Architecting software with security patterns

Riccardo Scandariato *Koen Yskout*
Thomas Heyman *Wouter Joosen*

Report CW 515, April 2008

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

Architecting software with security patterns

Riccardo Scandariato *Koen Yskout*
Thomas Heyman *Wouter Joosen*

Report CW 515, April 2008

Department of Computer Science, K.U.Leuven

Abstract

Security patterns, as domain-independent expert knowledge packaged in a reusable format, are able to offer significant guidance to the software engineer in developing secure systems. However, the overabundance of published security patterns complicates the process of finding the right pattern to solve the problem at hand. This is due to three reasons. First, not all security patterns are relevant to the software engineer. Second, the domain independence of patterns sometimes complicates finding a solution to a domain specific problem. Third, patterns exist on different levels of abstraction. Not all patterns can be applied to every step in the development process of a system.

This report proposes a method to facilitate the selection of a suitable set of security patterns to realize a specific set of security requirements. It is comprised of two parts. First, additional structure is superimposed on this collection. Second, a methodology is proposed that, given this structured inventory of patterns, guides the software engineer from the security requirements to an appropriate solution using patterns, taking into account various trade-offs and relations between patterns.

Keywords : Secure software, Security patterns, Software engineering.

1 Introduction

When developing an application, creating the full application from scratch is impossible, given the current complexity of an average application. Therefore, existing resources have to be reused as much as possible. For example, modern enterprise applications make heavy use of component technology and middleware facilities to reduce the overall implementation effort. However, this is not the full picture. Sometimes, it is not code or a service that can be reused, but it is, for instance, a certain reoccurring design structure. In this case, a different reuse strategy is necessary.

In the software engineering discipline, patterns represent a well-known technique to achieve this goal, namely by packaging domain-independent knowledge and expertise in a reusable fashion. Architectural and design patterns constitute solid solutions that can be employed out of the box by architects and designers in order to solve known, recurrent problems. For instance, the Gang of Four patterns [GHJV94] are the best known and are used extensively in today's frameworks and products. A pattern provides three main advantages. First, the solution is known to be sound because it is time-tested. Multiple successful instantiations of a pattern over time have proven its value, up until a certain extent. Second, benefits and drawbacks of a pattern are known in advance and they can be taken into account at development time, e.g., to trade-off among alternative design solutions. Indeed, because the pattern was used multiple times before, its advantages and disadvantages are known and can be described together with the pattern. This will avoid unpleasant surprises later on. Third, patterns establish a common vocabulary that can ease communication between different stakeholders. For example, every software engineer who has experience with design patterns knows what is meant by a *Singleton* or *Decorator*. It is therefore not surprising that almost fifteen years after the publication of the first catalog, design patterns have proven to be successful in software engineering [Gam06].

Also in the software *security* discipline, a recognized principle states to reuse community resources in order to avoid reinventing ad-hoc solutions from scratch [VM02], and this for two reasons. First of all, inventing a new solution schema (e.g., a new encryption protocol) is risky because of the likelihood of design flaws. Security solutions are not easily devised, and it requires a lot of expertise to get them right. Therefore, trying to 'do something smart' will most likely result in a solution that is easily circumvented. Likewise, it is not advisable to re-implement a well-known solution from scratch, because of possible development flaws. Small details matter a lot when securing an application. Therefore, reusing an existing, time-proven implementation, if available, is always better.

Security patterns offer invaluable help in order to enforce this security principle at the architectural and design level. First, design glitches can be avoided by applying well-known design solutions. Second, security patterns should include enough detailed information (down to the level of reference code) to help automate the implementation phase. In other words, security patterns are means to provide additional assurance that a software artifact is correct.

Security patterns have gained significant attention by the research community after the seminal work by Yoder and Barcalow [YB97]. Similarly to design patterns, they have been around for a significant amount of time. Indeed, in the period from 1997-2006, about 220 security patterns have been published.

However, it is remarkable that the adoption of security patterns is lagging behind, especially if compared with the vast success of design patterns. Despite the extensive literature, security patterns still have an inadequate reputation in the security community. As the main reasons for this, we see the high number of patterns and the heterogeneous nature of the patterns. Both of these hamper the usability of the security patterns. To make it easier for a developer to use security patterns, we will create a structured *inventory* of security patterns, and craft a *methodology* that employs this inventory in an efficient, systematic and user-friendly manner.

Patterns mentioned in this document will be typeset in Small caps. A short description and a reference to the source of the pattern is given in A.

2 Inventory

As stated in [YHSJ08], a clear definition of what makes a security pattern is missing. As a consequence, the pattern landscape is heterogeneous in nature and contains several outliers. A heterogeneous set of patterns can be confusing for the user and may generate frustration during the selection process. To create the inventory, we only make use of the set of 'core patterns' as determined in [YHSJ08]. There, a core pattern was defined as "a security pattern with a relevant problem description and a constructible, intensional solution which consists of software entities, arranged in a well-determined spatial and/or behavioral configuration, that interact externally with abstract participants". The instruments that will be described next in order to make the pattern collection easier to use, are specifically intended for this group. From the viewpoint of the software engineer, this restriction already constitutes a substantial reduction in the amount of patterns to be considered.

In Section 2.1, a categorization according to development process phases is presented. Next, the links between a pattern and the security objective(s) it tries to achieve is explicated in Section 2.2. Subsequently, in Section 2.3 non-functional labels are assigned to the patterns. The interrelationships between patterns are then explicited in Section 2.4. Finally, a unified template for the description of a security pattern is given in Section 2.5.

2.1 Development phase classification

In order to impose additional structure on the security pattern landscape, a categorization according to the development phase to which the pattern is applicable can be superimposed on the core pattern collection. Three such categories 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 development of the software application itself. These last patterns can be further detailed according to the design phase in which they are best applied: at the (higher level) architectural phase or on the (lower level) detailed design phase. Given these distinctions, the following categories naturally arise as shown in Figure 2.

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

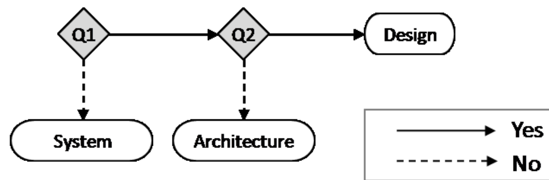

Figure 1: Classifying core patterns according to development level.

and/or inserts dependencies within an extensive part of the application. For example, it could introduce an abstraction layer or a security component. To make the definition more precise and operational, the locality principle introduced by Eden and Kazman [EK03] is adopted. Correspondingly, an architectural pattern is defined as being both intensional and non-local, meaning that it has an unbounded number of instances and it *cannot* be satisfied in a distinct part of the program without this affecting how the rest of the program functions.

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 can introduce some form of encapsulation of security data. More precisely, a design pattern is both intensional and local, meaning that it has an unbounded number of instances and it *can* be satisfied in a distinct part of the program without this affecting how the rest of the program functions [EK03].

Examples 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 excluded.

Examples for this category are the FIREWALL and the CONTAINER MANAGED SECURITY pattern.

2.1.1 Method and results

In order to categorize the core patterns according to the phase of the development process to which they belong, a second decision chart is used. Again, the classification process was performed independently by three reviewers with a final decision based on majority. The decision chart for this part of the process is depicted in Figure 1, and starts at question Q1. The questions are, in order:

Q1 *Is the source code of the original application required?* Does the instantiation of the pattern require modifications to the application that go beyond

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

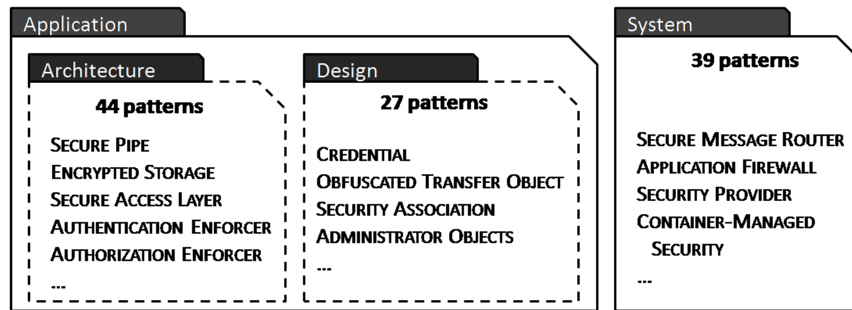

Figure 2: Classification of security patterns according to development phase.

configuration? If this is the case, then the pattern can only be instantiated during application development.

Q2 *Is the instantiated solution local?* Will a correct instantiation of the pattern remain correct, no matter how the application is extended? If the instantiation of the pattern does not necessarily remain correct, then the solution is not local, and this implies that the pattern is architectural. If the instantiation of the pattern remains correct, then the solution is local and the pattern is a security design pattern.

As shown in Figure 2, the 110 core patterns are classified as 39 system patterns (35.5% of the core patterns), 44 architectural (40.0%) and 27 design patterns (24.5%).

Concerning the execution of the classification algorithm, the main point of disparity was on the distinction between the system and architectural level. In total, the reviewers were not unanimous on 14 patterns (12.7% of the set of core patterns).

2.2 Security objective classification

Clearly, no single pattern can solve all security problems at once. Each pattern contributes to the achievement of one or more security objectives (or goals). In this work, the associations between the core patterns and the security objectives they try to achieve are made explicit, facilitating the selection of the right pattern for implementing the security requirements. An initial exploration of the relation between security patterns and security objectives has also been performed by Schumacher [Sch02]. Note that security requirements are typically linked to security objectives that must be achieved. As an example, a typical requirement would be formulated as: "To preserve the *confidentiality* of customer data, only authenticated users can view their own subscription information."

An initial, rough analysis of the association between a pattern and its security objectives was already presented in [YHSJ08]. While the set of objectives from that analysis (Confidentiality, Integrity, Accountability, Availability, Privacy, Authorization, and Authentication) is sufficient to cover all patterns, it lacks the granularity for using the patterns in a constructive way. This requires a more fine-grained assignment of objectives to patterns. For example, patterns that solve the problem of securely storing confidential data, as well as patterns

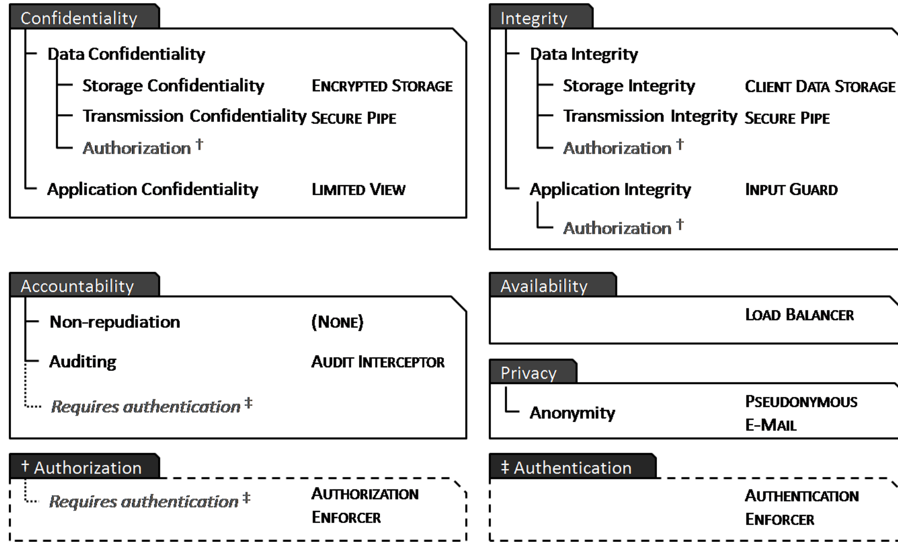

Figure 3: Decomposition of security objectives with example patterns.

for transmitting confidential data, exist. Although both problems are a confidentiality problem, storage and transmission are two distinct aspects of confidentiality. By using a more fine-grained classification, the number of potentially applicable security patterns can be reduced significantly for any given security problem. To accomplish this, a refinement of the list of security objectives is needed.

Refining an abstract objective (or goal) into more concrete sub-objectives is a well-known technique in goal-based requirements engineering methods such as KAOS [LDM95]. For security as a non-functional requirement, a decomposition of security concepts has been proposed by Chung et al. in [CNYM99]. That decomposition, however, is not constructive enough from a software engineering standpoint.

2.2.1 Method and results

Since there is no universal list of security objectives, the authors compiled a list from various sources. The proposed set is a restructuring of the traditional CIAA objectives (confidentiality, integrity, availability, and accountability), the trees of objectives suggested by Firesmith [Fir04], and the flat list mentioned by Menezes et al. [MVV96] (from which low-level mechanisms such as time stamping and certification were omitted). The result is the graph shown in Figure 3, together with an example pattern for each objective. It should be noted that constructing such a list is not straightforward, as all the sources use slightly different terminology and structuring.

The objectives are presented in an interconnected graph. The graph has the abstract security objectives (confidentiality, integrity, accountability, availability, and privacy) as its root nodes, and finer-grained objectives (e.g., data confidentiality and non-repudiation) as child nodes. At the lowest level, supporting objectives (authorization and authentication) can be found. These can

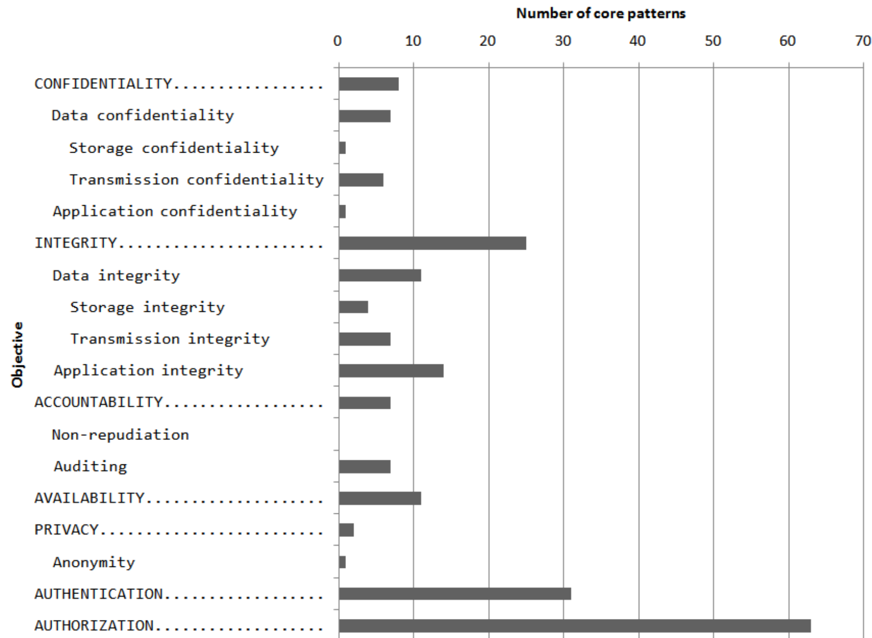

Figure 4: Number of core patterns per objective.

also be shared by different top-level objectives, and are therefore shown separately at the bottom. As an example, in [Fir04], access control is considered to be a top-level objective. In the classification proposed in this work, however, access control (labeled 'authorization') is a supporting objective for 'data confidentiality', 'data integrity', and 'application integrity'. This, in the authors' view, more accurately represents the actual role of authorization.

Starting from the set of objectives, the core patterns are assigned to the objectives by studying the detailed descriptions of the patterns. Each pattern may have multiple objectives associated with it. The same procedure as described in [YHSJ08] was used for the assignment, i.e., three independent reviews with a decision based on majority.

In Figure 4 the result of the assignment is shown. Note that the list is hierarchical; a pattern on a sub-level is counted again in its parent objective. In this way, a cumulative overview per objective is obtained. Also, the total number of patterns contained in the graph is larger that the size of the core set, due to the occasional attribution of multiple objectives per pattern. A short description of the objectives follow.

Confidentiality includes both confidentiality of data (e.g., a person's salary) and confidentiality of details concerning the application (e.g., the usage of a specific algorithm). The latter, dubbed *application confidentiality*, is somewhat related to 'security through obscurity'. Whether or not this is a good practice is out of scope of this work. *Data confidentiality* is achieved through three strategies: *storage confidentiality*, *transmission confidentiality*, and *authorization* (see later).

Integrity, similarly to confidentiality, contains both data and application integrity. *Data integrity* can be achieved through *storage integrity*, *transmission integrity*, and *authorization*. *Application integrity*, on the other hand, ensures that the application is always in a consistent state and produces the expected results. *Authorization* is a possible strategy to realize application integrity as well, e.g., by preventing unauthorized plugins to load.

Accountability contains strategies such as *non-repudiation* and *auditing*. For accountability purposes, some form of *authentication* must be performed.

Availability considers strategies for keeping a system accessible and responsive to its users.

Privacy —the right to be informed about the flow of information about oneself, and to control this flow—adds an extra dimension of purpose to confidentiality. Depending on the purpose of an action, executing the action may or may not be allowed (and certain obligations, such as a notification, may have to be fulfilled as well). Often cited together with privacy is *anonymity*, namely the state of being unidentifiable within a group of entities.

As mentioned before, some of the above objectives are using (directly or indirectly) one or more supporting objectives. The latter are never goals on their own, rather they are used in the achievement of other security objectives.

Authorization, in general, requires that the entity trying to perform the action has been authenticated in some way.

Authentication happens when a user claims to have a particular identity, and this claim is subsequently validated by some means.

The benefits of assigning patterns to security objectives are threefold. First, it eases the selection of a pattern by providing guidance about the strategy to follow, because of the decomposition and dependencies in the graph—for example, the decomposition of confidentiality in, among others, authorization, which depends on authentication. Second, it restricts the number of candidate patterns (e.g., a pattern for storage integrity is certainly not applicable when a solution for a transmission confidentiality problem is needed). Finally, the classification gives insight into the current coverage of the objectives by the core security patterns, i.e., the analysis performed in [YHSJ08] can be specialized for the restricted set of core patterns only.

2.3 Quality trade-off labels

Security patterns contribute, in general, to the achievement of one or more main security objectives. 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 2.2, other (non-security) qualities

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

Dependability	Manageability	Confidentiality
Portability	Auditability	Integrity
Maintainability	Cost	Availability
Performace		Accountability
Usability		Privacy

Figure 5: Quality labels

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). The list of proposed quality labels can be found in Figure 5. 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 (all traffic has to be checked) and dependability (if it fails, everything behind the firewall becomes unreachable).

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 with 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.

Although the quality labels are useful on their own, i.e., by giving a concise overview of the (non-functional) consequences of the pattern, their true value will show when they are used for trading off between different patterns, as prescribed by the methodology in Section 3.

2.4 Inter-pattern relationships

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 each other 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 and apply them to the set of security patterns.

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 were proposed in the area of feature interactions in the telecommunication domain (see for

example [STJ^+06]).

2.4.1 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.

2.4.2 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.

2.4.3 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 behavior 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.

2.4.4 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 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.

2.4.5 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. A methodology for using this system will be presented in Section 3.

2.5 Security pattern template

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.

Our template is shown in Figure 6. 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 document.

The template consists of two parts. We would, in particular, like to draw attention to the first part of the template (see the bold text in Figure 6), which provides a short overview of the pattern. This overview collects entries that enable a quick judgment 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, these 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

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 6: Security Pattern Template.

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 here for completeness.

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 2.4.

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

The template provides two additional values compared to the templates mentioned in other publications (e.g., [KETEH02, SNL05]), 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.

2.6 Conclusion

We provide an inventory for security patterns, which classifies the patterns among several dimensions. First, the patterns are classified according to the development phase (application architecture, application design or system) they belong to. This reduces the number of candidate patterns at any time to roughly one third. Second, the patterns are assigned suitable security objectives. This

Figure 7: The waterfall model.

enables the discarding of most irrelevant patterns for a given security problem. Third, patterns are assigned quality labels which make it easier to trade off different patterns based on non-functional requirements. Finally, the patterns are related to one another through five relationships, providing the software engineer (positively or negatively) related patterns for each chosen pattern. All patterns are also uniformally described using a template that accelerate the selection process.

Incidentally, all 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 and security problem. 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).

3 Secure development with patterns

Given the structured inventory of security patterns from the previous section, creating a security solution based on the patterns already becomes more manageable. However, due to the large amount of available metadata now associated to the patterns, it is not yet clear how to make use of this metadata in a systematic and efficient way. Therefore, this section will formulate a methodology that is designed for this very purpose.

The proposed methodology aims at reducing the effort of searching for and selecting the appropriate set of security patterns during the development of a secure application. It provides a systematic and practical strategy to arrive at a solution covering the security requirements of the application, by using security patterns. Note that this methodology does not need to be followed to the letter; at any time the software engineer may of course decide to take a different path, which might be more applicable in his situation, or apply a certain optimization or shortcut given the circumstances he finds himself in. In these cases, however, caution must be exercised to ensure that no important aspects, which are normally taken care of by the methodology (especially inter-pattern relationships), are overlooked.

In Section 3.1, a high-level overview of the methodology will be given. This bird's-eye view provides an overall rationale and feeling of the methodology, but lacks an exhaustive description. The omitted details and corner cases will then be dealt with in the detailed description in Section 3.2.

3.1 A brief overview of the methodology

For the description of the methodology, we employ a waterfall-like development model (Figure 7). It is clear that this is, at its best, only a rough approximation of development models in practice; the waterfall model is rarely, if ever, followed

to the letter. However, the proposed methodology is also applicable in other, iterative development processes. The choice of the waterfall model enables us to focus on the intricacies of the methodology, rather than being distracted by development process issues. Note that 'design' in Figure 7 comprises both the architectural as well as the detailed design phase.

The following description assumes that both the application and the environment in which the application will be deployed are created in parallel. In case either the application or the environment already exists, the proposed methodology is still applicable. In case of an existing *environment*, it is necessary to create a list of patterns that are, in some form, already instantiated in the existing environment. The patterns in this list must be considered as already instantiated patterns for the methodology. In the case of an existing *application*, the converse process can be followed. This allows for the methodology to resolve security requirements that are not yet implemented by the application itself.

Analysis phase In the analysis phase, the usual analysis steps should take place, i.e., create the domain model and elicit the functional requirements of the system. The security requirements should also be elicited, using any suitable requirements engineering method. Possible methods include misuse cases[SO05], goal-oriented requirements engineering like KAOS [LDM95] or Tropos [BPG$^+$04], or others. Before continuing, each requirement should be annotated with the security objective it belongs to. These objectives can be found in Figure 3. Note that the success of the methodology depends on a complete and correct list of security requirements.

Within the application space, the (detailed) design phase should be entered after the completion of the architecture phase. As an alternative (or complementary), a solution in the system space can be sought after. The decision to implement a requirement in the application space or in the system space is highly dependent on the specific situation, and is influenced by variables such as the nature of the application, the degree of control over the application, the infrastructure that is already in place, or the experience of the development and deployment team. In general, applying a solution in the application space is preferable because the solution can be more easily tailored for the application, and implementing fine-grained policies become possible. Of course, because of the Defense in Depth-principle[3], a solution in both the application and system space is superior to a solution in either space on itself.

Architecture phase In the architecture phase, the architecture of the system is designed using the principles of Attribute-Driven Design (ADD) [BCK03]. That means that the architecture of the application is created based on the non-functional requirements of the application, and their relative importance. To this aim, a prioritized list of quality attributes should be compiled. Possible attributes are given in Figure 5. This list will later be used to perform trade-offs between multiple suitable patterns.

Next, the following steps are repeated for each security requirement and its associated security objective that is to be solved in the architectural phase. For

[3] https://buildsecurityin.us-cert.gov/daisy/bsi/articles/knowledge/principles/347.html

the order in which the requirements are executed, consult the detailed description in Section 3.2.

1. Check if a previously instantiated pattern (or other solution) did not already sufficiently fulfill the current security requirement. If so, continue with the next requirement.

2. Select the set of security patterns from the inventory that are applicable in the architecture phase, and that include the current security objective in their objectives list.

3. From the patterns selected in the previous step, remove the patterns that conflict with any previously instantiated pattern (architectural or system), if any.

4. Screen the security patterns remaining after the previous step based on their short description header. Only retain applicable patterns.

5. Select and instantiate one or more of the applicable patterns until the security requirement has been resolved. This final selection in the first place employs matches of the quality labels of the pattern with the prioritized list, the relations of the pattern with already instantiated patterns and other information found in the description header. If the pattern deems appropriate, or when in doubt, the detailed pattern description can be consulted.

   If there is an impairment relationship between an already instantiated pattern and the selected pattern, make sure to handle this. If, after successful instantiation of the pattern, the requirement was entirely fulfilled, mark the requirement as resolved.

6. Check for dependency and benefits-relations of the instantiated pattern(s), and implement the related patterns if required and appropriate (i.e., the related pattern is also an architectural pattern). Select but do not implement non-architectural dependencies.

Design phase The design phase follows the same principles as the architectural phase. First, however, the relations between architectural and design security patterns must be taken into account. Therefore, select all unresolved dependencies from the instantiated architectural patterns to design patterns and instantiate those patterns.

Next, repeat the following steps for each unresolved or insufficiently resolved security requirement and its associated security objective that is to be solved in the design phase.

1. Check if a previously instantiated pattern (or other solution) did not already sufficiently fulfill the current security requirement. If so, continue with the next requirement.

2. Select the set of security patterns from the inventory that are applicable in the design phase, and that include the current security objective in their objectives list.

3. From the patterns selected in the previous step, remove the patterns that conflict with any previously instantiated pattern (architectural, design or system), if any.
4. Screen the security patterns remaining after the previous step based on their short description header. Only retain applicable patterns.
5. Select and instantiate one or more of the applicable patterns until the security requirement has been resolved. This final selection may utilize the detailed pattern description, the matching of the quality labels of the pattern with the prioritized list, or the relations of the pattern with already instantiated patterns. If there is an impairment relationship between an already instantiated pattern and the selected pattern, make sure to handle this. If, after successful instantiation of the pattern, the requirement was entirely fulfilled, mark the requirement as resolved.
6. Check for dependency and benefits-relations of the instantiated pattern(s), and implement the related patterns if required or appropriate.

System As mentioned before, solutions in the system space can be constructed in parallel with the architecture and design phase. It can also be tackled after the others have been completed, or it can be the only considered possibility, e.g., when deploying an application that cannot be modified or extended.

The steps in the system space are similar to the previous steps. For every requirement that has to be solved in the system space, perform the following steps.

1. Check if a previously instantiated pattern (or other solution) did not already sufficiently fulfill the current security requirement. If so, continue with the next requirement.
2. Select the set of security patterns from the inventory that are applicable in the system space, and that include the current security objective in their objectives list.
3. From the patterns selected in the previous step, remove the patterns that conflict with any previously instantiated pattern (architectural, design or system), if any.
4. Screen the security patterns remaining after the previous step based on their short description header. Only retain applicable patterns.
5. Select and instantiate one or more of the applicable patterns until the security requirement has been resolved. This final selection may utilize the detailed pattern description, the matching of the quality labels of the pattern with the prioritized list, or the relations of the pattern with already instantiated patterns. If there is an impairment relationship between an already instantiated pattern and the selected pattern, make sure to handle this. If, after successful instantiation of the pattern, the requirement was entirely fulfilled, mark the requirement as resolved.
6. Check for dependency and benefits-relations of the instantiated pattern(s), and implement the related patterns if required or appropriate.

3.2 Detailed description

In the detailed description, we will use the following notation.

- $\mathcal{R}$ is the full set of security *requirements*. $\mathcal{R}_{App}$ is the subset of security requirements that will be fulfilled in the application space. Similarly, we define $\mathcal{R}_{Sys}$ for requirements that will be fulfilled in the system space.
- $\mathcal{R}_f$ is the set of *fulfilled* security requirements.
- $\mathcal{P}$ is the full security pattern inventory. We define three subsets thereof. $\mathcal{P}_{Arch}$, $\mathcal{P}_{Des}$ and $\mathcal{P}_{Sys}$ are the sets of security patterns in the inventory on the architectural, design and system level respectively.
- $\mathcal{S}$ is the set of currently *selected* candidate patterns.
- $\mathcal{I}$ is the set of already *instantiated* patterns.
- $\mathcal{C}$ is the set of patterns that *conflict* with one or more patterns from $\mathcal{I}$. If we denote the set of patterns conflicting with a pattern p as $C(p)$, then

$$\mathcal{C} = \bigcup_{p \in \mathcal{I}} C(p).$$

- $\mathcal{B}$ is the set of patterns that is referenced by a pattern from $\mathcal{I}$ through a *benefits* relation. Using $B(p)$ as the set of benefits for pattern p,

$$\mathcal{B} = \bigcup_{p \in \mathcal{I}} B(p).$$

- $\mathcal{I}m$ is the set of patterns that is referenced by a pattern from $\mathcal{I}$ through an *impairment* relation. Using $Im(p)$ as the set of impairments for pattern p,

$$\mathcal{I}m = \bigcup_{p \in \mathcal{I}} Im(p).$$

- $\mathcal{D}$ is the set of patterns that is referenced by a pattern from $\mathcal{I}$ through a *dependency* relation, and that have not yet been instantiated. We use $D(p)$ to denote the set of dependee patterns for p.
- $\mathcal{O}$ is the set of objectives as defined in Section 2.2. If we use $O(p)$ for the set of security objectives for pattern p, then $\forall p \in \mathcal{P} : O(p) \subset \mathcal{O}$.
- $\mathcal{Q}$ is an ordered list of quality labels, from the list in Figure 5.

Initially,

$$\mathcal{R}_f = \mathcal{R}_{App} = \mathcal{R}_{Sys} = \mathcal{S} = \mathcal{I} = \mathcal{C} = \mathcal{B} = \mathcal{D} = \mathcal{I}m = \emptyset$$

If the application will be deployed in an existing environment, check whether patterns from $\mathcal{P}_{Sys}$ are implemented in that environment. If so, add them to $\mathcal{I}$. For example, if the existing environment already contains an instance of the FIREWALL pattern, add this pattern to $\mathcal{I}$.

In the **analysis phase**, each requirement r needs to be associated with a security objective $o_r \in \mathcal{O}$.

Then, for each requirement, determine whether the requirement is to be solved in the application or system space. Put the former in $\mathcal{R}_{App}$ and the latter in $\mathcal{R}_{Sys}$.

Finally, an ordered list of quality requirements $\mathcal{Q}$ is necessary. This list consists of the labels in Figure 5, ordered by their priority in the application that is being developed. This list will be used for performing trade-offs.

In the **architecture phase**, a partial ordering on $\mathcal{R}_{App}$ should be defined, using any suitable operator. A good candidate is the relative importance of the requirement as determined by the business stakeholders. How to determine this importance is not within the scope of this deliverable. Then, perform the following steps for each requirement r in $\mathcal{R}_{App}$.

1. Check if r is already sufficiently fulfilled in the current architecture. If this is the case, skip all the following steps and proceed with the next requirement.

2. Find all patterns from $\mathcal{P}_{Arch}$ that include o_r in their associated objectives. That is,

$$\mathcal{S} = \{p \in \mathcal{P}_{Arch} \mid o_r \in O(p)\}.$$

3. Remove all patterns from $\mathcal{S}$ that appear in $\mathcal{C}$, either directly or though (recursively) following the dependency links.

4. Scan the description of the patterns in $\mathcal{S}$ (primarily the header) to reduce $\mathcal{S}$ to the set of patterns that, at first sight, are appropriate for fulfilling r.

5. Select the most favorable pattern p for instantiation from $\mathcal{S}$. The following factors can make a pattern more favorable:

   - the detailed description of the pattern;
   - a match of the pattern's labels with one or more important qualities for the application, found in $\mathcal{Q}$;
   - the occurrence of the pattern in $\mathcal{B}$;
   - the absence of the pattern in $\mathcal{I}m$.

6. Instantiate p. If $p \in \mathcal{I}m$, make sure to resolve the problems when instantiating. Note that, when instantiating, one the following situations will occur.

   - $p \notin \mathcal{I}$: p was never instantiated in this application. A new instantiation is necessary.
   - $p \in \mathcal{I}$: p was already instantiated in this application. If this instantiation also covers r, no new instantiation is needed. Otherwise, create a new instantiation of p.

7. Add p to $\mathcal{I}$. Add all its conflicts $C(p)$ to $\mathcal{C}$, all its benefits $B(p)$ to $\mathcal{B}$ and all dependencies $D(p)$ to $\mathcal{D}$.

8. Instantiate all patterns in $\mathcal{D}$ by repeating step 6–8 for each pattern p in $\mathcal{D}$. Remove p from $\mathcal{D}$ when done.

9. If r is fulfilled by the patterns in $\mathcal{I}$, add r to the fulfilled requirements set $\mathcal{R}_f$ and continue with the next requirement.

If, at any point, $\mathcal{S}$ becomes empty, there are no suitable security patterns for the current situation. In this case, proceed with one of the following steps:

- Determine whether the requirement can be solved in the system space, and move it to $\mathcal{R}_{Sys}$ if possible.
- If the set became empty because the removal of patterns conflicting with patterns in $\mathcal{I}$, try backtracking on one of the conflicting patterns to see if an alternative solution is available.
- Defer the requirement to the design phase.

In the **design phase**, first instantiate any patterns in $\mathcal{D}$ that belong to the design phase, i.e., repeat steps 6–8 from the architecture phase for each pattern $p \in \mathcal{D} \cap \mathcal{P}_{Des}$.

Then, for each non-fulfilled requirement $r \in \mathcal{R}_{App} \backslash \mathcal{R}_f$, perform the following steps (similar to the architecture phase).

1. Check if r is already sufficiently fulfilled in the current design. If this is the case, skip all the following steps and proceed with the next requirement.
2. Find all patterns from $\mathcal{P}_{Des}$ that include o_r in their associated objectives. That is,
$$\mathcal{S} = \{p \in \mathcal{P}_{Des} \mid o_r \in O(p)\}.$$
3. Remove all patterns from $\mathcal{S}$ that appear in $\mathcal{C}$, either directly or though (recursively) following the dependency links.
4. Scan the description of the patterns in $\mathcal{S}$ (primarily the header) to reduce $\mathcal{S}$ to the set of patterns that, at first sight, are appropriate for fulfilling r.
5. Select the most favorable pattern p for instantiation from $\mathcal{S}$. The following factors can make a pattern more favorable:
   - the detailed description of the pattern;
   - a match of the pattern's labels with the important qualities for the application, found in $\mathcal{Q}$;
   - the occurrence of the pattern in $\mathcal{B}$;
   - the absence of the pattern in $\mathcal{I}m$.
6. Instantiate p. If $p \in \mathcal{I}m$, make sure to resolve the problems when instantiating. Note that, when instantiating, one the following situations will occur.
   - $p \notin \mathcal{I}$: p was never instantiated in this application. A new instantiation is necessary.
   - $p \in \mathcal{I}$: p was already instantiated in this application. If this instantiation also covers r, no new instantiation is needed. Otherwise, create a new instantiation of p.

7. Add p to $\mathcal{I}$. Add all its conflicts $C(p)$ to $\mathcal{C}$, all its benefits $B(p)$ to $\mathcal{B}$ and all dependencies $D(p)$ to $\mathcal{D}$.

8. Instantiate all patterns in $\mathcal{D}$ by repeating step 6–8 for each pattern p in $\mathcal{D}$. Remove p from $\mathcal{D}$ when done.

9. If r is fulfilled by the patterns in $\mathcal{I}$, add r to the fulfilled requirements set $\mathcal{R}_f$ and continue with the next requirement.

If, at any point, $\mathcal{S}$ becomes empty, there are no suitable security patterns for the current situation. Similar to the architecture phase, proceed with one of the following steps:

- Determine whether the requirement can be solved in the system space, and move it to $\mathcal{R}_{Sys}$ if possible.

- If the set became empty because the removal of patterns conflicting with patterns in $\mathcal{I}$, try backtracking on one of the conflicting patterns to see if an alternative solution is available.

After, or in parallel with, the application space, the requirements in the **system space** $\mathcal{R}_{Sys}$ have to be implemented.

For each non-fulfilled requirement $r \in \mathcal{R}_{Sys}$, perform the following steps (similar to the steps of the architecture and design phase).

1. Check if r is already sufficiently fulfilled in the current system. If this is the case, skip all the following steps and proceed with the next requirement.

2. Find all patterns from $\mathcal{P}_{Sys}$ that include o_r in their associated objectives. That is,
$$\mathcal{S} = \{p \in \mathcal{P}_{Sys} \mid o_r \in O(p)\}.$$

3. Remove all patterns from $\mathcal{S}$ that appear in $\mathcal{C}$, either directly or though (recursively) following the dependency links.

4. Scan the description of the patterns in $\mathcal{S}$ (primarily the header) to reduce $\mathcal{S}$ to the set of patterns that, at first sight, are appropriate for fulfilling r.

5. Select the most favorable pattern p for instantiation from $\mathcal{S}$. The following factors can make a pattern more favorable:

   - the detailed description of the pattern;
   - a match of the pattern's labels with the important qualities for the application, found in $\mathcal{Q}$;
   - the occurrence of the pattern in $\mathcal{B}$;
   - the absence of the pattern in $\mathcal{I}m$.

6. Instantiate p. If $p \in \mathcal{I}m$, make sure to resolve the problems when instantiating. Note that, when instantiating, one the following situations will occur.

   - $p \notin \mathcal{I}$: p was never instantiated in this application. A new instantiation is necessary.

- $p \in \mathcal{I}$: p was already instantiated in this application. If this instantiation also covers r, no new instantiation is needed. Otherwise, create a new instantiation of p.

7. Add p to $\mathcal{I}$. Add all its conflicts $C(p)$ to $\mathcal{C}$, all its benefits $B(p)$ to $\mathcal{B}$ and all dependencies $D(p)$ to $\mathcal{D}$.

8. Instantiate all patterns in $\mathcal{D}$ by repeating step 6–8 for each pattern p in $\mathcal{D}$. Remove p from $\mathcal{D}$ when done.

9. If r is fulfilled by the patterns in $\mathcal{I}$, add r to the fulfilled requirements set $\mathcal{R}_f$ and continue with the next requirement.

If, at any point, $\mathcal{S}$ becomes empty, there are no suitable security patterns for the current situation. Proceed with one of the following steps:

- Determine whether the requirement can be solved in the application space, and move it to $\mathcal{R}_{Arch}$ if possible.

- If the set became empty because the removal of patterns conflicting with patterns in $\mathcal{I}$, try backtracking on one of the conflicting patterns to see if an alternative solution is available.

3.3 Conclusion

Given the structured inventory of security patterns, creating a security solution based on the patterns already becomes more manageable. However, due to the large amount of available metadata now associated to the patterns, a methodology is proposed to make use of this metadata in a systematic and efficient way.

The described methodology reduces the effort of searching for and selecting the appropriate set of security patterns during the development of a secure application. It provides a systematic and practical strategy to arrive at a solution covering the security requirements of the application by using security patterns, taking into account the properties of the patterns and the relationships between them.

It is apparent from the description that the methodology could benefit greatly from tool support. Indeed, in the first place an automatic selection of patterns from the right development phase and security objectives is helpful. Furthermore, the relationships between patterns provide a further possible path than can be pursued by tool support, especially when combined with a history of instantiated patterns. Using the tool, full traceability information (why a specific pattern was chosen and implemented) would become available and easily accessible.

A Patterns description

For each pattern mentioned in the text, a short description, as well as a reference to the original source of the pattern, is provided.

Administrator Objects [KBZ01]
: In RBAC environment, user-role assignment and role-privileges assignment is the major administrative task. Each user has a unique subject that describes the user's permitted roles and user must activate roles associated with his subject to access information. This pattern creates subjects for users and delegates administrative responsibilities.

Application Firewall [DGFRLP04]
: This pattern filters calls and responses to/from enterprise applications, based on an institution access control policies. It does this by interposing a firewall that can analyze incoming requests for application services and check them for authorization.

Audit Interceptor [SNL05]
: To intercept and audit requests to the business tier, use an audit interceptor which centralizes the auditing functionality and enables declarative audit event definitions.

Authentication Enforcer [SNL05]
: To reduce authentication code duplication and allow for easy changes to the authentication mechanism, create a centralized authentication enforcer that performs authentication of users and encapsulates the details of the authentication mechanism.

Authorization Enforcer [SNL05]
: To verify that each request is authorized, create an access controller that will perform authorization checks using standard mechanisms.

Client Data Storage [KETEH01]
: It is often desirable or even necessary for a Web application to rely on data stored on the client, using mechanisms such as cookies, hidden fields, or URL parameters. In all cases, the client cannot be trusted not to tamper with this data. The Client Data Storage pattern uses encryption to allow sensitive or otherwise security-critical data to be securely stored on the client.

Container Managed Security [SNL05]
: 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. It employs declarative security that requires the developer to only define roles at a desired level of granularity through deployment descriptors.

Credential [MF06]
: Credential provides secure portable means of recording authentication and authorization information for use in distributed systems.

Credential Tokenizer [SNL05]
: A Credential Tokenizer encapsulates different types of user credentials as a security token that can be reusable across different security providers.

Demilitarized Zone [SFBH+06]
: The Demilitarized Zone pattern introduces a region of the system that is

separated from both the external users and the internal data and functionality. This region will contain the servers, such as Web servers, that expose the functionality of the Web-based application. Restrict access to this region from the outside by limiting network traffic flow to certain physical servers. Use the same techniques to restrict access from servers in the DMZ to the internal systems.

Encrypted Storage [KETEH01]
: The Encrypted Storage pattern provides a second line of defense against the theft of data on system servers. Although server data is typically protected by a firewall and other server defenses, there are numerous publicized examples of hackers stealing databases containing sensitive user information. The Encrypted Storage pattern ensures that even if it is stolen, the most sensitive data will remain safe from prying eyes.

Firewall [Sch03]
: To prevent attackers from accessing the internal network, restrict the ingoing and outgoing traffic at the border between the internal and external network.

Full View With Errors [YB97]
: To prevent users from performing illegal operations, design the application so users see everything that they might have access to. When a user tries to perform an operation, check if it is valid. Notify them with an error message when they perform illegal operations.

Input Guard [Sar03]
: To stop the propagation of an error from the outside to the inside of a component, place a guard at every access point of the component to check the validity of the input.

Keep Session Data in the Server [Sor02]
: Session specific data has to be stored in between requests, and made available to the code handling a request. Keep all session speci

c data on the server. Assign a unique token to each session, and create the protocols used in communication between users and system so that this token is made part of every interaction. Use this token as a key into the data structure in the server that holds the session speci

c data for all clients.

Limited View [YB97]
: To prevent users from performing illegal operations, hide the operations to which a user does not have access.

Load Balancer [Sor02]
: Implement a system through which all requests pass and that directs them to the server instance that should handle them.

Obfuscated Transfer Object [SNL05]
: To protect critical data as it is passed between tiers, use an obfuscated transfer object to protect access to the data.

Pseudonymous E-mail [Sch02]
: Despite the desire for anonymity users may be required to authenticate to the email service, e.g., to receive answers to email messages later. The question is how to use an email service without revealing your own identity?

Secure Access Layer [YB97]
: To integrate application security with the security of external systems, correctly use the existing security mechanisms. If these do not exist, build your own access layer.

Secure Logger [SNL05]
: To securely log application events and data, use a centralized logger that obstructs log message alteration, deletion or loss.

Secure Message Router [SNL05]
: To securely communicate with multiple endpoints using message-level security, establish a security intermediary infrastructure that aggregates access to multiple application endpoints. Dynamically provide security logic for routing messages to multiple endpoint destinations.

Secure Pipe [SNL05]
: Use a secure pipe to guarantee the integrity and confidentiality of data sent over a wire.

Security Association [BHmoTOGSF04]
: To avoid reinitializing secure communication for every message, define a structure to hold the information used to protect and verify the messages so the secure communication instance can be reused.

Security Provider [Rom01]
: A Security Provider is a central service to which are directed all authentication and authorization requests. Applications such as email, web, corporate applications and others, would communicate directly with the Security Provider. The Security Provider then communicates with a user or policy store to evaluate a user's credentials and privileges.

Session [YB97]
: Create a session object that holds the variables and state specific for a certain user. Pass this session object around to objects that need one if its values.

Single Access Point [YB97]
: A security model is difficult to validate when it has multiple "front doors", "back doors", and "side doors" for entering the application. Set up only one way to get into the system, and if necessary, create a mechanism for deciding which sub-applications to launch.

References

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

[BHmoTOGSF04] Bob Blakley, Craig Heath, and members of The Open Group Security Forum. Security design patterns. `http://www.opengroup.org/security/gsp.htm`, 2004.

[BPG+04] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. Tropos: An Agent-Oriented Software Development Methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3):203–236, 2004.

[CNYM99] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering*. Springer, 1999.

[DGFRLP04] Nelly Delessy-Gassant, Eduardo B. Fernandez, Sajeed Rajput, and Maria M. Larrondo-Petrie. Patterns for application firewalls. In *Pattern Languages of Programs Conference (PLoP 2004)*, 2004.

[EK03] Amnon H. Eden and Rick Kazman. Architecture, design, implementation. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, Washington, DC, USA, 2003. IEEE Computer Society.

[Fir04] Donald Firesmith. Specifying reusable security requirements. *Journal of Object Technology*, 3(1):61–75, 2004.

[Gam06] Erich Gamma. Design patterns – 15 years later. In *European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, July 2006.

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

[KBZ01] Saluka R. Kodituwakku, Peter Bertok, and Liping Zhao. Aplrac: A pattern language for designing and implementing role-based access control. In *European Conference on Pattern Languages of Programs (EuroPLoP 2001)*, 2001.

[KETEH01] Darrell M. Kienzle, Matthew C. Elder, David Tyree, and James Edwards-Hewitt. Security patterns repository, version 1.0. `http://www.modsecurity.org/archive/securitypatterns/dmdj_repository.pdf`, 2001.

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

[LDM95] A. Van Lamsweerde, R. Darimont, and P. Massonet. Goal-directed elaboration of requirements for a meeting scheduler: problems and lessons learnt. In *Second IEEE International Symposium on Requirements Engineering (RE'95)*, 1995.

[MF06] Patrick Morrison and Eduardo B. Fernandez. The credential pattern. In *The Conference on Pattern Languages of Programs (PLoP 2006), Portland*, 2006.

[MVV96] A.J. Menezes, P.C. Van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Inc., 1996.

[Rom01] Sasha Romanosky. Security design patterns, part 1 v1.4. `http://www.cgisecurity.com/lib/securityDesignPatterns.pdf`, 2001.

[Sar03] Titos Saridakis. Design patterns for fault containment. In *The 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), Germany*, 2003.

[Sch02] Markus Schumacher. Security patterns and security standards – selected security patterns for anonymity and privacy. In *The 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), Germany*, 2002.

[Sch03] Markus Schumacher. Firewall patterns. In *The 8th European Conference on Pattern Languages of Programs (EuroPLoP 2003), Germany*, 2003.

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

[SO05] G. Sindre and A.L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering*, 10(1):34–44, 2005.

[Sor02] Kristian Elof Sorensen. Session patterns. In *The European Conference on Pattern Languages of Programs (EuroPLoP 2002)*, 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.

[VM02] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, 2002.

[YB97] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *The 4th Conference on Patterns Language of Programming (PLoP 1997)*, 1997.

[YHSJ08] Koen Yskout, Thomas Heyman, Riccardo Scandariato, and Wouter Joosen. Security patterns: 10 years later. Technical Report CW-514, Katholieke Universiteit Leuven, Department of Computer Science, 2008.