Automated Analysis of Permission-Based Security using UMLsec

Jan Jürjens[1]*, Jörg Schreck[2], and Yijun Yu[1]

[1] Computing Department, The Open University, GB
[2] O_2 (Germany), Munich

Abstract. To guarantee the security of computer systems, it is necessary to define security permissions to restrict the access to the systems' resources. These permissions enforce certain restrictions based on the workflows the system is designed for. It is not always easy to see if workflows and the design of the security permissions for the system fit together. We present research towards a tool which supports embedding security permissions in UML models and model-based security analysis by providing consistency checks. It also offers an automated analysis of underlying mechanisms for managing security-critical permissions using Prolog resp. automated theorem provers for first-order logic.

A commonly used security concept is permission-based access control, i.e. associating entities (e.g. users or objects) in a system with permissions and allowing an entity to perform a certain action on another entity only if it has been assigned the necessary permissions. Designing and enforcing a correct permission-based access control policy (with respect to the general security requirements) is very hard, considering the complex interplay between the system entities. This is aggravated by the fact that permissions can also be delegated to other objects for actions to be performed on the delegating object's behalf.

In this tool demonstration, we present a tool which supports the integration of permissions into early design models, in particular for object-oriented design using UML. We describe both static modelling aspects, where we introduce owned and required permissions and capabilities for their delegation into class diagrams, and dynamic modelling aspects. Dynamic modelling aspects are characterized by the use and delegation of permissions within an interaction of the system objects, modelled as a sequence diagram. To gain confidence in the correctness of the permission-based access control policy, we define checks for the consistency of the permission-related aspects within the static and dynamic models and between these models. Using a translation from the UML models to Prolog resp. to the input notation of a first-order predicate logic automated theorem prover, based on a formal semantics for the UML diagrams used, these

* `http://www.jurjens.de/jan` . This work was partially performed when this author was at TU Munich and is partly funded by the Royal Society through an international joint project with TU Munich on model-based security analysis of crypto-protocol implementations.

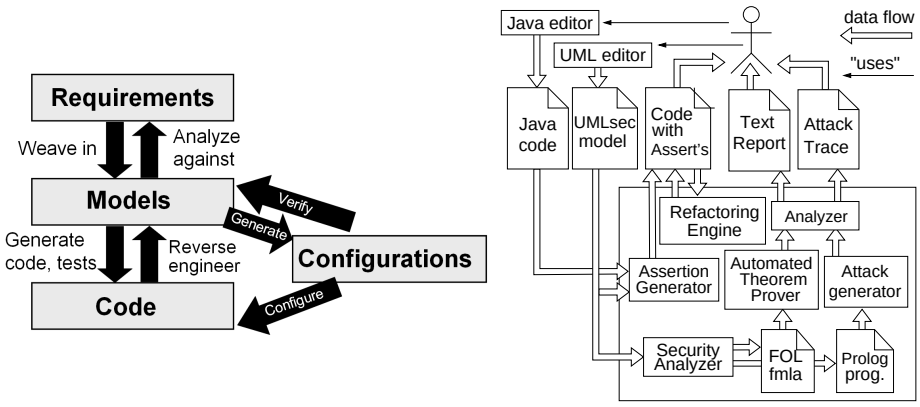

Fig. 1. a) Model-based Security Engineering; b) Model-based Security Tool Suite

permissions can then be verified for security properties, for example, the analysis of the correctness of authorization chains.

Security Analysis using UMLsec: Model-based Security Engineering (MBSE, [Jür04,Jür05]) is a soundly based approach for developing security-critical software where recurring security requirements (such as secrecy, integrity, authenticity) and security assumptions on the system environment can be specified either within a UML specification or within the source code as annotations (cf. Fig. 1a). Analysis plugins in the associated UMLsec tool framework [Too07] (Fig. 1b) generate logical formulas formalizing the execution semantics and the annotated security requirements. Automated theorem provers (ATPs) and model checkers automatically establish whether the security requirements hold. If not, a Prolog-based tool automatically generates an attack sequence violating the security requirement, which can be examined to determine and remove the weakness. Thus we encapsulate knowledge on prudent security engineering and make it available to developers who may not be security experts. Since the analysis that is performed is too sophisticated to be done manually, it is also valuable to security experts. Part of the MBSE approach is the UML extension UMLsec for secure systems development which allows the evaluation of UML specifications for vulnerabilities using a formal semantics of a simplified fragment of the UML. The UMLsec extension is given in form of a UML profile using the standard UML extension mechanisms. *Stereotypes* are used together with *tags* to formulate the security requirements and assumptions. *Constraints* give criteria that determine whether the requirements are met by the system design by referring to a precise semantics of the used fragment of UML.

The UMLsec tool-support in Fig. 1b) can then be used to check the constraints associated with UMLsec stereotypes mechanically, based on XMI output of the diagrams from the UML drawing tool in use [Too07,Jür05]. There is also a framework for implementing verification routines for the constraints associated with the UMLsec stereotypes. Thus advanced users of the UMLsec approach can use this framework to implement verification routines for the constraints of self-defined stereotypes. The semantics for the fragment of UML used for UMLsec is

defined in [Jür04] using so-called *UML Machines*, which is a kind of state machine with input/output interfaces and UML-type communication mechanisms. On this basis, important security requirements such as secrecy, integrity, authenticity, and secure information flow are defined.

Security permissions in UMLsec: The objects to which the access should be secured by permissions (which is for example supported by the Java Security Architecture, making use of the concept of Guarded Objects) are identified by marking classes in the class diagram at hand that define or own permission objects with the stereotype « permission − secured ». Whether an object owns certain permissions on other objects at instantiation time is stated as follows: A tagged value is associated with the « permission − secured » stereotype consisting of a list of pairs structured as follows: {permission = [(*class*, permission)]}. The first element of the pair indicates the class on which the permission is valid, the second names the permission. Methods and public attributes to which access is restricted are marked with the stereotype « permissioncheck » and an associated tagged value containing the list of permissions needed for access ({permission = [permission]}). The association to classes is given by the class implementing the method or containing the attribute. To allow objects of certain classes classified as reliable unrestricted access to particular methods and public variables, one can associate a second tag to the stereotype « permission_check ». The tagged value {no_permission_needed = [*class*]} indicates that objects of the named classes need no permissions for access. A permission is implemented as a message consisting of *permission* and *identifier* (of the object the permission is valid on). The object owning the permission will be specified by appending the object's public key. Therefore it is impossible for any other object to use this permission. A *certificate* is defined as a triple consisting of the identifier followed by the permission and the public key of the user of the *certificate*. For signing the permissions, there is a trusted instance in the system called security authority (SA). This instance releases all permissions and passes them on to the objects at their instantiation time. It is not possible to change the definition of a permission once signed by this authority. So a certificate defining a permission will be formally defined as follows: $Sign(identifier::permission::K_{legit}, {K_{SA}}^{-1})$.

For the specification of the workflow, a sequence diagram is created, allowing one to specify the connection between permissions and messages by regarding the exchange of messages between objects. First, we define which of the objects are permission-secured objects, using the same stereotype « permission − secured » as in the class diagram. To this stereotype, we attach the permissions the object owns on other objects, utilizing tagged values. These tags are defined the same way as in the class diagrams, by {permission = [(*object*, permission)]}. In contrast to the class diagram, here the first element of the pair means no longer a class but a concrete object on which the permission is valid. Permissions which are needed for executing a method are attached directly to the message which is to be protected by these permissions. If a message is protected by permissions, it is marked with the stereotype « permission_check », where the permissions are named as tagged values: {permission = [permission]}.

Delegation of permissions is stated by a tag as well: {delegation=[(class, permission, role/class)]}. It is executed by passing on certificates, which are formally defined as 7-tuples certificate $= (e, d, c, o, p, x, s)$ with emittent e, delegate d, class c of the delegate, object o, permission p which is valid on o, expiration timestamp x and sequence number s. In the sequence diagram, messages where permission certificates are sent are marked by the stereotype «certification», where a 7-tuple representing a certificate will be directly attached as a tagged value. The parameters of this tag correspond to the definition above. To implement the delegation of permissions, passing on the permission is not enough. The delegating object must issue a certificate containing the permission and restrictions for its use. In addition, the certificate contains the public key of the owner of the permission. This allows other objects to prove that this object originally was the owner of the permission. The certificate is signed with the private key of the permission's owner: $Sign(K_{legit} :: Sign(object :: permission :: K_{legit}, K_{SA}^{-1}) :: [properties], K_{legit}^{-1})$.

Checking the UML model: One can then use the UMLsec tool-support to check properties regarding the security permissions. The following are examples for properties which can be captured using the UMLsec notation and then checked by the tool.

- A permission certificate must be received before the corresponding action can be executed.
- The emittent of a certificate must own the permission to create the certificate, and the permission must be released for delegation.

The analysis is carried out by translating the permission-relevant information from the sequence diagrams to Prolog resp. first-order logic (depending on the property to be checked) and giving it as input to a Prolog compiler resp. an automated theorem prover for first-order logic (SPASS or e-SETHEO).

Conclusion: We presented a tool for the security analysis of permissions specified in UML models, using Prolog resp. automated theorem provers for first-order logic. It allows one to define security permissions to restrict the access to the systems' resources based on the workflows the system is designed for. Using the associated tool, one can automatically see if workflows and the design of the security permissions for the system are consistent and satisfy the overall security requirements on the system. The tool is mature and has been used in industrial projects, such as [BJN07,JSB08].

References

[BJN07] B. Best, J. Jürjens, and B. Nuseibeh. Model-based security engineering of distributed information systems using UMLsec. In *ICSE*. ACM, 2007.

[JSB08] J. Jürjens, J. Schreck, and P. Bartmann. Model-based security analysis for mobile communications. In *ICSE*. ACM, 2008.

[Jür04] J. Jürjens. *Secure Systems Development with UML*. Springer, 2004.

[Jür05] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *27th Int. Conf. on Softw. Engineering*. IEEE, 2005.

[Too07] Umlsec tool, 2001-07. http://www4.in.tum.de/~juerjens/sectooltut.