Moving from Requirements to Design Confronting Security Issues: A Case Study

Spyros T. Halkidis, Alexander Chatzigeorgiou, and George Stephanides

Computational Systems and Software Engineering Laboratory
Department of Applied Informatics
University of Macedonia
Egnatia 156, Thessaloniki 54006, Greece
halkidis@java.uom.gr, {achat,steph}@uom.gr

Abstract. Since the emergence of software security as a research area, it has been evident that security should be incorporated as early as possible in the software lifecycle. The advantage is that large gains can be achieved in terms of cost and effort compared to the introduction of security as an afterthought. The earliest possible phase to consider possible attacks is during requirements specification. A widely accepted approach to consider security in the requirements is the employment of misuse cases. In this paper we examine a case study to automatically generate a class diagram, based on the use and misuse cases present in the requirements. Particularly, we extend a natural language processing approach to move beyond a general domain model and produce a detailed class diagram. Moreover, security patterns are introduced in appropriate places of the design to confront the documented attacks and protect the threatened resources. Additionally, we perform an experimental study to investigate the tradeoff between the additional effort to mitigate the attacks and the security risk of the resulting system. Finally, the optimization problem of finding the smallest system regarding additional effort given a maximum acceptable risk is established and an appropriate algorithm to solve it is proposed.

Keywords: Software Security, Requirements Specification, Misuse Cases, Security Patterns, Risk Analysis.

1 Introduction

The consideration of software security techniques has been inevitable during the last years since it has been discovered that most attacks to all kinds of organizations exploit software vulnerabilities [19,50,36,18,48]. Additionally, research in this area has shown that the earlier we introduce security in the software lifecycle, the better [50, 36]. Therefore, the introduction of security already at the requirements phase is desirable.

When considering a software engineering methodology such as the Rational Unified Process [25] or a similar one such as the methodology proposed by Larman [28], one way to document security requirements in UML is the description of possible attacks through misuse cases [1, 45, 44]. Misuse cases are a way to document negative scenarios for the system under consideration [1] and report the steps required to

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part II, LNCS 5871, pp. 798–814, 2009.

perform specific attacks to systems. Usually, misuse cases can be specified by the system analyst with the help of a security expert. The design and analysis of secure software architectures based on misuse cases has been discussed by Pauli and Xu [40]. However, the entire process is based on the human analysis of misuse cases and the candidate architecture resulting from it is much more abstract than a detailed class diagram.

In this work we propose a method to automatically derive a class diagram based on use cases [28] and misuse cases present in the requirements. To achieve this we extend a natural language processing technique [12] in order to produce a class diagram corresponding to the text present in use cases.

Furthermore, we introduce security patterns [49, 4] to protect the system under consideration from the attacks described in the misuse cases.

Additionally, we examine the decrease of risk and accordingly the increase of effort in the system resulting from the consideration of misuse cases and their mitigation. This is achieved through analyzing the change in these variables when gradually including misuse cases in the requirements. For the computation of risk we use an earlier work where a fuzzy risk analysis technique is proposed. For the computation of effort, an object oriented function points metric [6] is used.

Moreover, we define the optimization problem of finding the minimum system (in terms of effort) with risk not exceeding a maximum acceptable value. Finally, we propose an algorithm that solves this problem and examine the resulting systems for different maximum acceptable risk values.

Additionally this is the case where the whole automated process seems interesting, since it is difficult for a software engineer to inspect the design of a large system without the use of automated tools.

To demonstrate our approach we have created a case study of an e-commerce system. Its requirements documented as use/misuse cases can be found below:

UC1-1. User enters login name and password to the System.
UC1-2. User logs in to the System.
UC2-1. User views the product catalog.
UC2-2. User selects the product from the product catalog.
UC2-3. System shows product details to the User.
UC3-1. User selects products from the product catalog and specifies product quantities.
UC3-2. The System adds the selected products to shopping line items.
UC3-3. The System adds the shopping line items to the shopping cart.
UC3-4. User enters personal information.
UC3-4a. User personal information is invalid.
UC3-4a. System asks the User to reenter personal information.
UC3-5. System produces an order line item for the new order.
UC3-6. System shows order information to the User.
UC4-1. User enters product name, product details and product price to the System.
UC4-2. The System adds new product to the product catalog.
UC4-1a. Product price is not a number.
UC4-1a1. System asks the User to reenter product name, product details and product price.

UC5-1. User selects product from product catalog.
UC5-2. System removes the selected product from product catalog.
UC6-1. User enters login name.
UC6-2. System forms order SQLstatement for the given login name.
UC6-3. System executes order SQLstatement.
UC6-1a. User name does not exist.
UC6-1a1. System asks the User to reenter login name.
UC7-1. User logs out.
MUC1-1. The attacker obtains access to the System host computer.
MUC1-2. The attacker eavesdrops sent messages to the System.
MUC1-3. The attacker analyzes messages possibly containing sensitive data.
MUC1-4. The attacker collects sensitive data (e.g. a password) through the whole eavesdropping process.
MUC1-5. The attacker uses sensitive data.
MUC1-6. The attacker obtains illegal rights to the System.
MUC2-1. The attacker exploits poor or non-existing authentication mechanism.
MUC2-2. The attacker obtains illegal access to the System.
MUC3-1. The attacker obtains access to the logs.
MUC3-2. The attacker modifies the logs.
MUC4-1. The attacker identifies a database related input field.
MUC4-2. The attacker forces the System to create a malicious SQLStatement.
MUC4-3. The attacker modifies the database.
MUC5-1. The attacker identifies an input field shown in another form.
MUC5-2. The attacker enters malicious script code to the input field in order to steal information, usually from cookies.
MUC5-3. The User executes the malicious script code and the attacker receives sensitive information.
MUC6-1. The attacker identifies a redirection link with user defined input parameters.
MUC6-2. The attacker chooses input parameters properly.
MUC6-3. The attacker modifies the HTTP headers.

2 Method

The core method of our approach extracts a detailed class diagram from the use cases and the misuse cases documented in the requirements. Based on the misuse cases appropriate security patterns are introduced in the design. We consider only associations and no generalizations,aggregations and/or compositions, since these can not be extracted from the use case format we have adopted.

2.1 Description of Process Input

Our method uses as input use case text for the requirements of the system under design and misuse case text to describe possible attacks to the aforementioned system. For the use case description we use the standard proposed by Larman [28], while for the misuse case description we use the standard proposed by Sindre and Opdahl [45].

For each use/misuse case the primary actor and the system under design (SuD) should be designated. Each use case is composed of one ore more use case steps. Each use case step is a sentence in active voice (active voice is a usual requirement in natural language processing of use cases). A representative example of a use case is UC4 from the previous description.

Each use case step is described by a unique id resulting from the concatenation of the use case number and the use case step id (e. g. 4-1, 4-2). Use cases can also contain alternative flows that are executed when specified conditions are met. For example use case step 4-1a is a condition linked to use case step 4-1. If it is satisfied when executing step 4-1, the alternative steps are executed, which in this case is only step 4-1a1.

A representative example of a misuse case is MUC4 from the previous description.

A misuse case follows the same rules for use case step ids and describes the steps required to perform the specific attack.

Our methodology requires some additional information concerning verbs that belong to specific categories:

1. **"*Input*" verbs:** Verbs designating that some input is entered to the system (e.g. "*input*", "*enter*", "*reenter*").
2. **"*Entry point*" verbs:** Verbs designating an entry point to the system (e.g. "log in", "*log on*").
3.**"*Exit point*" verbs:** Verbs designating the end of system use (e.g. "*log out*", "*log off*").

2.2 Natural Language Processing of Use Cases

A use case step consists of one or more verb phrases. The case where there are more than one verb phrases is when the use case step contains an auxiliary verb (e. g. "*ask*", "*choose*", "*request*") [2]. For example the sentence "*System asks the User to reenter personal information*" consists of two verb phrases. One verb phrase where *"System"* is the subject, *"ask"* is the verb and *"User"* is the object and one verb phrase where *"User"* is the subject, *"reenter"* is the verb and *"personal information"* is the object. In this case the first verb phrase, that contains the auxiliary verb, is ignored in the subsequent processing since it does not contain any action.

For each verb phrase the following information is extracted:

1. The **subject**.
2. The **verb**: The verb can be a simple verb or a phrasal verb (e. g. "*log in*", "*log out*").
3. **Direct objects**: A set of simple or compound (e. g. product information) direct objects of the verb phrase.
4. A **possible indirect object**: A simple or compound object following one of the prepositions *"from"*, *"to"*, *"for"*.

For example in the verb phrase *"User selects the product from the product catalog"*, the subject is *"User"*, the verb is "*select*", there is one direct object, namely *"product*" and the indirect object is "*product catalog*".

In order to extract the above information we extended a tool that at a first stage produces the sentence subject, simple verbs, and simple direct objects from the sentence verb phrases [12]. This tool uses as input the parse trees produced by three natural

language processing tools [10,8,3]. Next, the best parse tree is selected using a metric that evaluates how much each tree matches predefined rules for use cases [12].

In the proposed approach the compound direct/indirect objects of a verb phrase are recognized by merging sequences of objects that are neighbor siblings in the syntax tree. For example in the sentence "*System shows order information to the User*" the compound direct object is "*order information*". Adjectives are discarded in the formation of compound direct objects.

Phrasal verbs are recognized as verbs followed by a particle (preposition directly following a verb) [2, 34]. For example in the sentence "*User logs in to the system*" the phrasal verb is "*log in*".

2.3 Construction of Initial Class Diagram

The information present in the use cases is sufficient to initially construct a UML class diagram which is essentially an enhanced domain model. A domain model illustrates important conceptual classes (classes that correspond to real-world concepts) and their relationships in a problem domain [28]. The initial class diagram contains the additional information of directed associations and methods compared to the corresponding domain model.

The heuristic rules used in order to construct the initial class diagram are applied to each verb phrase and are the following:

1. If **the indirect object does not exist or is a primary actor/SuD**:

A class is created for each simple/compound direct object with a method having the name of the verb/phrasal verb and no parameters.

2. If **the indirect object exists and is not a primary actor/SuD**:

A class is created for each simple/compound direct object.

A class is created for the indirect object. For each simple/compound direct object a method is added to the class corresponding to the indirect object, having the name of the verb/phrasal verb and the simple/compound direct object as parameter.

Finally, associations from the class corresponding to the indirect object to all the classes corresponding to the simple/compound direct objects are added.

The primary actors/SuD are not modeled as classes.

An illustrative example for case 1 is the use case step *"User enters product name, product details and product price to the System"*, since the indirect object "*System*" is the SuD. The sentence subject is "*User*" and corresponds to the primary actor. There are three compound direct objects, namely "*product name*", "*product detail*" and "*product price*", which are modeled as classes. The verb is "*enter*" and is modeled as a method of each direct object class. The corresponding class diagram is shown in Figure 1.

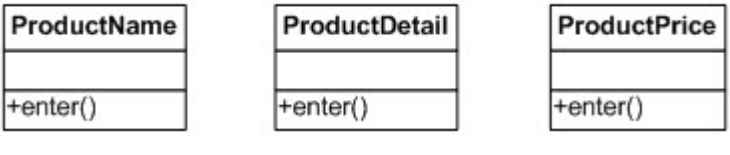

Fig. 1. The class diagram for use case step "User enters product name, product details and product price to the system"

An illustrative example for case 2 is the use case step "User selects the product from the product catalog", since the indirect object is "*product catalog*", which is not a primary actor/SuD of the use case and therefore is modeled as a class. The sentence subject is "*User*", which is the primary actor of the use case. There is one direct object, namely "*product"* which is modeled as a class. The verb is "*select*" and is modeled as a method of the indirect object class having a parameter corresponding to the direct object class. Additionally an association from the indirect object class to the direct object class is added. The corresponding class diagram is shown in Figure 2.

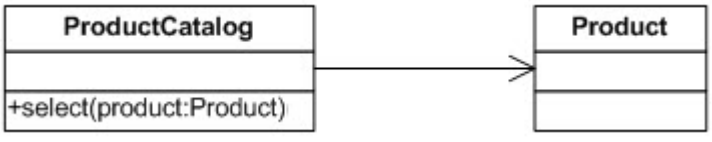

Fig. 2. The class diagram for use case step "User selects the product from the product catalog"

The enhanced domain model produced using the above rules for all use cases is shown in Figure 3.

The diagram is produced in XMI 1.4 for OMG format which can be read by various software engineering tools.

2.4 Addition of User Interface Classes

User interface (UI) classes are added according to heuristic rules, according to the category the verb of the phrase belongs to ("*entry point*", "*exit point*", other). Additionally appropriate stereotypes are added to these classes ("*ApplicationEntryPoint*", "*Input*" e.t.c.). Finally, appropriate associations from the UI classes to corresponding domain model classes are included.

We note here that we have chosen to use a simple stereotype addition to the classes instead of a more complex method like UMLSec [22]. We have adopted a similar technique to SecureUML [33] by means of the stereotype addition with an even smaller set of rules.

2.5 Inclusion of Security Patterns Based on Misuse Cases

2.5.1 Description of Employed Security Patterns

Since the suggestion of the first security patterns in the literature [52], various security patterns have been proposed. Patterns for Enterprise Applications [41], patterns for authentication and authorization [29, 13], patterns for web applications [23, 51], patterns for mobile Java code [32], patterns for cryptographic software [5] and patterns for agent systems [38] have been suggested. The first work trying to review all previous work on security patterns and establish some common terminology for this subject of research was [4].

Recently, a summary of security patterns has appeared in the literature [49]. 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.

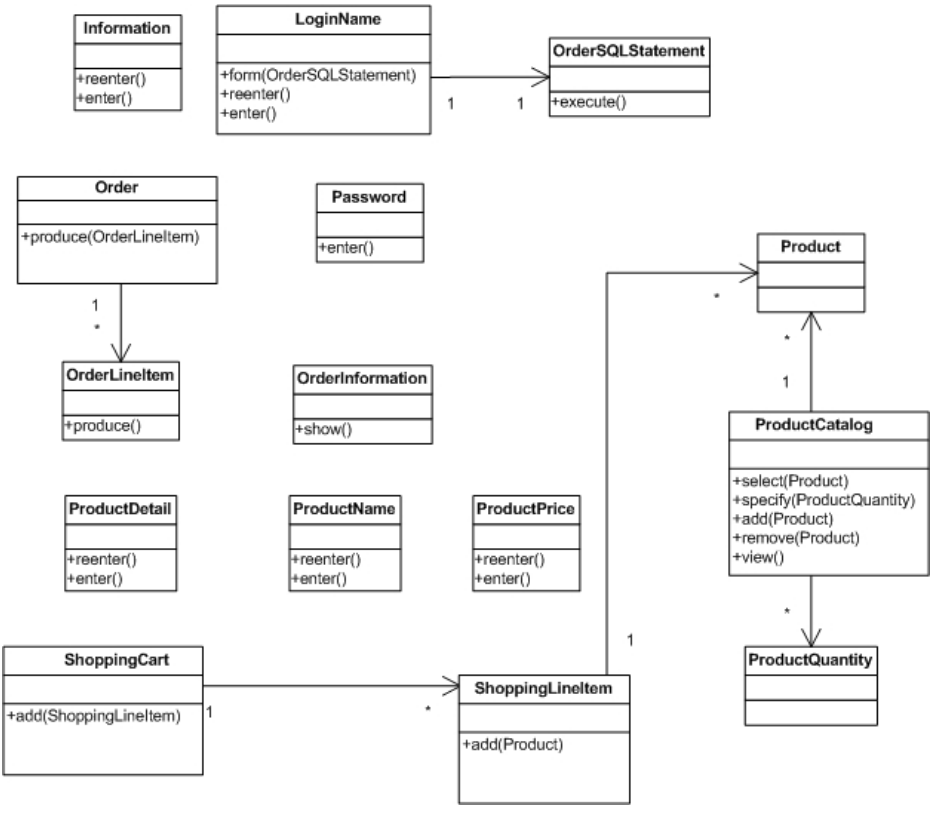

Fig. 3. The enhanced domain-model diagram for all use cases

The Secure Proxy pattern [4] is the only authentication pattern that uses two levels of security. Since it practices defense in depth [50] we have selected it for the authentication process i.e. the application entry points.

The Intercepting Validator pattern [49] offers a mechanism to check the validity of the data and allow access to the target only if the user input passes the security checks.

It therefore provides a technique to protect from *sql injection* [14, 47], *cross site scripting* [7, 46, 20] and *http response splitting* [24] attacks.

For the protection of the logs we have adopted the Secure Logger, Secure Log Store Strategy pattern [49].

Finally the pattern Secure Pipe offers an https connection in order that no eavesdropping attacks may occur.

This ensemble of patterns seems enough for our purposes since there exist patterns protecting from each attack previously described. We note here that we focus on the web and business tier and therefore use appropriate patterns [49] for these categories. These patterns are selected in an ad-hoc manner where the criteria are based on the protection from specific attacks. Additionally we do not deal with service oriented architectures and related security patterns. This seems to be beyond the scope of this paper.

2.5.2 Inclusion of Security Patterns

The design up to this point is complete in terms of functional requirements but until now no security considerations were taken into account. The attacks that the requirements engineer has made provision for are described in the misuse cases.

In order to identify the attacks documented in the misuse cases we have considered three different alternatives. The first possible approach would be to require the user to label the misuse case with the name of the corresponding attack. This approach would be too simplistic to adopt. The second possible approach would be to understand the theme of the sentences (discourse interpretation) present in the misuse cases based on natural language understanding techniques [2]. We have not adopted this approach since it requires the extraction of semantic information [2], which is beyond the scope of our work. The third approach is to recognize the attacks using a keyword matching technique. We have adopted an approach based on boolean expressions where the boolean variables take values based on the existence/non-existence of specific lemmas in the misuse case steps. This approach is adequate for our purpose since we assume that misuse cases are correctly documented and follow use case writing rules.

In order to explain the technique we used to identify the attacks from the misuse cases we will show the boolean expression for the Cross-Site Scripting attack [7, 46, 20]. The misuse case corresponding to the Cross-Site Scripting attack is MUC5 shown below:

MUC5-1. The attacker identifies an input field shown in another form.
MUC5-2. The attacker enters malicious script code to the input field in order to steal information, usually from cookies.
MUC5-3. The User executes the malicious script code and the attacker receives sensitive information.

The boolean expression we have used in order to identify the Cross-Site Scripting attack is:

(phrase="*malicious script*" or phrase="*harmful script*" or phrase="*crossSite script*") and (not (phrase="*header*")).

where the condition phrase=*word1 ... wordN* means that the lemmas *word1,...,wordN* coexist in the same use case step. In order to identify the Cross-Site Scripting attack we require that either the lemmas "*malicious*" and "*script*" or the lemmas "*harmful*" and "*script*" or the lemmas "*crossSite*" and "*script*" coexist in the same phrase and the lemma "*header*" does not exist in any phrase of this misuse case.

The *not* part of the condition is necessary in order not to misidentify an HTTP Response Splitting misuse case [24] as a Cross-Site Scripting misuse case. The boolean expressions used to recognize the attacks associated with the misuse cases are easily configurable.

Based on the attacks identified in the misuse cases and the class stereotypes present in the current class diagram, security patterns [49, 4] mitigating these attacks are added at appropriate places. For each attack specific class stereotypes are examined and corresponding security patterns are included. Table 1 shows the correspondence between attacks taken into account, class stereotypes existing in the diagram and security patterns.

When considering the "*Log Tampering*" and "*SQL Injection*" attacks a resulting part of the class diagram is shown in Figure 4 (The related patterns are added in various places of the whole class diagram).

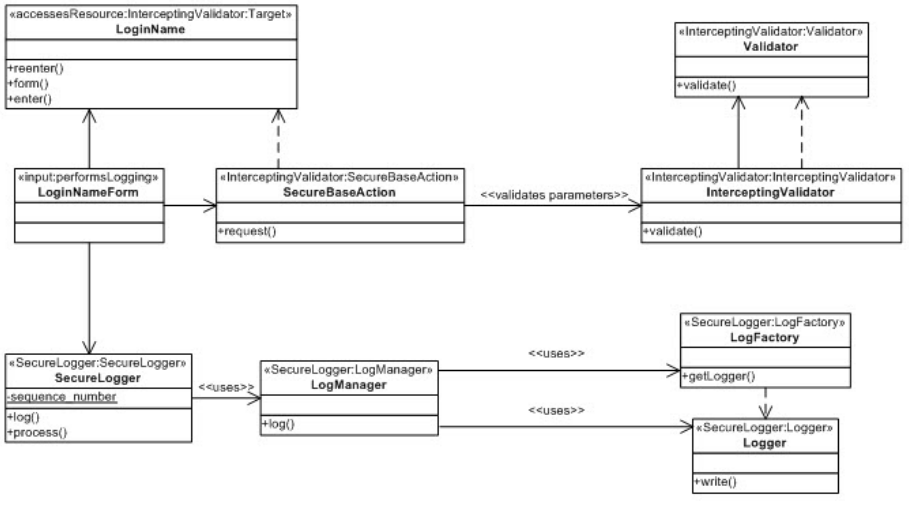

Fig. 4. Addition of appropriate security patterns based on the attacks identified from the misuse cases and the stereotypes of the class "LoginNameForm"

Table 1. Correspondence between documented attacks, class stereotypes and introduced security patterns

Attack	Stereotype	Security Pattern
Eavesdropping	"ApplicationEntryPoint	Secure Pipe
Exploitation of poor authentication	"ApplicationEntryPoint"	Secure Proxy
Log Tampering	"PerformsLogging"	Secure Logger
SQL Injection	"Input"	Intercepting Validator
Cross Site Scripting	"Input"	Intercepting Validator
HTTP Response Splitting	"Input"	Intercepting Validator

As we add security patterns to the system, risk is lowered but system development effort increases. Regarding risk we follow a fuzzy risk analysis approach from an earlier work examining risk for STRIDE [19] attacks [16]. A crisp weighting technique has been used in order to find total risk from risk related to each category. It is desirable to find an estimate of the trade-off between risk mitigation and effort for the system. There are two basic metrics in the literature for estimating effort at class diagram level, namely Class Point [11] and Object Oriented Function Points (OOFP) [6]. The first metric is more precise, but needs expert judgment to compute the Technical Complexity Factor which is part of its basic computation, contrary to OOFP. Therefore, we have chosen the OOFP metric, since it can be fully automated.

In order to investigate the aforementioned trade-off we have gradually included misuse cases in the requirements of our case study and computed the fuzzy risk and object oriented function points for each resulting system.

The diagram showing the trade-off between decrease in risk and increase in effort for the system is depicted in Figure 5.

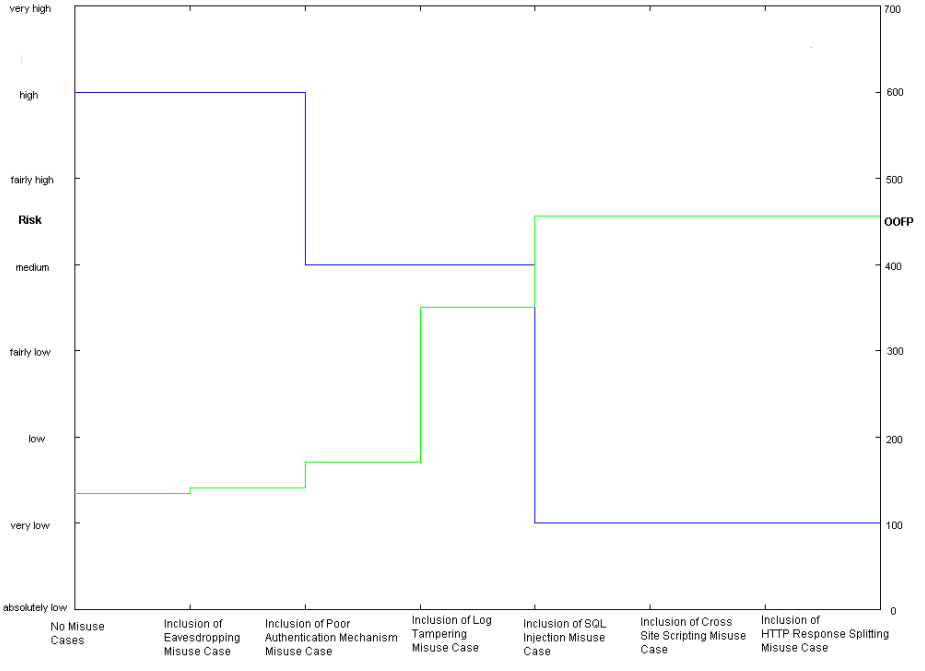

Fig. 5. Trade-off between decrease in risk and increase in effort for the system

From this diagram it becomes clear that the only place where the increase in effort is substantial is when the first misuse case related to the Intercepting Validator pattern (SQL Injection Misuse case) is included. This happens because different implementations of Intercepting Validators [49] have to be included in the diagram at different places, in order to validate data according to different rules. Additionally, if one misuse case related to the Intercepting Validator pattern is included in the requirements, inclusion of further misuse cases related to this pattern (Cross Site Scripting and HTTP Response Splitting Misuse Cases) causes no change in risk and OOFP.

3 An Optimization Problem Based on Risk and Effort

There are cases when some classes of the system access resources that do not contain crucial data (e.g. classes accessing cookies that do not contain important data). In these cases it is sometimes acceptable to employ a system that is not protected from attacks to low valued resources in order to reduce effort. Therefore, finding the minimal system given the maximum acceptable risk would be an interesting optimization problem.

This problem is similar to a 0-1 knapsack problem which is known to be NP-hard [35]; however, two different categories of algorithms can be used in order to solve knapsack problems. The first category contains algorithms that have a non-exponential average case and an exponential worst case and always find the optimal solution, while the second one includes algorithms that have a non-exponential worst case but find suboptimal solutions. To solve the optimization problem under examination we have used an algorithm similar to the greedy algorithm [35], which belongs to the second category.

The algorithm is described as follows:

S	current system
S_i	system after the introduction of security pattern i to the current system
$R(S)$	risk of system S
R_{max}	maximum acceptable risk
$OOFP_i$	additional object-oriented function points due to the introduction of security pattern I
$D(R(S_i),R(S_j))$	distance between risks of systems S_i and S_j
NonAddedPatterns	the set of security patterns that have not been added to the system
AllPatterns	The set of all security patterns that can be introduced to the system

```
S ← system without security patterns
NonAddedPatterns = AllPatterns
compute R(S)
while R(S)>Rmax and |NonAddedPatterns|>0
   compute d(R(Si),R(S)) ∀ i ∈ NonAddedPatterns
   compute OOFPi ∀ i ∈ NonAddedPatterns
   sort NonAddedPatterns in descending order according
to
      d(R(Si),R(S))/OOFPi
   S ← S with first element (security pattern instance)
       of NonAddedPatterns added
   Remove first element of NonAddedPatterns
   Calculate R(S)
end
```

The distance metric $d(R(S_i),R(S_j))$ between the risk of systems S_i and S_j used is given by the simple formula:

$$d(r_i, r_j) = 1 - s(r_i, r_j)$$

Where $s(r_i, r_j)$ is the similarity metric for fuzzy numbers described in [9].

The results of applying the optimization technique to our case study for all possible maximum acceptable risk values are shown in Table 2. Specifically we show the attained risk level, the corresponding number of object oriented function points and the security patterns to be included.

Table 2. Results of the optimization technique for different maximum acceptable risk values

Maximum acceptable risk	Attained risk for the solution of the optimization problem	Number of object oriented function points	Security patterns included
absolutely high very high high fairly high	fairly high	302	Attainable even with no use of security patterns
medium	medium	347	Secure Proxy Secure Pipe
fairly low	fairly low	487	Secure Proxy Secure Pipe Secure Logger 3 Intercepting Validators
low	very low	520	Secure Proxy Secure Pipe Secure Logger 4 Intercepting Validators
very low	very low	520	Secure Proxy Secure Pipe Secure Logger 4 Intercepting Validators
absolutely low	very low (absolutely low not achievable)	553	All Patterns (Secure Proxy Secure Pipe Secure Logger 5 Intercepting Validators)

4 Related Work

There has been substantial research in automatic transition of natural language to UML Design. In [31] a methodology to convert use case specifications to a class model is proposed according to the Rational Unified Process [25]. It is domain knowledge based, since it uses a glossary from the domain that the requirements belong to. First

use cases are generated from a natural language description by identifying candidate actors as nouns contained in the glossary and use cases as tasks performed by the actors. After the generation of use cases, candidate objects are identified as use case entities found in the glossary. A robustness diagram [42] is created as an intermediate step. When two objects, or an actor and one object exist in one statement an association between them is identified. Compositions and generalizations are induced using relations between use cases like inclusion of a use case by another and generalizations of use cases.

In several research papers automatic transition from textual descriptions to conceptual models, without examining use cases, is investigated [39, 21, 17]. The basic problems with these methodologies is that someone has to deal with the ambiguity of free text and that the model resulting from such input is over-specified to a larger extent (meaning that more classes are produced that a designer would normally not include) compared to when use cases are used as input. In [39] a semi-automatic approach to this problem is followed. The plain text that is the input to this methodology is subject to morphological analysis in order to recognize the part of speech corresponding to each word. From this analysis lexical items are recognized and assigned to model elements. Most frequently occurring nouns are assigned to classes, verbs are assigned to methods and adjectives are assigned to attributes. These model elements are assembled into a class model through user intervention. The inclusion of associations between classes is also left to the user. In [21] the textual descriptions are subject to natural language processing and a semantic network is produced, as an intermediate step. From the semantic network a domain model without associations is constructed. In this work phrasal verbs are also identified. In [17] an AI based approach is followed. After the natural language processing phase, a prolog system transforms parse trees to a predicate-argument structure. After this, the discourse of the requirements is interpreted and an ontology is constructed taking into account compound nouns. Attributes are identified from this ontology using a lexical database. Then, the domain model is built based on the extracted information. Examination of the results shows that there are possible methods that remain unidentified.

There has been also substantial work on security requirements engineering. Though, none of them deals with an automatic transition from use cases to design. In [26] a goal oriented approach is followed in order to provide an anti-model. The requirements are modeled as terminal goals under responsibility of an agent. The goals are formalized in temporal logic and the anti-goals are the attacker's goals. Thus attacker agents generate anti-requirements. Threats are derived through deductive inference from partial declarative goal/anti-goal models. A formal analysis can take place when and where needed so that evidence of security assurance can be provided. In [27] the related KAOS method is described. In [37] an ontology based approach called the SecureTROPOS technique is followed. At the first (lowest) level the main concepts are actors, goals, soft goals, tasks, resources and social dependencies. At the second level a set of organizational styles inspired by organization theory and strategic alliances is followed. At the last level social patterns focused on the social structure necessary to achieve a particular goal are presented. Formal TROPOS allows the specification in a first order linear time temporal logic.

In [30] a semi-automatic approach to translating use cases to sequence diagrams is examined also based on natural language processing techniques. The sentences have

to be in active voice and additionally if they are complex, specific rules are followed in order to simplify them. In this work the intervention of the user is required in some cases where the parser produces incorrect syntax trees.

The most related paper to the one presented here is [15]. In this aspect oriented programming approach security mechanisms are modeled as aspects. A case study for an authentication mechanism is given.

5 Conclusions and Future Work

In this work a complete method to move from requirements to class diagrams of secure systems is presented. To the best of our knowledge, this is the first attempt that confronts security issues documented in the requirements by employing security patterns in the design.

Additionally, the tradeoff between decrease of risk and increase in effort was studied, when gradually including misuse cases in the requirements. This study has shown that the increase in effort is substantial only for misuse cases corresponding to specific attacks.

Finally, an optimization problem regarding the minimum system achieving a desirable risk level was studied. The results show that a low level of risk is achievable without using all security pattern instances that can be possibly included.

All steps of the proposed method, as well as the techniques required for studying risk and effort related issues have been fully automated.

Future work includes an extension to a larger set of attacks/security patterns as well as considering service oriented architectures.

References

1. Alexander, I.: Misuse Cases: Use Cases with Hostile Intent. IEEE Software, 58–66 (January/February 2003)
2. Allen, J.: Natural Language Understanding. Addison Wesley, Reading (1994)
3. Bikel, D.,, M.: Design of a Multi-lingual Parallel-Processing Statistical Parser Engine. In: Proceedings of Human Language Technology Conference, HLT 2002 (2002), http://www.csi.upenn.edu/~dbikel/software.html#stat-parser
4. Blakley, B., Heath, C., Members of the Open Group Security Forum: Security Design Patterns. Open Group Technical Guide (2004)
5. Braga, A., Rubira, C.: Tropyc: A Pattern Language for Cryptographic Software. In: Proceedings of the 5th Conference on Pattern Languages of Programming, PLoP 1998 (1998)
6. Caldiera, G., Antoniol, G., Fiutem, R., Lokan, C.: A Definition and Experimental Evaluation of Function Points for Object-Oriented Systems. In: Proceedings of the Fifth International Symposium on Software Metrics-METRICS 1998, pp. 167–178 (1998)
7. Cgisecurity.com, Cross Site Scripting questions and answers, http://www.cgisecurity.com/articles/xss-faq.shtml
8. Charniak, E.: Statistical Techniques for Natural Language Parsing. AI Magazine 18(4), 33–44 (1997)
9. Chen, S.-J., Chen, S.-M.: Fuzzy Risk Analysis Based on Similarity Measures of Generalized Fuzzy Numbers. IEEE Transactions on Fuzzy Sets and Systems 11(1) (2003)

10. Collins, M.: A New Statistical Parser Based on Bigram Lexical Dependencies. In: Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics, pp. 184–191 (1996)
11. Costagliola, G., Ferruci, F., Tortora, G., Vitello, G.: Class Point: An Approach for the Size Estimation of Object Oriented Systems. IEEE Transactions on Software Engineering 31(1) (January 2005)
12. Dražan, J.: Natural Language Processing of Textual Use Cases. M.Sc. Thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague (2005)
13. Fernandez, E.: Metadata and authorization patterns (2000), http://www.cse.fau.edu/~ed/MetadataPatterns.pdf
14. Friedl, S.: SQL Injection Attacks by Example, http://www.unixwiz.net/techtips/sql-injection.html
15. Georg, G., Ray, I., Anastasakis, K., Bordbar, B., Toachoodee, M., Humb, S.H.: An Aspect Oriented Methodology for Desigining Secure Applications. Information and Software Technology 51, 846–864 (2009)
16. Halkidis, S.T., Tsantalis, N., Chatzigeorgiou, A., Stephanides, G.: Architectural Risk Analysis of Software Systems Based on Security Patterns. IEEE Transactions on Dependable and Secure Computing 5(3), 129–142 (2008)
17. Harmain, H.M., Gaizauskas, R.: CM-Builder: An Automated NL-based CASE Tool. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering, pp. 45–53 (2000)
18. Hoglund, G., McGraw, G.: Exploiting Software, How to Break Code. Addison Wesley, Reading (2004)
19. Howard, M., LeBlanc, D.: Writing Secure Code. Microsoft Press, Redmond (2002)
20. Hu, D.: Preventing Cross-Site Scripting Vulnerability. SANS Institute whitepaper (2004)
21. Ilieva, M.G., Ormanijeva, O.: Automatic Transition of Natural Language Software Requirements Specification into Formal Presentation. In: Montoyo, A., Muńoz, R., Métais, E. (eds.) NLDB 2005. LNCS, vol. 3513, pp. 392–397. Springer, Heidelberg (2005)
22. Jűrjens, J.: Secure Systems Development with UML. Springer, Heidelberg (2005)
23. Kienzle, D., Elder, M.: Security Patterns for Web Application Development. Univ. of Virginia Technical Report (2002)
24. Klein, A.: Divide and Conquer., HTTP Response Splitting, Web Cache Poisoning Attacks and Related Topics, Sanctum whitepaper (2004)
25. Kruchten, P.: The Rational Unified Process: An Introduction. Addison Wesley, Reading (2000)
26. van Lamsweerde, A.: Elaborating Security Requirements by Construction of Intentional Anti-Models. In: Proceedings of ICSE 2004, 26th International Conference on Software Engineering, Edinburgh, May 2004, pp. 148–157. ACM-IEEE (2004)
27. van Lamsweerde, A.: Engineering Requirements for System Reliability and Security, in Software System Reliability and Security. In: Broy, M., Grunbauer, J., Hoare, C.A.R. (eds.) NATO Security through Science Series - D: Information and Communication Security, vol. 9, pp. 196–238. IOS Press, Amsterdam (2007)
28. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice-Hall, Englewood Cliffs (2002)
29. Lee Brown, F., Di Vietri, J., Diaz de Villegas, G., Fernandez, E.: The Authenticator Pattern. In: Proceedings of the 6th Conference on Pattern Languages of Programming, PLoP 1999 (1999)

30. Li, L.: A Semi-Automatic Approach to Translating Use Cases to Sequence Diagrams. In: Proceedings of Technology of Object Oriented Languages and Systems, pp. 184–193 (1999)
31. Liu, D., Subramaniam, K., Eberlein, A., Far, B.H.: Natural Language Requirements Analysis and Class Model Generation Using UCDA. In: Orchard, B., Yang, C., Ali, M. (eds.) IEA/AIE 2004. LNCS (LNAI), vol. 3029, pp. 295–304. Springer, Heidelberg (2004)
32. Mahmoud, Q.: Security Policy: A Design Pattern for Mobile Java Code. In: Proceedings of the 7th Conference on Pattern Languages of Programming, PLoP 2000 (2000)
33. Lodderstedt, T., Basin, D., Doser, J.: SecureUML: A UML-Based Modeling Language for Model Driven Security. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 426–441. Springer, Heidelberg (2002)
34. Marcus, M., Kim, G., Marciniewicz, M.A., MacIntire, R., Bies, A., Ferguson, M., Katz, K., Schasberger, B.: The Penn Treebank: annotating predicate argument structure. In: Proceedings of the 1994 ARPA Human Language Technology Workshop (1994)
35. Martello, X., Toth, P.: Knapsack Problems: Algorithms and Computer Implementations. John Wiley and Sons, Chichester (1990)
36. McGraw, G.: Software Security, Building Security. Addison Wesley, Reading (2006)
37. Mouratidis, H., Giorgini, P., Manson, G.: An Ontology for Modelling Security: The Tropos Approach, in Knowledge-Based Intelligent Information and Engineering Systems. In: Palade, V., Howlett, R.J., Jain, L. (eds.) KES 2003. LNCS, vol. 2773. Springer, Heidelberg (2003)
38. Mouratidis, H., Giorgini, P., Schumacher, M.: Security Patterns for Agent Systems. In: Proceedings of the Eighth European Conference on Pattern Languages of Programs, EuroPLoP 2003 (2003)
39. Overmyer, S.P., Lavoie, B., Owen, R.: Conceptual Modeling through Linguistic Analysis Using LIDA. In: Proceedings of the 23rd International Conference on Software Engineering, pp. 401–410 (2001)
40. Pauli, J.J., Xu, D.: Misuse Case Based Design and Analysis of Secure Software Architecture. In: Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC 2005). IEEE, Los Alamitos (2005)
41. Romanosky, S.: Enterprise Security Patterns. Information Systems Security Association Journal (March 2003)
42. Rosenberg, D., Stephens, M.: Use Case Driven Modeling with UML: Theory and Practice. Apress (2007)
43. Sindre, G., Opdahl, A.L.: Capturing Security Requirements with Misuse Cases. In: Proceedings of the 14th annual Norwegian Informatics Conference, Norway (2001)
44. Sindre, G., Opdahl, A.L.: Eliciting Security Requirements with Misuse Cases. Requirements Engineering 10, 34–44 (2005)
45. Sindre, G., Opdahl, A.L.: Templates for Misuse Case Description. In: Proceedings of the 7th International Workshop on Requirements Engineering, Foundations for Software Quality, REFSQ 2001 (2001)
46. Spett, K.: Cross-Site Scripting, Are your web applications vulnerable? SPI Labs whitepaper
47. SPI Labs, SQL Injection, Are Your Web Applications Vulnerable? SPI Labs whitepaper
48. Spinellis, D.: Code Quality: The Open Source Perspective. Addison Wesley, Reading (2006)
49. Steel, C., Nagappan, R., Lai, R.: Core Security Patterns: Best Practices and Strategies for J2EE. In: Web Services and Identity Management. Prentice Hall, Englewood Cliffs (2006)

50. Viega, J., McGraw, G.: Building Secure Software, How to Avoid Security Problems the Right Way. Addison Wesley, Reading (2002)
51. Weiss, M.: Patterns for Web Applications. In: Proceedings of the 10th Conference on Pattern Languages of Programming, PLoP 2003 (2003)
52. Yoder, J.: Architectural Patterns for enabling application security. In: Proceedings of the 4th Conference on Pattern Languages of Programming, PLoP 1997 (1997)