Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android

Alexandre Bartel, Jacques Klein, Yves Le Traon
University of Luxembourg, SnT, Luxembourg
firstName.lastName@uni.lu

Martin Monperrus
University of Lille & INRIA
Lille, France
martin.monperrus@univ-lille1.fr

ABSTRACT

In the permission-based security model (used e.g. in Android and Blackberry), applications can be granted more permissions than they actually need, what we call a "permission gap". Malware can leverage the unused permissions for achieving their malicious goals, for instance using code injection. In this paper, we present an approach to detecting permission gaps using static analysis. Using our tool on a dataset of Android applications, we found out that a non negligible part of applications suffers from permission gaps, i.e. does not use all the permissions they declare.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contracts Validation*

General Terms

Experimentation, Security, Verification

Keywords

Permissions, permission-based software, call-graph, Android, security, Soot, static analysis

1. INTRODUCTION

Android is one of the most widespread mobile operating system in the world accounting 52% market share [10]. More than 300 000 Android applications available on dozens of application markets can be installed by end users. The other side of the coin is that all kinds of malware are waiting to be installed on thousands of Android devices. For instance, Zeus [13] sends banking information to malicious servers. This motivates researchers and engineers to devise security models, architectures and tools that are able to mitigate the malware harmfulness.

The security architecture of Android, the Google Chrome browser extension system and the Blackberry platform, all use a similar security model called the permission-based security model [2]. A permission-based security model can be loosely defined as a model in which 1) each application is associated with a set of permissions that allows accessing certain resources; 2) permissions are explicitly accepted by users during the installation process and 3) permissions are checked at runtime when resources are requested.

This permission model entails intrinsic risks. For instance, not all users may be able to cleverly reject powerful permissions at installation time. Malwares may also use platform vulnerabilities to circumvent runtime permission checks. Finally, applications can be granted more permissions than they actually need, what we call a "permission gap". Malwares can leverage the unused permissions for achieving their malicious goals and have many ways to do so, for instance using code injection or return-oriented programming [4]. Identifying permission gaps means reducing the risks for an application to be compromised, also known as reducing the application attack surface [16].

Permission gaps appear because the process of declaring application permissions is manual and error-prone: Android framework developers manually document which permissions are required for each system resource, and Android application developers manually declare the permissions they *think* are needed. This paper presents an approach to support those *manual* software engineering activities with an *automated* tool. This approach secures permission-based software in the sense that it reduces the attack risks (not in the sense that the resulting software is unattackable).

Our tool, called COPES (COrrect PErmissions Set), proceeds as follows. First, using static analysis, it extracts from the Android framework bytecode a table that maps every method of the API to a set of permissions the method needs to be executed properly. Second, COPES lists all framework methods used by an application, based on static analysis of the application bytecode. Third, COPES computes the set of permissions that are required for the application to run, which means that all permissions in this set are at least used once in the application, and consequently no permission gap remains. Eventually, COPES computes the permission gap as the difference between the declared permissions and the required permissions. By listing the permission checks per framework method, COPES can also help Android designers to comprehensively document the framework.

To sum up, the contribution of this paper is an approach to identify and fix permission gaps in permission based software. More specifically:

- We present a novel methodology to compute a close ap-

ASE '12, September 3-7, 2012, Essen, Germany

proximation of the required permission set and the permission gap based on static analysis, as opposed to concurrent work that uses testing [7].

• We evaluate our approach on 679 Android applications and we show that 124 of them suffer from a permission gap of one to more extraneous permissions.

The reminder of this paper is organized as follows. In Section 2 we propose generic method for inferring permission gaps. In Section 3 we explain some technical details. Experiments we conducted and results are presented and discussed in Section 4. We present the related work in Section 5. Finally we conclude the paper and discuss open research challenges in Section 6.

2. ANALYZING PERMISSIONS

In this section, we introduce the concept of permission-based software and propose a generic methodology to compute a mapping from code to permissions that are required for an application to run.

Permission-based software. Permission-based software is conceptually divided in three layers: 1) the core platform which is able to access all system resources (e.g. change the network policy), it is generally the operating system; 2) a middleware responsible for providing a clean application programming interface (API) to the OS resources and for checking that applications have the right permissions when they want accessing them; 3) applications built on top of the middleware, which have to explicitly declare the permissions they require. Layers #2 and #3 motivate the generic label "permission-based software". Since the middleware also hides the OS complexity and provides an API, it is sometimes called, as in the case of Android, a framework. Permissions can be checked at different levels in the system. We call high-level permissions the set of permissions that are checked at the framework level. Low-level permissions are permissions that are checked at the operating system level. They are 115 permissions in the Android system, while 8 permissions are checked at a low-level. This shows that the framework is responsible for much of the work related to permissions. Note that if a permission is checked at the operating system level, it is not possible to detect that an application uses it by only analyzing the framework. We distinguish 3 kinds of permissions for an application *app*: 1) a declared permission is in the permission list of *app*. 2) a required permissionis associated with a resource that *app* uses at least once. 3) an inferred permission is found by an analysis to be required by actually *app*.

Problem with permissions. When developers write manifests, they write declared permission list by trying to guess required permissions based on documentation and trial-and-errors. In this paper, we propose to automatically infer a permission list in order to avoid this manual and error-prone activity.

Inferring Permissions. Let *app* be an application. The *access vector* for *app* is a boolean vector AV_{app} representing the entry points[1] of the framework usable from *app*. Thus, the length of vector AV is the number of entry points of the framework. An element of the vector is set to *true* if the corresponding entry point is called by the application. We define the *permission access matrix* M as a boolean matrix which represents the relation between entry points of the framework and permissions. Rows represent entry points of the framework and columns represent permissions. A cell $M_{i,j}$ is set to *true* if the corresponding entry point (at row i) accesses a resource protected by the permission represented by column j. Let *app* and $\mathcal{F}$ be an application and a framework respectively. The inferred permissions vector, IP_{app}, is a boolean vector representing the set of inferred permissions for application *app*. A cell $IP_{app}(k)$ is set to *true* if the permission at index k is required by *app*. Otherwise it is set to *false*. We have $IP_{app} = AV_{app} \times M$ by using the boolean operators AND and OR instead of arithmetic multiplication and addition in the matrix calculus.

Extraction of M and AV. Our idea is to perform a static analysis of the framework to extract M and AV. To extract M, we compute a call graph for every entry point of the framework and then detect whether or not permission checks are present in the call graph. To extract AV, we search for occurrences of calls to entry points in the application code.

Computing the Permission Gap. The inferred permission list is computed as explained above. The permission gap is the difference between the permissions extracted from IP_{app} and the declared permissions $P_d(app)$. For instance If the application declares $\{p_1, p_2\}$ and $IP_{app} = \{p_1\}$, then the permission gap is $\{p_2\}$.

3. STATIC ANALYSIS FOR ANDROID

Our approach to detecting permission gaps presented in Section 2 is implemented with two tools. One extracts from a permission-based framework a binary matrix that maps framework methods to permissions, we call it the *mapper*. The other extracts from application code the list of framework methods used, we call it the *sniffer*. In COPES, both tools are based on static analysis using the Soot analysis toolkit.

The Mapper COPES' Mapper uses the Soot call graph analysis Spark [15]. We run Spark in context-insensitive, path-insensitive, flow-insensitive, field-sensitive mode to generate the call graph. In context-insensitive mode, every call to a same method are merged to a single edge independently of the context (receiver and parameters values). A path-insensitive analysis ignores conditional branching hence takes into account all paths of method bodies. The call graph construction is flow-insensitive since it does not consider the order of executions of instructions. It is also field-sensitive because it uses and propagates initialization data (e.g. constructor calls) to reduce the number of egdes.

Spark requires an entry point (usually a main) in order to apply its edge removal techniques. In the case of an API (such as the Android API), there is no "main". Hence, we build one call graph per public method of the Android API by creating one fake main method per public class of the framework (for Android, `android.*` and `com.android.*`). We can also build an artificial main calling all public methods, which is conceptually equivalent yet less scalable[2].

The Sniffer An Android application has no main as well. Hence, the sniffer does not need to construct a call graph. As a result, the Sniffer simply searches for all static occurrences of methods of the Android framework into the application bytecode.

[1] An *entry point* of a framework is a method that an application can use (e.g. public or documented)

[2] we were not able to extract such a call graph on a machine with 24GB RAM

Handling Services A "vanilla" mapper does not work since some calls to the framework are invisible to the static analysis because runtime binding are used within Android services. An Android servive is a component defined by an application to communicate within the application or with other components of the system. Applications also communicate with the operating systems using a special kind of services called *system services*where the system enforces permission checks for protected resources. The permission checks associated to services are mostly implemented in Java, but Android also checks permissions elsewhere (e.g. in C++ services).In this paper, we focus on the former (Java services). Applications synchronously communicate with others services (deployed from other applications or the OS) through a mechanism called *Binder*. When communicating with another service, the first step is to dynamically get a reference to the service by calling `Context.getSystem Service()`; and then a method is called on the reference. The main problem is that a default call graph does not see those runtime references.

Our solution is as follows. Since the binding uses a lookup table that is instantiated once at boot time within the *system server*, we intercept this lookup table and use it in a Soot plugin to redirect every proxy call to the concrete instance of the class which implements the service. In other terms, we feed the call graph engine with this domain specific information that it does not know from code. Note that when using a field-sensitive (such as Spark) or context-sensitive analysis, services must be properly initialized. Otherwise, their fields would point to null and method calls on those fields would not be considered during the call graph construction. We resolve this issue by providing a special initialization class to Spark containing services objects with proper initialization.

Our analysis considers other important technical details described in [3].

4. EVALUATION

Extracted Permission Maps. In the Android v2.2 framework, 115 permissions are defined. As already said, our static analysis method does not deal with: 8 low-level kernel permissions; 30 permissions checked at the level C++ services; 8 permissions checked at the level of content providers. Removing these permissions from the initial set of 115 permissions and by taking care of overlapping permissions (for instance, a permission can be checked at both C++ service and content provider levels) yields a set of *71 high-level permissions*. In the following, our discussion and comparison will only consider this set of 71 permissions. We ran the Mapper described in Section 3. As a result, on the 126660 entry points (methods of the Android framework), our tool infers that 112824 methods have no permission checks, whereas 9562 methods have at least one permission checks (with a median of 2 permission checks and a maximum of 50 permission checks). The total number of permission checks is 137408. In terms of CPU cost, the computation of the permission map M is performed in about 11 hours on a Desktop Dell dual quad-core 2.4GHz with 24 Gio RAM.

Permission Gaps in Real Applications. We ran our tool on a dataset of Android applications from the official Android Market. We consider the top 50 downloaded applications of all 34 top-level categories of the Android Market, as well as the top 500 of all applications and the top 500 of new applications (on February, 23^{rd} 2012). As a result, after deduplicating the applications that appear in several rankings, the dataset contains 2057 applications. For sake of soundness, we discard 1378 applications using reflection and/or class loading. On the 679 remaining applications, 124 are declaring one or more permissions which they do not use. In all, among applications suffering from a permission gap, 64.5% have an attack surface of 1 permission, 23.4% have an attack surface of 2 permissions, 12.1% of 3 or more permissions. We manually ran a sample of them without the extraneous permission to verify the correctness of our analysis (i.e. whether the extraneous permission is actually not used at runtime).

5. RELATED WORK

We have presented an approach to reduce the attack surface of permission-based software. The concept of "attack surface" was introduced by Manadhata and colleagues [16], it describes all manners *in which an adversary can enter the system and potentially cause damage*. This paper describes a method to identify the attack surface of Android applications, which is a important research challenge given the sheer popularity of the Android platform. In the context of Android, reducing the attack surface is adhering to the principle of least privileges introduced by Saltzer [19].

While the Android permission model is different from the one implemented in Java, the following pieces of research present related and relevant points to put our contribution in perspective. Koved and al. described a new static analysis [14] to generate a permission list for a Java2 program (in the Java permission model). An improved methodology was presented by Geay et al. [11]. We also use static analysis but in the context of Android which differs from a Java environment especially with respect to the binder mechanism linking Android API to services. As shown in our evaluation, the binder prevents off-the-shelf Java static analysis tools to resolve remote call to a service. Related to role-based access control, Pistoia et al. [18] formally model RBAC and statically check the consistency of a JavaEE based RBAC system. We check that permission lists of Android applications respect the principle of least priviledge. The concepts are the same (Android permissions could be approximated to roles, and we check which roles are needed at every point of the Android framework) but the target systems are not. We use a similar approach for solving the Binder problem as they do for solving the remote method invocation problem. A major difference though is that in the case of Android system services and context must be initialized beforehand to simulate a correct system state.

The Android security model has been described as much in the gray literature [6] as in the official documentation [1]. Different kinds of issues have been studied such as social engineering attacks [13], collusion attacks [17], privacy leaks [12] and privilege escalation attacks [9, 4]. In contrast, this paper does not describe a particular weakness but rather a software engineering approach to reduce potential vulnerabilities. Different authors empirically explored the usage of the Android model. For instance, Barrera et al. [2] presented an empirical study on how permissions are used. Other empirical studies include Felt's one [8] on the effectiveness of the permission model. Enck et al [5] presented an approach to detect dangerous permissions and malicious permission groups. They devised a language to express rules

which are expressed by security experts. Rules that do not hold at installation time indicate a potential security problem hence a high attack surface. Our goal is different, we don't aim at identifying risks identified by experts, but to identify the gap between the application's permission specification and the actual usage of platform resources and services. Contrary to [5], our approach is fully automated and does not involve an expert in the process. Finally, Felt et al. [7] concurrently worked on the same topic as this paper. They published a very first version of the map between developer's resources (e.g. API calls) and permissions. Interestingly, we took two completely different approaches to identify the map: while they use testing, we use static analysis. As a result, our work validates most of their results although we found several discrepancies [3]. But the key difference is that our approach is fully automated while theirs requires manually providing testing "seeds" (such as input values). However, in the presence of reflection, their approach works better if the tests are appropriate. Hence, we consider that both approaches are complementary, both at the conceptual level for permission-based architectures, and concretely for reverse-engineering and documenting Android permissions.

6. CONCLUSIONS AND PERSPECTIVES

In this paper, we have presented a generic approach to reduce the attack surface of permission-based software. The approach has been fully implemented for Android, a permission-based platform for mobile devices. Our prototype implementation is able to automatically find 9562 Android framework entry points which check permissions. In a permission-based framework, all those checks have to be documented, hence our approach does a significant job in achieving this task in a systematic manner. For end-user applications, our evaluation revealed that 124 applications from the Android Market indeed suffer from permission gaps.

The security architecture of permission based software in general and Android in particular is complex. In this paper, we abstracted over several characteristics of the platform such as low-level permissions. We are now working on a modular approach that would be able to analyze native code and bytecode in concert and to combine the permission information from both. Furthermore, we are exploring how to express permission enforcement as a cross cutting concern, in order to automatically add or remove permission enforcement points at the level of application or the framework, according to a security specification.

7. ACKNOWLEDGMENTS

This research is supported by the National Research Fund, Luxembourg. We also would like to thank Eric Bodden for his help in using the Soot analysis toolkit.

8. REFERENCES

[1] The android developer's guide, last-accessed: 2011-09. `http://developer.android.com/guide/index.html`.

[2] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *ACM Conference on Computer and Communications Security (CCS 2010)*, pages 73–84, Chicago, Illinois, USA, October 4-8, 2010.

[3] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android. Report ISBN: 978-2-87971-107-2, University of Luxembourg, Mar. 2012.

[4] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, 2011.

[5] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM CCS*, pages 235–245, New York, NY, USA, 2009.

[6] W. Enck, M. Ongtang, and P. McDaniel. Understanding android security. *IEEE Security and Privacy*, 2009.

[7] A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM CCS 2011*.

[8] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7, Berkeley, CA, USA, 2011. USENIX Association.

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

[10] Gartner.com. Gartner says sales of mobile devices grew 5.6 percent in third quarter of 2011; smartphone sales increased 42 percent. `http://goo.gl/HkyA4`, Last accessed: March 2 2012.

[11] E. Geay, M. Pistoia, T. Tateishi, B. G. Ryder, and J. Dolby. Modular string-sensitive permission analysis with demand-driven precision. In *ICSE*, pages 177–187. IEEE, 2009.

[12] C. Gibler, J. Crussel, J. Erickson, and H. Chen. Androidleaks detecting privacy leaks in android applications. Technical report, UC Davis, 2011.

[13] S. Hoffman. Zeus banking trojan variant attacks android smartphones. *CRN*, 2011. `http://goo.gl/xAEGr`.

[14] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. *ACM SIGPLAN Notices*, 37(11):359–372, Nov. 2002.

[15] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *12th International Conference on Compiler Construction*, 2003.

[16] P. Manadhata and J. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371 –386, may-june 2011.

[17] C. Marforio, A. Francillon, and S. Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, April 2011.

[18] M. Pistoia, S. J. Fink, R. J. Flynn, and E. Yahav. When role models have flaws: Static validation of enterprise security policies. In *ICSE*, 2007.

[19] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, 1975.