MOBILE APPLICATION SECURITY ON ANDROID

Context on Android security

Black Hat 2009

This document describes the Android Security Model and provides the context required to understand the tools and techniques that will be demonstrated by Jesse Burns at his Black Hat USA talk, currently scheduled for July 30th, 2009. Updates are available on the iSEC Partners website. A developer's guild to android is also maintained at that site, by the author at http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf.

Prepared for Black Hat USA by: Jesse Burns

Questions: AndroidSecurityPaper@isecpartners.com

Updates: https://www.isecpartners.com

Date: June, 2009

Version 0.1

Introduction

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. These extensions can help or could interfere with security, being able to analyze a distribution of Android is therefore an important step in protecting information on that system. This document takes the reader through the security model of Android, including many of the key security mechanisms and how they protect resources. This background information is critical to being able to understand the tools Jesse will be presenting at Black Hat, and the type of information you can glean from the tools, and from any running Android distribution or application you wish to analyze.

Before reading this paper you should already be familiar with Android's basic architecture and major abstractions including: *Intents*, *Activities*, *BroadcastReceivers*, *Services*, *ContentProviders* and *Binder*[1]. If you haven't yet encountered most of these platform features you might want to start with the Notepad[2] tutorial. As Android is open source you should also have this code available to you. Both the java and C code is critical for understanding how Android works, and is far more detailed than any of the platform documentation. As will be discussed in the Black Hat talk, the difference between what the documentation and the source say is often critical to understanding the systems security.

The Android Security Model

Android is a Linux platform programmed with Java and enhanced with its own security mechanisms tuned for a mobile environment[3]. 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,[4] 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[5] including regulatory compliance[6]. 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.

[1] Binders are a bit of an advanced topic. We will introduce them in the Services section.
[2] http://code.google.com/android/intro/tutorial.html
[3] Cylon heritage is perhaps the most problematic security aspect of Android's lineage.
[4] Hence getting code execution on most Android application doesn't fully compromise the device!
[5] Like the need to support emergency calling, even on screen-locked or out of service phones.
[6] Government regulations vary widely so the platform needs a lot of flexibility.

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[7]. 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[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 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 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.

Security Responsibilities of Developers

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. Developers 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 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[10] is launched with bad parameters[11] 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.

[7] This paper will cover how to safely do such sharing. Android offers several flexible mechanisms to choose from.
[8] For example they could play loud noises, interrupt the user or run down the battery.
[9] How to enter safe mode is device specific. One prototype platform required holding the Menu button while booting.
[10] By programs what is usually meant is an Activity or a Service. This will be covered in more detail later.
[11] Android applications don't usually have parameters — there isn't even a UI to specify them in the Home application.

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[12] signing however, as the actual identity of the developer isn't necessarily being validated by a third party to the device's user[13]. 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 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[14]. 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[15] permission to read the user's address book. A contact manager app needs the READ_CONTACTS permission, but a block stacking game shouldn't[16]. 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

[12]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.

[13] 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.

[14] 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.

[15] "android.permission.READ_CONTACTS" is the permission's full text.

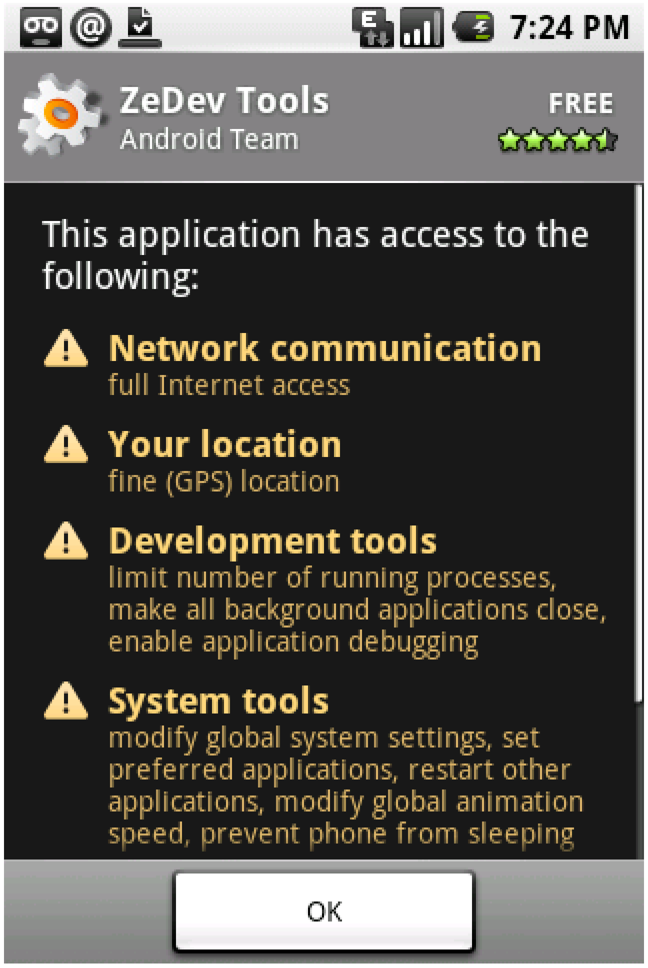
[16] If the game vibrates the phone, or connects to a high score Internet server it might need VIBRATE or INTERNET *permission*.

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[17] 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.

[17] The dialog lists permissions; the installation dialog gives a bit more information and the option to install as well.



This application has access to the following:

Network communication
full Internet access

Your location
fine (GPS) location

Development tools
limit number of running processes, make all background applications close, enable application debugging

System tools
modify global system settings, set preferred applications, restart other applications, modify global animation speed, prevent phone from sleeping

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[18]. 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.

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"
```

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

```
            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[19]:

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[20] 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

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

[19] See http://code.google.com/android/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel or search for "Android Manifest Permission protectionLevel" for platform documentation.
[20] 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*.

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,[21] it could define a new permission like

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

“SEND_LOCATION_MESSAGE”. This permission is all that applications using your service would need, making their installation simpler and clearer to 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. 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[22], 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*[23]. *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

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

override the *IntentFilter* match Activity Manager uses however. Any "exported"[24] *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*[25]. *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.

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

[25] 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[26] 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*[27] 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*:

```
// The browser's intent filter isn't interested in this action
```

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

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

```
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[28] 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.

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

Broadcasts

Broadcasts are a 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 permissions.

Receiving Broadcast Intents

Intents can be broadcast to *BroadcastReceivers*, allowing messaging between applications. By registering a *BroadcastReceiver* in your application's AndroidManifest.xml you can have your application's receiver class started and called whenever someone sends you a broadcast. Activity Manager uses the *IntentFilters* applications register to figure out which program to use for a given broadcast. As we discussed in the sections on *IntentFilters* and *Activity permissions*, filters are not a security mechanism and can't be relied upon[29] by *Intent* recipients. As with *Activities*, a broadcast sender can send a receiver an *Intent* that would not pass its *IntentFilter* just by specifying the target receiver component explicitly[30]. Receivers must be robust against unexpected *Intents* or bad data. As always in secure IPC programming, programs must carefully validate their input.

BroadcastRecievers are registered in the AndroidManifest.xml with the <receiver> tag. By default they are not exported, but can be exported easily by adding an <intent-filter> tag (including an empty one) or by setting the attribute *android:exported="true"*. Once exported, receivers can be called by other programs. Like *Activities*, the *Intents* that *BroadcastReceivers* get may not match the *IntentFilter* they registered. To restrict who can send your receiver an *Intent* use the *android:permission* attribute on the receiver tag to specify a manifest *permission*. When a *permission* is specified on a receiver, Activity Manager validates that the sender has the specified *permission* 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.

Safely Sending Broadcast Intents

When sending a broadcast, developers include some information or sometimes even a sensitive object like a *Binder*. If the data being sent is sensitive they will need to be careful who it gets sent to. The simplest way to protect this while leaving the system very dynamic is to require the receiver to have a *permission*. By passing a manifest permission name (receiverPermission is the parameter name) to one of Context's broadcastIntent() family of methods you can require recipients to have that permission. This lets developers control which applications can receive the *Intent*. Broadcasts are special in being able to very easily require permissions of recipients; when you need to send sensitive messages prefer this IPC mechanism.

---

[29] *IntentFilters* can sometimes help *Intent* sender safety by allowing the sending of an *Intent* that is qualified by a category. Receivers that don't meet the category requirements won't receive it, unless the sender forces delivery by specifying a component. Senders adding categories to narrow deliver therefore shouldn't specify a Component.

[30] See the examples given for *Activities* in the Activity Permissions section. These examples can be applied to broadcasts by using *sendBroadcast()* rather than *startActivity()* and adjusting the components appropriately for your test classes.

For example, an SMS application might want to notify other interested applications of an SMS it received by broadcasting an *Intent*. It can limit the receivers to those applications with the RECEIVE_SMS *permission* by specifying this as a required *permission* when sending. If an application sent the contents of an SMS message on to other applications by broadcasting an *Intent* without asserting that the receiver must have the RECEIVE_SMS *permission* then unprivileged applications could register to receive that *Intent* — creating a security hole. Applications can register to receive *Intents* without any special privileges. Therefore, applications must require that potential receivers have some relevant *permission* before sending off an Intent containing sensitive data.

> Developer Tip: It is easier to secure implementing *Activities* than *BroadcastReceivers* because *Activities* can ask the user before acting. However, it is easier to secure sending a broadcast than starting an *Activity* because broadcasts can assert a manifest *permission* the receiver must have.

Stick Broadcasts

Sticky broadcasts are usually informational and designed to tell other processes some fact about the system state. Sticky broadcasts stay around after they have been sent, and also have a few funny security properties. Applications need a special privilege, BROADCAST_STICKY, to send or remove a sticky Intent. You can't require a *permission* when sending a sticky broadcast, so don't use them for exchanging sensitive information! Also anyone else with BROADCAST_STICKY can remove a sticky *Intent* you create, so consider that before trusting them to persist.

> Developer Tip: Avoid using sticky broadcasts for sharing sensitive information, since they can't be secured like other broadcasts can.

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 an *Intent* and optionally communicated with over a *Binder* interface by calling *Context*'s bindService() method[31]. *Services* are similar to *BroadcastReceivers* and *Activities* in that they can be started independently of their *IntentFilters* by specifying a Component (if they are exported). *Services* can also be secured by adding a *permission* check to their <service> tag in the AndroidManifest.xml. The long lasting connections provided by bindService() create a fast IPC channel based on a *Binder* interface (see below). *Binder* interfaces can check permissions on their caller, allowing them to enforce more than one permission at a time or different permissions on different requests. *Services* therefore provide lots of ways to make sure the caller is trusted, similar to *Activities*, *BroadcastReceivers* and *Binder* interfaces.

Calling a *Service* 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[32] 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* onServiceConnected(ComponentName name, IBinder service) implementation. 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[33] as a way to validate a *permission*. We received the name of the implementing component when we receive the onServiceConnected() 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(permToCheck, name.getPackageName());
```

Figure 7 Checking a package has a permission

Compare the result of the checkPermission() call shown above with the constants PackageManager.PERMISSION_GRANTED or PackageManager.PERMISSION_DENIED. As documented the returned value is an integer, not a boolean.

[31] This is a slight oversimplification, but by using bindService() you can eventually get a binder channel to talk with a Service.

[32] An old attack on many IPC mechanisms is to "name-squat" on the expected IPC channel or name. Attackers listen on a port, name, etc. that trusted programs use to talk. Clients therefore end up talking to the wrong server.

[33] Also available to your ServiceConnection's onServiceConnected(ComponentName name, IBinder binder) method.

ContentProviders

Android has the *ContentProvider* mechanism to allow applications to share raw data. This can be implemented to share SQL data, images, sounds or whatever you like; the interface is obviously designed to be used with a SQL backend and one is even provided. *ContentProviders* are implemented by applications to expose their data to the rest of the system, the <provider> tag in the applications AndroidManifest.xml registers a provider as available and defines permissions for accessing it.

The Android security documentation mentions that there can be separate read and write *permissions* for reading and writing on a provider "…holding only the write permission does not mean you can read from a provider…" (Google Inc., 2008). People familiar with SQL will probably realize that it isn't generally possible to have write-only SQL queries. For example an updateQuery() or deleteQuery() call results in the generation of a SQL statement in which a *where* clauses is provided by the caller. This is true even if the caller has only write *permission*. Controlling a *where* clause doesn't directly return data, but the ability to change a statement's behavior based on the stored data value effectively reveals it. Through watching the side effects of a series of calls with clever *where* clauses, callers can slowly reconstruct whatever data is stored[34]. You could certainly create a provider for which this was not the case, especially if the provider is file or memory based, but it isn't likely that this will just work for simple SQL based providers. Keep this in mind before relying on write-only provider access.

Declare the read and write permissions you wish enforced by the system directly in the AndroidMainfext.xml's <provider> tag. These tags are android:readPermission and android:writePermission. These permissions are enforced at access time, subject to the limitations of the implementations discussed above. A general permission tag needed for any access can also be required.

Developer Tip: Assume clients with write access to a content provider also have read access. Describe any write *permission* you create as granting read-write access to SQL based providers.

Implementing a provider that is shared with other applications involves accepting some risks. For example, will those other applications properly synchronize their accesses of the data, and send the right notifications on changes? *ContentProviders* are very powerful, but you don't always need all that power. Consider simpler ways of coordinating data access where convenient.

An advanced feature providers may use is dynamic granting and revoking of access to other programs. The programs granted access are identified by their package name, which is the name they registered with the system on install (in their <manifest> tags, *android:package* attribute). Packages are granted

[34] Attackers exploiting "blind" SQL injection flaws use this technique for flaws that don't directly expose query results.

temporary access to a particular Uri. Generally, granting this kind of access doesn't seem like a great idea though as the granting isn't directly validated by the user, and there may not be correct restrictions on the query strings the caller can use. I also haven't worked with this option enough to give advice about using it securely[35]. It can be used by marking your provider tag with the attribute android:grantUriPermissions="true", and a subsequent <grant-uri-permission>[36] with attributes specifying which Uri's are permitted. Providers may then use grantUriPermission() and revokeUriPermission() methods to give add and remove permissions dynamically. The right can also be granted with special *Intent* flags: FLAG_GRANT_READ_URI_PERMISSION, and FLAG_GRANT_WRITE_URI_PERMISSION. Code doing this kind of thing would be a great place to start looking for security holes.

Avoiding SQL injection

To avoid SQL injection requests need to clearly delineate between the SQL statement and the data it includes. If data is misconstrued to be part of the SQL statement the resulting SQL injection can have difficult to understand consequences, from harmless bugs that annoy users to serious security holes that expose a user's data. SQL injection is easily avoided on modern platforms like Android by using parameterized queries which distinguish data from query logic explicitly. The *ContentProivder's* query(), update(), delete() and *Activities'* managedQuery() methods all support parameterization. These methods all take the "String[] selectionArgs" parameter, a set of values that get substituted into the query string in place of "?" characters, in the order the question marks appear. This provides clear separation between the content of the SQL statement in the "selection" parameter, and the data being included. If the data in selectionArgs contains characters otherwise meaningful in SQL, the database still won't be confused. You may also wish to make all your selection strings *final* in order to avoid accidentally contaminating them with user input that could lead to SQL injection.

SQL injection bugs in data input directly by the end user are likely to annoy users when they input friends whose name contains SQL meta-characters such as the single quote or apostrophe. A SQL injection could occur wherever data is received and then used in a query, that means data from callers of *Binder* interfaces, or data in *Intents* received from a broadcast, *Service* or *Activity* invocation, and these would be potential targets for malware to attempt to exploit. Always be careful about SQL injection, but consider more formal reviews of code where data for a query is from remote sources (RSS feeds, web pages, etc.). If you use parameterized types for all values you refer to and never use string concatenation to generate your SQL, you can avoid it completely.

[35] Expect an update to this paper with details on using this securely, or send me your tips.

[36] See the rather weak documentation for this here: http://code.google.com/android/reference/android/R.styleable.html#AndroidManifestGrantUriPermission

Intent Reflection

A common idiom when communicating on Android is to receive a callback via an *Intent*. For an example of this idiom in use you could look at the Location Manager, which is an optional service. The Location Manager is a binder interface with the method LocationManager.addProximityAlert(). This method takes a PendingIntent, which lets callers specify how to notify them. Such callbacks can be used any time, but occur especially frequently when engaged in IPC via an *Activity*, *Service*, *BroadcastReceiver* or *Binder* interface using *Intents*. If your program is going to send an *Intent* when called, you need to avoid letting a caller trick you into sending an *Intent* they wouldn't be allowed to. I call getting someone else to send an *Intent* for you "intent reflection", and preventing it is a key use of the *android.app.PendingIntent* class which was introduced in Android SDK 0.9.

If your application exposes an interface allowing its caller to be notified by receiving an *Intent*, you should probably change it to accept a *PendingIntent* instead of an *Intent*. *PendingIntents* are sent as the process that created them. The server making the callback can be assured that what they send will be treated as coming from the caller and not from themselves. This shifts the risk from the service to the caller. The caller now needs to trust the service with the ability to send this *Intent* as itself, which shouldn't be hard as they control the *Intent's* properties. The documentation for *PendingIntent* wisely recommends locking the *PendingIntent* to the particular component it was designed to send the callback to with setComponent(). This controls the *Intent's* dispatching.

Files and Preferences

UNIX-style file permissions are present in Android for file-systems which are formatted to support them, such as the root file system. Each application has its own area on the file system which it owns almost like programs have a home directory to go along with their user ids. An *Activity* or *Service's Context* object gives access to this directory with the getFilesDir(), getDir(), openFileOutput(), openFileInput(), getFileStreamPath(), methods but the files and paths returned by the context are not special and can be used with other file management objects like FileInputStream. The mode parameter is used to create a file with a given set of file permissions (corresponding to the UNIX file permissions). You can bitwise-OR these permissions together. For example, a mode of MODE_WORLD_WRITABLE | MODE_WORLD_READABLE makes a file world-readable[37] and writable. The value MODE_PRIVATE cannot be combined this way as it is just a zero. Somewhat oddly the mode parameter also indicates if the resultant file is truncated or opened for appending– with MODE_APPEND.

Figure 8 is a simple example of creating an example file that can be read by anyone.

```
fos = openFileOutput("PublicKey", Context.MODE_WORLD_READABLE);
```

Figure 8 Creating a World Readable file to write a public key into

The resultant *FileOutputStream* (called *fos* above) can be written to only by this process, but read by any program on the system should you wish to share it.

This interface of passing in flags that indicate files are world-readable or world-writable is simpler than the file permissions Linux supports but should be sufficient for most applications[38]. Generally any code that creates data that is world accessible must be carefully reviewed to consider:

- Is anything written to this file sensitive?[39]
- Could a change to this data cause something unpleasant or unexpected to happen?
    - Is the data in a complex format whose native parser might have exploitable vulnerabilities?[40]
- If world-writeable, do you understand that a bad program could fill up the phones memory and your application would get the blame?[41]

[37] World is also known as other, so MODE_WORLD_WRITEABLE creates other writeable files, like the command "chmod o+w somefile" does.
[38] To experiment with full Linux file permissions you could try executing chmod, or the "less documented" *android.os.FileUtils* class's static method *setPermissions()*, which takes a filename, a mode uid and gid.
[39] For example something you only know because of a permission you have.
[40] Historically a lot of complex file format parsers written in C or C++ have had exploitable parser bugs.
[41] This kind of anti-social behavior might happen. Because the file is stored under your application's home directory, the user might choose to fix the problem by uninstalling your program or wiping its data.

Obviously executable code like scripts, libraries or configuration files that specify which components, sites or folders to use would be bad candidates for allowing writes. Log files, databases or pending work would be bad candidates for world readability.

SharedPreferences is a system feature that is backed by a file with permissions like any others. The mode parameter for getSharedPreferences(String name, int mode) uses the same file modes defined by *Context*. It is very unlikely you have preferences so unimportant you don't mind if other programs change them. I recommend avoiding using MODE_WORLD_WRITEABLE, and suggest searching for it when reviewing an application as an obvious place to start looking for weaknesses.

Mass Storage

Android devices are likely to have a limited amount of memory on the internal file system. Some devices may support larger add on file systems mounted on memory cards however. For example, the emulator supports this with the –sdcard parameter, and it is referenced repeatedly in Android's documentation. Storing data on these file systems is a little tricky. To make it easy for users to move data back and forth between cameras, computers and Android, the format of these cards is VFAT. VFAT is an old standard that doesn't support the access controls of Linux, so data stored here is unprotected.

You should inform users that bulk storage is shared with all the programs on their device, and discourage them from putting really sensitive stuff there. If you need to store confidential data you can encrypt it, and store the tiny little key[42] in the application's file area, and the big ciphertext on the memory card. As long as the user doesn't want to use the storage card to move the data onto another system this should work. You may need to provide some mechanism to decrypt the data and communicate the key to the user if they wish to use the memory card to move confidential data between systems.

[42] A tiny 128 bit key is actually very strong. You can probably generate it at random as users will never need to see it. But think about the implications for backups before trying this.

Binder Interfaces

Binder is a kernel device driver that uses Linux's shared memory feature to achieve efficient, secure IPC. System services are published as *Binder* interfaces and the AIDL (Android Interface Definition Language) is used not just to define system interfaces, but to allow developers to create their own *Binder* clients and servers. The terminology can be confusing, but servers generally subclass *android.os.Binder* and implement the onTransact() method while clients receive a binder interface as an *android.os.IBinder* reference and call its transact() method. Both transact() and onTransact() use instances of *android.os.Parcel*[43] to exchange data efficiently. Android's support for *Binder* includes the interface *Parcelable*. *Parcelable* objects can be moved between processes through a *Binder*.

Under the covers, a *Binder* reference is a descriptor maintained by the *Binder* device (which is a kernel mode device driver). *Binder* IPC can be used to pass and return primitive types, *Parcelable* objects, file descriptors (which also allows memory maps), and *Binders*. Having a reference to a binder interface allows calls to its interface (i.e. call transact() and have a corresponding call to onTransact() occur on the server side) — but does not guarantee that the service exposing the interface will do what the caller requests. For example, any program can get a reference to the *Zygote* system service's *Binder* and call the method on it to launch an application as some other user, but *Zygote* will ignore such requests from unauthorized processes.

Binder security has two key ways it can enforce security: by checking the caller's identity, and by *Binder* reference security.

Security by Caller Permission or Identity Checking

When a *Binder* interface is called, the identity of the caller is securely provided by the kernel. Android associates the calling application's identity[44] with the thread on which the request is handled. This allows the recipient to use their *Context's* checkCallingPermission(String permission) or checkCallingPermissionOrSelf(String permission) methods to validate the caller's rights. Applications commonly want to enforce permissions they don't have on callers and so checkCallingPermissionOrSelf(String permission) allows the application to still call itself even if it lacks the normally needed permission. *Binder* services are free to make other binder calls, but these calls always occur with the services own identity (UID and PID) and not the identity of the caller.

Binder services also have access to the callers identity using the getCallingUid() and getCallingPid() static methods of the *Binder* class. These methods return the UID and process identifier (PID) of the process which made the *Binder* call. The identity information is securely communicated to the implementer of a *Binder* interface by the kernel[45].

[43] The native implementation of this Parcel formats data as it is expected by the kernel mode Binder device.

[44] The application's UID, and its process's current PID are provided.

[45] This is similar to how UNIX domain sockets can tell you the identity of the caller, or most IPC mechanisms on win32.

A *Binder* interface can be implemented a number of ways. The simplest is to use the AIDL compiler to create a Stub class which you then subclass. Inside the implementations of the methods the caller is automatically associated with the current thread so calling Binder.getCallingUid() identifies the caller. Developers who direct requests to handlers or implement their own onTransact() (and forego AIDL) must realize the identity of the caller is bound to the thread the call was received upon and so must be determined before switching to a new thread to handle a request. A call to Binder.clearCallingIdentity() will also stop getCallingUid() and getCallingPid() from identifying the caller. Context's checkPermission(String permission, int pid, int uid) method is useful for performing permission checks even after the callers identity has been cleared by using the stored UID and PID values.

Binder Reference Security

Binder references can be moved across a Binder interface. The Parcel.writeStrongBinder() and Parcel.readStrongBinder() methods allow this and provide some security assurances. When reading a *binder* reference from a Parcel with readStrongBinder() the receiver is assured (by the kernel's binder driver) that the writer of that binder had a reference to the received binder reference. This prevents callers from tricking servers by sending guesses of the numerical value used in the server's process to represent a *Binder* the caller doesn't have.

Getting a reference to a *Binder*[46] isn't always possible. Because servers can tell if callers had a particular binder, not giving out references to a *Binder* can effectively be used as a security boundary. While *Zygote* might not protect its binder interfaces from exposure, many *Binder* objects are kept private. To use reference security, processes need to carefully limit the revealing of *Binder* objects. Once a process receives a *Binder* it can do whatever it likes with it, passing it to others or calling its transact() method.

Binders are globally unique, which means if you create one nobody else can create one that appears equal to it. A *Binder* doesn't need to expose an interface, it might just serve as a unique value. A *Binder* can be passed between cooperating processes. A service could provide callers a *Binder* which acts as a key, knowing that only those who receive the key (or had it sent to them) could later send it back. This acts like an unguessable, easily generated password. The Activity Manager uses the reference nature of *Binders* to control management of *Surfaces* and *Activities*.

[46] By *Binder*, I mean a reference to a binder interface. When programming in Java these are represented by an *android.os.Binder* object.

Conclusion

Android applications have their own identity enforced by the system. Applications can communicate with each other using system provided mechanisms like files, *Activities*, *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 validate it by knowing the permission associated with the right you are exercising. If you are exposing your application 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 shipping, think about how you would patch a problem with your application.

A note on "root" access: code with root access, like knives or fire, is both useful and dangerous. Obviously some system processes like the *Zygote* need root to do their jobs. You have probably found that on the emulator it is easy to become root (su, adb shell, etc.) but Android distributions trying to slow recovery of data from lost phones or meet government regulations might not allow this. You may find yourself needing root access, for example to create some super encrypted and compressed virtual memory feature. This sort of thing is easy for custom builds of Android, running on emulators. Turning on security is actually a feature of an Android distribution. If the *SystemProperty* "ro.secure" is not set to 1, then the platform makes little effort at security. As much fun as it is to run Android on an emulator though, many developers will want their custom builds to work on phones too. For this a development platform[47] or a hardware platform with a friendly bootloader[48] that will let you load your code is needed.

This background information should be sufficient for you to be able to understand the presentation I will be making at Black Hat USA. It also includes most of the information provided in my developers guide for creating secure Android applications. Understanding Android application security is the starting point, not the end point for being able to understand Android distribution security. The tools I am presenting will help us look at whole distributions, and proprietary extensions to understand what we are dealing with and the security issues they

[47] OpenMoko is open, although porting hasn't been done as of this writing.

[48] A bootloader is the code that a device uses to start. Some loaders attempt to prevent the loading of custom software, this helps you feel better if you lost a phone that had personal data on it – but also inhibits deployment of custom builds.

Acknowledgements

This material is based on the developers guide the author published in 2008, shortly after the open-sourcing of the Android platform. Acknowledgements to the people that helped in writing that are given in the guild, which is maintained at http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf. I would also like to thank Black Hat USA, which is providing the venue for the release of my tools and my presentation of on Android internals and security.

Android Terminology

Activity – A screen that a user interacts with. When you create an Android application the default is to create an Activity for it. Android's "hello world" program uses an single Activity.

Activity Manager – A system program that manages *Activitys*, *Receivers*, *Services*, low memory conditions, as well as some IPC dispatching. Its Binder is available through getSystemService(String).

AIDL – Android Interface Definition Language, an object oriented interface description language that makes it easy to communicate over Binders between processes. Optimized for Java but workable in C.

AndroidManifest.xml – A main file every application uses to define itself to the system. The starting point for Android security where Activities, BroadcastReceivers, Services, ContentProviders, permissions and the need for permissions are all declared.

Binder – This is a real thing in your address space, it has a unique identity and is known about by the underlying OS. If you make one and I make one they aren't identical, if I send you mine and you send it back I can tell that it isn't some other Binder even though they don't have little names on them. You can use a Binder to talk between processes, or as an unforgeable token.

Linux – The famous Linux kernel. Used by Android and extended to support Binder and friends at http://git.android.com.

Parcelable – An interface that allows something to be put in a Parcel. This interface is usually needed to send an object over a Binder interface. Intents and Bundles are two common Parcelable classes.

Permission – The right to do something. There are many ways of expressing permissions; Android introduces Manifest Permissions which take up a few pages in this paper.

Reference Security – Capabilities is one notion of how security can be implemented by passing unforgeable tokens that represent authority around. Binders are references that can be used this way.

Side Loading – Direct user installation of applications, for example a website with an .APK users can directly install from the browser. Note that self-signed certificates don't prevent active network attackers from changing the code, so always install over SSL!

Works Cited

Chu, E. (2008, August 28). *Android Market: a user-driven content distribution system*. Retrieved August 30, 2008, from Android Developer's Blog: http://android-developers.blogspot.com/2008/08/android-market-user-driven-content.html

Google Inc. (2008, August 29). *Security and Permissions in Android*. Retrieved August 30, 2008, from Android - An Open Handset Alliance Project: http://code.google.com/android/devel/security.html

Burns, Jesse (2009, October) DEVELOPING SECURE MOBILE APPLICATIONS FOR ANDROID**.** http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf