Implementing Android Application security

[1]*Puli Rushyanth*
[1]*M.Tech Student,Dept of CSE*
KLUniversity,Guntur,Andhra Pradesh,India

Abstract:

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.

1.Introduction:

Android is a Linux platform programmed with Java and enhanced with its own security mechanisms tuned for a mobile environment . 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 including regulatory compliance . 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.

Our popularity-focused security analysis provides insight into the most frequently used applications. Our findings inform the following broad observations.

1. Similar to past studies, we found wide misuse of privacy sensitive information—particularly phone identifiers and geographic location. Phone identifiers, e.g., IMEI, IMSI, and ICC-ID, were used for everything from "cookie-esque" tracking to accounts numbers.

2. We found no evidence of telephony misuse, background recording of audio or video, abusive connections, or harvesting lists of installed applications.

3. Ad and analytic network libraries are integrated with 51% of the applications studied, with Ad Mob (appearing in 29.09% of apps) and Google Ads (appearing in 18.72% of apps) dominating. Many applications include more than one ad library.

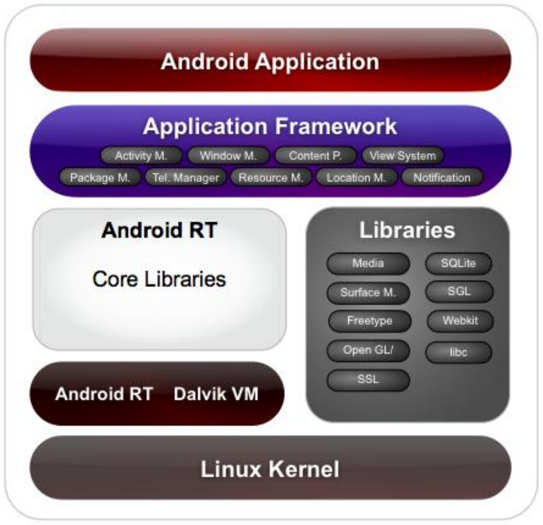

Figure 1: The Android system architecture

2 Background

Android: Android is an OS designed for smartphones. Depicted in Figure 1, Android provides a sandboxed application execution environment. A customized embedded Linux system interacts with the phone hardware and an off-processor cellular radio. The Binder middleware and application API runs on top of Linux. To simplify, an application's only interface to the phone is through these APIs. Each application is executed within a Dalvik Virtual Machine (DVM) running under a unique UNIX uid. The phone comes pre-installed with a selection of system applications, e.g., phone dialer, address book. Applications interact with each other and the phone through different forms of IPC. Intents are typed interprocess messages that are directed to particular applications or systems services, or broadcast to applications subscribing to a particular intent type. Persistent content provider data stores are queried through SQL-like interfaces. Background services provide RPC and callback interfaces that applications use to trigger actions or access data. Finally user interface activitiesreceive named action signals from the system and other applications. Binder acts as a mediation point for all IPC. Access to system resources (e.g., GPS receivers, text messaging, phone services, and the Internet), data (e.g., address books, email) and IPC is governed by permissions assigned at install time. The permissions requested by the application and the permissions required to access the application's interfaces/data are defined in its manifest file. To simplify, an application is allowed to access a resource or interface if the required permission .

3.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 install14. 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.

4. Evaluating Android Security

Our Android application study consisted of a broad range of tests focused on three kinds of analysis: a) exploring issues uncovered in previous studies and malware advisories, b) searching for general coding security failures, and c) exploring misuse/security failures in the use of Android framework. The following discusses the process of identifying and encoding the tests.

4.1 Analysis Specification We used four approaches to evaluate recovered source code: control flow analysis, data flow analysis, structural analysis, and semantic analysis. Unless otherwise specified, all tests used the Fortify SCA [2] static analysis suite, which provides these four types of analysis. The following discusses the general application of these approaches. The details for our analysis specifications can be found in the technical report .Control flow analysis. Control flow analysis imposes constraints on the sequences of actions executed by an input program P, classifying some of them as errors. Essentially, a control flow rule is an automaton A whose input words are sequences of actions of P—i.e., the rule monitors executions of P. An erroneous action sequence is one that drives A into a predefined error state. To statically detect violations specified by A, the program analysis traces each control flow path in the tool's model of P, synchronously "executing" A on the actions executed along this path. Since not all control flow paths in the model are feasible in concrete executions ofP, false positives are possible. False negatives are also possible in principle, though uncommon in practice. Figure 4 shows an example automaton for sending intents. Here, the error state is reached if the intent contains data and is sent unprotected without specifying the target component, resulting in a potential unintended information leakage. Init p1 p2 p3 p4 p5 p6

p1 = i.$new_class(...)

p2 = i.$new(...) |

i.$new_action(...)

p3 = i.$set_class(...) |

i.$set_component(...)

p4 = i.$put_extra(...)

p5 = i.$set_class(...) |

i.$set_component(...)

p6 = $unprotected_send(i) |

$protected_send(i, null)

targeted error

empty has_data

Figure 2: Example control flow specification Data flow analysis.

Data flow analysis permits the declarative specification of problematic data flows in the input program. For example, an Android phone contains several pieces of private information that should never leave the phone: the user's phone number, IMEI (device ID), IMSI (subscriber ID), and ICC-ID (SIM card serial number). In our study, we wanted to check that this information is not leaked to the network. While this property can in principle be coded using automata, data flow specification allows for a much easier encoding. The specifi cation declaratively labels program statements matching certain syntactic patterns as data flow sources and sinks. Data flows between the sources and sinks are violations. Structural analysis. Structural analysis allows for declarative pattern matching on the abstract syntax of the input source code. Structural analysis specifications are not concerned with program executions or data flow, therefore, analysis is local and straightforward. For example, in our study, we wanted to specify a bug pattern where an Android application mines the device ID of the phone on which it runs. This pattern was defined using a structural rule that stated that the input program called a method getDeviceId() whose enclosing class was android.telephony.TelephonyManager. Semantic analysis. Semantic analysis allows the specification of a limited set of constraints on the values used by the input program. For example, a property of interest in our study was that an Android application does not send SMS messages to hard coded targets. To express this property, we defined a pattern matching calls to Android messaging methods such assendTextMessage(). Semantic specifications permit us to directly specify that the first parameter in these calls (the phone number) is not a constant. The analyzer detects violations to this property using constant propagation techniques well known in program analysis literature.

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, While our findings of exposure of phone identifiers and location are consistent with previous studies, our analysis framework allows us to observe not only the existence of dangerous functionality, but also how it occurs within the context of the application.Its provide the security.

References:

[1] Fernflower - java decompiler. http://www.reversed-java.com/fernflower/.

[2] Fortify 360 Source Code Analyzer (SCA). https://www.fortify.com/products/fortify360/source-code-analyzer.html.

[3] Jad. http://www.kpdus.com/jad.html.

[4] Jd java decompiler. http://java.decompiler.free.fr/.

[5] Mocha, the java decompiler. http://www.brouhaha.com/~eric/software/mocha/.

[6] ADMOB. AdMob Android SDK: Installation Instructions. http://www.admob.com/docs/AdMob_Android_SDK_Instructions.pdf. Accessed November 2010.

[7] ASHCRAFT, K., AND ENGLER, D. Using Programmer-Written Compiler Extensions to Catch Security Holes. In Proceedings ofthe IEEE Symposium on Security and Privacy (2002).

[8] BBC NEWS. New iPhone worm can act like botnet bsay experts. http://news.bbc.co.uk/2/hi/technology/8373739.stm, November 23, 2009.

[9] BORNSTEIN, D. Google i/o 2008 - dalvik virtual machine internals. http://www.youtube.com/watch?v=ptjedOZEXPM.