Analyzing Inter-Application Communication in Android

Erika Chin Adrienne Porter Felt Kate Greenwood David Wagner

University of California, Berkeley
Berkeley, CA, USA
{emc, apf, kate_eli, daw}@cs.berkeley.edu

ABSTRACT

Modern smartphone operating systems support the development of third-party applications with open system APIs. In addition to an open API, the Android operating system also provides a rich inter-application message passing system. This encourages inter-application collaboration and reduces developer burden by facilitating component reuse. Unfortunately, message passing is also an application attack surface. The content of messages can be sniffed, modified, stolen, or replaced, which can compromise user privacy. Also, a malicious application can inject forged or otherwise malicious messages, which can lead to breaches of user data and violate application security policies.

We examine Android application interaction and identify security risks in application components. We provide a tool, ComDroid, that detects application communication vulnerabilities. ComDroid can be used by developers to analyze their own applications before release, by application reviewers to analyze applications in the Android Market, and by end users. We analyzed 20 applications with the help of ComDroid and found 34 exploitable vulnerabilities; 12 of the 20 applications have at least one vulnerability.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.4 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Android, message passing, Intents, mobile phone security

1. INTRODUCTION

Over the past decade, mobile phones have evolved from simple devices used for phone calls and SMS messages to sophisticated devices that can run third-party software. Phone owners are no longer limited to the simple address book and other basic capabilities provided by the operating system and phone manufacturer. They are free to customize their phones by installing third-party applications of their choosing. Mobile phone manufacturers support third-party application developers by providing development platforms and software stores (e.g., Android Market, Apple App Store [1, 3]) where developers can distribute their applications.

Android's application communication model further promotes the development of rich applications. Android developers can leverage existing data and services provided by other applications while still giving the impression of a single, seamless application. For example, a restaurant review application can ask other applications to display the restaurant's website, provide a map with the restaurant's location, and call the restaurant. This communication model reduces developer burden and promotes functionality reuse. Android achieves this by dividing applications into components and providing a message passing system so that components can communicate within and across application boundaries.

Android's message passing system can become an attack surface if used incorrectly. In this paper, we discuss the risks of Android message passing and identify insecure developer practices. If a message sender does not correctly specify the recipient, then an attacker could intercept the message and compromise its confidentiality or integrity. If a component does not restrict who may send it messages, then an attacker could inject malicious messages into it.

We have seen numerous malicious mobile phone applications in the wild. For example, SMS Message Spy Pro disguises itself as a tip calculator and forwards all sent and received SMS messages to a third party [25]; similarly, MobiStealth records SMS messages, call history, browser history, GPS location, and more [26, 4]. This is worrisome because users rely on their phones to perform private and sensitive tasks like sending e-mail, taking pictures, and performing banking transactions. It is therefore important to help developers write secure applications that do not leak or alter user data in the presence of an adversary.

We examine the Android communication model and the security risks it creates, including personal data loss and corruption, phishing, and other unexpected behavior. We present ComDroid, a tool that analyzes Android applications to detect potential instances of these vulnerabilities. We used ComDroid to analyze 20 applications and found 34 vulnerabilities in 12 of the applications. Most of these vulnerabilities stem from the fact that Intents can be used for both intra- and inter-application communication, so we provide recommendations for changing Android to help developers distinguish between internal and external messages.

MobiSys'11, June 28–July 1, 2011, Bethesda, Maryland, USA.

2. ANDROID OVERVIEW

Android's security model differs significantly from the standard desktop security model. Android applications are treated as mutually distrusting principals; they are isolated from each other and do not have access to each others' private data. We provide an overview of the Android security model and inter-application communication facilities next.

Although other smartphone platforms have a similar security model, we focus on Android because it has the most sophisticated application communication system. The complexity of Android's message passing system implies it has the largest attack surface. In Section 6, we compare and contrast Android to other mobile operating systems.

2.1 Threat Model

The Android Market contains a wide array of third-party applications, and a user may install applications with varying trust levels. Users install applications from unknown developers alongside trusted applications that handle private information such as financial data and personal photographs. For example, a user might install both a highly trusted banking application and a free game application. The game should not be able to obtain access to the user's bank account information.

Under the Android security model, all applications are treated as potentially malicious. Each application runs in its own process with a low-privilege user ID, and applications can only access their own files by default. These isolation mechanisms aim to protect applications with sensitive information from malware.

Despite their default isolation, applications can optionally communicate via message passing. Communication can become an attack vector. If a developer accidentally exposes functionality, then the application can be tricked into performing an undesirable action. If a developer sends data to the wrong recipient, then it might leak sensitive data. In this paper, we consider how applications can prevent these kinds of communication-based attacks.

In addition to providing inter-application isolation, the Android security model protects the system API from malicious applications. By default, applications do not have the ability to interact with sensitive parts of the system API; however, the user can grant an application additional permissions during installation. We do not consider attacks on the operating system; instead, we focus on securing applications from each other.

2.2 Intents

Android provides a sophisticated message passing system, in which *Intents* are used to link applications. An Intent is a message that declares a recipient and optionally includes data; an Intent can be thought of as a self-contained object that specifies a remote procedure to invoke and includes the associated arguments. Applications use Intents for both inter-application communication and intra-application communication. Additionally, the operating system sends Intents to applications as event notifications. Some of these event notifications are system-wide events that can only be sent by the operating system. We call these messages *system broadcast Intents*.

Intents can be used for *explicit* or *implicit* communication. An explicit Intent specifies that it should be delivered to a particular application specified by the Intent, whereas an implicit Intent requests delivery to any application that supports a desired operation. In other words, an explicit Intent identifies the intended recipient by name, whereas an implicit Intent leaves it up to the Android platform to determine which application(s) should receive the Intent. For example, consider an application that stores contact information. When the user clicks on a contact's street address, the contacts application needs to ask another application to display a map of that location. To achieve this, the contacts application could send an explicit Intent directly to Google Maps, or it could send an implicit Intent that would be delivered to any application that says it provides mapping functionality (e.g., Yahoo! Maps or Bing Maps). Using an explicit Intent guarantees that the Intent is delivered to the intended recipient, whereas implicit Intents allow for late runtime binding between different applications.

2.3 Components

Intents are delivered to application *components*, which are logical application building blocks. Android defines four types of components:

- *Activities* provide user interfaces. Activities are started with Intents, and they can return data to their invoking components upon completion. All visible portions of applications are Activities.

- *Services* run in the background and do not interact with the user. Downloading a file or decompressing an archive are examples of operations that may take place in a Service. Other components can *bind* to a Service, which lets the binder invoke methods that are declared in the target Service's interface. Intents are used to start and bind to Services.

- *Broadcast Receivers* receive Intents sent to multiple applications. Receivers are triggered by the receipt of an appropriate Intent and then run in the background to handle the event. Receivers are typically short-lived; they often relay messages to Activities or Services. There are three types of broadcast Intents: normal, sticky, and ordered. Normal broadcasts are sent to all registered Receivers at once, and then they disappear. Ordered broadcasts are delivered to one Receiver at a time; also, any Receiver in the delivery chain of an ordered broadcast can stop its propagation. Broadcast Receivers have the ability to set their priority level for receiving ordered broadcasts. Sticky broadcasts remain accessible after they have been delivered and are re-broadcast to future Receivers.

- *Content Providers* are databases addressable by their application-defined URIs. They are used for both persistent internal data storage and as a mechanism for sharing information between applications.

Intents can be sent between three of the four components: Activities, Services, and Broadcast Receivers. Intents can be used to start Activities; start, stop, and bind Services; and broadcast information to Broadcast Receivers. (Table 1 shows relevant method signatures.) All of these forms of communication can be used with either explicit or implicit Intents. By default, a component receives only internal application Intents (and is therefore not externally invocable).

To Receiver	`sendBroadcast(Intent i)`
	`sendBroadcast(Intent i, String rcvrPermission)`
	`sendOrderedBroadcast(Intent i, String recvrPermission, BroadcastReceiver receiver, ...)`
	`sendOrderedBroadcast(Intent i, String recvrPermission)`
	`sendStickyBroadcast(Intent i)`
	`sendStickyOrderedBroadcast(Intent i, BroadcastReceiver receiver, ...)`

To Activity	`startActivity(Intent i)` `startActivityForResult(Intent i, int requestCode)`
To Service	`startService(Intent i)` `bindService(Intent i, ServiceConnection conn, int flags)`

Table 1: A non-exhaustive list of Intent-sending mechanisms

2.4 Component Declaration

To receive Intents, a component must be declared in the application *manifest*. A manifest is a configuration file that accompanies the application during installation. A developer uses the manifest to specify what external Intents (if any) should be delivered to the application's components.

2.4.1 Exporting a Component

For a Service or Activity to receive Intents, it must be declared in the manifest. (Broadcast Receivers can be declared in the manifest or at runtime.) A component is considered *exported*, or public, if its declaration sets the `EXPORTED` flag or includes at least one *Intent filter*. Exported components can receive Intents from other applications, and Intent filters specify what type of Intents should be delivered to an exported component.

Android determines which Intents should be delivered to an exported component by matching each Intent's fields to the component's declaration. An Intent can include a component name, an action, data, a category, extra data, or any subset thereof. A developer sends an explicit Intent by specifying a recipient component name; the Intent is then delivered to the component with that name. Implicit Intents lack component names, so Android uses the other fields to identify an appropriate recipient.

An Intent filter can constrain incoming Intents by action, data, and category; the operating system will match Intents against these constraints. An *action* specifies a general operation to be performed, the *data* field specifies the type of data to operate on, and the *category* gives additional information about the action to execute. For example, a component that edits images might define an Intent filter that states it can accept any Intent with an `EDIT` action and data whose MIME type is `image/*`. For a component to be an eligible recipient of an Intent, it must have specified each action, category, and data contained in the Intent in its own Intent filter. A filter can specify more actions, data, and categories than the Intent, but it cannot have less.

Multiple applications can register components that handle the same type of Intent. This means that the operating system needs to decide which component should receive the Intent. Broadcast Receivers can specify a priority level (as an attribute of its Intent filter) to indicate to the operating system how well-suited the component is to handle an Intent. When ordered broadcasts are sent, the Intent filter with the highest priority level will receive the Intent first. Ties among Activities are resolved by asking the user to select the preferred application (if the user has not already set a default selection). Competition between Services is decided by randomly choosing a Service.

It is important to note that Intent filters are not a security mechanism. A sender can assign any action, type, or category that it wants to an Intent (with the exception of certain actions that only the system can send), or it can bypass the filter system entirely with an explicit Intent. Conversely, a component can claim to handle any action, type, or category, regardless of whether it is actually well-suited for the desired operation.

2.4.2 Protection

Android restricts access to the system API with permissions, and applications must request the appropriate permissions in their manifests to gain access to protected API calls. Applications can also use permissions to protect themselves. An application can specify that a caller must have a certain permission by adding a permission requirement to a component's declaration in the manifest or setting a default permission requirement for the whole application. Also, the developer can add permission checks throughout the code. Conversely, a broadcast Intent sender can limit who can receive the Intent by requiring the recipient to have a permission. (This protection is only available to broadcast Intents and not available to Activity or Service Intents.) Applications can make use of existing Android permissions or define new permissions in their manifests.

All permissions have a protection level that determines how difficult the permission is to acquire. There are four protection levels:

- *Normal* permissions are granted automatically.
- *Dangerous* permissions can be granted by the user during installation. If the permission request is denied, then the application is not installed.
- *Signature* permissions are only granted if the requesting application is signed by the same developer that defined the permission. Signature permissions are useful for restricting component access to a small set of applications trusted and controlled by the developer.
- *SignatureOrSystem* permissions are granted if the application meets the Signature requirement or if the application is installed in the system applications folder. Applications from the Android Market cannot be installed into the system applications folder. System applications must be pre-installed by the device manufacturer or manually installed by an advanced user.

Applications seeking strong protection can require that callers hold permissions from the higher categories. For example, the `BRICK` permission can be used to disable a device. It is a Signature-level permission defined by the operating system, which means that it will only be granted to applications with the same signature as the operating system (i.e., applications signed with the phone manufacturer's signature). If a developer were to protect her component with the `BRICK` permission, then only an application with that permission (e.g., a Google-made application) could use that component. In contrast, a component protected with a Normal permission is essentially unprotected because any application can easily obtain the permission.

3. INTENT-BASED ATTACK SURFACES

We examine the security challenges of Android communication from the perspectives of Intent senders and Intent recipients. In Section 3.1, we discuss how sending an Intent to the wrong application can leak user information. Data can be stolen by eavesdroppers and permissions can be accidentally transferred between applications. In Section 3.2, we consider vulnerabilities related to receiving *external* Intents, i.e., Intents coming from other applications. If a component is accidentally made public, then external applications can invoke its components in surprising ways or inject malicious data into it. We summarize guidelines for secure Android communication in Section 3.3.

Throughout our discussion of component security, we focus our attention on exported components. Non-exported components are not accessible to other applications and thus are not subject to the attacks we present here. We also exclude exported components and broadcast Intents that are protected with permissions that other applications cannot acquire. As explained in Section 2.4.2, Normal and Dangerous permissions do not offer components or Intents very strong protection: Normal permissions are granted automatically, and Dangerous permissions are granted with user approval. Signature and SignatureOrSystem permissions, however, are very difficult to obtain. We consider components and broadcast Intents that are protected with Signature or SignatureOrSystem permissions as private.

3.1 Unauthorized Intent Receipt

When an application sends an implicit Intent, there is no guarantee that the Intent will be received by the intended recipient. A malicious application can intercept an implicit Intent simply by declaring an Intent filter with all of the actions, data, and categories listed in the Intent. The malicious application then gains access to all of the data in any matching Intent, unless the Intent is protected by a permission that the malicious application lacks. Interception can also lead to control-flow attacks like denial of service or phishing. We consider how attacks can be mounted on Intents intended for Broadcast Receivers, Activities, and Services. We also discuss special types of Intents that are particularly dangerous if intercepted.

3.1.1 Broadcast Theft

Broadcasts can be vulnerable to passive eavesdropping or active denial of service attacks (Figure 1). An eavesdropper can silently read the contents of a broadcast Intent without interrupting the broadcast. Eavesdropping is a risk whenever an application sends a public broadcast. (A public broadcast is an implicit Intent that is not protected by a Signature or SignatureOrSystem permission.) A malicious Broadcast Receiver could eavesdrop on all public broadcasts from all applications by creating an Intent filter that lists all possible actions, data, and categories. There is no indication to the sender or user that the broadcast has been read. Sticky broadcasts are particularly at risk for eavesdropping because they persist and are re-broadcast to new Receivers; consequently, there is a large temporal window for a sticky broadcast Intent to be read. Additionally, sticky broadcasts cannot be protected by permissions.

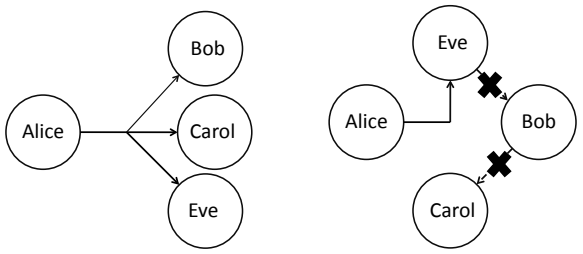

Figure 1: Broadcast Eavesdropping (left): Expected recipients Bob and Carol receive the Intent, but so does Eve. Broadcast Denial of Service for Ordered Broadcasts (right): Eve steals the Intent and prevents Bob and Carol from receiving it.

Furthermore, an active attacker could launch denial of service or data injection attacks on ordered broadcasts. Ordered broadcasts are serially delivered to Receivers in order of priority, and each Receiver can stop it from propagating further. If a malicious Receiver were to make itself a preferred Receiver by registering itself as a high priority, it would receive the Intent first and could cancel the broadcast. Non-ordered broadcasts are not vulnerable to denial of service attacks because they are delivered simultaneously to all Receivers. Ordered broadcasts can also be subject to malicious data injection. As each Receiver processes the Intent, it can pass on a result to the next Receiver; after all Receivers process the Intent, the result is returned to the sending component. A malicious Receiver can change the result, potentially affecting the sender and all other receiving components.

When a developer broadcasts an Intent, he or she must consider whether the information being sent is sensitive. Explicit broadcast Intents should be used for internal application communication, to prevent eavesdropping or denial of service. There is no need to use implicit broadcasts for internal functionality. At the very least, the developer should consider applying appropriate permissions to Intents containing private data.

3.1.2 Activity Hijacking

In an *Activity hijacking* attack, a malicious Activity is launched in place of the intended Activity. The malicious Activity registers to receive another application's implicit Intents, and it is then started in place of the expected Activity (Figure 2).

In the simplest form of this attack, the malicious Activity could read the data in the Intent and then immediately relay it to a legitimate Activity. In a more sophisticated active attack, the hijacker could spoof the expected Activity's user interface to steal user-supplied data (i.e., phishing). For

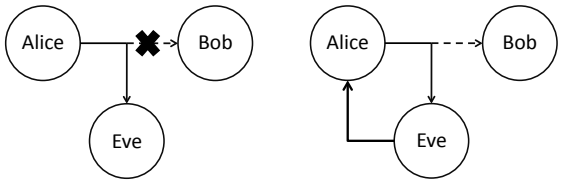

Figure 2: Activity/Service Hijacking (left): Alice accidentally starts Eve's component instead of Bob's. False Response (right): Eve returns a malicious result to Alice. Alice thinks the result comes from Bob.

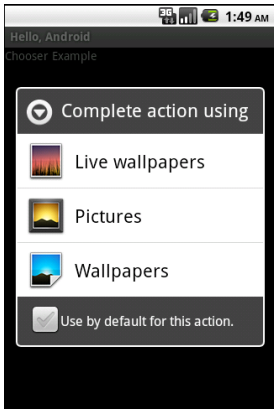

Figure 3: The user is prompted when an implicit Intent resolves to multiple Activities.

example, consider a legitimate application that solicits donations. When a user clicks on a "Donate Here" button, the application uses an implicit Intent to start another Activity that prompts the user for payment information. If a malicious Activity hijacks the Intent, then the attacker could receive information supplied by the user (e.g., passwords and money). Phishing attacks can be mounted convincingly because the Android UI does not identify the currently running application. Similarly, a spoofed Activity can lie to the user about an action's completion (e.g., telling the user that an application was successfully uninstalled when it was not).

Activity hijacking is not always possible. When multiple Activities match the same Intent, the user will be prompted to choose which application the Intent should go to if a default choice has not already been set. (Figure 3 shows the dialog.) If the secure choice is obvious, then the attack will not succeed. However, an attacker can handle this challenge in two ways. First, an application can provide a confusing name for a component to fool the user into selecting the wrong application. Second, the malicious application can provide a useful service so that the user willingly makes it the default application to launch. For example, a user might opt to make a malicious browser the default browser and never get prompted to choose between components again. Although the visibility of the Activity chooser represents a challenge for the attacker, the consequences of a successful attack can be severe.

If an Activity hijacking attack is successful, the victim component may be open to a secondary *false response* attack. Some Activities are expected to return results upon completion. In these cases, an Activity hijacker can return a malicious response value to its invoker. If the victim application trusts the response, then false information is injected into the victim application.

3.1.3 Service Hijacking

Service hijacking occurs when a malicious Service intercepts an Intent meant for a legitimate Service. The result is that the initiating application establishes a connection with a malicious Service instead of the one it wanted. The malicious Service can steal data and lie about completing requested actions. Unlike Activity hijacking, Service hijacking is not apparent to the user because no user interface is involved. When multiple Services can handle an Intent, Android selects one at random; the user is not prompted to select a Service.

As with Activity hijacking, Service hijacking can enable the attacker to spoof responses (a false response attack). Once the malicious Service is bound to the calling application, then the attacker can return arbitrary malicious data or simply return a successful result without taking the requested action. If the calling application provides the Service with callbacks, then the Service might be able to mount additional attacks using the callbacks.

3.1.4 Special Intents

Intents can include URIs that reference data stored in an application's Content Provider. In case the Intent recipient does not have the privilege to access the URI, the Intent sender can set the `FLAG_GRANT_READ_URI_PERMISSION` or `FLAG_GRANT_WRITE_URI_PERMISSION` flags on the Intent. If the Provider has allowed URI permissions to be granted (in the manifest), this will give the Intent recipient the ability to read or write the data at the URI. If a malicious component intercepts the Intent (in the ways previously discussed), it can access the data URI contained in the Intent. If the data is intended to be private, then an Intent carrying data privileges should be explicitly addressed to prevent interception.

Similarly, pending Intents delegate privileges. A *pending Intent* is made by one application and then passed to another application to use. The pending Intent retains all of the permissions and privileges of its creator. The recipient of a pending Intent can send the Intent to a third application, and the pending Intent will still carry the authority of its creator. If a malicious application obtains a pending Intent, then the authority of the Intent's creator can be abused.

3.2 Intent Spoofing

A malicious application can launch an *Intent spoofing* attack by sending an Intent to an exported component that is not expecting Intents from that application (Figure 4). If the victim application takes some action upon receipt of such an Intent, the attack can trigger that action. For example, this attack may be possible when a component is exported even though it is not truly meant to be public. Although developers can limit component exposure by setting permission requirements in the manifest or dynamically checking the caller's identity, they do not always do so.

3.2.1 Malicious Broadcast Injection

If an exported Broadcast Receiver blindly trusts an incoming broadcast Intent, it may take inappropriate action or operate on malicious data from the broadcast Intent. Receivers often pass on commands and/or data to Services and Activities; if this is the case, the malicious Intent can propagate throughout the application.

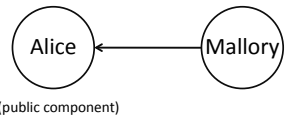

Figure 4: Intent Spoofing. A malicious application Mallory sends an Intent to an exported component Alice. Alice does not expect to receive a message from Mallory.

Broadcast Receivers that register to receive Intents with *system actions* are particularly at risk of malicious broadcast injection. As discussed in Section 2.2, some Intents can only be broadcast by the operating system to inform applications about system events. These Intents contain action strings that only the operating system may add to broadcast Intents. (See Appendix C for examples of system action strings). However, if a Broadcast Receiver registers to receive a system broadcast, the component becomes publicly accessible. In this case, a malicious application can send an Intent explicitly addressed to the target Receiver, without containing the system action string. If the Receiver does not check the Intent's action, then the Receiver will be tricked into performing functionality that only the system should be able to trigger.

3.2.2 Malicious Activity Launch

Exported Activities can be launched by other applications with either explicit or implicit Intents. This attack is analogous to cross-site request forgeries (CSRF) on websites [18, 6]. In most cases, a malicious Activity launch would just be an annoyance to the user because the target Activity's user interface would load. However, three types of Activity launching attacks are possible. First, launching an Activity can cause it to affect application state or modify data in the background. If the Activity uses data from the Intent without verifying the origin of the Intent, the application's data store could be corrupted. Second, a user can be tricked. For example, a user might click on a "Settings" screen in a malicious application, which directs the user to a screen in a victim application. The user might then make changes to the victim application while believing she is still interacting with the malicious application. Third, a victim Activity could leak sensitive information by returning a result to its caller upon completion.

3.2.3 Malicious Service Launch

If a Service is exported and not protected with strong permissions, then any application can start and bind to the Service. Depending on the duties of a particular Service, it may leak information or perform unauthorized tasks. Services sometimes maintain singleton application state, which could be corrupted.

A malicious Service launch is similar to a malicious Activity launch, but Services typically rely on input data more heavily than Activities. Consequently, a malicious launch attack where the Intent contains data is more likely to put a Service at risk. Additionally, there are more opportunities for a bound Service to return private data to its caller because Services often provide extensive interfaces that let their binders make many method calls.

3.3 Secure Communication Guidelines

Developers should be cautious about sending implicit Intents and exporting components. When sending private data, applications should use explicit Intents if possible. Internal communication can and should always use explicit Intents. If it is not possible to use explicit Intents, then the developer should specify strong permissions to protect the Intent. Results returned by other components in response to Intents need to be verified to ensure that they are valid results from an expected source.

To make components more secure, developers should avoid exporting components unless the component is specifically designed to handle requests from other applications. Developers should be aware that declaring an Intent filter will export the component, exposing it to attack. Critical, state-changing actions should not be placed in exported components. If a single component must handle both inter- and intra-application requests, perhaps that component should be divided into separate components, one for each type. If a component must be exported (e.g., to receive system broadcasts), then the component should dynamically check the caller's identity prior to performing any operations. The return values of exported components can also leak private data, so developers should check the caller's identity prior to returning sensitive values. Intent filters are not security measures and can be bypassed with explicit Intents. Requiring Signature or SignatureOrSystem permissions is an effective way of limiting a component's exposure to a set of trusted applications. See Appendix B for examples of how to create secure code.

4. COMDROID

We provide a tool, ComDroid, to detect potential vulnerabilities in Android applications. Applications for the Android platform include Dalvik executable (DEX) files that run on Android's Dalvik Virtual Machine. We first disassemble application DEX files using the publicly available Dedexer tool [24]. ComDroid parses the disassembled output from Dedexer and logs potential component and Intent vulnerabilities. We list the types of warnings ComDroid produces, separated by component and Intent type in Table 2.

4.1 Permission Map

Every permission can be associated with one of the four protection levels described in Section 2.4.2. We consider both system-defined permissions (found in the system's Android Manifest [2]) and application-defined permissions (found in application manifests). We view Normal and Dangerous permissions as easy to obtain (or "weak") and consider components protected with those permissions as public.

4.2 Intent Analysis

ComDroid examines Intent creation and transmission to detect the kinds of vulnerabilities outlined in Section 3.1. To do this, ComDroid statically analyzes disassembled output from Dedexer. Static analysis has commonly been used for bug finding [8, 21, 27]. ComDroid specifically performs flow-sensitive, intraprocedural static analysis, augmented with limited interprocedural analysis that follows method invocations to a depth of one method call. ComDroid parses translated Dalvik files and tracks the state of Intents, IntentFilters, registers, sinks (e.g., `sendBroadcast()`, `startActivity()`, etc.), and components. For each method that uses

Unauthorized Intent Receipt	
Intent type	Potential vulnerability
Sent Broadcasts	Broadcast Theft (without data) Broadcast Theft (with data)
Sent Activity requests	Activity Hijacking (without data) Activity Hijacking (with data)
Sent Service requests	Service Hijacking (without data) Service Hijacking (with data)
Intent Spoofing	
Component type	Potential vulnerability
Exported Broadcast Receivers	Broadcast Injection (without data) Broadcast Injection (with data) System Broadcast without Action Check
Exported Activities	Activity Launch (without data) Activity Launch (with data)
Exported Services	Service Launch (without data) Service Launch (with data)

Table 2: The list of different vulnerabilities associated with each type of Intent and component. "Without data" indicates the Intent involved in the attack does not contain extra data, whereas "with data" indicates the Intent does contain extra data in it and thus may additionally be vulnerable to data leakage or injection.

Intents (whether it is passed an Intent parameter, instantiates an Intent, or otherwise receives an Intent), ComDroid tracks the value of each constant, string, class, Intent, and IntentFilter. When an Intent object is instantiated, passed as a method parameter, or obtained as a return value, ComDroid tracks all changes to it from its source to its sink. An Intent sink is a call that transmits an Intent to another component, such as the calls listed in Table 1.[1]

For each Intent object, we track (1) whether it has been made explicit, (2) whether it has an action, (3) whether it has any flags set, and (4) whether it has any extra data. For each sink, we check whether it is possible for any implicit Intent object to flow to that sink. Some Intent-sending mechanisms allow the sender to specify a permission that restricts who the Intent can be delivered to; our analysis records this information.

ComDroid issues a warning when it detects an implicit Intent being sent with weak or no permission requirements. Intents sent through the sink may be vulnerable to action-based attacks (e.g., broadcast denial of service or Activity/Service launching). If any of these Intents contain extra data, then they may also be vulnerable to eavesdropping. We issue warnings with "without data" and "with data" tags to distinguish action-based attacks from eavesdropping. Intents containing data in excess of an action, categories, component name, or package name are considered as having extra data and therefore open to both the action- and data-based attacks.

4.3 Component Analysis

ComDroid's component analysis decides whether components might be susceptible to an Intent spoofing attack. ComDroid examines the application's manifest file and translates Dalvik instructions to get information about each component. For each component, ComDroid determines whether the component is public based on the presence of Intent filters or the `EXPORTED` flag.

Activities and Services are always declared in the manifest. Some Receivers are also declared in the manifest. An Activity can be multiply declared in the manifest using an Activity alias, which presents an existing Activity as a separate component with its own permission, Intent filters, etc. We treat Activities and their aliases as separate components for the purpose of our analysis because an alias's fields can increase the exposure surface of the component. Also, typically one Activity is marked as a main, launching Activity that the system opens when an application is started. This Activity is public but is generally less likely to be attackable, therefore we do not issue an exposure warning for this case.

Receivers can also be dynamically created and registered by calling `registerReceiver(BroadcastReceiver receiver, IntentFilter filter)`. The Intent filter is specified at registration time and can be changed each time `registerReceiver` is called, so we consider each registration of a Receiver as a unique component.

If a public component is protected with no permission or a weak permission, ComDroid generates a warning about a potential Intent spoofing attack (malicious Broadcast injection, malicious Activity launch, or malicious Service launch, depending on the component type). Again, we further separate the warnings into "without data" and "with data" warnings. Attacks without additional data only invoke the victim component; attacks with data additionally supply the victim component with malicious data. Both are attack surfaces, but attacks with data can potentially give an attacker more control and more opportunities to influence application state or pollute application databases. If the component receives an Intent and only reads the action, categories, component name, or package name, then it is considered to not use extra data. Otherwise, it is considered to use extra data.

ComDroid separately issues warnings for Receivers that are registered to receive system broadcast actions (actions only sent by the system). For these warnings, the solution is to add a call to `android.content.Intent.getAction()` to

[1]We do not consider `stopService()` as a vulnerable sink. If a Service is maliciously stopped, it can be restarted by a legitimate component when the Service is needed.

verify that the protected action is in the Intent (authenticating the sender of the Intent). This is in contrast to other Intent spoofing attacks where the more common solution is to make the component private.

ComDroid also notes when it appears that unique Intent actions are being used to communicate in place of explicit Intents. If ComDroid finds a public component that registers to receive Intents with a non-Android action string and also finds components that transmit implicit Intents with the same action string, ComDroid issues a warning. We call this "action misuse" to alert the developer that he or she may be using actions insecurely.

4.4 Limitations and Discussion

We currently track Intent control flow across functions, but we do not distinguish between paths through `if` and `switch` statements. Instead, we follow all branches. This can lead to false negatives; e.g., an application might make an Intent explicit in one branch and implicit in another, and our tool would always identify it as explicit. In retrospect, it would have been better to track both. Additionally, our tool does not yet detect privilege delegation through pending Intents and Intents that carry URI read/write permissions; we leave this for future work. Despite these limitations, our experimental results (Section 5) indicate that our analysis identifies many actual application vulnerabilities.

It is important to note that ComDroid issues warnings but does not verify the existence of attacks. Some Intents and components are intentionally made public without restriction, for the purpose of inter-application collaboration. It is not possible to automatically infer the developer's intention when making a component public. We defer to the developer to examine his or her own program and verify the veracity of the warnings. ComDroid supplies the location of the potential vulnerability (filename, method, and line number), the type (malicious Activity launch, broadcast theft, etc.), and whether data leakage/injection could be involved. It could further be extended to explicitly recommend a fix for developers (e.g., make the Intent explicit).

Although ComDroid is intended primarily as a tool for developers, it takes DEX files as input instead of source code. We made this choice due to the difficulty of obtaining source code for most applications. Using DEX files, we can examine the programming practices of the most popular applications in the Android Market. The use of DEX files also allows third parties (such as anti-virus vendors) to conduct security audits. That said, ComDroid requires the user to manually investigate the warnings, which may be difficult for third parties to do quickly (especially on a large scale). Ideally, ComDroid would be used by the developers themselves or security teams contracted by the developers, since they are familiar with the code or have access to the source code.

We considered a dynamic analysis approach to ComDroid as an alternative to our static approach. A dynamic analysis tool would have the benefit of confirming a vulnerability by exploiting it at run-time (although it still may not be able to make the human distinction of whether the bug is severe or not), but it may be challenging to explore the application state space to obtain full coverage. Static analysis has the benefit of discovering vulnerabilities that may not have been exposed at runtime. It is worth investigating a combined static and dynamic tool in future research to leverage the benefits of both approaches.

Type of Exposure	Percentage
Broadcast Theft	44 %
Activity Hijacking	97 %
Service Hijacking	19 %
Broadcast Injection	56 %
System Broadcast w/o Action Check	13 %
Activity Launch	57 %
Service Launch	14 %

Table 3: The percentage of applications that had at least one warning per exposure type

5. EVALUATION

We ran ComDroid on the top 50 popular paid applications and on 50 of the top 100 popular free applications on the Android Market [1].[2] We report ComDroid's warning rates and discuss common application weaknesses. We emphasize that ComDroid issues warnings about potential security issues; manual review is needed to determine the functionality of the Intent or component and decide whether the exposure can lead to a severe vulnerability. We manually examined 20 applications to check ComDroid's warnings, evaluate our tool, and detect vulnerabilities.

5.1 Automated Analysis

ComDroid detected a total of 1414 exposed surfaces across 100 applications. There were 401 warnings for exposed components and 1013 warnings for exposed Intents. In Figure 5 we show what fraction of sent Intents are implicit; on average, about 40% are implicit. In Figure 6, we show the frequency of exposed components out of the total number of components for each application, separated by component type. 50% of Broadcast Receivers are exposed, and most applications expose less than 40% of Activities to external applications. Our tool does not generate a warning for an application's primary launcher Activity; consequently, many applications that show zero exposed Activities warnings may have one public launcher Activity. There is no clear distribution for the exposure of Services because few applications have multiple Service components.

We also show the breakdown of warnings by type in Figure 7. Intuitively, there is more Intent communication between Activities so there are more exposure warnings for Activity-related Intents than Broadcast- and Service-related Intents combined.

Table 3 shows the percentage of applications that have at least one of a given type of surface exposure. Of sending-related vulnerabilities, 44% of applications have Broadcast-related warnings. Of these applications, none of them restrict the broadcast in any way or make the Intent explicit.

Although 97% of applications have Activity hijacking warnings, on average only 27.7% of Intents that involve starting an Activity are open to an Activity hijack. This is promising as it shows that developers are making a majority of their Activity communications explicit. 19% of applications contain Service hijacking warnings.

[2]Specifically, we considered the applications ranked 51-100 for the free applications. Dedexer was not able to disassemble a few applications. In those cases, we took the next application in the list.

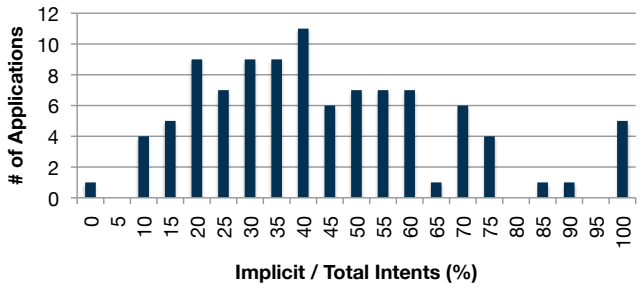

Figure 5: Histogram showing the percentage of implicit Intents out of total Intents for each application.

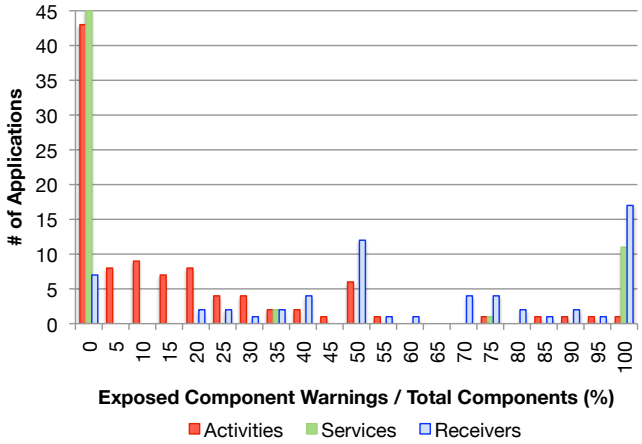

Figure 6: Histogram showing the percentage of components with warnings out of total components for each application.

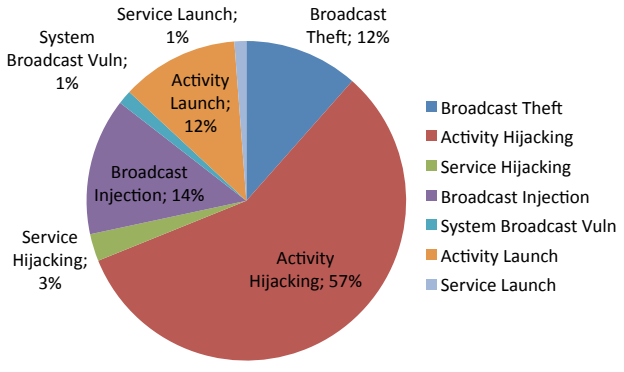

Figure 7: Breakdown of warnings by type.

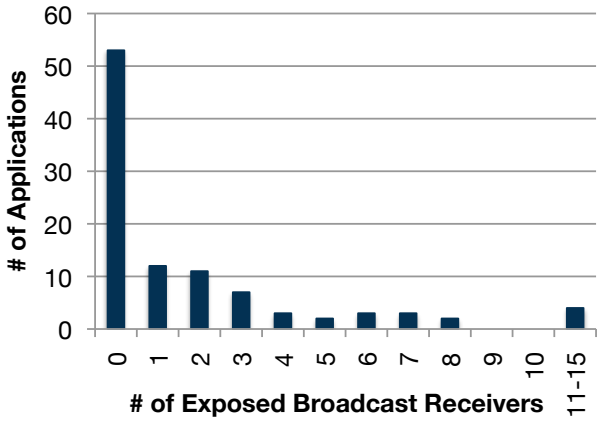

Figure 8: Histogram showing the number of Broadcast Receivers with warnings per application.

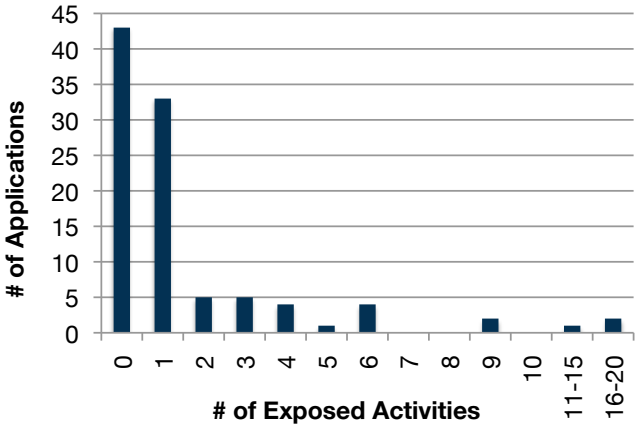

Figure 9: Histogram showing the number of Activities with warnings per application.

Over 56% of applications have a Broadcast Receiver that may be vulnerable to a Broadcast injection attack. Broken down by number of exposed Receivers per application (Figure 8), we see that most applications expose one or two Receivers, if any.

13% of applications have a public Receiver that accepts a system Broadcast action but does not check that the Intent actually contains that action (a definite bug that may also lead to a serious vulnerability).

57% of applications have at least one Activity that may be vulnerable to a malicious Activity launch. The other 43% only expose the main launching Activity. We display the break down of these malicious Activity launch warnings by number of malicious launch warnings per application (Figure 9). On average, applications have one exposed Activity in addition to the launch Activity. This is good news, as it seems that most applications are limiting their Activities from exposure. We can also see a handful of applications expose 11 to 20 Activities and can benefit from further investigation. Finally, 14% of applications have at least one Service that may be vulnerable to malicious Service launches.

Our results indicate that Broadcast- and Activity- related Intents (both sending to and receiving from) play a large role in application exposure.

5.2 Manual Analysis

We randomly selected 20 applicationis from the 100 mentioned earlier, and then we manually inspected ComDroid's warning for these applications to evaluate how many warnings correspond to vulnerabilities. In this section, we present the findings of our manual analysis and discuss three example applications with vulnerabilities. See Appendix A for the list of applications reviewed.

ComDroid generated 181 warnings for the 20 applications. We manually reviewed all of them and classified each warning as a vulnerability, not a vulnerability, or undetermined. We define a vulnerability as a component or Intent that exposes data or functionality that can be detrimental to the user. For example, an unprotected broadcast is only a vulnerability if it includes sensitive user data or if its theft results in a DoS to a legitimate service. Similarly, Activity launching is only a vulnerability if the victim Activity toggles state or operates on the Intent data in a way that negatively affects the application. Negative consequences may be context-dependent; consider an application that sends a user to another application to view a website. Normally, it will not matter if the website Activity is hijacked, but the hijacking could lead to phishing if the user expects to enter payment information at the website. We further divide vulnerabilities into two types: (1) dangerous vulnerabilities that do not rely on user interaction and (2) spoofing vulnerabilities that might occur if the user is tricked. We also separately note commonly found unintentional bugs. We classify warnings as bugs when the developer appears to be misusing or misunderstanding the Android communication design. This category includes action misuse and system broadcasts without action verification.

In order to detect vulnerabilities, we reviewed the disassembled code of the application components with warnings. We also installed the applications and interacted with them to dynamically observe their Intents. When necessary, we built "attack" code to confirm or disprove vulnerabilities. This review process does not reflect a developer's experience with ComDroid because developers would have access to the source code and knowledge of the application's intended functionality, which we did not have.

Of the 181 warnings, we discovered 20 definite vulnerabilities, 14 spoofing vulnerabilities, and 16 common, unintentional bugs (that are not also vulnerabilities). Of the 20 applications examined, 9 applications contain at least 1 definite vulnerability and 12 applications have either definite or spoofing vulnerabilities. This demonstrates the prevalence of insecure Intent communication. Table 4 shows the number of vulnerabilities and warnings for each category.

ComDroid has an overall vulnerability and bug detection rate of 27.6%. Broken down by Unauthorized Intent Receipt vulnerabilities/bugs and Intent Spoofing vulnerabilities/bugs, it has a rate of 22.6% and 38.6%, respectively. As shown in the table, Activity hijacking has the highest number of false positives, with a lower detection rate of 15.2%. Examining only the broadcast-related vulnerabilities (theft, injection, and system broadcasts without action check), ComDroid has a detection rate of 61.2%.

In 25 cases, we were unable to determine whether warnings were vulnerabilities. We cannot always determine whether a surface is intentionally exposed without knowing the developer's intentions. We were uncertain of 25 of the 181 warnings. The remaining 106 warnings were false positives, i.e., not dangerous or spoofing vulnerabilities or common bugs. Of these, 6 of the warnings should not have been generated and can be attributed to shortcomings in our implementation of ComDroid. (Two Broadcast Receivers were declared without receiving methods, meaning they could not actually receive Intents. In four cases, Intents were misidentified as implicit when they were actually explicit.) The remaining 100 false positives are still exploitable attacks. However, the impact of these attacks is minor: They would be merely a nuisance to the user. For example, an Activity that turns on a "flashlight" when launched or takes some other trivial action would fall into this category. Because they represent only a nuisance, we conservatively decided not to classify them as vulnerabilities. We now discuss a few applications and the vulnerabilities we discovered in them.

ICE: In Case of Emergency. "ICE: In Case of Emergency" is an application that can be launched from a locked screen by a paramedic in case of an emergency [5]. It stores medical information such as medications, allergies, medical conditions, insurance information, and emergency contact information. ICE contains multiple exploitable Broadcast Receivers. One Broadcast Receiver can be used to exit any running application and lock the screen. Several of ICE's Broadcast Receivers will temporarily remove the ICE widget from the locked screen, rendering ICE unusable.

ICE's vulnerable Broadcast Receivers are accidentally public due to developer confusion over Android's complexities. Two of ICE's Receivers are registered to receive protected system broadcasts, which causes Android to make them public. ICE does not check that the received Intent has the appropriate system action, so an explicit Intent to the Receiver will trigger the same behavior as if the OS had sent the Intent. For example, ICE disappears when the operating system broadcasts an Intent with the `BOOT_COMPLETED` action; a malicious application could send an explicit Intent with this action to ICE and fool it into exiting. Additionally, some of ICE's Receivers use broadcasts to pass internal notifications. The internal broadcasts have application-specific actions, e.g., `com.appventive.ice.unlock_finished`. This is a misuse of action strings. These components should be made private and invoked explicitly.

IMDb Mobile. "IMDb Mobile" is a movie resource application [17]. It presents facts about movies, and users can look up local showtimes. IMDb's showtime Activity has buttons for the user to select a location and refresh the showtime search results. When the user clicks on one of the buttons, the Activity relays the request to a background Service. The Service responds with a public broadcast, and a Broadcast Receiver listens for the broadcast and then updates the state of the Activity's user interface. For example, the Service can send a broadcast with a `com.imdb.mobile.showtimesNoLocationError` action, which will cause the Activity to display an error stating that no showtimes are available for the desired location.

IMDb Mobile's broadcast Intents are intended for internal use. No other application needs to know about them, and other applications should not be able to control the user interface. However, with the current implementation, the showtime Activity can be manipulated by a malicious application. The developer should have used explicit Intents to communicate internally, and the Receiver should not be exported.

Type of Exposure	Definite Vulnerabilities	Spoofing Vulnerabilities	Unintentional Bugs (no vuln.)	Total Warnings	Vuln. and Bug Percentage
Broadcast Theft (without data)	1	0	6	10	70.0%
Broadcast Theft (with data)	2	0	2	4	100.0%
Activity Hijacking (without data)	2	11	0	91	14.3%
Activity Hijacking (with data)	0	3	0	14	21.2%
Service Hijacking (without data)	0	0	1	5	20.0%
Service Hijacking (with data)	0	0	0	0	--
Broadcast Injection (without data)	10	0	2	24	50.0%
Broadcast Injection (with data)	3	0	0	7	42.9%
System Broadcast w/o Action Check	1	0	3	4	100.0%
Activity Launch (without data)	0	0	0	9	0.0%
Activity Launch (with data)	0	0	2	10	20.0%
Service Launch (without data)	0	0	0	2	0.0%
Service Launch (with data)	1	0	0	1	100.0%

Table 4: The number of vulnerabilities and bugs we found from the warnings in ComDroid.

Nationwide Bus. Nationwide Bus is an Android application that gives bus location and arrival information for Korean cities [20]. It uses public broadcasts for internal communication. The broadcasts are used to update map and bus state. One component fetches bus information from the server and then broadcasts the data, which is intended for an internal Receiver. This is a privacy violation if the user does not want other applications to know his or her location.

Two exported components expect bus data as input. A Receiver listens for the aforementioned broadcasts, and a Service in charge of bus arrivals is started with bus data. A malicious application could send these components Intents with fake bus information, which will then be displayed in the map as fake bus stations and arrival times. The developer should have used explicit Intents for internal communication, and the Receiver and Service should not be exported.

5.2.1 Financial Applications

We also used ComDroid to guide a review of 10 popular financial and commercial applications. None of these applications were part of our larger set of 100 applications. The financial and commercial applications are generally well-secured; we did not find any vulnerabilities that could put the user's financial information at risk. It is clear that the applications were built to be secure, with few exposed surfaces. For example, applications make use of `exported=false`, use explicit Intents, and do not declare Intent filters for most of their components. It appears that the financial and commercial applications were written or reviewed by security-conscious developers, who might benefit from a tool like ComDroid during the development process.

Despite their focus on security, we found vulnerabilities in 4 of the 10 applications. One application sends an Intent to ask the browser to open up a bank website to a login page. The Intent is implicit, which puts the user at risk of an Activity hijacking phishing attack. Three applications misuse actions: they use a class name as an action string to bind to a Service. An attacker could hijack the Service and mount a denial of service attack on the parts of the application that rely on the Service. One of the vulnerable applications also contains a number of bugs that are not exploitable; it registers for a system broadcast and does not check the sender, and it uses broadcasts for internal communication. We believe that these errors, made by security-conscious developers, are indicative of the fact that Android's Intent system is confusing and tricky to use securely.

5.3 Discussion

Our analysis shows that Android applications are often vulnerable to attack by other applications. ComDroid's warnings can indicate a misunderstanding of the Intent passing system, as we illustrated with the ICE, IMDb, and Nationwide Bus applications and can alert the developer (or a reviewer) to faulty software engineering practices that leak data and expose application internals.

Our analysis reveals that developers commonly use the action field of an Intent like an address instead of explicitly addressing the Intent. (For example, a developer might use actions prefixed with the application package name.) They add Intent filters to the components that listen for Intents with their action name, which has the undesirable side effect of making the component public. It is reasonable to assume that they are either forgetting or are not aware that they should be making their Receiver private. ComDroid can help developers be aware of surfaces that are accidentally exposed in this manner.

5.4 Recommendations

Along with more vigilant developer coding, we also recommend changes that can be made to the Android platform to prevent unintentional exposure. One of the fundamental problems is that Intents are used for both intra- and inter-application communication and using them within an application can expose the application to external attack if the developer is not careful. Ideally, intra- and inter- application communication should be carried out through different means. Similarly, component accessibility should be divided into three categories: internal, exported to the system only, and exported to other applications.

We acknowledge that this would only prevent bugs in future applications. To fix current bugs in legacy applications, we suggest another approach. To address unintentional Intent-sending vulnerabilities, we suggest that the system try to deliver any implicit Intents first to internal components. If the Intent can be delivered to an internal component (of the same application as the sender), it should not be delivered to any other applications. This would handle the case where developers use implicit Intents to communicate with other internal components. Of the 100 applications analyzed in Section 5, this change would eliminate 106 warnings. Of the 20 applications we manually reviewed, the change would eliminate 9 bugs and 2 vulnerabilities.

To address Intent-receiving vulnerabilities, we suggest that the system should not implicitly make a component public merely because it has declared an Intent filter. Instead, we propose that Android should make a component public only if it: (1) sets the `exported` flag, (2) has an Intent filter with a data field specified, (3) has an Intent filter that registers to receive protected system actions, (4) has a main, launcher specification, or (5) has an Intent filter that registers to receive Intents with one of the standard Android actions. This definition represents a compromise between security and backward compatibility. Of the 100 applications analyzed, this change would eliminate 50 warnings. Of the 20 applications we manually reviewed, it would eliminate 5 vulnerabilities and 1 bug. These are conservative changes; we verified that neither of these changes break any of the 20 applications we manually tested.

6. OTHER MOBILE PLATFORMS

Windows Phone 7 (WP7) and iOS also provide third-party application platforms. However, their inter-application communication systems are less complex than Android's. WP7 applications can only send messages to a small number of trusted system applications (e.g., the browser); third-party applications cannot receive messages. Consequently, we are not aware of any security risks associated with WP7 application communication.

iOS applications can choose to accept inter-application messages by registering custom URI schemes with the OS. When an application receives a message, the application is opened and moved to the foreground. (For example, sending a message to `skype://15554446666` opens the Skype application.) This is the equivalent of sending an Intent to start an Activity, and a malicious iOS application could mount an Activity hijacking attack by registering for another application's scheme. However, the remaining Android communication attacks are not applicable to iOS. iOS developers are unlikely to accidentally expose functionality because schemes are only used for public interfaces; different types of messages are used for internal communication. Our recommendations for Android (Section 5.4) aim to create the same distinction between internal and external messages in Android.

7. RELATED WORK

Attack Surfaces. The concept of examining systems to identify and quantify their attack surfaces is not new. Metrics have been proposed for evaluating the exposed attack surface of a generic system [16], and attack surface reduction has widely been recognized as an approach to improving system security [22].

Non-mobile Systems. The Android inter-application communication system is analogous to a (local) network system. As such, it must deal with standard threats that apply to all messaging systems, for example, eavesdropping, spoofing, denial of service, etc. [9]. As we have shown in this paper, these threats are present in Android's Intent system.

Problems in Android's communication model are similar to problems with decentralized information flow control (DIFC) in other systems. DIFC lets applications explicitly express their information flow policies (i.e., which applications communicate and how they communicate) to the OS or a language runtime, which then enforces the policies [19, 23, 28, 10]. The problems that arise in Android relate to developers' difficulty with setting appropriate communication policies; the same problems exist in DIFC models, which also require the developer to write policies.

Android. We are not the first to realize that Android developers make mistakes that can compromise security. Burns [7] discusses common developer errors, such as using Intent filters instead of permissions. He recommends using permissions to protect components and validate caller identity. Our work builds on these concepts and provides a tool for detecting these errors. Similarly, Enck et al. [14] examine Android security policies and discuss some developer pitfalls. They present a decompiler to recover application source code from DEX files and apply COTS Java static analysis tools to the source code to examine various properties in applications. They investigate how broadcast Intents can leak information and how information can be injected into Receivers [12]. Their investigation of broadcast Intents and Receivers is limited to data-based attacks, and they do not discuss attacks involving Activities or Services. We present non-data attacks on Receivers (e.g., denial of service attacks and state change) and extensively consider Activities and Services.

SCanDroid [15] is a static analysis tool that takes a data-centric approach to reasoning about the consistency of security specifications. It analyzes data policies in application manifests and data flows across Content Providers. Based on its analysis, it makes a recommendation on whether an application can be installed with the permissions it has without violating the permissions of other applications. Used together, ComDroid and SCanDroid could combine surface exposure with database permission violations. However, SCanDroid currently requires users to have access to application source code, which may not be feasible.

TaintDroid [11] provides system-wide dynamic taint tracking for Android. It discovered 68 potential information misuse examples in 20 applications. Unlike ComDroid, TaintDroid focuses solely on data flow and does not consider action-based vulnerabilities. We found many control-flow vulnerabilities using ComDroid. Also, TaintDroid is meant to be a post-production tool for real-time analysis, while ComDroid can be used as either a pre- or post-production tool. TaintDroid and ComDroid are complementary tools.

Kirin [13] approaches third-party application security from the opposite perspective of our tool. While we look for vulnerabilities in benign applications, Kirin looks for malicious applications. They limit their analysis to security configurations in the manifest and take a blacklist approach to detecting undesirable permission combinations.

8. CONCLUSION

While the Android message passing system promotes the creation of rich, collaborative applications, it also introduces the potential for attack if developers do not take precautions. We examine inter-application communication in Android and present several classes of potential attacks on applications. Outgoing communication can put an application at risk of Broadcast theft (including eavesdropping and denial of service), data theft, result modification, and Activity and Service hijacking. Incoming communication can put an application at risk of malicious Activity and Service launches and Broadcast injection.

We provide a tool, ComDroid, that developers can use to find these kinds of vulnerabilities. Our tool relies on DEX code, so third parties or reviewers for the Android Market can use it to evaluate applications whose source code is unavailable. We analyzed 100 applications and verified our findings manually with 20 of those applications. Of the 20 applications, we identified 12 applications with at least one vulnerability. This shows that applications can be vulnerable to attack and that developers should take precautions to protect themselves from these attacks.

Acknowledgments

This work is partially supported by National Science Foundation grant CCF-0424422 and a gift from Google. This material is also based upon work supported under a National Science Foundation Graduate Research Fellowship. Any opinions, findings, conclusions, or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

9. REFERENCES

[1] Android Market. `http://www.android.com/market/`.

[2] Android permissions. `http://android.git.kernel.org/?p=platform/frameworks/base.git;a=blob;f=core/res/AndroidManifest.xml`.

[3] iPhone App Store. `http://www.apple.com/iphone/apps-for-iphone/`.

[4] MobiStealth. `http://www.mobistealth.com/`.

[5] Appventive. ICE: In case of emergency. `http://www.appventive.com/ice`.

[6] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. of the 15th ACM Conference on Computer and Communications Security (CCS 2008)*, 2008.

[7] J. Burns. Mobile application security on Android. *Blackhat*, 2009.

[8] B. Chess and G. McGraw. Static analysis for security. *Security & Privacy, IEEE*, 2(6):76–79, 2004.

[9] W. Cheswick, S. Bellovin, and A. Rubin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[10] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proc. of the 20th ACM Symposium on Operating Systems Principles*, pages 17–30. ACM, 2005.

[11] W. Enck, P. Gilbert, B.-g. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, October 2010.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. of the 20th USENIX Security Symposium*, August 2011.

[13] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proc. of the 16th ACM Conference on Computer and Communications Security (CCS)*, November 2009.

[14] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[15] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated security certification of Android applications. Technical report, University of Maryland, 2009.

[16] M. Howard, J. Pincus, and J. Wing. Measuring relative attack surfaces. *Computer Security in the 21st Century*, pages 109–137, 2005.

[17] IMDb. IMDb Movies & TV. `http://www.androlib.com/android.application.com-imdb-mobile-jzEzw.aspx`.

[18] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Securecomm and Workshops, 2006*, pages 1–10. IEEE, 2006.

[19] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles*, pages 321–334. ACM, 2007.

[20] H. Lee. Nationwide bus. `http://www.androlib.com/android.application.net-hyeongkyu-android-incheonbus-Eqwq.aspx`.

[21] V. B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proc. of the 14th Conference on USENIX Security Symposium*, pages 18–18. USENIX Association, 2005.

[22] P. Manadhata, J. Wing, M. Flynn, and M. McQueen. Measuring the attack surfaces of two FTP daemons. In *Proc. of the 2nd ACM Workshop on Quality of Protection*, pages 3–10. ACM, 2006.

[23] A. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):410–442, 2000.

[24] G. Paller. Dedexer. `http://dedexer.sourceforge.net/`.

[25] M. A. Troy Vennon. Android malware: Spyware in the Android Market. Technical report, SMobile Systems, March 2010.

[26] T. Vennon. Android malware: A study of known and potential malware threats. Technical report, SMobile Systems, February 2010.

[27] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, 2000.

[28] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proc. of the 7th Symposium on Operating Systems Design and Implementation*, pages 263–278. USENIX Association, 2006.

APPENDIX

A. MANUAL REVIEW

We manually reviewed all ComDroid warnings for the following applications: Adobe Photoshop Express, App Protector Pro, Bubble, Halloween Live Wallpaper, ICE - In Case of Emergency, IMDb Movies & TV, Instant Heart Rate, Kindle for Android, Korean Nationwide Bus, Pageonce Pro - Money and Bills, PicSay Pro - Photo Editor, Retro Camera, Smart Keyboard PRO, Starlight Live Wallpaper, Steamy Window, SwiftKey Keyboard, Tango Video Calls, TweetCaster Pro for Twitter, Uninstaller, and WolframAlpha.

B. EXAMPLE CODE

Declaring Components. When declaring a component in the manifest, a developer can use the "exported" attribute to make the component explicitly internal (i.e., private):

```
<activity android:name=".TestActivity"
  android:exported="false">
</activity>
```

If a developer needs to make a component selectively accessible to external applications, he or she can use permissions to restrict access to applications with the given permissions:

```
<activity android:name=".TestActivity2"
  android:exported="true">
  android:permission="my.permission">
  <intent-filter>
    <action android:name="my.action.TEST"/>
  </intent-filter>
</activity>
```

If a component is protected with a new permission, the new permission can be declared as such:

```
<permission
  android:description="My test permission"
  android:name="my.permission"
  android:protectionLevel="signature"/>
```

Alternately, the component implementation can dynamically call `checkCallingPermission(String permission)` to verify that the caller has the specified permission.

Declaring Intents. When sending an Intent, the developer can make the recipient explicit by setting a destination class:

```
Intent i = new Intent();
i.setClassName("some.package.name",
  "some.package.name.TestActivity");
```

Or, equivalently, the class name can be set with one of the following three methods:

```
setClass(Context ctxt, Class<?> cls)
setClassName(Context ctxt, String className)
setComponent(ComponentName component)
```

Or it can be limited to be sent to components of a specific package:

```
setPackage(String packageName)
```

If a Receiver is intended to only accept system broadcast actions, then the developer should check the received action:

```
public void onReceive(Context ctxt, Intent i){
  if (i.getAction().equals("expected.action"))
    return;
}
```

C. SYSTEM BROADCAST ACTIONS

Several examples of system broadcast actions:

```
android.intent.action.ACTION_POWER_CONNECTED
android.intent.action.ACTION_POWER_DISCONNECTED
android.intent.action.ACTION_SHUTDOWN
android.intent.action.BATTERY_CHANGED
android.intent.action.BATTERY_LOW
android.intent.action.BATTERY_OKAY
android.intent.action.BOOT_COMPLETED
android.intent.action.CONFIGURATION_CHANGED
android.intent.action.DEVICE_STORAGE_LOW
android.intent.action.DEVICE_STORAGE_OK
```