CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 19th, 2011

Android security model

Overview

The **big difference** between Android and Windows/UNIX, is that on Android we have **mutually distrusting software**, and furthermore the **user may not trust the software he is installing**.

On Windows, the things are organized around the user. We're concerned about **protecting users from users**.

On Android, the issue is that **a program could take over your system** since you can't really know what a program does, and there are a lot of programs too.

- We want to **limit power of installed software** (sandboxing)
- We need ways to **let programs to communicate** or call each other (A program might want to use Yelp to search for a restaurant and then have it display the restaurant on a map using something like Google maps)
  - Who can call me?
    - **Example:** not all applications can "call" the address book application
  - Who can I call?
    - Giving out information, I need to be careful.
    - Maybe there's an evil Google Maps program that will do nasty things with the data I give it.

Android architecture

Android is built on top of Linux. Android applications are written in Java, although they can have some native code components.

The layers in Android are:

Application
Android API
JVM with native code support
libc, libssl
Linux Kernel

When you think JVM, you think **isolation**. That is not the case in Android where the JVM provides native code support. Also, in the Android JVM has no Java Security Manager.

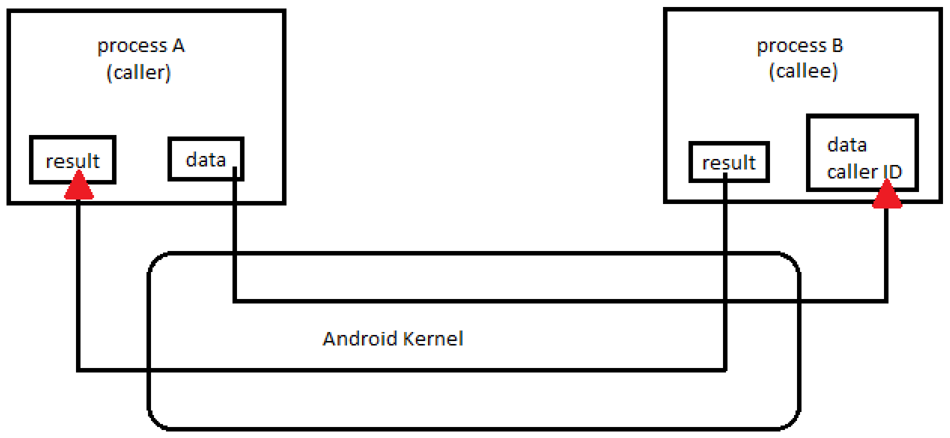
In Android, each application is installed as a **separate user**. Each application has its own home directory where its code and data reside and it is accessible only by that application's user ID. Right off the bat, one application can't go look at the files of another application. This makes all sharing explicit. So that's how Android solves the **sandboxing problem.**

Android inter-process communication

Definition: Inter-process communication (IPC) is a way for a process to get data from another process. It's basically how different processes/programs/services communicate inside an OS.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 19th, 2011

Android inter-process communication



How does **Android** do inter-process communication?

- Intents
- Activities
- Services
- Broadcast receiver
- Binder
- Files

How it works?

The **caller** will get the data it wants to pass, will **serialize** (**marshal**) the data into bytes, pass the message through the kernel to the **callee**, who then deserializes it (unmarshal), gets back the data, looks through it and determines what it needs to do, gets a result and sends it to the caller.

But who is allowed to call the callee? You could put this functionality into the kernel. **Android leaves it up the callee** to decide who can call it. When a message is sent to a callee, the callee needs some unique ID to identify all the possible callers. This is the user ID of the caller program. The kernel will be the one who will provide the ID of the caller to the callee. This is generally true for all IPC systems: provide caller ID to callee.

In **Android**, the messages are these things called **intents**. Intents are just messages. They get passed back and forth and applications can define intent filters which specify what intents an application is interested in receiving. The important thing is that these filters are not enforced! An application can force an intent that you do not want upon you.

Permissions

An application can define its own **permission variables** and the **permissions it needs** inside the application's `AndroidManifest.xml` **manifest file**.

Permissions required

One or more `<uses-permission>` tags can be used declaring the permissions that your application needs.

For example, an application that needs to monitor incoming SMS messages would specify:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
```

```
            ...
        </manifest>
```

The system comes with some **predefined permissions**. When you install an **Android** application, a pop-up tells you what kind of permissions the application needs. These are all default permissions, so they have names such as:

- Location fine (permit the application to use your GPS location)
- Location coarse (permit the application to only use the lousy location by GSM signal)
- Contacts (permit the application to access your contacts)
- SMS send (permit the application to send an SMS)

Application defined permissions

You can declare application-defined permissions in your `AndroidManifest.xml` using one or more `<permission>` tags.

An application that wants to control who can start one of its activities could declare a permission for this operation as follows:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.me.app.myapp" >
    <permission android:name="com.me.app.myapp.permission.DEADLY_ACTIVITY"
        android:label="@string/permlab_deadlyActivity"
        android:description="@string/permdesc_deadlyActivity"
        android:permissionGroup="android.permission-group.COST_MONEY"
        android:protectionLevel="dangerous" />
    ...
</manifest>
```

Example: Imagine you install a new application and it's a shopping list application so it will have a database of your shopping list items. This application might define an Add to shopping list permission or Read shopping list or Delete from shopping list that other applications can request/get in order to manipulate the shopping list of the application. These kind of permission solve the **who can call me problem**.

Activities

Activities are small actions that are performed by one application to another.

- Each applications defines the activities other applications can perform on it
- Some activities are generic like "browse to location"
- Activities are defined in an application's manifest file

An application that wants to perform an activity first constructs an **intent** and then **invokes an activity** with that intent. The sender does not know which application the intent will go to. The intent merely indicates the action an application wants to perform such as *"browse to location"*. It does not indicate how that action should be performed. Therefore, the intent should not contain sensitive info since it may be end up going to an **evil application**.

Activities provide **some control over "who can call me,"** since the recipient of an intent can specify **intent filters** to indicate who can call him.

Also, activity implementations inside an application can check the permissions of invoker. When they receive an intent, they retrieve the ID of the sender and they call a function to check that the sender has the needed permission, such as *"add to shopping list"*.

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 19th, 2011
Services are pretty much the same as activities, almost like a long-term activity.

Broadcast receivers

Broadcast receivers are defined in your application's manifest file.

Senders can require permissions on **receivers**. This way a sender can ensure it is safe to send sensitive information to an application.

Example: If you had an application that was a *password storage application,* then every sender application who wants to store a password for the user in a safe place will ensure that the receiver has the *"can store password"* permission. Your *password storage application* will have acquired the *"can store password"* permission before hand by prompting you to grant it this permission.

Binder

Binder is a low-level IPC system which implements **capabilities**.

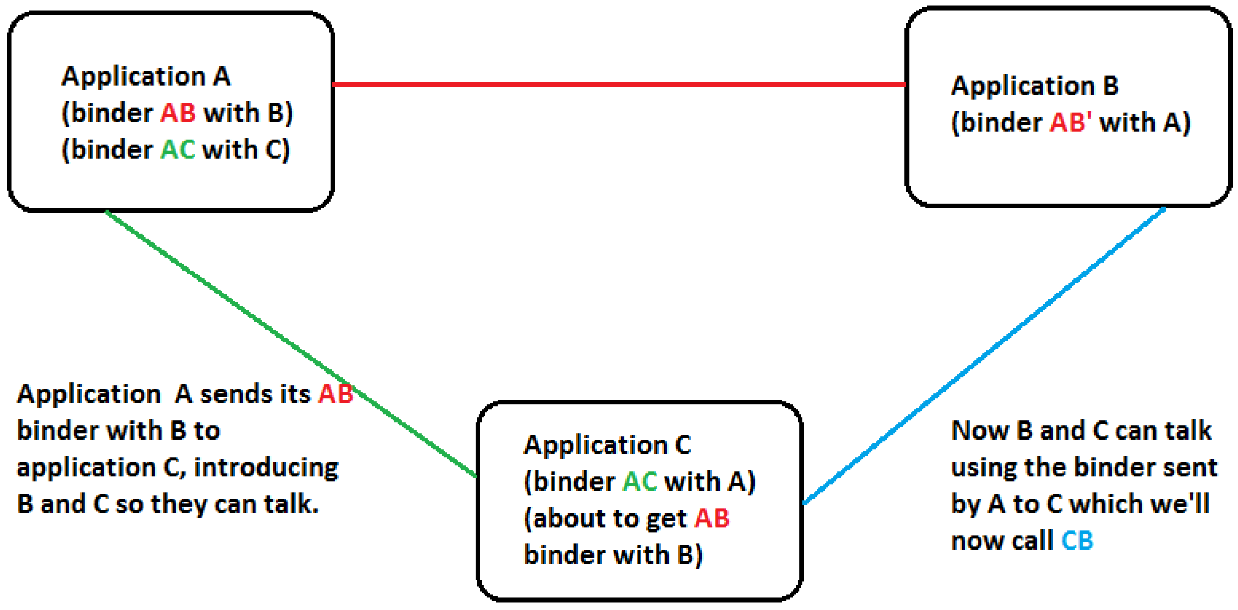
Binder works by allowing **binder objects** to communicate with each other. Externally, a binder object has:

- a public `transact` method which takes a **parcelable** message (a message that can be converted to a string of bytes)
- a protected method called `onTransact`

The system is set up with as a client- server scheme. The client will have a **binder reference object** and the server will have **a binder object**. These two objects are linked together through the **kernel binder module**. The binder object will have fields such as caller UID, caller GIDs.

It turns out that **Android** lets you serialize and send binder objects using the binder kernel module.

Imagine application **A** has a binder connection with application **B** and one with application **C**. Application **A** can send its application **B** binder to application **C**. This way application **A** acts as an *introducer* for B and C. Now application **B** can talk to application **C**.


Application A
(binder AB with B)
(binder AC with C)
Application B
(binder AB' with A)
Application A sends its AB binder with B to application C, introducing B and C so they can talk.
Application C
(binder AC with A)
(about to get AB binder with B)
Now B and C can talk using the binder sent by A to C which we'll now call CB

CSE 409, Fall 2011, Rob Johnson, http://www.cs.stonybrook.edu/~rob/teaching/cse409-fa11/
Alin Tomescu, September 19th, 2011
Note: A client cannot construct a binder reference object. A client **has to be given** one from a server possessing a binder object.

There are two ways to enforce binder permissions:

- Before you give binder references
- After you give binder references, inside the `onTransact` method
    - In your `onTransact` method, you look up the caller UID and check its permissions

Capabilities

"**Capability-based security** is a concept in the design of secure computing systems, one of the existing security models. A **capability** (known in some systems as a *key*) is a communicable, unforgeable token of authority. It refers to a value that references an object along with an associated set of access rights. A user program on a capability-based operating system must use a capability to access an object." --Wikipedia

Compared to the security community, Linux has a different notion of capabilities.

Capabilities have two *key properties*, they are **unforgeable** and they are **explicit**.

Story: When you go open a file on UNIX and Windows do you have to say what permissions you want to use? You have to specify whether you want to read or write the file, but *you don't have to say what subset of your permissions you want the OS to use*. This is called **ambient authority**. The OS looks in this cloud of authority for something that will allow you to do what you want. This leads to the **confused deputy** problem: someone acting as a *deputy*, gets two keys from its two masters, with one key having more access than the other. Now that someone might use the higher privileged key to perform an action for the lower privileged master. **BIG NO NO!**

If the program wants to access a resource it should explain to the OS how it has the **capability** of accessing that resource. That's why they can't be forgeable. Ideally, a program should start with its **capability set** as a parameter, and this set should suffice to determine what the program can do:

- Can I open this file for reading or for writing?
- Can I print this file?
- Can I create a raw socket?
- etc.

```
int main(int initial_capability)
{
    FILE * fd = open(initial_capability, "foo.txt", "rw");

    return 0;
}
```

There are OS-level capabilities and then there are language-level capabilities.

OS-level capabilities

After a file is opened using **ambient authority**, file access is done using OS-level capabilities. That is, when a `write` call is made on a file handle, the OS merely looks in a table which maps every file handle to its capabilities:

- Can the caller use this handle to write to the file?
- Can the caller use this handle to read the file?
- Can the caller use this handle to write the file's attributes?

- Can the caller delete this file?
- etc.