Automatic detection of inter-application permission leaks in Android applications

D. Sbîrlea
M. G. Burke
S. Guarnieri
M. Pistoia
V. Sarkar

The Android® operating system builds upon already well-established permission systems but complements them by allowing application components to be reused within and across applications through a single communication mechanism, called the Intent mechanism. In this paper, we describe techniques that we developed for statically detecting Android application vulnerability to attacks that obtain unauthorized access to permission-protected information. We address three kinds of such attacks, known as confused deputy, permission collusion, and Intent spoofing. We show that application vulnerability to these attacks can be detected using taint analysis. Based on this technique, we developed PermissionFlow, a tool for discovering vulnerabilities in the byte code and configuration of Android applications. To enable PermissionFlow analysis, we developed a static technique for automatic identification of permission-protected information sources in permission-based systems. This technique identifies application programming interfaces (APIs) whose execution leads to permission checking and considers these APIs to be sources of taint. Based on this approach, we developed Permission Mapper, a component of PermissionFlow that improves on previous work by performing fully automatic identification of such APIs for Android Java® code. Our automated analysis of popular applications found that 56% of the most popular 313 Android applications actively use intercomponent information flows. Among the tested applications, PermissionFlow found four exploitable vulnerabilities. By helping ensure the absence of inter-application permission leaks, we believe that the proposed analysis will be highly beneficial to the Android ecosystem and other mobile platforms that may use similar analyses in the future.

Introduction

USERS of modern smartphones can install third-party applications from markets that host hundreds of thousands of applications [1, 2] and even more from outside of official markets. To protect sensitive user information from these potentially malicious applications, most operating systems use permission-based access-control models (Android** [3], Windows Phone** 7 [4], Meego** [5], and Symbian** [6]).

Permissions are a well-known and powerful security mechanism, but as with any new operating system, there is the possibility that Android-specific features may reduce the guarantees of the classic permissions model. One such feature is the new communication mechanism (called *Intent*), which can be used to exchange information between components (called *Activity* classes) of the same application or of different applications.

One type of attack that exploits Intents for malicious purposes is *permission collusion*. In this attack, an application that individually only has access to harmless permissions augments its capabilities by invoking a collaborating application through sending and receiving Intents. To stage this attack, malevolent developers could trick users into installing such cooperating malicious applications that covertly compromise privacy.

Digital Object Identifier: 10.1147/JRD.2013.2284403

©**Copyright** 2013 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the Journal reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied by any means or distributed royalty free without further permission by computer-based and other information-service systems. Permission to republish any other portion of this paper must be obtained from the Editor.

0018-8646/13 © 2013 IBM

Listing 1 An Activity declaration in AndroidManifest.xml with declarations of used permissions and an intent-filter.

```
1 <manifest package="com.android.app.myapp" sharedUid="uidIdentifier">
2     <uses -permission name=" android.permission.VIBRATE" />
3     <activity name=" MyActivity ">
4         <intent - filter>
5             <action name="com.zxing.SCAN" />
6             <category name="category.DEFAULT" />
7         </intent - filter>
8      </activity>
9 </manifest>
```

A second type of attack using Intents is the *confused deputy attack*. Confused deputy attacks rely on misconfigured applications; components that interact with other applications are invoked by unauthorized callers and allow them to perform protected actions or access permission-protected information.

A third type of attack, *Intent spoofing* [7], is an Android-specific form of the confused deputy attack: it affects applications not meant to communicate with other applications. Even if a developer's intention was to disallow external invocation of internal Activity classes, other applications may be able to invoke them if the application does not have the necessary configuration. This is possible because Intents can be used for inter-application invocations as well as intra-application invocations.

In this paper, we focus on the use of the above types of attacks to obtain unauthorized access to permission-protected information via exploiting the Intent mechanism. We call these attacks permission-leak attacks.

Tested on 313 popular Android Market applications, our tool, PermissionFlow, identified that 56% of them use intercomponent information flows that may require permissions. Four exploitable vulnerabilities were found. Our contributions are as follows:

- We describe a static analysis-based technique that detects permission-leaking Intent vulnerabilities in Android applications. Based on this technique, we developed PermissionFlow, a tool for discovering vulnerabilities in the byte code and configuration of Android applications.
- We propose a static analysis-based technique for automatic identification of permission-protected information sources in permission-based systems. Our approach consists of identifying APIs whose execution leads to permission-checking and considering these APIs to be sources of taint. Based on this approach, we developed Permission Mapper, a component of PermissionFlow that improves on previous work by performing fully automatic identification of such APIs for Android Java code.
- We evaluate PermissionFlow on leading Android applications and show that a majority (177 out of 313) of applications tested use Intents to invoke Activity classes that return information. These applications could benefit from PermissionFlow to ensure that the use of this feature is secure. PermissionFlow found three permission-protected leaks in widely used applications and an additional vulnerability that allows leaking of information that should be protected by custom permissions.

Background

The vulnerabilities we identify involve knowledge about the Android development model, the Android interprocess communication mechanism, and its permissions system. These components are the focus of the following subsections.

Android development

Android applications are typically written in Java using both standard Java libraries and Android-specific libraries. On Android devices, the Java code does not run on a standard Java Virtual Machine (JVM) but is compiled to a different register-based set of byte code instructions and is executed on a custom virtual machine [Dalvik virtual machine (DVM)]. Android application packages (or "APKs," after their file extension) are actually ZIP archives containing the Dalvik byte code compiled classes, their associated resources such as images, and the application manifest file.

The application manifest is an XML configuration file (AndroidManifest.xml) used to declare the various components of an application, their encapsulation (public or private), and the permissions required by each of them.

Android APIs offer programmatic access to mobile device-specific features such as GPS, vibrator, address book, data connection, calling, SMS, and camera. These APIs are usually protected by permissions. Consider the Vibrator class as an example; to use the `android.os.Vibrator.vibrate(long milliseconds)` function, which starts the phone vibrator for a number of milliseconds, the permission `android.permission.VIBRATE` must be declared in the application manifest, as seen on line 2 of **Listing 1**.

Listing 2 Code snippet showing how a caller accesses information returned by a child Activity.

```
1 void onActivityResult ( int requestCode , int resultCode , Intent data) {
2    if ( requestCode == CREATE REQUEST CODE) {
3      if ( resultCode == RESULT OK) {
4        String info = intent.getStringExtra( ”key ”);
5       }
6     }
7 }
```

Application signing is a prerequisite for inclusion in the official Android Market. Most developers use self-signed certificates that they can generate themselves, which do not imply any validation of the identity of the developer. Instead, they enable seamless updates to applications and enable data reuse among sibling applications created by the same developer. Sibling applications are defined by adding a sharedUid attribute in the application manifest of both, as seen in line 1 of Listing 1.

Activity classes

The Android libraries include a set of graphical user interface (GUI) components built specifically for the interfaces of mobile devices, which have small screens and low power consumption. One type of such component is called Activity classes, which are windows on which all visual elements reside. An Activity can be a list of contacts from which the user can select one, or the camera preview screen from which he can take a picture, or the browser window, etc.

Intents

To spawn a new Activity, programmers use Intents and specify the name of the target class, as shown in the following snippet:

```
Intent i = newIntent ();
i.setClassName(this, “package.CalleeActivity”);
startActivity (i);
```

Usually the parent Activity needs to receive data from the child Activity. This is possible through the use of Intents with return values. The parent spawns a child by using `startActivityForResult()` instead of `startActivity()` and is notified when the child returns through a callback (the `onActivityResult()` function), as shown in **Listing 2**. This allows the parent to read the return code and any additional data returned by the child Activity.

As shown in **Listing 3**, the child Activity needs to call the `setResult` function, specifying its return status. If additional data should be returned to the parent, the child can attach an Intent along with the result code and supply extra key/value pairs, where the keys are Java Strings and the values are instances of Parcelable types, which are similar to Java Serializable classes and include Strings, arrays, and value types.

Listing 3 Code snippet showing how child Activity classes can return data to their caller.

```
1 Intent intent = new Intent();
2 intent.putExtra(“key”, “my value”);
3 this.setResult(RESULT_OK, intent);
4 finish();
```

Sending Intents to explicitly named Activity classes, as described above, is called *explicit Intent usage*. Android also allows creation of Intents specifying a triple (action, data type, category) and any Activity registered to receive those attributes through an intent filter will be able to receive the Intent. If there are multiple Activity classes that can receive the Intent, the user will be asked to select one.

The explicit Intent feature is mostly used in intra-application communication, as described in the following section, but can be useful for inter-application communication as well, and its existence is the root cause of the vulnerabilities discovered by us.

Inter-application Intents and data security

Interprocess communication with Intents

Intents can be used for communication between Activity classes of the same application or for inter-application communication. In the second case, Intents are actually interprocess message-passing primitives. To specify a subset of Intents that an Activity answers to, developers add to the application manifest an intent filter associated with the Activity. The intent filter in Listing 1 specifies that MyActivity can be invoked by sending an Intent with action `com.zxing.SCAN`; such an Intent is called an implicit Intent because it does not specify a particular Activity to be invoked. Implicit Intents are created using the single parameter constructor new `Intent(String)`.

Table 1 Different configurations lead to different levels of vulnerability.

Activity configuration		*Application configuration*	*Consequence*	
Exported	*Intent filter*	*SharedUid*	*Callers accepted*	*Risk level*
Exported="true"	Present	Any	Any	High
Exported="true"	Absent	Any	Any	High
Exported="false"	Present	Set	From same developer	Low
Exported="false"	Absent	Set	From same developer	Low
Default	Present	Any	Any	High
Default	Absent	Set	From same developer	Low

Component encapsulation

Developers enable or disable inter-application invocation of their Activity classes by setting the value of the Boolean exported attribute of each Activity in the application manifest. The behavior of this attribute is a detail that may be a source of confusion, as the meaning depends on the presence of another XML element, the intent filter:

- If an intent filter is declared and the exported attribute is not explicitly set to true or false, its default value is true, which makes the Activity accessible by any application.
- If an intent filter is not declared and the exported attribute is not set, by default the Activity is only accessible through Intents whose source is the same application.

An exception to the above rules is allowed if the developer specifies the attribute sharedUid in the manifest file. In that case, another application may run in the same process and with the same Linux user ID as the current application. This addition changes the behavior of Activity classes that are not exported: they can be invoked not only from the same application, but also from the sibling application with the same user ID. Listing 1 shows the use of the sharedUserId attribute.

It is important to realize that the intent-filter mechanism does not provide any security guarantees and is meant only as a loose binding between Activity classes and Intents; any Activity with an intent filter can still be sent an explicit Intent, in which case the intent filter is ignored. The presence of this attribute, however, changes the behavior of the security-related exported attribute, as detailed above. We found that many developers overlook the security-related implications when using intent filters.

Attacks on permission-protected information

All Android applications with a GUI contain at least one Activity, which means vulnerabilities related to Activity classes can affect most applications. All the vulnerabilities that we identify have in common the existence of information flows that are meant to allow child Activity classes to communicate with authorized parents but can instead be used by unauthorized applications to access sensitive information without explicitly declaring the corresponding permission.

We consider three different attack scenarios, which are discussed in the following paragraphs; our tool, PermissionFlow, identifies the flow vulnerabilities that enable all of them. PermissionFlow validation can be used as a prerequisite for applications before being listed in Android Market and by developers to ensure the security of their applications or by users.

Note that other operating systems run applications in a "sandbox" mode and do not offer a mechanism for direct inter-application communication; in spite of this, some of these attacks are still possible. For example, in iOS, the colluding applications attack can be performed through URL Schemes [8]. For these attacks to be possible, certain misconfigurations have to exist. These misconfigurations consist of a combination of implicitly public Activity classes (callable by unexpected callers) and misconfiguration of Activity permissions, which consists of failure to enforce the ownership of permissions on callers for Activity classes that return permission-protected information (previous work considered implicitly public Activity classes but did not test whether those Activity classes check for permissions of their callers, leading to some trustworthy applications being considered vulnerable (false positives).]

Misconfigured applications

Attacks on misconfigured applications occur when an attacker application installed on the device can exploit the flows of a misconfigured application. If an application has any one of the configuration parameter combinations listed in **Table 1** as high risk, then any application on the device can spawn it.

If in the application manifest an Activity is listed with an intent filter and is not accompanied by a exported = ''false'' attribute, any other application on the system can invoke it. Then, in the absence of declarative or dynamic permission checking by the developer, information returned to the caller through the Intent result may compromise

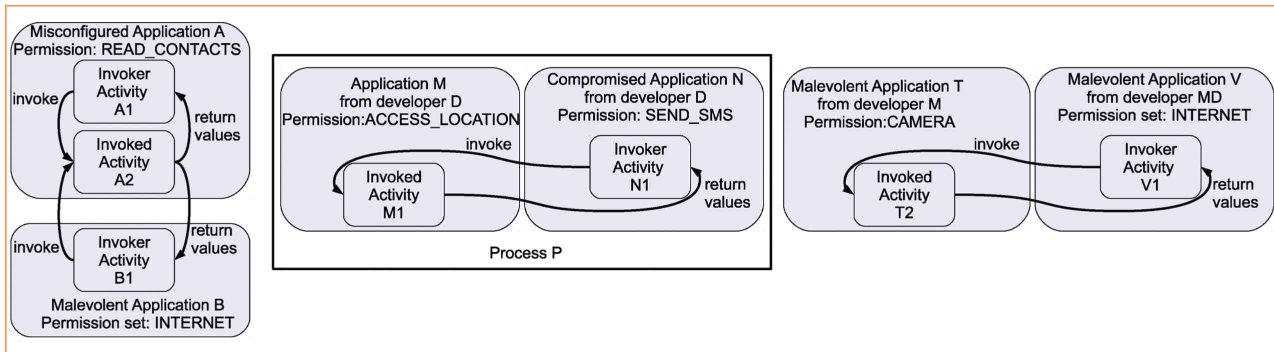

Figure 1

Possible attacks: internal Activity invocation or confused deputy, application sharing user ID with a compromised application (center) and permission collusion by malevolent applications (right).

permission-protected information, as no permission is required of the caller.

In the example from **Figure 1** (left), the user installed malevolent Application B (a music streaming app) with permission to access the Internet. Application B can exploit the honest but misconfigured contact manager Application A by invoking Activity A2 that returns the contacts; Application B can then send the contacts to a remote server. If A2 is built to reply to external requests and it simply failed to check that Application B has the proper permission, then the attack is a classic confused deputy attack. However, because in Android, Intents are also used as an internal (intra-application) communication mechanism, it is possible that A2 is not built for communicating with another application and is only misconfigured. This Intent spoofing is a more powerful attack than confused deputy for two reasons. First, it targets internal APIs, not just public entry points. These internal APIs are generally not regarded as vulnerable to a confused deputy attack and so they are not secured against it. By increasing the number of APIs that can be targeted, this attack increases the likelihood that the returned information is permission-protected. (Protected information tends to flow between internal components such as A1 and A2, even when it does not leave the application.) Second, the problem in this attack is not that the deputy performs a protected operation, but that it sends protected information to the callee.

For Activity classes that are designed to be invoked by unknown applications, developers can ensure that callers own a set of permissions in one of two ways: declaratively (in the manifest file, using the permission attribute of the Activity) or dynamically [by calling the function `checkCallingPermission(Stringpermission)`]. Note that the permission attribute can only be used to enforce a single permission and is different from the uses-permission node in Listing 1, which controls what permissions the application needs in order to function.

The safest approach is to completely disable outside access to internal Activity classes that may leak protected information. Table 1 shows the combinations of configuration parameters that may lead to information leaks.

Collusion

Collusion attacks obtain permission-protected information without requesting the permission, by exploiting the combination of assignment of Android permissions on a per-application basis and the exchange of applications information without making this explicit to the user.

In Figure 1 (right), we show a scenario in which a user is tricked by a malevolent developer MD into installing two separate applications that seem to have little risk associated with them. For example, a camera application that does not require Internet permission seems safe, as it cannot upload the pictures to the Internet. Likewise, a music streaming application that does not request camera permission would be acceptable. However, if the two applications are malicious, the music streaming application can invoke the camera application and send the pictures obtained from it remotely. The Android security system does not inform the user of this application collusion risk.

Applications sharing the user ID

We have not yet discussed an additional type of attack that our approach can recognize but that is improbable in practice because of its narrow applicability. This type of attack is on sibling applications.

The vulnerability in this case allows an attacker to access the permission-protected information of applications sharing the user ID with the already compromised application N. If N is configured to have the same user ID as application M (as shown in Figure 1 center), it can then obtain the

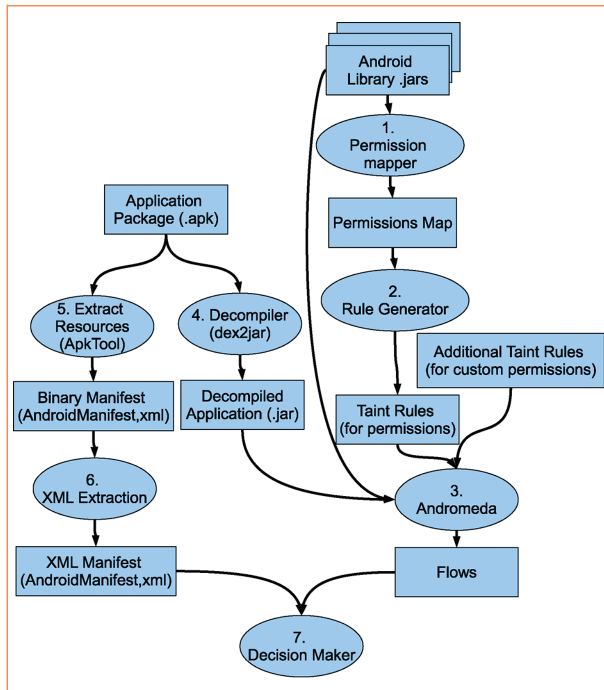

Figure 2

The components of PermissionFlow.

information from M. To set up this attack, an attacker would need to control application N of developer D, N should have any exfiltration permission (sending short messages, accessing the Internet, etc.), and N should share user ID with an application by D that returns permission-protected information. Then, N can invoke most Activity classes of M, even if M is configured according to the rows with low risk level in Table 1. PermissionFlow can detect this vulnerability as well.

PermissionFlow system

PermissionFlow has two main parts. The first one is a general, reusable taint analysis framework; the second consists of all other components, which are Android-specific.

To analyze real Android Market applications, whose source code is usually not available, we support input in the form of Android binary application packages (APK files). This means that PermissionFlow can also be used by Android users, developers, and security professionals.

The system design shown in **Figure 2** consists of the following components:

- The Permission Mapper (labeled 1 in the Figure 2) builds a list of method calls in the Android API that require the caller to own permissions. Its inputs are Android classes obtained by building the Android source code, with any modifications or additions performed by the device manufacturer. Having the complete system code as input allows the mapper to extract all the permissions-protected APIs that will be present on the device. It builds a permissions map, which maps permission-protected methods to their required permissions.
- The permissions map is passed to the Rule Generator, which builds the taint analysis rules relating the sources in the map with their corresponding sinks. In our case, the only sink is the Activity.setResult method with an Intent parameter.
- Our taint analysis engine (labeled 3) reads the generated rules and any extra rules manually added for detecting application-dependent private information. It outputs the flows that take the protected information from sources to sinks. For this, it needs access to the application classes and the Android library classes. The taint analysis engine also needs access to the Android library, in order to track flows that go through it, for example, callbacks that get registered and Intents that get passed to the system.
- The dex2jar decompiler [9] (labeled 4) is used to extract from the application APK a JAR archive containing the application byte code.
- To extract the binary application manifest from the application package, we use the ApkTool [11] (labeled 5); the decompilation step needed to get the textual XML representation is performed by AXMLPrinter2 [10] (labeled 6).
- The taint analysis engine outputs the flows from sources to sinks if there are any, but the presence of flows does not in itself imply that the application is vulnerable. The Decision Maker (labeled 7) looks for the patterns identified in Table 1 in the application manifest file; these patterns correspond to misconfigurations that allow successful attacks to take place. If an application contains a vulnerable information flow and is improperly configured, only then is it vulnerable. It is improperly configured if it is public (as shown in Table 1) and fails to enforce on its callers the permissions protecting the information it returns.

Permission Mapper

The Permission Mapper matches function calls used for permission enforcement in the Android libraries to Android library functions that use these calls. In short, it uses static analysis to identify permission-protected methods and to map them to their required permissions; this analysis is independent of any application analysis and needs to be performed only once for each input Android configuration.

Identifying sources of permission-protected information could also be attempted by crawling the documentation. However, it is incomplete even for public, documented classes and does not include public, but non-documented methods and does not account for Java reflection on

Listing 4 Code snippet showing how services check the permissions of applications.

```
1 public class VibratorService extends IVibratorService.Stub {
2   public void vibrate(long milliseconds, binder) {
3     if (context.checkCallingOrSelfPermission(VIBRATE)) !=
  PackageManager.PERMISSION.GRANTED) {
4               throw new SecurityException("Requires VIBRATE permission");
5     }
6   }
7 }
```

non-public methods. It also does not account for any modifications and additions to the Android API performed by the phone manufacturer. Felt et al. [12] showed that it is possible to identify which API calls require permissions through a combination of automated testing and manual analysis, but they use techniques that allow false negatives, need partial manual analysis and do not handle the version and feature fragmentation [13] of Android well. For these reasons, we built the Permissions Mapper, a reliable and automatic tool for identifying permission-protected APIs and their required permissions.

Most Android permissions are enforced through calls to the `checkPermission` function of the Context and PackageManager classes or the `checkCallingOrSelfPermission` function of the Context class; our work targets complete coverage of APIs enforced through this mechanism, which includes the majority of Android permissions. Other APIs are enforced through native code or Linux users, which we do not consider.

To illustrate how permission checks work, we can use, for example, the VIBRATE permission. To use the phone vibrator, an application needs to own the VIBRATE permission; all functions that require this permission check for it.

When this function is called, the Android API forwards the call to a system service. The service process is the one that makes the actual permission checks, before performing any protected operation, as shown in **Listing 4**. The proxy for the service performs the interprocess communication, and because of this, it appears as a leaf in the call graph. In our analysis, we automatically fill in the missing edges between proxies and their corresponding services (relying on the name correspondence between the two, which is enforced by the Android AIDL code generation).

The interprocedural dataflow analysis is built using IBM WALA [14] and starts by building the call graph of the Android libraries, including all methods as entry points. All call chains containing a `Context.checkPermission(String)` method call are then identified. To find the actual permission string that is used at a `checkPermission` call site, we follow the definition-use (def-use) chain of the string parameter. Once found, we label all callers upstream in the call chain as requiring that permission. Note that different call sites of `checkPermission` will have different permission strings. Each such string needs to be propagated correctly upstream, building a set of required permissions for each function.

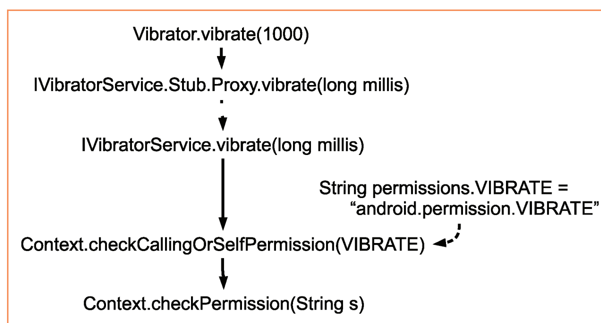

Figure 3

Permission analysis exemplified for the vibrate() call. The call graph edge from the proxy to its corresponding service is automatically added before the analysis.

For the `vibrate()` example, the permission enforcement call chain is in **Figure 3**. The string value of the vibrate permission is located by following the def-use chain of the `checkPermission` parameter (dashed lines) until the source `String` constant is found. Once the constant is found, we need to identify which functions on the call chain need this permission. We start by labeling the function that contains the first ("most downstream") call site through which the def-use chain flows. In our case, the def-use chain goes first through the call site of `checkCallingAndSelfPermission` in `IVibratorService.vibrate` (line 3 in Figure 3), where the VIBRATE variable is specified as a parameter, so `IVibratorService` is labeled with "`android.permission.VIBRATE`."

After labeling `IVibratorService.vibrate`, the same label must be propagated to callers of that function, but in our case there are no callers except the proxy stub.

Because the communication between Android proxy stubs and their corresponding services (shown as dotted edges in Figure 3) is done through message passing, it does not appear

in the actual call chain built by WALA. To work around this problem, we add the permissions labels of the service methods to the corresponding proxies; the labels are then propagated to any callers of those methods.

Taint analysis engine

For the taint analysis engine, we used Andromeda [15], which is highly scalable and precise as well as sound; it builds on IBM WALA [14]. Andromeda uses, as input, rules composed of two sets: sources and sinks. The sources are parameters and return values of functions that are the origin of tainted data and the sinks are security-critical methods. The engine tracks data flow from the source through assignments, method calls, and other instructions, until the data reaches a sink method. If the taint analysis engine discovers that tainted data reaches a sink method, the flow is included in the taint analysis engine output. We discuss the soundness of the complete PermissionFlow analysis in the section "Evaluation of PermissionFlow").

Experimental results

Evaluation of the permissions map

We evaluated the Android API Permissions Mapper by directly comparing its output permissions map with that of Felt et al. [12], based on automated and manual testing. To perform the comparison with the work by Felt et al., we eliminated permissions from their map that are enforced through mechanisms other than the `checkPermissions` calls, because our analysis only targets `checkPermissions`-enforced permissions.

Comparing the size of their reference map (which includes 1,311 calls that require permissions) with ours (4,361 calls with permissions) shows that our tool finds more functions that require permissions. The larger size of our map is partly explained by the lack of false negatives for the analyzed Java APIs. However, a direct comparison is not possible, because the input classes on which Felt et al. ran their analysis is not specified in their paper.

Through manual comparison, we identified one false negative in their map (probably due to the automated testing not generating a test and the subsequent analysis not detecting the omission). The function is `MountService.shutdown`, which usually needs the `SHUTDOWN` permission, but if the media is shared, it also needs the `MOUNT_UNMOUNT_FILESYSTEMS` permission. The existence of missing permissions in the testing-based methods shows that testing methods, even if party automated and enhanced by manual analysis, cannot offer guarantees with respect to false negatives.

Another reason for the higher number of methods found in our map is the existence of false positives: permissions that are reported as required but are not. We have identified the following sources of false positives, all of which are known weaknesses of static taint analysis:

- Checking for redundant permissions. For example, checking for `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` in `TelephonyManager.getCellLocation()`, where either one is sufficient for enforcement; our method reports both, because it is oblivious to control flow.
- Data-dependent checks. For example, a check for the `VIBRATE` permissions depends on the value of a parameter such as in `NotificationManager.notify()`.
- Android provides the pair of functions `clearIdentity` and `restoreIdentity` that are used to change all checks so that they are performed on the service instead of the application using the service.

Another advantage of testing-based analysis is that it can cover areas, such as the permissions enforced in native code, which static analysis does not target (such as `RECORD_AUDIO`).

To perform a more detailed comparison of our approach with the work by Felt et al., we compared results obtained for a simple security analysis, identification of overprivileged applications, based on the permission map. This analysis consists of identifying Android applications that request in their manifest more permissions than they actually need to perform their functions. The results are used to reduce the attack surface of applications by removing unused permissions from the manifest. To perform this analysis, we built another static analysis tool based on IBM WALA, which records the API calls that can be performed by an application and computes, based on the permissions map, the permissions required by that application. The set of discovered permissions is then compared to the set of declared permissions.

We used both permission maps as input for the analysis of the Top Android Market Free Applications (crawled in December 2011) that were compatible with Android 2.3 and available in the United States (354 applications). For a fair comparison, we removed from Felt et al.'s reference map the parts that relates content provider databases to their permissions, as these were outside the scope of our work. After eliminating applications that crashed the dex2jar [9] decompiler or generated incorrect byte code, we were left with 313 applications. Of these, both our analysis and theirs found 116 to be overprivileged. No permissions identified as unused by us were identified as used by Felt et al., which is consistent with the lack of false negatives expected from a static analysis approach. However, 47 permissions were identified as used by us and as unused by Felt et al., which is a false-positive rate of 4.8%.

PScout [16] is a permission mapper built through static analysis based on Soot. The number of overprivileged applications found by PScout is consistent with both

Felt et al. and our tool. If we compare the number of entries in the map, PScout finds 17,218 APIs, we find 4,361, and Felt et al. list 1,311. This inconsistency comes from four sources. First, different versions of Android are analyzed by each. Second, different subsets of classes may be selected as input from the Android code. Third, the tools have different levels of false positives (Felt has none, because it is a dynamic analysis, but PScout and our mapper may have some). Fourth, there are different levels of false negatives (Felt et al. may have missing permissions listed because of incomplete coverage). PScout reports a false-positive rate of 7% on real applications, whereas we have a rate of 4.8% (the false-positive rate is not a clear indicator, as the input applications used for the two systems were not the same, but it is consistent with the difference in the map sizes). As in our analysis, this tool omits permissions enforced through non-Java mechanisms. It includes permissions required for ContentProvider access, which we do not consider in our mapper (the following subsection explains how to include ContentProviders and Services in the PermissionFlow analysis).

The technical report [17] contains a comparison of the Permission Mapper to a similar tool by Bartel et al. [18].

Evaluation of PermissionFlow

We tested PermissionFlow on the same applications used to evaluate the permission map. To confirm the correctness of the results we manually inspected all applications. Out of the 313 applications, 177 use the Activity.setResult with an Intent parameter to communicate between components (both internal and external). These 56% of applications may be vulnerable if they also contain flows from taint sources to sinks and are not configured properly. They can use PermissionFlow to check that they are secure.

To check for correctness, we ran PermissionFlow with our permissions map and the one produced by Felt. Using the map from Felt et al., PermissionFlow correctly identified two applications as vulnerable and had no false positives. With the permissions map built by our analysis, PermissionFlow outputs a larger set of vulnerable applications, but the additional applications are all false positives. As we saw in the previous section, both permissions maps are incomplete: ours does not track permissions enforced through non-Java mechanisms and Felt et al.'s allows the possibility of missing permission checks. Choosing one of the two maps amounts to either using a possibly incomplete map (Felt et al.) and finding no false positives, or identifying the complete set of Java-based flows and accepting some false positives but missing flows based on native code or Linux permissions checks.

Our analysis is sound with respect to the subset of Android that we consider, which includes Activity classes. To maintain soundness with respect to chains of communicating Activity classes (from the same or different applications), we taint the information returned by the `Intent.getExtra` family of calls with the set of permissions owned by the current application. If the invoked Activity is from a different application, that application is assumed to be validated with PermissionFlow, so that it enforces the correct permissions on the current (caller) application. We need to guarantee that the incorporation of other Android components does not break the soundness of our analysis. To ensure security, these components should individually maintain the following invariant: if any information protected by permission P flows into the component, then the component must enforce permission P through its manifest, on any components that read that information. This invariant can be checked in the same manner that PermissionFlow checks Activity classes (on each component individually). To support Services, more sinks need to be considered other than `Activity.setResult()`. For inter-application ContentProviders, we need access to the manifest that declares them, but we still do not need to analyze the source code of both simultaneously (PermissionFlow already batches applications. This would be sufficient for a safe ContentProvider analysis). None of these changes alters the dataflow analysis itself, only its input and output. Thus, we maintain soundness when using a sound taint analysis engine such as Andromeda.

Out of the set of 313 applications, we found that three are vulnerable to permission-protected information leaks: Adobe Photoshop Express, SoundTracking, and Sygic GPS. Adobe Photoshop Express and SoundTracking allow any application to send Intents to components that return permission-protected information, so their risk is high. Sygic can only leak such information to applications with the same user ID, so the risk level is low. The permission information compromised is the user contacts (Adobe Photoshop Express), the location (SoundTracking), and the camera pictures (Sygic). All of the vulnerabilities lead to a window being displayed to the user, corresponding to the invoked Activity from the callee application. If the user performs some activity in the window (selecting a contact, taking a picture), that information will leak. The window that pops up is familiar and belongs to a trusted application, so the user may unwittingly cooperate. Further, even though we did not find any, there may be other applications for which attacks do not require user intervention. See the technical report [17] for more details.

As examples of confidentiality violations that PermissionFlow identifies when provided with application-specific sources of taints, the GO SMS and GO Locker applications leak the lockscreen password. Since they have an identical pattern, we consider these GO applications to be the fourth vulnerability identified by the PermissionFlow tool. More information about the specific information flows that lead to these vulnerabilities can be found in the technical report [17].

Related work

Privilege escalation attacks on Android applications have been previously mentioned in the literature [19]. However, such attacks require usage of native code, careful identification of buffer overflow vulnerabilities and high expertise. We focus only on vulnerabilities specific to Android and help protect the information before such attacks happen.

Grace et al. [20] focused on static analysis of stock Android firmware and identified confused deputy attacks that enable the use of permission-protected capabilities. Our analysis is complementary in that it identifies not actions that are performed, but information that flows to attackers. In addition, we focus not on stock applications, but third-party applications.

TaintDroid [21] uses dynamic taint tracking to identify information flows that reach network communication sinks. Both PermissionFlow and TaintDroid can potentially support other sinks, and their dynamic approach is complementary to our static approach because it can better handle control flow (e.g., paths that are never taken in practice are reported as possible flows by our tool). It can also enforce only safe use of vulnerable applications by denying users the capability to externalize their sensitive information.

SCanDroid [22] is the first static analysis tool for Android and can detect information flow violations. The tool needs to have access to both the vulnerable application and the exploitable application. To the best of our knowledge, SCanDroid is not easily extensible with new taint propagation rules.

CHEX [23], a system developed concurrently with our work, relies on static analysis to discover permission leaks in Android applications. CHEX uses an IR similar to the one used by WALA but does not use WALA dataflow analysis. For efficiency, they use a graph reachability analysis. Andromeda, our taint analysis engine, achieves efficiency through use of a demand-driven taint analysis. CHEX detects several types of vulnerabilities affecting Android applications, including permission-protected information leaks. However, CHEX does not check the application manifest to identify if Activity classes are exported or if Activity classes use the manifest to enforce permissions from their callers. CHEX requires a permissions map as input, in that it does not automatically generate it.

ComDroid [7] is a tool that analyses inter-application communication in Android. ComDroid does not track permission-leak vulnerabilities. None of the vulnerabilities described pertains to permission-protected information. ComDroid does emit misconfiguration warnings, but these are not necessarily vulnerabilities (some applications offer public services that need no checking). Permission-leak attacks are a special case of Intent spoofs in that they imply permission-protected information flow, not only control flow and misconfiguration. Contributions such as automatic rule generation and automatic permission map building separate our work from theirs.

Kirin [24, 25] is a tool based on a formal representation of the Android security model that checks if applications meet security policies. It can check for confused deputy vulnerabilities ("unchecked interface"), Intent spoofing ("intent origin"), and other attacks by using a powerful Prolog-based security policy enforcement mechanism, which takes into consideration the set of applications already installed on a device. The authors point out several difficulties with creating information flow policies in Android and discuss the future possibility of including source code analysis to make information flow policies for Android practical. If we consider the PermissionFlow rules as information flow security policies, then PermissionFlow is a step in the right direction for such a tool. It would solve the problem they mention of information flowing into any application with a user interface.

Felt et al. [12] perform a similar analysis to our PermissionMapper. Their work is based on automated testing rather than static analysis, which means incomplete coverage and the possibility of false negatives in the permissions map. They do not use the map to check for information flow-based vulnerabilities in applications. Their work was discussed earlier in this paper.

As discussed in this paper, PScout [16] builds a permission mapper through static analysis based on Soot.

Kantola et al. [26] propose to modify Android configuration semantics to implicitly mark fewer Activity classes as public. Their heuristic-based approach fixes most vulnerabilities and maintains backwards compatibility with applications written for the current Android semantics. The cost is the continuing reliance on heuristics, and vulnerabilities are still possible. A cleaner approach would be to disallow inter-application invocations if the caller does not own the permissions required by the callee. Such an approach prevents these vulnerabilities by correctly configuring applications, but breaks backwards compatibility. Another solution is to emit warnings based on the configuration and static analysis, which PermissionFlow would be well-suited for. In the technical report [17] we provide recommended practices for safe application configurations for security-aware Android developers.

Hornyack et al. [27] describe a tool that can be used to complement ours.

Mann and Starostin [28] propose a wider analysis based on typing rules that can discover flow vulnerabilities. We discuss both papers in the technical report.

Conclusion

This paper describes an automated solution for the problem of checking for leaks of permission-protected information; this is an important security problem for mobile devices,

as such leaks compromise users' privacy. We demonstrate the benefits of this analysis on Android, and identify the Intent mechanism as a source of permission leaks in this operating system. We found that permissions can leak to other applications even from components that are meant to be private, i.e., accessed only from inside the application.

Our automated analysis of popular applications found that 56% of the top 313 Android applications actively use inter-component information flows. Among the tested applications, PermissionFlow found four exploitable vulnerabilities. Because of the large scale usage of these flows, PermissionFlow is a valuable tool for security-aware developers, for security professionals and for privacy-conscious users. Our approach extends beyond Android, to permission-based systems that allow any type of inter-application communication or remote communication (such as Internet access). Most mobile operating systems are included in this category and can benefit from the proposed new application of taint analysis. By helping ensure the absence of inter-application permission leaks, we believe that the proposed analysis will be highly beneficial to the Android ecosystem and other mobile platforms that may use similar analyses in the future.

**Trademark, service mark, or registered trademark of Google, Inc., Microsoft Corporation, Linus Torvalds, Symbian Software, Sun Microsystems, Linus Torvalds, or Apple, Inc., or Adobe Systems, Inc., in the United States, other countries, or both.

References

1. IPhone App Store, *Apple*, (Accessed Dec. 2011).
2. *Android Market Insights*, Research2Market, Woodbridge, ON, Canada, 2011. [Online]. Available: http://www.research2guidance.com/shop/index.php/android-market-insights-september-2011
3. *Android Security Guide*, Google, Mountain View, CA, USA, 2012. [Online]. Available: http://developer.android.com/guide/topics/security/security.html
4. *Windows Phone 7 Security Model*, Microsoft, Redmond, WA, USA, 2012. [Online]. Available: http://www.microsoft.com/download/en/details.aspx? displaylang=en&id=8842
5. *Meego Help: Assigning Permissions*, The Linux Foundation, San Francisco, CA, USA, Accessed Dec. 2011. [Online]. Available: http://wiki.meego.com/Help:Assigning_permissions
6. *Nokia. Symbian v9 Security Architecture*, 2012. [Online]. Available: http://library.developer.nokia.com/
7. E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proc. 9th Int. Conf. MobiSys, Appl. Services*, New York, NY, USA, 2011, pp. 239–252.
8. *Passing Data Between iOS Applications*. [Online]. Available: http://kotikan.com/blog/posts/2011/03/ passing-data-between-ios-apps
9. *Opensource, Dex2jar Google Code Repository*. [Online]. Available: http://code.google.com/p/dex2jar/
10. *Opensource, Java 2 Me Port of Google Android*. [Online]. Available: http://code.google.com/p/android4me/downloads/list
11. *Opensource, Android Apktool Google Code Repository*. [Online]. Available: http://code.google.com/p/android-apktool/
12. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. CCS*, New York, NY, USA, 2011, pp. 627–638.
13. *Android Fragmentation Information*. [Online]. Available: http://www.androidfragmentation.com
14. *T. J. Watson Libraries for Analysis (WALA)*. [Online]. Available: http://wala.sourceforge.net/wiki/index.php/Main_Page
15. O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri, "Andromeda: Accurate and scalable security analysis of web applications," in *Fundamental Approaches to Software Engineering*. Berlin, Germany: Springer-Verlag, 2013, pp. 210–225, ser. Lecture Notes in Computer Science.
16. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the android permission specification," in *Proc. ACM Conf. CCS*, New York, NY, USA, 2012, pp. 217–228.
17. D. Sbýrlea, M. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar, "Automatic detection of inter-application permission leaks in Android applications," Rice Univ., Houston, TX, USA, Tech. Rep. [Online]. Available: http://www.cs.rice.edu/~vsarkar/PDF/PermissionFlow-TR.pdf
18. A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Automatically securing permission-based software by reducing the attack surface: An application to android Univ. Luxembourg, Walferdange, Luxembourg, Tech. Rep. 978-2-87971-107-2, 2011.
19. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on Android *Proc. 13th ISC*. Berlin, Germany: Springer-Verlag, 2011, pp. 346–360.
20. M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proc. 19th NDSS*, 2012, pp. 1–15. [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/pubs/NDSS12_WOODPECKER.pdf
21. 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 real-time privacy monitoring on smartphones," in *Proc. 9th USENIX Conf. OSDI*, 2010, pp. 1–6.
22. A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "SCanDroid: Automated Security Certification of Android Applications Dept. Comput. Sci., Univ. Maryland, College Park, MD, USA, Tech. Rep. CS-TR-4991, Nov. 2009.
23. L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. CCS*, New York, NY, USA, 2012, pp. 229–240.
24. W. Enck, M. Ongtang, and P. McDaniel, "Mitigating android software misuse before it happens," Netw. Sec. Res. Center, Pennsylvania State Univ., Pennsylvania, PA, USA, Tech. Rep. TR-0094-2008, Nov. 2008.
25. W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Sec. Privacy*, vol. 7, no. 1, pp. 50–57, Jan./Feb. 2009.
26. D. Kantola, E. Chin, W. He, and D. Wagner, "Reducing attack surfaces for intra-application communication in android," in *Proc. 2nd ACM Workshop SPSM Devices*, New York, NY, USA, 2012, pp. 69–80.
27. P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall, "These aren't the droids you're looking for: Retrofitting Android to protect data from imperious applications," in *Proc. 18th ACM Conf. CCS*, New York, NY, USA, 2011, pp. 639–652.
28. C. Mann and A. Starostin, "A framework for static detection of privacy leaks in android applications," in *Proc. 27th SAC, Comput. Sec. Track*, 2012, pp. 1457–1462.

Received January 21, 2013; accepted for publication March 13, 2013

Dragos Sbîrlea *Department of Computer Science, Rice University, Houston, TX 77005 USA (dragos@rice.edu).* Mr. Sbirlea is a Ph.D. candidate in the Habanero Multicore Software Research group at Rice University. He is the author or coauthor of five scientific publications. His research interests include analysis and optimization of parallel programs. He is a contributor to the Sandia Qthreads library and a member of the Association for Computing Machinery (ACM).

Michael G. Burke *Department of Computer Science, Rice University, Houston, TX 77005 USA (mgb2@rice.edu).* Dr. Burke

has been a Senior Research Scientist in the Computer Science Department at Rice University since 2009. He received a B.A. degree in philosophy from Yale University in 1973 and M.S. and Ph.D. degrees in computer science from New York University in 1980 and 1983, respectively. He was a Research Staff Member in the Software Technology Department at the IBM Thomas J. Watson Research Center from 1983 to 2009, and a manager there from 1987 to 2000. His research interests include programming models for high-performance parallel and big data computing, and static analysis-based tools for programming language optimization; database applications; mobile security; compiling for parallelism. He is an author or coauthor of more than 50 technical papers and 11 patents. Dr. Burke is an ACM Distinguished Scientist and a recipient of the ACM SIGPLAN Distinguished Service Award. He is a member of the Institute of Electrical and Electronics Engineers.

Salvatore Guarnieri *IBM Software Group, Yorktown Heights, NY 10598 USA (sguarni@us.ibm.com).* Mr. Guarnieri is a Software Engineer in the Security Division of IBM Software Group, and a Ph.D. student in the Department of Computer Science of University of Washington. He is an author or coauthor of six scientific publications. His research interests include program analysis and security of scripting languages.

Marco Pistoia *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (pistoia@us.ibm.com).* Dr. Pistoia is a Manager and Research Staff Member at the IBM T. J. Watson Research Center, where he leads the Mobile Technologies Group. He has worked with IBM for 17 years. He holds a Ph.D. degree in mathematics from New York University, Polytechnic Institute, and B.S. and M.S. degrees in mathematics from the University of Rome, Tor Vergata, Italy. His research interests include program analysis, programming languages, security, and mobile technologies. He is an author or coauthor of 10 books, 30 technical papers, 14 patents, and 68 patent applications, and the recipient of two ACM SIGSOFT Distinguished Paper Awards.

Vivek Sarkar *Department of Computer Science, Rice University, Houston, TX 77005 USA (vsarkar@rice.edu).* Dr. Sarkar is a Professor of Computer Science and the E.D. Butcher Chair in Engineering at Rice University. He conducts research in multiple aspects of parallel software including programming languages, program analysis, compiler optimizations, and runtimes for parallel and high performance computer systems. Prior to joining Rice in July 2007, he was Senior Manager of Programming Technologies at IBM Research. His past projects include the X10 programming language, the Jikes Research Virtual Machine for the Java language, the ASTI optimizer used in the IBM XL Fortran product compilers, the PTRAN automatic parallelization system, and profile-directed partitioning and scheduling of Sisal programs. He became a member of the IBM Academy of Technology in 1995 and was inducted as an ACM Fellow in 2008. He holds a B.Tech. degree from the Indian Institute of Technology, Kanpur, an M.S. degree from University of Wisconsin-Madison, and a Ph.D. degree from Stanford University.