs that depend on framework resources,

- Smali Debugging (Removed in 2.1.0 in favor of IdeaSmali),
- Helping with repetitive tasks.

To repackage an application using APKTool you only need the APKTool itself, which is free and can be downloaded without any restrictions and the target application APK.

The Mobiliteit.lu application has been downloaded from the Google Play Store and using ES File Explorer I created a backup of the application which created an APK file on the SD card.

Using the command **java -jar apktool_2.3.0.jar d -f -o *path_for_decoded_files* *path_for_APK*** the APK was decoded and in the folder containing the decoded files, among other files, we have:

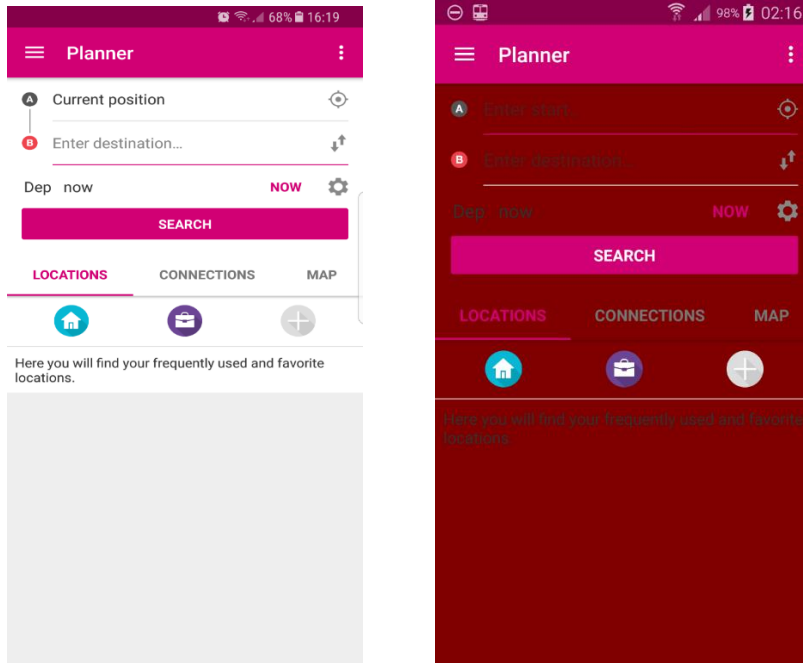
- the *AndroidManifest.xml* which is unencrypted and readable using a notepad similar application,
- in the smali folder, we can find the application's code in smali language,
- in the res folder, we can find all resources, which are decoded.

To test repacking on this application, I decided to change its colors, which was made by modifying the *colors.xml* file contained under **res\values** folder, using the value **#800000** for most of the colors.

The next step was to use the command **java -jar apktool_2.3.0.jar b *path_for_decoded_files*** for repackaging the application, which created the modified APK file named *Mobiliteit.apk*.

For signing the new APK file, I used ZipSigner application.

To test the success of the repackaging, I installed the modified APK on my device and below you can see the results: the first capture is of the original application and the second on if from the modified and repackaged application:



3.2 dex2jar Package Description

Dex2jar is a tool used for converting .dex format used in Android into .class format used in Java – from one binary format to another binary format, not Java source. If viewing the source code is needed, Java decompiler is needed.

Dex2jar contains following components:

- dex-reader – for reading the Dalvik Executable (.dex/.odex) format,
- dex-translator – to convert the file into ASM format,
- dex-ir used by dex-translator – designed to represent the dex instruction,
- dex-tools - which work with .class files.
- dex-writer [To be published] write dex same way as dex-reader.

To repack an application using Dex2jar you need to download Dex2jar and JD-GUI apps which are free and easy to use.

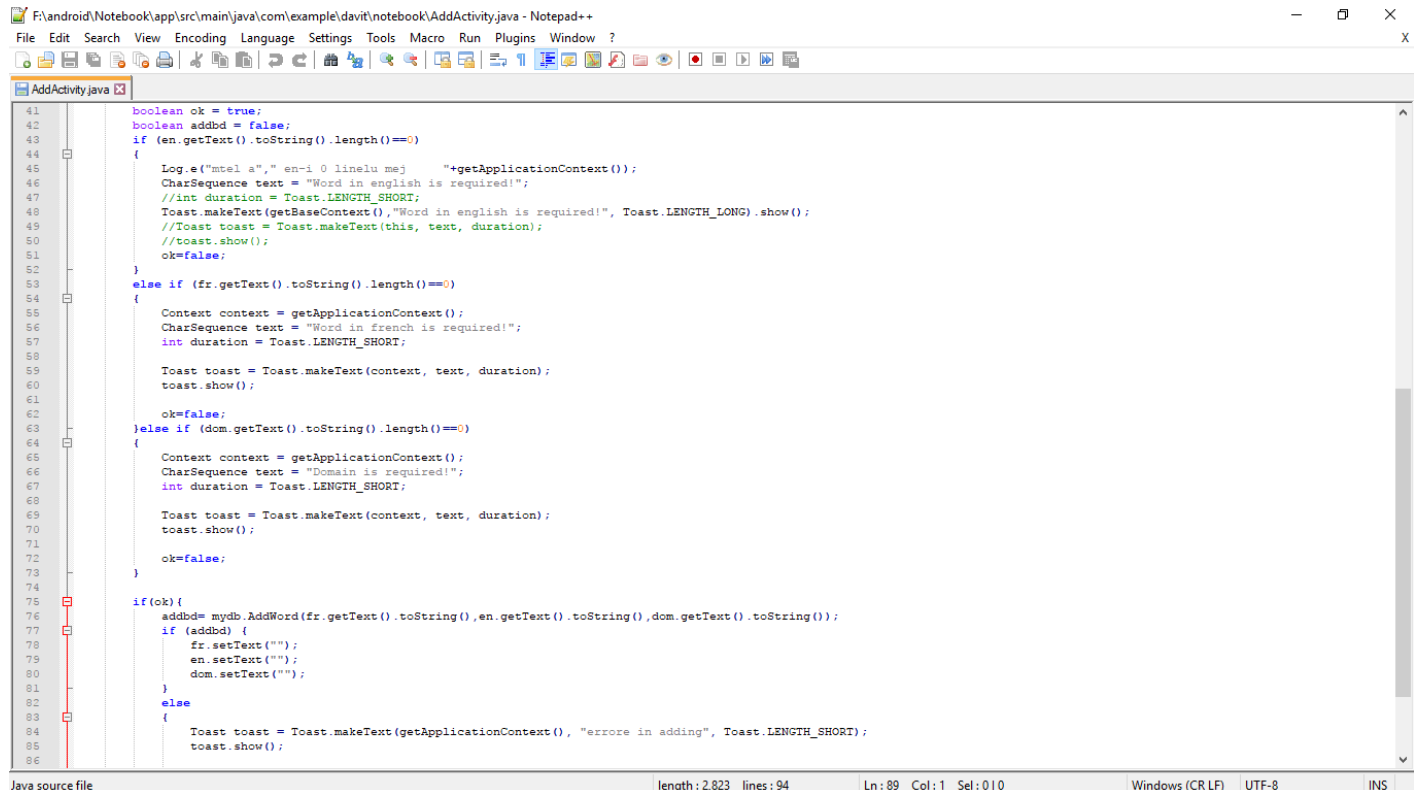
In order to repack an app, you need the original APK, which will be renamed and converted into a .zip. Once this activity is done, you can unzip the APK and find the AndroidManifest.xml, resources.arsc and classes.dex files. The file needed is classes.dex which must be copied to the Dex2jar folder .

Using the command `dex2jar classes.dex` the `classes.dex` file is decoded – the result can be seen by opening the decoded file using JD-GUI.

In regards to the code, original code and decoded code aren't the same – the order of variables, methods and functions are changed, the parameters are renamed with `param[parameter type][number]` (if there are more than one parameter with the same type).

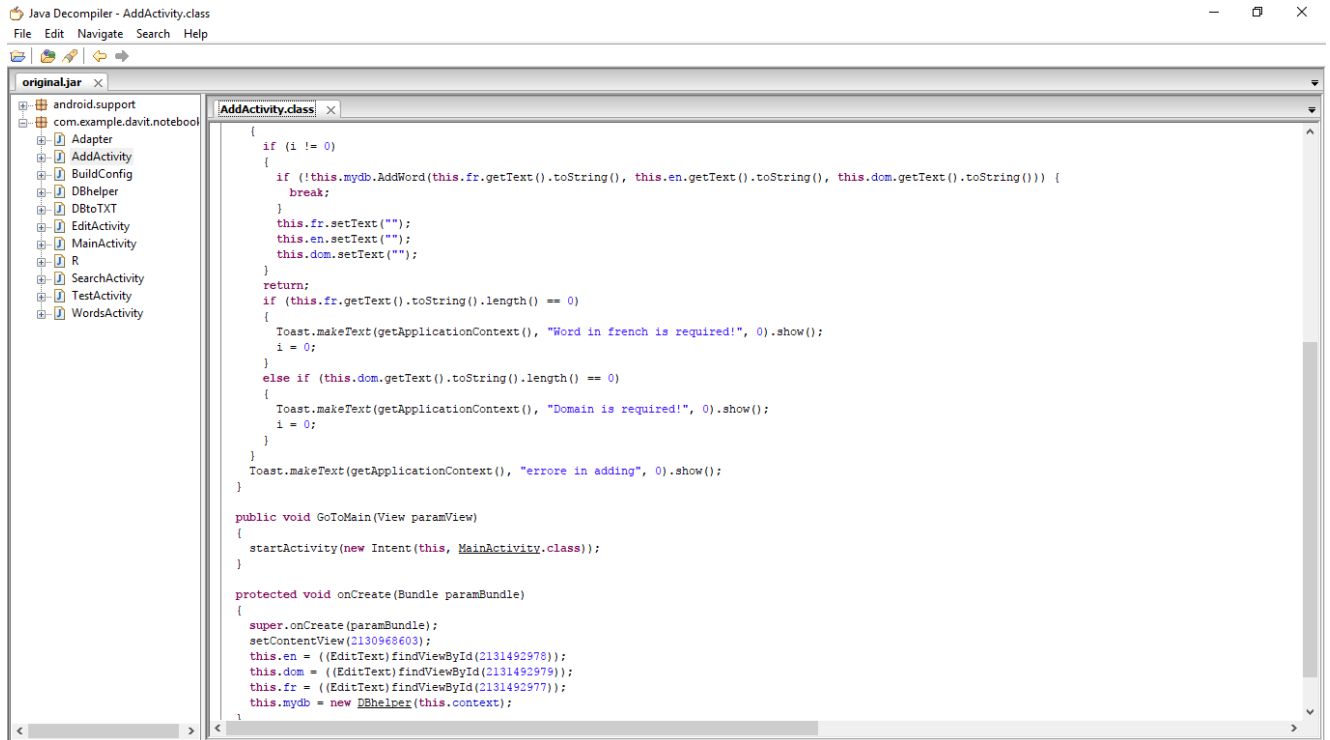
Below we can see the difference:

Original code



```
41 boolean ok = true;
42 boolean addbd = false;
43 if (en.getText().toString().length() == 0)
44 {
45     Log.e("mte1 a", "en-i 0 line1u mej" + getApplicationContext());
46     CharSequence text = "Word in english is required!";
47     //int duration = Toast.LENGTH_SHORT;
48     Toast.makeText(getApplicationContext(), "Word in english is required!", Toast.LENGTH_LONG).show();
49     //Toast toast = Toast.makeText(this, text, duration);
50     //toast.show();
51     ok=false;
52 }
53 else if (fr.getText().toString().length() == 0)
54 {
55     Context context = getApplicationContext();
56     CharSequence text = "Word in french is required!";
57     int duration = Toast.LENGTH_SHORT;
58
59     Toast toast = Toast.makeText(context, text, duration);
60     toast.show();
61
62     ok=false;
63 }
64 else if (dom.getText().toString().length() == 0)
65 {
66     Context context = getApplicationContext();
67     CharSequence text = "Domain is required!";
68     int duration = Toast.LENGTH_SHORT;
69
70     Toast toast = Toast.makeText(context, text, duration);
71     toast.show();
72
73     ok=false;
74 }
75 if(ok) {
76     addbd= mydb.AddWord(fr.getText().toString(), en.getText().toString(), dom.getText().toString());
77     if (addbd) {
78         fx.setText("");
79         en.setText("");
80         dom.setText("");
81     }
82     else
83     {
84         Toast toast = Toast.makeText(getApplicationContext(), "errore in adding", Toast.LENGTH_SHORT);
85         toast.show();
86     }
```

Modified code after decoding



3.3 Proguard

The Android SDK includes a tool called ProGuard, a command-line tool with an optional graphical user interface, which is a code obfuscation tool.

ProGuard is a default tool in development environments like Oracle's Wireless Toolkit, NetBeans, EclipseME, Intel's TXE SDK and Google's Android SDK.

ProGuard is a free tool used to shrink, optimize and obfuscate Java packages; it's primarily used to reduce the size of the APK files.

As every little byte counts when you're working with mobile devices, ProGuard can be used to reduce the size of an APK.

The reduction is done in a few different ways. ProGuard goes through all the Java code and figures out which bits of code, that is, which libraries, aren't being called anywhere in the application. This can be a problem if the application is using certain advanced Java techniques, such as indirect calls to classes that aren't explicitly named in the code. But, in a simple application, simply minifying the application should work fine.

ProGuard also reduces the size of the application file by renaming methods, packages and classes to reduce the size of the identifiers. And this also serves to obscure the code, making it harder to read if somebody tries to reverse engineer it.

A very important step is to test the new APK, and make sure that it still works the same as it did before reducing it with ProGuard.

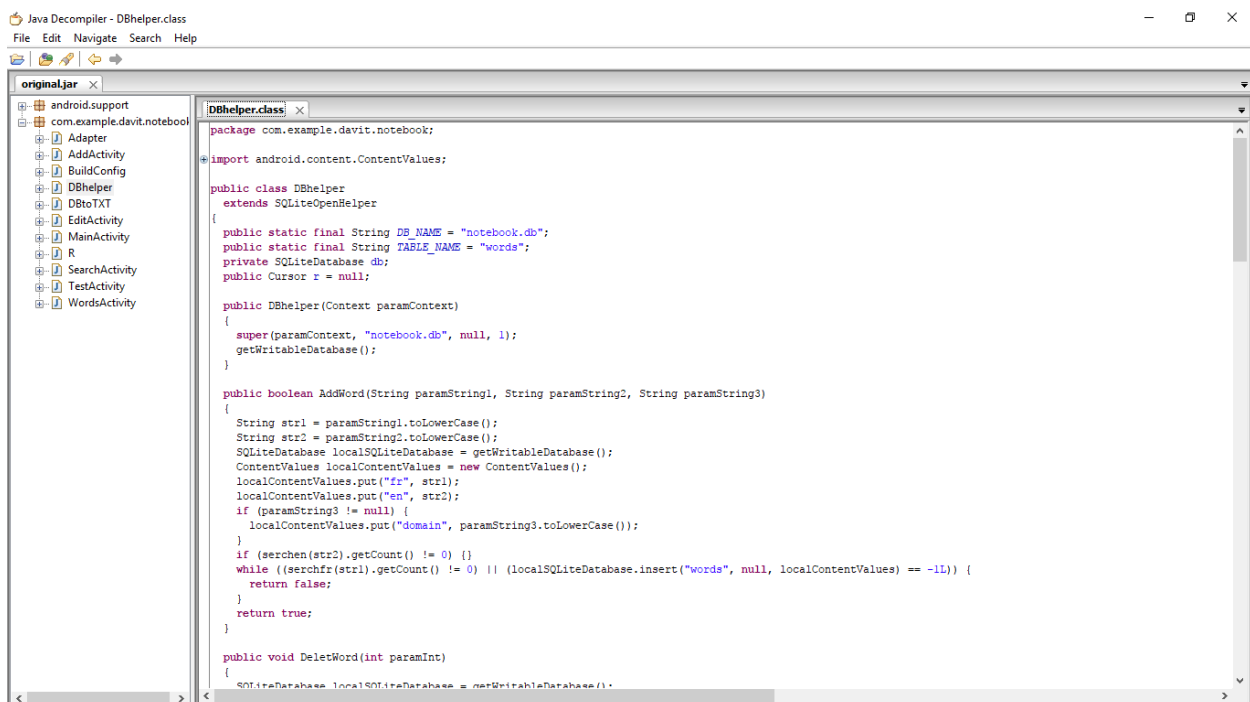
It is recommended to use ProGuard to obfuscate (minify) the compiled version of an app, as, apart from replacing all the class names, methods and variables with single-letter strings, it will also optimize the compiled bytecode and make it more complex. This makes ProGuard an additional barrier for hackers to hack an application.

To enable ProGuard, the build.gradle file needs to be modified as follows:

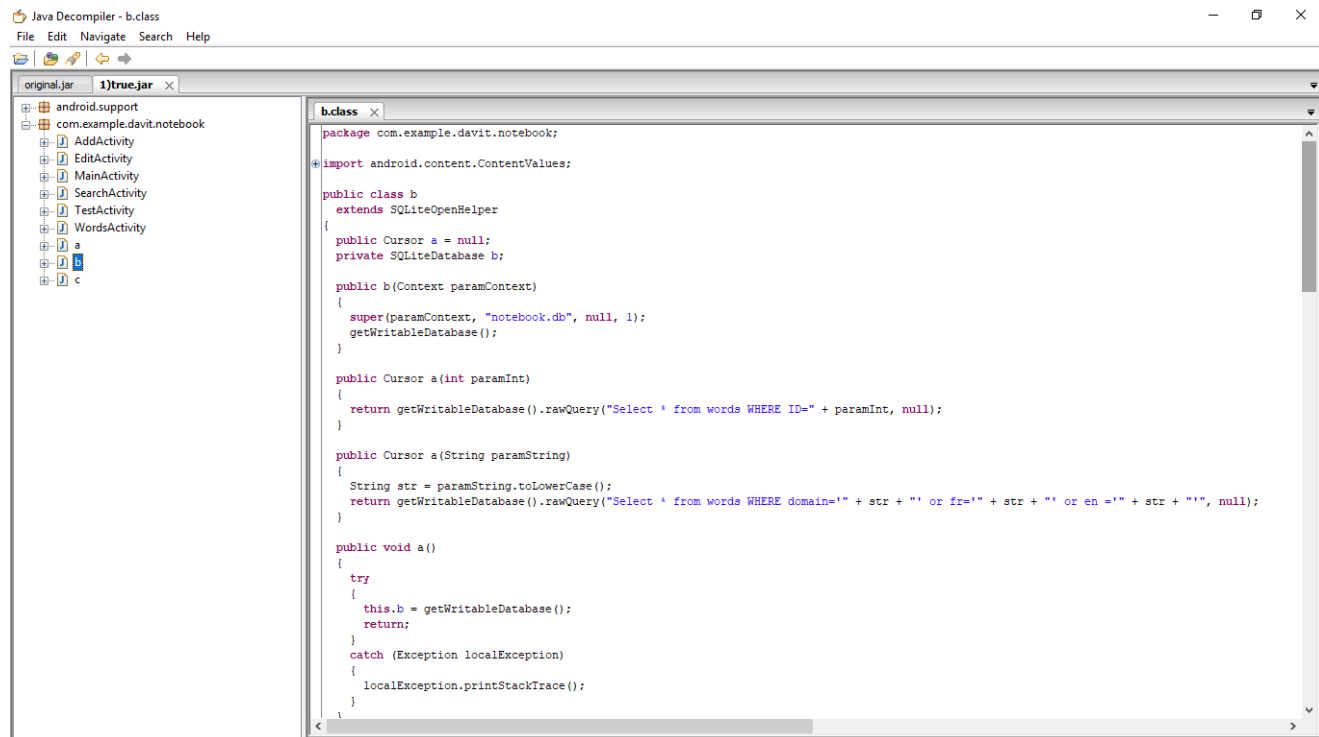
```
release {  
    minifyEnabled true // Enables code shrinking for the release build type.  
    proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-  
rules.pro'}
```

In the below print screens, we can see that variables, classes and method names are changed:

Before enabling ProGuard



After enabling Proguard



Some 3rd-party libraries can come with their own set of rules, ProGuard must be configured accordingly – these rules need to be added to the proguard-rules.pro file.

One of ProGuard's main pieces of impact is the optimization actions that the tool can perform, which help with the 64k method limit of the DEX files, which only allow 65536 methods to be referenced on a single DEX file.

If the app contains heavy-methods services which are split into several smaller packages, and not all of them used by the app, enabling ProGuard will remove all unused methods from the app, but also from the libraries included, which will decrease the number of methods used by the app.

When using APKTool to decompile an app built without enabling ProGuard, we can see rich information like source file name, the original variable and the method name of the code. But when using ProGuard, the above information will be hidden: no source file name is available, and the variables, classes and methods get renamed to a random alphabet character.

Using ProGuard can help a developer in adding an extra layer of security to the application, however, the obfuscation ability of ProGuard is limited, and shouldn't be used as an unique security mechanism.

4. Protecting against repackaging

4.1 Self verification

To protect against repackaging, the original developers of an Android app can use self-verification, which will check whether the app is modified or not at runtime.

There are several methods to enforce self-verification while developing apps:

- **DEX file verification**

The DEX file contains the fundamental functionalities of Android apps, so this makes it a target for app repackaging. The DEX file integrity verification is made by comparing its cyclic redundancy check (CRC) or hash to the predefined value. This predefined value can be stored in the app's local resources or it can be retrieved remotely from a server.

- **APK verification**

Sometimes, there's no need to modify the DEX file in order to accomplish an app repackaging. When using ads in free apps, there are some ads SDKs for which developers should specify the ad client IDs in the AndroidManifest.xml for the providers of the ads SDKs to pay the revenue generated by these ads. This client ID can be easily changed by hackers to redirect the revenue in their own favor. As the DEX file is not modified in this case, DEX file verification would not help. However, we can use the whole APK verification method, which has a similar logic as the verification of the Dex file, without any CRC or hash being involved.

- **Signature verification**

A main requirement when installing an Android app is for the app to be signed by the developer. The developer can use a self-signed certificate, or a certificate issued by a public Certification Authority. This signature serves to identify the developer and also to establish a trust between apps having the same signature. As each signature is unique, a repackaged app will be signed using 2 signatures: the original developer's one and the hacker's one – if the original one hasn't leak.

Self-verification cannot be used alone to achieve anti-repackaging – even if it ensures the integrity of the app and provides some mechanisms to fight repackaging. It is mandatory to use additional protection methods to ensure integrity of Android apps.

4.2 Packer

Android packers are used to compress or encrypt an original classes.dex file, decrypt the DEX file to memory at runtime, and then execute via DexClassLoader. The transformation is transparent for the end user as the app is automatically restored to its original state in memory at launching.

Packers can also use code obfuscation to harden the internal logics of an app, or can be used by attackers to reinforce malware to avoid signature-based detection and also hide the logic behind the malware.

Packers use different techniques to perform their actions:

- **Obfuscation** – aims to hide or prevent the understanding of the code. There are multiple techniques to perform obfuscation: modifying the name of the classes, fields or methods (ProGuard), reordering control flow graphs, encrypting constant strings, Java reflections,
- **Dynamic code modification** – malware can use native Java code to generate malicious bytecodes in a dynamic way and execute them in Dalvik Virtual Machine. Apart from changing the instructions in the Dex and Oat files, dynamic code modification can also impact key data structures in the memory (DexHeader, ClassDef, ArtMethod) to assure integrity of the contents during usage and then delete them after usage.
- **Dynamic loading** – at runtime, Android apps can load code (dex or jar format) from other sources, taking advantage of this feature, packers encrypt the original dex file and before running the app, the file is decrypted and loaded.
- **Anti-debugging** – packers can attach to another process for debugging using ptrace to prevent debugging through gdb. If a target process attaches to itself, gdb will not be able to attach to it, so further debugging operations are prevented.

5. CONCLUSION AND FUTURE WORK

In this paper we talk about ways of repackaging and ways of protecting application. It is important to understand that even if you protect your application you cannot be sure that no one can repackaging it, there are always vulnerabilities which are undiscovered. What can be done to fight hackers is to make repackaging harder by using same tools and technologies to secure the applications. In the future, I would like to create a method of protecting Android application using remote server.

Bibliography

Learning Android Application Development, Raimon Rafols Montane Laurence Dawson, Packt Publishing Ltd, 2016

Decompiling Android, Godfrey Nolan, Apress, 2012

Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces, Wu Zhou, Yajin Zhou, Xuxian Jiang, Peng Ning, North Carolina State University

Android Malware: Detection, Characterization, and Mitigation. (Under the direction of Xuxian Jiang.) Zhou Yajin

Protecting Mobile Networks and Devices: Challenges and Solutions, Weizhi Meng, Xiapu Luo, Steven Furnell, Jianying Zhou, CRC Press, 2016

Internet sources:

Apktool

<https://ibotpeaches.github.io/Apktool/>

dex2jar

<https://tools.kali.org/reverse-engineering/dex2jar>

ProGuard

<https://www.guardsquare.com/en/proguard>

Intents and Intent Filters

<https://developer.android.com/guide/components/intents-filters.html>

Security Tips

<https://developer.android.com/training/articles/security-tips.html>

Android obfuscation tools – ProGuard, Karlo Mravunac

<https://sgros-students.blogspot.be/2017/04/android-obfuscation-tools-proguard.html>

I, Davit Pogolian, declare this term paper to be my , own original work