The Android Security Model

Android is a Linux platform programmed with Java and enhanced with its own security mechanisms tuned for a mobile environment[4]. 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 resulting 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, Android applications are individually siloed from each other. 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,[5] and sharing data between applications must be done explicitly. The Android GUI environment has some novel security features that help support this isolation.

Mobile platforms are growing in importance, and have complex requirements[6] including regulatory compliance[7]. Android 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 security or granting the application additional rights.

Android *permissions* are rights given to applications 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 application is used to protect its data and developers need to be explicit about sharing data with other applications[8]. Applications can entertain users with graphics, play music, and launch other programs without special permissions.

Malicious software is an unfortunate reality on popular platforms, and through its features Android tries to minimize the impact of malware. However, even unprivileged malware that gets installed on an Android device (perhaps by pretending to be a useful application) can still temporarily wreck the user's experience[9]. 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 calls (which may incur tolls),
- disclosing the user's private data, or
- destroying address books, email, etc.

Generally a user's response to annoying, buggy or malicious software is simply to uninstall it. If the

[4] Cylon heritage is perhaps the most problematic security aspect of Android's lineage.
[5] Hence getting code execution on most Android application doesn't fully compromise the device!
[6] Like the need to support emergency calling, even on screen-locked or out of service phones.
[7] Government regulations vary widely so the platform needs a lot of flexibility.
[8] This paper will cover how to safely do such sharing. Android offers several flexible mechanisms to choose from.
[9] For example they could play loud noises, interrupt the user or run down the battery.

software is disrupting the phone enough that the user can't uninstall it, they can reboot the phone (optionally in safe mode[10], which stops non-system code from running) and then remove the software before it has a chance to run again.

Developer's Responsibilities

As a developer writing for Android, you will need to consider how you will keep users safe as well as how to deal with constrained memory, processing and battery power. You must protect any data users input into their device with your application, and not allow malware to access the application's special permissions. How you achieve this is partly related to which features of the platform you use.

One of the trickiest big-picture things to understand about Android 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 application, rather than every person, its own UID. For example, 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[11] is launched with bad parameters[12] the developer of that application has ensured it won't harm the system or do something the phone's user didn't intend. Any program can ask Activity Manager to launch almost any other application, which runs with the application's UID.

Fortunately, the untrusted entry points to your application are limited to the particular platform features you choose to use and are secured in a consistent way. Android 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*, *BroadcastReceivers* or *ContentProviders* with the system. After a brief refresher on Android *Permissions* and *Intents* we will cover securely using each of these features.

Android requires developers to sign their code. Android code signing usually uses self-signed certificates, which developers can generate without anyone else's assistance or permission. One reason for code signing is to allow developers to update their application without creating complicated interfaces and permissions. Applications 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 easily, including copying data from existing versions. The signing is different than normal Jar or Authenticode[13] signing however, as the actual identity of the developer isn't necessarily being validated by a third party to the device's user[14]. 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

[10] How to enter safe mode is device specific. One prototype platform required holding the Menu button while booting.
[11] By programs what is usually meant is an Activity or a Service. This will be covered in more detail later.
[12] Android applications don't usually have parameters — there isn't even a UI to specify them in the Home application.

[13]Authenticode is a popular Microsoft code signing technology documented here: http://msdn.microsoft.com/en-us/library/ms537364(VS.85).aspx. They usually identify a company and are validated by a certificate authority.
[14] Some mobile platforms use code signing as a way to control and track developers. Android's self-signing system just makes software easier to maintain and use. Certificate chains and the standards third parties used to validate identities are too complex for most users to understand, so Android uses a more social approach to developer identity.

little money to some authority. This approach is novel, and may well succeed, but it wouldn't be technically difficult to add trusted signer rules or warnings to an Android distribution if it proved desirable.

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. Android uses *manifest permissions* to track what the user allows applications to do. An application's permission needs are expressed in its AndroidManifest.xml and the user agrees to them upon install[15]. When installing new software, users have a chance to think about what they are doing and to decide to trust software based on reviews, 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[16] permission to read the user's address book. A contact manager app needs the READ_CONTACTS permission, but a block stacking game shouldn't[17]. Keeping the model simple, it's possible to secure the use of all the different Android inter-process communication (IPC) mechanisms with just a single kind of permission. Starting *Activities*, starting or connecting to *Services*, accessing *ContentProviders*, 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".

Developer Tip: Users won't understand how their device works, so keep permissions simple and avoid technical terms like *Binder*, *Activity* or *Intent* when describing permissions to users.

Once installed, an application's permissions can't be changed. By minimizing the permissions an application uses it minimizes the consequences of potential security flaws in the application and makes users feel better about installing it. When installing an application, users see requested permissions in a dialog similar[18] to the one shown in Figure 1. 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.

[15] The same install warnings are used for side loaded and Market applications. Applications installed with adb don't show warnings but that mechanism is only used by developers. The future may bring more installers than these three.
[16] "android.permission.READ_CONTACTS" is the permission's full text.
[17] If the game vibrates the phone, or connects to a high score Internet server it might need VIBRATE or INTERNET *permission*.
[18] The dialog lists permissions; the installation dialog gives a bit more information and the option to install as well.

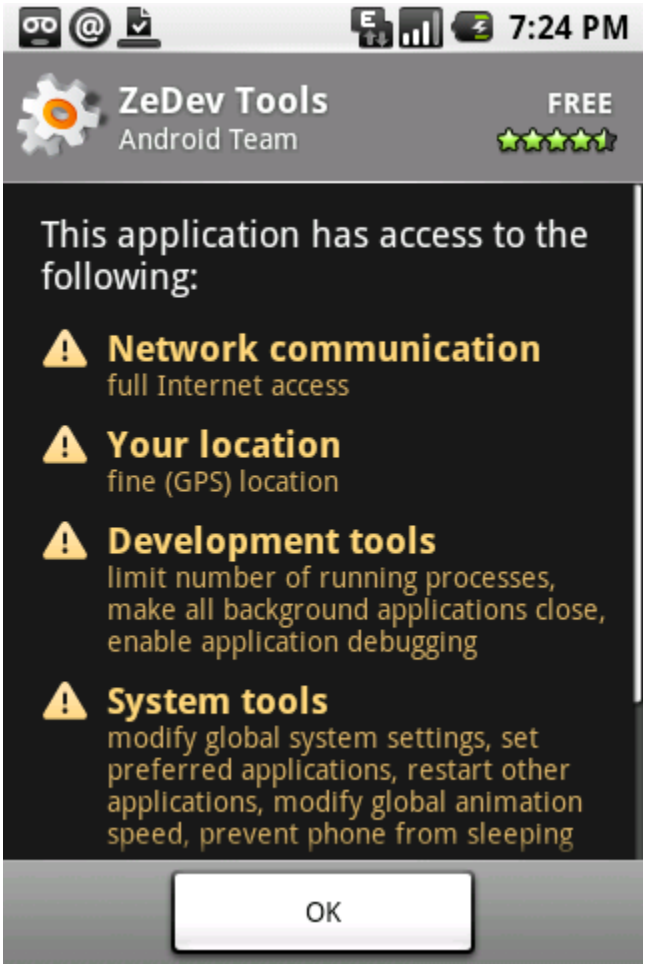

Figure 1 Dialog showing Application permissions to users.

(Chu, 2008)

From a developer's perspective permissions are just strings associated with a program and its UID. You can use the *Context* class' checkPermission(String permission, int pid, int uid) method to programmatically check if a process (and the corresponding UID) has a particular permission like READ_CONTACTS[19]. This is just one of many ways permissions are exposed by the runtime to developers. The user view of permissions is simple and consistent; the idiom for enforcement by developers is consistent too but adjusts a little for each IPC mechanism.

[19] You would pass the fully qualified value of READ_CONTACTS, which is "android.permission.READ_CONTACTS".

Figure 2 shows an example permission definition. Note that the description and label are resources to aid in localizing the application.

```
<permission
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:name="com.isecpartners.android.ACCESS_SHOPPING_LIST"
      android:description="@string/access_perm_desc"
      android:protectionLevel="normal"
      android:label="@string/access_perm_label">
</permission>
```

Figure 2 Custom permission definition in an AndroidManifest.xml file.

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 (but isn't in the example above). All permissions 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.checkPermission*. Permissions also have a protection level (called *protectionLevel* as shown above).

There are only four protection levels for permissions[20]:

Normal	Permissions for application features whose consequences are minor like VIBRATE which lets applications vibrate the device. Suitable for granting rights not generally of keen interest to users, users can review but may not be explicitly 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.
SignatureOrSystem	Similar to Signature except that programs on the system[21] image also qualify for access. This allows programs on custom Android systems to also get the permission. This protection is to help integrate system builds and won't typically be needed by developers.

Figure 3 Android manifest permission protection levels

[20] See http://code.google.com/android/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel or search for "Android Manifest Permission protectionLevel" for platform documentation.
[21] Of course custom system builds can do whatever they like; indeed you ask the system when checking permissions – but *SignatureOrSystem* level permissions intend for third party integration and so protects more stable interfaces then *Signature*.

If you try to use an interface which you don't have permissions for you will probably receive a *SecurityException*. You may also see an error message logged indicating which permission you need to enable. If your application enforces permissions you should consider logging an error on failure so that developers calling your application can more easily diagnose their problems. Sometimes (aside from the lack of anything happening) permission failures are silent. The platform itself neither alerts users when permission checks fail, nor allows granting of permissions to applications after installation.

> Developer Tip: Your application might be used by people who don't speak your language. Be sure to internationalize the label and description properties of any new permission you create. Have someone both technical and fluent in the target languages review to ensure translations are accurate.

In addition to reading and writing data, many permissions allow applications 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.

Creating New Manifest Permissions

Applications can define their own permissions if they intend other applications to have programmatic access to them. Using a manifest permission allows the end user to decide which programs get access, rather than having the developer just assume access is acceptable. For example, an application that manages a shopping list application could define a permission named "com.isecpartners.ACCESS_SHOPPING_LIST" (ACCESS_SHOPPING_LIST for short). If the application defines an exclusive *ShoppingList* object then there is now precisely one instance of *ShoppingList* and the ACCESS_SHOPPING_LIST permission is needed to access it. The permission would be required for callers trying to see or update the shopping list. Done correctly, only the programs that declare they use this permission could access the list, giving the user a chance to either consent or prevent inappropriate access. When defining permissions keep them clear and simple, make sure you actually have a service or some data you want to expose not to just interactive users but to other programs.

Adding permissions should be avoided using a little cleverness whenever possible. For example you could define an *Activity* that added a new item to the shopping list. When an application called *startActivity* and provided an *Intent* to add a new shopping list item, the *Activity* could display the data provided and ask for confirmation from the user instead of requiring permission enforcement. This keeps the system simple for users and saves you development effort. A requirement for *Activities* that immediately altered the list upon starting would make the permission approach necessary.

Creating custom permissions can also help you minimize the permission requirements for applications that use your program programmatically. For example, if an application needs permissions to both send SMS messages and access the users location,[22] it could define a new permission like "SEND_LOCATION_MESSAGE". This permission is all that applications using your service would need, making their installation simpler and clearer to the user.

[22] Location determination can require multiple permissions depending on which scheme the particular phone uses.

Intents

Intents are an Android-specific mechanism for moving data between Android processes and are at the core of much of Android's IPC. They don't enforce security policy themselves, but are usually the messenger that crosses the actual system security boundaries. 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 Context'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(), stopService(), 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 results or errors asynchronously with *PendingIntents* provided by clients to servers through their *Binder* interfaces

Intents have a lot of implementation details[23], 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 particular goal, and often some data in the form of a *Uri*[24]. *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". Generally APIs that take Intents can be restricted with manifest permissions. This allows you to create *Activities*, *BroadcastReceivers*, *ContentProviders* or *Services* that can only be accessed by applications the user has granted these rights to.

Intent Filters

Depending on how they are sent, *Intents* may be dispatched by the Android Activity Manager. For example an *Intent* can be used to start an *Activity* by calling Context.startActivity(Intent intent). The *Activity* to start is found by Android's Activity Manager by matching the passed in *Intent* against the *IntentFilters* registered for all *Activities* on the system and looking for the best match. *Intents* can

[23] Indeed the documentation for just the Intent class is far longer than this document.
[24] An instance of the android.net.Uri class.

override the *IntentFilter* match Activity Manager uses however. Any "exported"[25] *Activity* can be started with any *Intent* values for action, data, category, extras, etc. The *IntentFilter* is not a security boundary from the perspective of an *Intent* receiver. In the case of starting an *Activity,* the caller decides what component is started and creates the *Intent* the receiver then gets. The caller can choose to ask Activity Manger for help with figuring out where the *Intent* should go, but doesn't have to.

Intent recipients like *Activities*, *Services* and *BroadcastReceivers* need to handle potentially hostile callers, and an *IntentFilter* doesn't filter a malicious *Intent*[26]. *IntentFilters* help the system figure out the right handler for a particular *Intent*, but doesn't constitute an input filtering system. Because *IntentFilters* are not a security boundary they cannot be associated with permissions. While starting an *Activity* is the example I used to illustrate this above, you will see in the following sections that no IPC mechanisms using *IntentFilters* can rely on them for input validation.

Categories can be added to *Intents*, making the system more selective about what code the *Intent* will be handled by. Categories can also be added to *IntentFilters* to permit *Intents* to pass, effectively declaring that the filtered object supports the restrictions of the Category. This is useful whenever you are sending an *Intent* whose recipient is determined by Android, like when starting an *Activity* or broadcasting an *Intent*.

Developer Tip: When starting or broadcasting *Intents* where an *IntentFilter* is used by the system to determine the recipients, remember to add as many categories as correctly apply to the *Intent*. Categories often require promises about the safety of dispatching an *Intent*, helping stop the *Intent* from having unintended consequences.

Adding a category to an *Intent* restricts what it can do. For example an *IntentFilter* that has the "android.intent.category.BROWSABLE" category is indicating it is safe to be called from the web browser. Carefully consider why *Intents* would have a category and consider if you have met the terms of that contract before placing a category in an *IntentFilter*. Future categories could (for example) indicate an *Intent* was from a remote machine or un-trusted source but because this category won't match the *IntentFilters* we put on our applications today, the system won't deliver them to our programs. This keeps our applications from behaving unexpectedly when the operating environment changes in the future.

[25] An activity is automatically exported if it has an IntentFilter specified, it can also be exported explicitly by adding the attribute android:exported="true"

[26] You can enforce a permission check for anyone trying to start an Activity however. This is explained in the section on Activities.

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[27] 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*.

> Developer Tip: The easiest way to make *Activities* safe is just to confirm any changes or actions clearly with the user. If starting your *Activity* with an *Intent*[28] could result in harm or confusion you need to require a permission to start it.

Activities cannot rely on *IntentFilters* (the <intent-filter> tag in AndroidManifest.xml) to stop callers from passing them badly configured *Intents*. Misunderstanding this is actually a relatively common source of bugs. On the other hand, *Activity* implementers can rely on permission checks as a security mechanism. Setting the *android:permission* attribute in an <activity> declaration will prevent programs lacking the specified permission from directly starting that *Activity*. Specifying a manifest *permission* that callers must have doesn't make the system enforce an *intent-filter* or clean intents of unexpected values so always validate your input.

This code shows starting an *Activity* with an *Intent*. 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);
```

Figure 4 Starting an Activity based on its IntentFilter

The following code demonstrates forcing the web browser's *Activity* to handle and Intent with an *action* and *data* setting that aren't permitted by its *intent-filter*:

[27] *Activities* implemented by the caller's program may share a process, depending on configuration.

[28] An *Intent* received by an *Activity* is essentially untrusted input and must be carefully and correctly validated.

```
// The browser's intent filter isn't interested in this action
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 ComponentName("com.android.browser",
                                    "com.android.browser.BrowserActivity"));
this.startActivity(i);
```

Figure 5 Starting an Activity regardless of its IntentFilter

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*.

Figure 6 gives an example AndroidManifest entry that declares an *Activity* called ".BlankShoppingList". This example *Activity* clears the current shopping list and gives the user an empty list to start editing. Because clearing is destructive, and happens without user confirmation, this *Activity* must be restricted to trustworthy callers. The "com.isecpartners.ACCESS_SHOPPING_LIST" *permission* allows programs to delete or add items to the shopping list, so programs with that *permission* are already trusted not to wreck our list. 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"
      android:permission="com.isecpartners.ACCESS_SHOPPING_LIST">
      <intent-filter>
I           <action
                  android:name="com.isecpartners.shopping.CLEAR_LIST" />
      </intent-filter>
</activity>
```

Figure 6 Activity declaration requiring a caller permission.

When defining *Activities*, those defined without an *intent-filter* or an *android:exported* attribute are not publicly accessible, that is, other applications can't start them with Context.startActivity(Intent intent). These *Activities* are the safest of all, but other applications won't be able to reuse your application's *Activities*.

Developers need to be careful not just when implementing *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! For example Malware could register a higher priority *IntentFilter* and end up getting the user's sensitive data sent to their *Activity* instead.

When starting an *Activity* if you know the component you intend to have started, you can specify that in the *Intent* by calling its setComponent() method. This prevents the system from starting some other *Activity* in response to your *Intent*. Even in this situation it is still unsafe[29] to pass sensitive arguments in this *Intent*. You can think of the *Intent* used to start an *Activity* as being like the command line arguments of a program, which usually shouldn't include secrets either.

Developer Tip: Don't put sensitive data into *Intents* used to start *Activities*. Callers can't easily require Manifest permissions of the *Activities* they start, and so your data might be exposed.

---

[29] For example processes with the GET_TASKS permission are able to see "ActivityManager.RecentTaskInformation" which includes the "baseIntent" used to start *Activities*.