Towards a Practical Security Analysis Methodology

Alexander van den Berghe

iMinds-DistriNet, Department of Computer Science

KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium

E-mail: alexander.vandenbergh@cs.kuleuven.be

Website: <https://people.cs.kuleuven.be/alexander.vandenbergh>

Abstract—The research community has proposed numerous techniques to perform security-oriented analyses based on a software design model. Such a formal analysis can provide precise security guarantees to the software designer, and facilitate the discovery of subtle flaws. Nevertheless, using such techniques in practice poses a big challenge for the average software designer, due to the narrow scope of each technique, the heterogeneous set of modelling languages that are required, and the analysis results that are often hard to interpret. Within the course of our research, we intend to provide practitioners with an integrated, easy-to-use modelling and analysis environment that enables them to work on a broad range of common security concerns without leaving the software design's level of abstraction.

I. INTRODUCTION

Developing secure software is still a daunting task with considerable pitfalls. To be successful, security concerns should be considered right from the start of the development process, including the design phase. The research community has recognized this, and has produced an elaborate set of methodologies that embody this security by design principle. In earlier work, we have evaluated the state of the art regarding this topic with a systematic literature review [1], focussing on the possibilities for modelling and analysing security concerns at the design level. We have found that most existing methodologies support modelling only a small number of concerns, usually access control, and they offer limited support for analysis.

Based on these results, we identify the need for a new methodology that enables the designer to model security solutions for a broad range of concerns, and that offers precise, formal analysis techniques for the created models. In particular, our research goal is to provide a methodology, which should have the following three characteristics.

First, the methodology should offer an *adequate representation of security concepts in software design models*. This entails that the designer should be given the possibility to create a security-specific model using concepts at an appropriate level of abstraction. These models should offer support for a broad, integrated set of security concerns. Furthermore, the designer should be encouraged to reuse existing, well-known security solutions as much as possible. This can be achieved by offering a catalogue of common security concerns and solutions, based on best practices or security standards, for example.

Second, the methodology should *enable precise, formal analysis of the security concerns of these models*. Because existing verification techniques already exist, they should be

reused as much as possible. The underlying details of the analysis should nevertheless remain invisible to the designer.

Finally, after performing an analysis, *the results of this analysis should be meaningful to the designer*. This means that the output from the analysis tools must be translated back to the model's level of abstraction, so that it can be easily understood by the designer. Additionally, the analysis results may be used to further enrich the security model with relevant information and constraints that are necessary to provide the desired security guarantees.

II. RELATED WORK

Methodologies that support the analysis of software designs can be roughly divided into two groups.

The first group contains methodologies that require the designer to explicitly specify the property to analyse, allowing for an unlimited set of properties. Within this group, some methodologies rely only on common design languages such as UML and OCL. For instance, in SecureUML [2], designers specify their desired role-based access control (RBAC) properties as OCL queries executed against UML-based designs augmented with a formal OCL foundation. Sohr et al. [3] and Yu et al. [4] verify possible system states against OCL constraints that express RBAC properties. Other methodologies rely on notations besides the design languages. For example, UML AC [5] formalises the semantics of access control policies and properties using graphs. Xu and Nygard [6] and Kong et al. [7] formalise integrity threats using respectively graph grammars and Petri nets. Methodologies can also rely on external tools to perform the analysis. For example, Georg et al. [8] and AMF [9] employ Alloy to analyse respectively authentication and authorisation properties.

The second group contains methodologies that provide a pre-defined set of properties to analyse. UMLsec [10] allows designers to annotate their design with a range of properties and employs formal tools to analyse these designs. Unfortunately, UMLsec provides no guidance to the designer while annotating his or her design. Another example is Buyens et al. [11], who use set theory to detect least privilege and separation of duty violations.

The above methodologies do not provide means to reuse well-known security solutions. Nevertheless, such reuse is very common in software development, with patterns [12], [13] being a well known example. Uzunov et al. [14] survey a number of design methodologies that reuse security patterns.

Figure 1. A birds-eye-view of our security analysis methodology.

FDAF [15] provides an aspect repository containing, for example, aspects modelling RBAC and data origin authentication. Unfortunately, these methodologies barely offer security analysis of the created designs.

Heyman [16] combines security patterns with formal modelling, to allow designers to both reuse the patterns while providing the possibility to analyse their resulting designs. This methodology requires a background in both security and formal modelling, and provides only the set of patterns as guidance to mitigate discovered security violations.

Our methodology aims at combining formal analysis with the reuse of common security knowledge, while minimizing the required skills of the designer. The structure of our knowledge catalogue is partially inspired by the goal refinement and formalisation techniques from the KAOS [17] requirements engineering methodology.

III. PROPOSAL

From a birds-eye-view, our proposed methodology consists of four main elements (Figure 1): (1) a vocabulary which defines the common concepts that are used by the other three elements; (2) a catalogue that primarily consists of security concerns, corresponding solutions, strategies, and low-level facts; (3) a modeller through which the designer works with security-specific models; and (4) a verifier that analyses the models and generates feedback for the designer. The remainder of this section discusses each of these elements in detail.

In practice, the designer creates a security and attacker model using the modeller, based on the security concerns, solutions and strategies from the catalogue. The designer's choices determine the low-level facts that need to be verified. The verifier, hidden from the designer, transforms the models, together with these facts, into a verification model which is then automatically verified according to a specific formal

approach. Afterwards, the verification result is transformed back into feedback that is meaningful to the designer and can be incorporated back into the security model.

A. Vocabulary

The vocabulary provides a common theory for the rest of the methodology. It comprises the terminology and the corresponding semantics, and acts as the glue between the other elements. For example, the vocabulary should define what a cryptographic hash function is. The catalogue, modeller and verifier must then all treat hashing according to this common definition. Otherwise, it could happen that the verification result does not apply to the security model, for instance.

B. Catalogue

The catalogue is instrumental in our methodology in that it bundles the analysable security concerns and relates them to automatically verifiable facts. This enables designers to select predefined security requirements, instead of manually specifying the desired properties by hand. To achieve this, the elements in the catalogue are defined as templates. In this text, we represent a template parameter as $\langle \text{parameter} \rangle$. The elements of the catalogue are spread over three levels of abstraction, namely high, medium and low (cf. *Catalogue* in Figure 1).

Security concerns populate the highest level. They are derived from standards such as [18] and [19]. An example is the *storage confidentiality* concern, which states that $\langle \text{data} \rangle$ is stored confidentially on $\langle \text{persistent storage} \rangle$.

Solutions and strategies are practical means for achieving a security concern, and occupy the medium level in the catalogue. A solution can consist of multiple strategies, combined using the logical AND/OR operators. Security patterns [13] can be used as a basis for populating the catalogue with specific solutions and strategies. For example, enforcing *access control* is a solution to the *storage confidentiality* concern above. It involves of two strategies, *authentication* and *authorisation*, which must both be implemented (AND).

Finally, facts inhabit the lowest level. A correctly implemented strategy must satisfy all of its facts. For example, the *authorisation* strategy must satisfy both *authenticate first* (a user must be authenticated before he or she can be authorised) and *authorise before access* (the user must be authorised before accessing the persistent storage).

Only verifying facts obviously does not provide any formal guarantees about the fulfilment of the corresponding strategy, solution and security concern. This can only be attained if each relation in the catalogue is proven sound. More specifically, it must be proven that the set of facts associated to a strategy are sufficient to implement that strategy, and that the combination of strategies achieves the corresponding solution. Finally, each solution must actually solve the concerns to which it is associated. To allow for such proofs, all elements in the catalogue are accompanied by formal specification, e.g. in linear-time temporal logic, based on the concepts in the

vocabulary. Experts must write such proofs once, after which they can be re-used.

Since the security landscape always evolves, the catalogue must allow extensions with new elements and corresponding proofs.

C. Modeller

The modeller provides a domain-specific language (DSL) allowing the designer to create security and attacker models. A security model is a model of the software that contains only the security-relevant elements. For example, when analysing *confidential storage*, it should only include the locations that store the data in question, and the elements that act on that data, e.g. by encrypting it.

The attacker model associates the abilities of the attacker with each element of the security model. For example, an attacker can remove messages from a certain connection but cannot read or add messages. Furthermore, an attacker model also describes the initial attacker knowledge, e.g. the possession of a certain cryptographic key.

The modeller also receives feedback from the verifier. Such feedback, for example, shows violations of the security concern, or identifies the design elements that are crucial for achieving the security concern.

D. Verifier

The verifier automatically transforms the security and attacker models, and the formal definitions of the facts, to a verification model. An off-the-shelf tool (e.g. a model checker, automated theorem prover, or any already existing security analysis tool) verifies this model and returns a verification result. Our only requirement on such a tool is that it does not need manual interaction. The result of the tool must be transformed back to feedback meaningful to the designer, hence in terms of the security and attacker models.

Adding a new tool as verifier requires specifying two transformations. A first transformation to create a verification model in the correct language and a second one to convert the verification result to feedback. Both transformations require a proof that their output preserves the semantics of their input.

IV. PROOF OF CONCEPT

To show the viability of our methodology, we have developed a proof of concept, including a tool built on top of the Eclipse framework, that supports a small subset (storage confidentiality) of the envisioned catalogue (Figure 2). We explain our proof of concept using a simplistic online store example, in which customers can create an account that contains their credit card number. Furthermore, for accountability reasons, the store logs every change to a customer profile.

The designer creates a security model (Figure 3) for the online store using our drag-and-drop editor based on Sirius [20]. The credit card numbers are transmitted between the elements according to the connections drawn between them. Both `AccountDB` and `LogDB` store credit card numbers persistently (indicated by the `Storage` role in Figure 3).

Figure 2. Example catalogue containing the storage confidentiality concern.

Figure 3. The security model for the online store, red coloured elements indicate the violation.

Our example requires the *storage confidentiality* concern mentioned earlier. Instantiating this concern results in the requirement that *a <credit card number> is stored confidentially on <AccountDB> and <LogDB>*.

Assume that the designer selects the *no persistent plain text* solution to resolve this concern. This solution comprises four alternative strategies, each of them associated to one fact. Therefore, the verifier will need to prove that at least one of these four facts is true for every storage location. The designer follows the *encrypted storage* strategy, and specifies that `AccountDBMgr` encrypts the credit card numbers, indicated by the `Encryptor` role (Figure 3), before storing them.

Currently we have not defined an attacker model DSL in our proof of concept. Therefore, we assume an implicit attacker model in which the attacker is only able to read from all persistent storage.

In our proof of concept, we use the Spin model checker [21] to perform the verification. The security model, implicit attacker model, and the facts mentioned above, are transformed to a verification model, which here takes the form of a Promela file. The result of running Spin is a trail that indicates that the verification model violates the facts. This trail is transformed back to the corresponding flow through the elements of the security model, which are highlighted in red by the modeller (Figure 3). The feedback indicates that `LogDB` stores the credit card data as plain text, which violates the original concern.

To remedy this violation, the designer must also apply one of the available solutions to the violating path. The designer decides to apply the *no persistent plain text* solution again, but

now by implementing the *truncated storage* strategy. This is achieved by adding an element capable of truncating data in front of the LogDB. Repeating the analysis returns no more violations, indicating that the model satisfies the concern.

As a side note, we recognize that the violation in this simple example was trivial to spot without performing any formal analysis, but for realistic, more complex models, we believe that automated support quickly becomes essential to ensure that a system adheres to its security requirements.

V. GOING FORWARD

In the first part of our research, we have systematically surveyed the state of the art concerning support for modelling and analysis of security at the software design level [1]. Based on the gaps that emerged from this survey, we have outlined the important characteristics of a practical security analysis methodology, and we have provided a proof of concept of such a methodology. In the remainder of our research track, we intend to gradually convert this proof of concept into a mature methodology, which will involve a diverse set of activities.

Currently, our catalogue only supports a small subset of the intended security concerns, and will be extended with new security concerns. Also, the relationships between the elements in the catalogue and the transformations performed by the verifier needs to be proven correct and sound. Although the methodology is perfectly usable without such proofs, it would not provide any formal guarantees about the correctness of the analysis results. As a requirement to construct these proofs, the common vocabulary must be explicitly formalised.

Furthermore, we intend to increase the level of detail of both the models and the provided analysis. First of all this involves replacing the currently implicit attacker with an explicit attacker model, in which the developer can define the abilities of the intended attacker enabling a more fine-tuned analysis. Besides that, additional information about the elements in the catalogue should be specified and taken into account. For example, merely modelling that data is encrypted provides only a limited amount of information; other factors, such as the algorithm, are worth considering as well. This will require extensions to the modeller and catalogue, and the addition of new verifiers.

Simultaneously, we want to expand the feedback offered to the designer beyond merely signalling that the concern is satisfied or a violation was found. For example, the designer can be assisted with resolving the violations by presenting (or even automatically applying) possible resolutions. Furthermore, to guide the further development of the system, the security model can be annotated with conditions that are essential to preserve the security of the design.

Orthogonal to the aforementioned aspects, our methodology must be thoroughly evaluated to provide evidence that it actually helps software designers. Hence, we need to evaluate the methodology when executed by actual practitioners. One option for such an evaluation is a controlled empirical user study where students play the role of the practitioners. Such

empirical studies have already been successfully performed within our research group in the past [22], [23], [24], showing that they are a both feasible and useful evaluation method.

REFERENCES

- [1] A. van den Berghe, R. Scandariato, K. Yskout, and W. Joosen, "Design Notations for Secure Software: A Systematic Literature Review," *Software and Systems Modeling*, Submitted.
- [2] D. Basin, M. Clavel, J. Doser, and M. Egea, "Automated analysis of security-design models," *Information and Software Technology*, vol. 51, no. 5, pp. 815–831, 2009, SPECIAL ISSUE: Model-Driven Development for Secure Information Systems.
- [3] K. Sohr, G.-J. Ahn, M. Gogolla, and L. Migge, "Specification and Validation of Authorisation Constraints Using UML and OCL," in *Computer Security ESORICS 2005*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, vol. 3679, pp. 64–79.
- [4] L. Yu, R. France, Indrakshi Ray, and S. Ghosh, "A Rigorous Approach to Uncovering Security Policy Violations in UML Designs," in *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, 2009, pp. 126–135.
- [5] M. Koch and F. Parisi-Presicce, "UML specification of access control policies and their formal verification," *Software and Systems Modeling*, vol. 5, pp. 429–447, 2006.
- [6] D. Xu and K. E. Nygard, "Threat-Driven Modeling and Verification of Secure Software Using Aspect-Oriented Petri Nets," *Software Engineering, IEEE Transactions on*, vol. 32, no. 4, pp. 265–278, april 2006.
- [7] J. Kong, D. Xu, and X. Zeng, "UML-BASED MODELING AND ANALYSIS OF SECURITY THREATS," *International Journal of Software Engineering and Knowledge Engineering*, vol. 20, no. 6, pp. 875–897, sep 2010.
- [8] G. Georg, I. Ray, K. Anastasakis, B. Bordbar, M. Toahchoodee, and S. H. Houmb, "An aspect-oriented methodology for designing secure applications," *Information and Software Technology*, vol. 51, no. 5, pp. 846–864, 2009.
- [9] H. Hu and G.-J. Ahn, "Constructing Authorization Systems Using Assurance Management Framework," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 40, no. 4, pp. 396–405, july 2010.
- [10] J. Jürjens, *Secure Systems Development with UML*. Springer, 2004.
- [11] K. Buyens, R. Scandariato, and W. Joosen, "Least privilege analysis in software architectures," *Software and Systems Modeling*, pp. 1–18, 2011.
- [12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [13] E. Fernandez-Buglioni, *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.
- [14] A. V. Uzunov, E. B. Fernandez, and K. Falkner, "Securing distributed systems using patterns: A survey," *Computers & Security*, vol. 31, no. 5, pp. 681–703, 2012.
- [15] L. Dai and K. Cooper, "Using FDAF to bridge the gap between enterprise and software architectures for security," *Science of Computer Programming*, vol. 66, no. 1, pp. 87–102, 2007.
- [16] T. Heyman, "A Formal Analysis Technique for Secure Software Architectures," Ph.D. dissertation, KU Leuven, March 2013.
- [17] A. Van Lamsweerde, *Requirements engineering: from system goals to UML models to software specifications*. Wiley, 2009.
- [18] Standard, "PCI DSS v3.0," <https://www.pcisecuritystandards.org>, November 2013.
- [19] —, "The Common Criteria: Security functional components," <https://www.commoncriteriaportal.org>, September 2012.
- [20] Sirius, <https://www.eclipse.org/sirius/>.
- [21] G. J. Holzmann, *The SPIN MODEL CHECKER - PRIMER AND REFERENCE MANUAL*. Addison-Wesley, 2004.
- [22] K. Yskout, R. Scandariato, and W. Joosen, "Does organizing security patterns focus architectural choices?" in *34th International Conference on Software Engineering (ICSE 2012)*, vol. 1. IEEE, 2012, pp. 617–627.
- [23] K. Wuyts, R. Scandariato, and W. Joosen, "Empirical evaluation of a privacy-focused threat modeling methodology," *The Journal of Systems and Software*, vol. 96, pp. 122–138, October 2014.
- [24] K. Yskout, R. Scandariato, and W. Joosen, "Do security patterns really help designers?" in *37th International Conference on Software Engineering (ICSE'15)*, 2015.