CS161–Spring 2012 — Project Report

Project: Fine-grained permission control engine on Android

Team Noble 6 :
Edward Lu, Ashwin Kamath, Victor Chang, Jason Jiang, Josef John, Nathan Nandi

May 8, 2012

1 Introduction / Background

Our objective for our final CS 161 project was to create a proof of concept for a fine-grained permission control engine for the Android system. This engine allows users to define and manage their own access control policies for different applications dynamically and with specificity not provided in the current Android permission implementation. We use the term proof of concept to imply that we mean to create a fully functional engine, but have it implement only policies of interest in order to prove it can be done. Having implemented a complete list of policies spanning every possible permission is not our objective.

The current Android system application permission system is too coarse-grained to provide users with enough security. Users today must accept all permissions proposed by the application at installation time if they want to use said application. Not only are these permissions coarse in scope, but this all-or-nothing approach forces users into potentially unsafe situations if the will to use the application is strong enough. Thus we want to be able to allow users to specify finer-grained permissions, even after installation.

Our solution was to provide an application-level interface (referred to here on out as "Barley"), for users to select any application currently installed. Once selected, the application's current permissions are displayed and the user may then modify these permissions. The user can also specify more specific internet permissions through regular expressions. Barley stores this information in a SQLite database, which is interpreted and enforced by our modified version of the Android operating system at a series of Policy Enforcement Points (PEP). [1]

[1] The term "Policy Enforcement Points" is borrowed from a similar project proposal as described in "Taming Information-Stealing Smartphone Applications on Android", by Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh of the Department of Computer Science at NC State University (Reference 1).

2 The Team

The team was divided into two groups. Every week, these groups would switch off and do some work on the project in attempt to get exposed to all aspects of the development. This worked out very well because smaller groups are easier to manage. Weekly meetings and emails served as mediums through which the groups communicated their progress, questions, and ideas. The slow, logistically difficult task of handling a 6 person group was avoided and our project was finished efficiently.

3 The Architecture

To begin, we had to decide on an architecture. We wanted an engine that was easily configurable by users and one that involved the least amount of intrusion into the operating system as possible. Our final decision is shown in Figure 1 and will be elaborated upon for the rest of the report.

4 The Application

The first task for our project was to create the user interface, Barley. We created a GridView to view all the apps installed on the phone, in a similar fashion to the default Launcher2 application. We hooked up an event to each app icon in the GridView, which, when clicked, would send an intent to a tab view where a user can modify the selected app's policies. The tabs were organized into policy categories such as "Internet", "SMS", "Phone Info", "Regexes". Each tab had a list of checkboxes, along with the policy that checking the box would enable. The "Regexes" tab contained input boxes for a user to input any regular expression in. The input boxes we provided were for an Internet blacklist and an Internet whitelist. The idea was that the user could input a regular expression to match URLs into these boxes. If the user put it in the blacklist, all matching URLs would be blocked from the selected app's access. If the user put it in the whitelist, only matching URLs would be allowed to be requested from the selected app.

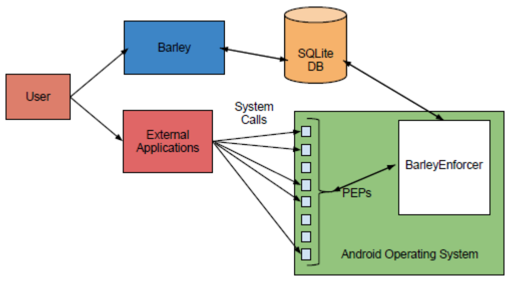

Figure 1: Architecture Diagram

5 The Database

For the application to have any use, it must have persistent storage. Thus we decided upon having the application store the user's policy choices in a SQLite database. This decision was made because SQLite has a simple interface and because the Android SDK has built in support for SQLite databases. The schema of the database is shown in Figure 2.

As shown in the figure, we identify applications using package names. These are readily accessible through Context and PackageManager APIs (to name just a few), so we believed this would be the best way to identify applications without fear of collision. Whenever an application is selected in Barley's GridView, a check is run to see if the its package name exists in the saved_apps table. If not, it will populate it accordingly. The user's policy choices are remembered in the apps_policies table and the user's regular expressions are saved in the saved_apps table. The all_policies table holds all the implemented policies and provides Barley with text to describe its policies for the user interface.

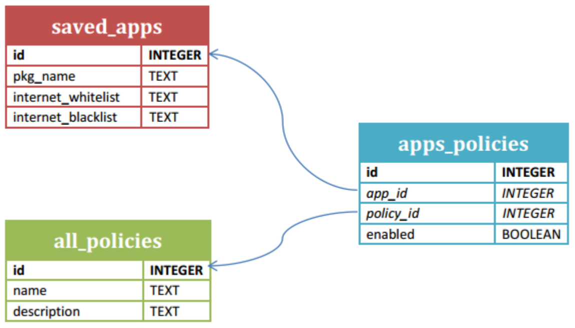

Figure 2: Barley's Database Schema

6 The Operating System

The final step in our project was to modify the Android operating system to enforce the policies outlined in Barley's database. To begin, we had to figure out where in operating system we need to inject code - the Policy Enforcement Points (PEPs) mentioned earlier in the introduction. These PEPs will be where the operating system accesses Barley's database to check to see if the application requesting to execute a system API is allowed to do so. We were greatly assisted in this process by work done by a team of UC Berkeley researchers, who successfully reverse engineered a complete list of system API calls and its required permissions into a permission map.[2] We treated this permission map as a map for PEPs, and began injecting code for a single PEP. To create an environment to test our project, we had to download the source code and inject our Barley application into it - making the operating system load the application as a built-in app. This was met with many small logisitical challenges (many external program dependencies were not

[2]The permission map was mapped out by Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song and David Wagner (Reference 2)

mentioned in the documentation and had to be researched and installed independently) and a few unexpected technical problems. For example, the Android operating system is stricter on string localization for built-in applications. Normal applications are allowed to have strings embedded within layout xmls, but they must be moved if it were to be installed as a built-in application.

Our first major challenge was met shortly after testing the first PEP implementation. The injected code needed to access Barley's database, but due to existing security measures used by Android, it is not a trivial matter to access an outside application's database. Since every application and the operating system is assigned its own linux user id, the linux kernel would not allow even the operating system to access Barley's database. We tried addressing this challenge by using the ContentProvider API - an API designed to share data across different applications. However, this was met with severe problems. First, the ContentProvider API required a Context to query its information but several PEPs do not have access to Contexts. Second, the ContentProvider API needed its requestor to register its intent to access the database in its AndroidManifest - but the operating system has no such thing. We realized this was a lost cause, so we tried a simplier approach: after every API call that modified the Barley database, chmod the database file so that it could be read globally. This worked splendidly and did not pose a serious security risk. Assuming outside applications could even guess at the location of the database, they could only read it. There is no way they could tamper with it.

We later tried to simplify the code by moving the policy-checking mechanism into Android's SDK. We created a class called "BarleyEnforcer" and put it in a new API folder. This greatly simplified the injected code's complexity. All we had to do now was `import android.barley.BarleyEnforcer;` and call `BarleyEnforcer.allowed(String pkg_name, String []permissions_needed)`.

Our second major challenge was caused because some PEPs would force close the application should it be denied privileges by Barley. This ungraceful exit was not desired by our GSI nor by our group, so we had to be careful in choosing PEPs that would not force close the application. We addressed the challenge simply by testing each and every PEP in a painstakingly long modify-build-modify-rebuild loop. Also, to make it even more graceful, we added Toast messages for PEPs that contained Contexts. Users would receive a friendly and unobtrusive (the toast never takes focus away from the Activity) message notifying them that Barley has prevented an unauthorized system call.

Our final major challenge was that some PEPs did not have access to the requesting application's package name, thus rendering checking of policies impossible. If we cannot figure out who requested the application, how could we identify what policies the user specified for the application? We managed to address this problem in a few PEPs. Some lucky ones had constructors with the Context of the requestor but simply did not save it in an instance variable, so we added it to one and used it. Others involved ContentResolvers, whose API we modified to implement a `.getContext()` method. Every other PEP could not be implemented and were ignored for the project. They symbolize the main limitation of our solution - if we cannot find a way to retrieve information about the requesting application, there is no way we can implement a related policy for it.

To conclude our project, we created a malicious application that would send SMS texts without the user's consent. This app, named Barlene, was used to demonstrate how our engine could easily let the user modify its policies after installation so that he/she could use the app without worrying about it texting anyone.

7 Conclusion

Our team worked with the Android SDK to create an application, hooked the application to a SQLite database, and modified the operating system to provide a fine-grain permission control application for the user. With our ”Barley” application, Android users have the option of denying permissions for certain apps after installation and specifying finer policies with the use of regular expressions. We have implemented several demonstrations as a proof of concept, including restricting Internet usage (with finer-grained control by supporting URL regular expressions) and blocking SMS sending privileges through a test app ”Barlene” that we wrote.

In conclusion, our project was a success. We accomplished all the goals that we set out to achieve, which was mainly to provide users with an additional layer of permission control security. In the process, we learned a lot about how the Android operating system worked. Our solution exceeded our expectations, but it is true that we found limitations to our approach. Because certain PEPs could not provide the requesting application's package name, we cannot implement policies for it. Thus suggestions for future work would include stack inspection, which may allow the programmer to discover the application's identity without having to resort to using an API to find its package name.

8 References

1. *Taming Information-Stealing Smartphone Applications (on Android)* by Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Department of Computer Science, NC State University.

2. *Android Permissions Demystified* by Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, David Wagner. University of California, Berkeley. 2011.