2012 IEEE Sixth International Conference on Software Security and Reliability Companion

Revising a Security Tactics Hierarchy through Decomposition, Reclassification, and Derivation

Jungwoo Ryoo and Phil Laplante
College of Information Sciences and Technology
The Pennsylvania State University
University Park, PA U.S.A.
{jryoo and plaplante}@psu.edu

Rick Kazman
Shidler College of Business
University of Hawaii
2404 Maile Way
Honolulu, HI 96822 U.S.A.
kazman@hawaii.edu

Abstract— **Software architecture is the set of important design decisions that address cross-cutting system quality attributes such as security, reliability, availability, and performance. Practitioners often face difficulty in beginning an architectural design due to the lack of concrete building blocks available to them. Tactics are fundamental design decisions and play the role of these initial design primitives and complement the existing design constructs such as architectural or design patterns. A tactic is a relatively new design concept, and tactics repositories are still being developed. However, the maturity of these repositories is inconsistent, and varies depending on the quality attribute. To address this inconsistency and to promote a more rigorous, repeatable method for creating and revising tactics hierarchies, we propose a novel methodology of *extracting* tactics. This methodology, we claim, can accelerate the development of tactics repositories that are truly useful to practitioners. We discuss three approaches for extracting these tactics. The first is to derive new tactics from the existing ones. The second is to decompose an existing architectural pattern into its constituent tactics. Finally, we extract tactics that have been misidentified as patterns. Among the many types of tactics available, this paper focuses on *security* tactics. Using our methodology, we revise a well-known taxonomy of security tactics. We contend that the revised hierarchy is complete enough for use in practical applications.**

Keywords-tactics; extraction; repository; security; patterns; taxonomy; derivation; decomposition; reclassification

I. INTRODUCTION

Making the right decisions in designing a software system is always a challenge. Experience plays an important role, and there is often no clear "right" decision. As a result, seasoned software architects often resort to their experience or intuition in making design 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 that produces consistent and appropriate solutions given the same set of requirements.

Architectural patterns and design patterns are one result of these efforts. A pattern is a well-known solution to a recurring software development problem [8]. Patterns are intended to help the software design activity 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 employed, a pattern may have implications throughout the system, or it may only affect a small portion of it. The former type is usually referred to as an architectural pattern while the latter is referred to as a design pattern [7]. This paper concentrates on architectural patterns.

Software architectures address global, crosscutting concerns that are common across many different types of software applications. These concerns include so-called "non-functional" requirements such as security, availability, reliability, performance, etc. We follow [2] in referring to these overarching concerns as *quality attributes*.

Quality attributes are critical in producing a satisfactory software product since functionality does not matter when quality attributes are not satisfied. For example, a stakeholder may switch to a different Operating Systems (OS). If that OS turns out to have major security flaws, it will quickly lose its appeal no matter how much functionality it provides.

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

To overcome this weakness, an architect needs finer-grained design primitives to be able to reason about one quality attribute at a time. To respond to this need, the idea of tactics has been proposed [2, 17]. Bass et al. define tactics as "a design decision that influences the control of a quality attribute response." Unlike architectural patterns, tactics are

978-0-7695-4743-5/12 $26.00 © 2012 IEEE
DOI 10.1109/SERE-C.2012.18

meant to address a single architectural force and therefore give more precise control to an architect when making design decisions.

Patterns package tactics. For example, consider the most common architectural pattern—the Layered Pattern. Layers group together similar sets of functions and separate those from other function sets that are expected to change independently. By doing so, the modifiability/maintainability of the system is expected to increase. 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 [1].

Structural decisions can thus be made more easily and confidently, since a quality attribute of concern can be explicitly mapped to an individual tactic. One way to think of a tactic is as a set of building blocks that, together, constitute an architectural pattern. Once the relationships between an architectural pattern and its corresponding tactics are understood, the uncertainties surrounding the pattern (in terms of quality attributes) can be addressed and, in doing so, minimized or even eliminated.

But where do tactics come from? As with patterns, they emerge from practice. The process of identifying tactics is empirical: we can "discover" tactics by examining and generalizing over many instances of patterns found in practice. This approach is useful because tactics are a relatively new concept, compared to patterns.

Based on this observation, this paper proposes a novel methodology to extract tactics from architectural patterns. The main motivation for developing the methodology is to help build a comprehensive repository of tactics; using this repository, practitioners can select tactics appropriate for their own problem domain and quality attributes. As the size of the repository becomes larger, searches for a desired tactic will become increasingly complex. To facilitate this process, it is crucial for the tactics repository to have well-defined top level structures (hierarchies). Such hierarchies already exist [2], but they are still in early stages of development. Therefore, the proposed methodology will also provide a mechanism to systematically build a top level tactics hierarchy based on refinement relationships. This requires a revision and extension of the existing tactics hierarchies. The new tactics extraction methodology allows the original taxonomy of tactics to naturally evolve into an improved form—one based on empirical evidence collected from patterns. In this paper we will demonstrate the methodology by revising the original security tactics taxonomy into one that is both richer and more accurate (because it is supported by an explicit chain of empirical observations, as derived from our extraction methodology). In this way we are helping to form a rigorous foundation for the process of architectural design.

This paper will focus on tactics relevant to security. There are many published security patterns (e.g. [18, 19, 20]). Among these sets, some are architectural patterns while others are design patterns; a distinction that is subtle but important. Another contribution of this paper, then, is to propose a principled set of criteria for classifying patterns as architecture patterns, design patterns, or tactics. Using these criteria we then revisit the existing categorizations of security patterns and reclassify them. As we will show, this process has led us to the discovery of additional tactics to supplement the existing security tactics hierarchy [2] by decomposing existing patterns into constituent tactics, by correctly identifying a tactic that had been misidentified as an architectural pattern, or by deriving tactics from one or more known tactics.

II. Background and Existing Research

It is well known that a software flaw costs an organization dramatically more if it is found later during the software development process [6]. Security flaws are particularly egregious in this regard. Major security glitches (other than localized programming mistakes) found during testing are much more expensive to correct than those identified in an earlier development phase. Security vulnerabilities left in the product after release can be catastrophic. If one wishes to introduce a security property that is architectural—pervading the system and lasting for the entire lifetime of the software—then it is crucial to have an appropriate architectural strategy in place during the earliest stages of software development. Architectural design is the earliest possible—and therefore most cost-effective—time to impact a software solution, which is why this paper focuses on architectural approaches to security (architectural patterns and tactics) rather than on later detailed design, implementation, and testing strategies.

A. Security Patterns

As the definition of design patterns suggests [8], the development of patterns is community-driven. These communities consist of groups of expert researchers and practitioners who can reflect on their development experiences in a certain problem domain and collectively decide which of the patterns that they have observed are worthy of collecting and categorizing. In addition, communities can find patterns that are to be avoided; these are referred to as anti-patterns [5, 13]. Architectural patterns exist only if they provide utility to practicing architects. Depending on the level of acceptance by a particular community, 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 of use, one or more recurring problems that they solve, a set of forces describing quality attributes involved and tradeoffs to be made among them, the actual design solution, and other related patterns [8]. As the number of security patterns grows, attempts are being made to find different ways to organize them [3, 9, 12, 18]. Unearthing categorization criteria for patterns in an exhaustive manner is meaningful since it helps efforts to identify new patterns by providing a reasoning framework to systematically explore unknown or missing patterns. The same principle applies to tactics, and we demonstrate how to derive new tactics this way in section IV.

One of the criteria used by [9] to classify security patterns is the classic security domain concept of confidentiality, integrity, and availability. Security patterns can be categorized according to which of these three security aspects they promote. Several patterns (5 out of 14) that Hafiz et al. examined 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.

We will further investigate some of these patterns when we discuss our methodology for extracting security tactics in section III.

B. Security Tactics

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

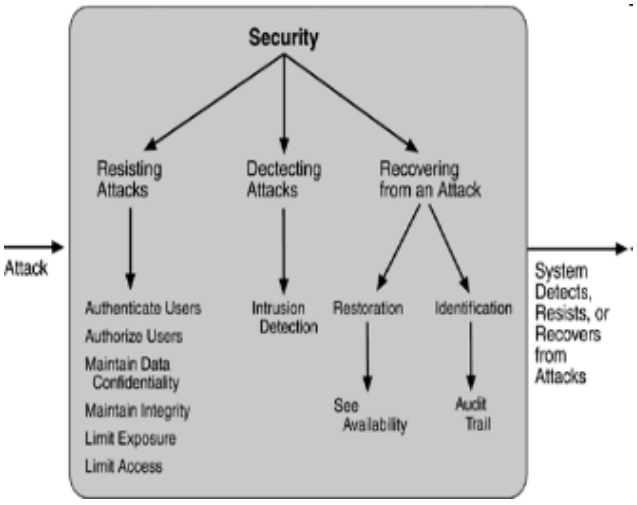

Figure 1. The original tactics hierarchy [2]

Each node in Figure 1 represents an individual tactic. The directed arrows show refinement relationships. For instance, the generic "security" tactic can be refined into more concrete tactics such as "resisting attacks," "detecting attacks," and "recovering from an attack." This notational convention continues in the ensuing figures depicting tactics in this paper.

The current hierarchy of tactics is relatively flat. For example, tactics such as Authenticate Users, Authorize Users, and Limit Access are all located at the same level.

Note that our ultimate goal in this work is to be able to refine and expand the existing graph by adding new mid-level nodes and to associate security tactics instances (i.e., concrete realizations of tactics) with each of the leaf nodes in the improved hierarchy. This work will provide a richer and hopefully more complete set of building blocks for security patterns and builds on our previous research [15, 16].

III. METHODOLOGY FOR EXTRACTING SECURITY TACTICS

A. Description

As mentioned earlier, there are three fundamental ways to extract tactics: one based on decomposition, the other based on reclassification of patterns, and another based on derivation.

Some (i.e., decomposition and reclassification) of these methods require expert scrutiny of existing patterns for their potential to be decomposed into constituent tactics or to be reclassified as tactics. Derivation is possible by reexamining the legacy tactics hierarchy first and its expanded version next.

Reevaluation of the revised tactics hierarchy is particularly important since the newly added tactics may provide new insights for identifying more tactics.

Despite its relatively short history, the security patterns community has already proposed a large number of patterns. This paper uses three published descriptions of security pattern collections [9, 10, 12]. For this part of our methodology, we adopted a variant of the Wideband Delphi approach [4] that is a consensus-based estimation technique for projecting the level of effort, and hence cost, required for a project. The Wideband Delphi approach, as Boehm first presented it, consisted of the following 6 steps:

1. Coordinator presents each expert with a specification and an estimation form.
2. Coordinator calls for a group meeting in which the experts discuss estimation issues with the coordinator and each other.
3. Experts fill out forms anonymously.
4. Coordinator prepares and distributes a summary of the estimates.
5. Coordinator calls for a group meeting, specifically focusing on having the experts discuss points where their estimates vary widely.

6. Experts fill out forms, again anonymously, and steps 4 to 6 are iterated for as many rounds as appropriate.

We implemented this process except that, in Step 1, rather than having each expert work with a specification and an estimation form, each expert had a copy of the existing security tactics hierarchy [2], the combined list of security patterns from Hafiz et al. [9], Kiiski [12], and Halkidis et al. [10], and a tactics classification form.

For Step 2, six experts with significant academic and industrial background in software architecture were called together for a meeting in which the task was discussed and any clarifying questions about the process were answered. For Step 3, the experts independently reviewed each pattern and determined the tactic or tactics that the pattern employed. Following Step 4, the coordinator prepared a summary of the six sets of analyses, distributed these back to the experts, and arranged a meeting to discuss the results.

For Step 5 the experts met, discussed their decisions, and tried to reach a final consensus on each pattern's relationships to a set of tactics. Step 6 proved to be unnecessary. No further iterations of the process were required, as complete consensus was reached during Step 5. The results of this Wideband Delphi process are discussed in section III.B.

B. Application of the Methodology

This section demonstrates how our methodology works by providing several examples. Since steps 1 through 4 are similar for any patterns under inspection, the examples discussed in this section largely concentrate on step 5 of our methodology, in which an explicit association between a tactic and a pattern was made. The types of possible associations include (1) reclassification, (2) decomposition, (3) derivation, (4) realization, and (5) validation.

Reclassification refers to changing the existing designation of a pattern from design/architectural pattern to tactic. The basis of this change is an argument that a tactic already exists but has been misidentified as a design or architectural pattern. Therefore, reclassification simply identifies an appropriate location in an existing tactics hierarchy and properly positions the reclassified pattern.

Decomposition disassembles an existing pattern into two or more constituent tactics.

Derivation refers to a natural process in which the maintainers of a tactics hierarchy recognize one or more missing tactics, based on their reasoning and intuition and add them. These tactics usually emerge from other existing tactics that provide clues for what the missing tactics might be. This tactics extraction process solely depends on an existing tactics hierarchy still under development.

Realization means an instantiated tactic being used in a particular problem context. Tactics in a tactics hierarchy are context-free and are templates/types for guiding an instantiation process that leads to realization. Once fully realized, tactics are parameterized with details supplied by a specific context or problem domain. For example, Authenticate Users is a tactic. There are however many ways to authenticate users in a particular software application. Authentication can be done via passwords, tokens, biometric scanners, etc. which are the realizations of the Authenticate Users tactic.

Validation refers to a scenario in which there already exist tactics comparable to known architectural or design patterns. Validation is similar to reclassification in that misidentified patterns (that need to be reclassified as tactics) are revealed in its process. However, it is different from reclassification since nothing new is added to the existing tactics hierarchy. Validation simply reinforces the credibility of a tactic and the reason for its existence.

IV. TACTICS EXTRACTION EXAMPLES

A. Audit Design: An Example of Reclassification

Kiiski [12] defines the Audit Design pattern as follows:

Auditing is a process to analyze logs or other information of an event. Audit Design helps to satisfy these audit requirements. In addition, the Audit Design helps satisfy the logging requirements.

This pattern deals with security-relevant events that have already occurred and is associated with the Audit Trail tactic in the original tactics hierarchy [2]. The Audit Trail tactic is defined as follows:

Audit Trail is a tactic that [attempts to] identify an attacker. It is a copy of each transaction applied to the data in the system together with identifying information. Audit information can be used to trace the actions of an attacker.

The Audit Design pattern refines the Audit Trail tactic since it emphasizes the event-logging aspect of the tactic. This has been renamed in the revised tactics hierarchy as the "Log Events" tactic.

Had there not been the Audit Design pattern identified by the tactics community, the tactic could also have been derived from the Audit Trail tactic.

B. Role-Based Access Control (RBAC): An Example of Decomposition

Kiiski [12] defines the RBAC pattern as follows:

The RBAC pattern describes how to assign rights to resources depending on the role of the user. This way the administrators

have to manage only a reasonable number of rights for different roles and not for every user separately.

This pattern is derived from multiple tactics. It is a direct descendent of the Authorize Users tactic, but the pattern assumes that identification and authentication are already done. Therefore, the RBAC pattern can be decomposed into: Identify Users, Authenticate Users, and Authorize Users.

C. Detect Service Denial: An Example of Derivation

In the security taxonomy (Figure 1), there is only one tactic (i.e., Intrusion Detection). Typical attacks mitigated by Intrusion Detection can compromise confidentiality (e.g., by stealing information) and integrity (e.g., by breaching secure boundaries). However, there are also attacks affecting availability (e.g., by deleting important files or simply taking down the system). Therefore, it is safe to deduce that there must be a tactic used to identify denial of service (DOS) type attacks where a large load of some kind is placed on the system to render it completely inoperable or to severely degrade its performance. We refer to this newly derived tactic as Detect Service Denial. As a result, we conclude that Detect Attacks consists of two tactics—Detect Intrusion and Detect Service Denial.

D. Compartmentalization: An Example of Realization

Hafiz et al. [9] defines the compartmentalization pattern as follows:

Put each part in a separate security domain. Even when the security of one part is compromised, the other parts remain secure.

A majority of experts agreed that the pattern could be traced back to the Limit Exposure tactic while one picked both Limit Exposure and Limit Access. The original definitions for the two tactics [2] are:

- **Limit Exposure**: *Attackers typically depend on exploiting a single weakness to attack all data and services on a host. The architect can design the allocation of services to hosts so that limited services are available on each host*.

- **Limit Access**: *As in firewalls and Demilitarized Zones (DMZs), this tactic protects resources from both known and unknown sources by blocking attempts to use the resources*.

The disagreement turned out to be partly due to the domain-specific nature of the original tactic definitions (i.e., network security architectures dealing with components such as hosts). As a result, to resolve the disagreement, the definitions of the tactics had to be revised to better highlight the differences between the two tactics in a non-domain-specific way.

- **Limit Exposure**: *This tactic concentrates on minimizing the attack surface [14] of a system. It can also be characterized as a passive defense or security design since it does not proactively prevent attackers from doing harm. The Limit Exposure tactics are more defensive in nature and do not interact with the manifestation of attack attempts (such as malware) directly. The focus of this tactic is more on protecting potential victims after an incident rather than thwarting attackers before their attacks are fully materialized. This tactic focuses on reducing the probability of and minimizing the effect of damage caused by hostile action without intention of taking the initiative (U.S. Department of Defense n.d.)*.

- **Limit Access**: *These are active defense interventions that directly constrain the behaviors of an attacker. For instance, firewalls actively examine packets for their header contents and drop the packets sent by the attackers*.

Based on these enhanced definitions, the experts could reach the following consensus. The compartmentalization pattern is an example of the Limit Exposure tactic since it:

- Minimizes the attack surface by partitioning a software application into different compartments separated by walls that keep a breach from spreading from one compartment to the other and

- Passively copes with an attack by changing the structure of the software (i.e. compartmentalization) itself and makes it less vulnerable instead of adding stand-alone security features that actively discourage an attack.

Note that the main goal of this tactic is to confine the effect of an attack in a compartment rather than in the entire software, which matches the new definition of the Limit Exposure tactic.

E. Distributed Responsibility: An Example of Validation

Hafiz et al. [9] defines the Distributed Responsibility pattern as follows:

Definition 1: *Partition responsibility across compartments such that compartments that are likely to fail do not have data that needs to be secure*.

Definition 2: *Assign responsibilities in such a way that several of them need to fail in order for the system as a whole to fail*.

The experts agreed that this pattern is a refinement of the Limit Exposure pattern. The definitions of the distributed responsibility pattern are, however, slightly different refinements. The first definition is distributed responsibility

for data while the second definition is distributed responsibility for function. However, since these refinements do not introduce any new architectural design primitives, they do not cause us to further refine the Limit Exposure tactic. They simply validate its generality.

F. Reevaluation of the Existing Hierarchy

Some of the existing tactics categories can also be extended to include subcategories by borrowing the criteria already used for classifying security patterns [9]. 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 the commonly accepted categorizations of security measures in the security community, the (1) limit access, (2) authenticate users, and (3) authorize users can be reclassified as shown in Figure 2.

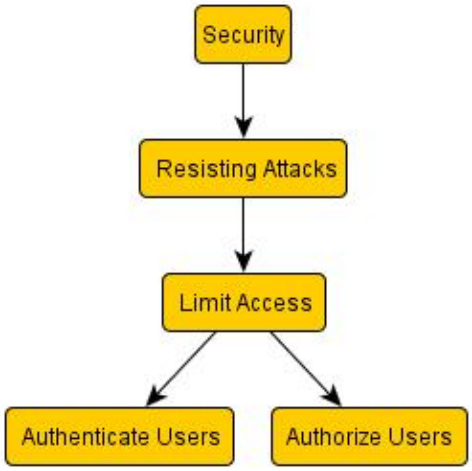

Figure 2. Restructuring the security tactics hierarchy

V. REVISION OF THE EXISTING TACTICS HIERARCHY

Based on the discussions provided in the previous sections, the original tactics hierarchy has been updated, as shown in Figure 3 below.

For consistency in wording, some of the names of the tactics have been reworded from their original manifestations. Since a majority of the tactics in the original tactics hierarchy are in the form of verb phrases, the tactics that did not conform to this rule were renamed. For example, the Detecting Attacks tactic was rephrased as Detect Attacks.

Another set of changes we propose to the nomenclature used in the original security tactics hierarchy is how it refers to users. The term "user" is misleading due to its human connotation. Since a "user" might be human or non-human (e.g., another system), the use of a more generic term is warranted. Therefore, we replaced the term "user" with "actor" in the revised security tactics hierarchy. The term actor [11] is widely used in the requirements engineering community, and it refers to both human and non-human entities with which a system interacts.

But by far the most important changes to the tactics hierarchy were the ones that changed its structure, making it more empirically based, complete, and internally consistent. Since tactics are meant to be an aid to designers, and since architectural design is notoriously difficult, the creation of a rich and consistent vocabulary of design primitives is a significant step in making architectural design regular and repeatable.

VI. RELATED RESEARCH AND FUTURE RESEARCH DIRECTIONS

Our ultimate goal is to be able to both qualitatively and quantitatively measure the effectiveness of a tactic. This verification of effectiveness is crucial because the adoption of a tactic depends on its usefulness. Unless one can provide objective evidence for how useful a tactic is in addressing a particular design concern (e.g., preventing escalation of privilege), it is difficult to make recommendations regarding the use of the tactic. Tactics and their taxonomy provided in this paper have a great potential to pass this litmus test since they have the implicit approval of the software architecture patterns community. However, they fall short of being formally certified due to the lack of a rigorous evaluation process. Such a process requires an infrastructure consisting of the following three core elements: specification, retrieval, and verification mechanisms.

VII. CONCLUSION

In this work we evaluated three different collections of security architecture patterns and used a Wideband Delphi technique with six experts to produce an updated security tactics hierarchy. This hierarchy significantly refines and improves the hierarchy previously developed by Bass et al. [2] which has been in use for almost a decade. Getting these tactics right is important because they are claimed to be design primitives and as such they form the basis for a theory of design.

Heretofore tactics hierarchies have been created in an informal manner, in much the same way that design patterns have been created: based on the judgments and experiences of practitioners. In this paper we have introduced a rigorous approach to defining the tactics hierarchies, which is essential for maintaining, amending, and improving tactics hierarchies of any kind. Indeed, the technique we have introduced is not limited to security; that was just our example in this paper. This technique can be used to rigorously produce tactics hierarchies for all desirable qualities of a software architecture.

REFERENCES

[1] Bachmann, F., Bass, L., & Nord, R. (2007). *Modifiability Tactics* (No. CMU/SEI-2007-TR-002). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University. Retrieved from http://www.sei.cmu.edu/library/abstracts/news-at-sei/architect200708.cfm

[2] Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.). Addison-Wesley Professional.

[3] Blakley, B., & Heath, C. (2004, April). Security Design Patterns. *The Open Group*. Retrieved from http://www.opengroup.org/pubs/catalog/g031.htm

[4] Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall.

[5] Brown, W., Malveau, R., McCormick, H., & Mowbray, T. (n.d.). AntiPatterns. Retrieved February 21, 2010, from http://www.antipatterns.com/

[6] Davis, A. M. (1993). Software Requirements: Objects, Functions and States (Revised Edition) (2nd ed.). Prentice Hall.

[7] Eden, A., & Kazman, R. (2003) Architecture, Design, and Implementation (pp. 149-159). *Presented at the 25th International Conference on Software Engineering (ICSE 25)*, Portland, OR.

[8] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (illustrated edition.). Addison-Wesley Professional.

[9] Hafiz, M., Adamczyk, P., & Johnson, R. E. (2007). Organizing Security Patterns. *IEEE Software*, *24*(4), 52-60.

[10] Halkidis, S. T., Chatzigeorgiou, A., & Stephanides, G. (2006). A qualitative analysis of software security patterns. *Computers & Security*, *25*(5), 379-392. doi:10.1016/j.cose.2006.03.002

[11] Jacobson, I. (1992). Object Oriented Software Engineering: A Use Case Driven Approach (Revised.). Addison-Wesley Professional.

[12] Kiiski, L. (2007). Security Patterns in Web Applications. Presented at the TKK T-110.5290 Seminar on Network Security.

[13] Laplante, P. A., & Neill, C. J. (2005b). *Antipatterns: Identification, Refactoring, and Management* (1st ed.). Auerbach Publications.

[14] Manadhata, P., & Wing, J. (2006). An Attack Surface Metric. Presented at the USENIX Security Workshop on Security Metrics (MetriCon), Vancouver, BC. Retrieved from http://www.cs.cmu.edu/~pratyus/as.html

[15] Ryoo, J., Laplante, P., & Kazman, R. (2009). In Search of Architectural Patterns for Software Security. *Computer*, *42*(6), 98-100. doi:10.1109/MC.2009.193

[16] Ryoo, J., Laplante, P., & Kazman, R. (2010). A Methodology for Mining Security Tactics from Security Patterns. Presented at *the 43rd Hawaii International Conference on System Sciences (HICSS 43)*. Koloa, Kauai, Hawaii, USA: The IEEE Computer Society Press.

[17] Scott, J. & Kazman, R. (2009). Realizing and Refining Architectural Tactics: Availability. CMU/SEI-2009-TR-006, Software Engineering Institute, Carnegie Mellon University.

[18] Schumacher, M. (2003). Security Engineering with Patterns: Origins, Theoretical Models, and New Applications (1st ed.). Springer.

[19] Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., & Sommerlad, P. (2006). *Security Patterns: Integrating Security and Systems Engineering* (1st ed.). Wiley.

[20] Yoder, J., & Barcalow, J. (1997). Architectural Patterns for Enabling Application Security. Presented at the Fourth Conference on Patterns Languages of Programs (PLoP '97), Monticello, Illinois. Retrieved from http://www.joeyoder.com/papers/patterns/

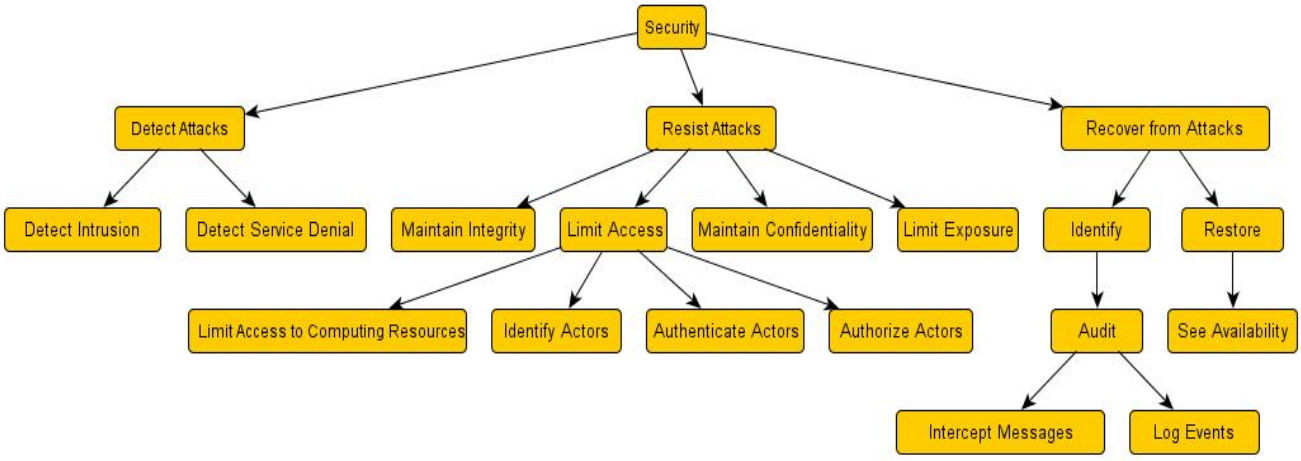

Figure 3. Updated tactics hierarchy