Developing Secure Embedded Systems: Pitfalls and How to Avoid Them

Jan Jürjens,[*] Computing Department, The Open University, GB. http://www.jurjens.de/jan

We give an overview over the challenges in developing secure embedded systems and show how to use the approach of Model-based Security Engineering (MBSE) to address them. In MBSE [Jür04, Jür05a, Jür05b, Jür06, BJN07], recurring security requirements (such as secrecy, integrity, authenticity and others) and security assumptions on the system environment, can be specified either within a UML specification, or within the source code (Java or C) as annotations. The associated tools [UML04] (Fig. 1b) generate logical formulas formalizing the execution semantics and the annotated security requirements. Automated theorem provers 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. This way we encapsulate knowledge on prudent security engineering as annotations in models or code 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. One can use MBSE within model-based development (Fig. 1a). Here one first constructs a model of the system. Then, the implementation is derived from the model: either automatically using code generation, or manually, in which case one can generate test sequences from the model to establish conformance of the code regarding the model. The goal is to increase the quality of the software while keeping the implementation cost and the time-to-market bounded. For security-critical systems, this approach allows one to consider security requirements from early on in the development process, within the development context, and in a seamless way through the development cycle: One can first check that the system fulfills the relevant security requirements on the design level by analyzing the model and secondly that the code is in fact secure by generating test sequences from the model. However, one can also use our analysis techniques and tools within a traditional software engineering context, or where one has to incorporate legacy systems that were not developed in a model-based way. Here, one starts out with the source code.

[*]This work has partly been performed while still at TU Munich.

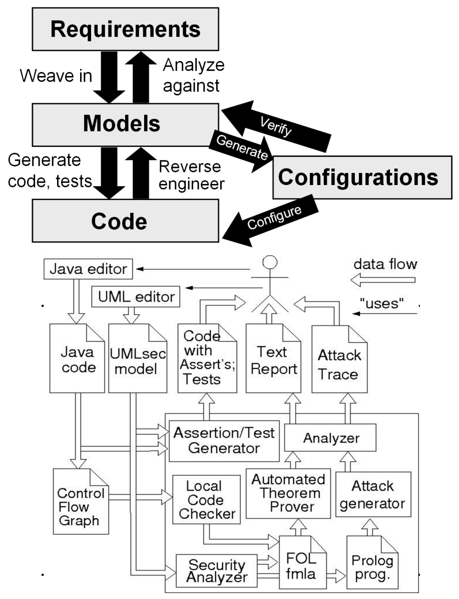

Figure 1. a) MBSE; b) Tool Suite

Our tools extract models from the source code, which can then again be analyzed against the security requirements. Using MBSE, one can incorporate the configuration data (such as user permissions) in the analysis, which is very important for security but often neglected.

Security Design Analysis using UMLsec [Jür04] 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 security-relevant information added using stereotypes includes security assumptions on the physical level of the system, security requirements related to the secure handling and communication of data, and security policies that system parts are supposed to obey. The UMLsec tool-support in Fig. 1b) can 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 [UML04, Jür05b]. 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. See [Jür04] for more details.

Code Security Assurance [Jür05a, Jür06] Even if specifications exist for the implemented system, and even if these are formally analyzed, there is usually no guarantee that the implementation actually conforms to the specification. To deal with this problem, we use the following approach: After specifying the system in UMLsec and verifying the model against the given security goals as explained above, we make sure that the implementation correctly implements the specification with techniques explained below. In particular, this approach is applicable to legacy systems. In ongoing work, we are automating this approach to free one of the need to manually construct the UMLsec model.

Run-time Security Monitoring using Assertions A simple and effective alternative is to insert security checks generated from the UMLsec specification that remain in the code while in use, for example using the assertion statement that is part of the Java language. These assertions then throw security exceptions when violated at run-time. In a similar way, this can also be done for C code.

Model-based Test Generation For performance-intensive applications, it may be preferable not to leave the assertions active in the code. This can be done by making sure by extensive testing that the assertions are always satisfied. We can generate the test sequences automatically from the UMLsec specifications. More generally, this way we can ensure that the code actually conforms to the UMLsec specification. Since complete test coverage is often infeasible, our approach automatically selects those test cases that are particularly sensitive to the specified security requirements.

Automated Code Verification against Interface Specifications For highly non-deterministic systems such as those using cryptography, testing can only provide assurance up to a certain degree. For higher levels of trustworthiness, it may therefore be desirable to establish that the code does enforce the annotations by a formal verification of the source code against the UMLsec interface specifications. We have developed an approach that does this automatically and efficiently by proving locally that the security checks in the specification are actually enforced in the source code.

Automated Code Security Analysis We developed an approach to use automated theorem provers for first-order logic to directly formally verify crypto-based Java implementations based on control flow graphs that are automatically generated (and without first manually constructing an interface specification). It supports an abstract and modular security analysis by using assertions in the source code. Thus large software systems can be divided into small parts for which a formal security analysis can be performed more easily and the results composed. Currently, this approach works especially well with nicely structured code (such as created using the MBSE development process).

Secure Software-Hardware Interfaces We have tailored the code security analysis approach to software close to the hardware level. More concretely, we considered the industrial Cryptographic Token Interface Standard PKCS 11 which defines how software on untrustworthy hardware can make use of tamper-proof hardware such as smart-cards to perform cryptographic operations on sensitive data. We developed an approach for automated security analysis with first-order logic theorem provers of crypto protocol implementations making use of this standard.

Analyzing Security Configurations We have also performed research on linking the UMLsec approach with the automated analysis of security-critical configuration data. For example, our tools automatically checks SAP R/3 user permissions for security policy rules formulated as UML specifications [Jür04]. Because of its modular architecture and its standardized interfaces, the tool can be adapted to check security constraints in other kinds of application software, such as firewalls or other access control configurations.

Industrial Applications of MBSE to Security-critical Embedded Systems: Pitfalls and How to Avoid Them:

- A biometric authentication system, where three significant security flaws were found [Jür05b, Jür05a].
- The Common Electronic Purse Specifications (CEPS), a candidate for a globally interoperable electronic purse standard supported by organizations representing 90 % of the world's electronic purse cards (including Visa International), where three significant security weaknesses were found and corrected [Jür04].
- The German Electronic Health Card in development by the German Ministry of Health.
- An electronic purse system developed for the Oktoberfest in Munich.
- An electronic signature pad based contract signing architecture at a German insurance company.

References

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

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

[Jür05a] J. Jürjens. Code security analysis of a biometric authentication system using automated theorem provers. In *ACSAC'05*. IEEE, 2005.

[Jür05b] J. Jürjens. Sound methods and effective tools for model-based security engineering with UML. In *ICSE*. IEEE, 2005.

[Jür06] J. Jürjens. Security analysis of crypto-based Java programs using automated theorem provers. In *ASE*. IEEE, 2006.

[UML04] UMLsec group. Security analysis tool, 2004. http://www.umlsec.org.