Focus

Understanding Android Security

William Enck, Machigar Ongtang, and Patrick McDaniel
Pennsylvania State University

The next generation of open operating systems won't be on desktops or mainframes but on the small mobile devices we carry every day. The openness of these new environments will lead to new applications and markets and will enable greater integration with existing online services. However, as the importance of the data and services our cell phones support increases, so too do the opportunities for vulnerability. It's essential that this next generation of platforms provide a comprehensive and usable security infrastructure.

Developed by the Open Handset Alliance (visibly led by Google), 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. Although the Android SDK has been available since late 2007, the first publicly available Android-ready "G1" phone debuted in late October 2008. Since then, Android's growth has been phenomenal: T-Mobile's G1 manufacturer HTC estimates shipment volumes of more than 1 million phones by the end of 2008, and industry insiders expect public adoption to increase steeply in 2009. Many other cell phone providers have either promised or plan to support it in the near future.

A large community of developers has organized around Android, and many new products and applications are now available for it. One of Android's chief selling points is that it lets developers seamlessly extend online services to phones. The most visible example of this feature is—unsurprisingly—the tight integration of Google's Gmail, Calendar, and Contacts Web applications with system utilities. Android users simply supply a username and password, and their phones automatically synchronize with Google services. Other vendors are rapidly adapting their existing instant messaging, social networks, and gaming services to Android, and many enterprises are looking for ways to integrate their own internal operations (such as inventory management, purchasing, receiving, and so forth) into it as well.

Traditional desktop and server operating systems have struggled to securely integrate such personal and business applications and services on a single platform; although doing so on a mobile platform such as Android remains nontrivial, many researchers hope it provides a clean slate devoid of the complications that legacy software can cause. Android doesn't officially support applications developed for other platforms: applications execute on top of a Java middleware layer running on an embedded Linux kernel, so developers wishing to port their application to Android must use its custom user interface environment. Additionally, Android restricts application interaction to its special APIs by running each application as its own user identity. Although this controlled interaction has several beneficial security features, our experiences developing Android applications have revealed that designing secure applications isn't always straightforward. Android uses a simple permission label assignment model to restrict access to resources and other applications, but for reasons of necessity and convenience, its designers have added several potentially confusing refinements as the system has evolved.

This article attempts to unmask the complexity of Android security and note some possible development pitfalls that occur when defining an application's security. We conclude by attempting to draw some lessons and identify opportunities for future enhancements that should aid in clarity and correctness.

Android Applications

The Android application framework forces a structure on developers. It doesn't have a `main()` function or single entry point for

execution—instead, developers must design applications in terms of *components*.

Example Application

We developed a pair of applications to help describe how Android applications operate. Interested readers can download the source code from our Web site (http://siis.cse.psu.edu/android_sec_tutorial.html).

Let's consider a location-sensitive social networking application for mobile phones in which users can discover their friends' locations. We split the functionality into two applications: one for tracking friends and one for viewing them. As Figure 1 shows, the FriendTracker application consists of components specific to tracking friend locations (for example, via a Web service), storing geographic coordinates, and sharing those coordinates with other applications. The user then uses the FriendViewer application to retrieve the stored geographic coordinates and view friends on a map.

Both applications contain multiple components for performing their respective tasks; the components themselves are classified by their *component types*. An Android developer chooses from predefined component types depending on the component's purpose (such as interfacing with a user or storing data).

Component Types

Android defines four component types:

- *Activity* components define an application's user interface. Typically, an application developer defines one activity per "screen." 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.
- *Service* components perform background processing. When an activity needs to perform some operation that must continue after the user interface disappears (such as download a file or play music), it commonly starts a service specifically designed for that action. The developer can also use services as application-specific daemons, possibly starting on boot. Services often define an interface for Remote Procedure Call (RPC) that other system components can use to send commands and retrieve data, as well as register callbacks.
- *Content provider* components store and share data using a relational database interface. Each content provider has an associated "authority" describing the content it contains. Other components use the authority name as a handle to perform SQL queries (such as `SELECT`, `INSERT`, or `DELETE`) to read and write content. Although content providers typically store values in database records, data retrieval is implementation-specific—for example, files are also shared through content provider interfaces.
- *Broadcast receiver* components act as mailboxes for messages from other applications. Commonly, application code broadcasts messages to an implicit destination. Broadcast receivers thus subscribe to such destinations to receive the messages sent to it. Application code can also address a broadcast receiver explicitly by including the namespace assigned to its containing application.

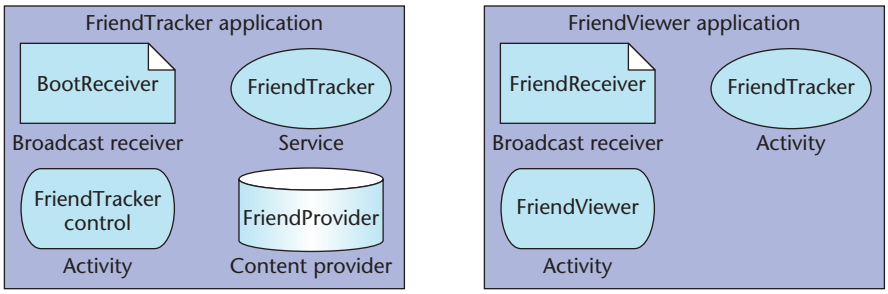

Figure 1. Example Android application. The FriendTracker and FriendViewer applications consist of multiple components of different types, each of which provides a different set of functionalities. Activities provide a user interface, services execute background processing, content providers are data storage facilities, and broadcast receivers act as mailboxes for messages from other applications.

Figure 1 shows the FriendTracker and FriendViewer applications containing the different component types. The developer specifies components using a manifest file (also used to define policy as described later). There are no restrictions on the number of components an application defines for each type, but as a convention, one component has the same name as the application. Frequently, this is an activity, as in the FriendViewer application. This activity usually indicates the primary activity that the system application launcher uses to start the user interface; however, the specific activity chosen on launch is marked by meta information in the manifest. In the FriendTracker application, for example, the FriendTrackerControl activity is marked as the main user interface entry point. In this case, we reserved the name "FriendTracker" for the service component performing the core application logic.

The FriendTracker application contains each of the four component types. The FriendTracker service polls an external service to discover friends' locations. In our example code, we generate loca-

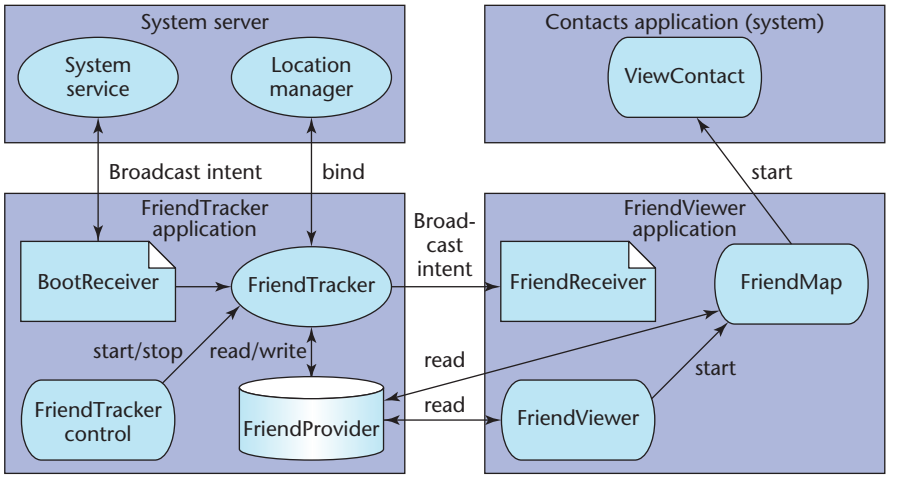

Figure 2. Component interaction. Android's application-level interactions let the FriendTracker and FriendViewer applications communicate with each other and system-provided applications. Interactions occur primarily at the component level.

tions randomly, but extending the component to interface with a Web service is straightforward. The FriendProvider content provider maintains the most recent geographic coordinates for friends, the FriendTrackerControl activity defines a user interface for starting and stopping the tracking functionality, and the BootReceiver broadcast receiver obtains a notification from the system once it boots (the application uses this to automatically start the FriendTracker service).

The FriendViewer application is primarily concerned with showing information about friends' locations. The FriendViewer activity lists all friends and their geographic coordinates, and the FriendMap activity displays them on a map. The FriendReceiver broadcast receiver waits for messages that indicate the physical phone is near a particular friend and displays a message to the user upon such an event. Although we could have placed these components within the FriendTracker application, we created a separate application to demonstrate cross-application communication. Additionally, by separating the tracking and user interface logic, we can create alternative user interfaces with different displays and features—that is, many applications can reuse the logic performed in FriendTracker.

Component Interaction

The primary mechanism for component interaction is an *intent*, which is simply a message object containing a destination component address and data. The Android API defines methods that accept intents, and uses that information to start activities (`startActivity(Intent)`), start services (`startService(Intent)`), and broadcast messages (`sendBroadcast(Intent)`). The invocation of these methods tells the Android framework to begin executing code in the target application. This process of intercomponent communication is known as an *action*. Simply put, an intent object defines the "intent" to perform an "action."

One of Android's most powerful features is the flexibility allowed by its intent-addressing mechanism. Although developers can uniquely address a target component using its application's namespace, they can also specify an implicit name. In the latter case, the system determines the best component for an action by considering the set of installed applications and user choices. The implicit name is called an *action string* because it specifies the type of requested action—for example, if the `"VIEW"` action string is specified in an intent with data fields pointing to an image file, the system will direct the intent to the preferred image viewer. Developers also use action strings to broadcast a message to a group of broadcast receivers. On the receiving end, developers use an *intent filter* to subscribe to specific action strings. Android includes additional destination resolution rules, but action strings with optional data types are the most common.

Figure 2 shows 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). In many ways, ICC is analogous to inter-process communication (IPC) in Unix-based systems. To the developer, ICC functions identically regardless of whether the target is in the same or different application, with the exception of the security rules defined later in this article.

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 background. 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.

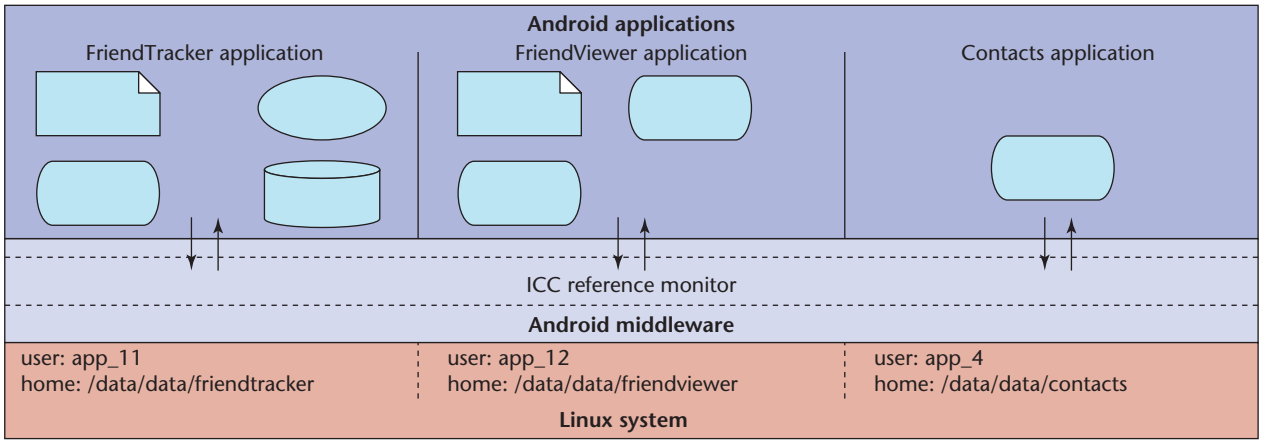

Figure 3. Protection. Security enforcement in Android occurs in two places: each application executes as its own user identity, allowing the underlying Linux system to provide system-level isolation; and the Android middleware contains a reference monitor that mediates the establishment of inter-component communication (ICC). Both mechanisms are vital to the phone's security, but the first is straightforward to implement, whereas the second requires careful consideration of both mechanism and policy.

Once bound, FriendTracker invokes methods to register a callback that provides updates on the phone's location. Note that if a service is currently bound, an explicit "stop" action won't terminate the service until all bound connections are released.

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 subscribes 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 addressed via an authority string embedded in a special content Uniform Resource Identifier (URI) of the form `content://<authority>/<table>/[<id>]`. Here, `<table>` indicates a table in the content provider, and `<id>` optionally specifies a record in that table. Components use this URI to perform a SQL query on a content provider, optionally including `WHERE` conditions via the query API.

Security Enforcement

As Figure 3 shows, 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. For example, the Web browser vulnerability discovered recently after the official release of T-Mobile G1 phones only affected the Web browser (http://securityevaluators.com/content/case-studies/android/index.jsp). Because of this design choice, the exploit couldn't affect other applications or the system. A similar vulnerability in Apple's iPhone gave way to the first "jail breaking" technique, which let users replace parts of the underlying system, but would also have enabled a network-based adversary to exploit this flaw (http://securityevaluators.com/content/case-studies/iphone/index.jsp).

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, `/dev/binder`. 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.

As the central point of security enforcement, the Android middleware mediates all ICC establishment by reasoning about labels assigned to applications and components. A reference monitor[1] provides mandatory access control (MAC) enforcement of how applications access components. 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

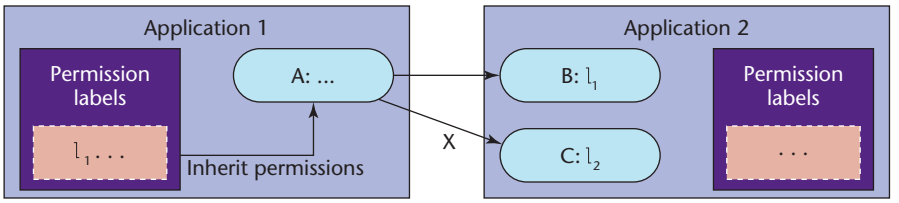

Figure 4. Access permission logic. The Android middleware implements a reference monitor providing mandatory access control (MAC) enforcement about how applications access components. The basic enforcement model is the same for all component types. Component A's ability to access components B and C is determined by comparing the access permission labels on B and C to the collection of labels assigned to application 1.

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. Figure 4 depicts this 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 security policy—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 to discretionary,[2] 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.[3]

Security Refinements

Android's security framework is based on the label-oriented ICC mediation described thus far, but our description is incomplete. 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. The rest of this section provides an exhaustive list of refinements we identified as of the v1.0r1 SDK release.

Public vs. Private Components

Applications often contain components that another application should never access—for example, an activity designed to return a user-entered password could be started maliciously. Instead of defining an access permission, the developer could make a component private by either explicitly setting the `exported` attribute to false in the manifest file or letting Android infer if the component should be private from other attributes in its manifest definition.

Private components simplify security specification. By making a component private, the developer doesn't need to worry which permission label to assign it or how another application might acquire that label. Any application can access components that aren't explicitly assigned an access permission, so the addition of private components and inference rules (introduced in the v0.9r1 SDK release, August 2008) significantly reduces the attack surface for many applications. However, the developer must be careful when allowing Android to determine if a component is private. Security-aware developers should always explicitly define the `exported` attribute for components intended to be private.

Implicitly Open Components

Developers frequently define intent filters on activities to indicate that they can handle certain types of action/data combinations. Recall the example of how the system finds an image viewer when an intent specifying the `VIEW` action and an image reference is passed to the "start activity" API. In this case, the caller can't know beforehand (much less at development time) what access permission is required. The developer of the target activity can permit such functionality by *not* assigning an access permission to it—that is, if a public component doesn't explicitly have an access permission listed in its manifest definition, Android permits any application to access it.

Although this default allows policy specification enables functionality and ease of development, it can lead to poor security practices and is contrary to Saltzer and Schroeder's principle of fail-safe defaults.[4] Referring back to our example FriendViewer application, if the FriendReceiver broadcast receiver isn't assigned an access permission, any unprivileged installed application can forge a `FRIEND_NEAR` message, which represents a significant security concern for applications making decisions based on information passed via the intent. As a general practice, security-aware developers should always assign access permissions to public components—in fact, they should have an explicit reason for not assigning one. All inputs should be scrutinized under these conditions.

Broadcast Intent Permissions

Components aren't the only re-

source that requires protection. In our FriendTracker 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 FriendViewer 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"` permission label). This lets the developer control how information is disseminated, but this refinement pushes an application's security policy into its source code. The manifest file therefore doesn't give the entire picture of the application's security.

Content Provider Permissions

In our FriendTracker application, the FriendProvider 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 content 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 read and write permissions, even if the distinction isn't immediately apparent.

Service Hooks

Although they weren't explicitly identified, the FriendTracker service defines RPC interfaces: `isTracking()` and `addNickname(String)`. The `isTracking()` method doesn't change the service's running state; it simply returns whether FriendTracker is currently tracking locations. However, `addNickname(String)` does modify the running state by telling FriendTracker to start tracking another friend. Due to this state modification, the developer might want to differentiate access to the two interfaces. Unfortunately, Android only lets the developer assign one permission label to restrict starting, stopping, and binding to a service. Under this model, any application that can start or stop FriendTracker can also tell it to monitor new friends. To address this, Android provides the `checkPermission()` method, which lets developers arbitrarily extend the reference monitor with a more restrictive policy. In effect, these service hooks let the developer write code to perform custom runtime security.

Service hooks provide much greater flexibility when defining access policy—in fact, several services provided in the base Android distribution use them. However, like broadcast intent permissions, service hooks move policy into the application code, which can cloud application security.

Protected APIs

Not all system resources (such as the network, camera, and microphone) are accessed through components—instead, Android provides direct API access. In fact, the services that provide indirect access to hardware often use APIs available to third-party applications. Android protects these sensitive APIs with additional permission label checks: an application must declare a corresponding permission label in its manifest file to use them. Bitfrost takes a similar approach (the "one laptop per child" security model[5]), but it allows controlled permission change after installation.

By protecting sensitive APIs with permissions, Android forces an application developer to declare the desire to interface with the system in a specific way. Consequently, vulnerable applications can't gain unknown access if exploited. The most commonly encountered protected API is for network connections—for example, the FriendViewer application requires Internet access for map information, so it must declare the `INTERNET` permission label. In general, protected APIs make an application's protection domain much clearer because the policy is defined in the manifest file.

Permission Protection Levels

Early versions of the Android SDK let developers mark a 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.

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—for example, FriendTracker defines the permission label associated with the `FRIEND_NEAR` intent broadcast as dangerous. However, the permission protection levels express only trivial granting policies. A third-party application still doesn't have much control if it wants another developer to use the permission label. Making a permission "dangerous" helps, but it depends on the user understanding the security implications.

Pending Intents

All the security refinements described up to this point fall within the realm of an extension to the basic MAC model. The v0.9r1 SDK release (August 2008) introduced the concept of a "pending intent," which is rather straightforward: a developer defines an intent object as normally done to perform an action (to start an activity, for example). However, instead of performing the action, the developer passes the intent to a special method that creates a PendingIntent object corresponding to the desired action. The PendingIntent object is simply a reference pointer that can pass to another application, say, via ICC. The recipient application can modify the original intent by filling in unspecified address and data fields and specify when the action is invoked. The invocation itself causes an RPC with the original application, in which the ICC executes with all its permissions.

Pending intents allow applications included with the framework to integrate better with third-party applications. Used correctly, they can improve an application's security—in fact, several Android APIs require pending intents, such as the location manager, which has a "proximity update" feature that notifies an application via intent broadcast when a geographic area is entered or exited. The pending intent lets an application direct the broadcast to a specific private broadcast receiver. This prevents forging without the need to coordinate permissions with system applications.

However, pending intents diverge from Android's MAC model by introducing *delegation*. By using a pending intent, an application delegates the ability to influence intent contents and the time of performing the action. Historically, certain delegation techniques have substantial negative effects on the tractability of policy evaluation.[6]

URI Permissions

The v1.0r1 SDK release (September 2008) introduced another delegation mechanism—URI permissions. Recall that Android uses a special content URI to address content providers, optionally specifying a record within a table. The developer can pass such a URI in an intent's data field—for example, an intent can specify the `VIEW` action and a content URI identifying an image file. If used to start an activity, the system will choose a component in a different application to view the image. If the target application doesn't have read permission to the content provider containing the image file, the developer can use a URI permission instead. In this case, the developer sets a read flag in the intent that grants the target application access to the specific intent-identified record.

URI permissions are essentially capabilities for database records. Although they provide *least privilege*[4] access to content providers, the addition of a new delegation mechanism further diverges from the original MAC model. As mentioned with pending intents, delegation potentially impacts the tractability of policy analysis. A content provider must explicitly allow URI permissions, therefore they require the data store developer's participation.

Lessons in Defining Policy

Our experiences working with Android security policy revealed that it begins with a relatively easy to understand MAC enforcement model, but the number and subtlety of refinements make it difficult for someone to discover an application's policy simply by looking at it. Some refinements push policy into the application code. Others add del-

egation, which mixes discretionary controls into the otherwise typical MAC model. This situation makes gathering a firm grasp on Android's security model nontrivial.

Even with all the refinements, holistic security concerns have gone largely unaddressed. First, what does a permission label really mean? The label itself is merely a text string, but its assignment to an application provides access to potentially limitless resources. Second, how do you control access to permission labels? Android's permission protection levels provide some control, but more expressive constraints aren't possible. As a purposefully simple example, should an application be able to access both the microphone and the Internet?

Will granting a permission break the phone's security? Do the access permission assignments to an application's components put the phone or the application at risk? Android currently provides no means of answering these questions.

We developed an enhanced installer and security framework to answer a variant of these questions—namely, "does an application break some larger phone-wide security policy?" Our tool, called Kirin,[7] extracts an application's security policy from its manifest file to determine if the requested permissions and component permission assignments are consistent with the stakeholders' definition of a secure phone (stakeholders in this context range from the network provider to an enterprise to a user). Kirin uses a formalized model of the policy mechanisms described in this article to generate automated proofs of compliance using a Prolog engine running on the phone. If an application's policy isn't compliant, it won't be installed. By defining security requirements in logic, which we call *policy invariants*, we significantly reduce the need to defer install-time decisions to the user—that is, the policy invariants capture the appropriate response. We've successfully used Kirin to identify multiple vulnerabilities in the base applications provided with Android and have subsequently established an ongoing relationship with Google to fix the flaws and further investigate Android's security via Kirin.

In many ways, Android provides more comprehensive security than other mobile phone platforms. However, learning how to effectively use its building blocks isn't easy. We're only beginning to see different types of applications, and as Android matures, we'll learn how faulty application policy affects the phone's security. We believe that tools such as Kirin and those like it will help mold Android into the secure operating system needed for next-generation computing platforms. □

References

1. J.P. Anderson, *Computer Security Technology Planning Study*, tech. report ESD-TR-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. J. Saltzer and M. Schroeder, "The Protection of Information in Computer Systems," *Proc. IEEE*, vol. 63, no. 9, 1975, pp. 1278–1308.
5. I. Krstic and S.L. Garfinkel, "Bitfrost: The One Laptop per Child Security Model," *Proc. Symp. Usable Privacy and Security*, ACM Press, 2007, pp. 132–142.
6. N. Li, B.N. Grosof, and J. Feigenbaum, "Delegation Logic: A Logic-Based Approach to Distributed Authorization," *ACM Trans. Information and System Security*, vol. 6, no.1, 2003, pp. 128–171.
7. W. Enck, M. Ongtang, and P. McDaniel, *Mitigating Android Software Misuse Before It Happens*, tech.reportNAS-TR-0094-2008, Network and Security Research Ctr., Dept. Computer Science and Eng., Pennsylvania State Univ., Nov. 2008.

William Enck *is a PhD candidate in the Systems and Internet Infrastructure Security (SIIS) Laboratory in the Department of Computer Science and Engineering at Pennsylvania State University. His research interests include operating systems security, telecommunications security, and systems and network security. Enck has an MS in computer science and engineering from Pennsylvania State University. Contact him at 344 Information Sciences and Technology Building, University Park, PA 16802; Email: enck@cse.psu.edu.*

Machigar Ongtang *is a PhD candidate in the Systems and Internet Infrastructure Security (SIIS) Laboratory in the Department of Computer Science and Engineering at Pennsylvania State University. Her research interests include pervasive computing, context-aware security, and telecommunications security. Ongtang has an MSc in information technology for manufacture from the University of Warwick, UK. Contact her at 344 Information Sciences and Technology Building, University Park, PA 16802; Email: ongtang@cse.psu.edu.*

Patrick McDaniel *is a co-director of the Systems and Internet Infrastructure Security (SIIS) Laboratory and associate professor in the Department of Computer Science and Engineering at Pennsylvania State University. His research interests include systems and network security, telecommunications security, and security policy. McDaniel has a PhD in computer science from the University of Michigan. Contact him at 360A Information Sciences and Technology Building, University Park, PA 16802; Email: mcdaniel@cse.psu.edu.*