App Monitoring in Android through Instrumentation

Project Report

(Technical Report: TR-20150909-1)

Derrick Ahipo

Applied Mathematics and Computer Science

August 2015

Computer and Information Sciences

Indiana University South Bend

Advisor: Dr. Raman Adaikkalavan

Table of Contents

1 Introduction

Created for the mobile world, Android [1] is an open source operating system (OS) built on top of the Linux Kernel [2] and developed by Google Inc. Currently, Android is the most used and popular mobile OS and is pre-installed on devices like phones, tablets, computers, TVs and watches. Android has about 64% of the mobile OS market share worldwide as of June 2015 according to [3] and has about 82% of the smartphone OS market share during the period of 2015 Q2 according to [4] worldwide. With such a ubiquitous use of Android OS and the fact that it was mainly developed for portable devices and the adoption of these devices are ever growing, more and more researchers are using it to develop new technologies and developers are developing new apps that can run on it. According to [5], as of August 2015 there are more than 1.6 million apps on the Google Play market store where users can downloads apps officially.

Although Android is built on top of the Linux kernel, it uses a different security model and has several improvements over Linux security model as discussed in [6]. Some of the improvements include sandboxing of apps, use of user IDs, app permissions and app signing. Security is an important issue in the world of computing and even more with mobile devices because these devices contain private information of the user such as contact, location, etc. There is an extreme threat to privacy due to personal mobile devices and a warrant is needed to search a mobile by a law enforcement officer in United States [25]. One of the main ways in which privacy leaks can be addressed is by implementing proper access control mechanisms and avoiding privilege escalation attacks. Currently, Android supports coarse-grained access control [6] and also a security model that can lead to privilege escalation attacks. In this project, we study the main security features of Android and their effects on privacy. Sandboxing isolates apps from each other by keeping their data and the process in which they run separate. The permissions system follows the all or none model. If the user grants all the permissions required by an app, the app will be installed, and if the user does not want to grant one of those required permissions, the app will not be installed. Once the required permissions are granted, users cannot remove one or more permissions explicitly.

There is a large body of work in Android security [7, 8, 9, 10, 22]. SRT Appguard [9] and Cyanogenmod [10] are two projects that deal with issues related to coarse-grained permissions and others. Cyanogenmod allows users to revoke one or more granted permissions and still have the app running, but it can lead to unstable apps. SRT Appguard allows users to monitor what the app is doing with its permissions. SRT Appguard follows an app based approach while Cyanogenmod modifies the Android OS. The app approach makes it easier for people to install and use the app but it has limitations as only user installed apps can be monitored. Android source code modification gives more freedom and access to all resources which need to be monitored. With Android source code modification, all apps including the system apps can be monitored.

In Android, privacy leaks happen through many ways. The goal of our project is to build the necessary foundation to understand what Android apps do with their user granted permissions, which can then be used to prevent privacy leaks. Our approach is based on Android OS source code modification where we instrument the source code and build a custom ROM. Below, we discuss the main ways in which privacy leaks may happen in Android.

1.1 Main Issues

1.1.1 Misuse of User Granted Permissions

Currently, Android access control has 152 system-defined permissions to allow apps to access resources [11]. In addition, new permissions can be created, but is not common. During app installation, user grants the permissions required by the app. System-defined permissions are not fine-grained so the user does not know what the app can actually do with these permissions. For example, a maligned messaging app may request the user to grant permission to access all the contacts in the user’s device so that it can add some information about all the contacts to the app's database and can let the user connect with his/her contacts. Though the app needs only phone numbers and name, due to the coarseness of the Android app permission model, the user provides access to the entire contact database, which may include personal information like addresses, birthdays, anniversaries, etc… and information like personal notes, information about external accounts (e.g., Twitter, Face Book), work information, email accounts, etc. Based on the nature of the app the user may think the app is going to use only the name and phone number, but the malicious app can read other information from the contact database, without the knowledge of the user leading to privacy leaks. In addition, after installing the app, the user will not even know if all the granted permissions are really used nor what information is being accessed with the permissions associated. As another example, permissions that grant access to internet and location are really important and need to be treated carefully. These two permissions when granted to an app can be used to gather information about the user such as where the user usually eats, goes to school or work, banks what website, etc. and can be transmitted via the internet permission which can also lead to privacy leaks.

1.1.2 Privilege Escalation Attacks

Apps can share the user granted permissions using a shared user ID. In other words, apps developed by a person or organization or company can share permissions between different apps when they use the same app signature. Since apps can share their user granted permissions it can lead to permission accumulation and privilege escalation attacks. Assume a user installed two different malicious apps from a particular developer: App A that has internet permission and App B with the permission to access the contacts. In this case, App B can access all the contact information and share it with App A, which can then send the information using the internet permission. On the other hand, user may not even know that this is possible.

Apps can also share data using content providers. Information can be shared among Apps using the content provider even if they have been developed by different developers. As discussed in [12] "All applications can read from or write to your provider, even if the underlying data is private, because by default your provider does not have permissions set. To change this, set permissions for your provider in your manifest file, using attributes or child elements of the <provider> element. You can set permissions that apply to the entire provider, or to certain tables, or even to certain records, or all three." For instance, App A could share information with app B using content provider. App A can create a content provider and save all data there and app B can retrieve data by accessing that content provider and this can be really dangerous when malicious apps collaborate.

1.2 Our Contributions

To address misuse of user granted permissions and privilege escalation attacks we need to understand what apps do with their permissions and whether they are collaborating. In this project, we have created an instrumentation based framework to capture and generate interesting events. Specifically, we list and briefly describe our contributions below.

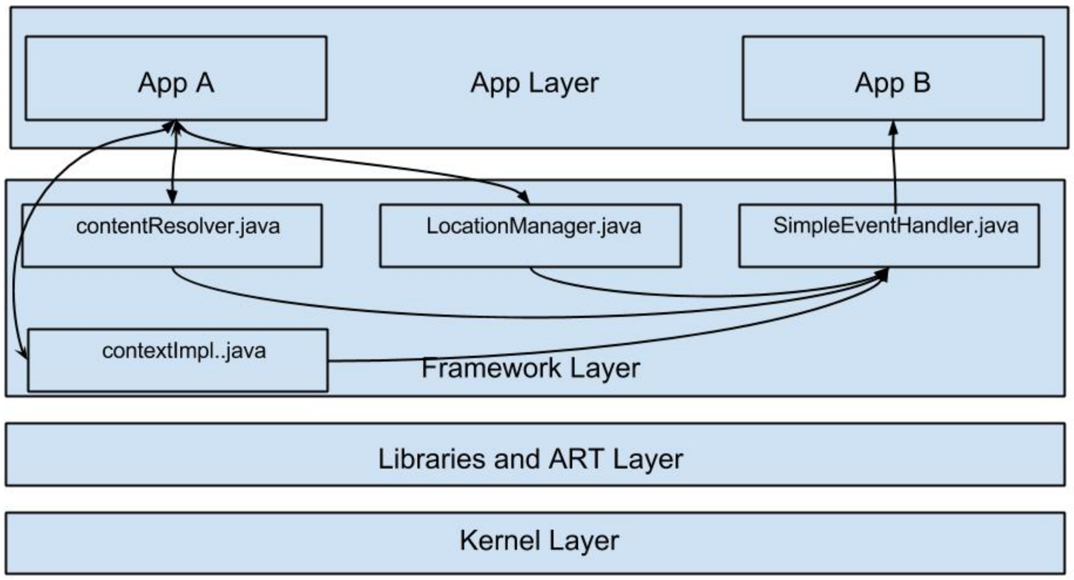

Figure 1: Contribution Architecture

Our main contributions are (shown in Figure 1):

- Created Android Custom ROM with instrumentation to address privacy leaks that can happen through components such as content provider, network activity and location.
- Create the new class "SimpleEventHandler.java" to handle events. The events are

formatted using XML, and generated via instrumentation. Events are broadcasted using the method “sendBroadcast()”. An event receiver app receives broadcast from the Custom ROM containing the XML message.

- Testing of content provider instrumentation using apps. This test is done using a contacts app which uses a content provider where the contacts are stored (also called contact provider). It confirms that any use of content providers will be monitored.
- Testing of location and network activity instrumentation using apps. This test is done using navigation apps. This shows that any call to a get the location or access network will be recorded.

1.3 Report Organization

This report is organized as follows. In Section 2, we explain Android security architecture in more detail. In Section 3, we discuss approaches that were explored and why we chose the custom ROM approach and how we captured interesting events via our instrumentation. In Section 4, we explain the events generated for content provider and location and how we tested them. In the Section 5, we discuss related work. In Section 6, we summarize our project and discuss future work.

2 Android Security Architecture

This section provides detailed description of the Android architecture, security model, and inter-process communication. In the Android architecture subsection we discuss different layers of Android system such as programming language used and their functionalities. In the Android security model subsection we describe different security features offered by Android OS. In the last subsection we discuss Android's inter-process communication methods.

2.1 Android Architecture

Android architecture is shown in Figure 2 and is made of four main layers that are (bottom to top): Linux kernel (with Android enhancements), libraries and Android Runtime (ART), frameworks, and apps. The bottom two layers are written in C/C++ and two top layers are written in Java.

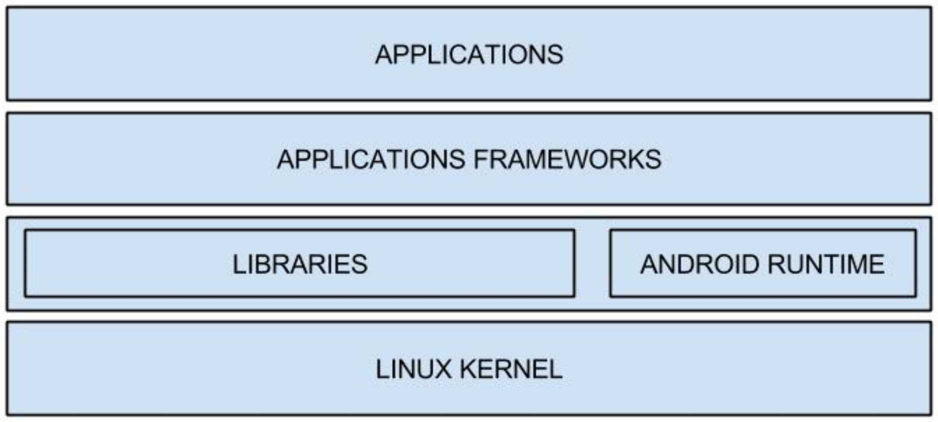

Figure 2: Android Architecture

The first layer is the Linux kernel with Android enhancements. Android OS is built on top of the Linux kernel as Android was primarily made for mobile devices where managing power and storage consumption and user privacy are critical. New features were added to the Linux kernel to optimize it [6]. As specified in [13], Android kernel inherits the following security features from the Linux kernel: "a user-based permissions model, process isolation, extensible mechanism for secure IPC, the ability to remove unnecessary and potentially insecure parts of the kernel, prevents user A from reading user B's files, ensures that user A does not exhaust user B's memory, ensures that user A does not exhaust user B's CPU resources, ensures that user A does not exhaust user B's devices (e.g. telephony, GPS, Bluetooth)" The optimizations carried out for the mobile devices bring new features to the Linux kernel. These new features include: enabling the kernel to kill apps in case of low memory availability, allow the mobile device to stay on or to turn off after a certain period of time (WakeLock), have the ability to share memory, and communicate between processes via Binders.

The second layer contains a set of libraries and the Android Runtime (ART) [14]. There are more than hundred libraries written in C/C++, and several daemons [15] to take care of the service requests and necessary boot files. The ART executes apps and was first introduced in Android 4.4 and it replaced the Dalvik virtual machine. The main difference between ART and Dalvik virtual machine is that former uses Ahead-Of-Time (AOT) compiler and the latter uses the Just-In-Time (JIT) compiler. The AOT compiles the app at install time and it is only done once whereas the JIT compiles the app every time the app is started.

The third layer holds the application framework that has several Java libraries, which contain the necessary classes to create apps and to access and communicate with different system services.

The last and topmost layer is the app layer. This is where all apps are installed on the device. There are two types of apps: pre-installed apps and user installed apps. Pre-installed apps are usually installed on the mobile device by the manufacturers or vendors. They are located in the `/system` folder and cannot be removed by the user. The user installed apps are the apps installed by the user. They are located in the `/data` folder and can be uninstalled any time by the user. For example, the native or stock contacts app installed in a new phone is a pre-installed app and is provided with the OS. On the other hand, users can download other apps from the Google Play store that access contacts and install it.

Both user apps and system apps could be created with one or combination of the following elements: activity, service, content provider, or broadcast receiver [6]. An *activity* is a graphical user interface (GUI) page that can be used for user interactions and carrying out operations. An app must have at least one activity, it is the only component required when creating an app. A *service* is like an activity but runs in the background, does not have a GUI, and is not required to be part of any app. It is usually used to run operations that are long running or could slow down the user interface making the device unresponsive. For example in a calculator app, an activity (or GUI) can send some data to a service that will execute the operation in the background and return it to the activity. Thus, the GUI will still be responsive while the operation is being executed in the background. The *content provider* is just like a database service and is used by apps to store and retrieve data. It can also be used to share data between apps. The *broadcast receiver* allows apps to receive events broadcasted by other services and apps and perform operations based on those events.

2.2 Android Security Model

There are four basic security features that Android offers where one is at kernel level and the others are at framework level. Although Android is built on top of the Linux kernel, it uses the security features provided by the kernel differently. The main difference between Android and Linux, is how user IDs are utilized. Other Android security features namely sandboxing, permissions and application signing use the assigned user IDs. We discuss these security features in detail below.

2.2.1 Assigning App ID and Protection of Data

One of the important differences between Android and Linux system is the use of user ID. Android assigns a Linux user ID to every app installed in the device while Linux assigns a user ID to every person profile on the machine. The assignment is done at app install time in Android. The given Linux user ID for the same app will be different between Android devices. An app's Linux user ID does not change as long as the app is still installed on the device. Just for our own purposes we call this as “app user ID” or AUID. Before Android 4.2 (Jelly Bean API 17), Android was a single user OS and the AUID concept worked. Now Android has enabled some devices (such as tablets) to allow two or more users on a single device. Nonetheless, it does not change the basic process of assigning Linux user IDs to apps as AUIDs. In order to support multiple users [6] [19], Android assigns a unique user ID (UUID) for every user. In order to support apps in the multi-user environment Android assigns an effective user ID (EUID) to each app installed on the device based on the AUID and UUID. Android creates private data folder for each installed app in a single user environment and for each user and app combination in the multi-user environment. The data folder is also mapped to an AUID/EUID. Thus, for an app to have read and write permission to a data folder and its contents, the AUID/EUID for the app and data folder should be same. Apps do not have permission to the files stored by other apps so they cannot read or write to those files unless the file was permission was set to “world readable”. Thus, apps are consequently isolated from each other via the AUID/EUID. Thus, even when multiple users are allowed in the Android system, apps cannot use data across users thereby protecting data between users.

2.2.2 App Isolation

Sandboxing is the method of separating the running processes. Sandboxing provides isolations that are needed for security purposes and help avoiding propagation of malwares or viruses. The isolation by assignment of AUID described above, is one part of the sandboxing scheme that provides data isolation. The other part is running apps as separate virtual machine processes to isolate them from each other. We can define every app as a virtual machine ready to be executed and this provides complete isolation and apps can only access their own files. For isolated apps to access any other Android features or services they need the appropriate user granted permissions and is discussed next.

2.2.3 Permissions to Access Protected Resources

An app needs permissions to use Android services and features like contacts, internet and location. An app does not need any permission if it does not use any protected feature. Permissions that are needed by an app are specified by the developer in the app's `AndroidManifest.xml` file using the tag `<uses-permission>`. The current list of system defined permissions is available at [16]. Once permissions are listed by the app developers, users have to grant those permissions to the apps during installation without which the apps are not installed. Android assigns a group ID (GID) to each app and it is same as the AUID. For instance, if the AUID of an app is 10039, then the GID will be also 10039. In addition, an app can also have more GIDs, as when an app is granted a permission the GID for that permission is also added to the app's set of GIDs. For example, assume the permission for "internet access" is GID "inept". If internet permission is given to an app with AUID 10001, that app will have 10001 and "inept" in its GID set. So when it comes to checking if an app has permission for a requested resource just comparing the GIDs should be sufficient. Once an app has the permission via the assigned GIDs, it could everything that the permission allows it on the device.

2.2.4 Application Signing

Each app is signed with a security certificate with the developer's private key when releasing the app on the market for users to download. This signature is used for app maintenance, updates, and to identify the developer. It also allows developer to share resources with other apps developed with the same signature.

2.3 Inter-Process Communication

Communication between Android system and apps or between apps can be used for malicious purposes leading to privacy leaks. Almost all communications require some kind of permission or receiver registration defined in the `AndroidManifest.xml` file. There are many ways in which communication can take place and they are: Android manager, broadcast, file sharing, content providers, and starting apps from other apps.

2.3.1 Android Manager

The first way is using Android managers to provide access to different resources. Apps use managers to access system services. For example, to access mobile device location information, an app needs to interact with the location manager by invoking appropriate methods of the location manager class. This process is shown in Figure 3.

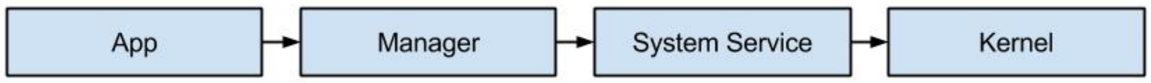

Figure 3: Communication between Apps and System

2.3.2 Broadcasting

The second way of communication is by sending and receiving broadcasts. It is the way the Android system communicates with apps. The broadcasts sent by the Android system usually contain status information. For example, a system broadcast could contain a message saying that the battery is low or the boot process is done. The apps that subscribe or listen to these broadcasts can receive the required information and can initiate appropriate actions. Apps can communicate with other apps in a similar manner. To send a broadcast, an app can use one of the overloaded `sendBroadcast` methods listed in [20].

In order to receive broadcasts, an app needs to have a class extending the `BroadcastReceiver` class and then register it in its `AndroidManifest.xml` file. To register the receiver, the app has to include the tag `<receiver>` between the start and end `<application>` tags. It is the same principle to receive a broadcast sent by the system. Once the receiver is created and registered, it is ready for use. On reception of a broadcast, the `onReceive()` method is executed.

2.3.3 Sharing using Files

The third way is to use file sharing and is the easiest way. When the app saves a file in its private folder with “world readable” and/or “world writable” permissions, the file could then be used by multiple apps to pass information among them. But this approach can lead to serious security issues.

2.3.4 Content Provider

The fourth way of communication is via content provider and is mainly used to save and retrieve common data. Using content provider, data can be saved in a structured manner and can be easily retrieved. For an app to create/use a content provider, the content provider needs to be defined in the `AndroidManifest.xml` using the `<provider>` tag. In between the start and end provider tags, the authority of the provider need to be specified. The authority is basically the name of the content provider created and/or being used. For system providers like contacts, having the necessary permission to access the protected feature is enough to get access to the provider. A content provider can be viewed similar to a database and apps can query, insert, update, and delete data. In order to perform those operations, apps need to use content resolver object in their application’s context which in turn communicates with the content provider object. For more information, refer to [17].

2.3.5 App Invocation

The final way of communication between apps is carried out by starting an app from another app. For example, a camera app can programmatically start an email app using methods like `startActivity()`. When the camera app starts an email app, it can also share the data with the email app. Developers of the email app have to register what activities can be opened by other apps via the `AndroidManifest.xml` file. The developers of the camera app can define the list of apps that could be opened by the camera app to process the data. The email app can be in the list of possible apps so that it could respond to the camera app’s request. This technique is mostly used to open system apps from a user app but it can also be used to open other user apps. A user app can only be opened by another user app if the app to be opened has been defined as such in its `AndroidManifest.xml` file as discussed above. Most system apps are already set up to be opened by user apps.

2.3.6 Summary

Two are more apps written by the same developer or different developers can escalate their privileges through collaboration through one or more of the above discussed inter-process communication methods. Thus, monitoring of inter-process communication is also crucial to understand and address the security and privacy risks posed by privilege escalation attacks.

3 App Monitoring

Monitoring apps and communication between apps will allow us to address the two major issues, coarse permissions and privilege escalation attacks, discussed in Sections 1.1 and 2. In this project, we investigated two different approaches to monitor apps and their communication. These approaches should be able to track all the activities performed by apps. The first approach was to create a user installed monitoring app that can be installed from the Google Play store by users. The second approach was to instrument the operating system and create custom Android ROM to monitor apps and communication between apps. Both the approaches are discussed below.

3.1 Understanding Permission Leaks

In order to explore situations where privacy leaks can happen we ran some experiments. Below, we discuss each of the experiment briefly.

As specified in [23] Apps installed in Android are sandboxed and do not have any permissions by default that can impact other apps, user, or the system itself. As the first experiment, we created an App without any permission but tried to access the contacts database. To our expectation the app crashed. The second experiment was to run the same app with incorrect permission and the app crashed and the result was as expected. The third experiment was to test escalation of privileges by apps through collaboration. We created two apps and kept it under same owner (com.android.owner) App A had permission to access contacts, and App B did not have permission to access contacts. As expected, keeping the code from two different apps in the same com.android.owner does not allow sharing of data or permissions as they both get different IDs, and the app crashed. In all the above experiments the apps crashed with error in acquire unstable provider. The fourth experiment was to access different parts of the contact database with an app that had permission to access contacts. This was mainly conducted to explore the coarse access control issue. As expected the app was able to access all parts of the contacts database. The final experiment was to test the privilege escalation by using shared

UserID between both the apps A and B. In this case, apps were able to share their permissions and data and app B was able to access contacts through app A's permission.

3.2 User Installed App

In this approach, the goal was to create a user installed app downloadable from the Google Play store, which will monitor apps and inter-process communication. Unfortunately, the user installed app approach did not work because a user installed app does not have enough privilege to access the system information and other app information. In fact, apps are isolated by sandboxing and cannot access each other information. One possible way is to reinstall all the user apps, as discussed in Section 5, after security upgrades for monitoring [9].

3.3 Android Instrumentation

Because of permissions limitations and access denials with the user installed app approach, we took the second approach of modifying the operating system through code instrumentation and adding classes to the Android OS and creating a custom ROM. We carried out multiple steps for this approach using Android Studio and Android SDK Tools. First, we performed experiments to understand how apps use their permissions and how communication between apps work in the Android OS to identify the instrumentation points and interesting events that can be captured and raised. Next, we created classes that can be used by any of the instrumentation point to capture, format, and raise events. Finally, we performed tests to verify that the identified instrumentation points worked and were able to capture activities within apps and their communication and generate and raise events.

3.3.1 Identifying Instrumentation Points

Our goal was to identity instrumentation points for three different services: content provider, location and network activity. Content provider, as explained in Section 2.3.4, allows apps to store and share data. When apps need to user Android contacts, calendar, album, etc. they have to use the content provider service. We also looked at location and network activity to instrument as they are really critical from a user's perspective to prevent privacy and information leak.

Our first investigation consisted of including different permissions in the `AndroidManinfest.xml` file for a contact app and looked at how the app behaved. We used a contact app for this purpose as we wanted to see what is needed to access the Android contacts database. This also allowed us to understand how services provided by content providers in general can be instrumented as Android contacts use content provider. Based on our investigation and using the Android developer website, to access contacts the app needs to use the contentRevolver class. We started testing the contact app without providing any permission in the AndroidManinfest.xml file. As soon the contact app started loading, it crashed due to lack of permissions. We traced the crash error to a method call in the contentResolver class. The method named query() in the contentResolver class has a uniform resource Identifier (URI) parameter which needs the name of the content provider (contact provider, in our case) we want to access. The URI is used to identify the content provider and verify if the app has the permission to access that content provider. In case of our contact app, the URI points to the contact provider and since the app does not have the required permission it crashed. The contact app also crashed when we used a permission that does not have anything to do with what we are accessing. We gave the internet permission to our contact app and it crashed at the query() method for the same reasons. The last part was to give the correct permission to the app. The app did not crash and was able to display all the contacts saved on the phone. With the contact permission, the app has access to all the information of a contact. For instance, the app was able to retrieve the full name, the address, e-mail, etc., basically everything stored in the contact content provider. There is no limit to what the app can access in that content provider. In addition, we were able to trace the call to the methods of the contentResolver class and URI used being CONTENT_URI. Figure 4 lists the steps involved in accessing the content provider, where we have termed all the CONTENT_URI related to contacts as ContactProvider URI. These experiments gave us a better understanding of how permissions and content providers work, and also the granularity of the contact permission.

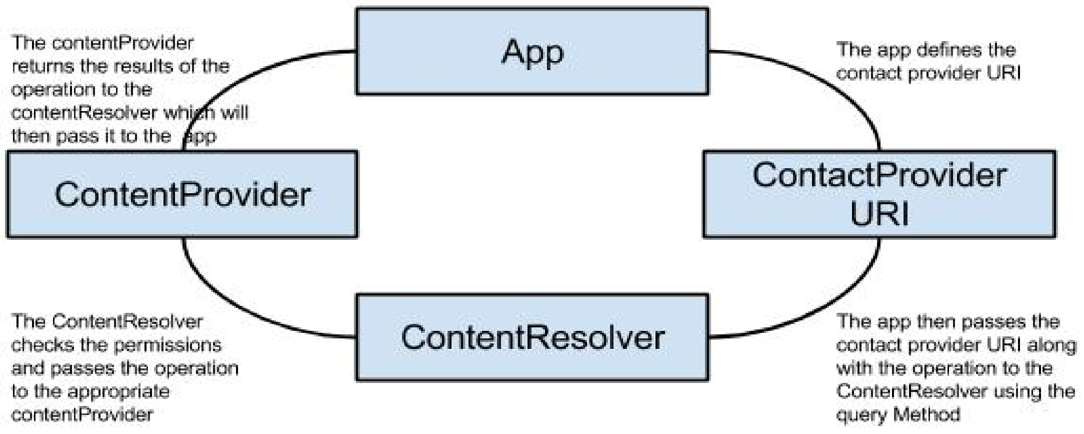
App
ContentProvider
ContactProvider URI
ContentResolver
The contentProvider returns the results of the operation to the contentResolver which will then pass it to the app
The app defines the contact provider URI
The ContentResolver checks the permissions and passes the operation to the appropriate contentProvider
The app then passes the contact provider URI along with the operation to the ContentResolver using the query Method

Figure 4: Accessing a ContentProvider

The next step was to understand how communications between apps work and particularly how they can share data between themselves. We tried two scenarios using two different apps. Both these are contact apps, but app A had the contact permission and app B had the internet permission.

- For the first scenario we created app A and app B to be from the same owner. We had both the apps under the same package name folder, but just by putting the code from two different apps in the same package did not allow sharing of data or permissions as they both get different Linux user IDs. We then used the sharedUserId attribute of the manifest tag in the `AndroidManinfest.xml` file to allow apps to share the data and permissions [18]. This allowed app B to read contacts from the Android contact database even when it did not have the required permissions but by using the permission granted to app A.
- The second scenario was to have the two apps being from two different owners and try to share data. In order for this scenario to work we used the "launch from another app" approach. App B opened an activity from app A and then used app A to access the

contacts. Not all apps are launchable from other apps. Apps are private and cannot be launched by any other apps unless explicit permissions are provided by app owners.

These experiments allowed us to understand coarse permission and privilege escalation issues and also provided us with information about where to carry out the instrumentation to monitor activities related to accessing Android apps that use content provider services. They helped us understand how content providers work, particularly the content provider for contacts. All apps (system and user apps) have to go through the content provider for contacts to access the contacts. And as it was described above, a content provider is accessed and managed using a content resolver. So we decided to instrument the contentResolver.java class. Similarly we ran other experiments and explored the Android system to identify instrumentation points for monitoring location and network connectivity. As a result our investigations, we identified the following instrumentation points:

- For content provider: query(), insert(), update(), delete(), and applyBatch() methods in contentResolver.java
- For location : getLastKnownLocation() and getLastLocation() in LocationManager.java and getSystemService() in contextImpl.java
- For network connectivity : getSystemService() in contextImpl.java

3.3.2 Event specification

In order capture interesting events through instrumentation we created a new class that can be invoked from various instrumentation points. We named the class as SimpleEventHandler.java and it contains multiple overloaded methods called SEvent() that can be invoked by other modules within Android to create, generate, and broadcast events. Below, we discuss the overloaded methods.

1. SEvent(String Tag, Context context, Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)

    - This methods takes seven different inputs. It uses the context object to retrieve the package name and app name of the app that invokes an activity related to content provider, location, or network connectivity. The uri object contains the URI requested by the app. The projection object contains the name of the attributes of the content provider accessed by the app. For instance, when an app opens a contact provider, the projection object may include different information about which columns of a contact is being accessed such as contact ID, first name, etc. The selection object contains the conditions (similar to SQL WHERE clause) and the conditions themselves can be generalized to substitute values from the selectionArgs object.

- This SEvent() then creates events based on the inputs, and broadcasts those events, which can be received by broadcast receivers.

2. SEvent(String Tag, Context context, ArrayList<ContentProviderOperation> operations)
   - This method gets context as an input, but also gets a set of operations (projection, selection, and selectionargs) instead of just one operation as in the first method. It then calls the first method internally.
3. SEvent(String Tag, String PackageName, Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)
   - This method is similar to the first one, except that it takes a package name as an input instead of the context. This SEvent() can then create events based on the inputs, and broadcasts those events, which can be received by broadcast receivers.
4. SEvent(String Tag, Context context, Location loc)
   - This method gets context and also the loc object. The location object is used to extract information about the location access requests by apps.
5. SEvent(String Tag, Context context)
   - This method calls the first method internally, and sets other required objects to

null before calling. This is used by the network and location requests where there is no specific operation requested.

All the above SEvent() methods create events using one of the XML event specifications defined in the following sections. First, we will explain the tags and then show the different specifications.

- In the `<FILE>` tag, attribute Name is the name of the instrumented java class. The Function attribute contains the name of the instrumented method. The AccessType attribute contains the type of activity such as read, write, delete, etc.
- `<Package_Name>` is the name of the package.
- `<Application_Name>` represents the name of the app.
- `<URI>` is the name of the provider.
- `<Accuracy>` is the estimated accuracy of the location in meters.
- `<Altitude>` is the altitude in meters.
- `<Latitude>` is the latitude in degree.
- `<Longitude>` is the longitude in degree.
- `<Provider>` is the name of the provider that returned the location.
- `<Time>` is the time when the function was called.
- The `<uid>` is the app user ID.

3.3.2.1 Specification for Content Provider Events

```
<File Name = , Function = , AccessType =  >

      <Package_Name> </Package_Name>
```

```
        <Application_Name> </Application_Name>

        <URI> </URI>

        <Projection> </Projection>

        <Selection> </Selection>

        <SelectionArgs> </SelectionArgs >

        [<Operation> </Operation>]

        <SortOrder> </SortOrder>

        <Time> </Time>

        <uid> </uid>

</File>
```

3.3.2.2 Specification for Location Events

```
<File Name = , Function = , AccessType =  >

        <Package_Name> </Package_Name>

        <Application_Name> </Application_Name>

        <URI> </URI>

        <Accuracy> </Accuracy>

        <Altitude> </Altitude>

        <Latitude> </Latitude>

        <Longitude> </Longitude>

        <Provider> </Provider>

        <Time> </Time>

        <uid> </uid>

</File>
```

3.3.2.3 Specification for Network Connectivity Events

```
<File Name = , Function = , AccessType =  >
       <Package_Name> </Package_Name>
       <Application_Name> </Application_Name>
       <URI> </URI>
       <Time> </Time>
       <uid> </uid>
</File>
```

3.3.3 Event Generation

In order to instrument content provider via query(), insert(), update(), delete(), and applyBatch() methods in contentResolver.java, which we identified in section 3.3.1, we used the following steps. In each of these methods, we constructed a tag to insert the class name, function name, and access type. We then invoked one of the SEvent() methods to create and broadcast the event. For example, we instrumented the query() method as:

```
String actionTag = "<File Name = \"ContentResolver.java\" Function =
\"query()\" AccessType = \"Read\">";

SimpleEventHandler.SEvent(actionTag,mContext, uri, projection,selection,
                          selectionArgs, sortOrder);
```

In general, every time an app accesses a content provider one of the instrumented methods is invoked. The instrumented method in turn invokes one of the SEvent() methods defined in Section 3.3.2. The SEvent() method generates an XML structured event by using the inputs and broadcasts it using the sendBroadcast() method. The broadcasted events can be received by any user app with the appropriate receiver registration. As shown below, in an SEvent() method, first an intent is created, then the XML message is added to the intent, and then the intent is broadcasted using the sendBroadcast() method.

```
Intent intent = new Intent("com.derrickAhipo.Event");
intent.putExtra("SEvent",XMLMsg);
try{
     context.sendBroadcast(intent);
}
catch(Exception e){
     Log.d("Error " , e.getMessage());
}
```

In addition, we also wanted to save the XML message to a file on the mobile device hard drive. The main issue we faced was the inability to find a good location to store the file in a secure manner and at the same time provide access to that file to other apps.

Similar to the content provider, we also instrumented the getSystemService(String name) in the ContextImpl.java class. The getSystemService() method is used to get any service manager. Since we limited our study to the location and internet services, the instrumentation is activated only when the parameter name is "location" or "connectivity". If the name is "location", the service manager returned will be the location manager and if it is "connectivity", the service manager returned will be internet manager. To access the results of the location manager we instrumented both getLastLocation() and getLastKnowLocation(String provider) in LocationManager.java as these methods are invoked when apps want information about the current location of the device. We instrumented these two methods similar to the query() method so that every time an app requests a location we can also monitor it. All the source code for these instrumented files are attached to the project development guide.

4 Receiver App and Testing

In order to receive events broadcasted by the SEvent() methods of SimpleEventHandler.java class, we created a user installed app containing a broadcast receiver. We registered the app to receive the events by adding the <receiver> tag to the `AndroidManifest.xml` file of the app as described in Section 2.3.2 A broadcast receiver within an app executes the onReceive() method only when it receives the appropriate broadcast. In the onReceive() method of the receiver app, we save the XML events as a file in the app's data folder using a file object. In the future, we are planning to integrate the event receiver to an event stream processing system [21].

Currently, through our instrumentation and broadcast receiver we are able to monitor all interactions made by an app with a content provider, and all accesses to the location manager and connectivity manager. To test our system, we used native apps and other apps from [24] which use content providers, location services and connectivity services. The apps that we downloaded from the [24] are: Firefox, F-Droid, Duck Duck Go, and Here GPS Location.

4.1 Content provider instrumentation

To test the content provider, we used the Android contact app. First, we opened the app which retrieved and displayed all the contacts stored on the phone. As expected, XML events were generated capturing the read operation. The below snippet shows the main XML events captured during opening of the app.

```
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
	<Package_Name>com.android.contacts</Package_Name>
	<Application_Name>Contacts</Application_Name>
	<URI>content://com.android.contacts/profile</URI>
	<Projection>_id display_name contact_presence contact_status photo_id
photo_thumb_uri lookup is_user_profile </Projection>
	<Time>2015-08-29 02:23:32 PM</Time>
```

```
        <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>com.android.contacts</Package_Name>
        <Application_Name>Contacts</Application_Name>
        <URI>content://com.android.contacts/contacts?directory=0&limit=100</URI>
        <Projection>photo_id </Projection>
        <selection>photo_id NOT NULL AND photo_id!=0</selection>
        <sort_Order>starred DESC, last_time_contacted DESC</sort_Order>
        <Time>2015-08-29 02:23:32 PM</Time>
        <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>com.android.contacts</Package_Name>
        <Application_Name>Contacts</Application_Name>
        <URI>content://com.android.contacts/contacts?directory=0&limit=100</URI>
        <Projection>photo_id </Projection>
        <selection>photo_id NOT NULL AND photo_id!=0</selection>
        <sort_Order>starred DESC, last_time_contacted DESC</sort_Order>
        <Time>2015-08-29 02:23:32 PM</Time>
        <uid>10002</uid>
</File>

        <Package_Name>com.android.contacts</Package_Name>
        <Application_Name>Contacts</Application_Name>
        <URI>content://com.android.contacts/contacts/strequent</URI>
        <Projection>_id display_name starred photo_uri lookup contact_presence
contact_status </Projection>
        <sort_Order>display_name COLLATE NOCASE ASC</sort_Order>
        <Time>2015-08-29 02:23:32 PM</Time>
        <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>com.android.contacts</Package_Name>
        <Application_Name>Contacts</Application_Name>
        <URI>content://com.android.contacts/contacts?android.provider.extra.ADDRESS_BOOK_IND
EX=true&directory=0</URI>
        <Projection>_id display_name contact_presence contact_status photo_id
```

```
photo_thumb_uri lookup is_user_profile </Projection>
	<selection></selection>
	<selectionArgs></selectionArgs>
	<sort_Order>sort_key</sort_Order>
	<Time>2015-08-29 02:23:32 PM</Time>
	<uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
	<Package_Name>com.android.contacts</Package_Name>
	<Application_Name>Contacts</Application_Name>
	<URI>content://com.android.contacts/provider_status</URI>
	<Projection>status data1 </Projection>
	<Time>2015-08-29 02:23:33 PM</Time>
	<uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
	<Package_Name>com.android.contacts</Package_Name>
	<Application_Name>Contacts</Application_Name>
	<URI>content://com.android.contacts/profile</URI>
	<Projection>_id display_name contact_presence contact_status photo_id
photo_thumb_uri lookup is_user_profile </Projection>
	<Time>2015-08-29 02:23:35 PM</Time>
	<uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
	<Package_Name>com.android.contacts</Package_Name>
	<Application_Name>Contacts</Application_Name>
	<URI>content://com.android.contacts/contacts?android.provider.extra.ADDRESS_BOOK_IND
EX=true&directory=0</URI>
	<Projection>_id display_name contact_presence contact_status photo_id
photo_thumb_uri lookup is_user_profile </Projection>
	<selection></selection>
	<selectionArgs></selectionArgs>
	<sort_Order>sort_key</sort_Order>
	<Time>2015-08-29 02:23:35 PM</Time>
	<uid>10002</uid>
</File>
```

Then we added a contact named “The Real Test” with the phone number “574-888-9999”. Adding a contact operation is executed by the applyBatch() method. The below snippet shows the XML event generated. As shown the event shows the add operation, including the name and phone number.

```
<File Name = "ContentResolver.java" Function = "applyBatch()" AccessType = "unknown">
	<Package_Name>com.android.contacts</Package_Name>
	<Application_Name>Contacts</Application_Name>
	<operations>mType: 1, mUri: content://com.android.contacts/raw_contacts, mSelection:
null, mExpectedCount: null, mYieldAllowed: false, mValues: account_type=null
aggregation_mode=2 data_set=null account_name=null, mValuesBackReferences: null,
mSelectionArgsBackReferences: null
mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount:
null, mYieldAllowed: false, mValues: data1=(574) 888-9999 data2=2
mimetype=vnd.android.cursor.item/phone_v2, mValuesBackReferences: raw_contact_id=0,
mSelectionArgsBackReferences: null
mType: 1, mUri: content://com.android.contacts/data, mSelection: null, mExpectedCount:
null, mYieldAllowed: false, mValues: data5=Real data1=The Real Test data6=null data2=The
data4=null data3=Test mimetype=vnd.android.cursor.item/name, mValuesBackReferences:
raw_contact_id=0, mSelectionArgsBackReferences: null
mType: 2, mUri: content://com.android.contacts/raw_contacts, mSelection: _id=?,
mExpectedCount: null, mYieldAllowed: false, mValues: aggregation_mode=0,
mValuesBackReferences: null, mSelectionArgsBackReferences: {0=0}
</operations>
	<Time>2015-08-29 07:27:14 PM</Time>
	<uid>10002</uid>
</File>

<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
	<Package_Name>com.android.inputmethod.latin</Package_Name>
	<Application_Name>Android Keyboard (AOSP)</Application_Name>
	<URI>content://com.android.contacts/contacts</URI>
	<Projection>_id </Projection>
	<Time>2015-08-29 07:27:16 PM</Time>
	<uid>10034</uid>
</File>
```

```
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
      <Package_Name>com.android.contacts</Package_Name>
      <Application_Name>Contacts</Application_Name>
      <URI>content://com.android.contacts/contacts/0/suggestions?limit=3&query=name%3AThe%
20Real%20Test</URI>
      <Projection>_id </Projection>
      <Time>2015-08-29 07:27:17 PM</Time>
      <uid>10002</uid>
</File>
```

Then we deleted the contact “The Real Test” we previously added. This executes the delete() method. Our system was able to detect the operation and raise the corresponding events. The below snippet shows the XML events captured during the deletion process.

```
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
      <Package_Name>com.android.contacts</Package_Name>
      <Application_Name>Contacts</Application_Name>
      <URI>content://com.android.contacts/contacts/lookup/0r21-
4F37314B31293F4F314D4F/21/entities</URI>
      <Projection>raw_contact_id account_type data_set contact_id lookup </Projection>
      <Time>2015-09-02 07:11:55 AM</Time>
      <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "delete()" AccessType = "Delete">
      <Package_Name>com.android.providers.media</Package_Name>
      <Application_Name>Media Storage</Application_Name>
      <URI>content://media/none/media_scanner</URI>
      <Time>2015-09-02 07:11:58 AM</Time>
      <uid>10005</uid>
</File>
<File Name = "ContentResolver.java" Function = "delete()" AccessType = "Delete">
      <Package_Name>com.android.contacts</Package_Name>
      <Application_Name>Contacts</Application_Name>
      <URI>content://com.android.contacts/contacts/lookup/0r21-
4F37314B31293F4F314D4F/21</URI>
```

```
        <Time>2015-09-02 07:12:01 AM</Time>
        <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>com.android.inputmethod.latin</Package_Name>
        <Application_Name>Android Keyboard (AOSP)</Application_Name>
        <URI>content://com.android.contacts/contacts</URI>
        <Projection>_id </Projection>
        <Time>2015-09-02 07:12:03 AM</Time>
        <uid>10034</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>com.android.contacts</Package_Name>
        <Application_Name>Contacts</Application_Name>
        <URI>content://com.android.contacts/provider_status</URI>
        <Projection>status data1 </Projection>
        <Time>2015-09-02 07:12:05 AM</Time>
        <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>com.android.contacts</Package_Name>
        <Application_Name>Contacts</Application_Name>
        <URI>content://com.android.contacts/contacts/strequent</URI>
        <Projection>_id display_name starred photo_uri lookup contact_presence
contact_status </Projection>
        <sort_Order>display_name COLLATE NOCASE ASC</sort_Order>
        <Time>2015-09-02 07:12:06 AM</Time>
        <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>com.android.contacts</Package_Name>
        <Application_Name>Contacts</Application_Name>
        <URI>content://com.android.contacts/contacts/strequent</URI>
        <Projection>_id display_name starred photo_uri lookup contact_presence
contact_status </Projection>
        <sort_Order>display_name COLLATE NOCASE ASC</sort_Order>
        <Time>2015-09-02 07:12:06 AM</Time>
        <uid>10002</uid>
```

```
</File>

      <Projection>name_raw_contact_id display_name_source lookup display_name
display_name_alt phonetic_name photo_id starred contact_presence contact_status
contact_status_ts contact_status_res_package contact_status_label contact_id
raw_contact_id account_name account_type data_set dirty version sourceid sync1 sync2 sync3
sync4 deleted data_id data1 data2 data3 data4 data5 data6 data7 data8 data9 data10 data11
data12 data13 data14 data15 data_sync1 data_sync2 data_sync3 data_sync4 data_version
is_primary is_super_primary mimetype group_sourceid mode chat_capability status
status_res_package status_icon status_label status_ts photo_uri send_to_voicemail
custom_ringtone is_user_profile times_used last_time_used </Projection>
      <sort_Order>raw_contact_id</sort_Order>
      <Time>2015-09-02 07:12:07 AM</Time>
      <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
      <Package_Name>com.android.contacts</Package_Name>
      <Application_Name>Contacts</Application_Name>
      <URI>content://com.android.contacts/profile</URI>
      <Projection>_id display_name contact_presence contact_status photo_id
photo_thumb_uri lookup is_user_profile </Projection>
      <Time>2015-09-02 07:12:07 AM</Time>
      <uid>10002</uid>
</File>
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
      <Package_Name>com.android.contacts</Package_Name>
      <Application_Name>Contacts</Application_Name>
      <URI>content://com.android.contacts/contacts?android.provider.extra.ADDRESS_BOOK_IND
EX=true&directory=0</URI>
      <Projection>_id display_name contact_presence contact_status photo_id
photo_thumb_uri lookup is_user_profile </Projection>
      <selection></selection>
      <selectionArgs></selectionArgs>
      <sort_Order>sort_key</sort_Order>
      <Time>2015-09-02 07:12:07 AM</Time>
      <uid>10002</uid>
</File>
```

4.2 Location service instrumentation

To test the location service instrumentation, we used the Here GPS Location app. At start, the app will request the location service using the getSystemService() method to get the device location. As shown in the below XML messages our system captures the location request from the app and broadcasts the events.

```
<File Name = "ContextImpl.java" Function = "getSystemService" AccessType =
"Access_Location">
      <Package_Name>com.borneq.heregpslocation</Package_Name>
      <Application_Name>Here GPS Location</Application_Name>
      <Time>2015-08-26 10:42:02 PM</Time>
      <uid>10062</uid>
</File>

<File Name = "LocationManager.java" Function = "getLastKnownLocation" AccessType =
"Access_Location">
      <Package_Name>com.borneq.heregpslocation</Package_Name>
      <Application_Name>Here GPS Location</Application_Name>
      <Time>2015-08-26 10:42:06 PM</Time>
      <uid>10062</uid>
</File>

<File Name = "LocationManager.java" Function = "getLastKnownLocation" AccessType =
"Access_Location">
      <Package_Name>com.borneq.heregpslocation</Package_Name>
      <Application_Name>Here GPS Location</Application_Name>
      <Accuracy>20.0</Accuracy>
      <Altitude>0.0</Altitude>
      <Latitude>37.422005</Latitude>
      <Longitude>-122.084095</Longitude>
      <Provider>gps</Provider>
      <Time>2015-08-26 10:42:38 PM</Time>
      <uid>10062</uid>
</File>
```

4.3 Network connectivity service instrumentation

To test the network connectivity service instrumentation, we used the Duck Duck Go app and the Firefox App. These apps use the network by requesting the network connectivity service using the getSystemService() method. As shown below, we opened the Duck Duck Go app first and then searched for the keyword 'iusb'. The app in turn launched the Firefox App and performed the search. The below snippet shows the events captured. As shown the events also include information from the content provider module as Firefox uses the content provider to access its bookmarks, reading list, etc.

```
<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
      <Package_Name>se.johanhil.duckduckgo</Package_Name>
      <Application_Name>Duck Duck Go</Application_Name>
      <URI>content://duckduckgo/search_suggest_query/iusb?limit=50</URI>
      <Time>2015-08-26 10:55:23 PM</Time>
      <uid>10073</uid>
</File>

<File Name = "ContextImpl.java" Function = "getSystemService" AccessType =
"Access_connectivity">
      <Package_Name>org.mozilla.firefox</Package_Name>
      <Application_Name>Firefox</Application_Name>
      <Time>2015-08-26 10:55:35 PM</Time>
      <uid>10075</uid>
</File>

<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
      <Package_Name>org.mozilla.firefox</Package_Name>
      <Application_Name>Firefox</Application_Name>

      <URI>content://org.mozilla.firefox.db.browser/bookmarks?profile=default&limit=1</URI
>
      <Projection>_id </Projection>
      <selection>url = ? AND parent != ?</selection>
      <selectionArgs>http://duckduckgo.com/?q=iusb -3 </selectionArgs>
```

```
        <sort_Order>url</sort_Order>
        <Time>2015-08-26 10:55:38 PM</Time>
        <uid>10075</uid>
</File>

<File Name = "ContentResolver.java" Function = "query()" AccessType = "Read">
        <Package_Name>org.mozilla.firefox</Package_Name>
        <Application_Name>Firefox</Application_Name>
        <URI>content://org.mozilla.firefox.db.readinglist/items?profile=default</URI>
        <Projection>_id </Projection>
        <selection>url = ? OR resolved_url = ?</selection>
        <selectionArgs>http://duckduckgo.com/?q=iusb http://duckduckgo.com/?q=iusb
</selectionArgs>
        <Time>2015-08-26 10:55:39 PM</Time>
        <uid>10075</uid>
</File>

<File Name = "ContextImpl.java" Function = "getSystemService" AccessType =
"Access_connectivity">
        <Package_Name>org.mozilla.firefox</Package_Name>
        <Application_Name>Firefox</Application_Name>
        <Time>2015-08-26 10:55:42 PM</Time>
        <uid>10075</uid>
</File>
```

5 Related Work

There is a lot of research works in Android security and in the area of app monitoring and privilege escalation attacks. Several approaches have been developed to monitor apps. Those approaches can be grouped in two categories: app-based and custom ROM. The Android permission extension (Apex) framework [8] uses the Android source code modification approach. Apex removes the all or none model of the Android Security system and allows the user to grant or deny a permission. Stowaway [7] is another tool developed to check if apps are overprivileged or not. Based on the method calls in the app, Stowaway produces a list necessary permissions and compare it to the permission the app asked for at install time. Aurasium [22] is an app repackaging tool that is used to monitor apps. The repackaging process adds instrumentation and monitoring code to the app package (apk) as opposed to the Android system. The Cyanogenmod [9] take the custom ROM approach and supports app permissions management where users can install apps without granting all the permissions required by the app during installation. SRT Appguard is an app sold on Google play. According to [9] SRT Appguard app allows users to remove permissions granted during installation and also to monitor other apps. Some of the key features are [9]: "dynamic permission management for apps, works without rooted device or modified firmware, riskscore assessment shows how potentially dangerous apps are, monitored apps still receive updates of Google Play, comprehensive logs show what apps really do, and new dashboard shows the security status of your device." SRT Appguard uninstalls user apps and adds its security upgrades to user apps and then reinstalls them. It cannot uninstall pre-installed system apps such as contacts and add its security upgrades. So this approach has limitation on what can be monitored.

6 Conclusions and Future Work

In this project, we developed a custom Android ROM that contains the necessary instrumentations to monitor apps accesses to the content providers, the location service and the connectivity service. To develop our custom ROM, we identified the instrumentation points, developed an XML event format, and created a common class that can be invoked by instrumented methods to capture, generate, and broadcast events. We created a receiver app that can receive the broadcasted events. These events can then can be used to monitor activities within apps and communication between apps.

As future work, we plan on integrating the event generation module with an event stream processing system that can receive events and perform on the fly processing of those events and triggering alerts. We also plan to log the events captured and perform log analysis.

7 References

[1] Android Studio, URL: http://developer.android.com/sdk/index.html , Last Accessed: August, 2015

[2] Beck, M., Bohme, H., Dziadzka, M., Kunitz, U., Magnus, R., Schroler, C., Verworner, D. Linux Kernel Programming. 3rd Ed. Pearson Education, Delhi, 2004. ISBN: 81-7808-805-3

[3] Mobile OS (Operating System) Percent Market Share. Retrieved August, 2015, from StatCounter Global Stats: http://stats.areppim.com/stats/stats_mobiosxtime_nam.htm

[4] Smartphone OS Market Share, 2015 Q2. Retrieved August, 2015, from IDC: http://www.idc.com/prodserv/smartphone-os-market-share.jsp

[5] Number of Android applications. Retrieved August, 2015, from AppBrain: http://www.appbrain.com/stats/number-of-android-apps

[6] Elenkov, N. Android Security Internals: An In-Depth Guide to Android's Security Architecture. No Starch Press, San Francisco, 2014. ISBN: 978-1-59327-581-5

[7] Felt, A. P., Chin, E., Hanna, S., Song, D., Wagner, D. 2011. Android permissions demystified. In Proceedings of the 18th ACM conference on Computer and communications security (CCS '11). ACM, New York, NY, USA, 627-638. Retrieved August, 2015, from University of California, Berkeley: https://www.cs.berkeley.edu/~dawnsong/papers/2011%20Android%20permissions%20demystified.pdf

[8] Nauman M., Khan, S., Zhang, X. 2010. Apex: Extending Android permission model and enforcement with user-defined runtime constraints. In Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS '10). ACM, New York, NY, USA, 328-332. Retrieved August, 2015, from Penn State University: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.193.2604&rep=rep1&type=pdf

[9] Mobile Android Security, URL: http://www.srt-appguard.com/en/ , Last Accessed: August,

2015

[10] Cyanogenmod, URL: http://www.cyanogenmod.org/ , Last Accessed: August, 2015

[11] Security Tips, URL: http://developer.android.com/training/articles/security-tips.html#declaring , Last Accessed: August, 2015

[12] Creating a Content Provider, URL: http://developer.android.com/guide/topics/providers/content-provider-creating.html , Last Accessed: August, 2015

[13] System and kernel security, URL: https://source.android.com/devices/tech/security/overview/kernel-security.html , Last Accessed: August, 2015

[14] ART and Dalvik, URL: https://source.android.com/devices/tech/dalvik/ , Last Accessed: August, 2015

[15] Deamon Tools, URL: http://www.daemon-tools.cc/home , Last Accessed: August, 2015

[16] Manifest.permission, URL: http://developer.android.com/reference/android/Manifest.permission.html , Last Accessed: August, 2015

[17] Content Providers, URL: http://developer.android.com/guide/topics/providers/content-providers.html , Last Accessed: August, 2015

[18] <manifest>, URL: http://developer.android.com/guide/topics/manifest/manifest-element.html#uid , Last Accessed: August, 2015

[19] Android 4.2 APIs, URL: http://developer.android.com/about/versions/android-4.2.html#MultipleUsers , Last Accessed: August, 2015

[20] Context, URL: http://developer.android.com/reference/android/content/Context.html , Last Accessed: August, 2015

[21] Jiang, Q., Adaikkalavan, R., & Chakravarthy, S. 2010. Event/Stream Processing for Advanced Applications. In A. Cuzzocrea (Ed.), Intelligent Techniques for Warehousing and Mining Sensor Network Data (pp. 305-325). Retrieved August, 2015, from IGI-Global: http://www.igi-global.com/chapter/event-stream-processing-advanced-applications/39551.

[22] Xu, R., Saidi, H., Anderson, R. 2012. Aurasium: practical policy enforcement for Android applications. In Proceedings of the 21st USENIX conference on Security symposium (Security'12). Retrieved August, 2015, from University of Cambridge: http://www.cl.cam.ac.uk/~rja14/Papers/aurasium-usenix-sec-12.pdf.

[23] System Permissions, URL: http://developer.android.com/guide/topics/security/permissions.html , Last Accessed: August, 2015

[24] F-DROID, URL: https://f-droid.org/ , Last Accessed: August, 2015

[25] Supreme Court requires warrants for cell phone searches on arrest. URL: https://www.washingtonpost.com/news/volokh-conspiracy/wp/2014/06/25/supreme-court-requires-warrants-for-cell-phone-searches-on-arrest/ , Last Accessed: August, 2015