POSTER: The Quest for Security against Privilege Escalation Attacks on Android

Sven Bugiel[†], Lucas Davi[†], Alexandra Dmitrienko[§], Thomas Fischer[‡], Ahmad-Reza Sadeghi[†§], Bhargava Shastry[§]

[†]System Security Lab/CASED
Technische Universität Darmstadt
Darmstadt, Germany

[‡]System Security Lab
Ruhr-Universität Bochum
Bochum, Germany

[§]Fraunhofer SIT
Darmstadt, Germany

ABSTRACT

In this paper we present the design and implementation of a security framework that extends the reference monitor of the Android middleware and deploys a mandatory access control on Linux kernel (based on Tomoyo [9]) aiming at detecting and preventing application-level privilege escalation attacks at *runtime*. In contrast to existing solutions, our framework is system-centric, efficient, detects attacks that involve communication channels controlled by both, Android middleware and the Linux kernel (particularly, Binder IPC, Internet sockets and file system). It can prevent known confused deputy attacks without false positives and is also flexible enough to prevent unknown confused deputy attacks and attacks by colluding applications (e.g., Soundcomber [11]) at the cost of a small rate of false positives.

Categories and Subject Descriptors: D.4.6 [Operating Systems]: Security and Protection

General Terms: Security

1. INTRODUCTION

Google Android [1] is a modern software platform for smartphones with rapidly expanding market share [8]. The software stack of Android includes Linux kernel, middleware and an application layer. Android devices run applications that are distributed through the Android market. Google's application distribution model is not very restrictive and allows anyone who has registered as an Android developer (and paid $25 fee) to publish apps. This implies that some of these apps are very likely to be malicious.

To mitigate the malware threat, Android deploys security mechanisms, such as application sandboxing and a permission framework. The standard Android permission system limits access to sensitive data (SMS, contacts, etc.), resources (battery or log files) and to system interfaces (Internet connection, GPS, GSM). Each application at installation time requests permissions (from the user), and, if granted, Android reference monitor enforces permission checks at runtime. These measures are intended to confine damage imposed by malware within the privilege boundaries of an application sandbox.

However, as it has been shown [4, 7, 5], Android's security model is vulnerable to application-level privilege escalation attacks. Privileges assigned to applications at install time can be escalated at runtime. This implies that in contrast to the general belief the damage imposed by Android malware is not limited to the application's sandbox.

Application-level privilege escalation attacks. The recent privilege escalation attacks range from unauthorized phone calls [6] and text messages [4] to illegal downloads of malicious files [10] and context-aware voice recording [11]. In most scenarios, an application under control of the adversary escalates privileges by misusing a so-called confused deputy, a privileged application that exposes privileged interfaces for public use and does not check if the caller has appropriate permissions. Resent research results show that confused deputy vulnerabilities are common in third party applications [4, 7] as well as in Android default apps (web-browser [10], Phone [6], DesckClock and Settings [7]). Further, a very recent privilege escalation attack [11] demonstrates a severe attack scenario where two malicious applications collude in order to merge their permissions to get a permission set which might not be approved by the user when requested by a single app.

Remarkably, all application-level privilege escalation attacks occur by means of communication with other applications. Standard mechanism for communication among applications is a Binder-based lightweight Inter-Process Communication (IPC) provided by Android middleware. However, applications can also communicate through other channels that bypass the middleware, e.g., by sharing files or by establishing a local network connection [4]. Further, colluding applications can communicate directly with each other, or indirectly, by using another application or component in between. In the latter case, a mediating component can provide either covert (e.g., via a power manager or screen settings [11]) or overt (e.g., via a contacts database) channels to communicating applications.

Very recent security extensions to Android, QUIRE [5] and IPC Inspection [7], aim to address confused deputy attacks that occur over Binder IPC. QUIRE tracks the IPC call chain and recognizes the initiator of a security-critical request, giving the possibility to a potential confuse deputy to validate privileges of the call originator. This approach has the same deficiencies as the basic Android system due to the fact that the application developer is obliged to enforce security checks, thus developers without appropriate skills in security will most likely continue to produce vulnerable applications. The key insight of IPC Inspection is to reduce the permissions of an application when it receives an input from a less-privileged one. This approach requires the creation and maintenance of

Copyright is held by the author/owner(s).
CCS'11, October 17–21, 2011, Chicago, Illinois, USA.
ACM 978-1-4503-0948-6/11/10.

multiple application instances with different sets of privileges, and thus seems to be inefficient. Moreover, IPC Inspection may result in false positives and requires to overprivilege applications to tackle this issue. Also, neither IPC Inspection nor QUIRE are able to deal with confused deputy attacks launched via channels that bypass Binder IPC, e,g., attacks over socket connections [4].

Our goal and contributions. We aim for a security framework that addresses privilege escalation attacks and considers the above mentioned problems. In particular, we require an efficient and system-centric monitoring that controls communication channels in Android's middleware and in the Linux kernel, and tackles the known confused deputy attacks without false positives, but is also flexible enough to cover future privilege escalation attacks. In particular, our contributions are the following:

- We present the design and implementation of a security framework which extends the Android's reference monitor concept at both middleware and kernel level, that monitors communication links between applications and verifies them against rules defined in a security policy. Our framework is system-centric, efficient and can prevent known confused deputy attacks without false positives.

- The framework allows to cover unknown confused deputy attacks and attacks by colluding applications that communicate either directly or indirectly. Particularly, the attacks of Soundcomber [11] can be prevented that rely on indirect communication of colluding malicious apps over covert channels in Android system components. However, prevention of unknown attacks may have false positives, thus would require a trade-off between security and usability.

2. FRAMEWORK DESCRIPTION

Our framework performs runtime monitoring and analysis of communication links across applications in order to prevent potentially malicious communication based on the defined system-centric security policy.

State of Android system is represented as a graph, where vertices are application sandboxes, world-wide readable files[1] and Internet sockets. Edges in the graph show granted IPC calls, allowed file access or allowed socket connections.

Our extension is invoked when applications establish IPC connection, access files or connect to sockets. The framework validates whether the requested operation can potentially be exploited for a privilege escalation attack (based on the underlying security policy).

Architecture.

The architecture of our framework is shown in Figure 1. Generally, it builds upon *XManDroid* [3] – a framework that extends Android middleware, and enhanced with a kernel-level module[2]. In the following, we will explain components of our architecture and their interaction in the following use cases: (i) IPC call handling (steps 1-11), (ii) application (un-)installation (steps a-b), (iii) file/socket creation (steps A-C), (iv) file/socket read/write access (steps i-iv), and (iv) policy installation (steps I-III).

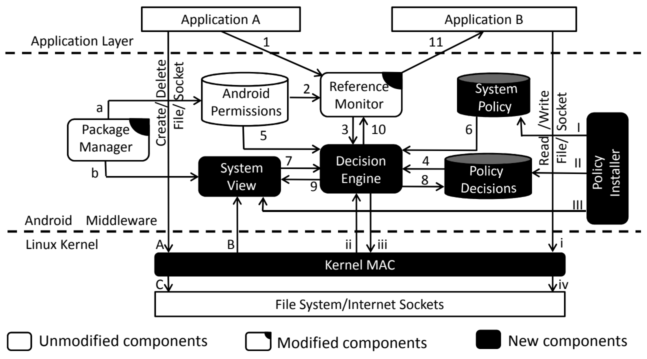

Figure 1: Framework architecture

IPC call handling. At runtime all IPC calls are intercepted by Android ReferenceMonitor (step 1). It obtains information about permissions from AndroidPermissions database (step 2) and validates permission assignments. If ReferenceMonitor grants access, it invokes DecisionEngine (step 3) to ensure the communication also complies to a system security policy. DecisionEngine first requests a record corresponding to this particular IPC call from PolicyDecisions (step 4). If the record is found, it means that a previously made decision can be applied. Otherwise, DecisionEngine makes a fresh decision which requires inputs from AndroidPermissions (step 5), SystemPolicy (step 6) and SystemView (step 7). The resulting decision is stored in PolicyDecisions (step 8), and if it is positive, SystemView is updated (step 9) reflecting that a communication link exists among applications A and B. Further, DecisionEngine informs ReferenceMonitor about the decision it has made (step 10), and ReferenceMonitor either allows (step 11) or denies the IPC call.

Application (un-)installation. Installation procedure involves a standard PackageManager of Android. Typically it extracts application permissions from the *Manifest file* (included in the application installation package) and stores them in a system database AndroidPermissions (step a). Additionally, PackageManager adds a new application in SystemView (step b).

Upon application un-installation, PackageManager revokes permission labels granted to the un-installing application from the AndroidPermissions database (step a) and removes this application from SystemView (step b).

File/socket creation/deletion. When an application requests to create or delete a world-wide readable file, the request is intercepted by KernelMAC - a mandatory access control (MAC) module deployed in Linux kernel (step A). KernelMAC updates SystemView (inserts/deletes a vertex which corresponds to a file/socket in the system graph) (step B) and performs the requested operation (step C).

File/socket read/write access. Read/write access to a wold-wide readable file or an Internet socket is intercepted by KernelMAC. Before access is granted, KernelMAC asks for a decision DecisionEngine. DecisionEngine runs a policy making algorithm on the graph (steps 4-9) and returns a decision (step iii). If decision is positive, access is granted (step iv).

[1]By default, all files created by applications are stored in private directory and inaccessible by other applications. However, files can be explicitly marked as world-wide readable to allow external access

[2]Our implementation integrates Tomoyo [9] Linux extension to enable kernel-level mandatory access control

Policy installation. PolicyInstaller writes/updates the system policy rules to the SystemPolicy database (step I). Next, it removes all decisions previously made by DecisionEngine, as those may not comply to a new system policy (step II). Also, SystemView component is reset into a clean state (step III). Note, that SystemView state is only reset upon update of SystemPolicy and persist across reboots.

3. POLICY

Policy consists of rules that define undesirable communication patterns among applications. Each rule describes application properties, a communication link among them and a decision to be made. Application properties include permissions, a trust level and (optionally) a name. Two trust levels are distinguished: untrusted (for third party apps) and trusted (for system applications). System apps are trusted not to be malicious, however, they may suffer from confused deputy vulnerabilities and design deficiencies that allow malicious applications to establish indirect communication links. Description of communication links includes the type of communication to be considered (direct or indirect) and may (optionally) include description of data to be transmitted over the channel. Decision has the following options: “accept”, “deny” or “ask the user”

Policy rules are expressed in a policy language which is inspired by VALID [2]. In the following we provide an example of the policy rule to defeat an attack [10]. The attack scenario is like following: A malicious app without INTERNET permission launches a web-browser to download archived and application package files. The following rule prevents applications without INTERNET permission from invoking web-browser to download “.zip” and “.apk” files.

```
Section types:
A,B          : Applications
L            : Communication Link
Section goals:
goal ProtectBrowser.decision(deny) := L.hasSender(A) ∧
L.hasReceiver(B) ∧ L.type(direct) ∧
(L.hasData(*.zip) ∨ L.hasData(*.apk))
A.trustLevel(untrusted) ∧ ¬(A.hasPermision(INTERNET)) ∧
B.trustLevel(trusted) ∧ B.name(Browser) ∧
B.hasPermission(INTERNET)
```

4. EVALUATION

We successfully evaluated our framework against attacks published in [6, 4, 10, 7, 11]. Moreover, we performed automated tests to evaluate performance. The average runtime overhead is 13.126ms in case of an uncached policy decision or 0.105ms in case of a cached one. Moreover, in 97% of all cases cached policy deicions were applied. Our results show that the performance overhead imposed by our architecture is below human perception and the user will not notice any performance delays.

Further, we performed user tests to evaluate our framework against false positives: 20 users tested 50 third-party apps loaded from Android market. Known confused deputy attacks can be prevented without false positives, because they typically feature a well-defined communication pattern (such as a fixed data string to be transmitted to a confused deputy). However, security rules against unknown confused deputy attacks, as well as against attacks by colluding applications may result in small rate of false positives. Our analysis shows that mainly false positives are induced by security rules that aim to prevent attacks by colluding apps that communicate indirectly, over overt/covert channels in Android system components, such as Soundcomber [11].

5. CONCLUSION

We present the design and implementation of a security framework for Android that aims at preventing privilege escalation attacks. The framework analyzes application communication and ensures it complies to a desired system policy. Our framework is efficient, does not rely on developers to perform security checks, can detect known confused deputy attacks without false positives and monitors communication channels in both, middleware and Linux kernel (namely, Binder IPC, Internet sockets and file system). Moreover, it is flexible enough to cover much broader class of privilege escalation attacks, including privilege escalation by colluding applications, however, at the cost of a small rate of false positives. Our framework can prevent recently published privilege escalation attacks [6, 4, 10, 7, 11].

6. REFERENCES

[1] Google Android. http://www.android.com/.

[2] S. Bleikertz and T. Groß. A virtualization assurance language for isolation and deployment. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY)*, 2011.

[3] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. XManDroid: A new Android evolution to mitigate privilege escalation attacks. Technical Report TR-2011-04, Technische Universität Darmstadt, 2011.

[4] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on Android. In *Proceedings of the 13th Information Security Conference (ISC)*, 2010.

[5] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight provenance for smartphone operating systems. In *20th USENIX Security Symposium*, 2011.

[6] W. Enck, M. Ongtang, and P. McDaniel. Mitigating Android software misuse before it happens. Technical Report NAS-TR-0094-2008, Pennsylvania State University, 2008.

[7] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *20th USENIX Security Symposium*, 2011.

[8] Gartner Inc., 2011. http://www.gartner.com/it/page.jsp?id=1689814.

[9] T. Harada, T. Horie, and K. Tanaka. Task Oriented Management Obviates Your Onus on Linux. In *Linux Conference*, 2004.

[10] A. Lineberry, D. L. Richardson, and T. Wyatt. These aren't the permissions you're looking for. BlackHat USA, 2010.

[11] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *18th Annual Network and Distributed System Security Symposium (NDSS)*, pages 17–33, Feb. 2011.