JYX JYVÄSKYLÄN YLIOPISTO UNIVERSITY OF JYVÄSKYLÄ

This is a self-archived version of an original article. This version may differ from the original in pagination and typographic details.

Author(s): Mazhelis, Oleksiy; Naumenko, Anton

Title: The Place and Role of Security Patterns in Software Development Process

Year: 2006

Version: Published version

Copyright: © 2006 SciTePress

Rights: CC BY-NC-ND 4.0

Rights url: https://creativecommons.org/licenses/by-nc-nd/4.0/

Please cite the original version:

Mazhelis, O., & Naumenko, A. (2006). The Place and Role of Security Patterns in Software Development Process. In E. Fernandez-Medina, & M. I. Yague (Eds.), Security in information systems : proceedings of the 4th International Workshop on Security in Information Systems, WOSIS 2006 (pp. 91-100). Insticc press. https://doi.org/10.5220/0002478700910100

The Place and Role of Security Patterns in Software Development Process

Oleksiy Mazhelis and Anton Naumenko

University of Jyväskylä
Information Technology Research Institute
P.O.Box35, FIN-40014, Jyväskylä, Finland,

Abstract. Security is one of the key quality attributes for many contemporary software products. Designing, developing, and maintaining such software necessitates the use of a secure-software development process which specifies how achieving this quality goal can be supported throughout the development lifecycle. In addition to satisfying the explicitly-stated functional security requirements, such process is aimed at minimising the number of vulnerabilities in the design and the implementation of the software. The secure software development is a challenging task spanning various stages of the development process. This inherent difficulty may be to some extent alleviated by the use of the so-called security patterns, which encapsulate knowledge about successful solutions to recurring security problems. The paper provides an overview of the state of the art in the secure software development processes and describes the role and place of security patterns in these processes. The current usage of patterns in the secure software development is analysed, taking into account both the role of patterns in the development processes, and the limitations of the security patterns available.

1 Introduction

Nowadays, security is one of the key quality attributes that many software products should possess. Development of such secure software necessitates both the implementation of appropriate security mechanisms, such as authentication and access control, and robust design and implementation of the software making it resistant to attacks [1].

The process of secure software development is highly complex, due to the need to take into account a variety of security-related aspects at different stages of the development process. For instance, possible threats should be identified and evaluated at the requirement phase; appropriate security mechanisms (controls) should be defined at the design phase, etc. With the aim to facilitate this process, so called security patterns are introduced [2, 3] that encapsulate knowledge about successful solutions to recurring security problems.

In our research project on mobile design patterns and architectures[1], we work in collaboration with software houses. The development of software in these organisations usually follows a specific method, derived from generic software development

[1] MODPA project, http://titu.jyu.fi/modpa/

Mazhelis O. and Naumenko A. (2006).
The Place and Role of Security Patterns in Software Development Process.
In *Proceedings of the 4th International Workshop on Security in Information Systems*, pages 91-100
Copyright © SciTePress

process models, such as the incremental or the spiral model [4]. These models historically were of little practical use for the security teams, because security aspects were not covered in them. Instead, a security team could follow a separate security-specific process model, such as the security waterfall lifecycle [5]. Thus, the security design was isolated from the functional design; furthermore, the security design implied the pre-existence of the functional design. As a result, conflicts arose between these two designs and between the responsible designers. Only relatively recently, efforts have started incorporating security into the software development process. However, the use of security patterns appears to be mostly ignored in these models except few recent works within the security patterns community [6, 7].

This paper aims at specifying the role of security patterns in the process of secure software development taking place in the companies. Having overviewed the available models for secure software development processes, the place of security patterns in these models is identified, and the function of the patterns is analysed. The paper is organised as follows. The next Section introduces software patterns in general and security patterns in particular. Section 3 reviews secure software development processes. After that, Section 4 describes the use of security patterns at different stages of a secure software development process. Finally, Section 5 summarises the benefits of the involvement of security patterns and indicates the areas where current work is limited.

2 Software Patterns

A pattern describes a solution to a specific type of problems recurring in a particular context. The idea of identifying patterns in design is generally attributed to the building architect Christopher Alexander and colleagues, who gave the following definition of patterns [8]:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

In the domain of software engineering, patterns became known mainly after Gamma, Helm, Johnson, and Vlissides (Gang of Four, or GoF) published their seminal book on design patterns for object-oriented software [9]. Patterns in object-oriented software "identify participating classes and instances, their roles and collaborations, and the distribution of responsibilities" [9, p. 3]. The description of the pattern specifies among other things the context in which the use of a pattern is appropriate as well as the consequences of pattern's use. As a result, the use of patterns supports producing architectural solutions with desired properties.

Patterns can be divided into several categories according to the level of abstraction at which they operate. Buschmann et al. identified architectural patterns, design patterns, and idioms [10] in decreasing order of abstraction level. Different stages of the development process can employ patterns of different abstraction levels:

- Architectural patterns are used to produce a high-level conceptual software architecture;

– Design patterns facilitate the creation of a more detailed concrete architecture;
– Idioms are of practical use at the implementation stage to address the peculiarities of a particular language or platform.

After the work of GoF, a number of other pattern catalogues, systems, and languages have appeared [10–12], addressing different problem areas (e.g. distributed computing, resource management, etc.), as well as different platforms. The first attempt to synthesise the patterns for typical design problems in the domain of information security may be attributed to Yoder and Barcalow who proposed a collection of architectural security patterns [2]. As well as other patterns, the security patterns encapsulate knowledge about successful solutions to recurring design problems, focusing on the re-occurring security problems. They support the design of the software, wherein the confidentiality, integrity, and availability of information (for authorised use) are crucial.

Figure 1 illustrates as an example the Protected system pattern [3] (also known as single access point pattern [2]). The pattern is used to protect the system resources against unauthorized access by ensuring that all the clients' requests to the protected resources are mediated by a guard. The guard's responsibility is to determine whether the requests are permitted by a predefined security policy. Thus, the guard enforces the security policy by intercepting the requests and matching them against this policy. The guard should be non-bypassable and incorruptible, and it is responsible for validating input data contained in the requests. A firewall is an example of the protected system, where the resources are represented by the IP addresses and ports of the systems behind the firewall.

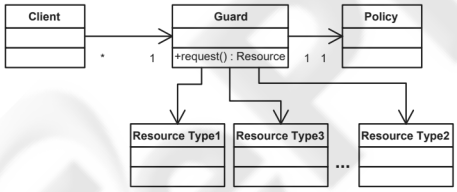

Fig. 1. The protected system pattern with a single centralized guard.

Since the work of Yoder and Barcalow, a number of security patterns or pattern collections have been proposed, usually targeting a narrow subset of problems in the security domain (such as patterns for access control [13]). The available collections of security patterns remain fragmented and sometimes inconsistent; e.g. the same patterns appear under different names in different sources. In response to this, more recently, the Open Group Security Forum attempted to create a more comprehensive list of existing security patterns [3]. According to the categories of problems being addressed, the patterns in [3] are divided into the available system patterns facilitating the provision of predictable and uninterruptible access to the resources and services, and the protected system patterns aimed at protecting valuable resources against unauthorized use, disclosure or modification.

3 Secure Software Development Processes

A software development process defines the roles, activities, stages and the outcomes to build a software product or to enhance an existing one [14]. The development process is usually guided according to a specific process model, such as the waterfall or Boehm's spiral model.

Security brings additional challenges in the development process, e.g. [15]:

- The actual customer of the system being developed is not adequate source of security requirements;
- The use case methodology, while proven useful in many software development endeavours, is difficult to apply to security design, since capturing all malicious activities during the requirements definition is infeasible and requires from the software architects deep knowledge of security domain;
- It is impossible to guarantee the absence of security bugs; therefore, the absence of security bugs cannot be used as an acceptance criteria.

The traditional process models are of little practical use for the designers and developers responsible for software security, because these models do not address security-specific challenges. Instead, groups of security experts within the development teams followed a separate security-specific process model, such as the security waterfall lifecycle [5]. As a result, the security design was isolated from the functional design and implied its pre-existence, and consequently resulted in conflicts between these two designs and between the designers.

Recently, however, several efforts were made to incorporate the security aspects into the software development process. We will refer to the development processes created in these efforts as the secure software development (SSD) processes. A secure software development process emphasises security-related concerns. In particular, such process is aimed at minimising the number of vulnerabilities in the design and the implementation of the software. In this section, several available process models for SSD are overviewed.

One of the attempts to merge the security development and the general lifecycle models was conducted at Microsoft. In the attempt to improve the software development processes and thereby to make the software resistant to malicious attacks, Microsoft developed the so-called Trustworthy Computing Security Development Lifecycle (SDL). This SDL modifies the organisation's development process by adding well-defined security check-points and security deliverables [1].

According to Microsoft, several main principles should be consistently applied in order to produce a more secure software [16]. These principles include i) "secure by design" (designing and implementing software able to protect the information being processed and able to resist attacks), ii) "secure by default" (software's default state should promote security), iii) "secure in deployment" (secure use of the software should be supported with guidelines and tools), and iv) "communications" (software vulnerabilities discovered after the deployment should be remedied timely, and the information about them should be delivered to the users). The integration of these principles into a development process has resulted in a set of security checkpoints and security deliver-

Table 1. Adding security check-points and deliverables to the software development process (adopted from [16]).

Process phase	Security check-points and deliverables
Requirements	Inception: security advisor assigned; ensure security milestones understood; identify security requirements
Design	Design & Threat modelling: design guidelines documented; threat models produced; security architecture documented; threat model and design review completed; ship criteria agreed to
Implementation	Guidelines & Best practices: coding and testing standards followed; test plans developed and executed; tools used for code analysis
Verification (Beta)	Security push: threat models reviewed; code reviewed; attack testing; new threats evaluated; security testing completed
Release	Final security review (FSR): threat models reviewed; unfixed bugs reviewed; new bugs reviewed; penetration testing; documentation archived
RTM	RTM & Deployment: signoff by security team
Response	Security response feedback: tools/processes evaluated; postmortems completed

ables. These checkpoints and deliverables are listed in Table 1, where they are mapped onto the phases of the traditional waterfall model.

Similarly to Microsoft's SDL, Peterson [17–19] describes what security concerns need to be addressed at various phases of an arbitrary secure development process. The reference process used by the author is iterative and includes the phases of analysis, design, development, and deployment. At the Analysis phase, the requirements are lumped together and analysed. The specification of security requirements can be facilitated by employing so-called misuse cases [17] that focus on the illegitimate behaviour of attacker(s), i.e. on the functionality that the system should prohibit [20]. Misuse cases can be as well used as a basis when (security-related) testing scenarios are defined. On the design phase, the software architecture provides the security team with the system scope, and "illuminates security risks" [18]. Security-specific artefacts produced at the design phase include threat models, data classification, and security integration design. At the coding phase, the unit testing and code development are supplemented with unit hacking as well as with countermeasure and detection/signature development [19]; here, a unit hack is a security focused unit test case verifying the code's resistance to a particular type of attack.

McGraw [1] presents the software security best practices, and describes how they can be applied to software artefacts. These best practices are essentially the same as described in Microsoft's SDL and/or Peterson's security concerns.

National Institute of Standards and Technology (NIST) also published recommendations of how security should/could be incorporated into the system development lifecycle (SDLC) [21]. The NIST guidelines align the security considerations along five basic SDLC phases including Initiation, Development/acquisition, Implementation, Operation/maintenance, and Disposition. In addition to the security aspects covered in the guidelines above, the NIST recommendations address other security issues, such as security certification and accreditation, or disposition-related issues (e.g. information preservation and media sanitisation).

Other processes for secure software development have been proposed as well. These include, among others, "Correctness by construction", "Cleanroom Software Engineer-

ing", and "Team software process", as described in [22]. The "Correctness by construction" process introduces a formal notation to specify the system and design components, and suggests conducting specific review and analyses aimed at ensuring the consistency and correctness. As a result of applying this process, near-defect-free software was reported to be produced. The "Cleanroom Software Engineering" is also a formal process based on the incremental development and testing. Its four key points include incremental development, function-based specification and design, functional correctness verification, and statistical testing. The "Team software process" is an operational process introduced by the Software Engineering Institute. It describes a set of best practices to be used by independent developers and development teams. According to these best practices, secure software development is supported by preventing and removing defects throughout the lifecycle, by applying measurement and quality management to control the process, and by using predictive measures for remaining defects. The process is reported to be effective way of producing near defect-free software within budget and time constraints [22].

4 Patterns in Secure Software Development Process

The place and role of patterns in traditional software development process has been discussed in literature; e.g. Yacoub and Ammar [23] introduced the Pattern-Oriented Analysis and Design (POAD) approach, where patterns are seen as key instruments in software analysis and design. In this subsection, the role of security patterns in secure software development is analysed, and the place of these patterns in process models is considered.

Role of security patterns. As discussed in [22], more than 90% of software security vulnerabilities are caused by known software defect types which can be classified as specification, design, and implementation defects. Some of vulnerabilities are caused by implementation defects, such as declaration errors, logic errors, loop control errors, conditional expressions errors, failure to validate input, interface specification errors, and configuration errors. At the same time, a number of vulnerabilities are caused by specification and design defects, e.g. failure to identify threats, inadequate authentication, invalid authorisation, incorrect use of cryptography, failure to protect data, and failure to carefully partition applications.

In order to minimise the number of vulnerabilities exposed in the software, it is crucial to reduce the number of the specification, design, and implementation defects. Security patterns may provide a support in amending these defects at various phases of the development process, as discussed below.

Place of security patterns. In this subsection, the use of security patterns is considered along the requirements, design, implementation, and deployment phases commonly present in the software development process. For the sake of brevity, only the pattern-specific activities are described for each phase.

Requirements. At this phase, explicit and testable security requirements are to be stated. Security patterns address the recurring security problems, and the descriptions of these problems may be employed as a checklist of the problems that the software being designed may potentially need to address.

For eliciting covert security requirements, misuse/abuse cases may be employed [17, 1]. To support the discovery of relevant misuse/abuse cases, some of the pattern descriptions may present or exemplify a security problem in a form of a misuse case.

Other pattern-related activities at this phase, as suggested by Yacoub and Ammar [23], include the identification of relevant design problems, the acquaintance with relevant patterns, retrieval of candidate patterns from domain-specific pattern databases, and the selection of patterns for further use in the design.

Design. At the design phase, the overall structure of the software is formed, and the security-critical components in this structure are identified. The risk of incorporating defects in the design may be mitigated, if the security patterns are employed during the design process. For instance, such design defects as inadequate authentication, invalid authorisation, failure to protect data may be avoided if the Protected System patterns [3] are consistently used.

The design phase can be seen as consisting of high-level design activities, and the design refinement activities [23]. During the high-level design, pattern-level diagrams with different levels of details are constructed; this results in a detailed diagram, where the patterns, interfaces between them, and the internal design of the patterns are shown. In turn, the design refinement phase focuses on instantiating the application-specific pattern internals, developing class diagrams, and, finally, optimising these diagrams to reach a "dense and profound design" [23].

Security patterns, as other patterns, provide necessary concepts to express the structure and interactions imposed by an architecture at a higher abstraction level [24]. Therefore, documenting complex security designs may be facilitated by these patterns.

Once the architectural design is produced, its fitness to the stated requirements needs to be assessed. In order to determine the appropriateness of an architecture with respect to these requirements, a formal architecture evaluation can be conducted. The process of evaluation can be based on a specific evaluation method, such as the Software Architecture Analysis Method (SAAM) and the Architecture Tradeoff Analysis Method (ATAM) [25]. An important precondition for a successful evaluation is a deep understanding of the architecture among the evaluators. For this, in ATAM, for example, several steps are devoted to the identification and the analysis of the architectural approaches and decisions (in particular in a form of architectural styles and patterns followed), which underpin the architecture being evaluated [25]. Security patterns may thus be useful in the evaluation process for revealing and analysing security-related architectural approaches and decisions.

Implementation. At this phase, the coding and testing are performed in accordance with standards and best practices. Depending on the particular platform and programming language, the developers are faced with lower-level problems specific to this platform or language. If improperly addressed, these problems may result in implementation defects such as loop control errors, conditional expressions errors, or failure to validate input.

Low-level security idioms are aimed at encoding the successful solutions to such problems, and are therefore useful at this phase. For example, Code Validator pattern can be employed to cope with buffer overflows when programming in C/C++, Syntax Validator pattern is useful for cleaning the strings fed to cgi-bin scripts, etc. [15].

Deployment. The implemented software at this phase is configured and integrated into the enterprise architecture. Even if all the security guidelines and best practices are carefully followed, the software produced is still highly likely to contain vulnerabilities which are going to be discovered while the software is in operational use [16].

The development team, therefore, should response to the discovered vulnerabilities by timely issuing patches and informing the users. The design and implementation of such security updates could be seen as a special case of software maintenance. As evidenced by experimental studies [26], this process can be completed faster and/or with smaller number of mistakes if the use of patterns is documented in the software code. Therefore, to facilitate further security updates, it is advisable to explicitly document the employed security patterns in the code.

5 Discussion and Concluding Remarks

Among the variety of non-functional requirements which contemporary software is often required to meet, security appears to be one of the most difficult to satisfy. This difficulty stems from the fact that making the software secure requires from the development team to i) introduce appropriate security mechanisms, and ii) implement the software (including the security mechanisms) appropriately. Therefore, the members of a team faced with the need to design and implement secure software should be, on one hand, skilful in software development, and on the other hand, literate in the security-related issues, such as threat modelling, security mechanisms design, implementation, verification, and integration.

The secure software development process models are aimed at alleviating the above difficulty, by defining the set of activities needed to transform user requirements into a product. Several such process models were proposed recently; among them are e.g. Microsoft Security Development Lifecycle process, Peterson's Secure Development process, NIST guidelines, etc. Security patterns are as well aimed at alleviating the difficulty of secure software development by encapsulating knowledge about successful solutions to recurring security problems. These patterns address various security-specific issues at different phases of the development process. A number of security patterns and pattern collections were elicited during last decade. Little attention, however, was devoted to the recommendations, which could be applied in companies, and which would specify how and at which phase to deploy security patterns in secure software development. While a number of works discussing security development process models can be located [5, 21, 16, 22, 17–19, 27], the use of security patterns in these models is not specified.

This paper could be seen as an initial attempt to amend this limitation by considering the place and role of security patterns in the secure software development. The role of security patterns is to minimise the number of vulnerabilities exposed in the software, through the minimisation of the design and implementation defects. Security patterns may provide a support in amending these defects at different phases of the development process. At the design phase, many design defects can be avoided, if the security architectural and design patterns such as the Protected System patterns are consistently used. At the implementation phase, the low-level security patterns (idioms) play an important

role in fighting against implementation defects, such as loop control errors, conditional expressions errors, and failure to validate input.

As concerns the security patterns themselves, while increasingly large body of knowledge is available on security patterns, several limitations in these works can be identified, thus indicating the areas where further work is needed.

First, no single source (repository) gathering all or major security patterns into a consistent pattern system is available. The available collections of patterns remain fragmented and sometimes inconsistent; e.g. same patterns can be discussed under different names in different sources. The work aimed at remedying this situation is in progress – e.g. the book solely devoted to security patterns by Schumacher et al. [7] has been just published; the web-page devoted to security patterns has been created (http://www.securitypatterns.org) and is developing, etc.

Second, the available collections of security patterns seem to largely ignore low-level security patterns. The book by Ramachandran [15] is one of few sources, where such low-level patterns (e.g. Sentinel, Validators) are considered. Noteworthy, the core of solutions to such low-level problems (e.g., "validating input in forms") can be found in a variety of books on secure software development; however, we have not managed to find any security pattern collections describing them.

Finally, security patterns may need to be presented in a format different from the traditional pattern presentation format, due to the peculiarities of the concepts and methods employed in the domain of information security. For example, such pattern presentation may include the description of the addressed threats, attacks, supported security goals, etc. Such security-specific presentation format is not yet provided, however, for security patterns.

Acknowledgments

This research was done in MODPA research project (http://www.titu.jyu.fi/modpa) at the Information Technology Research Institute, University of Jyväskylä. MODPA project was financially supported by the National Technology Agency of Finland (TEKES) and industrial partners Nokia, SysOpen Digia, SESCA Technologies, Tieturi, Metso Paper, and Trusteq. The work was partly supported by the Dept. of Mathematical Information Technology, University of Jyväskylä.

References

1. McGraw, G.: Software security. IEEE Security & Privacy Magazine **2**(2) (2004) 80–83
2. Yoder, J., Barcalow, J.: Architectural patterns for enabling application security. In: Proceedings of PLoP 97. (1997)
3. Blakley, B., Heath, C., members of The Open Group Security Forum: Security design patterns. Technical Guide No. G031, The Open Group (2004)
4. Pressman, R.S.: Software engineering: a practitioner's approach. European adaptation, 5-th edition edn. New York, McGraw-Hill (2000)
5. Baskerville, R.: Information systems security design methods: implications for information systems development. ACM Computing Surveys **25**(4) (1993) 375 – 414

6. Schumacher, M.: Security Engineering with Patterns: Origins, Theoretical Model and New Applications. Springer (2003)
7. Schumacher, M., Fernandez, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. Wiley (2006)
8. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press (1977)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley professional computing series. Addison-Wesley, Boston, Mass. (1995)
10. Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M.: Pattern-oriented Software Architecture. A System of Patterns. John Wiley & Sons, UK (1996)
11. Noble, J., Weir, C.: Small Memory Software: Patterns for Systems with Limited Memory. Software patterns series. Addison-Wesley Professional (2001)
12. Alur, D., Crupi, J., Malks, D.: Core J2EE Patterns: Best Practices and Design Strategies. Second edition edn. Prentice Hall PTR / Sun Microsystems Press (2003)
13. Priebe, T., Fernandez, E., Mehlau, J., Pernul, G.: A pattern system for access control. In: Proceedings of the IFIP WG 11.3 Conference on Data and Applications Security, Sitges (2004) 235–249
14. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison Wesley (1999)
15. Ramachandran, J.: Designing Security Architecture Solutions. Wiley Computer Publishing (2002)
16. Lipner, S.B.: The trustworthy computing security development lifecycle. In: 2004 Annual Computer Security Applications Conference, IEEE Computer Society (2004)
17. Peterson, G.: Collaboration in a secure development process. part 1. Information Security Bulletin **9** (2004) 165–172
18. Peterson, G.: Collaboration in a secure development process. part 2. Information Security Bulletin **9** (2004) 205–212
19. Peterson, G.: Collaboration in a secure development process. part 3. Information Security Bulletin **9** (2004) 263–266
20. Sindre, G., Opdahl, A.L.: Templates for misuse case description. In: Seventh International Workshop on Requirements Engineering: Foundation for Software Quality. (2001)
21. Grance, T., Hash, J., Stevens, M.: Security considerations in the information system development life cycle. NIST Recommendations, Special Publication 800-64 REV. 1, National Institute of Standards and Technology, Gaithersburg, MD 20899-8930 (2004)
22. Redwine, S.T.J., Davis, N.: Processes to produce secure software. towards more secure software. In: National Cyber Security Summit. Volume Volume I. (2004)
23. Yacoub, S.M., Ammar, H.H.: Pattern-Oriented Analysis and Design. Composing Patterns to Design Software Systems. Boston, Addison-Wesley (2004)
24. Schmidt, D.: Using design patterns to develop reusable object-oriented communication software. CACM (Special Issue on Object-Oriented Experiences, Mohamed Fayad and W.T. Tsai Eds.) **38**(10) (1995) 65–74
25. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison Wesley Longman (2001)
26. Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.: Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Transactions on Software Engineering **28**(6) (2002) 595–606
27. Apvrille, A., Pourzandi, M.: Secure software development by example. IEEE Security & Privacy Magazine **3**(4) (2005) 10–17