Making Marshmallow's Permissions Sweet Again

Ajay Mohan	Heli Modi	Kandarp K.	Shreeja K.	Yu Shen
ajmohan	hmodi	kkhandwala	shk149	yus067

ABSTRACT

Android 6 allows the user to place an app either in a position of absolute distrust or one with a high level of trust. We show how this actually does not offer the flexibility to allow certain actions while preventing undesirable ones. This paper aims to address the problem by bringing out potential misuse of permissions to the user's attention. To do so, we come up with a new way to assign and declare permissions that can minimize interactions between two sensitive resources which we argue to be what really needs to be monitored and controlled. We also perform static code analysis of seven open-source applications to measure the impact of the proposed model both in terms of usability and from the perspective of developers.

1. INTRODUCTION

The mechanism by which sensitive resources are protected in Android is the use of *permissions*. Only resources that have been authorized for use, by means of the corresponding permission(s) being granted, may be accessed by apps. Android started out with an "all or nothing" approach where users did not have control over individual permissions being granted to apps: they could be installed only with all requested permissions given away, not otherwise.

With the introduction of Android 6 *Marshmallow* last year, app permissions became granular and more user-friendly at the same time. At the first instance when an app would request any so-called "dangerous" permission, a prompt would be presented to the user to choose whether to allow or deny access to the resource, following which the choice is persistent. Also, users could revoke any permission that had been granted earlier, thus making permissions more transparent than previous iterations of Android.

However, the permission model has drawbacks. While it does well at restricting blanket access to specific resources, it does not control how the app uses those resources. Another issue is that some permissions touted as "normal" permissions are granted to apps without informing the user (note that these cannot be revoked), which includes access to the Internet. This imparts a false sense of security to the average user and allows attacks such as information leak to go through, by use of sensitive resources in ways the user did not wish to allow.

For example, a third-party keyboard app may access the Internet to update definitions in the dictionary used and also request access to external storage to store/backup preferences. These are legitimate enough use cases to convince most people to part with the corresponding permissions. However, a malicious app could misuse these permissions, say, by siphoning off private data in the external storage to a remote server, without the user realizing it. It is instructive to observe that an equivalent, but harmless app is indistinguishable for the user in terms of the number and specific permission requests the current security model mandates.

This paper attempts to address such behavior by making it harder to misuse different permissions. To do so, in the following sections, we present a new model that helps minimize *interactions* between two sensitive resources and allows the user to decide which ones to allow or not. In doing so, we try to deviate as little as possible from the user experience in Android Marshmallow to avoid any unnecessary confusion, while being able to provide a higher level of security.

2. THREAT MODEL

Threat modeling is a procedure to identify vulnerabilities that will assist in defining countermeasures to prevent threats to the system. In case of Android applications, the threat model is as follows. The attacker is a developer who makes an Android application ("app") which needs resources that are guarded by the corresponding permissions. The capabilities of the attacker are to get users to willingly install a seemingly innocuous app and having the app able to access resources protected by some (normal) permissions by default and additional (dangerous) permissions granted by the user on request for a purportedly legitimate purpose. The victim is a user of such a malicious app, who is unaware about the misuse of the permissions that may happen.

The point of contention is in the way the current and proposed permission models attempt to protect the sensitive resources. We argue that users most often wish to control how apps use different resources, not the resources by themselves. As an example, it may be okay for an app to access photos on the user's device, but not to upload them without authorization. This is something that is not in the scope of the current permission model, but at the center of our proposal in this paper.

3. PROPOSED PERMISSION MODEL

In essence, the way we envision permissions should be enforced is as follows. Android apps are composed of different components (activities, services, background receivers and content providers). There may be multiple instances of each of these with *different* sets of permissions limiting them. The user is prompted when either a single component with one or more conflicting permissions is run, or when two components with permissions that can cause harm interact with one another.

Our design (see Figure 1) has four major aspects. The secret sauce is the use of suspicious combinations of permissions (section 3.1). A new app manifest structure is defined next.

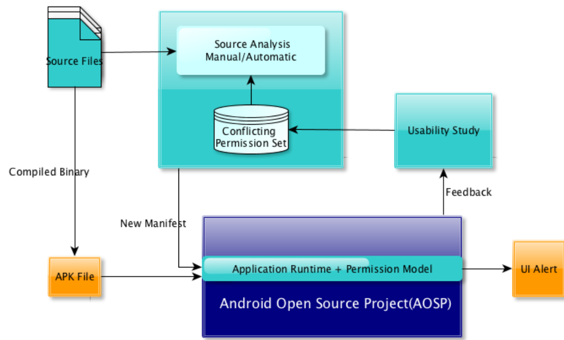

Figure 1. Architecture of our Permission Model

Section 3.3 elaborates on static code analysis for identifying boundary points (where components interact). Finally, we discuss how to evoke an appropriate popup for requesting permissions (section 3.4).

3.1 Suspicious Combinations of Permissions

Android provides a relatively high level of flexibility, by allowing developers to access fine-grained permissions such as `READ_CALENDAR` and `WRITE_CALENDAR`. In the interest of a better user experience, however, the dangerous permissions are "lumped" together into permission groups, which cater to more abstract behavior. There are nine such groups in Android Marshmallow, as described below.

1. Calendar (Managing calendars)
2. Camera (Taking photos and recording videos)
3. Contacts (Managing contacts)
4. Location (Current device location)
5. Microphone (Audio recording)
6. Phone (Dialing and managing phone calls)
7. Body Sensors (Heart rate and similar data)
8. SMS (Sending and viewing messages)
9. Storage (Accessing photos, media, and files)

For our model, we note *permissions of interest*, which include the nine permission groups above and additional groups formulated from the normal permission set, consisting of permissions that do entail security concerns — Internet, NFC, BT, Network and WiFi. In fact, `INTERNET` was a dangerous permission in previous iterations of Android. It is interesting to note that all the new permission groups identified are communication-related. As argued earlier, a permission in itself usually poses no harm and may simply be granted. There are certain exceptions, such as making phone calls (`CALL_PHONE`), sending text messages (`SEND_SMS`) or using NFC, since use of any of these can cost the user money or directly affect privacy. As an additional level of security, we can allow users to revoke (or not automatically grant) some individual permissions that he/she values.

Next, we empirically identify potentially malicious pairs of permissions and find them to be classifiable either as resulting in information leaks or context leaks.

Information leaks:
`INTERNET` with any other permission of interest except NFC
`MICROPHONE` with `PHONE` (ex: recording conversations)

Context leaks:
`CAMERA` with `CALENDAR` or `LOCATION` (ex: geotagging)
`LOCATION` with `BODY SENSORS` or `MICROPHONE`
(ex: augmented location)

3.2 Proposed Structure of App Manifest

Every app has an `AndroidManifest.xml` file which declares the various components (activities, services, broadcast receivers, and content providers) of the application and the "global" permission set. Figure 2 shows the usual structure of the app manifest in Android Marshmallow.

An *activity* is a component that provides a visual interface with which users can interact in order to do something, such as dial a number, take a photo, send an email or view a map. An app usually consists of multiple activities that are loosely bound to each other. The `<activity>` tag is used to declare an activity. A *service* is a component representing either an application's desire to perform a longer-running operation while not interacting with the user or to supply functionality for other applications to use. Each service has a corresponding `<service>` declaration. An *intent* is a messaging object used to request an action from another app component. The `<intent-filter>` tag describes the capabilities of its parent component, i.e. what an activity or service can do.

The manifest file also lists the permissions the application expects to be granted in order to function fully. The `<uses-permission>` tag in the manifest is used to declare permissions that the application will need to access the various resources protected by those permissions.

```
<manifest>

    <uses-permission />
    <permission />

    <application>

        <activity android:name="com.example.project.XYZ"
                  android:permission="ABC">
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <service android:name="string"
                 android:permission="string">
            <intent-filter> . . . </intent-filter>
            <meta-data/>
        </service>

        <uses-library />

    </application>

</manifest>
```

Figure 2. App Manifest File Structure (in Marshmallow)

The various components are protected by the permissions listed in the `<uses-permission>` tag or

<permission> tag (used to declare custom permissions), using the android:permission attribute. This attribute, which may be present in activity and service tags, lists any permissions that *clients* must have to launch the particular component (this is its primary purpose). For example, if a caller of startActivity() has not been granted the specified permission, its intent will not be delivered to the activity. If this attribute is not set, the component is not protected by any permissions, other than those declared for the app as a whole.

Note that we described how permissions are defined in Android Marshmallow above. We propose that the android:permission attribute must be present within every component and it is what should be used for declaring a permission, without the need to declare the same permission "globally" using the <uses:permission/> tag. Note that multiple permissions could be associated with a single activity (or component). Thus, in our model, there is no use of the <uses:permission/> tag.

3.3 Boundaries between Components

Since components may have conflicting permissions, any sharing between them must be controlled, otherwise it renders the enforcement of permission combinations meaningless.

We performed static code analysis to accomplish two goals:
(a) to spot transition points between activities and
(b) to note the current coding standards and methodology.

We analyzed seven open-source Android apps that were picked randomly across various categories: AdBlock Plus, Barcode Scanner, Dolphin Browser, Telegram, VLC, WordPress and Wikipedia.

The tool used for this was lint, which is a tool that is bundled with most Android (app development) IDEs. A linter is generically a tool that flags incorrect language usage in software. It comes with the ability to define a custom set of rules and patterns which will be looked for during the analysis. Upon finishing the analysis, it generates a HTML document which lists all the findings from the source code based on the rules provided beforehand (and some of its own general rules for missing API requirements).

To perform the analysis on the apps, we created our own lint rules, which are as follows:

- Look for the methods checkSelfPermission() and requestPermissions(). These methods are mandatory to be used for requesting a dangerous permission.
- Look for the method intent(). This method is the fundamental method used to navigate to another activity within the application as well as to invoke any system application for handling a task.
- Look for the pattern "*.class*". Activities are essentially java files which can use all the public methods and variables of another activities by creating the object of that activity.
- Monitor the usage of *global variables*, declared in the Application.class file, which can be accessed across the components of the application.

After performing static code analysis, we had the following information for each activity: *Permission set*, *Activities transitioned to* and *Point of transition*. Thus, from the static code analysis, we identified the exact points of transition which should be monitored by the Android operating system to generate a popup that prompts the user for permission to make use of conflicting permissions.

3.4 New Permission Request Popup

Figure 3 illustrates how an implicit intent is delivered through the system to start another activity in Android Marshmallow.
[1] Activity A creates an Intent with an action description and passes it to startActivity().
[2] The Android System searches all apps for an intent filter that matches the intent.
[3] When a match is found, the system starts the matching activity (Activity B) by invoking its onCreate() method and passing it the Intent.

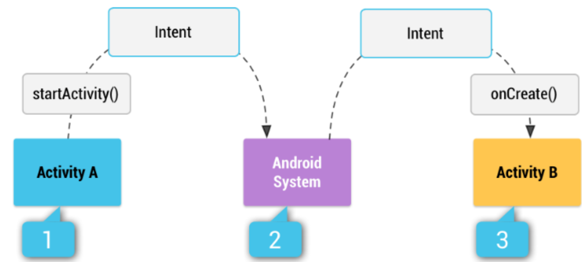

Figure 3. Flow of intent to start another activity

The system searches through the list of all intents using the startActivity(Intent) method in android.content.Context, which calls a native method in ActivityManagerNative.java that uses an Inter-Process Communication (IPC) mechanism to start the appropriate activity. Before the invocation of the next activity (or component), we must introduce our permission verification process, once the intent match is completed.

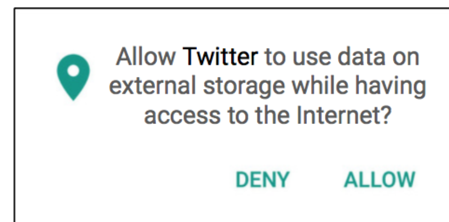

Figure 4. Permission Request Dialog

Based on the list of suspicious permission pairs and the newly structured manifest file, a popup is generated at any suspicious points of transition (or when the component uses multiple conflicting permissions), which will prompt the user (only for the first occurrence, after which the choice is persistent) for permission to go ahead or reject the permission request, as seen in Figure 4. Note that if a component remains isolated, i.e. does not interact with other components nor does it make use of multiple conflicting permissions, no prompts are required to be shown to the user.

3.5 Usability Study

This section enunciates the pros and cons of our model from the perspective of both users and developers. As it is not possible for us to use the proposed app manifest structure (because of incompatibility with existing IDEs), we used the findings from the previously discussed static code analysis.

For the usability study, we compared the number of permission request popups (for each app) in Marshmallow with our proposed model. This was done to measure the impact of our model on the user experience.

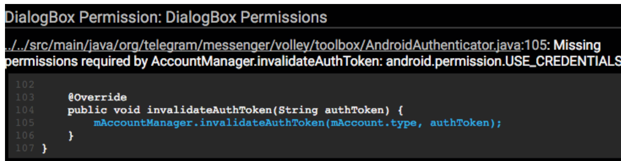

Figure 5. `lint` Output for Boundary Locations

Application	Reduction in Permission Requests
VLC	-33% (increase)
Dolphin Browser	0%
WordPress	0%
Telegram	20%
Barcode Scanner	25%
AdBlock Plus	33%
Wikipedia	100%

Table 1. Results of Usability Study

As seen in Table 1, by using our permission model, the number of permission request popups usually decreases. However, there is an increase of 33% in VLC. This is because the number of individual permissions requested in VLC is less than the number of permission combinations that are identified in our model. For such applications, refactoring the code to use fewer conflicting permissions in components will help.

From the developers' perspective, we tried to find how many changes a would developer have to implement to meet the new specifications. This is pretty straightforward as the only change a developer has to make (to ensure compatibility with our proposed model) is in the per-activity declaration of permissions in the app manifest. Also, developers can use our `lint` rules (stated in static code analysis) to identify places of improvement in the existing code. The developer can then try and modify the existing code to make use of fewer permissions within each activity.

4. RELATED WORK

Right from the release of the first version of Android OS in 2008, Android powered mobile devices have proliferated. The Google Play Store has also witnessed a large number applications that require a plethora of permissions from users in order to work as expected. Android application security has been a topic of great impact since then, although most research has been in sandboxing the execution environment of applications. This correlates to the traditional way of providing isolation and protection mechanisms by operating systems. From the application's perspective, research extended in directions that verified the sanity of applications. One such direction is verifying permissions that are specified in the app manifest. Every permission that is defined for Android has an associated API that it controls. The security subsystem will then verify if the permissions are present or not during the invocation of these APIs during runtime.

Orthogonal research like Taintdroid[3] track the tainting of critical user information from tainted source to tainted sinks. They use virtual taint map across various points in the execution flow of the system. We do not further discuss this line of research as our goal was to keep the execution overhead to the kernel module as minimal as possible and systems like Taintdroid suffer from performance bottlenecks.

PScout[2] is a tool that extracts permission specification from the Android OS using static analysis techniques. They extract this information from the existing permissions across various Android systems and establishing the fine grained results of their association. We base some of our work by reusing condition probabilities from PScout and also extend it by completely redefining the permission groups. To the best of our knowledge, we are the first to perform Android's individual component specific permission analysis. We do believe based on our results that this is an effective way in thinking about securing android applications in the future.

5. CONCLUSION

From the previous discussions, we now know that the permission model introduced with the latest iteration of Android did increase manageability of permissions, yet, it did not focus on the security aspect of how permissions are being used. We found that permissions can be used maliciously in conjunction, in ways the user did not wish to allow. Attacks such as information leaks are possible without the users knowledge (or interaction). Hence, to address this issue, we came up with a set of suspicious combinations of permissions, defined a new structure of the app manifest to declare permissions per-component rather than declaring them globally. Furthermore, we conducted static code analysis on seven open-source Android apps and identified the transition points between activities. We proposed amendments to the Android OS to monitor these transition points for generating OS-level permission request popups. Finally, we performed a usability study which showed that the number of permission requests a user faces usually decreases with our model.

REFERENCES

1. usenix.org/system/files/conference/usenixsecurity15/sec15-paper-wijesekera.pdf
2. eecg.toronto.edu/ lie/papers/PScout-CCS2012-web.pdf
3. usenix.org/legacy/event/osdi10/tech/full_papers/Enck.pdf