An Android Security Case Study with Bauhaus

Bernhard J. Berger, Michaela Bunke, and Karsten Sohr
Center for Computing Technologies (TZI), Universität Bremen, Germany
{berber|mbunke|sohr}@tzi.de

***Abstract*—Software security has made great progress; code analysis tools are widely-used in industry for detecting common implementation-level security bugs. However, given the fact that we must deal with legacy code we plead to employ the techniques long been developed in the research area of program comprehension for software security. In cooperation with a security expert, we carried out a case study with the mobile phone platform Android, and employed the reverse engineering tool-suite Bauhaus for this security assessment. During the investigation we found some inconsistencies in the implementation of the Android security concepts. Based on the lessons learned from the case study, we propose several research topics in the area of reverse engineering that would support a security analyst during security assessments.**

***Index Terms*—program comprehension, security assessment, software security, Android**

I. Introduction

Over the last years, static code analysis for security has made great progress. Commercial available tools are employed by software vendors to detect implementation-level security bugs, such as buffer overflows and injections vulnerabilities. Certainly, employing these tools is only the first step towards secure software as it is restricted to common bug classes.

More seriously are design-level flaws, since literature states that the later a change must be applied to the design of an application, the more costs will arise [1]. Methods such as Microsoft's threat modeling [2] or the architectural risk analysis proposed by McGraw [3] should help to discover security problems already during the software-design phase. In academia, more formal approaches to dealing with software security have been established, notably, language-based security [4], model-driven security [5], and stepwise refinement [6]. Although these approaches are promising, they can just be applied when software is developed from scratch, which is rarely the case.

Owing to the fact that our work is focused more on the aspect of program understanding we expect that program comprehension tools can help a security analyst detecting security flaws in code, such as divergences between documentation and code. We imply that none of such flaws can be detected in a fully automated way, and a security analyst must assess the situation by her own. What we yet can expect is that the tool helps us to assess the risks of the software and, for example, pinpoints security-critical areas of the code. Therefore, we chose a reverse-engineering tool-suite, called Bauhaus [7], to analyse a well-known security aspect, in the open-source software system Android. We compared the implementation of permission enforcement to the official Android documentation and discovered a divergence.

The rest of the paper is structured as follows. In Section II, we give an overview of related research. Thereafter, our motivation for the conducted case study is given in Section III, followed by a brief description of Android concepts. In Section V, we describe the case study focusing on a security assessment of permission mechanisms. Based on the experience gained by the case study, we discuss further research problems in Section VI. Afterwards, we conclude and give an outlook in Section VII.

II. Related Work

Software security is an emerging research area with a strong practical impact. For example, we have static analysis tools, that focus on common implementation-level bugs which are mostly related to improper input validation [8]–[11]. Nevertheless, these tools do not help one to understand the security aspects.

To our knowledge, only a few works deal with reverse engineering the security architecture out of code. Karppinen et al. use the Software Architecture Visualization and Evaluation tool to detect a security back door— they completely removed the security check —that they added for the case study. To detect the back door they used static as well as dynamic information and compared the resulting information with the results of the correct implementation [12].

Mancoridis reports about common bug classes and names techniques a software maintenance-engineer can use to improve the security of a software. Moreover, he emphasizes several techniques that must be developed to tackle these problems properly. He stresses that it is necessary to develop formal notations and tools to allow the specification of software-security architectures. Mancoridis assumes that the developer has the security architecture of her software in her mind, what is not necessarily the case [13].

Sometimes literature on static code analysis treats program comprehension as a side topic, for example, Chess and West briefly mention program comprehension tools such as CAST [14] and Fujaba [15] in their book on static analysis for software security, but do not give further details of how they might help one to address the problem of software security [16].

III. Motivation for the Case Study

It is expected that Android will become one of the major mobile phone platforms in the future [17] and is used for other devices as well. As it gains a lot of market share and is open source, it is an interesting target for security analyses. For this reason, we started a security assessment project. During the

analysis, we faced several challenges, mostly related to the lacking documentation of Android's security concepts and the complexity of the code.

We started our review of the Android platform with the assumption that not only the Linux kernel is security-critical, but also the Android middleware (the Android framework classes). For example, the permission enforcement and the reference monitor, which mediates the access to Android components, is implemented within the Java-based middleware, although the kernel is accessed to retrieve data for security decisions. We aimed to explore and understand the implementation of Android's security mechanisms.

Due to the fact that the structure of the code and specifically the software architecture are unknown to us at the beginning, we used a reverse engineering tool-suite called Bauhaus, to gain a better insight into the code. Other tools such as Fujaba or CAST could have been used, too. The reason for employing Bauhaus was that the tool is available at our institute and hence had experience with it. Generally, tools for program comprehension contain functionality to represent information about the program, which can be gathered statically as well as dynamically. With the help of these tools, one can obtain information on the components, modules, classes, methods, and member variables, as well as relationships between these elements, such as call relations or member accesses.

We focus our analyzes on permission checking and enforcement because access control is a basic security concept for IT systems and applications, going back to Lampson's access control matrix [18]. Further literature on authorization can be found in standard works on computer security (see [19]).

IV. The Android Concepts

We first describe Android's main concepts, before presenting the challenges related to analyzing the platform with respect to security. Note that there does not exist a comprehensive document on Android's security concepts. The information is scattered throughout the Android developer's website.

1) Android Components: An Android application consists of different parts, called components, having, according to its task, one of four basic component types. *Activities* are the presentation layer of an application, allowing a user to interact with the application. *Services* represent background processes without a user interface. *Content providers* are data stores that allow developers to share databases across application boundaries. Finally, *broadcast receivers* are components that receive and react to broadcast messages, for example, the Android OS itself sends such a broadcast message if the battery is low. Each component of an application runs as a separate task, making an Android device to a large distributed system, even if all processes are running on the same device.

2) Inter-Process Communication: The Android platform supports inter-process communication (IPC) for communication between components [20]. One foundation for this IPC is the Binder, an Android-specific kernel device that allows efficient but safe communication.

A way to communicate with components not known at the development time, are messages, which may include arbitrary data, called intents. An *intent* is an abstract description of an operation to be performed on the platform [21]. For example, an intent can start a new activity or service, or communicate with background services. An advantage of this technique is that a client application is no longer linked to a specific program, but can access any possible service for the specified need.

3) Android Security Mechanisms: Android has two basic methods of security enforcement. Firstly, applications run as Linux processes with their own user IDs and thus are separated from each other. This way, a vulnerability in one application does not affect other applications. In contrast to Java, the virtual machine is not a security barrier because the Linux kernel takes over the task of separating processes.

Since Android provides IPC mechanisms, which need to be secured, a second enforcement mechanism comes into play. Android implements a reference monitor to mediate access to application components based on permission labels. If an application intends to access another component, the end user must grant the appropriate permissions at installation time. Furthermore, the security model has several refinements that increase the model's complexity [20].

V. Android Case Study

We conducted our security assessment along with a security expert, the third author, to understand the implementation of some aspects of the Android security mechanism with the help of the Bauhaus tool-suite.

1) Issues: Starting from the documentation, we planned to focus on several aspects related to the implementation of permission checking. The process of permission enforcement consists of different steps, which we aimed to understand. After reading the security documentation, available at the Android website, we picked the Bluetooth API for further investigation. Other functionality of the platform could have been analyzed in a similar way.

The Bluetooth API documentation states that an application needs at least the `BLUETOOTH` permission to use the Bluetooth device. If a program wants to administer the Bluetooth device, it needs the `BLUETOOTH_ADMIN` permission in addition. It is explicitly noted that one needs the former permission, to use the latter [22].

During the security assessment we wanted to answer several questions the security expert had. We will discuss these in the following:

Question 1 Where are permissions enforced within the Bluetooth API (enforcement points)?

Question 2 Which permissions are enforced within the Bluetooth API (access control policy)?

Question 3 Can we check whether an application needs the `BLUETOOTH` permission, in order to use the `BLUETOOTH_ADMIN` permission?

Question 4 Can we identify the permission enforcement bundled with the IPC mechanism, described in the `Mix'n'Match` pattern [23]?

2) Policy Enforcement Points: In the beginning, the security expert wanted to know where permissions are enforced during the usage of a service. Therefore, we extracted in an interactive step all public methods belonging to the Bluetooth service and their relations to the method `enforceCallingOrSelfPermission`. It soon became clear that all public methods implement an enforcement point to make sure that the calling process has obtained certain permissions.

The security expert extracted the information that all public methods are protected against unauthorized usage. This knowledge leads to the next question, namely, which permissions are enforced within the identified enforcement points.

3) Access Control Policy: The security expert was also interested in a view that shows the specific permissions that are enforced, which would be a visualization of the access control policy. To extract such a view, we had to gather additional information from the source code because the actual representation did not contain sufficient details. The relevant information for the desired view, the permission, which will be enforced, is the first actual parameter passed to the method `enforceCallingOrSelfPermission`. In general, it is not always possible to determine the actual value of a parameter of a method. In our case, it was not necessary to apply expensive static analyses because of the constant propagation taken place in the compiler. With these additional data, we enhanced the data generated by our front end and added new nodes for the permissions, which are checked, and new edges between methods and the required permissions.

The security expert, conducting the security assessment, can now see the implemented access control policy of the Bluetooth service for the first time. On the basis of the generated view, he was able to identify those methods of the service belonging to the administrative part of the API, information that is not documented explicitly.

4) Correctness of Documentation: Having extracted the implemented access control policy of the Bluetooth service, the next question to be answered was, whether the condition mentioned in the security documentation was up-to-date, i.e., we wanted to check whether all methods, enforcing the `BLUETOOTH_ADMIN` permission, require the `BLUETOOTH` permission, too.

This check could be done manually. To automate this process, however, we wrote a small (Python) script that searches for the enforcement calls automatically. Since our representation is a graph, we can simply search for methods that are connected with the `BLUETOOTH_ADMIN` permission but are not connected with the `BLUETOOTH` permission. In total, we identified 14 methods that check the `BLUETOOTH_ADMIN` permission, but none of them seems to check the `BLUETOOTH` permission. To better understand this situation, we adapted the script to regard method calls as well as the permissions that are checked within the methods directly or indirectly (via called methods). We found that four methods check the `BLUETOOTH` permission indirectly and the remaining ten methods do not check for both permissions at all.

The findings were quite surprising for the security expert since the documentation stated that a calling process must have both permissions. Certainly, the fact that not both permissions are checked does not necessarily lead to a vulnerability, but a developer may develop her code with wrong assumptions in her mind after reading the documentation. Furthermore, it is necessary to review those cases where the `BLUETOOTH` permission is only checked indirectly if this can lead to an inconsistent state of the Bluetooth service if a calling process only has the `BLUETOOTH_ADMIN` permission.

5) Mix'n'Match Pattern: Another open question was to check whether the IPC mechanism is tied with an enforcement point for intents. First of all, our security expert wanted to identify the `Runtime Mix'n'Match` pattern [23] which is said to model the intent mechanism and to be used in the Android framework. Moreover, he wanted to find clues about the enforcement points in the architecture as they are described in the policy enforcement security pattern [24]. This policy pattern describes a specialized IPC mechanism to dynamically bind different actors like activities and services. In addition, it is said to enforce permissions "to prevent applications from launching activities of other applications" [25].

We analyzed a small application, which opens a *WebView* activity showing an assigned URL. According to the pattern description, the concept consists of five parts: `Intents`, `IntentHandler`, `IntentResponder`, `IntentFilter`, and `Client` [23]. To start a new activity, the `Client` writes the information for the activity into the intent. This intent is sent to the `IntentHandler`. It routes the requirements to the `IntentFilter`, which knows which requirements can be handled by which activity or service. To generate this knowledge base, any activity and service must define an `IntentResponder`. `IntentFilter` prepares a list of fitting activities and services and returns them to the `IntentHandler`. It chooses one appropriate activity or service if there exists more than one. This intent becomes initialized and is sent back to the requesting `Client`.

As described in the previous sections, the communication between components in the Android framework is secured with enforcement points, so we wanted to detect which code components join the Intent mechanism *and* enforcement points. We used the *hierarchical reflexion method* [26] to model a hypothetical pattern structure and tried to search for it iteratively by mapping parts of the code base to this structure.

The result determines convergences and differences among the architecture and the implementation model. It showed that the runtime `Mix'n'Match` pattern could be successfully identified by our security expert. Unfortunately, he did not encounter any enforcement point pattern, which ought to be connected closely to this IPC mechanism, but gained an overview of how the components interact in the IPC mechanism. We assume that this binding cannot be detected with static analysis or is bundled with other patterns which partly cover these security patterns. Furthermore, other problems concerning security patterns were identified and must be solved prior to a mature recognition process [27].

6) Conclusions of the Case Study: In summary, we conclude that the Bauhaus tool helped us during the review process. The results of our review are:

Result 1 We identified all enforcement points within the Bluetooth service and found out that all public methods were protected adequately.

Result 2 To identify the enforced permissions, we had to adapt our front end to extract additional information. Afterwards, we were able to visualize the access control policy.

Result 3 The statement that one needs the `BLUETOOTH` permission in order to use the `BLUETOOTH_ADMIN` could not be supported by our investigation since we found counterexamples.

Result 4 We were able to detect the IPC mechanism pattern, but not the permission enforcement. We conclude that the security documentation is imprecise at that point or we were limited by the chosen comprehension technique.

VI. Software-Security Comprehension

In the preceding section, we showed that program-comprehension and reverse-engineering techniques can be used in the area of software security. Now, we discuss research topics that need to be investigated more deeply, to develop useful techniques and tools for a security evaluator. For our more general discussion, we also consider experience gained in a research project called ASKS, which is currently being carried out with enterprises that made available their business applications, which are implemented using the Java platform, Enterprise Edition technology [28], for a security analysis.

One conclusion that we drew from our security review is that it is necessary to create more formal architectural security views (see also Mancoridis' statement [13]). These views need to be language- and platform-independent in order to be a common language to communicate with security experts who are not necessarily experts for the programming language. With the help of these views, it is easier to understand the security architecture of an application or even of a distributed system. In the following, we discuss some further ideas of how these views can be created and what security aspects may be of interest for such views.

A. Possible Architectural Views

There are many software aspects related to security. In companion with our security expert, we identified some aspects that are suitable to be extracted from source and be useful for a security specialist.

1) Visualization of Trust Zones: It is helpful to group the identified software parts into trust zones [29] based on the criticality of the data/components accessed. With the help of such a view, one can conduct a security-related impact analysis of changes and identified bugs, to balance out the improvements against the threats.

2) Visualization of Attack Surfaces: Beyond the decomposition of the code base into different zones, it is helpful to add information about the boundaries of components (architectural components or whole processes). Therefore, it is necessary to identify framework means that allow communications between processes. By means of this knowledge, it is possible to identify data sources and sinks. In combination with a dependence graph [30], it would be feasible to estimate the attack impact.

3) Access Control Policy: In Section V-3, we described how to extract parts of the access control policy of Android's Bluetooth service. Since access control is crucial to many platforms and applications, we can apply the task of extracting the access control policy on other platforms. For example, we extracted the access control policy of a Java enterprise application and compared that policy with the documentation employing the reflexion analysis [31].

B. Towards Automatic Extraction of Architectural Views

The aforementioned views must be created with the help of techniques already known in the reverse-engineering community, but that need to be tailored towards the specific security needs to give reasonable results.

1) Abstraction: From our point of view, it is inevitable to introduce graphical abstractions beyond the known visualizations, such as UML-diagrams and implementation-level dependence graphs, to make security comprehension easier.

The abstraction of constructs in the software which are imposed by the framework, such as IPC mechanisms and Java Beans, would help one to concentrate on the essential parts of the application. Furthermore, it is common in current frameworks that parts of the implementation are generated automatically. During an assessment a reviewer must analyze the generated parts to "understand" the whole application and he cannot differentiate between handwritten and generated source code. These technical entities hide the real intent behind the code. Therefore, it is necessary to remove these details and replace them with a presentation which is more meaningful to a security analyst.

2) Component Detection: A slightly different kind of abstraction is the process of component detection and aggregation, to allow a developer to build up a mental map of the system more easily. This is useful if the architectural components are spread over several classes and packages. The ideal case would be a supportive mechanism to restructure the application's representation semi-automatically such that it resembles the existing architecture, specified by domain experts.

Within the reverse-engineering and program-comprehension community, there already exists experience with various clustering techniques to extract components automatically from code [32]. The components that are of interest for detection are mostly domain- and implementation-specific, as well as the aforementioned abstractions we must introduce. Therefore, it is a necessity to involve framework experts to achieve reasonable results.

3) Security Pattern Detection: Often, security features are integrated into the software architecture by common and well-known aspects like enforcement points. Some of these aspects can be merged to security patterns which have the goal to harden software against attacks and misuse [33].

Existing design-pattern detection approaches, however, can

only detect a few of the common design patterns [34]. Presently, none of them supports the detection of security patterns, although ensuring security is a significant task [35].

Due to the fact that not everybody reengineering a system has appropriate security knowledge automated approaches of detection are desirable. When a security pattern has been detected, it can be highlighted in a software-architecture representation. Such visualized security aspects can support hardening software before it will be released or used by different user groups to post-check a software system.

VII. Summary

We conducted a case study focusing on permission checking in the Android framework and showed that the Bauhaus tool-suite can support a security expert during a security assessment. We were able to enhance our understanding of the Android framework, in particular, a divergence between the documentation of the Bluetooth API and the framework implementation has been found. Moreover, the comprehension of the IPC mechanism for intents and the unexpected missing of permission checks were other results of the case study. Based on our experience, we discussed new challenges and research problems for program comprehension in security assessments.

Further research must be carried out to apply the techniques of program comprehension to the field of software security. Our impression is that neither the security-research community discusses this topic adequately nor is industry making use of such techniques to better understand the security status of their software. Using state-of-the-art tools for finding security bugs cannot reveal logical security problems such as undesirable interactions between components.

With the increasing complexity of software, software companies need to understand the security risks of their code, and tools employing program comprehension functionality will support them with this challenging task. We truly believe that "software-security comprehension" will be a fruitful research topic for the future with also a broad practical impact.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) under the grant 01IS10015B (ASKS project).

References

[1] R. Pressman, *Software Engineering – A Practioner's Approach*, 4th ed. McGraw-Hill, 1997.

[2] F. Swiderski and W. Snyder, *Threat Modeling*. Microsoft Press, 2004.

[3] G. McGraw, *Software Security: Building Security In*. Addison-Wesley, 2006.

[4] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, jan. 2003.

[5] M. Clavel, V. Silva, C. Braga, and M. Egea, "Model-driven security in practice: An industrial experience," in *Proc. of the 4th European Conf. on Model Driven Architecture: Foundations and Applications*. Berlin, Heidelberg: Springer, 2008.

[6] H. Mantel, "Preserving information flow properties under refinement," in *IEEE Symposium on Security and Privacy*, 2001.

[7] A. Raza, G. Vogel, and E. Plödereder, "Bauhaus – A tool suite for program analysis and reverse engineering," in *Ada-Europe*, ser. LNCS, vol. 4006. Springer, 2006.

[8] V. B. Livshits and M. S. Lam, "Finding security vulnerabilities in java applications with static analysis," in *Proc. of th 14th USENIX Security Symposium*, August 2005.

[9] B. Chess, "Improving computer security using extended static checking," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002.

[10] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proc. of the 2007 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007.

[11] C. Nagy and S. Mancoridis, "Static security analysis based on input-related software faults," in *Proc. of European Conf. on Software Maintenance and Reengineering*. IEEE Computer Society, 2009.

[12] K. Karppinen, M. Lindvall, and L. Yonkwa, "Detecting security vulnerabilities with software architecture analysis tools," in *IEEE Intl. Conf. on Software Testing Verification and Validation Workshop*, vol. 1. Los Alamitos, CA, USA: IEEE Computer Society, 2008.

[13] S. Mancoridis, "Software analysis for security," in *Frontiers of Software Maintenance, 2008*, oct. 2008.

[14] CAST, 2011. [Online]. Available: http://www.castsoftware.com

[15] Universität Paderborn, 2011. [Online]. Available: http://www.fujaba.de/

[16] B. Chess and J. West, *Secure programming with static analysis*, 1st ed. Addison-Wesley Professional, 2007.

[17] R. Cozza, C. Milanesi, and A. Gupta, "Competitive landscape: Mobile devices, worldwide, 3q10," Gartner, Inc., Tech. Rep., 2011. [Online]. Available: http://www.gartner.com/it/page.jsp?id=1466313

[18] B. W. Lampson, "Protection," *ACM*, vol. 8, no. 1, jan 1974.

[19] R. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*, 2nd ed. Wiley, 2008.

[20] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android security," *IEEE Security Privacy*, vol. 7, 2009.

[21] Google Inc., "Android Development - Intent." [Online]. Available: http://developer.android.com/reference/android/content/Intent.html

[22] ——, "Android - Bluetooth." [Online]. Available: http://developer.android.com/guide/topics/wireless/bluetooth.html

[23] P. G. Austrem, "Runtime mix'n and match design pattern," in *Proc. of the 15th Pattern Languages of Programs Conf.* New York, NY, USA: ACM, 2008.

[24] Y. Zhou, Q. Zhao, and M. Perry, "Policy enforcement pattern," in *Proc. of the 7th Pattern Languages of Programs Conf.*, 2002.

[25] Google Inc., "Android - Security." [Online]. Available: http://developer.android.com/guide/topics/security/security.html

[26] R. Koschke and D. Simon, "Hierarchical reflexion models," in *Proc. of Working Conf. on Reverse Engineering*, 2003.

[27] M. Bunke and K. Sohr, "An architecture-centric approach to detecting security patterns in software," in *In Proc. 3rd ESSoS*, ser. LNCS, vol. 6542. Springer, 2011.

[28] Oracle, 2011. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[29] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, and N. R. Mead, *Software Security Engineering: A Guide for Project Managers*, 1st ed. Addison-Wesley Professional, 2008.

[30] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems, TOPLAS*, vol. 12, no. 1, jan. 1990.

[31] K. Sohr and B. Berger, "Towards architecture-centric security analysis of software," in *Engineering Secure Software and Systems*. Springer, 2010.

[32] R. Koschke, "Atomic architectural component recovery for program understanding and evolution," Ph.D. dissertation, Institute of Software Technology, University of Stuttgart, Germany, 1999.

[33] J. Yoder and J. Barcalow, "Architectural patterns for enabling application security," in *Proc. of 4th Pattern Languages of Programs Conf.*, Monticello/IL, 1997.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Object-Oriented Software*. Addison Wesley, 1994.

[35] M. VanHilst and E. B. Fernandez, "Reverse engineering to detect security patterns in code," in *Proc. of 1st Intl. Workshop on Software Patterns and Quality*. Information Processing Society of Japan, December 2007.