The Android Labyrinth: combating application repackaging attacks[1]

Authors
Rowena Harrison, MSc (Royal Holloway, 2014)
Keith Mayes, ISG, Royal Holloway

Overview

In the world of selfies, the twitterverse and the world wide web, smart phones have become an integrated part of everyday life for most, making them a target for malicious attacks. One particular threat facing the smart phone operating system Android, is from repackaged applications; that is legitimate applications that have been reverse engineered, modified to include malicious code, repackaged, and then distributed for download. This malicious code or malware within apps can have destructive consequences on unsuspecting smart phone users, such as sending premium rate SMS, gathering personal information or even stealing money.

Software developers wary of the reverse engineering process and attack method will obfuscate their code, trying to make it harder to read the code and understand its function, through a variety of techniques and software. One software tool available from the makers of Android, called ProGuard is designed to obfuscate Android applications. Its effectiveness was tested by reverse engineering two versions of an app created by the author, one with ProGuard applied to it, and one without, giving some interesting results.

ProGuard was found to be an effective obfuscation tool and also comes free with the development environment for Android, making it easy to incorporate and worth implementing. However, the tool was not infallible and several clues were left behind, leading again to the age-old security proverb: Don't rely on security through obscurity!

Smart phones have everything these days, the Internet, Global Positioning System (GPS), pedometers, notepads, you name it. But one of the most important aspects of smart phones is the ability to download applications (apps) from app-stores, sparking the shift from non-smart to smart phones in recent years. In fact, it's estimated that in 2016, the number of smart phone users worldwide will exceed 2 billion, and by 2018 will have surpassed the number of feature phones completely.

One significant player in the smart phone arena is Android, a Linux-based operating system, released in 2007 by Google. It has grown rapidly and now dominates - from a market share of 2.8% in 2009 to 84% at the end of 2014, and is the operating system used by several smart phone makers, for example, Samsung, Google, HTC, Sony, LG and Motorola.

Google Play (formerly Android Market) developed by Google, is the primary dispensing platform for Android apps, allowing users to browse and download paid or free apps. This

[1] This article is to be published online by Computer Weekly as part of the 2015 Royal Holloway info security thesis series. The full MSc thesis is published on the ISG's website.

app store is conveniently available via the Google Play app, automatically installed on each Android smart phone. There are also other third party app stores such as GetJar and SlideME.

The popularity of the Android operating system, and subsequently its applications, has made it a major target for malicious behaviour, mimicking the trend between Microsoft Windows' majority market share and the number of malware attacks on personal computers. In fact, 99% of all phone malware is targeted at Android devices.

With the widespread download of smart phone apps and the access to millions of apps via app-stores, there is a serious threat from reverse engineering of phone apps. Apps can be easily plagiarized, have malware incorporated in them through reverse engineering by malicious entities and re-distributed onto third-part markets, a so-called repackaging attack.

A **repackaging attack** is when an app is reverse engineered, modified to include malware and then re-uploaded to app stores.

What is reverse engineering?

In general, reverse engineering is the process of analysing an object or system's operation to determine its original design. This process has been used throughout the course of history, and has even influenced the outcomes of wars; the examination of encrypted German communications lead to the reverse engineering of the Enigma machine, allowing the allies to decrypt and understand German messages. However, in a software engineering context, it is the method of obtaining the high-level source code of a program from examining the machine code, or other types of low-level code.

Reverse engineering: obtaining high-level source code from examining low-level codes.

Reverse engineering is used at both ends of the spectrum. Malicious developers reverse engineer software and operating systems to find vulnerabilities that can be exploited to access sensitive information or to facilitate taking control of a system. Anti-virus developers scrutinise examples of malware in order to calculate the extent of damage that could be caused and find out ways to remove it from a system.

Repackaging of apps is when an app, created and signed by one developer, is reverse engineered and then re-released onto app stores, signed by another developer. The re-released app is usually modified from the original one. An example of a repackaging attack is shown in the figure below.

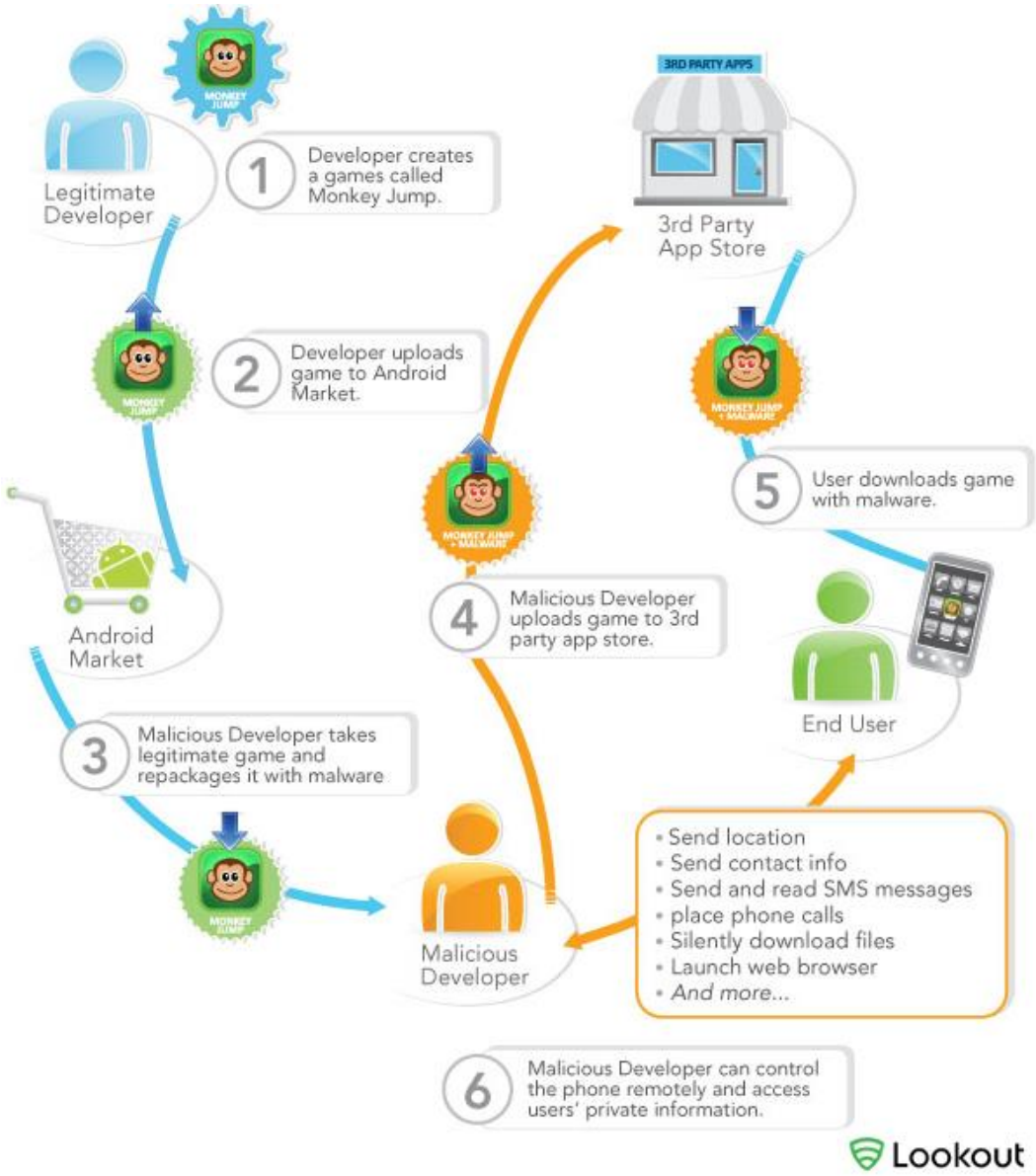

Figure 1. The Repackaging Process. Taken from "Lookout Mobile Threat Report 2011''. [Online]: https://www.lookout.com/resources/reports/mobile-threat-report-2011. [Accessed: Aug 2014].

How might a reverse engineered app be exploited?

So what is to gain from repackaging apps? Apps can be plagiarized, altered and repackaged for monetary reasons; a paid app could be re-released for free, ``cracking'' the application, and a plagiarist could divert revenue from advertising towards himself by altering the ad libraries. They can also be repackaged with malware for more dangerous and malicious purposes.

There are several types of malware known to computers such as worms, trojans, viruses and spyware. However, these malwares have morphed to suit their new environment of smart phones in recent years. In the 2012 survey "*Android Security, A survey. So far so good",* Khan and Tahir classify malware for smart phones:

- Ad-Ware - Malware advertisements on smart phones.
- Destructive - Examples include deletion of phone contacts with no user knowledge.

- Direct Payoff - Malware that sends an SMS without user consent.
- Premeditated Spyware - Remote listening and tracking.
- Proof of Concept - Malware that proves a particular attack can work (e.g. leaving Bluetooth running to drain the battery without user knowledge).
- Information Scattering - Checking address books, cookies and passwords without user consent.

As stated previously, attackers may have financial motives to change advertising libraries, but what other kind of things can repackaged malicious apps do with these different types of malware?

A paper by Jung et al. (*Repackaging Attack on Android Banking Applications and Its Countermeasures,* Wireless Personal Communications, 73(4), 1421-1437, 2013) showed that a serious attack on a banking app was "possible without having to illegally obtain any of the sender's personal information, such as the sender's public key certificate, the password to their bank account, or their security card''. The attack involves an innocent user installing a malicious repackaged banking app developed by an attacker. When the user tries to send an amount of money to the intended recipient, the money is transferred to the attacker instead. The attack is laid out in the figure below.

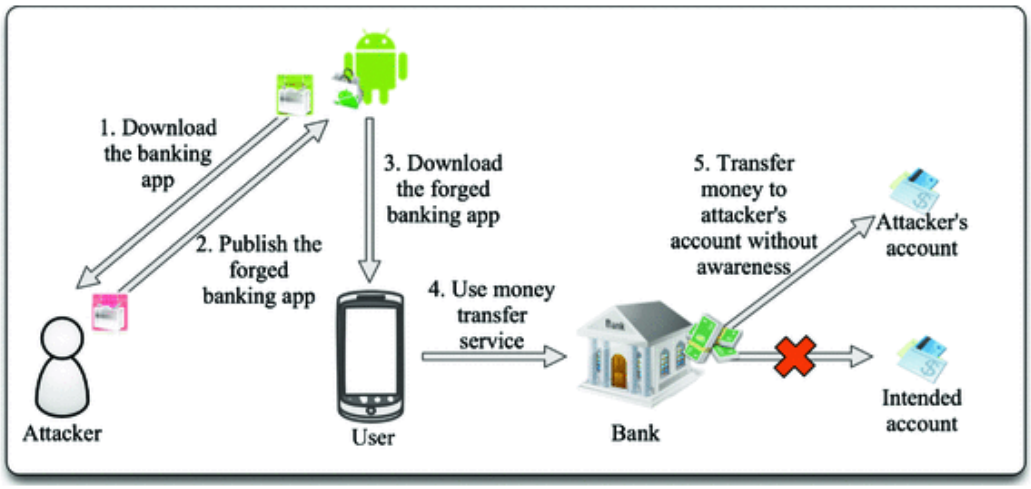

Figure 2. Repackaging Attack on Banking App. Figure taken from Jung et al. *Repackaging Attack on Android Banking Applications and Its Countermeasures,* Wireless Personal Communications, 73(4), 1421-1437, 2013.

The simulated attack was successful. The antivirus routine was skipped, integrity checks were forged and messages were replayed in order to trick the banking server into believing that it has sent the correct amount to the intended recipient, whereas it has sent the amount to the attacker. This scenario demonstrates the serious dangers of vulnerable code being repackaged and let loose on Android markets. With a total of 48 billion app downloads from Google Play alone, the potential impact of real repackaged banking apps similar to this one could be catastrophic.

Tools for reverse engineering apps?

If you were thinking of a hacker tapping away at a keyboard for hours in a dark room, you are mistaken. Any script kiddie with a basic understanding of code can do it; there is an

abundance of tools readily available to reverse engineer any app on an online store, which don't require too much thought to use.

The first tool designed for this process was *undx* created by Marc Schonefeld. This tool converts Dalvik bytecode (the code needed to run apps on smart phones) back into Java bytecode (the code which the app is written in), a .jar file, which then can be fed into tools such as a *Java Decompiler (JAD)* or *Java Decompiler Graphical User Interface (JD-GUI)* to obtain the source code. A more sophisticated tool called *dex2jar* was created to overcome issues *undx* had with more complex code.

Another way to alter apps is to use smali/baksmali, an assembler/disassembler. The baksmali disassembler translates the Dalvik bytecode in to a readable arrangement, the smali code. This code can then be changed and recompiled to use on an Android device. Many academic papers use baksmali to study app repackaging. *ApkTool* uses the baksmali disassembler to obtain smali code of an app, but also has the ability to repackage it.

With ever expanding online resources, it is easy to see why so many Android apps are repackaged. Numerous tools are readily available online with tutorials which walk through the steps to reverse engineer and repackage apps, making this process straightforward, even for those who know little about app development.

Experiments in reverse engineering

To test the reverse engineering process, I created my own app called RowenApp using the Android Software Development Kit (SDK). RowenApp is designed to store personal banking information of the user or users in a database and then find and retrieve this information when requested by the user. The app stores the personal information in an encrypted format. An example of one of the screens in RowenApp is shown below.

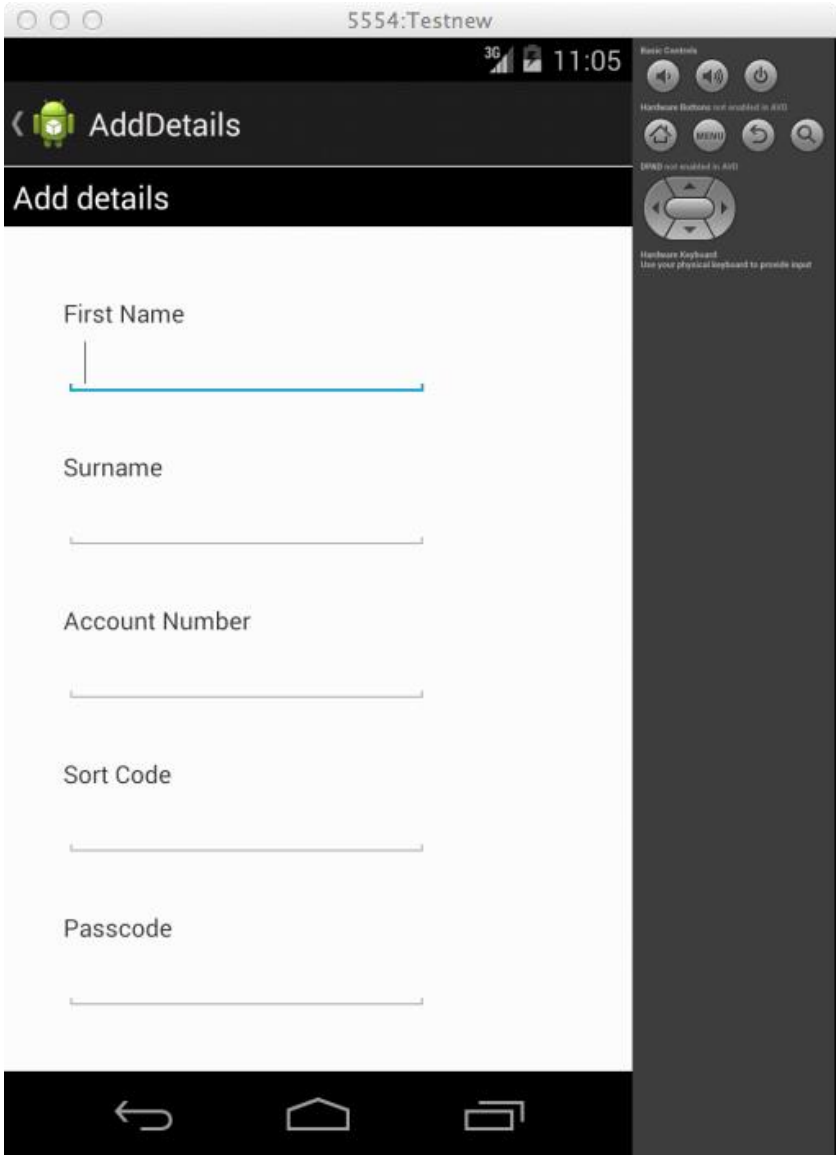

Figure 3. The AddDetails activity screen of RowenApp. This is where users input their information to be stored by the app.

I also had a go at reverse engineering RowenApp using *ApkTool* and *dex2jar* and found that the output of both tools were scarily accurate. It was easy to determine the nature and purpose of the application with both tools however, the two tools created different results.

The output of *ApkTool* is in the smali code format, which has a low-level assembler-like syntax, making it difficult to read as someone who has the majority of their coding experience in high-level languages like C and Java. This type of syntax also increases the number of lines of code, making it a more time-consuming process to determine the purpose of RowenApp.

Dex2jar's output is in Java, a high-level language, making it a lot easier to understand the code and what it does. However, unlike the smali code outout of *ApkTool*, variable names were changed to non-contextual names, which made keeping track of variables throughout

the code more confusing. *Dex2jar* also didn't come with any of the application resources like *ApkTool* did. The application resources are things like layout, style and icon files.

To get a full picture of the application, both tools could be used together, *dex2jar* for the source code in Java which is easier to understand, and *ApkTool* for finding the resources which are not present in the dex2jar output.

Can app developers protect against reverse engineering?

Code obfuscation: making code as unreadable to humans as possible.

So two freely available tools I got off the Internet were able to successfully reverse engineer my app, and have accurate enough outputs so someone could understand the app code. Is there anything developers can do to make repackaging attacks more difficult? Yes, and the answer is code obfuscation.

Code obfuscation purposefully tries to make code as hard as possible for humans to read, in order to conceal its inner workings. Obfuscation, like reverse engineering, is done by both legitimate software developers, and creators of malware; the first, to make it harder to plagiarise their work, and the latter, to elude anti-virus detectors. Code obfuscators are often steered towards combating particular reverse engineering stages, to reduce the effectiveness of tools and heuristic methods.

Why obfuscate?

- Make legitimate code hard to plagiarise, but also
- Elude anti-virus detectors.

There are a number of techniques that can be employed to obfuscate code:

- *Identifier mangling* - changing the names of classes, methods and fields, called identifiers, from meaningful words to meaningless characters. Identifiers, in personal programs, can provide information about the code and the intention of the program, making the reverse engineering process less time-consuming as human inspection and analysis will be easier.
- *String Obfuscation* – removes meta-data about the program by passing strings (arrays of characters used within the program for user interaction) through an invertible transformation function and storing them within the code. By running the inverse transformation on this result, the original string is generated when needed at runtime.
- *Dynamic Code Loading* – this is where the whole program can be encrypted. Like string obfuscation, the program can be decrypted at runtime before it is executed.
- *Self-modifying Code*- this type of code transforms itself whilst being executed; the instructions observed in the code initially may not be the code running when the application is being executed.
- *Junkbytes* - This technique alters the flow of the program by adding in lines of code, which is never executed. This confuses disassemblers by introducing errors and diverts attention away from the true meaning of the code.

There are also several obfuscation tools available which use a variety of the techniques listed above. I used the tool *ProGuard*, a Java source code obfuscator included in the Android SDK to obfuscate my app and create RowenAppobsf. *ProGuard* obfuscates the code by applying identifier mangling to the code, but also shrinking the code to remove unused

methods and field. It was very easy to implement as well, I simply uncommented a piece of code in one of the configuration files.

Experimental findings and conclusions

From studying the tool outputs from RowenApp and RowenAppobsf, the obfuscation process used by ProGuard definitely made understanding the application more complex; the output from both tools operating on RowenAppobsf, found the context and purpose removed from classes, methods and variables.

A more laborious and time-consuming examination of the code was required to understand RowenAppobsf, taking me over four hours to examine the output from both tools, compared to RowenApp which took only one hour. From this, the code obfuscation by *ProGuard* was therefore effective, also reinforced by the fact that I (although not a knowledgeable reverse engineer) had the advantage of knowing the app's function and source code, and could compare the unobfuscated and obfuscated versions.

Configuring and applying *ProGuard* to RowenApp was very simple and it definitely adds difficulty into the reverse engineering process, so it is worth doing. However, I was still able to deduce the context and purpose of the application through the examination of a file called f. This file included column headings in the database I used that were not obfuscated by *ProGuard*, allowing me to understand small pieces, which I could then fit into a bigger picture.

The obfuscation offered by *ProGuard* did make the reverse engineering process more difficult, but it was still practical. This leads to the conclusion that free tools may not be sufficient at protecting against reverse engineering, so developers generally should not rely on any aspect of the app code remaining secret for security. However the bigger problem when trying to protect apps is the unrestricted access of malicious parties to powerful tools, online tutorials and resources, which cannot be solved by applying tools or processes to the app code.

To conclude, reverse engineering of phone apps is practical and appears widespread, and whilst obfuscation techniques make reverse engineering more time consuming, they do not prevent it. Therefore app developers and service providers should assume that attackers may have complete knowledge of their published apps and the potential for creating maliciously modified variants.

Biographies

Rowena Harrison first started out at Imperial College London, studying a Physics Masters, where her passion for all things technical and computing began. After graduating in 2013, she moved to Royal Holloway to complete an MSc in Information Security, following her desire to develop a strong theoretical foundation in the subject. Rowena now works at Corsaire, a security consultancy based in Woking, Surrey, gaining practical and technical expertise in the information security sector, building upon the knowledge from her MSc. She is currently working towards becoming a CREST certified tester.

Keith Mayes B.Sc. Ph.D. CEng FIET A.Inst.ISP, has spent much of his life working in/with industry, yet is also an active researcher/author with 100+ publications. He is Director of the ISG Smart Card Centre at Royal Holloway University of London and of Crisp Telecom Limited. He has worked in hardware/software development, DSP and sensors, standardisation,

mobile communications, smart cards/RFIDs, embedded systems, systems modelling plus diverse aspects of information security. Current interests include (but are not restricted to) mobile comms and trusted execution technologies, NFC, Smart cards/RFIDs, transport ticketing systems, automotive security, m-commerce, attacks and attack resistant system implementations.