ANDROID SECURING YOU

Mohd. Salman Khan[1], Abhishek Sharma[2], Arpit Kansal[3]

[1]*Assistant Professor, ABESIT Ghaziabad, (India)*

[2,3] *B.Tech IT, ABESIT Ghaziabad, (India)*

ABSTRACT

Android is a software stack for mobile devices that includes an operating system, middleware and key applications. The Android provides the tools and APIs necessary to begin developing applications on the Android platform using programming language. Android is a widely anticipated 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 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. This paper we should already be familiar with Android's basic architecture and major abstractions including: Intents, Activities, Broadcast Receivers, Services, Content Providers and Binder. As android is open source we should also have this code available to us. Both the java and C code is critical for understanding how Android works, and is far more detailed than any of the platform documentation.

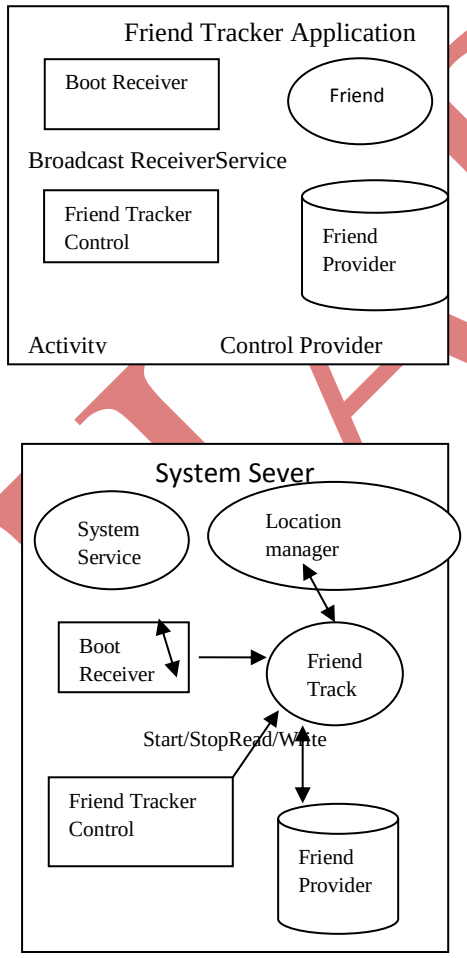
I INTRODUCTION

Android has security features built into the operating system that significantly reduce the frequency and impact of application security issues. The system is designed so we can typically build applications with default system and file permissions and avoid difficult decisions about security. Android is a Linux based operating system with native libraries programmed in C / C++ and with enhanced 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 soiled 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, and sharing data between applications must be done explicitly. The Android GUI environment has some novel security features that help support this isolation. 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.

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. 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.

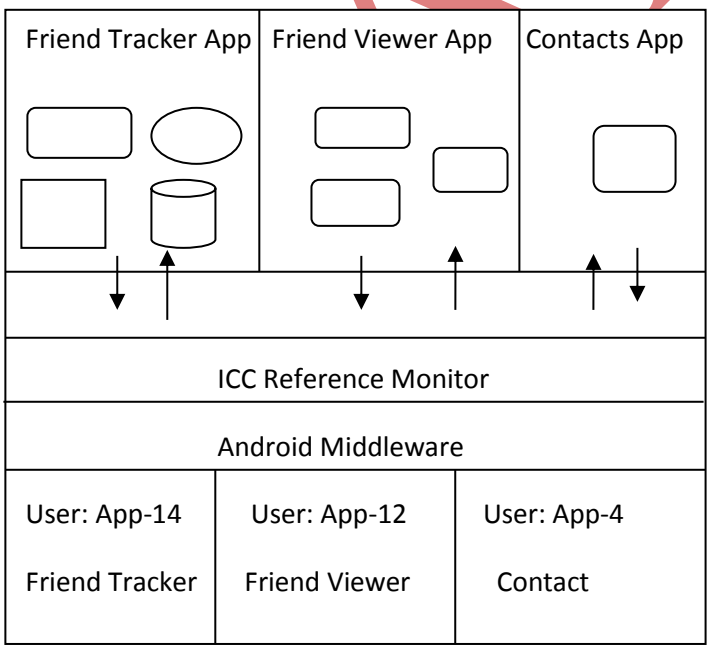
II ANDROID APPLICATIONS

The Android application framework forces a structure on developers. It doesn't have a main() function for execution—instead have entry point through its components. An Android developer chooses from predefined component types depending on the component's purpose (such as interfacing with a user or storing data). One such example of android application is:-



III ACTIVITY AND INTERACTION OF COMPONENTS

Activities start each other, possibly passing and returning values. Only one activity on the system has keyboard and processing focus at a time; all others are suspended. In Figure 2, the interaction between components in the FriendTracker and FriendViewer applications and with components in applications defined as part of the base Android distribution. In each case, one component initiates communication with another. For simplicity, we call this inter-component communication (ICC). The available ICC actions depend on the target component. Each component type supports interaction specific to its type for example, when FriendViewer starts FriendMap, the FriendMap activity appears on the screen. Service components support start, stop, and bind actions, so the FriendTrackerControl activity, for instance, can start and stop the FriendTracker service that runs in the back-ground. The bind action establishes a connection between components, allowing the initiator to execute RPCs defined by the service. In our example, FriendTracker binds to the location manager in the system server.Once bound, FriendTracker invokes methods to register a callback that provides updates on the phone's location.Broadcast receiver and content provider components have unique forms of interaction. ICC targeted at a broadcast receiver occurs as an intent sent (broadcast) either explicitly to the component or, more commonly, to an action string the component sub-scribes to. For example, FriendReceiver subscribes to the developer-defined "FRIEND_NEAR" action string. FriendTracker broadcasts an intent to this action string when it determines that the phone is near a friend; the system then starts FriendReceiver and displays a message to the user. Content providers don't use intents—rather, they're ad-dressed via an authority string embedded in a special content Uniform Resource Identifier (URI) of the form content://<authority>/<table>/[<id>]. Components use this URI to perform a SQL query on a content provider, optionally including WHERE conditions via the query API.



As Figure 3, Android protects applications and data through a combination of two enforcement mechanisms, one at the system level and the other at the ICC level. ICC mediation defines the core security framework and

is this article's focus, but it builds on the guarantees provided by the underlying Linux system. In the general case, each application runs as a unique user identity, which lets Android limit the potential damage of programming flaws. ICC isn't limited by user and process boundaries. In fact, all ICC occurs via an I/O control *command on a special device node. Because the file must be world readable and writable for proper operation, the Linux system has no way of mediating ICC. Although user separation is straightforward and easily understood, controlling ICC is much more subtle and warrants careful consideration.In its simplest form, access to each component is restricted by assigning it an access permission label; this text string need not be unique. Developers assign applications collections of permission labels. When a component initiates ICC, the reference monitor looks at the permission labels assigned to its containing application and—if the target component's access permission label is in that collection—allows ICC establishment to proceed. If the label isn't in the collection, establishment is denied even if the components are in the same application.

IV ACCESS PERMISSION LOGIC

The developer assigns permission labels via the XML manifest file that accompanies every application package. In doing so, the developer defines the application's securitypolicy—that is, assigning permission labels to an application specifies its protection domain, whereas assigning permissions to the components in an application specifies an access policy to protect its resources. Because Android's policy enforcement is mandatory, as opposed todiscretionary, all permission labels are set at install time and can't change until the application is reinstalled. However, despite its MAC properties, Android's permission label model only restricts access to components and doesn't currently provide information flow guarantees, such as in domain type enforcement. Partially out of necessity and partially for convenience, the Google developers who designed Android incorporated several refinements to the basic security model, some of which have subtle side effects and make its overall security difficult to understand.

V BROADCAST INTENT PERMISSIONS

Components aren't the only resource that requires protection. In our example, the FriendTracker service broadcasts an intent to the FRIEND_NEAR action string to indicate the phone is physically near a friend's location. Although this event notification lets the Friend-Viewer application update the user, it potentially informs all installed applications of the phone's proximity. In this case, sending the unprotected intent is a privacy risk. More generally, unprotected intent broadcasts can unintentionally leak information to explicitly listening attackers. To combat this, the Android API for broadcasting intents optionally allows the developer to specify a permission label to restrict access to the intent object. The access permission label assignment to a broadcasted intent for example, sendBroadcast (intent, "perm. FRIEND_NEAR") restricts the set of applications that can receive it (in this example, only to applications containing the "perm.FRIEND_NEAR" per-mission label). This lets the developer control how information is disseminated, but this refinement pushes an application's security poli-cy into its source code. The manifest file therefore doesn't give the entire picture of the application's security.

VI CONTENT PROVIDER PERMISSIONS

In our application, the Friend Provider content provider stores friends' geographic coordinates. As a developer, we want our application to be the only one to update the contents but for other applications to be able to read them. Android allows such a security policy by modifying how access permissions are assigned to con-tent providers—instead of using one permission label, the developer can assign both read and write permissions. If the application performing a query with write side effects (INSERT, DELETE,UPDATE) doesn't have the write permission, the query is denied. The separate read and write permissions let the developer distinguish between data users and interactions that affect the data's integrity. Security-aware developers should define separate readand write permissions, even if the distinction isn't immediately apparent.

VII PERMISSION PROTECTION LEVELS

Early versions of the Android SDK let developers mark permission as "application" or "system." The default application level meant that any application requesting the permission label would receive it. Conversely, system permission labels were granted only to applications installed in /data/system (as opposed to /data/app, which is independent of label assignment). The likely reason is that only system applications should be able to perform operations such as interfacing directly with the telephony API. The v0.9r1 SDK (August 2008) extended the early model into four protection levels for permission labels, with the meta information specified in the manifest of the package defining the permission. "Normal" permissions act like the old application permissions and are granted to any application that requests them in its manifest; "dangerous" permissions are granted only after user confirmation. Similar to security checks in popular desktop operating systems such as Microsoft Vista's user account control (UAC), when an application is installed, the user sees a screen listing short descriptions of requested dangerous permissions along with OK and Cancel buttons. Here, the user has the opportunity to accept all permission requests or deny the installation. "Signature" permissions are granted only to applications signed by the same developer key as the package defining the permission (application signing became mandatory in the v0.9r1 SDK). Finally, "signature or system" permissions act like signature permissions but exist for legacy compatibility with the older system permission type.

Normal	Permissions for app 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.
Signature or System	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 is to help integratesystem builds and won't typically beNeeded by developers.

The new permission protection levels provide a means of controlling how developers assign permission labels. Signature permissions ensure that only the framework developer can use the specific functionality (only Google applications can directly interface the telephony API, for example). Dangerous permissions give the end user some say in the permission granting process. However, the permission protection levels express only trivial granting policies.Making a permission "dangerous" helps, but it depends on the user understanding the security implications.

VIII SECURITY RESPONSIBILITIES FOR 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. One ofthe trickiest big-picture things to understand about An-droid 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. Android requires developers to sign their code.Android codesigning 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 signing however, as the actual identity of the developer isn't necessarily being validated by a third party to the de-vice's user.

IX CONCLUSION

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. When installing new software, users have a chance to think about what they are doing and to decide to trust software based on re-views, the developer's reputation, and the permissions required. 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. 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 to the one shown in 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.

REFERENCES

[1] J.P. Anderson, Computer Security Technology Planning Study, tech. report ESDTR-73-51, Mitre, Oct. 1972.

[2] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman, "Protection in Operating Systems," Comm. ACM vol. 19, no.8, 1976, pp. 461–471.

[3] L. Badger et al., "Practical Domain and Type Enforcement for UNIX," Proc. IEEE Symp. Security and Privacy, IEEE CS Press, 1995, pp. 66–77.

[4] Google Inc. (2008, August 29). Security and Permissions in Android. Retrieved August 30, 2008, from Android - An Open Handset Alliance Project.

[5] W. Enck, M. Ongtang, and P. McDaniel, Miti gatingAndroid Software Misuse before It Happens, tech.report NAS-TR-0094-2008, Network and Security Research Ctr., Dept. C S E, Pennsylvania State Univ., Nov. 2008.