Final Technical Report: Security Patterns for Web Application Development

DARRELL M. KIENZLE, PH.D.
PRINCIPAL INVESTIGATOR
(703) 885-4816

MATTHEW C. ELDER, PH.D.
RESEARCH SCIENTIST
(703) 885-4814

DARPA Contract # F30602-01-C-0164
Effective Date of Contract: June 7, 2001
Contract Expiration Date: May 30, 2002

Security Patterns for Web Application Development

DARRELL M. KIENZLE, PH.D.
MATTHEW C. ELDER, PH.D.

Our final technical report documenting patterns of security in web applications.

EXECUTIVE SUMMARY

A *security pattern* is a well-understood solution to a recurring information security problem. They are patterns in the sense originally defined by Christopher Alexander (the basis for much of the later work in design patterns and pattern languages of programs), applied to the domain of information security. A security pattern encapsulates security expertise in the form of worked solutions to these recurring problems, presenting issues and trade-offs in the usage of the pattern. This document presents our research into security patterns for Web application development.

We have produced Version 1.0 of our Security Patterns Repository, consisting of 26 patterns and 3 mini-patterns. (A mini-pattern is a shorter, less formal discussion of security expertise in terms of just a problem and its solution.) We focused on the domain of Web application security to focus the scope of the problems that our patterns address. We also constructed a worked example system using some of our security patterns to help validate the approach; this example system was a patterns repository to present our security patterns, spur discussion, and collect feedback. The Security Patterns Repository, both patterns and application, can be found in its entirety at **http://www.securitypatterns.com**.

In this paper, we present the context for security patterns, including discussions of design patterns, patterns templates, and our security patterns. We outline our approach to the security patterns project, then introduce the security patterns in our repository, along with their abstracts. Next, we examine some of the issues that we uncovered during our documenting of and experimenting with security patterns. Finally, we discuss some other related efforts in the security patterns space and present our tentative conclusions.

Introduction

There is a huge disconnect between security professionals and systems developers. Security professionals are primarily concerned with the security of a system, while developers are primarily concerned with building a system that works. While security is one of the non-functional goals with which developers must be concerned, it is only one of many. And while security professionals complain that developers don't take security seriously, developers are just as frustrated that security professionals don't understand that security is not their only, or even primary, concern.

OBJECTIVES

Security patterns are proposed as a means of bridging the gap between developers and security experts. Security patterns are intended to capture security expertise in the form of worked solutions to recurring problems. Security patterns are intended to be used and understood by developers who are not security professionals. While the emphasis is on security, these patterns capture the strengths and weaknesses of different approaches in order to allow developers to make informed trade-off decisions between security and other goals.

Above all, security patterns are meant to be constructive. Far too much of the available security expertise is presented in the form of laundry lists of what not to do. The poor developer who attempts to understand security is likely to be overwhelmed by these lists. Furthermore, security patterns are meant to be educational, and research in pedagogical techniques has indicated that simply telling people what not to do will not change their behavior. Instead, successful teaching methodologies focus on modeling and then rewarding good behavior. Security patterns try to provide constructive assistance in the form of worked solutions and the guidance to apply them properly.

REPORT OVERVIEW

In this paper, we present our work on security patterns, including an overview of Version 1.0 of our Security Patterns Repository. We discuss the origin of security patterns and our approach to this project, as well as our security patterns template and repository organization. We outline both our structural and procedural security patterns in the repository, providing abstracts for each pattern. We then present some of our experiences and observations developing and using security patterns. Finally, we include some related work and conclusions.

An Overview of Security Patterns

The idea of security patterns evolved from design patterns. This section provides a high-level overview of design patterns and presents our concept of a security pattern.

DESIGN PATTERNS

One of the most exciting developments in software engineering is the emergence of design patterns as an approach to capturing, reusing, and teaching software design expertise. This movement first gained widespread visibility with the publication of *Design Patterns* [9] in 1994. The "Gang of Four" book, as it is commonly known, defined design patterns as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" [9].

The patterns approach has infiltrated all areas of software engineering. For example, the Java APIs, the OpenStep libraries, and the Microsoft Foundation Classes all use the patterns catalogued in *Design Patterns*. There are numerous workshops, books, and Web sites devoted to the further study and development of design patterns.

The design patterns movement actually has its roots outside of software engineering entirely. Alexander invented the concept in his writings on architecture and urban planning [1]. He developed the approach in order to capture the essential knowledge of his field and provide a methodology in place of ad hoc craftsmanship. Alexander's patterns ranged from high level (placement of buildings within a community) to low level (the layout of an individual room). An excellent discussion of Alexander's work can be found in Gabriel's *Patterns of Software* [8]. Gabriel's book should be required reading for the design patterns community in that it presents a balanced view of both the successes and the failures of Alexander's approach.

Alexander defined a pattern as describing "a problem which occurs over and over again in our environment, and that describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice". Vlissides paraphrased Alexander's definition of a pattern as "a solution to a problem in a context". Vlissides posited that in addition to the problem, context, and solution, a pattern must include recurrence, a teaching component, and a name [18].

Alexandrian patterns generally include the following major elements:

- A standard name by which the pattern can be discussed
- A concise summary of the problem addressed by the pattern
- A description of the solution
- The impact that the pattern has on the "essential forces" at play

Design patterns generally adhere to a much more restrictive format. They include UML diagrams of the structure and the dynamic interactions of the objects that constitute the pattern. They require examples of the patterns in use. They include pointers to related patterns. And they generally include sample code implementing the pattern.

The design patterns approach has been very successful at capturing the recurring code-level patterns that occur in object-oriented software. There is a large community developing exactly this type of pattern. However there are at least three significant variations worth mentioning:

- *Architectural patterns*. Coad recognized that many software design patterns could actually be applied at higher levels of abstraction. For example, a publish-subscribe model described in a design pattern could actually be used at the enterprise level between entire systems. Coad developed a repository of "architectural patterns" to capture this. While similar in nature to design patterns, this work demonstrated the general utility of Alexander's patterns approach applied to software.
- *AntiPatterns*. Another group of four authors published "AntiPatterns" as an approach to documenting common mistakes in software development [5]. These antipatterns document common implementation, architecture, and even software engineering process mistakes. The *AntiPatterns* books attempt to provide guidance for "refactoring" inappropriate approaches into more acceptable alternatives. However, the major focus of the AntiPatterns approach is in enumerating missteps to avoid.
- *Pattern Languages*. Alexander actually developed the notion of a pattern language as a mechanism for expressing collections of interrelated patterns. Using a pattern language, it is possible to define families of solutions that allow plug-replacement of specialized components. This approach is quite popular, as it lends itself well to the development of object-oriented frameworks of interrelated components.

SECURITY PATTERNS

A security pattern is a well-understood solution to a recurring information security problem. They are patterns in the sense originally defined by Christopher Alexander, applied to the domain of information security. While some of these patterns will take the form of design patterns, *not all security patterns are design patterns*.

Because of the popularity of design patterns in the software engineering community, the natural inclination is to assume that anything going by the name "security patterns" should be described using a UML diagram and include sample source code. While it is true that many interesting security patterns can be presented this way, there are many other important patterns that do not fit within these constraints.

For informational purposes, we identify two broad categories of security patterns:

- *Structural patterns*. These are patterns that can be implemented in the final product. They encompass design patterns, such as those used by the Gang of Four. They often include diagrams of structure and descriptions of interaction.
- *Procedural patterns*. These are patterns that can be used to improve the process for development of security-critical software. They often impact the organization or management of a development project.

It is important to note that there are a number of different efforts bearing the name "security patterns". There is no single, agreed-upon definition of a security pattern, though general consensus is emerging. Please see the section on *Other Security Patterns Efforts* for a discussion of other approaches to (and other definitions of) security patterns.

Our Approach

Our security patterns effort focused on three components: a template and tutorial, the patterns repository, and a worked example system. In this section we outline our approach in each of these areas.

TEMPLATE AND TUTORIAL

Our approach on the Template and Tutorial consisted of surveying related work in design patterns and pattern languages, then constructing a template specifically for use with security patterns.

The Hillside Group maintains a resource for writing general patterns and pattern languages [11]. The security patterns template that we developed for the Template and Tutorial was derived in part from research into existing patterns templates, including ones used by AG Communication Systems [1] and the Gang of Four in *Design Patterns* [9]. We identified common fields and overlapping categories in existing patterns templates as the basis for our generic security patterns template. We then augmented those categories with additional security-specific elements. This security patterns template was the most general case. Wherever fields were not applicable in particular patterns they were labeled as such.

During our work soliciting feedback and revising patterns in the months following the Template and Tutorial, we identified a number of issues with the existing template. As a result, we produced a second version of the template that we used in Version 1.0 of our Security Patterns Repository. We present and explain the new template in the next full section, *Security Patterns Template*.

PATTERNS REPOSITORY

Our approach to developing a patterns repository was to collect security expertise from a variety of sources, including relevant security literature and personal experience, then package this information in the template from

above. Throughout the course of the project, we did extensive work documenting security patterns and revising those patterns based on review and feedback.

Our approach also included significant investigation into categorizations of the security patterns in our repository. We determined a number of alternative classification schemes for our patterns repository over the course of the project; the Web site presents some of those classification schemes.

Version 1.0 of the Security Patterns Repository, discussed in this document, consists of 26 patterns and 3 mini-patterns. A mini-pattern is a shorter, less formal discussion of security expertise in terms of just a problem and its solution. There are 13 structural patterns and 3 structural mini-patterns. There are 13 procedural patterns. The next section presents the structural patterns with their abstracts, and the section following that presents the procedural patterns with their abstracts.

Abstracts from the security patterns in Version 1.0 of the repository are presented in two subsequent sections of this paper, *Structural Patterns* and *Procedural Patterns*.

WORKED EXAMPLE SYSTEM

Our approach towards experimental assessment of our security patterns was to construct a worked example system: a public Web repository for our security patterns. This application utilized a number of the security patterns in our repository. This sample system provided us with concrete code examples that could be folded back into the repository to better illustrate the patterns used. In addition, it provided developers with a worked example of building security into a Web application using the security patterns repository.

Most importantly, the development of this system was intended to force us to "eat our own dog food." By developing a Web system with an eye towards a public deployment, we were able to better understand the needs of our intended audience. Ultimately, we hope it will give us greater credibility if our patterns repository actually implements the patterns that we suggest others be using.

The functionality of the example system includes management of the patterns in the repository, enrollment, access control, collection of feedback from users, and on-line editing of patterns.

Our worked example system is an Apache Tomcat application serving Java servlets and Java Server Pages, which connects to a MySQL database containing the patterns on the back end. The Web application works on both Windows and Linux platforms.

Security Patterns Template

Many patterns researchers have found that describing patterns in a consistent format aids both patterns readers and writers. There are numerous existing templates for design patterns, security patterns, and other patterns efforts. We have examined previous patterns templates and settled on the structure described below for our security patterns.

We do not claim that our template is perfect or complete. We started the project with a general idea of what should be in a security patterns template. Over time, our template has evolved as fields were added, eliminated, or combined. Nevertheless, we feel that it is important that our patterns follow some basic structure, conveying the same basic information and answering the same basic questions.

Our patterns adhere to the following security patterns template:

- *Pattern Name*

  The *Pattern Name* captures the essence of the pattern in a concise and, if possible, catchy manner. Ideally, pattern names avoid terms that already have strong connotations.

  If possible, structural patterns are nouns that describe the thing that can be built, e.g., *Partitioned Application* or *Trusted Proxy*. More abstract nouns are appropriate when the pattern is itself more abstract. For example, the *Password Authentication* pattern is a general pattern that describes many facets of password authentication mechanisms.

  Procedural patterns are names using active verb phrases that describe the action the pattern recommends. For example, *Patch Proactively* is preferable to "Proactive Patching".

  Parenthetically, the *Pattern Name* field includes aliases or other appropriate names for the pattern ("a.k.a."). These alternate names can include names by which others might have referenced the pattern in the literature.

- *Abstract*

  The *Abstract* summarizes the pattern briefly (i.e., in a few sentences). The summary includes what the purpose or intent of the pattern is but does not go into implementation details. Because readers will often skim lists of abstracts, they must (a) stand completely independent of any context and (b) be concise.

  The abstract allows users to decide which patterns to explore in greater detail. Thus, they give some indication of any limits on pattern applicability. For example, if a pattern should not be used to protect high-value data, then that is something noted in the abstract.

- *Problem*

  The *Problem* describes the conditions that motivate the usage of the pattern. This section outlines the context in which the pattern is applicable. When multiple patterns address the same basic problem, the *Problem* section for each pattern provides the more detailed context that would make that pattern specifically appropriate.

  The problem statement does not contain a lengthy discussion of secondary effects. For example, the *Problem* section for the *Password Authentication* pattern does not include the need to protect against password-guessing attacks. The *Password Authentication* pattern addresses the problem of authenticating users. Susceptibility of this approach to password-guessing attacks is a secondary effect of using passwords.

- *Solution*

  The *Solution* describes at a high level how the pattern solves the problem described in the problem statement. This section explains how the pattern is applicable to the problem and the rationale for applying the solution.

  Optionally, the *Solution* section will include a diagram to describe the solution structure visually. A solution will also be explained in terms of particular components and their interactions, if appropriate. Significant scenarios comprising the solution are presented in detail in this section.

- *Issues*

  The *Issues* provide some wisdom regarding the pattern in the form of detailed explanation and hints. It is important to not overwhelm the reader though. The most significant issues are addressed first and in the most detail. Minutiae are given relatively little attention.

  This section identifies the most common mistakes in the usage of this pattern and provides the reader with guidance for avoiding them. This section also enumerates residual vulnerabilities by explaining circumstances wherein a properly implemented pattern might still be attacked. Finally, possible attacks against the pattern are identified and defenses against those attacks addressed.

- *Examples*

  The *Examples* cite known uses of this pattern. Explicit references to products or systems can be used. However, in recognition that many systems do not want their security internals discussed, it is appropriate to refer to “a known system” without identifying that system. Likewise, it might be necessary to provide only general details of a specific system.

  Examples can be identified as occurring at different implementation levels, such as the code level, system level, or network level. Although

many patterns target a specific level, examples from all three levels are included wherever possible in order to communicate the generality of a pattern.

Wherever available, sample code from our repository application will be referenced. Developers appreciate sample code that they can link directly into their application and begin using immediately. These patterns are not a sourcebook, but the inclusion of sample code makes the material more tangible and gains credibility with the intended readership.

- *Trade-Offs*

  The *Trade-Offs* describe the possible impact of using the pattern with respect to various functional and non-functional requirements. This section hypothesizes about the competing forces that come into play in the usage of the pattern.

  Each of the following areas of potential consequence is discussed: Accountability, Availability, Confidentiality, Integrity, Maintainability, Usability, Performance, and Cost. If the pattern appears to have no impact on the consequence, the field indicates "No effect" or "No direct effect".

  The trade-offs discussed in this section focus on primary effects. Clearly, system compromises could result in many consequences, but a pattern that targets integrity (for example) will not include speculation on all possible secondary effects.

- *Related Patterns*

  The *Related Patterns* list and include links to any other related patterns in the repository. The listing of related patterns includes a brief description as to the nature of the relationship. Two patterns can be related in any number of ways: one pattern can use another, one pattern can be dependent upon another, one pattern can replace another, etc. Also, if related security patterns from other sources are known, they are indicated here.

- *References*

  The *References* enumerate citations related to the pattern in the literature and on the Web. These references are cited throughout the pattern descriptions.

Structural Patterns

The following are the structural patterns and mini-patterns in Version 1.0 of the Security Patterns Repository. Please visit **www.securitypatterns.com** for complete pattern descriptions and the opportunity to submit feedback.

ACCOUNT LOCKOUT

Passwords are the only approach to remote user authentication that has gained widespread user acceptance. However, password-guessing attacks have proven to be very successful at discovering poorly chosen, weak passwords. Worse, the Web environment lends itself to high-speed, anonymous guessing attacks. Account lockout protects customer accounts from automated password-guessing attacks, by implementing a limit on incorrect password attempts before further attempts are disallowed.

AUTHENTICATED SESSION

An authenticated session allows a Web user to access multiple access-restricted pages on a Web site without having to re-authenticate on every page request. Most Web application development environments provide basic session mechanisms. This pattern incorporates user authentication into the basic session model.

CLIENT DATA STORAGE

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

CLIENT INPUT FILTERS

Client input filters protect the application from data tampering performed on untrusted clients. Developers tend to assume that the components executing on the client system will behave as they were originally programmed. This pattern protects against subverted clients that might cause the application to behave in an unexpected and insecure fashion.

DIRECTED SESSION

(Mini-Pattern) The *Directed Session* pattern ensures that users will not be able to skip around within a series of Web pages. The system will not expose multiple URLs but instead will maintain the current page on the server. By guaranteeing the order in which pages are visited, the developer can have confidence that users will not undermine or circumvent security checkpoints.

ENCRYPTED STORAGE

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

HIDDEN IMPLEMENTATION

(Mini-Pattern) The *Hidden Implementation* pattern limits an attacker's ability to discern the internal workings of an application—information that might later be used to compromise the application. It does not replace other defenses, but it supplements them by making an attacker's job more difficult.

MINEFIELD

The *Minefield* pattern will trick, detect, and block attackers during a break-in attempt. Attackers often know more than the developers about the security

aspects of standard components. This pattern aggressively introduces variations that will counter this advantage and aid in detection of an attacker.

NETWORK ADDRESS BLACKLIST

A network address blacklist is used to keep track of network addresses (IP addresses) that are the sources of hacking attempts and other mischief. Any requests originating from an address on the blacklist are simply ignored. Ideally, breaking attempts should be investigated and prosecuted, but there are simply too many such events to address them all. The *Network Address Blacklist* pattern represents a pragmatic alternative.

PARTITIONED APPLICATION

The *Partitioned Application* pattern splits a large, complex application into two or more simpler components. Any dangerous privilege is restricted to a single, small component. Each component has tractable security concerns that are more easily verified than in a monolithic application.

PASSWORD AUTHENTICATION

Passwords are the only approach to remote user authentication that has gained widespread user acceptance. Any site that needs to reliably identify its users will almost certainly use passwords. The *Password Authentication* pattern protects against weak passwords, automated password-guessing attacks, and mishandling of passwords.

PASSWORD PROPAGATION

Many Web applications rely on a single database account to store and manage all user data. If such an application is compromised, the attacker might have complete access to every user's data. The *Password Propagation* pattern provides an alternative by requiring that an individual user's authentication credentials be verified by the database before access is provided to that user's data.

SECURE ASSERTION

The *Secure Assertion* pattern sprinkles application-specific sanity checks throughout the system. These take the form of assertions – a popular technique for checking programmer assumptions about the environment and proper program behavior. A secure assert maps conventional assertions to a system-wide intrusion detection system (IDS). This allows the IDS to detect and correlate application-level problems that often reveal attempts to misuse the system.

SERVER SANDBOX

Many site defacements and major security breaches occur when a new vulnerability is discovered in the Web server software. Yet most Web servers run with far greater privileges than are necessary. The *Server Sandbox* pattern builds a wall around the Web server in order to contain the damage that could result from an undiscovered bug in the server software.

TRUSTED PROXY

A trusted proxy acts on behalf of the user to perform specific actions requiring more privileges than the user possesses. It provides a safe interface by constraining access to the protected resources, limiting the operations that can be performed, or limiting the user's view to a subset of the data.

VALIDATED TRANSACTION

(Mini-Pattern) The *Validated Transaction* pattern puts all of the security-relevant validation for a specific transaction into one page request. A developer can create any number of supporting pages without having to worry about attackers using them to circumvent security. And users can navigate freely among the pages, filling in different sections in whatever order they choose. The transaction itself will ensure the integrity of all information submitted.

Procedural Patterns

The following are the procedural patterns in Version 1.0 of the Security Patterns Repository. Please visit **www.securitypatterns.com** for complete pattern descriptions and the opportunity to submit feedback.

BUILD THE SERVER FROM THE GROUND UP

Many Web compromises and defacements occur because of unnecessary and potentially vulnerable services present on the Web server. Default installations of many operating systems and applications are the source of many of these services. This pattern advocates building the server from the ground up: understanding the default installation of the operating system and applications, simplifying the configuration as much as possible, removing any unnecessary services, and investigating the vulnerable services that are a part of the Web server configuration.

CHOOSE THE RIGHT STUFF

Many security problems can be avoided during system design if components, languages, and tools are selected with security in mind. This is not to say that security is the only criterion of concern – merely that it should not be ignored while making these decisions. This pattern provides guidance in selecting appropriate Commercial-Off-the-Shelf components and in deciding whether to use build custom components.

DOCUMENT THE SECURITY GOALS

In order for developers to make consistent, intelligent development choices regarding security, they have to understand the overall system goals and the business case behind them. If the security goals are not documented and disseminated, individual interpretation could lead to inconsistent policies and inappropriate mechanisms.

DOCUMENT THE SERVER CONFIGURATION

Web servers and application servers are extremely complex, and complexity is a major impediment to security. In order to help manage the complexity of Web server and application configurations, developers and administrators must document the initial configuration and all modifications to Web servers and applications.

ENROLL BY VALIDATING OUT OF BAND

When enrolling users for a Web site or service, sometimes it is necessary to validate identity using an out-of-band channel, such as postal mail, telephone, or even face-to-face authentication. The out-of-band channel can

be used to establish a shared secret, which can then be used to establish identity during enrollment.

ENROLL USING THIRD-PARTY VALIDATION

When enrolling users for a Web site or service, it is always easier to allow some other party to take on the difficult task of authenticating user identity. When a third-party service is available and sufficiently reliable, the Web application can offload this task on the third party. This approach is becoming more common as third-party services become available. The most common form of transaction authentication—credit card authentication—is a form of third-party validation.

ENROLL WITH A PRE-EXISTING SHARED SECRET

When enrolling users for a Web site or service, sometimes it is sufficient to validate identity using a pre-existing shared secret, such as a social security number or birthday. The use of a pre-existing shared secret enables enrollment without prior communication specific to setting up an account.

ENROLL WITHOUT VALIDATING

When enrolling users for a Web site or service, sometimes it is not necessary to validate the identity of the enrolling user. When there is no initial value involved in the Web site or service for which enrollment is occurring, validation is an unnecessary procedure and can be eliminated.

LOG FOR AUDIT

Applications and components offer a variety of capabilities to log events that are of interest to administrators and other users. If used properly, these logs can help ensure user accountability and provide warning of possible security violations. The *Log for Audit* pattern ties logging to auditing, to ensure that logging is configured with audit in mind and that auditing is understood to be integral to effective logging.

PATCH PROACTIVELY

During the lifetime of a software system, bugs and vulnerabilities are discovered in third-party software, and patches are provided to address those issues. Rather than waiting for the system to be compromised before applying patches ("patching reactively"), administrators of software systems should monitor for patches often and apply them proactively.

RED TEAM THE DESIGN

Red teams, which examine a system from the perspective of an attacker, are commonly used to assess the security of a finished system. However, the earlier in development that a problem is found, the easier it is to fix. The *Red Team the Design* pattern effects a security evaluation of the application at the stage when it is most possible to fix any problems identified.

SHARE RESPONSIBILITY FOR SECURITY

The *Share Responsibility for Security* pattern makes all developers building an application responsible for the security of the system. Security consists of more than just encryption, anti-virus software, and firewalls. Any element of a system can have security concerns, and system developers have to understand and address those concerns. Use of this pattern avoids the common problem of "the security guy" or security team being pitted against the rest of the development team.

TEST ON A STAGING SERVER

Web site development requires extensive testing to enable availability, protect confidentiality, and ensure integrity. While unit testing can be done on development machines, system and integration testing should take place on machines as similar to the production servers as possible. The use of a staging server enables necessary testing while preventing the outages that often occur when developers and administrators experiment with the live production system on the fly.

Observations from Documenting Security Patterns

In our time documenting and exploring security patterns, we uncovered a number of interesting research issues. We present some of our problems, lessons learned, and significant observations experiences in this section.

ABSTRACTION LEVEL

There is no single correct level of detail for security patterns. Different potential consumers of security patterns work at different levels. A developer may be primarily concerned with patterns of code-level objects, an architect may build network models, and a CIO may be primarily interested in trust relationships between organizations. All are valid uses of the security patterns approach, though each target audience might find little of value in patterns at a much different level of detail.

We have found it useful to distinguish between four different categories of pattern detail:

1. *Concepts*. This is the highest level of abstraction, and encompasses general strategies such as diversity, obscurity, and least privilege. These are abstract nouns and cannot be implemented directly by developers. For example, one cannot build a “Least Privilege”.
2. *Classes of patterns* (also called pattern families). A class of pattern represents a general problem area for which multiple solutions are possible. While a class of pattern cannot be directly implemented, it can be used as a placeholder for tradeoff analysis between alternative approaches.
3. *Patterns*. A pattern is specific enough to allow basic properties to be specified and trade-off analysis to be conducted against other patterns. It is general enough to allow it to be used in multiple circumstances with multiple targets.
4. *Examples*. An example is a worked solution to a very specific problem instance. An example is typified by sample code. It is the most immediately useful, but in a very narrow context.

As an example, consider the desire for secret communication between two parties. An executive may identify the need for confidentiality. An architect

could suggest an encrypted communication as a class of pattern. The software designer could select a session encryption pattern supported by an out-of-band authentication pattern. Finally, a developer would instantiate the pattern using specific algorithms, libraries, source code, and key strengths.

We must reiterate that this is an arbitrary selection of levels of abstraction. An architect might refer to level 2 as "patterns" and believe that level 3 and below represent nonessential details. Alternatively, the businessman might see levels 2, 3, and 4 as all nonessential details that the technical types can deal with. The implementor might see level 3 and above as hopelessly abstract. In all cases, people are most comfortable working at a certain level of abstraction.

We have selected level 3 for our patterns (with examples from level 4) because we believe it represents the point where we can get the most benefit in terms of transfer of security knowledge. Any more concrete and we run the risk of losing the message amidst the details. Any more abstract, and we lose interest of the developers who, ultimately, build the software upon which we depend.

REPOSITORY ORGANIZATION

An area of investigation that generated considerable but inconclusive discussion concerned the possible organizations of the Security Patterns Repository. As we were documenting security patterns and as the repository achieved a certain critical mass, we began to note various relationships between particular patterns. Certain patterns were related in a general sense topically; other patterns were obvious alternatives for achieving a particular goal; still other patterns seemed to follow in a sequential manner. Noticing the myriad relationships between patterns, we naturally investigated categorizations and classifications within our patterns repository.

We investigated the following organizing strategies for the patterns repository:

- Subject Matter – the problem area the pattern addresses.
- Life Cycle Stage – the development stage when the pattern would be used.
- Pattern Type – the class of pattern, such as structural or procedural.
- Audience – the intended reader/user of the pattern.

Agreeing upon a definitive organization for the repository proved to be an elusive goal. Even coming to agreement on the relative superiority of the various possible organizations was difficult. There are many valid organizations for the collection of patterns that we documented, and the utility of a particular classification scheme is dependent on the goals of the repository user.

To support the multiple possible organizations of the patterns repository, we presented the user with different repository classification schemes in a previous version of the repository application. In the current version of the repository application, we present only broad pattern type categories (structural, procedural, and enrollment). We support keyword search to enable organization by subject matter, and other pattern relationships are described in the *Related Patterns* section of each pattern.

SECURITY PATTERNS TEMPLATE

Another area of investigation that generated significant discussion involved a template specifically for security patterns. As mentioned previously, there are numerous existing templates for design patterns. The topic of a definitive template for design patterns is a hot button issue in that community; agreeing on a template for security patterns throughout the course of this work was non-trivial for us as well.

Our initial version of a security patterns template, presented in our "Security Patterns Template and Tutorial" document [13], was an amalgam of various existing design patterns templates, including those of the Gang of Four [9] and AG Communication Systems [1]. We incorporated security-specific concerns in fields for common attacks and consequences (such as the impact on availability, confidentiality, and integrity). This version of the template consisted of 14 fields and another 13 sub-fields.

As the patterns repository evolved and we utilized our initial version of the template, we recognized the shortcomings of our all-encompassing, detailed template. In particular, we noted that many of the particular fields and sub-fields did not apply to both structural and procedural patterns. The repeated "N/A" entries in certain patterns made those patterns seem empty or less complete.

As a result of this informal experimentation, we simplified the security patterns template to the one presented previously. By generalizing some of the pattern fields and dropping other detailed sub-fields, the template is applicable to both structural and procedural patterns. As we used the new template on the current version of the repository, we found that the same information could be conveyed as before, but the structure was more flexible to enable easier use with certain patterns.

EXAMPLES OF/FOR SECURITY PATTERNS

A significant issue that we encountered when documenting our security patterns was the difficulty in finding concrete examples of pattern usage. In some circumstances, a pattern's usage is hidden behind an application's implementation, so we are not in a position to identify and present the example. In other circumstances, we were familiar with specific examples of a pattern's usage but confidentiality concerns prevented us from citing the example explicitly. Finally, while we can draw upon our own experience in Web development, we are not in a position that enables us to cite examples from a sufficiently wide range of Web development security projects.

To help validate our security patterns, we utilized various patterns in the development of our repository application whenever possible. This also enables us to cite one more example for those patterns' usage.

In other cases where we could not cite many specific examples of a pattern's usage, we have cited literature that helps validate the concept.

One of the benefits of presenting our security patterns in a public forum for feedback is the possibility that others can provide us with examples of usage for various patterns. The Security Patterns Repository will be kept up to date with such information as feedback is provided to us.

Observations from Using Security Patterns

Throughout the course of building our example application—a repository for displaying patterns, editing patterns, and soliciting feedback—we identified a number of issues with both procedural and structural patterns. This section continues to explore our problems, lessons learned, and significant observations.

BUILD THE SERVER FROM THE GROUND UP

One of the first procedural patterns that we utilized was *Build the Server from the Ground Up*. We chose Red Hat Linux 7.3 as the operating system for our Web server and Apache Tomcat to serve pages. We experimented extensively with configuration options.

We originally thought that we would simply use the Red Hat installer's default "Server" configuration. However, we quickly learned that the default installation includes literally thousands of unnecessary files. By judiciously using the installer's "Select Individual Packages" option, we were able to approximate the *Build the Server from the Ground Up* pattern. We repeatedly installed the system using fewer packages each time, until we reached the minimum that still allowed the system to function. And the process was actually quite fast because of the minimal nature of the installation.

When building the server, we found it very useful to keep copies of all of the files that were needed. There were a number of files (such as the Java Development Kit, the Tomcat application server, and the MySQL JDBC driver) that we had to download from the Web. In order to ensure that the server could be rebuilt identically, we maintained local copies of all of the necessary files.

DOCUMENT THE SERVER CONFIGURATION

Another procedural pattern that we utilized in concert with the previous one was *Document the Server Configuration*. As we experimented with the myriad configuration options on the operating system and Web server software, we documented the entire process.

We discovered that the goals of experimentation and documentation are often contradictory. Getting the various components to work together was often a difficult process – attempting to document every step made the process considerably more painful.

However, when one member of the team performed some configuration experiments without documenting them, the result was a working system that we could not recreate. We were in the awkward position of having a system that finally worked properly, and being afraid of touching the system for fear of causing it to stop working. After several weeks of further experimentation, we finally rooted out the problem and our solution.

As a result of this experience, we can conclusively say that while documenting the server configuration was painful, not documenting it resulted in considerably more pain.

HIDDEN IMPLEMENTATION

One of the structural mini-patterns that we utilized throughout the development process was *Hidden Implementation*. This mini-pattern is actually a collection of conceptual techniques that advocate obscuring specific implementation details from users and potential attackers.

This pattern proved much harder than we had anticipated. We had intended to make changes to our configuration so that end users would not be able to tell that we were using Apache Tomcat. We began by simply changing all of the Java Server Pages to use .html extensions instead of .jsp. That simple change was actually quite problematic. The program makes a number of assumptions about extensions, treating .html differently from .jsp. For example, at one point pages were being delivered to the browser with sensitive scripting tags intact, because the application server did not automatically execute tags in files ending with .html.

Furthermore, there are a number of places in the HTTP protocol where products are expected to identify themselves in order to facilitate compatibility. We could not eliminate these references without risking incompatibilities.

Finally, there are any number of different error screens buried within the Tomcat application that identify the program and version number. Changing these would have required making changes directly to the application source code. This was something we wished to avoid for maintenance reasons.

PASSWORD AUTHENTICATION

We implemented *Password Authentication* in our web application in order to meet DARPA's restrictions on public release of work products. In addition, we felt strongly that this pattern should be fully tested and sample source code developed. As a result of this process, we were forced to revise our pattern considerably.

The earliest drafts of the pattern repository stated unequivocally that passwords should always be communicated using encrypted sessions. This is standard security advice and we had no intention from deviating from the standard.

However, in developing the site, we intuitively understood that the value of data we were trying to protect had no need for encrypted passwords. We were not protecting bank transactions or credit card numbers. Like many other sites that wished to associate users with remarks, but recognized that there would be little harm if this mechanism were breached.

As a result of this experience, we revised the *Password Authentication* pattern to recognize that encryption of password communication is not always necessary.

ACCOUNT LOCKOUT

We implemented an *Account Lockout* mechanism that blocks further access after a certain number of failed login attempts. The mechanism itself proved to be quite simple to implement. However, the additional functions needed to administer the lockout mechanism were quite complex. We wound up compromising and implementing a mechanism that automatically resets locked out accounts after some period of inactivity. We believe that this approach strikes a good balance between usability and account security, but we are interested in getting feedback from other developers.

AUTHENTICATION SESSION

We utilized the *Authenticated Session* pattern to support the editing capabilities in our repository application. This pattern worked very well, maintaining sensitive information in server-resident session data, where it cannot be tampered with.

We added the concept of "roles" to this pattern in order to simplify the process of rendering role-specific pages. For example, when a user who is identified as an "editor" logs in the session variable "editor" is created. Whenever a pattern page is assembled on the server, if this variable is present, the user is presented with the option of editing the pattern. When roles are used, it is important that a single logout function be created in order to ensure that all roles are cleared in a single place.

We also learned that it is very useful to add a "logout" option early in the development process. Otherwise, it is very tedious to repeatedly quit the web browser application in order to clear the session data.

MINEFIELD

We implemented a number of customizations in order to make a potential attacker less comfortable on the server. We will not give any particular details here, but we can say that developers thoroughly enjoy this work and can be very creative and devious.

SERVER SANDBOX

The server sandbox pattern was originally written using the tried-and-true

technique of dropping root privileges once a service has bound to the necessary privileged ports. While preparing to deploy the production server, we learned of a superior approach that is now available in Red Hat Linux.

This approach involves using *capabilities*. The application server can be given the privilege to bind to a privileged port, but none of the other administrative privileges associated with the *root* account. Without using capabilities, we were forced to install both the Apache web server (which can run as nobody) and the Tomcat application server (which must be able to bind directly to a low port). Once we learned of this capability, we were able to do away with Apache and implement a server sandbox around Tomcat.

This experience reinforces the importance of using these patterns in production, and improving them based on real-world experience. We will update the *Server Sandbox* pattern to include this technique, as well as others that readers discover and report.

Other Security Patterns Efforts

There are other groups working in the same general space, attempting to apply the concepts of design patterns to security. There is no "one-size-fits-all" solution. This section presents a very brief overview of other work, presented in rough chronological order, so that the reader can determine which approach will best address his/her specific needs.

SECURITY PROPERTIES OF DESIGN PATTERNS

The very first reference to design patterns in a security context was a presentation by Deborah Frincke at NISSC 1996. The presentation discussed the security ramifications of some of the original 23 patterns in *Design Patterns* [9]. Her basic idea was that one could start with a security requirement on a pattern and decompose it into constituent object requirements. In this way, if a proof were developed at the pattern level, it could be reused in tandem with the pattern. This was a powerful idea, but limited to patterns that were not originally intended to address security requirements. Unfortunately, this work was not continued and was never published.

NAVAL RESEARCH LAB'S PATTERNS

The Naval Research Lab (NRL) is currently involved in applying patterns to the problem of formal verification of security-critical software. The earliest work using logic trees to organize security arguments [12] indicated that claim trees could be made more cost-effective through reuse in tandem with common patterns of software architecture. An assurance professional could create a generic argument to accompany a software pattern. Software developers could then instantiate that pattern in circumstances that require the functional and assurance properties that it provides.

The NRL patterns are "heavyweight," in that they are costly to develop, being geared at automated analysis and manipulation. In contrast, our patterns are "lightweight," as they are aimed at communication with practitioners. While NRL focuses on how patterns should be best represented, the primary thrust of our research is to collect the correct patterns. Ultimately, it is conceivable that the two efforts can be merged, with the structural patterns from our repository being encoded using the notation that NRL develops.

SECURITY PATTERNS AT PLOP

The annual Pattern Languages of Programs (PLoP) workshop brings together patterns researchers working in many different domains. Two papers have been presented at PLoP that discuss the use of patterns to address security requirements. Both of these are very low-level approaches, providing techniques for developing object-oriented software that is flexible with regard to security. The first such paper presents an object-oriented abstraction of existing cryptographic APIs [3]. Using these patterns, a developer can construct software that can easily be ported from one cryptographic API to another. The second paper presents developers with strategies for leaving security placeholders within code, so that basic access control mechanisms can be added at a later date [19]. These papers are important in that they introduce the idea of security-specific patterns. However, they do not really raise the user's awareness of the pitfalls and trade-offs that are present.

SECURITY PATTERNS MAILING LIST

Over the past year, a small group of researchers working in this space has used a sporadic mailing list to share their work. Details of the mailing list can be found at **http://www.security-patterns.de**. The individual researchers have developed a number of patterns, largely in isolation. The Web site has a fairly comprehensive list of links in this area.

Most of the patterns presented on the mailing list have been conventional design patterns and have used a template similar to the Gang of Four. A few, however, have been procedural patterns.

OPENGROUP SECURITY FORUM

The OpenGroup Security Forum is currently developing a library of architectural security patterns, available on-line. The patterns they are developing exist at a higher level of abstraction than our patterns, essentially being what we have termed "pattern families." Ultimately, our patterns may serve as concrete examples of their patterns, and their patterns may provide precise statements of need for our patterns.

Conclusions

We have identified a number of candidate security patterns, collected in Version 1.0 of our Security Patterns Repository at

http://www.securitypatterns.com. Our initial investigation into security patterns has produced a promising package for collecting and conveying security expertise.

The next step is to evaluate the utility of the specific patterns in our repository. In the patterns community, formal evaluation does not occur on a specific pattern prior to publication. The evaluation process consists of feedback and discussion in a public forum to reach consensus concerning the validity and utility of a particular pattern. By publishing our repository of security patterns on the Web and providing a mechanism for collecting feedback, we hope that the security and patterns communities will assess our existing patterns and provide suggestions for new security patterns. We will incorporate feedback and maintain the repository on-line.

Evaluation of the security patterns in Version 1.0 of the repository sets the context for more formal evaluation of the security patterns concept as a whole. We hope to collect structured feedback from developers who attempt to use the patterns on actual projects. Potentially, we could evaluate the security patterns in a university course where students would utilize patterns in development of a Web application.

We consider our Security Patterns Repository Version 1.0 a positive result from this project. Only extensive evaluation from the community at large will determine whether the security patterns concept itself produces a positive or negative result.

Acknowledgements

This project was funded by the Defense Advanced Research Projects Agency (DARPA) through the Air Force Research Laboratory under contract #F30602-01-C-0164. We would like to thank our DARPA program manager, Brian Witten, for supporting this project. We would also like to thank David Tyree, Jim Edwards-Hewitt, and James Croall for their significant contributions to the Security Patterns Repository.

References

[1] AG Communication Systems. “Pattern Template”. http://www.agcs.com/supportv2/techpapers/patterns/template.htm, 2001.

[2] Alexander, C., S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[3] Braga, A., C. Rubira, and R. Dahab. "Tropyc: A Pattern Language for Cryptographic Software". Pattern Languages of Programs 1998, Monticello, IL, 1998.

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

[5] Brown, W., R Malveau, H. McCormick, and T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, 1998.

[6] Fernandez, E. "Metadata and Authorization Patterns". Technical report TR-CSE-00-16, Computer Science and Engineering Department, Florida Atlantic University, 2000.

[7] Fernandez, E. and R. Pan. "A Pattern Language for Security Models". Pattern Languages of Programs 2001, Monticello, IL, 2001.

[8] Gabriel, R. Patterns of Software: Tales from the Software Community. Oxford University Press, 1997.

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

[10] Hays, V., M. Loutrel, and E. Fernandez. "The Object Filter and Access Control Framework". Pattern Languages of Programs 2000, Monticello, IL, 2000.

[11] The Hillside Group. "Patterns Home Page – Your Patterns Library". http://www.hillside.net/patterns, 2002.

[12] Kienzle, D. *Practical Computer Security Analysis*. Ph.D. Dissertation, Department of Computer Science, University of Virginia, January 1998.

[13] Kienzle, D., M. Elder, D. Tyree, and J. Edwards-Hewitt. "Security Patterns Template and Tutorial". http://www.securitypatterns.com/documents.html, February 2002.

[14] NSTISSC (National Security Telecommunications and Information Systems Security Committee). *National Information Systems Security (Infosec) Glossary*. NSTISSI No. 4009, September 2000.

[15] The Open Group. "Guide to Security Patterns". http://www.opengroup.org/security/gsp.htm, 2002.

[16] Schumacher, M. and U. Roedig. "Security Engineering with Patterns". Pattern Languages of Programs 2001, Monticello, IL, 2001.

[17] Security Patterns Mailing List. http://www.security-patterns.de, 2002.

[18] Vlissides, J. Pattern Hatching: Design Patterns Applied. Addison-Wesley, 1998.

[19] Yoder, J. and J. Barcalow. "Architectural Patterns for Enabling Application Security". Pattern Languages of Programs 1997, Monticello, IL, 1998.

Final technical report: Security pattern for web application development, 2002. http://www.scrypt.net/celer/securitypatterns/final.Google Scholar. P. Morrision and E. B. Fernandez. Securing the broken pattern.Â J. Yoder and J. Barcalow. Architectural patterns for enabling application security. In International Conference on Pattern Language of Programs. PLoP, 1997.Google Scholar. Index Terms. Evaluation of web application security risks and secure design patterns. Security and privacy. Cryptography. Secure web application development should be enhanced by applying security checkpoints and techniques at early stages of development as well as throughout the software development lifecycle. Special emphasis should be applied to the coding phase of development.Â OWASP is the emerging standards body for web application security. In particular they have published the OWASP Top 10,[8] which describes in detail the major threats against web applications. The Web Application Security Consortium (WASC) has created the Web Hacking Incident Database (WHID) and also produced open source best practice documents on web application security. The WHID became an OWASP project in February 2014.[9]. Security technology[edit]. Final Technical Report: Security Patterns for Web Application Development. Article. Darrell M. Kienzle.Â A well-designed user interface is important for security applications, but it is critical if the adequate use, and the effectiveness of security features, depend on it. Currently, many criteria are available to facilitate the design of a user interface, like the new HCI-S or Security Human Computer Interaction, which is focused in the design of user interfaces for security applications. Similar approaches have emerged recently, such as the use of sonification alerts to notify to the users about malicious attacks either in real time or during the analysis of network logs, in forensics.