Security Patterns related to Security Requirements

David G. Rosado[1], Carlos Gutiérrez[2], Eduardo Fernández-Medina[1] and Mario Piattini[1]

[1] ALARCOS Research Group. Information Systems and Technologies Department
UCLM-Soluziona Research and Development Institute. University of Castilla-La Mancha
Paseo de la Universidad, 4 – 13071 Ciudad Real, Spain

[2] STL. Calle Manuel Tovar 9, 28034 Madrid, Spain

Abstract. In the information technology environment, patterns give information system architects a method for defining reusable solutions to design problems. The purpose of using patterns is to create a reusable design element. We can obtain, in a systematic way, a security software architecture that contains a set of security design patterns from the security requirements found. Several important aspects of building software systems with patterns are not addressed yet by today's pattern descriptions. Examples include the integration of a pattern into a partially existing design, and the combination of patterns into larger designs. Now, we want to use these patterns in our architectures, designs, and implementations.

1 Introduction

It is very common not to consider security at the first stages of systems development but to deal with it once the system has been designed and implemented. However, those aspects known as "quality requirements" [4] [16], being security one of them, must be described in a concrete way before the system architecture is designed [3]. The worldwide security/business continuity market is showing good growth and is forecasted to gain momentum by 2005, reaching a 15% growth rate, all translating into over $118 billion in spending by 2007. All segments - hardware, software, and services - will lead growth uniformly as enterprises seek to improve their infrastructures to manage organisational risks more effectively [20].

Ignoring security issues is dangerous because it can be difficult to retrofit security in an application [30]. As the statistics presented by the CERT show, the number of incidents related to security have exponentially grown during the last years (they have passed from 3734 incidents reported in 1998 to 137529 in 2003; Total incidents reported (1988-2003): 319,992) [9].

Security patterns are proposed as a means of bridging the gap between developers and security experts. Security patterns are intended to capture security expertise in the form of worked solutions to recurring problems. Security patterns are also intended to be used and understood by developers who are not security professionals [21]. The first person who used the pattern approach was Christopher Alexander [1], and in his book he indicated that each pattern describes a problem which occurs over and over

Delgado J. (2006).
Protecting Notification of Events in Multimedia Systems.
In *Proceedings of the 4th International Workshop on Security in Information Systems*, pages 163-173

again in our environment, and then states 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. The "Gang of Four" book, as it is commonly known, defined design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [19].

Design patterns tell their readers how to design a system, given a statement of a problem and a set of forces that act upon the system. In the information technology environment, patterns give information system architects a method for defining reusable solutions to design problems without ever having to talk about or write program code; they are truly programming language-independent.

The purpose of using patterns is to create a reusable design element. Each pattern is useful in and by itself. The combination of patterns assists those responsible for implementing security to produce sound, consistent designs that include all the operations required, and so assure that the resulting implementations can be efficiently completed and will perform effectively [6].

One of problems we face in everyday practice is that we want to secure an application without spending excessive time and effort; for this reason, we are tempted to use some known solutions like putting up a firewall or using simple password authentication. Applying a pattern, a solution that has already been extensively used in practice, might seem to be a reasonable idea. In many cases, however, a solution applied without a thorough understanding of security requirements does not provide adequate protection within the specific context [23].

In this paper, we will study the most important security requirements types, and then we will look for a set of security patterns that covers all of the security requirements specified and these patterns will help us create reference security architecture where all security requirements are covered.

The rest of the paper is organized as follows: in Section 2, we will show several security requirements types. In Section 3, we will discuss security patterns, we will give a catalogue of architectural patterns and design patterns grouping together by requirement types that fulfil and we will finish showing a pattern example. Finally, we will put forward our conclusions and future work.

2 Requirements Security

In this section, we will select the most commonly used attributes and security properties in the security dominion; the defined criteria are based on the works of Babar [2] and Firesmith [17]. The selected security properties are the following:

- Authentication: the system verifies the identities of its externals before interacting with them. It must be validated the identity of customers to frustrate any disauthorized access.
- Authorization: access and usage privileges of authenticated externals are properly granted and enforced. This attribute defines the access privileges of entities to different resources and services of a system.
- Integrity: there should be a mechanism to protect data from unauthorized modification while data are stored in an organizational repository or are

transferred into a network. It should ensure that data and communications will not be compromised by active attacks.

- Confidentiality: sensitive information is not disclosed to unauthorized parties (e.g., individuals, programs, processes, devices, or other systems). A system should ensure data and communication privacy from unauthorized access. Resource hiding is an important aspect of confidentiality.
- Attacker detection: attempted or successful attacks (or their resulting harm) are detected, recorded, and notified. It consists of being able to detect and register access or modification intents in the system coming from disauthorized users.
- Non-Repudiation: a party of an interaction (e.g., message, transaction, transmission of data) is prevented from successfully repudiating (i.e., denying) any aspect of the interaction. It prevents that certain participant in certain interaction can deny to have participated in it.
- Security Auditing: security personnel are enabled to audit the status and use of security mechanisms by analyzing security-related events. This means keeping a log of users' or other systems' interaction with a system. It helps detect potential attacks, find out what happened after assaults, and gather evidence of abnormal activities.
- Maintainability: It facilitates the introduction or modification of the security policy during the software development life cycle.
- Availability: It assures that authorized users can use the resources when they are required. It ensures that authorized users can access data and other resources without any obstruction or disturbance. If a disaster occurs, it ensures that a system recovers quickly and completely.

3 Searching/Defining Security Patterns Based on Security Requirements

3.1 Security Patterns

Patterns can be grouped into three categories according to their level of abstraction [8] (high-level, mid-level and low-level): i) An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. ii) A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context. iii) An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

For Ramachandran [24], a pattern is a common and repeating idiom of solution design and architecture. Ramachadran defines a pattern as a solution to a problem in the context of an application. Security components tend to focus on hardening the system against threat to the exclusion of other goals. Patterns bring balance to the

definition of security architecture because they place equal emphasis on good architecture and strong security. Our choices of security properties, authentication mechanisms, and access control models can either drive our architecture towards some well-understood pattern of design or turn us towards some ad hoc solution with considerable architectural tensions. Without a model for security architecture, if we take the latter path we might discover flaws or risks in the solution's construction only at deployment. That might be too late.

Something is a security pattern if: we can give it a name; we have observed its design repeated over and over in many security products; there is a benefit to defining some standard vocabulary dictated by common usage rules to describe the pattern; there is a good security product that exemplifies the pattern; there is value in its definition. The pattern might capture expertise or make complex portions of an architecture diagram more intelligible. The pattern might make the evaluation of a product easier by highlighting departures from a standard way of approaching a problem. It might also raise a core set of issues against these departures [24].

Security patterns provide techniques for identifying and solving security issues; they work together to form a collection of best practices (to support a security strategy) and they address host, network and application security. The benefits of using patterns are: they can be revisited and implemented at anytime to improve the design of a system; less experienced practitioners can benefit from the experience of those more fluent in security patterns; they provide a common language for discussion, testing and development; they can be easily searched, categorized and refactored; they provide reuseable, repeatable and documented security practices; they do not define coding styles, programming languages or vendors [5].

Design strategies determine which application tactics or design patterns should be used for particular application security scenarios and constraints. Security Design patterns are an abstraction of business problems that address a variety of security requirements and provide a solution to the known security related problem(s). They can be architectural patterns that depict how a security problem can be architecturally resolved, or they can be defensive design strategies upon which secure code can be later built [28].

3.1.1 Architectural Patterns

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used. A pattern can be thought of as a set of constraints on an architecture -on the element types and their patterns of interaction - and these constraints define a set or family of architectures that satisfy them. For example, client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients. Use of the term client-server only implies that multiple clients exist; the clients themselves are not identified, and there is no discussion of what functionality, other than implementation of protocols, has been assigned to any of the clients or to the server. Countless architectures are of the client-server pattern under this (informal) definition, but they are different from each other. An architectural pattern is not an architecture but it still conveys a useful image of the system—it imposes useful constraints on the architecture and, in turn, on the system [4].

An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them [8]. An architectural pattern is a high-level abstraction. The choice of the architectural pattern to be used is a fundamental design decision in the development of a software system. It determines the system-wide structure and constrains the design choices available for the various subsystems. It is, in general, independent of the implementation language to be used. Examples of architectural patterns are broker, multi-layer, pipe and filter, transaction-processing, etc.

One of the most useful aspects of patterns is that they exhibit known quality attributes. This is why the architect chooses a particular pattern and not one at random. Some patterns represent known solutions to performance problems, others lend themselves well to high-security systems, still others have been successfully used in high-availability systems. Choosing an architectural pattern is often the architect's first major design choice [4].

3.1.2 Design Patterns

Mature software design patterns, like patterns in any other discipline, capture solutions that have developed and evolved over time. Hence they are not the designs that people tend to generate initially. Mature patterns reflect many iterations of untold redesign and recoding, as developers have struggled for greater reuse and flexibility in their software. Design patterns capture refined solutions in a succinct and easily applied form.

The purpose of using patterns is to create a re-usable design element. Each pattern is useful in and of itself. The combination of patterns assists those responsible for implementing security to produce sound, consistent designs that include all the operations required, and so assure that the resulting implementations can be completed efficiently and will perform effectively [6].

A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context [8]. A design pattern is a mid-level abstraction. The choice of a design pattern does not affect the fundamental structure of the software system, but it does affect the structure of a subsystem. Like the architectural pattern, the design pattern tends to be independent of the implementation language to be used. Examples of design patterns are adapter, composite, delegation, façade, observer, etc.

3.2 Relation between Patterns and Requirements

We propose a set of security patterns that guarantee, in some way, one or several security requirements types, that is, for a security requirement type we know one or several patterns that insure the mentioned requirement. The use of patterns helps us develop a secure system.

In table 1, we can see the relation between security requirements, security services, security architectural patterns and security design patterns that we can use to be sure

that our design based on these patterns fulfils and guarantees these security requirements for the system designed. We have selected a subset of security requirements types of the aforementioned; we have studied several security patterns that drive and guide us towards a secure development as well as towards a security software architecture based on security patterns. Due to space constraints, we will only briefly describe some patterns of those shown in table 1. For more information about these patterns, please see the shown references. Some of the patterns can be described as follows:

- *Audit Interceptor:* This pattern [28] works in conjunction with the Secure Logger pattern and provides instrumentation of the logging aspects in the front. Besides, this pattern enables the administration and manages the logging and audit in the back-end.
- *Secure Logger:* This pattern [28] defines how to capture the application-specific events and exceptions in a secure and reliable manner to support security auditing.
- *Assertion Builder:* This pattern [28] defines how an entity assertion (for example, authentication assertion or authorization assertion) can be built.
- *Secure Pipe:* This pattern [28] shows how to secure the connection between client and server, or between servers when connecting between trading partners. In a complex distributed application environment, there will be a mixture of security requirements and constraints between clients, servers, and any intermediaries. It adds value by requiring mutual authentication and establishing confidentiality or non-repudiation between trading partners. This is particularly critical for B2B integration using Web services.
- *Authoritative Source of Data*: This pattern [25] is used to verify the validity of data and their origin. It prevents the system from using outdated and incorrect information and reduces the potential risk of processing and propagating fraudulent data.
- *Layered Security*: This pattern [26] is aimed at dividing a system's structure into several layers to improve the security of the system by securing all of the layers. One major drawback of using this pattern is that is increases complexity at the architecture level.
- *Check Point*: This pattern [29] [30] centralizes and enforces security policy and encapsulates the algorithm to put the security policy into operation. The algorithm can contain any number of security checks. This pattern can be also used to keep track of the failed attempts of security breaches, which helps take appropriate action if the failures are malicious activities.
- *Data Filter*: This pattern [18] filters the contents of client requests in a distributed system, according to predefined policies. Filtering can occur locally or remotely. In many distributed systems, e.g., the Internet, requests for services or data need to be filtered according to institution policies, legislative restrictions, privacy needs, etc.

Table 1. Requirements, Architectural Patterns and Design Patterns.

Security Requirements	Security Service(s)	Architectural Patterns	Design Patterns
Authentication	Authenticity and Integrity	Data Filter [18], SSO [22] Check Point [29] [30] Cryptographic [27]	Authenticator [12] SSO Delegator [28] Assertion Builder [28] Sender Authentication [7]
Authorization	Authorization Service	Firewall [15] PEP+PDP+PRP+PIP+PAP Data Filter [18] Bodyguard [10] Check Point [29] [30] Cryptographic [27]	RBAC [14] Application Firewall [11] XML Firewall [11] Assertion Builder [28] Authorization [14] Session [29] [30]
Confidentiality	Confidentiality Service	Firewall [15] Layered Security [26] Check Point [29] [30] Cryptographic [27] Encryption [27] Pipes and Filter [8]	Secure Pipe [28] Multilevel Security [14] Session [29] [30] Information Secrecy [7]
Integrity	Integrity Service	Firewall [15] Layered Security [26] Cryptographic [27] Encryption [27] Data Filter [18] Pipes and Filter [8]	Authoritative Source of Data [25] Message Integrity [7] Multilevel Security [14] Session [29] [30]
Non-repudiation	Non-Repudiation Service	Encryption [27] Cryptographic [27]	Secure Pipe [28] Signature [7]
Audit	Audit Service	Check Point [29] [30] Single Access Point [29] [30]	Audit Interceptor [28] Secure Logger [28]

- *Authenticator*: The Authenticator pattern [12] describes a general mechanism for providing identification and authentication to a server from a client. It has the added feature of allowing protocol negotiation to take place using the same procedures. The pattern operates by offering an authentication negotiation object which then provides the protected object only after authentication is successful.
- *BodyGuard*: This pattern [10] allows us to share objects and control their access in a distributed environment without system level support for distributed objects. The Bodyguard is a pattern that simplifies the management of object sharing over a network. It provides message dispatching validation and assignment of access rights to objects in non-local environments, to prevent the incorrect access to an object in collaborative applications.

For example, for the requirement type 'Authorization', the security service that orders the authorization requirement is the Authorization Service (Transport level and Application level), and we know some patterns for developing the mentioned service as the architectural patterns *Firewall*, *BodyGuard*, *Check Point,* etc. that apply the architecture necessary for the carrying out of the authorization and the security design patterns *RBAC*, *Authorization*, *Session*, etc, that implement in some way, the service required by fulfilling the selected security requirements type. Therefore, if we use

these security patterns, we will be able to develop and design the architecture and the mechanisms that achieve and fulfil the desired requirement type.

3.3 Example: Cryptographic Pattern

Modern cryptography is been widely used in many applications, such as word processors, spreadsheets, databases, and electronic commerce systems. The widespread use of cryptographic techniques and the present interest and research on software architectures and patterns led us to cryptographic software architectures and cryptographic patterns. This architecture is composed of many patterns that offer the cryptographic services address application requirements. The focus of these patterns is on the security properties of confidentiality, integrity, authentication and non-repudiation. The *Information Secrecy* pattern describes how to keep messages secret from an attacker (confidentiality). The *Message Integrity* pattern shows how to prevent that an attacker modifies or replaces messages without the sharing of cryptographic keys. The *Sender Authentication* pattern illustrates how messages can be authenticated with the usage of cryptographic keys. The *Signature* pattern describes how it can be prevented that communicating parties cannot repudiate a message (non-repudiation). Based on these generic cryptographic patterns, we can generate more patterns composing one pattern with other, obtaining in a same pattern the fulfilment of several security requirements, for example, *Secrecy with Integrity* pattern is the result of linking the patterns *Information Secrecy* and *Message Integrity*, where the properties of confidentiality and integrity are ensured at the same time with the use of this pattern.

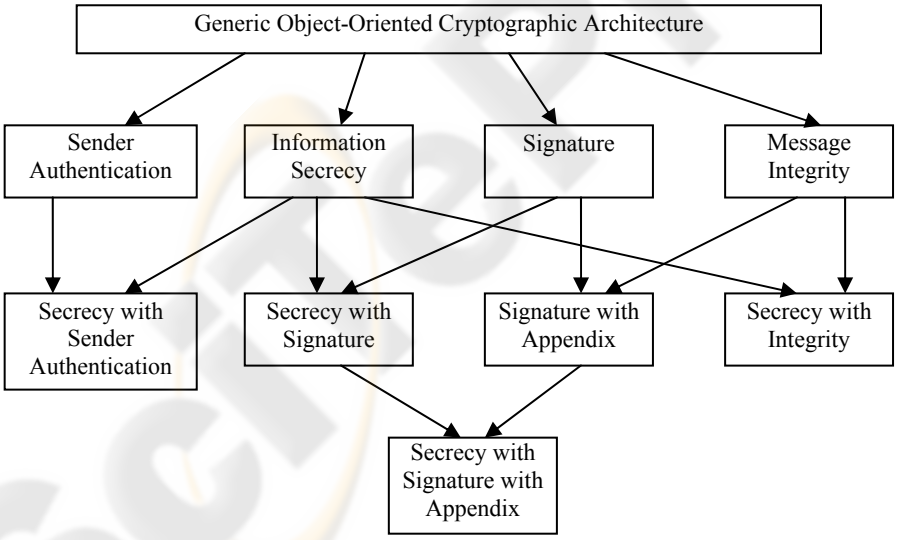

Fig. 1. Relationships between cryptographic design patterns.

In [7], it is defined an architecture based on these patterns called *Generic Object-Oriented Cryptographic Architecture (GOOCA)*, that is a abstraction of all these patterns together forming a generic architecture. Fig. 1 is a directed acyclic graph of dependences among patterns. An edge from pattern A to pattern B shows that pattern A generates pattern B. A pattern that is pointed at by more than one edge has as many generators as the number of edges arriving in it. The GOOCA generates the microarchitecture for the four basic patterns. All other patterns are generated from combinations of these.

4 Conclusions

Architects must make design decisions early in a project lifecycle. Many of them are difficult, if not impossible, to validate and test until parts of the system are actually built. Due to the difficulty of validating early design decisions, architects sensibly rely on tried and tested approaches for solving certain classes of problems. This is one of the great values of architectural patterns. They enable architects to reduce risk by leveraging successful designs with known engineering attributes.

In the past, only software architects engaged in military application development had to learn complex security methodologies. The rapid expansion of e-commerce and internet applications has increased the need for an adequate application security for practically all enterprise applications. The software architects of enterprise applications are faced with a difficult choice.

This paper has presented a security patterns catalogue both architectural and design destined, based on security requirements types that we can consider important for security information systems.

We will take these security requirements specified to create a draft of the candidate security architecture. This candidate architecture will also identify a set of security patterns that covers all of the security requirements within the component architecture and will detail them in a high-level way, addressing the known risks, exposures, and vulnerabilities.

Our future work will be that of studying the different security patterns and getting a method with that we can classify what pattern is best, for requirement type selected, of between the possible patterns to use according to some security property (performance, reliability, degree security, flexibility, etc) and we can use them in our reference architecture previously created.

Acknowledgements

This research is part of the following projects: DIMENSIONS (PBC-05-012-2) financed by FEDER and by the “Consejería de Ciencia y Tecnología de la Junta de Comunidades de Castilla-La Mancha” (Spain), RETISTIC (TIC2002-12487-E) and CALIPO (TIC2003-07804-C05-03) granted by the “Dirección General de Investigación del Ministerio de Ciencia y Tecnología” (Spain).

References

1. Alexander, C., Ishikawa, S., and Silverstein, M., A pattern language: towns, builings, construction. 1977, New York: Oxford University Press.
2. Babar, M.A., Wang, X., and Gorton, I. Supporting Security Sensitive Architecture Design. in QoSA-SOQUA 2005. 2005: Springer-Verlag.
3. Barbacci, M.R., Ellison, R., Lattanze, A.J., Stafford, J.A., Weinstock, C.B., and Wood, W.G., Quality Attribute Workshops (QWAs). Third Edition., in Architecture Tradeoff Analysis Initiative. 2003, Carnegie Mellon. Software Engineering Institute. p. 36
4. Bass, L., Clements, P., and Kazman, R., eds. Software Architecture in Practice. 2nd ed. 2003, Addison-Wesley.
5. Berry, C.A., Carnell, J., Juric, M.B., Kunnumpurath, M.M., Nashi, N., and Romanosky, S., Chapter 5: Patterns Applied to Manage Security, in J2EE Design Patterns Applied.
6. Blakley, B., Heath, C., and TheOpenGroup, Technical Guide. Security Design Patterns. 2004. http://www.opengroup.org/
7. Braga, A.M., Rubira, C., and Dahab, R. Tropyc: A Pattern Language for Cryptographic Software. in 5th Pattern Languages of Programming (PLoP'98) Conference. 1998. Allerton Park, Illinouis, USA.
8. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M., Pattern-Oriented Software Architecture: A System of Patterns. 1st ed. 1996: John Wiley & Sons. 476 Pg.
9. CERT. http://www.cert.org/stats/cert_stats.html#incidents
10. Das Neves, F. and Garrido, A., BodyGuard, in Pattern Languages of Programs III, Addison-Wesley, Editor. 1998.
11. Delessy-Gassant, N., Fernandez, E.B., Rajput, S., and Larrondo-Petrie, M.M. Patterns for Application Firewalls. in 11th Conference on Pattern Languages of Programs (PLoP'2004). 2004. Allterton Park, Monticello, Illinois.
12. F. Lee Brown, J., DiVietri, J., Diaz de Villegas, G., and Fernandez, E.B. The Authenticator Pattern. in 6th Conference on Pattern Languages of Programs, PLoP 1999. 1999. Allterton Park, Monticello, Illinois.
13. Fernandez, E.B. Patterns for Operating Systems Access Control. in 9th Conference on Pattern Languages of Programs, PLoP 2002. 2002. Allerton Park, Illinois, USA.
14. Fernandez, E.B. and Pan, R. A pattern language for security models. in 8th Conference on Pattern Languages of Programs, PLoP 2001. 2001. Allerton Park, Illinois, USA.
15. Fernandez, E.B., Petrie, M.L., Seliya, N., and Herzberg, A. A Pattern Language for Firewalls. in 10th Conference on Pattern Languages of Programs (PLoP'2003). 2003. Allterton Park, Monticello, Illinois.
16. Firesmith, D.G., Commom Concepts Underlying Safety, Security, and Survivability Engineering. 2003, SEI
17. Firesmith, D.G., Specifying Reusable Security Requirements. Journal of Object Technology, 2004. 3: p. 61-75.
18. Flanders, R. and Fernandez, E.B. Data Filter Architecture Pattern. in 6th Conference on Pattern Languages of Programs, PLoP 1999. 1999. Allterton Park, Monticello, Illinois.
19. Gamma, E., Helm, R., Johnson, R., and Vlissides, J., Design Patterns: Elements of Reusable Object-Oriented Software. 1994: Addison-Wesley.
20. IDC. 2005. http://www.idc.com/getdoc.jsp?containerId=prUS00190705
21. Kienzle, D.M. and Elder, M.C., Final Technical Report: Security Patterns for web Application Development. 2005
22. King, C., Osmanoglu, E., and Dalton, C., Security Architecture. 2001: McGraw-Hill.
23. Kis, M. Information Security Antipatterns in Software Requirements Engineering. in 9th Conference on Pattern Languages of Programs (PLoP'2002). 2002. Allterton Park, Monticello, Illinois.
24. Ramachandran, J., Designing Security Architecture Solutions. 2002: John Wiley & Sons.

25. Romanosky, S. Enterprise Security Patterns. in 7th European Conference on Pattern Languages of Programs (EuroPlop'02). 2002. Irsee, Germany.
26. Romanosky, S., Security Design Patterns. 2001. http://www.cgisecurity.com/lib/securityDesignPatterns.html
27. Schumacher, M. and Roedig, U. Security Engineering with Patterns. in 8th Conference on Patterns Lnaguages of Programs, PLoP 2001. 2001. Monticello, Illinois, USA.
28. Steel, C., Nagappan, R., and Lai, R., Core Security Patterns. 2005: Prentice Hall PTR. 1088 Pg.
29. Wassermann, R., Using Security Patterns to Model and Analyze Security Requirements. 2004. p. 151
30. Yoder, J. and Barcalow, J. Architectural Patterns for Enabling Application Security. in 4th Conference on Patterns Language of Programming, PLop 1997. 1997. Monticello, Illinois, USA.