The Security Twin Peaks

Thomas Heyman[1], Koen Yskout[1], Riccardo Scandariato[1], Holger Schmidt[2], and Yijun Yu[3]

[1] IBBT-DistriNet, Katholieke Universiteit Leuven, Belgium
`first.last@cs.kuleuven.be`
[2] Technische Universität Dortmund, Germany
`holger.schmidt@cs.tu-dortmund.de`
[3] Open University, United Kingdom
`y.yu@open.ac.uk`

Abstract. The feedback from architectural decisions to the elaboration of requirements is an established concept in the software engineering community. However, pinpointing the nature of this feedback in a precise way is a largely open problem. Often, the feedback is generically characterized as additional qualities that might be affected by an architect's choice. This paper provides a practical perspective on this problem by leveraging architectural security patterns. The contribution of this paper is the Security Twin Peaks model, which serves as an operational framework to co-develop security in the requirements and the architectural artifacts.

Keywords: security, software architecture, requirements, patterns

1 Introduction

Often, the requirements specification is regarded as an independent activity with respect to the rest of the software engineering process. In fact, both literature and practice have pointed out that requirements cannot be specified in isolation and "thrown over the wall" to the designers and implementers of the system. In contrast, the requirements specification (describing the problem) and the architectural design (shaping a solution) are carried on concurrently and iteratively, while still maintaining the separation between the problem and solution space. This process of co-developing the requirements and the software architecture is referred to as the Twin Peaks model [22]. As depicted in Figure 1, the specification process (i.e., refinement) in the Twin Peaks model continuously jumps back and forth between the requirements and architectural peaks, in order to embrace the decisions made in the other peak.

Some work already exists that focuses on the forward transition from security requirements to software architectures [18, 11, 31, 26]. This work leverages standardized solutions, such as security patterns. These solutions are related to the security requirements via traceability links, facilitating both the selection of the right architectural solutions and documentation of the rationale for

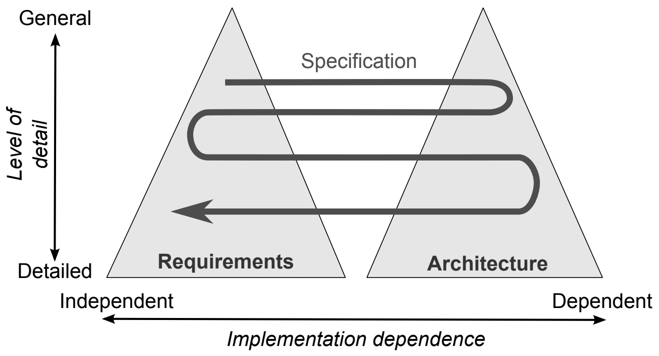

Fig. 1. The original Twin Peaks model [22]

the architectural choice [29]. This, in turn, facilitates impact analysis in face of change.

Concerning the backward transition, even if the importance of the feedback from the architecture to the requirements is an established concept in the software engineering community, the literature fails in pinpointing the nature of this feedback in a precise and operational way. This is also true for software qualities such as security. Often, the feedback is generically characterized as additional qualities, such as performance, that might be affected by a security architectural choice [28].

This paper presents an elaboration of the original Twin Peaks model in the context of security, called the Security Twin Peaks. By leveraging architectural security patterns, the model provides constructive insights in the process of specifying and designing a security-aware system, by pinpointing interaction points between the software architect's and the requirements engineer's perspective. In particular, we illustrate that an architectural security pattern actually consists of three elements that are key with regard to the Twin Peaks: (1) components supporting the security requirement by fulfilling a security functionality, (2) roles (connecting the generic solution to the specific architecture) and the expectations on such roles, and (3) residual goals. As our main contribution, we show how these elements are related to the requirements specification and can be leveraged to drive the refinement process, thereby substantiating the Security Twin Peaks model. KAOS is used to represent (security) requirements [27]. However, the presented model is not specific to the chosen requirements engineering methodology.

The rest of this paper is organized as follows. The related work is presented in Section 2. Architectural security patterns are analyzed in Section 3, in order to identify the root causes of feedback from the architectural design to the requirements specification. The Security Twin Peaks model is introduced and discussed in Section 4. Finally, Section 5 presents the concluding remarks.

2 Related Work

The problem peak in secure software engineering. In the realm of security software engineering, Haley et al. [10, 9] present a framework for representing security requirements in an application context and for both formal and informal argumentation about whether a system satisfies them. The proposed argumentation process specifies several iterative steps for the problem part of the Twin Peaks. Mouratidis et al. [19, 18, 14] present a procedure to translate the Secure Tropos models [8] into UMLsec diagrams [16]. However, they do not provide explicit feedback from the chosen architectural solution back to the requirements phase.

The solution peak in secure software engineering. Côté et al. [5] propose a software development method using problem frames for requirements analysis and using architectural patterns for the design. For the benefit of evolving systems, evolution operators are proposed to guide a pattern-based transformation procedure, including re-engineering tasks for adjusting a given software architecture to meet new system demands. Through application of these operators, relations between analysis and design documents are explored systematically for accomplishing desired software modifications, which allows for reusing development documents to a large extent, even when the application environment and the requirements change. In parallel, Hall et al. [12] propose the A-struct pattern in problem frames to explore the relationship between requirements and architectures in a problem-oriented software engineering methodology. Both work deals with feedback for general software engineering problems, but they had not focused on specific difficulties in secure software development.

Security patterns. Many authors have advanced the field of security design patterns during the last years [30, 17, 3, 24, 25, 6]. A comprehensive overview and a comparison of the different existing security design patterns can be found in [13], which establishes, among others, that the quality of the documentation of some existing security design patterns is questionable. A recent survey by Bandara et al. [1] compares various software engineering methods in application to address a concrete RBAC security pattern to reveal that there is still a need to systematically support the Security Twin Peaks by linking security requirements with security architectures. To shed more lights on the mechanisms for Secure Twin Peaks, another extensive survey on security requirements for evolving systems [21] categories the literature in terms of how an evolving system is related to its evolving requirements and changing contexts. Fernandez et al. [7] propose a methodology to systematically use security design patterns in UML activity diagrams to identify threats to the system and to nullify these threats during fine-grained design. Mouratidis et al. [20] present an approach to make use of security design patterns that connects these patterns to the results generated by the Secure Tropos methodology [8].

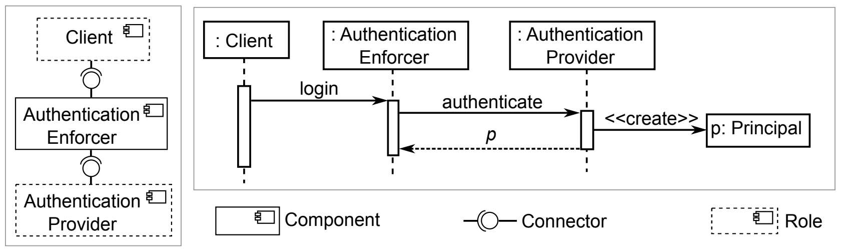

Fig. 2. Structure and behavior of the Authentication Enforcer pattern [25]

3 Architectural Security Patterns Revisited

Patterns are a well-known and recognized technique to build software architectures. This section revisits architectural security patterns and highlights the key elements that are used in Section 4 as stepping stones to link the solution domain (architectural peak) to the problem domain (requirements peak).

To illustrate the concepts, the Authentication Enforcer [25] pattern is used as an example. This pattern describes how to solve the problem of authenticating users in a systematic way by creating an authentication layer, and provides a number of different implementation alternatives to realize this layer in the architecture. One of the alternatives is the provider-based authentication method depicted in Figure 2. The solution consists of an Authentication Enforcer component that mediates all access requests originated by Clients and delegates the implementation of an authentication method to a third-party Authentication Provider component.

3.1 Key Notions for Co-development

Besides the many pieces of information that are traditionally documented in a pattern (e.g., the problem description and the known uses), we observe that an architectural security pattern can be seen, at its core, as a combination of three parts. They are:

1) Components and behavioral requirements. The participants of a pattern can be grouped into components that are newly introduced, and roles (further discussed in point 2) referring to components that are external to the pattern. A new component has a security-specific purpose, i.e., it adds new functionality to the system that is specific to a security requirement the system should uphold. This corresponds to the operationalization of a secondary functional requirement, as in Haley et al. [9]. Hence, new components introduce (finer-grained) behavioral requirements, which they fulfil and to which they can be linked, as clarified in Section 4.

Example. In the example, the Authentication Enforcer pattern introduces an Authentication Enforcer component, which encapsulates the authentication logic. This component is only needed to address the security problem statement.

2) Roles and expectations. Patterns are generic solutions that need to be instantiated in the context of concrete (possibly partial) architectures. Roles are used for that purpose. A role is a reference that needs to be mapped to a component (or sub-system) that is already present in the existing architecture. Hence, the roles provide the connective between the new components and the existing components, and define how both should interact.

Often, a pattern introduces expectations specific to its roles that need to be fulfilled by the concrete architecture. The pattern can impose constraints on both the way an external component is supposed to play a given role, as well as the way the external component interacts via the connectors with the rest of the patten internals. Hence, roles introduce finer-grained requirements in the problem domain.

EXAMPLE. The Authentication Enforcer pattern introduces two roles: the Client, which is mapped to the actual component that invokes the Authentication Enforcer component, and the Authentication Provider, which needs to be mapped to a third-party system providing an authentication mechanism. One expectation that should be realized by the Authentication Provider role is that the result of the authentication process is passed back as a Principal object. The pattern specifies the responsibilities but does not dictate how the Authentication Provider should be implemented (e.g., it does not specify the authentication mechanism). The pattern also imposes certain expectations on the interaction between the Authentication Enforcer and the Client. It suggests to protect the confidentiality of credentials, especially during transit. For instance, in a web context, the pattern suggests to avoid clear-text communication.

3) Residual goals. These are security considerations to take into account when instantiating the pattern. For instance, the pattern might make (trust) assumptions on the environment in which the system is deployed, that fall outside the scope of the solution presented by the pattern. These residual goals are not under the responsibility of either the newly introduced components or the roles.

EXAMPLE. One residual goal of the Authentication Enforcer pattern is to localize all authentication logic in the Authentication Enforcer component. Realizing this goal is out of scope of the pattern itself—it is impossible for the Authentication Enforcer pattern to enforce, in some way, that no other component contains custom authentication code. A residual goal externalizes this concern, placing the responsibility back in the hands of the software architect. Another residual goal is that it should be impossible for an attacker to obtain the user's credentials. This manifests itself in residual requirements such as "make credentials hard to forge" (e.g., implement a strong password policy) and "ensure that credentials do not leak" (e.g., store salted hashes locally, do not store the passwords in plain text).

As a final note, although this section focuses on architectural security patterns, the three parts presented above can be identified in any generic security architectural solution, irrespective from whether it is described as a pattern or not.

3.2 Revisiting the Pattern Documentation

The above three parts have been implicitly mentioned (often in a scattered way) in the literature, e.g., in the documentation of existing architectural patterns [4]. These patterns are usually documented by means of the following information [3]. The *problem and forces* describe the context from which the pattern emerged. The *solution* describes how the pattern resolves the competing forces and solves the problem. This solution consists of two parts. The *structure* of the solution depicts the different participants that play a role in the pattern, and the relationships among them. The *behavior* of the solution describes the collaborations among the different participants, by which they realize their common goals and solve the problem. Apart from the solution, a pattern should document its *consequences*, that highlight both the strengths and weaknesses of the proposed solution. Finally, an *example* of the pattern in an easily understood software setting shows how this is applied in practise.

The three elements from the previous section are often present in a general pattern description. The participants from the solution description correspond to both the newly introduced components and roles. The behavior of the solution introduces the behavioral requirements on the new components of the pattern, and possibly also expectations on the roles. The consequences of the pattern (and in particular the weaknesses) identify potential residual goals. This clearly shows that the three key notions for co-development are not so abstract and are, in most cases, already implicitly documented in existing patterns. This work contributes to the subject by bringing these three parts to the front and illustrating the primary role they play in the context of the Security Twin Peaks.

4 The Security Twin Peaks

In the previous section, we discussed the fundamental parts comprising an architectural security pattern. In this section, these concepts are leveraged to outline a constructive process for co-developing secure software architectures and security requirements. Particular focus is placed on the feedback loop between architecture and requirements, and the more subtle intricacies that need to be taken into account.

This process is not a new development process or paradigm by itself. Rather, it gives constructive guidance on what is mostly left implicit, i.e., how to interleave the requirements and architectural peaks when designing a secure software system using security patterns or other generic security solutions. Hopefully, the awareness of this process contributes to the effectiveness of requirements engineering and software architecture design.

For the solution peak, we apply an attribute-driven architectural design approach, such as Bass et al. (ADD, [2]). In attribute-driven design, non-functional concerns such as performance, maintainability, security, and so on, are referred to as quality attributes, which are orthogonal to the functionality expected of the system and drive the design of the software architecture. Qualities are realized through fundamental design decisions, referred to as tactics (a.k.a. solution

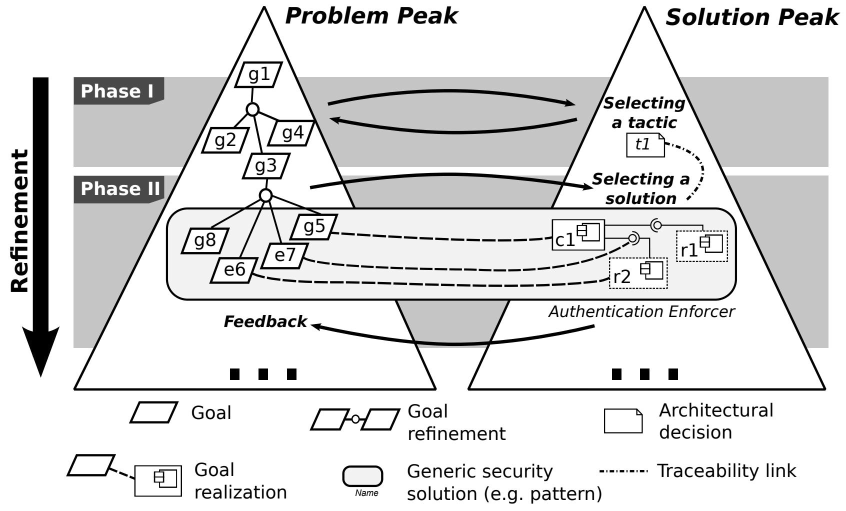

Fig. 3. The Security Twin Peaks.

strategies). An architectural pattern (or style) is a domain-independent solution to a recurring problem, which packages and operationalizes a number of tactics.

For the problem peak, we apply a goal-based requirements engineering approach, such as the one by van Lamsweerde (KAOS, [27]). In goal-based methods, goals (prescriptive statements of intent that the system should satisfy) are used for requirements elicitation, analysis and elaboration. Agents (active system components playing a specific role in goal satisfaction) achieve these goals through cooperation.

We illustrate the process on a simple application. Consider an online shop which allows customers to buy products over the Internet. Customers have an account to which costs are billed. For billing purposes, an important goal of the system is that all purchases are securely traced back to the customers.

4.1 Overview

The process is sketched in Figure 3. Table 1 complements the figure by explaining the meaning of the labels. The process progresses through 8 activities grouped in 2 phases. These phases are repeated over several iterations until the requirements specification and the architectural design are deemed as complete (i.e., detailed enough). In KAOS terminology, the goal decomposition stops once the leaf goals are realizable by software agents, i.e., a solution is selected and instantiated. In Phase I (activities 1-4), a tactic is selected to realize a goal. In Phase II (activities 5-9) a pattern is selected and instantiated. For each activity, a graphical representation is given of the current peak (with a filled triangle), and the transition between the peaks (with an arrow) where applicable. After a bird's-eye view on the whole process, each activity is described in more detail.

Table 1. The Security Twin Peaks: example.

Label	Type	Description
g_1	Goal	All purchases are securely traced
t_1	Tactic	User shall be authenticated before purchasing
g_3	Goal	An identified user shall be authenticated
g_5	Goal	Mediate requests and delegate authentication
e_6	Expectation	Return a Principal object
e_7	Expectation	Avoid clear-text communication
g_8	Residual goal	Ensure that credentials do not leak
c_1	Component	Authentication Enforcer
r_1	Role	Client
r_2	Role	Authentication Provider

Act. 1. ▲ △ Select an **initial security goal** that will be refined in this iteration of the process. In the example, goal g_1 ('all purchases are securely traced') is selected. Obviously, g_1 would be part of a larger goal tree that is not shown here.

Act. 2. △ ⤳ ▲ Choose and assess a **solution tactic** for the goal. In the example, a prevention tactic is chosen: users shall be authenticated before purchase orders can be placed (t_1). The architect (together with other stakeholders), must decide whether, for example, the assurance gained from authentication outweighs the decrease in usability that goes together with enforcing authentication.

Act. 3. ▲ ↜ △ **Refine** the goal based on the chosen tactic. In the example, this leads to the introduction of sub-goals g_2 ('users shall be identified'), g_3 ('an identified user shall be authenticated') and g_4 ('only authenticated users can purchase products'). While manual refinement by an expert is possible, a problem decomposition pattern could be associated to the tactic. The decomposition pattern can provide extra guarantees of soundness, e.g., by ensuring that the set of sub-goals is indeed complete.

Act. 4. ▲ △ Check for **conflicts** between the newly introduced and the previous goals. Resolve conflicts where possible. If a conflict cannot be satisfactorily resolved, backtrack to Activity 2 to select a different solution. For instance, the identification goal g_2 may conflict with an anonymity goal elsewhere in the goal tree, aimed at protecting the customer's privacy. The stakeholders can, for example, weaken the anonymity goal so that customers can still anonymously browse the shop, but anonymity is no longer required when an actual purchase is made.

Act. 5. ▲ △ **Select** a sub-goal introduced in the previous step (e.g., g_3) that has to be resolved using an architectural security pattern.

Act. 6. △ ⤳ ▲ Choose an **architectural security pattern** whose problem statement matches the selected sub-goal. In the example, the Authentication Enforcer pattern is chosen to resolve goal g_3. For instance, this can be a solution pattern that is linked to the used problem decomposition pattern. If no suitable solution is found, backtrack to activity 2 to select a different tactic.

Act. 7. △ ▲ **Instantiate** the architectural security pattern by performing the following activities:

(a) *Instantiate the newly introduced components* from the pattern in the architecture. In the example, an Authentication Enforcer component (c_1) is added to the architecture.

(b) Connect the new components to the rest of the architecture by *binding the roles* to concrete architectural elements. If no suitable architectural elements are present already in the architecture, create them. In the example, the Client role (r_1) is mapped to the existing component that handles purchases, and the Authentication Provider role (r_2) is mapped to a to-be-introduced component, responsible for checking the customer's credentials.

If the instantiation of the pattern becomes infeasible, backtrack to activity 6 to select a different pattern.

Act. 8. ▲ ↩ △ **Update** the requirements model so that it corresponds to the new architecture, by performing the following activities.

(a) Introduce new requirements that describe the functionality of the newly introduced components. For instance, g_5 ('mediate requests and delegate authentication').

(b) Introduce the expectations that need to be achieved by the elements that play one of the pattern's roles. For instance, e_6 ('return a Principal object') is an expectation for role r_2 and e_7 ('avoid clear-text communication') is an expectation on the connector between c_1 and r_2. Note that this list of expectations is not complete for the given example.

(c) Add the residual goals described by the pattern to the goal tree. For instance, only g_8 ('ensure that credentials do not leak') is shown in the example.

Act. 9. ▲ △ Check for **conflicts** between the newly introduced and the previous goals. Resolve conflicts where possible. If a conflict cannot be satisfactorily resolved, backtrack to activity 6 to select a different solution.

4.2 Discussion

The previous process description should be complemented with the following considerations.

ACTIVITY 1. The security goal that is selected in this activity can originate from known requirements engineering techniques, which we do not consider further in this work. Also, the order in which goals are selected for refinement is not fixed, and should be decided by the requirements engineer and the other stakeholders. It should be noted, however, that additional security goals can be introduced by Activity 8 later in the process. These goals should also be considered for selection in a next iteration.

Activity 2. In choosing the solution tactic, the focus is on determining a suitable tactic to guide the goal decomposition, possibly led by a catalog of tactics. For instance, security can be handled by preventing attacks (e.g., authenticate users), detecting attacks (e.g., intrusion detection) or recovering from an attack (e.g., using audit trails) [2]. While not having a direct manifestation in the primary architectural artifacts (at this stage), choosing a tactic does involve the architectural peak, as potential architectural constraints need to be taken into account. This is why the architect should assess whether the current architecture is able to support the considered tactic. Also, care must be taken to choose a tactic that does not conflict with the important qualities of the existing architecture. Note that it is still uncertain whether it is feasible to fulfill the goal using the chosen tactic. This only becomes apparent after a solution has been chosen and instantiated in Activity 7.

To determine the suitability of a tactic, various factors need to be taken into account and a risk assessment should be performed to decide whether the potential losses outweigh the implementation costs. For instance, the tactic can be too costly or even impossible to implement, e.g., a tactic may require full mediation, which is not supported by the current architectural environment.

Finally, notice that the selection of a tactic is an important architectural decision that belongs to the body of architectural knowledge. As shown in Figure 3, the tactic can be linked to the pattern that will be selected later on. In this respect, the tactic documents the rationale that will lead to the selection of a certain pattern and complements the rationale represented by linking the pattern and the goal it realizes.

In the online shop example, the prevention tactic is chosen: a user will be asked to authenticate before the shopping process continues, ensuring that the identity of the user is known before the billing procedure starts. An alternative tactic (while not as straightforward in the e-commerce context) would be detection and recovery: send the invoice to the address the user entered, without performing rigorous authentication first. If the bill is paid, the item gets shipped. Otherwise, the order is canceled.

Activity 3. Performing a goal decomposition based on a tactic represents the completion of a first round-trip between the problem peak and the solution peak. Note that the influence of architectural decisions on goal decomposition is mentioned in KAOS as well. In particular, in KAOS, a goal can be decomposed into several alternative branches and it is acknowledged that the selection of an alternative leads to a different architecture. In KAOS, the selection of an alternative is driven by soft goals (i.e., system qualities, development goals or architectural constraints) [26].

Activity 4 (and 9). In some cases, the new goals (resulting from a decomposition or introduced by a pattern instantiation) will fit naturally in the existing goal tree. However, conflicts may emerge and, hence, there is a need to explicitly incorporate conflict resolution in the secure development process.

Requirements engineering methodologies such as KAOS already define techniques to resolve conflicts (e.g., avoidance, restoration, anticipation or weakening) [27]. If the conflict cannot be sufficiently resolved, however, backtracking and selecting a different tactic or pattern can be considered. Of course, it can also be decided that the currently selected pattern remains in place, and the other (conflicting) part of the system is revisited.

Activity 5. The selection of a sub-goal initiates the second part of the process, where a concrete solution is chosen and instantiated. Like in Activity 1, the order of selection is left to the insight of the requirements engineer and the other stakeholders, which may mandate certain priorities.

Activity 6. The selection of the architectural pattern defines a traceability link, connecting the selected sub-goal and the pattern, i.e., the goal provides the rationale for the pattern. As mentioned before, this explicit relationship enriches the architectural knowledge.

Note also that, sometimes, a pattern may be able to solve a collection of goals simultaneously. This leads to more complex traceability links, but the outlined process can still be used.

Conversely, an architectural pattern may not be a complete match for the selected sub-goal, and can only fulfill a part of it. In this case, the goal can be additionally refined, such that one of its children match the pattern, or, alternatively, the initial refinement can be adapted.

Activity 7. By instantiating new components and connecting these to existing components, the software architecture gets refined. This refinement comes in two flavors. A pattern can extend an architecture with new components, while largely leaving the existing system untouched, as is the case with the Authentication Enforcer. As a second category of refinements, a pattern can substitute one or more components with a more refined subsystem. An example of such a pattern is the Secure Message Router [2], which can replace an existing message broker.

Of course, when instantiating the pattern, established software engineering practices need to be applied. For instance, related functionality should be grouped together. This also implies that the solution should be merged with existing components where possible. For instance, corresponding roles from different patterns can be mapped to the same component.

It can be expected that new components pose no difficulties to their introduction in the system, because they are independent from the context. Concerning the role bindings, however, more problems can arise. In some cases, it can be straightforward to select an existing component to fulfill a role, or to extend an existing component with new responsibilities. In general, however, the expectations imposed on a role might fundamentally conflict with other parts of the system. For example, consider a pattern dictating that communication between a new component and a role should be encrypted. If the element to which the role gets bound to requires that all its connections are plain-text to support auditing, then this clearly triggers a non-trivial conflict that prevents the pattern from being instantiated correctly. In general, it can be observed that conflicting expectations are the root cause of conflict among patterns, which are informally documented in pattern catalogs. Similarly, pattern languages (such as [30]) contain a cohesive set of patterns resolving each others residual goals as much as possible, while not introducing conflicts.

Activity 8. There are three distinct types of feedback, arising from the three parts of an architectural security pattern described in the Section 3. Each type of feedback introduces new elements in the requirements model.

The first type of feedback is the addition of new requirements assigned to the newly introduced components from the security pattern. These requirements describe the functionality of the new components and are necessary to ensure that all behavior implemented in the system is traceable to some requirement.

The second type of feedback is the set of expectations (i.e., constraints) imposed on the elements that play one of the roles from the pattern. It is then up to the architect to iterate over the architecture and assess whether the expectations (1) are already met by the component and the connectors involved in the corresponding role (e.g., it might be that the web server hosting the shop already supports SSL in order to securely transmit user credentials) or (2) require a refinement of these architectural elements so that they meet the expectations.

The third type of feedback is the set of residual goals. These goals are not assigned to a concrete element, because it is unspecified (from the pattern perspective) which element is responsible for them, or even how to achieve them. Therefore, they serve as candidate initial goals for refinement in a next iteration of the process. Obviously, it could be the case that the residual goals are already met by some sub-system of the architecture as is. In summary, the unbounded responsibility associated to residual goals distinguishes them from the previous type of feedback.

The goals generated by the feedback need to be reconciled with the goal tree. All goals introduced in this activity are prescribed by the pattern, and are necessary to ensure its correct functioning. As the pattern itself was selected to achieve the sub-goal selected in Activity 5 (e.g., g_3 in the example), it is natural to expect that the feedback goals need to be inserted as children of that sub-goal.

5 Conclusion

This paper has presented an elaboration of the Twin Peaks model, specific to co-developing secure software architectures and security requirements using patterns. The precise interaction points between architectural design and requirements engineering (in the context of software security) have been identified by decomposing the instantiation of an architectural security pattern into the interwoven process of (a) introducing new components and binding existing components to roles, and (b) introducing security behavioral requirements, expectations and residual goals. By pinpointing these interaction points, it is easier for the security-minded requirements engineer and software architect to predict where feedback might arise during the development process, and identify its root cause. Furthermore, this model can be leveraged to build more robust secure software engineering methods.

We plan to evaluate the secure twin peaks by applying it to other security requirements engineering methods such as SEPP [23], which is based on Jackson's problem frames [15]. Also, we would like to validate our approach in the context of industrial case studies. We believe that the explicit documentation of traceability links between architectural design and requirements analysis artifacts helps to (1) systematically evolve software systems, and (2) increase the applicability of security patterns in practice.

Acknowledgements. This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U. Leuven.

References

1. Bandara, A., Shinpei, H., Jürjens, J., Kaiya, H., Kubo, A., Laney, R., Mouratidis, H., Nhlabatsi, A., Nuseibeh, B., Tahara, Y., Tun, T., Washizaki, H., Yoshioka, N., Yu, Y.: Security patterns: Comparing modeling approaches. Technical Report 2009/06 (2009)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, 1st edn. (1998)
3. Blakley, B., Heath, C., members of The Open Group Security Forum: Security design patterns. The Open Group (2004)
4. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A system of Patterns. Wiley (1996)
5. Côté, I., Heisel, M., Wentzlaff, I.: Pattern-based Exploration of Design Alternatives for the Evolution of Software Architectures. International Journal of Cooperative Information Systems, World Scientific Publishing Company Special Issue of the Best Papers of the ECSA'07 (December 2007)
6. Dougherty, C., Sayre, K., Seacord, R.C., Svoboda, D., Togashi, K.: Secure design patterns. Tech. Rep. CMU/SEI-2009-TR-010, Carnegie Mellon Software Engineering Institute (2009)
7. Fernandez, E.B., Larrondo-Petrie, M.M., Sorgente, T., Vanhilst, M.: Integrating Security and Software Engineering: Advances and Future Visions, chap. A Methodology to Develop Secure Systems Using Patterns, pp. 107–126 (2007)
8. Giorgini, P., Mouratidis, H.: Secure tropos: A security-oriented extension of the tropos methodology. International Journal of Software Engineering and Knowledge Engineering 17(2), 285–309 (2007)
9. Haley, C.B., Laney, C.R., Moffett, D.J., Nuseibeh, B.: Security requirements engineering: A framework for representation and analysis. IEEE Transactions on Software Engineering 34(1), 133–153 (2008)
10. Haley, C.B., Moffett, J.D., Laney, R., Nuseibeh, B.: A framework for security requirements engineering. In: Proceedings of the International Workshop on Software Engineering for Secure Systems (SESS). pp. 35–42. ACM Press, New York, NY, USA (2006)
11. Haley, C.B., Nuseibeh, B.: Bridging requirements and architecture for systems of systems. In: Proceedings of the International Symposium on Information Technology (ITSim). vol. 4, pp. 1–8 (2008)

12. Hall, J.G., Rapanotti, L., Jackson, M.: Problem oriented software engineering: Solving the package router control problem. IEEE Transactions on Software Engineering 34(2), 226–241 (2008)
13. Heyman, T., Yskout, K., Scandariato, R., Joosen, W.: An analysis of the security patterns landscape. In: Proceedings of the International Workshop on Software Engineering for Secure Systems (SESS). pp. 3–10. IEEE Computer Society (2007)
14. Islam, S., Mouratidis, H., Jürjens, J.: A framework to support alignment of secure software engineering with legal regulations. Journal of Software and Systems Modeling (March 2010), published online first
15. Jackson, M.: Problem Frames. Analyzing and structuring software development problems. Addison-Wesley (2001)
16. Jürjens, J.: Secure Systems Development with UML. Springer (2005)
17. Kienzle, D.M., Elder, M.C., Tyree, D., Edwards-Hewitt, J.: Security patterns repository (2002)
18. Mouratidis, H., Jürjens, J.: From goal-driven security requirements engineering to secure design. International Journal of Intelligent Systems – Special issue on Goal-Driven Requirements Engineering 25(8), 813 – 840 (June 2010)
19. Mouratidis, H., Jürjens, J., Fox, J.: Towards a comprehensive framework for secure systems development. In: Dubois, E., Pohl, K. (eds.) Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE) (LNCS 4001). pp. 48–62. LNCS 4001, Springer (2006)
20. Mouratidis, H., Weiss, M., Giorgini, P.: Modelling secure systems using an agent oriented approach and security patterns. International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 16(3), 471–498 (2006)
21. Nhlabatsi, A., Nuseibeh, B., Yu, Y.: Security requirements engineering for evolving software systems: A survey. Journal of Secure Software Engineering 1(1), 54–73 (2009)
22. Nuseibeh, B.: Weaving together requirements and architectures. Computer 34(3), 115–117 (2001)
23. Schmidt, H.: A Pattern- and Component-Based Method to Develop Secure Software. Deutscher Wissenschafts-Verlag (DWV) Baden-Baden (April 2010)
24. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns: Integrating Security and Systems Engineering. Wiley & Sons (2005)
25. Steel, C., Nagappan, R., Lai, R.: Core security patterns: Best practices and strategies for J2EE, web services, and identity management (2005)
26. van Lamsweerde, A.: From system goals to software architecture. In: Formal Methods for Software Architectures. pp. 25–43. LNCS 2804, Springer (2003)
27. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (March 2009)
28. Weiss, M.: Modeling security patterns using NFR analysis. In: Integrating Security and Software Engineering, pp. 127–141. Idea Group (2007)
29. Weiss, M., Mouratidis, H.: Selecting security patterns that fulfill security requirements. In: IEEE International Requirements Engineering Conference (2008)
30. Yoder, J., Barcalow, J.: Architectural patterns for enabling application security. In: Proceedings of the International Patterns Language of Programming (PLoP) Conference (1997)
31. Yskout, K., Scandariato, R., De Win, B., Joosen, W.: Transforming security requirements into architecture. In: Proceedings of the International Conference on Availability, Reliability and Security (AReS). pp. 1421–1428. IEEE Computer Society, Washington, DC, USA (2008)