Challenges in Secure Software Evolution - The Role of Software Architecture

Stephan Seifermann, Emre Taşpolatoğlu
FZI Research Center for
Information Technology
{seifermann,taspolat}@fzi.de

Ralf Reussner, Robert Heinrich
Karlsruhe Institute of Technology (KIT)
{reussner,heinrich}@kit.edu

Abstract

Achieving quality properties for software systems and maintaining them during evolution is challenging. Especially, security properties often degrade during software evolution. This is often not noticed and can lead to monetary loss and serious damage to the company's image. Approaches for maintaining security properties exist but fail to exploit the knowledge of the architectural design phase. This results in high effort and slow reactions on evolutionary changes. In this paper, we describe five key challenges in maintaining security properties during software evolution and show how architecture supports mastering them.

1 Introduction

Developing software with sufficient quality is challenging. Even if functional requirements are adequately realized, satisfying quality requirements such as efficiency, availability and security is difficult. Even worse, quality properties once established for a system often vanish in later steps of evolution. Systems once performing in an acceptable manner may lose one or several of these quality properties after an evolutionary change. Consequences on the quality when introducing new functionality are not well understood. This is particularly bad for security properties as their degradation during system evolution is rarely noticed immediately. In addition, establishing and guaranteeing security properties is challenging, as

C1 it is hard to imagine all potential future steps of intelligent attackers in advance,

C2 security depends on many assumptions on the execution context. As this context evolves, some assumptions may not hold anymore and hence, the security argument becomes invalid,

C3 security properties not only strongly depend on the software, but also the execution context's configuration, such as access right configurations,

C4 formal analyses and ideally verification of software scales bad, and

C5 usually not all attack paths through involved libraries, operating systems, middleware-platforms etc. can be seriously taken into account.

There is a research gap in exploiting software architecture to tackle these challenges. Instead, security evolution often focuses on fine-grained design and code. This prohibits efficient identification of parts to be adapted for maintaining security during evolution.

This paper is structured as follows: In Section 2, we present the state of the art on secure software evolution. The mentioned challenges of keeping established security properties during software evolution with focus on the architectural level are discussed in Section 3. Section 4 presents sketches of potential solution strategies and Section 5 concludes the paper.

2 State of the Art

Gained attention on the combined field of software security and evolution in the last years lead to many approaches with various focuses on the targeted development phases, artifacts, used methods, and so on. Felderer et al. [6] call for integrating security in the software development lifecycle and considering all artifacts and phases of the development process. In their survey, they focus on UML-based modeling and evolution of security-related artifacts. This includes co-evolution of the security and system artifacts. The survey identifies a research gap in the evolution of architectural security artifacts and lists approaches for separate security modeling and architecture evolution. Dai and Cooper [5] also list such modeling approaches but do not consider evolution as well.

Many security evolution approaches focus on specific development phases. iObserve [8] eases transitions between operation-level adaptation and development-level evolution for distributed cloud-based systems by extending the MAPE control loop with shared component-oriented models. A model-driven engineering approach [7] generates artifacts to be executed during run-time. Monitoring probes keep models up-to-date and detects violated security requirements such as location of sensitive data [14].

Developers incorporate security during the implementation mainly from three viewpoints: a) The constructive viewpoint aims for high initial code quality by applying best practices such as security patterns

[15], increasing the test coverage [11, chap. 4], or performing reviews of the produced code [11, chap. 3]. b) Code analyses focus on certain aspects. For instance, FindBugs [10] detects common bugs by bug patterns and JOANA [17] analyzes security by information flows. These analyzes are fast because of their narrow focus. c) Formal approaches such as model checking [4] and verification ensure conformance to a given specification. Model checking is applicable for finite-state systems or components. Verification does not limit the states and therefore usually cannot be fully automated. KeY [1] provides interactive verification of specified contracts, for instance. This introduces high effort w.r.t. the verification itself and the formalization of the specification as already outlined in [2]. Modular and incremental analyses or verification enable fast reactions on evolutionary changes.

The SecVolution approach [3] supports the evolution of common security apsects such as access control or confidentiality in all development phases. A security context knowledge database holds information necessary to check the fulfillment of security requirements. Developers use the UMLSec UML profiles to document requirements. Verification tools such as CARiSMA[1] check conformance to requirements. When the system or its environment changes, security maintenance rules reestablish security properties by suggesting changes or using the knowledge delta to transform software or security artifacts such as access control rules. Monitoring context changes and defining maintenance rules is a manual task and can be domain-specific. UMLSec focuses on design-time but could be applied to architecture as well.

ModelSec [13] spans from requirements elicitation to implementation. Model transformations incorporate security models for software artifacts in various phases to generate implementation code. However, evolution and maintenance phases are out of scope.

The following approaches do not elicitate or analyze security properties by software artifacts such as architectural models or formal requirements directly. Useful results require considerable security knowledge. Threat modeling [18] is a manual process of identifying, rating and treating security issues in software systems. Misuse cases [16] elicitate security requirements by formulating unwanted use cases. Both approaches suffer from the error-prone manual process that is extensive because of fuzzy stop criteria. Additionally, the only implicit relation to concrete software artifacts such as code makes result interpretation hard.

3 Open Issues

We derived five open issues from the identified challenges (C1-C5): As the UMLSec approach [3] stresses, system changes and especially to its environment including the threats drive the evolution. Treating the fast evolution of the attacker's capabilities is hard (C1). Additionally, existing approaches do not exploit the system context and runtime configuration for rating the security completely. UMLSec [3] emphasizes coverage of the system context (C2) but provides no comprehensive description of relevant information nor a fully formalized extraction process. The same holds true for runtime configuration (C3). iObserve [14] exploits runtime configurations for detecting security issues in cloud environments but gathers and monitors runtime configuration for a small cloud-relevant subset only. Trade-offs between precise and fast results are often necessary because of a) badly scaling high-precision analyses (C4) such as verification approaches [2] and b) finding a feasible reduction of the analysis' complexity (C5) by narrowed analysis scopes as done by JOANA [17] for instance.

The following paragraphs describe the issues from an architectural viewpoint because this is a research gap according to Felderer et al. [6]. Therefore, we focus on scenarios that could benefit from exploiting architectural artifacts. Nevertheless, the issues are generalizable for other artifacts as well. When talking about security analyses, we consider two types: Determining security properties of the current system state and observing changes affecting these properties. Along with security properties, we focus on attacks for virtual scenarios in contrast to physical ones such as breaking a padlock. Attacks are "the act of carrying out an exploit, an instance of an attack pattern created to compromise a particular piece of target software" [9, chap. 2]. We do not limit software evolution to certain artifacts or phases because they often depend on each other. Nevertheless, changes must influence architectural artifacts such as components, deployment, design decisions or security policies.

C1 Threat Evolution occurs permanently, affects all development phases, and introduces new problems for security evolution approaches. Threat modeling is not limited to attackers relevant for a certain phase only anymore but spans many phases. Therefore, developers have to exploit information of all phases including the often neglected operations phase (C3). UMLSec [3] proposes this as well. Gathering all relevant information is still an open issue. The architectural viewpoint is a collection of aspects of different stakeholders and combines information from various phases such as structural and deployment information. Unfortunately, little research about exploiting architecture for security information extraction is available. An resulting attacker model must, however, still be extended with phase-specific information. Analyses have to reveal emerging attacks timely because the development team usually does not recognize security issues by accident. Therefore, fast issue detecting analyses are required (C4, C5). Additionally, assumptions that affect the security of a system

[1] http://carisma.umlsec.de

become invalid fast (C2). A structured approach for treating invalidated assumptions is necessary.

C2 Context Evolution occurs in a timely similar matter as (C1). It is permanent and happens in every software development phase. The context of a software consists of not only its environment or configuration, but further includes stakeholders that interact with the system. Still, due to intentions of attackers, we separate them from general stakeholders. This allows distinguishing two different evolution aspects – the execution context and the attackers – despite the tight coupling between the evolutionary changes of them (C1, C3). An attacker's intention and performance is based heavily on how the system's context is defined. Also the effectiveness of security measures relies on the kind of contextual execution environment of the software. Despite the importance also outlined by Bürger et al. [3], there is hardly support for evolutionary context changes especially on architectural level. Tracking evolution in multiple steps and especially in larger long-living systems is hard without explicit knowledge regarding the context. This introduces uncertainty about validity of security-relevant components and aspects of the software because implicit context information that is not properly formalized and documented tends to get lost. To overcome this challenge, it is important to document even the most implicit security-relevant context assumptions and possibly their influence on architectural elements.

C3 Changing Runtime Configuration is often ignored because interfaces between architecture models and configuration properties do not exist. As the integration of runtime configuration into the software and its context environment fails, it is hard to include into analyses. Existing approaches such as iObserve often derive component structures, deployments and usage characteristics but do not provide information required for security analysis [14]. Nevertheless, failing to integrate runtime configuration influences the system security massively. It can invalidate security mechanisms e.g. by introducing holes in a firewall. Compliance checks of configuration and requirements are necessary. Tracking of configurations and their changes enable this but are hard to realize because multiple stakeholders introduce these changes and the linking to software artifacts such as deployment description or code is ambiguous.

C4 Bad Scaling of Analyses occurs especially for analyses that produce high-precision results such as verification approaches [2]. At a certain complexity, a full analysis is not feasible anymore. A guideline has to define a trade-off between the time needed and the accuracy or coverage. Individuals must not decide this on their own because of incomparable results or missed issues in important modules. To cope with this issue, the guideline can lower the coverage of the analysis by selecting important modules for a more detailed analysis or lower the accuracy by increasing the level of abstraction. A high level of abstraction can still produce important results. Bürger et al. [3] showed this in their case study for design level analyses using UMLSec. Increasing the abstraction level further to the architectural level reduces the analysis effort even more but also implies a different set of detectable issues. Nevertheless, detecting a few issues early with little effort can reduce the overall effort because of cheaper fixing of issues in earlier development stages. Analyses on the architectural level are, however, still rare as pointed out by Felderer et al. [6].

C5 Complexity of Analyses occurs because there are too many possible attack paths through a software system and future attacks cannot be foreseen because of fast evolving attackers (C1). We cannot solve the latter issue but reduce its impact by providing means to react fast to emerging attacks. Fast analyses enable fast reactions. Demanding full path coverage, however, often does not allow fast analyses. Therefore, a feasible abstraction level is necessary. Nevertheless, creating such an analysis model can be challenging because of the abstraction gap between the concrete software artifacts such as code or requirements and the model: Developers have to map the known elements to the analysis counterparts when creating and interpreting the results. During evolution, they must calculate the change in the software artifacts, transform and apply this change to the analysis model, and find the difference in the analysis result. Furthermore, choosing the correct abstraction level can be challenging e.g. in threat modeling [18]. In this case, reducing the analysis effort increases the preparation and evolution effort for analyses. General analysis construction guidelines cannot eliminate this effort completely but reduce it: The level of abstraction of the concrete software artifacts to be analyzed and the analysis elements should be similar. Reusing existing artifacts of software and also of analysis as much as possible is a way to achieve this. This reduces the analysis specific part that has to be evolved separately and enables comprehensibility. Using software artifacts with an inherent high level of abstraction such as the architectural description is beneficial to achieve fast analyses.

4 Vision

We do **not** suggest replacing existing analyses. Instead, we argue for improving and running them more efficiently by incorporating architectural knowledge: Architectural security analyses enable targeted fine-grained analyses by suggesting meaningful targets with less effort. Additionally, developers can fix architecture-related security issues cost-efficiently in an early development stage or can find new types of security issues such as deployment issues.

There is, however, still a research gap in considering architectural artifacts in security evolution as outlined in the state of the art. We suggest two approaches to reduce this gap: 1) An architectural data flow analysis for finding security issues in design decisions and 2) a documentation and validation approach for security patterns, threats and their respective context assumptions regarding software evolution.

Data flow analyses usually operate on code to determine security properties. We plan to lift the analysis to the architectural level. This requires high level data flow specifications similar to control flow specifications used in Palladio [12], which is a software architecture modeling and simulation approach to determine quality attributes. The specification will cover data sources, sinks, properties and processing operators. Runtime configuration information like access control policies is another specification input thus tackling (C3). The architect defines types of data and data flows instead of individual data flows for the analysis. Reusing existing architectural description artifacts lowers analysis' complexity (C5) and is possible by an integration in modeling frameworks such as Palladio. The architect defines analysis goals in terms of normative data properties derived from requirements or user preferences. The analysis provides actual properties at any point in architecture, ensures compliance with user preferences and requirements, and derives requirements for external service providers. This becomes possible by using all information available in the architecture such as often neglected deployment information (C1). Fast analysis execution on high level artifacts enables fast reactions to changes and tackles the scaling issue (C4).

Evaluating the system regarding its security properties after an evolutionary change focuses heavily on information that is available based on limited views and stakeholders. The applicability and validity of security solutions depends on many software artifacts ranging from business matters to usage as well as code itself. This joint information needs to be persisted over several cross-cutting borders during the entire life-cycle of the software. Formal consideration of context evolution to analyze security properties plays a crucial role to provide secure application against changing conditions. Therefore, we plan to provide a formal documentation approach which covers (C1) and (C2) as well as (C3). It contains security patterns and threat models and combines them logically by using formal context assumptions covering large aspects of the underlying software system. As these security-related artifacts are to be represented on architectural level, they are model-based and combined with the Palladio approach [12]. The architect documents context information formally by considering security relevant aspects including threats, solutions, stakeholders, or configurations. The planned analysis uses this information as input to evaluate the security state of the software after evolutionary changes (e.g. in the runtime configuration or attacker profiles) by tracking down and validating the impacted architectural elements related to made security decisions.

5 Conclusion

In this paper, we introduced five key challenges in detail for the evolution of security properties in software systems that cover complexity and context dependencies of security analyses. We motivated, that including knowledge and analyses of various software artifacts on the architectural level is essential in tackling the mentioned challenges, and introduced our planned approaches to achieve this.

Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (grant 01IS13023C), by the German Federal Ministry of Economic Affairs and Energy (IGF-Vorhaben 18363 N) and German Research Foundation in the Priority Programme SPP 1593. We want to thank the research groups of Jörn Müller-Quade at the KIT and the FZI for the fruitful discussion about the topic.

References

[1] W. Ahrendt et al. The KeY tool. *SoSyM*, 4:32–54, 2005.

[2] C. Baumann et al. Lessons learned from microkernel verification: Specification is the new bottleneck. In *SSV'12*, number 102 in EPTCS, 2012.

[3] J. Bürger et al. Restoring Security of Long-Living Systems by Co-Evolution. In *COMPSAC'15*, volume 2, pages 153–158. IEEE, 2015.

[4] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT, 2001.

[5] L. Dai and K. Cooper. A survey of modeling and analysis approaches for architecting secure software systems. *I. J. Network Security*, 5(2):187–198, 2007.

[6] M. Felderer et al. Evolution of security engineering artifacts: A state of the art survey. *IJSSE*, 5(4):48–98, 2014.

[7] R. Heinrich et al. Integrating run-time observations and design component models for cloud system analysis. In *Models@run.time*, pages 41–46. CEUR, 2014.

[8] R. Heinrich et al. Architectural run-time models for operator-in-the-loop adaptation of cloud applications. In *MESOCA*, pages 36–40. IEEE, 2015.

[9] G. Hoglund and G. McGraw. *Exploiting software: How to break code.* Pearson Education India, 2004.

[10] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *ACM SIGPLAN Notices*, volume 39, pages 92 – 106. ACM, 2004.

[11] G. J. Myers, T. Badgett, and C. Sandler. *The art of software testing.* John Wiley & Sons, 3rd edition, 2012.

[12] Reussner, Ralf H. et al., editor. *Modeling and Simulating Software Architectures – The Palladio Approach.* MIT, 2016. ISBN: 978-0-262-03476-0, to appear.

[13] Ó. Sánchez et al. Modelsec: A generative architecture for model-driven security. *J. UCS*, 15(15):2957–2980, 2009.

[14] E. Schmieders, A. Metzger, and K. Pohl. Architectural runtime models for privacy checks of cloud applications. In *PESOS'15*, pages 17–23. IEEE, 2015.

[15] M. Schumacher et al. *Security Patterns: Integrating Security and Systems Engineering.* John Wiley & Sons, 2005.

[16] G. Sindre and A. Opdahl. Eliciting security requirements with misuse cases. volume 10, pages 34–44. Springer, 2005.

[17] G. Snelting et al. Checking probabilistic noninterference using joana. *it - Information Technology*, 56:280–287, 2014.

[18] F. Swiderski and W. Snyder. *Threat Modeling.* Microsoft, 1st edition, 2004.