BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Mobile development and security

Session 15

Karim Karimov
Lecturer

Android security

The Android platform claims to be “***open***,” the meaning of which seems to be largely open to interpretation. Many companies claim their patented or even closed-source software is “***open***” because they published an API. Android, on the other hand, is open because developers can see and change its source code without restrictive licenses or fees. It is also open because it’s designed to be securable and is able to run third-party applications. Platforms with weaker security models sometimes need to cover up their weakness by locking the applications down to only the “***known good***” ones, thus throwing up barriers to application development and hindering user choice. The Open Handset Alliance takes the idea of “***open***” further and even states “***Android does not differentiate between the phone’s core applications and third-party applications***” (Open Handset Alliance, 2009).

Android security

Android distributions can be configured so that their owners **do not have root access** or can’t change certain aspects of the system or settings. So far this approach is common on phones, and it can help users feel safe if they lose their phone, or install third-party programs. It also helps reassure carriers that **licensed content** such as ringtones is somewhat protected. Phones can also go through a process of being **“locked” to a particular network**, which helps protect the business model of carriers who sell devices at a loss to encourage subscriptions. However, with any phone, someone technical with physical access to the device can probably “fix” either of these configurations with a bit of time and effort. An owner **breaking root** on own device shouldn’t harm the security model of Android, however, and sensible people probably expect users to get full control of their phones.

Development on Android

Android has two types of developers: **application developers** who build software for the platform based on the Android SDK, and **system developers** who extend or adapt the Android platform for their devices or to contribute back to the platform. Most system developers work for Google or phone manufacturers. Application developers are much more numerous and better supported.

Debugging on Android

Debugging support is built into Android and provided in such a way that working with a device or with the emulator is mostly interchangeable. Android's support for debugging is provided primarily through a debugging deamon (**/sbin/adbd**), which allows software on your development machine to connect to the software running on the device. There are two distinct ways to debug on the platform—one for **native code** and the other for **code running in the virtual machine (Dalvik)**.

For the device-based debugging to work, you need to have the **Android Debug Bridge Daemon** (**adbd**) running on the device. This is usually started by going into **Settings | Applications | Development** and then enabling **USB debugging** on your device.This program runs as the user "shell" and provides data stream forwarding services for TCP and UDP, Unix domain sockets, and more. It also has the ability to execute commands on the device as the "shell" account and can therefore install packages as well as copy files onto or off of the device.

Android’s Security Model

Android is based on the Linux kernel, which provides a security model. Android has abstractions that are unique to it, however, and they are implemented on top of Linux, leveraging Linux user accounts to silo applications. Android permissions are rights given to applications to allow them to take pictures, use the GPS, make phone calls, and so on. When installed, **applications are given a unique user identifier** (**UID**); this is the familiar Unix UID seen on desktops and servers. It is a small number like ***1011*** that is unique on a given system and used by the kernel to control access to files, devices, and other resources. Applications will always run as their given UID on a particular device, just like users always have their same UID on a particular server but different UIDs on unrelated systems. The UID of an application is used to protect its data, and developers need to be explicit about sharing data with other applications. Applications can entertain users with graphics, play music, run native code and launch other programs without needing any permissions.

Android’s Security Model

The need for permissions minimizes the impact of malicious software, unless a user unwisely grants powerful rights to dubious software. Preventing people from making bad but informed choices is beyond the scope of the security model—**the permission model is designed to make the choice an informed one**. The Android permission model is extensible, and developers need to keep in mind what is reasonable for a phone user to understand when defining new permissions for them. A confused user can’t make good choices. To minimize the extent of abuse possible, permissions are needed for programs that perform potentially dangerous operations that the phone needs to support, such as the following:

- Directly dialing calls (which may incur tolls)
- Accessing private data
- Altering address books, e-mail, and so on

Android Permissions Review

Applications need approval to perform tasks their owner might object to, such as 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 file, and the user agrees to these upon install (up to API version 26 - android 6.0). Newer version of Android platform requires runtime permission based on code requirement.

To be useful, permissions must be associated with some goal that a user can understand. For example, an application needs the **READ_CONTACTS** permission to read the user's address book (the permission's full name is "***android.permission.READ_CONTACTS***"). A contact management program needs **READ_CONTACTS** permission, but a block stacking game shouldn't.

Manifest Permission Protection Levels

Protection Levels	Protection Behavior
Normal	Permissions for application features whose consequences are minor (for example, VIBRATE, which lets applications vibrate the device). Suitable for granting rights not generally of keen interest to users. Users can review them but may not be explicitly warned.
Dangerous	Permissions such as WRITE_SETTINGS and SEND_SMS are dangerous because 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 upon install, although the specific behavior may vary according to the version of Android or the device upon which it is installed.
Signature	These permissions are only granted to other applications signed with the same key as the program. This allows secure coordination without publishing a public interface.
SignatureOrSystem	Similar to Signature, except that programs on the system image also qualify for access. This allows programs on custom Android systems to also get the permission. This protection helps integrate system builds and won't typically be needed by developers. **Note:** 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 thus protect more stable interfaces than Signature.

Android Permissions Exception

If you try to use an interface 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.

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 them to ensure translations are accurate.

Creating New Manifest Permissions

Applications can define their own permissions if they intend other applications to have programmatic access to them. If your application doesn't intend for other applications to call it, you should just **not export** any Activities, BroadcastReceivers, Services, or ContentProviders you create and not worry about permissions. Using a manifest permission allows the end user to decide which programs get programmatic access. For example, an application that manages a shopping list application could define a permission named ***"com.isecpartners.ACCESS_SHOPPING_LIST"***. 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. 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.

Securable IPC Mechanisms

Android implements a few key tools used to communicate with or coordinate between programs securely. These mechanisms give Android applications the ability to run processes in the background, offer services consumed by other applications, safely share relational data, start other programs, and reuse components from other applications safely.

Much of the **interprocess communication (IPC)** that occurs on Android is done through the passing around of a data structures called **Intents**. These are collections of information that have a few expected properties the system can use to help figure out where to send an Intent if the developer wasn't explicit.

Each of these IPC mechanisms uses Intents in some capacity and is probably somewhat familiar to most Android developers. However, because using these safely is key to Android security.

Intents

Intents are used in a number of ways by Android:

- To start an Activity (by coordinating with other programs) such as browsing a web page.
  - Example: Using Context's ***startActivity()*** method.
- As Broadcasts to inform interested programs of changes or events.
  - Example: Using Context's ***sendBroadcast()***, ***sendStickyBroadcast()***, and ***sendOrderedBroadcast()*** family of methods.
- As a way to start, stop, or communicate with background Services.
  - Example: Using Context's ***startService()***, ***stopService()***, and ***bindService()*** methods.
- As callbacks to handle events, such as returning results or errors asynchronously with **PendingIntents** provided by clients to servers through their Binder interfaces.

IntentFilters

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 override the **IntentFilter** match Activity Manager uses, however. Any **"exported"** Activity can be started with any Intent values for action, data, category, extras, and so on. (Activities can exported explicitly via the **android:exported="true"** attribute at manifest file.) 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 Manager for help with figuring out where the Intent should go, but it doesn't have to.

IntentFilters

Intent recipients such as Activities, Services, and BroadcastReceivers need to handle potentially hostile callers, and an IntentFilter doesn’t filter a malicious Intent.
IntentFilters help the system figure out the right handler for a particular Intent, but they don’t constitute an input-filtering or validation system. Because IntentFilters are not a security boundary, they cannot be associated with permissions.
Adding a category to an Intent restricts what it will be resolved to. For example, an IntentFilter that has the ***“android.intent.category.BROWSABLE”*** category indicates that it is safe to be called from the web browser. Carefully consider why Intents would have a category and consider whether you have met the terms of that, usually undocumented, contract before placing a category in an IntentFilter.

Broadcasts

Broadcasts provide 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.
Intents can be broadcast to BroadcastReceivers, allowing messaging between applications. By registering a BroadcastReceiver in your application's AndroidManifest .xml file, you can have your application's receiver class started and called whenever someone sends a broadcast your application is interested in. Activity Manager uses the IntentFilter's applications register to figure out which program to use to handle a given broadcast.
BroadcastReceivers are registered in AndroidManifest.xml with the **<receiver>** tag. By default they are not exported. However, you can export them 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.

Safely Sending Broadcast Intents

When sending a broadcast, developers include some information or sometimes even a sensitive object such as a **Binder**. If the data being sent is sensitive, they will need to be careful who it is sent to. The simplest way to protect this while keeping the system dynamic is to **require the receiver to have 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 you 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, you should use this IPC mechanism.
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.

Sticky 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 sticky broadcasts, 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.

Avoid using sticky broadcasts for sharing sensitive information because they can’t be secured like other broadcasts can.

Services

Services are long-running background processes provided by Android to allow for background tasks such as playing music and running a game server. They can be started with an Intent and optionally communicated with over a **Binder** interface via a call to Context's **bindService()** method. Services are similar to BroadcastReceivers and Activities in that you can start them 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.

Calling a Service is slightly trickier. If you need to make sensitive calls into a Service, such as storing passwords or private messages, you'll need to validate that the Service you're connect to is the correct one and not some hostile program that shouldn't have access to the information you provide. If you know the exact component you are trying to connect to, you can specify that explicitly in the Intent you use to connect. Alternatively, you can verify it against the name provided to your SeviceConnection's **onServiceConnected (ComponentName name, IBinder service)** implementation.

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 application's AndroidManifest.xml file registers a provider as available and defines permissions for accessing it.
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. Declare the read and write permissions you wish enforced by the system directly in AndroidMainfext.xml's **<provider>** tag. These tags are **android:readPermission** and **android:writePermission**.
ContentProviders are very powerful, but you don't always need all that power. Consider simpler ways of coordinating data access where convenient.

Avoiding SQL Injection

To avoid SQL injection requests, you 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 resultant 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 such as Android via parameterized queries that distinguish data from query logic explicitly. The ContentProvider's **query()**, **update()**, and **delete()** methods and Activity's **managedQuery()** method 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.

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 that 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 (prior to which intent reflection was endemic).

Intent Reflection

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 it sends will be treated as coming from the caller and not from itself. 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 because they control the Intent’s properties. The PendingIntent documentation 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 that are formatted to support them, such as the root file system. Each application has its own area on the file system that 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()**, and **getFileStreamPath()** methods, but the files and paths returned by the context are not special and can be used with other file-management objects such as **FileInputStream**. Generally, any code that creates data that is world-accessible must be carefully reviewed to consider the following:

- Is anything written to this file sensitive? For example, something only you know because of a permission you have.
- If the file is world-writeable, a bad program could fill up the phone's memory and your application would get the blame! This kind of antisocial behavior might happen—and 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.

Files and Preferences

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. It is recommended to avoid using **MODE_WORLD_WRITEABLE** and search 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. To make it easy for users to move data back and forth between cameras, computers, and Android, the format of these cards is **VFAT**, which is an old standard that doesn't support the access controls of Linux. Therefore, data stored here is unprotected and can be accessed by any program on the device.

You should inform users that bulk storage is shared with all the programs on their device, and discourage them from putting really sensitive data there. If you need to store confidential data, you can **encrypt it and store the tiny little key in the application's file area and the big cipher-text on the shared 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.

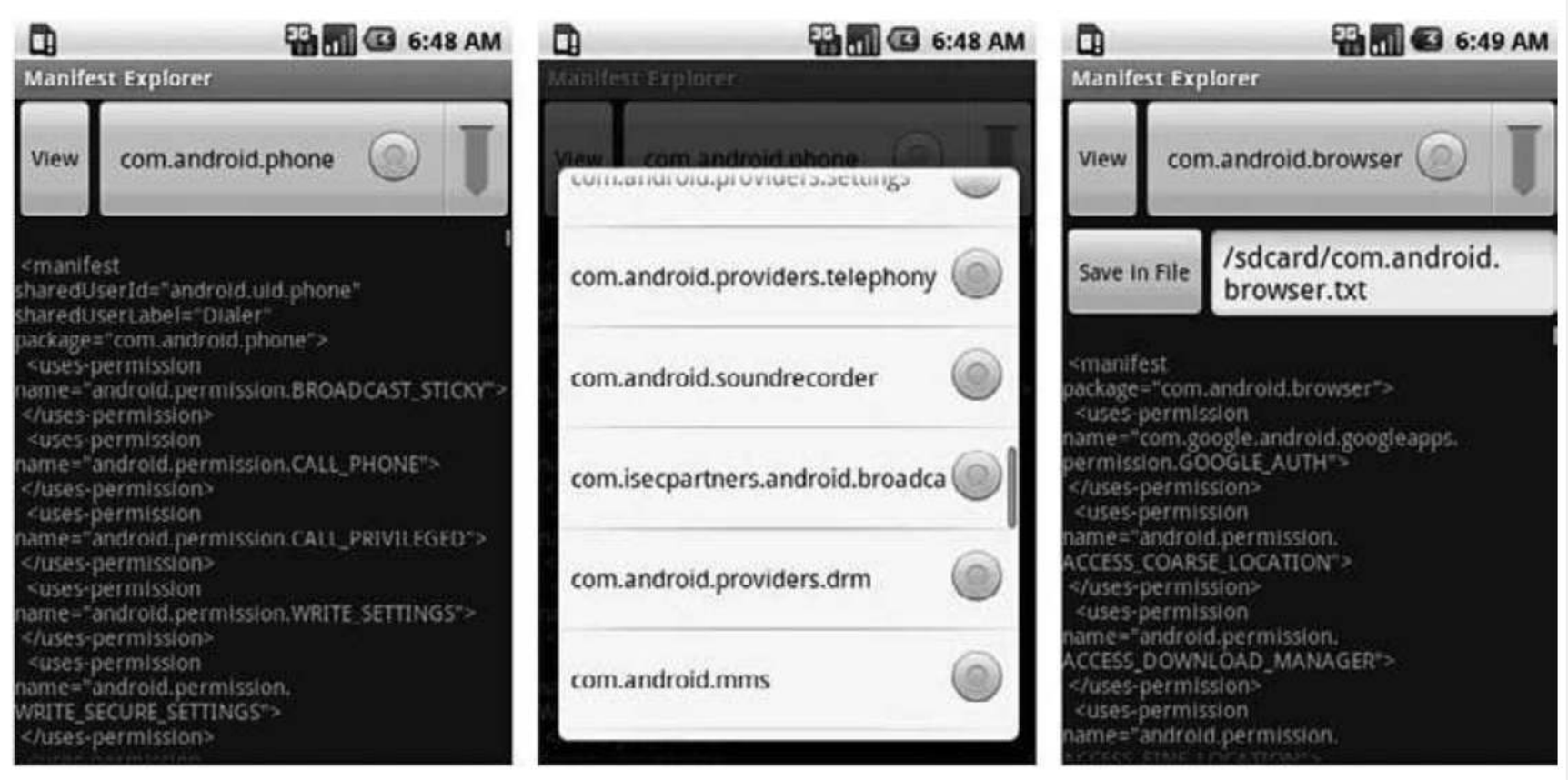
Android Security Tools

Manifest Explorer

Both Android distributions, and every application installed on them must have an **AndroidManifest.xml** policy file, which Manifest Explorer helps the user find and view. The file is of great interesting when analyzing system security because it defines the permissions the system and applications enforce and many of the particular protections being enforced. The Manifest Explorer tool can be used to review the AndroidManifest.xml file, the security policies and permissions of applications and the system, as well as many of the IPC channels that applications define and which end up defining the attack surface of applications. This attack surface outline is a common starting point for understanding the security of application and Android distributions.

Android Security Tools

Manifest Explorer



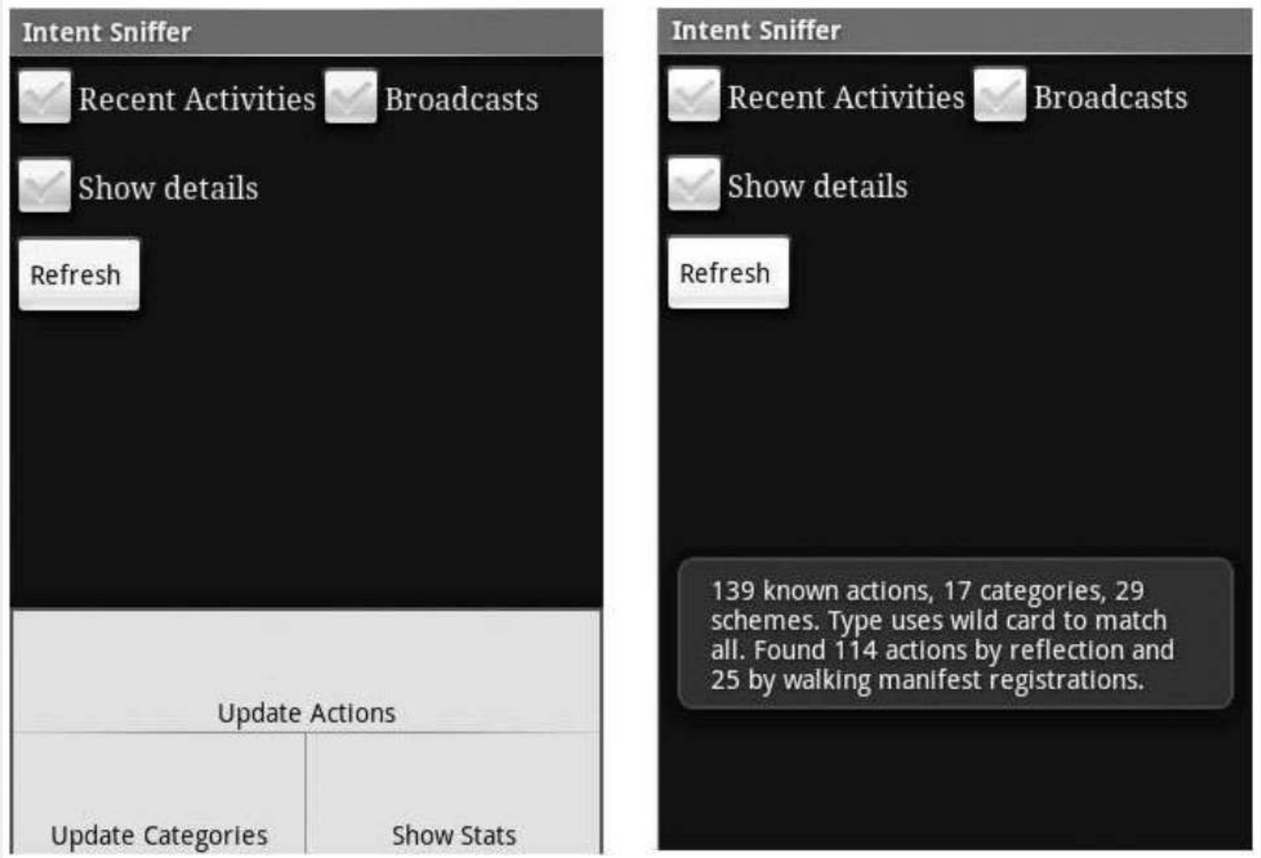
Android Security Tools

Intent Sniffer

On Android, an Intents are one of the most common ways applications communicate with each other. The Intent Sniffer tool performs monitoring of runtime routed broadcasts Intents, sent between applications on the system. It does not see explicit broadcast Intents, but defaults to (mostly) unprivileged broadcasts. There is an option to see recent tasks' Intents (**GET_TASKS**), as the Intent's used to start Activities are accessible to applications with GET_TASKS permission like Intent Sniffer. The tool can also dynamically update the Actions and Categories it scans for Intents based on using reflection and dynamic inspection of the installed applications.

Android Security Tools

Intent Sniffer



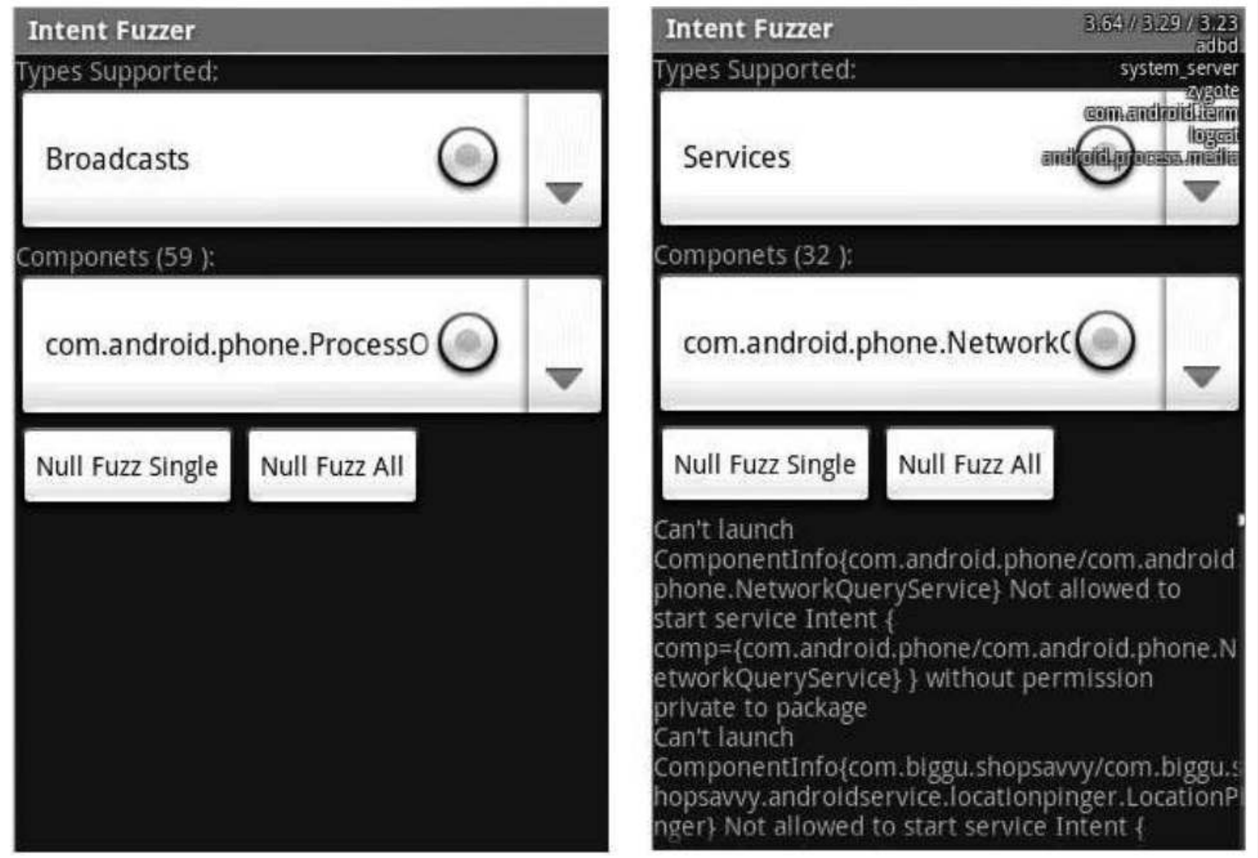
Android Security Tools

Intent Fuzzer

A fuzzer is a testing tool that sends unexpected or incorrect input to an application in an attempt to cause it to fail. Intent Fuzzer is exactly what is seems—it is a fuzzer for Intents. It often finds bugs that cause the system to crash as well as performance issues on devices, applications or custom platform distributions. The tool can fuzz either a single component or all installed components. It works well on BroadcastReceivers but offers less coverage for Services, which often use Binder interfaces more intensively than Intents for IPC. Only single Activities can be fuzzed, not all them at once. Instrumentations can also be started using this interface, and although ContentProviders are listed, they are not an Intent-based IPC mechanism and so cannot be fuzzed with this tool. Developers ma want to adapt Intent Fuzzer so that it can provide Intents more appropriate for their application.

Android Security Tools

Intent Fuzzer



End of the session