Growing a Pattern Language (for Security)

Munawar Hafiz
Auburn University
munawar@auburn.edu

Paul Adamczyk
paul.adamczyk@gmail.com

Ralph Johnson
University of Illinois at Urbana-Champaign
rjohnson@illinois.edu

Abstract

Researchers and practitioners have been successfully documenting software patterns for over two decades. But the next step–building pattern languages–has proven much more difficult. This paper describes an approach for building a large pattern language for security: an approach that can be used to create pattern languages for other software domains.

We describe the mechanism of growing this pattern language: how we cataloged the security patterns from books, papers and pattern collections written by all security experts over the last 15 years, how we classified the patterns to help developers find the appropriate ones, and how we identified and described the relationships between patterns in the language. To our best knowledge, this is the largest pattern language in software. But the most significant contribution of this paper is the story behind how the pattern language is grown; it illustrates the steps that can be adapted to create and grow pattern languages for other domains.

Categories and Subject Descriptors D.2.11 Software [*Software Engineering*]: Software Architectures; K.6.5 Computing Milieux [*Management of Computing and Information Systems*]: Security and Protection

General Terms Design, Documentation, Security

Keywords Patterns; Pattern Language; Architecture.

1. Introduction

Christopher Alexander was the first to introduce the concept of a pattern language. He has inspired many computer scientists to develop pattern languages for software, but so far we have not produced a result that is as impressive as his. This paper describes our experiences developing a pattern language for security. We did not start out to develop a pattern language, but instead just collected patterns related to security. We found many security pattern documented by various authors, who were targeting different audiences and trying to solve different security problems. One of our first tasks was to eliminate duplication. We also wanted to unify the different collections of patterns without rewriting them. Soon we realized that the only way to make our collection of patterns useful to practitioners was to augment the patterns with guidance for using them in practice.

Our first approach was to categorize all security patterns into distinct categories to make it easier to find the patterns relevant to a specific problem. We tried several different ways to categorize patterns and we documented that process in our earlier paper [15]. However, we realized that even the most precise categorization does not provide sufficient help to practitioners. A good categorization enables the user to *identify* a set of patterns that are applicable to a specific problem. But categorization fails to *guide* the user in selecting the correct pattern (or sequence of patterns) from the set of related patterns in the same category.

When we tried to organize our catalog, we realized that the organizational properties of a pattern language can also provide guidance. Rather than focusing on categories, a pattern language shows connections between patterns. There are many different types of connections, many of which are only relevant to a few patterns. Pattern languages replace the rigid focus on categories or types by considering individual patterns first, and expressing different connections between them. In particular, we discovered that the very act of trying to draw diagrams of our pattern language helped to design the language.

This paper describes a pattern language for security, which has 96 patterns. It documents the steps we took, the decisions we made and discarded in the process, and other insight we acquired while developing the pattern language. These lessons are not limited to security: they can be applied to organize patterns in other domains of software.

2. Pattern Languages

A pattern language is the set of patterns used by a designer or by a group of designers. It corresponds to a design style, or to a design school. Thus, one could talk about the pattern

Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1562-3/12/10. . . $10.00

language of Frank Lloyd Wright or Mies van der Rohe. When we refer to a pattern we can either mean an artifact that shows the pattern or a document that describes it. Similarly, when we refer to a pattern language we can either mean the language used by a designer or a community of designers, or we can refer to a document that describes the pattern language. In this paper, we always refer to the document.

Alexander's book "A Pattern Language" [3] describes a pattern language for making buildings that he thought would achieve "The timeless way of building" [2]. He came up with this pattern language by examining buildings all over the world, by discussing the patterns and writing them with a small group of collaborators, and then by trying them out on a variety of projects. In his opinion, he was not really inventing a pattern language but instead was discovering the pattern language used by people through the ages to create livable spaces.

Alexander has inspired a number of computer scientists to try to describe pattern languages for software design. But Alexander's pattern language is much larger than anything so far written for software. "A Pattern Language" has 253 patterns ranging from very high level patterns that describe the organization of a country into cities and towns to detailed patterns describing the trim on doors and windows. In contrast, the pattern languages for programming described so far are a fraction of this size and focus on narrow topics. For example, the first published patterns language for programming was in "Documenting Frameworks with Patterns" [24], which gives 10 patterns for designing graphical editors using HotDraw. The largest pattern language we know is Hanmer's "Patterns for Fault Tolerant Software" [18] with 63 patterns.

Many collections of patterns are larger than this (e.g. Fowler's enterprise patterns book [11] has 52 patterns), but they are generally not considered pattern languages. The authors of these collections usually call them catalogs, because they do not have the structure that Alexander associated with pattern languages, and they do not claim to be complete. Alexander said that there was a relationship between patterns, that some patterns lead to others. He organized the patterns in the language in the order that they should be considered. Although Alexander opposed master plans, his pattern language encouraged a kind of top-down design. Thus, there are "top patterns" and "bottom patterns". In contrast, in a catalog like "Design Patterns" [12], patterns are mostly at the same level, and there is no top or bottom. "Pattern Oriented Software Architecture" [7] described three levels of patterns, with one level clearly the top and another clearly the bottom, but there was no clear relationships between individual patterns.

Alexander's work remains an inspiration to researchers of software patterns, but it is important to remember that building architecture is different from software. Buildings exist in a three-dimensional space, though designers often must take a fourth dimension (time) into account. Alexander's patterns simplify the problem by either eliminating a dimension or making a dimension smaller, such as by breaking up a house into rooms. Software has more dimensions, and they are not well defined. Most software patterns do not eliminate dimensions, but rather redistribute the original problem among different dimensions. Software patterns cover many more dimensions, each with its own pattern language. There is a pattern language for object-oriented design, and a pattern language for security, and another pattern language for performance.

2.1 Diagramming a pattern language

"A Pattern Language" [3] used diagrams to describe the relationship between patterns. In these diagrams, the patterns are organized in a directed acyclic graph, from most general to most concrete; an arrow from one pattern to another represents a (structural and/or temporal) refinement.

Majority of pattern papers at the early PLoP conferences did not have diagrams, even though many described pattern languages or parts of pattern languages. The papers that did include diagrams follow the Alexandrian form. One often-cited example describes a pattern language for fault-tolerance patterns [1]. It uses lines (not arrows) to show connections between the 8 patterns in the paper. Another early example is a pattern language for client/server frameworks [39], which includes an Alexandrian-style diagram of 15 related patterns. One interesting aspect of their diagram is that the sequential progression through the patterns culminates in few patterns at the bottom, so the entire diagram looks like a circle rather than a triangle, which is the typical shape of a hierarchy. Moreover, this pattern language includes design patterns [12] in addition to new patterns described in the paper. The Alexandrian diagrams have been used in later patterns as well, e.g., in the C++ idioms pattern language [8] and the pattern language for fault-tolerant software [18].

Most pattern diagrams are similar to the design patterns book and rely on textual annotations rather than plain arrows and hierarchy. The meaning and expressiveness of the annotations varies considerably: they might be verb phrases, noun phrases, or generic terms such as *uses* or *extends*. One example of a pattern language diagram is in the book on domain driven design [9], where the pattern language is divided into 3 parts with the two essential patterns (Ubiquitous Language and Model-Driven Design) serving as the only connection points between them. Although the patterns are written using the Alexandrian style, the diagram is not. The arrow connections have textual annotations and some closely-related patterns are bunched together (shown as ovals). It is one of the most comprehensive diagrams and includes 24 patterns, however not all the patterns from the book are shown.

A more typical approach to presenting larger collection of patterns is to break them into smaller pieces. Schumacher et al. [31] describe 46 security patterns, divided into 8 sections. Some sections have a corresponding diagram. Unfortunately, the diagrams were produced by different authors, following different styles. It is not easy to combine them.

Text-annotated diagrams include significantly more details than Alexandrian diagrams. Heer et al. [20] describes a pattern language containing 21 patterns: some design patterns and some patterns presented in their paper. Typically for this style, almost all the annotations have unique text and although each pattern is only related to few others, the resulting diagram is quite complex. The perfectly rectangular shape of the diagram does not represent any specific quality (such as hierarchy), but rather is designed to minimize the amount of space occupied.

Design patterns [12] are the only patterns that appear consistently in other pattern languages. It is due to their popularity, but also a testament to their ubiquitous nature, proving that they are indeed *recurring solutions*. But beyond incorporating design patterns, very little work has been done to combine multiple pattern languages. Coplien has pioneered the application of Alexander's concept of pattern sequences to show examples of combining patterns from multiple pattern languages [27]. He combined C++ idioms with half object plus protocol [26] pattern language and described a procedure for adding other languages, such as fault tolerance patterns. However, we are not aware of any follow up work.

In summary, the state of the art in documenting pattern languages of software has not changed since mid-1990s. Two types of diagrams are used. The Alexandrian-style diagrams rely on hierarchy and make it easy for the reader to identify which patterns to consider at any point in the pattern language. The text-annotated diagrams pack more information, giving the reader a much quicker overview, but they lack standard meanings or reusable vocabulary for annotations and they trade off the ability to follow patterns in a sequence. In the process of deriving our pattern language for security, we attempted to explore all of these variations.

2.2 Growing a Pattern Language

We began the derivation process with the catalog of security patterns described in our earlier paper [15]. In that work, we evaluated several classification schemes that enable breaking up the large catalog into smaller clusters of related patterns. The final classification schema from that paper, STRIDE threat model [34] augmented with hierarchical view, served as the starting point for developing this pattern language.

Following the lessons from the prior work on patterns, we considered both the Alexandrian and annotation-based approaches. We quickly noticed that creating an Alexandrian-style diagram meant sacrificing readability—a diagram with 96 names is difficult to follow. Somewhat surprisingly, adding textual descriptions to arrows made the diagram easier to read by giving some instructions to follow in an otherwise unruly grouping. We decided to draw all the arrows pointing down, in Alexandrian style, meaning that more general patterns appear higher in the diagram.

Having made those decisions, we proceeded to create small pattern languages, one for each cluster of patterns corresponding to one cell in the STRIDE classification. In each grouping, we ordered the patterns temporally (i.e., in the typical order they would be applied in practice), and annotated each relevant relationship with a short description. Then, we combined the small diagrams into one large diagram, adding more inter-group relationships. Finally, we looked for missing patterns and connections, and also considered patterns outside of the security domain.

The details of this process, along with some lessons learned, are presented in the following sections.

3. Security Pattern Catalog

Security patterns provide a way of collecting and disseminating security knowledge. We have created a comprehensive catalog of all security pattern that have been published in the last fifteen years (when the first security pattern paper [40] appeared). Currently, our catalog has 96 entries. All 96 patterns are summarized in Appendix A.

3.1 Security Pattern Sources

Our security pattern catalog is a union of all security patterns that appear in many books, catalogs and papers on security patterns. It accumulates the experience of the entire security pattern community and is a fair representation of the solution domain of security.

There have been 3 books on security patterns. Markus Schumacher led a working group on security patterns to write a security pattern book [31]. This book had 46 security patterns from the domain of enterprise security and risk management, identification and authentication, access control, accounting, firewall architecture, and secure internet application. This book aggregates many previous works on security patterns, including the first catalog of security patterns [40], Markus Schumacher [32] and Eduardo Fernandez's [6, 10] work on security patterns, etc. Steel et al. [33] described 23 security patterns for J2EE based applications, Web services and identity management in their book. Microsoft's Patterns and Practices group published the third book [21] that included 18 patterns.

People have worked on pattern catalogs covering various domains. Bob Blakley and Craig Heath [5] compiled a security pattern catalog with the members of the Open Group forum. It contained 13 patterns: 5 patterns for improving reliability, and 8 for security. Kienzle et al. [25] listed 26 patterns and 3 mini patterns in their security patterns catalog. Sasha Romanosky [28, 29] compiled two pattern catalogs containing 12 patterns in total.

We have created two pattern catalogs. The first stems from our study of the evolution of mail transfer agent architecture [16]. This contains 11 security patterns, and 4 reliability patterns. These patterns are not specific to mail transfer agents; they can be used by the designers of other secure systems. Our second catalog compiles a collection of 12 privacy design patterns [14].

3.2 Building the Catalog

The pattern sources describe 174 security patterns, but with many overlaps. We developed our catalog by removing the overlaps. For example, patterns for authentication have been listed in a number of pattern catalogs under different names.

- The AUTHENTICATOR pattern, listed in conjunction with other patterns of operating systems security [10], was later included in the Wiley security patterns book [31]. The authentication protocol may depend on some user input (i.e., something that the user knows e.g., password, or possesses e.g., a smart card, or has e.g., a biometric characteristic), or the input provided by a trusted third party (i.e., brokered authentication).
- Sasha Romanosky described the same pattern under the name SECURITY PROVIDER [28].
- Kienzle et al. lists it as AUTHENTICATED SESSION [25] in their catalog.
- The Core Security Patterns book described the AUTHENTICATION ENFORCER pattern [33] from Java tool-oriented perspective. This pattern creates a centralized authentication enforcement component that performs authentication of users and encapsulates the details of the authentication mechanism.
- Microsoft security patterns book [21] lists this pattern as DIRECT AUTHENTICATION [21]. The authors described authentication mechanism using WSE 3.0 framework.

We have one AUTHENTICATION ENFORCER pattern for authentication, removing language-specific and catalog-specific descriptions of this pattern. However, we did not remove the overlapping patterns in separate application contexts. For example, AUTHENTICATION ENFORCER and INTERCEPTING WEB AGENT [33] are both about authentication: one in a single application context, and the other in single sign on context. These are separate patterns in the catalog.

3.3 Attributes of the Catalog

Cataloging is the first step of growing a pattern language in a domain. We exhaustively covered all the related works in security patterns to build the catalog. Our catalog does not have the bias of one person or a group. It spans the security pattern literature of fifteen years and includes both classic and emerging security solutions. Some of the patterns may not have the most descriptive names; but we retain the names so that they can be easily traced to the sources.

The main criteria of including a security pattern in the catalog was the maturity of its description, and the importance of the security problem and solution. Many of the patterns in the catalog are overlapping—they are described in more than one source. For these patterns, we chose the one with the most mature description. The overlapping indicates that these are important patterns; hence they are identified by many people during their independent explorations of patterns.

Stakeholder	Viewpoint / Role	Motivation	People	Data	Function	Network	Time	Scorecard
Business Architecture		SECURITY	NEEDS	IDENTI-	FICATION	FOR	ENTERPRISE	ASSETS
Integration Arch.	Enterprise Architect				2 SINGLE SIGN ON			
Application Architecture	Architect			2 ERROR DETECTION & CORRECTION	23 SINGLE ACCESS POINT	1 SECURE COMMUNICATION		
	Designer			13 ENCRYPTED STORAGE	31 SERVER SANDBOX	7 STATEFUL FIREWALL		
	Developer			1 SAFE DATA STRUCTURE WHITE	HATS		HACK	THYSELVES
Operational Arch.	System Architect	LOW		HA	NG	ING		FRUIT

Table 1. Classification of Security Patterns using the Enterprise Architectural Space Organizing Table

Our next step was to organize the patterns in the catalog.

4. Organizing Security Patterns

A catalog of 96 patterns is overwhelming. It is hard for developers to find a pattern they need, let alone know when to use it. We attempted to organize the security patterns [15] by grouping them into small, correlated sets. We applied several schemes and found that a hierarchical classification scheme using threat models worked the best. This section summarizes our classification approach.

Good organization also facilitates navigation; it cross-references the related patterns and guides a user among them so that the user knows how to use a pattern. The classification scheme facilitates lookup; it is not a guide. But classifying the patterns based on threat models (i.e., the problem domain of patterns) groups related patterns. We derive a pattern language by exploring pattern relationships in small groups, and then unifying these smaller pattern languages. Section 5 elaborates this.

4.1 Approaches for Classifying Security Patterns

Recognizing the importance of organizing patterns into small groups, security pattern authors attempted various classification schemes. The Open Group pattern catalog [5] is classified into two broad groups based on applicability: patterns for protected systems and patterns for available systems. Kienzle et al. [25] groups their patterns in two broad classes: structural patterns and procedural patterns. We thought that these partitionings are too broad to be useful.

We first tried to classify the patterns using domain concepts, confidentiality, integrity and availability, but found that they were too general. Also, most of the patterns are about more than one concept. Another attempt was to use the application context, i.e., classifying based on the part of the system that a pattern secures—application core, application perimeter and application exterior. There are fewer overlaps using this scheme. The higher level patterns relate to more than one application context, and they cannot be classified. This outlined the need for a hierarchical classification scheme. We integrated this scheme with threat models to create a hierarchical classification scheme in the section 4.2.

But before that, we applied a classification scheme based on the Zachman framework, which was used by other security pattern authors [19, 22, 31]. Zachman framework [41] was introduced in 1987 as a table. Its 5 rows describe the levels of information model from the perspective of various stakeholders, e.g., the customer or the owner, the designer, the builder etc. The 6 columns describe different ways of describing an artifact: data (what?), function (how?), network (where?), people (who?), time (when?) and motivation (why?).

We have used a classification scheme based on the Zachman framework. Our tabular classification scheme [15] uses the 'Enterprise Architectural Space Organizing Table' [36]. Out of the 7 columns of the table, 6 are from the Zachman framework table. The seventh column includes patterns for testing. The rows describe stakeholders similar to the Zachman framework, but they have been specified in roles of finer granularity using the architectural standards description from IEEE 1471 [23] and 'Enterprise Architecture Framework' [13].

Table 1 shows how the security patterns are classified using this tabular scheme. It shows only relevant rows of the larger table. As an example of classification, consider the SAFE DATA STRUCTURE [17] pattern. This pattern is applied to remove the array bounds checking vulnerability in a programming language with no garbage collection. A system written in C is vulnerable to buffer overflow attacks because of unsafe array operations, e.g., unsafe string handling. SAFE DATA STRUCTURE advocates the inclusion of length and allocated memory information with a data structure. This pattern is considered in the development phase of an application when the safe string processing libraries are written or re-used. Hence this pattern fits into the cell defined by the *Developer* row of *Application Architecture* perspective and the *Data* column.

The tabular scheme can not classify 16 patterns because they fall into multiple cells in the table. For example, the SECURITY NEEDS IDENTIFICATION FOR ENTERPRISE ASSETS [31] pattern is used by various roleplayers in *Business Architecture* and covers all viewpoints. Thus the pattern straddles many rows and columns.

Another problem of the tabular classification scheme is the skewness, i.e., patterns are concentrated in a few cells of the table. Table 1 lists 56 out of 80 patterns in the *Function* column; 16 patterns that are not in the table are also related to the *Function* column. This is because most software

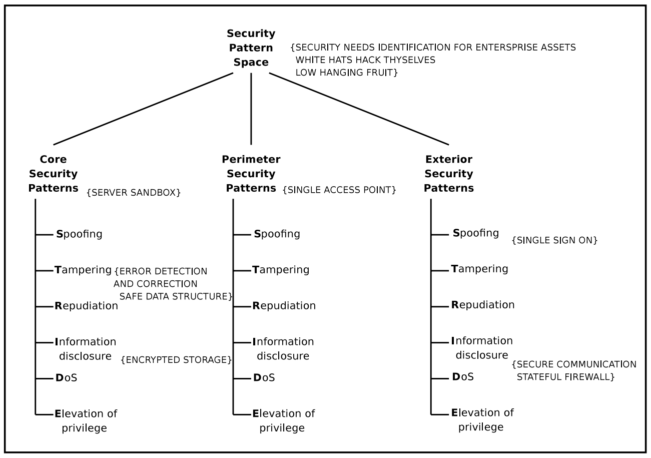

Figure 1. Hierarchical Classification Scheme

patterns provide a mechanism to achieve some functionality. A pattern classification scheme that considers *Function* as a viewpoint is skewed.

The tabular classification scheme fails to distinguish low level patterns from high level patterns, that cover multiple perspectives and are used by multiple stakeholders. Its use of perspectives and viewpoints concentrate most of the patterns in a few cells. We improved this by adopting a classification scheme that uses domain knowledge (threat model) and embraces hierarchy.

4.2 Classifying Security Patterns using Threat Model

We used a hierarchical scheme based on threat models in combination with the tabular scheme for classifying security pattern. A hierarchical scheme can classify and distinguish low level patterns from high level patterns; using domain vocabulary (threat models) makes a more intuitive classification. Furthermore, this classification scheme created small groups of patterns that solve similar security problems. This formed the basis for growing a pattern language in the next section.

Figure 1 shows our hierarchical classification scheme and how it classifies the security patterns listed in Table 1. Our tree-based scheme allows us to classify a pattern by placing it on a leaf node (low level patterns) as well as internal nodes (high level patterns), therefore creating the hierarchy. Patterns at the root of the tree are applicable to multiple contexts. The application contexts (core, perimeter and exterior) are in the internal nodes. Each context may have several security problems, categorized by the STRIDE threat model [34]. STRIDE is an acronym of the following concepts.

- *Spoofing*. Attempt to gain access to a system using a forged identity. A compromised system would provide an authorized user access to sensitive data.
- *Tampering*. Corruption of data during communication through the network. Integrity of the data is under threat.

Part of System \ Threat Model	Spoofing	Tampering	Repudiation	Information Disclosure	DoS	Escalation of Privilege	Multi-purpose Patterns
Core		Checkpointed System chroot Jail Content Dependent Processing Error Detection and Correction Safe Data Structure Trust Partitioning Unique Location for Each Write Request	Audit Interceptor Secure Logger	Container Managed Security Client Data Storage Directed Session Dynamic Service Management Encrypted Storage Exception Shielding Full Access with Errors Limited Access Obfuscated Transfer Object Policy Delegate Reference Monitor Secure Session Object Security Session Subject Descriptor	DoS Safety Small Processes	Compartmentalization Controlled Object Factory Controlled Process Creation Controlled Virtual Address Space Distributed Responsibility Execution Domain Secure Resource Pooling	Multilevel Security Secure Service Facade Server Sandbox
Perimeter	Authentication Enforcer Account Lockout Brokered Authentication Credential Tokenizer	Intercepting Validator Message Inspector Message Intercepting Gateway Secure Message Router		Authorization Enforcer Role Based Access Control			Policy Enforcement Point Single Access Point Single Threaded Facade
Exterior	Assertion Builder Intercepting Web Agent Message Replay Detection Network Address Blacklist Password Synchronizer Single Sign On Single Sign On Delegator			Anonimity Set Batched Routing Chaining Constant Length Padding Cover Traffic Integration Reverse Proxy Layered Encryption Link Padding Morphed Representation Oblivious Transfer Packet Filter Firewall Pseudonymous Identity Proxy based Firewall Random Exit Random Wait Stateful Firewall Secure Communcation Security Association Security Context			Demilitarized Zone Front Door Information Obscurity Protection Reverse Proxy Replicated System Secure Service Proxy Standby Tandem System Trusted Proxy
Higher Level Patterns	Asset Evaluation Defense In Depth Enterprise Security Approaches Enterprise Security Services Hidden Implementation Low Hanging Fruit Minefield Policy Protected System Risk Determination Security Needs Identification for Enterprise Assets Vulnerability Assessment White Hats Hack Thyselves						

Table 2. Classification of Security Patterns using Hierarchical Scheme with Threat Models

- *Repudiation.* Users refusal to acknowledge participation in a transaction.
- *Information disclosure.* Unwanted exposure and loss of confidentiality of private data.
- *Denial of service.* Attack on system availability.
- *Elevation of privilege.* Attempt to raise the privilege level by exploiting some vulnerability. Confidentiality, integrity and availability of a resource are under threat.

Table 2 lists all 96 patterns using the hierarchical classification scheme. Some cells of the table are empty. This may indicate 2 things: 1) not all threats are relevant for all contexts, and 2) not all patterns have been discovered. Perhaps, future pattern miners will look at this table for inspirations.

At the core, application developers are primarily concerned about three threats: tampering, information disclosure, and privilege escalation. Patterns that protect from tampering are about validating user inputs, running processes inside a sandbox, and ensuring that a system can restart gracefully from a compromise. Core patterns that prevent information disclosure describe how to create and manage user sessions, and how to hide information and control access. Core patterns for preventing privilege escalation vulnerabilities are about enforcing least privilege principle [30] in the process level. Table 2 also lists 2 patterns each for preventing repudiation and denial of service. Patterns for preventing repudiation are about collecting and managing logging and auditing information. Monitoring everything that is happening in a running system is important for management and forensics; these two patterns are therefore referenced by many other patterns about security and other domains. Patterns for preventing denial of service attacks are about monitoring and parsimoniously using system resources, so that a resource exhaustion attack cannot happen.

3 patterns are listed on the rightmost column on core context; these are patterns that are in the 'core' internal node of the hierarchical classification. SECURE SERVICE FAÇADE is about creating a single point for implementing various types of security checks. The actual policies implemented can target various types of security threats, e.g., tampering and information disclosure. MULTILEVEL SECURITY is about implementing a policy model such as the Bell-LaPadula model [4] to prevent information disclosure, tampering and escalation of privilege.

The higher level patterns in perimeter security context are about minimizing the number of access points in a system and creating a policy enforcement point, where authentication, authorization, and input validation policies can be enforced. Consequently, the leaf patterns for perimeter security are about implementing authentication mechanism to prevent spoofing, authorization mechanism to prevent information disclosure, and input validation mechanism to prevent tampering.

The external security patterns prevent two types of attacks. By implementing a single sign on mechanism, the patterns prevent identity spoofing, and allow users to reuse their credentials to access multiple services. Patterns for preventing information disclosure groups patterns from two domains. There are 8 patterns for implementing various types of gatekeeper mechanism to prevent external users from entering the system. There are also 12 patterns that are about implementing privacy enhancement technologies (PETs). The higher level patterns in this context are about creating infrastructure for firewalls.

Repudiation and denial of service threats only have patterns in the core context. This is because logging and auditing are activities that are implemented in the core of an application, but are used throughout the application. On the other hand, there are several solutions for preventing external denial of service attacks. However, most of these solutions are not about software applications. We are not aware of patterns that describe these solutions.

13 patterns in the bottom of Table 2 are higher level patterns, classified at the root of the hierarchical scheme. These patterns address many security threats and may necessitate changes in all parts of an application.

5. A Pattern Language for Security

The hierarchical classification scheme groups patterns that target similar problems at an application context. Naturally, these patterns should be related. We developed a pattern language for our pattern catalog by first exploring relationships among the patterns in the small scale. These are then unified to create the larger pattern language.

A pattern language offers the reader a guidance in selecting the next pattern to consider. It shows all the closely related patterns. But how does one determine that there exists a relationship between patterns? In Alexander, the pattern at the head of the connecting arrow is typically mentioned in the 'Context' section of the pattern that appears at the end of the arrow. Alternatively, the pattern at the end of the arrow is mentioned in the 'Resulting Context' of the pattern at the head of the arrow. Unfortunately, most of the security patterns were not written in that format, so we had to re-create those relationships by carefully considering the typical order in which they would be applied. We listed them in that temporal order, and then described their relationships. Typically, two related patterns either appear in a sequence, or represent two separate branches, which are related to another, more general pattern.

Appendix A contains snippets of patterns descriptions to make it easier for the reader to follow the diagrams.

5.1 Pattern Language in Core Application Context

35 core patterns are in 6 out of 7 columns in Table 2. We started with the 7 patterns that prevent privilege escalation. The first step of preventing privilege escalation is partitioning an application, described by two patterns. We related them by an arrow from COMPARTMENTALIZATION to DISTRIBUTED RESPONSIBILITY, meaning that the latter follows the former. In practice, these two patterns typically go hand in hand. A user compartmentalizes a system, then checks whether the responsibilities are properly distributed, and may compartmentalize again and so on. So, we added an arrow in the reverse direction. EXECUTION DOMAIN specializes DISTRIBUTED RESPONSIBILITY by restricting access to resources available to a process. Applications with a stronger security requirement may need finer control on the execution domain and how privilege is handled. It is especially important to check process creation (CONTROLLED PROCESS CREATION), object creation (CONTROLLED OBJECT FACTORY), and memory usage (CONTROLLED VIRTUAL ADDRESS SPACE)—all three follow EXECUTION DOMAIN. SECURE RESOURCE POOLING eases process creation by pre-forking a process pool; it also adds control on process lifetime to prevent privilege escalation vulnerabilities in long-running processes. Figure 2 shows the relationship between patterns.

We found that patterns for preventing tampering are closely related with the first 7 patterns. TRUST PARTITIONING and UNIQUE LOCATION FOR EACH WRITE REQUEST are two different ways to enforce DISTRIBUTED RESPONSIBILITY. EXECUTION DOMAIN explicitly indicates all the resources a process can use during its execution. This simplifies the creation of a CHECKPOINTED SYSTEM, that periodically saves the application state so that a system can restart gracefully. Also, limiting the execution domain makes it easy to reason about the type of data handled by the system. Especially for user inputs, it is safe to represent them using a SAFE DATA STRUCTURE.

A CHROOT JAIL pattern follows EXECUTION DOMAIN because it limits resource access. We knew that CHROOT JAIL is a refinement of SERVER SANDBOX, a multi-purpose

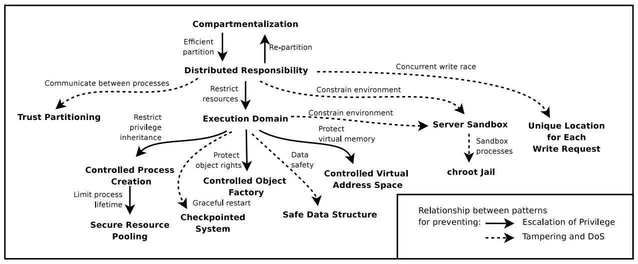

Figure 2. Pattern Language of Core Patterns for Preventing Privilege Escalation and Tampering

pattern. We added SERVER SANDBOX in the relationship. Figure 2 shows the pattern language ensuing from this pass. We could not relate these 13 patterns with 2 remaining patterns for preventing tampering: CONTENT DEPENDENT PROCESSING and ERROR DETECTION AND CORRECTION. We later found that these are related with tampering patterns at a different application context.

Next, we explored the relationship between 14 patterns to prevent information disclosure. 10 are about managing a session. Naturally, we started with SECURITY SESSION. A session may guide user activity (DIRECTED SESSION). Another way of controlling user activity is how a GUI is presented to user, LIMITED ACCESS and FULL ACCESS WITH ERRORS being two competing design choices. OBFUSCATED TRANSFER OBJECT hides session communication using cryptography, which can be applied at the client end (CLIENT DATA STORAGE) or at the server end (ENCRYPTED STORAGE). POLICY DELEGATE and DYNAMIC SERVICE MANAGEMENT are about implementing a loosely coupled security session. They are not directly related to the session management patterns. Instead, they are directly related to a multi-purpose pattern in core context, SECURE SERVICE FAÇADE, which is a refinement of SECURITY SESSION.

Among the 4 remaining patterns, SUBJECT DESCRIPTOR, REFERENCE MONITOR and CONTAINER MANAGED SECURITY are about access control. A reference monitor enforces access requests from a subject. There are no other relationships. However, we found that these patterns are related with ROLE BASED ACCESS CONTROL, which is classified in the perimeter context in Table 2. Although there are various access control mechanisms, role based access control is probably the best and most widely documented option. It influences GUI creation; so, it is related with LIMITED ACCESS and FULL ACCESS WITH ERRORS. We could not relate EXCEPTION SHIELDING directly with this set of patterns, and omitted it during this pass. REFERENCE MONITOR enforces various constraints in execution domain, e.g., process creation constraints (CONTROLLED PROCESS CREATION). There are therefore relationships between patterns listed in Figure 2 and 3, but we did not consider them until the final pattern language diagram.

One multi-purpose pattern remained in the core context. MULTILEVEL SECURITY combines role based access control with a security model. It is a refinement of ROLE BASED ACCESS CONTROL because it takes the idea of roles, organizes them in security tiers and imposes a security model on the read and write privileges between tiers. Figure 3 shows the pattern language containing 15 patterns: 13 from information disclosure column, and 2 from multi-purpose column.

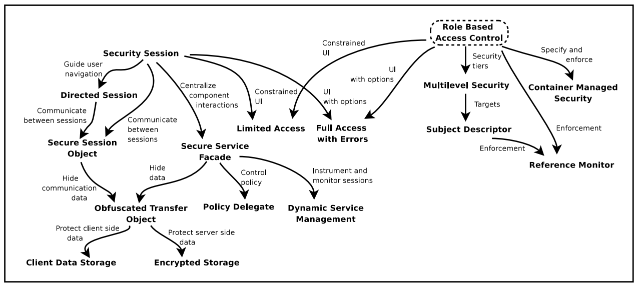

Figure 3. Pattern Language of Core Patterns for Preventing Information Disclosure

4 patterns for preventing DoS and repudiation based attacks remained. SMALL PROCESS goes hand in hand with COMPARTMENTALIZATION and act as a partitioning guideline similar to DISTRIBUTED RESPONSIBILITY in Figure 2. DOS SAFETY checks for resource exhaustion attacks. These checks are enforced by a REFERENCE MONITOR, listed in Figure 3. 2 patterns for logging and auditing are connected with many patterns in the entire pattern language, since logging and auditing is a recurrent activity. We omitted these in this pass, and only included them in the final pattern language.

5.2 Pattern Language in Perimeter Context

13 perimeter patterns are in 4 columns in Table 2. We realized that they were more closely related together by the perimeter context than by the threat-specific responses, so we explored them together. The main concern for perimeter security is to enforce authentication, authorization and input validation policies. Applications typically have separate components for performing these tasks, and a policy enforcement point acts as a MEDIATOR [12] between the components. Enforcing policies at multiple points is hard to manage. The SINGLE ACCESS POINT pattern minimizes the number of access points in a system. POLICY ENFORCEMENT POINT follows SINGLE ACCESS POINT in the pattern language. Access points denote the perimeter processes. To make perimeter secure, these processes should be simplified by making them single-threaded (SINGLE-THREADED FAÇADE).

Authentication, authorization and input validation policies are enforced by the policy enforcement point; they are therefore related. AUTHENTICATION ENFORCER introduces an authentication component. Upon successful authentica-

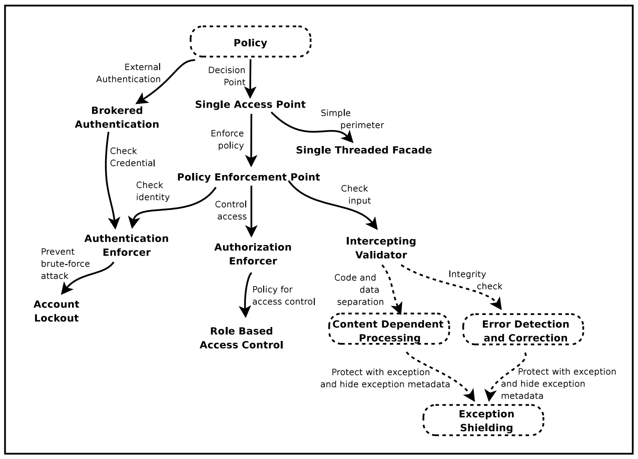

Figure 4. Pattern Language of Perimeter Patterns

tion, it creates a SECURITY SESSION, a core pattern. ACCOUNT LOCKOUT pattern is applied to an authentication component to protect the component from brute-force attacks. The AUTHORIZATION ENFORCER pattern creates policies, such as role based access control, which are eventually refined by various core security patterns. Similarly, an interceptor component validates input using core patterns CONTENT DEPENDENT PROCESSING and ERROR DETECTION AND CORRECTION. Input validation processes often modify user input in order to remove malicious data, but this may have the unfortunate side effect of putting an application in an unsafe state. Input validation routines are therefore protected by exceptions. The EXCEPTION SHIELDING pattern also ensures that an exception handling routine do not reveal sensitive internal information to an attacker.

AUTHENTICATION ENFORCER enforces authentication requests by checking a user's credentials. This model works well if a user trusts the authentication component. If there is no trust, a user cannot present his or her credential. Instead, both parties use a mutually-trusted authentication broker. This is BROKERED AUTHENTICATION. It is implemented by the client side, since the client communicates with the broker, and eventually presents the broker-provided credentials to the authentication component. AUTHENTICATION ENFORCER therefore follows this pattern. BROKERED AUTHENTICATION is further related with POLICY, a higher level pattern; it enforces an external authentication policy.

We could not relate 3 patterns that prevent tampering, and 1 pattern that prevent spoofing with these patterns. These are related with input validation and credential management, both done at an external component.

Figure 4 shows 9 perimeter patterns as well as 3 core patterns.

5.3 Pattern Language in Exterior Context

35 perimeter patterns are in 3 columns in Table 2. We first identified the relationships between 7 patterns that prevent spoofing. 5 out of 7 patterns are related to enabling single sign on. INTERCEPTING WEB AGENT is an external authentication component, installed in a web server. It is a variant of brokered authentication mechanism described by BROKERED AUTHENTICATION, a perimeter pattern. However, INTERCEPTING WEB AGENT does not create credentials for clients, it authenticates the client and generates interoperable security tokens.

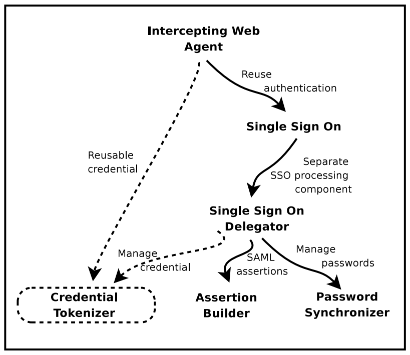

Figure 5. Pattern Language of Exterior Patterns to Prevent Spoofing

One of the main tasks of INTERCEPTING WEB AGENT is to provide single sign on capability (SINGLE SIGN ON). It is typically delegated to a separate SINGLE SIGN ON DELEGATOR. Single sign on depends on various components, especially for identity and credential management. These patterns are directly related with SINGLE SIGN ON DELEGATOR. ASSERTION BUILDER provides a service for managing and distributing SAML assertions. PASSWORD SYNCHRONIZER creates a hub for synchronizing user credentials across different application systems. CREDENTIAL TOKENIZER is a perimeter pattern that encapsulates various types of user credentials as a security token.

2 patterns, MESSAGE REPLAY DETECTION and NETWORK ADDRESS BLACKLIST, are not about single sign on. They did not have direct relationships with the other patterns that prevent spoofing. The remaining 5 patterns are shown in Figure 5.

Next, we explored the 9 multi-purpose patterns. 4 of these patterns are about implementing firewalls and proxy-based services, 2 are about secure communication, and 3 are about adding redundancy to implement fault-tolerant systems.

The first step of implementing a firewall is to create a DEMILITARIZED ZONE and enforce policy at a FRONT DOOR. The relationship between these two patterns is similar to the relationship between SINGLE ACCESS POINT and POLICY ENFORCEMENT POINT. A PROTECTION REVERSE PROXY is the firewall at the FRONT DOOR. Alternatively, a third party can be used as a TRUSTED PROXY. Among the secure communication patterns, INFORMATION OBSCURITY sets the high-level goal. A SECURE SERVICE PROXY follows the pattern by creating a secure façade. 3 patterns for fault-tolerant and available systems are not related. They are instead competing solutions. These three patterns are Replicated System, Standby and Tandem System.

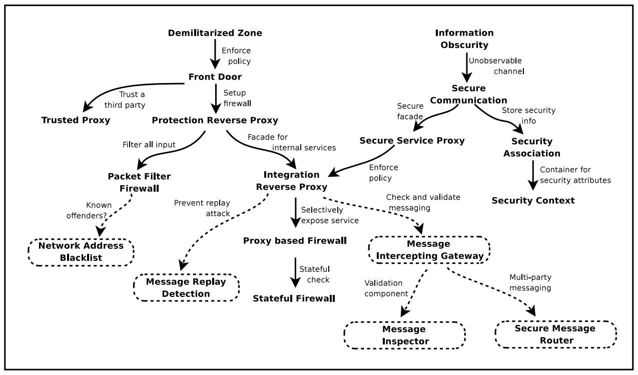

Figure 6. Pattern Language of Exterior Patterns to Prevent Information Disclosure

These three cliques of patterns are not related. However, we found 7 patterns in 'information disclosure' column that are about firewall creation and secure communication. Protection Reverse Proxy includes a Packet Filter Firewall and an Integration Reverse Proxy that hides the internal services. Packet Filter Firewall looks at incoming packets at the network layer and filters them. It accesses the services of a Network Address Blacklist, a pattern in the 'spoofing' column that was previously omitted. Proxy based Firewall filters at the application layer; it exposes a few internal services. Stateful Firewall is a further refinement of a proxy based firewall; it keeps track of requests coming in multiple sessions. Integration Reverse Proxy is the policy enforcement point for a Secure Service Proxy. It enforces policies for message replay detection and message validation. Multi-party messaging (Secure Message Router) can also be implemented by a Message Intercepting Gateway.

Secure Communication pattern lies between Information Obscurity and Secure Service Proxy, related in the previous pass. Setting up a secure communication involves multiple steps, including exchanging and storing Security Associations and keeping track of the Security Context.

Figure 6 shows the relationship between 18 patterns: 13 patterns for firewall and secure communication, and 5 patterns that were omitted in the previous passes.

We were left with 12 patterns for information disclosure prevention. These are about implementing privacy enhancement technologies (PETs). The privacy patterns are related with the multi-purpose pattern, Information Obscurity; all of them hide information. We explore two goals of information obscurity: providing anonymity in a communication scheme and getting involved in a zero-knowledge secure multiparty transaction.

We started with Pseudonymous Identity that hides anonymity targets behind a pseudonym. The pseudonymous communication is then blended with other communication and an Anonymity Set is created. A mix based network applies Morphed Representation to hide the relation between incoming and outgoing data. In online communication, mix networks adopt Layered Encryption to hide packet content. Constant Length Padding is used in conjunction with encryption to make all the encrypted packets of the same size. The defense in depth principle [38] suggests Chaining anonymity solutions, so that failure of a layer of anonymity does not expose sensitivity information. To prevent attackers from only monitoring exit nodes, a Random Exit strategy is employed with Chaining.

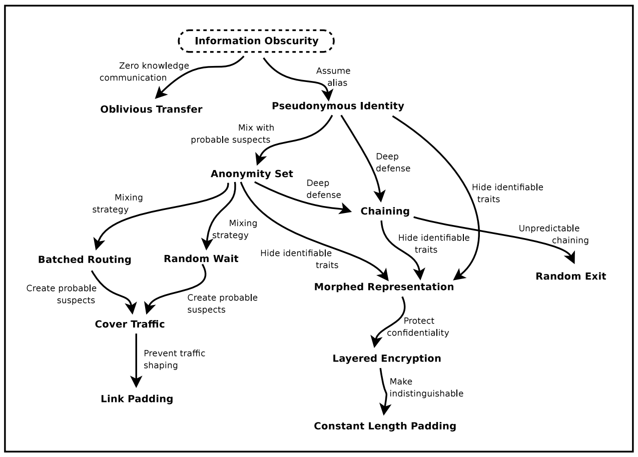

Figure 7. Pattern Language of Exterior Patterns to Prevent Attacks on Privacy

Batched Routing and Random Wait are two competing mixing strategies. Cover Traffic ensures proper mixing even in a network with low traffic. Cover Traffic is typically followed by Link Padding to create a continuous flow of dummies between nodes.

Another goal of anonymity is to enable zero-knowledge communication. Oblivious Transfer describes cryptographic operations for secure multi-part computation. Figure 7 shows the pattern language.

5.4 Higher Level Pattern Language

Higher level patterns from Table 2 are mostly about enterprise-wide security and risk management. The scope of patterns includes policies, directives, or constraints that apply to all systems and all operations across the enterprise. As we were identifying their relationships, we noticed two almost-parallel hierarchies of patterns, one for process management (e.g., Asset Evaluation, Risk Determination), the other for software (Low Hanging Fruit, Hidden Implementation, Minefield). But both sets of tasks need to be considered together, so we added relationships between them. Figure 8 shows 13 patterns and their relationships.

Security Needs Identification for Enterprise Assets is the root pattern for all enterprise security concerns. It assesses the need for security in an organization,

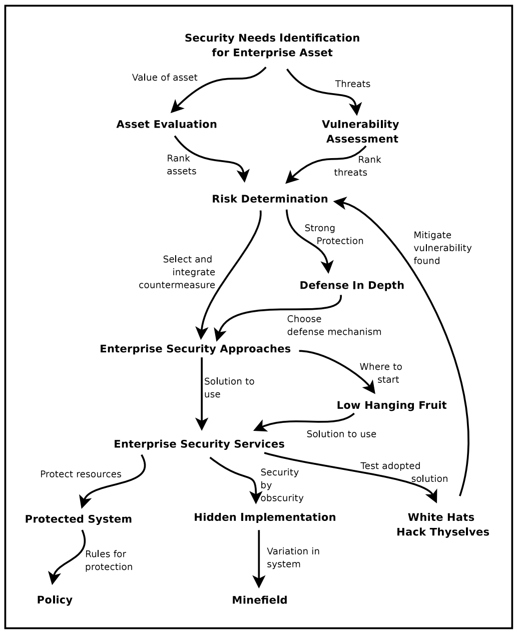

Figure 8. Pattern Language of Higher Level Patterns

which is then explored by understanding the assets and vulnerabilities in an organization. RISK DETERMINATION is the final stage of assessment process, when an organization evaluates and prioritizes the risks. A key security principle is DEFENSE IN DEPTH: having multiple security solutions in different layers of an application. This pattern guides the selection of security approaches, and eventually the choice of security services.

A core security goal is to create a PROTECTED SYSTEM, i.e., an application where the resources at the core are protected by a series of guards at the application perimeter. This leads to identifying the policies to be adopted in an application and applying them at the perimeter. Figure 4 lists the perimeter security patterns of implementing a policy enforcement point.

Hiding internal details from attackers is an important security goal. This involves hiding details at all levels, from hiding specification to hiding application status messages. MINEFIELD makes it hard for an attacker to explore internal details, because it suggests varying every instance of an application. Thus a vulnerability present in one application may not be found in another instance of the same system.

A key security activity is testing the security measures that have been adopted, and possibly modifying/upgrading systems to mitigate emerging security threats. WHITE HATS HACK THYSELVES pattern describes this process.

5.5 A Unified Pattern Language for Security

Let us look again at figures 2-7. They are relatively self-contained, because they describe solutions for different classes of security problems. But patterns in Figure 8 are fundamentally different. They explain how to approach solving any security problem, but stop short of solving any. In other words, patterns in Figure 8 leave off where all the other figures start. In order to produce a single diagram of the full pattern language, we identified the relationships between the higher level patterns in Figure 8 and the starting points of figures 2-7. This step is shown in Figure 9. Figures 2, 4, and 6 are connected directly to the ENTERPRISE SECURITY SERVICES, PROTECTED SYSTEM, or POLICY patterns from Figure 8. The remaining figures represent refinements of patterns in those 3 figures.

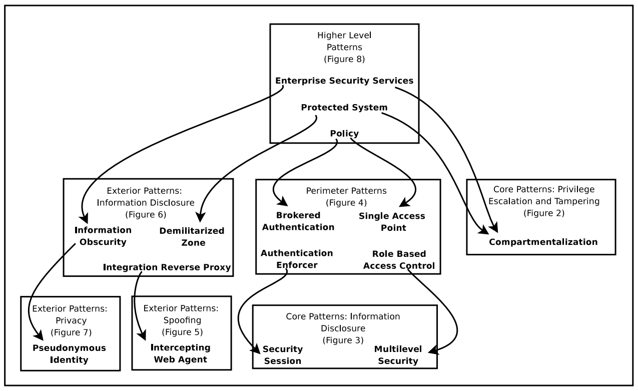
Higher Level Patterns (Figure 8)
Enterprise Security Services
Protected System
Policy
Exterior Patterns: Information Disclosure (Figure 6)
Information Obscurity
Demilitarized Zone
Integration Reverse Proxy
Perimeter Patterns (Figure 4)
Brokered Authentication
Single Access Point
Authentication Enforcer
Role Based Access Control
Core Patterns: Privilege Escalation and Tampering (Figure 2)
Compartmentalization
Exterior Patterns: Privacy (Figure 7)
Pseudonymous Identity
Exterior Patterns: Spoofing (Figure 5)
Intercepting Web Agent
Core Patterns: Information Disclosure (Figure 3)
Security Session
Multilevel Security

Figure 9. Unifying the Smaller Pattern Languages

To create the final diagram, shown in Figure 10, we expanded Figure 9 with all the contents from the other figures. Then we added some additional relationships and included patterns that were not shown in the previous figures, because they did not fit them. Here are some of the additional connections we made for this diagram.

COMPARTMENTALIZATION prevents privilege escalation attacks. It is a security service to be chosen by enterprise architects, as well as a means to create a protected system, hence connected with two patterns. Also related with ENTERPRISE SECURITY SERVICES are INFORMATION OBSCURITY pattern, and three patterns for building available systems (REPLICATED SYSTEM, STANDBY, TANDEM SYSTEM).

PROTECTED SYSTEM contains a gatekeeper component that enforces policies. The gatekeeper component can be at the application perimeter in the form of a POLICY ENFORCEMENT POINT, or implemented by a separate firewall infrastructure. The policies enforced in these two classes have some similarity. Both employ authentication and input validation. But the contexts are very different, e.g., one considers credential based authentication, while the other is about reusing authentication and single sign on.

AUTHENTICATION ENFORCER creates a SECURITY SESSION, that eventually enforces the policies. These core patterns are shown in figure 3.

We show the AUDIT INSPECTOR and SECURE LOGGER patterns at the bottom, being related with SECURITY SESSION and REFERENCE MONITOR. In practice, logging and auditing are associated with many other security solutions;

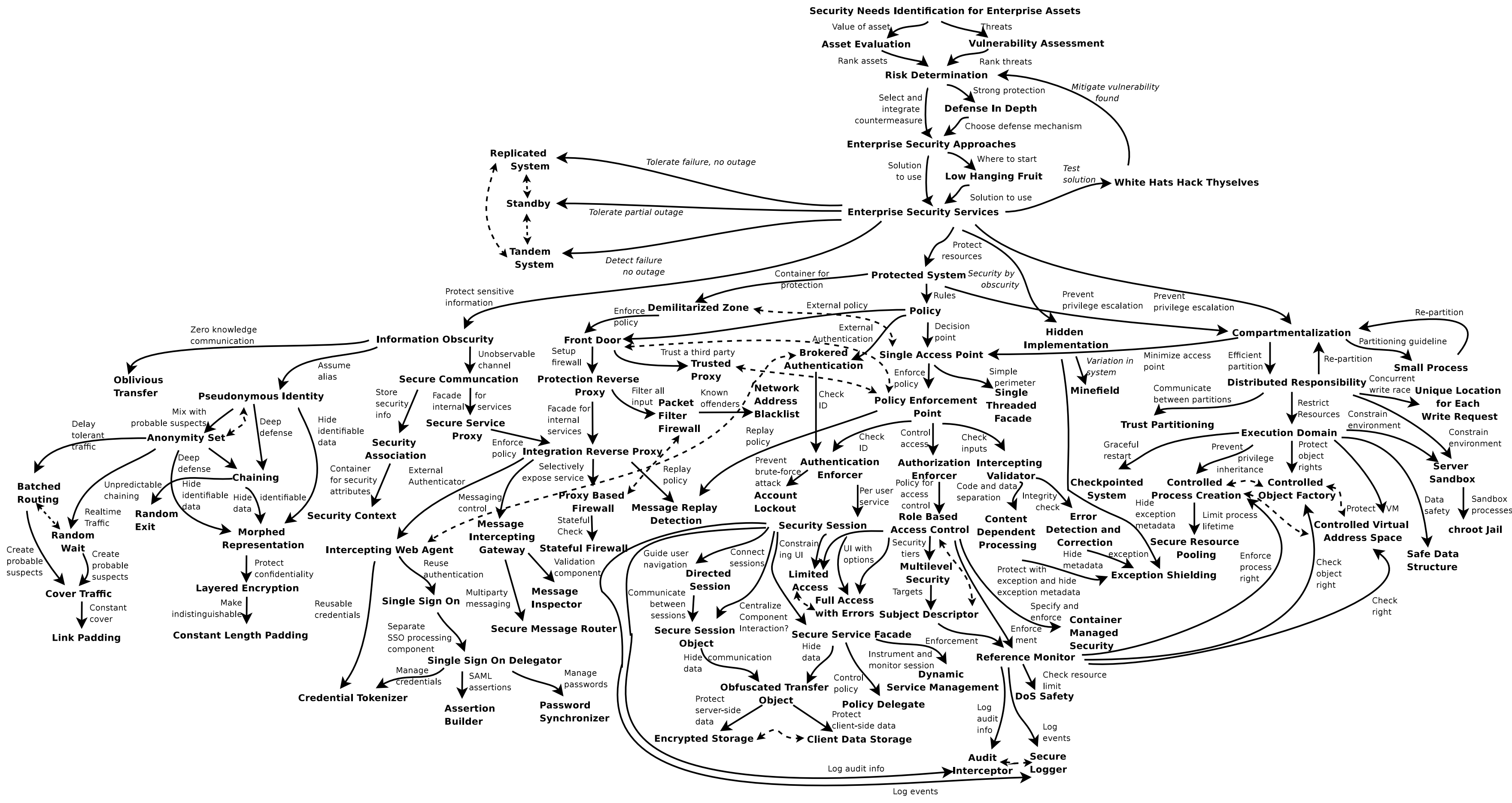

Figure 10. The Full Pattern Language

these activities are also not limited to the security domain. Hence, these patterns are typically connected with many other patterns.

6. On the Pattern Language Diagram

Reading pattern language diagrams is hard. Considered in isolation, Figure 10 is large and intimidating. But since its contents were introduced in small chunks, the reader can easily recognize the top-level patterns from smaller diagrams (e.g., INFORMATION OBSCURITY, DEMILITARIZED ZONE, or COMPARTMENTALIZATION) and quickly locate other patterns from the corresponding category. Similarly to the small diagrams, most relationships between patterns flow top-down, indicating how to apply patterns sequentially to refine the design.

The cliques of related patterns from the smaller pattern languages are preserved in the combined figure. Aside from the added relationships that show how the top-level patterns from each category are related, few arrows connect multiple original pattern language figures (e.g., MESSAGE REPLAY DETECTION). This indicates that most patterns solve only one specific problem, but each can be refined in different ways by patterns that extend it.

There are three general observations about the annotations describing relationships between patterns. First, they are very short, summarizing in a few words the essence of the relationship. The exact description of the relationship is not as important as the indication that a relationship exists. No single-phrase summary can substitute for actually reading the related pattern, but being aware that the relationship exists adds significant knowledge. Second, similarly to other work on pattern languages, the relationship descriptions we defined are unique; they summarize the change to the system that comes from applying the "target" pattern (i.e., the pattern listed at the end of the arrow). The few relationship descriptions that *are* used repeatedly point to the same "target" pattern. Third, the relationships are described as either verb phrases (e.g., "Rank threats") or noun phrases (e.g., "Threats"). While it is not difficult to turn noun phrases into verb phrases, the extra verb often adds little to clarify the description. We opted for brevity over consistency.

With such a large collection of relationships to analyze, it would be tempting to try to find commonalities and other characteristics that would enable us to define a "language of relationships between patterns". We observed three classes of relationships: (1) pattern A *refines* or *extends* pattern B, (2) pattern A *uses* pattern B to achieve a (part of its) goal, and (3) patterns A and B are *alternatives*. Most relationships between patterns fit the first two categories. (The same type of arrow depicts both types; pattern alternatives are connected by a dashed, doubly-ended arrow.) The specific instances of "uses" and extends" do not form obvious subcategories but rather stand on their own. We opted not to pursue the language approach.

7. From Pattern Catalogs to Pattern Languages

The process of creating a *Pattern Language for Security* has enabled us to better appreciate the difficulties of putting together a large number of patterns into more than a mere collection. It is difficult to grow pattern languages, because adding more patterns to a language means increasing the possible scope (or context) and having to account for many more relevant associations (or dimensions of) patterns. This explains, in part, why so few pattern languages for software feature more than 25 patterns. And if they do, as in the case of the Wiley book [31], the authors focus on smaller sections and do not try to piece them together in a single diagram. In each section, they focus only on specific types of relationships that are relevant to the patterns in that section, and they do not attempt to reconcile the differences between sections. As the length of Section 5 indicates, combining smaller diagrams into a complete one is very time consuming. If the relationships between the smaller groups of patterns are not well understood, the task becomes nearly impossible.

Large collections of patterns are hierarchical by nature. The larger, architectural, patterns appear at the top of the hierarchy, while smaller patterns that complement and refine them are shown underneath. This hierarchy is easy to spot in Figure 10. It was built bottom up from small hierarchies in the previous figures, which represent specific security contexts. It is possible to simplify the diagram by replacing all the patterns from each smaller figure with a single label or box, as we did in Figure 9. Such a hierarchical understanding of how the smaller pieces are related, while ignoring the details of the smaller pieces, makes it easier to work with pattern languages; both to write and to read them. But there is a benefit to documenting larger collections and hierarchies in detail: accumulating more domain knowledge.

Having a good classification scheme is the key in evolving a pattern collection into a pattern language. It might seem that all our previous work on categorizing patterns had little added value, because we reorganized the patterns while drawing the smaller figures. However, most of that classification *is* retained in the diagrams. Had there been no classification to start with, we would not be able to draw smaller diagrams and we would get bogged down in the details while attempting to draw the full diagram first.

The *Pattern Language for Security* represents a small portion of software patterns. In order to apply these patterns to a software system, they need to be integrated with the functional domain of the system. In the process of designing such a system, other types of pattern languages become handy. For example, implementing the EXCEPTION SHIELDING pattern requires some knowledge of user interfaces [35], building a CREDENTIAL TOKENIZER or ASSERTION BUILDER is likely to benefit from design patterns [12], while the CHECKPOINTED SYSTEM can only improve with the application of other patterns for fault tolerance [18].

These types of patterns would likely be applied *after* security patterns, in order to refine the solution even further. In contrast, higher level patterns, e.g., for requirements analysis or system engineering, would be applied *before* any security patterns. Even though security patterns in the pattern language range from very generic (and vague) to small and very specific, related patterns in other domains can be both much more generic and much more concrete. A more "complete" pattern language for building secure software systems will likely consist of patterns from all these pattern languages, and others, depending on the actual problem domain. But in the sequence of pattern applications, patterns from different domains are not mixed, but rather applied in order, one domain at a time, because a mature pattern language for a domain has no holes, and patterns from outside of the domain need to be applied only when the resulting context is outside of the scope of the pattern language.

8. Next Steps

This paper describes work in progress. The security pattern language has not yet been used on a large project, or reviewed by more than a few security experts, or used to teach students how to make secure systems. Doing these things will undoubtedly point out ways to improve it. However, its current version has already improved the state of the art.

Earlier work consisted of collections of patterns, most of which were not connected to each other. The pattern language for security shows how all the patterns are connected. Earlier work had a lot of duplication, and different collections used different notations to describe the relationships of the patterns in those collections. The security pattern language has removed overlaps and uses a single diagramming notation for the entire pattern language. We not only have a more complete set of patterns than any of the previous works, we have an approach for developing a pattern language that can be used in other domains.

There are many reasons to document a pattern language. It standardizes the vocabulary of experts, and helps them to communicate with each other. It serves as a guide to novices, helping them to learn how to design. By describing a particular way of design, it opens that way to scrutiny and improvement. Finally, as a way of design becomes more standardized, it becomes possible to automate parts of it.

Does the security pattern language match the vocabulary of experts? Often different authors use different names for the same pattern, so we had to select one. In some cases, none of the names seemed ideal but we did not want to invent a new one. We plan to have the pattern language reviewed by more experts to help refine the choice of names and to clarify their descriptions.

Each use of a pattern language leads to a way to evaluate and improve it. Using a pattern language to design a system shows missing patterns and shows when descriptions of individual patterns are incomplete. Using a pattern language to teach a course makes it clear where it is hard to understand and ways in which people are likely to misinterpret it. We intend to study more existing software systems to compare their security architectures with the sequences of patterns captured in the security pattern language. Patterns descriptions are informal since they are designed to be read by people, but automating them, whether in a design tool, a checker, or a program transformation system, requires the pattern to be formalized. This forces people to be more explicit about the pattern, and often brings some new insight.

Our next steps are to circulate the pattern language for feedback and review by a larger audience. Then we will start using it and encouraging others to start using it. We expect to continue improving it for years.

We expect a more immediate impact from our other main contribution—a detailed description of the process of creating a pattern language. We have shown how to integrate many patterns from several disparate source into a full pattern language. Our approach can serve as a recipe for designing pattern languages in other domains. While the details—pattern names, descriptions of relationships, or the fidelity of relationships—are open to debate, our approach is sound. We encourage other researchers to follow our approach.

9. Onward

Growing a pattern language takes time. We have been studying security patterns for over 7 years and this pattern language is not finished. Let's summarize the main steps.

Start with a comprehensive survey of relevant patterns. It is natural to encounter duplication, overlap, and seemingly irreconcilable differences in scope or style. Prepare summaries of all patterns in a common format to ensure you have good understanding of the content. There is a fine line between overly-simplistic summaries and paraphrasing the entire documents, so try out different formats. Explore various classification and organization schemes relevant to the domain of the language (like threat modeling for security patterns) in order to categorize the patterns in different ways. Organize patterns in each category into small pattern languages with all relationships made explicit. Look for links between patterns in different categories while incorporating them into the complete pattern language. In time, you will see structure.

Going forward, we imagine seeing larger, unified pattern languages bringing "the nature of order" to software design. Security is only a small part of software design, yet our pattern language for security is already too big to be easily understood. Other non-functional domains, such as reliability, performance, or usability are just as complex; so are functional domains. It will take a concerted effort on the part of all software practitioners to organize that body of knowledge, but if we are ever to match Alexander's vision, we must keep trying.

References

[1] M. Adams, J. Coplien, R. Gamoke, R. Hanmer, F. Keeve, and K. Nicodemus. *Pattern Languages of Program Design 2*, chapter 33: Fault-Tolerant Telecommunication System Patterns. Addison-Wesley, 1996.

[2] C. Alexander. *The Timeless Way of Building*. Number 1 in Center for Environmental Structure series. Oxford University Press, New York, 1980.

[3] C. Alexander, S. Ishakawa, and M. Silverstein. *A Pattern Language: Towns, Building and Construction*. Oxford University Press, New York, 1977.

[4] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report ESD-TR-73-278, MITRE Corporation, 1973.

[5] B. Blakley and C. Heath. Security design patterns technical guide–Version 1. Technical report, Open Group(OG), 2004.

[6] F. L. Brown Jr., J. DiVietri, G. D. Villegas, and E. B. Fernandez. The authenticator pattern. 1999.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley series in Software design patterns. John Wiley & Sons, 1996.

[8] J. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.

[9] E. Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] E. B. Fernandez and J. C. Sinibaldi. More patterns for operating systems access control. In Proceedings of the European Conference on Patterns Language of Programming (EuroPLoP'03), 2003.

[11] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[13] M. Goodyear, editor. *Enterprise System Architectures: Building Client Server and Web Based Systems*. CRC Press, Sep 28 1999.

[14] M. Hafiz. A Pattern Language for Developing Privacy Enhancing Technologies. *To appear in Software—Practice and Experience*, 2012.

[15] M. Hafiz, P. Adamczyk, and R. E. Johnson. Organizing security patterns. *IEEE Software*, 24(4):52–60, July/August 2007.

[16] M. Hafiz and R. Johnson. Evolution of the MTA architecture: The impact of security. *Software—Practice and Experience*, 38(15):1569–1599, Dec 2008.

[17] M. Hafiz, R. Johnson, and R. Afandi. The security architecture of *qmail*. In Proceedings of the 11th Conference on Patterns Language of Programming (PLoP'04)., 2004.

[18] R. Hanmer. *Patterns For Fault Tolerant Software*. Wiley, 2007.

[19] J. Heaney, D. Hybertson, A. Reedy, S.Chapin, T. Bollinger, D. Williams, and M. Kirwan Jr. Information assurance for enterprise engineering. In Proceedings of the 9th Conference on Patterns Language of Programming (PLoP'02), 2002.

[20] J. Heer and M. Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12:853–860, 2006.

[21] J. Hogg, D. Smith, F. Chong, D. Taylor, L. Wall, and P. Slater. *Web Service Security: Scenarios, Patterns, and Implementation Guidance for Web Services Enhancements (WSE) 3.0*. Microsoft Press, March 2006.

[22] D. Hybertson, J. Heaney, and A. Reedy. Conceptual aspects of security patterns. 2002.

[23] IEEE Std 1471-2000. IEEE recommended practice for architectural description of software-intensive systems, 2000.

[24] R. E. Johnson. Documenting frameworks using patterns. In A. Paepke, editor, *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 63–76. ACM Press, Oct. 1992.

[25] D. Kienzle, M. Elder, D. Tyree, and J. Edwards-Hewitt. Security patterns repository version 1.0. http://www.scrypt.net/~celer/securitypatterns/repository.pdf, 2002.

[26] G. Meszaros. *Pattern Languages of Program Design 1*, chapter 8: Pattern: Half-object + Protocol (HOPP). Addison-Wesley, 1995.

[27] R. Porter, J. O. Coplien, and T. Winn. Sequences as a basis for pattern language composition. *Science of Computer Programming*, 56(1-2):231 – 249, 2005.

[28] S. Romanosky. Security design patterns part 1. http://citeseer.ist.psu.edu/575199.html, Nov 2001.

[29] S. Romanosky. Enterprise security patterns. http://citeseer.ist.psu.edu/romanosky02enterprise.html, 2002.

[30] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sep 1975.

[31] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley and Sons, December 2005.

[32] M. Schumacher and U. Roedig. Security engineering with patterns. In Proceedings of the 8th Conference on Patterns Language of Programming (PLoP'01)., 2001.

[33] C. Steel, R. Nagappan, and R. Lai. *Core Security Patterns : Best Practices and Strategies for J2EE(TM), Web Services, and Identity Management*. Prentice Hall PTR, Oct 2005.

[34] F. Swiderski and W. Snyder. *Threat Modeling*. Microsoft Press, 2004.

[35] J. Tidwell. *Designing interfaces : Patterns for Effective Interaction Design*. O'Reilly, 2005.

[36] D. Trowbridge, W. Cunningham, M. Evans, L. Brader, and P. Slater. Describing the enterprise architectural space. *MSDN*, June 2004.

[37] R. Veryard and A. Ward. Trusting components and services, 2001.

[38] J. Viega and G. McGraw. *Building Secure Software: How to Avoid Security Problems The Right Way*. Addison-Wesley, 2002.

[39] K. Wolf and C. Liu. *Pattern Languages of Program Design 1*, chapter 4. New Clients with Old Servers: A Pattern Language for Client/Server Frameworks. Addison-Wesley, 1995.

[40] J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In Proceedings of the 4th Conference on Patterns Language of Programming (PLoP'97)., 1997.

[41] J. A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3), 1987.

A. Security Patterns in Our Catalog

- ACCOUNT LOCKOUT [25]. How can we protect a system's login mechanism from password guessing attacks?
  Limit the number of incorrect password entry attempts. Once a pre-defined threshold value of consecutive failures is reached, lock the account.
- ANONYMITY SET [14]. How can we retain the anonymity of an entity or a personal information?
  Mix the private information with other data so that the private information is not distinguishable. Create a set of equally probable data and hide the user information by making it a part of the set.
- ASSERTION BUILDER [33]. How can we keep the security information about a subject and use it for single sign on?
  Encapsulate the processing control logic in order to create SAML authentication statements, authorization decision statements and attribute statements. Expose the assertion builder as a service.
- ASSET EVALUATION [31]. How can an enterprise determine the overall value of its assets?
  Systematically determine the overall value of the assets identified in the scope of the risk assessment. Determine the financial value and security value of assets as well as impact to business.
- AUDIT INTERCEPTOR [33]. How can we easily support additions or changes to the auditing events in an application?
  Intercept requests and responses in the business tier. Specify which events to audit outside of the application, so they can be re-configured at run-time.
- AUTHENTICATION ENFORCER [31]. How can we prevent impostors from accessing the system?
  Create a single point of access where all requests to enter the system are checked and apply an authentication protocol to verify the identity of the agent. On successful authentication, create a proof of identity of the agent.
- AUTHORIZATION ENFORCER [31]. How can we specify access rights to specific resources in a system?
  For each active entity that can access resources, specify which resources it can access and with what rights.
- BATCHED ROUTING [14]. How can the output of a mix node hide timing information?
  Collect the input data packets from multiple nodes. When the collection reaches a threshold, output all the data packets together, in a batch.
- BROKERED AUTHENTICATION [21]. How can an application authenticate when a client does not have a direct trust relationship with it?
  Use an authentication broker that both parties trust to independently issue a security token to the client. The client can then present credentials, including the security token, to the authenticating application.
- CHAINING [14]. How can anonymity solutions be strengthened to have defense in depth?
  Adopt multiple instances of an anonymity mechanism in layers. Chain multiple mix nodes instead of one and route traffic through multiple nodes. Adopt multiple pseudonym covers to protect users.
- CHECKPOINTED SYSTEM [5]. How can we recover and restore system state to a known valid state when its component fails?
  When the system is running, save its current state periodically and store this information outside of the running system. When the system fails, restore its current state from the backup.
- CHROOT JAIL [16]. How can we design a system so that a security compromise in one process does not affect other processes?
  Run the processes under separate, least privilege user ids. Run the programs/processes in a controlled environment with limited access to system files.
- CLIENT DATA STORAGE [25]. In case when server data needs to be stored on the client, how can we protect the data from unauthorized access by the client?
  Encrypt the data on the server before passing it to the client. Keep a hash value of the data to detect if the content is tampered with. Change the session key often to protect against guessing attacks.
- COMPARTMENTALIZATION [38]. How can we prevent a security failure in one part of a system from being exploited in another part of the system?
  Separate the system into unique security domains, e.g., multiple processes that run at different privilege levels.
- CONSTANT LENGTH PADDING [14]. How can we prevent attackers from correlating packets that enter a node with packets that exit it?
  Make all the sent data packets have the same length. Add dummy padding at the end of the packet, after the payload.
- CONTAINER MANAGED SECURITY [33]. How can we add security roles declaratively to an application?
  Use standard security features provided by the application container. Define application-level roles at development time. Map these logical roles to users in the deployment environment at deployment time.
- CONTENT DEPENDENT PROCESSING [16]. How can a mail program be made secure so that the message content cannot be used maliciously?
  Treat the received contents as mail message only and do not perform any processing on them. When treating programs and files as addresses, minimize the risk by executing them as a less privileged user.
- CONTROLLED OBJECT FACTORY [31]. How can we ensure that objects and resources created by a program do not automatically inherit all the rights of their creator?

Create new objects with limited rights. Intercept new object creation requests and force the requester to fully specify the rights to be associated with the new object.

- CONTROLLED PROCESS CREATION [31]. How can we ensure that processes created during a program execution are not granted too many access rights?

  Create child processes with a subset of privileges of their parent process. Have the parent process assign the privileges of the child processes explicitly.

- CONTROLLED VIRTUAL ADDRESS SPACE [31]. How can we ensure that programs cannot access memory of other programs?

  Divide the virtual address space into segments according to logical units in the programs. Use descriptors to indicate access rights.

- COVER TRAFFIC [14]. How can we preserve anonymity of data when network traffic is slow?

  Create dummy packets that look like real data packets. Keep dummy traffic flowing between anonymity-preserving nodes to act as decoy for actual data traffic.

- CREDENTIAL TOKENIZER [33]. How can we use different types of security tokens in a system?

  Encapsulate different types of user credentials as a security token that can be used by different security providers. Define a security API for creating and retrieving the user information from given user credentials.

- DEFENSE IN DEPTH [38]. How can we make the system robust against all types of security failures?

  Employ various security measures at multiple layers of an application and throughout its operating environment.

- DEMILITARIZED ZONE [31]. How can we protect the systems from direct attacks?

  Define a physical or logical subnetwork that contains and exposes an organization's external services to a larger untrusted network. Separate it from the internals of the system with a firewall.

- DIRECTED SESSION [25]. How can we prevent the attacker from guessing URLs in a session?

  Expose a single URL to the user. Provide access to all pages relevant to a session from that URL and guide the user's path throughout the session.

- DISTRIBUTED RESPONSIBILITY [37]. How can we mitigate the effects of one portion of the system becoming compromised?

  Partition responsibility across system components such that the components that are likely to fail do not have critical data. Ensure that several components need to fail in order for the whole system to fail.

- DOS SAFETY [16]. How can we minimize the effects of the denial of service attack on a system?

  Set limits on the use of critical resources. Adopt resource management features of the operating system or apply per-process resource management.

- DYNAMIC SERVICE MANAGEMENT [33]. How can we proactively prevent intrusions by modifying objects or invoking operations before an attack can conclude?

  Enable fine-grained dynamic instrumentation of business objects to detect and prevent improper user actions.

- ENCRYPTED STORAGE [25]. How can we protect the server data from unauthorized access?

  Encrypt the critical data before storing it on the server and decrypt it in memory before using it by the server. Use a single key for encrypting all the data and change it periodically.

- ENTERPRISE SECURITY APPROACHES [31]. How can we determine preferred security approaches for achieving the security properties of enterprise assets?

  Specify an integrated set of approaches that achieve the required security protection for each asset type. For each asset type, systematically examine a set of risk criteria to determine appropriate security approaches and their suggested business priorities.

- ENTERPRISE SECURITY SERVICES [31]. How can we select and integrate security services across the organization to support security properties using preferred security approaches?

  Specify an integrated set of security services to address identified security approaches and security properties for each asset type and for the overall organization.

- ERROR DETECTION AND CORRECTION [5]. How can we protect data in transit from corruption and tampering?

  Add redundancy to data to enable detection of and recovery from errors.

- EXCEPTION SHIELDING [21]. How can we prevent a Web application from disclosing information about its internal implementation when an exception occurs?

  When an exception occurs, send it to the client only after sanitizing its contents. Sanitize exceptions by removing sensitive information and detailed stack trace from the exception message.

- EXECUTION DOMAIN [31]. How can we protect system resources from unauthorized access?

  Define an execution environment for processes. Explicitly indicate all the resources a process can use during its execution and the type of access to each resource.

- FRONT DOOR [31]. How can we provide a single sign on for several Web applications or services integrated under a single reverse proxy?

  Keep user identity and session information in the reverse proxy. Have the proxy pass the information to all of the back-end applications.

- FULL ACCESS WITH ERRORS [31]. How can we present to the user functionality that might be partially inaccessible?

  Design the application so that users see every available option. When a user performs an operation, check if it is allowed and generate an error notification if it is not.

- HIDDEN IMPLEMENTATION [25]. How can we prevent the attacker from gathering knowledge about inner workings of the system?

  Limit the communication with clients, because any communication might reveal information about the internals of the system. Make it impossible for attackers to query the inner workings of the system.

- INFORMATION OBSCURITY [31]. How can we ensure that sensitive data of the system is protected from unauthorized access?
  Identify the most sensitive information. Obscure the more sensitive data that may be exposed to attacks by encrypting it while leaving the bulk of the application data unencrypted.
- INTEGRATION REVERSE PROXY [31]. How can we ensure a consistent Web application space while hiding the topology from the users?
  Use a reverse proxy to integrate all Web servers as back-end servers with a common host address. Map internal URIs to individual back-end server functions.
- INTERCEPTING VALIDATOR [33]. How can we ensure that input data is validated before use?
  Define pluggable filters that target different types of input attacks. Apply the filters declaratively based on the URL, so that different requests are mapped to different filter chains.
- INTERCEPTING WEB AGENT [33]. How can we retrofit authentication and authorization into an existing Web application?
  Provide authentication and authorization functionality outside of the application. Install an intercepting agent on the Web server and route incoming requests to authentication and authorization services.
- LAYERED ENCRYPTION [14]. How can we make the mix network secure against an active adversary?
  In the sending node, determine the path to the recipient node. Encrypt the payload with the symmetric key of the recipient. Encrypt that message with the symmetric key of the preceding node, adding an encrypted layer for each node in the path.
- LIMITED ACCESS [31]. How can we present the functionality that might be partially inaccessible to the user?
  Design the application so that the users see only the options they have access to. Dynamically adjust the options when the user's permissions change.
- LINK PADDING [14]. How can we prevent attackers from correlating packets that enter a node with packets that exit it?
  Distribute data traffic equally among all the outgoing nodes from an anonymity preserving node. Send dummy messages when traffic flow is too low.
- LOW HANGING FRUIT [28]. How can we remove an identified vulnerability from the system really quickly?
  Remove obvious vulnerabilities by applying simple fixes. Do not attempt to re-design the environment or reinstall applications.
- MESSAGE INSPECTOR [33]. How can we design a simple XML security implementation to process application-specific content?
  Use modular or pluggable components that can be integrated with infrastructure service components that handle pre- and post-processing of incoming and outgoing XML messages. Chain those components together to build more advanced checks.
- MESSAGE INTERCEPTING GATEWAY [33]. How can we validate XML messages at the system entry point?
  Use a proxy infrastructure to provide a centralized entry point that encapsulates access to all target service endpoints of a Web services provider.
- MESSAGE REPLAY DETECTION [21]. How can we protect a service from an attacker who replays an intercepted message?
  Uniquely identify each incoming message. Cache identifiers of incoming messages and reject messages that match an entry in the replay detection cache.
- MINEFIELD [25]. How can we hide the internal workings of COTS components from the attacker?
  Customize some visible aspects of the system so that they appear different from the standard implementation.
- MORPHED REPRESENTATION [14]. How can we obfuscate the representation of the data in a mix network?
  Change the representation of the data in the incoming packets so that they look different in the outgoing packets. On reception, decrypt the data and encrypt it using the key shared with the receiver node.
- MULTILEVEL SECURITY [31]. How can we control access in an environment with sensitive data to prevent leakage of information?
  Assign classification to users and data. Separate different institutional units into categories. Enforce confidentiality and integrity by adopting concrete security models.
- NETWORK ADDRESS BLACKLIST [25]. How can we identify a malicious user at a system access point?
  Maintain a list of network addresses that exhibit suspicious behavior. Drop requests received from a blacklisted address.
- OBFUSCATED TRANSFER OBJECT [33]. How can we transfer large data sets between components without the components getting unauthorized access to data?
  Obfuscate the data in the object that needs to be protected. Encrypt the data using an agreed upon key between the source and the target component.
- OBLIVIOUS TRANSFER [14]. How can a user involve in a zero-knowledge communication?
  Adopt an oblivious transfer protocol, in which a sender transfers one of potentially many pieces of information to a receiver, but remains oblivious as to what piece (if any) has been transferred.
- PACKET FILTER FIREWALL [31]. How can we protect the system from packets sent by malicious hosts?
  Intercept all incoming packets at the single access point and filter them based on the ingress/egress security policy. Reject packets coming from untrusted sources.
- PASSWORD SYNCHRONIZER [33]. How can we use the same password across multiple systems?
  Centralize the management of user credentials across different application systems via a programmatic interface. Issue password service commands to all the connected application systems.
- POLICY [5]. How can policies be enforced effectively?
  Isolate the part of the system that makes policy enforcement decisions in a discrete component. Ensure that policy enforcement functions are performed in proper sequence.
- POLICY DELEGATE [33]. How can security services be loosely coupled with applications to which they are applied?
  Implement a mediator to co-ordinate requests between applications and security services. Perform pertinent message translation to accommodate disparate message formats and protocols.

- POLICY ENFORCEMENT POINT [31]. How can we enforce access control policies at all entry points to the system?
  Channel all outside communication through one point of the system. Apply identification, authorization, and other security mechanisms at that point.
- PROTECTED SYSTEM [5]. How can we prevent unauthorized access to resources?
  Structure a system so that all client access to resources is mediated by a guard which enforces a security policy. The guard can be any type of firewall that acts as the policy enforcement point.
- PROTECTION REVERSE PROXY [31]. How can we protect the server infrastructure from attacks exploiting vulnerabilities in application layer protocols?
  Create a proxy component consisting of two packet filter firewalls with a reverse proxy in between. In the outer firewall, allow only HTTP port to access the reverse proxy. In the reverse proxy, perform application layer checking and forward the valid packets only to the inner firewall.
- PROXY BASED FIREWALL [31]. How do we protect our network from potential attacks that might be embedded within the data segment of the packets?
  Make the client interact only with a proxy of the service requested, which in turn communicates with the protected service. Each application proxy has its own access rules predefined by the administrator that may be used to authenticate, inspect, change, and filter the incoming (or outgoing) messages.
- PSEUDONYMOUS IDENTITY [14]. Anonymity targets can be personal or impersonal, but everything has some form of identity. If identities are left exposed, entities are exposed too. How can entities be saved from exposure?
  Hide identity by adopting a random pseudonym that does not relate to the original.
- RANDOM EXIT [14]. How can an anonymity service prevent exit node abuse?
  Allow traffic to exit an anonymity network not only at the endpoints of a path, but also in the middle of a path.
- RANDOM WAIT [14]. How can the output of a mix node hide timing information?
  Add random delays to the processing of incoming data traffic in an anonymity preserving node to thwart timing attacks.
- REFERENCE MONITOR [31]. How can we enforce authorization policies to prevent users and processes from performing illegal actions?
  Define a process that intercepts all requests for resources and validates access to them.
- REPLICATED SYSTEM [5]. How can we ensure availability of transactional services in the midst of communication and system failures?
  Replicate services and place them at various points in the network. During failure replace the failed service with an available one. Perform load-balancing between available services.
- RISK DETERMINATION [31]. How can we explicitly identify realistic enterprise security needs?
  Make a list of all the business assets. Classify them and identify the types of protection they need.
- ROLE BASED ACCESS CONTROL [31]. How can we reduce the number of individual rights when there are many users and resources involved?
  Group users into roles based on similarities in duties performed. Assign the rights for accessing resources to specific roles.
- SAFE DATA STRUCTURE [16]. How can string routines be made safe from buffer overflow attacks?
  Define a data structure for strings that includes the length information and allocated memory information. In every string-manipulating function, check for length and available memory before proceeding.
- SECURE COMMUNICATION [5]. How can we ensure that the data being passed across public or semi-public networks is secure despite security threats?
  Create a secure channel for sensitive data that obscures the data in transit. Reduce the performance overhead on the system by using ordinary communication channels for non-sensitive data.
- SECURE MESSAGE ROUTER [33]. How can we provide a security intermediary infrastructure that can handle various messages produced by different standards-based frameworks and workflows?
  Define a security intermediary infrastructure that aggregates access to multiple application endpoints in a workflow or among partners participating in a Web service transaction. Use it to route messages between these components.
- SECURE LOGGER [33]. How can we log system events correctly, securely and in a timely manner?
  Use a centrally controlled logging component or service that can be used in various places throughout the application. Protect the logs by encrypting the contents and limiting the access to them.
- SECURE RESOURCE POOLING [16]. How can we minimize the vulnerability associated with daemon processes?
  Limit the lifetime of daemon processes and fork them again after a configurable, short lifetime. Limit the number of requests handled by a daemon process. Run the daemons in a contained environment to minimize the exploits.
- SECURE SERVICE FAÇADE [33]. How can we provide a secure interface for a fine-grained and loosely coupled security service?
  Integrate fine-grained, security-unaware service implementation into a unified, security-enabled interface to clients. Use it as a gateway where client requests are securely validated and routed to the appropriate fine-grained service implementation.
- SECURE SERVICE PROXY [33]. How can we adapt an existing system to use newer security protocols efficiently?
  Provide the new security service as a wrapper. Intercept all the requests from clients, identify the requested service, enforce the security policy as required by the service, and forward the request to the appropriate destination service.
- SECURE SESSION OBJECT [33]. How can the security context associated with a client session be saved and routed securely?
  Design a standardized structure and interface to access the security context. Encapsulate authentication and authorization information like credentials, roles and privileges and use them for secure transport.

- SECURITY ASSOCIATION [5]. How can we eliminate the overhead of passing security-related information in each message?

  Define a data structure for storing security-related information at each end of the channel. Have all participants exchange that information once, when the channel is established, and apply it to all the messages sent and received over the channel.

- SECURITY CONTEXT [5]. How can an execution context, program or process have access to information about a subject whenever it needs to take an action?

  Provide a container for security attributes and data related to a particular user. Make the container accessible to execution context, process, operation or action acting on behalf of the user.

- SECURITY NEEDS IDENTIFICATION FOR ENTERPRISE ASSETS [31]. How can we identify realistic enterprise security needs explicitly?

  Identify all business assets. Classify them and identify the types of protection they need.

- SECURITY SESSION [31]. How can the user data be shared between components?

  Create a session object that holds all the variables that need to be shared by many objects. Associate every action of the user with the session.

- SERVER SANDBOX [25]. How can we make the server applications safe?

  Limit the privileges that Web components possess at run time. Create a user account to be used only by the server and limit its privileges so that it has no administrative rights.

- SINGLE ACCESS POINT [31]. How can we provide security at all entry points to the system?

  Set up only one way to get into the system. Use a login screen as the single access point.

- SINGLE SIGN ON [25]. How can the user be relieved of re-authentication after he successfully authenticates once?

  Create an authenticated session that keeps track of user's authenticated identity for the duration of the user's session. Authenticate the user the first time he requests access. Provide the user with some credentials that he can present with every new request.

- SINGLE SIGN ON DELEGATOR [33]. How can we make the single sign on mechanism interact with diverse entities and yet have loose coupling?

  Encapsulate the access to identity management and single sign on functionality. Decouple the physical security service interfaces and hide the details of service invocation, retrieval of security configuration, and credential token processing from the client.

- SINGLE THREADED FAÇADE [16]. How can the processes at system perimeter be more resilient to attacks?

  Design the processes at the perimeter of the system to perform a single task. Make these processes single-threaded to minimize the risks involved in complex resource management.

- SMALL PROCESSES [16]. How can we protect programs consisting of many processes from resource exhaustion?

  Make the processes small. Design each process to perform only one task. This means that processes allocate fewer resources and are less likely to exhaust system memory.

- STANDBY [5]. How can we design a system that is tolerant of component failure?

  Structure a system with backup components so that the service provided by one component can be resumed by the backup component in the case of system failure.

- STATEFUL FIREWALL [31]. How can we correlate incoming packets (e.g. to identify a potential attack)?

  At the firewall, keep a list or table of the connections that have been opened and correlate the types of messages received and sent to the connections. To improve performance, do not check the packets from well-established connections.

- SUBJECT DESCRIPTOR [5]. How can we associate attributes with a subject?

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

- TANDEM SYSTEM [5]. How can we make a system tolerant to domino-effect of failures, where a single failure is propagated and brings about failure of the entire system?

  Structure a system so that an independent failure of one component is detected quickly. Perform each task on multiple components and use the result only if they all produce similar results.

- TRUSTED PROXY [25]. How can we hide the shortcomings of security mechanisms of components from malicious users?

  Use a trusted proxy as a buffer between inadequately protected components and untrusted users. In the proxy, intercept and filter all communication between the users and components to enforce appropriate security mechanisms.

- TRUST PARTITIONING [16]. How can we ensure system security even after some part of the program is compromised?

  Assign minimum privilege level to components. Classify the owners of processes into different trusted and untrusted groups. Design the components to distrust inputs from other groups and to validate inputs.

- UNIQUE LOCATION FOR EACH WRITE REQUEST [16]. How can we mitigate the effects of failure when multiple operations update the same resource?

  Create a unique file for each write request. If the write fails, only one operation needs to be re-executed.

- VULNERABILITY ASSESSMENT [31]. How can an enterprise identify vulnerabilities to its assets and determine the severity of those vulnerabilities?

  Systematically identify and rate probable vulnerabilities of the enterprise assets. Create a threat model and identify vulnerabilities. Rate the severity of vulnerabilities.

- WHITE HATS HACK THYSELVES [28]. How can we approximate real-world security attacks before the system is deployed?

  Apply grey hat hacking techniques against your own system before it is deployed. Plan and execute attacks under controlled but non-trivial circumstances.