Repackaged Android Apps: How to Fake and How to Fend

Remmy Chen

COMP 116: Introduction to Computer Security

May 1, 2018

Abstract

Android is an open source mobile operating system released by Google [1], with around 86% of global market share as of 2017 [2]. The openness of Android operating systems can be seen in the high level of device fragmentation, operating system fragmentation, and app market fragmentation. Combined with the ability to reverse engineer existing apps, Android becomes particularly susceptible to security concerns relating to confidentiality, integrity and availability. This paper will explore principles behind Android that makes it prone to security vulnerabilities, methodologies that allow original apps to be recompiled and distributed to end users, and ways to fend against reverse engineered Android apps.

Introduction: Brief background on Android and repackaged Android apps

Android is an open source mobile operating system based on the Linux kernel and other open source software [1]. It was first unveiled by Google in 2007, the same year as when IOS, a mobile operating system created and developed by Apple exclusively for its hardware, was unveiled [3]. Android is the most popular mobile operating system globally, with around 86% of global market share as of 2017 [2].

Android is carried by devices from a number of manufacturers, including Google, Samsung, Sony, HTC, and Nokia [4]. Android devices come in all shapes and sizes, with variations at all levels including low level chipset support, screen sizes and performance levels [5]. There are many different versions of Android that are in use, with very low adoption rate of the latest operating system updates [5]. For example, as of February 2017, less than 2% of Android phones used the latest major version of Android, Android Nougat, which were released at the end of August 2016 [6]. An implication of the low adoption rate of the latest operating system updates is that the majority of Android devices in use have Android versions that are likely to have known, unpatched security vulnerabilities.

Apart from fragmentation in brands, models, operating system versions, Android devices also have a fragmented application market. While Google Play Store is the official and the predominant source of Android applications, many other channels exist, such as websites that offer Android applications and other Android applications marketplaces. Reasons that some Android applications don't get published to Google Play Store but get published elsewhere include that there is a cost of publishing applications to Google Play Store, that the application doesn't comply with Google Play Store rules, and that Google Play Store is not accessible to a desired market, such as China. The abundance of application sources makes it difficult for applications downloaded to Android devices to be regulated.

The lack of regulation surrounding Android perhaps is intentional in the spirit of open source collaboration. However, from a security standpoint, this makes Android incredibly vulnerable. Without sufficient checks in the Android ecosystem, an end user, the target of malicious actors, may inadvertently download a mobile application from some source, grant more than necessary permissions to the application, and allow the application pretty much to lurk and attack without the end user's knowledge.

An example of malicious applications is repackaged applications, which will be the focus of this paper. Disguised as legitimate applications, repackaged applications carry malicious modifications, such as advertisement wrappers, data logging, and malware, and distributed as nearly identical copies of legitimate applications. Repackaged applications are created by unpackaging legitimate applications to source code, injecting, modifying, or lifting proprietary code, and repackaging the source code into applications. Repackaged applications then get resigned, published, distributed through various channels, and finally (re)installed on an unsuspecting user's device.

To the Community: Why this topic is important

Mobile security becomes increasingly important as more personal and business information get stored on smartphones. In particular, defending against repackaged applications is important because there are few security and protective measures that prevent republished, malicious applications from being distributed [7]. For example, in 2017, it was discovered that a fake version of the WhatsApp messenger application, which contained an advertisement wrapper, was downloaded more than a million times from the Google Play Store before it was removed [8]. The application, "Update WhatsApp Messenger", appeared to have been developed by the firm behind the real application - WhatsApp Inc. by adding an invisible Unicode character space after WhatsApp Inc. [8]. The million plus downloads before flags were raised is worrisome because it demonstrates how even the Google Play Store vulnerability scans may be easily bypassed. It is no surprise then, that code tampering and reverse engineering are both in the Open Web Application Security Project (OWASP) Top 10 Mobile Risks 2016 List, to stress again of the urgency that should be paid to repackaged applications [9].

Repackaging an application
(source code: https://github.com/RemmyChen/RepackagedAndroidApplications)

PHASE 1: INVESTIGATION - APPLICATION DECOMPILATION

In the first quarter of 2018, Japan's hit mobile game Travel Frog by Hit-Point Co.,Ltd. topped application stores in mainland China and Taiwan [10]. The first time I opened the Mandarin version of Travel Frog, which was downloaded from the mobile application store Mi Store on my Xiaomi phone, it asked for a number of permissions. Each time I opened the application, a splash page showing advertisement (of Xiaomi phones sold on a third party platform) would be shown, and if I clicked on the page, an automatic background download of the third party platform's application would be started presumably so that I could purchase what was advertised. The Mandarin verison of Travel Frog had tens of millions of downloads on Mi Store, with over ten thousand reviews many of which complained about advertisements. Similarly, the Japanese version of Travel Frog hosted on Mi Store displayed advertisement and received many negative, advertisement-related reviews. Interestingly, looking at the Japanese version of Travel Frog on Google Play Store, there weren't any advertisement-related reviews.

To investigate further, I downloaded the .apk files of the Japanese version of Travel Frog from Google Play Store (via https://apkpure.com/ on my laptop) and of the Mandarin version of Travel Frog from Mi Store (on my smartphone, transferred to my laptop via Android File Manager). Then I ran $apktool d [apk file] to decompile the .apk files into AndroidManifest.xml [11], resource files, and Smali code. Looking at AndroidManifest.xml, one would immediately notice that in addition to permissions for internet, vibrate, billing, and access network state requested as with Google Play Store's version, Mi Store's version also requested permissions for internet, write external storage, read phone state, get tasks, access wifi state, access course location, and access fine location. In addition, one would notice that instead of having net.gree.unitywebview.CUnityPlayerActivity as the launcher, leanback launcher, and main as with Google Play Store's version, Mi Store's version had com.google.littleDog.SplashActivity as the launcher and main, with net.gree.unitywebview.CUnityPlayerActivity as the leanback launcher in AndroidManifest.xml. Poking around further, while advertisement images weren't in resource files, one would find that in the assets folder of Mi Store's decompiled apk were two more apk files, one called AdServer.apk and one called analytics_core.apk. Searching on the web, a page on the Mi Developer Platform website https://dev.mi.com showed instructions of how one can insert a package of wrappers into applications, including one called AdServer.apk and one called analytics_core.apk, possibly allowing developers to profit from advertisements and Xiaomi to collect data [12]. Checking on the Mi Store again, the developer is listed as Hit-Point Co.,Ltd., which is the same developer as for the Japanese version of Travel Frog from Google Play Store. Suppose the Mandarin version of Travel Frog on Mi Store, contrary to what is listed, is not be from the original developer, one might wonder about possible ramifications if the developer made the app more malicious when tampering with the source code.

PHASE 2: ATTACK - APPLICATION RECOMPILATION

The decompiled .apk file of Google Play's Japanese version of Travel Frog may be used as the source code to build a simple repackaged application. To ensure the source code works, I ran $apktool b [source code folder] to get an .apk file in the /dist folder that is then transferred to my phone [11] [12]. Android requires that mobile applications be digitally signed with a certificate in release mode before they can be installed on phones so as to identify the author of an app, so the modified .apk file needs to be resigned, which will change the certificate [13]. The signing may be done via command line (i.e. using keytool, jarsigner, and zipalign) or via mobile applications (i.e. ZipSigner) [11]. The repackaged and resigned application is then installed and run on my phone.

I then built a demo application called HelloWorld in Android Studio with a main activity that just displays Hello World into an .apk file [11] [12]. In addition, I built a demo application called HackedToast in Android Studio with a splash page containing an image displayed for nine seconds prior to the main activity plus a toast that says "Gotta love ads!" - these are the modifications I want to transplant to Google Play's Japanese version of Travel Frog. The two .apk files were decompiled using apktool to determine where the differences in source code are. To practice, I planted the ten-second advertisement splash page and the toast from the HackedToast application into the HelloWorld application. Specifically, I added the image file and a splash.xml file to a res/drawable folder, added a splash theme in res/values/styles.xml,

modified the string "HelloWorld" in the main activity in res/values/strings.xml, added the splash theme as a label in AndroidManfifest.xml, and added to the Smali code. Smali is the assembly language used by the Android Dalvik Virtual Machine. An advantage of working with Smali is that because it maps quite directly to Dalvik Executable, it is not lossy to convert between Smali and Dalvik Executable. A disadvantage is that it is in assembly language, which is harder to work in when dealing with large code modifications. An alternative that is unexplored in this demo is using dex2jar and JD-GUI to obtain Java source code from Dalvik Executable that is good enough to understand what the code is doing but that is not necessarily lossless enough to rebuild into an .apk file [12].

Taking a look at the code base in Travel Frog, which is significantly bigger and built in game development platform Unity, the first task is to determine what the first activity is. This may be done by looking at which activity is the main and launcher in AndroidManifest.xml. Having determined the main activity, I planted the Smali code for the ten-second advertisement splash page and the toast in Google Play's Japanese version of Travel Frog. Then I moved the resources in just as I had with HackedToast. It took a number of tries to get the repackaged Travel Frog to run because though the main and launcher activity listed in AndroidManifest.xml probably is the starting activity, it has a super or parent class activity, perhaps because of how the code base is structured in an application that uses Unity.

The repackaged Travel Frog ultimately displays an advertisement briefly before the game runs, whichi s relatively benign. The repackaged Travel Frog doesn't hold the splash screen for the nine seconds that I hardcoded the runnable handler to do and I had to silence the toast because I couldn't find the memory address for the main activity needed by the Smali code I was using - it should have been in res/values/public.xml or in layout.smali like HelloWorld but I suspect here again Unity is slightly different with the way layouts are coded. However, even then, the repackaged application seems sufficient to demonstrate how one may repackage an application.

Conclusion

Generally, all mobile applications are susceptible to repackaging. there are a few ways to defend against repackaged malicious applications. On the implementator side, one may use Android ProGuard and other tools to shrink and obfuscate code such that the application's control flow path and methods are made obscure, while keeping in mind of performance impact [15]. It is also possible to use checksums, digital signatures, and other validation mechanisms to detect tampering and to prevent repackaged applications from running. On the client side, one may look at the developer's information, the number of downloads, the reviews, and the permissions that an application requests [16], which is a necessary but insufficient step to guard against repackaged applications, as illustrated with the Chinese and Japanese versions of Travel Frog hosted on Mi Store, the default application store for Xiaomi phones. One may also visit the developer's website to download an application. However, ultimately these steps may all be bypassed and perhaps it is most fruitful for action to be taken on the market side, with more regulations, standardization and checks for applications' and developers' authenticity across the Android universe.

References

[1] "Android Open Source Project." Android Open Source Project. Accessed May 01, 2018. https://source.android.com/.

[2] Mobile OS Market Share 2017." Statista. Accessed May 01, 2018. https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/.

[3] "Which Came First? IOS or Android." TechizBlogSG. Accessed May 01, 2018. http://techizblogsg.weebly.com/which-came-first-ios-or-android.html.

[4] Betts, Andy. "Which Smartphone Manufacturers Are Best for Android Updates?" November 16, 2017. Accessed May 01, 2018. https://www.makeuseof.com/tag/smartphone-manufacturers-best-android-updates/.

[5] OpenSignal.com. "Android Fragmentation Report August 2015." Android Fragmentation Report August 2015 - OpenSignal. Accessed May 01, 2018. https://opensignal.com/reports/2015/08/android-fragmentation/.

[6] Perez, Roi. "Fragmentation-nation: Only 1.2% of Android Devices Use Latest OS." SC Media UK. February 24, 2017. Accessed May 01, 2018. https://www.scmagazineuk.com/fragmentation-nation-only-12-of-android-devices-use-latest-os/article/640080/.

[7] Kassner, Michael. "Malware Scanning of Mobile Apps Needs Serious Help." TechRepublic. Accessed May 01, 2018. https://www.techrepublic.com/article/malware-scanning-of-mobile-apps-needs-serious-help/.

[8] Sulleyman, Aatif. "Fake WhatsApp App Tricks More than a Million People into Downloading It." The Independent. November 06, 2017. Accessed May 01, 2018. https://www.independent.co.uk/life-style/gadgets-and-tech/news/whatsapp-app-android-messaging-apps-fake-google-play-a8039806.html.

[9] "OWASP Mobile Security Project." OWASP Mobile Security Project - OWASP. Accessed May 01, 2018. https://www.owasp.org/index.php/OWASP_Mobile_Security_Project#tab=Top_10_Mobile_Risks.

[10] Chow, Vivienne. "Japan's Hit Mobile Game "Travel Frog" Teaches a Philosophical Lesson about Letting Go." Quartz. February 08, 2018. Accessed May 01, 2018. https://qz.com/1198149/japans-hit-mobile-game-travel-frog-is-teaching-a-philosophical-lesson-about-letting-go/.

[11] Mahaseth, Neil. "Decompile APK Get Java Xml and Mod APP [Ultimate Guide]." How I Solve. November 11, 2017. Accessed May 01, 2018. https://www.howisolve.com/decompile-apk-get-java-xml-change-apps/.

[12] “How to Decompile, Get Java Source Code, Build and Sign an Android App." Master-Console_. February 24, 2018. Accessed May 01, 2018. https://masterconsoleblog.wordpress.com/2017/07/01/how-to-decompile-get-java-source-code-build-and-sign-an-android-app/.

[13] 小米广告 SDK 集成指南 - 小米开放平台 (Xiaomi Advertisement SDK Instruction Guide – Xiaomi Development Platform). Accessed May 01, 2018. https://dev.mi.com/docs/gameentry/手机&pad 游戏接入文档/小米广告 SDK 集成指南/#4-faq.

[14] "Signing Your Applications." Android Developers. Accessed May 01, 2018. http://www.androiddocs.com/tools/publishing/app-signing.html.

[15] "Shrink Your Code and Resources | Android Developers." Android Developers. Accessed May 01, 2018. https://developer.android.com/studio/build/shrink-code.html.

[16] Kleinman, Jacob. "How to Spot Fake Apps in Apple's App Store and Google Play." Lifehacker. December 19, 2017. Accessed May 01, 2018. https://lifehacker.com/how-to-spot-fake-apps-in-apples-app-store-and-google-pl-1821428717.

Acknowledgements

Thanks to Tufts University for the resources and education that I've been able to receive and thanks to Professor Ming Chow and the instruction team for a wonderful course and guidance!

Addendum 1: Steps of Repackaging [11] [12]

- decompile and recompile app:
  1) get apk via phone or web (i.e. adb, https://apkpure.com/)

  2A) apktool: android manifest, resource files, Smali files (difficult to manipulate large code)
  - install apktool (i.e. https://ibotpeaches.github.io/Apktool/)
  - $apktool d [apk] to decompile apk
  - make modifications
  - $apktool b [decompiled apk folder] to build apk

  2B) dex2jar, JD-GUI: android manifest, resource files, Java files (lossy)
  - change .apk file's extension to .zip
  - $unzip [zip file]
  - download dex2jar (https://github.com/pxb1988/dex2jar)
  - $./d2j-dex2jar.sh classes.dex to generate classes-dex2jar.jar file
  - download JD-GUI (http://jd.benow.ca/ or http://macappstore.org/jd-gui/)
  - drag and drop classes-dex2jar.jar into the JD-GUI window.
  - go to File > Save All Sources to save -dex2jar.jar.src.zip
  - extract the zip file and to get all java files of the application
  - make modifications
  - $apktool b [decompiled apk folder] to build apk

- copy conterfeit app back to phone to test:
  1) transfer modified apk to phone
  2) install ZipSigner application
  3) generate resigned .apk
  4) install .apk and run application

- distribute counterfeit app:
  1) pretend to be original developer and put app onto app stores
  2) get high ratings and many downloads
  3) don't get caught