2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications

Systematic Analysis and Detection of Misconfiguration Vulnerabilities in Android Smartphones

Zhihui Han, Liang Cheng, Yang Zhang, Shuke Zeng, Yi Deng, Xiaoshan Sun
Trusted Computing and Information Assurance Laboratory, Institute of Software, Chinese Academy of Sciences
Email: hanzhihui@tca.iscas.ac.cn

***Abstract*—Android is a modern and popular software platform for smartphones. To manage information and features on smartphones, Android employs intent-based mechanism for inter-application or intra-application communication and provides a permission-based security model that requires each application to explicitly request permissions in its manifest file. However, misconfiguration defined in manifest files and that embedded in application code may result in vulnerabilities due to developer confusion and general misuse of the features provided by Android. In this paper, we propose a logic-programming-based approach to analyze smartphones and discover misconfiguration vulnerabilities in Android manifest file and application code. To enable misconfiguration vulnerability analysis and detection, we develop a static technique to extract security related information from application code, and employ logic predicates to describe various vulnerabilities. Based on this approach, we developed a tool called SADroid to systematically analyze and detect misconfiguration vulnerabilities in Android smartphones. Our results with two representative phones show that the inherent weakness of Android permission model and developers' programming errors make Android vulnerable to some attacks.**

***Keywords*—Android smartphone, static analysis, misconfiguration vulnerability, logic programming**

I. Introduction

Android smartphones are increasingly prevalent in recent years. The tremendous popularity can be attributed to phones' evolution from simple devices used for phone calls and SMS messages to powerful communication and entertainment platforms for web surfing, social networking, GPS navigation, and online banking. End users can customize their smartphones by installing third-party applications(or simply apps) from markets that host hundreds of thousands of applications. Abundant functionalities and useful features of smartphone make users relying on smartphones to store and handle personal data, such as emails, photos, contact information and even bank accounts. Increasing personal and sensitive data stored in smartphones attracts attackers to target mobile smartphones.

There are many attacks and threats on Android, such as privilege escalation [18], [11], remote control [30], phishing [26], intent spoofing [10] and component hijacking [24]. These attacks and threats are caused by different vulnerabilities. Some are caused by users' incaution, such as phishing and component hijacking. Some result from bugs in application code, such as privilege escalation caused by an application with a heap overflow vulnerability. And some are results of misconfiguration, such as privilege escalation and intent spoofing.

Although previous works [12], [25], [28] have implied that Android's permission-based model is not strong enough to completely protect Android from permission re-delegation attack, developers and vendors can make their effort to reduce security threats through using Android features correctly and securely in app and phone image development. Unfortunately, although lots of security develop notes are provided in Android document, most of app developers, who are lack of security knowledge, do not realize the importance of these notes and do not configure their apps to use Android features securely. The misconfiguration makes apps and Android system vulnerable to some threats and attacks. For example, developers always use the default configuration set by develop tools, e.g. Eclipse with ADT, which often lead to vulnerabilities, such as intent spoofing [10] and confused deputy attack [23].

In this paper, we propose a logic-based approach to systematically analyze and detect misconfiguration vulnerabilities and threats in Android smartphone images coming from some leading manufactures, including Google and Samsung. Our work focuses on the vulnerabilities which are caused by app developers who do not follow Google's security notes and misconfigure their apps, especially the apps developed by Android manufactures and pre-installed in Android phone images. Two types of configuration are analyzed in our work. The first type is the configuration defined in app's manifest file, including components, enforced permissions, exported attribute value and intent filters. The second type is dynamically defined in app code, which consists of dynamic receiver constructors, intents, and privileged Android APIs. Our approach leverages well-studied static analysis to extract the two types of configuration for each pre-installed apps, and then use logic programming to emulate Android's inter-component communication (ICC) mechanism to examine how pre-installed apps interact with each other under the constrain of the extracted configuration and generate an ICC graph of the whole image. By modeling some security rules provided by security notes in Android document and some security threats derived from attacks with logic predicates, we can check security rules and detect security threats over the computed ICC graph. With the help of check and detection results, developers can secure their apps by modifying and recompiling source code and Android manufactures can fix the vulnerabilities in their phone images with binary rewriting.

We have implemented a SADroid prototype to analyze and detect misconfiguration vulnerabilities in Android smartphones. Its main components are Static Analyzer, which performs static analysis on each pre-installed apps to extract security configuration, and Logical Checker, which logically specifies security rules and security threats with logic predicates and check them with logical inference. We have used our prototype to analyze four smartphone images for Google

978-1-4799-6513-7/14 $31.00 © 2014 IEEE
DOI 10.1109/TrustCom.2014.56

Nexus 4 and Samsung Galaxy S3. SADroid finds out many violations of security rules and some potential attacks in all of the four phone images. Our results also show that all of the four images have misconfiguration and some components are incorrectly protected in some ways by intent-filters instead of protected by permissions. Our contributions are the followings:

- We describe a static analysis based method to discover exhaustive security-related configuration of Android apps.
- We propose a scalable approach to analyze and detect vulnerabilities and potential threats caused by misconfiguration due to developer confusion and general misuse of Android features.
- We implemented a prototype tool called SADroid, and used our tool to analyze four mainstream Android phone images. Our tool finds out various misconfiguration and illustrates that it is common for a developer to ignore permission protection and use intent filters instead of permissions to protect his components .

The rest of the paper is organized as follows. In section II we present an overview of Android, focusing on Android configuration and intent-based inter-component communication. Section III and section IV describe our system design and implementation. In section V, we present and discuss our experiment results when applying SADroid to analyze four popular Android phone images. Section VI provides discussions of SADroid. Section VII is dedicated to related work, and in section VIII we conclude.

II. Android Overview

A. Android Manifest File

Every app must have an AndroidManifest.xml file (with precisely that name) in its root directory. The manifest file presents essential security-related information about the app to the Android system, and this information constitutes the static configuration of the app. First, the manifest describes the components of the application - the *activities*, *services*, *broadcast receivers* and *content providers* that the application is composed of. It names the classes that implement each of the components and publishes their capabilities through *exported* attribute value and *intent filters* (except content provider). A broadcast receiver can also be dynamically declared with *registerReceiver()* API in the runtime. Second, the manifest declares which permissions the app must have in order to access protected parts of the API and interact with other apps, which must be granted in order to complete the installation. Third, the manifest also declares the permissions that others are required to have in order to interact with the app's components.

B. Intent-Based Inter-Component Communication

Android applications are written in Java, and Android defines four types of app components to build applicaitons: *activities*, *services*, *broadcast receivers* and *content providers*. Three of the four component types - activities, services, and broadcast receivers - are activated by an asynchronous message called *Intent*, and intents can be sent between these three types of components. Intents can be used to start activities; start, stop or bind services; and broadcast information to broadcast receivers. There are two primary forms of intents: explicit intents and implicit intents. An explicit intent specifies an exact component to which it should be delivered, and explicit intents are typically used to start a component in the same app. Whereas an implicit intent does not name a specific component but includes enough information for the system to determine which of the available components is best to run for that intent. To advertise which implicit intents an app can receive, it is need to declare one or more intent filters for components in its manifest file. The system will deliver an implicit intent to the app component only if the intent can pass through one of the intent filters. Using an explicit intent guarantees that the intent is delivered to the intended recipient, while implicit intents allow for runtime matching between different apps. It is important for developers to realize that the intent-filter mechanism does not provide any security guarantees but only a matching mechanism between components and intents.

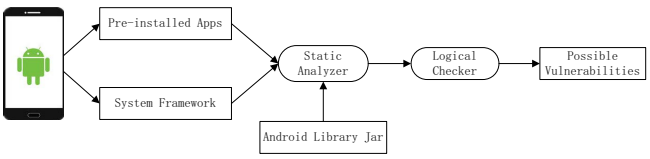

Fig. 1. Overview of SADroid

III. System Design

We aim to analyze and detect vulnerabilities and threats due to developers' misconfiguration. Fig. 1 provides an overview of our system named SADroid. SADroid consists of two primary components: Static Analyzer and Logical Checker. For the Android phone to analyze, SADroid first extracts all the pre-installed apps and system framework files, where the set of pre-installed apps is our analysis target and the framework files provide essential assistant information of the target Android system.

Then for each pre-installed app, Static Analyzer parses its manifest file to extract the static configuration, including app declared permissions, app enforced permissions, component declarations, component permissions and some other information. There is also some dynamic configuration in app code which is constructed in the runtime, such as dynamic receivers, intents and privileged API employment. To save analysis cost and time in getting the dynamic configuration, we do not run each pre-installed app, but apply well-studied static analysis technique on each of them to gain approximate but overall results. All the configuration is encoded as logic predicates and input into Logical Checker.

Based on the encoded configuration, Logical Checker formally describes Android intent-based mechanism and permission model with logic predicates, and then examines how components interact with each other to generate an overall ICC graph. Finally, Logical Checker makes some logical inference on the formal description or ICC graph to check security rules and detect potential attacks, which we will present later.

A. Static Analyzer

Given each pre-installed app in the phone image to inspect, most of its configuration is defined in its manifest file, so our system first extracts the binary app manifest file from

the application package suffixed by apk, and decompiles the manifest file to a textual XML representation. By parsing the XML file, we can get the app's static security configuration. Such information includes application's name, declared permissions, application enforced permissions, exported flag attribute value, sharedUserId attribute value, components and associated enforced permissions, and intent filters.

Besides the static configuration, some other security information is defined in app's Dalvik bytecode (either classes.dex or its odex variant), including dynamic broadcast receivers, intents or broadcasts, content provider access, and uses of privileged permissions. To obtain this information, Static Analyzer performs static analysis on app's Dalvik bytecode.

Dynamic broadcast receivers can be dynamically created and registered in app's code by calling *registerReceiver(BroadcastReceiver receiver, IntentFilter filter)* or *registerReceiver(BroadcastReceiver receiver, IntentFilter filter, String broadcastPermission, Handler scheduler)*. The intent filter or broadcast permission is specified at registration time and can be changed each time *registerReceiver* is invoked, so we consider each registration of a receiver as a unique component and label it with the location where it is registered. For intent filter and permission parameters associated with registered receivers, our system first builds a control-flow-graph (CFG) and gains their values by identifying and tracking their class constructors, instantiations and setting methods on this CFG.

Intents or broadcasts are used as parameters of some Android ICC APIs, such as *startActivity(Intent intent)*. In this paper, we call such ICC APIs *intent sink methods*. In order to get the Intent parameter's value, Static Analyzer first traverses the app bytecode to locate each intent sink method. Next, Static Analyzer backtracks app bytecode from the location of each intent sink method, and builds a control-flow-graph. Based on the control-flow-graph, Static Analyzer computes each Intent parameter's value by tracking each Intent object's construction, instantiation, and transition until the sink location. For each Intent parameter, not only the value is needed to record, but also the location of intent sink method must be record, including app's package name, class and method name invoking the intent sink method in user app. A broadcast is a special intent associated with a certain receiver permission. So we can obtain a broadcast's value in the similar way with an intent.

Content provider access can be issued by a component in the form of Uri, and Android system can address the Uri to the requested content provider. Content provider access is security sensitive, so we should identify all the content accesses in user app. A component can request a content access through calling methods in *android.content.ContentResolver* class with an Uri as its parameter, including *delete, insert, query* and *update*. Static Analyzer computes each Uri parameter's value similarly to Intent parameters.

Security-sensitive APIs are such APIs that are protected by permissions providing for apps to access private user data or certain device components(e.g., Bluetooth) [22]. There are more than one hundred permissions defined by Android system [6], and developers can also define their own permissions. We choose 28 representative permissions in our study considering three principles: (1)The permission must be frequently used in apps [17], (2) The protection level of the permission is *dangerous, signature, or systemorsignature*, (3) The permission may compromise users or system if misused. For each chosen permission, we identify the list of related Android APIs that exercise the permission. However, such a list is not easy to come by because Android API document is incomplete about the relationship between APIs and permissions. Fortunately, Felt's work [17] gives out permission-API mapping result, although their target platform is Android 2.3 which is old for our system. Our system aims at Android 4.2 (API level is 17) or higher versions, so based on the mapping result, we remove some deprecated APIs, and add some new related APIs to generate a new permission-API mapping file.

Finally, Static Analyzer builds a call graph for each entry point, and searches the call graph to uncover paths from the entry point to intent sink methods, content provider access and security-sensitive APIs. Entry points we focus on are the callbacks defined by Android to be implemented by developers to handle notifications when some component life-cycle changes, which are used for component interaction. Such entry points are standard and can be determined by the type of components contained within the app. Specially, there are in total four types of components, and each type has some predefined interfaces to other components or system. For example, *onReceive* method in *BroadcastReceiver* class or its subclasses is an entry point. The paths are attack surface which can be accessed by other components to achieve some unauthorized behaviors, such as privilege escalation.

All information Static Analyzer uncovers is encoded and saved in the form of logic predicates.

B. Logical Checker

Formal methods help us confirm whether a certain security rule holds in Android system or detect whether a potential security threat exists in the system. We first use predicate logic to model Android intent-based ICC mechanism and permission-based protection mechanism, which is the kernel of Logical Checker. And this model can be instantiated when applying the extracted configuration as input of it, where the model instance formally describes the security state of Android phone. Additionally, Logical Checker also generates and maintains an overall ICC graph for the system over the model instance by logically examining all possible intent-based component interaction. The ICC graph consists of edges and nodes: each node indicates the call graph for a component, each edge is an intent or a broadcast to connect two components who can communicate with each other through this intent or broadcast.

Inputting formal specifications of security rules and threats into Logical Checker, we can drive Logical Checker to make logical inference to check the security properties and detect the threats over the model instance and ICC graph. Logical Checker will report all violations of security rules and all scenarios of threats.

Logical Checker enables our approach scalable, and users or analysts can specify and check security rules or threats which they are interested in. Security rules and threats are described in terms of predicate logic. Next, we will present some samples embedded in Logical Checker.

We first define some symbols:

- *APPS*, the set of apps
- *COMPS*, the set of components
- *PERMS*, the set of permissions
- *ACTIVITIES*, the set of activities
- *SERVICES*, the set of services
- *INTENTS*, the set of all extracted intents
- *APIS*, the set of privileged APIs we select
- *CERTS*, the set of certificates
- *SUIDS*, the set of sharedUserId values

1) Security Rules: Android document provides many *Notes* and *Cautions* to guide developers how to build a secure app. If developers follow the guidance, they can effectively reduce apps' attack surface. Unfortunately, lots of developers ignore these *Notes* and *Cautions*, which makes their apps vulnerable. We will give some samples to illustrate how to derive security rules from Android document in terms of predicate logic.

Rule 1. There is a *Note* in Android document as follows [5]: *"Note: In order to receive implicit intents, you must include the CATEGORY_DEFAULT category in the intent filter. The methods startActivity() and startActivityForResult() treat all intents as if they declared the CATEGORY_DEFAULT category. If you do not declare it in your intent filter, no implicit intents will resolve to your activity."* From this *Note*, we can derive that if an activity's intent filters do not specify CATEGORY_DEFAULT category, this activity can only receive explicit intents. Since explicit intents are mostly used in intra-application communication, we assume that this activity should be private, and its exported flag should be false. This rule can be encoded with logic formula as follows:

$$\forall activity \in ACTIVITIES. \quad (\neg launcher(activity) \wedge \neg contain\_category(activity, CATEGORY\_DEFAULT)) \rightarrow public(activity, false)$$

Rule 2. Android document notes that *"Caution: To ensure your app is secure, always use an explicit intent when starting a Service and do not declare intent filters for your services"* [5]. Since services run in the background, using an implicit intent to start a service is security hazard because you cannot be certain what service will respond to the intent. And in this case, a malicious service may be started instead of the intended service, and performs some dangerous actions, such as stealing private data. We use following formula to model whether an Android phone satisfies this rule:

$$\forall service \in SERVICES, \forall intent \in INTENTS. (parameter(intent, startService) \rightarrow explicit(intent)) \wedge \neg has\_filter(service)$$

Rule 3. Android API document advises that *"You'll typically use an explicit intent to start a component in your own app"* [5]. Based on this advise, we assume that if an intent can be resolved to a component within the same app, this intent is very possibly intended for intra-application communication and should be explicit type. Using explicit intents is an effective measure to reduce the risk of data leakage caused by implicit intents. This rule can be specified with following formula:

$$\forall intent \in INTENTS, \exists \{comp1, comp2\} \in COMPS, \exists app \in APPS. (source(intent, comp1) \wedge resolve\_to(intent, comp2) \wedge component(comp1, app) \wedge component(comp2, app)) \rightarrow explicit(intent)$$

where, *source(intent, comp1)* means that *intent* comes from *comp1*, *resolve_to(intent, comp2)* states that *intent* can be resolved to *comp2*.

2) Security Threats: Android faces various security threats, including data leakage, unauthorized intent receipt, intent spoofing, and privilege escalation. We will present some common threats and formally describe them with logic predicates, and analysts can use our system to examine more threats.

Threat 1. An implicit intent can have key-value pairs that carry additional information required to accomplish the requested action in its *extra* filed. Sometimes, extra field may contain some very secret information, such as username, password, even online bank account information. However, a malicious app can intercept any implicit intent without permission protection and access the privacy information stored in the extra field. Therefor, implicit intents with extra information may lead to data leakage, and we specify this threat scenario with following formula:

$$\exists intent \in INTENTS. \ implicit(intent) \wedge has\_extra(intent)$$

Threat 2. Explicit privilege escalation in Android is a kind of confused deputy attack[23]. In this attack, an app called *requester* lacks a certain permission which another app called *deputy* has. The requester can invoke the deputy's public interface directly or indirectly, causing the deputy to call a privileged API method. In this case, the API call will success since the deputy has the required permission, and the requester is successful in executing a privileged action, which also violates principle of least privilege. We formally describe this scenario by following logic predicate:

$$\exists \{comp1, comp2\} \in COMPS, \exists app1 \in APPS, \exists api \in APIS, \exists permission \in PERMS. \quad invoke(comp1, comp2) \wedge path(comp2, api) \wedge api\_permission(api, permission) \wedge component(comp1, app1) \wedge \neg has\_permission(app1, permission)$$

Informally, *invoke(comp1, comp2)* means that *comp1* can communicate with *comp2* based on intents directly or indirectly, e.g., an activity starts anther activity. Predicate *path(comp2, api)* says that there is a call path from *comp2's* entry point to a sensitive *api*. Predicate *api_permission(api, permission)* implies *permission* is required to invoke this privileged *api*. This formula describes not only one-step privilege escalation, but also multi-step escalation called chained capability leaks [20].

Threat 3. Contrast to explicit privilege escalation, there is another type of privilege escalation called implicit privilege escalation, which arises from the abuse of an optional attribute in the manifest file, i.e. *"sharedUserId"*. Multiple apps singed by the same developer certificate will share the union of all the permissions granted to each app if they have the same *"sharedUserId"* value. If one of these apps uses a certain permission that is not granted to itself but to other apps, an implicit privilege escalation happens. We specify this threat with following formula:

$$\exists \{comp1, comp2\} \in COMPS, \exists \{app1, app2\} \in APPS, \exists id \in SUIDS, \exists cert \in CERTS, \exists api \in APIS, \exists permission \in PERMS. \ component(comp1, app1) \wedge signed(app1, cert) \wedge component(comp2, app2) \wedge signed(app2, cert) \wedge$$

$userid(app1, id) \wedge userid(app2, id) \wedge path(comp2, api) \wedge$
$api\_permission(api, permission) \wedge$
$has\_permission(app1, permission) \wedge$
$\neg has\_permission(app2, permission)$

Threat 4. Similar to explicit privilege escalation, an app with a ceratin permission can access a content provider on behalf of another app who does not have this permission, which can be called content leakage. Since content leakage in not only an explicit privilege escalation but also a kind of data leakage, we specify this threat separately.

$\exists\{comp1, comp2\} \in COMPS, \exists app1 \in APPS, \exists api \in$
$APIS, \exists permission \in PERMS.invoke(comp1, comp2) \wedge$
$path(comp2, api) \wedge api\_permission(api, permission) \wedge$
$content\_access(api) \wedge component(comp1, app1) \wedge$
$\neg has\_permission(app1, permission)$

Predicate *content_access(api)* means that *api* is a method to access content provider, e.g., query, insert, etc. Informally, this formula states that if *comp1* can invokes *comp2's* public interfaces directly or indirectly, and *comp2* has a path from a entry point to a content-access *api*, content leakage takes place.

IV. Implementation

We have developed a SADroid prototype with a mixture of Java code, shell scripts and Prolog code. Two primary components constitute SADroid: Static Analyzer and Logical Checker. SADroid works on Android phone image, i.e., Android ROM. To analyze vulnerabilities existing in an Android phone, SADroid first employs `ext4-unpacker` tool [2] to extract all of the pre-installed apps and framework files from phone image. Since some manufactures may odex their pre-installed apps and remove the classes.dex from apk files, e.g., Samsung, we write a shell script file to convert *.odex* files to *.dex* files with the help of `smali/baksmali` tool [7].

A. Static Analyzer

For each pre-installed app, Static Analyzer adopts `apktool` [1] to decompress the related apk file to extract its manifest file (AndroidManifest.xml). Parsing the textual manifest file, SADroid obtains static configuration knowledge for each app and stores the results as Prolog facts, e.g., `component/2, public/2, has_permission/2, signed/2`.

The core of Static Analyzer bases on WALA [8], which is an open-source libraries for Java code analysis providing a framework to parse a set of Java classes and generate a call graph over all reachable methods. For each *.dex* file standing alone or *.apk* file containing a classes.dex file, Static Analyzer first translates dex instructions to WALA's intermediate representation based on `dexlib` library [4] which is used in smali tool, and further analysis is performed on the intermediate representation. For each entry point, we generate a call graph, and extract the information mentioned in section III-A. Finally, all of the results are encoded to Prolog facts.

B. Logical Checker

The input of Logical Checker consists of two parts: Prolog facts and deriving rules. Prolog facts are collected by Static Analyzer to generate a model instance, and security rules and security threats mentioned in section III-B are encoded to Prolog deriving rules. Each rule contains two parts:(1)an evaluation part that evaluates the pre-conditions (2)a logging part that logs the evaluation if the evaluation succeeds. For each component, Logical Checker checks whether it can communicate with any other component with an intent, if so, there is a directed edge from the source component to the destination component. By logging all of the edges,Logical Checker generates a general *ICC* graph for all of the pre-installed apps, which describes the communication relationship between components based on intent mechanism. Based on the ICC graph, Logical Checker detects possible vulnerabilities by checking the security rules and security threats, and logs all the violation of each security rule and all the scenarios of each threats. For example, the deriving rule for threat 1 can be programmed as follows:

```
intent_leakage(Intent) :-
    % evaluation part
    implicit_intent_with_extra(Intent),
    % logging part
    unique_assert(log_intent_leakage(Intent))
```

The predicate implicit_intent_with_extra(Intent) checks whether an intent contains *extra* field. log_intent_leakage is a dynamic term, if implicit_intent_with_extra(Intent) succeeds, a log_intent_leakage term will be asserted and this threat instance is logged.

The following Prolog rule detects all instances of threat 1, and checks for other Prolog rules are nearly the same:

```
find_all_intent_leakage :-
    intent(AppName, Value),
    Intent = intent(AppName, Value),
    intent_leakage(Intent),
    fail.
```

To speed up the execution and avoid infinite loops, we use tabling evaluation provided by Prolog implementation XSB [21]. All the predicates used to generate the ICC graph and the predicates to check the security rules and threats are declared as tabled predicates.

TABLE I. Four Analyzed Android Phones

Vendor	Distribution	Android Version	# of Apps
Google	Nexus 4	4.3	87
Samsung	Galaxy S3	4.2	196
CyanogenMod	CM11 for Nexus 4	4.4	81
	CM11 for Galaxy S3	4.4	83

V. Evaluation

In this section, we present the evaluation results of applying SADroid to some real Android smartphones. In order to assess misconfiguration vulnerabilities posed in the wild, we selected four representative smartphone images from three vendors, including some flagship phones billed as having significant additional bundled functionality and some mainstream third-party images optimized by CyanogenMod (CM) [3]. Table I shows the phone images and their versions we analyzed using SADroid. For each phone image, we apply SADroid to check the security rules and threats mentioned in Section III-B to uncover misconfiguration vulnerabilities and demonstrate its effectiveness.

A. Results

SADroid reports all violations of the security rules and possible threats mentioned in Section III-B. Table II presents the total number of different types of components on the four images, e.g., data '809/70' means there are 809 activities in official Nexus4 and all of these activities come from 70 apps. In order to make the statistics more clearly, we list activity alias and dynamic receiver separately.

TABLE II. STATISTICS OF TOTAL COMPONENTS

component	Nexus4	GalaxyS3	CM-Nexus4	CM-GalaxyS3
activity	809/70	2170/146	510/63	513/65
activity-alias	79/15	59/5	51/13	51/13
service	284/64	381/143	161/55	161/55
provider	85/45	125/62	57/31	57/31
receiver	229/52	489/127	146/43	149/45
dynamic-receiver	268/42	777/116	123/34	123/34

For simplicity, we name violation of security rule 1 as improper-exported-activity, violation of security rule 2 as service-with-filter and implicit-intent-for-service, violation of security rule 3 as local-implicit-intent, threat 1 as intent-leakage, threat 2 as explicit-privilege-escalation, threat 3 as implicit-privilege-escalation, threat 4 as content-leakage.

In Table III, we list the results for improper-exported-activity(IEA), service-with-filter(SWF), implicit-intent-for-service(IIFS), local-implicit-intent(LII) and intent-leakage(IL). The data in every cell means *number of vulnerabilities/number of apps with that kind of vulnerabilities(percentage of vulnerable objects out of the total number of the same type objects/the frequency of vulnerable apps out of the total apps)*, e.g., the data 122/36(0.15/0.42) for improper-exported-activity implies that exported flag values of 122 activities from 36 apps are improperly specified, and the vulnerable activities account for 15% out of the total activities, and 42% of apps contain at least one vulnerable activity.

Explicit privilege escalation is a multi-step attack. SADroid searches the generated ICC graph to find out all the attack paths leading to explicit privilege escalation. We call the component who starts an explicit privilege escalation *source*, and the component whose permission is obtained by the source is called *goal*. We compute the distribution of source components and goal components, and additionally we examine the number of goal components without permission protection. The results are listed in Table IV. Although the number of attack paths is huge, the fraction of involved components out of all the components is limited. Source components account for about 20%, and the frequency of goal components is much less than 10%. Furthermore, more than half of goal components are unprotected by any permission.

TABLE III. STATISTICS OF SOME VULNERABILITIES

vulnerability	Nexus4	GalaxyS3	CM-Nexus4	CM-GalaxyS3
IEA	122/36 (0.15/0.41)	224/52 (0.10/0.27)	89/29 (0.17/0.36)	91/31 (0.18/0.37)
SWF	122/45 (0.43/0.52)	195/99 (0.51/0.51)	69/28 (0.43/0.35)	69/28 (0.43/0.34)
IIFS	164/33 (0.42/0.38)	256/80 (0.33/0.41)	71/22 (0.34/0.27)	71/22 (0.34/0.27)
LII	621/38 (0.65/0.44)	1016/73 (0.51/0.37)	252/39 (0.54/0.48)	252/39 (0.54/0.47)
IL	199/29 (0.21/0.33)	589/57 (0.30/0.29)	111/29 (0.24/0.36)	111/29 (0.24/0.35)

TABLE IV. STATISTICS OF EXPLICIT PRIVILEGE ESCALATION

	Nexus4	GalaxyS3	CM-Nexus4	CM-GalaxyS3
# of paths	37269	547116	26930	26930
# of sources	374	938	228	228
# of goals	85	274	76	76
% of sources	21.3%	23.4%	21.8%	21.8%
% of goals	4.8%	6.8%	7.3%	7.3%
% of unprotected goals	71.8%	59.9%	65.8%	65.8%

Implicit privilege escalation arises from the abuse of an optional attribute in the manifest file, i.e. *"sharedUserId"*, which may lead to collusion attack. Fortunately, we do not find any implicit privilege escalation in the four phone images.

Content leakage can be considered as a special class of explicit privilege escalation, which only focuses on the permissions used to access content providers. Similar to explicit privilege escalation, we examine the paths of content leakage in Table V. Here, we call the components accessing content providers *goal components*. Only a few components can be exploited to leak content data, but most of these components are not protected by any permission, which is the root cause leading to content leakage.

TABLE V. STATISTICS OF CONTENT LEAKAGE

	Nexus4	GalaxyS3	CM-Nexus4	CM-GalaxyS3
# of paths	6843	201655	7441	7441
# of sources	377	927	240	240
# of goals	15	64	16	16
% of sources	21.4%	23.2%	22.9%	22.8%
% of goals	0.8%	1.6%	1.5%	1.5%
% of unprotected goals	73.3%	84.3%	75%	75%

B. Discussion on Results

Developers are expected to build secure apps or secure system following security tips in Android document. However, it is not the fact. Our results reveal that many developers do not understand Android permission mechanism very well, especially the usage of permission mechanism provided for inter-component-communication, which leads developers to mis-configuring their apps and results in some vulnerabilities. Although Android permission model is not perfect to provide security protection, developers' misconfiguration make Android's security much worse. Unfortunately, it is common for a developer to mis-configure his app.

Although Android document recommends that developers should neither declare intent filters for a service nor use an implicit intent to start a service, many developers do not follow these security recommendations. About half of developers add intent filters to a service that listen for intents with their action, which has the undesirable side effect of making the component public. It is reasonable to assume that they are not aware that they should make the service private. Using implicit intents to start a service violates Android document's recommendation, since the system determine which service to start when multiple services match the implicit intent, and users could not realize which service is running. However, More that 30% of developers use implicit intents to start services.

Our results for security rules also show that most developers prefer an implicit intent for component communication

even if the intent is intended for inter-application communication, in which case an explicit intent should be preferred.

From Table IV, we can conclude that only a few components are exploited to escalate privileges, however these components are distributed in more than half of apps and most of these components are not protected with any permissions, which is a universal phenomenon. Privilege escalation is a logical vulnerability, and developers can not foresee all the ways of communication, so the privilege escalation vulnerability is difficult to avoid. But developers can reduce this kind of vulnerabilities through declaring proper permissions for related components.

Content providers are protected best, however, some content leakages still exist. We find these leakages are caused by the app components who access content providers directly, and these components are not protected by proper permissions. By enforcing proper permissions on these components, developers can effectively reduce this kind of data leakage.

By examining the results, we conclude that most of vulnerabilities result from components lack of permission protection. There are some reasons why developers often ignore to declare permission protection:(1) most of development tools use none permission protection for components as default policy, where some developers lack of security knowledge do not change the default policy, (2) some developers use intent filters instead of permissions to prevent other components from communicating with their components, however, this measure is incorrect. Intent filters are designed to address implicit intents, and not for security protection. Without permission protection, a malicious app can always access your component by declaring an intent filter with all of the listed actions, categories and data.

Our experiment shows that smartphones with more pre-installed apps tend to be more vulnerable, and CM-Nexus4 and CM-GalaxyS3 nearly have the same evaluation results, because the two images have the same pre-installed apps except a small one.

VI. Discussion

SADroid is a well scalable tool to find violation of security rules recommended by Android document and detect security threats based on known threat patterns. Our tool focuses on vulnerabilities caused by security misconfiguration. We consider 28 representative privileged permissions which are commonly used for developers. To involve more permissions, new permissions and related APIs should be added to Static Analyzer. There are many recommendations for developers to build more secure apps, and we just extract and formally specify some typical suggestions. Users and analysts can specify more security rules and input them into Logical Checker to detect vulnerabilities which they are interested in. Our tool only detects local-implicit-intent, intent-leakage, explicit-privilege-escalation, implicit-privilege-escalation and content-leakage, but more threat specifications can be added. SADroid serves as a prototype to demonstrate the effectiveness of our analysis model, therefor it only includes the representative rules and well-known threats.

SADroid not only detects vulnerabilities, but also assesses deveolpers' misconfiguration in Android system. The result is worrisome. Our result shows that it is common for a developer to misconfigure his app, mainly including lack of protection permissions, misuse of implicit intents, and abuse of intent filters.

The accuracy of SADroid depends on the accuracy of static analysis on each app, especially the values of intents which are difficult to get precise values. So SADroid can benefit from more accurate static analysis method, which consumes much more time.

VII. Related Work

A lot of research works focus on Android security. From the perspective of attacks and malwares, Vidas et al. [29] provide a taxonomy of attacks to the platform. Davi et al. [11] show that it is possible to mount privilege escalation attacks on the well-established Android platform, and implement a privilege escalation attack with return-oriented programming without returns. Felt et al. [18] demonstrate that permission re-delegation attack can help a less privilege app to perform a privilege task through another third-party app with higher privilege. Zhou et al. [30] present a systematic characterization of existing Android malwares consisting of 1260 samples in 49 different families.

There are a number of works focusing on analyzing and improving the Android permission model, including privacy and capacity leak detection. Kirin [15], [16] develops a logic-based tool to determine whether the permissions declared by an application meet a certain global policy invariants to ensure only policy compliant applications will be installed. Shin et al. [27] present a formal model of Android permission scheme, which enables users to express authorization and permission-protected interactions among app components, to check whether the system meets given security requirements. However, these logic-based works only consider the policy defined in manifest file, which is not enough to construct a complete model. SADroid also uncovers the dynamic security policy defined in app's Dalvik bytecode through static analysis technique, and build a more precise model to detect vulnerabilities, which improves the accuracy of detection.

ScanDroid [19] is the first static tool for Android, which checks data flow consistency. TaintDroid [13] is an efficient, system-wide dynamic analysis tool using taint tracking to simultaneously track multiple sources of sensitive data. Enck et al. [14] design and implement a Dalvik decompiler, ded, to recover an application's Java source code. ComDroid [10] is a tool that detects application communication vulnerabilities through examining application interaction and identifying security risks in application components. Felt et al. [17] develop a static tool Stowaway to detect over-privilege in Android apps. PScout [9] builds a permission mapper through static analysis based on Soot. Han et al. [22] investigate applications that run on both Android and iOS and examine the difference in the usage of their security sensitive APIs. Woodpecker [20] statically analyzes eight popular Android smartphone to identify capability leak caused by confused deputy attacks. CHEX [24] is a static analysis method to automatically vet Android apps for component hijacking vulnerabilities by modeling these vulnerabilities form a data-flow analysis perspective.

Comparing with previous research, SADroid builds a abstract model to describe Android based-intent communication and permission mechanism, which can be initialized as different model instance by inputting different configuration collected by Static Analyzer on various Android phones. Besides, SADroid constructs a more precise logical model instance with more detailed security configuration including static configuration in manifest files and dynamic configuration embedded in Dalvik bytecode. SADroid can accept vulnerability specifications encoded with logic predicates as input to detect vulnerabilities, which enables SADroid well scalable. Users or analysts could use SADroid to detect vulnerabilities which they are interested in. We have employed SADroid to detect seven vulnerabilities.

Instead of analyzing one app, SADroid constructs an overall ICC graph for the system with logical inference. The ICC graph consists of edges and nodes: each node indicates the call graph for a component, each edge is an intent or a broadcast to connect two components who can communicate with each other through this intent or broadcast. Over this ICC graph, more kinds of vulnerabilities can be detected, especially the vulnerabilities related to component interaction. Furthermore, detecting vulnerabilities on ICC graph can effectively improve the accuracy of detection.

VIII. Conclusion

In this paper, we propose a logic-based approach to analyze and detect the misconfiguration vulnerabilities existing in Android smartphones and implement a prototype called SADroid. SADroid adopts static analysis to extract security configuration from manifest file and app code. SADroid then specifies some security rules that developers should follow to build apps and some threats that a phone may face. Based on the configuration and specifications of security rules and threats, SADroid reasons about the vulnerabilities. The results are worrisome: there are programming errors in many apps, permissions are always misused and developers do not understand the permission model very well, which may lead to various vulnerabilities and attacks.

Acknowledgment

This work was supported in part by NSFC grant No.60970028, NSFC grant No.61100227 and National 863 Program of China under Grant 2011AA01A203.

References

[1] "android-apktool," https://code.google.com/p/android-apktool/.

[2] "Android ics jb ext4 imagefile unpacker," http://sourceforge.net/projects/androidicsjbext/.

[3] "Cyanogenmode-android community operating system," http://www.cyanogenmod.org/.

[4] "dexlib," https://code.google.com/p/smali/source/browse/dexlib/src/main/java/org/jf/dexlib/.

[5] "Intents and intent filters," http://developer.android.com/guide/components/intents-filters.html.

[6] "Manifest.permission," http://developer.android.com/reference/android/Manifest.permission.html.

[7] "Smali and baksmali," https://code.google.com/p/smali/.

[8] "T.j. watson libraries for analysis (wala)," http://wala.sourceforge.net.

[9] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228.

[10] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

[11] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," in *Information Security*. Springer, 2011, pp. 346–360.

[12] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, "Quire: Lightweight provenance for smart phone operating systems." in *USENIX Security Symposium*, 2011.

[13] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones." in *OSDI*, vol. 10, 2010, pp. 1–6.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX Security Symposium*, 2011.

[15] W. Enck, M. Ongtang, and P. McDaniel, "Mitigating android software misuse before it happens," 2008.

[16] W. Enck, M. Ongtang, P. D. McDaniel *et al.*, "Understanding android security." *IEEE Security & Privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[17] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

[18] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses." in *USENIX Security Symposium*, 2011.

[19] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[20] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[21] T. Group, "The xsb programming system," http://xsb.sourceforge.net/.

[22] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng, "Comparing mobile privacy protection through cross-platform applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA*, 2013.

[23] N. Hardy, "The confused deputy:(or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, no. 4, pp. 36–38, 1988.

[24] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 229–240.

[25] R. A. Rimando, "Development and analysis of security policies in security enhanced android," Ph.D. dissertation, Monterey, California. Naval Postgraduate School, 2012.

[26] P. Schulz, "Android security-common attack vectors," *Rheinische Friedrich-Wilhelms-Universitat Bonn, Germany, Tech. Rep*, 2012.

[27] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A formal model to analyze the permission authorization and enforcement in the android framework," in *Social Computing (SocialCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 944–951.

[28] S. Smalley and R. Craig, "Security enhanced (se) android: Bringing flexible mac to android," in *20th Annual Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[29] T. Vidas, D. Votipka, and N. Christin, "All your droid are belong to us: A survey of current android attacks." in *WOOT*, 2011, pp. 81–90.

[30] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (S&P), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.