Using Model Driven Security Approaches in Web Application Development

Christoph Hochreiner[1], Zhendong Ma[3], Peter Kieseberg[1], Sebastian Schrittwieser[2], and Edgar Weippl[1]

[1] SBA-Research, Austria
{chochreiner,pkieseberg,eweippl}@sba-research.org
[2] St. Poelten University of Applied Sciences, Austria
sebastian.schrittwieser@fhstp.ac.at
[3] Austrian Institute of Technology
zhendong.ma@ait.ac.at

Abstract. With the rise of Model Driven Engineering (MDE) as a software development methodology, which increases productivity and, supported by powerful code generation tools, allows a less error-prone implementation process, the idea of modeling security aspects during the design phase of the software development process was first suggested by the research community almost a decade ago. While various approaches for Model Driven Security (MDS) have been proposed during the years, it is still unclear, how these concepts compare to each other and whether they can improve the security of software projects. In this paper, we provide an evaluation of current MDS approaches based on a simple web application scenario and discuss the strengths and limitations of the various techniques, as well as the practicability of MDS for web application security in general.

1 Introduction and Related Work

Model Driven Engineering (MDE) has gained a lot of attention during the past few years. The rise of modeling languages, especially UML, drove the development of MDE techniques as well as more and more sophisticated tool support for the automated generation of code. One of the most important motivations for applying MDE techniques is software correctness. Generally, software defects can result from two sources during the software development process: First, problems can originate from bad design decisions in the planning phase of the software development process. This type of defects, often referred as flaws, is fatal as elimination of the fundamental design misconceptions in later phases of the development process may require a general overhaul of the entire architecture. Modeling techniques can support development in this early design phase. The second type of defect is based on implementation errors (bugs). Even if the software was designed to work correctly, the actual implementation can introduce errors which led to the development of tools for automated code generation. In this case, the availability of automated tools that allow the translation of the abstract model into code that can be compiled or directly interpreted by a machine is of crucial importance. Furthermore, techniques such as model validation, checking and model-based testing can be used to support the reliability of a program in reference to its model.

With the success of MDE approaches the idea of bringing these concepts to the security domain was raised by the scientific community almost a decade ago [3,6]. The basic idea is similar to MDE: The process of modeling security aspects of a software project should enhance its quality - in this case related to security. The theoretical consideration is to deal with the same two categories like in MDE (flaws in the design and the implementation phase) by modeling the security requirements before the implementation. Design-based vulnerabilities can be addressed with model checking techniques and goal oriented system analysis and the number of implementation errors can be reduced by using automated code generation for sensitive, security-related parts of the software.

In the last years, a vast amount of different techniques for Model Driven Security (MDS) in software applications has been developed. The main purpose of this paper lies in providing a novel comparison of several major modeling approaches for designing secure software based on the example of a simple web application. In particular, we not only wanted to analyze how typical mistakes in web application scenarios could be described by security modeling techniques but also if these techniques actively push the developer towards a more secure implementation by incorporating security essential within the modeling process. In contrast to MDE, the modeling of security is heavily influenced by the open world assumption. Security aspects, as being non-functional requirements of a software project, can be left out of the model and the implementation without having direct influence on the functionality of the software. We strongly believe that the benefit of security modeling techniques is limited, if their sole purpose is to offer the possibility of modeling security aspects without actually enforcing them. In 2011 Kasal et al. [4] provided a taxonomy evaluation of different state-of-the-art approaches for model driven engineering. The taxonomy was proposed purely theoretically, still, to the best of our knowledge, there has not been a structured practical comparison of the actual techniques with respect to implementing a real-life scenario. Our work is focused towards the practical applicability and effectiveness of model driven engineering approaches such as Lloyd and Juerjens [5] did when they applied the UMLsec and JML approaches to practically evaluate a biometric authentication system. The main contributions of this paper can be defined as follows: We show what types of common threats in web application scenarios can be modeled and to what degree corresponding security measures are enforced by the different modeling techniques. Furthermore, we provide the analysis of our experimental assessment of current security modeling techniques based on a typical web application scenario. Additionally, we discuss the practicability of MDS for the secure development of web applications.

2 Methodology

2.1 Evaluation Scenario

For our evaluation we designed a typical basic web application scenario, which covers the threats outlined by the Open Web Application Security Project (OWASP) in their 2010 published version of their TOP 10 list [9]. This allows us to evaluate the modeling techniques and compare their functionality. In detail, the scenario consists of three machines: A client accesses a web server that is connected to a database server. On the web service, there exist two different user roles, normal user and administrator, which have different access permissions regarding the database server. Figure 1 shows the basic scenario. Please note that the model in the figure does not follow the concepts of any common modeling language in order to be formulated as neutral as possible before modeling the scenario with different MDS approaches. In this simple use case, the threats of the OWASP Top 10 can be identified (see [9]).

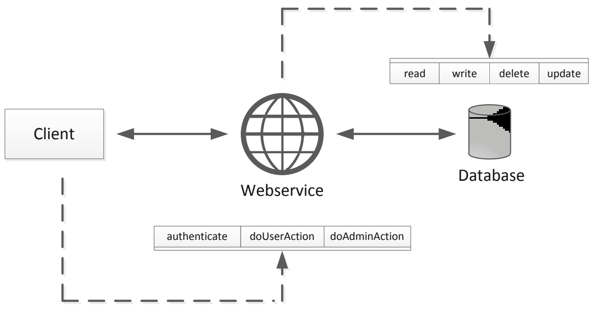

Fig. 1. Simple Web Application Scenario

3 Selection of Methods

In this section, we evaluate the possibility of modeling the threats of the OWASP Top10 with different MDS approaches. We give a short introduction on each concept and then model our web application use case with respect to the OWASP threats.

3.1 UML Based

UML [10] is a widely used model notation method for analyzing software system objects. Several diagrams are defined to express different aspects of the systems from an abstract to an implementation perspective. Original UML notations have been extended to integrate non-functional system properties such as security or the threat environment in an explicit way. The extended UML diagrams allow the developer to model threats as well as countermeasures.

Secure UML. Secure UML is an extension of the standard UML specification that encapsulates the modeling aspect of Role Based Access Control (RBAC) to include security aspects [11]. It is a single purpose extension and solely allows for modeling the access control aspects of the example by adding roles, permissions and constraints on the method level to the existing syntax. The authors of Secure UML created a prototypical tool to automatically transform the model into an EJB (Enterprise Java Beans) based architecture incorporating all standard access controls and primitive comparison functions (e.g. $<, >, \neq \emptyset$), all other functions have to be implemented by the user. With these additions, the model can be transferred automatically into executable code, thereby taking care of the first two OWASP entries (A1) and (A2). SecureUML derives input validation [2] through implementation of a separate validation class which takes care of input content. RBAC is a fundamental part of SecureUML, access control restrictions for objects, databases and files are ensured, thus covering (A3) and (A4), furthermore, RBAC relates to URL access restriction, thus (A8) can be modeled. The Secure UML specification does not provide the functionality to model the aspects of transport security or the required logging of queries.

UMLsec. As an extension to the classical UML standard, UMLsec provides additional methods to model security aspects of software systems based on so-called *secure guards* resulting in models that are compatible to standard UML diagrams.

When applying the OWASP Top 10 threats, there are some aspects that can be prevented with proper UMLsec modeling. The first two threats, Injection (A1) and Cross-Site Scripting (A2), concern the data provided by the user. To prevent attacks on the web service based on this external input, every external input has to be checked. The threats concerning the Broken Authentication and Session Management (A3) cannot be dealt with proper modeling, because the authentication mechanism is encapsulated within the authenticate method and the evaluation of this functionality was omitted, because they are not in the focus of UMLsec. It is possible to model countermeasures against Direct Object Reference (A4), Cross-Site Request Forgery (CSRF) (A5), Failure to Restrict URL Access (A8) and Unvalidated Redirects and Forwards (A10) with secure guards. There have to be secure guards for every possible attack scenario. One example is a special guard that checks the feasibility of the called method to prevent CSRFs.

The terminal aspect that can be modeled with UMLsec, thus covering the problem of Insufficient Transport Layer Protection (A9). With UMLsec it is possible to tag specific communication paths with security requirements like encryption. Beside the aspects that can be modeled with UMLsec, there are some that cannot be taken care of with this engineering technique, including Security Misconfiguration (A6) and Insecure Cryptographic Storage (A7). It is not feasible to handle these two types of errors with model engineering techniques, because these techniques only cover the architecture of the program and not the deployment environment.

Misusecase. The misusecase specification is an extension to the use case specification of the UML use case diagram. This extension was developed by Guttorm Sindre and Andreas L. Opdahl [12] to describe malicious acts against a system, which are added to the normal use case diagram with inverted colors. Because of the high level of abstraction it is not possible to provide any tool support to generate code out of the use case diagram.

When applying the OWASP Top 10, we can identify some problems that can be covered with the misusecase diagram. The misusease diagram can model any attack like injection (A1), cross-site scripting (A2) or the failure to restrict URL access (A8). The issue of broken authentication (A3) can be tackled with the modeling of unauthorized actions, but the use case diagram cannot model any temporal or causal dependencies. The configurational aspects like security misconfiguration (A6) or insecure cryptographic storage (A7) as well as technical requirements like insufficient transport layer protection (A9) can thus not be covered with misusecase diagrams.

3.2 Aspect Oriented Software Development

Aspect oriented software development (AOSD) is an emerging approach with the goal of promoting advanced separation of concerns. The approach allows system properties such as security to be analyzed separately and integrated into the system environment.

Aspect Oriented Modeling. The framework proposed by Zhu et. al. [15] is designed to model potential threats on a system in an aspect-oriented matter. These additions are designed to model an attacker-and-victim-relation in different types of UML diagrams. Due to page limitations, in the evaluation we only describe the class diagram that already shows most of the additional features compared to standard UML specifications. The basis of the class diagrams is an abstract *attacker class* that provides basic attributes and methods.

This framework is applicable in the context of risk oriented software development. After a risk analysis of the system, all high impact attacks have to be identified and can subsequently be model. These models can be transformed into aspect-oriented code that is weaved into the existing code base. The code generator published by Zhu et. Al. is capable of producing AspectJ and AspectC++ code. These extensions to the standard UML specification are not practical enough in order to model basic security aspects like RBAC or transport layer security, they are only useful for handling specific attack scenarios and adding specific countermeasures to a given system. Still, in general it is possible to model all aspects of the OWASP Top10 using aspect oriented modeling.

SAM. Besides UML-based modeling approaches, there exist also some modeling techniques based on Petri nets and temporal logic, like the AOD framework proposed by H.Yu et al. [14] This framework is designed to model complex workflows and join them with security aspects. Nodes in the petri net represent single steps of the workflow and the security aspects handle the transitions between these nodes. The constraints for the workflow are modeled in a temporal logic that allows a formal verification of the system.

Protocol Checker. The AVISPA Tool for automated validation of Internet security protocols and applications is mainly concerned with verifying (cryptographic) protocols with respect to known vectors like man-in-the-middle- or replay-attacks. At the heart of AVISPA lies a definition language for protocols called HLPSL (High Level Protocol Security Language), which is specifically designed for modeling protocol flows together with security parameters and requirements. Furthermore, AVISPA provides four different analysis engines that can either be targeted at a problem separately, or together.

Another tool for analyzing synchronous as well as asynchronous protocols is the Symbolic Model Verifier (SMV), which is based on temporal logic. Models are specified in the form of temporal logic formulas, the tool is able to specify and handle finite automata and to check temporal logic formulas for validity. A speciality lies in the ability to handle asynchronous protocols and distributed systems. Still it is not possible to model executable software systems using SMV.

The modeling language Alloy is based on a first-order relational logic, its primary goal lies in the realm of modeling software designs. The logical structures of the systems are modeled using relations, existing properties are modeled by relational operators. Furthermore, Alloy provides the user with means for typing, sub-typing as well as type-checking on runtime and the building of reusable modules. The actual analysis is done by using the tool Alloy Analyzer which is based on a SAT-solver, since due to the construction of the language, the analysis of a model is basically a form of constraint solving.

Since these techniques aim at providing a detailed security analysis on the protocol level, using them for modeling whole software applications is not practically feasible, especially since they are not concerned with architectural decisions, but with the execution of actual protocols using cryptographic primitives. Still, they can be useful for analyzing cryptographic primitives or transport layer protocols, thus being a good strategy for thwarting insufficient transport layer protection.

3.3 Goal Driven Approaches

Goals are the objectives, expectations and constraints of the system environment. Goal driven approaches address the problems associated with business goals, plans and processes as well as systems to be developed or to be evolved in order to achieve organizational objectives. Goals cover different types of issues - both functional and non-functional. Goal models demonstrate how the different goals contribute to each other through refinement links down to particular software requirements and environmental assumptions. Functional goals focus on the services to be provided while non-functional goals are inked with the quality of services like security or availability.

KAOS. The KAOS model originates from the requirements engineering domain and was designed by researchers at the University of Lauvain and the University of Oregon. The name of the methodology KAOS stands for "Knowledge Acquisition in autOmated Specification" [13] and it describes a framework to model and refine goals including the selection of alternatives. The framework is supported by a software solution called *Objectiver* [1], which supports the developer in designing the goal models and refining them, as well generating object or operation models, but does not provide any code generation functionality. This modeling approach allows the developer to model all OWASP Top 10 threats as goals that can be further used for the requirements generation.

Secure Tropos. The Tropos methodology [1] supports the software development process by describing the environment of the system and the system itself. It is used to model dependencies between different actors that want to achieve different goals by executing plans. There are four different abstraction layers defined that describe different stages of requirements and layers of design. Secure Tropos [8] is an extension to the original Tropos methodology by adding security constraints and secure entities as well as the concepts of ownership, trust and dependency. The Secure Tropos methodology does not allow the designer to model any OWASP TOP 10 threat directly within the model, still there are some software solutions, like ScTro [2], that support the software engineer during the design and requirements analysis phase.

4 Evaluation

Secure UML. Beside the intention to use the constraints only for access restrictions and preconditions to these access restrictions like the UserAuthenticated constraint, it is possible to add more complex requirements to provide input validation as the application of the framework to our use case shows. In Figure 2 we have added the InputValidated constraint, which assures that the parameters do not contain any strings that can be used for XSS or SQL injections. The additional functionality to cover XSS and SQL injection checks has to be implemented by the user, since the tool only covers primitive comparison functionality for constraints. The Secure UML specification does not provide the functionality to model the aspects of transport security or the required logging of queries to the database.

UMLsec. This evaluation focuses on the class and the deployment diagram, because these two diagrams cover all security requirements of our simple web application scenario.

[1] `http://www.objectiver.com`
[2] `http://sectro.securetropos.org`

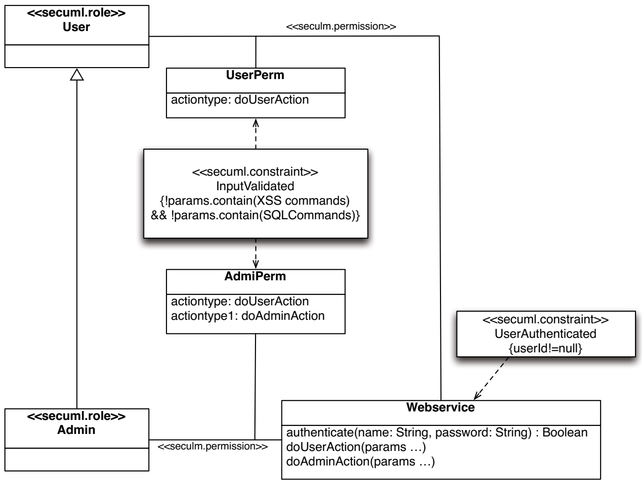

Fig. 2. Use case modelled with Secure UML

The aspect of transport security tangles the communication among the three components, as shown in Figure 1. The communication between the client machine and the application server is done over the Internet and therefore all service-calls and the resulting replies have to be encrypted. The communication between the application server and the database server is not that critical, especially because they are situated in the same local network. In this case it is enough to reduce the requirement to integrity instead of encryption. These two stereotypes are expressed with the UMLsec specification. The environments like Internet and LAN are added to the link between the systems and the calls are tagged with the required stereotypes. Although these transport requirements are easy to model, it is not feasible to automatically generate code ensuring compliance with these requirements, because these systems are too heterogeneous.

The two aspects *authentication* and *RBAC* are modeled within the class diagram. The UMLsec specification only supports class based access restrictions and it is necessary to extend the basic model with two additional classes (UserAction and AdminAction) to define user specific access control. These two classes are simple wrapper classes, which are annotated with two different guards. These two guards are called from the web service class and check if the current user has a specific role, which has been assigned to him by a successful authentication.

Due to this implicit mechanism, it is not necessary to model additional constraints, like that the user has to be authenticated. In [7] one can find a successful evaluation of how UMLsec properties can be transferred into actual code. The downside of this kind of modeling is that it does not scale well for additional roles and it increases the complexity of the model. The proper input validation is modeled with a secure dependency between the web service and the InputValidator which is called for every input, as shown in Figure 5. The model (Figure 4) that shows the usage of secure guards covers this scenario. These guards check, whether the users have enough privileges to perform actions.

The final aspect is the assertion that all queries get logged. This aspect is modeled with the secure dependency addition of UMLsec. By means of this addition it is possible to model the constraint that every call to a method that is provided by the database class is succeeded by the log method of the logger class. In this scenario every user could submit malicious input to the system, there has to be some input validation to prevent attacks like SQL-injection or XSS. This aspect is modeled using the secure dependency addition: Every input that is passed on to a method provided by the web service has to by checked for malicious input.

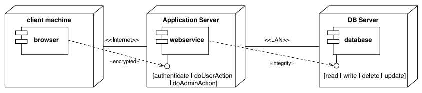

Fig. 3. Secure Links in UMLsec

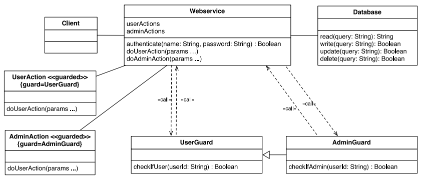
Client
Webservice
userActions
adminActions
authenticate(name: String, password: String) : Boolean
doUserAction(params ...)
doAdminAction(params ...)
Database
read(query: String): String
write(query: String): Boolean
update(query: String): Boolean
delete(query: String): Boolean
UserAction <<guarded>> {guard=UserGuard}
doUserAction(params ...)
AdminAction <<guarded>> {guard=AdminGuard}
doUserAction(params ...)
«call»
«call»
«call»
«call»
UserGuard
checkIfUser(userId: String) : Boolean
AdminGuard
checkIfAdmin(userId: String) : Boolean

Fig. 4. Secure Guards in UMLsec

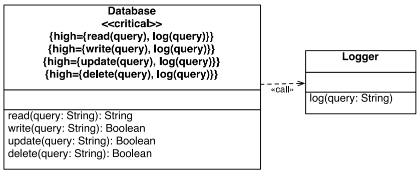
Database
<<critical>>
{high={read(query), log(query)}}
{high={write(query), log(query)}}
{high={update(query), log(query)}}
{high={delete(query), log(query)}}
read(query: String): String
write(query: String): Boolean
update(query: String): Boolean
delete(query: String): Boolean
«call»
Logger
log(query: String)

Fig. 5. Secure Dependency in UMLsec

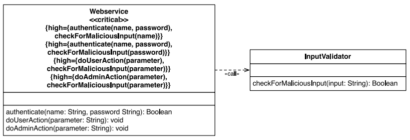
Webservice
<<critical>>
{high={authenticate(name, password), checkForMaliciousInput(name)}}
{high={authenticate(name, password), checkForMaliciousInput(password)}}
{high={doUserAction(parameter), checkForMaliciousInput(parameter)}}
{high={doAdminAction(parameter), checkForMaliciousInput(parameter)}}
authenticate(name: String, password String): Boolean
doUserAction(parameter: String): void
doAdminAction(parameter: String): void
«call»
InputValidator
checkForMaliciousInput(input: String): Boolean

Fig. 6. Input Validation in UMLsec

Misusecase. The use case diagram (Figure 7) shows the modeling of different threats to the system. The threats are carried out by the attacker indicated with an ordinary use case actor that has the background color black. The same applies to the misusecases in the diagram, that are ordinary use case elements with a black background.

The misusecase diagram provides the functionality to model high level threats that are executed by different actors of the system, but is does not provide the functionality to model any countermeasures or mitigation approaches. The only possibility to model countermeasures is to extend the existing use cases to implement organizational countermeasures, like additional permission checks.

Aspect Oriented Modeling. In our example, the attacker tries to tamper with the authentication using invalid input. This attack is modeled as an aspect that provides some methods to execute checks to prevent this attack, the remaining part of the diagram is a simplified representation of our basic UML class diagram. In the context of this framework it is feasible to omit all classes or methods that are not used in this attack, every diagram that is modeled within this framework visualizes one single attack.

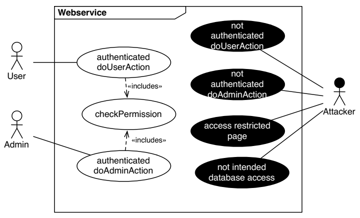
Webservice
User
Admin
Attacker
authenticated doUserAction
«includes»
checkPermission
«includes»
authenticated doAdminAction
not authenticated doUserAction
not authenticated doAdminAction
access restricted page
not intended database access

Fig. 7. Modeling of malicious acts with misusecase diagrams

Thus, the approach is only feasible for covering the most pressing topics like injections and XSS, since every possible attack needs to be modeled with respect to its effects on the system, which implies that all possible attacks need to be known beforehand. Furthermore, in case of real-life-size applications, the number of possible attack scenarios that need to be modeled separately will grow drastically.

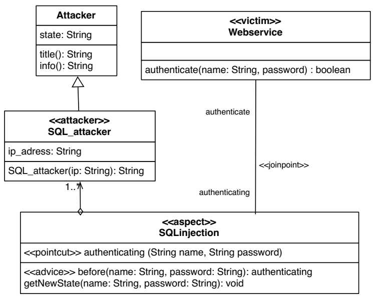
Attacker
state: String
title(): String
info(): String
<<victim>> Webservice
authenticate(name: String, password) : boolean
<<attacker>> SQL_attacker
ip_adress: String
SQL_attacker(ip: String): String
authenticate
<<joinpoint>>
authenticating
1..*
<<aspect>> SQLinjection
<<pointcut>> authenticating (String name, String password)
<<advice>> before(name: String, password: String): authenticating
getNewState(name: String, password: String): void

Fig. 8. Aspect Oriented Modeling

SAM. Due to the lack of complex workflows in our scenario, we omitted a detailed analysis of this framework. The single method calls do not trigger any workflows within the web service. Currently there is no tool support for this framework that provides automatic code generation, but this framework can be used in order to perform a detailed risk analysis of a complex workflow.

KAOS. The KAOS model itself starts at a high level that describes abstract requirements for the system, which are separated in functional and non-functional requirements, while the security requirements lie in the non-functional section as one can see in Figure 9 (i). Figure 9 (ii) shows a refinement of the secure system requirement, where most OWASP Top 10 issues can be modeled. Figure 9 (iii) shows a model for a concrete requirements model for the call of the method doAdminAction. This model already includes actors and specific requirements that are linked to the rather high level requirements like restricted access or authenticity. These goal models can be further used to generate object models, operation models or responsibility models to derive concrete software development requirements and restrictions.

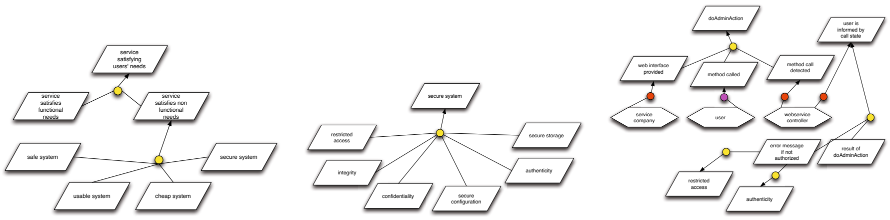

Fig. 9. Goal model: (i) Basic, (ii) refined, (iii) for a specific action

Secure Tropos. Figure 10 shows a simple dependency model with security constraints, which are modeled in the cloud shaped elements. It shows the three actors of the system, namely the user, admin and the service provider itself and the two plans that can be executed by the first two actors. The security constraints can be used to introduce requirements for actions between two actors, but none for systems, like a database server. This methodology is designed to model dependencies and trust relations within multiple stakeholders, but it is not feasible to apply this methodology to our evaluation scenario to improve the security of the system.

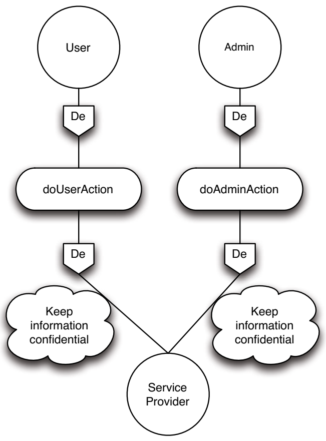

Fig. 10. Security constraints modeled with Tropos

Table 1. Summary of OWASP Top 10 mitigation coverage

OWASP Top 10	Secure UML	UMLsec	Misuse-case	Aspect Oriented	KAOS	Protocol Checker	Secure Troposker
Injection (A1)	✓	✓	✓	✓	✓	✗	✗
XSS (A2)	✓	✓	✓	✓	✓	✗	✗
Broken Auth. and Session Mgmnt. (A3)	✓	✗	✓	✓	✓	✗	✗
Insecure Direct Object Ref. (A4)	✓	✓	✓	✓	✓	✗	✗
CSRF (A5)	✗	✓	✓	✓	✓	✗	✗
Security Misconfiguration (A6)	✗	✗	✗	✓	✓	✗	✗
Insecure Cryptographic Storage (A7)	✗	✗	✗	✓	✓	✗	✓
Failure to Restrict URL Access (A8)	✓	✓	✓	✓	✓	✗	✗
Insufficient Transport Layer Protection (A9)	✗	✓	✗	✓	✓	✓	✓
Unvalidated Redirects and Forwards (A10)	✗	✓	✓	✓	✓	✗	✓
Toolsupport	✓	✗	✗	✓	✓	✗	✓

5 Conclusions

Most of the UML based modeling methodologies support the modeling of mitigation and countermeasures to the OWASP Top 10 threats, mostly by adding additional constraints on an implementation level. The misuse case diagram and the goal based approaches do not handle the implementation, they model a higher abstraction layer that shows real world interactions and requirements. Some threats can be described with these high level requirements, as can be see for the KAOS methodology. Apart from the Secure UML approach there is no feasible tool support to transform the actual models into source code that mitigates the mentioned threats, and these models can be rather used to identify potential security issues or potential collisions for conflicting goals. The aspect of detecting conflicts and resolving them is crucial for large systems that have several different stakeholders with conflicting requirements. The second major outcome of our evaluation is that model driven engineering does not make the software more secure in general by adding implicit mitigation procedures or checking the models for potential flaws, like the OWASP Top 10. These methodologies are only supposed to support the developers by indicating the location of conflicts, which can be done with goal based methodologies or the addition of standard mitigation features to existing systems, which can be done with the UMLsec and the Secure UML methodologies. Table 1 presents an overview about the capabilities of the evaluated methodologies. Overall it can be said that model driven engineering can reduce the occurrence of threats that are listed in the OWASP Top 10 by indicating them within the model, but this indication does not ensure that the software architect who designs the model, plans the appropriate countermeasures or mitigation features and that the actual implementation is compliant with the model.

Acknowledgements. This work has been supported by the Austrian Research Promotion Agency (FFG) under the Austrian COMET Program.

References

1. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An agent-oriented software development methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
2. Hayati, P., Jafari, N., Rezaei, S., Sarenche, S., Potdar, V.: Modeling input validation in uml. In: 19th Australian Conference on Software Engineering, ASWEC 2008, pp. 663–672. IEEE (2008)
3. Jürjens, J.: Umlsec: Extending UML for secure systems development. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 412–425. Springer, Heidelberg (2002)
4. Kasal, K., Heurix, J., Neubauer, T.: Model-driven development meets security: An evaluation of current approaches. In: 2011 44th Hawaii International Conference on System Sciences (HICSS), pp. 1–9. IEEE (2011)
5. Lloyd, J., Jürjens, J.: Security analysis of a biometric authentication system using UMLsec and JML. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 77–91. Springer, Heidelberg (2009)
6. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-based modeling language for model-driven security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
7. Montrieux, L., Jürjens, J., Haley, C., Yu, Y., Schobbens, P., Toussaint, H.: Tool support for code generation from a umlsec property. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 357–358. ACM (2010)
8. Mouratidis, H., Giorgini, P.: Enhancing secure tropos to effectively deal with security requirements in the development of multiagent systems. In: Barley, M., Mouratidis, H., Unruh, A., Spears, D., Scerri, P., Massacci, F. (eds.) SASEMAS 2004-2006. LNCS, vol. 4324, pp. 8–26. Springer, Heidelberg (2009)
9. OWASP. Open web application security project top 10, `https://www.owasp.org/index.php/Top_10_2010-Main` (last access: January 15, 2013)
10. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Pearson Higher Education (2004)
11. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-based access control models. Computer 29(2), 38–47 (1996)
12. Sindre, G., Opdahl, A.: Templates for misuse case description. In: Proceedings of the 7th International Workshop on Requirements Engineering, Foundation for Software Quality (REFSQ 2001), Switzerland. Citeseer (2001)
13. van Lamsweerde, A., Dardenne, A., Delcourt, B., Dubisy, F.: The kaos project: Knowledge acquisition in automated specification of software. In: Proceedings AAAI Spring Symposium Series, pp. 59–62 (1991)
14. Yu, H., Liu, D., He, X., Yang, L., Gao, S.: Secure software architectures design by aspect orientation. In: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2005, pp. 47–55. IEEE (2005)
15. Zhu, Z., Zulkernine, M.: A model-based aspect-oriented framework for building intrusion-aware software systems. Information and Software Technology 51(5), 865–875 (2009)