# Architectural Risk Analysis of Software Systems Based on Security Patterns

Spyros T. Halkidis, Nikolaos Tsantalis, *Student Member*, *IEEE*, Alexander Chatzigeorgiou, *Member*, *IEEE*, and George Stephanides, *Member*, *IEEE*

**Abstract**—The importance of software security has been profound, since most attacks to software systems are based on vulnerabilities caused by poorly designed and developed software. Furthermore, the enforcement of security in software systems at the design phase can reduce the high cost and effort associated with the introduction of security during implementation. For this purpose, security patterns that offer security at the architectural level have been proposed in analogy to the well-known design patterns. The main goal of this paper is to perform risk analysis of software systems based on the security patterns that they contain. The first step is to determine to what extent specific security patterns shield from known attacks. This information is fed to a mathematical model based on the fuzzy-set theory and fuzzy fault trees in order to compute the risk for each category of attacks. The whole process has been automated using a methodology that extracts the risk of a software system by reading the class diagram of the system under study.

**Index Terms**—Security patterns, fuzzy risk analysis, dependability analysis, software security, design patterns, security architecture, software architecture.

## 1 Introduction

The importance of software security has been evident since the discovery that most attacks to real software systems are initiated by poorly designed and developed software [49], [47], [20], [21]. Furthermore, it has been shown that the earlier we incorporate security in a software system, the better this would be in terms of effort and cost [49], [34]. Therefore, in analogy to design patterns [17], which aim at making software well structured and reusable, Security Patterns [48], [4] have been proposed, targeting at imposing some level of security already at the design phase.

One reasonable research aim would be to estimate the security imposed in a software system by examining which security patterns are used and where they reside in the design. To achieve this, we first determine to what extent several security patterns are robust to known categories of attacks [21]. In order to determine this kind of information, we have built two Web applications and studied them under known attacks [43]. Having made this study, we propose estimates for the resistance of the examined security patterns to Spoofing, Tampering-with-Data, Repudiation, Information Disclosure, Denial-of-Service, and Elevation-of-Privilege (STRIDE) [21] attacks. Additionally, we propose a new security pattern based on our findings for an attack not covered by existing security patterns. Then, we use results from fuzzy reliability [8] and fault trees [1], [7] to propose a mathematical model that examines the proper use of the security patterns and calculates risk for each category of STRIDE [21] attacks. We have automated the process of determining risk based on this model by building software that takes as input the XML file of the class diagram that corresponds to the system under consideration and produces the calculated risk for each category of STRIDE attacks. This methodology can be a part of a larger risk management framework [34]. Finally, we measure the change in calculated risk when gradually adding security patterns in order to estimate the impact of each pattern on the security level of the system.

### 1.1 Related Work

Existing approaches for quantifying the security of software systems can be divided into two categories: dependability-based approaches and risk-analysis-based approaches. The former, in general, aim at calculating security measures such as the mean time to security failure employing Markov models. The latter aim at calculating the total risk for a system based on the analysis of fault trees, considering the likelihood and impact of events. However, both approaches extract risk or security measures and thus can be considered very similar, at least within the context of software engineering.

The dependability community [38] tried addressing the problem of quantifying the security of software systems by finding analogies between security and reliability problems. The first investigation into mathematical models for the security of systems was done by Littlewood et al. [29]. In their work, analogies between notions used in reliability with notions used in security have been examined. Since then, various mathematical models have appeared in the literature. Madan et al. proposed a mathematical model applicable to fully implemented systems based on Markov Chains [32]. Goseva-Popstojanova and Trivedi

- *The authors are with the Computational Systems and Software Engineering Laboratory, Department of Applied Informatics, University of Macedonia, GR-54006 Thessaloniki, Greece. E-mail: {halkidis, nikos}@java.uom.gr, {achat, steph}@uom.gr.*

*Manuscript received 31 Aug. 2006; revised 11 June 2007; accepted 22 Oct. 2007; published online 2 Nov. 2007.*
*For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0121-0806.*
*Digital Object Identifier no. 10.1109/TDSC.2007.70240.*

proposed a component-based approach. In their approach, the mathematical model for a software system is derived based on the components that it contains, and the parameters of the Markov Chain model corresponding to these components are assumed to be known [18].

In risk analysis methodologies the likelihood, exposure and consequences for events related to security are determined, and then, through mathematical techniques, the risk for the system is calculated [1]. Our technique belongs to this category. The need for risk analysis at design level has been particularly emphasized by McGraw [34]. According to him, "Design flaws account for 50 percent of security problems, and architectural risk analysis plays an essential role in any solid security program."

The importance of security patterns to secure software systems has been recently illustrated [48], [4]. The pioneering work on security patterns was by Yoder and Barcalow [54] in 1997. Since then, various security patterns have been introduced: patterns for enterprise applications [41], patterns for authentication and authorization [27], [13], patterns for Web applications [25], [51], patterns for mobile Java code [33], patterns for cryptographic software [6], and patterns for agent systems [36]. However, all these efforts did not share some common terminology.

The first effort to provide a comprehensive review of existing security patterns was done by the OpenGroup Security Forum [4]. In this work, security patterns are divided into Available System Patterns, which are related to fault tolerance [39] and Protected System Patterns, which aim at protecting resources.

In an earlier work [19], we have performed a qualitative evaluation of these security patterns.

Recently, a summary of security patterns has appeared in the literature [48]. In this text, security patterns were divided into Web-tier security patterns, business-tier security patterns, security patterns for Web services, security patterns for identity management, and security patterns for service provisioning. In this paper, we focus on Web-tier and business-tier security patterns.

### 1.2 Contribution

To the best of our knowledge, this paper is the first to experimentally examine the resistance of several security patterns to known categories of attacks. The main contribution of this paper is to propose a complete methodology for calculating the risk of STRIDE [21] attacks on a software system composed of security patterns *already from its design*. Additionally, we make use of a fuzzy risk analysis framework. Using fuzzy terms is more appropriate when examining the design of a system for security. We cannot apply exact numbers due to the lack of exact information about the security of the system [20]. We note here that we make use of nine levels of risk, which leads to better granularity compared to using fewer levels. Additionally, our approach is security pattern centric. All security estimates are based on used and missing security patterns in places where they are needed. Finally, in this paper, we propose a new security pattern against an attack that we discovered during our experiments and that existing security patterns do not protect against.

### 1.3 Organization

The rest of this paper is organized as follows: Section 2 describes the systems that we used to experimentally determine the resistance of several security patterns to known categories of attacks. Section 3 contains preliminaries on the fuzzy-set theory and calculations on fuzzy fault trees. In Section 4, the methodology for constructing fuzzy fault trees from UML-class diagrams is described. In Section 5, experimental results are presented, concerning the resistance of security patterns to known attacks, risk evaluation of a nonsecure and a secure system, and the risk evolution when patterns are introduced in different orders. In Section 6, we propose and evaluate a new security pattern named "Secure GET Parameters." Finally, in Section 7, we draw some final conclusions and propose future work.

## 2 Subject Systems for the Experiments

In order to experimentally examine the robustness of various security patterns to known attacks, we have developed two systems. The first system, hereafter denoted as nonsecure application, is a typical e-commerce application with no usage of security patterns, except for Protected System [4], where various sources for attacks were deliberately included. The second system, hereafter denoted as secure application, is a variant of the nonsecure application, where the sources for attacks were not removed, but security patterns have been used to protect against the attacks.

The criteria for selecting the specific systems were

1. containment of most common sources for attacks,
2. knowledge of the exact location of each security hole,
3. accessibility to the source code, and
4. selection of a typical Web application such as an e-commerce system.

Moreover, there is no widely accepted benchmark system for assessing software security.

Both applications under examination are typical Java 2 Enterprise Edition (J2EE; now referred to as Java EE) [40]. We have chosen J2EE as a platform for both applications, since J2EE is widely used for business applications and offers a wide variety of security features [48], [3].

The architecture of a typical J2EE system is shown in Fig. 1. The client, typically a Web browser, accesses the Web Tier where servlets reside. Servlets can forward requests to Enterprise Java Beans (EJBs) residing in the Business Tier, some of which provide access to the database. We have used JBoss 4.0.3 [23] as an application server for the Web and business tiers and MySQL 5.0 [37] for the database tier.

The nonsecure system consists of 46 classes. It has 16 servlets and seven EJBs, where one EJB serves as a Web service end point [40].

First, three sources for SQL injection attacks were included in this application. An SQL injection attack [43], [2], [46], [15] occurs when an attacker is able to insert a series of SQL statements in a query formed in an application by exploiting nonexisting or improper validation of data [2]. An

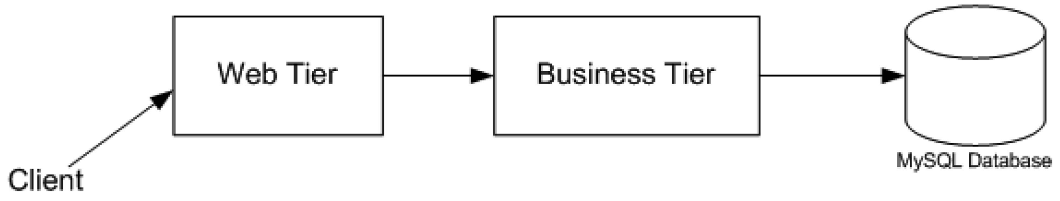

Fig. 1. A typical J2EE architecture.

SQL injection attack can cause unauthorized viewing of database data and database modification.

Additionally, 11 sources for cross-site scripting (XSS) have been included. An XSS attack [43], [11], [45], [22] occurs when not properly validated data input in one page is shown in another. In this case, script code can be input in the former page to be consequently executed in the latter. This way, it is easy to perform an Information Disclosure attack [21], for example, by inserting Javascript code in the former page, that shows cookie values containing sensitive information such as credit card numbers in the latter.

Furthermore, a source for HTTP Response Splitting [26] was included. HTTP Response Splitting attacks occur when user data that was not properly validated is included in the redirection URL of a redirection response or when improperly validated data is included in the cookie value or name when the response sets a cookie. In both cases, it is easy to create two responses, instead of one, by manipulating headers. In the second response, an XSS attack can be performed [26].

Moreover, there was no SSL connection used in the nonsecure application, and as a result, crucial information such as credentials and cookie values containing credit card information could be eavesdropped.

Finally, six servlet member variables race conditions were included, which could be exploited by having a number of users acting simultaneously. A summary of the security vulnerabilities present in the nonsecure application is shown in Table 1.

The secure application consists of 62 classes. It has 17 Servlets and nine EJBs, where, again, one EJB serves as a Web service end point [40]. The security patterns that were added in this system are the Secure Proxy (Login Tunnel variant) [4], the Secure Pipe [48], the Secure Logger (Secure Log Store Strategy variant) [48], and the Intercepting Validator [48] (where a white listing approach for validation of input was used [20]).

## 3 Fuzzy Analysis of the Subject Systems

The main target in our research was to build a mathematical model for systems that use security patterns based on our findings for the level of security that each pattern offers (as explained in Section 5.1). The most appropriate models for our purpose seem to be risk analysis models [1]. We have chosen to avoid the use of deterministic numbers, because it is impractical to specify security characteristics of software systems using exact values. As Hoglund and McGraw [20] note, in software risk analysis, exact numbers as parameters work worse than having values such as high, medium, and low. These kinds of values can be termed as fuzzy. Additionally, the inapplicability of exact numbers is more prominent when we try applying a mathematical model already at the design phase, where less information is available.

Concerning the use of a probabilistic or fuzzy uncertainty model for describing selected system design properties, the choice depends on the characteristics of the available information [35]. Employing a probabilistic model assumes the knowledge of sufficient statistical information to extract the required probability distribution functions. In our case, this information is not available, since it requires results from a large number and variety of attacks to systems with and without the considered security patterns.

Within the context of our methodology, the engineer has no access to such large amount of information and has to quantify risk on the basis of few data, which is additionally characterized by vagueness. In other words, he/she has only a rough idea concerning the value of likelihood, exposure, and consequences for each attack. To this end, the fuzzy-set theory provides a plausible alternative to probabilistic models, taking into account that it is convenient to employ linguistic terms for assessing risk.

Furthermore, the applicability of fuzzy techniques to security problems has already been proposed [9], and the use of fault trees for secure system design has also been suggested [1], [7]. In this paper, we perform an analysis of the security of systems employing security patterns by

TABLE 1
Summary of Vulnerabilities Present in the Nonsecure Application

| Type of vulnerability | Number of sources for attack |
|---|---|
| SQL Injection | 3 |
| Cross-Site Scripting | 11 |
| HTTP Response Splitting | 1 |
| Servlet member variable race conditions | 6 |
| Eavesdropping | 3 |

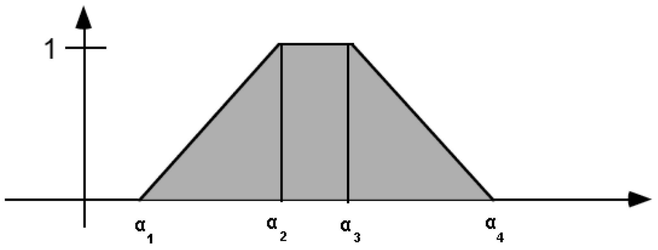

Fig. 2. Example of a trapezoidal fuzzy number membership function.

using results from the fuzzy-set theory [55] and fuzzy fault trees [8].

## 3.1 Preliminaries on the Fuzzy-Set Theory

Commonly used types of fuzzy numbers are triangular and trapezoidal fuzzy numbers. Trapezoidal fuzzy numbers are more appropriate when there is a higher degree of uncertainty in the variable values [55].

$A$ is referred to as a trapezoidal fuzzy number, denoted $A = (a_1, a_2, a_3, a_4)$ if its membership function $\mu_{\mathrm{A}}(x)$ is determined by [8]

$$\mu_{\mathrm{A}}(x) = \begin{cases} \frac{(x-a_1)}{(a_2-a_1)}, & a_1 \leq x \leq a_2, \\ 1, & a_2 \leq x \leq a_3, \\ \frac{(a_4-x)}{(a_4-a_3)}, & a_3 \leq x \leq a_4, \\ 0, & otherwise. \end{cases}$$

In Fig. 2, an example of a trapezoidal fuzzy number membership function is depicted.

Another tool in our analysis is fuzzy fault trees [8]. The root node of fault trees represents an undesirable top event [7], which, in our case, is a successful security breach, and the other nodes represent either primary events, which, in our case, initiate an attack, or intermediate events, which are described by the logical composition of primary and/or other intermediate events. In fuzzy fault trees, all events are represented by fuzzy variables [8].

Additionally, we use results from risk analysis. When performing risk analysis for a system, a common formula used by the risk engineering community is the following [9]:

$$R = L \cdot E \cdot C,$$

where $L$ is the likelihood of occurrence of a risky event, $E$ is the exposure of the system to the event, $C$ is the consequences of the event, and $R$ is the computed risk. Examining this equation in comparison to the risk analysis performed by Hoglund and McGraw [20], in our case, the likelihood $L$ is the likelihood of a successful attack, the exposure $E$ is a measure of how easy it is to carry out the attack, and $C$ is the impact of the attack.

## 3.2 Fuzzification, Fuzzy Calculations, and Defuzzification

The proposed analysis differs from a general fuzzy inference system in the sense that no Rule base and Inference Engine exists [28]. Instead, the risk of the system under study is obtained explicitly by fuzzy calculations. The primary inputs to our analysis are the values corresponding to Likelihood, Exposure, and Consequences for each malicious event that can take place in the examined system. These values can be described by appropriate linguistic terms [12], as shown in the left column of Table 2. Fuzzification is performed according to the mapping proposed by Chen and Chen [12], as shown in Table 2. It should be mentioned that at all steps, simple fuzzy numbers, instead of generalized fuzzy numbers, are used; that is, all the weights in the generalized fuzzy numbers and in the formulas are equal to 1.0. For each category of STRIDE attacks, a separate fault tree is constructed. Each gate in the tree corresponds to a fuzzy calculation, meaning that based on the fuzzy values of the input variables, a fuzzy value for the output variable is obtained. Input variables correspond to the risk of each event, calculated by the fuzzy multiplication of likelihood, exposure, and consequences. For the AND and OR gates of the fault tree, the output is obtained by the classical set operators of the fuzzy-set theory, namely, the *min* and *max* of the input fuzzy values, respectively [28]. Obviously, the value corresponding to the top event of the fault tree is the risk of the entire system under study.

TABLE 2
Mapping of Linguistic Terms to Fuzzy Numbers

| Linguistic Term | Fuzzy Number |
|---|---|
| absolutely-low | (0.0, 0.0, 0.0, 0.0) |
| very-low | (0.0, 0.0, 0.02, 0.07) |
| low | (0.04, 0.1, 0.18, 0.23) |
| fairly-low | (0.17, 0.22, 0.36, 0.42) |
| medium | (0.32, 0.41, 0.58, 0.65) |
| fairly-high | (0.58, 0.63, 0.80, 0.86) |
| high | (0.72, 0.78, 0.92, 0.97) |
| very-high | (0.93, 0.98, 1.0, 1.0) |
| absolutely-high | (1.0, 1.0, 1.0, 1.0) |

The defuzzification process consists of finding the linguistic level corresponding to the fuzzy risk value. This is achieved by employing a similarity metric [12] based on a center-of-gravity technique. Eventually, a linguistic term representing the risk level of the entire system with regard to each category of STRIDE attacks is extracted.

## 4 Description of the Methodology

As already mentioned, the methodology proposed can be applied already at the design phase. However, in order to appropriately perform our analysis, the information present in a plain UML [14] diagram is not sufficient. We have considered using UMLSec [24] in our analysis, but we have concluded that a much smaller extension to UML is sufficient. A Java tool applying the proposed methodology can be found in [16].

### 4.1 Annotation of the Unified Modeling Language Class Diagram

An important piece of information for our analysis is which security patterns exist in the design and where they reside. We have used a commercial software engineering tool [5] for defining templates of all security patterns. For example, the Intercepting Validator [48] security pattern annotated

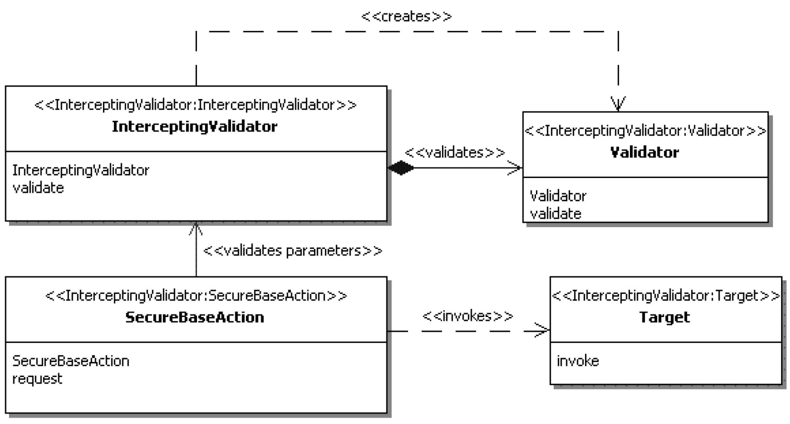

Fig. 3. Annotated class diagram of the Intercepting Validator security pattern.

with the appropriate information in stereotypes is depicted in Fig. 3. The annotations required in our extension are summarized by the following conventions, which are relatively easy to apply:

1. For each class that is part of a security pattern, the class stereotype should contain the name of the security pattern and the role of the class in this pattern. This requirement is automatically satisfied through the use of templates.
2. For each class where data is entered, the stereotype should contain the word "Input."
3. For each class that accesses a resource (for example, a database) or displays input data, the stereotype should contain the general word "accessesResource."
4. For each class acting as an application entry point (where users should be authenticated before logging in to the system), the stereotype should contain the word "ApplicationEntryPoint."
5. For each class that performs logging, the stereotype should contain the word "performsLogging."
6. For each class that sends parameters in the URL through a GET request to another class, the association/dependency between these classes should contain in the stereotype the phrase "parametersThroughGET."
7. In the case that a class or association or dependency should contain multiple words/phrases in its stereotype, these are separated by a ":".

After having designed the annotated class diagram of the system, the information present in this diagram has to be extracted in a form that can be automatically processed. We have chosen to export the class diagram in the form of XML, specifically in XMI for the UML 1.4 (OMG) format [53]. An XMI parser that we have developed extracts all the required information for our analysis such as classes, associations, dependencies, generalizations, and stereotypes.

## 4.2 Generation of Fault Trees

For each category of STRIDE attacks, a separate fault tree is constructed. A fault tree is represented as an expression, where a "$*$" corresponds to an AND gate, and a "$+$" corresponds to an OR gate. The factors of the fault trees are added gradually by examining one by one the implemented/missing security patterns of the design. Each factor represents a possible attack to the system under study. For each factor, the values for likelihood, exposure, and consequences are as shown in the corresponding cases in Tables 4, 5, and 6, respectively (Section 5.1). Specifically.

### 4.2.1 Intercepting Validator (Case 1)

A graph is built, where the classes are the vertices, and the associations and dependencies are the edges. Then, all paths from classes, which serve as input forms to classes that access resources, are examined. For each such path where no association or dependency from a vertex of the path to an Intercepting Validator pattern instance exists, a factor to the fault trees for Information Disclosure and Tampering with data attacks is added (as an input to the OR gate leading to the top event).

### 4.2.2 Protected System/Secure Proxy (Cases 2-5)

For each class acting as an application entry point, we examine whether there exists an association or dependency to a Protected System or Secure Proxy pattern. If no authentication mechanism exists, a factor corresponding to case 2 is added to the fault trees for Spoofing Identity, Elevation of Privilege, and Information Disclosure. If Protected System is used as an authentication mechanism, a factor corresponding to case 3 is added to the same fault trees for the case that the guard is compromised. Finally, if Secure Proxy is used as an authentication mechanism, both guards must be compromised for an attack to succeed, and therefore, an AND gate (whose output serves as input to the OR gate leading to the top event) having as input the factors corresponding to cases 4 and 5 is added to the same fault trees.

### 4.2.3 Secure Pipe (Case 6)

The system is examined for the existence of an SSL connection. If no Secure Pipe pattern is present in the system, a factor to the fault trees for Spoofing Identity, Information Disclosure, and Elevation of Privilege is added, since information could be eavesdropped.

### 4.2.4 Secure Logger (Case 7)

For each class that logs messages and is not part of a Secure Logger pattern instance, a factor to the fault trees for Tampering-with-Data, Information Disclosure, and Repudiation attacks is added.

#### 4.2.5 Secure Get Parameters (Case 8)

The system is examined for associations or dependencies, where parameters are sent through a GET request. If the communicating classes are not part of a "Secure GET Parameters" pattern instance (see Section 6), a factor to the fault tree for Information Disclosure attacks is added.

#### 4.2.6 Servlet Member Variable Race Conditions Exploited (Case 9)

Since there is no security pattern at the architectural level to avoid servlet member variable race conditions, a factor is always added to the fault tree for Information Disclosure attacks.

## 5 Experiments and Results

To evaluate the usefulness of security patterns to the remediation of risk, three experiments have been carried out: the first aims at estimating the resistance of each security pattern to known attacks, the second quantifies the difference in risk level between the secure and nonsecure systems, and the third investigates the impact on risk when introducing patterns in different orders to the nonsecure system.

According to Sjøberg et al. [44], the above experiments do not fall in the category of controlled experiments in software engineering, since the subjects are projects, where data is collected at several levels rather than individuals or teams conducting one or more software engineering tasks. Particularly, within the context of software reliability analysis [10], the above experiments belong to empirical studies, since data of interest concerning software artifacts is collected and analyzed, with the purpose of testing a known theory (that is, investigating whether Security Patterns can fortify software systems). Essentially, the difference between the two approaches is that according to Sjøberg et al. [44], several subjects are analyzed within the context of the same experimental study, whereas according to Cai [10], a single project (possibly with variations) is subjected to several trials of testing. Consequently, our experimental methodology is closer to the approach of Cai [10], since a single-subject program is employed (and variants of it), which is subjected to several kinds of attacks.

### 5.1 Evaluation of Likelihood, Exposure, and Consequences for Attacks Corresponding to Missing Security Patterns

The first experiment consists of subjecting the two systems described in Section 2 to known attacks employing two different approaches. First, we have used a commercial Web application penetration testing tool [50]. Second, we have employed a static analysis tool [30], [31] that won a contest initiated in various newsgroups, where the participants performed attacks to the systems that we described in Section 2.

Both approaches found the major security flaws of the nonsecure application, that is, the three SQL Injection and the 11 XSS vulnerabilities. Denial-of-Service attacks and the HTTP response-splitting source in the nonsecure application were found by neither approach.

The Web Application Penetration Testing tool found minor application errors that pose no threat to security, not found by the static approach (like the lack of checking for proper session variable value ranges). It also found the unencrypted login request flaw in the nonsecure application that did not use SSL. Furthermore, it found unencrypted GET parameter flaws in the secure application. This approach had several false positives by finding sources for buffer overflows when Java was used.

Race conditions for servlet member variables were found only by the static analysis approach. This approach had also several false positives by finding sources for SQL Injection in the secure application by examining the code for the EJBs, whereas proper input validation was done by patterns in the Web Tier.

The analysis of the attack results for the secure application shows that proper use of the security patterns leads either to the total remediation or to a high level of remediation of all major security flaws. The flaws that are not totally eliminated are related to the authentication mechanism. Specifically, dictionary attacks can succeed [52], [42], even when a proper authentication pattern is used. Moreover, attacks can also exploit servlet member variable race conditions, which cannot be confronted at the design level.

What follows is an evaluation of the resistance of security patterns to STRIDE attacks. Additionally, we perform a likelihood-exposure-consequences analysis of attacks on security patterns or more commonly attacks based on the lack of a specific security pattern in a place where it is needed. The analysis is based on the results of the attacks to the nonsecure and the secure system described in Section 2, as well as on subjective judgment. However, the proposed methodology is not affected from the particular values for likelihood, exposure, and consequences, which are actually input parameters to our approach and can be set freely according to personal estimates.

The Intercepting Validator pattern [48], when used for all kinds of input, including session variables that are not entered by the user but are still posted, protects from SQL Injection, XSS, and HTTP Response-Splitting attacks. Therefore, it offers good protection against Tampering-with-Data and Information Disclosure attacks [21]. We consider the resistance of this pattern as very high and not absolutely high, although it offers absolute protection from these attacks when used properly. We lower the estimate of its resistance in order to account for the improper implementation of the pattern during development. We make conservative estimates for the other security patterns as well.

The Secure Proxy pattern Login Tunnel variant [4] has two levels of authentication in order to protect from Spoofing Identity, Elevation-of-Privilege, and Information Disclosure attacks. Its resistance to the related attacks can be estimated by considering it to be equivalent to the existence of two guards [4] connected in a series. The resistance of this pattern to attacks is dependent upon the resistance of each

TABLE 3
Resistance of the Security Patterns Examined against STRIDE Attacks

| Security Pattern | S | T | R | I | E |
|---|---|---|---|---|---|
| Intercepting Validator | | very high | | very high | |
| Protected System with Secure Pipe | high | | | high | high |
| Secure Proxy with Secure Pipe | very high | | | very high | very high |
| Secure Logger | | very high | very high | very high | |

guard to dictionary attacks. Specifically, in order for both guards to be compromised, two consecutive dictionary attacks to the authentication mechanism of a guard must succeed. Recent studies [52], [42] have shown that dictionary attacks with a usual distribution of the complexity of the passwords selected succeed 15 percent to 20 percent of the times. The authentication mechanism of a guard can still be marked as of high security.

All authentication patterns and, consequently, the Protected System [4] and the Secure Proxy pattern should be resistant to eavesdropping attacks to serve their purpose. Thus, they should always be used together with the Secure Pipe pattern that enforces the use of the SSL protocol [48]. The Secure Pipe pattern offers protection from Information Disclosure attacks.

Finally, the Secure Logger [48] pattern offers a strong protection mechanism from reading/tampering the logs, preventing from Tampering-with-Data, Repudiation, and Information Disclosure attacks.

Based on the above analysis, we can make conclusions about the resistance of the security patterns under consideration to known categories of attacks [21]. The results are summarized in Table 3. Irrelevant entries to the specific security pattern are left blank. Since we have not considered security patterns that can confront Denial-of-Service attacks, the corresponding category has been eliminated from our analysis.

Next, we perform a likelihood-exposure-consequences [1], [9] investigation for attacks that occur in cases where specific security patterns are missing and cases where the security patterns used do not offer total protection. Our investigation is based on the previous analysis, together with knowledge on possible attacks on Web Applications [43].

We note that the likelihood and the exposure (ease) of an attack are the same, regardless of the application, whereas the consequences depend on the data affected and, thus, on the specific application. Although in our investigation, consequences for the specific applications could be considered, we examined the worst case scenario for the consequences, considering that all system data is of crucial importance.

Regarding the authentication mechanism, the categories of attacks affected when the authentication mechanism is broken are Spoofing, Information Disclosure, and Elevation of Privilege (if someone gets administrator rights) [21]. The most trivial case is when no authentication is used at an application entry point. In this case, the likelihood of an attack is very high, the ease of performing an attack is very high, and the consequences are damaging (very high). When the Protected System pattern is used, the likelihood of successfully attacking a guard of this pattern is low, the ease (exposure) of a dictionary attack can be regarded high, and the consequences are very high. When the Secure Proxy pattern is used, two guards must be compromised for an attack to succeed. The likelihood and exposure of compromising the first guard are the same as in the case of a guard of Protected System. The consequences of attacking the first guard are very low, since the first guard only acts as a front end to the second guard, and no resources are compromised yet when the first guard is compromised. The likelihood, exposure, and consequences of attacking the second guard are the same as in the case of a guard of Protected System. The consequences of attacking the second guard of Secure Proxy are very high, because if the second guard is compromised, then all the protected resources are compromised.

In case the Secure Logger pattern is not used in a place where logging is performed, the categories of attacks affected are Tampering with Data, Repudiation, and Information Disclosure. If the server where the logs reside is compromised, the log data can be read and changed, letting a user deny having performed an action. The likelihood of such an attack and the ease of such an attack are low, since generally, it is not easy to compromise the server where the logs reside. The consequences regarding Tampering with Data and Information Disclosure are low, since the data kept in the logs is not usually of high importance. The importance of the logs is, however, very high when considering Repudiation (someone could deny having performed an action that he/she performed, or conversely, someone could accuse someone else of having performed an action that he/she did not), and therefore, the consequences are also very high.

When the Secure Pipe pattern is not used, the application may not be configured to work with an SSL connection. In this case, important data could be eavesdropped, leading to an Information Disclosure attack, and additionally, if the credentials are eavesdropped, this would lead to Spoofing and Elevation of Privilege. The likelihood of an eavesdropping attack in this case can be considered high, the ease of such an attack is high, and the consequences for all categories affected are very high.

When no Intercepting Validator is used in a path from a class where data is input to a class where this data is shown or a resource (for example, a database) is accessed, having this data as a parameter, then an SQL Injection and/or an XSS attack could occur. The categories of attacks affected are Information Disclosure, which can occur in both types of attacks, and Tampering with Data, which can occur in an

TABLE 4
Likelihood of an Attack in All Examined Cases

| Case | S | T | R | I | E |
|---|---|---|---|---|---|
| **1. Missing Intercepting Validator** | | high | | high | |
| **2. No Authentication** | very high | | | very high | very high |
| **3. Protected System compromised** | low | | | low | low |
| **4. First Guard of Secure Proxy compromised** | low | | | low | low |
| **5. Second Guard of Secure Proxy compromised** | low | | | low | low |
| **6. Missing Secure Pipe** | high | | | high | high |
| **7. Missing Secure Logger** | | low | low | low | |
| **8. Missing “Secure GET parameters”** | | | | low | |
| **9. Servlet member variable race conditions exploited** | | | | low | |

TABLE 5
Exposure of an Attack in All Examined Cases

| Case | S | T | R | I | E |
|---|---|---|---|---|---|
| **1. Missing Intercepting Validator** | | high | | high | |
| **2. No Authentication** | very high | | | very high | very high |
| **3. Protected System compromised** | high | | | high | high |
| **4. First Guard of Secure Proxy compromised** | high | | | high | high |
| **5. Second Guard of Secure Proxy compromised** | high | | | high | high |
| **6. Missing Secure Pipe** | high | | | high | high |
| **7. Missing Secure Logger** | | low | low | low | |
| **8. Missing “Secure GET parameters”** | | | | very high | |
| **9. Servlet member variable race conditions exploited** | | | | low | |

SQL Injection attack, where the database is modified. When an Intercepting Validator is missing, the likelihood of such an attack is high, the ease of such an attack is high, and the consequences are damaging (very high).

When information entered by users is transferred through the GET method, the corresponding variable values are encoded in the URL (independent of whether the Web Application uses an HTTPS protocol or not). In this case, attacks that can exploit (locally or remotely) the history of visited URLs in the client’s browser by harvesting GET request parameter values might occur. One way of confronting this kind of attacks is to use the proposed “Secure GET Parameters” pattern described in Section 6. The category of attacks affected is Information Disclosure attacks. The likelihood of such attacks is low (since in general, it is more difficult to access directly sensitive data in the client than packets across a network), the ease is very high (since the parameters are easily accessible through the URL), and the consequences are very high (considering the worst case scenario, where the parameters are crucial).

Finally, there exists no pattern yet that provides a proper synchronization and locking mechanism ensuring that no servlet member variable race conditions can be exploited. Thus, there is always the possibility of such an attack with low likelihood, low ease, and very high consequences.

The results about the likelihood, exposure, and consequences of an attack are summarized in Tables 4, 5, and 6, respectively.

## 5.2 Risk Quantification for the Systems under Study

The second experiment consists of evaluating the risk level for all kinds of STRIDE attacks for the secure and the nonsecure applications. Based on the likelihood-exposure-consequences results in Tables 4, 5, and 6, we derive tables of all primary events and the STRIDE categories of attacks that they belong to for the nonsecure and secure applications. The primary events for the nonsecure system are shown in Table 7. We note here that many events can correspond to the same case, as what happens for events 4-14, which correspond to 11 missing uses of the

TABLE 6
Consequences of an Attack in All Examined Cases

| Case | S | T | R | I | E |
|---|---|---|---|---|---|
| **1. Missing Intercepting Validator** | | very high | | very high | |
| **2. No Authentication** | very high | | | very high | very high |
| **3. Protected System compromised** | very high | | | very high | very high |
| **4. First Guard of Secure Proxy compromised** | very low | | | very low | very low |
| **5. Second Guard of Secure Proxy compromised** | very high | | | very high | very high |
| **6. Missing Secure Pipe** | very high | | | very high | very high |
| **7. Missing Secure Logger** | | low | very high | low | |
| **8. Missing "Secure GET parameters"** | | | | very high | |
| **9. Servlet member variable race conditions exploited** | | | | very high | |

TABLE 7
Primary Attack Events for the Nonsecure System

| Primary Event | Likelihood of Occurrence | Exposure | Consequences | Categories of Attacks |
|---|---|---|---|---|
| **Event 1.** Dictionary attack to the guard of Protected System is Successful | low | high | very high | S, I, E |
| **Event 2.** POST/GET Parameters are eavesdropped because of missing Secure Pipe | high | high | very high | S, I, E |
| **Event 3.** Exploitation of Servlet member variables race conditions | low | low | very high | I |
| **Events 4-14.** Attacker reads and/or tampers logs because of missing Secure Logger | low | low | T, I: low<br>R: very high | T, R, I |
| **Events 15-25.** No input validation because of missing Intercepting Validator | high | high | very high | T, I |
| **Event 26.** Exploitation of unencrypted GET parameters because of missing "Secure Get Parameters" (see section 6) | low | very high | very high | I |

Secure Logger pattern, and events 15-25, which correspond to 11 missing instances of the Intercepting Validator pattern, in several places of the design.

The resulting fault tree for Information Disclosure attacks is shown for illustration in Fig. 4. We performed a similar analysis for the secure system. The primary-attack events for the secure system are shown in Table 8. (In the secure system, the Protected System pattern, which essentially consists of one level of authentication, has been replaced by the Secure Proxy pattern, which consists of two levels of authentication. These two levels can still be compromised by a successful dictionary attack.)

The outputs of the fuzzy fault trees for the nonsecure and secure systems are summarized in Table 9. These results show the calculated risk of the entire systems based on the security patterns contained and security patterns missing in places where they are needed. We therefore have managed to quantify the difference between the two systems under STRIDE [21] attacks, which is prominent in all categories, except for Repudiation attacks.

## 5.3 Risk Evolution for Different Pattern Sequences

The third experiment consists of gradually employing security patterns in the nonsecure system in order to evaluate whether the order of patterns affects the evolu-

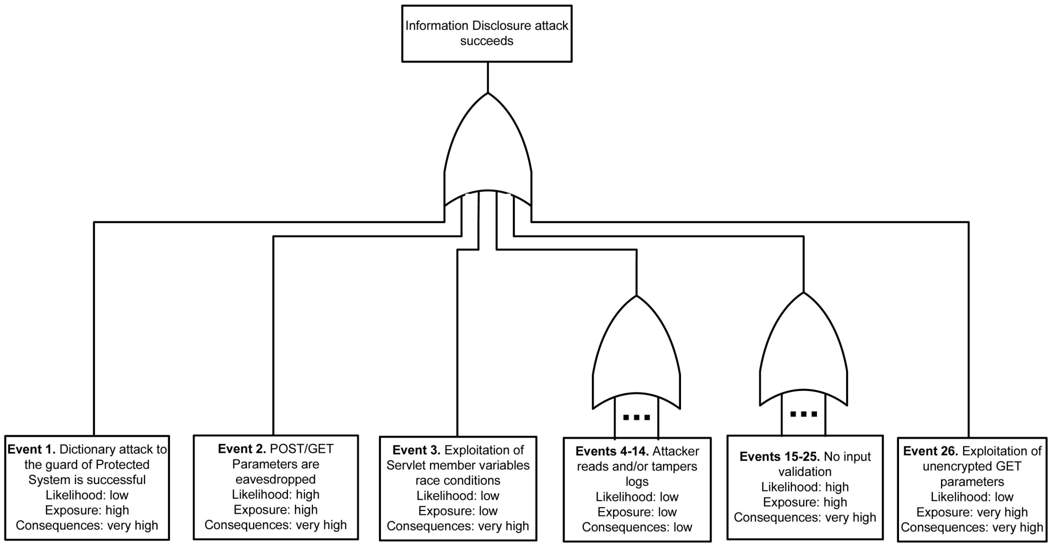
Information Disclosure attack succeeds
...
...
**Event 1.** Dictionary attack to the guard of Protected System is successful Likelihood: low Exposure: high Consequences: very high
**Event 2.** POST/GET Parameters are eavesdropped Likelihood: high Exposure: high Consequences: very high
**Event 3.** Exploitation of Servlet member variables race conditions Likelihood: low Exposure: low Consequences: very high
**Events 4-14.** Attacker reads and/or tampers logs Likelihood: low Exposure: low Consequences: low
**Events 15-25.** No input validation Likelihood: high Exposure: high Consequences: very high
**Event 26.** Exploitation of unencrypted GET parameters Likelihood: low Exposure: very high Consequences: very high

Fig. 4. Fault tree for Information Disclosure attacks when considering the nonsecure system.

TABLE 8
Primary Attack Events for the Secure System

| Primary Event | Likelihood of Occurrence | Exposure | Consequences | Categories of Attacks |
|---|---|---|---|---|
| **Event 1.** Dictionary attack to the first guard of Secure Proxy is successful | low | high | very low | S, I, E |
| **Event 2.** Dictionary attack to the second guard of Secure Proxy is successful | low | high | very high | S, I, E |
| **Event 3.** Exploitation of Servlet member variables race conditions. | low | low | very high | I |

TABLE 9
Calculated Risk for the Two Systems under Examination

| | S | T | R | I | E |
|---|---|---|---|---|---|
| **Non-secure system** | fairly high | fairly high | very low | fairly high | fairly high |
| **Secure system** | very low | absolutely low | absolutely low | very low | very low |
| **Difference in number of linguistic levels** | 4 | 5 | 1 | 4 | 4 |

tion of risk. In particular, we have studied the changes in the calculated risk when employing security patterns in two different sequences. For the sake of brevity, we will present the results for Information Disclosure attacks.

The first sequence consists of gradually employing the security patterns, as shown in Table 10.

The diagram shown in Fig. 5 depicts graphically the evolution of the calculated risk for the first sequence. The risk remains at the level "fairly high" during steps 1-23 and drops to "low" at step 24 when the last Intercepting Validator is employed. The drastic reduction of risk after the employment of the last Intercepting Validator happens, because it eliminates the last source of attacks that has high values for likelihood, exposure, and consequences (other sources for attacks remain, such as attacks that exploit servlet member variables race conditions and dictionary attacks that succeed in compromising the two levels of

TABLE 10
First Sequence of Employed Security Patterns

| | |
|---|---|
| **1** | Secure Proxy |
| **2** | Secure Pipe |
| **3-13** | Secure Logger |
| **14-24** | Intercepting Validator |

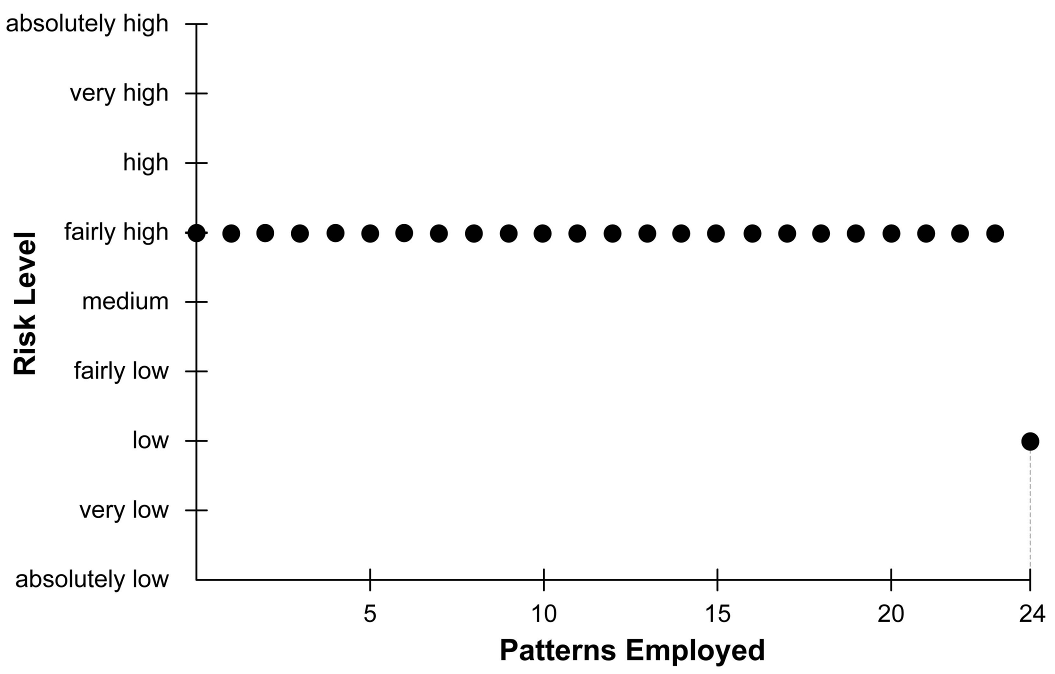

Fig. 5. Evolution of the calculated risk for the first sequence (Information Disclosure attacks).

protection of Secure Proxy, but these attacks correspond to lower values for likelihood, exposure, and consequences). Since the output of the OR gate leading to the top event of the fault tree depends on the highest input (that is, corresponds to a max operation), we have a relatively high risk for the resulting system up to the employment of the last Intercepting Validator pattern. The same behavior would be observed in case any other source of attack with high values for likelihood, exposure, and consequences remained until the last steps.

The second sequence consists of gradually introducing the security patterns, as shown in Table 11. In this sequence, the Intercepting Validator patterns are employed earlier. This evolution of the calculated risk is depicted graphically in Fig. 6. The risk level is "fairly high" during steps 1-12 and drops to "low" at step 13, where the Secure Pipe pattern is employed (after the employment of Intercepting Validator in all required places), eliminating all major sources for attacks that correspond to high values for likelihood, exposure, and consequences.

Based on Fig. 6, we can conclude that when we employ Intercepting Validator and Secure Pipe at an early stage, the calculated risk declines earlier compared to when we employ them at the last steps of our experiments. This happens, because the Intercepting Validator and the Secure Pipe patterns are important when regarding Information Disclosure attacks.

TABLE 11
Second Sequence of Employed Security Patterns

| | |
|---|---|
| **1** | Secure Proxy |
| **2-12** | Intercepting Validator |
| **13** | Secure Pipe |
| **14-24** | Secure Logger |

## 6 Introduction and Evaluation of the "Secure GET Parameters" Pattern

### 6.1 Pattern Description

The analysis of attacks performed in the first experiment shows that even when existing security patterns are used properly, unencrypted parameters might be used in a GET request. This means that even when an SSL connection is used throughout the whole application, parameters that are sent unencrypted via a GET request could be eavesdropped if someone obtains access to the client's browser history of visited URLs. Taking this into account, we propose a new security pattern, named "Secure GET Parameters," that provides resistance to this attack. The class diagram of the proposed security pattern is shown in Fig. 7.

In an actual application, the Client can be any servlet that sends parameters through a GET request to any other servlet that can play the role of a Target. The corresponding sequence diagram is shown in Fig. 8. The *Client* that wants to send encrypted parameters through a GET request to the *Target* sets an *Encryptor* object for self use and a *Decryptor* object for use by the *Target* through the use of appropriate methods of the *SecureBaseAction* object. Then, the *Client* invokes the method `encryptParameters()` of *SecureBaseAction*, which is delegated to `encryptParameters()` of the *Encryptor* object. A collection of the encrypted parameters is returned to the *SecureBaseAction* object and, consequently, to the *Client*. The *Client* then sends these encrypted parameters through the GET request to the *Target*. Consequently, in the browser history of visited URLs, these parameters will appear encrypted, eliminating this source of possible attacks. The *Target* should then decrypt these parameters to get their original values. Thus, the *Target* invokes method `decryptParameters()` of *SecureBaseAction*, which is delegated to `decryptParameters` of the *Decryptor* object. The *Decryptor* object returns a collection of the decrypted

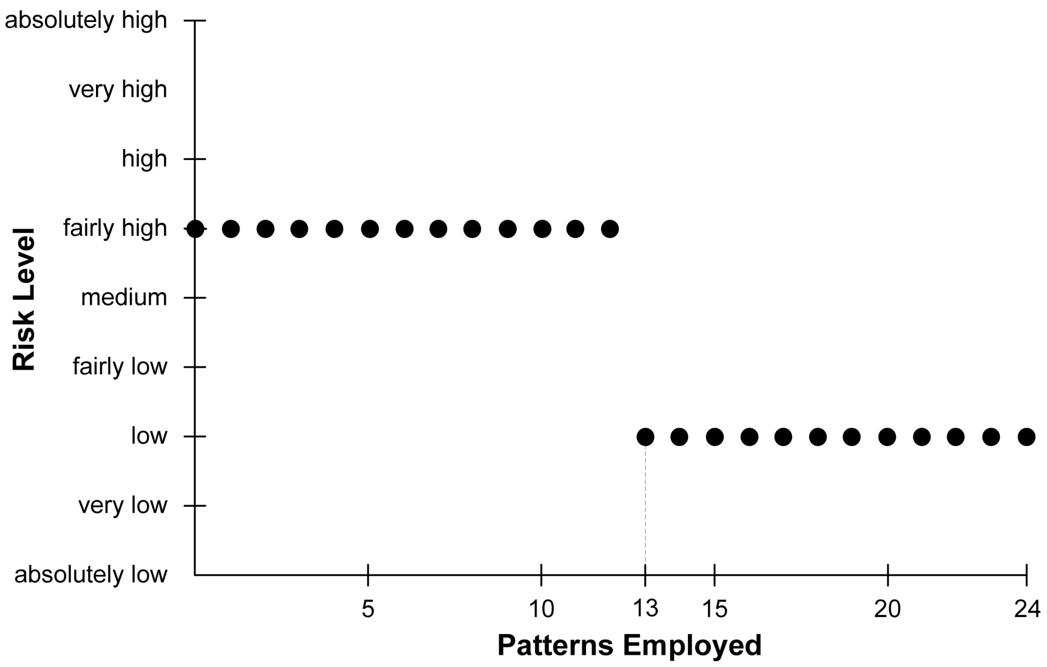

Fig. 6. Evolution of the calculated risk for the second sequence (Information Disclosure attacks).

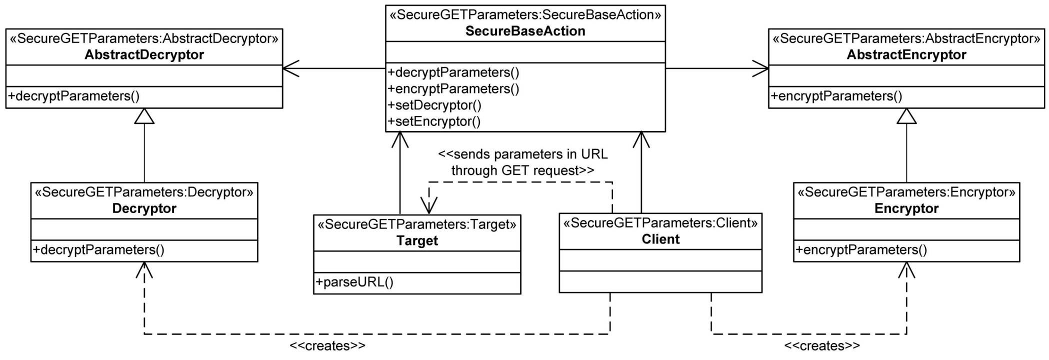
«SecureGETParameters:SecureBaseAction»
SecureBaseAction
+decryptParameters()
+encryptParameters()
+setDecryptor()
+setEncryptor()
«SecureGETParameters:AbstractDecryptor»
AbstractDecryptor
+decryptParameters()
«SecureGETParameters:AbstractEncryptor»
AbstractEncryptor
+encryptParameters()
<<sends parameters in URL through GET request>>
«SecureGETParameters:Decryptor»
Decryptor
+decryptParameters()
«SecureGETParameters:Target»
Target
+parseURL()
«SecureGETParameters:Client»
Client
«SecureGETParameters:Encryptor»
Encryptor
+encryptParameters()
<<creates>>
<<creates>>

Fig. 7. Class diagram of the Secure GET Parameters security pattern.

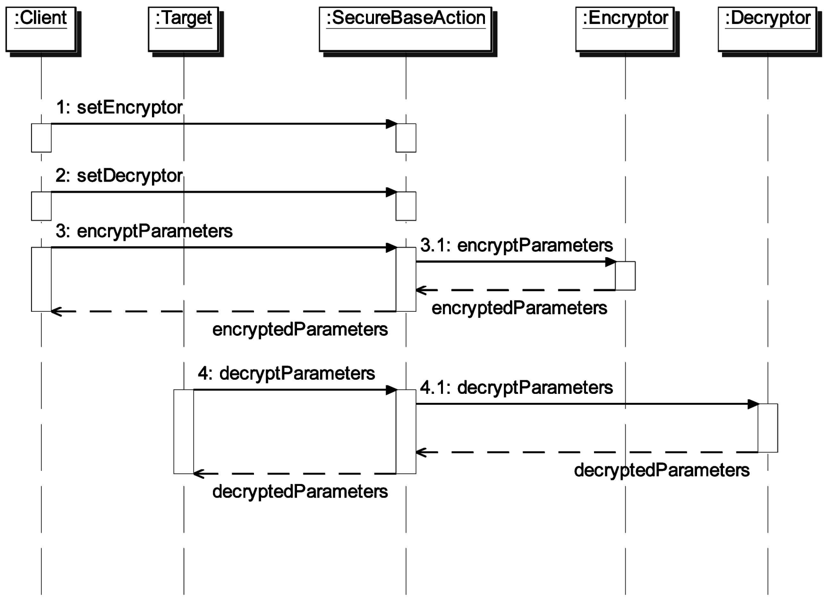

Fig. 8. Sequence diagram of the Secure GET Parameters security pattern.

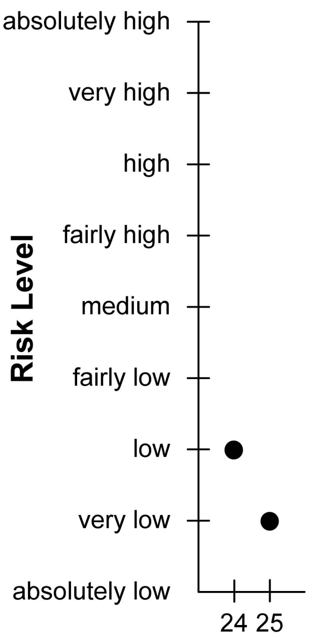

Fig. 9. Evolution of the calculated risk after the introduction of the "Secure GET Parameters" pattern (Information Disclosure attacks).

parameters to the *SecureBaseAction* object, which is consequently returned to the *Target*.

In the above pattern, the *Client* is able to choose any encrypting and decrypting algorithm among available implementations, as represented by the concrete classes in the design. The abstractions for the *Encryptor* and the *Decryptor* allow for the use of different implementations of encrypting and decrypting algorithms, without affecting both the *Target* and the *Client*.

### 6.2 Experimental Evaluation

To evaluate the impact of the proposed "Secure GET Parameters" pattern on risk mitigation, we have introduced this pattern as a last step in the two sequences of patterns employed to improve the nonsecure system, described in Section 5.3. The introduction of the proposed pattern in both cases (that is, the sequence in Tables 10 and 11, respectively) reduced the risk concerning Information Disclosure attacks from "low" to "very low," which corresponds to one linguistic level (see Fig. 9). This further reduction occurs, because the introduction of the "Secure GET Parameters" pattern eliminates the possibility of such an eavesdropping attack, which, according to Tables 4, 5, and 6 for Information Disclosure, corresponds to values "low," "very high," and "very high" for likelihood, exposure, and consequences, respectively. These values are significantly higher than those corresponding to the remaining sources of attacks that cannot be eliminated (attacks that exploit servlet member variable race conditions and dictionary attacks that succeed in compromising the two levels of protection of Secure Proxy).

## 7 Conclusions and Future Work

In this paper, we have proposed a methodology for quantifying the security level of a software system based on the implemented/missing security patterns. Moreover, the estimation can be performed already at the design phase. Thus, security problems can be detected at an early stage, which reduces the cost compared to the introduction of security during implementation. The comparison of two e-commerce systems having the same functionality, one without and one with security patterns, has shown that the nonsecure application has a high risk of being affected by each category of STRIDE attacks, whereas the secure application has a significantly lower risk.

An interesting extension to this work would be the automatic introduction of missing security patterns either at the design phase of a system being developed or in already-implemented software systems.

## Acknowledgments

The authors would like to thank the Web Application Security Mailing List of SecurityFocus and the comp.lang.java.security Mailing List for letting them organize a contest, Benjamin Livshits of Stanford University, the winner of the contest, and Watchfire Corp. for providing an evaluation license for AppScan, and the anonymous reviewers for their constructive comments.

## References

[1] E. Amoroso, *Fundamentals of Computer Security Technology.* Prentice Hall, 1994.
[2] C. Anley, "Advanced SQL Injection in SQL Server Applications," white paper, NGSSoftware, 2002.
[3] C.A. Berry, J. Carnell, M.B. Juric, M.M. Kunnumpurath, N. Nashi, and S. Romanosky, *J2EE Design Patterns Applied.* Wrox Press, 2002.
[4] B. Blakley, C. Heath, and Members of the Open Group Security Forum, *Security Design Patterns: Open Group Technical Guide,* 2004.
[5] *Borland Together Architect Home Page,* http://www.borland.com/together, 2007.
[6] A. Braga, C. Rubira, and R. Dahab, "Tropyc: A Pattern Language for Cryptographic Software," *Proc. Fifth Conf. Pattern Languages of Programming (PLoP),* 1998.
[7] P.J. Brooke and R.F. Paige, "Fault Trees for Security System Design and Analysis," *Computers and Security,* vol. 22, no. 3, pp. 256-264, Apr. 2003.
[8] K.-Y. Cai, *Introduction to Fuzzy Reliability.* Kluwer Academic Publishers, 1996.
[9] K.-Y. Cai, "System Failure Engineering and Fuzzy Methodology: An Introductory Overview," *Fuzzy Sets and Systems,* vol. 83, no. 2, pp. 113-133, Oct. 1996.
[10] K.-Y. Cai, "Software Reliability Experimentation and Control," *J. Computer Science and Technology,* vol. 21, no. 5, pp. 697-707, Sept. 2006.
[11] *Cross Site Scripting (XSS) Questions and Answers,* Cgisecurity.com, http://www.cgisecurity.com/articles/xss-faq.shtml, 2007.
[12] S.-J. Chen and S.-M. Chen, "Fuzzy Risk Analysis Based on Similarity Measures of Generalized Fuzzy Numbers," *IEEE Trans. Fuzzy Sets and Systems,* vol. 11, no. 1, pp. 45-56, Feb. 2003.
[13] E. Fernandez, *Metadata and Authorization Patterns,* http://www.cse.fau.edu/~ed/MetadataPatterns.pdf, 2007.
[14] M. Fowler, *UML Distilled: A Brief Guide to the Standard Modeling Language.* Addison Wesley, 2003.
[15] S. Friedl, *SQL Injection Attacks by Example,* http://www.unixwiz.net/techtips/sql-injection.html, 2007.
[16] *Fuzzy Risk Analysis Framework,* http://java.uom.gr/~halkidis/fuzzyrisk, 2007.
[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.
[18] K. Goseva-Popstojanova and K.S. Trivedi, "Architecture-Based Approach to Reliability Assessment of Software Systems," *Performance Evaluation,* vol. 45, nos. 2-3, pp. 179-204, July 2001.

[19] S.T. Halkidis, A. Chatzigeorgiou, and G. Stephanides, "A Qualitative Evaluation of Security Patterns," *Proc. Sixth Int'l Conf. Information and Comm. Security (ICICS),* 2004.
[20] G. Hoglund and G. McGraw, *Exploiting Software: How to Break Code.* Addison Wesley, 2004.
[21] M. Howard and D. LeBlanc, *Writing Secure Code.* Microsoft Press, 2002.
[22] D. Hu, "Preventing Cross-Site Scripting Vulnerability," white paper, SANS Inst., 2004.
[23] *JBoss Home Page,* http://www.jboss.com, 2007.
[24] J. Jürjens, *Secure Systems Development with UML.* Springer-Verlag, 2005.
[25] D. Kienzle and M. Elder, "Security Patterns for Web Application Development," technical report, Univ. of Virginia, 2002.
[26] A. Klein, "Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks and Related Topics," white paper, Sanctum, 2004.
[27] F. Lee Brown, J. Di Vietri, G. Diaz de Villegas, and E. Fernandez, "The Authenticator Pattern," *Proc. Sixth Conf. Pattern Languages of Programming (PLoP),* 1999.
[28] H.W. Lewis III, *The Foundations of Fuzzy Control.* Plenum Press, 1997.
[29] B. Littlewood, S. Brocklehurst, N. Fenton, P. Mellor, S. Page, D. Wright, J. Dobson, J. McDermid, and D. Gollman, "Towards Operational Measures of Computer Security," *J. Computer Security,* vol. 2, no. 3, pp. 211-229, 1993.
[30] B. Livshits and M.S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," *Proc. 14th Usenix Security Symp.,* Aug. 2005.
[31] B. Livshits and M.S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," technical report, Stanford Univ., 2005.
[32] B.B. Madan, K. Goseva-Popstojanova, K. Vaidyanathan, and K.S. Trivedi, "A Method for Modeling and Quantifying the Security Attributes of Intrusion Tolerant Systems," *Performance Evaluation,* vol. 56, nos. 1-4, pp. 167-186, Mar. 2004.
[33] Q. Mahmoud, "Security Policy: A Design Pattern for Mobile Java Code," *Proc. Seventh Conf. Pattern Languages of Programming (PLoP),* 2000.
[34] G. McGraw, *Software Security: Building Security In.* Addison Wesley, 2006.
[35] B. Möller, M. Beer, and M. Liebscher, "Fuzzy Analysis as Alternative to Stochastic Methods: Theoretical Aspects," *Proc. Fourth German LS-DYNA Forum '05,* pp. D-I-29-D-I-43, 2005.
[36] H. Mouratidis, P. Giorgini, and M. Schumacher, "Security Patterns for Agent Systems," *Proc. Eighth European Conf. Pattern Languages of Programs (EuroPLoP),* 2003.
[37] *MySQL Home Page,* http://www.mysql.com, 2007.
[38] D.M. Nicol, W.H. Sanders, and K.S. Trivedi, "Model-Based Evaluation: From Dependability to Security," *IEEE Trans. Dependable and Secure Computing,* vol. 1, no. 1, pp. 48-65, Jan.-Mar. 2004.
[39] L.L. Pullum, *Software Fault Tolerance Techniques and Implementation.* Artech House, 2001.
[40] E. Roman, R.P. Sriganesh, and G. Brose, *Mastering Enterprise JavaBeans.* Wiley Publishing, 2005.
[41] S. Romanosky, "Enterprise Security Patterns," *Information Systems Security Assoc. J.,* Mar. 2003.
[42] B. Ross, C. Jackson, N. Miyake, D. Boneh, and J.C. Mitchell, "Stronger Password Authentication Using Browser Extensions," *Proc. 14th Usenix Security Symp.,* 2005.
[43] J. Scambray and M. Shema, *Hacking Exposed Web Applications.* McGraw-Hill, 2002.
[44] D.I.K. Sjøberg, J.E. Hannay, O. Hansen, V. By Kampenes, A. Karahasanović, N.-K. Liborg, and A.C. Rekdal, "A Survey of Controlled Experiments in Software Engineering," *IEEE Trans. Software Eng.,* vol. 31, no. 9, pp. 733-753, Sept. 2005.
[45] K. Spett, "Cross-Site Scripting: Are Your Web Applications Vulnerable?" white paper, SPI Laboratories, 2005.
[46] K. Spett, "SQL Injection: Are Your Web Applications Vulnerable?" white paper, SPI Laboratories, 2005.
[47] D. Spinnelis, *Code Quality: The Open Source Perspective.* Addison Wesley, 2006.
[48] C. Steel, R. Nagappan, and R. Lai, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management.* Prentice Hall, 2006.
[49] J. Viega and G. McGraw, *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison Wesley, 2002.
[50] *Watchfire Home Page,* http://www.watchfire.com, 2007.
[51] M. Weiss, "Patterns for Web Applications," *Proc. 10th Conf. Pattern Languages of Programming (PLoP),* 2003.
[52] T. Wu, "A Real-World Analysis of Kerberos Password Security," *Proc. Network and Distributed System Symp. (NDSS),* 1999.
[53] *XML Metadata Interchange,* http://www.omg.org/technology/documents/formal/xmi.htm, 2007.
[54] J. Yoder and J. Barcalow, "Architectural Patterns for Enabling Application Security," *Proc. Fourth Conf. Pattern Languages of Programming (PLoP),* 1997.
[55] H.-J. Zimmerman, *Fuzzy Set Theory and Its Applications.* Kluwer Academic Publishers, 1996.

**Spyros T. Halkidis** received the BS and MS degrees in computer science from the University of Crete, Crete, Greece, in 1996 and 1998, respectively, the MBA degree from the University of Macedonia, Thessaloniki, Greece, in 2000. He is currently working toward the PhD degree in the Computational Systems and Software Engineering Laboratory, Department of Applied Informatics, University of Macedonia. His research interests include software engineering, secure software, and security patterns.

**Nikolaos Tsantalis** received the BS and MS degrees in applied informatics from the University of Macedonia in 2004 and 2006, respectively. He is currently working toward the PhD degree in the Computational Systems and Software Engineering Laboratory, Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. His research interests include design patterns, refactorings, and object-oriented quality metrics. He is a student member of the IEEE.

**Alexander Chatzigeorgiou** received the diploma in electrical engineering and the PhD degree in computer science from the Aristotle University of Thessaloniki, Thessaloniki, Greece, in 1996 and 2000, respectively. From 1997 to 1999, he was with Intracom, Greece, as a telecommunications software designer. He is currently an assistant professor of software engineering in the Computational Systems and Software Engineering Laboratory, Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece. His research interests include software metrics, object-oriented design, and software maintenance. He is a member of the IEEE and the IEEE Computer Society.

**George Stephanides** received the PhD degree in applied mathematics from the University of Macedonia, Thessaloniki, Greece. He is currently an associate professor in the Computational Systems and Software Engineering Laboratory, Department of Applied Informatics, University of Macedonia. His current research and development activities include the applications of mathematical programming, security and cryptography, and application-specific software. He is a member of the IEEE and the IEEE Computer Society.