Android Security

Scribe: Eleanor Cawthon and Ben Zhang

Nov 12, 2015

1 Understanding Android Security

1.1 Background

This week's discussion is based on the paper "Understanding Android Security." Android is an open source operating system for mobile devices that provides a base operating system, an application middleware layer, a Java software development kit (SDK), and a collection of system applications.

Android applications can either be downloaded from the official store, or installed directly. The **threat model** is that some Android applications can be compromised or malicious, while the OS's **goal** is to protect data and other applications from these compromised apps.

Typical Android applications consists of four components as below. Figure 1 shows these components in example applications.

- *Activity* defines an applications's user interface (UI).
- *Service* represents background processes (e.g., music).
- *Content Provider* contains database (store and share data).
- *Broadcast Receiver* receives messages from other applications.

1.2 Security Mechanisms

The security enforcement of Android include the following three mechanisms:

1. Each application is assigned a unique user ID. This prevents applications from accessing information of another application (isolation).
2. ICC (inter component communication), which is the flip side of isolation — a way for components to talk to each other. We will talk about more later.
3. Code signing. Each developer has SK_d and PK_d to sign the code and provide a certificate. The certificate can be self-signed or signed by third parties like Symantec.

   Note: Self-signing is still useful—it allows linking of multiple apps by a single developer. This makes it easy to

   - ensure updates are from the same developer,

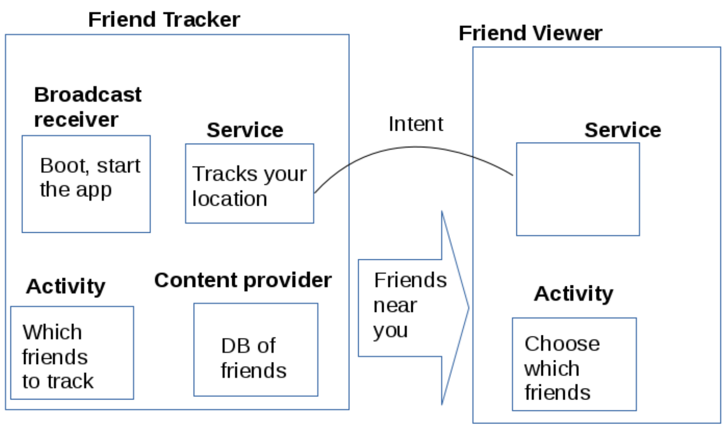

Figure 1: The *FriendTracker* and *FriendViewer* example Android applications (the figure is based on the one drawn in class. See also: the copyrighted image in the original paper).

- to have apps by the same developer communicate with/trust each other, and
- to flag malicious developers.

1.2.1 Inter-Component Communication

Inter-component communication (ICC) is based around *intents*. An intent is a message that contains a destination component, an action, and the data.

The destination component can be an activity (use `startActivity(intent)`), a service (use `startService(intent)`). It can also be any interested components by `broadcastIntent(intent)`.

Android implements Mandatory Access Control (MAC) that is based on permissions labels to arbitrate ICC communications. The OS checks that permissions of an application include the permission of the target component.

1. Applications. Developer specifies permissions in the manifest. Some permissions could be dangerous—which means the user must be asked for approval at install time.
2. Components. Developer can create new permission labels for each component.

Types of permissions:

- *Normal*: applications get the permission automatically.
- *Dangerous*: the user is asked at install time whether they want to grant permission or not.
- *Signature*: Application written by developer A gets permission to component written by developer B if A=B.
- *Sig & system*: either it satisfies the above or it's a system application.

2 Student Presentations

2.1 Android Permissions: User Attention, Comprehension, and Behavior

This paper looks at user perspectives on the useability of all these Android security features. Survey of 300 random (hopefully representative) Android users, followed up by in-depth interviews with 25 of them (in which researchers watched them install an app).

1. Q: Do users notice permissions before installing?

   A: 17% of users notice. This number stayed the same across the survey and interview components. Users mostly pay attention to reviews.
2. Q: Do users understand how permissions correspond to application privileges?

   A: They had a survey with 3 questions showing a permission and asking what that means it can do (multiple choice). 3% got all 3 questions right.

Conclusion: Users don't understand Android permissions!

What's happened in 3 years?

Bad: apps can automatically add more permissions in updates!

Better: Improvement in most recent android version: Apps can not ask for the permission until it's used.

Users want to be able to block requests — 8% would block at least some if they could.

PiOS: Detecting Privacy Leaks in iOS Applications

Definition: reading sensitive information and sending it to third party through network

Overview:

- Analyze iOS objective-C binary to create Control-Flow-Graphs
- Reachability analysis on CFGs to identify path from sensitive information to third parties
- Results: Analyzed >1400 iPhone applications. Most of them leak deviceID, but other information was not found to be leaked severely.

Challenges:

- Encrypted binaries (solved by jailbreak and debug tools)
- Class hierarchy has to be partially reconstructed by "backwards slicing" and with SDK header files
- Limitation: Only static analysis

Result: Most apps leaked the unique device ID (which was generally being used for advertising). Leaking the device ID matters — e.g. Google can use this to link behavior across apps.

The second-most leaked datum was location.

TaintDroid

Overview

- Taint-tracking system on android: tracks explicit info flows
- Goals: detect when sensitive data leaves the phone dynamically; improve application analysis for users and 3rd parties; low performance overhead
- Challenges: 3rd party apps can access a lot of information since they just ask for permissions at install time. It's non-trivial to decide what is and isn't sensitive.

a message either is tainted or not, and it remains tainted if any chunk of data containing it is passed around. This minimizes overhead but introduces false positives.

Taint *sources* are sensors (location...), info databases (SMS...), and device identifiers. The *sink* is network interface flows outside the device

Taint is only propagated if leaky methods are called on the sensitive data (e.g., length() doesn't count for purposes of this paper)

Results:

- 9.3% of apps leaked tainted data (Google Play Services considered consensual, when that's excluded it's still 6.0%)
- 30% sent identifiers without consent
- 15 apps (50%) sent location data

Optimization: Previous taint tracking approaches for android were a lot slower. This adds spatial locality and variable semantics: double the size of stack, and then double the index to find the associated data.

Takeaway: A lot of apps leak a lot of data, and this is a pretty efficient way of measuring this.

Limitations of the system include false positive on IPC taint tracking, false negatives on JNI method taint tracking, and the system only uses data flow but not control flow.