The ISDF Framework: Integrating Security Patterns and Best Practices

Abdulaziz Alkussayer and William H. Allen

Department of Computer Science
Florida Institute of Technology
Melbourne, FL, USA
alkussaa@fit.edu, wallen@fit.edu

Abstract. The rapid growth of communication and globalization has changed the software engineering process. Security has become a crucial component of any software system. However, software developers often lack the knowledge and skills needed to develop secure software. Clearly, the creation of secure software requires more than simply mandating the use of a secure software development lifecycle, the components produced by each stage of the lifecycle must be correctly implemented for the resulting system to achieve its intended goals. In this paper, we demonstrate that a more effective approach to the development of secure software can result from the integration of carefully selected security patterns into appropriate stages of the software development lifecycle to ensure that security designs are correctly implemented. The goal of this work is to provide developers with an Integrated Security Development Framework (ISDF) that can assist them in building more secure software intuitively.

1 Introduction

Until recently, security in software development was viewed as a patch deployed to solve security breaches, or sometimes as an enhancement to an already completed software package. As a result, security considerations were located towards the end of the development lifecycle; particularly as add-on mechanisms and techniques before the system was deployed at the client's premises. Security issues were often raised only after some undetected vulnerability had been compromised. It was not yet understood that developing secure software requires a careful injection of security consideration into each stage of the software development lifecycle [1,2,3,4]. However, once the importance of designed-in security was recognized, attention was directed towards improving the development process by considering security as a requirement instead of a corrective measure.

The inspiration for our previously proposed ISDF framework [5] came from recognizing the existence of two common software development pitfalls: i) security is often only an afterthought in software development; ii) many security breaches exploit well-known security problems. The first issue can only be corrected by mandating the use of a secure development lifecycle to incorporate security considerations across all software development stages. The second must be solved by ensuring that software developers make use of security patterns to avoid insecure development practices.

Fortunately, software security engineering has matured in recent years. Software developers have become more conscious of the fact that security has to be built within the system rather than on the system [4,6,7]. Thus, software security research has been active in two areas: improving engineering best-practices and increasing the use of security knowledge during development.

To address the first area, significant work has been done to formulate a methodology that considers security throughout the secure software development lifecycle (SDLC). The objective is to provide a set of development guidelines and rules on how to build more secure software. Among the many advantages of such methodologies is the ability to equip software developers with easy-to-follow security guidelines. These methodologies represent the best known engineering practices for building secure software. Two well-documented approaches are the Security Development Lifecycle (SDL) [8] and Software Security TouchPoints [9]. A recent discussion of both approaches can be found in the Fundamental Secure Software Development initiative by SAFECode[1] [10].

It has recently been recognized that security knowledge may be encapsulated within security patterns. A pattern describes a time-tested generic solution to a recurring problem within a specific context [11]. Since 1977 when patterns were first introduced by Alexander, et al. [12], they have become a very popular method of encapsulating knowledge in many disciplines. In software engineering, design patterns and security patterns have gained significant attention from the research community. Moreover, design patterns have become increasingly popular since publication of the Gang-of-Four (GoF) book [13]. Although design patterns have been widely adopted in most of today's development libraries and programming tools, the use of security patterns is more recent. They gained popularity following the seminal work by Yoder and Barcalow [14] which presented seven architectural patterns that are useful in developing the security aspects of a system. They used natural language and (GoF) templates to describe their patterns. Since then, many other security patterns have been published. Although many of the published security patterns are considered to be merely guidelines or principals [15], security patterns have been proven to be effective methods of dealing with security problems in a software system. Nevertheless, significant effort and security expertise are needed to properly apply them to a real software development situation.

In this paper, we demonstrate how the ISDF framework can be used to integrate the two independent security solutions mentioned above. First, we describe how the ISDF framework incorporates the best features of existing secure SDLCs. Then, we explain a four-stage utilization process for employing security patterns during the development lifecycle. Finally, we present a practical example that illustrates the benefits of using the ISDF framework during software development

[1] SAFECode is a global industry-led effort to identify and promote best-practices for developing and delivering more secure software and hardware services.

and shows that our combined approach consolidates the secure development best practices that are incorporated into a secure SDLC with the security knowledge built into security patterns. The authors are aware that the addition of a metrics component is necessary to measure the effectiveness of the framework and are working to incorporate security metrics into the ISDF. The results of our metrics-related work will be presented in a future paper.

The rest of this paper is structured as follows. Section 2 provides a brief overview of secure software development and security patterns. Section 3 provides an overview of related work. Section 4 describes the ISDF framework. Section 5 presents a practical example that illustrates the use of the ISDF framework to effectively develop more secure software. Section 6 contains our conclusions and a brief discussion of future work.

2 Background

In recent literature, a number of approaches for developing secure software are discussed. The Fundamental Secure Software Development guide by SAFECode [10] presents a six-phase software development cycle and discusses the best industry practices required during each phase to produce more secure software. The development phases are: requirements, design, programming, testing, code integrity and handling, and documentation [10]. This guide serves as the main source of security best practices that are incorporated into our framework.

Two well-known secure development methodologies are Microsoft's SDL, which first appeared as a result of the Trustworthy Computing Initiative in 2002 [8], and Software Security TouchPoints, which was proposed by Gary McGraw in 2004 [9]. Although there are differences between these methodologies, they agree on three key points [6,7,10]:

1. Advocate security education
2. Risk management is essential
3. Utilization of best practices is crucial

Microsoft's SDL is based on thirteen stages, spanning the entire development lifecycle [6]. These stages are: education & awareness, project inception, defining and following design best practices, product risk assessment, risk analysis, creating security documents/tools/best-practice for customer, secure coding policies, secure testing policies, the security push, the final security review, security response planning, product release, and security response execution. The software development artifacts mandated by Microsoft's SDL methodology are: requirements, design, implementation, verification, release, and support & services [6].

The Software Security Touchpoints methodology depends on the following seven best practices: code review, architectural risk analysis, penetration testing, risk-based security testing, abuse cases, security requirements, and security operations [9]. It also mandates six development phases: requirements and use cases, architecture and design, test plan, code, tests and test results, and feedback from the field [7].

Security patterns have become a reliable approach for effectively addressing security considerations during implementation of a software system. The security patterns book [11] includes forty six patterns. Twenty five of these security patterns address security issues during the design phase. Many of these patterns are well structured and hence the use of UML diagrams to represent such patterns is common. For example, Fernandez and Pan [16] used UML diagrams to illustrate four security patterns: Authorization, Role Based Access Control, Multilevel Security, and File Authorization patterns.

There are also several model-based security patterns. Hatebue et al. [17] presented security patterns using the Security Problem Frame which is used to capture and analyze security requirements. Horvath and Dorges [18] use Petri Nets to model patterns for multi-agent systems, such as the Sandbox and Message Secrecy patterns. Supaporn et al. [19] proposed a more formal method by constructing an extended-BNF-based grammar of security requirements from security patterns.

The use of security patterns during development is essential for building secure software. The richness of the number of security patterns published is encouraging. However, in many cases pattern designers do not provide clear information on when to apply their patterns within the software development lifecycle [20] and selection of the right pattern at the right development stage is not an easy task.

3 Related Work

Researchers have begun to focus on integrating security patterns into a software development lifecycle. For example, Aprville and Pourzandi [21] investigated the use of security patterns and UMLsec [22] in some phases of a secure software development lifecycle but were hampered by the limited range of patterns available at the time. They also did not describe how patterns could be incorporated into a secure SDL to create a development framework.

Valenzuela recommended a methodology that integrates the ISO 17799 (an international Information Security Standard) with a software development lifecycle [23]. This approach proposes parallel security activities for each stage of the SDL and included a mapping of each stage to the appropriate phase of the ISO 17799 process.

Fernandez et al. [24] proposed a methodology that incorporates security patterns in all development stages of their own lifecycle. Their approach includes guidelines for integrating security from the requirements phase through analysis, design, and implementation. In a more recent paper, Fernandez et al. [25] proposed a combination of three similar methodologies into a single unified approach to build secure systems using patterns, but did not integrate them into an industry-recognized Secure SDL.

Existing studies have focused on using either security patterns or best practices - or a loose combination of the two - to build secure software. However, none have explored the need for a concrete method to incorporate the full strength of the two approaches. In the following section, we present a framework that integrates the strengths of both of these well-proven software development techniques.

4 The ISDF Framework

The Integrated Security Development Framework (ISDF) consists of two main components, as shown in Figure 1. The first component is the secure software development best practice, represented on the left-hand side of Figure 1. The second component is a four-stage security pattern utilization process which appears in the right-hand side of the figure. The left hand side of Figure 1 shows the ISDF mandating a development lifecycle. However, this framework does not represent a particular development lifecycle and hence a conventional development model with six phases is used. These phases are very common in any development model. Also, the short activities included in each phase are summarized from [6,7,10] collectively. These activities represent the best engineering practices for developing secure software.

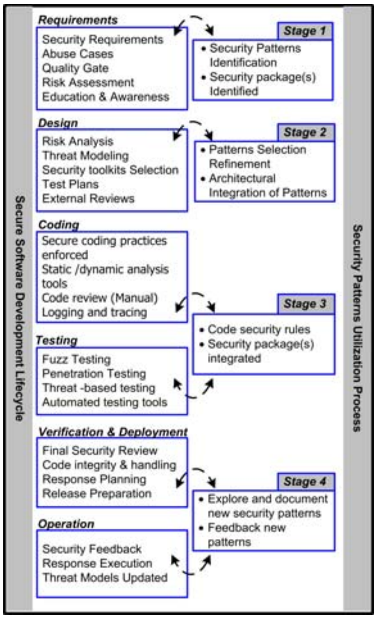

Fig. 1. Integrated Security Development Framework

The relation between the best practices activities and security patterns activity is bidirectional. Thus, the key success factor for seamless integration involves interweaving best practices and security pattern activities at every development stage.

Next, we explain how our framework effectively merges security patterns into the secure software development process.

4.1 Requirements Stage

In this stage, security patterns are selected based on the security requirements and on an analysis of potential threats that are determined from the preliminary risk assessment. For example, Access Control patterns and Identification and Authorization patterns [11] can be identified during this stage. Unfortunately, many practitioners unconsciously postpone the decision of identifying and selecting security patterns to the design phase. Since security patterns evolved from design patterns, however, identifying security patterns reveals more than just a design solution. It places security constraints on the system as a whole, as well as on its subcomponents. These security constraints must be rationalized by measurable security requirements and their associated risks must be mitigated. The relationship between security requirements and security patterns is vital. Some researchers [17,19] have investigated this relationship and proposed promising solutions. However, more research is needed in this area.

Moreover, security components (e.g. firewalls) should be identified at this stage in parallel with the corresponding identification of security tools based on best practices. In fact, many practitioners often don't consider security components until the implementation phase. While this delay is somewhat understandable in the sense that the selected security component must be integrated programmatically into the system, the selection of a security component may best be addressed at the requirement stage because subsequent risk assessment and security pattern selection may be affected by this decision. Security pattern repositories, such as [26], and pattern classification and categorization methods, such as [27], can be useful during the pattern selection and identification process.

4.2 Design Stage

Of course, not all security patterns can be identified as early as the requirement stage. Security patterns may be identified during the design stage to leverage some design constraints. Furthermore, some of the security patterns selected at the previous stage may not adhere to the proposed architecture of the system. Hence, a refinement selection activity should be expected at this stage to resolve such issues.

The SAFECode documentation [10] suggests that every security pattern reveals a solution that "consists of a set of interacting roles that can be arranged into multiple concrete design structures". If the structures of security patterns are aligned with the other design structures of the system, an architectural integration between all the structural components and their interrelationships is produced.

Although many security design pattern studies have been published, as described in section 2.2, the Open Group presented a very coherent design methodology to improve security [28]. This technical report proposed the use of three generative sequences (one main and two sub-sequences) for applying security patterns during the design stage. The main sequence is the System Security Sequence and the sub-sequences are the Available System Sequence and the Protected System Sequence [28].

4.3 Implementation Stage

In this stage, the security rules produced during the design phase are coded based on the secure coding best practice. The selected security components are integrated with the corresponding system components according to the architectural design. No security patterns exist specifically for this stage [20]. However, many secure development methodologies can use published attack patterns as a security education tool and sometimes as test case drivers.

Also, rigorous threat-based testing for structural components of the pre-selected patterns is fundamental in this stage. Thus, the ISDF anticipates the adherence to the best practices of coding and testing mandated by the secure development lifecycle in the coding and testing phases, respectively.

4.4 Post Implementation Stage

This stage corresponds to the last two stages of the secure software development lifecycle in the ISDF, namely deployment and operation. The transition between deployment and operation always raises a critical security concern; carrying the integrity and authenticity of the software source code throughout its chain of custody. The code integrity and handling practice during the deployment phase addresses this concern. However, we strongly believe that there is a need for a new security pattern to effectively safeguard this transition in parallel with the above mentioned practice.

After the software is deployed into its operational environment, it is important to monitor responses to flaws and vulnerabilities of the system to check for new evolved patterns. Note that it is important to avoid simply declaring that the individual code batches and bug fixes represent new patterns. Once a new security pattern has been found and documented, then feedback of the new pattern has to go back to the requirement stage for further security improvement in the consequent releases.

During the operational lifetime of the system, it is essential to revisit the requirement and design stages before implementing the new security countermeasures that resulted from new security threats and attacks. In fact, many recent software security vulnerabilities exist because of the lack of a thorough consideration of the countermeasure defenses implemented at the earlier stages.

5 An Example

To better illustrate the advantages of our framework, we use a simple e-commerce system (called eShop) as an example. An e-commerce example is used because of the popularity and clarity of its prime functionalities. The aim is to demonstrate the effectiveness of the interweaving between best practices and security patterns provided by our framework. While it is impossible to present the entire case study due to space limitations, a subset of the case will be presented covering the requirements and design stages described in Section 4.

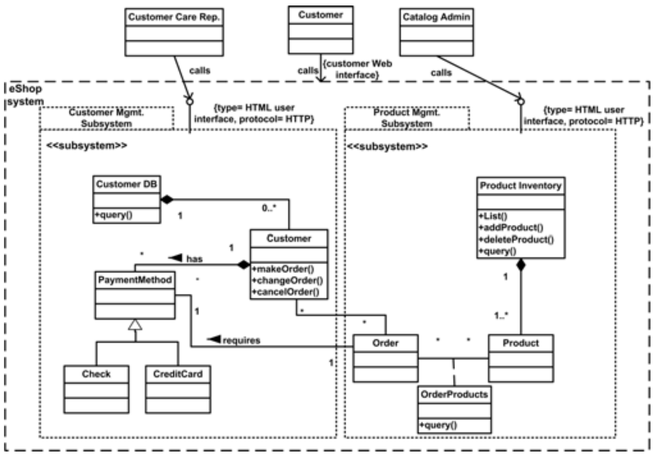

Fig. 2. eShop Preliminary Design

The system, as depicted in Figure 2, has three external (remotely connected) user groups: customers, product catalog administrators, and customer care representatives. Note that, the intent of this diagram is to show some structural components of the eShop system and hence it does not strictly follow the formal UML class diagram notation. The functional requirements of each user group can be summarized as follows.

- *Customers*: browse products and place orders.
- *Product catalog administrators*: remotely manage the product catalog.
- *Customer care representatives*: remotely manage customers' data and orders.

For the non-functional requirements, we focus exclusively on security.

5.1 Stage 1: Requirements

As mentioned earlier, to better employ the strength of the framework, one must work through the best practice activity and the security pattern activity in parallel fashion. The following is a subset of the security requirements of the eShop.

Sq1: *The system shall enforce authentication of users in a secure manner.*
Sq2: *The system shall be able to log and trace back all customer transactions.*
Sq3: *The system shall ensure the privacy and protection of customer data and order transactions.*

These security requirements explicitly impose the need to satisfy some related security properties. The first requirement forces the confidentiality of the access control technique. The second requirement imposes the accountability of customers. The third requirement impresses the confidentiality and integrity of the customer data and transactions. Now that the security properties of the requirements are clear, security patterns can be identified. For the first requirement, `Authentication Enforcer` [29] pattern is selected to handle the problem of how to verify that a subject (customer) is really who they say they are. Next, the accountability property imposes two sub objectives: auditing and non-repudiation [11]. Non-repudiation focuses on capturing evidence so that users who engage in an event cannot later deny that engagement. Auditing refers to the process of monitoring and analyzing logs to report any indication of a security violation. The `Audit Interceptor` [29] is selected to intercept and log requests to satisfy the auditing objective imposed by the second requirement. Finally, the `Secure Pipe` [29] was chosen to fulfill the third requirement and prevent 'man-in-the-middle' attacks. Note that even though the process of identifying the correct patterns is a bit difficult and requires a certain level of security expertise, it can be simplified by consulting an organized security patterns inventory like [26].

5.2 Stage 2: Design

It is expected that not all security patterns can be discovered as early as the requirement stage. In fact, most patterns identified here are a result of architectural design constraints or as a mitigation strategy to identified threats. Thus, the patterns selection refinement process is crucial in this stage.

It is obvious that with multiple entry points to the system (i.e., an entry point for each user group), some of the patterns identified during the requirement stage may need to be replicated (e.g., the authentication mechanism). As shown in Figure 2, three user group entities are interacting with the system at three distinct entry points. However, allowing more entry points may increase the system's risk exposure. For example, a malicious attacker could attempt to impersonate a legitimate user to gain access to his/her resources. This could be particularly serious if the impersonated user has a high level of privilege like a *customer care representative role* or a *product catalog administrator role*. The imposter may then be able to compromise the system and disclose customer credit information. In addition, an intensive threat modeling process must be utilized during this stage to capture the range of potential threats. The design constraints and potential threats identified in this stage collectively influence the refinement process for the preselected patterns. As mentioned above, replicating the authentication mechanism over multiple entry points is problematic and may increase exposure to risks.

One possible solution is to unify the system's entry points into a single point of responsibility. This simplifies the control flow since everything must go through a `Single Access Point` [14]. Figure 3 depicts a refinement of the selected patterns integrated into the eShop architectural design. As an abstraction to simplify the design, we encapsulated the eShop internal entities in a single entity called *eShop inner components*.

Along with the `Single Access Point`, access requests must be validated and authenticated by some type of `Check Point` [11]. A `Check Point` establishes a critical section where security must be enforced through a security policy

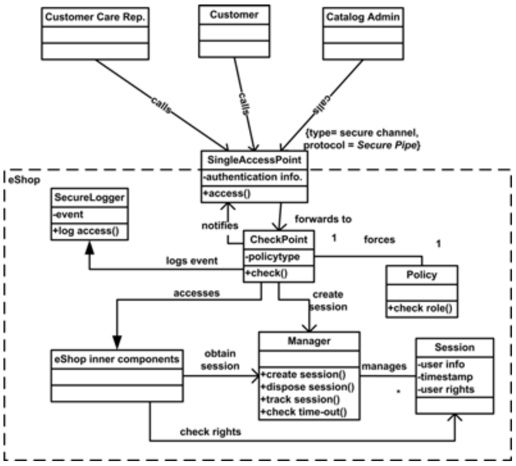

Fig. 3. eShop Integrated Security Patterns

that incorporates different organizational security policies. In terms of the eShop system, the `Check Point` receives a request from the `Single Access Point` and provides validation for a user's options within the restrictions of group policy. Then, the `Check Point` uses a `Secure Logger` [29] to log the event in a secure manner. If access is granted, the `Check Point` then instantiates a `Session` [29] object for the user. The `Session` object holds those security variables associated with the user that may be needed by other components. The `Check Point` uses a *Manager* component to keep track of active session objects. The *eShop inner components* entity authorizes the user by asking the *Manager* for the underlying session object and checking the user's data stored there.

Finally, access to sensitive resources (such as *product inventory* and *customer DB*) may require additional authentication. Therefore, an `Authenticator` [11] can be used to further verify the identity of the subject prior to granting access to these resources. This places an extra, yet important, defensive shield against malicious attacks like those described earlier. However, the `Authenticator` is not shown in Figure 3 due to simplification reasons.

6 Conclusion and Future Work

Our work proposes an integrated framework for developing secure software based on the combination of secure development best practices and security patterns. We also present a four-stage development engineering process to better utilize security patterns with software secure development methodologies. Furthermore, we illustrated how the ISDF framework can be utilized to build more secure software.

Our framework yields two main contributions toward efforts to advance the engineering process to construct more secure software. First, the ISDF framework uniquely consolidates the security patterns with software development best practices. Combining the two will not only simplify the process of building more secure software, but also reduce the risks associated with using ad-hoc security approaches in software development. Second, the ISDF framework enables developers with limited security experience to more easily and more reliably develop secure software.

Our approach also helps to resolve two issues noted in the security patterns literature. The first is the observation that 35 percent of the published patterns do not pass the soundness test for patterns and, therefore, are considered to be guidelines or principals and not formal patterns [15]. For example, security patterns like `Asset Valuation` and `Threat Assessment` [11] don't conform to the formal definition of a security pattern [15,20]. However, since the ISDF incorporates best practices to guide secure development, there is no need to utilize those types of pattern.

The second issue is the lack of patterns for some parts of the development phase (e.g., the small number of attack patterns for use in the design phase) [20]. Our framework resolves this limitation by mandating concrete best practices in parallel with security patterns.

In the future, we will continue to work towards the formalization of the ISDF framework and the discovery of security metrics that can be measured at early stages of the development lifecycle. Also, we will investigate the reversibility of the framework on a legacy system.

Acknowledgment

We would like to thank the Institute of Public Administration (IPA)in Saudi Arabia for their support to this work.

References

1. Viega, J., McGraw, G.: Building Secure Software. Addison-Wesley, Reading (2002)
2. Davis, N., Humphrey Jr., W., Zibulski, S.R., McGraw, G.: Processes for producing secure software. IEEE Security & Privacy 2(3), 18–25 (2004)
3. Howard, M.: Building more secure software with improved development process. IEEE Security & Privacy 2(6), 63–65 (2004)
4. Jayaram, K.R., Mathur, A.: Software engineering for secure software- state of the art: A survey. Technical report, Purdue University (2005)
5. Alkussayer, A., Allen, W.H.: Towards secure software development: Integrating security patterns into a secure SDLC. In: The 47th ACM Southeast Conference (2009)
6. Howard, M., Lipner, S.: The Security Development Lifecycle SDL: A Process for Developing Demonstrably More Secure Software. Microsoft Press (2006)
7. McGraw, G.: Software Security: Building Security. Addison-Wesley, Reading (2006)
8. Howard, M., Lipner, S.: Inside the windows security push. IEEE Security & Privacy 1(1), 57–61 (2003)
9. McGraw, G.: Software security. IEEE Security & Privacy 2(2), 80–83 (2004)

10. Simpson, S.: Fundamental practices for secure software development: A guide to the most effective secure development practices in use today (2008), `http://www.safecode.org`
11. Schumacher, M., Frenandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. John Wiley & Sons, Chichester (2006)
12. Alexander, C., Ishikawa, S., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, Oxford (1977)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison- Wesley Professional (1995)
14. Yoder, J., Barcalow, J.: Architecural patterns for enabling application security. In: PLoP 1997 Conference (1997)
15. Heyman, T., Yskout, K., Scandariato, R., Joosen, W.: An analysis of the security patterns landscape. In: 3rd International Workshop on Software Engineering for Secure Systems (2007)
16. Fernandez, E.B., Pan, R.: A pattern language for security models. In: PLoP 2001 Conference (2001)
17. Hatebur, D., Heisel, M., Schmidt, H.: Security engineering using problem frames. In: International Conference on Emerging Trends in Information and Communication Security (ETRICS) (2006)
18. Horvath, V., Dorges, T.: From security patterns to implementation using petri nets. In: International Conference on Software Engineering (2008)
19. Supaporn, K., Prompoon, N., Rojkangsadan, T.: An approach: Constructing the grammar from security patterns. In: 4th International Joint Conference on Computer Science and Software Engineering (JCSSE 2007) (2007)
20. Yoshioka, N., Washizaki, H., Maruyama, K.: A survey on security patterns. Progress in Informatics (5), 35–47 (2008)
21. Aprville, A., Pourzandi, M.: Secure software development by example. IEEE Security & Privacy 3(4), 10–17 (2005)
22. Jurjens, J.: Secure System Development with UML. Springer, Heidelberg (2004)
23. Valenzuela, I.: Integration ISO17799 into your software development lifecycle. Secure 11, 29–36 (2007)
24. Fernandez, E.B.: A methodology for secure software design. In: International Conference on Software Engineering Research and Practice (2004)
25. Fernandez, E.B., Yoshioka, N., Washizaki, H., Jurjens, J.: Using security patterns to build secure systems. In: 1st International Workshop on Software Patterns and Quality (SPAQu 2007) (2007)
26. Yskout, K., Heyman, T., Scandariato, R., Joosen, W.: An inventory of security patterns. Technical report, Katholieke Univerity Leuven, Department of Computer Science (2006)
27. Hafiz, M., Adamczyk, P., Johnson, R.E.: An organizing security patterns. IEEE Software 24(4), 52–60 (2007)
28. Blakley, B., Heath, C.: of the Open Group Security Forum, M.: Security design patterns. Technical report, Open Group (2004)
29. Steel, C., Nagappan, R., Lai, R.: Core Security Patterns: Best Practices and Strategies forJ2EE, Web Services, and Identity Managment. Prentice-Hall, Englewood Cliffs (2005)