ArcSecure – Model Driven Security

David Basin, ETH Zürich (basin@inf.ethz.ch)
Martin Buchheit, Interactive Objects Software GmbH (buchheit@io-software.com)
Jürgen Doser, Universität Freiburg (doser@uni-freiburg.de)
Bernhard Hollunder, Interactive Objects Software GmbH (hollunder@io-software.com)
Torsten Lodderstedt, Interactive Objects Software GmbH (lodderstedt@io-software.com)

Zusammenfassung

Im Rahmen des vom BMWA geförderten Leitprojektes „VERNET – Sichere und verlässliche Transaktionen in offenen Kommunikationsnetzen" (http://www.vernetinfo.de/) wurde innerhalb des ArcSecure Projektes das Thema *Model Driven Security* erstmalig systematisch aufgearbeitet und wesentliche Beiträge hinsichtlich der formalen Grundlagen sowie der praktischen Umsetzbarkeit erzielt.

Das ArcSecure Projekt wurde in der Zeit vom 01.07.2001 bis 31.12.2003 durchgeführt; das Projektkonsortium bestand aus der Interactive Objects Software GmbH[1] und der Universität Freiburg, Lehrstuhl für Softwaretechnik[2].

Der folgende Artikel gibt eine Einführung in die *Model Driven Security* und beschreibt den im ArcSecure Projekt verfolgten Ansatz. Weitere Artikel und Präsentationen zum ArcSecure Projekt können über http://www.vernetinfo.de aufgefunden werden.

---

[1] Interactive Objects Software GmbH, Basler Straße 61, D-79100 Freiburg, http://www.io-software.com.

[2] Universität Freiburg, Institut für Informatik, Lehrstuhl für Softwaretechnik, Georges-Köhler-Allee 52, D-79110 Freiburg, http://www.informatik.uni-freiburg.de/~softech.

1. Model Driven Security

Security is an integral part of most modern IT systems and designing such systems requires properly identifying, integrating, and configuring different security technologies. Examples include access control for preventing unauthorized access to system resources, encryption to ensure the confidentiality of data during network transmissions, and digital signatures for electronic contract signing. Although a large number of security architectures and technologies are available, we hear daily accounts of security vulnerabilities and failures.

Why is it so difficult to engineer robust, secure systems? A glance at the system development processes typically used suggests one reason: security is often managed in an ad-hoc fashion where requirements are analyzed and mechanisms are implemented shortly before, or even during, the system installation and deployment phase. Concerns are often separated and handled separately by software engineers and security specialists. This has technological reasons as well as cultural ones. In particular, there is a lack of methods and tools for tightly integrating security into the development process. The results are systems where security is often dealt with as an afterthought and where security mechanisms are poorly integrated into the system. The need to overcome this deficiency has been noted before, e.g. in [3].

We propose Model Driven Security as an approach to overcoming this deficiency. Model Driven Security provides methods and tools to tightly integrate security into the development process. The key idea is to use high-level, visual models that integrate system design and security design and to use generative techniques to automate the construction of systems from these designs. We have based Model Driven Security upon Model Driven Architecture, which is an emerging standard for model-centric and generative software development and is defined by the Object Management Group [4, 8]. Model Driven Architecture provides standards like the Meta-Object Facility [7] for defining modeling languages and the Unified Modeling Language (UML) [6] as the standard modeling language in the area of object-oriented software development. Furthermore, a language for specifying transformation rules is currently being standardized as well. Many development tools now implement these standards and they can be extended to support Model Driven Security.

In Model Driven Security, security is integrated into modeling. Design models are combined with security models, leading to new kinds of models that we call *security design models*. In these models, security policies refer to the elements of the core system model, e.g., components, business objects, methods, attributes, etc. Figure 1 suggests how this works: the design model is given with, for example, a class diagram, which is enriched with new modeling elements that represent security roles and permissions. This combination can be used to define an application's access control policy. Figure 1 also depicts the generative part of Model Driven Security. We extend existing transformation rules (as part of Model Driven Architecture) in order to automatically generate a policy-conform security infrastructure using the security mechanisms that are available in the target system platform.

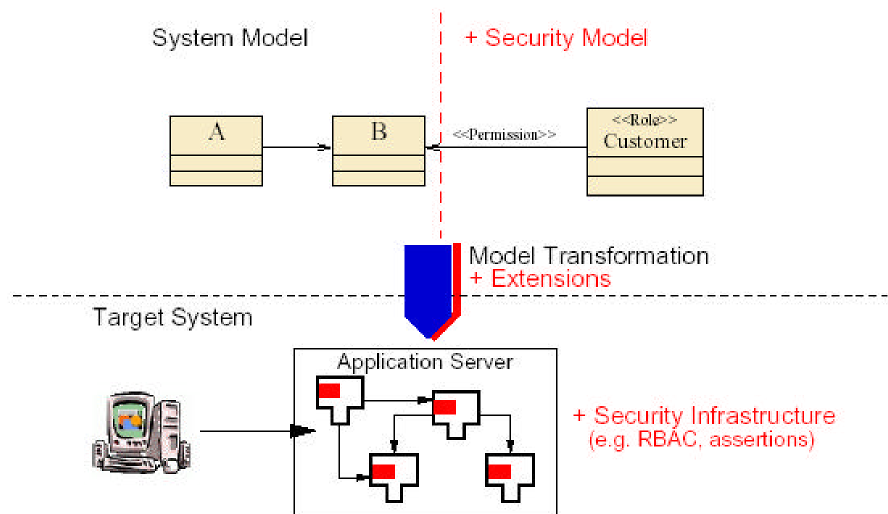

Figure 1: Model Driven Security: system design models are combined with security models and automatically transformed to system architectures.

We have built a tool that supports modeling access control policies, and which generates security infrastructures that are compatible with different standards for component-based systems such as J2EE/EJB or .NET. Our experience using this tool supports our thesis that the specification of security policies is simplified by combining them with design models. Moreover, it suggests that Model Driven Security constitutes a real "sweet spot" of Model Driven Architecture, i.e., an area where there are compelling advantages in using a model-driven process: security holes can be identified early in the development process; the implementation can be kept consistent with the modeled security policy; and implementations can be ported to new platforms by changing the transformation functions used.

2. An Example Security Design Language

The starting point of Model Driven Security is a security design language that combines vocabularies for expressing security and design policies. We give an example of such a language, which features a security language for modeling access control policies based on role-based access control (RBAC) [9]. We illustrate how to formalize a specific access control policy by means of an example.

Our example is a banking application, which has been considerably simplified but is adequate to convey the main ideas. The entities in this example are accounts, which have owner and balance attributes as well as methods for withdrawing and depositing money. The accounts, their attributes, and their methods together represent the resources that should be protected. Moreover, there are three roles, which formalize different kinds of system users: customers, bank employees, and managers. Within this setting, consider how we might formalize the following (informally given) security policy:

(P1) Each employee can read information associated with all accounts as well as create new accounts. Furthermore, he can change the owner of an account.

(P2) In addition to the employees' permissions, each manager can delete accounts.

(P3) Each customer can read all information associated with his own accounts.

The model in Figure 2 is given in a security design language that we proposed in [1] and formalizes this policy. In the right-hand part, a UML class is used to define the entity Account with its attributes and methods. The left-hand part defines the roles Customer, Employee, and Manager. Manager is defined to be a subrole of Employee, which means that a manager inherits all permissions granted for employees. The middle part connects roles with resources in order to model the system's access control policy.

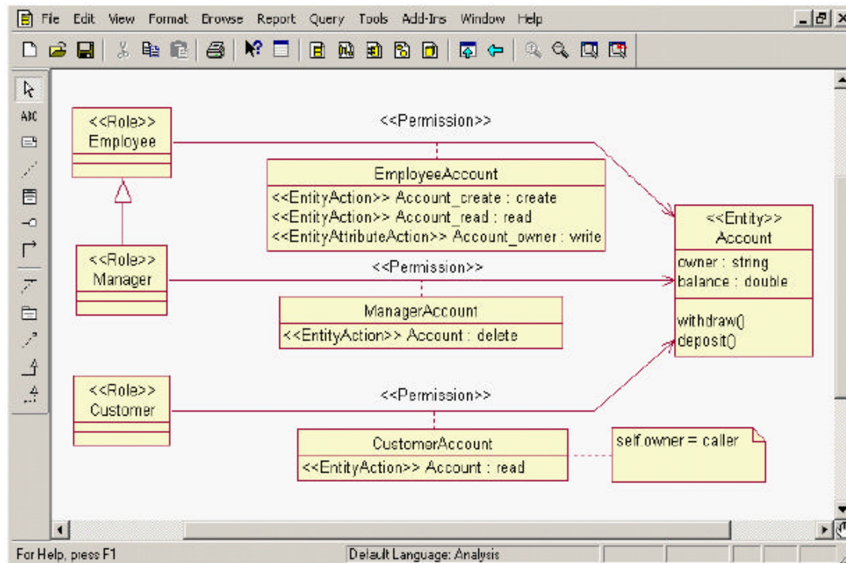

Figure 2: Modeling a small banking scenario.

The general permission of an Employee to access an Account is defined as a UML association labeled with the stereotype Permission between the two UML classes. In order to restrict this permission, the association is refined by the association class EmployeeAccount. An attribute of the association class specifies a so-called *action type*, i.e., the type of access the permission allows on the protected resource. In the example, the association class EmployeeAccount contains three attributes. Two of them are labeled with the stereotype EntityAction and have the types create and read, respectively. This formalizes that an employee has the right to create new accounts and to read, but not change, the attributes' values. Additionally, the action type read allows a customer to execute all business methods that are modeled as being side-effect free. In the example, however, the account's methods deposit and withdrawal modify the account's state, and hence cannot be accessed with a read permission. The third attribute in the association class EmployeeAccount expresses that an employee can change the value of the attribute owner of an account, thus completing the formalization of (P1).

The security policy (P2) is modeled as an inheritance between roles and by defining an additional permission with action type delete between Manager and Account.

To model (P3), we must formalize that each customer can read accounts, but only those accounts he owns. The general ability to read accounts is formalized analogously to the case of an employee. To express the restriction to owned accounts, we need a further modeling construct, which we call *authorization constraints*. An authorization constraint is specified using the Object Constraint Language (OCL), which is part of UML. Authorization constraints can be attached to association classes (cf. Figure 2) and restrict permissions. In our example, we restrict the permission Customer-

Account by the constraint self.owner = caller, which states that the caller of an access method must be the account's owner.

3. Security Design Language: A General Schema

Above we described one particular security design language in which we combined access control policies with UML system models based on class diagrams. However, there exist both a wide range of design languages (UML alone comprises several) and a wide range of security policy modeling languages (e.g., for modeling access control, information flow, and privacy policies). Because of this and because the development of any modeling language is a time-consuming task, it is in our opinion neither feasible nor desirable to develop a single "universal" security design language comprising all of these languages. We are better served by working with smaller domain or view-specific specification languages and developing general methods for combining them. We therefore propose a schema for building these languages in a modular way.

Looking back, we observe that a security design language has three parts:

1. a system design modeling language for constructing system models (e.g., components with their business methods, attributes, and associations to other components),
2. a security modeling language for expressing security aspects (e.g., role based access control policies), and
3. rules describing how to combine both modeling languages (e.g., selecting resource types and defining the corresponding action types).

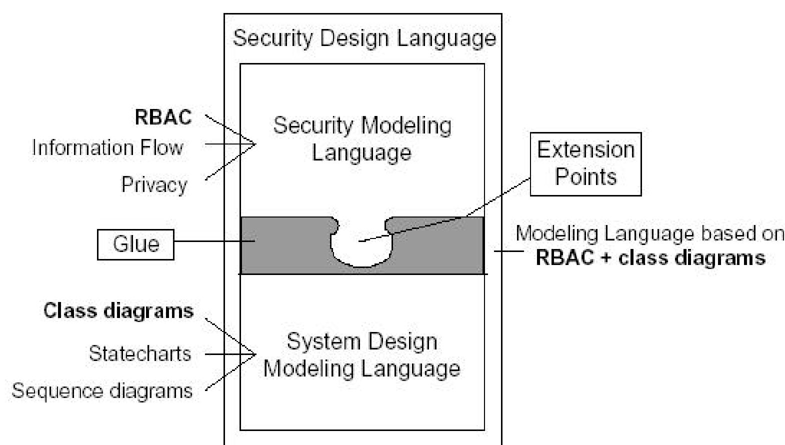

Figure 3: Constructing a Security Design Language.

These parts can vary and the previously given security design language can therefore be considered as just one instance of a more general language combination scheme, as shown in Figure 3. By different instantiations of the three parameters in this schema, we obtain different security design languages, tailored for expressing different kinds of designs and security policies.

Generally, one begins with both a system design modeling language and a security modeling language. Next, one needs some sort of *glue* that connects them, as well as a notation for the resulting combined language. The security modeling language used in the previous example, which we call SecureUML, is basically just the language of RBAC (extended with authorization constraints) and talks about roles, permissions, and actions on protected resources. This language can be *glued with* or *plugged into* various different system design modeling languages. The glue used to plug Se-

cureUML into a system design modeling language consists of defining which parts of the system design modeling language are protected resources in the sense of SecureUML, and what types of actions are defined on these resources. In the previous example, we defined methods and classes as protected resources, where one can, e.g., *execute* methods, or *read* (the attributes of) classes. But class diagrams are only one of many different possibilities. If the design language is UML statecharts, the protected resources might be (the ability to enter or leave) states or actions associated with transitions, etc. cf. [1].

The abstract syntax and the concrete UML notations of SecureUML, as well as of the system design modeling language, are defined by metamodels according to the Meta-Object Facility standard and UML profiles. Combining these languages on the level of abstract syntax then just means merging and gluing the metamodels. The glue consists of the definition of relationships between the elements of both languages. For example, classes in the system design modeling language are defined to be resources in SecureUML by specifying an inheritance relationship between resources and classes in the combined metamodel.

4. Generating System Infrastructures

One of the advantages of Model Driven Security is that by implementing specific transformation rules we can generate security infrastructures for different platforms out of a platform-independent model. As we have demonstrated in [1], UML models expressed in the security design language sketched above can be transformed into executable EJB applications as well as .NET applications. These applications fulfill the modeled access control policy without the need to manually program a single line of code.

In the following we focus on the generation of EJB systems. We can distinguish two types of generation rules:

1. rules for generating the EJB component infrastructure, and
2. rules for generating the access control infrastructure.

The first set of rules is responsible for generating the artifacts that are required by the EJB standard (cf. [10]). These comprise Java source code fragments such as the component interface, the home interface, the bean class including stubs for the business methods, and access methods for attributes and associations, as well as XML files for the deployment descriptor and build environment. For example, the UML class representing an account is transformed into an EJB Entity Bean component with all required interfaces and an implementation class.

A detailed description of these rules can be found in [1]. Here we will focus on the second kind of rules.

The security architecture of EJB supports both *declarative* and *programmatic* access control mechanisms. When applying the declarative approach, the access control policy is configured in the deployment descriptor of the EJB component. It is then the EJB application server's responsibility to enforce this policy. Complementarily, Java assertions that perform specific security checks can directly be inserted in the methods of the bean class. To support this, EJB's programmatic access control mechanism provides interfaces for retrieving security relevant user data, like their names or whether they act in a particular role.

Our rules for generating the access control infrastructure employ both mechanisms. The declarative access control mechanism is used to implement the permissions specified in the UML model. More precisely, each permission is translated into a collection of XML statements for the deployment de-

scriptor in such a way that the generated access control configuration enforces the modeled security policy, ignoring the authorization constraints. Programmatic access control is then used to enforce the authorization constraints: for each method that is restricted by an authorization constraint, an assertion is generated and placed at the start of the method body. As a consequence, when an EJB's method is called, the authorization constraint is checked first. If the authorization constraint is violated, a security exception will be thrown directly; otherwise, the method will be executed. The following Java code fragment shows the precondition of the method getBalance of the entity Account.

```
1.  if (!( (context.isCallerInRole("Employee")
2.          || context.isCallerInRole("Manager"))
3.          || (context.isCallerInRole("Customer")
4.                  && context.getCallerPrincipal().getName().equals(getOwner()) )
5.  )) throw new AccessControlException("Access denied");
```

The first two lines ensure that employees and managers have unrestricted access to this method, whereas the fourth line enforces the ownership constraint for all customers.

Having generated the application system's architecture, including the complete configured security infrastructure, the implementation of the modeled business methods (such as deposit and withdraw in the above example) is left to the application developers. Afterwards, the application can be deployed in an application server.

5. Towards the Development of Certifiably Secure Systems

The goal of our work is to support a robust methodology for developing secure systems. As our example suggests, the use of a security design modeling language provides a basis for formalizing security requirements produced during the requirements analysis. Tool support provides assistance in formalizing and refining the design and in making the subsequent transition to implementation. Hence, Model Driven Security is compatible with, and supports, phase-oriented development processes. It is especially well suited for iterative development, as much of the application generation is automated.

We have evaluated our concepts in an extensive case study based on a model driven version of the J2EE "Pet Store" application, which is a standard example application designed to demonstrate the use of the J2EE platform. Pet Store is an e-commerce application with web front-ends for shopping, administration, and order processing. The application model consists of 30 components and several front-end controllers. We have extended this model with an access control policy formalizing the principle of least privileges [5], i.e., a user may only have those access rights that are necessary to perform a job. The modeled policy comprises of six roles and 60 permissions, 15 of which are restricted by authorization constraints. The corresponding infrastructure is generated automatically and consists of roughly 5,000 lines of XML (overall application: 13,000) and 2,000 lines of Java source code (overall application: 20,000).

This large expansion is due to the high abstraction level provided by the modeling language. For example, it allows one to grant "read" access to a role, whereas EJB only supports permissions for whole components or single methods. Therefore, a modeled permission to read the state of a component may require the generation of many method permissions, e.g., for every get-method of an attribute. Clearly, this amount of information cannot be managed at the source code level. The low abstraction level provided by the access control mechanisms of today's middleware platforms therefore forces developers to take shortcuts and make compromises in the use of access control mechanisms. For example, roles are assigned full access privileges where they only require read access, or

access control is enforced by providing different applications for different viewpoints while the backend is only protected against external attackers by using firewalls and virtual private networks. However, this ignores the threats of internal attacks. Model Driven Security can help to ease the transition from security requirements to secure applications and to formalize and meet exact application requirements.

We can even go one step further. Although we use a UML-based notation, the modeling languages we use are equipped with a formal set-theoretic semantics (see [1]). As a result we can make formal guarantees about the properties of the resulting systems. First, it is possible to carry out analysis at the model level. This is useful when security policies are sufficiently complex; consider a modern banking application with hundreds of roles and thousands of special cases. Analysis here could be used to calculate with models and reason about their consequences, for example, under what conditions a user can access the banking data of others. Second, a formal language provides a starting point for reasoning about the correctness of the transformation process. With some effort, it should be possible to prove, once and for all, that a given transformation function respects the semantics of the models, i.e., generated applications, when executed on the target platform, are faithful to the security policy formalized by the model. For example, when access to protected resources is disallowed by the model, it is denied by the application.

Hence, long term, we see Model Driven Security as providing a key technology for building *certifiably* secure systems. Certification standards, like the Common Criteria [2], require the use of semi-formal methods for the higher levels of certification (Evaluation Assurance Levels 5 and 6) and the use of formal methods to validate models and a link between models and code in the highest level (Evaluation Assurance Level 7). The construction, analysis, and refinement of models is best carried out using specialized tools. Current tool support for Model Driven Security provides an auspicious beginning and going the distance will be an exciting, high-impact area for future research.

References

[1] D. Basin, J. Doser, and T. Lodderstedt. Model driven security: from UML models to access control infrastructures. Technical Report 414, ETH Zürich, July 2003.

[2] Common Criteria Recognition Arrangement. *Common Criteria for Information Technology Security Evaluation*, 1999. http://www.commoncriteria.org/cc/cc.html.

[3] P. T. Devanbu and S. Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 227–239. ACM Press, 2000.

[4] A. Kleppe, W. Bast, J. B. Warmer, and A. Watson. *MDA Explained: The Model Driven Architecture–Practice and Promise*. Addison-Wesley, 2003.

[5] T. Mayfield, J. E. Roskos, S. R. Welke, and J. M. Boone. Integrity in automated information systems. Technical report, National Computer Security Center, 1991.

[6] Object Management Group. *OMG Unified Modeling Language Specification, Version 1.4*, 2001. http://www.omg.org/technology/documents/formal/uml.htm.

[7] Object Management Group. *Meta-Object Facility (MOF), version 1.4*, 2002. http://www.omg.org/technology/documents/formal/mof.htm.

[8] Object Management Group. *MDA Guide Version 1.0.1*, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[9] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[10] Sun Microsystems, Inc. *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/ejb/docs.html.