A Methodology for Mining Security Tactics from Security Patterns

Jungwoo Ryoo
Penn State

Phil Laplante
Penn State

Rick Kazman
University of Hawaii/SEI

Abstract

Although many aids such as architectural styles and patterns are now available for software architects, making optimal design decisions on appropriate architectural structures still requires significant creativity. In an effort to introduce a more direct link between an architectural decision and its consequences, a finer grained architectural concept called a tactic has emerged. Since its introduction, many tactics have been identified and used in real- life applications. However, the number of tactics discovered is not sufficient to cover all the necessary aspects of architectural decision making. The tactics could be created from scratch, but it would be more efficient if tactics could be mined from a proven source. One possible source is any architectural pattern that consists of tactics. Therefore, in this paper we propose a novel way to retrieve tactics from well known patterns. Among the many different types of existing patterns, this paper focuses on security patterns.

1. Introduction

Making the right decisions in designing a system or software architecture is always a challenge. Experience plays an important role, and there is often no clear "right" decision; as a result experienced designers often resort to their experience or intuition in making architectural decisions. Since one of the primary goals of software engineering is to remove this artistic aspect as much as possible, researchers have been working to develop a more scientific process that is repeatable and produces uniform conclusions given the same set of requirements.

Architectural patterns and design patterns are the culmination of one such effort. A *pattern* is a well known solution to a recurring software development problem [1]. Patterns are intended to help software design by capturing past experiences and allowing for communication and propagation of best practices thus improving the architect's productivity, quality and confidence. Depending on the abstraction level being employed during a particular phase of a software development process, a pattern can be that of an architectural nature or of a lower level design. The former is usually referred to as an architectural pattern while the latter is referred to as a design pattern. This paper concentrates on architectural patterns.

Software architectures rarely deal with functionality. Rather, they address universal concerns common across many different types of software applications. These concerns include so-called "non-functional" requirements such as security, availability, modifiability, performance, reusability, etc. Quality attribute is another common term for these overarching concerns. Quality attributes are critical in producing a satisfactory software product since functionality does not matter when one or more of these quality attributes are not satisfied. For example, one of the main motivations for switching to another Operating Systems (OS) or to a later version of an existing OS is to improve security. If an OS turns out to be insecure, it will quickly lose its popularity no matter how much functionality it provides.

Despite their popularity, architectural patterns have some weaknesses. One of these is the fact that an architectural pattern usually addresses multiple quality attribute concerns (or forces). This characteristic is problematic because the relationship between a certain quality attribute and a pattern is not clearly defined—it is dependent on implementation decisions, detailed design decisions, and context. This, in turn, makes it difficult for an architect to confidently select a pattern that solves a specific problem at hand.

To overcome this weakness, one needs a finer grained design concept that tackles one quality attribute at a time. To respond to this newly emerging need, the idea of *tactics* has been proposed [2]. Unlike architectural patterns, tactics are meant to address a single architectural force and therefore give more precise control to an architect.

Patterns package tactics. For example, let us examine the most common architectural pattern—the Layered Pattern—to see how this works in practice. Layers group together similar sets of functionality and separate those from other functions that are expected to change independently. By doing so, the modifiability of the system is expected to increase. For example, layering is often used to insulate a system from changes in the underlying platform (hardware and software), increasing maintainability while reducing integration and verification costs. To do this the architect creates one or more platform-specific layers to abstract the details of the underlying hardware and operating system. The rest of the system's functionality then accesses the underlying platform via these abstractions. To achieve this effect the Layered Pattern employs two Localize Change tactics—"Semantic Coherence" and "Abstract Common Services"—to increase cohesion, and it employs three Prevent Ripple Effects tactics—"Use Encapsulation", "Use an Intermediary", and "Restrict Communication Paths"—to reduce coupling [10].

With tactics, structural decisions can be made more easily since a quality attribute of concern can be explicitly mapped to individual tactics. In fact, one way to think of a tactic is a set of building blocks that constitute an architectural pattern. Once the missing links between a pattern and its corresponding tactics are fully recovered, the uncertainties surrounding an architectural pattern (in terms of quality attributes) can be minimized or even eliminated.

But where do tactics come from? We can, in fact, discover tactics by generalizing over many instances of design and architectural patterns. This approach is beneficial because tactics are relatively new, compared to patterns, and existing tactics repositories are still evolving.

Based on this observation, this paper proposes a novel methodology to mine tactics from well known architectural patterns, particularly, those relevant to security.

2. Security patterns and tactics

It is well known that a software flaw costs an organization dramatically more if it is found relatively later during a software development process [3]. Security flaws are no exception. Major security glitches (other than programmer mistakes) found during testing are much more expensive to correct than those identified in an early design phase. If one would like to introduce a security property or safeguard that is *architectural*—and hence overarching and lasting during the entire life time of the software— it is crucial to have an appropriate architectural strategy in place during the earliest stages of software development. Design, particularly at the architectural design stage, is considered the earliest possible (and therefore most cost-effective) place to impact a software solution, which is why this paper focuses on architectural approaches to security (such as architectural patterns and tactics) rather than lower level design, implementation, and testing.

There are many published security patterns [4]. Among these, some are architectural patterns while others are design patterns. In addition, there are architectural tactics that have been misidentified as patterns. The distinction is not always clear, which often confuses the users of the patterns. One of the main contributions of this paper is to improve the current categorization of patterns so that one can clearly distinguish patterns from tactics. This achievement also leads to the discovery of more tactics to supplement the existing tactics repositories by making it possible to either break down an existing pattern into constituent tactics or correctly identifying a tactic misidentified as an architectural pattern.

2.1. Security Patterns

As its definition suggests [1], the development of patterns tends to be community-driven. This community could consist of a group of expert practitioners who can reflect on their development experiences in a certain problem domain and collectively decide which of the patterns they know are worthy of repeating or to be avoided at all costs. Depending on the level of acceptance by the software development community in general, some patterns may fade away while the popularity of other patterns may be bolstered over the time.

Patterns are defined by a **name**, a **context** in which their use is ideal, one or more recurring **problems** they solve, **forces** describing quality attributes involved and tradeoffs to be made among them, the actual design **solution**, and other **related patterns** [1].

As the number of security patterns grows, attempts are being made to find different ways to organize them [5,6,7]. Unearthing various categorization criteria for patterns in an exhaustive manner is meaningful in patterns research since it helps efforts to identify new patterns by providing a reasoning framework to systematically explore unknown or missing patterns.

One of the criteria used by Hafiz at al. is a classic security domain concept such as confidentiality, integrity, or availability [7]. Security patterns can be categorized by which of these three security control parameters they promote. It turns out that a significant number of patterns (5 out of 14) affect multiple security domain concepts (e.g., both integrity and availability) while a dominant number of patterns (9 out of 14) support a single security domain concept.

These findings support our hypothesis that some of the security tactics are being misidentified as architectural patterns. That is, the patterns associated with only one security domain concept are prime candidates to be reclassified as tactics since their effects are limited to having an effect on just a single quality attribute. The patterns belonging to this category include those promoting:

- Confidentiality: authenticator, authorization, exception shielding, policy enforcement point, secure pre-forking, single access point, and subject descriptor,
- Integrity: safe data buffer, and
- Availability: checkpointed system.

We will further investigate some of these patterns by discussing a methodology for mining security tactics in Section 3.

2.2. Security Tactics

Compared to security patterns, security tactics are still in an early stage of development, which is another motivation for finding a reliable mechanism to mine and categorize security tactics and to eventually galvanize the research community in this area.

The known set of security tactics can be described in a single hierarchy based on the following three categories: (1) resisting attacks, (2) detecting attacks, and (3) recovering from an attack [2]. Individual tactics are then placed directly under these categories as shown in Figure 1.

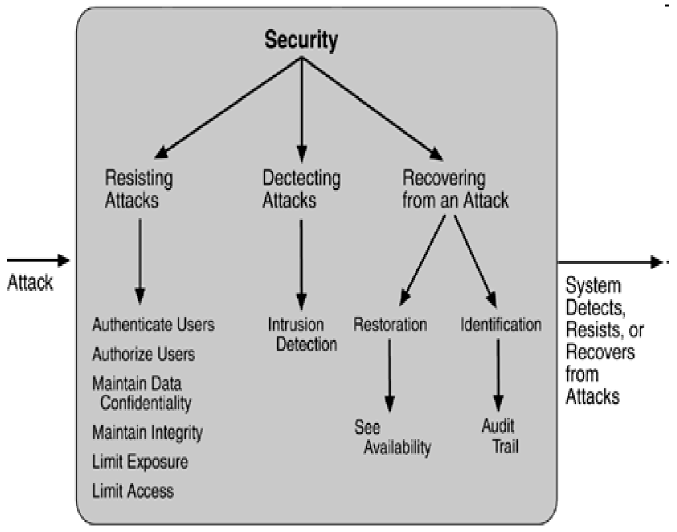

Figure 1: A hierarchy of security tactics.

The hierarchy appears to be flat, suggesting ample room for many more intermediate nodes in the graph. A richer hierarchy will naturally develop as newly discovered tactics fill the gap. Some of the existing tactics categories can also be extended to include subcategories by borrowing the criteria already used for classifying security patterns. For example, items such as (1) maintain data confidentiality and (2) maintain integrity can be mapped to the corresponding security patterns categories promoting security domain concepts like confidentiality and integrity respectively. In addition, based on commonly accepted categorizations of security measures in the security community, the (1) limit access, (2) authenticate users, and (3) authorize users can reclassified as shown in Figure 2.

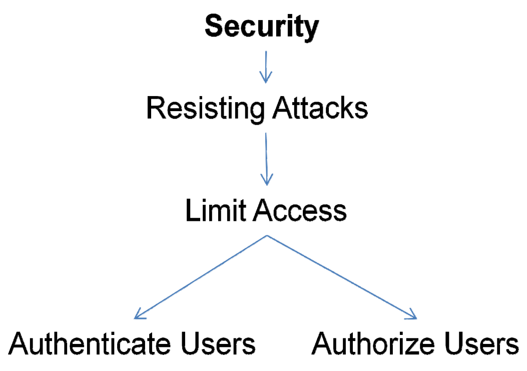

Figure 2: An example of restructuring the security tactics hierarchy.

Note that the authors' ultimate goal is to be able to add new concrete tactics underneath each of the leaf nodes in the existing graph, such as the (1) authenticate users and (2) authorize users categories as well as adding new mid-level nodes. This will provide a richer and hopefully more complete set of building blocks for security patterns.

3. A new methodology for mining security tactics

We now introduce a new methodology for mining security from well known security patterns. The core concept of the methodology is to examine existing security patterns to verify if any known security pattern satisfies all the conditions necessary to qualify it as a security tactic. These conditions include: (1) atomicity, (2) force limitation, (3) problem-specificity, (4) completeness, and (5) trade-offs between forces. They are explained in detail in the following subsections.

3.1. Atomicity

Atomicity condition requires tactics to be the highest level design concept (almost at the boundary between a problem and a solution domain) that cannot be further divided into other multiple tactics.

However, it can be refined into a more concrete tactic. For instance, the two arrows extending from the limit access tactic in Figure 2 do not indicate that the tactic consists of authenticate users and authorize users tactics. Rather, the arrows show that the limit access tactic can be either refined into authenticate users or authorize users tactics.

Tactics are primitives that serve as building blocks for architectural patterns. It is also true that tactics and patterns are not at the same abstraction level. Patterns implement tactics; therefore tactics are a more abstract design concept than patterns.

3.2. Force limitation

A security tactic addresses just the quality attribute of security, while a security pattern has an effect on other quality attributes as well. We believe that a typical pattern's effect on multiple quality attributes (for example, positively affecting security and negatively affecting performance) is partly due to the fact that there exist different constituent tactics affecting each of the forces (i.e., tactic A positively affecting security and anti-tactic B negatively affecting performance).

As a result, if a security pattern consists of only one security tactic, that pattern is most likely to be a tactic misidentified as a pattern. Of course, to be fully qualified as a tactic, it needs to satisfy all the other conditions, too.

3.3. Problem specificity

Patterns tend to be domain-specific and developed around a particular problem (e.g., patterns for information visualization []). Tactics are domain-neutral and simpler in their definition (e.g., authenticate users). As a result, they are generic and require fewer adjustments before they are ready to be applied to any problems in a given domain.

These behaviors of patterns and tactics have a lot to do with the fact that patterns tend to me more closely tied to a specific problem, while tactics only focus on a quality attribute response.

3.4. Completeness

The different levels of problem specificity required for patterns and tactics also influence how complete they can be in their original forms. Patterns are all underspecified: they tend to be incomplete, and many details need to be added before a pattern can be implemented. Tactics, on the other hand, require less tailoring, and there is no need for additional details because they are earlier-stage design decisions and as a result, are less attached to a particular problem. In this sense, one can claim that tactics can afford to be specified with less information and that they are more complete in their own right and for their own purposes.

3.5. Tradeoffs between forces

As already discussed in Section 3.2, a tactic addresses a single force. Different forces and their effects on each other are considered only after the tactic is translated into a pattern along with other tactics. Therefore, architects do not need to reason about tradeoffs between tactics.

The reasoning begins after individual tactics are woven into a pattern at which time they have more information to make better decisions on what tradeoffs can be made between implemented versions of tactics (e.g., a proxy as an element of a pattern rather than an intermediary as a tactic). That is, by the time one is considering tradeoffs among different forces, one is already dealing with a pattern.

4. Examples

This section provides two straightforward scenarios demonstrating how the criteria described in Section 3 can be used to mine security tactics from existing security patterns.

In the first scenario, the Authenticator Pattern [8] (one of the security patterns mentioned as a prime candidate for being misidentified as a tactic) is used to show how a candidate pattern is categorically denied reclassification. This rejection process also shows how some tactics can still be salvaged as byproducts.

According to Brown et al., "The authenticator pattern performs authentication of a requesting process before deciding access to distributed objects." This description already gives a strong impression that the pattern is strongly tied to a specific problem domain. Phrases such as 'requesting process' and 'distributed objects' strongly suggest domain-specificity. It is obvious that the authenticator pattern does not meet the problem specificity condition.

Figure 3 shows the authenticator pattern is relying on another pattern (i.e., factory method). The factory method pattern is mapped to the information hiding tactic (through the use of an interface) and the intermediary tactic, both of which promote the modifiability quality attribute of the software. Based on these facts, one can conclude that the authenticator pattern violates the atomicity, force limitation, completeness conditions. The pattern satisfies the tradeoffs between forces condition since the contention between security and modifiability is minimal. Table 1 summarizes the discussion so far, which is a step-by-step application of the tactics mining methodology this paper is proposing.

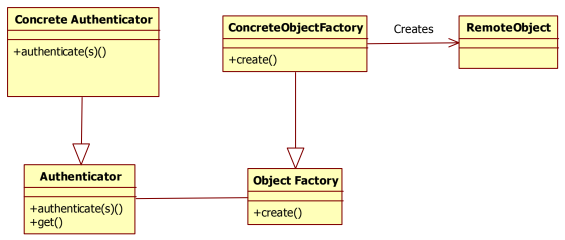

Figure 3: A UML diagram for the authenticator Pattern [8].

Although the authenticator pattern itself is disqualified as a tactic, one might still be able to recognize a tactic behind the pattern, which is authenticate users. Nonetheless, the recovered tactic is already known one in this case.

Table 1: A summary of the authentication pattern reclassification test.

Pattern Name	**Authenticator**
Atomicity	Fail
Force Limitation	Fail
Problem Specificity	Fail
Completeness	Fail
Tradeoffs between Forces	Pass
Tactic?	**No**

The second scenario involves a pattern that is clearly misidentified and should be reclassified as a tactic. The pattern is referred to as Compartmentalization [9]. The pattern is described as: "Put each part in a separate security domain. Even when the security of one part is compromised, the other parts remain secure." Unlike the authenticator pattern discussed earlier, this description does not have anything indicating problem specificity. It is also atomic and satisfies the force limitation (addressing only the security quality attribute), completeness, and tradeoffs between forces conditions. Therefore, the compartmentalization pattern should be reclassified as a tactic. In the tactics hierarchy, the new tactic should be located under the limit access tactic.

5. Conclusions and further research

A great deal of research has been directed at security architectures and patterns over the last few years. Yet this research seems to slowly penetrate into actual practice. Perhaps this lag is due to the relative lack of clearly demonstrated linkages between well-known and trusted tactics, and the architectures that embody them. We hope this research will begin to help bridge that gap and to produce new architectures through tactics mining

6. References

[1] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, Reading, 1995.

[2] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, Reading, MA, 2003.

[3] A. Davis, "Software Requirements: Objects, Functions and States. 3rd ed," PTR Prentice-Hall, Indianapolis, 1993.

[4] J. Yoder and J. Barcalow, "Architectural Patterns for Enabling Application Security," Proc. 4th Conf. Pattern Languages of Programs (PLoP97), 1997, http://jerry.cs.uiuc.edu/plop/plopd4-submissions/P60.doc.

[5] M. Schumacher, E. Fernandez-Buglioni , D. Hybertson, F. Buschmann, and P. Sommerlad "Security Patterns: Integrating Security and Systems Engineering," John Wiley & Sons, Hoboken, 2005.

[6] B. Blakley and C. Heath," Security Design Patterns Technical Guide— Version 1" Open Group, 2004; www.opengroup.org/security/gsp.htm.

[7] M. Hafiz, P. Adamczyk, and R. E. Johnson, "Organizing Security Patterns". *IEEE Software*. July/August 2007, Vol. 24, 4, pp. 52-60.

[8] F. Brown, F DiVietri, G. Villegas, and E. Fernandez. "The Authenticator Pattern".
Pattern Languages of Programs 1999, Monticello, IL, 1999.

[9] M. Hafiz, R.E. Johnson, and R. Afandi. "The security architecture of qmail." In 11th Conference on Pattern Languages of Programs, 2004.

[10] F. Bachmann, L. Bass, R. Nord, "Modifiability Tactics", Software Engineering Institute Technical Report CMU/SEI-2007-TR-002, 2007.