A system of security patterns

Koen Yskout, Thomas Heyman,
Riccardo Scandariato, Wouter Joosen

Report CW469, December 2006

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A system of security patterns

Koen Yskout, Thomas Heyman,
Riccardo Scandariato, Wouter Joosen

Report CW469, December 2006

Department of Computer Science, K.U.Leuven

Abstract

For the past 5 years, MITRE has been tracking the types of errors that lead to publicly reported vulnerabilities. The results show that the number of vulnerabilities is not getting any smaller. On the contrary, they observed a 55% increase over the past two years. Furthermore, over 4500 vulnerabilites that were tracked in 2005, only 25% were due to infrastructural software, like the OS. More than 75% of all vulnerabilities were actually due to faulty application software. This suggest an inadequate adoption of proven secure software engineering techniques, as also recognized by the research community.

Designing secure software is a hard endeavor, requiring unique skills, which one cannot expect from an average development team. It would be beneficial if a set of easily usable bricks would be made available to a larger base of designers upon which sound and secure software can be built, without the need for them to fully grasp all the underpinnings of security engineering. A viable solution is represented by security patterns, which provide domain-independent, time-tested security knowledge and expertise. Furthermore, they preserve this knowledge in a reusable format, so that other (non-expert) designers may benefit.

This technical report presents an extensive inventory in which security patterns are collected to form a coherent system. Based on a broad survey of existing literature, we applied a reduction process in order to both simplify the patterns landscape according to several dimensions and remove heterogeneity. For the resulting set of core security patterns, we provide a uniform description and, most importantly, enhance the patterns by means of meta-information enabling and facilitating both the search for and the selection of the right pattern for the job.

A System of Security Patterns

Koen Yskout, Thomas Heyman, Riccardo Scandariato, Wouter Joosen

DistriNet Research Group
Katholieke Universiteit Leuven
Belgium
first.last@cs.kuleuven.be

Technical report

Date: January 23, 2007

Version: 1.0 release candidate 1

Document status: draft

Document History

11/10/06	0.1	Thomas Heyman	Initial version.
13/01/07	1.0	Thomas Heyman	Integrated feedback.

Executive Summary

Given the level of interdependency between today's society and computerized systems, the present level of security achieved by software is unsatisfactory. Further, the situation is not improving. As an estimate of this trend, MITRE has been tracking the types of errors that lead to publicly reported vulnerabilities for the past 5 years. The results show that the number of vulnerabilities is not decreasing— on the contrary, they observed a 55% increase over the past two years. If one analyzes the root causes of that increasing rate, MITRE reports, quite interestingly, that of over 4500 vulnerabilites that were tracked in 2005, only 25% were due to infrastructural software, like the OS. More than 75% of all vulnerabilities were actually due to faulty application software.

High quality security building blocks, such as authentication technologies, encryption libraries, and policy enforcing engines are already available to developers. Although those building blocks are essential ingredients, they are not sufficient on their own to build secure distributed software, as many vulnerabilities in software systems are not directly related to security components. Rather, the impressive number of application-level vulnerabilities suggests an inadequate adoption of proved secure software engineering techniques, e.g., to integrate the above-mentioned security components in developed applications.

This work starts from the observation that there is a lack of guidance for developers during the design phase. As it is notoriously known in the research community, plugging in security as an afterthought is a very unsuccessful strategy. A better practice suggests to develop software systems in a secure way, i.e., with security in mind throughout the development cycle, especially during the design phases. However, designing secure software is a hard endeavor, requiring unique skills, which one cannot expect from an average development team. It would be beneficial if a set of easily usable design bricks would be made available to a larger base of designers upon which sound and secure software can be built, without the need to fully grasp all the underpinnings of security engineering.

A viable solution to ease up the hard endeavor of designing security-savvy applications is represented by security patterns, which provide domain-independent, time-tested security knowledge and expertise. Furthermore, they preserve this knowledge in a reusable format, so that other (non-expert) designers may benefit. This technical report presents an extensive inventory in which security patterns are collected to form an improved and coherent system.

The first contribution is in the area of rationalization of existing work. This was a necessary step to *enable* the simplification of the pattern selection process. Based on a broad survey of existing literature of about eighty security patterns, we applied a reduction process in order to both simplify the patterns landscape and remove heterogeneity. Indeed, not all security patterns in literature have the same level of quality, e.g., compared to software design patterns. In particular, many patterns are too generic and lack a well-defined, solution-oriented description. Furthermore, existing security patterns are meant to be applied at different levels of abstraction, e.g., architectural design rather than detailed design. In this respect, by simplifying the landscape (by removing overlapping or lower quality patterns) and by bringing in homogeneity (by classifying) we enabled a simplification of the pattern selection process.

Indeed, the second major contribution is a set of support meta-information that *guide designers* in easily selecting the right patterns for the job. In particular, each pattern is linked to the security objective it fulfills (e.g., confidentiality, controlled access, or availability) so that the rationale that led to the introduction of a given pattern in the design is traceable. Furthermore, each pattern comes with a set of annotations describing the side effects of its adoption on other software qualities. These annotations are highly beneficial to drive informed trade-off decisions among alternatives. Finally, we made explicit the relationships among the pattern in the inventory. Indeed, security patterns are not isolated entities. They may interact with one another, both advantageously or disadvantageously. In order to select the best pattern for a

given design challenge, these inter-pattern relationships are made explicit to the user.

Contents

1 Introducing security patterns 6
- 1.1 Introduction . . . 6
- 1.2 Related work . . . 7

2 Utilities for secure design with patterns 9
- 2.1 Security patterns inventory . . . 9
  - 2.1.1 Development phase classification . . . 10
  - 2.1.2 Security objective classification . . . 11
  - 2.1.3 Quality trade-off labels . . . 13
  - 2.1.4 Security pattern template . . . 13
- 2.2 A system of patterns . . . 16
  - 2.2.1 Inter-pattern relationships . . . 16
- 2.3 Conclusion . . . 19

3 An overview of the inventory 20
- 3.1 Classification by development phase . . . 20
- 3.2 Classification by security objective . . . 22
- 3.3 Inter-pattern relationships . . . 24

4 The inventory of security patterns 26
- 4.1 Application Firewall . . . 27
- 4.2 Audit Interceptor . . . 28
- 4.3 Authentication Enforcer . . . 29
- 4.4 Authorization Enforcer . . . 30
- 4.5 Checkpointed System . . . 31
- 4.6 Comparator Checked Fault Tolerant System . . . 32
- 4.7 Container Managed Security . . . 33
- 4.8 Controlled Object Factory . . . 34
- 4.9 Controlled Object Monitor . . . 35
- 4.10 Controlled Process Creator . . . 36
- 4.11 Credential Tokenizer . . . 37
- 4.12 Demilitarized Zone . . . 38
- 4.13 Firewall . . . 39
- 4.14 Full View with Errors . . . 40
- 4.15 Input Guard . . . 41
- 4.16 Limited View . . . 42
- 4.17 Load Balancer . . . 43
- 4.18 Obfuscated Transfer Object . . . 44
- 4.19 Output Guard . . . 45
- 4.20 Replicated System . . . 46
- 4.21 Reverse Proxy . . . 47

4.22 Secure Access Layer 48
4.23 Secure Logger 49
4.24 Secure Message Router 50
4.25 Secure Pipe 51
4.26 Secure Service Façade 52
4.27 Secure Session Object 53
4.28 Security Association 54
4.29 Security Context 55
4.30 Server Sandbox 56
4.31 Session 57
4.32 Session Failover 58
4.33 Session Timeout 59
4.34 Single Access Point 60
4.35 Subject Descriptor 61

List of Figures

2.1 Classification according to development phase 10
2.2 Classification according to security objective 11
2.3 Comparing tactics and scenario's with quality trade-off labels. 14
2.4 Security Pattern Template. 15

Chapter 1

Introducing security patterns

1.1 Introduction

Plugging in security as an afterthought is more the norm than the exception in today's software practice and, notoriously, this is a very unsuccessful strategy. A better practice suggests to develop software systems in a secure way, i.e. with security in mind throughout the development cycle, especially during the design phases.

Still, designing secure software is a hard endeavor, since this software has to be designed and implemented with failure in mind. That is, developers have to explore and exercise the failure conditions of systems they are building as much as they devote time to implement the functional part of the specification. This requires a particular (counter-intuitive) mindset, high commitment and unique skills, which one cannot expect from an average development team. It would be beneficial if a set of easily usable bricks would be made available to a larger base of designers upon which sound and secure software can be built, without the need for them to fully grasp all the underpinnings of security engineering.

In the software engineering discipline, software design patterns represent a well-known technique to provide domain-independent, time-tested knowledge and expertise. Furthermore, they preserve this knowledge in a reusable format, so that other (non-expert) designers may benefit.

According to us, security patterns[1] should play a key role during the design stage of the secure software process. However, in order for a non-expert to know which pattern to apply where, he or she must first have a thorough overview of the security pattern landscape. This can be trickier than it seems. First, the large amount of security patterns and their heterogeneity make the selection of "the right pattern for the job" a daunting task. Second, inter-pattern relationships complicate this selection even further, as not all patterns complement each other constructively. Indeed, to assist the selection process, a *system of security patterns* is needed. We use the term system both in the meaning of "a manner of classifying, symbolizing, or schematizing" and "a regularly interacting or interdependent group of items forming a unified whole" (definitions from Merriam-Webster).

In order to realize the first definition of system, the body of existing security patterns shows some shortcomings. First, not all security patterns have the same level of quality, e.g. compared to software design patterns. In particular, many patterns are too generic and lack a well-defined, solution-oriented description. Second, existing patterns are meant to be applied at different levels of abstraction, e.g., architectural design rather than detailed design. In this respect, standardizing the description of patterns through a uniform template and making their classification more explicit greatly simplifies the selection process.

Concerning the latter definition of system, security patterns are not isolated entities. They may interact with one another, both advantageously or disadvantageously. In order to select

[1] We use pattern and security pattern interchangeably hereafter. Whenever a clear distinction is needed, we use software design patterns to identify non-security patterns.

the best pattern for a given design challenge, these inter-pattern relationships must be made explicit to the user.

As its main contribution, this paper presents an inventory in which security patterns are collected to form a system of patterns. Based on an extensive survey of available literature from the security pattern community, we have gathered an initial set of roughly eighty security pattern candidates. The candidates were screened according to several dimensions: (1) complexity-wise, we simplified the pattern landscape by removing the overlaps between pattern variants coming from different sources, (2) quality-wise, we removed some candidates that were of a less appropriate level of detail, and (3) taxonomy-wise, we removed some patterns that were not at the right level of abstraction, i.e., they were neither at architectural nor at detailed design level. This reduced the amount of core security patterns to a final thirty five.

For this resulting set, a uniform description is provided according to a template we contribute and, most importantly, the patterns are enhanced by means of meta-information, which enables and facilitates both the search and the selection processes. Namely, the patterns are annotated to include information about their role in the process (does the pattern pertain to the architectural or design phase), their intent in fulfilling a security objective (does the pattern provide integrity or confidentiality), and their inter-relationships (which patterns conflict with the selected one, and which could be beneficial).

The effectiveness of the inventory in designing secure software is showcased by means of a proof-of-concept application that we designed. The application models a simplified distributed calendaring system, with the usual set of security requirements (e.g., confidentiality and accountability).

Finally, note that this work is part of a larger research project investigating a pattern-based software building method to improve the design quality of secure software. In that context we define a support methodology by which designers are guided through a sequence of standardized and repeatable steps in order to select the patterns of interest and to integrate them in design artifacts (while dealing with trade-offs between security and other qualities). However, the description of that methodology is not in the scope of this paper.

The rest of this report is organized as follows. In the remainder of this Chapter, Section 1.2 provides an overview of related literature. The tools for using the inventory are presented in Chapter 2. Next, an overview of the inventory, facilitating the use of the full inventory, will be given in Chapter 3. Finally, the full inventory of security patterns is given in Chapter 4.

1.2 Related work

After the seminal paper by Yoder and Barcalow [YB97], several works on security patterns have been published. Blakley and Heath collect an inventory of about fifteen security patterns [BHm04]. In that work, the authors also provide a (limited) description of a system of patterns that distinguishes between availability and security concerns. They also discuss what makes a good pattern and propose a minimal definition of what is a pattern. The work by Steel et al. contains a very extensive collection of security patterns, specifically meant for the Java Enterprise platform [SNL05]. We acknowledge the level of granularity used in that work as the most suitable for our purposes. Indeed, the majority of members of our final inventory come from that source. A third systematic collection of security patterns is presented by Schumacher et al. [SFBH+06], whose work contains about fifty security patterns. The work approaches the use of patterns at several levels of abstraction, not limited to architectural and design patterns, but also including patterns for, e.g., performing risk assessment and mitigation. In further work Schumacher et al. discuss the role of security patterns in secure software engineering and present a system of patterns for network security [SR02]. Finally, an analytical body of over thirty patterns can be found in earlier work by Kienzle et al. [KETEH].

Security patterns have also been quite popular at the latest editions of the PLoP (Pattern Languages of Programs) conference, the main venue for the software patterns research community [YB97, Sor02, Sch03, DGFRLP04, Sar03, Som03].

The first observation is that consistency in the level of detail between these works is still largely missing and there is *no standard format for describing security patterns.* Although some sources are more succinct in the details they provide for each security pattern, others provide an overabundance. While the Gang-of-Four design patterns [GHJV94] are the de-facto standard template for describing design patterns, there is no such agreed upon format in the security pattern community. Examples of security pattern templates can be found in [KETEH02, SNL05]. For instance, consider the description of the Check Point and the Authorization Enforcer patterns. The former contains the name, aliases, a motivation, the problem statement and conflicting forces, the solution, an example, consequences, related patterns and known uses. The latter contains all of these, but elaborates on the solution by providing structure, participants and responsibilities, strategies for implementation with code snippets and sequence diagrams, security factors and risks, and concludes with a brief reality check. We recognize that the second option is preferable.

A related comment is that the *level of abstraction of patterns varies.* While some patterns pertain to the same level of abstraction as the design patterns of the Gang-of-Four [GHJV94], others are either too abstract and programming agnostic (e.g. Asset Valuation) or, conversely, too low-level and implementation-bound (e.g. File Authorization). In this work we define (by examples) what is expected to be the right level of abstraction, as far as design stages are concerned. Note that questioning and surveying the quality of existing patterns have been carried out in the past already, albeit in a limited way. For instance, Halkindis et al. performed a qualitative analysis of patterns in the Blakley and Heath inventory [HCS04]. The evaluation criterion the authors used was the adherence to the ten security principles by Viega and McGraw [VM01]. In a previous study, Konrad et al. applied the same evaluation criteria to the Yoder and Barcalow inventory[KCC03].

The final observation is that *there is no agreed-upon "system" of security patterns.* Some initial work exists. For instance, Mazhelis and Naumenko relate patterns to security requirements [MN06]. The same criterion is used by Rosado et al., along with a distinction between architectural and design patterns [RGFMP06]. Konrad et al. applied different classification criteria, such as the nature of the patterns (creational, structural or behavioral, as defined in GoF [GHJV94]) and the abstraction level (network, host, application) of the pattern [KCC03]. However, inter-pattern relationships are neglected in the above-mentioned works.

Chapter 2

Utilities for secure design with patterns

2.1 Security patterns inventory

Significant overlap between existing security patterns exists. In this section, we present an inventory of patterns that, firstly, removes overlap between identical patterns and, secondly, groups intimately related patterns. Each group is represented by one exemplary pattern. Obviously, the process of comparing, filtering and grouping can be cumbersome and is subject to arbitrary choices from time to time. Nevertheless, at the end of the reduction process the list was narrowed down to fifty patterns, of which only thirty-five are "core" security patterns.

The core patterns are those that can be used by software architects and designers to add security during the design of an application. The other (non-core) security patterns are either too high-level (i.e. they are too abstract to integrate in the design of an application) or too low-level (i.e. they are too implementation-oriented). Hence, they are categorized as follows:

Security objectives are high-level security goals such as 'confidentiality', 'controlled access', 'identification', and 'availability'. Since the patterns in this category are mainly rewordings of these security goals, we put these patterns aside in favor of a more systematic (and widely accepted) classification of security objectives, as described in Section 2.1.2.

Some examples of patterns found in the literature that belong to this category are the Known Partners (which is basically the same as 'identification') and Secure Communication (corresponding to 'secure data transmission') patterns.

Building blocks are patterns that do not solve a particular security problem by themselves, but are used as primitives by other patterns or security solutions. As for the security objectives, some broader and more systematic concepts (inspired by the Common Criteria security functional components [CCp06]) can be used, e.g., 'transport level encryption', 'message-level encryption', and 'access control models'. The full list is given in Section 3.2.

Example patterns for the building block category are the File Authorization and Multilevel Security pattern.

From now on, we will focus only on the thirty-five core patterns. In particular, the instruments to make the pattern collection easier to use are specifically intended for this group. In Section 2.1.1, a categorization according to development process phases is presented. Next, the links between each pattern and the security objective(s) it tries to achieve is explicated in Section 2.1.2. Finally a unified template for the description of a security pattern is presented in Section 2.1.4.

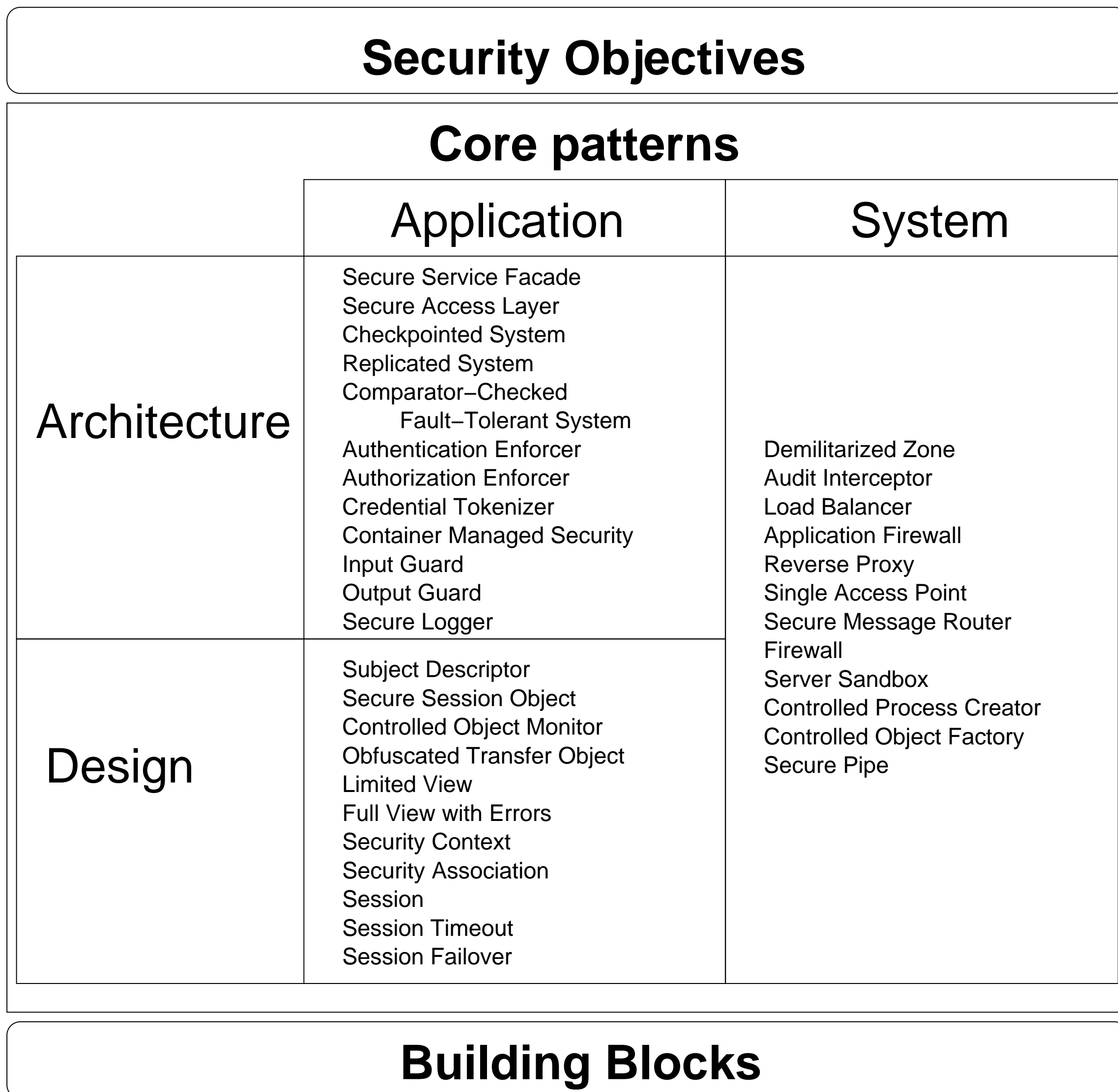

Figure 2.1: Classification according to development phase

2.1.1 Development phase classification

Two kinds of core patterns can be distinguished. The first set of patterns only affects the environment (infrastructure, middleware) in which the application will eventually be deployed. In contrast, the other set is used in the design of the software application itself. Furthermore, these application patterns differ in the development phase in which they are best applied: at the higher level architectural phase or on the more lower level (detailed) design phase. Given these distinctions, the following categories naturally arise as shown in the center part of Figure 2.1.

Application architecture. A pattern is an application architectural pattern if its introduction has system-wide implications. That is, the pattern introduces new components in the application, modifies existing components and/or inserts dependencies within an extensive part of the application. For example, it could introduce an abstraction layer or a security component. Examples of patterns for this category are the SECURE ACCESS LAYER and the AUTHORIZATION ENFORCER.

Application design. A pattern is an application design pattern if its introduction only has local implications. That is, the adoption of the pattern affects only a small subset of elements, which are part of the detailed design of the application. For example, a pattern

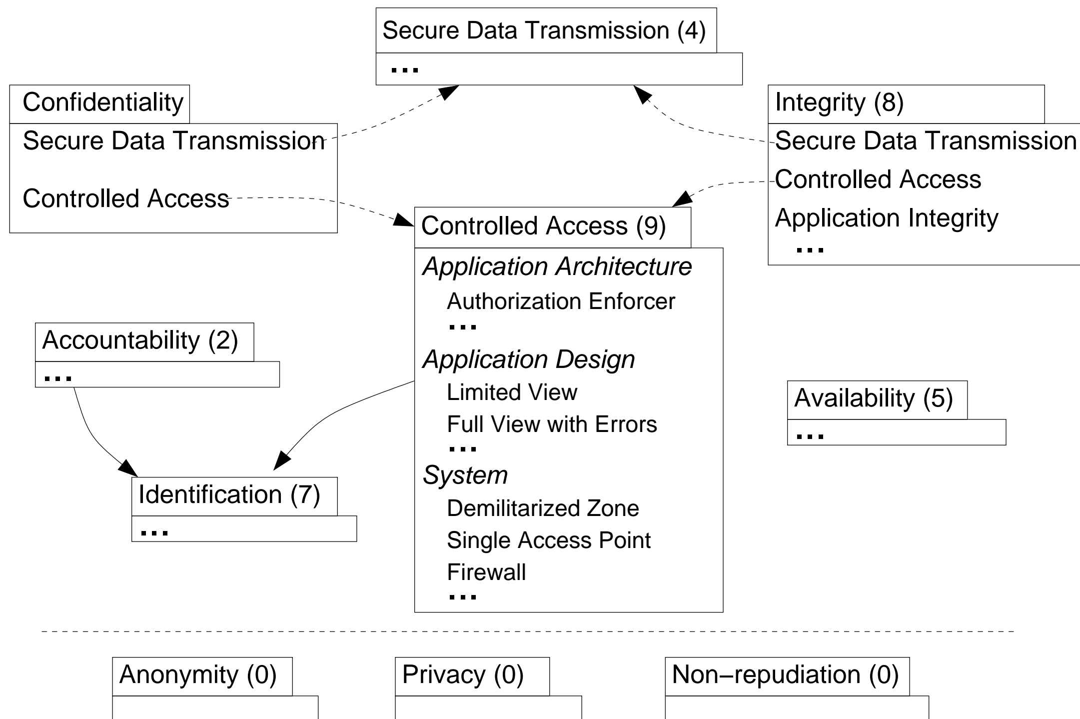

Figure 2.2: Classification according to security objective

can introduce some form of encapsulation of security data. A few illustrations for this category are the SESSION pattern and the OBFUSCATED TRANSFER OBJECT.

System. A pattern is a system pattern[1] if its introduction has its main effects on the environment in which the application will be deployed, ideally independent of the application (which can be considered as a black box). Sometimes however, small changes to the application cannot be precluded. Two typical patterns for this category are the FIREWALL and the SECURE PIPE.

2.1.2 Security objective classification

No single pattern can solve all security problems at once. Each pattern contributes to the achievement of one specific security objective. Sometimes, a pattern might contribute to more than one objective, but this is the exception rather than the rule. We made explicit the association between a pattern and the security objectives it tries to achieve, this facilitates the selection of the right pattern when it comes to implementing the security requirements (which, in turn, are typically related to the security objectives).

The set of security objectives we use is shown in Figure 2.2. In the picture, the relationships among objectives and sub-objectives are represented by arrows: a solid arrow represents a dependency between two objectives (e.g., there is no accountability without identification), while a dashed arrow means optionality (e.g., integrity may or may not require secure data transmission). Optionality is context-specific: some cases may require both secure data transmission and controlled access in order to implement confidentiality, while in other cases controlled access suffices. Also note that every objective can contain patterns for the different phases as mentioned in Section 2.1.1. To avoid clutter in the figure, this is only shown for the controlled access objective, while the other objectives only contain the number of included patterns.

[1] A distinction between a 'system architecture' and 'system design' category turned out to be arbitrary and hard to sustain.

Not surprisingly, the complete set of objectives is an extension of the classic CIAA set (i.e., confidentiality, integrity, availability, and accountability). This set is well-known and complements the software construction process, as the security requirements are typically related to the objectives. Another interesting set of objectives could be derived from the eleven classes of functional requirements in the Common Criteria (CC) [CCp06]. This is subject to future work.

The security objectives of Figure 2.2 are:

- **Confidentiality** means that sensitive data can only be read by the intended audience. To achieve this main goal, two sub-goals have to be considered: *secure data transmission* for protecting data transmitted over a channel, and *controlled access* for restricting access to stored data. Examples of confidentiality patterns are given when the sub-goals are discussed.

- **Integrity** refers to the fact that resources have not been tampered with. Similar to confidentiality, multiple aspects are considered. *Application integrity* makes sure that the application is always in a consistent state and produces the expected results. Whenever transmitting data that should be resistant to modifications, *secure data transmission* solutions can be utilised. Finally, for access to stored data, *controlled access* is needed again to prevent unauthorized modifications. An example of an application integrity pattern is the INPUT GUARD (at the application architecture level).

- **Availability** is the concern of keeping a system available and responsive for its users. One of the system availability patterns is the LOAD BALANCER.

- **Accountability** enables the tracing of important (or all) actions performed on the system back to a particular user, usually by means of logging. This requires that some form of *identification* has been performed before. An architectural example for accountability is the SECURE LOGGER.

As mentioned, some of the above objectives depend (directly or indirectly) on other common sub-objectives. The latter are never goals on their own, rather they are used in the achievement of other security objectives. Sub-objectives are:

- **Secure data transmission** intends to protect data in transit. This protection can take various forms: guarantee the confidentiality of the message, preventing (undetected) modification or deletion of the message, assuring the identity of the sender or receiver of the message, or any combination hereof. An illustration is the SECURITY ASSOCIATION at the design level.

- **Controlled Access** is the restriction of access to operations on resources to authorized entities (users) only. Of course, this requires that the entity trying to access the resource has been identified. A well-known architectural-level pattern for this objective is the AUTHORIZATION ENFORCER.

- **Identification** happens when a user claims to have a particular identity. The process of verifying this claim is commonly called the authentication of the user. At the architectural level, this could for instance be accomplished by the AUTHENTICATION ENFORCER pattern.

Other security objectives, but for which to the best of our knowledge no patterns have appeared in the literature, are:

- **Non-repudiation**, the undeniability of having performed an action, e.g. having sent or received a particular message.

- **Anonymity**, the state of being unidentifiable within a group of entities.
- **Privacy**, the right to be informed about the flow of information about oneself, and to control this flow.

It is not clear whether patterns for these three objectives are missing due to a gap in the pattern landscape, because there is no need for patterns for these objectives, or even because these branches are not yet mature enough or widely used in order for patterns to be discovered.

2.1.3 Quality trade-off labels

Security patterns contribute, in general, to the achievement of one main security objective. They do have an influence on other security and non-security qualities[2] as well, however. The impact of a pattern on these other qualities is expressed through quality trade-off labels, or simply "labels".

Not only security objectives are considered for quality trade-offs. Next to the security objectives described in Section 2.1.2, other non-security qualities include: dependability, portability, maintainability, performance and usability (as defined in ISO 9126); manageability and auditability (from the Common Criteria); and cost (as the main example of a business quality). Each pattern can have a beneficial or detrimental impact on these qualities. To give an example: while the main goal of a FIREWALL is to control access, it can also improve auditability. On the other hand, it has a negative impact on both performance and dependability.

It is important to note that we consider the impact of the pattern on the quality compared to the system with *the same (security) functionality, but not implemented in the pattern under consideration*. For example, an AUTHORIZATION ENFORCER improves maintainability compared to the same system that implements authorization, however without using an AUTHORIZATION ENFORCER.

The trade-off labels are related to the tactics and scenarios from [BCK03]. In that work, system qualities such as availability, performance and security can be provided by a system architecture by incorporating so-called "tactics" in the architecture. To incorporate security into an architecture, the architect may implement *tactics* such as the authentication of users, maintaining integrity, limiting exposure and limiting access. Tactics for availability include voting, rollbacks, heartbeat messages and similar methods. These tactics then influence *scenario's* (which are ways to express quality requirements of an architecture). In this context, patterns are a collection of different tactics to improve certain qualities.

The left hand side of Figure 2.3 illustrates how availability tactics are methods to increase a certain quality of a system, and that architectural patterns group certain tactics to achieve a certain quality goal in the architecture. When compared to trade-off labels (on the right hand side of Figure 2.3), the quality of the main tactic incorporated in a pattern corresponds to the security objective. Secondary tactics correspond to positive labels, i.e. a pattern that improves confidentiality and includes a performance tactic is assigned a positive performance label. The main difference between both approaches is that we focus on security as the main quality.

2.1.4 Security pattern template

As described earlier, the method and the level of detail in which security patterns are described vary largely from pattern to pattern. This facilitates neither selecting nor applying a security pattern. For instance, information needed to understand the full consequences of selecting a particular pattern may be missing, or the instructions on how to apply a pattern can be vague. Therefore, it is beneficial to have a unified template.

[2] We refer to "qualities" as those defined in [BCK03].

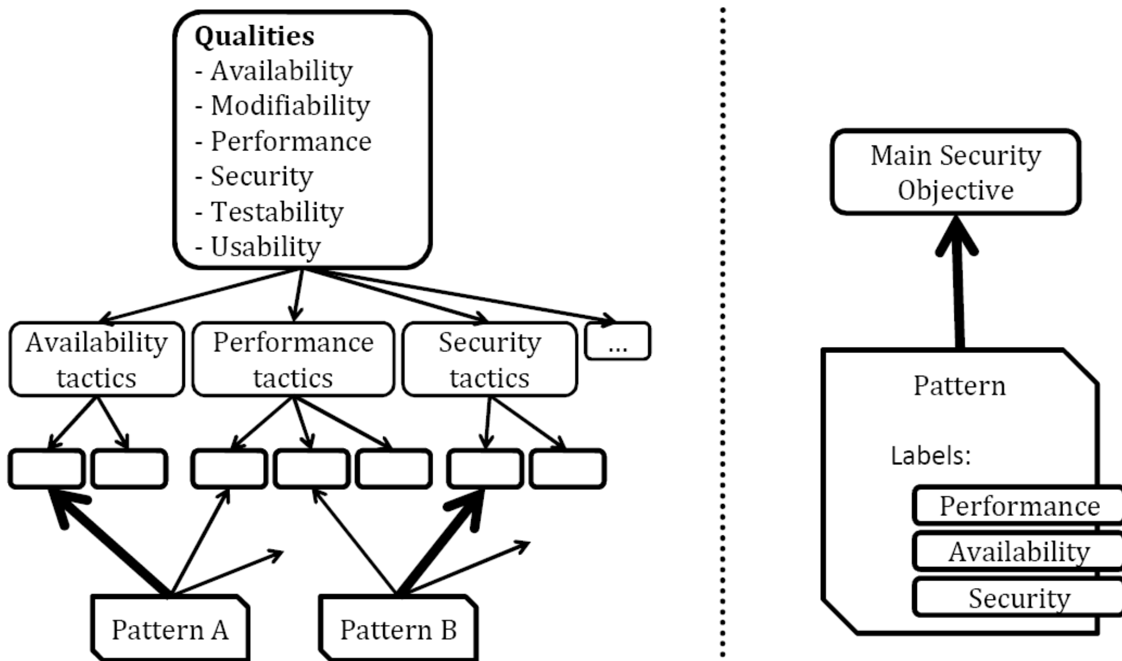

Figure 2.3: Comparing tactics and scenario's with quality trade-off labels.

Our template is shown in Figure 2.4. This template is based on the Gang of Four [GHJV94] template for software design patterns, augmented with additional entries that support the system of patterns as described in this paper. Hence, our main contribution resides in the first part of the template entities (see bold text in Figure 2.4).

The template consists of two parts. The first part provides a short overview of the pattern, collecting entries that enable a quick judgement about the suitability of the pattern for the problem at hand. The entries in this part are:

Pattern name The pattern name should describe the security pattern in a short but clear, depicting way.

Intent The intent of the pattern should be given in a concise way, i.e., what is the purpose of the pattern? What problem does it solve?

Also known as (optional) If the pattern is commonly known by some different names, they can be mentioned here.

Applicability The applicability describes under what circumstances the pattern can be used. This includes both the scope of the pattern and the development phase in which the pattern is most easily applied.

Security objectives The objectives describe the main security objective the pattern tries to solve. Multiple objectives can be given, but this should be rare. If a pattern has an influence on another than its main objective, this can be mentioned in the labels section.

Labels As a part of our contribution, the labels of the pattern describe the impact (both positive and negative) of the pattern on different qualities such as performance and usability, as well as the impact of the pattern on other security objectives.

Relationships A short summary of the inter-pattern relationships should be given. For more details about these relations, the detailed description in the second part should be consulted.

The second part of the template contains the elaborate description of the pattern. This part is similar to the software design pattern template from the Gang of Four, we include it for completeness.

Pattern Name

Intent
Also known as (optional)
Applicability
Security objectives
Labels
Relationships

1. Problem
   - Forces
2. Example
3. Solution
   - Structure
   - Dynamics
   - Participants
   - Collaborations
4. Implementation (optional)
5. Pitfalls (optional)
6. Consequences
7. Related patterns
   - Dependencies
   - Impairments
   - Conflicts
   - Benefits
   - Alternatives
8. Known uses

Figure 2.4: Security Pattern Template.

Problem The problem for which the pattern offers a solution. As part of this description, the different forces which — by their conflicting nature — lead to the problem can be mentioned as well.

Example An example of the application of the pattern. This example should provide an easy case to map the upcoming solution to.

Solution A complete and detailed description of the solution provided by the pattern. This description is comprised of the static structure and the dynamic behavior of the solution (preferably in a graphical notation like UML), the participants together with their responsibilities and, finally, the collaborations between the participants.

Implementation (optional) In this section, clues or ideas about the implementation can be given, possibly including sample code. When alternative implementation methods exist for the pattern, these should be mentioned here.

Pitfalls (optional) The application of the pattern might include some (possibly subtle) pitfalls and risks. Known pitfalls can be mentioned here, optionally including a possible solution.

Consequences The advantages and possible disadvantages of applying the pattern. This can also include a more thorough description of the labels mentioned in the first part.

Related patterns The relations of the pattern with other patterns are noted down here. For a detailed discussion on the different types of relations, consult Section 2.2.1.

Known uses In this part, successful uses of the pattern are given.

The template provides two additional values compared to the templates mentioned in Section 1.2 in order to facilitate a quick selection of the appropriate pattern. First, by explicitly separating the compact overview from the body of the pattern description, the relevant information needed to make a first quick selection is separated from the details. Second, the inclusion of the development phase and security objectives supports the pruning of irrelevant patterns, based on the particular development phase one is in or the security objective one has to deal with.

Incidentally, this information could be used to support the selection process by means of a GUI tool, e.g., to present only the subset of relevant patterns that can be useful for the present design phase. Furthermore, the tool could support and even automate the analysis of the inter-pattern relationships (e.g., hiding conflicting patterns and suggesting the use of beneficial related patterns).

2.2 A system of patterns

Some security patterns complement one another. Suppose the Secure Pipe pattern has been implemented. This pattern *benefits* from the addition of a Security Association to the system. When implementing an Authentication Enforcer, advantages will definitely be gained from a Single Access Point. Also, a case could be made that the correct functioning of the Audit Interceptor *depends* on the presence of a Secure Logger. Conversely, some patterns hamper or even conflict with one another when implemented simultaneously.

In this section, we look at a system of patterns as a regularly interacting (or interdependent) group of items forming a unified whole. To this aim, we propose several types of relationships between security patterns and apply them to the inventory of patterns presented in Section 2.1.

2.2.1 Inter-pattern relationships

We distinguish *five* inter-pattern relationships. These are different gradations of how the implementation of a second pattern, say B, impacts the advantages one has gained by already having implemented a first pattern A. These relationships are, from positive interactions to negative: *depends*, *benefits*, *alternative*, *impairs* and *conflicts*. Note that similar classifications

	System	Firewall	Single Access Point	*Application architecture*	Authentication Enforcer	Authorization Enforcer	Secure Logger	*Application design*	Security Association	Limited View	Full View With Errors	Session
System												
Demilitarized Zone		D										
Secure Pipe									B			
Load Balancer												I
Audit Interceptor							D					
Application architecture												
Authentication Enforcer			B			B						
Authorization Enforcer			B		B							
Application design												
Limited View											AC	
Full View With Errors										AC		

Table 2.1: Relationships among security patterns.

were proposed in the area of feature interactions in the telecommunication domain (see for example [STJ+06]).

Included in Table 2.1 are all identified relationships between a small subset of the security patterns in the inventory (the full table is a 35x35 matrix, which cannot be displayed due to space limitations). In particular, the table will be used by the examples in next subsections. The relationships should be read from left to top and are abbreviated as follows: *D* depends on, *B* benefits from, *A* is alternative to, *I* impairs, *C* conflicts with.

Depends

This is the strongest reinforcement relationship. If pattern *A* depends on pattern *B*, then *A* will not function correctly without *B*.

E.g.: Demilitarized Zone depends on Firewall. Without a firewall, it is impossible to implement a demilitarized zone (or DMZ), as it uses the firewall to partition the network into an external, internal and demilitarized section.

The *depends*-relationship is not symmetrical. In the example given above: while Demilitarized Zone depends on Firewall, the converse is not true. *Depends* is, however, a transitive relationship.

Benefits

Not quite as strong as depends, if pattern *A* benefits from *B*, then implementing *B* will add to the value already provided by implementing *A*. This might be because *B* enables extra functionality in *A*, decreases development time, improves the security added to the system by *A*, etc.

E.g.: Secure Pipe benefits from Security Association. If you want to secure a communication channel using cryptographic techniques, you can drastically improve both security and

computational overhead by first setting up a security association between both communication partners.

Benefits is, in general, not symmetrical (a SECURITY ASSOCIATION does not benefit from a SECURE PIPE). *Benefits* is also not transitive: while a SECURE MESSAGE ROUTER benefits from an AUTHENTICATION ENFORCER and the AUTHENTICATION ENFORCER benefits from a CREDENTIAL TOKENIZER, the SECURE MESSAGE ROUTER does not necessarily benefit from a CREDENTIAL TOKENIZER.

Alternative

This neutral relationship indicates that two patterns, A and B, while not identical, are functionally equivalent. If A is implemented, then the system will lose nothing by replacing A with B. A may be substituted for B without impacting the overall system behaviour and quality.

E.g.: LIMITED VIEW is an alternative to FULL VIEW WITH ERRORS. While both operate in a different way, they are functionally equivalent as they both prevent the user from executing actions that he or she is not authorized to perform.

Alternative is both symmetrical and transitive.

It is important to make the distinction between alternatives and the grouping of similar patterns in the inventory. The grouping in the inventory reduces overlap both by imposing an equivalence relationship on the grouped patterns and by replacing each group of identical patterns with its most prominent member. Note that, since security patterns are not formally defined, it is hard to define “identical”. We treat two patterns as identical if they attempt to achieve the same goal in the same way, and are found in the same category of the inventory.

The alternative relationship, however, connects the representatives of these identical classes that are functionally equivalent, i.e., it connects patterns that are not fully equivalent (otherwise those patterns would already be in the same equivalence class), but have the same function. This difference is subtle and sometimes arbitrary, as the distinction between two different implementation strategies of the same pattern and two alternatives can be small.

As an example, consider KEEP SESSION DATA IN THE CLIENT and KEEP SESSION DATA IN THE SERVER. We consider those to be different implementation strategies of SESSION (i.e. they are in the same group), but a case could be made that these are alternatives and not just two different implementations of the same pattern.

Impairs

If pattern A is impaired by B, then the correct functioning of A might be hampered by implementing B. This does not mean that it is impossible to implement both A and B together, but care must be taken that this does not result in errors.

E.g.: LOAD BALANCER impairs SESSION (more specifically, KEEP SESSION DATA IN THE SERVER). While it is possible to implement a load-balanced web application that uses sessions to maintain user state, the developer must ensure that all requests of one user are forwarded to the server that maintains the session state of that user.

Impairs is symmetrical and not transitive.

Conflicts

If pattern A conflicts with pattern B, then implementing B in a system that contains A will result in inconsistencies. It is not useful to implement both patterns to solve the same problem.

E.g.: LIMITED VIEW conflicts with FULL VIEW WITH ERRORS. Given that both patterns attempt to achieve the same goal in totally different ways, implementing them both at the same time would leave the system inconsistent.

Conflicts is symmetrical and not transitive.

These five relationships, when imposed on the pattern inventory, provide us with a system of security patterns, usable throughout the secure design of an application. We demonstrate the use of the pattern system in the next section.

2.3 Conclusion

While security patterns exist to help architects and designers increase the overall security of a software system, the usability of these patterns is greatly hampered by need for extensive prior knowledge and lack of support to the selection process. That is, typical questions like “Which patterns exist”, “Which pattern helps me to reach a certain security objective”, “Will this pattern conflict with my system and do I have alternatives”, and so forth, are hard to answer without either expertise or guidance. Furthermore, the heterogeneity of the security pattern landscape makes it difficult to integrate automated approaches in the software design process.

In the context of this work, we surveyed the majority of existing publications describing security patterns and filtered them in order to produce a homogeneous, usable system of security patterns. Patterns have been extended with meta-information in order to both facilitate the search process (e.g., see categorizations in Section 2.1) and support consistency during the pattern selection process (e.g., see inter-relationships in Section 2.2).

The presented work focuses on rationaling and improving (quality-wise) the existing security patterns. The work has an added value as a framework to qualify and map out newly discovered patterns. Furthermore, this work constitutes the enabling step to apply security patterns in the context of a pattern-based methodology to build secure software, which is subject to ongoing and future work. Additionally, this work also outlines some gaps in the existing body of security patterns. For instance, to our knowledge, there is no coverage of non-repudiation and privacy concerns, as well as policy patterns. Extending the patterns landscape is a natural continuation of this work and is subject to future work.

Chapter 3

An overview of the inventory

3.1 Classification by development phase

Application Architecture

Authentication Enforcer	Section 4.3 on page 29
Authorization Enforcer	Section 4.4 on page 30
Checkpointed System	Section 4.5 on page 31
Comparator-Checked Fault-Tolerant System	Section 4.6 on page 32
Container-Managed Security	Section 4.7 on page 33
Credential Tokenizer	Section 4.11 on page 37
Input Guard	Section 4.15 on page 41
Output Guard	Section 4.19 on page 45
Replicated System	Section 4.20 on page 46
Secure Access Layer	Section 4.22 on page 48
Secure Logger	Section 4.23 on page 49
Secure Service Facade	Section 4.26 on page 52

Application Design

Controlled Object Factory	Section 4.8 on page 34
Controlled Object Monitor	Section 4.9 on page 35
Full View with Errors	Section 4.14 on page 40
Limited View	Section 4.16 on page 42
Obfuscated Transfer Object	Section 4.18 on page 44
Secure Session Object	Section 4.27 on page 53
Security Association	Section 4.28 on page 54
Security Context	Section 4.29 on page 55
Session	Section 4.31 on page 57
Session Failover	Section 4.32 on page 58
Session Timeout	Section 4.33 on page 59
Subject Descriptor	Section 4.35 on page 61

System	
Application Firewall	Section 4.1 on page 27
Audit Interceptor	Section 4.2 on page 28
Controlled Process Creator	Section 4.10 on page 36
Demilitarized Zone	Section 4.12 on page 38
Firewall	Section 4.13 on page 39
Load Balancer	Section 4.17 on page 43
Reverse Proxy	Section 4.21 on page 47
Secure Message Router	Section 4.24 on page 50
Secure Pipe	Section 4.25 on page 51
Server Sandbox	Section 4.30 on page 56
Single Access Point	Section 4.34 on page 60

3.2 Classification by security objective

Items with a * are references to another objective.

Confidentiality	
– *Secure Data Transmission**	
– *Controlled Access**	

Integrity	
– *Application Integrity*	
Application Architecture	
Checkpointed System	Section 4.5 on page 31
Comparator-Checked Fault-Tolerant System	Section 4.6 on page 32
Input Guard	Section 4.15 on page 41
Output Guard	Section 4.19 on page 45
Secure Access Layer	Section 4.22 on page 48
Application Design	
Controlled Object Factory	Section 4.8 on page 34
System	
Controlled Process Creator	Section 4.10 on page 36
Server Sandbox	Section 4.30 on page 56
– *Controlled Access**	

Availability	
Application Architecture	
Replicated System	Section 4.20 on page 46
Application Design	
Session Failover	Section 4.32 on page 58
Session Timeout	Section 4.33 on page 59
System	
Load Balancer	Section 4.17 on page 43
Reverse Proxy	Section 4.21 on page 47

Accountability *Accountability requires Identification**	
Application Architecture	
Secure Logger	Section 4.23 on page 49
Application Design	
System	
Audit Interceptor	Section 4.2 on page 28

Secure Data Transmission	
Application Architecture	
Application Design	
Obfuscated Transfer Object	Section 4.18 on page 44
Security Association	Section 4.28 on page 54
System	
Secure Message Router	Section 4.24 on page 50
Secure Pipe	Section 4.25 on page 51

Controlled Access *Controlled Access requires Identification**	
Application Architecture	
Authentication Enforcer	Section 4.3 on page 29
Authorization Enforcer	Section 4.4 on page 30
Container-Managed Security	Section 4.7 on page 33
Secure Service Facade	Section 4.26 on page 52
Application Design	
Controlled Object Monitor	Section 4.9 on page 35
Full View with Errors	Section 4.14 on page 40
Limited View	Section 4.16 on page 42
System	
Application Firewall	Section 4.1 on page 27
Demilitarized Zone	Section 4.12 on page 38
Firewall	Section 4.13 on page 39
Single Access Point	Section 4.34 on page 60

Identification	
Application Architecture	
Container-Managed Security	Section 4.7 on page 33
Credential Tokenizer	Section 4.11 on page 37
Application Design	
Secure Session Object	Section 4.27 on page 53
Security Context	Section 4.29 on page 55
Session	Section 4.31 on page 57
Subject Descriptor	Section 4.35 on page 61
System	

3.3 Inter-pattern relationships

The rationale for each relationship is described in the Related Patterns-section for each pattern.

Depends on...	
Audit Interceptor (4.2, p28)	Secure Logger (4.23, p49)
Controlled Object Factory (4.8, p34)	Controlled Object Monitor (4.9, p35)
Controlled Object Monitor (4.9, p35)	Authorization Enforcer (4.4, p30)
Demilitarized Zone (4.12, p38)	Firewall (4.13, p39)
Replicated System (4.20, p46)	Load Balancer (4.17, p43)
Reverse Proxy (4.21, p47)	Demilitarized Zone (4.12, p38)
Secure Session Object (4.27, p53)	Session (4.31, p57)
Security Association (4.28, p54	Secure Pipe (4.25, p51), Security Context (4.29, p55)
Security Context (4.29, p55)	Subject Descriptor (4.35, p61)
Session Failover (4.32, p58)	Load Balancer (4.17, p43), Session (4.31, p57)
Session Timeout (4.33, p59)	Session (4.31, p57)

Benefits from...	
Audit Interceptor (4.2, p28)	Secure Service Façade (4.26, p52)
Authentication Enforcer (4.3, p29)	Secure Pipe (4.25, p51), Secure Service Façade (4.26, p52)
Authorization Enforcer (4.4, p30)	Authentication Enforcer (4.3, p29), Secure Service Façade (4.26, p52)
Checkpointed System (4.5, p31)	Comparator-Checked Fault-Tolerant System (4.6, p32)
Controlled Object Monitor (4.9, p35)	Authentication Enforcer (4.3, p29)
Credential Tokenizer (4.11, p37)	Secure Pipe (4.25, p51)
Firewall (4.13, p39)	Demilitarized Zone (4.12, p38), Application Firewall (4.1, p27)
Input Guard (4.15, p41)	Output Guard (4.19, p45)
Load Balancer (4.17, p43)	Replicated System (4.20, p46), Session Failover (4.32, p58)
Output Guard (4.19, p45)	Input Guard (4.15, p41)
Secure Logger (4.23, p49)	Secure Pipe (4.25, p51)
Secure Pipe (4.25, p51)	Security Association (4.28, p54)
Security Context (4.29, p55)	Security Association (4.28, p54)
Session (4.31, p57)	Session Timeout (4.33, p59), Secure Session Object (4.27, p53), Session Failover (4.32, p58)
Subject Descriptor (4.35, p61)	Security Context (4.29, p55)

Is an alternative for...

Application Firewall (4.1, p27)	Reverse Proxy (4.21, p47), Input Guard (4.15, p41)
Authentication Enforcer (4.3, p29)	Container Managed Security (4.7, p33)
Authorization Enforcer (4.4, p30)	Container Managed Security (4.7, p33)
Container Managed Security (4.7, p33)	Authorization Enforcer (4.4, p30), Authentication Enforcer (4.3, p29)
Full View with Errors (4.14, p40)	Limited View (4.16, p42)
Input Guard (4.15, p41)	Application Firewall (4.1, p27)
Limited View (4.16, p42)	Full View with Errors (4.14, p40)
Load Balancer (4.17, p43)	Reverse Proxy (4.21, p47)

Impairs...

Checkpointed System (4.5, p31)	Audit Interceptor (4.2, p28)
Firewall (4.13, p39)	Reverse Proxy (4.21, p47)
Load Balancer (4.17, p43)	Session (4.31, p57)
Reverse Proxy (4.21, p47)	Secure Pipe (4.25, p51)

Conflicts with...

Full View with Errors (4.14, p40)	Limited View (4.16, p42)
Limited View (4.16, p42)	Full View with Errors (4.14, p40)

Chapter 4

The inventory of security patterns

4.1 Application Firewall [DGFRLP04]

Intent To filter calls and responses to/from enterprise applications, based on an institution access control policies.

Also known as Content Firewall

Applicability System

Security objectives Controlled Access

Labels -Performance, +Manageability, +Auditability

Relationships A: Reverse Proxy, Input Guard

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

As an alternative, Reverse Proxy could be used to implement basic functionality of an application firewall. The Reverse Proxy is mainly focused on availability, however. Also, the Reverse Proxy requires the implementation of additional Firewall instances.

Another alternative to an application firewall is an Input Guard. The Input Guard protects individual components, while the Application Firewall is placed in front of an entire system, protecting an application.

4.2 Audit Interceptor [SNL05]

Intent You want to intercept and audit requests and responses to and from the Business tier, in a flexible and modifyable way.

Applicability System

Security objectives Accountability

Labels -Performance, +Auditability, +Maintainability, +Manageability

Relationships D: Secure Logger; B: Secure Service Façade

Related patterns

Dependencies

An Audit Interceptor depends on some secure logging facility, preferably a Secure Logger, to store audit events. Without it, it is impossible to guarantee the integrity of the audit trails, rendering them useless, as the attacker might have forged them.

Benefits

The Audit Interceptor benefits from the presence of a Secure Service Façade. As the purpose of the Façade is to forward to the application services and business objects, it is the preferred place to audit requests.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.3 Authentication Enforcer [SNL05]

Intent You need to verify that each service request is from an authenticated entity.

Applicability Application Architecture

Security objectives Controlled Access

Labels +Maintainability, +Manageability, +Auditability, -Anonymity, +Privacy, +Accountability, +Portability

Relationships B: Secure Pipe, Secure Service Facçade; A: Container Managed Security

---

Related patterns

Dependencies

None identified.

Impairments

None identified.

Conflicts

None identified.

Benefits

The Authentication Enforcer benefits from a Secure Pipe to protect the user's credentials during transmit at login time. It also benefits from a Secure Service Façade, as these two patterns are commonly combined to implement authentication: the facçade delegates requests to the Authentication Enforcer, which then retrieves the appropriate credentials and performs the authentication.

Alternatives

Authentication Enforcer may be replaced with Container Managed Security, with the added functionality that Container Managed Security also provides authorization.

4.4 Authorization Enforcer [SNL05]

Intent Verify that requests for services are properly authorized at the method and link level.

Applicability Application Architecture

Security objectives Controlled Access

Labels +Maintainability, +Manageability, +Auditability, +Accountability, +Portability

Relationships B: Authentication Enforcer, Secure Service Façade; A: Container Managed Security

---

Related patterns

Dependencies

None identified.

Impairments

None identified.

Conflicts

None identified.

Benefits

The Authorization Enforcer benefits from a Secure Service Façade, as these two patterns are commonly combined to implement authentication: the facçade delegates requests to the Authorization Enforcer, which then retrieves the appropriate information and performs the authorization.

The Authorization Enforcer also benefits from an Authentication Enforcer, as the latter can authenticate the users before an authorization decision is made.

Alternatives

The functionality provided by an Authorization Enforcer can also be implemented by using Container Managed Security, where the Container Managed Security provides the additional functionality of authentication.

4.5 Checkpointed System [BHm04]

Intent Structure a system so that its state can be recovered and restored to a known valid state in case a component fails.

Also known as Snapshot, Undo

Applicability Application Architecture

Security objectives Integrity (more specifically, application integrity)

Labels +Dependability, -Performance, -Cost

Relationships B: Comparator-Checked Fault-Tolerant System; I: Audit Interceptor

---

Related patterns

Dependencies

None identified.

Benefits

To detect failures, a Comparator-Checked Fault-Tolerant System can be implemented. Since that pattern also uses memento's to store the state of a component, the effort for implementing that pattern may be lowered significantly.

Impairments

Care must be taken when an Audit Interceptor is used together with a Checkpointed System. When a component fails and is reverted back to a previously saved state, the currently executing operation might not have been fully completed, and/or its effects might have been undone. This must be reflected in the audit logs, so that no false operations or events are logged.

Conflicts

None identified.

Alternatives

None identified.

4.6 Comparator Checked Fault Tolerant System [BHm04]

Intent Structure a system so that an independent failure of one component will be detected quickly and so that an independent single-component failure will not cause a system failure.

Also known as Tandem System

Applicability Application Architecture

Security objectives Integrity (application integrity)

Labels -Cost, -Performance, +Dependability

Relationships I: Audit Interceptor; A: Output Guard; B: Checkpointed System

---

Related patterns

Dependencies

None identified.

Benefits

A Checkpointed System largely conforms to the structure of a comparator checked fault tolerant system. The main difference is that the Checkpointed System uses the memento's to possibly roll the application back to the last known "correct" state. This check for correctness can then be implemented using a comparator check of these memento's between two similar components.

Impairments

When an Audit Interceptor is used to log the actions of a system, care must be taken that the actions of components that fail the comparison in a comparator checked fault tolerant system are marked as "failed", or otherwise distinguished from correctly completed operations.

Conflicts

None identified.

Alternatives

Functionality of a comparator checked fault tolerant system can be partially replaced by an Output Guard. While the fault tolerant system compares two different calculations of the output, an Output Guard only validates the syntactical correctness of the output.

4.7 Container Managed Security [SNL05]

Intent You need a simple, standard way to enforce authentication and authorization in your J2EE applications and don't want to reinvent the wheel or write home-grown security code. Using a Container Managed Security pattern, the container performs user authentication and authorization without requiring the developer to hard-wire security policies in the application code.

Applicability Application Architecture

Security objectives Identification, Controlled Access

Labels +Manageability, +Portability, +Integrity

Relationships A: Authentication Enforcer, Authorization Enforcer

---

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

The authentication functionality of Container Managed Security may be replaced by an Authentication Enforcer. Similarly, its authorization functionality may be replaced by an Authorization Enforcer.

4.8 Controlled Object Factory [SFBH+06]

Intent This pattern addresses how to specify the rights of processes with respect to a new object. When a process creates a new object through a factory (see Factory Method [GHJV94] and Abstract Factory [GHJV94]), the request includes the features of the new object. These features include a list of rights to access the object.

Applicability Design

Security objectives Integrity (application integrity)

Labels -Performance, +Manageability

Relationships D: Controlled Object Monitor

Related patterns

Dependencies

The Controlled Object Factory depends on a Controlled Object Monitor to correctly enforce the supplied access privileges when invocations are made on objects generated by the factory.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.9 Controlled Object Monitor [SFBH+06]

Intent This pattern addresses how to control access by a process to an object. Use a reference monitor to intercept access requests from processes. The reference monitor checks whether the process has the requested type of access to the object.

Applicability Application Design

Security objectives Controlled Access

Labels +Maintainability, +Manageability, +Auditability, +Accountability

Relationships D: Authorization Enforcer; B: Authentication Enforcer

---

Related patterns

Dependencies

The Controlled Object Monitor depends on the Authorization Enforcer, as it is a specialization of this pattern (it extends it with added functionality).

Benefits

As with the Authorization Enforcer, the Controlled Object Monitor benefits from an Authentication Enforcer to identify and authenticate entities before attempting to access objects.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.10 Controlled Process Creator [SFBH+06]

Intent This pattern addresses how to define and grant appropriate access rights for a new process, in an operating system in which processes or threads need to be created according to application needs.

Applicability System

Security objectives Integrity (Application Integrity)

Labels -Performance, +Manageability, +Dependability

Relationships —

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.11 Credential Tokenizer [SNL05]

Intent You need a flexible mechanism to encapsulate a security token that can be used by different security infrastructure providers.

Applicability Application Architecture

Security objectives Identification

Labels +Portability, +Manageability, +Useability

Relationships B: Secure Pipe

Related patterns

Dependencies

None identified.

Benefits

The Credential Tokenizer benefits from a Secure Pipe, as this allows for secure communication between the client and server, or servers of different business partners.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.12 Demilitarized Zone [SFBH+06]

Intent Any organization conducting e-commerce or publishing information over Web technologies must make their service easily accessible to their users. However, any form of Web site or e-commerce system is a potential target for attack, especially those on the Internet. A Demilitarized Zone (DMZ) separates the business functionality and information from the Web servers that deliver it, and places the Web servers in a secure area. This reduces the "surface area" of the system that is open to attack.

Applicability System

Security objectives Controlled Access

Labels -Performance, -Cost, -Manageability, -Dependability, +Maintainability

Relationships D: Firewall

Related patterns

Dependencies

The Demilitarized Zone depends on a Firewall, since these are needed to compartimentalize the network, thereby implementing a DMZ.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.13 Firewall [Sch03]

Intent Control incoming and outgoing network connections, restrict access to certain hosts on the network level.

Applicability System

Security objectives Controlled Access

Labels +Accountability, -Performance, -Dependability

Relationships B: Demilitarized Zone, Application Firewall, I: Reverse Proxy

---

Related patterns

Dependencies

None identified.

Impairments

A Firewall is impaired by a Reverse Proxy (or potentially all non-transparent intermediaries), as the network connections do not appear to be originating from the proxied servers anymore, but from the proxy itself. The firewall should be positioned so that it is able to make the necessary distinctions between authorized and unauthorized connections.

Conflicts

None identified.

Benefits

A Firewall benefits from a Demilitarized Zone, as the latter pattern provides a time-tested method of partitioning a network with respect to an internal part, an external part and a so-called demilitarized zone, to add an extra layer of security.

The Firewall might also benefit from an Application Firewall, especially when implementing "defence in depth", as it adds another layer of security (application-level security in addition to network-level security).

Alternatives

None identified.

4.14 Full View with Errors [YB97]

Intent Prevent users to perform illegal operations by showing an error message when the user tries to perform an illegal operation.

Also known as Full View With Exceptions, Reveal All and Handle Exceptions, Notified View

Applicability Application Design

Security objectives Access Control

Labels +Maintainability, -Usability

Relationships C: Limited View; A: Limited View

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

The Full View with Errors cannot be combined with the LIMITED VIEW, especially when applied to a single operation. For different operations different strategies could be chosen, although this does not result in a consistent user interface and will most likely increase the users' confusion.

Alternatives

By using the LIMITED VIEW pattern, the operations that are not available can be hidden.

4.15 Input Guard [Sar03]

Intent Protect components from input that does not conform to the system specification.

Applicability Application Architecture

Security objectives Integrity

Labels +Auditability, -Performance

Relationships A: Application Firewall; B: Output Guard

Related patterns

Dependencies

None identified.

Impairments

None identified.

Conflicts

None identified.

Benefits

The Output Guard complements the Input Guard, as it helps to ensure that components also provide valid output.

Alternatives

For a system-wide alternative, the Input Guard may be replaced with an Application Firewall. Note, however, that the Application Firewall does not operate on a component level; the Input Guard enables a much finer level of control.

4.16 Limited View [YB97]

Intent Prevent users to perform illegal operations by hiding all operations that cannot be performed by the user.

Also known as Blinders, Child Proofing, Invisible Road Blocks, Hiding the cookie jars, Early Authorization

Applicability Application Design

Security objectives Access Control

Labels +Usability

Relationships C: Full View with Errors; A: Full View with Errors

---

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

The Limited View cannot be combined with the Full View with Errors, especially when applied to a single operation. For different operations different strategies could be chosen, although this does not result in a consistent user interface and will most likely increase the users' confusion.

Alternatives

By using the Full View with Errors pattern, the operations that are not available are shown but result in an error message being displayed. This improves the user interface consistency but decreases the usability.

4.17 Load Balancer [Sor02]

Intent Distribute the load from multiple users over several servers.

Applicability System

Security objectives Availability

Labels +Performance

Relationships B: Replicated System, Session Failover; I: Session, A: Reverse Proxy

---

Related patterns

Dependencies

None identified.

Benefits

When combining sessions with a load balancer, it is beneficial to also implement the Session Failover pattern, so that users do not loose their sessions when a server becomes unavailable.

When distributing the load over different identical systems, the Replicated System pattern can be used. The load balancer plays the role of the Workload Management Proxy in this case.

Impairments

When combining a Session pattern with the load balancer, care must be taken that the requests belonging to the same session are always forwarded to the same server (at least, if session data is stored on the server). This implementation of the load balancer can become quite complicated.

Conflicts

None identified.

Alternatives

None identified.

4.18 Obfuscated Transfer Object [SNL05]

Intent You need a way to protect critical data as it is passed within application and between tiers.

Applicability Application Design

Security objectives Confidentiality (Secure Data Transmission)

Labels +Manageability, +Maintainability, +Portability

Relationships —

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.19 Output Guard []

Intent Confine an error in the component that contains the fault which led to that error.

Applicability Application Architecture

Security objectives Integrity

Labels +Auditability, -Performance

Relationships B: Input Guard

Known uses

Related patterns

Dependencies

None identified.

Benefits

The Output Guard benefits from an Input Guard, as it helps to ensure that the component does not receive erroneous input (potentially causing errors).

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.20 Replicated System [BHm04]

Intent Structure a system which allows provision of service from multiple points of presence, and recovery in case of failure of one or more components or links.

Also known as Redundant Components, Horizontal Scalability

Applicability Application Architecture

Security objectives Availability

Labels -Manageability, -Cost

Relationships D: Load Balancer

Related patterns

Dependencies

The Workload Management Proxy is an instance of the LOAD BALANCER pattern.

Benefits

None identified.

Impairments

When the SESSION pattern is used together with the Replicated System (and state is stored on the server side), care must be taken that the requests belonging to the same session are forwarded to the same replica each time.

Conflicts

None identified.

Alternatives

None identified.

4.21 Reverse Proxy [Som03]

Intent Protect your web server infrastructure on an application protocol level, without hindering accessibility.

Applicability System

Security objectives Availability

Labels +Manageability, -Maintainability, +Integrity

Relationships D: Demilitarized Zone; I: Secure Pipe

Related patterns

Dependencies

The Reverse Proxy depends on the correct implementation of a Demilitarized Zone, so that the proxy can be situated in the DMZ and the actual web server in the internal network.

Benefits

None identified.

Impairments

The Reverse Proxy impairs the Secure Pipe, as it acts as a person-in-the-middle, and therefore breaks direct connections that might be assumed by the Secure Pipe. The impairment may be circumvented by deploying additional Secure Pipes: one from the client to the proxy, and potentially one from the proxy to the backend server.

Conflicts

None identified.

Alternatives

An alternative to the Reverse Proxy, without the implied Demilitarized Zone, is the Application Firewall. This pattern also enables application layer filtering.

4.22 Secure Access Layer

Intent Application security will be insecure if it is not properly integrated with the security of the external systems it uses. On top of the lower-level security, build a secure access layer for communicating in and out of the program.

Also known as Using Low-level security, Using Non-application security, Only as strong as the weakest link

Applicability Application Architecture

Security objectives Application Integrity

Labels +Maintainability, +Portability

Relationships —

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.23 Secure Logger [SNL05]

Intent Application events must be logged in a centralized way, and it should be impossible to alter lo files.

Applicability Application Architecture

Security objectives Accountability

Labels +Confidentiality, +Integrity, +Maintainability, +Manageability, -Performance

Relationships B: Secure Pipe

Related patterns

Dependencies

None identified.

Benefits

As indicated in the Implementation section, a Secure Pipe can be used to implement the secure logger.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.24 Secure Message Router [SNL05]

Intent Securely communicate with multiple partner endpoints using message-level security and identity-federation mechanisms.

Applicability System

Security objectives Secure Data Transmission

Labels +Manageability, +Maintainability, +Confidentiality, +Integrity, -Availability

Relationships —

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.25 Secure Pipe [SNL05]

Intent You need to provide privacy and prevent eavesdropping and tampering of client transactions caused by man-in-the-middle attacks.

Applicability System

Security objectives Secure Data Transmission

Labels —

Relationships B: Security Association

Related patterns

None identified.

Dependencies

None identified.

Impairments

None identified.

Conflicts

None identified.

Benefits

A Secure Pipe benefits from a Security Association, as this can be used to enhance both performance as the assurance that both parties are who they claim to be.

Alternatives

None identified.

4.26 Secure Service Façade [SNL05]

Intent You need a secure gateway mandating and governing security on client requests, exposing a uniform, coarse-grained service interface over fine-grained, loosely coupled business services that mediates client requests to the appropriate services.

Applicability Application Architecture

Security objectives Controlled Access

Labels +Identification, +Auditing, +Manageability, +Maintainability

Relationships —

Related patterns

Dependencies

None identified.

Impairments

None identified.

Conflicts

None identified.

Benefits

None identified.

Alternatives

None identified.

4.27 Secure Session Object [SNL05]

Intent You need to facilitate distributed access and seamless propagation of security context and client sessions in a platform-independent and location-independent manner.

Applicability Application Design

Security objectives Identification

Labels +Portability, +Confidentiality, +Performance

Relationships D: Session

Related patterns

Dependencies

Since the Secure Session Object encapsulates certain session-related data, it depends on the Session pattern.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.28 Security Association [BHm04]

Intent Define a structure which provides each participant in a secure communication with the information it will use to protect messages to be transmitted to the other party, and with the information which it will use to understand and verify the protection applied to messages received from the other party.

Applicability Application Design

Security objectives Secure Data Transmission

Labels +Performance

Relationships D: Secure Pipe, Security Context

Related patterns

Dependencies

- A Security Association is associated to a SECURE PIPE and stores information used to protect message traffic.
- A Security Association instance is set up using information stored in a SECURITY CONTEXT.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.29 Security Context [SFBH+06]

Intent Provide a container for security attributes and data relating to a particular execution context, process, operation, or action.

Applicability Application Design

Security objectives Identification

Labels +Controlled Access, +Maintainability

Relationships D: Subject Descriptor; B: Security Association

Related patterns

Dependencies

A Security Context refers to information related to a subject, which is contained in a SUBJECT DESCRIPTOR.

Benefits

A Security Context is often used in combination with a SECURITY ASSOCIATION.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.30 Server Sandbox [KETEH]

Intent Many site defacements and major security breaches occur when a new vulnerability is discovered in the Web server software. Yet most Web servers run with far greater privileges than are necessary. The Server Sandbox pattern builds a wall around the Web server in order to contain the damage that could result from an undiscovered bug in the server software.

Also known as Privilege Drop, Untrusted Server, Constrained Execution Environment, Unprivileged/Restricted User Account, Run as Nobody

Applicability System

Security objectives Integrity, more specifically Application Integrity

Labels -Manageability, -Performance, -Cost

Relationships —

Related patterns

Dependencies

None identified.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.31 Session [YB97]

Intent Many objects need access to shared values, but the values are not unique throughout the system.

Also known as User's Environment, Namespace, Threaded-based Singleton, Localized Globals

Applicability Application Design

Security objectives Identification

Labels -Anonymity, -Privacy, +Usability

Relationships B: Session Timeout, Secure Session Object, Session Failover

---

Related patterns

Dependencies

None identified.

Benefits

The Session can offer increased security functionality through the Session Timeout or the Session Failover. If there is a need to transfer session data securely across different tiers, the session also benefits from a Secure Session Object.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.32 Session Failover [Sor02]

Intent Avoid inconveniencing users that lose session data in a system restart.

Applicability Application Design

Security objectives Availability

Labels -Performance, -Cost

Relationships D: Load Balancer, Session

Related patterns

Dependencies

Session Failover depends on a Load Balancer, as it is useless without one. It also depends on Session, as it "inherits" most of its functionality from this pattern. The Session Failover benefits from the *Keep Session Data in Server* implementation alternative of the Session on which it is based, as this implementation allows for the survival of session data across system restarts.

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.33 Session Timeout [Sor02]

Intent Prevent the system from running out of resources because abandoned sessions are not cleaned up.

Applicability Application Design

Security objectives Availability

Labels -Usability, -Privacy, +Cost

Relationships D: Session

Related patterns

Dependencies

This pattern extends the Session pattern and is necessary when session data is kept on the server (to limit the amount of data on the server).

Benefits

None identified.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

4.34 Single Access Point [YB97]

Intent Reduce the "attack surface" by imposing a single access point on the system, providing an ideal place to do access control and policy enforcement.

Also known as Login Window, One Way In, Guard Door, Validation Screen

Applicability System

Security objectives Controlled Access

Labels +Manageability

Relationships —

Related patterns

Dependencies

None identified.

Impairments

None identified.

Conflicts

None identified.

Benefits

None identified.

Alternatives

None identified.

4.35 Subject Descriptor [SFBH+06]

Intent Provide access to security-relevant attributes of an entity on whose behalf operations are to be performed.

Also known as Subject Attributes. The entity described may be referred to as a subject or principal.

Applicability Application Design

Security objectives Identification

Labels +Controlled Access

Relationships B: Security Context

Related patterns

Dependencies

None identified.

Benefits

A Security Context can use the Subject Descriptor to represent the attributes of subjects.

Impairments

None identified.

Conflicts

None identified.

Alternatives

None identified.

Bibliography

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition.* Addison Wesley, 2003.

[BHm04] Bob Blakley, Craig Heath, and members of The Open Group Security Forum. *Security Design Patterns.* The Open Group Security Forum, 2004.

[CCp06] Common criteria for information technology security evaluation; part 2: Security functional components (version 3.1, revision 1), 2006.

[DGFRLP04] Nelly Delessy-Gassant, Eduardo B. Fernandez, Sajeed Rajput, and Maria M. Larrondo-Petrie. Patterns for application firewalls. *PLoP*, 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional, 1994.

[HCS04] Spyros T. Halkidis, Alexander Chatzigeorgiou, and George Stephanides. A qualitative evaluation of security patterns. Malaga, Spain, October 2004. International Conference on Information and Communications Security (ICICS).

[KCC03] Sascha Konrad, Betty H.C. Cheng, and Laura A. Campbell. Using security patterns to model and analyze security requirements. In *IEEE International Conference on Requirements Engineering (RE)*, Monterey Bay, CA, USA, 2003.

[KETEH] Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. Security patterns repository, version 1.0.

[KETEH02] D. Kienzle, M. Elder, D. Tyree, and J. Edwards-Hewitt. Security patterns template and tutorial. February 2002.

[MN06] Oleksiy Mazhelis and Anton Naumenko. The place and role of security patterns in software development process. In *Security in Information Systems, Proceedings of the 4th International Workshop on Security in Information Systems*, 2006.

[RGFMP06] David G. Rosado, Carlos Gutiérrez, Eduardo Fernández-Medina, and Mario Piattini. Security patterns related to security requirements. In *Security in Information Systems, Proceedings of the 4th International Workshop on Security in Information Systems*, 2006.

[Sar03] Titos Saridakis. Design patterns for fault containment. 2003.

[Sch03] Markus Schumacher. Firewall patterns. *EuroPLoP*, 2003.

[SFBH+06] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns.* Wiley, 2006.

[SNL05] Christopher Steel, Ramesh Nagappan, and Ray Lai. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management.* Prentice Hall Ptr, 2005.

[Som03] Peter Sommerlad. Reverse proxy patterns. 2003.

[Sor02] Kristian Elof Sorensen. Session patterns. 2002.

[SR02] Markus Schumacher and Utz Roedig. Security engineering with patterns. 2002.

[STJ+06] Frans Sanen, Eddy Truyen, Wouter Joosen, Andrew Jackson, Andronikos Nedos, Siobhan Clarke, Neil Loughran, and Awais Rashid. Classifying and documenting aspect interactions. pages 23–26, 2006.

[VM01] John Viega and Gary McGraw. *Building Secure Software.* 2001.

[YB97] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. *PLoP*, 1997.