Android Security Model that Provide a Base Operating System

Prof. Parag Gholse ,Asst.Professor,SRPCE College,Nagpur
Mr.Shashank Thakre,Student,SRPCE College,Nagpur.

Abstract— Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The android provides the tools and APIs necessary to begin developing applications on the Android platform using programming language. Android is a widely anticipated open source operating system for mobile devices that provides a base operating system, an application middleware layer, a Java software development kit (SDK), and a collection of system applications. Android has a unique security model, which focuses on putting the user in control of the device. Android devices however, don't all come from one place, the open nature of the platform allows for proprietary extensions and changes.This paper we should already be familiar with Android's basic architecture and major abstractions including: Intents, Activities, Broadcast Receivers, Services, Content Providers and Binder. As android is open source we should also have this code available to us. Both the java and C code is critical for understanding how Android works, and is far more detailed than any of the platform documentation.

Keywords: *Android, Android Tools, API, System Application, Security Model and Architecture of Android.*

*****

1 INTRODUCTION

T HE next generation of open operating systems won't be on desktops or mainframes but on the small mo- bile devices we carry every day. The openness of these new environments will lead to new applications and markets and will enable greater integration with existing online services. However, as the importance of the data and services our cell phones support increases, so too do the opportunities for vulnerability. It's essential that this next generation of platforms provide a comprehensive and usable security infrastructure. Developed by the Open Handset Alliance (visibly led by Google), Android is a widely anticipated open source operating system for mo- bile devices that provides a base operating system, an application middleware layer, a Java software develop- ment kit (SDK), and a collection of system applications. Although the Android SDK has been available since late 2007, the first publicly available Android-ready "G1" phone debuted in late October 2008. Since then, Android's growth has been phenomenal: T- Mobile's G1 manufacturer HTC estimates shipment vo- lumes of more than 1 million phones by the end of 2008, and industry insiders expect public adoption to increase steeply in 2009. Many other cell phone providers have either promised or plan to support it in the near future. A large community of developers has organized around Android, and many new products and applications are now available for it. One of Android's chief selling points is that it lets developers seamlessly extend online services to phones. The most visible example of this feature is—unsurprisingly—the tight integration of Google's Gmail, Calendar, and Contacts Web applications with system utilities. Android users simply supply a username and password, and their phones automatically synchronize with Google services. Other vendors are rapidly adapting their existing instant messaging, social networks, and gaming services to Android, and many enterprises are looking for ways to integrate their own internal opera- tions (such as inventory management, purchasing, receiv- ing, and so forth) into it as well. Traditional desktop and server operating systems have struggled to securely inte- grate such personal and business applications and servic- es on a single platform; although doing so on a mobile platform such as Android remains nontrivial, many re- searchers hope it provides a clean slate devoid of the complications that legacy software can cause. Android doesn't officially support applications developed for oth- er platforms: applications execute on top of a Java mid-dleware layer running on an embedded Linux kernel, so developers wishing to port their application to Android must use its custom user interface environment. Addi- tionally, Android restricts application interaction to its special APIs by running each application as its own user identity. Although this controlled interaction has several beneficial security features, our experiences developing Android applications have revealed that designing secure applications isn't always straightforward. Android uses a simple permission label assignment model to restrict access to resources and other applications, but for reasons of necessity and convenience, its designers have added several potentially confusing refinements as the system has evolved. This article attempts to unmask the complex-ity of Android security and note some possible develop-ment pitfalls that occur when defining an application's security. We conclude by attempting to draw some les-sons and identify opportunities for future enhancements that should aid in clarity and correctness.

2. ANDROID SECURITY MODEL

Android is a Linux platform programmed with Java and enhanced with its own security mechanisms tuned for a mobile environment3. Android combines OS features like efficient shared memory, preemptive multi-tasking, Unix user identifiers (UIDs) and file permissions with the type safe Java language and its familiar class library. The re-sulting security model is much more like a multi-user server than the sandbox found on the J2ME or Blackberry platforms. Unlike in a desktop computer environment where

a user's applications all run as the same UID, An-droid applications are individually soiled from each oth-er. Android applications run in separate processes under distinct UIDs each with distinct permissions. Programs can typically neither read nor-write each other's data or code[4], and sharing data between applications must be done ex-plicitly. The Android GUI environment has some novel security features that help support this isolation. Mobile platforms are growing in importance, and have complex requirements[5] including regulatory compliance6. Andro-id supports building applications that use phone features while protecting users by minimizing the consequences of bugs and malicious software. Android's process isolation obviates the need for complicated policy configuration files for sandboxes. This gives applications the flexibility to use native code without compromising Android's se-curity or granting the application additional rights.

Android permissions are rights given to applica-tions to allow them to do things like take pictures, use the GPS or make phone calls. When installed, applications are given a unique UID, and the application will always run as that UID on that particular device. The UID of an ap-plication is used to protect its data and developers need to be explicit about sharing data with other applications7. Applications can entertain users with graphics, play mu-sic, and launch other programs without special permis-sions.

Malicious software is an unfortunate reality on popular platforms, and through its features Android tries to mi-nimize the impact of malware. However, even unprivi-leged malware that gets installed on an Android device (perhaps by pretending to be a useful application) can still temporarily wreck the user's experience[8]. Users in this unfortunate state will have to identify and remove the hostile application. Android helps users do this, and

minimizes the extent of abuse possible, by requiring user permission for programs that do dangerous things like:

– Directly dialing the calls (which may incur

tolls),

– Disclosing the user's private/public data, or

– Destroying address books, email, ID, etc.

Generally a user's response to annoying, buggy or mali-cious software is simply to uninstall it. If the software is disrupting the phone enough that the user can't uninstall it, they can reboot the phone (optionally in safe mode[9], which stops non-system code from running) and then remove the software before it has a chance to run again.

2.1 ANDROID APPLICATIONS

The Android application framework forces a structure on developers. It doesn't have a main() function or single entry point for execution—instead, developers must de-sign applications in terms of components.

The user then uses the Friend Viewer application to re-trieve the stored geographic coordinates and view friends on a map. Both applications contain multiple components for performing their respective tasks; the components themselves are classified by their component types. An Android developer chooses from predefined component types depending on the component's purpose (such as interfacing with a user or storing data).

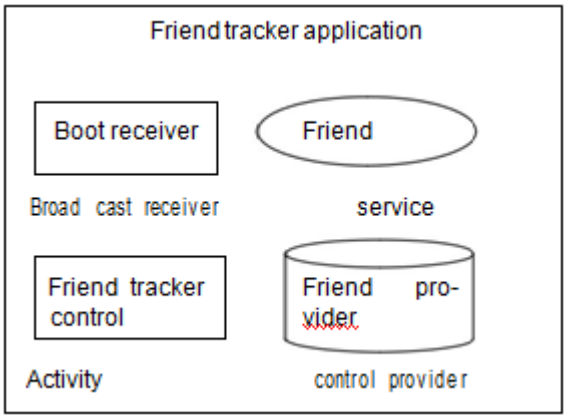

Figure 1: Android Application

2.2 ACTIVITY AND INTERACTION OF COMPO-NENTS

An application developer defines one activity per "screen." Activities start each other, possibly passing and returning values. Only one activity on the system has keyboard and processing focus at a time; all others are suspended. In Figure 2, the interaction between compo-nents in the FriendTracker and FriendViewer applications and with components in applications defined as part of the base Android distribution. In each case, one compo-nent initiates communication with another. For simplici-ty, we call this inter-component communication (ICC). In many ways, ICC is analogous to inter-process communi-cation (IPC) in Unix-based systems. To the developer, ICC functions identically regardless of whether the target is in the same or different application, with the exception of the security rules defined later in this article. The availa-ble ICC actions depend on the target component. Each component type supports interaction specific to its type for example, when FriendViewer starts FriendMap, the FriendMap activity appears on the screen. Service com-ponents support start, stop, and bind actions, so the FriendTrackerControl activity, for instance, can start and stop the FriendTracker service that runs in the back-ground. The bind action establishes a connection between components, allowing the initiator to execute RPCs de-fined by the service. In our example, FriendTracker binds to the location manager in the system server.

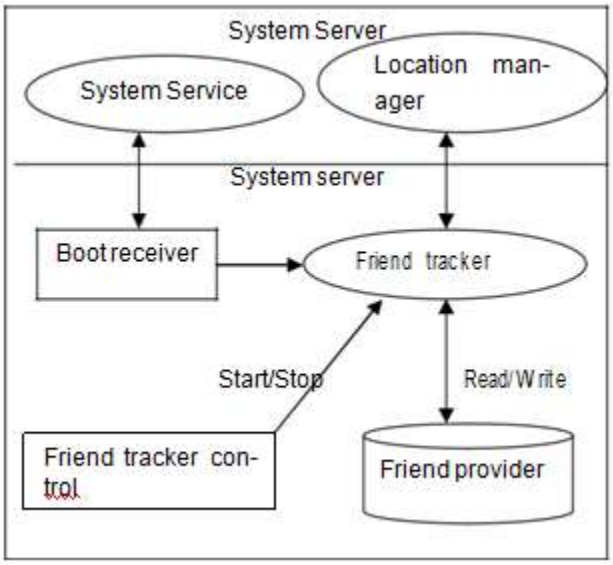

Figure 2: Component Interaction

Once bound, FriendTracker invokes methods to register a callback that provides updates on the phone's location. Note that if a service is currently bound, an explicit "stop" action won't terminate the service until all bound connections are released. Broadcast receiver and content provider components have unique forms of interaction. ICC targeted at a broadcast receiver occurs as an intent sent (broadcast) either explicitly to the component or, more commonly, to an action string the component sub-scribes to. For example, FriendReceiver subscribes to the developer-defined "FRIEND_NEAR" action string. FriendTracker broadcasts an intent to this action string when it determines that the phone is near a friend; the system then starts FriendReceiver and displays a message to the user. Content providers don't use intents—rather, they're ad-dressed via an authority string embedded in a special content Uniform Resource Identifier (URI) of the form content:// <authority>/<table>/[<id>]. Here, <ta-ble> indicates a table in the content provider, and <id> optionally specifies a record in that table. Components use this URI to perform a SQL query on a content provid-er, optionally including WHERE conditions via the query API.

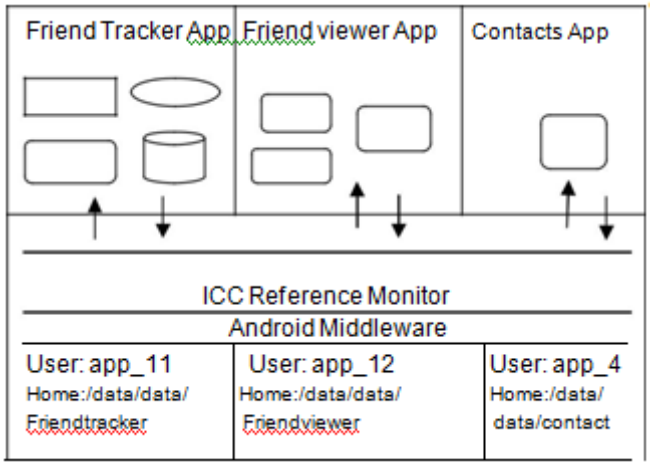

Figure 3: Protection

As Figure 3, Android protects applications and data through a combination of two enforcement mechanisms, one at the system level and the other at the ICC level. ICC mediation defines the core security framework and is this article's focus, but it builds on the guarantees provided by the underlying Linux system. In the general case, each application runs as a unique user identity, which lets An-droid limit the potential damage of programming flaws. A similar vulnerability in Apple's iPhone gave way to the first "jail breaking" technique, which let users replace parts of the underlying system, but would also have enabled a network-based adversary to exploit this flaw (http://security evaluators.com/content/case stu-dies/iphone/index.jsp). ICC isn't limited by user and process boundaries. In fact, all ICC occurs via an I/O con-trol command on a special device node, /dev/binder. Because the file must be world readable and writable for proper operation, the Linux system has no way of mediat-ing ICC. Although user separation is straightforward and easily understood, controlling ICC is much more subtle and warrants careful consideration. As the central point of security enforcement, the Android middleware me-diates all ICC establishments by reasoning about labels assigned to applications and components. A reference monitor provides mandatory access control (MAC) en-forcement of how applications access components. In its simplest form, access to each component is restricted by assigning it an access permission label; this text string need not be unique. Developers assign applications col-lections of permission labels. When a component initiates ICC, the reference monitor looks at the permission labels assigned to its containing application and—if the target component's access permission label is in that collec-tion—allows ICC establishment to proceed. If the label isn't in the collection, establishment is denied even if the components are in the same application.

3. ACCESS PERMISSION LOGIC

The developer assigns permission labels via the XML ma-nifest file that accompanies every application package. In doing so, the developer defines the application's security policy—that is, assigning permission labels to an applica-tion specifies its protection domain, whereas assigning permissions to the components in an application specifies an access policy to protect its resources. Because Andro-id's policy enforcement is mandatory, as opposed to dis-cretionary, all permission labels are set at install time and can't change until the application is reinstalled. How-ever, despite its MAC properties, Android's permission label model only restricts access to components and doesn't currently provide information flow guarantees, such as in domain type enforcement. Security Refine-ments Android's security framework is based on the la-bel-oriented ICC mediation described thus far, but our description is incomplete. Partially out of necessity and partially for convenience, the Google developers who designed Android incorporated several refinements to the basic security model, some of which have subtle side ef-fects and make its overall security difficult to understand. The rest of this section provides an exhaustive list of re-finements we identified as of the v1.0r1 SDK release.

4. BROADCAST INTENT PERMISSIONS

Components aren't the only resource that requires protec-tion. In our FriendTracker example, the FriendTracker service broadcasts an intent to the FRIEND_NEAR action string to indicate the phone is physically near a friend's location. Although this event notification lets the Friend-Viewer application update the user, it potentially informs all installed applications of the phone's proximity. In this case, sending the unprotected intent is a privacy risk. More generally, unprotected intent broadcasts can unin-tentionally leak information to explicitly listening attack-ers. To combat this, the Android API for broadcasting intents optionally allows the developer to specify a per-mission label to restrict access to the intent object. The access permission label assignment to a broadcasted in-tent for example, sendBroadcast (intent, "perm. FRIEND_NEAR") restricts the set of applications that can receive it (in this example, only to applications containing the "perm.FRIEND_NEAR" per-mission label). This lets the developer control how information is disseminated, but this refinement pushes an application's security poli-cy into its source code. The manifest file therefore doesn't give the entire picture of the application's security.

5. CONTENT PROVIDER PERMISSIONS

In our FriendTracker application, the FriendProvider con-tent provider stores friends' geographic coordinates. As a developer, we want our application to be the only one to update the contents but for other applications to be able to read them. Android allows such a security policy by modifying how access permissions are assigned to con-tent providers—instead of using one permission label, the developer can assign both read and write permissions. If the application performing a query with write side effects (INSERT, DELETE, UPDATE) doesn't have the write permission, the query is denied. The separate read and write permissions let the developer distinguish between data users and interactions that affect the data's integrity. Security-aware developers should define separate read and write permissions, even if the distinction isn't imme-diately apparent.

6. PERMISSION PROTECTION LEVELS

Early versions of the Android SDK let developers mark permission as "application" or "system." The default ap-plication level meant that any application requesting the permission label would receive it. Conversely, system permission labels were granted only to applications in-stalled in /data/system (as opposed to /data/app, which is independent of label assignment). The likely reason is that only system applications should be able to perform operations such as interfacing directly with the telephony API. The v0.9r1 SDK (August 2008) extended the early model into four protection levels for permission labels, with the meta information specified in the manifest of the package defining the permission. "Normal" permissions act like the old application permissions and are granted to any application that requests them in its manifest; "dan-gerous" permissions are granted only after user confirma-tion. Similar to security checks in popular desktop operat-ing systems such as Microsoft Vista's user account control (UAC), when an application is installed, the user sees a screen listing short descriptions of requested dangerous permissions along with OK and Cancel buttons. Here, the user has the opportunity to accept all permission requests or deny the installation. "Signature" permissions are granted only to applications signed by the same develop-er key as the package defining the permission (applica-tion signing became man-datory in the v0.9r1 SDK). Fi-nally, "signature or system" permissions act like signa-ture permissions but exist for legacy compatibility with the older system permission type. The new permission protection levels provide a means of controlling how de-velopers assign permission labels. Signature permissions ensure that only the framework developer can use the specific functionality (only Google applications can di-rectly interface the telephony API, for example). Danger-ous permissions give the end user some say in the per-mission granting process—for example, FriendTracker defines the permission label associated with the FRIEND_NEAR intent broadcast as dangerous. However, the permission protection levels express only trivial granting policies. A third-party application still doesn't have much control if it wants another developer to use the permission label. Making a permission "dangerous" helps, but it depends on the user understanding the secu-rity implications.

7. SECURITY RESPONSIBILITIES FOR DE-VELOPERS

Developers writing for Android need to consider how their code will keep users safe as well as how to deal with constrained memory, processing and battery power. De-velopers must protect any data users input into the device with their application, and not allow malware to access the application's special permissions or privileges. How to achieve this is partly related to which features of the platform an application uses, as well as any extensions to the platform an Android distribution has made. One of the trickiest big-picture things to understand about An-droid is that every application runs with a different UID. Typically on a desktop every user has a single UID and running any application launches runs that program as the users UID. On Android the system gives every appli-cation, rather than every person, its own UID. For exam-ple, when launching a new program (say by starting an Activity), the new process isn't going to run as the launcher but with its own identity. It's important that if a program is launched with bad parameters the developer of that application has ensured it won't harm the system or do something the phone's user didn't intend. Any pro-gram can ask Activity Manager to launch almost any oth-er application, which runs with the application's UID. Fortunately, the untrusted entry points to your applica-tion are limited to the particular platform features you choose to use and are secured in a consistent way. Andro-id applications don't have a simple main function that always gets called when they start. Instead,

their initial entry points are based on registering Activites, Services, Broadcast Receivers or Content Providers with the sys-tem. After a brief refresher on Android Permissions and Intents we will cover securely using each of these fea-tures. Android requires developers to sign their code. Android code signing usually uses self-signed certificates, which developers can generate without anyone else's as-sistance or permission. One reason for code signing is to allow developers to update their application without creating complicated interfaces and permissions. Applica-tions signed with the same key (and therefore by the same developer) can ask to run with the same UID. This allows developers to upgrade or patch their software easi-ly, including copying data from existing versions. The signing is different than normal Jar or Authenticode sign-ing however, as the actual identity of the developer isn't necessarily being validated by a third party to the de-vice's user. Developers earn a good reputation by making good products; their certificates prove authorship of their works. Developers aren't trusted just because they paid a little money to some authority. This approach is novel, and may well succeed, but it wouldn't be technically dif-ficult to add trusted signer rules or warnings to an An-droid distribution if it proved desirable.

8. ANDROID PERMISSIONS REVIEW

Applications need approval to do things their owner might object to, like sending SMS messages, using the camera or accessing the owner's contact database. Andro-id uses manifest permissions to track what the user al-lows applications to do. An application's permission needs are expressed in its AndroidManifest.xml and the user agrees to them upon install. When installing new software, users have a chance to think about what they are doing and to decide to trust software based on re-views, the developer's reputation, and the permissions required. Deciding up front allows them to focus on their goals rather than on security while using applications. Permissions are sometimes called ―manifest permissions― or ―Android permissions― to distinguish them from file permissions. To be useful, permissions must be associated with some goal that the user understands. For example, an application needs the READ_CONTACTS permission to read the user's address book. A contact manager app needs the READ_CONTACTS permission, but a block stacking game shouldn't Keeping the model simple, it's possible to secure the use of all the different Android in-ter-process communication (IPC) mechanisms with just a single kind of permission. Starting Activities, starting or connecting to Services, accessing Content Providers, sending and receiving broadcast Intents, and invoking Binder interfaces can all require the same permission. Therefore users don't need to understand more than ―My new contact manager needs to read contacts. Once in-stalled, an application's permissions can't be changed. By minimizing the permissions an application uses it mini-mizes the consequences of potential security flaws in the application and makes users feel better about installing it. When installing an application, users see requested per-missions in a dialog similar to the one shown in Installing software is always a risk and users will shy away from software they don't know, especially if it requires a lot of permissions. From a developer's perspective permissions are just strings associated with a program and it's UID. You can use the Context class' check Permission (String permission, int pid, int uid) method to programmatically check if a process (and the corresponding UID) has a par-ticular permission like READ_CONTACTS. This is just one of many ways permissions are exposed by the run-time to developers. The user view of permissions is sim-ple and consistent; the idiom for enforcement by develop-ers is consistent too but adjusts a little for each IPC me-chanism.

```
<permi
    ssio
    n
    and
    ro-
id:name="com.isecpartners.android.ACCESS_SHOPPIN
G _LIST"
 android:description="@string/access_perm_des
 c" android:protectionLevel="normal"
 android:label="@string/access_perm_label">
</permission>
```

Manifest permissions like the one above have a few key properties. Two text descriptions are required: a short text label, and a longer description used on installation. An icon for the permission can also be provided. All permis-sions must also have a name which is globally unique. The name is the identifier used by programmers for the permission and is the first parameter to Context. Check Permission. Permissions also have a protection level (called protection Level as shown above).

There are only four protection levels for permissions.

Normal	Permissions for application features whose consequences are minor like VI-BRATE which lets applications vibrate the device. Suitable for granting rights not generally of keen interest to users, users can review but may not be explicit-ly warned.
Dangerous	Permissions like WRITE_SETTINGS or SEND_SMS are dangerous as they could be used to reconfigure the device or incur tolls. Use this level to mark permissions users will be interested in or potentially surprised by. Android will warn users about the need for these permissions on install.
Signature	These permissions can only be granted to other applications signed with the same key as this program. This allows secure

	coordination without publishing a public interface.
Signature Or System	Similar to Signature except that programs on the system image also qualify for access. This allows programs on custom Android systems to also get the permis-sion. This protection is to help integrate system builds and won't typically be needed by developers.

If you try to use an interface which you don't have per-missions for you will probably receive a Security Excep-tion. You may also see an error message logged indicating which permission you need to enable. In addition to read-ing and writing data, much permission allows applica-tions to call upon system services or start Activities with security sensitive results. For example, with the right permission a video game can take full control of the screen and obscure the status bar, or a dialer can cause the phone to dial a number without prompting the user.

Intents

Intents are an Android-specific mechanism for moving data between Android processes and are at the core of much of Android's IPC. To allow their communication role Intents can be sent over Binder interfaces (since they implement the Parcelable interface). Almost all Android IPC is actually implemented through Binder, although most of the time this is hidden from us with higher level abstractions.

Intent review

Intents are used in a number of ways by Android:

- To start an Activity – coordinating with other programs like browsing a web page Using Con-text's startActivity() method.
- As broadcasts to inform interested programs of changes or events
  - Using Context's sendBroadcast(), sendStickyBroadcast(), and sendOrderedBroadcast() family of methods.
- As a way to start, stop or communicate with background Services
  - Using Context's startService(), stop-Service(), and bindService() methods
- To access data through ContentProviders, such as the user's contacts.
  - Using Context's getContentResolver() or Activities managedQuery()

  As call backs to handle events, like returning re-sults or errors asynchronously with PendingIn-tents provided by clients to servers through their Binder interfaces

Intents have a lot of implementation details but the basic idea is that they represent a blob of serialized data that can be moved between programs to get something done. Intents usually have an action, which is a string like ―android.intent.action.VIEW― that identifies some particu-lar goal, and often some data in the form of an Uri. Intents can have optional attributes like a list of Categories, an explicit type (independent of what the data's type is), a component, bit flags and a set of name value pairs called ―Extras―.

Activities

Activities allow applications to call each other, reusing each other's features, and allowing for replacements or improvement of individual system pieces whenever the user likes. Activities are often run in their own process, running as their own UID, and so don't have access to the caller's data aside from any data provided in the Intent used to call the Activity. The Activity Manager will likely decide to start the web browser to handle it, because the web browser has an Activity registered with a matching intent-filter.

```
Intent i = new Intent(Intent.ACTION_VIEW);
i.setData(Uri.parse("http://www.isecpartners.com"));
this.startActivity(i);
Intent i = new Intent("Cat-Farm Aardvark Pidgen");
    // The browser's intent filter isn't interested in this Uri scheme
    i.setData(Uri.parse("marshmaellow:potatochip?"));
    // The browser activity is going to get it anyway!
    i.setComponent(new Component-Name("com.android.browser", "com.android.browser.BrowserActivity"));
    this.startActivity(i);
```

If you run this code you will see the browser Activity starts, but the browser is robust and aside from being started just ignores this weird Intent.

This example Activity clears the current shopping list and gives the user an empty list to start editing. Because clear-ing is destructive, and happens without user confirma-tion, this Activity must be restricted to trustworthy cal-lers. The description of that permission also explains to users that granting it gives an applications the ability to read and change shopping lists. We protect this Activity with the following entry:

```
<activity
 android:name=".BlankShoppingList"
 andro-
id:permission="com.isecpartners.ACCESS_SHOPPING_LIST">
  <intent-
  filter>
  <action
    andro-
id:name="com.isecpartners.shopping.CLEAR_LIST" />
</intent-filter> </activity>
```

Developers need to be careful not just when implement-ing Activities but when starting them too. Avoid putting data into Intents used to start Activities that would be of interest

to an attacker. A password, sensitive Binder or message contents would be prime examples of data not to include!

9. BROADCASTS

Broadcasts are way applications and system components can communicate securely and efficiently. The messages are sent as Intents, and the system handles dispatching them, including starting receivers, and enforcing permis-sions. Receiving broadcast intents, Intents can be broad-cast to Broadcast Receivers, allowing messaging between applications. As with Activities, a broadcast sender can send a receiver an Intent that would not pass its Intent Filter just by specifying the target receiver component explicitly. Receivers must be robust against unexpected Intents or bad data. As always in secure IPC program-ming, programs must carefully validate their input. Broadcast Recievers are registered in the AndroidManif-est.xml with the <receiver> tag. By default they are not exported, but can be exported easily by adding an <in-tent-filter> tag (including an empty one) or by setting the attribute android: exported="true". Once exported, re-ceivers can be called by other Programs. Like Activities, the Intents that Broadcast Receivers get may not match the Intent Filter they registered. To restrict who can send your receiver Intent use the android: permission attribute on the receiver tag to specify a manifest permission. When permission is specified on a receiver, Activity Manager validates that the sender has the specified per-mission before delivering the Intent. Permissions are the right way to ensure your receivers only gets Intents from appropriate senders, but permissions don't otherwise affect the properties of the Intent that will be received.

10. SERVICES

Services are long running background processes provided by Android to allow for background tasks like music playing or running of a game server. They can be started with Intent and optionally communicated with over a Binder interface by calling Context's bind Service () me-thod. Services are similar to Broadcast Receivers and Ac-tivities in that they can be started independently of their Intent Filters by specifying a Component (if they are ex-ported). Services can also be secured by adding a permis-sion check to their <service> tag in the AndroidManif-est.xml. The long lasting connections provided by bind Service () create a fast IPC channel based on a Binder in-terface (see below). Binder interfaces can check permis-sions on their caller, allowing them to enforce more than one permission at a time or different permissions on dif-ferent requests. Services therefore provide lots of ways to make sure the caller is trusted, similar to Activities, Broadcast Receivers and Binder interfaces. Calling a Ser-vice is slightly trickier. This hardly matters for scheduling MP3s to play, but if you need to make sensitive calls into a Service, like storing passwords or private messages, you'll need to validate the Service you're connect to is the correct one and not some hostile program that shouldn't have access to the information you provide. If you know the exact component you are trying to connect to, you can specify that explicitly in the Intent you use to connect. Alternately, you can verify it against the name provided to your SeviceConnection's onServiceCon-nected(ComponentName name, IBinder service) imple-mentation. That isn't very dynamic though and doesn't let users choose to replace the service provider.

To dynamically allow users to add replacement services, and then authorize them by means of checking for the permission they declared and were granted by the user we can use the component name's package as a way to validate permission. We received the name of the imple-menting component when we receive the onServiceCon-nected() callback, and this name is associated with the applications rights. This is perhaps harder to explain than to do and comes down to only a single line of code!

res = getPackageManager(). checkPermission (permTo-Check, name.getPackageName());

11. CONCLUSION

Android applications have their own identity enforced by the system. Applications can communicate with each oth-er using system provided mechanisms like files, Activi-ties, Services, BroadcastReceivers, and ContentProviders. If you use one of these mechanisms you need to be sure you are talking to the right entity — you can usually vali-date it by knowing the permission associated with the right you are exercising. If you are exposing your applica-tion for programmatic access by others, make sure you enforce permissions so that unauthorized applications can't get the user's private data or abuse your program. Make your applications security as simple and clear as possible. When communicating with other programs, think clearly about how much you can trust your input, and validate the identity of services you call. Before ship-ping, think about how you would patch a problem with your application.

REFERENCES

[1] http://www.javatpoint.com/android-tutorial
[2] https://www.codementor.io/android/tutorial
[3] Android Software Misuse before It Happens, tech. report NAS-TR-0094-2008, Network and Security Re search Ctr., Dept. Computer Science and Eng., Pennsyl vania State Univ., Nov. 2008.
[4] Chu, E. (2008, August 28). Android Market: a user- driven content distribution system. Re trieved August 30, 2008, from Android Devel oper's Blog.
[5] Google Inc. (2008, August 29). Security and Permis sions in Android. Retrieved August 30, 2008, from An droid - An Open Handset Al liance Project.
[6] Burns, Jesse (2009, October) developing secure mobile applications for android.