Author(s): Falcarin, Paolo; Morisio, Maurizio
Title: Developing Secure Software and Systems
Year of publication: 2004
Citation: Falcarin, P. and Morisio, M. (2004) 'Developing Secure Software and Systems', in *IEC Network Security: Technology Advances, Strategies, and Change Drivers*, Chicago: International Engineering Consortium (IEC), pp. 15-22

Developing secure software and systems

Paolo Falcarin
Paolo.Falcarin@polito.it
Politecnico di Torino
Control and Computer Engineering Dept.
Corso Duca degli Abruzzi, 24
I-10129 Torino, Italy

Maurizio Morisio
Maurizio.Morisio@polito.it
Politecnico di Torino
Control and Computer Engineering Dept.
Corso Duca degli Abruzzi, 24
I-10129 Torino, Italy

Abstract

The development and maintenance of network and data security in software systems is done in a late phase of design and coding or during deployment, often in an ad-hoc manner. Network monitoring and recovery, encryption protocols, best practices for combating cyber-crime, or disaster recovery planning are useful methodologies applied to enforce security of a deployed system. Nevertheless these are not enough to protect from attacks directed to software vulnerabilities hidden at design and code level.

Introducing security aspects in all the phases of the software development process is an emerging approach to limit costs of adding security features when it's too late and very expensive in terms of time and resources. In this paper we illustrate some proposals to consider security issues in the software process from the early phase of requirements to design and coding.

1. Introduction

Network security in all its aspects has become more crucial in recent years, in terms of protecting and preserving corporate data and network resources in cases of attacks and disruption. A complementary and needed strategy is building secure software applications. These are key-enabler for the development of complex services integrated through open standard interfaces, but currently the security concerns are often considered after deployment, when attacks or security bugs are found and it is very expensive to fix the problems: that is because the implementation of security features is widespread through several software modules and it is not fully understood by the majority of developers.

This is mainly due to the fact that software engineering and security engineering are quite independent disciplines. The two research and practitioners communities are just starting to collaborate, most often security experts are not software engineering experts and vice versa, the software engineering process does not consider security concerns in depth.

As a result, security issues are often considered only too late (when the system is deployed), with high economical costs, both for fixing the software security bugs and for losses in services or products.

The introduction of engineering-style development for secure software systems requires different methodologies for different process phases; moreover the process has to be designed having security issues in mind, and additional process-related activities (such as the definition of security requirements or security inspections) have to be considered.

Before to proceed, we want to underline that all other facets of system/software dependability (like reliability, safety, and availability) are not within the focus of this paper: we mean security in the definition given by the Common Criteria [2], e.g.: confidentiality, access control, protection, integrity, authentication, non-repudiation, etc...

In the next sections we will discuss existing approaches to introduce security early in the software development process. Then we describe new proposals we consider necessary to enhance state-of-the-art.

2. Existing approaches

Among the existing approaches, we briefly describe some existing solutions; for example there are ideas for enhancing UML with security, different tools to check and correct security vulnerabilities in the source code; moreover Aspect-Oriented Programming is used to separate security concerns from service logic code, in order to obtain a more modular and maintainable source code.

2.1 Security at design level

Given the widespread use of UML it is important to include explicitly in a formalized manner the security characteristics of a system within UML diagrams. Such additional information can be added using stereotypes and attributes in the easiest case, or by means of UML extensions if necessary. When design models (UML) are annotated with security related information it is possible to carry on automatic verification of the design. Such tools enable the identification of defects and security threats as early as possible in the life cycle, thus reducing the cost of their correction.
SecureUML is a proposal for a UML profile for security; more specifically, SecureUML extends UML to support authorization constraints. The capability of specifying significant information related to access control, and of deriving complete access control infrastructures (using EJB components), is presented in [11]. Basically, SecureUML defines a set of stereotypes and uses them to automatically produce the code that controls how users can access the different parts of the application. These stereotypes only consider the static parts of UML; dynamic ones are part of the future work.
A more ambitious UML profile is UMLsec [14]. In this case, the approach is quite different: it extends statecharts and component diagrams to enforce the following security requirements: fair exchange, secrecy/confidentiality, secure information flow, and secure communication link. The author describes the notation with a formal semantics based on Abstract State Machines language, to precisely analyse designed models. The main goals of UMLsec are:

- Evaluate UML specifications for vulnerabilities in design.
- Encapsulate established rules of security engineering.
- Make available to developers not specialized in security a simpler way to check UML models
- Consider security from early design phases
- Make verification cost-effective

UMLsec offers mostly-used security requirements as stereotypes with tags (secrecy, integrity, etc…); it uses associated constraints to evaluate specifications and indicate possible vulnerabilities; it ensures that stated security requirements enforce given security policy and that UML specification satisfies the defined requirements.
Among the mandatory requirements, UMLsec provides basic security requirements such as secrecy and integrity and it allows considering different threat scenarios depending on adversary strengths. Moreover it allows including important security concepts and mechanisms (e.g. access control) and it provides security primitives (e.g. (a)symmetric encryption).
The main goal of UMLsec is to automatically check security properties on an extended UML model, stored in XMI standard format; then this model can be checked by a formal tool [19] to verify security requirements.

2.2 Security and AOP

An emerging approach, coping with design and coding, is to express security issues with Aspect-Oriented Programming (AOP) [12]: with this approach security experts write security code in aspects and developers write application logic: then the designer uses particular tools (called aspect-weavers) to generate at compile-time the complete application code. Aspect-Oriented programming extensions to common programming languages (like AspectJ [4] for Java and Aspectc [5] for C++) have become widespread and more stable in recent years as a way for developers to obtain a more modular code even for crosscutting concerns like security. Usually design decisions on how and where inserting security aspects are made by developers, instead of designers. This does not fully protect against the introduction of security bugs due to developer's inexperience in security. What is missing is a methodology to integrate AOP at design level in order to delegate compositions rules of aspects to designers and security experts and not to developers. This goal could be achieved with extensions of UML for AOP: different ideas have been recently proposed [6] [10] to cover this issue, and a working group in UML standardization body is currently trying to introduce these extensions in the upcoming UML 2.0.

The AOP approach will allow developers to design applications in a security conscious way from the ground up, instead of applying security in an ad hoc manner during and after development. To reach this goal: developers should be able to write secure programs easily, without having to be security experts; developers should be able to apply security to legacy source code, without requiring revisions to the legacy code; developers should be able to apply security to COTS components for which they do not have source code.

AOP approach can be used by the security expert to handle, for example, SSL communications between components, or certificates exchange in a modular way. A key advantage offered by AOP is the possibility to change security parts of applications even in a later phase of design and implementation with a minimal cost, allowing to use more flexible software development processes.

2.3 Security at code level

Modern programming languages and systems provide much support for security. Through strong typing, they can substantially reduce the opportunities for coding errors that could result in buffer overflows and other vulnerabilities. They also allow protection by encapsulation and in addition, they sometimes include rich security infrastructures, for example libraries for authentication and digital signature. Although common programming languages are not primarily concerned with security, there is also the need of developing compilers considering secure programming issues, in order to give the possibility to developers to build more trustworthy code without being security gurus. For instance, access-control techniques that depend on the contents of the execution stack have been already investigated [13] and they trigger discussions on how to enhance programming languages in order to cope these issues; but until these problems are not faced by main compiler vendors, there will not be a widespread diffusion of these to industry. Currently, instead, the main approach is based on making a deeper debugging using tools that recognize bad design-patterns in source code, which can expose security vulnerabilities. This is due to the fact that most security attacks exploit instances of well-known classes of implementation flaws. Many of these flaws could be detected and eliminated before software is deployed. These problems continue to be present with growing frequency, not because they are not sufficiently understood by the security community, but because techniques for preventing them have not been integrated into the software development process phase of verification and validation of source code. Different strategies, more or less automated, can be used to find out security vulnerabilities at source code level.

Following the classification of [16] we can list these main classes of security vulnerabilities. Buffer-overflow is one of the most common problems of bad programming that hides memory-handling vulnerabilities that can be exploited in a Denial-of-Service attack; buffers, such as strings and arrays, can only contain a limited amount of data. When too much data is input to the buffer, it can overflow, overwriting other parts of memory and allowing new code to be inserted and executed. Another problem is due to race conditions, which happen when more than one thread (process) are competing for the same resource. A malicious program could attempt to grab resources before anyone else can get them, creating a problem of loss of secrecy and integrity.

Another issue to be considered are format strings: these define how the output of a command will be formatted. If an untrusted source has the ability to dictate what format strings will be used in a formatting command, such as print, the control can be leveraged to overwrite arbitrary memory locations just like a buffer overflow attack.

Another issue is trust management: developers often make poor decisions about who and what to trust. Servers should only trust one another if absolutely necessary to meet requirements. A firewall does not remove the need to secure those servers behind it.

Moreover, database account names and passwords are often hard coded into client applications that need access. These names and passwords can be pulled out and used to access the database directly. Other attacks on databases included bypassing poorly set up access controls, statistical inference attacks, and they can be damaged by malicious input.

Malicious input is the one received from users and external systems, which may be structured to maliciously affect the behaviour of the program. This input should be checked to ensure it adheres to the allowed values before it is used.

Latest but not least, bad implemented cryptography and cryptographic algorithms with known flaws can both lead to severe vulnerabilities. Whenever possible, widely reviewed cryptographic libraries should be used. Finally, as cryptography relies on random numbers, without a good source of random numbers, the cryptographic output will be weaker. If an attacker can guess what random numbers you are getting, they will be able to decipher the message with cryptanalysis.

In the code development phase, the goal is the protection of data and functions from unauthorized access, e.g.: unauthorized read/write of data, unauthorized use of function or service, focus on software faults that lead to security leaks. Current strategies are based on tools to find security vulnerabilities in the source code. For example, an extensible open-source tool like Splint [15] uses lightweight static analysis to detect common security vulnerabilities (including buffer overflows and format string vulnerabilities) and can be extended to detect new vulnerabilities.

Instead Flawfinder [18] is a program that examines source code and reports possible security weaknesses (``flaws'') sorted by risk level. It's very useful for quickly finding and removing at least some potential security problems during debugging. Flawfinder works by using a built-in database of C/C++ functions with well-known problems, such as buffer overflow risks, format string problems, race conditions, and poor random number acquisition. Flawfinder produces a list of ``hits'' (potential security flaws), sorted by risk in descendant order. This risk level depends not only on the function, but also on the values of the parameters of the function. For example, constant strings are often less risky than fully variable strings in many contexts. In some cases, Flawfinder may be able to determine that the construct isn't risky at all.
Not every hit is actually a security vulnerability, and not every security vulnerability is necessarily found. In fact, Flawfinder doesn't really understand the semantics of the code at all - it primarily does simple text pattern matching (though it does ignore comments and strings as it should). Anyway, Flawfinder can be a very useful aid in a coarse-grained finding and removing of security vulnerabilities.
Finally, RATS [16] as its name states, it is a “Rough Auditing Tool for Security “, because it performs only a rough analysis of source code. It will not find every error and will also find things that are not errors (false positives).
Better tools are based on formal methods but for space reason we will not describe them here.

3. New proposals

In Figure 1 the different phases of a generic process are mapped to different methodologies used to introduce security at different levels of the software development process; in the next paragraphs we will describe some proposals to integrate these currently separated methodologies and tools.

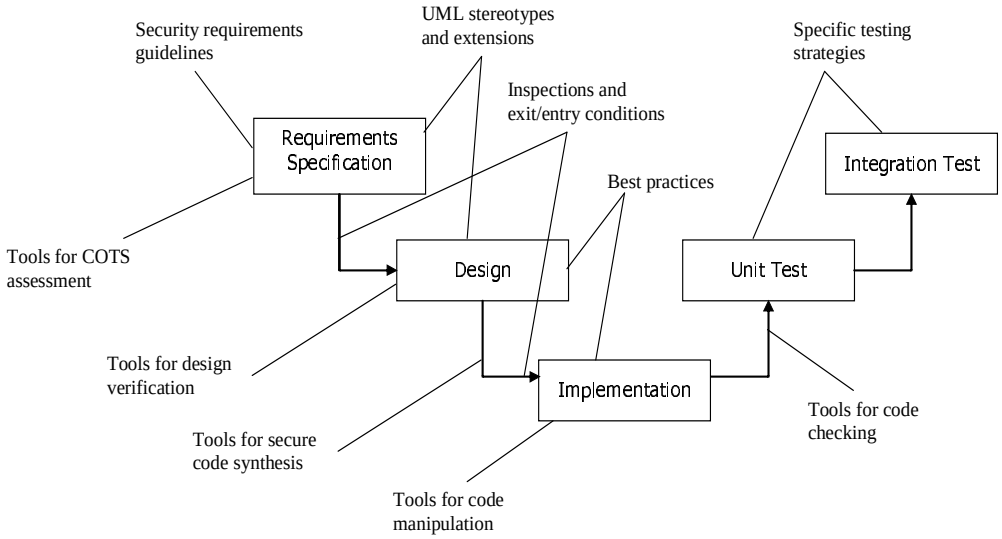

Figure 1. Mapping of methodologies and tools for secure software to the process

3.1 Process

Definition and validation of a software process capable of handling the security issue from the very beginning is a key issue: this must be capable of integrating tools, methods and organization in the most suitable way. As all these tools and techniques come from different domains and are tailored for particular systems and languages, in future they should have to be meaningfully integrated in a comprehensive process. This is currently an open issue and a desirable goal in order to have better results and faster development; these are key aspects for a broad adoption by industry.
As security requirements often change in time, because of discovering new attacks and protocols’ faults, the process must be enough agile to permit fast correction of security bugs.

The necessary condition for agility is having security requirements traceability through design and code; moreover it will be useful to include external prerequisites (derived from standards, laws, regulations etc.) in a prescriptive way in the models. This leads to better understanding of the processes. An explicit process is the basis for measurement, evaluation and improvement; and for external security process assessments.
All these goals can be met with different actions, like: elicitation of industrial practices with respect to security, process modelling, process instrumentation with measures, process analysis (identification of potential improvements), use of security entry/exit conditions. These conditions formally define if and how to proceed from one phase to the other of the process: on the one side they can enforce application of inspections and other practices, and on the other side can sensibly improve the quality of the final product.

3.2 Security Requirements

Developing methods and tools to specify and analyse security requirements from the very beginning is a fundamental issue to merge software development and security.
While there is a stringent need to increase the importance of security in software engineering curricula, not all software engineers will be ever able to become security experts, and vice versa.
A promising approach is to delegate security concerns to security experts and use methodologies and tools to integrate the separate work in a final common result.
Among methodologies for the detection of defects in earlier phases, one is based on security inspections. Inspecting documents for security (such as requirements or design documents) would further contribute to finding security risks. Early detection of defects with respect to security leads to reduction of rework efforts. Inspections and reading techniques in general are recognized to be effective practices to identify defects in software artefacts, overall when these are delegated to security experts. Inspections are complementary to testing, especially for the possibility of using them on early phases of the lifecycle. However, inspections are normally targeted to functional defects, while there is an emerging need of defining a family of inspection techniques targeted to security defects, to be applied on requirements, design and coding artifacts.

3.3 UML and AOP

Often security is not well understood by all developers of a complex system; forcing every developer to change his/her code to be compliant to existing or upcoming security requirements could slow the entire development process. AOP can help to have a more agile process of maintenance and evolution of security concerns: these tasks can be triggered by new security requirements coming out in a later phase of design, during implementation, or, most frequently, after deployment when attacks reveal security pitfalls. In this case it is necessary to individuate which security concerns are involved and then rewrite the related security aspects: this approach can lead to an easier evolution of the system and to a clearer tracing of security requirements through design to code. This idea could be implemented with a model-driven architecture able to provide designers with a complete set of views of the system. Therefore, there is an urgent need at design level of an integration of UML and AOP: it is important to extend UML diagrams with AOP constructs to be able to trace aspects (e.g.: security aspects) in classes of applications and to give to designers an easier way to control the system development and easily change security concerns in a later phase of software process cycle. Different proposals have been recently published [10] [7] [9] [8], but a complete theory and related tool are currently lacking. These UML extensions are realized extending UML meta-model or extending the notation with ad-hoc UML profiles. The real need is increasing as more as possible automation of pointcuts generation from design models to aspects' source code: this is a key issue in order to give full control on distribution of security aspects to security designers, without risking to delegate these ones to common developers as it frequently happens.

4. Conclusions

The main problem of current software development is to consider security concerns from the beginning of the process. We have presented some existing approaches we have evaluated, and some new proposals for further work State of the art is still immature in some parts of the process and lot of things are needed to offer a more integrated and reliable solution. We believe that a

process-centered view is fundamental to trace security requirements through design and source code, and that AOP is a key technology that can lead to a more agile process; this is because evolution of security parts of a system is triggered by discovery of new attacks and vulnerabilities: what is needed now is the possibility to easily upgrade and secure a system from design until after deployment. For a more detailed description of the presented topics we suggest the following texts on secure programming [1] [17] and to check out the following references.

5. Acknowledgments

The authors would like to thank for the precious collaboration, the following people: Wouter Joosen (Katholieke Universiteit of Leuven, Belgium), Luciano Baresi (Politecnico di Milano, Italy), Juergen Muench (Fraunhofer IESE, Kaiserslautern, Germany).

6. References

[1] Special Issue on security software, Software Practice and Experience, vol. 33, issue 5, 2003.
[2] Common Criteria. http://www.commoncriteria.org/
[3] T. Lodderstedt, D. Basin, and J. Doser, "SecureUML: A UML-Based Modeling Language for Model-Driven Security", 5th International Conference on the Unified Modeling Language, 2002.
[4] The AspectJ Programming Guide, the AspectJ team, available at http://aspectj.org
[5] The Aspectc project, available at http://www.aspectc.org/
[6] J. Suzuki and Y. Yamamoto. "Extending UML with Aspects: Aspect Support in the Design Phase," In Proc. of the 3rd Aspect-Oriented Programming (AOP) Workshop at The 13th European Conference on Object Oriented Programming (ECOOP'99), Springer LNCS 1743, Lisbon, Portugal, June 1999.
[7] O. Aldawud, T. Elrad, and A. Bader, "A UML Profile for Aspect Oriented Modeling". OOPSLA 2001 workshop on Aspect Oriented Programming.
[8] R. Pawlak, L. Duchien, G. Florin, F. Legond-Aubry, L. Seinturier, and L. Martelli, "A UML Notation for Aspect-Oriented Software Design". On-line at http://jac.aopsys.com/papers/uml_short/uml.html
[9] M. M. Kandé, J. Kienzle, and A. Strohmeier, "From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach".
[10] Third International Workshop on Aspect-Oriented Modeling. On-line at http://lglwww.epfl.ch/workshops/aosd2003/
[11] Tool for automatic J2EE deployment of secureUML: http://www.io-software.com/products/data/mda_security_factsheet_020902.pdf
[12] B. De Win, B. Vanhaute, and B. De Decker, "Security Through Aspect-Oriented Programming, Advances in Network and Distributed Systems Security". B. De Decker, F. Piessens, J. Smits and E. Van Herreweghen, eds. Kluwer Academic Publishers, 2001, pp. 125-138.
[13] M. Abadi, "Built-in Object Security", In proceedings of ECOOP 2003, Darmstadt, Germany, July 2003.
[14] J. Jürjens, "Model-based Security with UMLsec", UML Forum, Tokyo, April 2003.
[15] Splint homepage. http://www.splint.org/
[16] RATS tool, http://www.securesoftware.com/download_form_rats.htm
[17] D.A. Wheeler, Secure Programming for Linux and Unix. On-line at http://www.dwheeler.com/secure-programs/
[18] Flawfinder homepage: http://www.dwheeler.com/flawfinder/
[19] Autofocus homepage: http://autofocus.in.tum.de/index-e.html