A taxonomy of privilege escalation attacks in Android applications

Mohammed Rangwala

Department of Computer and Information Science,
Indiana University Purdue University Indianapolis,
Indianapolis, IN 46202, USA
E-mail: mmrangwa@iupui.edu

Ping Zhang

Department of Computer Science and Engineering,
Henan Institute of Engineering,
Zhengzhou, HN 451191, China
E-mail: zpings@sina.com

Xukai Zou*

Department of Computer and Information Science,
Indiana University Purdue University Indianapolis,
Indianapolis, IN 46202, USA
E-mail: xkzou@cs.iupui.edu
*Corresponding author

Feng Li

Department of Computer and Information Technology,
Indiana University Purdue University Indianapolis,
Indianapolis, IN 46202, USA
E-mail: fengli@iupui.edu

Abstract: Google's Android is one of the most popular mobile operating system platforms today, being deployed on a wide range of mobile devices from various manufacturers. It is termed as a privilege- separated operating system which implements some novel security mechanisms. Recent research and security attacks on the platform, however, have shown that the security model of Android is flawed and is vulnerable to transitive usage of privileges among applications. Privilege escalation attacks have been shown to be malicious and with the wide spread and growing use of the system, the platform for these attacks is also growing wider. This provides a motivation to design and implement better security frameworks and mechanisms to mitigate these attacks. This paper discusses; 1) the security features currently provided by the Android platform; 2) a definition, few working examples and classifications of privilege escalation attacks in Android applications; 3) a classification and comparison of different frameworks and security extensions proposed in recent research.

Keywords: Android; privilege escalation; smartphone security.

Reference to this paper should be made as follows: Rangwala, M., Zhang, P., Zou, X. and Li, F. (2014) 'A taxonomy of privilege escalation attacks in Android applications', *Int. J. Security and Networks*, Vol. 9, No. 1, pp.40–55.

Biographical notes: Mohammed Rangwala is a Master's student with the Department of Computer and Information Science at Indiana University-Purdue University Indianapolis. He obtained his BE in Computer Engineering from University of Mumbai, India in 2012. His research interests include network security, android security and digital provenance. He received the University Fellowship in 2012.

Ping Zhang is a Faculty Member with Department of Computer Science and Engineering, Henan Institute of Engineering, China. Her research area covers networking, operating systems, network management and protocol design.

Copyright © 2014 Inderscience Enterprises Ltd.

Xukai Zou is a Faculty Member with the Department of Computer and Information Science at Indiana University-Purdue University Indianapolis. He completed his PhD in Computer Science from University of Nebraska-Lincoln. His current research focus is applied cryptography, network security, biometrics, authentication and communication networks. His research has been supported by NSF, the Department of Veterans Affairs and Industry such as Cisco and Northrop Grumman.

Feng Li received his PhD in Computer Science from Florida Atlantic University in August 2009. His PhD advisor is Dr. Jie Wu. He joined the Department of Computer, Information, and Leadership Technology at Indiana University-Purdue University Indianapolis as an Assistant Professor in August 2009. His research interests include the areas of wireless networks and mobile computing, security, and trust management. He has published more than 30 papers in conferences and journals.

1 Introduction

Google's Android is a modern and popular operating system platform for smartphones and tablet devices. Since the first release in 2008, its popularity and sales of devices hosting the system have increased at a very fast rate. A report by Strategy Analytics in January 2013 states that smartphone sales grew 38% in the last quarter of 2012 to reach 217 million units worldwide, and over 700 million units for the entire year (Strategy Anayltics, 2013; Yahoo! Finance, 2013). Of this number, 68.4% devices operate the Android platform. In October 2012, Google said that there were about 700,000 applications available for downloading onto Android devices matching the number of applications on Apple's App Store for iOS devices (Bloomberg Businessweek, 2012). There are a large number of end user devices and a large number of applications being used on them. Handsets today have become full-fledged computing platforms supporting complete operating systems and complex applications. However, this brings new security challenges. A recent mobile security report states that:

> "The sheer number of mobile applications at a time when the technology in mobile security is still in its infancy presents complex, multifaceted, and unprecedented security challenges to enterprises while putting individual privacy at high risk." (Li and Clark, 2013)

The current model of the Android Application Market allows developers to upload arbitrary applications at a minimal fee.[1] This creates a large attack surface for malicious applications to be published on the market and installed on end user devices. According to the Kaspersky Security Bulletin 2012, 99% of the newly discovered mobile malicious programs target the Android platform (Kaspersky, 2012). Soundcomber (Schlegel et al., 2011) is a trojan that uses innocuous permissions and context-aware tone- and speech-analysis to extract small amounts of targeted private data. The trojan GGTracker (The Lookout Blog, 2012) sends SMS to a premium-rate number and can steal private information from the device. Apple screens applications posted to its App Market, which protects users from malicious applications. Nevertheless, applications could still have vulnerabilities that may be exploited. Google, on the other hand, does not screen applications being published to its market, making it possible for malicious applications to easily reach users. It occasionally takes down applications that are found to contain malware.

Android implements a permissions and sandboxing mechanism through its middleware layer to control access to resources and mediate inter-application communication. It is a privilege separated system, with each application having its own distinct system identity. This model is not able to prevent transitive usage of permissions that can be leveraged to launch privilege escalation attacks (Davi et al., 2010). It is possible for a malicious application to gain capabilities leaked from benign applications making its capabilities more than it is permitted to have. It is possible to prevent this by having strong checking of permissions; however, since developers are not security-minded, this is not used in practice. As a result, there is need for a more secure framework to be implemented to prevent these attacks.

The main contributions of this paper are:

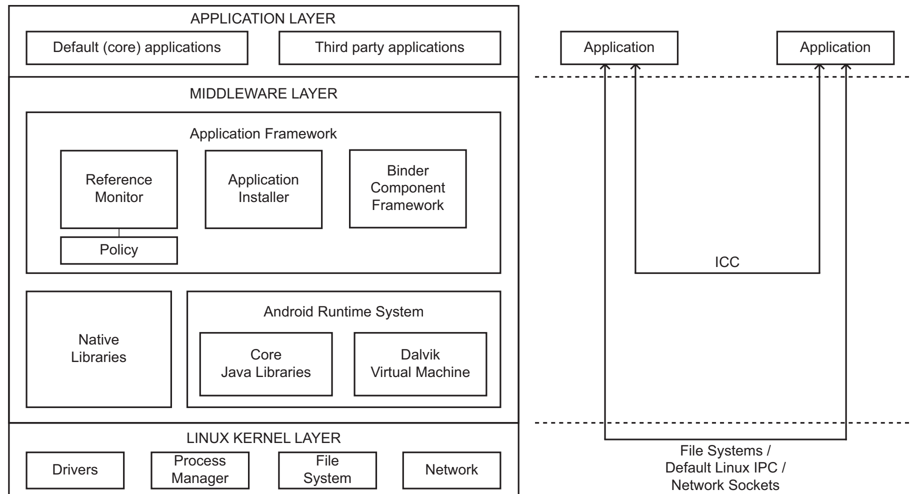
- discusses a classification of privilege escalation attacks depending on the channels and mechanisms used, explained by Figure 4 in Section 4
- provides comparisons of different analysis tools and security extensions, explained by Tables 1–3 in Section 6
- this paper not only provides tutorial knowledge for the inquisitive reader, but also provides guidance to security researchers and application developers.

The rest of this paper is organised as follows: Section 2 discusses background information about Android along with the security mechanisms currently implemented. The privilege escalation attack is formally defined and explained in Section 3. A classification of the application level privilege escalation attacks is provided in Section 4. Section 5 discusses defense goals and models. Section 6 discusses different analysis tools and extensions proposed in recent research, and their comparison. Security guidelines that must be followed along with some recoomendations are discussed in Sections 7 and 8 concludes the paper.

2 The Android platform

Android is an open source mobile platform built on a Linux kernel. It implements certain security mechanisms

Figure 1 Android architecture diagram



that impose restrictions on how applications are allowed to behave. This section reviews some of these concepts.

2.1 The architecture

The Android platform, as depicted in Figure 1 is built on a Linux kernel which provides the basic functionalities of drivers, file system support, networking and process scheduling. On top of the kernel are the native libraries, written in C/C++ (http://elinux.org/ Android_Architecture). The runtime environment contains an Android-specific optimised Java Virtual Machine called the Dalvik Virtual Machine and the core Java libraries. Every application runs on its own instance of the virtual machine providing isolation. An application framework layer provides the applications with basic functionalities such as resource management, activity life cycle management, window management, etc. Android applications are written primarily in Java, but it is possible to include C/C++ libraries using the Java native interface (JNI). However, the security mechanisms of Java, e.g., bounds checking, are lost when such libraries are included. Applications run over the application framework layer. The reference monitor is a component of the application framework that mediates inter component communication (ICC) on the basis of a system policy (Enck et al., 2009). It provides a mandatory access control (MAC) enforcement of how applications access components. The application installer installs new applications and the Binder component framework provides a synchronous RPC mechanism for ICC within the same application as well as between applications. The right portion of the figure shows the different communication channels possible between applications at the different layers of the Android architecture.

2.2 Inter component communication

Applications make use of components in their logic that communicate with each other through the mechanism of Intents. Intent is a passive data structure that contains two items: action to be performed, and data to be operated upon. Applications can make use of four components and Figure 2 shows the standard interaction between components through the use of intents. Activity – provides a user interface and handles a single focused thing a user can perform. An application can contain many Activities, one for each screen presented to the user. The interface progression is a sequence of one Activity starting another, possibly expecting a return value (Enck et al., 2008). Activities can start other or return to other activities using intents as shown in Figure 2(a). Service – used to perform a long-running background operation even after an application loses focus. Service components are bound to activities that can start/stop them as seen in Figure 2(b). Content Provider – used to manage a structured set of data, such as, a local database. Content providers provide interfaces that support SQL-like queries, e.g., SELECT, INSERT, UPDATE, through which components of other applications can access the data. They are queried by activities through intents as seen in Figure 2(c). Broadcast Receiver – used to receive broadcasted intents from multiple components. It acts as an asynchronous mailbox for directed broadcasts of system and application event messages and sometimes invokes a Service or Activity to handle the task. Broadcast receivers gather broadcasted intents from multiple components as seen in Figure 2(d).

2.3 Security mechanisms

Android makes use of the following security mechanisms (Chan et al., 2011; Davi et al., 2010):

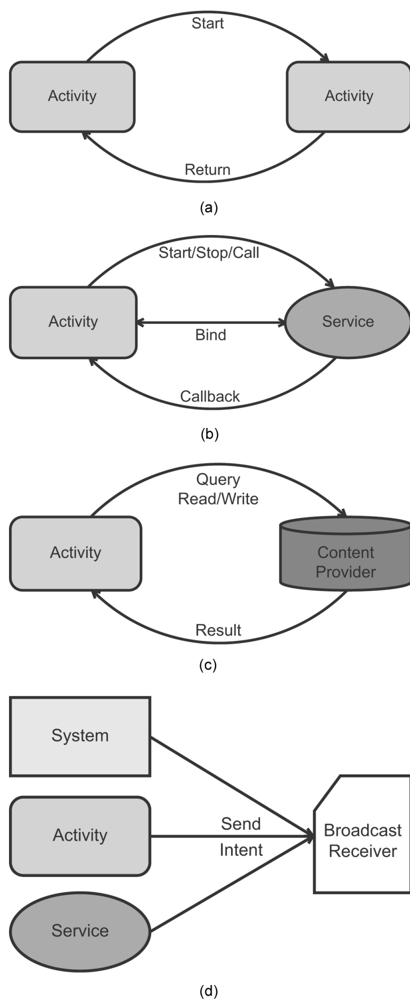
Discretionary access control. Each running process is assigned a UserID and each file (i.e., object) has certain access rules which is inherited from Linux. For example in Linux, every file has read, write, execute privileges for a user, group of users and everyone. Similar mechanisms exist in Android as well.

Sandboxing. Every application is logically isolated from other applications and system resources. An application can only access the files owned by it or files of other applications that

explicitly give permissions for other applications to access them.

Component isolation. Application components may be specified as public or private. A private component is only accessible to components of the same application. This imposes restrictions by default. A public component can be accessed by other applications and needs to perform permission checking to restrict other applications from accessing it.

Figure 2 Inter component communication (ICC) of four components: (a) activity components ICC; (b) service components ICC; (c) content providers ICC and (d) broadcast receivers ICC



Source: Chan et al. (2012)

Permissions. Android provides finer-grained security features through the use of permissions mechanism. It enforces restrictions or specific permissions that a particular process can perform. It is provided by the middleware layer and the reference monitor enforces a MAC on ICC calls. Android has roughly 100 built-in permissions that control operations ranging from dialing the phone (CALL_PHONE), taking pictures (CAMERA), using the Internet (INTERNET), listening to key strokes (READ_INPUT_STATE), whereas, applications may also declare custom sets of permissions to restrict access to them (Shabtai et al., 2010). Permissions can be classified into four categories (Felt et al., 2011a; Chin et al., 2011):

- Normal permissions protect API calls that would annoy a user but not cause any harm (e.g., SET_WALLPAPER) and do not require user approval
- Dangerous permissions allow an application to perform potentially harmful operations. They are granted by the user on installation (e.g., RECORD_AUDIO)
- Signature permissions only granted if the requesting application is signed by the same developer that defined the permission
- SignatureOrSystem permissions are granted if the application meets the Signature requirement or if the application is installed in the system applications folder.

Permissions are declared in *AndroidManifest.xml*, which is a compulsory file for all applications and is used at installation time by the Android system for user's consent. It relies on the user to judge whether he or she permits the application to use all the permissions it requires or does not install the application. The current Android platform only performs a static checking of permissions, but does not take into account the context in which these permissions are used by applications (Schlegel et al., 2011). This leads to vulnerabilities in the system.

Application signing. Android uses certain cryptographic signatures to verify the origins of applications and establish trust relationships among them. For this, developers sign their code using a certificate with a private key held by them. This certificate is present in the installation APK file of an application and is validated at install time. It allows to enable signature-based permissions, or to allow applications from the same origin (i.e., signed by the same developer) to share the same UserID.

Ongtang et al. (2009) state that:

> "The Android system protects the phone from malicious applications, but provides severely limited infrastructure for applications to protect themselves."

They identify three missing policies not available to applications:

- *Permission assignment policy*: Applications have limited ability to control which other applications are granted permissions to access its interfaces.
- *Interface exposure policy*: The system provides limited ability for applications to control how their interfaces are used by other applications.
- *Interface use policy*: At run-time, applications have limited ability to select which other application's interfaces they use.

Shabtai et al. (2010) provide an assessment of the Android security framework. They state that the Android system in its normal state is secure since the owner (and the attacker) can not modify the kernel without hardware changes. The only way

to circumvent this is to exploit vulnerabilities in the kernel modules or core libraries giving the attacker root privileges. This is an indication of a privilege escalation at the kernel level, but this paper focusses on attacks at the application level. This classification will be further discussed in Section 4.

3 The privilege escalation attack

It has been shown in the works (Davi et al., 2010; Chan et al., 2011; Bugiel et al., 2012; Chan et al., 2012; Felt et al., 2011a) that the Android security mechanisms, though novel, are flawed and it is possible for transitive usage of privileges to take place between applications. This section formally defines the privilege escalation attack along with working examples.

3.1 Definition

Davi et al. (2010) first defined the privilege escalation attack as:

> "An application with less permissions (a non-privileged caller) is not restricted to access components of a more privileged application (a privileged callee)."

This definition is better explained through an example depicted in Figure 3, which illustrates a privilege escalation attack between components of different applications (Davi et al., 2010). Consider three applications *A*, *B* and *C*, each running in its own sandbox and each having two components. In application C, component C_{C1} is protected by permission *p1*, similarly, C_{C2} by *p2*. Application B is granted permission *p1*, hence C_{B1} and C_{B2} can access C_{C1}. However, B is not protected by any permission and its components are publicly accessible. A does not have any permissions, but its component C_{A1} can access C_{B1}. C_{A1} is not able to access C_{C1} directly since it does not have permission *p1*, however, it can do so via component C_{B1}. This is a Privilege Escalation Attack since the privileges of application A (non-privileged caller) are escalated to the privileges of application C (privileged callee) indirectly through B which indicates the transitive usage of privileges.

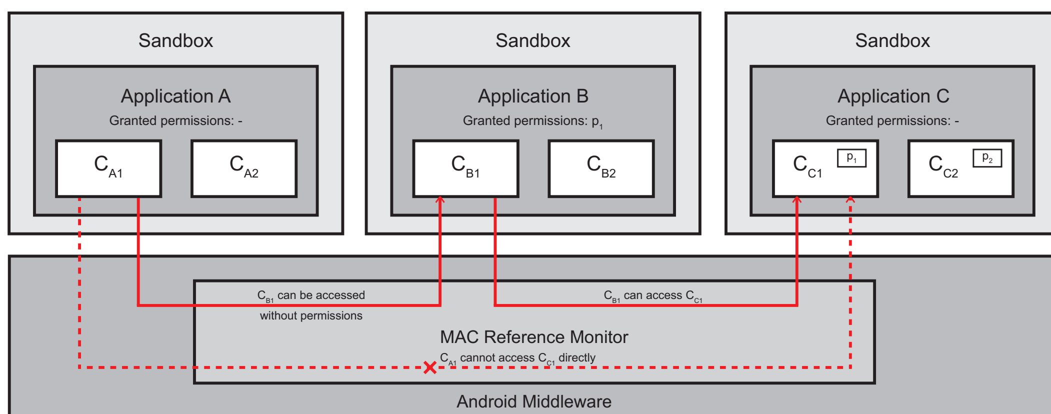
The attack described here is called the confused deputy privilege escalation attack and will be discussed in further detail in Section 4. In order to prevent this type of an attack, application B must ensure that another application calling it must have permission $p1$. This is done in Android through the use of permission checking mechanisms in the code and this task is delegated to application developers. Since developers are not security-minded and applications usually face a time-to-deliver deadline, this permission checking is generally ignored and making it an error-prone approach. Similar to other security vulnerabilities in software systems, these vulnerabilities are a result of undertrained developers, lack of application quality assurance mechanisms, usability issues of existing security mechanisms – issues that are difficult to avoid in reality (Lu et al., 2012).

3.2 Working examples

Davi et al. (2010) discussed a vulnerablility in the early version of Android that could be used to launch a privilege escalation attack. They stated that the core Android application Phone had an unprotected component that provided an interface to other applications that could be used to make unauthorised phone calls. This attack could be mapped to the example scenario of the previous section as: application Phone in place of B, the system interface in place of C. Any other application, e.g., Activity Manager could be in place of A. The Activity Manager could access the unprotected component of the Phone application sending a phone number to it, which would invoke the system interface to call that number, making an unauthorised phone call.

As a proof-of-concept attack in their work DroidChecker, (Chan et al., 2012) exploit a capability leak in the *Adobe Photoshop Express 1.3.1 (PEA)*. They created an attacker application that used the vulnerability in APE to retrieve e-mail addresses of contacts on the phone. The attacker application is first launched, which then launches the vulnerable component of APE. The user then gets tricked into selecting a contact from the contact list, which is then passed back to the attacker application. This indicates that the attacker application gains more privileges by exploiting the vulnerability of the APE application.

Figure 3 Component-based permission escalation attack (see online version for colours)



Source: Davi et al. (2010)

Hobarth and Mayrhofer (2011) authors discuss four example exploits that allow applications to temporarily escalate their privileges to root privilege. They are created using the Android Native Development Kit and depend on the Android system version being used, and can cause missing input sanitisation for applications receiving input from untrusted sources, overflow the limit of supported number of processes created by the same uid, remap the shared memory and restrict access to the ashmem (Android shared memory) shared memory. Though these exploits do not fall into the specific categories of attacks discussed in Section 4, they give an indication of vulnerabilities in the Android system and applications that can be exploited.

4 Classification of application-level privilege escalation attacks

Privilege escalation is possible at two levels in an Android system: the kernel level or the application level. Attacks at the kernel level exploit vulnerabilities in the Linux kernel or core system libraries (Shabtai et al., 2010). This paper focusses on privilege escalation attacks at the application level and this section gives a classification of them depending on the channel used to exploit vulnerabilities in applications. A classification of privilege escalation attacks is discussed by Bugiel et al. (2012) which is further extended here, as described by Figure 4. They classify them, based on the mechanism used, into two broad categories: confused deputy attacks and attacks by colluding applications.

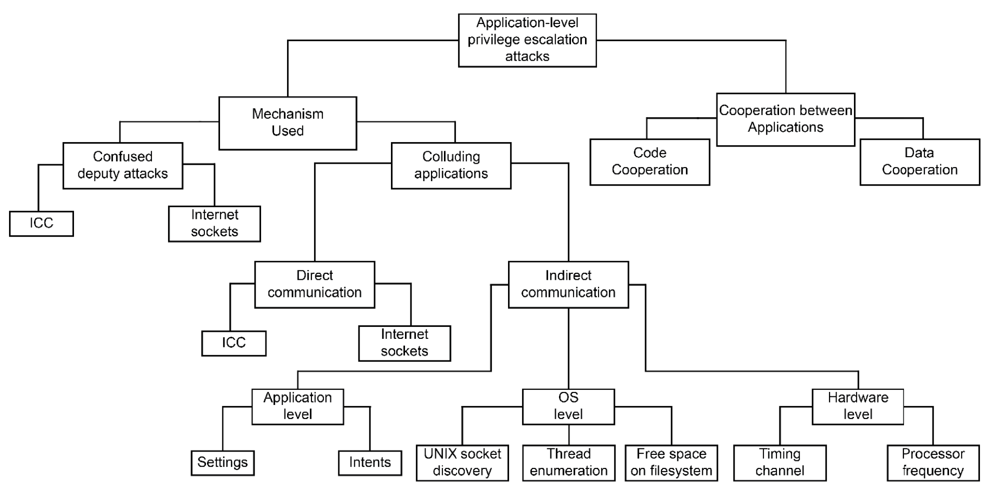
The first category, confused deputy attacks, involves an attacker application exploiting vulnerabilities of a target benign application (called a deputy) (Hardy, 1988) to perform some unauthorised operation. This is also referred to as permission re-delegation (Felt et al., 2011a). Permission re-delegation can occur in three ways:

- if an application accidentally exposes some internal functionality to other applications
- a confused deputy may intentionally expose some functionality but an attacker may use it for malicious purposes
- a developer may expose functionality with the goal of attenuating authority, but may implement the attenuation policy incorrectly or inconsistently.

Recent research shows that there are vulnerabilities in both the default core Android applications as well as third party developed applications that can be exploited (Bugiel et al., 2012). Relying on developers to prevent this is not a good practice, since developers are not security experts and they do not have independent motivation because a confused deputy attack does not affect the deputy itself. This attack can be further categorised, depending on whether vulnerabilities are leveraged using inter component communication (ICC) or through the use of internet sockets.

In the second category, both applications are malicious and they collude to gain a permission set that is not allowed for either of them. An example of this type of attack is the Soundcomber (Schlegel et al., 2011), in which one application has permission to monitor and record audio call activity and another application has internet access permission. Both applications collude to extract credit card information using targeted tone- and speech-analysis and send the data to a remote attacker. Marforio et al. (2012) analysed different mechanisms that could be used for communication by colluding applications, which may be used further for escalation of privileges. Mechanisms for communication channels can be classified into three categories: Application, Operating System and Hardware levels; and may be either overt or covert. It can be argued that if applications collude such that, a privileged application sends data to an unprivileged application (which does not have the requisite permissions to access the data), it results in a privilege escalation attack. Colluding applications may communicate directly using ICC or internet sockets, or through indirect communication like covert channels. Concentrating on covert channels as a means of communication, there are different mechanisms possible at each of the levels: Application level – through the use of

Figure 4 Classification of application-level privilege escalation attacks



single/multiple settings, types of Intents, automatic Intents; Operating system level – through thread enumeration, UNIX socket delivery, free space on file systems; Hardware level – through timing channels and processor frequency. Colluding applications can either communicate directly through the use of ICC or internet sockets, or indirectly by sharing files or through covert/overt channels in system components. Since covert channels are not as popular or effective in terms of time of attack, this paper restricts the discussion to file systems and Unix sockets. As will be discussed in Section 5, the existing security solutions to prevent privilege escalation attacks are insufficient to adequately prevent collusion attacks (Marforio et al., 2012), which remains an open research problem.

Zhongyang et al. (2013) discuss another classification for privilege escalation attacks based on the type of cooperation between applications code cooperation and data cooperation. In code cooperation, a less privileged application invokes the components of a more privileged application, whereas in data cooperation, it gets sensitive information or data from the more privileged application. Each of these types may use mechanisms discussed in the previous classification.

5 Defense discussion

After defining and categorising different types of privilege escalation attacks, it is important to focus on the defense strategies to use to prevent them. It is essential to identify the requirements that a defense mechanism must satisfy, in order to build an efficient system.

5.1 Requirements of security mechanisms

Defense mechanisms should satisfy the following (Felt et al., 2011a):

- *Prevention of privilege escalation*: Defense mechanisms should be able to prevent privilege escalation attacks of both categories – confused deputy and colluding applications that use either code or data cooperation.
- *Developer independence*: The mechanism should not rely on the developers to write their applications cautiously to meet some of the security requirements (if developers were themselves security minded, privilege escalation would be prevented at the root cause level).
- *Ease of development*: Applications should retain their functionality without imposing excessive burden on the developers.
- *Dynamic*: The mechanism should be dynamic, consider different runtime behaviours and not require application analysis.

5.2 Adversary model

Bugiel et al. (2012) propose a scalable adversary model that can be used as reference to build a defense mechanism:

- *WeakAdversary* is able to launch only known confused deputy attacks through ICC mechanisms. This is the simplest case of an adversary, and since the type of attacks are known, defense is relatively easier.
- *BasicAdversary* is able to launch confused deputy attacks using both ICC mechanisms as well as through internet sockets. Most research addresses this type of Adversary, since preventing privilege escalation attacks with colluding applications is a tougher challenge.
- *AdvancedAdversary* is able to launch any type of confused deputy attack as well as unknown collusion attacks that use the direct ICC mechanism.
- *StrongAdversary* is able to launch any type of application-level privilege escalation attack (confused deputy or collusion attacks) via all types of communication mechanisms discussed earlier.

An ideal security solution should be able to prevent a StrongAdversary from attacking the system. Although this model gives a coarse categorisation, it will be used to compare different security frameworks and techniques in Section 6.2.

5.3 General strategies for defense

Felt et al. (2011a) discuss some general strategies for defense against privilege escalation attacks. These methods include:

Capabilities: A capability is an un-forgeable, shareable token that, when used, grants access to a privilege (Hardy, 1988). A method of preventing privilege escalation attacks is to have the deputy (or any other application) ask for a capability token from its requester to be able to make API calls. This method does not satisfy developer independence and could result in malicious developers circumventing this mechanism.

Taint tracking: When a requester application makes a request, its data can be a source of the taint which can then be tracked through the application till it reaches a sink (i.e., API call). If tainted data reaches a corresponding sink, privilege escalation is possible. This method incurs a large overhead when used to track both control and data flows through an application and can also result in taint explosion.

MAC: Mandatory access control systems involve the operating system enforcing certain access and control flow policies across different levels of confidentiality and integrity. In such systems, no information can flow from low-integrity principals to high-integrity principals or from high-confidentiality principals to low-confidentiality principals. Android applications do not follow strict relationships of confidentiality and integrity between applications, and hence it is not feasible to have strict MAC rules.

Stack Inspection: When a privileged API call is made, the system can check the call stack (with some modifications) to determine the applications in the call chain and verify their privileges. However, this approach can not prevent against attacks if calls are asynchronous and they do not appear in the same call stack.

HAC: History based access control involves reduction of permissions of trusted code if it interacts with untrusted code. This mechanism places constraints on application functionality.

After looking at the requirements and the adversary model, the analysis tools and security extensions to the platform can be discussed.

6 Analysis tools, security frameworks and techniques

Bugiel et al. (2012) discuss the security extensions for the Android platform which is extended and depicted in the Figure 5. Most proposed solutions require changes to Android's middleware with extensions to components of the application installer, the reference monitor, the permissions database and the Dalvik Virtual Machine. These tools aim at achieving objectives that can be commonly listed as:

- system-centric protection mechanism
- a general solution to prevent all (or most) types of attacks
- compatibility with legacy Android applications
- low performance overhead in mobile phone usage.

Monitoring techniques face certain general challenges:

- smartphones are resource-constrained limiting the use of heavyweight techniques
- techniques need to distinguish between different types of information, requiring more storage and processing
- privacy information can be dynamic, making it difficult to track
- applications share information amongst each other which requires monitoring between them.

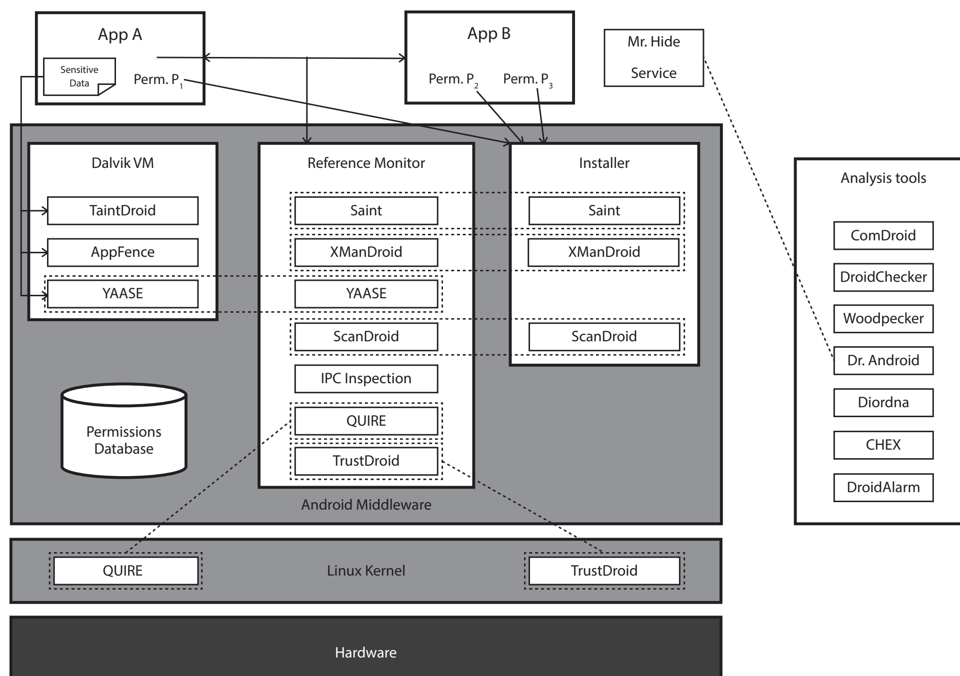
There are various tools proposed, some perform analysis of applications to detect capability leaks, some track the flow of sensitive data through an application, whereas others focus on prevention of privilege escalation (Enck, 2011). This section discusses a classification of these methods according to the technique used and also provides a comparison between some of them.

6.1 Analysis tools

SCanDroid (Fuchs et al., 2009) is an analysis tool that performs incremental checking of applications as they are installed on an Android device. It extracts security specifications from the manifest file and determines whether data flows through the application satisfy and are consistent with these specifications. It was the first program analysis tool aimed at providing security certification of Android applications. It uses the tool *WALA* (http://wala.sourceforge.net/ wiki/index.php/Main_Page) that parses through a set of Java classes and generates a call graph for all reachable methods. It involves string analysis to recover addresses of components, pointer analysis to track flows through the heap and handles interprocedural flows through JVML byte code. However, it remained in the research stage and was not implemented on real world applications.

ComDroid (Chin et al., 2011) is a static application analysis tool that detects vulnerabilities in Android

Figure 5 Security frameworks for Android



Source: Bugiel et al. (2012)

applications. The authors examined security challenges from the perspectives of Intent senders and receivers. ComDroid disassembles applications using the tool *Dedexer* (http://dedexer.sourceforge.net/) and then parses the output to detect potential Intent and component vulnerabilities. It examines Intent creation and transmission to detect any unauthorised receipt of Intents which could result in Broadcast theft, Activity hijacking or Service hijacking. A malicious application may launch an Intent spoofing attack by sending an Intent to an exported component that is not expecting Intents from that application, resulting in malicious Activity or Service launch. ComDroid's component analysis decides whether components might be susceptible to an Intent spoofing attack by examining the application's manifest file and performing intraprocedural analysis on each component. 20 applications were analysed with ComDroid and manually verified in which 34 exploitable vulnerabilities were found – 12 of the 20 applications had at least one vulnerability. ComDroid can lead to false negatives because it does not consider the reason for control flow in code.

DroidChecker (Chan et al., 2012) is another static application analysis tool which searches for vulnerabilities in Android applications. It performs a check on the manifest file of the application to detect vulnerable components in the application on the basis of a decision making process. This list of potentially vulnerable components is then used along with the decompiled source files obtained using the tool *dex2jar* (https://code.google.com/p/dex2jar/) to extract a call graph. Static taint analysis is then performed on this graph to identify paths between public entry points and API calls data as well as action calls, which could result in capability leaks. Their previous work (Chan et al., 2011) only checks for vulnerabilities based on the manifest file which was then further enhanced to determine capability leaks. An experiment with 1179 Android applications revealed six applications with true capability leaks. The sample attack in the Adobe Photoshop Express 1.3.1 application was discussed in Section 3.2.

Woodpecker (Grace et al., 2012) is also a static analysis application tools for determining capability leaks in Android applications. It performs data flow analysis of pre-loaded applications to find the reachability of dangerous permissions from a public interface. Woodpecker categorises capability leaks into two types: explicit – similar to capability leaks in a confused deputy problem and implicit – similar to maliciously colluding applications. Implicit leaks are caused by the abuse of the 'sharedUserId attribute in the manifest which gives applications of the same author the same ID allowing all such applications to acquire a union of permissions. Woodpecker extracts the pre-loaded applications and the manifest file from the phone image, then uses the disassembler *baksmali* (https://code.google.com/p/smali/) to get an intermediate representation. It then constructs a control flow graph and examines public unprotected interfaces that have paths leading to capabilities to detect explicit leaks. The system detects and reports the use of an unrequested capability by another application. 8 phone images were tested to find that among 13 privileged permissions examined, 11 were leaked, with individual phones leaking up to 8 permissions.

Diordna (Zhong et al., 2012) is another static analysis tool that checks the various sources of permission re-delegation and generates test cases for IPC unit testing. It uses the tool *dex2jar* (https://code.google.com/p/dex2jar/) to first convert the application into jar and then *WALA* (http://wala.sourceforge.net/ wiki/index.php/Main_Page) to construct a call graph of the application and a control flow graph for each method in it. Diordna works on the bytecode obtained from *WALA* to find all the permission-redelegation points and then traverses back from them towards the public entry points, thus marking the method trace. It then uses these method traces to generate test cases along with taint propagation analysis. Diordna detected permission re-delegation vulnerabilities and generated appropriate test cases for Music and DeskClock system applications focusing on the MediaPlaybackService and AlarmKlaxon. It is limited by the completeness of the graph generated by *WALA* and can not take into account runtime dynamic features of applications.

CHEX (Component Hijacking EXaminer) (Lu et al., 2012) is a static analysis tool that detects component hijacking vulnerabilities, in which an unauthorised application *A* reads sensitive data from or writes to the critical region of another application *B* using the public component interfaces of *B*. It is designed for a category of attacks different from privilege escalation attacks, but can indirectly detect them. CHEX is built on the Dalysis (Dalvik bytecode analysis) framework that works directly on Dalvik bytecode, which overcomes the limitation of incomplete reconstruction in other analysis tools. It treats the analysis as a data-flow problem, by splitting the application code into different parts to assess the asynchronous invocations of different interfaces. It first detects entry point methods in the application obtained from Dalysis and generates a data-flow summary by splitting the application code. Depending on the permutations of possible split sequences, it creates a permutation data-flow summary and data dependence graph. By testing source to sink connectivity in this graph, it detects the existence of vulnerabilities. CHEX targets at general vulnerabilities in all Android applications, and can only detect privilege escalation vulnerabilities at the ICC level, but not data cooperation that uses file systems or network sockets.

AndroidLeaks (Gibler et al., 2012) is a static analysis framework that detects leakage of sensitive information from applications on a large scale. It considers the exfiltration of data off the phone to be a privacy leak, regardless of whether it is a malicious leak or a part of the application's functionality. A permission map is created between APIs and their respective required permissions by analysing the Android source code. AndroidLeaks first converts an application into a JAR file using *dex2jar* (https://code.google.com/p/dex2jar/) and then builds a call graph from the byte code using *WALA* (http://wala.sourceforge.net/ wiki/index.php/Main_Page). It then iterates through the call graph to find source and sink API calls. A source API call is one that generates sensitive data, whereas a sink API call is one which can send the data

off the device. If such API calls are found, it uses static taint analysis to determine if the data from a source reaches a network sink and also handles taint analysis for data returned from callback functions. AndroidLeaks detected 57,299 leaks of various kinds in 7414 Android applications. It does not take into account Android-specific data and control flows, and covers a wide range of data flow leakages, a large number of which can be false positives for privilege escalation vulnerability analysis. It can, thus, detect vulnerabilities that can lead to data cooperation attacks, but can not detect code cooperation vulnerabilities.

DroidAlarm (Zhongyang et al., 2013) is a more recent static analysis tool that conducts a complete analysis of the application to first identify potential capability leaks and then analyse the leak paths. It works directly on the Dalvik bytecode as well. It is specifically used to detect leaks in Android malware, recognising that privilege-escalation malware can hide their sensitive functions by using exposed interfaces of other susceptible applications, making them more difficult to detect. It uses the tool *apktool* (https://code.google.com/p/android-apktool/) to extract permissions from the manifest file and checks each item for its permission level to detect sensitive components. For each sensitive component, it then checks for public interfaces by extracting information from the dex file using the tool *androguard* (https://code.google.com/p/androguard/). Based on the results of these steps, it flags an application as suspicious. For a suspicious application, DroidAlarm then uses separate techniques to detect code or data cooperation type of attacks. To detect code cooperation, it uses an iterative process to construct a path between a public interface and a sensitive API. For detecting data cooperation, it analyses the usage of sensitive registers from APIs to public interfaces, through series of function calls. DroidAlarm does not consider legitimate exposure of public interfaces in applications since it is designed mainly to analyse capability leaks in malware, which are initially known to be malicious applications. It also can not detect data cooperation through file systems or network sockets.

A summary of the analysis tools is provided in Table 1, which serves as a comparison between them. The analysis tools are likely to be incomplete because they can not completely predict runtime behaviour and communication. They rely on tools such as *dex2jar* (https://code.google.com/p/dex2jar/), *WALA* (http://wala.sourceforge.net/ wiki/index.php/Main_Page), *Dedexer* (http://dedexer.sourceforge.net/), etc. to generate byte codes, control flow graphs and call graphs which may not be complete. They can detect vulnerabilities that could prevent privilege escalation due to ICC, but not other methods. Analysis tools may not be specifically targeted at preventing privilege escalation attacks, but they can be used in combination with other tools to satisfactorily detect vulnerabilities and prevent attacks.

6.2 Security extensions

Secure application interaction (*Saint*) (Ongtang et al., 2009) is an extension to the Android security architecture with policies that address the missing abilities for applications as discussed in Section 2.3. It allows application developers to define access control rules for their components, providing a fine-grained access control model. The decision mechanism is based on signatures, configuration of the calling application and context in which the call is made (e.g., location) and decisions are enforced at both install-time and run-time. Thus, Saint relies on developers to ensure that the caller application has at least the same permissions

Table 1 Summary of analysis tools

Analysis tool	*Technique used*	*Works with*	*Detects*
ScanDroid (Fuchs et al., 2009)	Data flow tracking across application	Bytecode	Determines whether data flows are consistent with security specifications
ComDroid (Chin et al., 2011)	Intraprocedural vulnerablity analysis	Decompiled files	Intent and component vulnerabilities
DroidChecker (Chan et al., 2012)	Static taint analysis	Decompiled files	Component vulnerabilities and capability leaks through Intents (using forward approach source to sink)
Woodpecker (Grace et al., 2012)	Control flow graph analysis	Stock phone image, Bytecode	Capability leaks (using forward approach source to sink)
Diordna (Zhong et al., 2012)	Call graph and control flow graph analysis	Bytecode	Component vulnerabilities (using backward approach sink to source)
CHEX (Lu et al., 2012)	Data flow analysis	Bytecode	Component vulnerabilities (using forward approach source to sink)
AndroidLeaks (Gibler et al., 2012)	Call graph and static taint analysis	Bytecode	Leakage of sensitive information through network sinks
DroidAlarm (Zhongyang et al., 2013)	Function call tracking	Bytecode	Capability leaks in Android malware

as the callee to prevent confused deputy attacks, thus, not providing developer independence which makes it an error-prone approach. If no Saint policy exists, the access is implicity allowed. It is an application centric mechanism and not system-centric. Saint can not address the problem of malicious developers building applications that maliciously collude to achieve privilege escalation without being controlled by Saint's policies.

TaintDroid (Enck et al., 2010) is a security extension that tracks the flow of sensitive data through third party applications in real-time. It performs dynamic taint analysis, where it tracks how data flows between applications through variables, messages and files, and also out to the network through Binder IPC. This real-time data allows users to monitor data flow and detect suspicious activity of malicious applications. It labels private data with a taint mark, and follows tainted data as it propagates through the system, alerting the user when it leaves the system through a taint sink (e.g., network interface). TaintDroid can track and detect data leakages occurring from privilege escalation attacks. It performs taint analysis at four levels of granularity: variable-level, method-level, message-level, and file-level. It can track data flows through the system, however, it can not efficiently track control flows (where privilege escalation is also possible) since it causes heavy performance issues. TaintDroid detects leakages but can not distinguish between legitimate and malicious data leakage without a policy based decision making mechanism. Since it only tracks data flows, it can detect only data cooperation privilege escalation attacks.

AppFence (Hornyack et al., 2011) is an extension of TaintDroid framework to allow users to transparently enable privacy control mechanism on existing unmodified Android applications. It allows users to withhold data from suspicious applications that do not need the data to perform their advertised functionality by providing two approaches: shadowing sensitive data and blocking sensitive data from being ex-filtrated off the device. In data shadowing, if an application does not require sensitive data, shadowed data is provided to it, for example, an application requesting the device's location is returned the coordinated dummy coordinates. To block exfiltration, AppFence intercepts attempts to associate domain names to sockets and to write tainted data to a socket. It then either drops the message overtly or covertly. It also tackles the usability issue of applications when their permissions are removed. However, like TaintDroid, AppFence can only detect data leakages, thus, data cooperation attacks and can not directly detect privilege escalation attacks.

QUIRE (Dietz et al., 2011) is a security extension that provides a lightweight provenance system to prevent confused deputy attacks. It tracks the IPC call chain to determine if the caller application has the corresponding permissions, allowing an application to choose if it wishes to operate with reduced privileges or act explicitly on its own behalf. It annotates IPCs such that a receiver application has knowledge of the history of the complete call chain. It also extends the network module in the Android Linux kernel to provide a lightweight signature scheme that allows the application to create a signed statement. This can be used to verify remote procedure calls (RPCs) when they leave the device. QUIRE employs simple cryptographic mechanisms to protect data moving over IPC and RPC channels. However, QUIRE is application centric unlike XManDroid (Bugiel et al., 2011a) or YAASE (Rusello et al., 2011) and can not prevent against attacks by colluding applications because they may circumvent QUIRE's mechanism by dropping the IPC call chain and acting on their own behalf. It can not attacks that exploit covert channels in Android's core. Also, it is not transparent to the application developer since existing applications need to be rewritten.

IPC Inspection (Felt et al., 2011a) is a mechanism similar to QUIRE that prevents confused deputy attacks using Binder IPC. IPC Inspection reduces a deputy application's privileges if it receives a message from a less privileged application. If the privilege is reduced, it indicates that a deputy is under the influence of another application. In a chain of influence between applications, IPC Inspection reduces the privileges if any application lacks appropriate permission(s). IPC Inspection does not require a policy framework and thus, does not suffer from the problem of unknown attacks due to policy incompleteness. However, it does not provide a solution for privilege escalation attacks due to colluding applications or situations where a malicious deputy can directly abuse its privileges. Colluding applications can reside in the same sandbox without proper isolation and communicate freely. It also does not defend against attacks where an unprivileged application returns a malicious value in a request-reply IPC. IPC Inspection also poses an overhead since it requires multiple instances of applications with different privilege sets to be maintained – a primary interface that the user interacts with and multiple instances for each time a message is sent to the application.

XManDroid (Bugiel et al., 2011a) is a security framework that extends the Android system's reference monitor for real-time detection and prevention of privilege escalation attacks. XManDroid dynamically monitors communication between applications through the extension in the reference monitor and verifies them with security rules defined in a system centric policy, thereby, detecting any transitive usage of privileges. XManDroid can successfully detect communication links between dynamically created components, handle exceptional cases of pending intents, and prevent communication through covert channels between system components. It can detect all privilege escalation attacks (confused deputy and colluding types) that use ICC mechanisms. The authors claim that XManDroid is efficient and causes negligible performance overhead for the user. However, a large number of false positives in their experimental manual usability test motivated them to further enhance it as XManDroid2[2] (Bugiel et al., 2012) with a kernel module to enable MAC using TOMOYO Linux. This employs a faster and more efficient ICC call-chaining mechanism that is based on modifications to the Binder mechanism instead of Intents alone. The novelty of this technique is the runtime interaction of security extensions with the Android middleware and TOMOYO Linux, allowing dynamic runtime policy mapping from the middleware to the

kernel. It tracks ICC calls, operations on files, Unix domain sockets and internet sockets, thus protecting against the StrongAdversary model. However, it relies on a set of system policies which, if incomplete, will not be able to prevent against unknown attacks.

Yet another Android security extension ($YAASE$) (Rusello et al., 2011) was designed with the motivation of having a single solution with mechanisms that are needed for user privacy protection. YAASE provides a fine-grained policy enforcement mechanism by having different levels of control granularity over accesses to phone resources. It employs data tracking and tagging, similar to TaintDroid, for user-defined labels (private, confidential, public, etc.) associated with inter-application and application-to-internet data flows. Policies define the data labels that an application can access and violations cause alerts to users. YAASE can work without requiring collaboration with applications and controls which labels can be associated with each application. With this strategy, it can detect privilege escalation attacks of both confused deputy and colluding applications types. Since YAASE does data tracking, it can detect only data cooperation based attacks and not purely ICC based attacks. Experimental results show that accessing un-optimised data structures incurs heavy overheads, whereas, the overhead for tracking data sent over the internet is acceptable.

TrustDroid (Bugiel et al., 2011b) is a security extension that aims at providing isolation between applications belonging to different domains depending on their trust levels. This framework works at the middleware layer to prevent inter-domain application communication and data access, at the kernel layer to enforce MAC on the file system and on IPC channels, and at the network layer to filter network traffic. TrustDroid performs fine-grained analysis filtering on application data and data stored in databases preventing unauthorised access to it. It relies on a method of classifying applications into separate domains at install-time and assigns each application a separate colour according to the domain. A colour is associated with system Content providers and service providers as well to confine their usage of data to a particular domain. Malicious applications can not use interfaces of applications belonging to other domains, even if the interfaces are exposed as public, because the colours assigned would be different. This prevents cross-domain direct ICC, IPC, file system sharing and consequently cross-domain privilege escalation attacks. However, it can not provide protection against attacks (both confused deputy and colluding applications) taking place within the same domain. The access control imposed is static as compared to other works like XManDroid (Bugiel et al., 2011a) which offers a more dynamic solution.

Kantola et al. (2012) propose modifications to the Android platform (DK[3]) that detect and protect accidental inter-application messages that should have actually been intra-application messages. They identify a subclass of communication vulnerabilities in which applications unintentionally expose their components or messages to third party applications and provide a solution to detect and patch such vulnerabilities automatically. A heuristic is developed depending on combinations of permissions and component behaviours to determine when components should be exported. Although the main focus of this work is not to prevent privilege escalation attacks, by making unintentionally exposed components private, this work can prevent some access by third-party Intents, thus preventing a subset of privilege escalation attacks. It can solve both attacks due to both confused deputy and colluding applications, restricted to only those applications where components are exposed unintentionally.

A summary of the techniques used in the different security extensions is provided in Table 2. Table 3 gives a comparison of the different security extensions. Each mechanism is focused at preventing particular security vulnerabilities and has its limitations. Mechanisms like Saint (Ongtang et al., 2009), TaintDroid (Enck et al., 2010), AppFence (Hornyack et al., 2011) and TrustDroid (Bugiel et al., 2011b) are not aimed at preventing privilege escalation but their mechanisms can indirectly prevent some types of attacks. On the other hand frameworks like IPC Inspection (Felt et al., 2011a), XManDroid (Bugiel et al., 2011a), XManDroid2 (Bugiel et al., 2012), YAASE (Rusello et al., 2011), QUIRE (Dietz et al., 2011), DK (Kantola et al., 2012) are designed to prevent specific categories of attacks, but they have their own limitations.

Saint (Ongtang et al., 2009) does not provide developer independence because it requires developers to specify access control rules. TaintDroid (Enck et al., 2010) does not provide runtime independence since it requires user action when data leakage is detected. The other frameworks provide all the general requirements discussed in Section 5.

Table 2 Summary of security extensions

Security extension	*Technique used*	*Adversary model*
Saint (Ongtang et al., 2009)	Monitoring of runtime permission usage	Strong (without unknown attacks)
TaintDroid (Enck et al., 2010)	Tracking flow of sensitive data, taint analysis	N/A
AppFence (Hornyack et al., 2011)	Tracking flow of sensitive data, taint analysis	N/A
QUIRE (Dietz et al., 2011)	Tracking of call-chain IPC	Weak
IPC Inspection (Felt et al., 2011a)	Tracking of call-chain IPC	Weak
XManDroid (Bugiel et al., 2011a)	Customised framework	Advanced
XManDroid2 (Bugiel et al., 2012)	Customised framework	Strong (without unknown attacks)
YAASE (Rusello et al., 2011)	Tracking flow of sensitive data, taint analysis	N/A
TrustDroid (Bugiel et al., 2011b)	Data colouring and domain isolation	N/A
DK (Kantola et al., 2012)	Heuristic of permissions and component behaviours	N/A

An ideal framework would provide a complete solution that takes into account all types of privilege escalation attacks – confused deputy and colluding (with all mechanisms of achieving them) – known as well as unknown; and requirements with respect to usability, development, resource constraints and dynamism. The frameworks discussed individually satisfy and provide a subset of these requirements.

6.3 Other analysis tools and extensions

Analysis tools and extensions that are concerned with permissions and not directly related to privilege escalation attacks are discussed here:

Kirin (Enck et al., 2008) is an extension to the Android application installer. It provides a framework that checks the permissions requested by applications at install-time by analysing the *AndroidManifest.xml* file and allows only those applications whose permissions comply with a system-centric policy. The main goal of Kirin is to mitigate malware in a single application. Also, the Kirin framework can identify communication links that are security critical by analysing the interfaces that the application is authorised to communicate with. Enck et al. (2008) discuss certain policy invariants with respect to core system functionality, user privacy and applications that represent realistic security requirements of the phone. The model allows automated analysis which can identify insecure policy configurations that can leave the phone in a vulnerable state. It is static in nature and does not consider real-time behaviour, thus, it has to consider all possible communication links over unprotected interfaces. Also, it could result in a large number of false positives since Kirin would disallow applications that can potentially establish arbitrary communication links over the unprotected interfaces.

Android permission extension (*Apex*) (Nauman et al., 2010) is an extension to the Android permission mechanism. It provides a policy enforcement framework which allows a user to selectively grant or deny permissions to applications at install-time. It also allows the user to constraint the

Table 3 Comparison of security extensions

Security extension	*CDA-ICC*	*CDA-IS*	*CA-ICC*	*CA-I*	*Limitations*
Saint (Ongtang et al., 2009)	✓	✓	✓	✓	• Can prevent only known attacks only if predicted by the developer • Relies on (honest) developers to ensure security
TaintDroid (Enck et al., 2010)				✓	• Can not track control flows without performance overhead • Can not distinguish between legitimate and malicious data leakages • Can detect only data cooperation based attacks • Can not detect colluding attacks
AppFence (Hornyack et al., 2011)				✓	• Does not directly detect privilege escalation attacks • Can detect only data cooperation based attacks • Can not detect colluding attacks
QUIRE (Dietz et al., 2011)	✓				• Not a system centric approach • Can not prevent collusion attacks • Not a developer independent solution
IPC Inspection (Felt et al., 2011a)	✓				• Overhead of multiple instances • Cannot prevent collusion attacks • Can detect only confused deputy attacks based on ICC
XManDroid (Bugiel et al., 2011a)	✓		✓		• Large number of false positives due to policy incompletion or over estimation • Can detect only ICC based attacks
XManDroid2 (Bugiel et al., 2012)	✓	✓	✓	✓	• Relies on policy completion for prevention against all attacks • Can not prevent unknown attacks
YAASE (Rusello et al., 2011)	✓	✓	✓	✓	• Incurs heavy overheads for accessing unoptimised data structures • Can detect only data cooperation attacks
TrustDroid (Bugiel et al., 2011b)	✓	✓	✓		• Not a dynamic approach – relies on pre-classification of applications • Can not prevent intra-domain privilege escalation attacks
DK (Kantola et al., 2012)	✓		✓		• Only for applications whose components are exposed unintentionally

CDA-ICC: Confused deputy attacks due to ICC.
CA-ICC: Colluding Attacks due to ICC.
CDA-IS: Confused Deputy Attacks due to internet Sockets.
CA-I: Colluding Attacks due to Indirect.

usage of resources by defining certain runtime policies. Poly, an advanced application installer, provides the user with an interface for fine-grained control over individual permissions demanded by the application allow, deny or constrain in number of times and time of the day. The Apex framework monitors the usage of permissions to ensure that the application conforms to this defined policy. Apex makes the Android permission system more flexible but it relies on users to make the security decisions which may be error-prone. Although this approach may allow a user to prevent confused deputy attacks, it can not prevent attacks by colluding applications in which permissions are split over different applications.

Stowaway (Felt et al., 2011b) is a static analysis tool built on top of *ComDroid* and it checks if developers follow the principle of least privilege in terms of permission requests. Stowaway determines the set of API calls that an application uses and then maps those API calls to permissions. Automated testing tools were used on the Android API in order to build a permission map to detect over privilege. The permission map correlates permissions to API calls, Intents and ContentProviders and covers 85% of the Android API. Stowaway decompiles Android applications using *Dedexer* (http://dedexer.sourceforge.net/) and then performs static analysis on the source code to determine the API calls, accesses to ContentProviders and transmission of Intents to detect vulnerabilities. 940 applications were checked and one third of them were found to be over privileged. The study also shows many applications leak sensitive information.

Dr. Android and Mr. Hide (Jeon et al., 2012) are a suite of tools that allow finer-grained permissions to be to be inferred on existing applications; to be enforced by developers on their own applications; and to be retrofitted by users on existing applications. Android implements coarse permissions some of which can be split into finer permissions that can reduce unnecessary privilege levels and prevent threat from vulnerabilities. The suite consists of 2 tools:

- Mr. Hide (the Hide interface to the droid environment) – a set of services that wrap several privileged Android APIs and dynamically enforces a specific set of finer-grained permissions onto applications
- Dr. Android (Dalvik Rewriter for Android) – a tool that removes existing permissions and replaces them with finer-grained permissions from Mr. Hide to be able to retrofit applications.

The advantage is that these tools do not require source code or recompilation, they work directly with downloaded applications. Mr. Hide consists of a service that runs in its own process on the device, and hidelib which is a replacement for sensitive APIs that manages interprocess communication with the Mr. Hide service. Dr. Android is a separate tool that uses *apktool* (https://code.google.com/p/android-apktool/) to decompress the application into its constituent files. It modifies classes.dex file and concatenates hidelib.dex to it; modifies the list of permissions in the manifest file and may modify some other resource files. By altering permissions to be finer-grained, this suite of tools prevents certain combinations of finer privileges between applications, indirectly preventing privilege escalation. However, it can not prevent developer-made maliciously colluding applications.

These tools can help reduce some of the causes of privilege escalation attacks, and combined with other security extensions could prove useful in defense.

7 Security guidelines and recommendations

Chin et al. (2011) and Chan et al. (2012) discuss guidelines on how applications should be developed to prevent capability and data leakages. They can be summarised as follows:

- Developers should avoid making components accessible to components of other applications to prevent capability leakage. This can be done by either not declaring any Intent filters, or setting *Android:exported* attributes to false in the manifest file.
- For components that must be public, they must be protected by appropriate permissions such that only other applications that have the permissions can access it.
- Developers should be aware of the distinction between inter-application and intra-application communication mechanisms.
- Different components, if possible, should handle inter- and intra-application communication.
- The *checkPermission()* system call should be used before invoking an API call to check if the caller/sender application possesses the required permissions.
- Applications should use explicit Intents when sending private data and also specify strong permissions to protect it.
- Applications should check the identity of a caller application, as well as the identity of the application returning any value since privilege escalation can take place through data leakage as well.

Though these guidelines are developer-dependent, they help eliminate some of the causes of vulnerabilities in the Android system and applications. Developer – independent solutions are needed to effectively prevent attacks on the vulnerabilities.

Based on these requirements, some recommendations can be made to make changes in the Android platform to effectively limit, if not completely prevent, vulnerabilities in a developer-independent manner. Recommendations by Chin et al. (2011) are extended here as follows:

- The Android system should use different mechanisms for inter- and intra-application communication. Currently Intents are used for both purposes, which unknowingly can lead to vulnerabilities.

- Accessibility of components available is either public or private. It should be extended to include three types: internal, exported to only the system and exported to other applications. By adding such constructs in the programming, the true intentions of developers can be captured while developing the application.
- To avoid unintentional Intent-sending vulnerabilities, the system should try to deliver an Intent to internal components first, and then if required to other external components.
- The Android system implicitly considers a component to be public if it declares an Intent filter. To avoid Intent-receiving vulnerabilities, the system should consider a component to be public only if it
  - sets the *exported* flag
  - has an Intent filter with a data field
  - has an Intent filter that registers to receive protected system actions
  - has a main launcher specification, or
  - has an Intent filter that registers to receive Intents with one of the standard Android actions.

Although these recommendations may introduce issues of backward-compatibility with existing applications and systems, a suitable trade-off between backward compatibility and security needs to be made to ensure that such vulnerabilities are limited.

8 Conclusion

Google's Android is a fast growing mobile operating system that implements some novel security mechanisms such as privilege separation. This paper discusses how these mechanisms have flaws resulting in vulnerabilities in applications that can be used to escalate privileges. Privilege escalation attacks are classified in this paper and various different security frameworks and techniques proposed in recent research are discussed and compared. The solutions developed do not individually provide a complete solution to prevent all types of attacks. With the number of applications rising at a very fast rate, vulnerabilities are also increasing, making the platform for attacks wider. Thus, there is a need for a technique that provides a complete solution against privilege escalation attacks along with satisfying all the usability requirements. This paper provides knowledge to inquisitive readers as well as guidelines to security researchers and application developers.

References

Bloomberg Businessweek (2012) *Google Says 700,000 Applications Available for Android*, http://www.businessweek.com/news/2012-10-29/google-says-700-000-applications-available-for-android-devices

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T. and Sadeghi, A. (2011a) *XManDroid: A New android Evolution to Mitigate Privilege Escalation Attacks*, Technische Universitt Darmstadt Technical Report of Center for Advanced Security Research Darmstadt, TR-2011-04.

Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A. and Shastry, B. (2011b) 'Practical and lightweight domain isolation on Android', *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*, Chicago, IL, USA, pp.51–62.

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A. and Shastry, B. (2012) 'Towards Taming Privilege-Escalation Attacks on Android', *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA, pp.346–360.

Chan, P., Hui, L. and Yiu, S.M. (2011) 'A privilege escalation vulnerability checking system for Android applications', *Communication Technology (ICCT), 2011 IEEE 13th International Conference on. IEEE*, Jinan, China, pp.681–686.

Chan, P., Hui, L. and Yiu, S.M. (2012) 'DroidChecker: analyzing Android applications for capability leak', *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, Tucson, Arizona, USA, pp.125–136.

Chin, E., Felt, A.P., Greenwood, K. and Wagner, D. (2011) 'Analyzing inter-application communication in Android', *Proceedings of the 9th International Conference on Mobile Systems, Applications*, Bethesda, MD, USA, pp.239–252.

Davi, L., Dmitrienko, A., Sadeghi, A. and Winandy, M. (2010) 'Privilege escalation attacks on Android', *Proceedings of the 13th International Conference on Information Security*, Boca Raton, FL, USA, pp.346–360.

Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A. and Wallach, D.S. (2011) 'Quire: lightweight provenance for smart phone operating systems', *Proceedings of the 20th USENIX Conference on Security*, San Francisco, CA, USA, pp.23–23.

Enck, W. (2011) 'Defending users against smartphone apps: techniques and future directions', *Proceedings of the 7th International Conference on Information Systems Security*, Kolkata, India, pp.49–70.

Enck, W., Ongtang, M. and McDaniel, P. (2008) *Mitigating Android Software Misuse Before It Happens*, Technical Report NAS-TR-0094-2008 The Pennsylvania State University, pp.22–22.

Enck, W., Ongtang, M. and McDaniel, P. (2009) 'Analysis of the communication between colluding applications on modern smartphones', *Proceedings of the 28th Annual Computer Security Applications Conference*, Honolulu, Hawaii, USA, pp.51–60.

Enck, W., Gilbert, P., Chun, B.G., Cox, L. P., Jung, J., McDaniel, P. and Sheth, A.N. (2010) 'TaintDroid: an information-flow tracking system for realtime privacy Monitoring on smartphones', *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, Vancouver, BC, Canada, pp.1–6.

Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S. and Chin, E. (2011a) 'Permission re-delegation: attacks and defenses', *Proceedings of the 20th USENIX Conference on Security*, San Francisco, CA, USA, pp.22–22.

Felt, A.P., Chin, E., Hanna, S., Song, D. and Wagner, D. (2011b) 'Android permissions demystified', *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, pp.627–638.

Fuchs, A., Chaudhuri, A. and Foster, J.S. (2009) *SCanDroid: Automated Security Certification of Android Applications*, Technical Report University of Maryland, College Park, http://www.cs.umd.edu/ avik/projects/scandroidascaa

Gibler, C., Crussell, J., Erickson, J. and Chen, H. (2012) 'AndroidLeaks: automatically detecting potential privacy leaks in Android applications on a large scale', *Proceedings of the 5th International Conference on Trust and Trustworthy Computing (TRUST '12)*, Vienna, Austria, pp.291–307.

Grace, M., Zhou, Y., Wang, Z. and Jiang, X. (2012) 'Systematic detection of capability leaks in stock Android smartphones', *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, San Diego, CA, USA.

Hobarth, S. and Mayrhofer, R. (2011) 'A framework for on-device privilege escalation exploit execution on Android', *Proceedings of the 3rd International Workshop on Security and Privacy in Spontaneous Interaction and Mobile Phone Use*, San Francisco, CA, USA.

Hornyack, P., Han, S., Jung, J., Schechter, S. and Wetherall, D. (2011) "These aren't the droids you're looking for': retrofitting Android to protect data from imperious applications', *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, pp.639–652.

Jeon, J., Micinski, K.K., Vaughan, J.A., Fogel, A., Reddy, N., Foster, J.S. and Millstein, T. (2012) 'Dr. Android and Mr. Hide: fine-grained permissions in Android applications', *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, Raleigh, NC, USA, pp.3–14.

Kantola, D., Chin, E., He, W. and Wagner, D. (2012) 'Reducing attack surfaces for intra-application communication in Android', *Proceedings of the Second ACM workshop on Security and Privacy in Smartphones and Mobile Devices*, Raleigh, NC, USA, pp.69–80.

Kaspersky (2012) *99% of all Mobile Threats Target Android Devices*, http://www.kaspersky.com/about/news/virus/2013/99_of_all_mobile_threats_target_Android_devices

Li, Q. and Clark, G. (2013) 'Mobile security: a look ahead', *IEEE Security and Privacy*, pp.78–81.

Lu, L., Li, Z., Wu, Z., Lee, W. and Jiang, G. (2012) 'CHEX: statically vetting Android apps for component hijacking vulnerabilities', *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, Chicago, IL, USA, pp.229–240.

Marforio, C., Ritzdorf, H., Francillon, A. and Capkun, S. (2012) 'Analysis of the communication between colluding applications on modern smartphones', *Proceedings of the 28th Annual Computer Security Applications Conference*, Orlando, Florida, USA, pp.51–60.

Nauman, M., Khan, S. and Zhang, X. (2010) 'Apex: extending Android permission model and enforcement with user-defined runtime constraints', *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, Beijing, China, pp.328–332.

Hardy, N. (1988) 'The confused deputy: (or why capabilities might have been invented)', *ACM SIGOPS Operating Systems Review*, Vol. 22, No. 4, pp.36–38.

Ongtang, M., McLaughlin, S., Enck, W. and McDaniel, P. (2009) 'Semantically rich application-centric security in Android', *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC '09)*, Honolulu, Hawaii, USA, pp.340–349.

Russello, G., Crispo, B., Fernandes, E. and Zhauniarovich, Y. (2011) 'YAASE: yet another Android security extension', *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust and 2011 IEEE Third International Conference on Social Computing*, Boston, MA, USA, pp.1033–1040.

Schlegel, R., Zhang, K., Zhou, X., Intwala, M., Kapadia, A. and Wang X. (2011) 'Soundcomber: a stealthy and context-aware sound Trojan for smartphones', *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, pp.17–33.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S. and Glezer, C. (2010) 'Google Android: a comprehensive security assessment', *IEEE Security and Privacy*, Vol. 8, No. 2, pp.35–44.

Strategy Anayltic (2013) http://www.strategyanalytics.com

The Lookout Blog (2012) *UPDATE: Security Alert: Android Trojan GGTracker Charges Premium Rate SMS Messages*, https://blog.lookout.com/blog/2011/06/20/security-alert-android-trojan- ggtracker-charges-victims-premium-rate-sms-messages

Yahoo! Finance (2013) *Strategy Analytics: Android and Apple iOS Capture a Record 92 Percent Share of Global Smartphone Shipments in Q4 2012*, http://finance.yahoo.com/news/strategy-analytics-android-apple-ios-114300553.html

Zhong, J., Huang, J. and Liang, B. (2012) 'Android permission re-delegation detection and test case generation', *2012 International Conference on Computer Science and Service System*, Nanjing, China, pp.871–874.

Zhongyang, Y., Xin, Z., Mao, B. and Xie, L. (2013) 'DroidAlarm: an all-sided static analysis tool for Android privilege-escalation malware', *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, Hangzhou, China, pp.353–358.

Notes

[1] A user can register as a developer for $25 and publish applications freely thereafter.

[2] Called XManDroid2 for easier referencing.

[3] Called DK for easier referencing.

Websites

androguard: Reverse Engineering, Malware and Goodware Analysis of Android Applications, https://code.google.com/p/androguard/

apktool: Tool for Reverse Engineering Android Apps, https://code.google.com/p/android-apktool/

smali/baksmali: Assembler/Disassembler for the dex Format, https://code.google.com/p/smali/

Dedexer: Disassembler Tool for DEX Files, http://dedexer.sourceforge.net/

dex2jar: Tool to Convert Android .dex to .jar Files, https://code.google.com/p/dex2jar/

eLinux.org, Android Architecture, http://elinux.org/Android_Architecture

WALA: T.J. Watson Libraries for Analysis, Android Architecture, http://wala.sourceforge.net/wiki/index.php/Main_Page