Transforming security audit requirements into a software architecture

Koen Yskout, Bart De Win, and Wouter Joosen

IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium*
first.last@cs.kuleuven.be

Abstract. In this paper, an approach for automated transformations from a security requirements model to a consistent architectural model is presented. The approach can be used with an existing architectural model, and allows input from the architect to be taken into account. The transformation from audit requirements into a UML model is implementated using QVT and Eclipse EMF, and is illustrated by means of a small case study.

1 Introduction

Security is, more than ever, an important software quality. Nevertheless, it is hard to build a secure application, as demonstrated by the vast amount of security advisories, patches and updates published regularly. Of all security flaws, architectural ones are the most difficult and costly to correct once the system has been produced and deployed. On the other hand, during architectural design, modifying the architecture is cheap.

In this paper, we are concerned with the transition from security requirements to software architecture. This transition is interesting, because it crosses the semantic gap between the problem and solution domain. Correctly implementing security requirements across this gap is not an easy task, though. Therefore, we propose to apply automated, model-driven techniques, supporting the architect as much as possible in this process. The envisaged approach aims at automatically transforming security requirements into a consistent architecture, and consists of three main parts:

1. the definition of a suitable security requirements metamodel, in which security requirements can be modeled. Such metamodel is different from a functional requirements model, in that it needs to have security-specific semantics. Also, it needs to lie close enough to the architectural domain to make an automated transition possible. A first metamodel specifically for this purpose was already presented in earlier work [1].
2. the creation of a set of transformations from a security requirements model to an architectural solution. The existence of these transformations is crucial

* This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

for the success of the approach. The transformations should be able to take an existing architecture into account, and the architect should still be able to influence parts of them. This paper focuses on such a transformation from audit requirements to architecture.

3. a methodology for the verification of the correspondence of the generated architecture to the requirements. The success of the approach is proportional to the degree of assurance it can offer on the correctness of the end result. This part is still an open challenge, that will be addressed in future work.

Note that the approach focuses on security requirements that directly influence the architecture of the application, for example auditing or authorization requirements. Security requirements that aim at improving design or code quality (e.g., sanitize all input, check for buffer overflows, ...) are not considered. Also, we do not focus on the modeling of security properties of cryptographic protocols. Finally, we will focus on distributed applications, although the approach is more generally applicable.

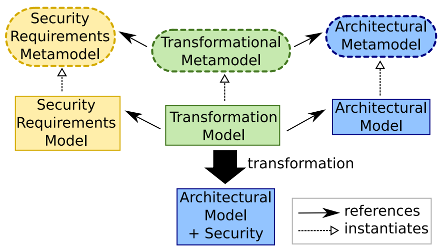
The rest of the paper is organized as follows. In Section 2, some general considerations regarding transformations suitable for this approach are given. Section 3 subsequently outlines the implementation of transformation from audit requirements to architecture, and is illustrated by an example in Section 4. Related work is given in Section 5, followed by a discussion on the approach in Section 6. Conclusions and future work are highlighted in Section 7.

2 Transformations

Each transformation will depend on both its source and target metamodels. For the transformations of interest in this paper, the source metamodel is a security requirements metamodel. The target metamodel is the metamodel of some architectural description method. The architectural metamodel can follow any paradigm (e.g., component-based), and could (but does not need to) have special constructs for dealing with security. In this paper we use UML as the target metamodel.

The simplest transformation takes a security requirements model as its input and generates a corresponding architecture. The architecture is based purely upon the information contained in the source model. However, transformations created in this way are too rigid to be used in practice, since they cannot use an existing architectural model, and they do not allow input from the architect.

This problem can be overcome by adding extra input to the transformation. Rather than extending the security requirements metamodel, we introduce a 'transformational' metamodel, which is defined separately for each transformation. This metamodel decorates the original security requirements metamodel: it references the original metamodel elements, but adds transformation-specific options. For instance, when multiple techniques can be used to implement a requirement, the transformational model allows the architect to select the technique he considers as the best given the situation. Also, the transformational model can



Security Requirements Metamodel
Transformational Metamodel
Architectural Metamodel
Security Requirements Model
Transformation Model
Architectural Model
transformation
Architectural Model + Security
references
instantiates

Fig. 1: Relations between the models and metamodels.

Requirements	UML
Agent (internal)	Component
Operation	Operation in Interface
Entity	Class in Component
Attribute of an Operation	Parameter
Attribute of an Entity	Property or Association
Attribute of an Agent	Property
DomainType	DataType or Class

Fig. 2: Possible mapping between requirement model types and UML types.

contain references to elements of the pre-existing target model, hereby providing the mapping between requirements and architectural elements. A graphical overview of the relations between the various models and metamodels is given in Figure 1.

Note that the transformational metamodel may want to impose certain constraints on these references, in order to restrict the set of allowed target elements. For example, the transformation could rely on these elements to be of a certain type, have certain properties, etc. These constraints represent assumptions made by the implementation, which can of course be relaxed or removed by more elaborate transformations.

Most of the constraints follow the same pattern: they state that a certain element from the requirements model will correspond to an element of a given type in the UML model. For instance, a rule can require that each agent element from the requirements model is represented as a UML component. In the transformational model, it is then possible to specify the corresponding component for each agent; if none is specified, a new component can be created.

3 Implementation

This section presents the implementation of a transformation from audit requirements to an architecture described in a UML model. First, the used technologies are outlined, followed by a more detailed discussion of the implemented transformation.

3.1 Technology choices

We have implemented the proposed approach as an Eclipse plug-in, as a proof of concept. All used metamodels (i.e., the security requirements metamodel plus the transformational metamodel) are specified using the Eclipse Modeling Framework (EMF). As a benefit, this allows the instant generation of a minimal, but usable, GUI editor for creating and viewing instances of the metamodels.

For the description of the transformations, the QVT Operational Mapping Language is used. The actual implementation is performed using the Eclipse implementation of QVTO.

Finally, UML is used as the target model for demonstration purposes. We generate an instance of the UML2 metamodel using the implementation of the Eclipse UML2 project, and use an external tool to visualize the result as a UML diagram. Note that a UML model usually contains much more information than the information that can be visualized on a diagram. It should be stressed that we aim for the generation of a fully consistent UML model; visualization is only a last (and currently manual) step.

3.2 Implementation overview

This section elaborates a possible implementation of a transformation for audit requirements. Our implementation makes the following assumptions:

1. Mappings between the requirement model and UML are performed as shown in Figure 2 and described in [1].
2. The behavior of UML operations is expressed using activities.
3. Audit logs are represented by classes, containing an operation for each type of event they can log.

These assumptions are restricting the applicability of the transformation, but they could be relaxed (within certain bounds). For example, in the current implementation, it would be relatively straightforward to represent an agent by more than one component.

The transformational metamodel for this implementation corresponds largely to the Audit Data Generation (FAU_GEN.1) component of the Common Criteria (CC) [2]. It offers the following customization options.

Audit levels can be assigned to each logged operation. The audit levels are the ones defined in the CC, i.e., 'minimal', 'basic', 'detailed' and 'unspecified'.

The **outcome** option allows to specify whether success and/or failure of the operation needs to be audited. It also provides the possibility to specify the UML actions from the architecture that denote success and failure.

The **information** of the operation that needs to be logged. This allows some parameters to be excluded from logging.

The **strategy** option offers a choice between logging strategies. Currently, the supported strategies are the use of explicit log calls, and the use of an audit interceptor. These strategies are detailed in the next two paragraphs. As a side note, notice that the available strategies are heavily influenced by the target metamodel. For example, in an aspect-oriented architecture, before and after advice are the natural candidates for invoking the auditing behavior.

Explicit log calls This strategy represents the most straightforward way of implementing auditing. It prescribes the adaptation of the behavior of each operation that has to be logged, by inserting a call to a logging operation at the start and at the end, when necessary. These logging operations are defined in a log interface, and implemented by a log class. A reference to an instance of the log class is added to the component corresponding to that agent. Explicit log calls do not allow any variability at runtime.

Audit interceptor For a more flexible approach, the audit interceptor strategy can be used. This strategy implements the Audit Interceptor security pattern [3]. It introduces three classes: (1) the audit interceptor, which gets invoked whenever a security-sensitive operation is invoked; (2) the event catalog, which determines whether the event needs to be logged or not based on its runtime configuration; and (3) the audit log itself. The pattern does not specify the mechanism by which operations are intercepted. Given that UML is our target metamodel, we can use UML signals for this purpose. In that case, it is assumed that a signal object is broadcast that represents the invocation of an operation (a functionality which could be provided by middleware, for instance). The audit interceptor then listens for signals it is interested in, and triggers its behavior upon reception.

Due to space constraints, it is impossible to discuss the complete implementation of the transformation. In its entirety, it comprises over 900 lines of QVT. This may seem excessive, but only about 50% is specific to the audit requirements. Around 40% of the code provides support for the core security requirement model, and roughly 10% provides helper functions for working with UML models. Thus, a large part can be reused in the definition of mappings for other classes of security requirements. Also, since transformations are written independently of any application, they should be written only once and can subsequently be applied multiple times.

4 Example

In this section, we will illustrate the approach with a small example. As a case study, we use a design for an ATM (Automated Teller Machine) of a bank, inspired by [4]. We will apply the system's audit requirements to an existing architecture described as a UML model.

The system to be designed is the ATM's main controller. It will be connected to, a.o., a magnetic stripe reader, an input/output console, a cash dispenser, a printer, and the bank's ATM network. We consider the operations that can be carried out through the customer console, i.e., withdraw, transfer, or deposit money, and a balance inquiry. Furthermore, an operator can start and stop the ATM via a separate, administrative console. We will implement two audit requirements:

R1 For each operation performed by a customer, an audit record needs to be generated at the ATM machine.
R2 The starting and stopping of the ATM machine needs to be audited.

The security requirements model corresponding to this case study contains an internal agent element representing the ATM controller. This agent is responsible for the following operations: 'start', 'stop', 'inquire balance', 'withdraw', 'transfer' and 'deposit', each with relevant parameters. The audit requirement instances require the auditing of each of these operations to an audit log provided by the ATM controller itself.

Element type	Element name	Property	Value
Agent	ATM Machine	Target component	ATM Controller (from UML model)
Operation	Withdraw	Target operation	withdraw (from UML model)
Operation	Start	Target operation	start (from UML model)
		Signal	start (from UML model)
Audit Requirement	R1	Audit level	Basic
		Strategy	Explicit Log Call
		Log success	true
		Success actions	Dispense cash (from UML model)
Audit Requirement	R2	Audit level	Minimal
		Strategy	Audit Interceptor

Fig. 3: A subset of the elements, their properties and values in the transformational model for the ATM case.

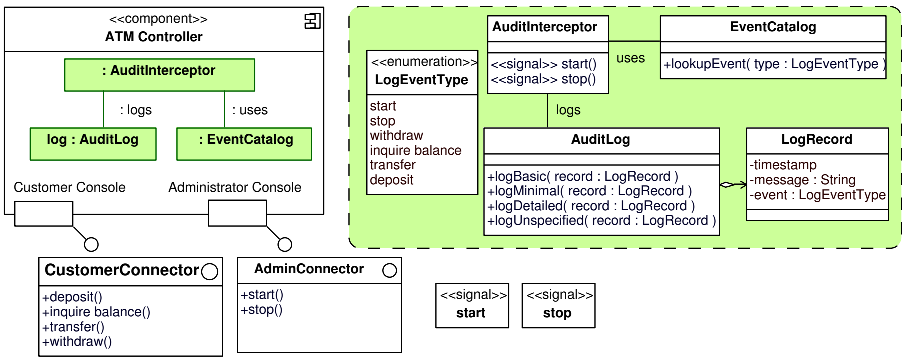
For the example, we choose to log the start and stop operations via an audit interceptor using UML signals, because these signals are already present in the existing architecture. Both operations are logged on the 'minimal' level. The customer's operations will use explicit log calls, and are logged at the 'basic' level. We will include all parameters of these operations in the log record, and log both success and failure.

Some elements and their properties in the transformational model for the first requirement are shown in Figure 3. Figure 4 shows some views of the generated architecture. The elements in green are added or modified by the transformation. Not all output of the transformation is visible on the diagrams, however. An invisible, yet crucial property of the result is the internal consistency of the UML model, e.g., actions only use inputs available to them, and all required and provided interfaces match. As a consequence, all views generated from the model will be mutually consistent.

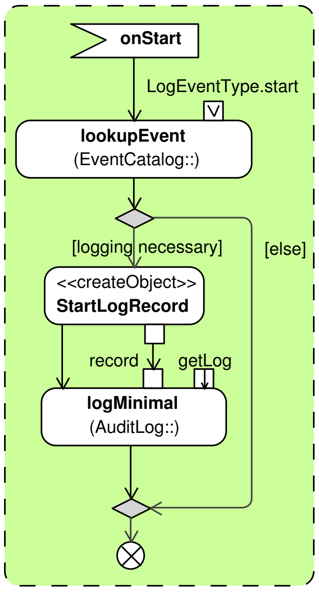
5 Related work

The work related to this paper can be divided in four groups. First, work on modeling security requirements has to be considered. The goal-oriented requirements approach KAOS [5] has extensions for eliciting security requirements. The resulting security requirements are modeled using the general KAOS metamodel. However, this prevents the inclusion of additional semantic information that is necessary to define an automated transformation. The Secure Tropos extension to Tropos [6] formally introduces and refines the notions of ownership, delegation and trust between agents.

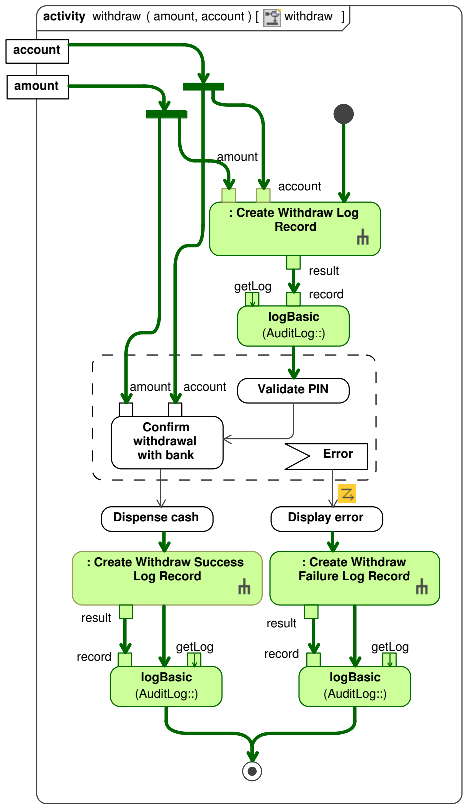
The second group of related work deals with the transition of requirements to architecture. Riemenschneider [7] gives initial ideas on refining architectures based on KAOS. This idea is further refined and illustrated in [8]. Brandozzi [9] introduces an intermediate language (called the 'architecture prescription language') that lies in between requirements and architecture. By means of this language, the translation between the problem and solution domain is made easier. A similar idea is proposed in [10]. Finally, Schmidt [11] presents an extension to the problem frames approach, utilizing patterns to preserve usability



(a) Structural view.



(b) Behavioral view: interception.



(c) Behavioral view: explicit log calls.

Fig. 4: Part of the architecture after transformation.

and security quality characteristics from requirements analysis to architecture. In contrast to our approach, none of these approaches are automated. Nevertheless, they provide valuable inspiration and guidelines for transformation creators.

A third, and large, part of related work to this approach deals with enriching architectural metamodels with security concepts. Considerable work has been performed extending UML with security concepts, i.e., to make the specification and verification of security properties easier. A widely acknowledged method in this area is UMLsec [12], which defines a UML profile and formal semantics for various security properties. The approach has also been applied to auditing [13]. Rodríguez [14] presents a UML 2.0 extension to incorporate security requirements into activity diagrams. Abie [15] presents a language with formal semantics to express communication-related security requirements. An integration of the language with UML is also provided. SecureUML [16] integrates the specification of role-based access control with UML, allowing transformations to access control infrastructures. All mentioned works focus on a particular metamodel (usually UML). Our approach, however, envisages the creation of transformations from the same security requirements metamodel to different architectural metamodels. This makes each of the aforementioned extensions a suitable target metamodel for our approach.

Finally, work has been performed on the preservation of security properties with transformations. In [17], Santen presents a formal framework to reason about the preservation of confidentiality requirements under refinement. Seehusen [18] describes an approach to analyze and refine UML interaction diagrams, preserving secure information flow properties.

6 Discussion

The proposed approach exhibits some interesting benefits for an architect using it.

Although the approach as discussed up till now made use of an existing architectural model, it can also be used to generate an architecture from scratch. Of course, since the security requirements model contains only elements necessary for requirements, the architecture will be incomplete. However, this allows the architect to experiment and get familiar with different strategies and options, without having to specify a complete mapping with an existing architecture.

The use of transformations implicates another appealing feature: traceability. Indeed, every creation or modification of an element by the transformation generates a trace from the source to the target elements. By means of these traces, it can be determined which elements were added or modified by a requirement. Also, they can be used to analyze which requirements were implemented in the architecture, and which were not.

Finally, the approach does not prescribe a fixed set of metamodels and transformations to be used. The security requirements metamodel is extensible, and no architectural metamodel is insisted upon. For the transformations, a major differentiation point between them will be their flexibility. A transformation (and its accompanying transformational model) should be easily customizable by the architect. Beware, however, that the aim is not to create one single, ideal transformation that offers any imaginable option.

The main drawbacks and shortcomings of the approach in its current form are the following.

First, the approach cannot (and will never be able to) deal with each possible security requirement. As mentioned earlier, it only focuses on security requirements that directly influence the architecture of the system. Also, some security requirements or its solutions are too specific or require too much input from the architect to be dealt with in an automated manner.

Second, possible interactions between security requirements are not taken into account. For instance, when access control and auditing are combined, some interesting questions arise. Should a request be audited before or after access control is performed? Or, how to deal with access control for inspecting the audit logs? In our opinion, the answers to these questions should be provided as part of the security requirements, instead of being (silently) handled by the transformations themselves.

Last, the end result should be verified for correctness. Two possible strategies could be followed here. On one hand, the correctness of a transformation could be proven, i.e., it preserves the security properties expressed by the requirements, no matter what (valid) input it is given. This proof should only be created once for each transformation. However, since transformations can quickly become very complex, this will probably not be feasible for each transformation. On the other hand, the correctness of the end result could directly be proven, i.e., prove that the result guarantees a number of properties defined by the requirements. This proof should be created over and over for each individual end result. It also requires that the target model is formal enough to reason about it. When pursuing this track, UMLsec [12] should be mentioned as a natural and promising target model.

7 Conclusion and future work

In this paper, an approach is presented delivering an automated transformation from security requirements to architecture. Transformations start from a security requirements model, and can be defined using any architectural metamodel. The approach can be applied to an existing architecture, and allows input from the architect. So far, the approach is implemented for transforming audit requirements to UML. An illustration by means of a case study is provided.

The approach still needs considerable extension in the future, both conceptual and implementation-wise. The security requirements metamodel needs to be extended with other types of requirements, e.g., confidentiality and integrity. Furthermore, a method to validate the correctness of the transformations or end result needs to be determined. Concurrently, more transformations can be implemented, possibly using different architectural metamodels. Finally, the approach should be validated more thoroughly in an industrial case study.

References

1. Yskout, K., Scandariato, R., Win, B.D., Joosen, W.: Transforming security requirements into architecture. In: Third International Conference on Availability, Reliability and Security, 2008 (ARES 08). (2008) 1421–1428
2. Common Criteria: Common criteria for information technology security evaluation, v3.1. Part 2: Security functional components (September 2007) http://www.commoncriteriaportal.org/thecc.html.
3. Steel, C., Nagappan, R., Lai, R.: Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management. Prentice Hall (2005)
4. Bjork, R.C.: An example of object-oriented design: An atm simulation http://www.cs.gordon.edu/courses/cs211/ATMExample/.
5. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. Proceedings of the 5th IEEE International Symposium on Requirements Engineering (2001) 249–263
6. Giorgini, P., Massacci, F., Mylopoulos, J., Zannone, N.: Modeling Security Requirements Through Ownership, Permission and Delegation. Volume 5. (2005)
7. Riemenschneider, R., Dutertre, B., Stavridou, V., van Lamsweerde, A.: From system requirements to system architecture. In: Proceedings of ISAW-4 - Fourth International Software Architecture Workshop, Limerick. (June 2000)
8. van Lamsweerde, A.: From System Goals to Software Architecture. Formal Methods for Software Architectures (2003) 25–43
9. Brandozzi, M., Perry, D.: Transforming Goal-Oriented Requirement Specifications into Architecture Prescriptions. Workshop From Software Requirements to Architectures (STRAW'01) at ICSE (2001)
10. Grünbacher, P., Egyed, A., Medvidovic, N.: Reconciling software requirements and architectures with intermediate models. Software and System Modeling **3**(3) (2003) 235–253
11. Schmidt, H., Wentzlaff, I.: Preserving Software Quality Characteristics from Requirements Analysis to Architectural Design. Proceedings of the European Workshop on Software Architectures (EWSA) **4344** (2006) 189–203
12. Jürjens, J.: Secure Systems Development With UML. Springer (2004)
13. Jürjens, J.: Modelling audit security for smart-card payment schemes with UMLsec. In Dupuy, M., Paradinas, P., eds.: Trusted Information: The New Decade Challenge, Kluwer Academic Publishers (June 2001) 93–108 Proceedings of SEC 2001 – 16th International Conference on Information Security.
14. Rodríguez, A., Fernández-Medina, E., Piattini, M.: Towards a UML 2.0 Extension for the Modeling of Security Requirements in Business Processes. 3rd International Conference on Trust, Privacy and Security in Digital Business (TrustBus), Krakow-Poland (2006) 51–61
15. Abie, H., Aredo, D., Kristoffersen, T., Mazaher, S., Raguin, T.: Integrating a Security Requirement Language with UML. UML 2004-The Unified Modelling Language: Modelling Languages and Applications **3273** 350–364
16. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-Based Modeling Language for Model-Driven Security. UML **2460** (2002) 426–441
17. Santen, T.: Preservation of probabilistic information flow under refinement. Information and Computation (2007)
18. Seehusen, F., Stølen, K.: Information flow property preserving transformation of UML interaction diagrams. In: Proceedings of the eleventh ACM symposium on Access control models and technologies, ACM Press New York, NY, USA (2006) 150–159