Giving the User Control over Android Permissions

6.858 Final Project - Fall 2012

Jonas Helfer & Ty Lin

{helfer,ty12}@mit.edu

December 15, 2012

Abstract

In this project, we investigate the possible options for users to restrict application permissions in a more fine-grained way and provide a proof of concept for a command-line tool that can remove permissions from applications before installation. We tested our tool with Android 2.2 (Froyo) on an emulated NexusS and a real Samsung Galaxy S GT-I9000.

Introduction

Android's permission system is a crucial part for the overall security of the OS, because it ensures that thrid-party applications can safely be run on the phone without negatively impacting security or usability [2]. However, under Android's current permission system, when a user downloads an application, he has the choice between accepting all permissions the application asks for, or not installing it at all. There is no user discretion for granting permissions and there is currently no way to modify an android app's permission after it was installed.

On the Android market (Google play) there are a number of apps for download that restrict permissions at runtime on a per-app basis, but for the most part they are not very effective and tend to lead to application crashes according to user reviews. We did not try the apps ourselves, because they require a rooted phone. The fact that there are a number of apps out there, that aim to restrict permissions shows that users would appreciate a more fine-grained permission control in Android. It also shows that restricting permissions under the current model is challenging and has no perfect solution yet. There are a few applications on the Android market such as the Privacy Blocker [1] which costs $1.99 to download. Unfortunately, quite a few reviews indicate that the application is actually not very effective and one reviewer even quantifies that it might work on one out of ten apps. There are also free applications on the market, including PDroid Privacy Protection [2] which is free but requires rooting your Android device.

Even with a permission restricting application that works perfectly, the user would still need to trust the application developer that the app actually does what it promises. Furthermore, if the blocking application requires root access, the user is essentially giving it all permissions and more, thus implicitly trusting it as much as the OS.

We propose to modify Android applications before installation, thus requiring no more trust from the user in our tool than he is already placing in the application. A clear benefit of our approach is that enforcing permissions is still up to the OS, thus giving the user more confidence that the permissions are effectively enforced.

The number of phone models using the Android platform is very large and unfortunately there is no guarantee that an application developed for one phone will work on another, even when they use the same version of Android OS. Google provides an emulator for standard Android phones such as the Nexus S, but there is no way of telling, whether an app will also work on a real phone without testing it on that particular phone. Because the only phone we had was a Samsung Galaxy S GT-I9000 with Android 2.2, we ran all our tests on that phone and an emulated Nexus S with the same OS version. The applications that we focused on include Whatsapp, Angry Birds, RMaps, and the MIT Mobile App, which we either identified with the app Permission Friendly as requesting particularly many permissions. The permissions we focused on were the INTERNET, ACCESS_FINE_LOCATION, READ_PHONE_STATE and WRITE_EXTERNAL_STORAGE.

The Android permission system

The permissions an application requests are declared in the manifest file of the package (.apk). For example, the line that requests the INTERNET permission: < uses-permission android:name="android.permission.INTERNET" >< /uses-permission>

There is also the possibility for packages to declare their own permissions, through which it can restrict access to its broadcasts or require permissions for intents.

When a user downloads an application, he is presented a list of permissions that the application may need; the user is not allowed to pick and choose which ones to grant or deny. This makes application development easier as there is no need to check at runtime which permissions are available. It also guarantees to application vendors that users cannot cherry-pick by, for example, using the application but denying it internet access in order to avoid advertisement (which is usually loaded from a server).

A key paper that provided great insight for our project is Stowaway: A static analysis tool and permission map for identifying permission use in Android applications [3]. The permission map (available at android-permissions.org) published with the paper is the most detailed documentation of Android permission enforcement that we could find.

The permission map was particularly useful to our project because it told us which permissions each API call requires. There are over 1000 API calls that require permissions and it would have been nearly impossible for us to find them in the little time we had.

Issues with the current system

Neither developers nor users are usually very familiar with the implications of requesting or granting specific permissions. A recent paper on the Android permission system identified that many apps are over-privileged (i.e. they request more permissions than they actually used). It appears that developers request certain permissions in the manifest file "just in case". A very common misconception that leads to over-privileged apps is that sending intents to the system calendar, phone or contact application requires the permissions READ/WRITE_CALENDAR, CALL_PHONE, READ/WRITE_CONTACTS. This is not the case, since the user is still required to interact with the system application to complete the action.

Preventing over-privilege is important. Extra permissions may (1) unnecessarily deter users from installing applications, (2) unnecessarily accustom users to accepting lots of permissions, and (3) needlessly increase the potential damage of application vulnerabilities.

Of course, the problems are not only on the side of developers, but also with the users. In fact, many users do either not check or not understand the permissions an app asks for before installing.

It even seems that the users and developers are not the only ones who have trouble understanding the permission system. According to the paper by Felt et al [3], there are a number of permissions which can be declared in the manifest but are never checked in any API call, such as android.permission.BRICK, which according to the specification is "required to be able to disable the device (very dangerous!).

Approach

Our approach for removing permissions from Android apps is very straightforward if the details are left aside. In its basic form, our plan is as follows:

1. Unzip apk package
2. Remove permissions from AndroidManifest.xml
3. Modify application code to make sure it doesn't crash because of permission issues
4. Zip modified apk package
5. Run on phone

Implementation

Of course, practise is never as simple as theory and there are a couple of obstacles along the way, the first of which is to obtain the application packages. It is not possible to download applications from Google play without having an Android phone. If browsing on the computer, Google will ask the user to log in with his Gmail account tied to the phone when trying to download an app. After logging in, it will propose to install the app to the user's phone next time it is connected to the internet. If the user agrees, he will not be asked again and the application will silently be installed on his phone (there is a small notification after the installation). Apart from the fact that this keeps developers from downloading apk files to their computer directly, we also consider this a security issue as anyone who obtains another person's Google credentials can install applications to that person's phone. Not to mention what could go wrong if Google were compromised...

Copying apk packages from the phone to the computer is also not possible directly without root access. In the end, we found two ways of getting apk packages to our computer:

1. Use the chrome extension APK downloader (requires you to input your Google credentials and the device ID of your phone, maybe a little bit of a security risk)
2. Use a file manager (like Astro) to back up the applications (which makes a copy of the apk on the SD card), then transfer the packages to the computer

Another issue one can run into when reverse-engineering Android applications is that after being modified, the apk files have to be signed before they can be installed on a device. Google's official signing tools require a certificate signed by a CA, but there are free tools out there that work with self-signed certificates generated with openSSL.

Simply removing a permission from the AndroidManifest only worked reliably with two of the permissions we wanted to investigate. Neither removing INTERNET nor WRITE_EXTERNAL_STORAGE. We assume that this is the case because developers do not expect phones to always have internet access or always have an SD card inserted.

With all other permissions, removing them from the manifest is hit and miss. Some applications do fine while others crash immediately or freeze.

Therefore, it was not enough to simply remove permissions from the manifest, we also needed to modify the source. We found *apktool* to be a convenient tool for decompiling the bytecode in apk files. After some initial difficulties with missing frameworks, we got it to reliably decompile and recompile our applications. Apktool does not decompile the bytecode to java, but to smali, which is an assembly language for the dex format used by Android's Dalvik (the VM that runs the apps).

Smali is not exactly a human-friendly language to modify, especially not when dealing with such a large codebase. It is nearly impossible to tell which lines of code are causing the problems and debugging smali is rather impractical. There are not many people out there who are decompiling android apps, so the documentation or support for this kind of endeavour is sparse or inexistent. Decompiling the dex code to java is also not an option. While this leads to readable code, recompiling is much more difficult and requires many manual fixes, which is not only impractical for large applications such as the ones we modified, but it also makes it nearly impossible to use an automated tool in the future.

Given the sparse documentation about smali, we decided to write a few dummy applications, which we then compiled and decompiled to see what was changing. From this, we got the idea that all we needed to do is implement a class which extends the class whose function requires the permission we are removing and then substitute the call to the original class with a call to our class, which instead of interfacing with the API would do nothing and return a default value. We did this for the Apache HttpClient and it seemed to work quite well, with the only issue being that the method we were trying to override was final, so we had to rename our method and make sure that we rewrite the smali code to call that new method.

While this approach works fine for the functions of the HttpClient, it does not work for functions of other classes, such as TelephonyManager and LocationManager, because those classes can not be extended (i.e. they are either declared final or have a private constructor). While it is possible to program a class that it identical to TelephonyManager, it is not a solution either, because an instance of TelephonyManager cannot be cast to an instance of the new class. It is possible to replace the cast statements with the creation of an instance of the new class, but it is rather impractical, because changing a cast to a constructor will require changes over multiple

smali instructions and registers, which in turn may require a ripple of changes through the rest of the code. We consider this rather impractical.

An approach which we found to work quite well is to leave the cast statements alone (we discovered that they do not require any permissions) and simply replace the critical function calls with calls to a static function of an "impostor" class. Because the function is static, it does not require the class to be instantiated, thus every change will only affect one line of smali code. By making static functions that do not take any arguments but return a valid dummy value, all changes can be limited to one smali instruction. We call the class that provides the static functions StaticFakeEverythingManager because it can be used to replace the TelephonyManager, LocationManager, RequestManager, ConnectionManager etc.

Our success and especially our final implementation relies heavily on the permission map by Felt et al [3]. Without it, we would have been unable to identify the API calls that require certain permissions.

In its final form, our implementation is as follows:

1. pull the apk file from the device with the help of a file manager
2. uninstall the package from the device using *adb uninstall*
3. decompile apk with *apktool*
4. remove permissions from *AndroidManifest.xml*
5. parse smali code to identify API calls that require the removed permissions
6. copy *StaticFakeEverythingManager.smali* into the smali source of the app
7. replace all critical API calls with calls to a static function of *StaticFakeEverythingManager*
8. re-parse smali code to ensure that all critical API calls were replaced
9. recompile into apk
10. sign the apk
11. install to device using *adb install*

We implemented this process with a number of shell scripts and a python script that calls the individual shell scripts. This modular approach proved to be very useful, because it allows manual intervention at any step of the process. The python script can be invoked as follows: *python appsafe.py AppName PERMISSION_TO_REMOVE_1 PERMISSION_TO_REMOVE_2 ...* (it requires the directories to be set up correctly as well as some external applications (such as *apktool* to be in the PATH).

While we wrote the code in StaticFakeEverythingManager by hand, we believe that it could easily be automated (or semi-automated) by parsing the permission map and Android's API definition and generating Java code based on that. The shell scripts that replaces the API calls in the smali code could easily be generated dynamically in python. It should be noted however, that our script only fixes permissions on API calls and does not affect content providers, intents or broadcasts. While this would be necessary for a complete permission removal tool, we had neither the time nor the manpower to do this.

Results

We tested our tool as well as the manual removal of permissions on several applications, the results are documented here.

Angry Birds

ACCESS_NETWORK_STATE
The application crashes (*quit unexpectedly*). An analysis of the code shows that the game does not need the permission, but third-party advertisement software that ships with AngryBirds uses it. We are unsure what the ad software uses this information for.

ACCESS_WIFI_STATE
Works fine without. In fact, there is no API call in the code of AngryBirds that needs this permission. This is a case of over-privilege.

ACCESS_PHONE_STATE
The application crashes. This permission is required by third-party advertisement software. We managed to remove it by hand and play AngryBirds without having to worry about evil advertisers stealing our phone number and device id (which is exactly what they could do with that permission, which shows that the name of the permission was badly chosen).

WRITE_EXTERNAL_STORAGE
The game runs fine without this permission and we could not detect any loss of functionality. Even saving games works.

BILLING
Works fine without.

MIT Mobile

The MIT Mobile app has many permissions for which it is not immediately clear what they are used for. It has so many permissions that the free app *App permission watch* classifies it as potentially dangerous along with Skype and WhatsApp.

CAMERA
The camera is used for the QR reader that comes with the app. If it is removed, the rest of the app works fine, but the QR reader reports that the camera may be broken.

INTERNET
Works fine without, but since the MIT Mobile app is essentially an app that pulls together information from the internet, the functionality is severely reduced. In fact, not event the emergency numbers can be accessed when offline.

CALL_PHONE
Works fine without. We believe that the app doesn't actually use the permission and that it was a misconception of the developers that sending an intent to the normal calling service on Android requires the permission.

READ_CALENDAR and WRITE_CALENDAR
Like CALL_PHONE, the app does not use these permissions.

READ_CONTACTS
The app does not seem to use this permission either.

ACCESS_FINE_LOCATION
Everything works fine without this permission. The map still loads but no longer shows the user's position.

RECEIVE_BOOT_COMPLETE
Works without. We do not know why the MIT Mobile app wants to receive the boot complete broadcast.

WRITE_EXTERNAL_STORAGE
The application works without this permission. While we did not find a feature that used it, it is possible that the app uses it somewhere.

RMaps

RMaps is a map application that allows the user to navigate using offline maps. Maps can be downloaded from various sources.

INTERNET
: The application works just fine without this permission.

ACCESS_FINE_LOCATION
: The application crashes unless the code is fixed. After fixing the code with our automated tool, it works normally but of course thinks that the user is a few hundred miles off the coast of western Africa ($0\,\mathrm{deg}$ latitude, $0\,\mathrm{deg}$ longitude).

ACCESS_COARSE_LOCATION
: We could not find any API call that required this permission and would not accept ACCESS_FINE_LOCATION instead.

WhatsApp

Of all the applications we tried to modify, WhatsApp has by far the longest list of permissions. Unfortunately we were unable to test the application with removed permissions, because it requires a server login before any functionality can be used. The login happens with the phone number and device ID, both of which are not valid on an emulator. We did not try it on the Samsung Galaxy because that would have made the app on the phone unusable for the time of the testing.

Conclusion

The current Android permission system has several weaknesses. It does not allow users to selectively grant or deny permissions to applications and it is hard for developers to use correctly. In this project we looked at a number of Android applications and tested whether permissions could be removed without impairing too much of the application functionality. We found that almost all applications request permissions they do not actually use. The MIT Mobile app was an especially bad example for this.

We also found that while certain permissions, such as INTERNET and WRITE_EXTERNAL_STORAGE can be removed from the manifest without leading to the crash of an application, this is not possible for other permissions such as READ_PHONE_STATE or ACCESS_FINE_LOCATION. For those other permissions, we successfully wrote a collection of python and shell scripts as a proof-of-concept to show that we can fix the broken applications with an automated script. Based on our collection of scripts it should be fairly straightforward to fix any API calls in Android applications that cause it to crash when permissions are missing.

Project experience

Overall, we both greatly enjoyed this project and have learned a lot in the process. Neither of us had any previous Android programming experience, so there was a certain learning curve, but once we got the hang of it, we think we managed to do some cool stuff. We overcame a number of obstacles on the way to our proof-of-concept implementation and we are very happy that we managed to succeed in using tools and languages that have very little available in terms of documentation or support.

We also realized that it was impossible for the two of us to achieve all we would have liked to achieve in the little time available for the project.

References

[1] Privacy blocker. https://play.google.com/store/apps/details?id=com.xeudoxus.privacy.blocker&hl=en.

[2] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security Privacy*, 7(1):50 –57, February 2009.

[3] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, page 627638, New York, NY, USA, 2011. ACM.