A Study of Security Architectural Patterns.

Conference Paper · January 2006

DOI: 10.1109/ARES.2006.18 · Source: DBLP

David Garcia Rosado
University of Castilla-La Mancha
65 PUBLICATIONS **688** CITATIONS
SEE PROFILE

Eduardo Fernández-Medina
University of Castilla-La Mancha
282 PUBLICATIONS **2,961** CITATIONS
SEE PROFILE

Mario Piattini
University of Castilla-La Mancha
1,097 PUBLICATIONS **11,315** CITATIONS
SEE PROFILE

A Study of Security Architectural Patterns

David G. Rosado[1], Carlos Gutiérrez[2], Eduardo Fernández-Medina[1] y 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
{David.GRosado, Eduardo.Fdez-Medina, Mario.Piattini}@uclm.es
(2) STL. Calle Manuel Tovar 9, 28034 Madrid, Spain
carlos.gutierrez@stl.es

Abstract.

Security and reliability issues are rarely considered at the initial stages of software development and are not part of the standard procedures in development of software and services. Security patterns are a recent development as a way to encapsulate the accumulated knowledge about secure systems design, and security patterns are also intended to be used and understood by developers who are not security professionals. In this paper, we will compare several security patterns to be used when dealing with application security, following an approach that we consider important for measuring the security degree of the patterns, and indicating a fulfilment or not of the properties and attributes common to all security systems.

1. Introduction

A good percentage of the software deployed in industrial/commercial applications is of poor quality and contains numerous flaws that can be exploited by attackers [1, 2]. There are many reasons for this and there is no doubt that we have a serious problem, every day the press reports of attacks to web sites or databases around the world, resulting in millions of dollars in direct or indirect losses [3]. Security is a very important aspect of any computing system, and has become a serious problem since institutions have opened their databases to the Internet [4-6]. Most web systems in current use have not been designed with security in mind and patches have failed to make them more resistant to attacks [7]. It is important to develop systems where security has been considered at all stages of design and at all architectural levels, which not only satisfy their functional specifications but also satisfy security and other non-functional requirements [8, 9].

There is very little work concerning the full integration of security and systems engineering from the earliest phases of software development. Although several approaches have been proposed for some integration of security, there is currently no comprehensive methodology to assist developers of security sensitive systems. Lack of support for security engineering in those approaches for software systems development is usually seen as a consequence of: i) security requirements being generally difficult to analyse and model, and ii) developers lacking expertise in secure software development [10, 11].

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. The first person who used the pattern approach was Christopher Alexander [12], and in his book he indicated that each pattern describes a problem which occurs over and over 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.

This paper will study a set of security patterns that help us to implement security requirements in the applications design. They are patterns that guide the systems design to make them more secure in a comfortable and efficient way. The rest of the paper is organized as follows: In section 2, we will define a template to define patterns and we will study a set of patterns to make the comparison. Then, we will describe the comparison framework that we have used and we will perform the patterns comparison. Finally, we will put forward our conclusions.

2. Template and Security Architectural Patterns Selected

A software pattern can be described through a set of properties (a template) such as name, problem, solution and so on. These templates allow authors to define new patterns, but respecting this structure [13].

In this section, a template will be defined composed of the following properties (based on [14, 15]): i) *Intent*: It describes what the pattern does, which its rationale and intent are, and what particular design issue it addresses. ii) *Context*: It describes the context of the problem. iii) *Problem*: It gives a statement of the problem that this pattern solves. iv) *Description*: A scenario that illustrates a design problem. v) Solution: To give a statement of the solution to the problem. vi) *Consequences*: To describe the trade-offs and results when we use the pattern. vii) *Known uses*: Examples of the patterns found in real systems viii) *Related patterns*: To list other related patterns that use this pattern as a reference.

Once the template has been defined, we are join to present some of the most important security architectural patterns, analyzing characteristics and find out the degree of security that they supply to the systems that use them. These patterns are as follows [13, 16-18] : 1) Authorization Pattern; 2) RBAC Pattern (Role-Based Access Control); 3) Multilevel Security Pattern; 4) Reference Monitor Pattern; 5) Virtual Address Space Access Control; 6) Execution Domain Pattern; 7) SAP Pattern (Single Access Point),; 8) Check Point Pattern; y 9) Session Pattern.

There are many others security patterns that, due to space constraints we can not described with detail, but we can find more information in [13, 17-22].

2.1. Authorization Pattern

i) *Intent*: It describes who is authorized to access the resources systems. ii) *Context*: Any computational environment where there are active entities that request resources whose access must be controlled. iii) *Problem*: The permissions granted for security subjects that have access to protected objects need to be explicitly indicated. On the contrary, any subject could access any resource. iv) *Description*: To structure the different access policies, we distinguish between active entities (subjects) and passive resources (protection objects). v) *Solution*: The Authorization structure (see Figure 1) can be captured from classes and relationships or associations. The active entities are represented by the *Subject* class and the passive resources (or resources to be protected) are represented the by *Object* class. The relationship between subject and object describes what subject is authorized to access certain objects (*Rights*). vi) *Consequences*: The solution is independent of the resources to be protected. The subjects can be executions of processes, users, roles and group of users; the objects to be protected can be transactions, memory area, I/0 devices, files or other resources of the operating system and the type of access can be reading, writing, execution or methods in higher level objects. vii) *Known uses*: It is the basis for the access control systems of most commercial products as Unix, Windows, Oracle and others. viii) *Related patterns*: The RBAC pattern shown later is a specialization of this pattern.

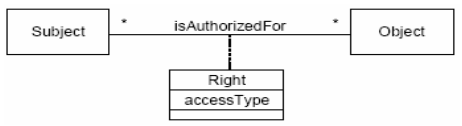

Figure 1. Authorization Pattern.

2.2. RBAC Pattern

i) *Intent*: To control the access resources only based on the subject role. ii) *Context*: Any environment where we need to control the access to computing resources and where users can be classified according to their jobs and tasks. iii) *Problem*: It is necessary to assign rights and permissions (central authority) in an appropriate way for users to be able to access the protected objects. iv) *Description*: It improves the administration by using roles that can be assigned to individual users or groups. We may need to have hierarchies of roles, with inheritance of rights. A role may be assigned to individual users or to groups of users. v) *Solution*: It extends the idea of the *Authorization* pattern by translating roles as subjects. A basic model for RBAC is shown in Figure 2. User and Role classes describe registered users and predefined roles, respectively. Users are assigned to roles, roles are given rights according to their functions and the *Right* association class defines the types of access that a user within a role is authorized to apply to the protection object. The combination Role, Protection Object and Rights is an instance of the Authorization pattern. vi) *Consequences*: When introducing roles, the administrative effort is reduced because there is no need of assigning rights to individuals. The roles structure let us manage big groups as well as reduce rules. vii) *Known uses*: RBAC is implemented in Sun's

J2EE, Microsoft's Windows 2000, IBM's WebSphere, and Oracle, among others.

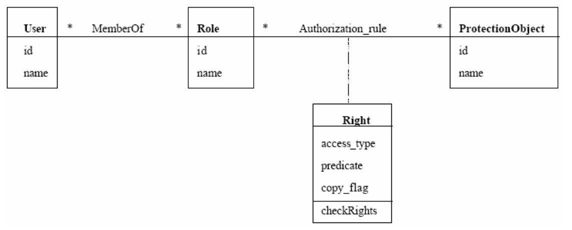

Figure 2. RBAC Pattern.

2.3. Multilevel Security Pattern

i) *Intent*: It provides a mechanism of access management in a system with several levels of security classification. ii) *Context*: It is applicable to systems that need to provide several security levels. iii) *Problem*: How to decide access in an environment with security classifications. iv) *Description*: In many systems, data integrity and confidentiality need to be guaranteed. This model would be able to be used in any architecture level and it provides a structure that allows us to have differente security levels for both subjects and objects. v) *Solution*: To represent the structure of Multilevel Security, there must be an instance of the class Subject Clasification for each subject and an instance of the class Object Classification for each object (see Figure 3). These instances are used to add levels and objects security categories to a subject. vi) *Consequences*: It facilitates the administrative work in an environment that requires the classification of subjects and objects. The multilevel security can be expensive since subjects and objects need to be classified into certain levels of sensitiveness. vii) *Known uses:* The model has been used by several military-sponsored projects and in a few commercial products, including DBMSs (Informix) and operating systems (Pitbull [23] and HP's Virtual Vault [24]). viii) *Related patterns*: The concept of roles can also be applied here.

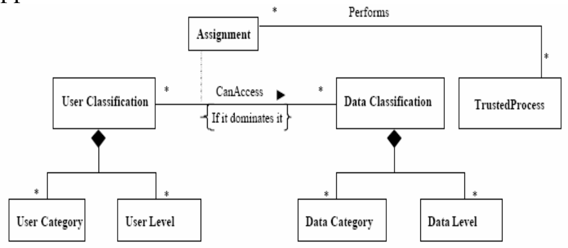

Figure 3. Multilevel Security Pattern

2.4. Reference Monitor Pattern

i) *Intent*: To make it possible that all authorizations are fulfilled when a process requires resources. ii) *Context*: A multiprocess environment making petitions by resources. iii) *Problem*: If the defined authorizations are not fulfilled, processes can execute all kind of illegal actions, for instance, any user could read any file. iv) *Description*: To define authorization rules is not enough; these rules must be imposed when a process makes a petition to a resource. There are many implementations and we need an abstract execution model. v) *Solution*: To define an abstract process that intercepts all petitions from resources and confirms them. Figure 4 shows us a class diagram in which we can see a Reference Monitor. Authorization rules indicate a collection of authorization rules organized as ACLs (access control lists) vi) *Consequences*: If all petitions are intercepted, we can assure that they fulfil the rules. The specific implementations are necessary for any kind of resource. To check each petition can mean a performance loose. vii) *Known uses*: Most modern operating systems implement this concept, e.g., Solaris 9, Windows 2000, AIX, and others. The Java Security Manager is another example. viii) *Related patterns*: This pattern is a special case of the *Checkpoint* pattern (section 2.9).

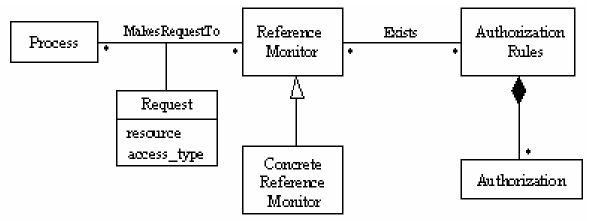

Figure 4. Reference Monitor Pattern.

2.5. Virtual Address Space Access Control Pattern

i) *Intent*: To control the access by processes to specific areas of their virtual address space (VAS) according to a set of predefined access types. ii) *Context*: Multiprogramming systems with a variety of users. Processes executing on behalf of these users must be able to share memory areas in a controlled way. Each process is executed in its own address space. iii) *Problem*: Processes must be controlled when they access memory, otherwise they could overwrite areas from other processes or gain access to private information. iv) *Description*: There is a variety of structures of virtual memory addresses space: some systems use a separate set, others an only level address space. Furthermore, VAS can be divided into users and operating system. We would like to control the access to all these kinds in a uniform way. This implies that an implementation of the solution will require specific hardware architecture. However, the solution must be independent of the hardware. v) *Solution*: To divide VAS into segments corresponding to logical units within the programs. To use descriptors to indicate access rights such as the beginning address of the accessible segment, the limit of the accessible segment and the type of allowed access (reading, writing, executing). Figure 5 shows a diagram to indicate the solution to the class. A process (*Process* class) must have a descriptor (*Descriptor* class) to access a segment in the VAS. vi) *Consequences*: This pattern provides a protection of the required segment because a process cannot access a segment without an own descriptor. If all resources are outlined in a virtual address space, the pattern can control the access to any kind of resource, including files. The solution is dependent of the hardware. In systems that use separate address spaces it is necessary to add an extra identifier to the descriptor registers to indicate the address space number. vii) *Known uses*: IBM S/38, IBM S/6000, Intel X86 [25], and Intel Pentium use some type of descriptors for memory access control. The operating systems in these machines must use this approach for memory management. viii) *Related patterns*: This pattern is a direct application of the Authorization pattern to the processes' address space.

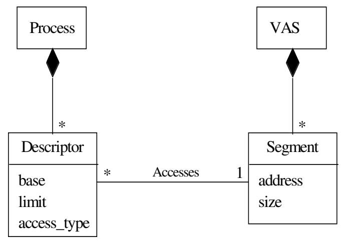

Figure 5. Virtual Address Space Access Control Pattern.

2.6. Execution Domain Pattern

i) *Intent*: Define an execution environment for processes, indicating explicitly all the resources a process can use during its execution, as well as the type of access for the resources. ii) *Context*: A process executes on behalf of a user, group, or role (a subject). A process must have access rights to use the resources defined for its subject during execution. The set of access rights given to a process define its execution domain. At times the process may also need to enter other domains to perform its work; for example, to extract a statistical value (avg, mean) from a file in another user's domain. iii) *Problem*: Restricting a process to a specific set of resources is a basic step to control malicious behaviour. Otherwise, unauthorized processes could destroy or modify information in files or databases with obvious results or could interfere with the execution of other processes. iv) *Description*: There is a need to restrict the actions of a process during its execution; otherwise it could perform illegal actions. Resources typically include memory and I/O devices, but can also be system data structures and special instructions. A process needs the flexibility to create multiple domains and to enter inner domains for specific purposes. v) *Solution*: Attach to the process a set of descriptors that represent the rights of the process. In Figure 6, class Domain represents domains and in conjunction with the Composite pattern it describes nested domains. Operation enter in class Domain lets a process enter a new domain. A domain includes a set of descriptors that define rights for resources. vi) *Consequences*: It could be applied to describe access to any type of resource if the resource is mapped to a specific memory address. The model does not restrict the implementation of domains. It has extra complexity and special hardware is needed. In capability systems the descriptors are part of the process code and are enabled during execution. vii) *Known uses*: The Plessey 250 and the IBM S/6000 running AIX [26] are good examples of the use of this pattern. The Java Virtual Machine defines restricted execution environments in a similar way [27].

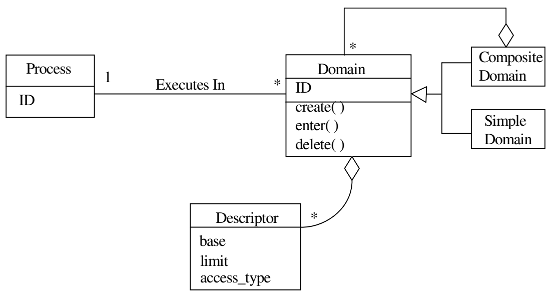

Figure 6. Execution Domain Pattern

2.7. Session Pattern

i) *Intent*: To provide us with an environment where a user's rights can be restricted and controlled. ii) *Context*: Any environment where we need to control the access to computing resources. iii) *Problem*: Depending on the context, for example, within a certain application, a user will only activate a subset of the authorizations he/she has. This will avoid that users use their rights wrongly (for instance, to accidentally delete certain files). In this way, if an attacker endangers a process, the damage potential is reduced. iv) *Description*: In many systems, global information is necessary in several points. To overcome this problem, Session objects that provide the necessary information are used. v) *Solution*: Figure shows us elements of a class diagram session. A subject can be in several sessions at the same time and it has a limited lifetime. When we start a session (for example, when registering ourselves), a user only activates a set of authorization contexts assigned to him/her, then, only the necessary rights are available within this session. The *Subject* class describes an active entity that accesses the system and asks for resources. The *AuthorizationContext* class describes a set of contexts of executions or active rights that the user has in a given interaction. vi) *Consequences*: Each session gains all privileges that are necessary to carry out the desired tasks. Thus, damage will be potentially reduced when a session is in danger because only an activated subset of authorization can be wrongly used. vii) *Known uses*: This concept appears in many computational environments, e.g. RBAC use sessions as defined by this pattern. UNIX ftp and telnet services use a Session for keeping track of requests and restricting user actions. viii) *Related patterns*: Session is an alternative to a Singleton [10] in a multi-threaded, multi-user, or distributed environment. SAP validates a user through Check Point (sections 2.8 and 2.9). It gets a Session in return if the user validation is acceptable.

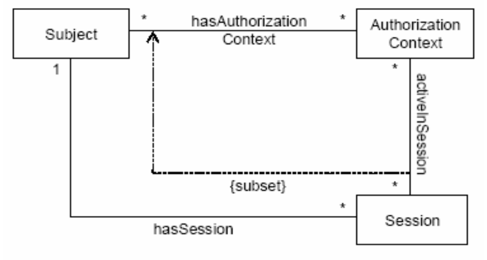

Figure 7. Session Pattern.

2.8. Single Access Point Pattern (SAP)

i) *Intent*: The SAP pattern defines a simple interface for all communications performed with entities external to the system. ii) *Context*: SAP can be applied to self-contained systems that need to communicate with external entities. iii) *Problem*: A security model is difficult to confirm when it has multiple main, back and lateral doors to come in the application. iv) *Description*: The application of a SAP pattern avoids that external entities are directly communicated with components of the system. All input traffic is carried out through a channel, where the supervision can be easily performed and this channel will collect information about occurred access petitions, their origins and authorization information. It will generate actions or transmit data to parts inside the system. v) *Solution*: SAP represents the only connection of the system with outside (see Figure 8). All incoming communication petitions are taken to the SAP instance that works as a mediator. If certain policies need to be imposed, all petitions should be sent to a *Check Point* class before they are transmitted to their addresses. vi) *Consequences*: SAP will provide a good place to capture register information as well as to carry out authorization tasks. The undesirable modification of data can be avoided with efficient checks that let us access the system. vii) *Known uses*: UNIX telnet and Windows NT login application use SAP for logging into the system. These systems also create the necessary Roles for the current Session. viii) *Related patterns*: SAP validates the user's login information through a Check Point and uses that information to initialize the user's Roles and Session. A Singleton [10] could be used for the login class especially if you only allow the user to have one login session started or only log into the system once.

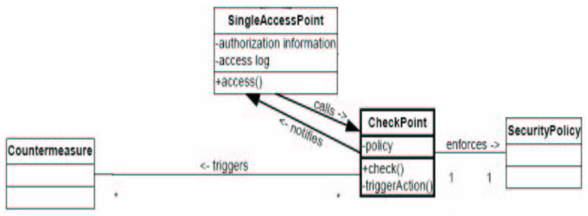

Figure 8. SAP Pattern and Check Point Pattern

2.9. Check Point Pattern

i) *Intent*: It states a structure to check the incoming petitions. If it finds violations, this pattern is in charge of taking the appropriate countermeasures. ii) *Context*: *Check Points* are applicable to any relevant security communication. iii) *Problem*: To avoid a disauthorized access, it is crucial to check who and how is interacting within a system and take measures if it is necessary. iv) *Description*: It needs to take any kind of action, if there are mistakes depending on the seriousness. v) *Solution*: A *Check Point* is a component that analyzes all petitions and messages. A SAP is predestined to be combined with a Check Point for all messages to be supervised (see Figure 8). It implements a method to check messages according to the current security policy. It gives place to actions that could be necessary to protect the system against attacks. vi) *Consequences*: Its application can benefit the system confidentiality, if the checking algorithm is correct. Undesirable modifications can be filtered if the checking algorithm is able to detect those attacks. Complex checking routines can make both the system and the message interchange work slower. vii) *Known uses*: The login process for an ftp server uses Check Point. viii) *Related patterns*: Single Access Point is used to insure that *Check Point* gets initialized correctly and that none of the security checks are skipped. Roles are often used for Check Point's security checks and could be loaded by *Check Point*. *Check Point* usually configures a *Session* and stores the necessary security information in it. It can also interact with the *Session* to get the user's Role during the authorization process.

3. Comparative Framework

In this section, we will put forward a comparison based on certain criteria that we consider important for security with the purpose of distinguishing all properties and characteristics of all previous patterns as well as showing a general vision of the subject. There are some comparisons [13] of patterns with certain criteria or security principles [28]. Some of these defined criteria are based on the works of Babar [29] and Firesmith [30], in which they select the most commonly used attributes and security properties in the security dominion. The security properties considered to make our comparison are the following: *Authentication*: It must be validated the identity of customers to frustrate any unauthorized access. *Authorization*: This attribute defines the access privileges of entities to different resources and services of a system. *Integrity*: To guarantee that data and communications will not be compromised by active attacks. *Confidentiality*: The guarantee that information is not accessed by unauthorized parts. *Attacker detection*: To be able to detect and register access or modification intents in the system coming from unauthorized users. *No-Repudiation*: It prevents that certain participant in certain interaction can deny to have participated in it. *Auditability*: To keep a log of user's or other system's interaction with a system and it helps detect potential attacks. *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. *Reliability*: It assures the system operations due to failures or configuration mistakes. Besides, it assures the system availability even when the system is being attacked. *Error management*: A system must provide a robust error management mechanism.

Also, we consider some criteria to evaluate patterns as they are: *Performance*: It indicates the impact of the pattern on the functioning of a system. *Implementation cost*: Costs accompanying the pattern use. *Security degree*: It indicates the security level that the pattern has for the function it fulfils, that is, the more security properties the pattern covers, the more security degree will have.

Many patterns fulfil security properties without constraints; others are only fulfilled according to certain conditions, as it can be seen in table 1. The majority of patterns are based on guaranteeing access control, supplying confidentiality and in some cases, also integrity and reliability, but they do not take into account properties as important as error management, flexibility or maintenance, etc. In table 2, we can see the use of these patterns in the software development to allow us to increase or to reduce the performance of the global application. For example, "—^{a}" means that the pattern can reduce performance because there are many users in the system and it is complex to manage and implement the pattern with many users. There are patterns with a high degree of security (see Table 3) but they are complex patterns. Then, if we want to have a system with a high degree of security, they will be also more complex systems, affecting their performance.

Table 1. Comparative table for the security criteria

	Authentication	Authorization	Integrity	Confidentiality	Attackers detection	No-Repudiation	Auditability	Maintainability	Availability	Reliability	Error management
Authoriz.		0		0	1			0	0		
RBAC		0		0	1			0	0		
Multilevel		0	5	6	1				0	0	
Reference Monitor		0	7	7	1				0	7	
Virtual Address		0	7	7	1				0		
Execution Domain		0	7	7	1				0	4	
SAP	0	0	2	0	0	0	0		0	0	0
Check Point	0	0	2	2	0	0			0	0	0
Session		0	0	0	1			3	0	4	

0. Always fulfilled. 1. Only detection. 2. Efficient check algorithm. 3. First step development. 4. Subset of the authorizations activated. 5. Biba model. 6. Bell LaPadula model. 7. To process level.

Developers (not security experts) can find many security patterns but it is very difficult to determine which pattern is better to be used or which pattern guarantees certain degree of security. For this reason, we find a lack of a method or a flexible model of security architectures that guarantees security of the system in many aspects and that guides developers in the right way for the implementation of security into their systems, according to the specific requirements of them [31].

Table 2. Comparative table of the performance

	Performance		
Authorization	—[a]	┼[e]	Virtual Address
RBAC	┼[c]	—[f]	Execution Domain
Multilevel	—[d]	—[b]	SAP
Reference Monitor	—	—[b]	Check Point
Session	┼c		

┼: Increase. — : Reduce. Conditions: a) Many users. b) Complex checks. c) Efficient implementation. d) Evaluation access rights. e) If it uses Reference Monitor. f) Domains management

Table 3. Comparative table for the evaluation criteria

	Implem. Cost	Security degree
Authorization	L	M
RBAC	M	M
Multilevel	H	H
Reference Monitor	H	H
Virtual Address	L	M
Execution Domain	M	H
SAP	H	H
Check Point	H	H
Session	H	H

L: Low M: Medium H: High

4. Conclusions and Future Work

Patterns are a promising proposal towards security, useful to build and evaluate systems. Security patterns help us keep in mind non-functional security requirements at the beginning of the design. In the critical applications of security, it is extremely important to avoid mistakes, since we must guarantee the security of such applications and provide all operations and interactions that are performed in the application with a high level of security. Therefore, the use of security patterns is important to develop a secure system.

A software architecture constructed in this way is more reusable and extensible than an architecture defined directly from the requirements or where patterns are applied later. It is clear that combinations of patterns are extensible because of the possibility of replacing a pattern with another concrete realization of the same pattern. They are reusable because of the possibility of replacing several of the used patterns to fit the requirements of a new application.

There are many patterns with different purposes, reason why developers must combine many patterns to establish a high degree of security within the system but they have the problem of choosing which pattern must be used and deciding which pattern will better adapt to the system security requirements.

Our future work will be studying the different security architectures existing in the systems design together with defining a method to specify flexible security architectures that can be easily adapted to systems with very different security requirements as well as guarantee security using security patterns.

5. 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).

6. References

[1] E. B. Fernandez, "An Overview of Internet Security", presented at World's Internet and Electronic Cities Conference (WIECC 2001), Kish Island, Iran, 2001.

[2] A. Toval, J. Nicolás, B. Moros, and F. García, "Requirements Reuse for Improving Information Systems Security: A Practitioner's Approach", *Requirements Engineering Journal*, vol. 6, pp. 205-219, 2001.

[3] D. Dennings, "Reflections on cyberweapons controls", *Computer Security*, vol. 16, pp. 43-53, 2000.

[4] A. Ghosh, C. Howell, and J. Whittaker, "Building software securely from the ground up", *IEEE Software*, vol. 19, pp. 14-17, 2002.

[5] E. Ferrari and B. Thuraisingham, "Secure Database Systems in: M. Piattini, O. Díaz," in *Advanced Databases: Technology Design*. Artech House, 2000

[6] P. Devanbu and S. Stubblebine, "Software engineering for security: a roadmap in: A. Finkelstein", *The Future of Software Engineering, ACM Press*, pp. 227-239, 2000.

[7] A. Boulanger, "Catapults and grappling hooks: The tools and techniques of Information warfare", *IBM Sys.*, vol. 37, pp. 106-114, 1998.

[8] ISACF, "Information Security Governance," in *Guidance for Boards of Directors and Executive Management*. USA: Information Systems Audit and Control Foundation, 2001

[9] E. B. Fernandez, "Metadata and authorization patterns", Departament of Computer Science and Eng., Florida Atlantic University TR-CSE-00-16, May 2000 2000.

[10] D. G. Firesmith, "Engineering Security Requirements", *Journal of Object Technology*, vol. 2, pp. 53-68, 2003.

[11] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems", presented at Proceedings of the IEEE, 1975.

[12] C. Alexander, S. Ishikawa, and M. Silverstein, *A pattern language: towns, builings, construction*. New York: Oxford University Press, 1977.

[13] R. Wassermann, "Using Security Patterns to Model and Analyze Security Requirements", 032.04/E, 9th March 2004.

[14] AGCS, "AG Communication System. Template Pattern", 1996.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1994.

[16] E. B. Fernandez, "Patterns for Operating Systems Access Control", presented at 9th Conference on Pattern Languages of Programs, PLoP 2002, Allerton Park, Illinois, USA, 2002.

[17] E. B. Fernandez and R. Pan, "A pattern language for security models", presented at 8th Conference on Pattern Languages of Programs, PLoP 2001, Allerton Park, Illinois, USA, 2001.

[18] J. Yoder and J. Barcalow, "Architectural Patterns for Enabling Application Security", presented at 4th Conference on Patterns Language of Programming, PLop 1997, Monticello, Illinois, USA, 1997.

[19] E. B. Fernandez and J. C. Sinibaldi, "More patterns for operating systems access control", presented at 8th European Conference on Pattern Languages of Programs (EuroPlop'2003), Irsee, Germany, 2003.

[20] E. B. Fernandez, M. L. Petrie, N. Seliya, and A. Herzberg, "A Pattern Language for Firewalls", presented at 10th Conference on Pattern Languages of Programs (PLoP'2003), Allterton Park, Monticello, Illinois, 2003.

[21] E. B. Fernandez, "Two patterns for web services security", presented at The 2004 International Symposium on Web Services and Applications, Las Vegas, Nevada, USA, 2004.

[22] S. Lehtoren and J. Pärssinen, "Pattern Language for Cryptographic Key Management", presented at 7th European Conference on Pattern Languages of Programs (EuroPlop'2002), Irsee, Germany, 2002.

[23] Argus Systems Group, "Trusted OS Security: Principles and practice," 2001 http://www.argus-systems.com/product/white_paper/pitbull/oss/2.shtml.

[24] HP, "Hewlett Packard Corp.,Virtual Vault," http://www.hp.com/security/products/virtualvault.

[25] R. E. Childs Jr., J. Crawford, D. L. House, and R. N. Noyce, "A processor family for personal computers", presented at Proceedings of the IEEE, 1984.

[26] N. A. Camillone, D. H. Steves, and K. C. Witte, "AIX operating system: A trustworthy computing system," in *IBM RISC System/6000 Technology*, 1990, pp. 168-172

[27] S. Oaks, *Java Security*, 2nd ed: O'Reilly Media, Inc., 2001.

[28] J. Viega and G. McGraw, *Building Secure Software - How to Avoid Security Problems the Right Way.*, 1st ed: Addison-Wesley, 2002.

[29] M. A. Babar, X. Wang, and I. Gorton, "Supporting Security Sensitive Architecture Design", presented at QoSA-SOQUA 2005, 2005.

[30] D. G. Firesmith, "Specifying Reusable Security Requirements", *Journal of Object Technology*, vol. 3, pp. 61-75, 2004.

[31] C. Gutiérrez, E. Fernández-Medina, and M. Piattini, "Towards a Process for Web Services Security", presented at WOSIS'05, Miami, Florida, USA, 2005.