Idea: Towards Architecture-Centric Security Analysis of Software

Karsten Sohr and Bernhard Berger

Technologie-Zentrum Informatik, Bremen, Germany,
{sohr|berber}@tzi.de

Abstract. Static security analysis of software has made great progress over the last years. In particular, this applies to the detection of low-level security bugs such as buffer overflows, Cross-Site Scripting and SQL injection vulnerabilities. Complementarily to commercial static code review tools, we present an approach to the static security analysis which is based upon the software architecture using a reverse engineering tool suite called Bauhaus. This allows one to analyze software on a more abstract level, and a more focused analysis is possible, concentrating on software modules regarded as security-critical. In addition, certain security flaws can be detected at the architectural level such as the circumvention of APIs or incomplete enforcement of access control. We discuss our approach in the context of a business application and Android's Java-based middleware.

1 Introduction

More and more technologies find their way into our daily life such as PCs, mobile phones with PC-like functionality, and electronic passports. Enterprises, financial institutes, or government agencies map their (often security-critical) business processes to IT systems. With this dependency of technologies new risks go along which must be adequately addressed. One of the main problems arise from faulty software which led to most of the security incidents reported to CERT [3]. For this reason, a lot of work has been done on statically detecting common low-level security bugs in software such as buffer overflows, simple race conditions or SQL injection vulnerabilities. These efforts led to research prototypes [4, 5, 14] and even to commercial products [11, 7, 18]. In the future, however, it will be expected that attackers will also exploit other software errors such as logical flaws in access control (e.g., inconsistent role-based access control in Web Service applications) or the wrong usage of the security mechanisms provided by software frameworks such as JEE or Spring [17].

Moreover, even if the software architecture has been specified with modeling languages such as UML, it is not clear whether the actual software architecture manifesting in the source code is synchronized with the specified (intended) architecture. As software is often changed in an ad hoc fashion due to the customers demands, the software architecture is steadily eroded. In particular, this

problem also applies to the security aspects of the software architecture. For example, if access to a security-critical resource ought to be protected by an API which enforces access control decisions, but an application for reasons of convenience can directly call the functionality in question without using this API, then the application's security can be easily subverted.

As a consequence, a methodology (including supporting tools) is desirable which allows one to analyze software on a more abstract level than the detailed source code. This way, certain kinds of logical security flaws in the software architecture can be detected as those mentioned above. In addition, a more abstract view on the software allows the security analyst to focus her analysis with already available tools more on the relevant software parts, reducing the rate of the false positives. For example, internal functionality, which is only called by a limited number of users, need not be analyzed to such an extent as modules exposed on the Internet. Last but not least, analyses at the architectural level can be carried out at design time such that security flaws can be detected at early stages of the software development process. This reduces the costs of resolving the security problems.

In this paper, we sketch a methodology for an architecture-based security analysis of software. The basis of this approach is the Bauhaus tool suite, a general-purpose reverse engineering tool, which has been successfully used in research projects as well as in commercial projects [19, 21]. With the help of the Bauhaus tool, different kinds of analysis can be conducted at the architectural level. First, one can directly analyze the high-level architecture of an application w.r.t. security requirements. Due to the fact that the Bauhaus tool allows one to extract a low-level architecture from the source code, called resource flow graph (RFG), one can also check this architecture against the expected high-level architecture. This way, one can detect security problems such as missing access control checks in security-critical APIs.

Having carried out the analyses at this architectural level, one can switch to an analysis at the detail level, i.e., at the source code. This analysis can then be done with the help of commercially available tools or research tools such as model checkers and SAT solvers or Bauhaus itself.

The remainder of this paper is organized as follows. In Section 2, we describe the main concepts of the Bauhaus tool suite, concentrating on the RFG. We explain the principles of our architecture-based methodology for the security analysis of software in Section 3. Section 4 discusses some early results of our approach in the context of two different case studies, namely, a tutorial application for JEE and the Java-based middleware of the Android platform. Section 5 gives a short overview of related work, whereas Section 6 concludes and discusses possible research directions.

2 The Bauhaus tool suite

The Bauhaus tool suite is a reverse engineering tool which lets one deduce two abstractions from the source code, namely the Intermediate Language (IML) and

the resource flow graph (RFG)[19]. The former representation in essence is an attributed syntax tree (an enhanced AST), which contains the detailed program information such as loop statements, variable definitions and name binding. The latter works on a higher abstraction level and represents architecturally relevant information of the software.

An RFG is a hierarchical graph, which consists of typed Nodes and edges. Nodes represent objects like routines, types, files and components. Relations between these elements are modeled with edges. The information stored in the RFG is structured in views. Each view represents a different aspect of the architecture, e.g., the call graph or the hierarchy of modules. Technically, a view is a subgraph of the RFG. The model of the RFG is fully dynamic and may be modified by the user, i.e., by inserting or deleting node/edge attributes and types. For visualizing the different views of RFGs, Graphical Visualiser (Gravis) has been implemented [8]. The Gravis tool facilitates high-level analysis of the system and provides rich functionality to produce new views by RFG analyses or to manipulate generated views.

3 Security analyses with the help of a RFG

We now discuss different aspects of analyzing software w.r.t. security based upon the RFG. Specifically, we describe how the methods and techniques, which have been well-established in the context of software quality assurance, can be adjusted for security analysis.

Notation of the architecture. Due to the fact that the elements of the RFG can be represented by a meta model, it is possible to define RFG profiles, that are specific to the software frameworks and security mechanisms used in the analyzed application. The elements of the profile have a well-defined semantics, which simplifies the analyses at the architectural level and leads to a better understanding of the security aspects of the architecture. For example, in case of a JEE business application one might have nodes of types such as “role”, “permission”, or “user” and edges of the types “user assignment” and “permission assignment” to express role-based access control (RBAC) [1]. In order to secure the communications, one might have “encrypted RPC channel” or “encrypted SOAP channel” edges.

Recovery of the software architecture. With the help of the reflexion method [15], the software architecture can be reconstructed and documented as a separate view of the RFG. This is done in a semi-automatic and iterative process starting from an abstract to a more detailed architecture description. Usually, this process is carried out in workshops with the software architects and developers.

Often the *mental* architectures, i.e., the architecture each developer / architect has in mind, might differ for the different participants of this process. These *mental* architectures are unified and written down as a hypothesized architecture which is then checked against the implemented low-level architecture. This

way, discrepancies between both architectures, the hypothesized and the implemented one, can be automatically detected. If there are references (edges) in the implemented architecture which are absent in the hypothesized architecture, then we speak of *divergences*. An *absence* is a reference occurring in the hypothesized architecture and not being present in the source code. The architecture is manually refined and enhanced with new knowledge gained from previous steps in an iterative process until the architecture remains stable.

So far, the reflexion analysis was carried out having software quality in general rather than software security in mind. However, this step neatly fits to the "Ambiguity analysis" introduced by McGraw [17] because a different understanding of the software architecture might lead to security holes. This way, a natural application of the reflexion method in the security context is possible. In particular, the RFG representing the architecture can be enhanced with security modeling elements.

This reflexion method is used in 4.1 to find violations of the RBAC policy of the JEE demo system *Duke's Bank*.

Security views. As indicated in Section 2, the Bauhaus tool suite allows one to define views on the software architecture to concentrate on the aspects to be analyzed and hence to carry out a more focused analysis. In a JEE-based business application one might define a view which comprises all remote access or a view on RBAC. In the context of a mobile phone platform, one might define a view for the mechanism which implements the enforcement of permissions for protected resources such as Bluetooth or WiFi.

Further analyses on the RFG. Owing to the fact that applications process data of different sensitivity (security levels), the data flow through the modules of the software must be identified and the communications adequately secured. Those paths through an application's modules and functions should be identified where sensitive data flow without appropriate protection such as encryption or digital signatures. For this to accomplish, we can assign security labels to the data (e.g., member variables in Java or global variables in C) and also to the modules and functions of the application. We can further define which data can be accessed by which module and function, respectively. At the RFG level, we then can check whether the defined access control policy is violated.

Security requirements which the RFG must satisfy itself can be represented as a graph. An example of such a requirement is shown in Figure 1. Here, it is stated that if we have a remote method call on Entity Java Beans (EJBs), then the Java annotation "RolesAllowed" is mandatory. This means that remote method calls are only allowed if (the appropriate) roles are assigned to the caller's principal. One now can search for all matching occurrences of remote method calls on EJBs—the "condition" part in Figure 1—within the RFG and check whether the requirement is fulfilled for all occurrences of the condition. However, note that subgraph problems in general are known to be NP-complete such that heuristics are to be applied [12].

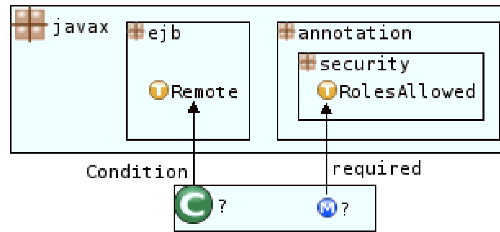


Fig. 1. A security requirement represented as a graph.

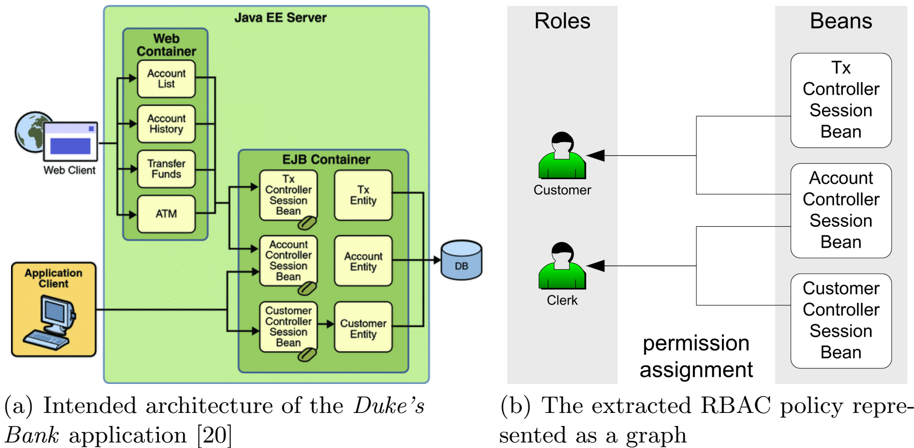
Analyses on the RFG and the detailed program representation. As the Bauhaus tool suite also makes available the detailed program representation in form of the IML, code analyses can be carried out at the source code level. This way, one can start analyzing the software at the more abstract RFG level. Having identified security-relevant locations of the application via the RFG, one can analyze the corresponding source code locations more deeply, i.e., both representations can be used in conjunction for the security analysis. Alternatively, one can employ the RFG to pinpoint security-critical parts of the software and then use commercially available analysis tools to detect low-level security bugs. Research prototypes based on SAT solvers or theorem provers might also be useful in order to check the code against constraints (such as invariants or pre- and postconditions). Two promising examples of such tools are JForge [9] and ESC/Java2 [6].

4 Early Case Studies

We now discuss our architecture-centric security analysis in the context of two case studies. The first one is named "Duke's Bank", a simple application from Sun's JEE tutorial [20]. We chose this application because on the one hand, it is simple, and on the other hand, it has a widely-used architecture for business applications. In order to show that our approach can also be used in the context of embedded systems' software, the second case study is from the mobile phone domain, namely, the Java-based middleware of the Android platform.

4.1 Analysis of a JEE application

Duke's Bank is a demo banking application allowing clerks to administer customer accounts and customers to access their account histories and perform transactions. It is a typical JEE application with a Web-based as well as a rich client interface (see Figure 2(a)). A customer can access information about his account via the Web interface, whereas the rich client interface can only be used by the clerk. The functionality of the application is provided by EJBs (`AccountControllerSessionBean`, `TxControllerSessionBean`, `CustomerControllerSessionBean`). In these EJBs, access to the database containing the



(a) Intended architecture of the *Duke's Bank* application [20]

(b) The extracted RBAC policy represented as a graph

Fig. 2. Description of the applications architecture

account data is encapsulated via the Java Persistence framework.

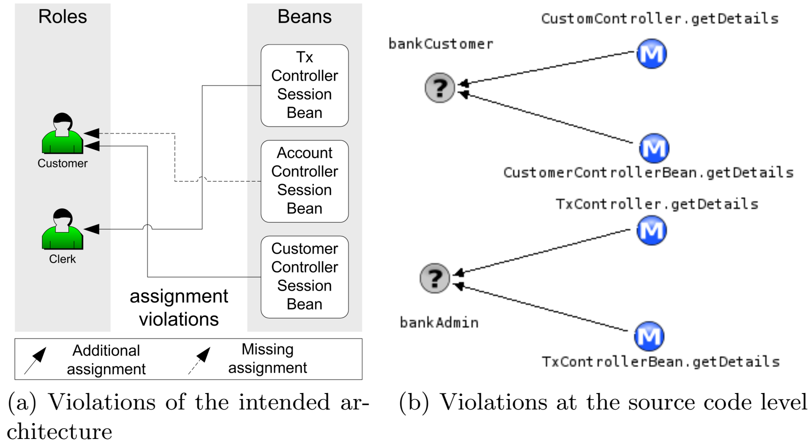
In Figure 2(a), one can see that the `CustomerControllerSessionBean` component cannot be accessed by the Web interface, i.e., the customers have no access to this bean. Figure 2(b) then displays the intended RBAC policy for the JEE application. The nodes represent the roles as well as EJBs and the edges correspond to the permission to access an EJB (permission assignment). Two roles *Clerk* and *Customer* have been defined, and specifically, there is no access from *Customer* to the component `CustomerControllerBean`.

We now briefly describe how the reflexion analysis can be applied to the *Duke's Bank* application with the focus on RBAC for EJBs. In a first step, we loaded the source code of this application into the Bauhaus tool and obtained an RFG as the low-level architecture, which is not given in this paper for reasons of brevity. The RBAC policy displayed in Figure 2(b) can then be regarded as the hypothesized architecture or more precisely as a security view representing the RBAC policy for the *Duke's Bank* application. This architecture is checked against the RFG gained from the source code.

Figure 3(a) shows the results of this reflexion analysis. Notice that the edges with the solid lines are representing divergences. The consequence of this divergence is that the code allows access that ought to be forbidden according to the RBAC policy, i.e., security violations are possible. The graph depicted in Figure 3(b) shows these violations manifesting in the source code's RFG [1]. For example, now every principal (be it a customer or a clerk) can access the method `getDetails()` of the `CustomerControllerSessionBean`. This way, she can query information on all customer data such as account numbers. Clearly,

[1] Note that the roles *Clerk* and *Customer* are mapped to the source code roles *bankAdmin* and *bankCustomer* within the frameworks of the reflexion analysis. Therefore, we have different names in Figure 3(b).

this is only a demo application, but it shows the kinds of problems which our analysis technique can detect.



(a) Violations of the intended architecture

(b) Violations at the source code level

Fig. 3. Analysis results

4.2 Analysis of Android

We also loaded the Android framework classes, implementing the Java-based middleware of Android into the Bauhaus tool suite to gain a better program understanding. Then we constructed an RFG, which can be used to identify the parts implementing the security concepts of the Android middleware such as mandatory access control for inter process communication (IPC), protected APIs, protection levels of applications [10]. This can be done by traversing the RFG and defining views which correspond to the security concepts implemented within the framework classes. The views let the analyst conduct a more focused analysis on the code. After constructing such views, she can switch to source code analysis as indicated above.

Employing our RFG-based analysis technique, we detected a kind of backdoor meant for Android developers, which does not seem to be officially documented: The end user can assign permissions to applications via a system-wide permission file. If there had not been additional security checks put in place, an end user could have given access rights defined by the operator to her own applications.

Beyond this, we can also check at the architectural level whether the permissions on security-critical resources (e.g., sending SMS, Bluetooth or WiFi access) are appropriately enforced. This means we can check on the RFG if the `enforcePermission()` methods are called for the permissions listed in the `Manifest.permission` class.

5 Related Work

There exist a plethora of works for the static security analysis of software. Some of those tools are research prototypes such as MOPS [4] and Eau Claire [5], others are successful commercial tools such as Fortify Source Code Analyzer [11], Coverity Prevent [7], and Ounce [18]. Our approach is complementary to all those works because we utilize architectural information to focus the analysis on the source code and carry out our analyses *directly* on the architecture. Common low-level security bugs can clearly be detected by well-established analysis tools. Only, Coverity Prevent considers the software architecture for analyses, which at this time is limited to the software visualization not supporting more complex analyses such as the reflexion method.

In addition, as new modeling elements can be added to the RFG through meta-modeling, domain- and framework-specific security analyses can be performed. Little work has been done in the context of software frameworks before such as the LAPSE tool, which can find low-level security bugs in JEE applications [16].

Our work can also be compared with approaches to modeling and analyzing security requirements, specifically, in the context of UML profiles. Two such specification languages are UMLsec [13], which allows one to formulate security requirements w.r.t. access control and confidentiality, and SecureUML [2], which allows one to model RBAC policies. In addition, Basin et al. present an approach to analyzing RBAC policies based on UML meta-modelling [2]. Our work currently is focused on checking the intended architecture against the implemented low-level architecture, although analyzing the architecture itself remains future work.

6 Conclusion and Outlook

We presented an architecture-centric approach to the security analysis of software which can be seen as complementary work to available security review tools. This allows one to conduct the analyses on a more abstract level detecting also logical security flaws in the software such as erroneous RBAC of business applications. Our approach is based upon a reverse engineering tool suite called Bauhaus. With the help of two early case studies, we showed that our approach could be employed in different domains, namely, JEE-based business applications and mobile phones.

As this paper only reports on early results, a lot of work remains to be done in the future. First, we intend to systematically analyze more comprehensive business applications which employ software frameworks. Specifically, this will be done in the context of the Service Oriented Architecture. In addition, we intend to investigate how security views can be extracted and how far this process can be automated in the context of an encompassing case study such as the Android middleware. Last but not least, one can contemplate how to integrate our architecture-based analysis method into the Software Development Lifecycle with the focus on the continuous monitoring of the security architecture.

References

1. American National Standards Institute Inc. Role Based Access Control, 2004. ANSI-INCITS 359-2004.
2. D. Basin, M. Clavel, J. Doser, and M. Egea. Automated analysis of security-design models. *Information and Software Technology*, 51:815–831, 2009.
3. CERT/CC. CERT statistics, 2008. http://www.cert.org/stats/.
4. H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244, 2002.
5. B. Chess. Improving Computer Security Using Extended Static Checking. In *IEEE Symposium on Security and Privacy*, pages 160–, 2002.
6. David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.
7. Coverity. Coverity Prevent, 2009. http://www.coverity.com.
8. J. Czeranski, T. Eisenbarth, H. Kienle, R. Koschke, and D. Simon. Analyzing xfig Using the Bauhaus Tool. In *Working Conference on Reverse Engineering*, pages 197–199. IEEE Computer Society Press, November 2000.
9. G. Dennis, K. Yessenov, and D. Jackson. Bounded Verification of Voting Software. In *Verified Software: Theories, Tools, Experiments, Second International Conference*, volume 5295 of *Lecture Notes in Computer Science*, pages 130–145. Springer, 2008.
10. W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, 2009.
11. Fortify Software. Fortify Source Code Analyzer, 2009. http://www.fortify.com/products/.
12. M. R. Garey and D. S. Johnson. *Computers and Intractability*. Freeman, San Francisco, 1979.
13. J. Jürjens and P. Shabalin. Automated verification of UMLsec models for security requirements. In *Proc. of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications*, volume 3273 of *LNCS*, pages 365–379. Springer, 2004.
14. K. Ashcraft and D.-R. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *IEEE Symposium on Security and Privacy*, pages 143–159, 2002.
15. R. Koschke and D. Simon. Hierarchical Reflexion Models. In *Working Conference on Reverse Engineering*, pages 36–45. IEEE Computer Society Press, November 2003.
16. V.B. Livshits and M.S. Lam. Finding Security Vulnerabilities in Java Applications Using Static Analysis. In *Proceedings of the 14th USENIX Security Symposium*, August 2005.
17. G. McGraw. *Software Security: Building Security In*. Addison-Wesley, 2006.
18. Ounce Labs Inc. Website, 2009. http://www.ouncelabs.com/.
19. A. Raza, G. Vogel, and E. Plödereder. Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering. In *Ada-Europe*, volume 4006 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 2006.
20. Sun Microsystems. The Java EE 5 Tutorial, 2008. http://java.sun.com/javaee/5/docs/tutorial/doc/bnclz.html.
21. Universitaet Stuttgart. Project Bauhaus—Software Architecture, Software Reengineering, and Program Understanding, 2009. http://www.bauhaus-stuttgart.de/bauhaus/index-english.html.