Semiautomatic Security Requirements Engineering and Evolution using Decision Documentation, Heuristics, and User Monitoring

Tom-Michael Hesse*, Stefan Gärtner†, Tobias Roehm‡, Barbara Paech*, Kurt Schneider†, and Bernd Bruegge‡
*Institute of Computer Science, University of Heidelberg, Germany
{hesse, paech}@informatik.uni-heidelberg.de
†Software Engineering Group, Leibniz Universität Hannover, Germany
{stefan.gaertner, kurt.schneider}@inf.uni-hannover.de
‡Institut für Informatik, Technische Universität München, Germany
{roehm, bruegge}@in.tum.de

***Abstract*—Security issues can have a significant negative impact on the business or reputation of an organization. In most cases they are not identified in requirements and are not continuously monitored during software evolution. Therefore, the inability of a system to conform to regulations or its endangerment by new vulnerabilities is not recognized. In consequence, decisions related to security might not be taken at all or become obsolete quickly. But to evaluate efficiently whether an issue is already addressed appropriately, software engineers need explicit decision documentation. Often, such documentation is not performed due to high overhead.**

To cope with this problem, we propose to document decisions made to address security requirements. To lower the manual effort, information from heuristic analysis and end user monitoring is incorporated. The heuristic assessment method is used to identify security issues in given requirements automatically. This helps to uncover security decisions needed to mitigate those issues. We describe how the corresponding security knowledge for each issue can be incorporated into the decision documentation semiautomatically. In addition, violations of security requirements at runtime are monitored. We show how decisions related to those security requirements can be identified through the documentation and updated manually. Overall, our approach improves the quality and completeness of security decision documentation to support the engineering and evolution of security requirements.

***Index Terms*—Security requirements engineering, decision knowledge, decision documentation, heuristic analysis, user monitoring, software evolution, knowledge carrying software.**

I. Introduction

To consider security issues when developing long-living information systems is of particular importance for the engineering and evolution of security requirements. But security requirements engineering and evolution in practice is often performed manually. As a consequence, decisions on security issues are not documented properly due to high manual overhead for collecting and structuring all related knowledge. But this is important to enable software engineers a quick access and review of decisions because security decisions and requirements remain important after deployment. As newly discovered vulnerabilities might affect the security requirements, software engineers need to find out whether the security requirements are adhered and used as intended during system operation. Thus, security requirements need to be continuously monitored and tracked during usage and evolution of the system. If new vulnerabilities occur, the affected requirement specifications and their development decisions have to be identified and updated, what again causes manual effort.

To overcome these problems, we propose a semiautomatic approach for documenting decisions, which address security requirements. For this purpose, our approach combines *structured documentation* of security-related development decisions with knowledge from *heuristic analysis* and *user monitoring*. The structured decision documentation enables software engineers to gain comprehensive and fast access to security decisions, their implementation and related requirements. This supports a quick reaction to new vulnerabilities and helps in verifying that the system conforms to security requirements, for instance in case of an application audit.

Since software engineers are usually not aware of security issues in requirements, they cannot identify the related security requirements and address them appropriately in their development decisions. On this account, the heuristic analysis of textual requirements enables them to identify potential vulnerabilities in early project stages without requiring the team to be security experts themselves. The knowledge from the analysis can be incorporated semiautomatically into decision documentation, what helps in decreasing the documentation overhead. Additionally, user monitoring at runtime enables software engineers to identify new potential vulnerabilities, which are not covered by heuristic analysis so far. The related security requirements and their development decisions can be updated easier, as the decision documentation allows to access them faster and makes decisions more comprehensive. By integrating different sources of knowledge, our approach improves the quality and completeness of decision documentation. This is beneficial for software vendors and end users, as system security is improved from the beginning and can be kept secure over time more easily.

The remainder of this paper is structured as follows. In Section II, we introduce the components we integrate in our approach and describe our running example. Then, we describe in Section III how results from heuristic analysis can be incorporated into decision documentation. Afterwards, we explain how documented decisions can be updated with data from monitoring of security-relevant user behavior in Section IV. Finally, we present related work for automated decision documentation in security requirements engineering in Section V and conclude our insights in Section VI.

II. Overview of Components

The required components for our approach are depicted in Figure 1: Decision documentation, security heuristics and user monitoring. We will briefly introduce them in the following subsections. As a prerequisite, our approach requires an explicit specification of requirements in form of use cases. Its textual descriptions in natural language should be accessible as structured or modeled data.

A. Decision Documentation

The decision documentation component provides the structure for documenting knowledge about security-relevant decisions. Such *decision knowledge* addresses the implementation of security requirements, for instance, which type of authentication is used to hinder unauthenticated users to access sensible data. The documentation needs to be updated, when the decision is modified in order to cope with changing security requirements, for instance, when the encryption standard for authentication is changed due to a new vulnerability. Decision documentation helps software engineers, in particular requirement engineers and software architects, to find and exploit previous security-related decisions and plan upcoming decisions. So, they can understand and assess more easily why a certain requirement, design or library was considered and reflect this knowledge in future decisions.

Decision knowledge consists of the decision problem, its context, and rationales justifying the decision. A *decision problem* at least comprises the problem description, a set of alternatives and a set of criteria to evaluate each alternative [12]. But decision knowledge may contain many additional pieces of information explaining the *context* of the decision, such as assumptions about related requirements or implications of an alternative [14]. Moreover, decision knowledge can consist of *rationales*, which are justifications for and reasons behind a decision [5], such as arguments supporting or challenging an alternative. In order to document all this knowledge, we decided to use an iterative documentation model for decisions as presented in [9]. It offers a set of different knowledge elements, which can be aggregated iteratively and are depicted in Figure 2. The basic element of this model is the *Decision*, which represents the set of knowledge elements for one decision as aggregated *DecisionComponents*. Amongst others, DecisionComponents can be refined to a decision problem description as an *Issue*, to context information like an *Assumption*, to a solution description as an *Alternative*, or to a description of a rationale as an *Argument*. The decision knowledge can be linked to the requirements specification and other system information and stored in a central place. For instance, DecisionComponents can be related to other knowledge elements like related requirements or concrete implementation artifacts. This allows software engineers a fast access to security knowledge, which enables a fast reaction to new vulnerabilities.

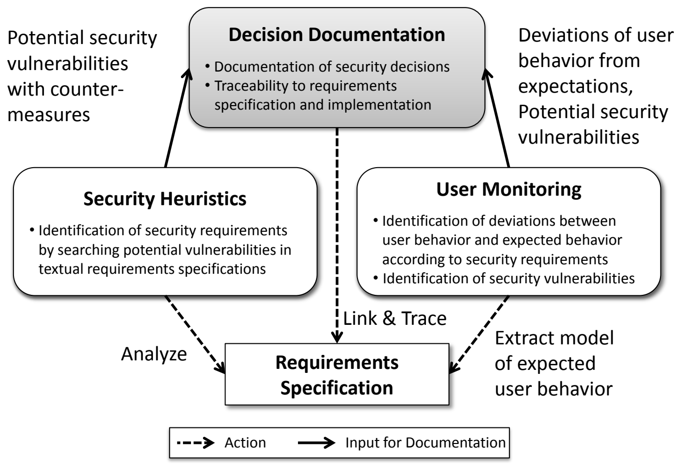
Decision Documentation
• Documentation of security decisions
• Traceability to requirements specification and implementation
Potential security vulnerabilities with counter-measures
Deviations of user behavior from expectations, Potential security vulnerabilities
Security Heuristics
• Identification of security requirements by searching potential vulnerabilities in textual requirements specifications
User Monitoring
• Identification of deviations between user behavior and expected behavior according to security requirements
• Identification of security vulnerabilities
Link & Trace
Analyze
Extract model of expected user behavior
Requirements Specification
Action
Input for Documentation

Fig. 1. Incorporating Decision Knowledge From Security Heuristics and User Monitoring Into Decision Documentation

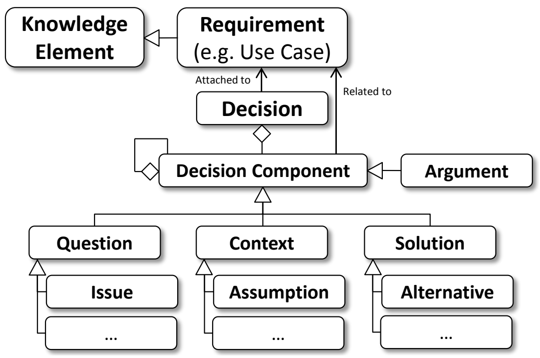

Fig. 2. Knowledge Elements of the Decision Documentation Model

We decided to use this decision documentation model, because all DecisionComponents can be aggregated without structural restrictions and may be refined iteratively. This flexibility supports the knowledge transformation from security heuristics, as the content of heuristic analysis results may vary and software engineers might even want to add additional contents to the incorporated knowledge. In addition, the ability of iterative refinement helps in updating the documentation with monitoring results, as extensions and changes to documented decisions can be made easily.

B. Security Heuristics

To find vulnerabilities in natural language requirements and use them for decision documentation, it is usually necessary to analyze a huge amount of requirements manually. On this

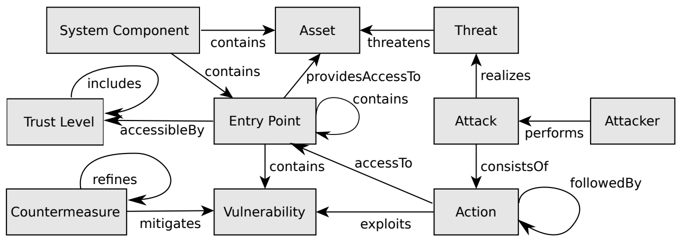

Fig. 3. Overall structure of the security knowledge required to derive heuristics [6].

account, the security heuristics component analyzes textual requirements specifications automatically. Therefore, we used our assessment approach as described in [6], which leverages previously reported incidents. An incident is an attempt to violate security policies or to gain unauthorized access to data [8].

To use an incident in our assessment approach, it must be modeled according to the knowledge structure as depicted in Fig. 3. According to that, an incident is described by corresponding actions which use several entry points to access assets (see [6] for further details). Additionally, discovered vulnerabilities are assigned to the incident. To incorporate expert knowledge for decision documentation, modeled incidents are enriched with additional security knowledge. This knowledge consists of a mixture of textbook knowledge, security obligations and laws, as well as experiences.

The assessment approach takes requirements in form of uses cases as well as modeled security incidents as input. The scenario of the use case is scanned for instances of relevant vocabulary and their chronological order. The found instances are heuristically matched with instances used to model incidents. If the comparison results in a positive match, the modeled incident has been found in the use case indicating a potential security vulnerability. The heuristic finding comprises the steps of the usage scenario which describe suspicious interaction. It is reported to the software engineer who has to investigate the vulnerability and to decide how to handle it. Decisions taken on base of heuristic findings are documented by the decision documentation component. The application of the security heuristics component and its interaction with the decision documentation component is described in detail in Section III.

C. User Monitoring

The user monitoring component analyzes the interaction between an end user and the application. Usually two things are necessary for an exploit to happen: A vulnerability and a malign end user. In case an application does not exhibit a vulnerability, no security problem exists. When a vulnerability is present, it might remain without effect if there is no malign user who exploits it. Hence, the user monitoring component monitors and analyzes end user actions at runtime, thereby implementing a dynamic analysis approach and complementing the static analysis of requirements using security heuristics.

The user monitoring component monitors end users during their interaction with the application to assess security requirements as described in [15]. Our goal is to check whether users adhere to security policies and to uncover potential new vulnerabilities. We hypothesize that a deviation of monitored user behavior from expected user behavior (as defined by security requirements) may indicate a vulnerability, e.g. when a user accesses data, which he or she is not allowed to access. An identified deviation is reported to software developers who have to investigate the deviation and decide how to handle it. These decisions are documented by the decision documentation component. The application of the user monitoring component and its interplay with the documentation component are detailed in Section IV.

D. Running Example

To illustrate the usage and integration of the three components, we introduce the medical application iTrust [11] as running example. It supports patients to manage their health records as well as medical staff to organize their work. Thus, security and privacy issues play an important role. The iTrust project was founded at North Carolina State University and is currently maintained by the Realsearch Research Group [11]. It consists of 55 use cases written in natural language (version 23). Based on these requirements, the iTrust system (version 17) has been developed as a web application. Thus, the requirements are sufficient to develop a working system. Since for iTrust no security incidents have been documented yet, we assume in the following sections that incidents have been discovered with a focus on technical aspects.

III. Incorporating Decision Knowledge From Security Heuristics

Regarding our running example iTrust, we consider use case 6 as described in [11]. In this use case, the patient choose to view a list of all licensed health care professionals (HCP) she has ever had an office visit with. The HCP's name, specialty, address, date of office visit is presented to the patient. Moreover, the patient can also add a HCP to her list by searching for the HCP's name or specialty. We further assume that an incident has been reported describing that the patient view with respect to the address field contains a vulnerability so that Cross-Site Scripting (XSS) attacks become possible. This is one of the most dangerous vulnerabilities in web applications according to the Common Weakness Enumeration (CWE) [17]. Cross-site scripting becomes possible through improper neutralization of input. Attacker can inject malicious browser-executable content into the patient view to steal sensitive information (e.g. medical identification number, patient information).

To use the reported incident in our assessment approach, we modeled the incident according to Fig. 3. For this purpose, we extracted affected system components, assets, trust levels, and entry points from the use case as listed in Table I.

After our heuristic analysis of requirements was performed, each finding belongs to one of the following categories:

A) The requirement contains security issues (true positive hit) and all steps of the usage scenario have been marked

TABLE I
EXTRACT OF SECURITY KNOWLEDGE USED IN THE ITRUST EXAMPLE

Concept	Individuals
System Component	iTrust Medical Records System
Asset	Address
Entry Point	Address field, Health record, View, Display
Trust Level	Patient, HCP
Threat	Execute unauthorized code or commands, Bypass protection mechanism, Read application data
Attack	Inject malicious script in address field
Attacker	Inside or outside attacker (unknown)
Vulnerability/ Security Issue	Cross-Site Scripting (XSS)
Countermeasure	Sanitize input (see CWE-79)

correctly
$\rightarrow$ Decisions to address these issues must be taken and documented.

B) The requirement contains security issues (true positive hit), but not all steps of the usage scenario have been marked correctly
$\rightarrow$ Decisions to address these issues must be taken and documented and an adapted documentation structure is required.

C) Security issues were incorrectly identified for the requirement (false positive hit)
$\rightarrow$ The finding can be discarded and no decision must be taken.

Transforming heuristic findings to decision knowledge decreases the documentation effort for decisions, as the incorporated security knowledge highlights decision points. To transform relevant findings into an initial set of documented decision knowledge, we have identified three necessary steps, which can be performed semiautomatically:

1) Security issues are grouped according to their origin in requirements specification as potential decisions.
2) Heuristic findings are mapped to elements of decision knowledge.
3) Missing knowledge is elicited and added to the related elements of a decision.

In the first step, the software engineers have to choose how to group decisions and where to attach them. This is of particular importance for enabling traceability between system specification and decision knowledge, so that decisions on security can be revisited in future development activities. To support this task automatically, one decision is created by default for each security issue which belongs to category A and linked to all steps of the regarded usage scenario. For example, after performing the heuristic analysis on iTrust use case 6, a potential XSS vulnerability is found for step 3 correctly. To document the decision for preventing this vulnerability, the decision is created and linked to step 3 directly. However, if a finding belongs to category B, it might be suitable to choose a different grouping, as some steps of the usage scenario are not affected in the actual system. For instance, if the incident was not modeled in detail, heuristic results would indicate the possibility of XSS for the whole use case. Then, the grouping of the decision should be changed to step 3 manually. If a finding belongs to category C, it can be discarded and does not have to be considered in the next steps. This may happen because the heuristic analysis of requirements is performed semantically. Thus, some findings might not be appropriate in all contexts. For example, in case the term "enter" was found in step 1 of use case 6, this does not necessarily mean that data is entered, but that a user enters a certain view. Then, this finding does not indicate a security issue.

Additionally, it has to be determined in the first step of our transformation, whether a decision that addresses the same heuristic finding is already documented. Then, the software engineer has to decide, whether the current finding shall only be linked to existing decisions instead of creating new ones. As a result of the first step, all potential decisions address a particular security issue and are attached to all suitable parts of the specification.

In the second step, the content of each heuristic finding is mapped to the decision knowledge elements in order to automatically generate proposals for the textual descriptions and links of the decision knowledge elements. Based on the modeled security knowledge as depicted in Fig. 3, a default mapping between security knowledge and decision knowledge is presented to the software engineers. The mapping is presented in Table II, where it is exemplarily applied to use case 6 from iTrust.

The *Vulnerability* of a *System Component* illustrates what security-related decision is needed to address the security issue and is therefore created as *Decision*. *Attack* and *Attacker* describe the details within the security issue corresponding to the description of a decision problem, so they are mapped to *Issue*. This Issue element is linked to the concrete requirement part such as a use case step or a subscenario, where the security issue was found. *Entry Points* and *Trust Level* are forming an *Assumption*, as they are the assumed circumstances permitting a potential security issue. A *Criterion* as specialized context knowledge serves to evaluate the decisions' alternatives by combining *Threat* and *Asset*. Then, alternatives are favored, which mitigate or overcome the given threat for the mentioned

TABLE II
DEFAULT MAPPING BETWEEN HEURISTIC AND DECISION KNOWLEDGE APPLIED ON THE ITRUST EXAMPLE

Heuristic Knowledge	Decision Knowledge	Example for Generated Textual Content
Vulnerability, System Component	Decision	Prevent *Cross-Site Scripting* in *iTrust Medical Records System*
Attack, Attacker	Issue	*Inject malicious script in address field* might be performed by *inside or outside attacker (link to use case step)*
Entry Points, Trust Level	Assumption	Attack can be performed through *address field* in *health record* for the roles *patient*, *health care personnel*
Threat, Asset	Criterion (as Context)	Prevent *execute unauthorized code* for *address*
Counter-measure	Alternative	*Sanitize input (link to mitigation strategies of CWE-79)*

asset. Finally, a *Countermeasure* describes how to react on a security issue. It is mapped to an *Alternative*. The software engineers have to decide for each finding if the default mapping is appropriate. Otherwise, they need to rectify it manually. For instance, for use case 6 the software engineer might want use the Countermeasure as part of the Decision and map the Vulnerability to Issue instead, as they are quite similar for this heuristic finding.

In the third step, the automatically generated proposals for textual contents and links of decision knowledge elements can be adapted and refined manually if necessary. Furthermore, the software engineers are requested to fill in missing description parts for knowledge elements that are not included in the findings. A major reason is that some aspects of the security knowledge might not have been modeled completely within the incidents. For instance, if no particular entry point is specified for a particular system component, the software engineer will be asked to add this as a part of the decisions' description. Another example is a missing countermeasure, so that no alternative can be added to the decisions' initial documentation automatically. If a decision becomes relevant in further development, it is likely to be enriched iteratively with more information.

IV. Updating Decision Knowledge Based on User Monitoring

User monitoring can serve our approach in two ways: It can help to prioritize heuristic findings and to uncover required updates for decision knowledge. First, the heuristic findings need to be prioritized, since not all use cases are relevant in operation equally. If the monitoring would show that an affected use case is used more frequently, heuristic findings would be prioritized as more relevant. The reason is that the damage is higher if the vulnerability is exploited by an attacker. Regarding iTrust, a XSS vulnerability on a site, which is shown to each patient by default, would execute a malicious script on many computers. Therefore, the vulnerability should be handled first for this use case.

Second, decision knowledge and its documentation need to be updated to address evolving security requirements. Again, we consider the iTrust, which is realized as a web application. It requires users to login (use case 3) before they can view their list of health care professionals and change it (use case 6). We use the login/ logout functionality to illustrate how the monitoring component identifies potential security issues using three steps [15]: Extracting a model of expected user behavior, monitoring user behavior, and comparing monitored and expected user behavior. First, a model of expected user behavior is extracted from the requirements semiautomatically. Therefore, the flow of events of each use case is transformed in a machine-readable format. In our example, a model of expected user behavior contains that users first login, then they perform an arbitrary number of tasks, and finally logout. Second, actions of users are monitored by software sensors, which are incorporated in the application and capture user actions. Depending on the implementation technology these software sensors can be implemented as dedicated logging code or reuse callback mechanisms of application development frameworks. Further, user monitoring can be performed on the user device, on the server (in case of a web application), or as a combination of both. Third, monitored user actions are compared to the extracted model of expected user behavior. In our example, a deviation will probably be detected for many users: They forget to logout and therefore deviate from expected behavior which requires to logout explicitly. This deviation is forwarded to the decision component. Now, software engineers have to investigate the deviation whether it is security-relevant or not. In the first case, they have to decide how to address the deviation properly. In our example, we are dealing with a medical application requiring a strict security policy. Thus, the deviation - the missing logout - is security-relevant and will probably be addressed by adding an automated logout feature, which terminates a user session after a certain time of inactivity. For a different type of application (e.g., a gaming app) with a less strict security policy, software engineers might decide to simply ignore the deviation.

The user monitoring component partially overlaps with an intrusion detection system: Both aim at the identification of anomalies in user behavior. But while intrusion detection systems usually operate on network and system level, our user monitoring component operates on the interaction level between user and application. This allows to assess whether the actual behavior of end users conforms to the given use cases.

The forwarded deviation is processed by the decision documentation component in order to prepare an update of the existing decision documentation. All decisions are retrieved that are linked with the related requirement. In our example, it is likely that a decision on how to implement the login/logout mechanism was made, which we assume to be documented already. This decision is linked to use case 3 and would, therefore, be reissued to software engineers for updating it in order to address the identified deviation for the logout step. As mentioned, it will probably be decided to add an automated logout feature. The decision for an automated logout and consequent changes of security requirements can be documented directly by adding this knowledge to the given decision documentation. In consequence, an effective reaction and a comprehensive documentation of the requirements' evolution was supported by integrating user monitoring in our approach.

V. Related Work

Our approach extends the current approaches for decision documentation in requirements engineering, as it introduces the semiautomatic documentation of security-related decisions with low effort and improved quality and completeness of informal documentation. Aurum et al. [2] outline the need for a knowledge sharing environment for decisions in requirements engineering, but they do not provide a computer-supported approach for knowledge acquisition and structuring. Nuseibeh et al. [13] describe their experiences when performing a security requirements analysis for an air traffic control systems. This security analysis based on a method introduced by Haley et al. [7] requires software engineers to create a formal context description for the system. Then, this context is validated against security requirements automatically. Although decisions are considered indirectly by proving that the system conforms to the security requirements, there exists no explicit decision representation in this approach. Moreover, modeling the formal context is performed manually. In consequence, the effort for identifying and documenting security-related decisions is high and error-prone. Whereas this is no problem for safety-critical air traffic systems, high analysis and documentation effort is not suitable for the development of most information systems.

The approaches of Sindre and Opdahl [16] and Braz et al. [3] focus on elicitation support using misuse cases to uncover security requirements. But they do not describe how to keep the security requirements specification and their related decisions up-tp-date over time. Tun et al. present an approach to analyze the evolution of security requirements [18]. They propose a meta-model to automatically generate templates for evolving security requirements. But this requires software engineers to specify manually and formally, which requirements are given for the system and what changes have occured in the environment. Burge et al. [4] present an ontology-based tool to provide rationale support for software engineers. Jansen et al. [10] describe a tool-supported approach to discover decisions within given architectural descriptions. However, both approaches focus on design and implementation and do not cover requirements.

VI. Conclusion

We have described a semiautomatic approach for incorporating knowledge from heuristic analysis and user monitoring into decision documentation. Therefore, we use heuristic analysis to identify potential vulnerabilities in textual requirements and user monitoring to identify potential vulnerabilities at runtime. Then, we outlined several steps how knowledge from these sources can be transformed to be a suitable starting point for or extension to decision documentation. As a result, our approach improves the quality and completeness of security requirements and their related decisions. Moreover, it lowers the effort for decision documentation. This supports the engineering and evolution of security requirements of long-living software systems, as software engineers can react faster when security issues become known and have to be addressed in implementation decisions.

Currently, we are developing a prototype of our approach utilizing Unicase [1], which is a model-based management tool for project and system knowledge in Eclipse. Future work should consider two directions: First, evaluation of our approach in form of a controlled experiment with software engineers. And second, investigation of other measures and information sources to improve engineering and evolution of security requirements and related decisions. This may include educating software engineers or enforcing a certain security requirements engineering process.

Acknowledgement

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future — Managed Software Evolution.

References

[1] Unicase. http://unicase.org/ (Retrieved in 05-2014).

[2] A. Aurum, C. Wohlin, and A. Porter. Aligning Software Project Decisions: A case study. *International Journal of Software Engineering and Knowledge Engineering*, 16(06):795–818, Dec. 2006.

[3] F. a. Braz, E. B. Fernandez, and M. VanHilst. Eliciting Security Requirements through Misuse Activities. In *Proc. of the 19th Int. Conference on Database and Expert Systems Applications*, pages 328–333. IEEE, 2008.

[4] J. E. Burge and D. C. Brown. Software Engineering Using RATionale. *Journal of Systems and Software*, 81(3):395–413, Mar. 2008.

[5] A. Dutoit, R. McCall, I. Mistrik, and B. Paech. *Rationale Management in Software Engineering*, chapter Rational Management in Software Engineering: Concepts and Techniques. Springer, 2006.

[6] S. Gärtner, T. Ruhroth, J. Bürger, K. Schneider, and J. Jürjens. Maintaining Requirements for Long-Living Software Systems by Incorporating Security Knowledge. In *Proc. of the 22th International Requirements Engineering Conference*. IEEE, 2014.

[7] C. B. Haley, R. C. Laney, J. D. Moffett, and B. Nuseibeh. Security Requirements Engineering: A Framework for Representation and Analysis. *IEEE Trans. Software Eng.*, 34(1):133–153, 2008.

[8] S. Hansman and R. Hunt. A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31–43, 2005.

[9] T.-M. Hesse and B. Paech. Supporting the Collaborative Development of Requirements and Architecture Documentation. In *3rd Int. Workshop on the Twin Peaks of Requirements and Architecture*, pages 22 – 26. IEEE, 2013.

[10] A. Jansen, P. Avgeriou, and J. S. van der Ven. Enriching software architecture documentation. *Journal of Systems and Software*, 82(8):1232–1248, Aug. 2009.

[11] A. Meneely, B. Smith, and L. Williams. iTrust electronic health care system case study. In *Software and Systems Traceability*, pages 425–438. Springer, 2012.

[12] T. Ngo and G. Ruhe. Decision Support in Requirements Engineering. In *Engineering and Managing Software Requirements*, pages 267–286. Springer, 2005.

[13] B. Nuseibeh, C. B. Haley, and C. Foster. Securing the Skies: In Requirements We Trust. *IEEE Computer*, 42(9):64–72, 2009.

[14] B. Paech, A. Delater, and T.-M. Hesse. Integrating Project and System Knowledge Management. In *Software Project Management in a Changing World*, pages 161–198. Springer, 2014.

[15] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, and W. Maalej. Monitoring user interactions for supporting failure reproduction. In *Proc. of the 21st International Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE, 2013.

[16] G. Sindre and A. L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[17] The MITRE Corporation. Common Weakness Enumeration (CWE), 2014.

[18] T. T. Tun, Y. Yu, C. Haley, and B. Nuseibeh. Model-Based Argument Analysis for Evolving Security Requirements. In *4th International Conference on Secure Software Integration and Reliability Improvement (SSIRI)*, pages 88–97. IEEE, 2010.