



Assisting in secure application development and testing

Loukmen Regainia

► To cite this version:

Loukmen Regainia. Assisting in secure application development and testing. Cryptography and Security [cs.CR]. Université Clermont Auvergne [2017-2020], 2018. English. NNT : 2018CLFAC018 . tel-01959245

HAL Id: tel-01959245

<https://theses.hal.science/tel-01959245>

Submitted on 18 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE CLERMONT AUVERGNE

ECOLE DOCTORALE
SCIENCES POUR L'INGENIEUR DE CLERMONT-FERRAND

T h è s e

Présentée par

Loukmen REGAINIA

pour obtenir le grade de

D O C T E U R D ' U N I V E R S I T É

SPECIALITE : INFORMATIQUE

Assistance au développement et au test d'applications sécurisées

Assisting in secure application development and testing

Soutenue publiquement le 12/06/2018

devant le jury :

Pr. Ana Rosa CAVALLI
Pr. Jean-Michel BRUEL
Pr. Michel MISSON
Dr. Barbara KORDY
Dr. Cédric BOUHOURS
Pr. Sébastien SALVA

Rapporteur et examinateur
Rapporteur et examinateur
Examineur
Examineur
Encadreur
Directeur de thèse



Tous les mots du monde ne sauraient exprimer l'immense amour que je vous porte, ni la profonde gratitude que je vous témoigne pour tous les efforts et les sacrifices que vous n'avez jamais cessé de consentir pour mon instruction et mon bien-être. En ce jour mémorable, pour moi ainsi que pour vous, recevez ce travail en signe de ma vive reconnaissance et mon profond
estime.

à mes chères parents, à ma tendre épouse, à ma petite Jana et à mes deux frères ...

Acknowledgements

Research supported by the industrial chair on Digital Confidence

Recherche prise en charge par la chaire industrielle de confiance numérique

<http://confiance-numerique.clermontuniversite.fr>

Remerciements

Je voudrais tout d’abord remercier grandement mon directeur de thèse, Sébastien SALVA, pour toute son aide. Je suis ravi d’avoir travaillé en sa compagnie car outre son appui scientifique, il a toujours été là pour me soutenir et me conseiller au cours des années de thèse.

Je tiens à remercier mon encadrant Cédric BOUHOURS, pour sa proximité, sa disponibilité permanente et pour les nombreux encouragements qu’il m’a prodigués. Il a été pour moi un mentor dans le monde de l’enseignement et la recherche scientifique.

J’adresse tous mes remerciements à Madame Ana Rosa CAVALLI, Professeur à l’institut Mines-Telecom/TELECOM SudParis, ainsi qu’à Monsieur Jean-Michel BRUEL, Professeur à l’Institut de Recherche en Informatique de Toulouse, de l’honneur qu’ils m’ont fait en acceptant d’être rapporteurs de cette thèse.

J’exprime ma gratitude à Madame Barbara Kordy, Maître de conférences à l’institut National des Sciences Appliquées de Rennes et à Monsieur Michel MISSON Professeurs à l’Université Clermont-Auvergne, qui ont bien voulu être examinateurs.

Il m’est impossible d’oublier toute l’équipe du département informatique: Pascale, François, Chafik, Carine, Laurent, Guenal, Abdel, Nathalie, Isabelle, pour leur accueil et toutes les choses qu’ils m’ont appris au long de mes premières années d’expérience en enseignement.

Je remercie toutes les personnes avec qui j’ai partagé ces années de thèse: Xavier, Hamadoun, Thomas, Pape, David, William, Illiot, Anne, Samir,

Abstract

Assisting in the development and testing of secure applications

by Loukmen REGAINIA

Ensuring the security of an application through its life cycle is a tedious task. The choice, the implementation and the evaluation of security solutions is difficult and error prone. Security skills are not common in development teams. To overcome the lack of security skills, developers and designers are provided with a plethora of documents about security problems and solutions (i.e, vulnerabilities, attacks, security principles, security patterns, etc.). Abstract and informal, these documents are provided by different sources, and their number is constantly growing. Developers are drown in a sea of documentation, which inhibits their capacity to design, implement, and the evaluate the overall application security. This thesis tackles these issues and presents a set of approaches to help designers in the choice, the implementation and the evaluation of security solutions required to overcome security problems. The problems are materialized by weaknesses, vulnerabilities, attacks, etc. and security solutions are given by security patterns.

This thesis first introduces a method to guide designers implement security patterns and assess their effectiveness against vulnerabilities. Then, we present three methods associating security patterns, attacks, weaknesses, etc. in a knowledge base. This allows automated extraction of classifications and help designers quickly and accurately select security patterns required to cure a weakness or to overcome an attack. Based on this knowledge base, we detail a method to help designers in threat modeling and security test generation and execution. The method is evaluated and results show that the method enhances the comprehensibility and the accuracy of developers in the security solutions choice, threat modeling and in the writing of security test cases.

Keywords. Security patterns, weaknesses, attacks, principles, life cycle, model checking, security testing.

Résumé

Assistance au développement et au test d'applications sécurisées

par Loukmen REGAINIA

Garantir la sécurité d'une application tout au long de son cycle de vie est une tâche fastidieuse. Le choix, l'implémentation et l'évaluation des solutions de sécurité est difficile et sujette à des erreurs. De plus, les compétences en sécurité ne sont pas répandues dans toutes les équipes de développement. Pour pallier à ces difficultés, les développeurs ont à leurs dispositions une multitude de documents décrivant des problèmes de sécurité et des solutions requises (vulnérabilités, attaques, principes de sécurité, patrons sécurité, etc.). Ces documents, proposés par des sources hétérogènes, sont souvent abstraits et informels. De plus, leur nombre est en constante croissance. Les développeurs sont noyés dans une multitude de documents ce qui fait obstruction à leur capacité à choisir, implémenter et évaluer la sécurité d'une application. Cette thèse aborde ces problématiques et propose un ensemble de méthodes pour aider les développeurs à choisir, implémenter et évaluer les solutions de sécurité face à des problèmes de sécurité. Ces derniers sont matérialisés par des vulnérabilités, des attaques, etc., et les solutions fournies par des patrons de sécurité.

Cette thèse introduit en premier une méthode pour aider les développeurs dans l'instantiation de patrons de sécurité dans un modèle et l'estimation de leurs efficacités face aux vulnérabilités. Puis, elle présente trois méthodes associant patrons de sécurité, vulnérabilités et attaques au sein d'une base de connaissance. Cette dernière permet une extraction automatique de classifications de patrons et améliore la rapidité et la précision des développeurs dans le choix des patrons de sécurité. En utilisant la base de connaissance, nous présentons une méthode pour aider les développeurs dans la modélisation de menaces ainsi que la génération et l'exécution de cas de test de sécurité. L'évaluation de la méthode montre qu'elle améliore l'efficacité, la

compréhensibilité et la précision des développeurs dans le choix des patrons de sécurité et l'écriture des cas de test de sécurité.

Mots-clés. Patrons de sécurité, failles, attaques, principes de sécurité, cycle de vie, vérification des modèles, test de sécurité.

Table of contents

List of figures	xv
List of tables	xix
1 Introduction	1
1.1 General Context	1
1.2 Open issues	2
1.3 Contributions	4
1.4 Overview of the thesis	5
2 State of the art	7
2.1 Introduction	7
2.2 Security concepts and databases	8
2.2.1 Weaknesses	9
2.2.2 Vulnerabilities	9
2.2.3 Threats	10
2.2.4 Attacks	11
2.2.5 Security principles	11
2.2.6 Security by design	13
2.3 Patterns	13
2.3.1 Design patterns	14

Table of contents

2.3.2	Patterns selection	16
2.3.3	Patterns instantiation	18
2.4	Security patterns and Security patterns classifications	18
2.4.1	Security patterns	20
2.4.2	Security pattern instantiation	25
2.4.3	Inter-patterns relationships	28
2.4.4	Security patterns classifications	32
2.4.5	Classification schemes	33
2.4.6	Classification quality criteria	35
2.5	Security modeling	37
2.5.1	Vulnerability Cause Graphs (VCG)	37
2.5.2	Security Activity Graphs (SAG)	40
2.5.3	Attack Defense Trees ADTree	42
2.6	Chapter conclusions	45
3	Assisting Designers in Security Pattern Integration	47
3.1	Introduction	47
3.2	Context and Motivations	48
3.3	Prerequisites	50
3.4	Assisting designers in the security pattern instantiation	52
3.5	Case study	59
3.6	Discussion	69
3.7	Chapter Conclusions	72
4	Security analysis with patterns and pattern classifications	75
4.1	Introduction	75
4.2	Context and motivations	76

4.3	Data-stores architectures	78
4.3.1	Data sources	78
4.3.2	Data-stores Meta-models	79
4.4	Data integration	82
4.4.1	Integrating security patterns and security principles	84
4.4.2	Integrating security patterns and weaknesses	87
4.4.3	Attacks and security patterns integration (v_1)	90
4.4.4	Attacks and security patterns integration (v_2)	92
4.5	Classification extraction and SAG/ADTree generation	96
4.5.1	Weakness Security Patterns Classification (WSPC)	96
4.5.2	Security Activity Graph (SAGs) generation	98
4.5.3	Attacks and Security Patterns Classification (v_1)	100
4.5.4	ADTree generation (v_1)	101
4.5.5	Attacks and Security Patterns Classification (v_2)	106
4.5.6	ADTree generation (v_2)	107
4.6	Discussion	111
4.6.1	Comparaison of the ADTrees generated with the approaches (v_1, v_2)	114
4.6.2	Comparaison with other approaches	117
4.6.3	Limitations	118
4.7	Chapter conclusion	120
5	Assisting developers in the generation of security test cases	123
5.1	Introduction	123
5.2	Context and motivations	124
5.3	Data-store extension and data integration	127
5.3.1	Data-Store meta-model extension	127
5.3.2	Data acquisition and integration	130

Table of contents

5.4	An approach for guiding developers devise more secure applications	132
5.4.1	Threat modelling	133
5.4.2	Test generation and execution	137
5.5	Evaluation	143
5.5.1	Experiment Results	145
5.5.2	Result interpretation	149
5.6	Chapter conclusion	152
6	Conclusions and perspectives	155
6.1	Summary of the achievements	155
6.2	Towards a complete toolkit	157
6.2.1	Knowledge base completion	157
6.2.2	Documentation Generation	158
6.2.3	Runtime verification of security pattern properties in application traces	159
6.3	Security pattern landscape	161
6.3.1	Threat management with security patterns	161
6.3.2	Security reference architectures	162
6.4	Publications	164
	References	167

List of figures

2.1	The Number of Vulnerabilities per year	10
2.2	Authorization Enforcer Structure	22
2.3	Authorization Enforcer Behavior	22
2.4	Authentication Enforcer Structure	24
2.5	Authentication Enforcer Dynamics	24
2.6	Intercepting Validator Class Diagram	25
2.7	Intercepting Validator Sequence Diagram	26
2.8	Bad Instantiation of <i>Intercepting Validator</i>	27
2.9	Organization of patterns proposed by Hafiz et al.	29
2.10	The types of the nodes in a Vulnerability Cause Graphs [17]	39
2.11	CVE-2003-0161 Vulnerability Cause Graph [17]	40
2.12	The elements of a SAG [8]	41
2.13	strcat Security Activity Graph [17]	42
2.14	Nodes and operators of an ADTree [51]	43
2.15	Example of an ADTree [49]	43
2.16	ADTerms [50]	44
2.17	SANDTree Example	45
3.1	The context of the proposed approach	51
3.2	An approach to assist designers to devise more secure applications	53

List of figures

3.3	Moodle Quiz Engine Classe Diagram	59
3.4	Moodle Quiz Engine Sequence Diagram	60
3.5	Structural instantiation of intercepting validator	61
3.6	Behavioral instantiation of intercepting validator	62
3.7	<i>inputValidator</i> Statechart	63
3.8	The buchi automaton corresponding to the negation of v'_1	68
3.9	A part of v_4 counter example trail	69
3.10	LTL patterns	71
3.11	“Response” LTL pattern	72
4.1	Relations among security artifacts	79
4.2	WSPC : Weaknesses based Security Patterns Classification	82
4.3	AWSPC : Attacks and Weaknesses based Security Patterns Classification	83
4.4	ASSPC : Attack Steps based Security Patterns Classification	83
4.5	Integrating Security patterns and Security principles	84
4.6	Exemplary hierarchical organization of security principles	85
4.7	Integrating weaknesses and mitigations	88
4.8	Integrating weaknesses, mitigations and security patterns	89
4.9	Attacks extraction and hierarchical organisations	91
4.10	Weaknesses based attacks and security patterns integration	92
4.11	Integrating attacks and security patterns (Second approach)	93
4.12	Exemplary dendogram	95
4.13	CWE-285 Security patterns tree	100
4.14	Generic ADTree	103
4.15	ADTree of the attack CAPEC-39	105
4.16	Generic example of ADTree (second method)	109
4.17	ADTree of the Attack CAPEC-34	110

4.18	Distribution of fixed weaknesses per pattern	113
4.19	Distribution of fixed attacks per pattern (V_1)	114
4.20	Distribution of fixed attacks per pattern (V_2)	114
4.21	CAPEC-244 ADTree (AWSPC)	115
4.22	CAPEC-244 ADTree (ASSPC)	116
4.23	“CAPEC-66: SQL Injection” ADTree	117
5.1	Meta-model of Testing Data Store (TDS)	128
5.2	Test cases generation and execution steps	132
5.3	Initial ADTree example	133
5.4	CAPEC-66 ADTree	135
5.5	CAPEC-66 ADTree after patterns choice	138
5.6	A GWT test case Example	138
5.7	The procedure related to the When section of Figure 5.6	139
5.8	The procedure related to the Then section (Input Guard) of Figure 5.6	140
5.9	Test case Figure 5.6 result	141
5.10	Response rates for Q1 to Q3	146
5.11	Response rates for Q7	147
5.12	Response rates for Q5	147
5.13	Response rates for Q9	148
5.14	Pattern choice correctness (Q6)	148
5.15	Test case correctness (Q10)	149
6.1	Injection Threat Tree	162

List of tables

2.1	Design pattern sections	15
2.2	Authorization Enforcer [100]	21
2.3	Authentication Enforcer [100]	23
2.4	The satisfaction of Intercepting Validator String points in the example Figure 2.8	27
2.5	Security Models [52]	38
3.1	The symbols of the LTL	51
3.2	Coefficients values and interpretations	58
3.3	Behavioral properties of <i>Intercepting Validator</i>	64
3.4	CWE-89 properties	67
4.1	Extraction of the pattern classification for the weakness CWE-285	97
4.2	Data extraction for the attack CAPEC-39	102
4.3	Data extraction for the attack CAPEC-34	108
4.4	Attack techniques descriptions	111
5.1	Verdict Summary and solutions	143

Chapter 1

Introduction

1.1 General Context

Despite the indisputable improvements recently made in modeling, coding and testing, software engineering is still regarded as a complex field. One reason for this complexity is well-known: software engineering must not only address the functional aspects of an application, but also have to cover other aspects such as security. Indeed, providing secure models and code is recognized as an important factor of quality, but, devising them is a difficult task.

It is well admitted that software security is essential and has to be considered through all the software life cycle. Many developers, researchers and organizations have hence made security their hobby-horse and brought several improvements with the proposal of numerous digitized security bases and documents. These documents take security into consideration at different stages of the software life cycle and are presented with different viewpoints, abstraction levels or contexts.

A large set of papers and tools have been proposed to help in the integration of security in the software engineering steps [74, 30]. Among them, the pattern community proposed the notion of security patterns as reusable solutions to security issues in the modeling stage [114]. Specifically, a pattern is a generic and reusable solution presented with a structure, a behavior,

Introduction

or some intents that have to be applied in models to meet security properties or to prevent security problems.

In addition to security patterns, a plethora of software security documents, knowledge bases or papers are publicly available. These numerous digitalized resources are also presented with different viewpoints (attackers, defenders, etc.), formats (text, database, etc.), abstraction levels (security principles, attack steps, exploits, etc.) or contexts (system, network, etc.). Furthermore, these different documents provide interest at different stages of the software life cycle.

This plethora of (often complex) documents exposes engineers to the difficult choice of the most suitable security solutions in a given context. Indeed, they cannot be experts in any field and they clearly lack of guidance for conceiving and testing secure software or systems.

We develop a little bit more each one of these issues in the next section, then we present the main contributions of the thesis.

1.2 Open issues

Using security patterns in a whole threat management and secure design process is characterized with many difficulties related to the nature of security patterns. Security pattern documents are often expressed with at a high level of abstraction with texts and sometimes with UML diagrams to be reusable in different kinds of context. Integrating a security pattern in an application model requires an instantiation phase, i.e., adapting its structure and behavior to meet the application model specificities. This phase is tricky and error especially when used for the first time. Hence, designers should be guided when using security patterns and should have at their disposal tools allowing to assess whether security pattern are instantiated correctly in an application model.

Another issue is related to the growing number of security patterns. Since the introduction of the first set of security patterns in 1997 [112], we are counting around 180 security patterns

at the moment. Since the pattern documents are not structured in the same manner, the choice of the most appropriate pattern to solve a security problem is difficult with regard to a given context and somehow perilous for novice designers [4]. Indeed, a wrong choice may imply the use of useless solutions or the addition of new vulnerabilities in the design and code of the application. This is exacerbated with the fact that “*a security pattern is not an island*” [95]. In other words, a security pattern can be related to a set of other patterns. If these related patterns are not considered by the designer, the effectiveness of the security pattern against the aimed problem is strongly harmed and, in the worst cases, security inconsistencies can be resulted in the application model.

In order to help designers, many researchers emphasize the importance of classifying security patterns [6, 4, 5, 110, 102, 4, 104]. Alvi et al. outlined 24 security patterns classifications [5] and compared them against a set of desirable quality criteria (Comprehensibility, Usefulness, etc.). They observed that several classifications were built in reference to a unique classification attribute, which appears to be insufficient. After reviewing these classifications, we observed that these classifications are steadily manually conceived by interpreting different documents (e.g., weaknesses, attacks, and security patterns documents) to find abstract relationships. Justifying these classifications or updating them is often difficult. The relations among patterns are often not given, yet we noticed that some patterns are compatible together and that others are conflicting. As a consequence, a designer may be still confused about the pattern choice.

So, it appears fruitful to provide security patterns classifications with regard to well known security notions (e.g., Security principles, weaknesses, attacks) in order to enable a fast, comprehensive and accurate choice of the appropriate security patterns to cure a weakness or to overcome an attack. These classifications should be able to express the relationships among security patterns and to be comprehensible by both novice and expert designers. In addition, the steps of building the classification should be strictly presented to justify the classification itself and to allow its replicability, maintenance and reusability.

Introduction

Besides the choice and the instantiation of security patterns, developers have to be able to identify threats to which the application is faced, in addition to testing whether the application is vulnerable to attacks and whether security solutions are applied.

We observed that developers lack of guidance to write threat models or test scripts in order to detect security problems and to detect security patterns properties. The writing of detailed threat models requires a lot of expert knowledge and of documents. Actually, developers are neither guided the threat modeling phase nor provided with security solutions needed to prevent security problems. Hence, developers lack of recommendation on how to write and structure tests to make them at least reusable.

1.3 Contributions

With regard to the issues exposed in Section 1.2, we summarize here the main contributions covered by this manuscript.

1. we present an original approach to guide designers for checking whether a set of security patterns are correctly integrated into models of applications and whether vulnerabilities are yet exposed in models despite the use of these patterns. This approach relies upon the analysis of the structural and behavioral properties of security patterns and on formal methods to check if these properties hold in the application model completed with patterns. We also provide a metric computation to assess the integration quality of patterns. Afterwards, our approach checks whether the vulnerability properties, which should be cured by the use of patterns, are not detected in the application model;
2. we propose a set of semiautomatic methods of classifying security patterns and the classifications themselves, which expose relationships among software weaknesses, attacks, security principles and security patterns. They express which patterns cure a given weakness or overcome an attack with respect to security principles. These methods

are based on a set of data acquisition and integration steps, which anatomize patterns, attacks and weaknesses into set of more precise sub-properties that are associated through a hierarchical organization of security principles. These steps provide the detailed justifications of the resulting classifications and allow their upgrade;

3. we propose a method, which assists developers from the Threat modeling stage to the security test case writing and execution. The method firstly proposes a knowledge based Attack Defense Trees (ADTrees) generation. These trees show attacks, steps and defenses given under the form of security patterns. These trees are then used for the writing of concrete security test cases. This approach also yields Test verdicts showing whether attack scenarios of ADTrees hold and whether security patterns properties are detected in the application behavior. We evaluate our approach on 24 participants and show encouraging results on the use of data acquisition in software engineering;

1.4 Overview of the thesis

The remainder of this thesis is organized as follows :

1. **Chapter 2:** introduces the main notions on which the thesis is based. First, we give an insight on some security concepts (weaknesses, vulnerabilities, attacks, etc.) used in the manuscript. We present the notions of design and security patterns, some security patterns catalogs and classifications. We also investigate on the reasons that make the pattern choices and use difficult and error prone. Then, we present some formal and informal security modeling methods used in the thesis;
2. **Chapter 3:** introduces an approach for guiding designers in the instantiations of security patterns. We express security patterns and vulnerabilities with generic properties. Then the approach assesses the instantiation quality of a set of security patterns and their effectiveness against a set of vulnerabilities. The security pattern instantiation quality

and the vulnerability of the application are exhibited with a set of indicators, these are calculated along with the approach;

3. **Chapter 4:** presents three classifications of security patterns against security notions (e.g., weaknesses, attacks, security principles, etc.) and the methods to perform these classifications. The process building these classifications is also presented in the form of a set of semiautomated data acquisition steps. A set of data-stores is built, associating security patterns, weaknesses, attacks and security principles. In addition, these data-stores allow an automated extraction of security patterns classifications. The readability of these classifications is enhanced by presenting them graphically in the form of Attack Defense Trees (ADTrees) and Security Activity Graphs (SAG). The data-stores are used in order to generate these models;
4. **Chapter 5:** takes advantage of the data acquisition and integration performed in the previous chapter to devise an approach helping developers write concrete security test cases. The approach firstly assist them in the Threat modeling. The resulting threat model is used is then used to generate test cases. A tool is developed in order to help in their generation and execution and verdicts are given. These assess whether attacks can be performed on the application and whether security pattern properties can be observed in its behavior. The method is evaluated with a public of 24 participants in order to assess its Comprehensibility, Accuracy and Efficiency;
5. **Chapter 6:** closes the main body of the thesis with concluding comments and proposals for future work;

Chapter 2

State of the art

2.1 Introduction

The development of secure applications is a very difficult task, as security is not a common skill in development teams. In addition, the security aspects are often considered lately in the application life, which makes the security decisions and choices very clumsy. Hence, it is very important to provide designers with a panoply of approaches, tools, methodologies and resources in order to help them in designing and implementing secure applications.

Our work is based upon different notions: basic security notions (weaknesses, vulnerabilities, attacks, etc), methods of security modeling, design and security patterns. This chapter aims at pointing out these aspects. We firstly present the notion of patterns then its use in software design and security. We highlight the challenges to which developers are faced in using patterns. We give an insight on the different approaches of graphical security modeling and analysis. We give an insight on the currently available security models and we illustrate some of them through examples.

This chapter is structured as follows: in Section [2.2](#), we give a brief introduction of some basic concepts of security. We present the differences among weaknesses, vulnerabilities,

attacks and threats in addition to the relations among them. Then, we introduce some noticeable publicly available data sources of weaknesses, attacks and vulnerabilities.

In Section 2.3, we give a chronological presentation of the notion of patterns and the design patterns. We present the differences of presentation in design patterns documentation in addition to a short vision on design patterns writing, selection and use. In Section 2.4, we introduce security patterns by means of some examples, we describe the major difficulties in security patterns selection and application in addition to the proposed approaches for helping designers in these tasks. Hence, we give a birds eye on the different security patterns classifications, repositories and relationships. Section 2.5 gives an insight on graphical models for security analysis. We give some discussions in Section 3.6 and we conclude the chapter in Section 3.7.

2.2 Security concepts and databases

As we pointed out earlier, devising a secure application is a difficult task. The major reason is that security concepts and artifacts are not common skills in development teams. Models of applications, quickly designed, often contain flows, which harm the Confidentiality, the Integrity and the Availability of the resources provided by the application. Indeed, an insecure application often encloses **Weaknesses**, each weakness being an error that can lead to a **Vulnerability**. A vulnerable application is **Threatened** by **Attacks**, each attack targets one ore more vulnerabilities.

It is important that designers understand these concepts. In this section, we shortly recall them and how they interact together. In addition, we present some public resources about attacks and vulnerabilities available for designers.

2.2.1 Weaknesses

A weakness in an application is a flaw, a fault, a bug or an error in one or more levels of the application. Hence, a weakness can be introduced in the implementation, code, design or the architecture level [69, 76]. When a weakness is left untreated, this could result in a vulnerability. Thus, a weakness is more abstract than a vulnerability since a weakness can be implemented in different ways giving many vulnerabilities. Furthermore, a weakness can lead to either a vulnerability or an exposure. An exposure is an error or a misconfiguration that can be used by a hacker as “*stepping-stone*” into the application. In other words, an exposure does not allow a direct compromise of the application security, but it can be considered as an important step for a successful attack. For instance, an exposure can allow information gathering, hiding the attacker activities or can be a primary entry-point to the application, etc [68]. Hence, a weakness is more abstract and covers more notions than a vulnerability.

The most important resource of information about weaknesses is the CWE (Common Weakness Enumeration) database. The CWE database lists 705 software weaknesses (Version 2.11) and makes available a specific and succinct definitions of each common weakness type. The construction of the CWE database aims at providing a common unified baseline standard for weakness identification, mitigation and prevention efforts [69].

2.2.2 Vulnerabilities

A vulnerability is a result of an untreated weakness introduced in the application, which should be fixed once it is discovered. In contrast to a weakness, a vulnerability is specific to a distinct version of a product. In other words, a weakness is a type of a failure that can occur in all the applications in a context, while a vulnerability is defined for an individual version of a product.

The most important resource of vulnerabilities information is the CVE (Common Vulnerabilities and Exposures) database. The CVE database lists at the moment 85684 vulnerabilities (May 16, 2017 version) and the number of vulnerabilities is continuously growing. In average

State of the art

4400 vulnerabilities are discovered each year. Figure 2.1 illustrates the number of vulnerabilities discovered per year. Since 1999 the number of the vulnerabilities varied between 894 vulnerabilities and 7946 vulnerabilities [97].



Fig. 2.1 The Number of Vulnerabilities per year

The growing number and the frequent discoveries of vulnerabilities implies that maintaining an up to date documentation is a tedious problem. It was vital to abstract these information in the form of weaknesses in the CWE database in order to help practitioners in understanding software security failures, which is impossible with the information provided with vulnerabilities.

2.2.3 Threats

A threat is a potential risk or cause of an undesirable incident that could harm or regress the security of one ore more valuable resources of an application [25]. In other words, a threat

is the possibility that a valuable asset of the application could be vulnerable to one or many types of attacks. Security is often relative to the elements being protected, the skills and the resources of stakeholders and the cost of the potential mitigations and countermeasures. Hence, the threat management helps identify the potential security problems and evaluates their impact and severity [78].

2.2.4 Attacks

An attack is the description of a sequencing of actions that an attacker would do in order to compromise the Confidentiality, the Integrity or the Availability of the application assets. It is important to be careful with describing a successiveness of actions when talking about attacks. Understanding attacks is crucial for designers since it allows them to understand the attacker point of view, capabilities, and how-to in order to protect their applications.

A noticeable resource of attacks is the Common Attack Pattern Enumeration and Classification (CAPEC) base. The CAPEC base provides a publicly available list of 510 attack patterns in a comprehensive schema [67]. In the CAPEC database, an attack pattern describes, through a set of sections, the elements and techniques generally used in attacks against vulnerable systems. In addition, it describes the execution steps of the attack, the prerequisites, solutions, related attacks and weaknesses, etc.

2.2.5 Security principles

A security principle is a desirable property, structure or behavior of software that aims at reducing the impact and the likelihood of a threat realization [106]. They give an insight on the nature of close security tasks whose contexts are not taken into consideration. Saltzer and Schroeder firstly proposed a set of eight best practices for system security [90], which were widely expanded in the last decades to form security principles [106, 62]. We recall in the following some basic security principles:

1. **Access Control:** the access control security principle defines the mechanisms of the *Identification/Authentication* of entities (users or services), the definition and the verification of their permissions and access rights in addition to the *Accounting* [94]. For instance, it includes the implementation of the AAA (Authentication, Authorization, Accounting) protocols;
2. **In Depth Defense:** inspired from a military strategy, this security principle aims at protecting a system with a layered set of security facilities. It may include a set of principles that interact with each other in a layered architecture like: The *Complete Mediation*, *Perimeter Security*, *Firewalling*, etc [101];
3. **Fault Tolerance:** describes the ability of an application or a system to continue operating normally (or in a reduced way) when the application is in case of failure, in addition to enhancing the manageability of failures in the application [62];
4. **Sensitive Data:** this principle addresses the prevention of sensitive data disclosure. For instance, it can describe the encryption, privacy, etc [62];
5. **Configuration management:** in software systems, it is crucial to protect the configuration items, their storing, and the access to them. This principle expresses these needs, it includes Configuration protection, Privilege management, Fail-safe defaults, etc [62];
6. **Security Simplification:** As software complexity increases, so does the risk of vulnerabilities. This principle refers to the KISS (keep it simple stupid) concept. It includes the economy of mechanisms, psychological acceptability, open design, etc [62, 106].

2.2.6 Security by design

2.3 Patterns

Originally, the idea of patterns was laid by the architect Christopher Alexander. In the context of urban planning and building architecture, Alexander and his team identified more than 250 patterns [96]. They were used by builders over centuries and, every time, these patterns were adapted to the environment. They also noticed that even with the adaptations of the patterns, some recurrences in conception solutions was very efficient [13]. Hence, they identified the context-problem-solution of patterns, known as the Alexandrian form. This form is a pattern language allowing non-architects to base their conceptions on patterns in order to build their own solutions [13]. Hence, a pattern can be defined as a reusable solution, issued from human experience, to a recurrent problem.

Applying the pattern approach in the software development was firstly introduced in 1987 by Ward Cunningham and Kent Beck. They adapted the Alexandrian form on software design problems, they proposed five patterns dealing with the design of user interfaces. This marks the birth of patterns in software engineering [95].

In 1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, four software design experts, known as the “Gang-of-Four” (GoF) [32] in the patterns community, paved the way for the acceptance of patterns in software engineering. They introduced a template allowing the description of software patterns [107]. Numerous design experts followed this path producing a large list of patterns [95]. Moreover, they distinguished different pattern families for each step of software life-cycle. Namely, there is “Architectural”, “Analysis” and “Design” patterns.

Whatever the context in which they are used, patterns present an effective mean to exchange human knowledge and experience of experimented designers. A pattern allows the expression of a reusable solution, practiced by an expert, to a problem that he/she met many times. Besides

the advantage of being reusable, a pattern allows novice designers to take advantage of the human experience in order to gain time and to produce a qualitative design. The abstract nature of design patterns requires their adaptation to the application context, this task is called “instantiation”. Nevertheless, the selection and the adaptation of a pattern in a specific context is a difficult and error prone task because of the abstract and the textual nature of patterns and the large number of available patterns in the literature.

In this section, we shortly recall the foundations of software patterns. We introduce the different types of patterns and how they are expressed. Furthermore, we highlight the difficulty of selecting and implementing a pattern for a specific context.

2.3.1 Design patterns

The aim of a design pattern is to describe an abstract, reusable and experimented solution to a recurrent problem in the design phase of the software life cycle. It helps in reducing considerably the time needed to solve design problem in addition to enhancing the quality, the documentation and the maintainability of the application under design [13]. According to [64], a design pattern have to be essentially described with a *Name*, *Intent*, *Context*, *Problem*, *Forces*, *Solution*, and *Consequences* Table 2.1 gives an insight on each one of these sections.

The textual sections of a pattern document may differ from an author to another. Indeed, some other elements may be added in order to make the pattern easier to understand. The pattern document if often provided with concrete *Examples* showing how the pattern can be used in code. In addition, section *Known Uses* points out the known occurrences of the pattern in existing systems. The responsibilities of the different parts of the pattern (described in the class diagram) and the interactions among them can be outlined in two sections named *Participants* and *Collaborations*. Some sections may have different names from a documentation to another. For instance, the section *Context* is some times called *Motivation* [96].

Table 2.1 Design pattern sections

Name	The name of a pattern has to be unique and evocative giving an image of what the pattern is about. The relation between the pattern and its name is determined according to the conventions established in the patterns community.
Intent	The description of what the pattern is about. In other words, it describes the objective of using the pattern in an application.
Context	The context of a pattern presents the environment in which the pattern can be applied. The context shows in which situation the problem described by the pattern may appear and where the pattern will work.
Problem	It expressed the problem to which the pattern may be the solution. The problem targeted by the pattern has to be recurrent and non-trivial, which means that it is a typical problem in the given context for which the appropriate solution is known only by expert designers.
Forces	The set of the forces of a pattern highlights its specificities making it better in a given situation. It helps to determine why the choice of the pattern is justified for a problem.
Solution	It gives a proven solution to the stated problem. The solution has to be abstract and reusable. Usually, it is described with UML diagrams but it can also be described in different forms like textually, with petri networks, etc. The solution described by a pattern has usually two faces, the first one is structural expressed usually with a class diagram presenting how to structure the application with the pattern, it allows also to outline the participants of the pattern. The second face is behavioral, it can be expressed with sequence or state diagrams in order to present how the application has to behave with the application of the pattern.
Consequences	The consequences of a pattern describe the impact of the pattern on the structure and the behavior of the application in which it is used. In other words, it describes how the application will be after the use of the pattern.

A design pattern can be characterized with a set of strong points. Firstly proposed by Bouhours et al., strong points present are desirable structural and behavioral properties brought by the pattern use [13]. Strong points are partially extracted from the pattern consequences and forces text sections.

Furthermore, since “*no pattern is an island*” [95], a pattern is often related to other patterns. There are many types of relationships among patterns [29]. For instance, the solution provided by a pattern can be often implemented by another pattern resolving a sub-problem of the general problem targeted by the original pattern, we talk here about a refinement relationship

[95]. The use of a pattern may depend on the use of one or more patterns. In other words, the application of a pattern *A* depending on a pattern *B* is possible only if the pattern *B* is also applied. Therefore, a section called *Related Patterns* is often added to the documentation of patterns to highlight the set of the related patterns and the nature of these relationships.

These differences in the pattern sections is one of the sources of heterogeneities in the patterns documents. These heterogeneities lead to one of the major origins of difficulty for understanding a pattern.

2.3.2 Patterns selection

As mentioned earlier, the use of a pattern in an application may lead to the use of a set of related patterns. In addition, the number of patterns proposed in the literature is still growing and the pattern documents are often heterogeneous. These results in the difficulty of choosing the appropriate pattern in a given context, especially for novice designers.

A proposed solution to help designers in the patterns selection is the creation of collections of patterns. The aim of a pattern collection is to provide a unique template to describe a pattern set, making the documentation as homogeneous as possible. In [16], Buschmann et al. identified three types of patterns collections “*Pattern Catalogs*”, “*Pattern Systems*” and “*Pattern Languages*”, which are considered as the steps of a patterns collection evolution.

- *Pattern Catalogs*: the first objective of a pattern catalog is to gather several patterns into one bigger collection of patterns. The two main challenges of building a pattern catalog are: to deal with the heterogeneity of the sources of patterns and to find the relationships among patterns. Hence, a pattern catalog can be provided with a non-uniform structure and presents a loosely coupled set of patterns [96];
- *Pattern Systems*: a pattern system is an evolution of a pattern catalog. In contrast to a pattern catalog, a pattern system expresses a more uniform and a more precise description

of the inter-pattern relations [96]. Therefore, this increases the readability and the understandability of patterns, which helps in selecting and using patterns for a specific problem. Hence, a pattern system is a more coupled set of patterns;

- *Pattern Languages*: they exhibit the last stage of a pattern collection evolution. A pattern language is a tightly interlacing set of patterns as a “*super-pattern*” in which patterns share a pre-defined goal and each pattern contributes in this goal at its level. Hence, pattern languages are more robust and comprehensive than pattern systems. Nonetheless, it remains difficult to prove the completeness of the patterns languages this is why some experts prefer to talk about patterns systems [96].

Even though a pattern collection can help choose the right pattern, it is also important to provide a pattern organization to enable concrete and fast choice of patterns. In other words, it is well admitted that classifying patterns is important to make the choice of patterns easier to enhance the accuracy of the selection and to reduce the time needed to the selection.

Since the birth of design patterns, a plethora of classifications have been proposed. A pattern classification is an organization of patterns over some criteria. For instance, the GoF classifies design patterns by two criteria: the first one is the *purpose* of the pattern (what the pattern does) and the second one is the *scope* of the pattern (whether the pattern is related to classes or objects) [96]. Buschmann et al. defined the POSA (Pattern Oriented Software Architecture) approach in which they depicted the categories of patterns (Architectural patterns, Design patterns and Idioms) from high to low level patterns [16]. Many other pattern classifications have been proposed in the literature to organize patterns from different points of view.

After its selection, a pattern is adapted in order to meet the specific context of the application. In other words, the abstract nature of patterns requires the instantiation of the pattern on the context of the application under design.

2.3.3 Patterns instantiation

After being chosen, a pattern has to be instantiated to meet the context and the environment of the application under design. Instantiating a pattern in an application is an error prone task. Indeed, a pattern is an abstracted solution. It can be misunderstood and misused, which can lead to a considerable deterioration of the design [13]. Hence, the designer must ensure that the pattern has been appropriately instantiated in an application model.

In order to help designers in this task, multiple works proposed the detection of design patterns in models [7, 11, 41, 13, 46]. This field of research attracted researchers from academia and industry. Al-Obeidallah et al. analyzed 32 works dealing with design pattern detection and they classified these works into four families (Database Query Approaches, Metrics-Based Approaches, UML Structure, Graph and Matrix Based Approaches and Miscellaneous Approaches) in order to guide designers in the choice of the appropriate approach [3]. The growing number of design patterns detection methods shows the difficulty of instantiating patterns.

2.4 Security patterns and Security patterns classifications

Software designers usually keep in mind what the application should do (the functional requirements) and forget what the application should not allow to do (the non-functional requirements) especially the security. Indeed, the security aspects in early stages of an application life-cycle is often under-considered and then less documented than the functional requirements of the application.

When the design of an application does not consider the security aspects, developers often have to retrofit the application to meet security requirements in the implementation. This phase is time consuming and very difficult. Considering security since the design phase enhances the efficiency of implementing and maintaining the security mechanisms up to the last stages of the

2.4 Security patterns and Security patterns classifications

application life-cycle. Security patterns represent a solution to take into consideration security at the design stage. Security patterns was proposed in order to capture human experience about the frequently met security problems and to propose abstract an reusable solutions to these problems since early stages of the application life cycle.

Based on the template of Gof, the first set of patterns dealing with security problems was presented by Yoder and Barcalow in 1997, which makes them the pioneers of security patterns. In their article “*Architectural Patterns for Enabling Application Security*” [112], they proposed seven security patterns (Single Access Point, Check Point, Roles, Session, Full View With Errors, Limited View and Secure Access Layer). They made analogy to a military base in order to clarify the application contexts, the problems and the solutions of the proposed patterns in addition to real-world software examples where these patterns were successfully used. Furthermore, they recognized that security should be considered from the beginning of any software development [112]. Step by step, the recognition of security patterns as a new pattern category started to take place. Many security patterns have been proposed covering most of the software security aspects. As a result, we count nowadays 176 security patterns [87].

Schumacher et al. defined security patterns with: “*A security pattern describes a particular recurring security problem that arises in specific contexts, and presents a well-proven generic solution for it. The solution consists of a set of interacting roles that can be arranged into multiple concrete design structures, as well as a process to create one particular such structure*” [96]. From this definition, three concepts can be extracted, the first one is a security problem frequently met in the design stage. The second concept is a context in which the security problem arises. The third concept is the solution provided by the pattern, which has to be reusable, abstract and adaptable into different concrete design structures. Therefore, a security pattern SP can be defined with a triple $SP := (C, P, S)$ where C is a security context, P a security problem and S a security solution [96].

In the next section, we present security patterns through some examples. Then, we highlight some challenges in using security patterns, which mainly relate to the difficulties of selection and the verification of their instantiations. For the pattern selection, we introduce some noticeable collections of patterns and the different classifications of patterns proposed in the literature along with quality criteria allowing to evaluate these classifications. Then, we outline the different inter-patterns relationships and the importance of these relations in pattern driven secure design.

2.4.1 Security patterns

The first two examples of patterns we chose to “*Authorization Enforcer*”. The Intent of these two patterns is 1) to provide the application with the ability of controlling the access to a protected component with an authentication mechanism 2) to check whether an entity is allowed to access this component. Moreover, the instantiation of these patterns structures the application in such a way that the implementation of custom authentication and authorization mechanisms is easier [100].

Table 2.2 summarizes the pattern *Authorization Enforcer* description. It provides the application with an encapsulated and centralized permission checking mechanism. By applying *Authorization Enforcer*, the designer takes advantage of structuring the application in such a way that access control implementation is easier and more maintainable. As illustrated in Figure 2.2, the pattern *Authorization Enforcer* is composed of six classes: a **Client**, which is the source of the request to the protected resource, **Secure Base Action**, which is a security front controller offering an intercepting point. In addition, the **Subject** is an object gathering the user identity, credential informations and a set of permissions stored in the **Permissions Collection**, which are set by the **Authentication Provider** according to the user identity. Based on this set of the permissions the **Authorization Enforcer** decides whether the requester is allowed or not to access the protected component [100].

2.4 Security patterns and Security patterns classifications

Table 2.2 Authorization Enforcer [100]

Name	Authorization Enforcer
Intent	The intent of this pattern is to enable the application to verify that entities requesting for protected components are properly allowed to access these components.
Problem	In complex applications, requests can cross multiple paths to access the protected functionalities that the application provides. Authenticating these entities is not sufficient since all the different roles in the application are not allowed to access all the functionalities of the applications. Hence, the application has to be able to verify the permissions of each entity requesting these protected components. Therefore, the application has to be able to check for each request whether the requesting entity is allowed or not to access to the solicited functionality or component.
Forces	Minimize the coupling between the view presentation and the security controller. Authorization logic required to be centralized and should not spread all over the code base in order to reduce risk of misuse or security holes. Authorization should be segregated from the authentication logic to allow for evolution of each without impacting the other.
Solution	Structure Figure 2.2 Dynamics Figure 2.3
Consequences	The application of Authorization Enforcer provides the application with an encapsulated authorization mechanism, which is isolated from the functional parts. This helps in reducing code and enhancing the reusability, the readability and the maintainability of the application. In addition, Authorization Enforcer promotes the separation of responsibility by decoupling the authorization and the authentication mechanisms in order to allow the implementation of more complex access control techniques.

Besides this structuration of the application provided by the instantiation of *Authorization Enforcer*, Figure 2.3 illustrates the behavior aspect of the pattern. The **Secure Base Action** retrieves the **Subject** from the **Request Context** then **Secure Base Action** invokes the “authorize” method of **Authorization Enforcer** with the **Subject** as parameter. The **Authorization Enforcer** delegates the authorization to the **Authentication Provider**, which retrieves the set of permissions for the subject and creates a permission collection according the identity of the **Subject** (stored in the subjects public credentials set). Afterwards, the **Authorization Enforcer** retrieves the **Subject** permission collection in order to decide whether it is allowed to access the protected component [100].

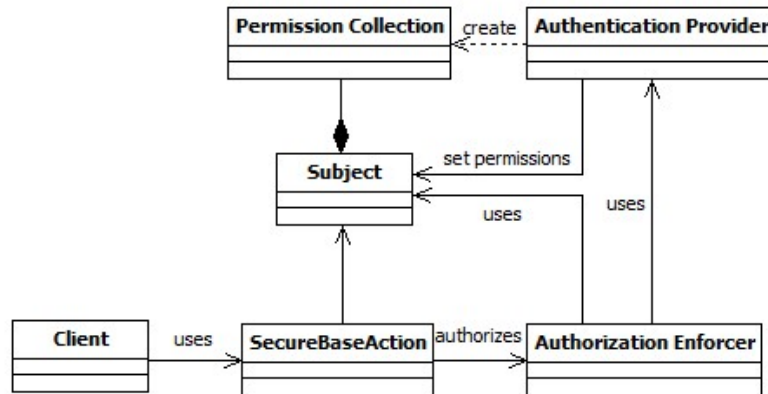


Fig. 2.2 Authorization Enforcer Structure

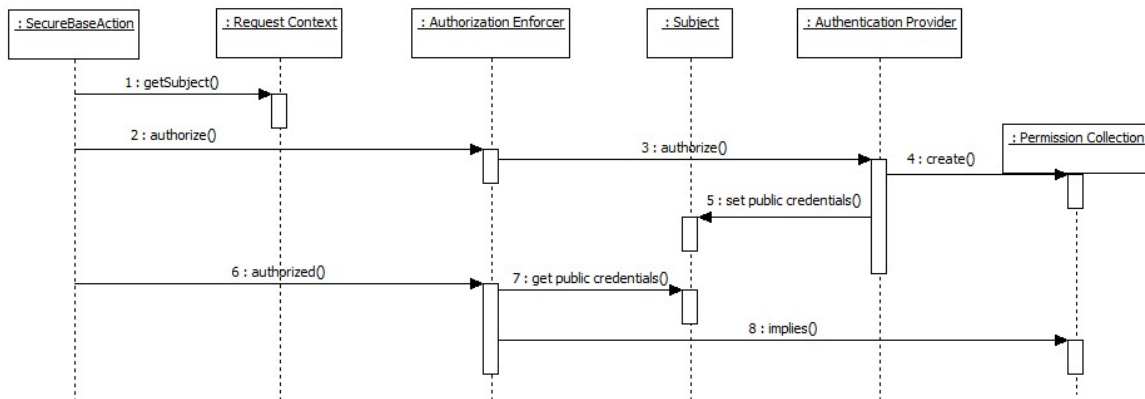


Fig. 2.3 Authorization Enforcer Behavior

The ability of the pattern **Authorization Enforcer** to control access is conditioned by the set of rights provided with the object **Subject**. To check the correctness of these rights, the application of the *Authorization Enforcer* takes advantage on the use of *Authentication Enforcer* in order to authenticate entities before giving an authorization decision.

Outlined in Table 2.3, the pattern *Authentication Enforcer* helps designers implement a custom authentication mechanism. The use of this pattern aims at checking the identity of each entity requesting the application and makes the authentication logic centralized and isolated from the other parts of the application. Then, the authentication code is reduced and it is less replicated over the application components. As illustrated in Figure 2.4, this pattern

2.4 Security patterns and Security patterns classifications

Table 2.3 Authentication Enforcer [100]

Name	Authentication Enforcer
Intent	The intent of this pattern is to enable the application to verify that each request is from an authenticated entity.
Problem	When an application contains a protected components, it is important to check whether the requests to these components are from authenticated entities. Since different classes can handle the requests, the authentication code can spread over these classes, which leads to the replication of the authentication code in many places in the application. In addition, implementing the authentication mechanisms often implies changes in the functional logic of the application, which harms the simplicity and the maintainability of the application. Therefore, it is important to make the authentication mechanism centralized, encapsulated and isolated from the functional logic of the application. Moreover, the user credentials should be kept secret and unreachable from the other parts of the application or the other coexisting applications.
Forces	Access to the application is restricted to valid users, and those users must be properly authenticated. There may be multiple entry points into the application, each requiring user authentication. It is desirable to centralize authentication code and keep it isolated from the presentation and business logic.
Solution	Structure Figure 2.4 Dynamics Figure 2.5
Consequences	With the use of the Authentication Enforcer pattern, the authentication mechanism is centralized and isolated from the other parts of the application. This allows the designer to benefit from the reduced code and the enhanced maintainability and the readability of the code since the authentication logic can change frequently in the lifetime of the application.

is composed of four classes: the **Client** requests are given to the **Authentication Enforcer**, which aims at authenticating the **Client** with regard to the credentials of the user provided by means of the **Request Context**. The **Subject** instance is created if the user is authenticated. Figure 2.5 illustrates the behavior of *Authentication Enforcer*. The **Client** creates a **Request Context** instance containing the user credentials. These credentials are retrieved from the **Authentication Enforcer** and compared to those stored in the **User Store**. If the credentials provided by the user are correct, an instance of **Subject** containing the user information and credentials representing the users identity is created.

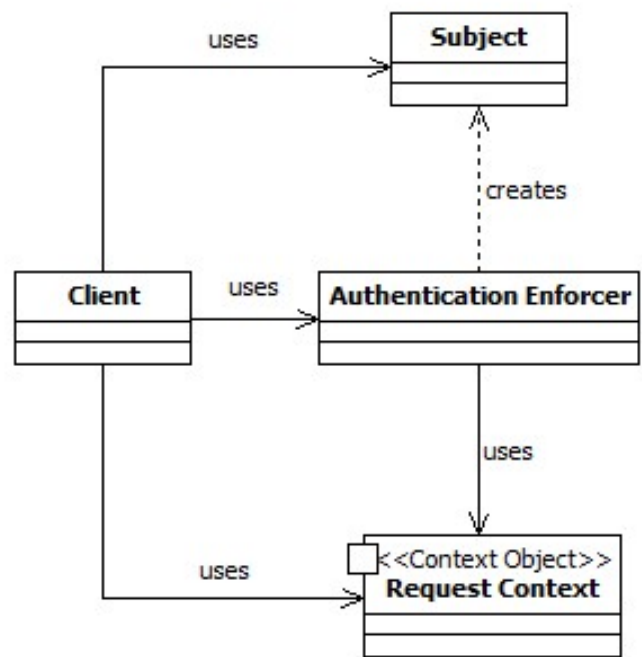


Fig. 2.4 Authentication Enforcer Structure

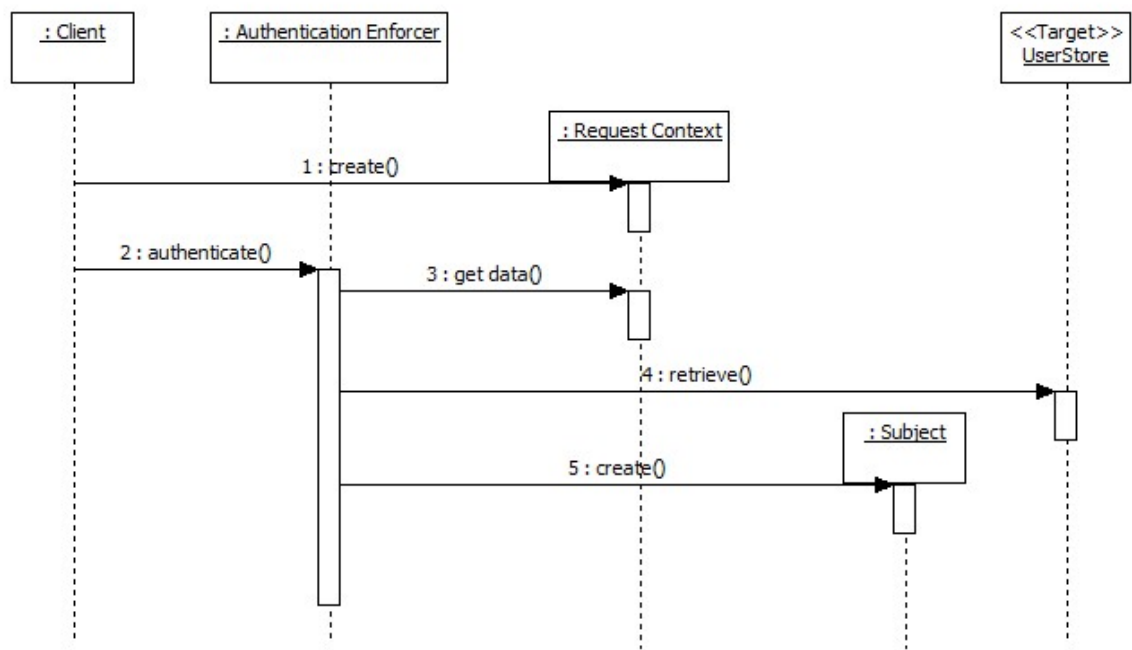


Fig. 2.5 Authentication Enforcer Dynamics

2.4 Security patterns and Security patterns classifications

These examples show that the pattern *Authorization enforcer* is related to the Authentication Enforcer and how it is important to highlight these relations in order to ease the use of these two patterns. Indeed, one can consider that the authorization and the authentication are two sequential steps.

Many security patterns are interrelated, and these inter-pattern relationships was heightened since the presentation of the first set of the security patterns by Yoder and Barcalow [112].

2.4.2 Security pattern instantiation

The instantiation of a security pattern in an application consists in adapting the structural and the behavioral properties of the security pattern [95] on the context of the application. This implies that the designer has to understand both the security pattern properties and the application context.

In this section, we illustrate a pattern instantiation example in addition to some examples of classical errors. We present how a bad instantiation of security pattern harms its effectiveness. We consider as example the security pattern “*Intercepting Validator*”, whose structure is

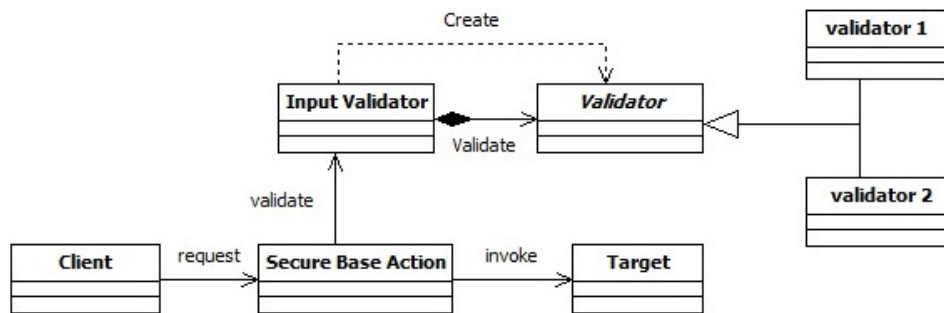


Fig. 2.6 Intercepting Validator Class Diagram

illustrated in Figure 2.6 and behavior is illustrated in Figure 2.7. It aims at providing the application with a centralized validation mechanism for each data type used in the application model. This validation mechanism is decoupled from the other parts of the application and

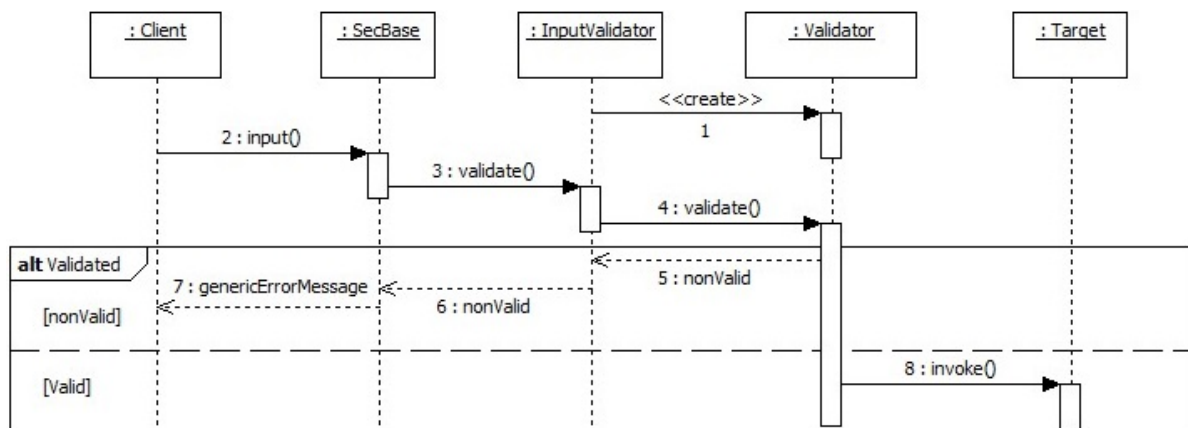


Fig. 2.7 Intercepting Validator Sequence Diagram

each data supplied by the client is validated before being used. The validation of input data prevents attackers from passing malformed input in order to inject malicious commands.

A security pattern is appropriately instantiated if the application model satisfies all of the patterns strong points. Otherwise, the security pattern is not efficient against the targeted security problem, or worst, its instantiation may imply security problems. The pattern “*Intercepting Validator*” is characterized by four strong points:

- each input has to be validated before being used by the application;
- a validation logic for every data-type used in the application;
- a unique and centralized validation mechanism;
- the separation of the validation logic from the presentation logic;

2.4 Security patterns and Security patterns classifications

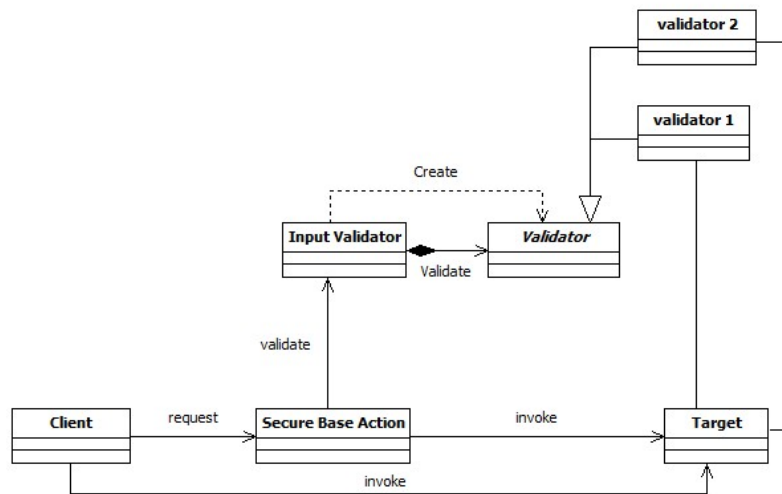


Fig. 2.8 Bad Instantiation of *Intercepting Validator*

Let us consider the UML class diagram of Figure 2.8 in which a designer has tried to instantiate the security pattern. The security pattern structure seems to be into the model. However, this instantiation is incorrect for two reasons: the client is directly connected to the target, which means that some inputs can be passed to the target without validation. The second reason is that the validators are directly connected to the target, which means that, in case of the use of a new data-type, many elements of the application have to be modified. Hence, the validation logic is not well decoupled from the functional logic of the application. Table 2.4 summarizes these issues; two strong points of the “*Intercepting Validator*” pattern are not satisfied, which indicates that it is not properly instantiated.

Table 2.4 The satisfaction of Intercepting Validator String points in the example Figure 2.8

Satisfied	Strong Point
X	each input has to be validated before being used in the application
✓	a validation logic for every data-type used in the application
✓	a unique and centralized validation mechanism
X	the separation of the validation logic from the presentation logic

Hence, it is crucial for designers to well understand a security pattern before its use. They have to be able to check that all the patterns strong points are present in the application model, which is a difficult and error prone task.

2.4.3 Inter-patterns relationships

As stated previously, highlighting the inter-pattern relationships helps the designers select patterns selection and combine them. These relations among patterns are outlined since the proposition of the first set of patterns by Buschmann et al. in [16]. It worths mentioning that these relations among patterns was also introduced in the GoF book [32].

Since the proposition of the first collection of security patterns, the template of security patterns proposed by Yoder and Barcalow [112] was provided with a section called related patterns. However, with the growing number of patterns, the choice of the appropriate solution becomes a very difficult task.

Hafiz et al. proposed an organization of security patterns to highlight the different relations among them [36]. They gathered a set of 97 security patterns from many sources and organized them manually. As illustrated in Figure 2.9 the resulted map of their organization covers many different types of relationships [35]. Hence, the designer has to look over this complex map in order to choose the appropriate solution. In addition, she/he has to understand the natures of these relationships to take decisions. Other works proposed to unify these relationships to clarify the security pattern combination.

In the literature, two types of security patterns organization are proposed, vertical and horizontal organizations. Vertical organizations present how more abstract patterns are related to more concrete security patterns. Horizontal organizations deal with the different types of collaborations among security patterns and how each couple of patterns work together.

Yskout et al. proposed in 2006 a horizontal unification of the inter-patterns relationships. They considered that a pattern A can the following relations with a pattern B [116].

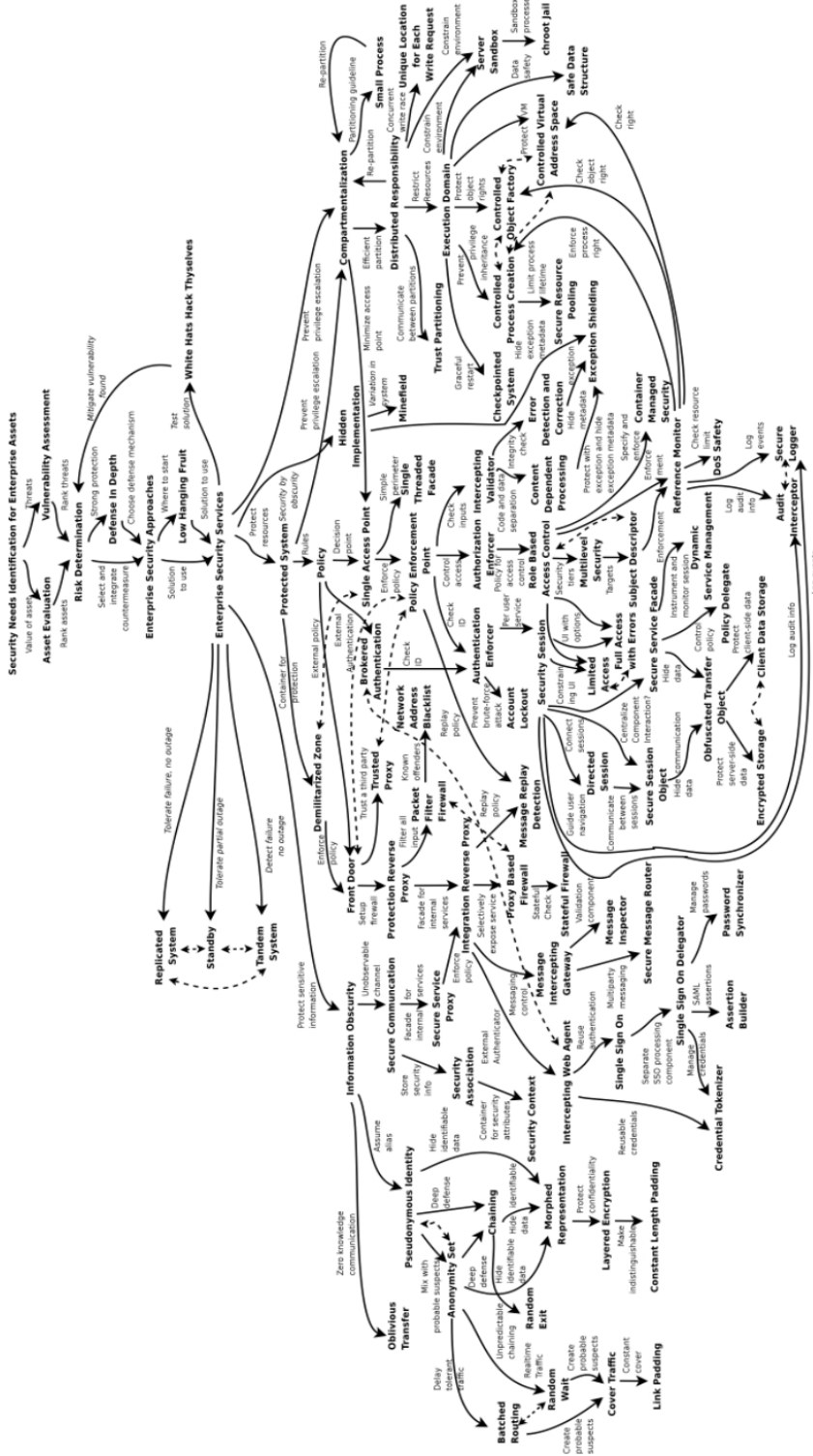


Fig. 2.9 Organization of patterns proposed by Hafiz et al

- **Depend:** means that two patterns have to be used together: if a pattern *A* depends on a pattern *B*, the pattern *A* can not be correctly implemented without the application of the pattern *B*. For instance, the use of the security pattern *Audit Interceptor* to intercept and audit requests and responses to and from the application strongly depends on the use of *Secure Logger* pattern, which provides the application with a secure logging facility. Otherwise, it is impossible to ensure the integrity of the audit trails [116];

“Depend” relationship is not symmetrical, if a pattern *A* depends on a pattern *B* it does not mean that the pattern *B* depends also on the pattern *A*;

- **Benefit:** if a pattern *A* benefits from a pattern *B*, then the use of *B* will enhance the security capability provided by the pattern *A*. For instance, *Authorization Enforcer* benefits from *Authentication Enforcer* since the use of the authentication improves the authorization by authenticating the identities;

“Benefit” relationship is not symmetrical, if a pattern *A* benefits from *B* this does mean that *B* benefits from *A*. “Benefit” is also non-transitive. For instance, the *Authentication Enforcer* benefits of the use of the pattern *Secure Pipe* to provide the application with encrypted channels between the users and the application. *Authorization Enforcer* does not necessarily benefit from the use of *Secure Pipe*;

- **Alternative:** this relationship indicates that two patterns, can have similar functional capabilities. If *A* and *B* are alternative patterns, this means that if a pattern *A* is used in an application, then *A* can be replaced by *B* without any negative impact on the overall application structure and behavior. For example, the *Limited View* and *Full View With Errors* patterns are alternative. While they are not totally equivalent (i.e., only authorized operations are displayed using *Limited view* and all the functionalities are displayed with *Full View With Errors*), they are functionally equivalent since both of them prevent users from using the application functionalities that they are not authorized to perform.

2.4 Security patterns and Security patterns classifications

The “alternative” relation is symmetrical, if A is alternative to B , then B is also alternative to A . It is also a transitive relationship;

- **Impair:** this relation means that two patterns can harm each other when used together. If a pattern A impairs a pattern B , then the use of A can decrease the correct functioning of B . For instance, the *Checkpointed System* pattern structures the application so that it is possible to store the application state and recover it in case of fail. This pattern impairs the *Audit Interceptor* pattern. When an application fails, the actual operation might not be completed so it can be reflected on the log as a false operation. It is important to notice that when a pattern A impairs a pattern B it is not totally impossible to use them together, however, the designer should take care on the inconsistencies that might result on their simultaneous use.

The “impair” relation is symmetrical and non-transitive;

- **Conflict:** The use of two conflictual patterns considerably degrades the behavior of the application. In other words, if a pattern A conflicts with a pattern B , then the use of both patterns in the same application will result in inconsistencies. For example, the use of the two alternative patterns *Limited View* and *Full View With Errors* would bring inconsistencies in the application and might lead to privilege escalation, authorization bypass, etc. “Conflicts” is symmetrical and non-transitive.

The difference between impair and conflict relationships is that if the pattern A impairs B this means that the use of A can harm the efficiency of B , while if A and B are conflictual, then the application will be vulnerable.

An exemplary vertical organization of security patterns was proposed by Delessy et al. in 2008 [22]. They categorized these relationships into four classes:

- **Realization relationship:** a pattern A realizes a pattern B when A is less abstract than B and A provides the solution to the same problem as B in a more specific context;

- **Specialization relationship:** a pattern A specializes a pattern B when the patterns can be applied at the same architectural level and the pattern gives A a more detailed solution than the pattern B for the same security problem. The relationships Specialization and Realization are opposites;
- **Containment relationship:** a pattern A is contained in a pattern B when the security problem solved by the pattern B includes the problem solved by the pattern A and the functioning of B uses the pattern A in its solution;
- **Collaboration relationship:** two patterns A and B collaborates when the two patterns are often used together, which means that A can depend or benefits of B .

Outlining the relations among security patterns in some collections helps considerably in the selection and the combination of the security patterns. However, it is still very difficult for novice designers to select the appropriate set of patterns in some specific contexts. Because of the different documents about security patterns, some researchers proposed to classify them [28].

2.4.4 Security patterns classifications

The classification of security patterns is considered by many authors as one of the important means for helping designers in security patterns selection and use [28, 4, 6]. Indeed, since the appearance of the security patterns, a plethora of classifications were proposed in the literature.

Classifying security patterns consists in categorizing the patterns in disjoint divisions with regard to specific criteria. The classification criteria can have different forms: they can be based on some pattern properties like the purpose, the consequences, etc. It also can be based on the application contexts such as the life cycle, the architectural layers or the categories of users. Moreover, classification criteria can be based on security concepts like vulnerabilities, attacks,

2.4 Security patterns and Security patterns classifications

weaknesses, threats, etc. A classification can also be built with multiple attributes, which is considered as finer than relying on only one attribute [5].

The growing number of classifications shows the efforts of the security pattern community in making them more understandable. The security patterns classification is a very difficult and error prone task though because of the textual and the abstract nature of the patterns documentation, and because of the diversity of the patterns sources. Hence, classifications are often made manually comparisons and evaluations of the criteria.

In the following, we review some classifications proposed in the literature and their classification schemes. Then we will discuss about the quality of these classifications.

2.4.5 Classification schemes

In 2001, Schumacher et al. [96] organized security patterns regard to application contexts. More precisely, the proposed approach organizes security patterns according to two dimensions: the first dimension is scaled on the software life-cycle phases and the second dimension on the architectural layers of the application. Moreover, they considered that the pattern problem domain can be seen as a third dimension. This classification does not consider the security concepts (vulnerabilities, Weaknesses, etc.).

In 2002, Kienzle et al. gathered 29 security patterns into two types: structural and procedural patterns [47]. They developed a system of security patterns called “*public Web repository system*” as an experimental base for security patterns.

In 2006, Laverdiere et al. found five levels of security patterns representation quality and defined them as “*undesired properties*” [53]. From better to worst, they considered that security patterns can be over-specified, under-specified, lacking generality, lacking consensus, or misrepresented. Then, they classified a set of security patterns over these criteria.

The same year, Hafiz et al. [36] proposed a multi-dimensional security patterns classification. They firstly based the classification of security patterns on the CIA (Confidentiality,

Integrity, Availability) model. Then, they classified the security mechanisms into three layers: core, perimeter and exterior security mechanisms. They conjugated these three layers over the STRIDE (Spoofing, Tampering, Repudiation, Information disclosure, Dos, Elevation of privilege) method, which is one of the important threat modeling and management methods. Then, they arranged 97 security patterns through these criteria. This repository of security patterns is considered as one of the most important collection and classification of the security patterns.

The classifications proposed in [6, 4] give another point of view by helping designers in the choice of patterns to fix software vulnerabilities and weaknesses. This choice of categorization seems quite interesting and meaningful since security vulnerabilities are often known by designers and are the natural causes of attacks on software systems. Alvi et al. proposed a vulnerability based scheme putting together security patterns and weaknesses documented in the Common Weakness Enumeration (CWE [69]) database [4]. They considered that the CWE weaknesses are appropriate to document flaws added through the design phase and they manually linked security patterns to CWE weaknesses from their textual descriptions. Anand et al. proposed another security pattern catalog composed of 12 families of vulnerabilities and identified some missing security patterns [6]. They focused on vulnerabilities because they considered that the CWE database is bigger than the scope of their work. They grouped software vulnerabilities into families and manually collected relationships between families and security patterns from the pattern textual descriptions and their vulnerability family definitions.

Besides classifying patterns against software vulnerabilities, other authors proposed security pattern classifications with regard to the concept of attacks [110, 102, 4, 104]. Wiesauer et al. initially presented in [110] a short taxonomy of security design patterns made from links found in the textual descriptions of attacks and the purposes and intents of security patterns. Tondel et al. presented in [102] the combination of three formalisms of security modeling (misuse cases, attack trees and security activity models) in order to give a more complete security modeling

2.4 Security patterns and Security patterns classifications

approach. In their methodology of building attack trees, they linked some activities of attack trees with CAPEC (Common Attack Patterns Enumeration and Classification) [67] attacks; they also connected some activities of SAGs (security activity diagrams) with security patterns. The relationships among security activities and security patterns are manually extracted from documentation and are not explained.

Alvi et al. presented a natural classification scheme for security patterns putting together CAPEC attacks and security patterns for the implementation phase of the software life cycle [4]. They analyzed some security pattern templates available in the literature and proposed a new template composed of the essential elements needed for designers. They manually augmented the CAPEC attack documentation with a section named “*Relevant security patterns*” composed of some patterns [4]. After inspecting the CAPEC base, we observed that this section is seldom available, which limits its use and interest. Uzunov et al. proposed in [104] a taxonomy of security threats and patterns specialized for distributed systems. They proposed a library of threats and their relationships with security patterns in order to reduce the expertise level required for the design of secure applications [104]. They considered that the use of the CAPEC base is cumbersome and assumed that their threat patterns are abstract enough to encompass security problems related to the specific context of distributed systems [4].

Motii et al. proposed in [71] a methodology to guide developers in the selection of security patterns based on threat management and security pattern classification. In the latter, security patterns are classified according to security properties and application domains.

2.4.6 Classification quality criteria

Alvi et al. performed a comparative study of 23 works dealing with the classification of security patterns and depicted 29 classification attributes [5]. They observed that several classifications were built in reference to a unique classification attribute, which appears to be insufficient.

They indeed concluded that the use of multiple attributes enables the pattern selection in a faster and more accurate manner [5].

They explained that a pertinent classification of security patterns has to satisfy a set of properties in order to give a real help to the designers in the selection of the suitable set of security patterns. A set of classification quality criteria was proposed in the literature [37, 40, 5]. In the following we present some of these classification quality criteria:

- **Navigability:** the ability of the classification to provide a guidance to the designers among the collaborations and the relationships among patterns;
- **Acceptability:** the ability of the classification to be applicable to all the security patterns;
- **Comprehensibility:** the capacity of the classification to be accessible to be used for expert and beginner designers;
- **Determinism:** the clearness of the classification definition and building process;
- **Mutual Exclusivity:** the ability of the classification to put each security pattern in one and only one category;
- **Repeatability:** the ability of repeating and reusing the classification over time in addition to the capacity of extending the classification on a bigger number of patterns;
- **Unambiguity:** each category presented in the classification has to be clearly defined;
- **Usefulness:** the ability of the classification to be used in an industrial collaborative software development process.

It is very important to focus on these criteria in order to provide a suitable classification of security patterns. A good classification of security patterns has to be able to present in a comprehensive manner the categories of the patterns in addition to the classification process. In addition it has to be helpful for all the designers, experts and novice, independently from the application environment and its development process organization.

2.5 Security modeling

In collaborative development environment, the security aspects are often loosely documented. The textual nature of the current security documents is repeatedly a source of misunderstanding. Hence, it is important to come up with user-friendly, intuitive and visual approaches in order to analyze qualitatively and quantitatively security concepts [52].

Many graphical security analysis approaches were proposed in the literature. The main objective of these approaches is to facilitate threat assessment and risk management with visual and formal facilities. The first graphical security model was proposed in 1991 by Weiss et al. [109]. This model corresponds to a kind of threat logic tree, which is considered as the origin of most of the attack tree models found in the literature. Kordy et al. counted more than 30 different models in 2013 [52]. They depicted 13 classification aspects (purpose, year, tool availability, etc). We present in Table 2.5 these works through four aspects: 1) whether the model is formally defined, hence, we distinguish three types of approaches, Formal, Semi-Formal and Informal models; 2) whether the model is about modeling attacks or defenses; 3) the capacity of the formalism to express order dependencies; 4) we present the number of papers dealing the model in order to give an idea about its usability.

In this section, we briefly recall some model examples. We present examples of informal, semi-formal and formal models.

2.5.1 Vulnerability Cause Graphs (VCG)

The informal approaches are textually described without providing a mathematical structure. An example of informal approaches is the *Vulnerability cause graphs*. Proposed by Byers et al. in 2006 as a structured method for analyzing and documenting the causes of a vulnerability, *Vulnerability cause graphs* relate vulnerabilities with their causes [17].

A vulnerability cause graph is a directed acyclic graph in which each node has an outgoing directed edge, except for the root that is the vulnerability. All the other nodes are the causes

Table 2.5 Security Models [52]

Name of the model	Attack or defense	Sequential or static	Formalization	Paper count
Attack countermeasure trees [89]	Both	Static	Formal	4
Attack–defense trees [51]	Both	Static	Formal	6
Attack trees [109]	Attack	Static	Formal	>100
Augmented attack trees [80]	Attack	Static	Formal	6
Bayesian attack graphs [55]	Attack	Sequential	Formal	10
Bayesian defense graphs [99]	Both	Sequential	Formal	5
Bayesian networks for security [79]	Attack	Sequential	Formal	14
Parallel model for multi-parameter attack trees [15]	Attack	Static	Formal	5
Protection trees [24]	Defense	Static	Informal	4
Security activity graphs [8]	Both	Static	Semi-Formal	2
Security goal indicator trees [77]	Defense	Sequential	Semi-Formal	3
Security goal models [60]	Both	Sequential	Formal	2
Serial model for multi-parameter attack trees [45]	Attack	Sequential	Formal	3
Vulnerability cause graphs	Attack	Sequential	Informal	4

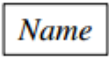
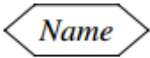
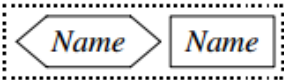
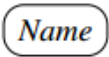
Visual representation	Model element
	Simple node
	Compound node
	Conjunction node
	Exit node

Fig. 2.10 The types of the nodes in a Vulnerability Cause Graphs [17]

of the vulnerability (one vulnerability is presented by each VCG) [17, 52]. As illustrated in Figure 2.10 four types of visual presentations are available: simple nodes address the causes that may lead to the vulnerability, they are the basic elements of a Vulnerability Cause Graph. Compound nodes can be considered as functions or procedures in software development. They enhance the reusability and readability of the models. The Conjunctions are associations among two or more nodes of different types excluding the exit node. The exit node represents the vulnerability studied in the graph.

For instance, the vulnerability CVE-2003-0161, which refers to the incapacity of properly handling the conversions from char and int can be expressed with the vulnerability cause graph of Figure 2.11. The main cause of the vulnerability is the capacity of copying external data to an internal buffer. This can lead to a data copied to an unchecked buffer, then either to an unsafe conditional length check (i.e., the length check is disabled) or that range check will be separated from the copy location, which means that the mechanism checks another input than the input of interest.

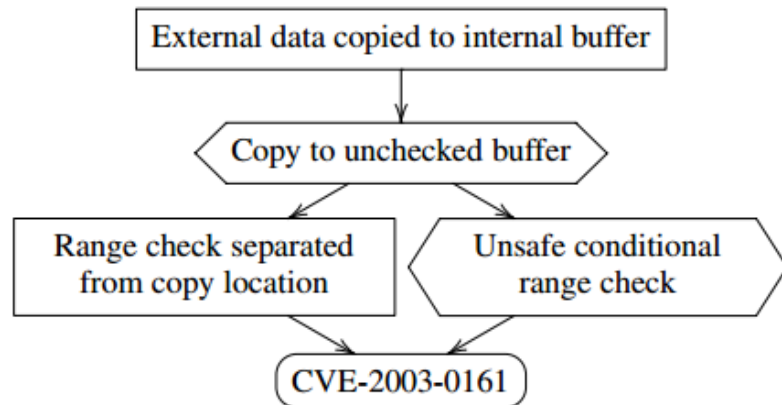


Fig. 2.11 CVE-2003-0161 Vulnerability Cause Graph [17]

The capacity of VCGs to express compound nodes helps model elements from different abstraction layers. In addition, VCG are considered as the starting point of building Security Activity Graphs (SAG) [52].

2.5.2 Security Activity Graphs (SAG)

With semi-formal models, some parts of the model are formally described while the other parts are described textually. Security activity graphs is one of these models.

A Security Activity Graph is a graphical representation of first order predicate calculus proposed by Ardi et al. [8]. A SAG illustrates the possible mitigations of the vulnerability causes presented in a VCG.

As illustrated in Figure 2.12, a Security Activity Graph is a graph whose root is the **Vulnerability**. It is compound of **Activity** nodes connected with gate nodes (Conjunctive, Disjunctive and Split gates) in order to mitigate the Vulnerability.

For instance one of the mitigations of the vulnerability CVE-2003-0161 is about the use of “strcat” primitive in order to properly concatenate strings. As illustrated in Figure 2.13, the use of “strcat” is composed of the conjunction of four sub actions, each one is the disjunction of two activities. These activities are described as bellow:

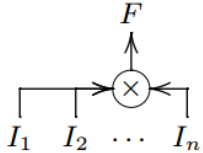
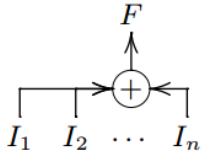
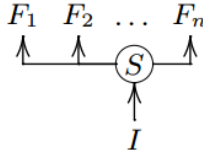
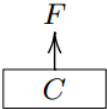
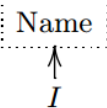
Graphical representation	Function	Logic equivalent
	And gate	$F = I_1 \wedge I_2 \wedge \dots \wedge I_n$
	Or gate	$F = I_1 \vee I_2 \vee \dots \vee I_n$
	Split gate	$F_1 = I, F_2 = I, \dots, F_n = I$
	Activity	$F = C$
	Vulnerability	N/A

Fig. 2.12 The elements of a SAG [8]

- **A:** Use preprocessor to prevent calls to strcat;
- **B:** Use strcat instead of strcat;
- **C:** Use preprocessor to prevent calls to strcat;
- **D:** Use safe string library;
- **E:** Use code inspection to find calls to strcat;
- **F:** Use strcat instead of strcat;

- **G:** Use code inspection to find calls to `strcat`;
- **H:** Use safe string library.

In this way, a Security Activity Graph helps in the analysis and the combination of activities in order to perform actions to countermeasure security problems. While the authors provide a formal description of the elements (nodes and gates) of the model, the approach is not totally formally described. Hence it is considered as a semi-formal model [52].

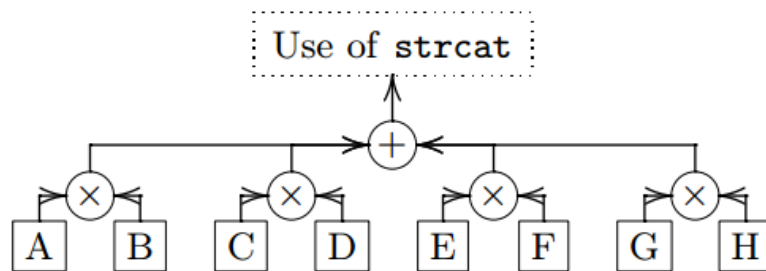


Fig. 2.13 strcat Security Activity Graph [17]

2.5.3 Attack Defense Trees ADTree

The formal models are clearly described with formal frameworks and proper syntaxes. Attack Defense Trees is an example of formal security graphical modeling approaches. Proposed by Kordy et al. in 2010 [50], an Attack Defense Tree (ADTree) is a node labeled and rooted tree allowing the description the attackers and defenders actions in an attack scenario.

As illustrated in Figure 2.14, an ADTree is compound of two types of nodes: the first type is the description of the actions an attacker can make in order to achieve an attack on a system (red circles). The second type of nodes is the expression of the actions a defender can make in order to protect the system (green squares). Each node can have many nodes of the same type, which is a refinement relationship, and one node of different type, which is the countermeasure relationship. In an ADTree, sibling nodes can be related with two types of operators: Conjunctive (AND) and Disjunctive (OR) [51].

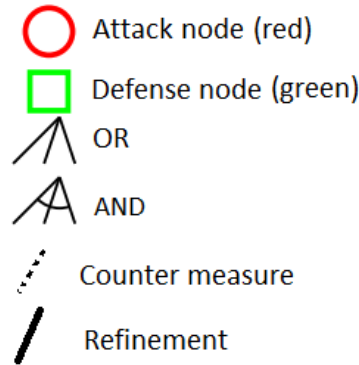


Fig. 2.14 Nodes and operators of an ADTree [51]

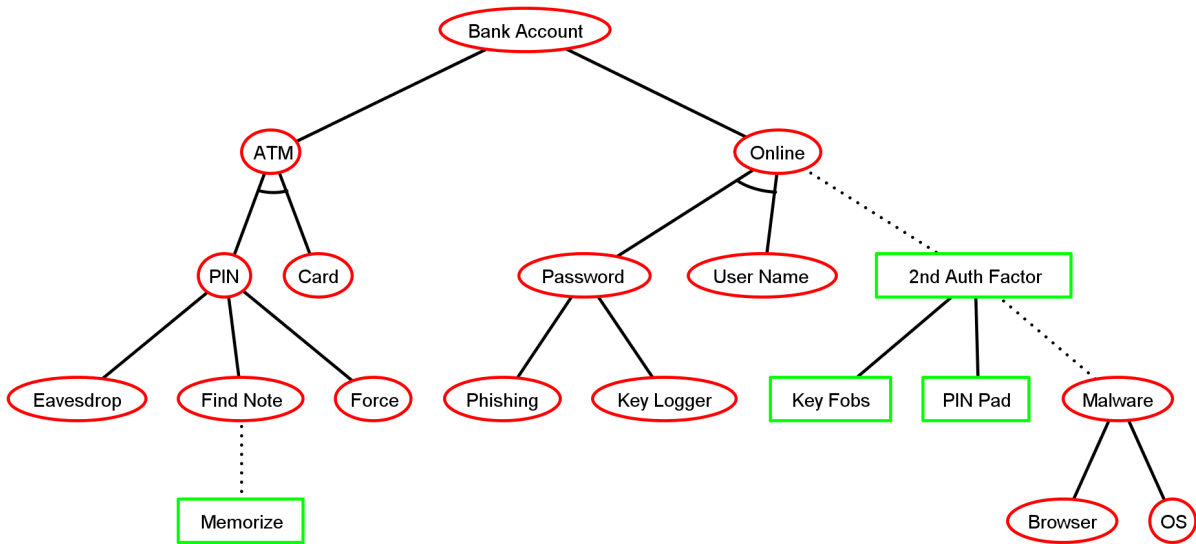


Fig. 2.15 Example of an ADTree [49]

For instance, Figure 2.15 presents an example of an ADTree expressing attack and defense actions in a bank account hack. The root of the tree is the attacker goal, which can be done either online or by means of an Automated Teller Machines (ATM). In order to get the money from an account using an ATM, we have to get the card and the pin, the latter can be got by eavesdropping, forcing or by finding a note. The countermeasure of the last attack is to memorize the pin.

Besides the visual aspect of ADTrees, they are provided with a mathematical framework. An ADTree T is described with an algebra associating attack and defense actions with a set of



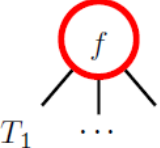
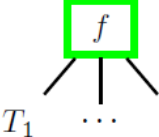

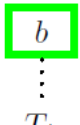
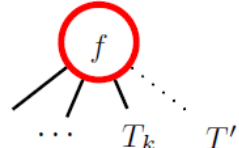
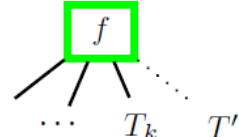
T				
	where $b \in \mathbb{B}^p$	where $b \in \mathbb{B}^o$	where $f \in \{\vee^p, \wedge^p\}, k \geq 1$	where $f \in \{\vee^o, \wedge^o\}, k \geq 1$
$\iota(T)$	b	b	$f(\iota(T_1), \dots, \iota(T_k))$	$f(\iota(T_1), \dots, \iota(T_k))$
T				
	where $b \in \mathbb{B}^p$	where $b \in \mathbb{B}^o$	where $f \in \{\vee^p, \wedge^p\}, k \geq 1$	where $f \in \{\vee^o, \wedge^o\}, k \geq 1$
$\iota(T)$	$c^p(b, \iota(T_1))$	$c^o(b, \iota(T_1))$	$c^p(f(\iota(T_1), \dots, \iota(T_k)), \iota(T'))$	$c^o(f(\iota(T_1), \dots, \iota(T_k)), \iota(T'))$

Fig. 2.16 ADTerms [50]

Attack Defense Terms (ADTerms) noted $\iota(T)$. Figure 2.16 gives the ADTerms of the possible ADTree refinements. ADTerms allow the expression of two types of roles: opponent “o”, and proponent “p” in addition to two types of operators: Conjunction ($\wedge^{o/p}$) and Disjunction ($\vee^{o/p}$) of attack and defense actions (o/p) [50]. The *countermeasure* relationship between two actions a and b is notated $c^{o/p}(a, b)$.

However, ADTree is a static model, which means that it does not allow the expression of order among actions [52]. In order to allow the expression of this order notion, the authors proposed an attack tree extension named SANDTree. In addition to the operators supplied with ADTrees, SANDTrees are provided with a Conjunction operator called the Sequential And (SAND) denoted $\overrightarrow{(\wedge^{o/p})}$ [44, 31].

For instance, in the example of SANDTree presented in Figure 2.17, getting money from a bank account has to be preceded by supplying the pin code, which is preceded by inserting the card. However, SANDTrees does not provide the capacity of modeling defense nodes since it only expresses attack actions.

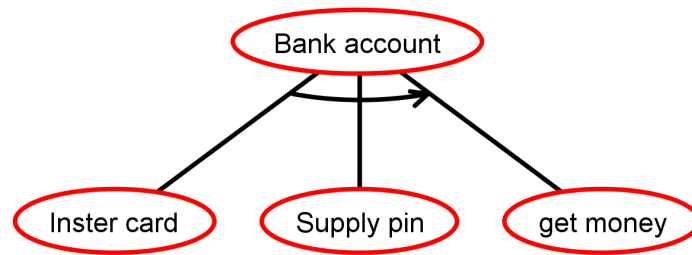


Fig. 2.17 SANDTree Example

The same authors provide a tool called ADTool (The Attack-Defense Tree Tool), which is used to model and analyze attack-defense scenarios represented with ADTrees and SAND trees ¹.

2.6 Chapter conclusions

In this chapter, we reviewed the challenges faced to devise secure applications by designers. We portrayed how security patterns can help novice designers in taking advantage of experts knowledge in order to give solutions to recurrent security problems. Even with the use of security patterns, secure application design is still a difficult task though. The growing number and the abstract textual nature of security patterns are the major challenges to which designers are faced when using security patterns.

Security patterns catalogs, classifications and relations was proposed in order to help designers in security patterns choices. But, many improvements has to be done in order to make these organizations of security patterns more useful. We observed that many of these classifications lack of navigability among patterns , which is an important property defined as the ability to guide the choice of designers among related patterns [5]. More precisely, we noticed that some patterns for the same vulnerability family are not compatible together. As a consequence, a designer may be confused about the appropriate selection of patterns.

¹<http://satoss.uni.lu/members/piotr/adtool/>

We also observed that the main issue of many security patterns classifications lies in the lack of a precise methodology to build the classification. All of them are based upon the interpretation of different documents, which are converted to abstract relationships. The first consequence is that understanding the structure of categories and relationships found in classifications is sometimes tricky. In addition, it becomes very difficult to extend these classifications.

Another issue of using security patterns is related to their abstract nature. Adding a security pattern in an application requires an instantiation step, which is not a trivial step. Hence, it is crucial to guide designers in the instantiation and the verification of security patterns. In the next chapter, we propose a model based checking approach to help designers in assess the instantiation quality of security patterns and their effectiveness against a set of vulnerabilities.

Chapter 3

Assisting Designers in Security Pattern Integration

3.1 Introduction

Despite the large improvements recently made in modeling, coding and testing, software engineering is still regarded as a very complex field. This complexity is due to many aspects, one of these is that software engineering has to address both functional and non-functional requirements. Functional requirements usually express what the application has to do or what the application is about, while non-functional requirements express the other criteria of the application such as its quality, fault tolerance, security, etc. Security is often expressed with more refined properties such as the confidentiality, the integrity or the availability of the application valuable assets.

Security patterns are proposed to help designers devise secure applications. We mentioned in the previous chapter that it is difficult to use security patterns because of their abstract nature, growing number and the complex relationships.

In this chapter we focus on these problems and propose an approach that aims at evaluating the instantiation quality of a security pattern. We first present in Section [3.2](#) the context and

the motivations of the proposed approach. Next, we present in Section 3.3 the prerequisites of the approach, which helps designers check whether a set of security patterns is appropriately instantiated in the UML model of an application, and whether the model, augmented with patterns, is protected against a set of vulnerabilities. We present in Section 3.4 the manual and automated steps of our approach. In Section 3.5, we illustrate a case study based on the Moodle educational platform, more precisely on its Quiz engine ¹. In Section 3.6 we discuss the practical use of the approach and the possible enhancements. We conclude the chapter in Section 3.7.

3.2 Context and Motivations

We presented in Chapter 2 how the security pattern classifications can help software designers in the selection of security patterns. However, the pattern documents do not cover the instantiation of security patterns in a model, since its instantiation may be different with regard to the application. It is very difficult for novice designers to judge whether a security pattern is properly instantiated. Furthermore, pattern classifications do not provide a mean to assess whether the application security is enhanced. Hence, we observed two important points related to the difficulty of instantiating security patterns. The first point is that security patterns documents lack of sturdiness (formal expressions, etc.), which makes the assessment of the instantiation quality of the security patterns difficult. The second point is the difficulty of checking security properties in an application model:

1. Konrad et al. outlined in [48] the difficulties frequently met by designers to clearly understand security patterns (e.g., the lack of comprehensive structure, the difficulty of identifying patterns and to assess their effectiveness). Therefore, they introduced a novel security patterns presentation. The structural part of a security pattern is given with

¹<https://moodle.org/>

UML (Unified Modeling Language) and the behavioral aspect of the patterns is defined with LTL (Linear Temporal Logic) [33]. This way, they formally provided the security constraints supported by a pattern, which allows designer to check, with model checking tools, which constraint is satisfied in the application model. Motii et al. proposed in [73] a methodology for security patterns integration and verification based on UML and OCL, they illustrated their methodology through a VPN security pattern;

2. a plethora of works addressed the verification of security properties on applications models, we summarize some of them below. Security properties can be expressed formally, especially with temporal logics [2, 56, 102]. For instance, Tanvir et al. proposed in [2] a methodology for the verification of the security properties and the impact of the *Role Based Access Control* (RBAC) security pattern in CSCW (Computer Supported Cooperative Work) systems. The global security requirements of the system are expressed as LTL formulas, the system is modeled with the PROMELA language and then requirements are verified with Spin [42] on the PROMELA specification of the application model. In [56], Mallouli et al. combined two modeling formalisms in order to express both the functional and security requirements of a system. The functional behavior is modeled using TEFSM (Timed Extended Finite State Machine) and the security requirements are specified with the Nomad language, which allows the expression of security properties with time considerations. The resulting model allows security properties checking and correction. In [102], Tøndel et al. proposed an approach using misuse cases, attack trees and UML activity graphs to describe how a threat can be mitigated with regard to the behavioral aspect of the application model, expressed with UML activity graphs;
3. Hamid et al. proposed in [39] a framework and a methodology associating model-driven paradigm and a repository of security and dependability patterns to support the design of trusted Resource Constrained Embedded System (RCES) applications. A set of artifacts is generated from each security pattern and then the conformance the security pattern

instantiation is validated with regard to these artifacts. They evaluated the approach and results show that designers are satisfied and the approach paved the way to let them organize themselves their own Pattern-based System Engineering (PBSE) methodology.

Most of these approaches are still difficult to use and error prone for designers because they have to write LTL or OCL formulas. They do not propose a reusable way to cover a big number of patterns and security properties, because security properties are often not generic and they have to be rewritten for each application. The approach we propose in this chapter considers the previous points, but it especially contributes to propose a complete process, which can be followed by designers in order to check the instantiation quality of security patterns and their effectiveness against security problems. In contrast with the approach proposed in [39], we propose to express the security patterns structural and behavioral properties with formal and reusable properties. These are used to check and evaluate the correct security pattern instantiation in the application model. In the same way, we model the vulnerabilities with reusable formal properties and we check the effectiveness of the security patterns in protecting the application model against vulnerabilities.

3.3 Prerequisites

As explained in Chapter 2, the instantiation of a security pattern in an application model is a difficult and error prone task. A pattern can be steadily misunderstood and then misused, which considerably harms its security purposes in the application. As illustrated in Figure 3.1, the approach we present in this section aims at helping designers instantiate a set of security patterns. Then, it checks whether the application model, including a set of security patterns, does not have a set of vulnerabilities.

We assume that the UML model of the application, denoted M , describes both structural and behavioral aspects of the application. The application model structure is described with

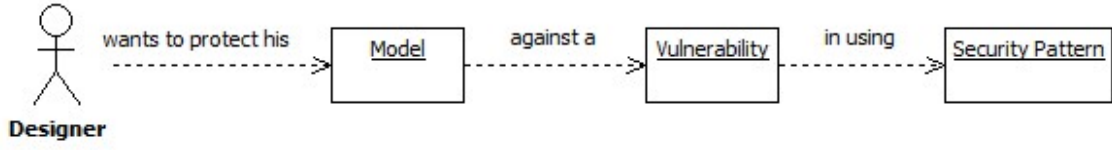


Fig. 3.1 The context of the proposed approach

a class diagram. Its behavior aspects are described with sequence diagrams and statecharts. Ideally, each class of M is completed with a statechart in order to detail its behaviors. In addition, we assume the designer has chosen a set of vulnerabilities $V = \{v_1, \dots, v_l\}$ and a set of security patterns $SP = \{sp_1, \dots, sp_k\}$ that are supposed to be the solutions to the vulnerabilities.

Table 3.1 The symbols of the LTL

Symbol	Signification	Diagram
$\Box \phi$	Globally : means that ϕ is always true	$\phi \xrightarrow{\cdot} \phi \xrightarrow{\cdot} \phi \xrightarrow{\cdot} \phi \xrightarrow{\cdot} \phi$
$\bigcirc \phi$	Next: means that ϕ will be true in the next state	$\xrightarrow{\cdot} \phi \xrightarrow{\cdot} \xrightarrow{\cdot} \xrightarrow{\cdot} \xrightarrow{\cdot}$
$\Diamond \phi$	Eventually: means that ϕ will finally be true	$\xrightarrow{\cdot} \xrightarrow{\cdot} \xrightarrow{\cdot} \phi \xrightarrow{\cdot} \xrightarrow{\cdot} \xrightarrow{\cdot}$
$\psi \mathcal{U} \phi$	Until: ψ is true till the occurrence of ϕ	$\psi \xrightarrow{\cdot} \psi \xrightarrow{\cdot} \psi \xrightarrow{\cdot} \phi \xrightarrow{\cdot} \xrightarrow{\cdot}$

As stated previously, Konrad et al. [48] extended the security pattern template with a set of LTL formulas to formally express the behavioral properties of a security pattern. A behavioral property is considered as a precise sequencing of actions. Modeling the behavioral properties of a security pattern in LTL allows to check whether the model M satisfies the behavior intended by the security pattern.

We recall that the Linear Temporal Logic (LTL) is a first order logic augmented with supplementary temporal operators. In addition to negation “!”, disjunction “ \vee ”, conjunction “ \wedge ”, etc, the LTL is provided with a set of temporal operators. Illustrated in Table 3.1, the temporal operators aim at specifying in which time a propositional variable is true. We present

for each operator (first column), its semantic signification (second column), and a linear diagram addressing how the operator is traduced over the time.

Aderhold et al. [1] presented in 2010 an exemplary set of secure coding guidelines formally described in LTL. They considered that validating input, enforcing authentication, etc, can be written in LTL formulas. We considered this form of secure coding properties to describe vulnerability properties. Therefore, we consider that a vulnerability v_j can also be expressed by a set of LTL properties.

3.4 Assisting designers in the security pattern instantiation

Now we are ready to expose the approach. It aims at helping designers in two ways: it assists designers in the security pattern instantiation by checking whether the model M meets every structural and behavioral property $P_{Sp_i} \in SP$ with SP a set of security patterns. In addition, the approach checks the effectiveness of the security patterns SP against the vulnerability set V by checking whether the model M meets any property P_{v_j} of any vulnerability $v_j \in V$.

The approach is composed of four steps, illustrated in Figure 3.2. In a nutshell, the first one is the instantiation of a security pattern set SP in the UML model M . The next step aims at checking whether the structural properties $P_s(Sp_i)$ of each security pattern Sp_i are satisfied in the model M . A structural instantiation quality coefficient $c_s(Sp_i)$ is calculated. Then, we check whether the behavioral properties $P_b(Sp_i)$ of each security pattern are satisfied and compute a behavioral instantiation quality coefficient $c_b(Sp_i)$. Based on these coefficients, the overall instantiation quality of the security patterns set Q is calculated.

If the patterns are well instantiated, then the next step aims at checking whether the model M has security flows. Hence we check whether the model M meets the properties set P_{v_j} of each vulnerability $v_j \in V$. If M , is still vulnerable, the designer has to review the choice of the security pattern set.

The steps of the approach are detailed bellow:

3.4 Assisting designers in the security pattern instantiation

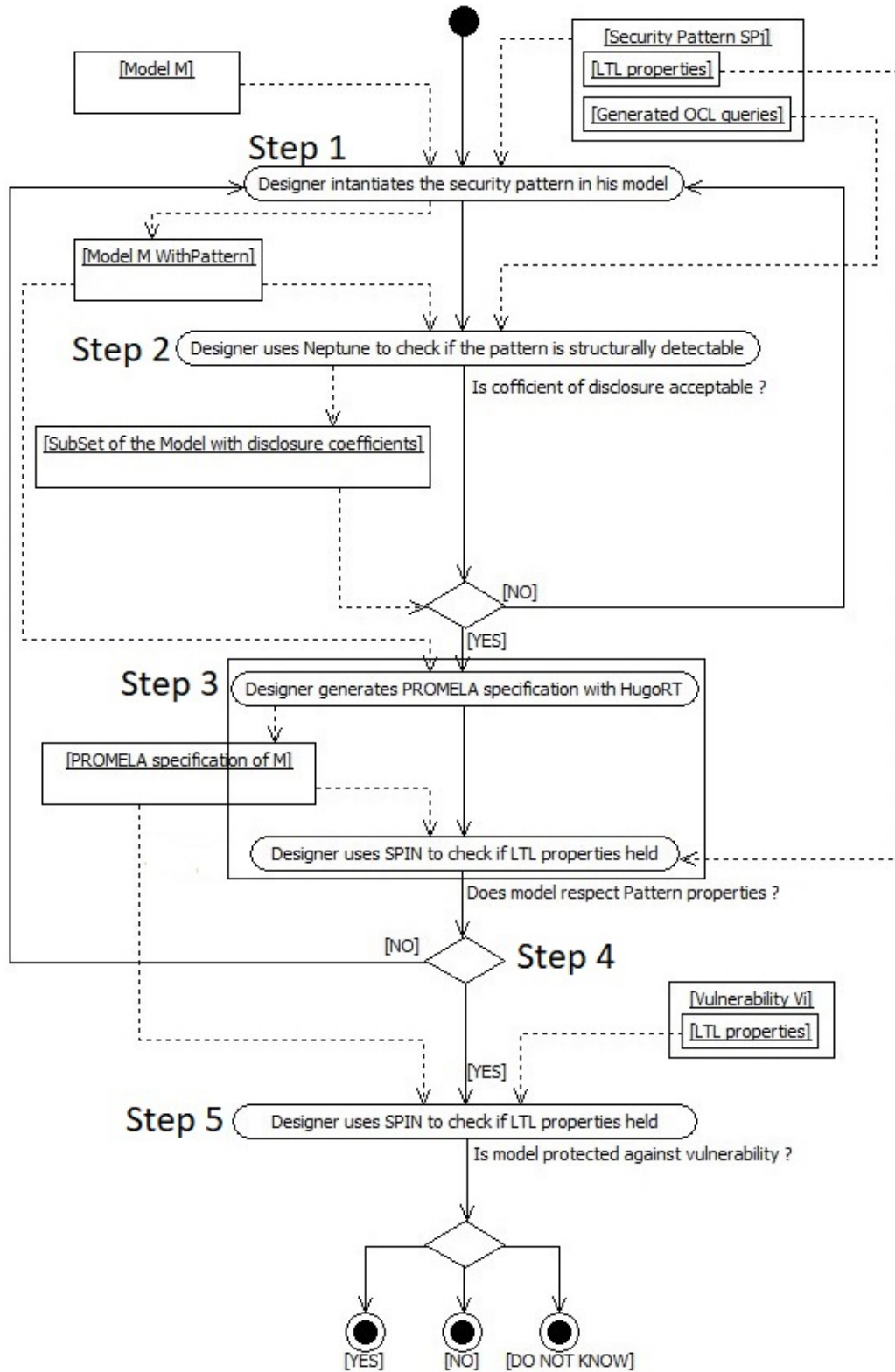


Fig. 3.2 An approach to assist designers to devise more secure applications

Step 1: Security pattern instantiation

Because of the abstract and generic nature of security patterns, the initial step of designers is the instantiation of each security pattern $Sp_i \in SP$ on the application model M . To do so, the security patterns structural and behavioral properties are adapted on the application specific context. Concretely, the designer have to modify the UML diagrams by adding classes (in the class diagram) in order to meet the pattern structure. In addition to interactions (in the sequence diagram) and behaviors (in state charts).

Step 2: Security patterns structural instantiation quality

In this step, we check whether each security pattern structural property holds in the application model M . We use the tool proposed in [14] to extract the structural properties $P_s(Sp_i)$ of each security pattern.

An OCL request is automatically generated for a pattern Sp_i expressing its structural properties. These requests are then executed on the model M using Neptune tool [93]. A list $\langle (p_1, C_1, c_1), \dots, (p_m, C_m, c_m) \rangle$ is obtained, where C_i is a class of M , p_i a class of the security pattern Sp_i and $0 \leq c_i \leq 1$ expresses the proximity of C_i and P_i , denoted proximity coefficient. The more the structure of C_i is close to the one of p_i , the more c_i tends to 1. This way, a proximity coefficient is calculated for each set of classes of M in order to express its proximity to the security pattern Sp_i .

The set of classes $\in M$ having the highest coefficients, noted $c_s(Sp_i)$, is proposed to be the instance of the security pattern Sp_j in M . However, the designer can choose another part of M with a lower coefficient as the instance of the security pattern. We consider that the security pattern Sp_i is structurally well instantiated if the coefficient $c_s(Sp_i)$ tends to 1. In this case the designer can start the next step. Otherwise, the designer has to review the instantiation of the security pattern.

Step 3: Security pattern behavioral instantiation quality

This step aims at checking whether the application model M satisfies the behavioral properties $P_b(Sp_i)$ of each security pattern Sp_i .

These behavioral properties, written in LTL, are extracted from the pattern description. As these properties are generic, the designer has to concretize them with regard to the elements of the model M .

To check the satisfiability of these properties, M is automatically translated into a Promela (PROtocol MEta LAnguage) specification with the HugoRT tool [63]. The Promela specification is achieved by traducing the state diagrams of each class of the model M on a finite state automaton within a scenario presented by a sequence diagram in the model M . The automaton expresses the overall interactions among the objects of the model M in a specific scenario.

With the Spin LTL model checker [42], we check whether the Promela specification of the model M satisfies each behavioral property $p \in P_b(Sp_i)$ of each security pattern Sp_i .

A behavioral instantiation quality coefficient $c_b(Sp_i)$ is computed from the results of Spin for each security pattern Sp_i . We define a function: $mc : P_b(Sp_i) \rightarrow \{0, 1\}$, expressing whether M satisfies the behavioral property p , with $mc(p) = 1$ if $M \models p$ and $mc(p) = 0$ otherwise. Then, $c_b(Sp_i)$ is defined by:

$$0 \leq c_b(Sp_i) = \sum_{p \in P_b(Sp_i)} mc(p) \cdot w_p \leq 1$$

where $mc(p)$ expresses the satisfiability of each behavioral property $p \in P_b(Sp_i)$, and w_p reflects the designer preferences about the behavioral properties of a security pattern Sp_i . The weights of the security pattern properties are defined with a *Simple Additive Weighting* (SAW) technique [113] with $w_p \in \mathbb{R}_0^+$ and $\sum_{p \in P_b(Sp_i)} w_p = 1$. Initially, all w_p are identical and equal $1/\text{card}(P_b(Sp_j))$.

The closer $c_b(Sp_i)$ is to 1, the more M respects the behavioral properties $P_b(Sp_i)$. However, the interest of this coefficient is conditioned by the cardinality of the set $P_b(Sp_i)$.

Step 4: Overall security patterns instantiation quality

In the previous two steps two coefficients are calculated for each security pattern Sp_i : a coefficient $c_s(Sp_i)$, expressing the structural instantiation quality and $c_b(Sp_i)$ the behavioral instantiation quality coefficient.

Based on these two coefficients, the final instantiation quality $q(Sp_i)$ of a security pattern Sp_i is defined as:

$$0 \leq q(Sp_i) = \frac{(C_s(Sp_i) + C_b(Sp_i))}{2} \leq 1$$

- When $q(Sp_i) = 1$, the security pattern Sp_i is structurally and behaviorally well instantiated. However, if the properties set $P_{Sp_i} (P_s(Sp_i) \cup P_b(Sp_i))$ has a low cardinality, it remains difficult to assess the behavioral instantiation quality of Sp_i ;
- If $q(Sp_i)$ is close to 1, and either $C_s(Sp_i)$ or $C_b(Sp_i)$ is low, then the designer has to review the structural or the behavioral instantiation of the security pattern Sp_i ;
- If $q(Sp_i)$ is close to 0, then the designer has to review the overall instantiation of the security pattern.

For the set of security patterns $SP = \{Sp_1, \dots, Sp_k\}$, the overall security pattern set instantiation quality coefficient, denoted Q , is defined as:

$$0 \leq Q = \sum_{i=1}^k q(Sp_i) \cdot w_{Sp_i} \leq 1$$

with $w_{Sp_i} \in \mathbb{R}_0^+$ and $\sum_{Sp_i \in SP} w_{Sp_i} = 1$ reflecting the preferences of the designers about the security patterns.

If Q is equal to 1, then all the security patterns are structurally and behaviorally instantiated in the model M . Otherwise, enhancements can be performed on the instantiation of a pattern Sp_i with regard to the coefficient $C_s(Sp_i)$ or $C_b(Sp_i)$. In addition, as explained by Yskout et al. [116], the instantiation quality of a security pattern Sp_1 can be reduced by another security

pattern Sp_2 since two security patterns Sp_1, Sp_2 might be conflictual. Moreover, Q depends on the size of properties set P_{Sp_i} of a security pattern Sp_i . In other terms, the more a pattern is formulated with generic properties, the more accurate the coefficient will be. A complete documentation of a security pattern Sp_i enhances considerably the precision of $q(Sp_i)$.

Step 5: Application vulnerability assessment

This last step occurs when the designer judges that each security pattern $Sp_i \in SP$ is appropriately instantiated in the model M . This step aims at checking the effectiveness of the set SP of security patterns to protect the model M from every vulnerability $v_i \in V$.

In addition to detect vulnerabilities this step may help designers to:

1. ensure that the chosen security pattern set as the solution to these vulnerabilities. If the model M , after a good instantiation of the security pattern set, is still vulnerable, then the pattern choice has to be reviewed. For instance, the pattern set has to be completed;
2. detect inconsistencies in M , which can be caused by the use of conflicting patterns. As presented in [116], when two security patterns are conflictual, their use in the model M can lead to vulnerabilities.

We assumed that a vulnerability $v_i \in V$ is defined with a set of LTL properties P_{v_i} describing behaviors that should never happen. These properties are generic and reusable. The designer has to instantiate these properties in accordance with the model M . Then, we call the tool Spin to automatically generate a never claim for each LTL property. A *never-claim* is a finite state automaton describing the negation of the vulnerability property. Next, the *never-claim* is compared against the Promela model of M . An execution (counter-example) is returned by Spin if the model M meets the *never-claim*. This means that M is vulnerable to the vulnerability v_i .

Table 3.2 Coefficients values and interpretations

Result	Signification
$q(Sp_i) = 1$	The security pattern Sp_i is well instantiated in the application model. The instantiation quality strongly depends on the size of the properties set $P(Sp_i)$.
$c_s(Sp_i) \rightarrow 1$, $c_b(Sp_i) \rightarrow 1$	Some structural and behavioral instantiation of Sp_i have to be reviewed.
$c_s(Sp_i) \rightarrow 0$, $c_b(Sp_i) \rightarrow 1$	The structural instantiation of Sp_i has to be reviewed, the security pattern structure has not been detected in the application model.
$c_s(Sp_i) \rightarrow 1$, $c_b(Sp_i) \rightarrow 0$	The behavioral instantiation of Sp_i has to be reviewed, security pattern properties have not been satisfied in the application model.
$c_s(Sp_i) \rightarrow 0$, $c_b(Sp_i) \rightarrow 0$	Both behavioral and structural patterns have to be reviewed.
$Q = 1$	All the security patterns of the set SP are correctly instantiated in the application model.
$Q \rightarrow 1$	Some structural or behavioral properties of the security patterns are not satisfied. Inconsistencies can be detected by means of $c_s(Sp_i)$, $c_b(Sp_i)$.
$Q \rightarrow 0$	It strongly recommended to review all the application model. Inconsistencies can be brought by the use of conflictual patterns.
$c_v(v_i) = 0$	The application model does not contain the vulnerability v_i .
$c_v(v_i) \neq 0$	The application model contains the vulnerability v_i the choice of the security patterns has to be reconsidered or some conflictual patterns have been used.

Once all the properties of P_{v_i} have been addressed, a vulnerability coefficient $c_v(v_i)$ is calculated as:

$$0 \leq c_v(v_i) = \frac{\sum_{p \in P_{v_i}} Sat(p)}{card(P_{v_i})} \leq 1$$

with $Sat : P_{v_i} \rightarrow \{0, 1\}$ and $Sat(p) = 1$ if p holds in the application model M , and 0 otherwise. When $c_v(v_i)$ is equal to 0 then we consider that the application model M does not contain the vulnerability v_i . Otherwise, the application model M has to be reconsidered.

The counter-examples given by Spin are presented as trails in a scenario. Presented graphically, these trails can be used by the designer to backtrack the source of the problem in the application behavior. An example of trail will be given in the next section.

Table 3.2 summarizes the different kinds of results that can be obtained. Each one is related to an action that the designer should do.

3.5 Case study

In this section we present a case study, which illustrates the steps defined previously. We consider the security pattern “*Intercepting validator*” and the vulnerability “*CWE-89 SQL Injection*”. We have taken as example the UML model of the application Moodle Quiz Engine (MQE) (part of the Moodle educational platform)². The structure of MQE, as illustrated in Figure 3.3, is composed of a client interface (*attempt.php*), which requests the quiz engine to get the *question_behavior*. The *question_behavior* class gathers the set of questions and interactions (the mechanisms dealing with a user choices and answers).

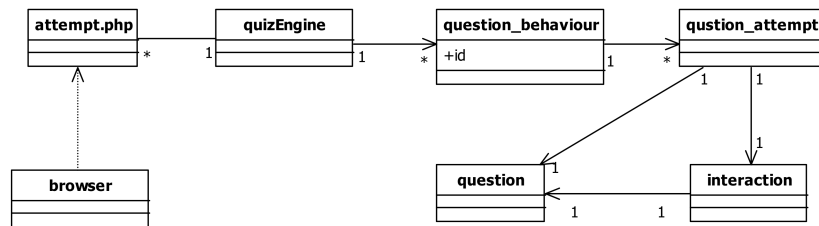


Fig. 3.3 Moodle Quiz Engine Classe Diagram

Figure 3.4 illustrates the sequence diagram of the scenario dealing with the retrieval of the questions. The client accesses his questionnaire *question_behavior* with a request of the form “GET /moodle/quiz/attempt.php?id=123”. With this request, the student is identified with the variable “id=123” and a questionnaire (*question_behavior*) is created by the *quizEngine*. Then, the questions and their correspondent interactions are retrieved from the database and returned to the user in order to start the exam. This model exhibits several security flows. For instance, it allows code injections.

²<https://moodle.org/>

Indeed, the MQE documentation does not express the need to an input validation logic. Therefore, the request “GET /moodle/quiz/attempt.php?id=123” can be manually modified with malicious values of *id* (e.g. an SQL command) in order to alter the database.

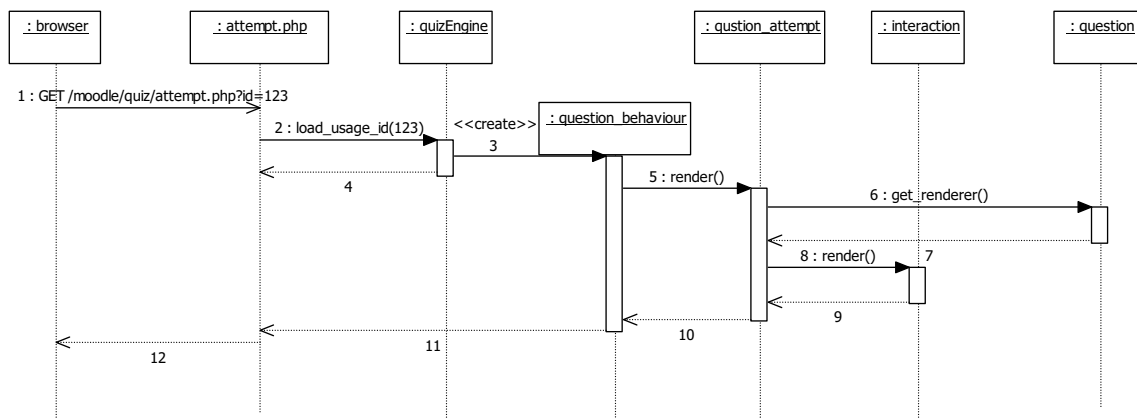


Fig. 3.4 Moodle Quiz Engine Sequence Diagram

An application allowing this type of interactions is vulnerable to the weakness “*CWE-89 SQL Injection*”. This weakness is targeted by well known SQL Injection attacks. Since 2004, 17 SQL injection vulnerabilities was published for Moodle educational platform [68]. Recently, in 2017, the vulnerability “*CVE-2017-2641*”³ was published. It describes that SQL injections can occur via user preferences in the versions 2.x and 3.x of Moodle. In addition, the vulnerability “*CVE-2015-2273*”⁴ describes that in the quiz module of many Moodle versions, identity usurpation is possible and allow students to craft identities in order to interact with the quiz module.

In the literature, “*Intercepting Validator*” is often considered as the solution to the “*CWE-89*” weakness [4, 110, 36, 117] as it supplies the application with a centralized validation mechanism. The interest of this validation mechanism is that it protects the application from mis-formed inputs by checking each input coming to the application.

³<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-2641>

⁴<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-2273>

In this section, we illustrate the proposed approach with this pattern. We start by instantiating Intercepting Validator in the MQE model. We check the instantiation quality of the security pattern in the context of the application. Then, we check whether the application model is still vulnerable to “CWE-89”.

Step 1: Security pattern instantiation

We firstly contextualized the structure of the security pattern, presented in Figure 2.6, on the class diagram of Figure 3.3. The inputs are supplied through *attempt.php*, and used as parameter by *quizEngine* without verification of its content. An “Intercepting Validator” has to be instantiated between the Client (*attempt.php*) and the target of the request (*quizEngine*). The resulting class diagram is given in Figure 3.5. Classes in white are provided from the initial class diagram of MQE (Figure 3.3) and the yellow classes come from the security pattern “Intercepting Validator” instantiated in the application model in order to validate each input providing from the client before being used in the *quizEngine*.

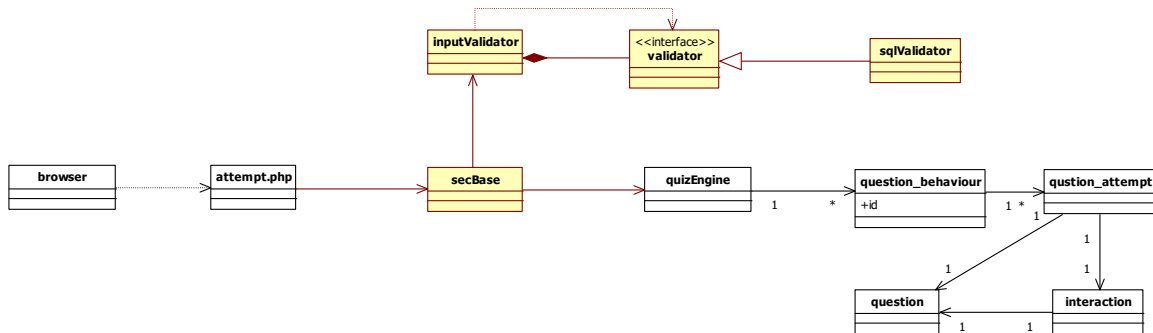


Fig. 3.5 Structural instantiation of intercepting validator

Then, the security pattern sequence diagram of Figure 2.7, is instantiated in the sequence diagram of MQE of Figure 3.4. Figure 3.6 depicts the UML sequence diagram obtained after this instantiation.

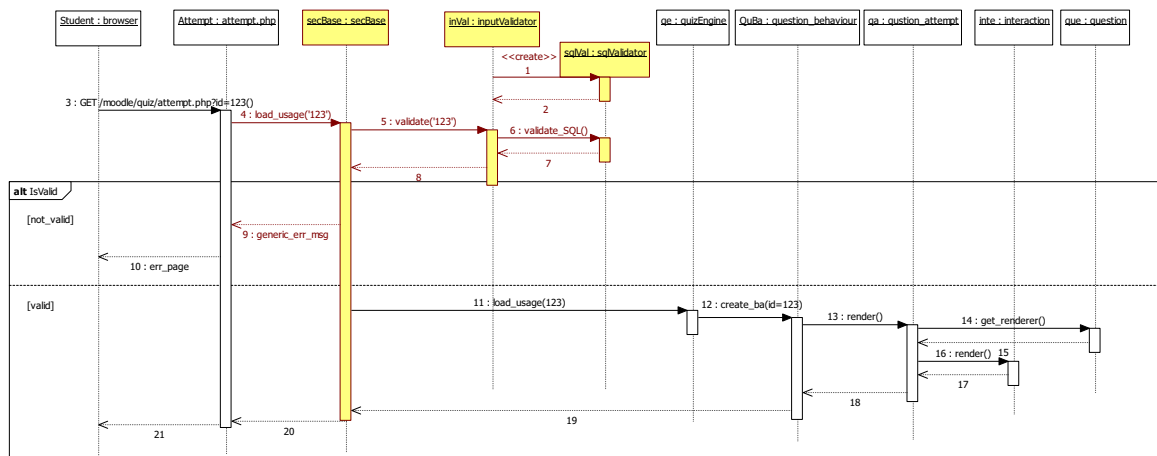


Fig. 3.6 Behavioral instantiation of intercepting validator

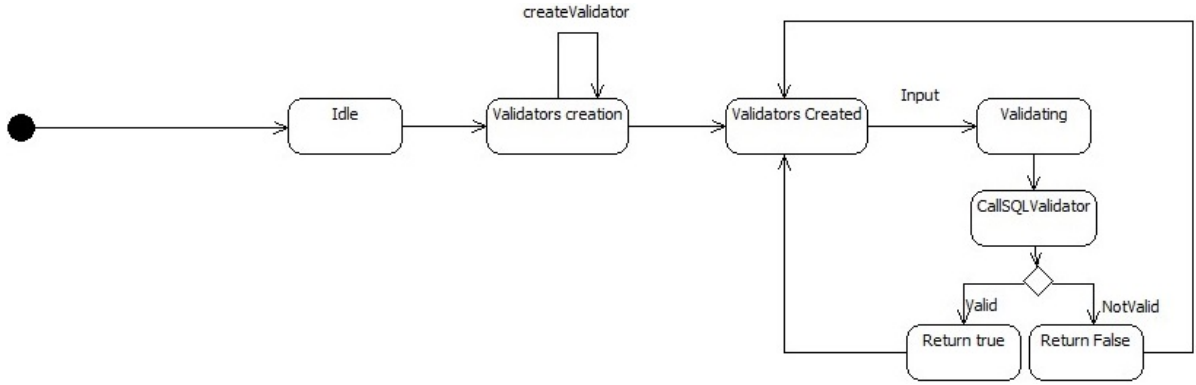
The *id* of the user is now intercepted by the class *secure_base_action*, which calls *inputValidator*. When the client request is validated, it is later processed in the application. Otherwise, an error message is returned to the user [100].

In addition, all the classes have been completed with state-charts in order to express their behaviors. For instance, the behavior of the class *inputValidator* is given in Figure 3.7. This statechart gives some details about the states of the class. It has two activities referring to the creation and the communication with the *sqlValidator*. The statechart expresses that *inputValidator* creates the validator then it waits for inputs. When an input is forwarded to *inputValidator*, the input is forwarded to *sqlValidator*, then the validation verdict is returned to *secBase*.

Now, the security pattern instantiation quality assessment can be calculated in two parts.

Step 2: Security pattern structural instantiation quality

In this step, we check whether the structural properties are satisfied in the model *M*, (Figures 3.5 and 3.6). As stated in the previous section, we use the methodology proposed by Bouhours et al. [14] in order to generate a set of OCL requests from the class diagram of the security pattern “*Intercepting Validator*”. The Neptune tool executes these requests with regard to the

Fig. 3.7 *inputValidator* Statechart

class diagram of the application model M , illustrated in Figure 3.5. The tool returns a set of tuples $\langle (p_1, C_1, c_1), \dots, (p_m, C_m, c_m) \rangle$, with p_i a class of the security pattern, C_i is class of the model M and c_i is a structural proximity coefficient. In our example, the security pattern structure is detected on the model M with $c_s(\text{InterceptingValidator}) = 1$, illustrated in yellow on the Figure 3.5. This detection is precise because of the simplicity of our model. In more complex diagrams, two or more instances of the security pattern might be detected. In this case, the designer has to choose the appropriate one.

Step 3: Security patterns behavioral instantiation quality

This step evaluates the behavioral instantiation quality of “*Intercepting Validator*”. The behavioral properties of the security pattern, presented in Chapter 3, Table 2.4, are modeled with LTL formulas.

For instance, the property “*a validation logic for every data-type used in the application*” means that for each data-type used in the application, a corresponding validator has to be created before any validation of inputs. This behavioral property can be written as :

$$p_2 : \square (Clientinput(Data) \rightarrow !Validate(data))$$

$$\mathcal{U} createValidator(Data.type))$$

Table 3.3 Behavioral properties of *Intercepting Validator*

P	Generic LTL property	instantiated LTL property
p_1	$\Box(\text{ClientInput}(\text{Data}) \rightarrow \neg \text{CallTarget}(\text{Data}) \mathcal{U} \text{Validate}(\text{data}))$	$\Box(\text{attempt.inState}(\text{input}) \rightarrow (\neg \text{quizEngine.inState}(\text{loadUsage}) \mathcal{U} \text{secBase.inState}(\text{valid})))$
p_2	$\Box(\text{ClientInput}(\text{Data}) \rightarrow \neg \text{Validate}(\text{Data}) \mathcal{U} \text{createValidator}(\text{data}:\text{type}))$	$\Box(\text{attempt.inState}(\text{input}) \rightarrow (\neg \text{secBase.inState}(\text{WaitingValidation}) \mathcal{U} \text{inVal.inState}(\text{validatorsCreated})))$
p_3	$\Box(\text{inputValidator.isUnique})$	$\Box(\text{secBase.isUnique} \wedge \text{inVal.isUnique})$
p_4	$\Box(\text{clientInput}(\text{data}) \wedge \neg \text{ServerValidate}(\text{data}) \wedge \Diamond \text{ServerValidate}(\text{data})) \rightarrow (\neg \text{returnGeneric}(\text{message}) \mathcal{U} \text{ServerValidate}(\text{data}))$	$\Box((\text{attempt.inState}(\text{input}) \wedge \neg \text{secBase.inState}(\text{nomvalid})) \wedge \Diamond \text{secBase.inState}(\text{nomvalid})) \rightarrow \neg \text{attempt.inState}(\text{err_page}) \mathcal{U} \text{secBase.inState}(\text{nomvalid}))$

which literally means that globally (\Box), for each client input, the data should not be validated until (\mathcal{U}) the creation of a validator. This formula is abstract and requires an instantiation step before checking its satisfiability in the application model M of Figure 3.6. This means that the generic terms have to be replaced by some concrete terms of M .

With regard to the behavioral properties of M , the *Clientinput* is replaced with the state *input* of the class *attempt.php* and *Validate(data)* corresponds to the state *WaitingValidation* of the class *secureBaseAction*.

When all the terms of the formula p_2 are substituted with the events of the model M , then the formula is instantiated. Otherwise, if there is no concrete events in the application model to express the generic items of the formula, then the application model has to be completed.

The formula p'_2 , given bellow, is the result of the instantiation of p_2 with respect to M .

$$p'_2 : \Box(\text{attempt.inState}(\text{input}) \rightarrow \neg \text{secBase.inState}(\text{WaitingVal}) \mathcal{U} \text{inVal.inState}(\text{valCreated}))$$

Table 3.3 gives the generic behavioral properties of the security pattern “*Intercepting Validator*” and the instantiated properties with regard to model M Figure 3.6.

Because of the semi formal nature of the UML diagrams, checking the LTL behavioral properties of the security pattern in the model M is not possible. Hence, we convert the model M into PROtocol MEta LAnguage (Promela) specification [12] by using the tool HugoRt tool.

With the model-checker Spin [42], each instantiated formula is automatically traduced into a *claim*, which corresponds to an automaton expressing the formula. Then, each *claim* is checked in the Promela specification of the application model M . If a property is not satisfied in the application, Spin returns a counter-example. The counter-example is given as a trail of the actions leading to the unsatisfiability of the behavioral property. In our example, all the behavioral properties are satisfied, which means that $c_b(\textit{InterceptingValidator}) = (1/4) + (1/4) + (1/4) + (1/4) = 1$, if we consider that all the properties have the same weight w_i . The coefficient c_b shows that the security pattern “*Intercepting Validator*” is behaviorally well instantiated in the application model M .

Step 4: Overall security pattern instantiation quality

In the two last steps, two quality coefficients was calculated, ($c_s(\textit{InterceptingValidator}) = 1$) and ($c_b(\textit{InterceptingValidator}) = 1$). The instantiation quality of the security pattern is ($q(\textit{InterceptingValidator}) = 1$), which means that the security pattern *Intercepting Validator* is structurally and behaviorally well instantiated in the application model.

In case $q(\textit{InterceptingValidator})$ tends to 0, the structural or the behavioral instantiation of the security pattern should be reviewed with regard to the values of ($c_s(\textit{InterceptingValidator})$) or ($c_b(\textit{InterceptingValidator})$).

Step 5: Application vulnerability assessment

Once the security pattern “*Intercepting Validator*” has been instantiated correctly in the application model M , it remains to check its effectiveness to protect M against the vulnerability “*CWE-89 SQL Injection*”.

The documentation of this vulnerability [69] shows that an application is vulnerable to SQL Injection attacks if it meets the properties:

1. v_1 : Inputs are not validated;
2. v_2 : Bad Input validation;
3. v_3 : Bad privilege management;
4. v_4 : Information disclosure in error messages.

For instance, the property v_1 expresses that, with no input validation, SQL Injections are possible to occur. This property can be written in LTL as :

$$v_1 : \Box(\text{clientInput}(\text{data}) \rightarrow \Diamond \text{invokeTarget}(\text{data}))$$

which means that each client input provided to the target component is directly processed without a prior verification. As previously, these formula have to be instantiated with regard to M . The instantiation of v_1 is :

$$v'_1 : \Box(\text{attempt.inState}(\text{input}) \rightarrow \Diamond \text{quizEngine.inState}(\text{loadUsage}))$$

where *attempt.php* is the client Input event and the generic term *invokeTarget* is replaced by *quizEngine* in the state *loadUsage*, because at this event the input is processed in the class *quizEngine*.

Table 3.4, gives the generic properties of the vulnerability and the instantiated properties with regard to the model M .

We may denote that the application model does not contain the required events to substitute all the terms of a generic LTL property. Indeed, the property v_3 in Table 3.4 cannot be instantiated in M because there is no states or events in the application model M dealing with identities and rights (*client.right()*). The application model should be improved to address

Table 3.4 CWE-89 properties

V	Generic property	Instantiated property	Sat
v_1	$\Box(\text{clientInput}(\text{data}) \rightarrow \Diamond \text{invokeTarget}(\text{data}))$	$\Box(\text{attempt.inState}(\text{input}) \rightarrow \Diamond \text{quizEngine.inState}(\text{loadUsage}))$	0
v_2	$\Box(\text{clientInput}(\text{data}) \rightarrow \Box(\neg \text{Valid}(\text{data}) \rightarrow \Diamond \text{invokeTarget}(\text{data})))$	$\Box(\text{attempt.inState}(\text{input}) \rightarrow \Box(\neg \text{secBase.inState}(\text{nonvalid}) \rightarrow \Diamond(\text{quizEngine.inState}(\text{loadUsage}))))$	0
v_3	$\Box(\text{clientInput}(\text{data}) \wedge \text{client.right}(\text{Min}) \rightarrow \Diamond \text{client.right}(\text{Max}))$?	?
v_4	$\Box(\neg \text{valid}(\text{data}) \rightarrow \Diamond(\neg \text{genMessage}))$	$\Box(\text{secBase.inState}(\text{nonvalid}) \rightarrow \Diamond(\neg \text{attempt.inState}(\text{err_page})))$	1

these elements. In our case, the designer has to search a complementary security pattern allowing the management of rights and privileges. For instance, the pattern *Least Privileges* meets these needs.

Then, each instantiated property is automatically traduced into a *never-claim*, with Spin, which is the negation of the property.

For instance, the buchi automaton of the never-claim corresponding to the negation of the formula v'_1 is illustrated in Figure 3.8. It models a behavior that should never occur, otherwise, the application is vulnerable. We call once more Spin to check if the never-claims hold in the Promela specification of M .

When Spin builds an execution that satisfies the vulnerability property in the Promela, it detects that the specification is vulnerable. In this case, $\text{Sat}(v_i) = 1$ and the execution is returned by Spin. The counter-example has the form of a trail, which retraces the steps of the application behavior leading to the vulnerability occurrence. Otherwise, $\text{Sat}(v_i) = 0$ and the vulnerability property is not satisfied.

Table 3.4 shows the results with our example. The two properties v'_1 and v'_2 was not detected in the model M . The property v'_3 cannot be instantiated in the application model because of the absence of the rights and roles notions in the application model. So, v_3 cannot be used to check whether the application satisfies the least privilege property. The property v'_4 was

[116]. In addition, the application has to be able to manage rights using the pattern “*least privilege*”, “*Trust partitioning*”, “*Authorization Enforcer*”, etc.

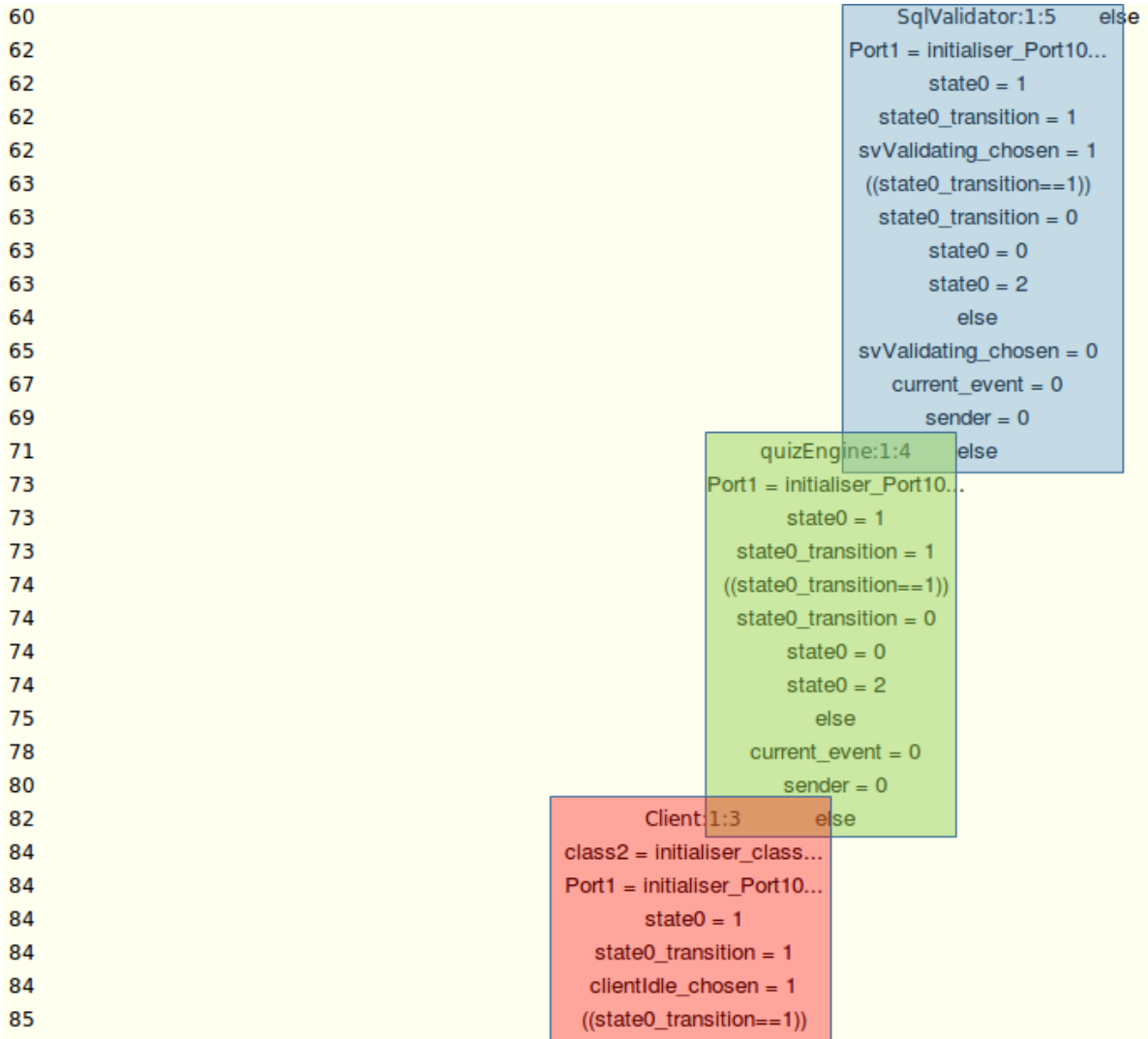


Fig. 3.9 A part of v_4 counter example trail

3.6 Discussion

In this chapter, we proposed a model-oriented approach, composed of manual and automated steps, to help designers secure an application model against a set of vulnerabilities using a set of security patterns. The proposed approach is based upon the formalization of the

security patterns and vulnerability properties with LTL formula. This can be perceived as a supplementary level of difficulty. Yet, this work trend the bond of using formal languages or methods with patterns, in order to automate some steps of the approach. We believe this kind of approach is realistic, seeing the growing number of works pointing to the formal verification of protocols and application models in order to detect security problems e.g., [9, 20, 23].

We considered properties to model vulnerabilities and patterns. These are generic and reusable. The use of these properties requires an instantiation phase though, which consists in substituting the events presented in the properties with the corresponding events of the application model. This can be considered as a disadvantage of the approach. In this area, some works are based on “*text-mining techniques*” for inferring LTL properties [54]. They introduced the tool TEXADA⁵, which dynamically mines LTL properties from application activity traces in order to investigate its behavior.

In [19] a language framework is proposed to help in write behavioral software properties with the use of a set of reusable patterns of LTL properties. Alavi et al. also presented in⁶ a set of reusable LTL patterns. Illustrated in Figure 3.10, these reusable LTL formulas are distributed over two dimensions, Scopes and Behaviors:

1. **Scopes:** illustrated in Figure 3.10b, they express in which part of the application model the property is verified (in the global application behavior, before/after an event or between two events);
2. **Behavioral patterns:** illustrated in Figure 3.10a, they express the “*kinds*” of application behaviors. They are divided into two classes:
 - (a) **Occurrence patterns:** they express the occurrence of an event or a state in the application behavior;

⁵<https://bitbucket.org/bestchai/texada/overview>

⁶<http://patterns.projects.cs.ksu.edu/documentation/patterns/ltl.shtml>

- (b) **Order patterns:** they express the order among events or states in an application behavior.

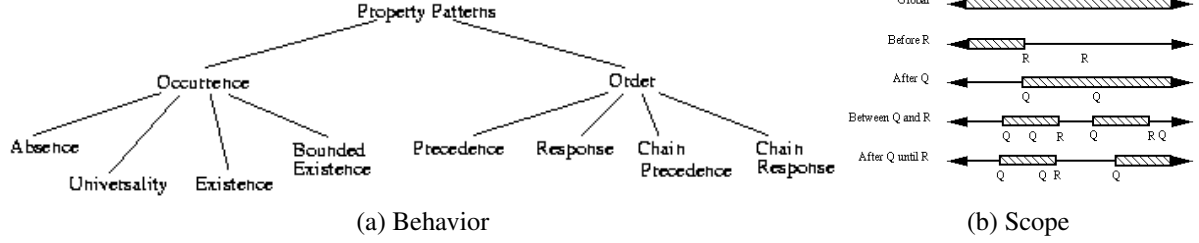


Fig. 3.10 LTL patterns

For instance, checking the cause effect property in an application behavior corresponds to the junction of the behavioral pattern Response with the scope Globally. As illustrated in Figure 3.11 (surrounded with a red square), a cause effect relation between two events (S, P) globally in the application behavior is written as:

$$F : \Box(P \longrightarrow \Diamond S)$$

which means that an occurrence of P is always followed with an occurrence of S . If we take back as example the formula v_1 in Table 3.4, it expresses a cause effect relationship between the *clientInput* event and the invocation event (*invokeTarget*). We search to assess whether the invocation of the target always **responds** to the receipt of a client input.

$$v_1 : \Box(\text{clientInput}(\text{data}) \rightarrow \Diamond \text{invokeTarget}(\text{data}))$$

clientInput(data) substitutes the cause P and *invokeTarget(data)* substitutes the effect S . This way, with 5 scopes and 8 behavioral patterns, a set of 40 reusable LTL properties is proposed. In addition, these LTL properties can be combined to address more complex ones. We believe that, besides a good documentation of security patterns and vulnerabilities, writing LTL properties can be facilitated with the use of this LTL language framework.

Response Property Pattern	
Intent	
To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to .	
Example Mappings	
S responds to P :	
Globally	$[] \langle P \rightarrow \langle \rangle S \rangle$
(*) Before R	$\langle \rangle R \rightarrow (P \rightarrow (! R \cup (S \ \& \ ! R))) \cup R$
After Q	$[] \langle Q \rightarrow [] \langle P \rightarrow \langle \rangle S \rangle \rangle$
(*) Between Q and R	$[] \langle (Q \ \& \ ! R \ \& \ \langle \rangle R) \rightarrow (P \rightarrow (! R \cup (S \ \& \ ! R))) \cup R \rangle$
(*) After Q until R	$[] \langle Q \ \& \ ! R \rightarrow ((P \rightarrow (! R \cup (S \ \& \ ! R))) \cup R) \rangle$
Examples and Known Uses	
Response properties occur quite commonly in specifications of concurrent systems. Perhaps the most common example is in describing a requirement that a resource must be granted after it is requested.	
Relationships	
Note that a Response property is like a converse of a Precedence property. Precedence says that some cause precedes each effect, and Response says that some effect follows each cause. They are not equivalent, because a Response allows effects to occur without causes (Precedence similarly allows causes to occur without subsequent effects).	
Note that this pattern does not require that each occurrence of a cause will have its own occurrence of an effect.	
This is an Order pattern.	

Fig. 3.11 “Response” LTL pattern

3.7 Chapter Conclusions

We proposed in this chapter a method to help designers devise secure applications. The method evaluates the instantiation of a security patterns set in a UML model of an application. In addition, it checks whether the application model does not include a set of vulnerabilities, supposed cured by the security patterns.

We proposed to express security patterns and vulnerability properties with OCL and generic LTL formulas. The security pattern properties express the structural and behavioral properties. Vulnerability properties express the behavioral properties that application model should not

hold. Thus, the generic nature of the proposed LTL properties makes them reusable regardless the application context.

We observed, through a case study, that it is not trivial to choose the appropriate security patterns to protect an application against a vulnerability set. Security patterns can have similar objectives and some patterns can depend on other ones. Some patterns cannot be used together, otherwise it will results in security inconsistencies in the application model.

We conclude here that it sounds interesting to organize security patterns in order to help designers in the security patterns choice. This is actually the topic of the next chapter. It proposes three methods to classify security patterns with regard to weaknesses, attacks and security principles.

The works presented in this chapter have been published in [\[81, 86, 82\]](#).

Chapter 4

Security analysis with patterns and pattern classifications

4.1 Introduction

As stated in the previous chapter, security patterns provide designers with reusable solutions to recurrent security problems. The choice and the use of security patterns is difficult because of their abstract nature and growing number. The choice of the appropriate security patterns for a security problem is difficult for two main reasons:

1. the growing number of security patterns and their relationships implies that designers can be lost while reading their descriptions;
2. the association of security patterns with other security notions (e.g., vulnerabilities, weaknesses, attacks, etc) is a tricky problem, because security patterns are very abstract by nature. Hence, including security patterns in the software life-cycle and approaches is still tedious at the moment.

In order to ease the selection of security patterns, many works classify them according to various criteria [4, 96, 53, 36, 6, 119, 71]. We showed in Section 2.4 the difficulties and the

solutions employed to organize them. We also gave an insight on the different classification schemes and the quality factors assessing the usefulness of a security pattern classification for fast and accurate choice.

In this chapter, we propose a new approach to generate pattern classifications by using the concept of data acquisition with respect to classification quality criteria. This chapter is structured as follows. In Section 4.2 we recall some related works and we give the motivations and the main contributions of this chapter. In Section 4.3, we present the data sources considered by our approach and the architecture of the databases used to gather security notions, e.g., security patterns, weaknesses, attacks and security principles. In Section 4.4, we detail the set of manual and automated steps of three methods for later generate classifications. We exhibit in Section 4.5 three classifications extracted from the resulting databases. In addition, we give the generation of graphical representations of these classifications given under SAGs and ADTrees. In Section 4.6, we discuss on the quality criteria of the obtained classifications along with some statistical results extracted from the databases. Finally, we conclude this chapter in Section 4.7.

4.2 Context and motivations

Several security pattern catalogs are available in the literature [35, 87, 116], gathering a total of 176 patterns. These catalogs make difficult the choice of the appropriate patterns for overcoming a security problem. Many classifications were proposed in the literature to ease the pattern choice with regard to a given context [6, 4, 5, 110, 102, 4, 104, 72].

After reviewing these classifications, we observed these all are manually conceived by interpreting different documents to find abstract relationships. Justifying these classifications or updating them is often unfeasible.

Since most of the security pattern classifications do not define the steps to build the classification, they are not *deterministic* [5] and they are difficult to reproduce and update. In addition, the relationships among security patterns are often not given making classifications

not *navigable* [5], yet we noticed that some patterns are compatible together and that others are conflicting. As a consequence, a designer may be still confused about the pattern choice. We tackle these issues in this chapter and we present three methods to generate three different pattern classifications associating security patterns, security principles, attack patterns and weaknesses. These steps break out security notions and data into sub-properties that can be linked with more evidence (i.e., less ambiguity). Security data are transformed and integrated (data acquisition) into data-stores.

The data-stores allow an automated extraction of security pattern classifications. The main contributions of this chapter are:

- we describe the data-stores architectures required to integrate security data to generate security pattern classifications;
- we list the steps required to fulfill the data-stores with security data publicly available. Security patterns, attacks and weaknesses are split into properties that are associated with manual or automatic steps. We avoid the direct textual comparison of security patterns with weaknesses or attacks. In addition, this helps in the reproduction of the data integration and the classifications extraction;
- we automatically generate three classifications: the first one organizes patterns with regard to weaknesses and shows the pattern combination that can cure a weakness. The two other ones organize patterns with regard to attacks. They give the pattern combinations that counter a given attack. The two classifications are obtained with different kinds of data and give different viewpoints. We generate the graphical representations of these classifications by means of SAGs and ADTrees, in order to help the analysis of the relationships between a weakness or an attack with a set of security patterns. We believe these models help designers in understanding the security problems and their related solutions regardless their security skills. This way, we increase the *Comprehensibility* of the generated classifications;

- as a proof of concept, we applied these methods on the Web applications context. The data-stores gather 215 CAPEC attacks, 136 CWE weaknesses, 66 security principles and 26 security patterns. Databases and classifications extraction tools are publicly available in [\[84\]](#);

4.3 Data-stores architectures

In this section, we firstly present the security data sources we studied to define the meta-models of the data-stores required for the data acquisition. Then, we expose the meta-models used by the pattern classification methods.

4.3.1 Data sources

After reviewing the literature and the publicly sources available available on Internet, we choose to focus on four security notions : security patterns, weaknesses, attack patterns and security principles. These give detailed information about the security problems, methods, solutions, etc. that a developer should consider.

1. Mitre corporation gathers most of the security problems from many points of view. They provide both researchers and security professionals with a plethora of security databases. These databases are periodically updated by the community for a better comprehensibility of vulnerabilities, attacks, weaknesses, etc. Among these public databases, we chose to focus on CAPEC database [\[67\]](#) and CWE database [\[69\]](#) the weaknesses (CWE) and attack patterns (CAPEC) databases. We chose to focus on these bases to extract information about security weaknesses and attacks because these appear to be the most complete bases composed of the largest number of attacks and weaknesses explained in detail (mitigations, steps, techniques, risks, countermeasures, etc.);

2. we considered a set of security patterns catalogs [116, 35, 100, 87]. The completeness of the documentations (including forces, consequences, related patterns, etc.), the number of patterns and the relations among them are the main criteria of security pattern catalogs choice. These are not exposed in all of these sources, hence, we mostly worked with detailed patterns repositories [116, 100];
3. we also explored the security principles and good practices proposed in many works [106, 90, 62, 94]. We reviewed these works and we extracted a set of security principles having different abstraction levels.

4.3.2 Data-stores Meta-models

We initially studied the meta-models proposed in [103, 65]. Figure 4.1 illustrates an insight on the relations among countermeasures and the security artifacts presented in Section 2.2.

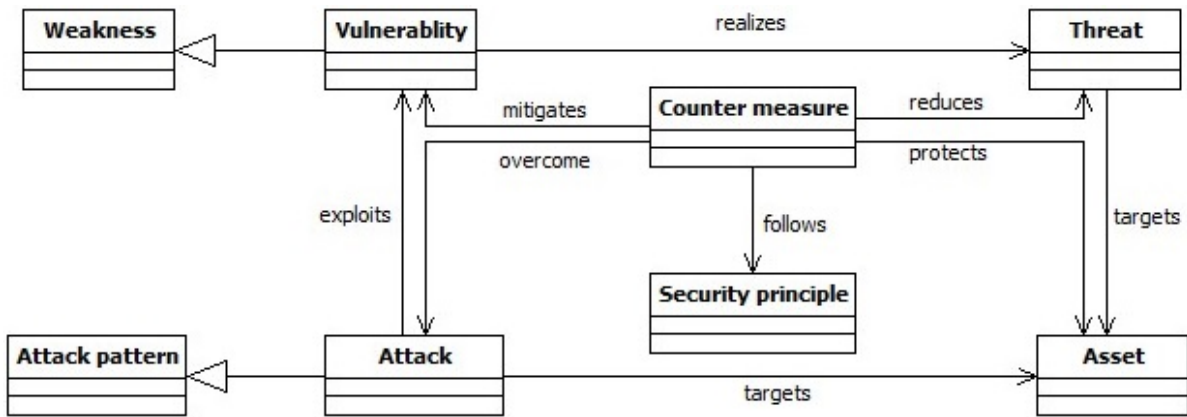


Fig. 4.1 Relations among security artifacts

- Countermeasures are the solutions implemented in order to reduce the impact of a threat on an asset and protect the application from being vulnerable to attacks. Countermeasures follow more abstract security notions expressed by security principles;

Security analysis with patterns and pattern classifications

- Vulnerabilities are the specialization of weaknesses since a weakness (CWE_i) can be implemented by different vulnerabilities (CVE_j) regarding products, technologies, versions, etc;
- An attack is the specialization of an attack pattern in a specific technology, product, version, etc.

After reviewing the meta-model of Figure 4.1 and the literature, we observed the following relations among attacks, weaknesses, patterns and security principles.

1. **Security patterns** are defined with a set of **strong points**, which are extracted from the sections forces and consequences of pattern documents. In addition, a security pattern is related to other patterns. The binary **relation** between two security patterns (a, b) is defined as follows:
 - **a Depends on b** : the use of b is required when using a ;
 - **a Benefits of b** : the pattern a effectiveness is enhanced when using b ;
 - **a is Alternative to b** : a and b have similitudes and a can be substituted by b ;
 - **a Impairs b** : the pattern a effectiveness is harmed when using b ;
 - **a Conflicts b** : inconsistencies can be resulted from using a and b together;
2. the **Weaknesse** documents provided in the CWE database show that a weakness can be cured by a set of **mitigations**. A mitigation is exposed in the CWE database with an id, a mitigation strategy and a textual description;
3. **Attacks** are organized hierarchically from the more abstract attacks to the more concrete ones. An attack can target a set of weaknesses, the relationships among attacks and weaknesses are already provided in the databases CAPEC and CWE. Attacks are split up in the CAPEC data-base into an ordered sequence of **steps**. Each step can be detected,

corrected, or prevented by a set of **countermeasures**, which are presented under the section *Security controls* of each step. In addition, a step is materialized by a set of concrete **techniques** allowing the implementation of the attack step;

4. All these relations breach the general security notions (attacks, weaknesses, etc.) into more concrete properties, which can be associated more precisely. These more precise relations shall help generate precise security pattern classifications. But it still remains to associate the strong points of security patterns with either mitigation cluster or countermeasure clusters.

To do so, we rely on **Security principles** as intermediary. In other words, a **strong point** is related to a **mitigation** or a **countermeasure** if they target the same security notion expressed by a **security principle**.

Mitigations and countermeasures are often more concrete and detailed compared to security principles. Though, we chose here to group these solutions in groups or **Clusters** in order to meet the abstraction level of the security principles. A security principle reflects the security notions of a set of analogous mitigations or countermeasures.

As result, in order to later generate classifications, three meta-models are built (**WSPC**: Weaknesses based Security Patterns Classification, **AWSPC**: Attacks and Weaknesses based Security Patterns Classification and **ASSPC**: Attack Steps based Security Patterns Classification):

1. **WSPC** meta-model (Figure 4.2) organizes **security patterns** with regard to the **weakness** notion. **Security patterns** (respectively **weaknesses**) are interrelated and associated to a set of **strong points** (respectively **mitigations**). **Mitigations** are grouped in **clusters**, and related to **strong points** with regard to in common **security principles**. The latter are hierarchically organized;

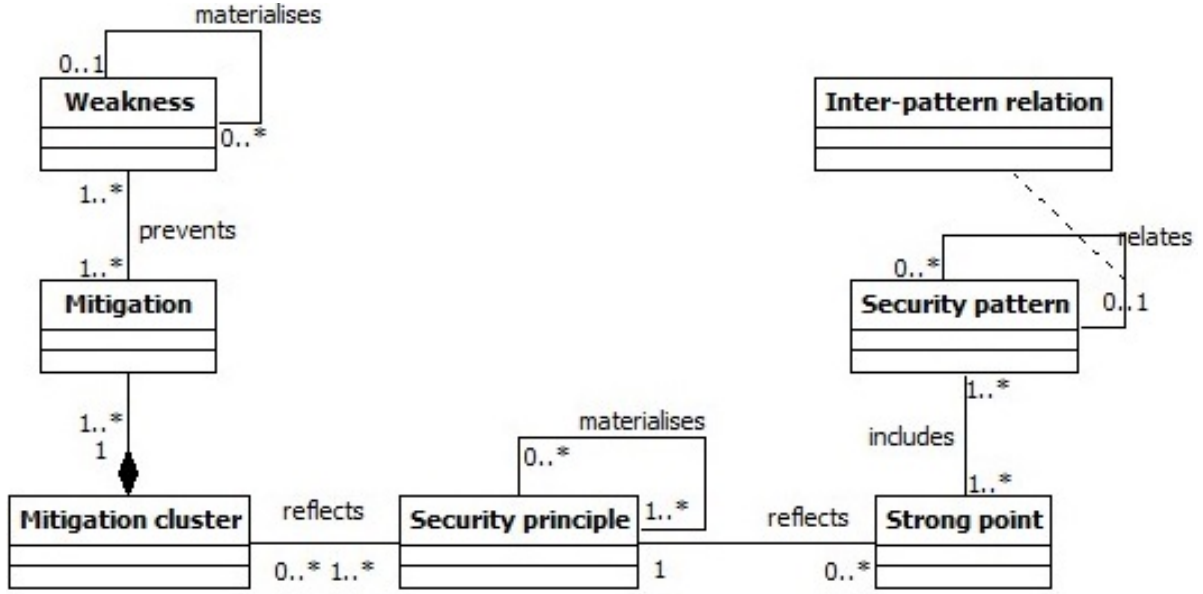


Fig. 4.2 **WSPC**: Weaknesses based Security Patterns Classification

2. **AWSPC** meta-model (Figure 4.3) associates attacks and patterns with regard to weaknesses. It is an extension of **WSPC** with attacks such that an attack targets a set of weaknesses;
3. **ASSPC** meta-model (Figure 4.4) also associates **attacks** and **security patterns** from another point of view. **Attacks** are divided into **steps**, each one is characterized with a set of **techniques** and **countermeasures**. As in **WSPC**, **countermeasures** are grouped into **clusters**, which are associated to **strong points** relying on **security principles**.

4.4 Data integration

In this section, we detail the set of manual and automatic steps used to fulfill the databases of Figures 4.2, 4.3 and 4.4 with elements extracted from the data sources described in Section 4.3. We implemented these steps by means of the Extraction Transformation and Loading (ETL)

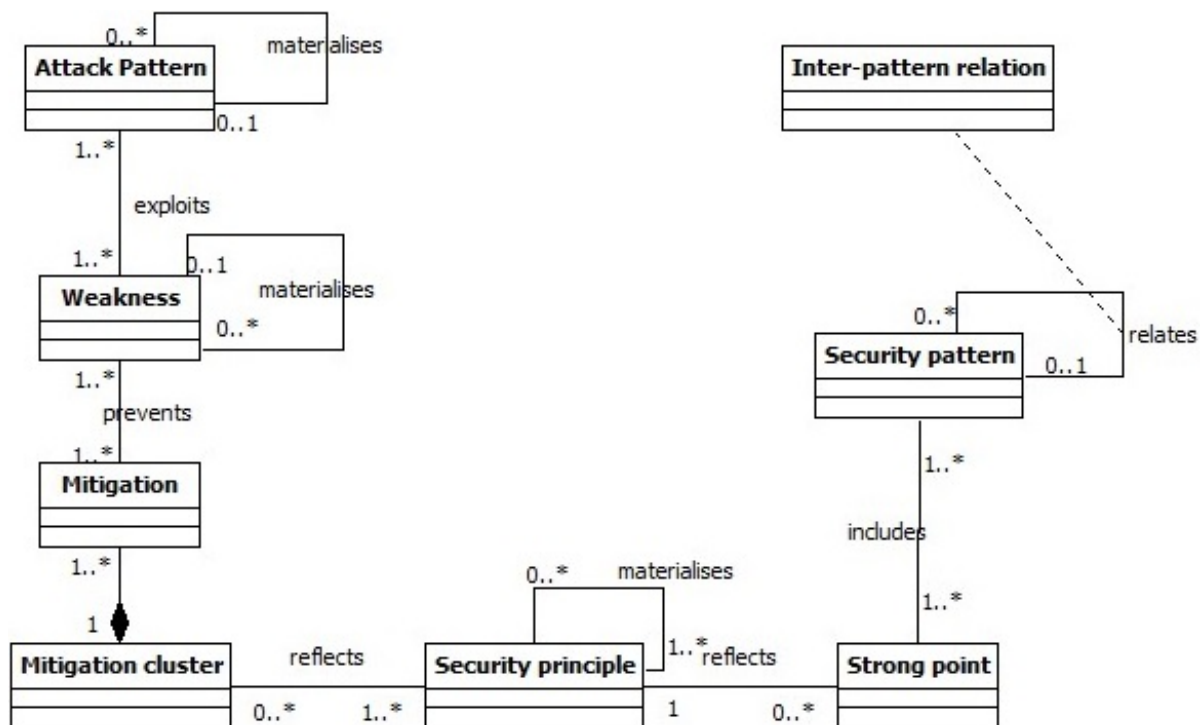


Fig. 4.3 AWSPC: Attacks and Weaknesses based Security Patterns Classification

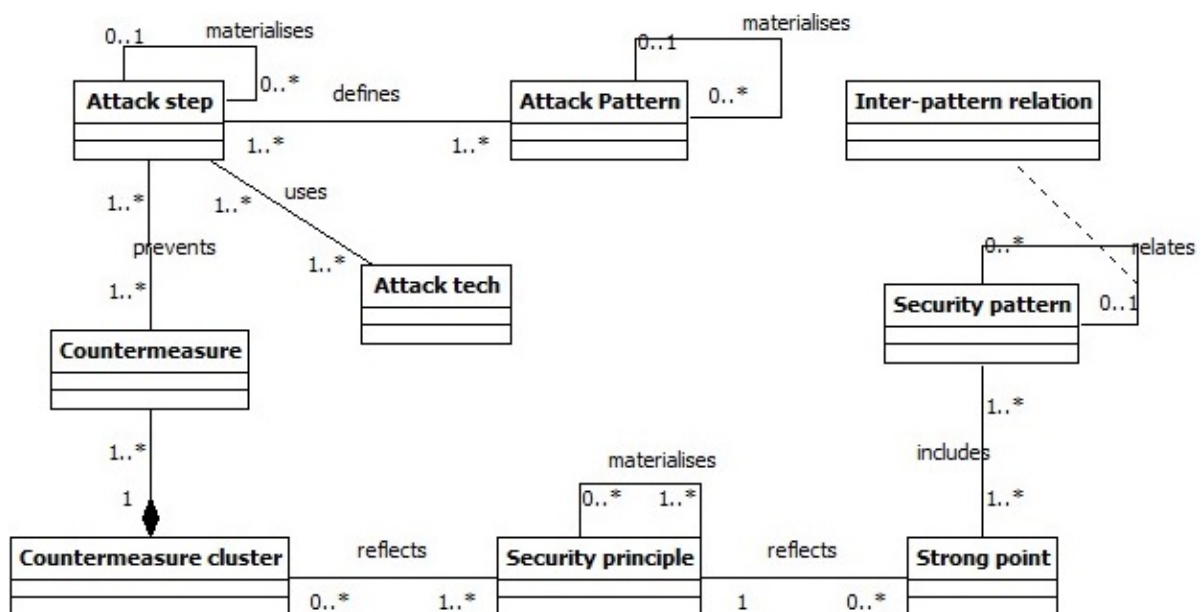


Fig. 4.4 ASSPC: Attack Steps based Security Patterns Classification

tool Talend¹. As a proof of concept, we focused the data-integration on the web application context. The data integration steps can be divided into four phases:

¹<https://www.talend.com/>

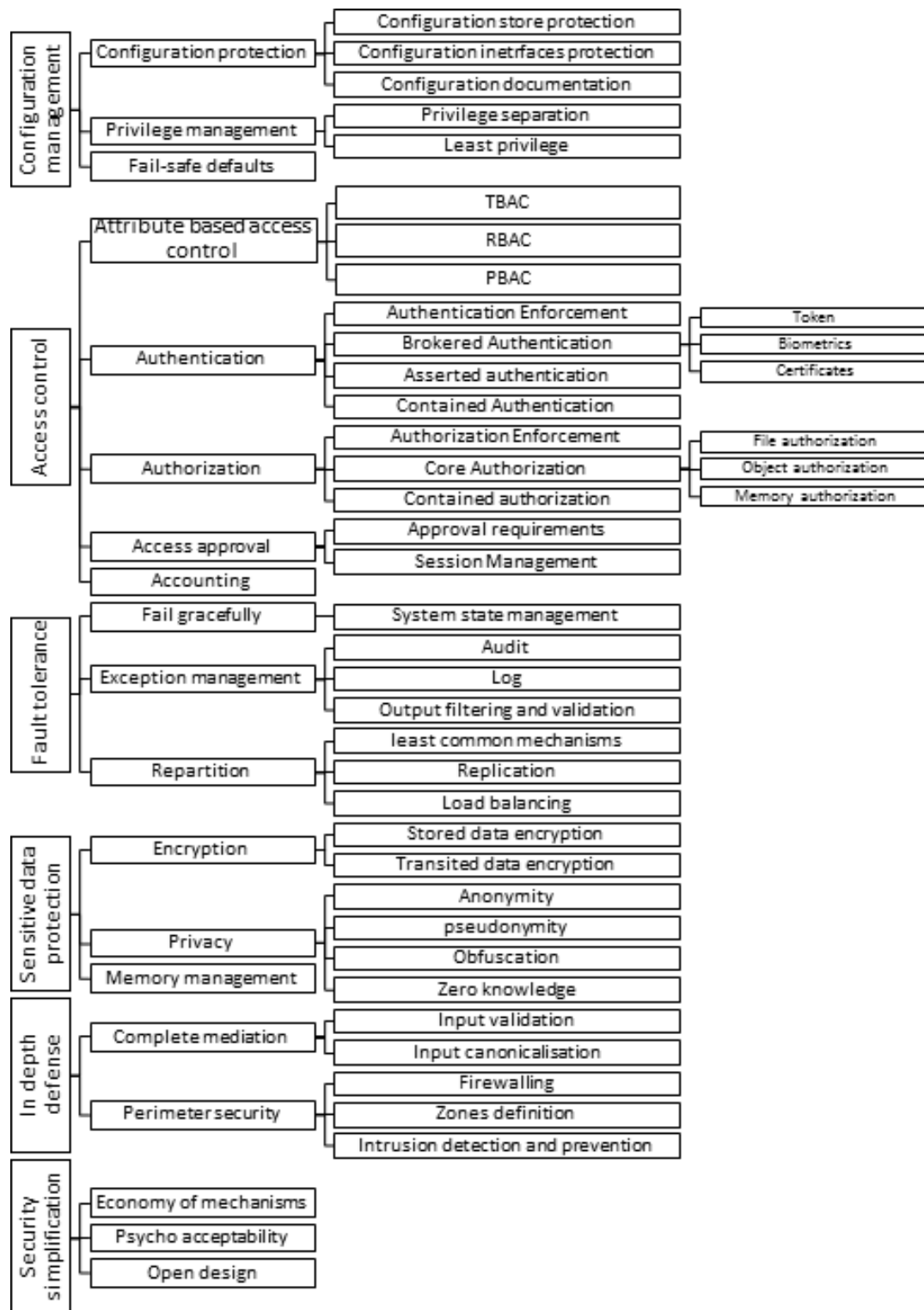


Fig. 4.6 Exemplary hierarchical organization of security principles

Step 2: Security pattern integration

We manually extracted two relations among patterns and strong points from the catalogs [100, 116]:

1. the first one is a many-to-many relation between security patterns and strong points, each pattern being characterized by a set of strong points that can be shared with other patterns. For example, the patterns “Authorization enforcer” and “Container managed security” share the strong point “Providing the application with authorization mechanism”. This relation is established by manually extracting the strong points of each pattern from its textual description (forces and consequences of the pattern);
2. the second relation materializes the inter-pattern relationships, annotated “depend”, “benefit”, “impair” or “alternative” [116]. With P a set of patterns, this relation is defined as a mapping from P^2 to the annotation set $\{\text{“depend”}, \text{“benefit”}, \text{“impair”}, \text{“alternative”}\}$, which provides for every pair of patterns $(p1, p2)$ an annotation about the relationship between $p1$ and $p2$.

We collected 26 patterns and 36 strong points.

Step 3: associating strong points with security principles

As introduced in [108], security patterns are classifiable with regard to security principles. Instead of looking for a direct relation between patterns and security principles, we focus on the strong points, which are more precise properties.

This step establishes a many-to-many relation between strong points and principles. It was manually done since strong points and principles are mostly presented with textual documents. During this step, we observed that the abstraction level of strong points better fit with the most concrete principles exposed in our hierarchical organization of Figure 4.6. But, if a strong point

is related to a principle sp that is not at the lowest level, then we also link the strong point with all the children of sp .

For example, the strong point “Minimize the decoupling between authorization and business logics” of the pattern “Authorization Enforcer” has a security objective also found in the principle “Economy of mechanism”, which is a sub-principle of “Security simplification”.

All these elements are automatically consolidated into an intermediary database denoted IDB_1 storing information about security patterns and security principles.

4.4.2 Integrating security patterns and weaknesses

In this second phase, we integrate security patterns and CWE weaknesses in two steps:

Step 1: Weakness and mitigation extraction

As illustrated in Figure 4.7, we automatically extracted the weaknesses from the CWE database (Version 2.9), and for each weakness, its potential mitigations. These weaknesses are organized in the CWE base into a hierarchy of four levels reflecting their abstraction levels (Category, Class, Base, Variant).

This extraction process is implemented under Talend and the result is stored in the database denoted IDB_2 , which gathers information about 185 CWE weaknesses and 65 mitigations.

For example, the weakness “*CWE-285: Improper Authorization*” is commonly found in Web applications and occurs when an application does not correctly manage authorization checks. When the application does not have a correct authorization mechanism, it exposes several vulnerabilities, e.g., denial of service or arbitrary code execution. This weakness can be fixed by means of several mitigations [69]. We expose some of them here:

- Use of a framework that correctly performs the authorization checks;
- Divide the software into anonymous, normal, privileged, and administrative areas;

Security analysis with patterns and pattern classifications

- Use Role Based Access Control;
- Default deny in access control lists (ACLs);
- Authorization correctly enforced at the server side;
- Restrict access to requests having an active authenticated session.

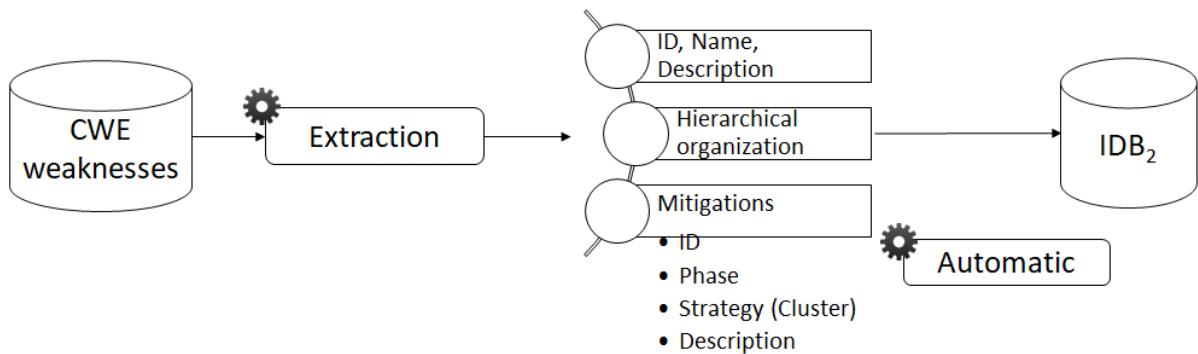


Fig. 4.7 Integrating weaknesses and mitigations

Step 2: Associations between weaknesses and security patterns

This step associates weaknesses and security patterns as follows :

1. mitigations are automatically grouped into clusters to meet the abstraction level of security principles. A set of mitigations are grouped into a cluster if they have the same Strategy (defined in the Section “Mitigation Strategy” of the CWE database);
2. A) in Figure 4.8, with the use of the database *IDB₂* we associate mitigations clusters and security principles This was manually done by interpreting the strategies found in mitigations. Indeed, the meaning of a mitigation strategy is often very close or comprised into some security principle definitions. As a strategy may cover several security principles, we have to consider a many-to-many relation here;

For instance, the weakness “*CWE-285: Improper Authorization*” can be fixed with the mitigation “Divide the software into anonymous, normal, privileged, and administrative areas”, which is related to two security principles:

- Least common mechanisms: so that a failure in an area does not affect another area;
- Privilege separation: so that a privilege given in an area is not valid in the other area.

A spoofed identity does not give the possibility to compromise all the authorization in the application;

Mitigations are usually described with concrete mechanisms given with a low abstraction level. Hence, we observed that mitigations are often associated with the most concrete security principles in reference to the hierarchical organization of Figure 4.6;

3. B) With the use of the databases IDB_1 and IDB_2 , we associate security patterns and mitigations clusters by means of security principles. On the one hand IDB_1 stores the relations among weaknesses, mitigations and security principles. On the other hand, IDB_2 stores the relations among security patterns, strong points and security principles. It is now manifest that the security principle hierarchy becomes the central point that helps match attacks with security patterns. Afterwards, these elements are consolidated under Talend into the database $WSPC$ associating 26 Security patterns, 36 strong points, 185 weaknesses, 65 mitigations and 66 security principles.

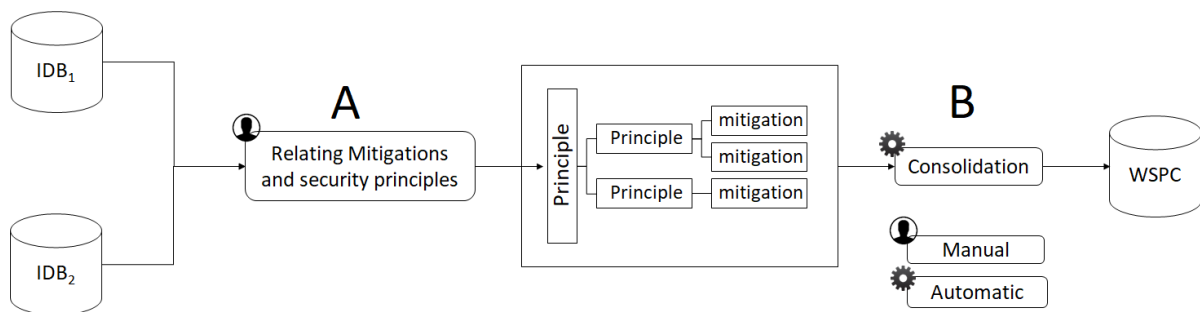


Fig. 4.8 Integrating weaknesses, mitigations and security patterns

If the weakness belongs to the level “category” or “class” then we consider the mitigations related to its children. While if variant, we consider the mitigations given for its alternatives (variants).

4.4.3 Attacks and security patterns integration (v_1)

In this phase, as illustrated in Figures 4.9, 4.10, we extract attack information from the CAPEC database (version 2.8) and link them to security patterns with regard to the weaknesses targeted by each attack and the relations among weaknesses and security patterns previously established in the database *WSPC*.

Step 1: CAPEC attack extraction and organisation

The attack acquisition and integration is illustrated in Figure 4.9. We automatically extracted attacks of the CAPEC base and organized them into a single tree, which describes a hierarchy of attacks from the most abstract to the most concrete ones, so that we can get all the sub-attacks of a given attack. To reach that purpose, we rely on the relationships among attack descriptions found in the CAPEC section called “Related Attack Patterns”. By scrutinizing all the CAPEC documents, it becomes possible to develop a hierarchical tree whose root node is unlabeled and connected to the most abstract attacks of the type “Category”. These nodes are parents of attacks that belong to the type “Meta Attack pattern” and so on. The leaves are the most concrete attacks of the type “Detailed attack pattern”.

The abstraction level of an attack is expressed in the column “Type” (C stands for Category, M for Meta pattern, S for Standard pattern and D for Detailed pattern), the links with other attacks are listed in the column “Related Attack Patterns Nature”.

From the section “Related Weaknesses” of the CAPEC documents, we extracted for each attack the set of weaknesses directly targeted by the attack. Thus, each CAPEC attack is related to CWE weaknesses. These relationships are provided by the CAPEC database and

are extracted automatically. Weaknesses are grouped into two categories, “Targeted” and “Secondary” ranking the impact degree of the attack on a weakness. We only focused on the type “Targeted” even though it could also be relevant to consider both types.

The outcome of this systematic extraction encodes an intermediary database *IDB₃* mapping from 215 attacks to 185 CWE weaknesses. Unsurprisingly, we observed that the attacks having a high level of abstraction (those of Category and Meta Pattern) are not related to any CWE weakness.

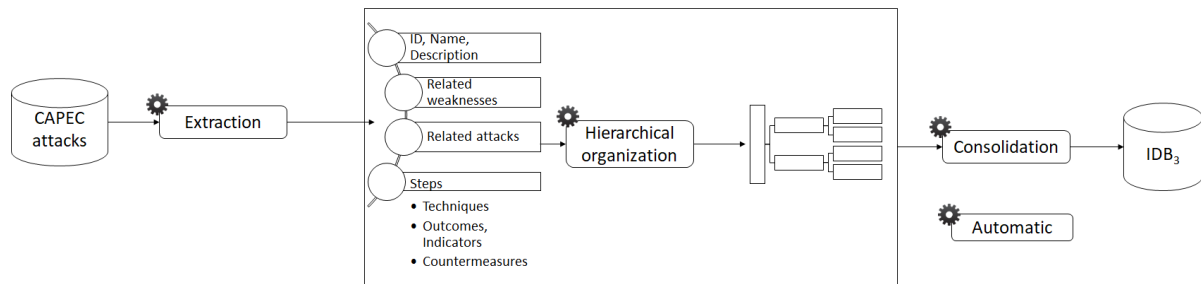


Fig. 4.9 Attacks extraction and hierarchical organisations

Step 2: associations of attacks and security patterns

As illustrated in Figure 4.10, this step consolidates the two databases *IDB₃* and *WSPC*. On the one hand, *WSPC* holds relations among weaknesses and security patterns, and on the other hand *IDB₃* gathers relations among attacks (hierarchically organized) and weaknesses. Hence, we combine and consolidate these two databases in order to dress relations among attacks and security patterns.

This step is performed automatically (implemented under Talend) and provides the database *AWSPC* gathering the CAPEC attacks organized hierarchically in one big tree, the set of CWE weaknesses targeted by each attack and the security patterns related to each CWE weakness. Hence, each attack is now related to a set of weaknesses and each weakness is related to a set of security patterns. We consider that an attack is possible on an application if the application contains at least one of the weaknesses targeted by the attack.

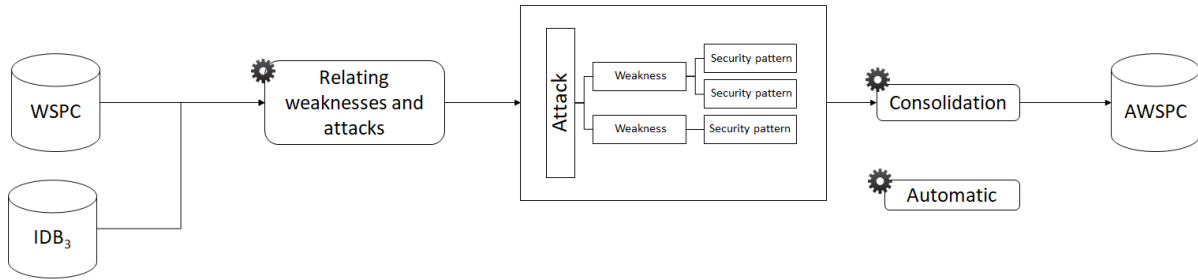


Fig. 4.10 Weaknesses based attacks and security patterns integration

4.4.4 Attacks and security patterns integration (v₂)

In this second approach, we want to link attacks and security patterns with regard to the countermeasures provided in the CAPEC database (version 2.8) and not according to weaknesses. We collected for each attack (from the section “Attack Execution Flow”) its ordered set of steps. Each step may be composed of more concrete sub-steps. Then, for each step, we extracted the corresponding techniques and countermeasures. From the CAPEC base Version 2.8, we extracted these elements automatically under Talend and collected them in the intermediary database *IDB₃*, which gathers 215 attacks, 209 steps, 448 techniques and 217 countermeasures, knowing that attacks can share steps, techniques, etc.

This phase of data integration is illustrated in Figure 4.11 and explained below:

Step 1: Countermeasures hierarchical clustering

The countermeasure number grows quickly while reading the attacks of the CAPEC base. Many of them have a close meaning though, which can be explained by the number of different contributors that added them. These countermeasures can be hence grouped into families to be later associated with security principles (Part A in Figure 4.11).

We semi-automated this process by applying a hierarchical clustering technique of documents. We firstly used the tool KHcoder ² to measure similarities among countermeasure

²<http://khc.sourceforge.net/en/>

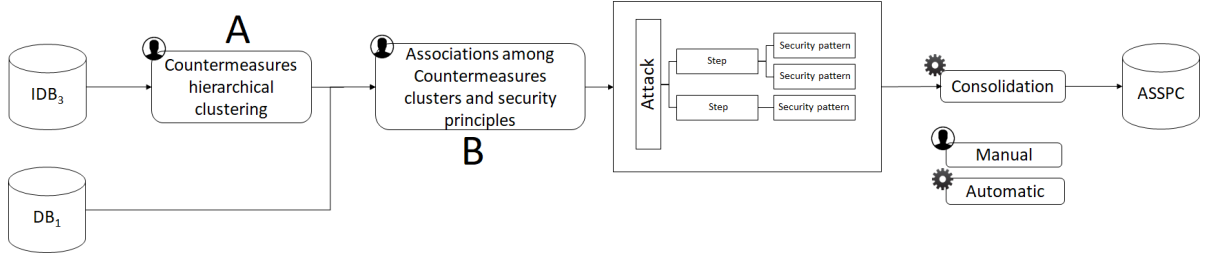


Fig. 4.11 Integrating attacks and security patterns (Second approach)

descriptions. KHcoder is a free tool, referred by numerous works, which performs quantitative content analysis (text mining). We applied KHcoder as follows:

1. the Stanford (Part Of Speech) POS tagger is called to sort the keywords (log, input, credentials, etc.) by their frequencies and types (noun, verb, adverb, etc.). In addition, we defined a set of ignored words (stop words) like *{punctuations, and, therefore, or, etc.}*;
2. from the frequencies, weights are computed and scaled with the Jaccard coefficient to measure a distance among countermeasures. The distance between two countermeasures a, b is defined by: $0 \leq d_{a,b} = q + r / p + q + r \leq 1$ where p is the number of keywords occurring in a and b , q is the number of keywords occurring in a and not in b and r is the number keywords occurring in b and not in a . At the end of this step, a distance matrix is obtained for all the countermeasures. The distance between two countermeasures is minimized when they have more common key words;
3. we used Ward, an agglomerative hierarchical clustering method, to make a hierarchy of countermeasures clusters [111]. Given two clusters $A = \{a_1, a_2, \dots, a_n\}, B = \{b_1, b_2, \dots, b_n\}$, the distance between the clusters A, B is calculated with the formula: $\Delta_{A,B} = (\frac{n_A n_B}{n_A + n_B}) (\frac{1}{n_A + n_B}) \sum_{i=0}^{n_A} \sum_{j=0}^{n_B} d(a_i, b_j)$ where n_A (resp n_B) is the number of the elements in the cluster A (resp B) and $d(a_i, b_j)$ is the Jaccard distance between the elements a_i, b_j of the two clusters A, B [75].

Security analysis with patterns and pattern classifications

We chose to use Ward methodology because this is a supervised technique that has the advantage to construct small clusters that are merged iteratively. This avoids the construction of a too big cluster, which might cover a lot of countermeasures an a lot of different security aspects.

Algorithm 1 Hierarchical clustering

Require: *Jaccard distance matrix;*

Require: *Ward distance matrix;*

repeat

Find the closest pair of clusters (A,B);

Merge them;

Update Ward distance matrix for (A,B);

until *There is one cluster*

Its algorithm is recalled in Algorithm 1. At the beginning, every countermeasure is encompassed into a new cluster. The algorithm merges the pairs of clusters if having the closest distance. The algorithm re-calculates the weight of the new cluster with regard to the Jaccard coefficient and updates the matrix distance. At the end of the process, all the clusters are grouped into one big cluster.

Finally, the number of clusters to select is chosen manually, as it is supervised in the domain of natural languages [118]. The cluster number can be selected with a dendrogram. Figure 4.12 illustrates an example of dendrogram, obtained with 23 countermeasures. At the lowest level, the dendrogram shows all the countermeasures and its top level represents one final cluster. The choice of the number of clusters comes down to draw an horizontal line in the dendrogram and to enumerate the number of cut vertical lines. There are two basic criteria to consider when inserting the line: a low cut is divisive, i.e., it may place two similar countermeasures in different clusters; a high cut is agglomerative, i.e., it may put in the same cluster two unrelated countermeasures. Therefore, in order to get a coherent clustering, the most suitable level has to be chosen after some iterations by checking whether the countermeasures obtained in the clusters refer to the same security principle or set of principles. In the example of Figure 4.12,

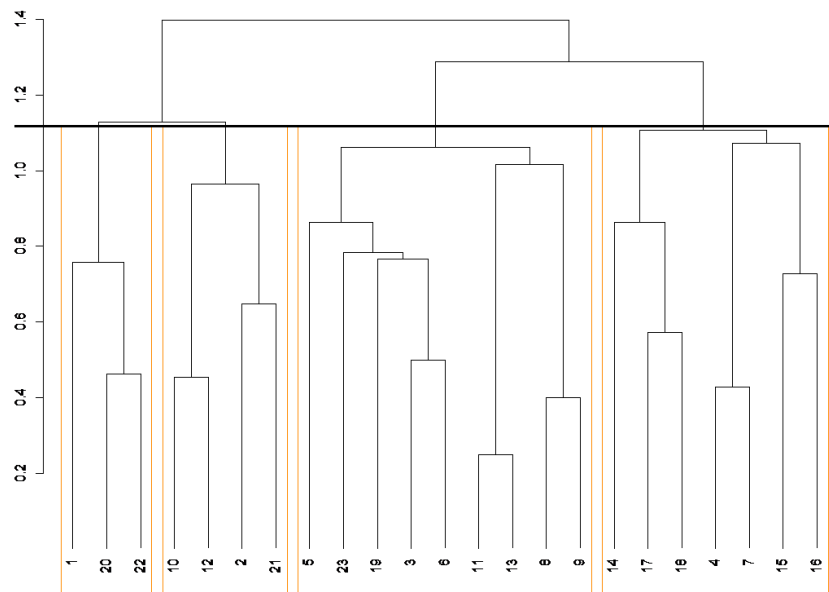


Fig. 4.12 Exemplary dendrogram

we obtained four clusters. From the 217 countermeasures collected in the database *IDB₃* we obtained 21 clusters of countermeasures.

Step 2: Associating security patterns and clusters of countermeasures

We manually established a many-to-many relation between countermeasures clusters (obtained in the previous step) and the hierarchical organization of security principles, illustrated in Figure 4.6 (part B Figure 4.11). The clusters obtained in the previous step include countermeasures sharing the same security aspects, e.g., Input validation, authentication, authorization, etc. Once these aspects are manually deduced, linking clusters and security principles becomes straightforward.

Then, these information are automatically consolidated under Talend, which results in the database *ASSPC* associating 215 attacks organized hierarchically, 217 countermeasures, 21 countermeasures clusters, 36 strong points and 66 principles organized hierarchically.

4.5 Classification extraction and SAG/ADTree generation

We presented in Section 4.4 a set of consecutive steps for populating three data-stores (*WSPC*, *AWSPC* and *ASSPC*). They associate security patterns, weaknesses, attacks and security principles. Based on these databases, we automatically generate three security patterns classifications which are :

- **WSPC**: Weakness oriented Classification of Security Patterns;
- **AWSPC**: Attack and Weakness Classification of Security Patterns;
- **ASSPC**: Attack Steps Classification of Security Patterns;

In addition, we graphically represent these classifications with Security Activity Graphs (SAGs) and Attack Defense Trees (ADTrees). We present how we generate these visual supports for each CWE weakness and CAPEC attack available in the stated databases.

4.5.1 Weakness Security Patterns Classification (WSPC)

The information about security patterns, weaknesses and security principles and the relationships among them are gathered in the database *WSPC*. The later is used to extract a first security pattern classification expressing the combinations of patterns that can cure a given weakness. We automatically extract for every weakness w :

- the information about w (name, identifier, description, etc.);
- the complete hierarchy of security principles Sp related to w , i.e. the arrangements of principles from the most abstract ones to the most concrete principles. The principles of Sp are associated with the weakness according to its potential mitigations given by the CWE database (Section 4.4.1). The latter does not precise if one of the proposed mitigations is sufficient to fix the weakness or if all the mitigations have to be applied.

4.5 Classification extraction and SAG/ADTree generation

Table 4.1 Extraction of the pattern classification for the weakness CWE-285

Weakness ID	Weakness name	Principle Level 1	Principle Level 2	Principle Level 3	Security pattern	relation type	Related security pattern
285	Improper Authorization	access control	attribute based access control	-No Value-	PBAC	-No Value-	-No Value-
				-No Value-	RBAC	-No Value-	-No Value-
			authorization	-No Value-	Authorization Enforcer Container Managed security	alternative alternative	Container Managed security Authorization Enforcer
		configuration management	privileges management	least privilege	Least privilege	-No Value-	-No Value-
				privilege separation	Compartmentalization Trust Partitioning	-No Value- benefits	-No Value- Compartmentalization
				least common mechanism	Compartmentalization	-No Value-	-No Value-
		fault tolerance	repartition		Distributed Responsibility	benefits	Compartmentalization

As a consequence, we suppose that all the security principles have to be considered to overcome the weakness;

- for every principle $sp \in Sp$, the set of patterns P_{sp} , the set of patterns P_{sp_2} not in P_{sp} that have relations with any pattern of P_{sp} , and the nature of these relations defined for couples of patterns by the annotations in $\{\text{"depend"}, \text{"benefit"}, \text{"impair"}, \text{"alternative"}, \text{"conflict"}\}$.

Table 4.1 shows an example of data extraction achieved for the weakness “CWE-285: Improper Authorization”. The tabular provides the *ID* and the name of the weakness (col. 1,2), the security principles and their levels in the tree of Figure 4.6 (col. 3-5), the related security patterns (col. 6) and the relations with patterns (col. 7,8). After applying these steps on all the weaknesses, we obtain the security patterns classification under the form of tabular. This classification is available in [84].

4.5.2 Security Activity Graph (SAGs) generation

At this stage, we think that Comprehensibility, which refers to the ability to use the classification by experts or novices, is not yet totally satisfied. Indeed, the classification is given under a tabular form only, which does not appear to be the most user-friendly way of representation. Hence, we propose here to portray this classification with SAGs, organizing the security principles and patterns related to a weakness. SAGs are semi-automatically drawn by these steps:

1. a weakness w has its own SAG whose root is labeled by the identifier of w . The root node is linked to the most abstract principles found in Sp connected together with the *AND* operator, since we consider that all the security principles have to be considered to remove the weakness. The security principles are themselves connected, from the most to the less abstract ones, by keeping the hierarchical ordering depicted in Figure 4.6;

2. as our classification provides the relations among patterns, we propose, to complete the SAG with new nodes encoding these relations. We replace these relations with logic operators to infer a Boolean expression from all these relations. Given a security principle sp in Sp and a couple of patterns (p_1, p_2) of the set P_{sp} , if we have:
 - $(p_1 \text{ depend } p_2)$ or $(p_1 \text{ benefit } p_2)$, we use the expression $(p_1 \text{ AND } p_2)$;
 - $(p_1 \text{ alternative } p_2)$, we use $(p_1 \text{ OR } p_2)$. The use of these two patterns together increases the complexity of the system, but is not problematic;
 - $(p_1 \text{ impair } p_2)$, we use $(p_1 \text{ XOR } p_2)$ since the presence of p_2 can decrease the efficiency of p_1 ;
 - $(p_1 \text{ conflict } p_2)$, we use the expression $(p_1 \text{ XOR } p_2)$ meaning that only one of the pattern should be used;
 - p_1 having no relation with any pattern $p_2 \in P_{sp}$, we add the term p_1 ;
 - All these expressions are assembled with the “AND” operation.
3. the number of relations among patterns may be large and not always relevant. As a consequence, a designer may still be confused about the choice of the patterns to use, especially when there are conflicted patterns. Hence, we propose to simplify the Boolean expressions and to update SAGs. The Boolean expression reduction is here performed with the tool BExpRed³. For instance, with the three patterns p_1 , p_2 and p_3 having the relations $(p_1 \text{ benefit } p_2)$, $(p_1 \text{ alternative } p_3)$ and $(p_2 \text{ alternative } p_3)$, we obtain $(p_1 \text{ AND } p_2) \text{ AND } (p_2 \text{ OR } p_3) \text{ AND } (p_1 \text{ OR } p_3)$, which can be simplified into the expression $(p_1 \text{ AND } p_2)$. It is manifest that this expression is clearer than the first one. The resulting Boolean expression is graphically shaped with an expression tree, whose nodes are logic operations and leaves are patterns. The resulting expression tree is linked to the hierarchical organization of security principles sp obtained in step 1.

³<https://sourceforge.net/projects/bexpred>

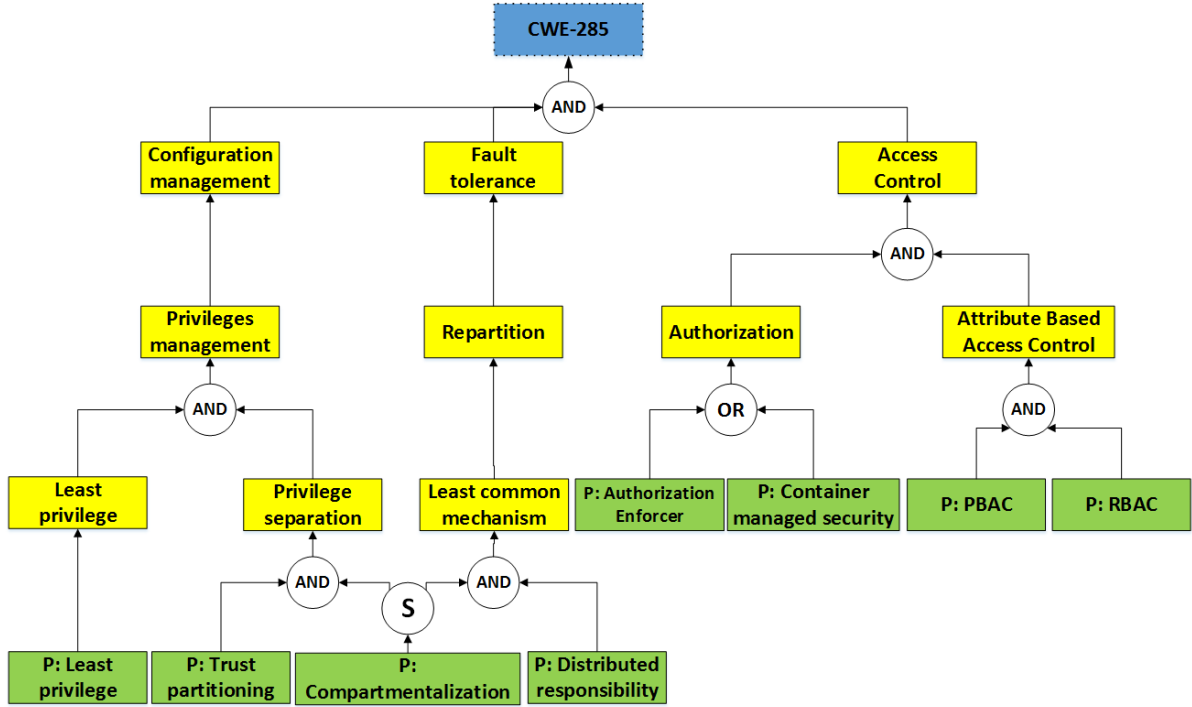


Fig. 4.13 CWE-285 Security patterns tree

A SAG achieved by these steps depicts the combinations of patterns, which overcome a weakness. We believe that these SAGs offer a good point of view on the potential solutions to a weakness, Figure 4.13 depicts the final SAG obtained for the weakness CWE-285. It shows that designers can implement either the security patterns “Authorization Enforcer” or “Container managed security” and should implement all the other patterns (“PBAC”, “RBAC”, “Least privileges”, etc.) in order to fix the weakness. The SAG also shows the security principles applied here.

4.5.3 Attacks and Security Patterns Classification (v_1)

We propose here to catalog the combinations of patterns that aim to mitigate a given attack. With the use of the second data-store *AWSPC*, which associates attacks, weaknesses and security patterns. The classification is extracted as follows. Given an attack $Att \in AWSPC$, we extract:

- the information about Att (name, identifier, description, etc.);

- the hierarchical organization of the sub-attacks of Att ;
- the set of weaknesses W targeted by Att (resp its sub-attacks);
- the set of security principles related to each weaknesses cluster;
- the set of patterns P that are related to every w and the set of patterns P_2 not in P that have relations with any pattern of P , and the nature of these relations.

Table 4.2 depicts an extraction example for the attack “CAPEC-39: Manipulating Opaque Client-based Data Tokens”. The tabular gives the attack ID and name (col. 1,2), the security patterns allowing to prevent the attack (col. 3), the relationships with other patterns (col. 4,5). As the attack “CAPEC-39”, has a sub attack “CAPEC-31 Accessing/Intercepting/Modifying HTTP Cookies”, (col. 6-10) also give the security patterns allowing to overcome the attack CAPEC-31 and their relations with other patterns.

4.5.4 ADTree generation (v_1)

We propose to generate ADTrees, organizing the attacks and their related security patterns. We recall that, with ADTrees, attacks are illustrated with red nodes, which can be conjunctively or disjunctively refined. An attack node can be mitigated with one defense node (in green squares) composed of sub defenses.

In our context, an ADTree shall be rooted by an attack of the CAPEC (stored in the database *AWSPC*). This root node can be connected to other attack nodes, expressing sub-attacks, which can be connected to defense nodes, representing security patterns. Figure 4.14a illustrates a general form of the ADTrees we generate.

We automatically build ADTrees from the database *AWSPC* as follows:

1. every attack Att found in *AWSPC* has its own ADTree whose root node is labeled by its identifier. This root node is linked to other attack nodes with a disjunctive refinement

Table 4.2 Data extraction for the attack CAPEC-39

ID	Name	Security pat	Rela	Relate	ID	Name	Security patterns	Relati	Relationship
39	Manipulating Opaque Client-based Data Tokens	Application Firewall	-No Value-	-No Value-	31	Accessing/Intercepting/M HTTP Cookies	Application Firewall	alternative	Container Managed Security
			alternative	Input Guard			Authentication Enforcer	benefits	Application Firewall
		Input Guard		Output Guard			Compartmentalization	-No Value-	-No Value-
			-No Value-	-No Value-			Container Managed Security	benefits	Demilitarized Zone
			alternative	Application Firewall			Controlled Object Factory	-No Value-	-No Value-
		Pathname Canonicalization	benefits	Output Guard			Input Guard	benefits	Authentication Enforcer
			-No Value-	-No Value-			Output Guard	benefits	Secure Service Facade
			alternative	Compartment			Pathname Canonicalization	-No Value-	-No Value-
							Secure Pipe	-No Value-	-No Value-
							Secure Service Facade	benefits	Demilitarized Zone

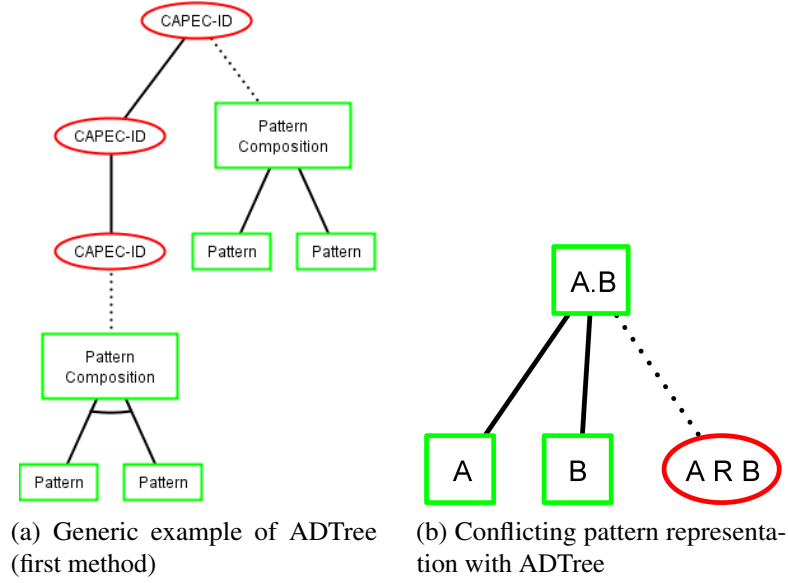


Fig. 4.14 Generic ADTree

if *Att* has sub-attacks. This step is repeated for every sub-attack. In other words, we generate a sub-tree of the original hierarchical tree extracted in *AWSPC*, whose root is *Att*;

2. thanks to the relations defined in the meta-model *AWSPC* (Figure 4.3) for every attack node *A*, we collect the set *P* of security patterns that counter the attack. The inter-pattern relationships are illustrated in the ADTree with new nodes and logic operations. Given a couple of patterns $(p_1, p_2) \in P$, if we have:

- $(p_1 R p_2)$ with R a relation $\in \{depend, benefit\}$, we build three defense nodes, one parent node labeled by $p_1 R p_2$ and two nodes labeled by p_1, p_2 combined with this parent defense node by a conjunctive refinement;
- $(p_1 alternative p_2)$, we build three defense nodes, one parent node labeled by $p_1 alternative p_2$ and two nodes labeled by p_1, p_2 , which are linked by a disjunctive refinement to the parent node;
- $(p_1 R p_2)$ with R a relation in $\{impair, conflict\}$. In this particular case, we would want to use the *xor* operation since both patterns can be used but the implementation

of p_2 decreases the efficiency or conflicts with p_1 . Unfortunately, this operation is not available with this tree model. Therefore, we replace the operator by the classical formula $(A \text{ xor } B) \longrightarrow ((A \text{ or } B) \text{ and not } (A \text{ and } B))$. The *not* operation is here replaced by an attack node meaning that two conflicting security patterns used together constitute a kind of attack. The corresponding sub-tree is depicted in Figure 4.14b;

- p_1 having no relation with any pattern p_2 in P , we add one parent defense node labeled with p_1 .

The parent defense nodes, resulting from the above steps, are combined to a defense node labeled by “Pattern Composition” with a conjunctive refinement. This last defense node is linked to the attack node A .

If we take back our example of CAPEC-39 attack, we obtain the ADTree of Figure 4.15, which shows that the attack has the sub-attack CAPEC-31. Unlike the SAG generation (Section 4.5.2), we do not express graphically the hierarchical organization of security principles (ADTrees are not tailored for this kind of purpose). The security patterns are here directly related to each attack. In the version 2.8 of the CAPEC database, the two attacks CAPEC-39 and CAPEC-31 target 17 weaknesses (6 for the CAPEC-39 and 11 for the CAPEC-31). The attack and all its concrete forms can be countered by several combinations over 10 security patterns. The number of security patterns related to both attacks CAPEC-39 and CAPEC-31 is explained here by the diversity of the targeted weaknesses. We assume for the classification generation that all of them have to be mitigated. As these ones cover different security issues here, e.g., input validation problems, privilege management, encryption problems, external control of the application state, etc., several patterns are required to fix the weaknesses and hence block the attacks.

This example illustrates that a designer can follow the concrete materializations of an attack in an ADTree. He/she can choose the most appropriate attack with respect to the context of the

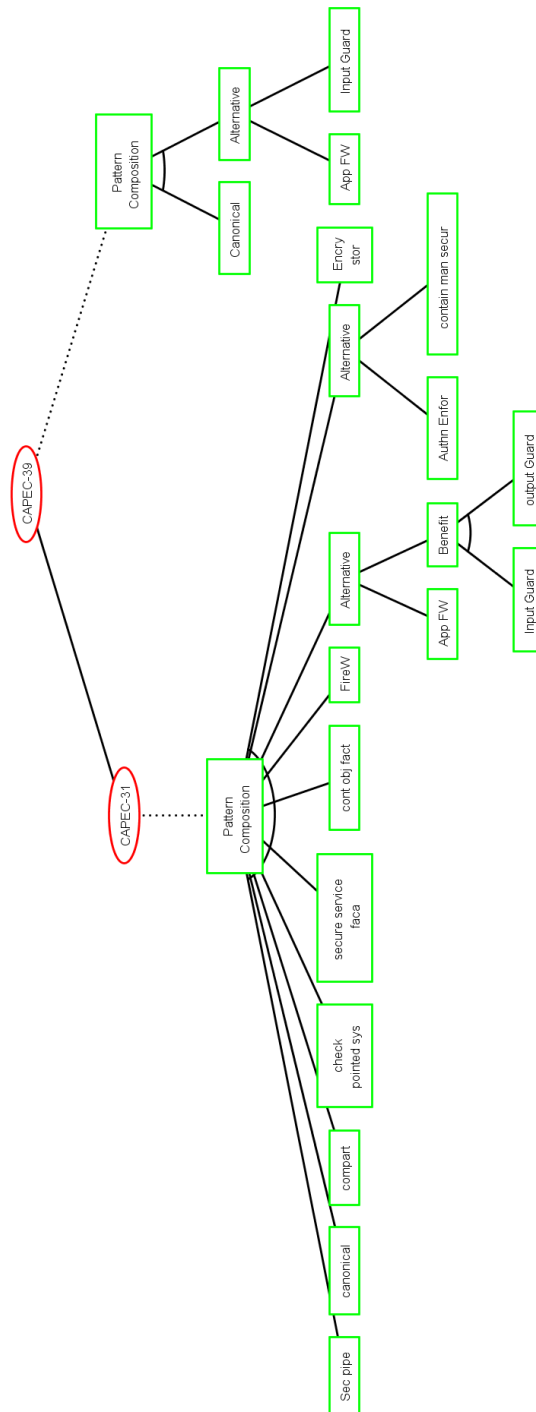


Fig. 4.15 ADTree of the attack CAPEC-39

application being designed. The ADTree provides the different security pattern combinations that have to be used to prevent this attack.

In the worst case, an attack node is not linked to a defense node, which means that either the classification is incomplete (the related weaknesses are not provided in the CAPEC database) or the attack is relatively new and cannot be yet overcome by security patterns.

4.5.5 Attacks and Security Patterns Classification (v_2)

The third database *ASSPC* holds other information about attacks that can be used to classify security patterns that counter attacks by means of another point of view. The database gathers the hierarchical organization of attacks, the sequences of steps required to successfully apply attacks, attack techniques and countermeasures. We propose here to use these relations to automatically generate a third pattern classification denoted *ASSPC*.

Given an attack $Att \in ASSPC$, the following data and relations are hence extracted:

- the information about Att (name, identifier, description);
- the tree $T(Att)$ of the sub-attacks of Att (Section 4.5.3), whose root is Att ;
- For every attack found in $T(Att)$, we extract the sequence of its attack steps. These can be also refined by more concrete steps. We extract also the a set of techniques implementing a step;
- for each step st , the complete hierarchy of security principles $Sp(st)$ by means of the successive relations established among st , countermeasure clusters and security principles in the database *AWSPC* Figure 4.3. $Sp(st)$ represents the complete hierarchy of security principles related to a step, i.e., if a principle sp associated to the step st is not a leaf of our hierarchical organization, then we also extract all the principle sub-tree whose root is sp ;
- for each principle $sp \in Sp(st)$, the set of security patterns P_{sp} , the set of patterns P_{sp2} not in P_{sp} that have relations with any pattern of P_{sp} , and the nature of these relations

defined for couples of patterns by the annotations in $\{“depend”, “benefit”, “impair”, “alternative”, “conflict”\}$.

Table 4.3 depicts an extraction example for the CAPEC attack 34 “HTTP Response Splitting”. The first col gives the ID of the chosen attack. This attack belongs to the category “Detailed” of the CAPEC, therefore it has no sub attacks (otherwise, the next columns would list them too). Cols 2 to 4 index the attack steps and techniques. For instance, we illustrate the step “Experiment” of the attack. The security patterns allowing to prevent the step are given in col 5. These four patterns have to be contextualized in the application model and implemented to prevent the attack. The last two cols add the security patterns being associated with the patterns of col 5 and their relations. Figure 4.3 shows that “Application Firewall” and “Input guard” are alternative patterns, hence using one of them is enough (although using both is not incorrect).

4.5.6 ADTree generation (v_2)

Once more, we propose to generate ADTrees with the general form illustrated in Figure 4.16. This ADTree points out how an attack is sequenced with steps and how to prevent them with countermeasures given under the form of security pattern combinations. An ADTree root node is labeled by an attack ID. In comparison to the ADTrees of Section 4.5.4, an attack node is refined with other nodes expressing steps and techniques.

We automatically generate these ADTrees from the data-store *ASSPC* as follows:

1. every attack $Att \in ASSPC$ has its own ADTree whose root node is labeled by its identifier. This root node is linked to other attack nodes with a disjunctive refinement if the attack has sub-attacks. This step is repeated for every sub-attack;
2. for each attack Att of the preceding tree, we collect its sequence of steps. The node labeled by Att is refined with a sequential conjunction of attack nodes, one for each step.

Table 4.3 Data extraction for the attack CAPEC-34

attack_ID	Attack_StepTitle	Attack_Step	tech_desc	Security_pattern	SP_relationship	related_Security_pattern
34	Experiment	Attempt variations on input parameters	Use CRLF characters (encoded or not) in the payloads in order to see if the HTTP header can be split.	Application Firewall	alternative	Input Guard
				Audit Interceptor	benefits	Output Guard
					depends	Secure Service Facade
				Input Guard	alternative	Secure Logger
					benefits	Application Firewall
			Use a proxy tool to record the HTTP responses headers.	Secure Logger	benefits	Output Guard
						Audit Interceptor
						Secure Pipe
				Application Firewall	alternative	Input Guard
						Output Guard
				Audit Interceptor	benefits	Secure Service Facade
					depends	Secure Logger
				Input Guard	alternative	Application Firewall
					benefits	Output Guard
				Secure Logger	benefits	Audit Interceptor
						Secure Pipe

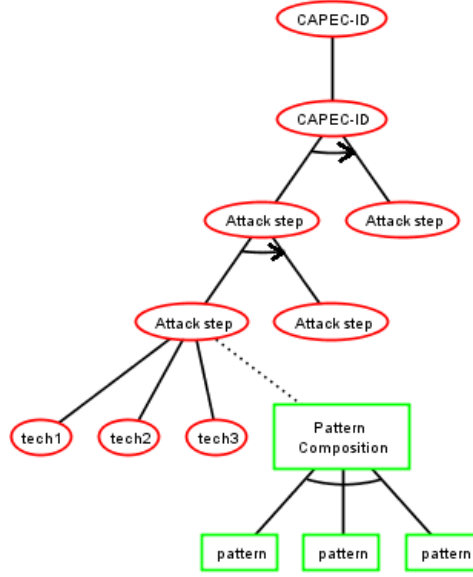


Fig. 4.16 Generic example of ADTree (second method)

We repeat this process if a step is itself composed of steps. In the same way, for each step St , the related techniques are extracted from the data-store *ASSPC* and are associated to the node labeled by St with a disjunctive refinement;

- for each step St , we extract from the data-store *ASSPC* the set P of security patterns that are countermeasures of St . Given a couple of patterns $(p_1, p_2) \in P$, we illustrate these relations with new defense nodes in the same way as the previous ADTree given in Section 4.5.3.

Figure 4.17 illustrates the ADTree obtained for the attack CAPEC 34. The root of the tree is the main goal of the attacker. Its second and third levels relate to the attack steps. These nodes are sequential conjunctive refinements of the root node. For instance, the step Exploit is achieved if both steps 3.1 and 3.2 are successfully executed in the right order (from left to right). An attack step has a disjunctive refinement of attack nodes labeled by techniques. The step is achieved if one of the attack techniques is applied with success. Defense nodes (square nodes) illustrate security pattern combinations. For instance, the step “1.1 Spider” refers to the Web application exploration through Graphical user interfaces in order to get all the URLs of the

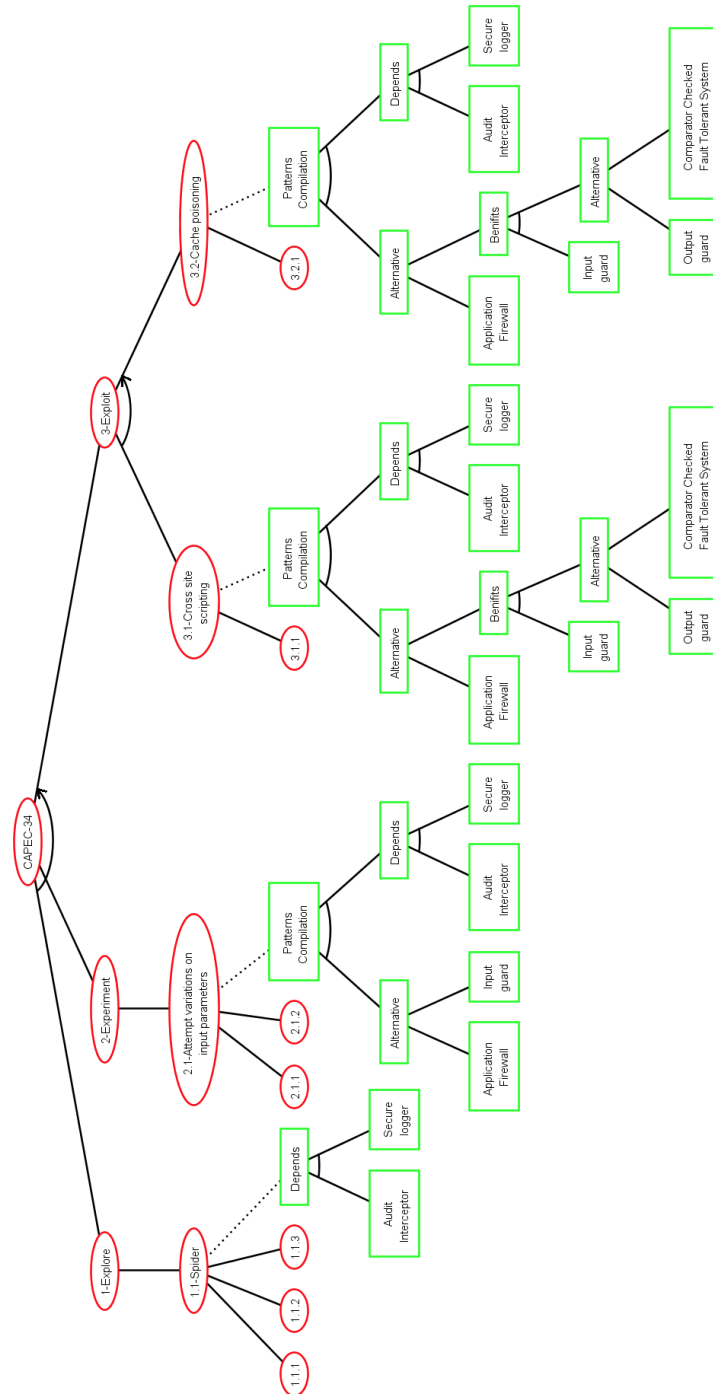


Fig. 4.17 ADTree of the Attack CAPEC-34

application. This step can be prevented by designing the application with both patterns “Audit interceptor” and “Secure logger”. “Audit interceptor” can be used to detect the application crawling and to warn an administrator. The audit logs are secured by means of “Secure logger”.

The latter guarantees that the audit logs cannot be accessed or altered by unauthorized users. This example illustrates that a designer can easily follow the concrete materializations of an attack in an ADTree and can directly choose security patterns.

Table 4.4 Attack techniques descriptions

Attack technique	Technique description
1.1.1	Use a spidering tool to follow and record all links and analyze the web pages to find entry points. Make special note of any links that include parameters in the URL, forms found in the pages (like file upload, etc.).
1.1.2	Use a proxy tool to record all links visited during a manual traversal of the web application.
1.1.3	Use a browser to manually explore the website and analyze how it is constructed. Many browsers' plugins are available to facilitate the analysis or automate the discovery.
2.1.1	Use CR/LF characters (encoded or not) in the payloads in order to see if the HTTP header can be split.
2.1.2	Use a proxy tool to record the HTTP responses headers.
3.1.1	Inject cross-site scripting payload preceded by response splitting syntax (CR/LF) into user-controllable input identified as vulnerable in the Experiment Phase.
3.2.1	The attacker decides to target the cache server by forging new responses. The server will then cache the second request and response. The cached response has most likely an attack vector like Cross-Site Scripting; this attack will then be serve to many clients due to the caching system.

The description of the different techniques related to the attack CAPEC-34 steps are given in Table 4.4, which gives an insight on how the different steps of the attack can be implemented.

4.6 Discussion

In Section 4.4, we proposed a set of manual and automatic steps integrating 185 CWE weaknesses, 215 CAPEC attacks, 26 security patterns and 66 security principles in three databases: **WSPC**: Weaknesses based Security Patterns Classification, **AWSPC**: Attacks and Weaknesses based Security Patterns Classification and **ASSPC**: Attack Steps based Security Patterns Clas-

sification. These databases enable multi-attribute based decision since security patterns can be selected according to: weaknesses, attacks, attack steps or security principles.

As presented in Section 4.5, several patterns classifications can be automatically extracted from the databases. In addition, the readability of these classifications is enhanced by expressing them graphically with SAGs and ADTrees. We have developed a tool in order to generate ADTrees with the two methods (Sections 4.5.4, 4.5.4). It generates automatically for each attack the corresponding ADTree in the form of an XML file, which can be loaded under the tool ADTool⁴ allowing designers the visualization and the analysis of each attack and its related set of security patterns. We have evaluated the quality of our classification with regard to the quality criteria defined by Alvi et al. [5], we have denoted that the proposed classifications meet the following quality criteria:

- **Navigability:** our classifications, accompanied by SAGs and ADTrees, satisfy the Navigability criteria the relations among security patterns are given. Thus, designers are guided in the choice of the appropriate security patterns set by highlighting depended or conflictual patterns;
- **Determinism:** the three classifications are clearly defined by means of the relations of the methodology steps. All these steps justify the soundness of the classification;
- **Unambiguity/Comprehensibility:** patterns are classified with regard to defined categories, i.e., attacks, steps, weaknesses or security principles. This organization, which is illustrated by means of SAGs/ADTrees, makes our classification readable and comprehensible even for novices in security patterns;
- **Usefulness:** we believe that the classifications can be used in practice since they are based upon a known security pattern catalog [116] and upon the bases CAPEC [67] and CWE [69], which are more and more employed in the industry. In addition, we

⁴<http://satoss.uni.lu/members/piotr/adtool/>

the evaluation of Section 5.5, which we conducted with 24 participants, shows that our method enhanced the Comprehensibility, the Accuracy and Efficiency of the participants in choosing security patterns;

- **Repeatability:** the databases and the classifications can be updated and generated semi-automatically. In addition, the classifications and the tools developed for the data integration are available in [84].

A plethora of statistical information can be extracted from the obtained data-stores. For instance, Figures 4.18, 4.19 and 4.20, illustrate that the patterns (“Input Guard”, “Application Firewall”) cover a considerable part of weaknesses and attacks. From these graphics, one can deduce that it is important to provide an application with an input validation mechanism using the pattern “Input Guard” or “Application Firewall” since both of them partially cover up to 17% of the weaknesses and 27% of the attacks. These statistical information can be useful to know the number of attacks or weaknesses partially cured by a security pattern (respectively a set of security patterns).

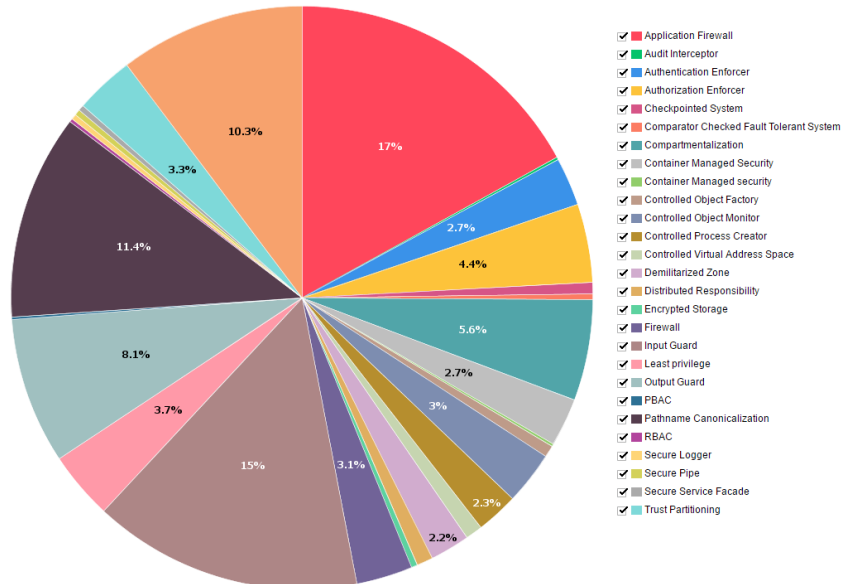


Fig. 4.18 Distribution of fixed weaknesses per pattern

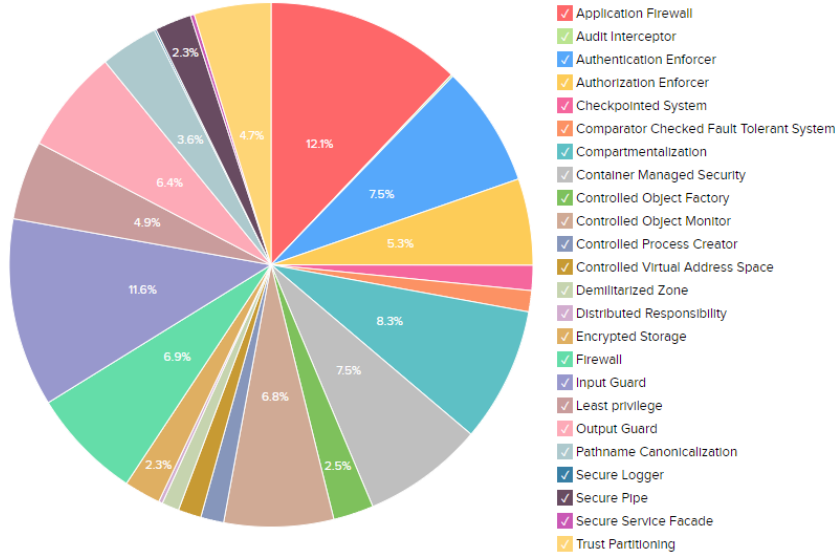


Fig. 4.19 Distribution of fixed attacks per pattern (V_1)

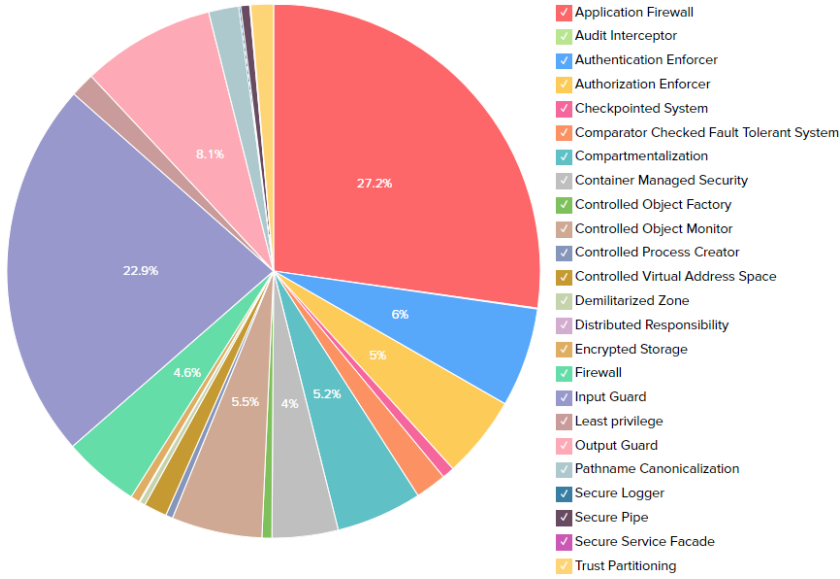


Fig. 4.20 Distribution of fixed attacks per pattern (V_2)

4.6.1 Comparaison of the ADTrees generated with the approaches (v_1, v_2)

In this chapter, we classify security patterns and attacks and then we generate ADTrees with two different approaches. In the classification *AWSPC*, we classify security patterns and attacks with regard to the notion of weaknesses.

We compared the ADTrees generated with the two classifications for 21 attacks and we concluded that they are complementary. The first method is related to the static part of the attack (weaknesses targeted by the attack), while the second is related to the dynamic part of the attack (the sequencing of the attack steps). In the first one, attacks are related to a bigger number of security patterns compared to the second one. But the patterns given by the classification *ASSPC* seem to better target the security problem raised by the attack. Thus, the combination of the two approaches provides a wider insight on the attack and the related solutions materialized by the different combinations of security patterns. The ADTrees generated with the two approaches are available in [84].

Attack purposes are subdivided by considering the related mitigations. As the CWE base used by our data integration technique is rich, this classification offers a lot of patterns combinations. Figure 4.21, illustrates an example of ADTree generated for the “CAPEC-244: Cross-Site Scripting via Encoded URI Schemes”. In this example, the main solutions provided by the set of the security patterns are input and output validation in addition to the authentication.

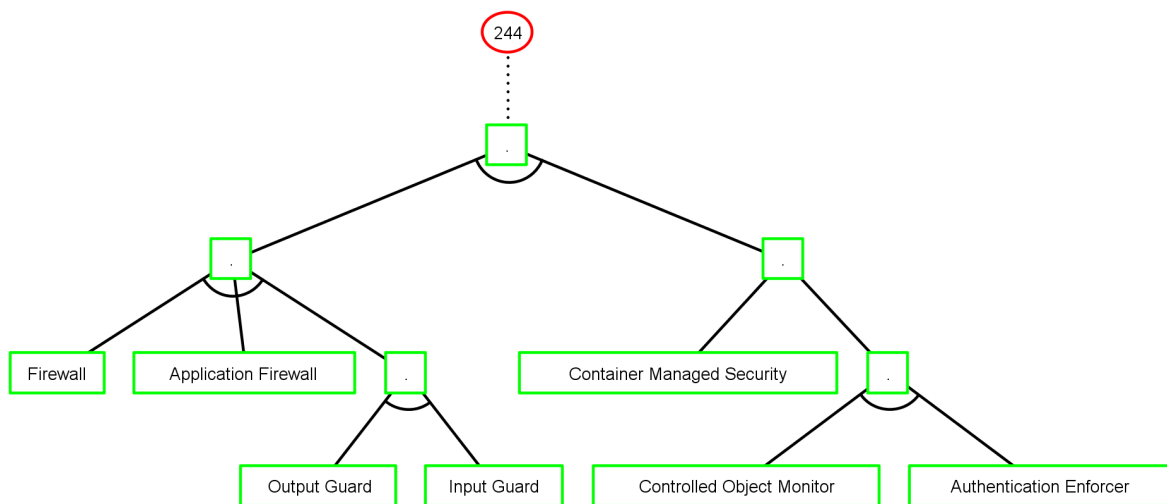


Fig. 4.21 CAPEC-244 ADTree (AWSPC)

Security analysis with patterns and pattern classifications

With the second classification (*ASSPC*), each attack is subdivided into a sequence of steps and each step is characterized with a set of countermeasures clusters, themselves associated to patterns. The CAPEC base holds less information about countermeasures. As a consequence, the classification provide less patterns per attack. Figure 4.22 illustrates the ADTree of the attack CAPEC-244 generated from the classification *ASSPC*. The set of the security patterns are related to input and output validation in addition to the logging and audit facilities.

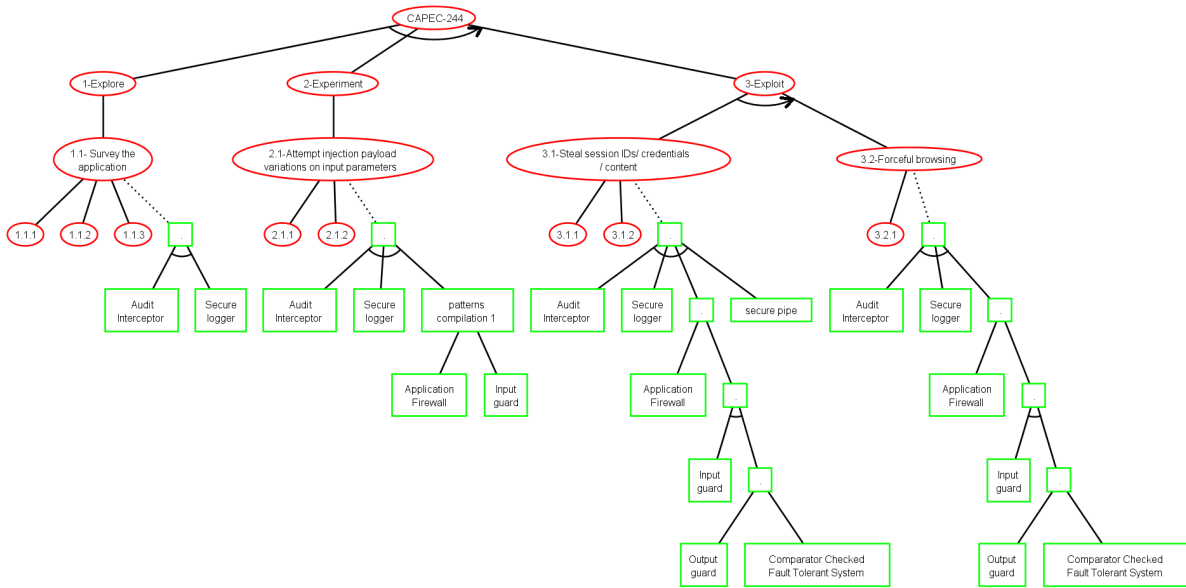


Fig. 4.22 CAPEC-244 ADTree (ASSPC)

If we compare these two ADTrees (Figures 4.21, 4.22), the patterns in common are (Input guard, Output guard and Application Firewall) related to input and output validation. The contrast between the two approaches is that in the first one (4.21) the attack CAPEC-244 is related to security patterns dealing with the authentication (Authentication enforcer, Controlled object monitor and Container managed security). This is explained by the fact that many of the weaknesses targeted by the attack (in version 2.8) (e.g., CWE-20, CWE-79, CWE-220) are related to authentication flows. While in the second approach (Figure 4.22), the attack CAPEC-244 is related to audit and logging mechanisms because of the importance of tracking

the actions of the attacker expressed by the countermeasures. Hence, the two models are complementary and allow a wider insight on the security patterns related to an attack.

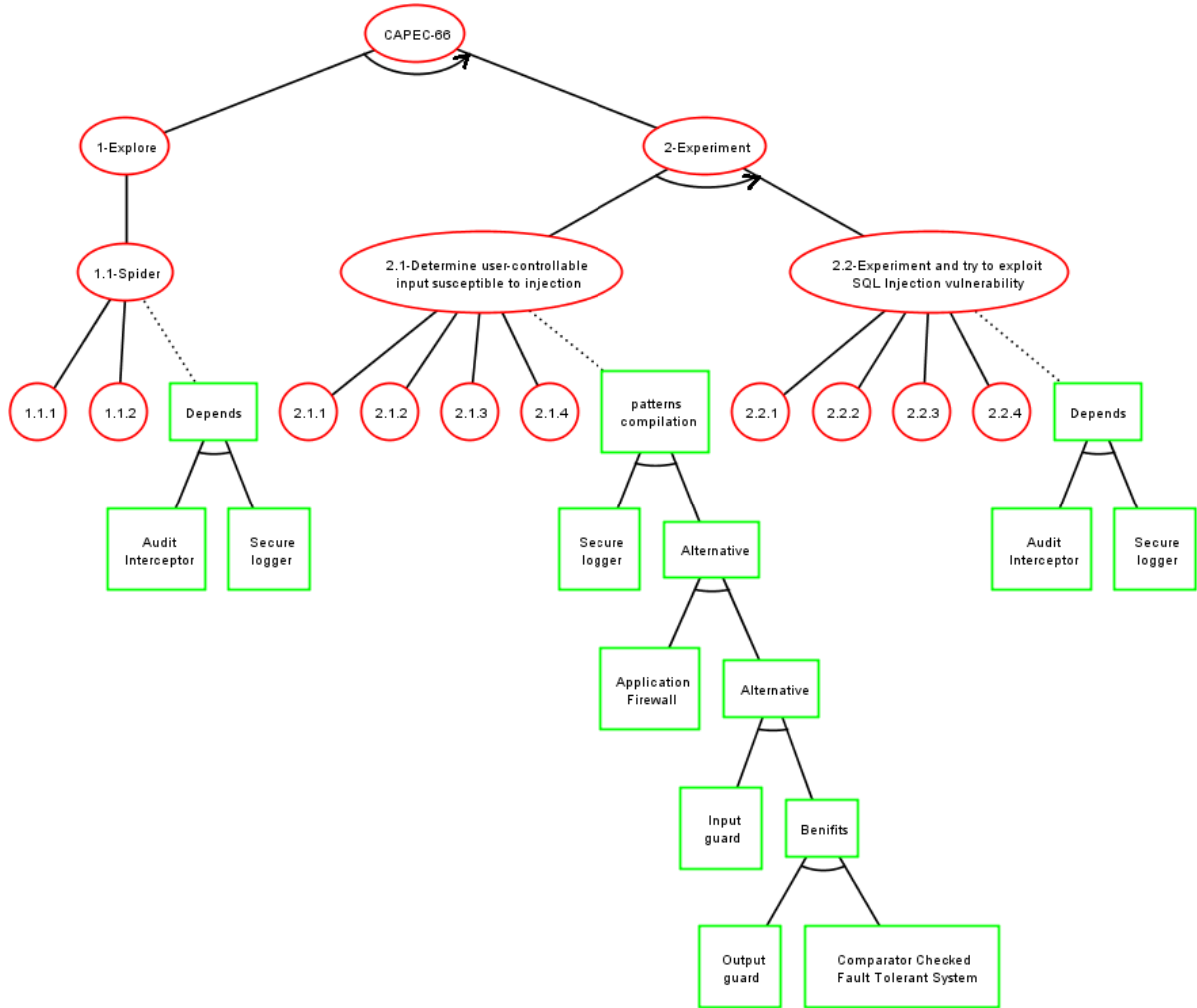


Fig. 4.23 “CAPEC-66: SQL Injection” ADTree

4.6.2 Comparison with other approaches

We compared our classification with the two papers providing relations between security patterns and attacks [110, 4]. In these works, the security pattern intents are manually compared to the summaries of the attacks. As these textual sections are abstract, few relations were found. The largest contribution is provided by Alvi et al. who considered around 20 attacks

and manually linked them to 5 patterns. In contrast to these works, our classification is more complete: we provide 26 security patterns as solutions against 215 attacks of the CAPEC base. Our classification exposes more pattern combinations per attack; the more choice is not always the better though. After inspection, we observed that more than one or two patterns are generally required to counter attacks. A last important point is that the classifications exposed in [110, 4] do not contradict our relations between attack and patterns. For instance, the attack “CAPEC-66 SQL Injection” is related to the security patterns “Intercepting Validator” and “Input validation” in [110]. The attacks “CAPEC-244” and “CAPEC-66” are only associated with the pattern “Intercepting Validator” in [4]. For these attacks, our method generates two ADTrees, which provide 4 combinations of 7 patterns for the CAPEC-244 and 8 combinations of 9 patterns for the CAPEC-66. As in [110, 4], the ADTrees exhibit the pattern “Input Guard”, which can be implemented by “Intercepting Validator”. But, the ADTrees also list other patterns. For the CAPEC-244, some of these patterns are alternative to “Input Guard”, e.g., “Application Firewall”. Other patterns, e.g., “Authentication Enforcer” or “Controlled Object Monitor” are related to specific weaknesses targeted by the attack CAPEC-244. We believe these patterns, which are not given in the previous classifications, are required to counter the attack with regard to the application context.

4.6.3 Limitations

Our classifications and methods present some limitations, which could lead to some research future work:

- the notion of attack combination is not considered in these methods. Such a combination could be seen as several attacks or as one particular attack. If an attack combination can be identified and documented with its sub-attacks, then it can be integrated in our data-store;

- the ADTree size limit is not supported by our ADTree generator. When an attack has a high level of abstraction, we observed that the resulting ADTree size can become large because it includes a set of sub-attacks, themselves linked to several patterns. This is a strong limitation since large trees are usually unreadable, which contradicts the method purposes;
- the classification is not exhaustive: it includes 215 attacks out of 569 (for any kind of application) and 26 security patterns out of 176. It can be completed with new attacks automatically. But the completion of the data-store with new security patterns requires some manual steps. It could be interesting to investigate whether text mining techniques would help partially automate them. The classification exhaustiveness also depends on the available security data. In the ADTree of Figure 4.17, all the lowest attack nodes are linked to defense nodes. We sometimes observed that no defenses are provided with other attacks. This can be usually explained by three main reasons:
 1. security databases or pattern catalogs are incomplete (lack of mitigation, countermeasure, etc.). More data are required while the data integration process. In particular, we observed that some countermeasures are missing for some attacks of the CAPEC base;
 2. the attack is relatively new. It is not yet documented or no pattern based solution is available;
 3. security data are missing because we did not considered them in the manual data integration steps. For instance, as the pattern descriptions do not clearly provide strong points, it is easy to skip one of them;
- several steps require manual interventions, which are prone to errors. These steps may lead to associations among security data that are bound to be controversial. We compared our results with other papers in Section 4.6.1 , but this is insufficient to ensure all the

associations are correct. Validating every relation is a hard problem. It could be partially solved by the use of verification methods. But the writing of formal expressions for modeling the entities and associations of our meta-model is another long and error-prone task that should be addressed;

- the inter-pattern associations are defined with binary relations only, as presented in [33]. These relations could be updated to link several patterns together.

4.7 Chapter conclusion

In this chapter, we introduced the architectures of three databases gathering information about security patterns, weaknesses, attacks and security principles.

We detailed a set of semi-automated data acquisition and integration steps allowing the fulfillment of these data-stores (*WSPC*, *AWSPC* and *ASSPC*) with data extracted from security pattern, weakness and attack documentations. These data-stores associate 215 attacks, 185 weaknesses, 26 security patterns and 66 security principles specialized to the Web applications context. The data-store (*ASSPC*) allows the extraction of the attack steps sequences and will be used in the next chapter (Chapter 5) for devising a test case generation method.

From these databases, three classifications are extracted:

- **WSPC**: Weaknesses based Security Patterns Classification;
- **AWSPC**: Attacks and Weaknesses based Security Patterns Classification;
- **ASSPC**: Attack Steps based Security Patterns Classification.

The goal of these classifications is to assist designers in the choice of the appropriate set of security patterns to protect an application against a weakness or an attack. In addition, the relations among security patterns are highlighted to help designers in the security patterns choice. We enhance the readability of the obtained classifications by representing them

graphically using Security Activity Graphs (SAGs) and Attack Defense Trees (ADTrees). The visual support provided by SAGs and ADTrees helps designers understand the activities needed to mitigate a weakness, the scenarios of attacks (steps, techniques, etc.) and the relations among security patterns.

The three classifications and the tool-set allowing the generation of the classifications and ADTrees are available in [84]. The work proposed in this chapter has been published in [83, 85, 92].

Chapter 5

Assisting developers in the generation of security test cases

5.1 Introduction

We introduced in Chapter 4 pattern classification methods using a set of data-stores. These draw up relationships among attacks, weaknesses and security patterns. The purpose of these data-stores is to provide a developer with a set of information allowing him or her to choose security patterns.

Developers often lack of expertise in security [96, 95] and we observed in the literature that they still lack of guidance in conceiving and implementing security tests. This chapter focuses on this issue and takes advantage of the data gathered in the data-stores developed in Chapter 4 to assist developers in the Threat modeling stage, the generation of security test cases and their executions.

This chapter is organized as follows: Section 5.2 introduces the context and the motivations of the proposed method. In section 5.3 we extend the data-store introduced in Chapter 4 to include new relations and data required for the test case generation. We provide in Section

5.4 the steps of test case generation and execution method. We perform an evaluation of the method in Section 5.5. We finally conclude this chapter in Section 5.6.

5.2 Context and motivations

Model-based security testing has been broached by a plethora of works. Among these, we are interested in those dealing with the non-functional aspect of an application, more precisely the security aspect. We focused on works where models do not describe the functional behavior of the application but rather express the attacker goals, interactions and resources along with the causes of the system vulnerabilities [58, 59, 70, 98]. Among these papers, some authors focused on the representations of security concerns with models such as Attack trees, Threat trees, Security activity graphs, etc. expressing security problems and their associated solutions. Besides the graphical expressions about threats, attacks or vulnerabilities these models provide, they can be used for testing whether an application is vulnerable to attacks. Indeed, in many works, security test cases are manually extracted from these graphical models. Below, we present some of these works.

A security testing approach was introduced by Morais et al. to assess the security of protocols [70]. From attack trees, a set of scenarios are manually extracted and then converted to attack patterns and UML specifications. Based on these elements, attack scripts are then manually written and completed with the injection of network faults. In [58], Marback et al. proposed an approach deriving attack trees from data flow diagrams. From the obtained attack trees, a set of events sequences are then extracted. These sequences are combined with parameters associated to regular expressions in order to generate concrete values. These values are finally replaced manually by blocks of code to produce security test cases. In [59], test cases are derived from threat trees, which are completed with parameters associated to regular expressions. Scenarios are then extracted from these threat trees and converted manually to security test scripts. Shahmehri et al. proposed in [98] a passive testing approach in order

to detect vulnerabilities. The undesired properties carried by vulnerabilities are materialized graphically in the form of Security Goal Models (SGMs), which are kind of Directed Acyclic Graphs (DAGs) expressing goals, vulnerabilities and eventually their associated mitigations. The detection criteria are then semi-automatically extracted and given to a monitoring tool, which returns test verdicts.

In these works, the proposed approaches take as input the threat models, which are often manually drawn. The test cases are also manually written. In addition, when models do not express enough details about an attack or a vulnerability (steps, parameters, etc.) the resulting test cases are often very abstract. As a consequence, test cases are still abstract, i.e., developers have to concretize them

If we take back the notion of security patterns, few works tackled the problem of testing their correct instantiation (i.e., the application satisfies their behavioral properties) or their efficiency [115, 48]. Yoshizawa et al. proposed in [115] an approach to check whether the structural and behavioral properties can be observed in the method calls and execution traces of an instrumented application. For each security pattern, two test templates, materialized by OCL expressions, are written to express the structure and the behavior of the security pattern. In addition, developers have to write Selenium scripts to experiment the application. The scripts stimulate the application, which returns traces. Then, the satisfiability of the OCL expressions is checked against these traces. Konrad et al. proposed in [48] a testing method where security requirements are modeled with UML State charts and security pattern consequences are expressed with LTL formulas. They proposed to argue security pattern

Hence, the security patterns documentation is enhanced by adding a formal expressions of the security patterns. These are used to check whether an application model satisfies the security properties provided by a pattern. However, the LTL formulas are not generic, which makes them hardly reusable. The developer has to write the LTL properties with regard to the application model.

On the one hand, the writing of detailed threat models requires a lot of expert knowledge and of documents. The referred papers neither guide developers in the threat modeling phase nor provide security solutions. On the other hand, some methods propose to generate test cases from (formal) specifications. These test cases are often at an abstract level. They cannot be directly used to experiment an application under test. Some methods tried to tackle this problem using a mapping technique. However, this kind of technique is usually very limited in its capability to translate abstract tests into concrete ones. Hence, most of the security testing approaches, especially those taking threat models as inputs, rely on developers to write concrete test cases. But, they do not give any recommendation on how to write and structure executable tests or to make them reusable.

The contributions of this chapter aim to help developers in these tasks. Once he or she has given its initial test requirements with a first ADTree, our approach semi-automatically completes it with attack steps, techniques and defenses given under the form of security patterns. Our approach assists the developer in the test suite generation, by structuring test cases and by completing them with comments or blocks of code. The test case execution provides verdicts expressing whether the application is vulnerable to the threats modeled in the ADTree or whether its behavior includes the observable consequences of the security patterns. This chapter provides an overview of the approach from the developer's viewpoint, proposes to take advantage of the ordering of the attack steps in ADTrees to reduce the test costs, revisits the test verdict definitions, details the ADTree generation, and includes additional evaluation results and examples.

The approach we propose relies on the data gathered in the data-store *ASSPC* (Section 4.5.5) in order to automatically infer ADTrees. These express the steps, techniques, countermeasures in addition to the set of interconnected security patterns allowing the implementation of these countermeasures. Furthermore, Rojas et al. [88] studied the effects of using an automated test generation tool during development and evaluated some criteria on human subjects. They

concluded that the Readability of the test cases, the Integration of the test generation approaches in the software life cycle and Education are the most important criteria to improve the Efficiency and Effectiveness of developers. We have concentrated our efforts on these criteria to generate readable and reusable test cases, which are clearly associated with some parts of the threat model. Efficiency and Effectiveness are the criteria used for evaluating our approach.

5.3 Data-store extension and data integration

The approach introduced in this chapter strongly relies on the information gathered in the data-store *ASSPC* detailed in Section 4.5.5. The latter stores information about attacks, the ordered steps sequences of attacks, the countermeasures and the set of security patterns allowing to counter an attack step. We provide in Section 4.5.6 a technique to generate ADTrees showing the sequences of attack steps and security patterns compilations.

In this section, we extend the database *ASSPC* with supplementary information to generate security test cases. We first introduce the meta-model of the extended data-store, then we present the data acquisition and integration steps in order to fulfill the data-store.

5.3.1 Data-Store meta-model extension

We extend the meta-model of the data-store *ASSPC* (white entities in Figure 5.1) with the new elements (yellow entities in Figure 5.1). We denote the resulting data-store Testing Data Store (*TDS*).

In Figure 5.1, every security pattern is also characterized with a set of consequences, which can be observed in the application behavior. These are enumerated under the section “**Consequences**” in the security pattern documentation.

An attack step is now associated to one or several application contexts. Each one is related to one test architecture. The context refers to an application family, e.g., Android applications,

To make test cases readable and re-usable, we use the behavior driven approach using the pattern “Given When Then” (shortened GWT) to break up test cases into several sections:

- Given sections aim at putting the application into a known state;
- When sections trigger some actions (stimuli);
- Then sections are used to check whether the conditions of success of the test case are met with assertions. We consider two kinds of Then sections. We use Then sections to check if an application is vulnerable to an attack step st . In this case, the Then section returns the verdict “ $Pass_{st}$ ”. Otherwise, it provides the verdict “ $Fail_{st}$ ”. Furthermore, we consider other Then sections for testing the detection of pattern consequences in the application behavior. These Then sections return “ $Fail_{sp}$ ” if a consequence of the security pattern sp is not detected and “ $Pass_{sp}$ ” otherwise.

Each test case section is linked to one procedure stored in the Procedure table of the data-store TDS , which implements the section. A Given, When or Then section can be reused with several attack steps or security patterns. With regard to the meta-model given in Figure 5.1, a GWT test case section (and procedure) is classified according to one application context and one attack step or pattern consequence. In some specific application contexts, the procedures can be completed with comments or with blocks of code to ease the test case development. When the procedure content can be reused with any application in a precise context, we call it *Generic procedure*:

Definition 5.1 (Generic procedure). *Let C be an Application context. A Generic procedure is a block of code, related to a Given, When or Then test case section, that can be used with any application of C ;*

The data-store must only contain Generic procedures related to an application context. It worth mentioning that an empty procedure is generic.

5.3.2 Data acquisition and integration

We developed a set of data acquisition and integration steps to complete the new tables of the data-store *TDS*. We chose to focus on the CAPEC base to extract information about the attacks (e.g., Indicators, Outcomes, prerequisites, etc.). We also based our method on the security pattern catalog published in [116]. With the data-store *ASSPC* (Section 4.5.5), the security pattern catalog and the CAPEC database, we fulfill the data-store as follows :

1. from the security patterns catalog, we extract for each pattern the set of its **consequences**. These are manually extracted from the section consequences of the security pattern documents;
2. from the CAPEC database, we automatically extracted, using Talend ETL, for each attack step:
 - **application context** of the attack step, which is the domain in which the attack step is applicable (web, mobile, client/server, etc.);
 - **prerequisites** and the **ressources required**, which the developer has to setup before launching the attack. These are automatically extracted from the sections “Attack Prerequisites” and “Resources Required” of the attack documentation;
 - **outcomes** and the **indicators** of the attack step success are automatically extracted from the sections “Outcomes” and “Indicators”;
3. for each **application context**, we manually supply its related **test architecture**. Thus, an attack step can be applied in different application contexts using different test architectures;
4. the data is consolidated into the data-store *TDS* which will be used in the test generation process.

5.3 Data-store extension and data integration

We provide each procedure with a set of guidelines, in the form of comments. The information provided in these guidelines originate from the data-store *TDS* as follows:

- for the procedures of **Given** sections, we provide comments about what the developer has to prepare before each attack step, which are extracted from the **prerequisites** and **resources required** of the attack steps;
- for the procedures of **When** sections, we provide comments about the different techniques that implement the attack step, which are provided in the **attack step techniques** of the attack;
- for the procedures of **Then** sections, we add comments indicating:
 1. how the developers can assert the success of the attack step the comments come from **indicators** and **outcomes** of the attack steps;
 2. how to detect security patterns **consequences** these comments come from the consequence of Figure 5.1.

Besides the 215 attacks, 209 attack steps, 448 attack techniques, 217 countermeasures, 26 security patterns and 66 security principles already gathered in the original data-store, we extended the data-store with 627 GWT test cases and 632 procedures automatically generated, knowing that attack steps GWT sections can share procedures. In addition, we collected 43 security patterns consequences.

For the Web applications context, we observed that several procedures can be completed with blocks of code calling penetration testing tools. We manually completed 32 procedures, which cover 43 attack steps. We used the testing framework Selenium¹ and the penetration testing tool ZAPProxy², which covers varied Web vulnerabilities. This set of procedures are

¹<http://www.seleniumhq.org/>

²<https://www.owasp.org/index.php/OWASPZedAttackProxyProject>

usable on any Web application. The scripts allowing the data integration and data-store update are available in [84].

5.4 An approach for guiding developers devise more secure applications

In this section, we present a security test case generation method that is based on the data-store (*TDS*), Figure 5.2 illustrates the method steps.

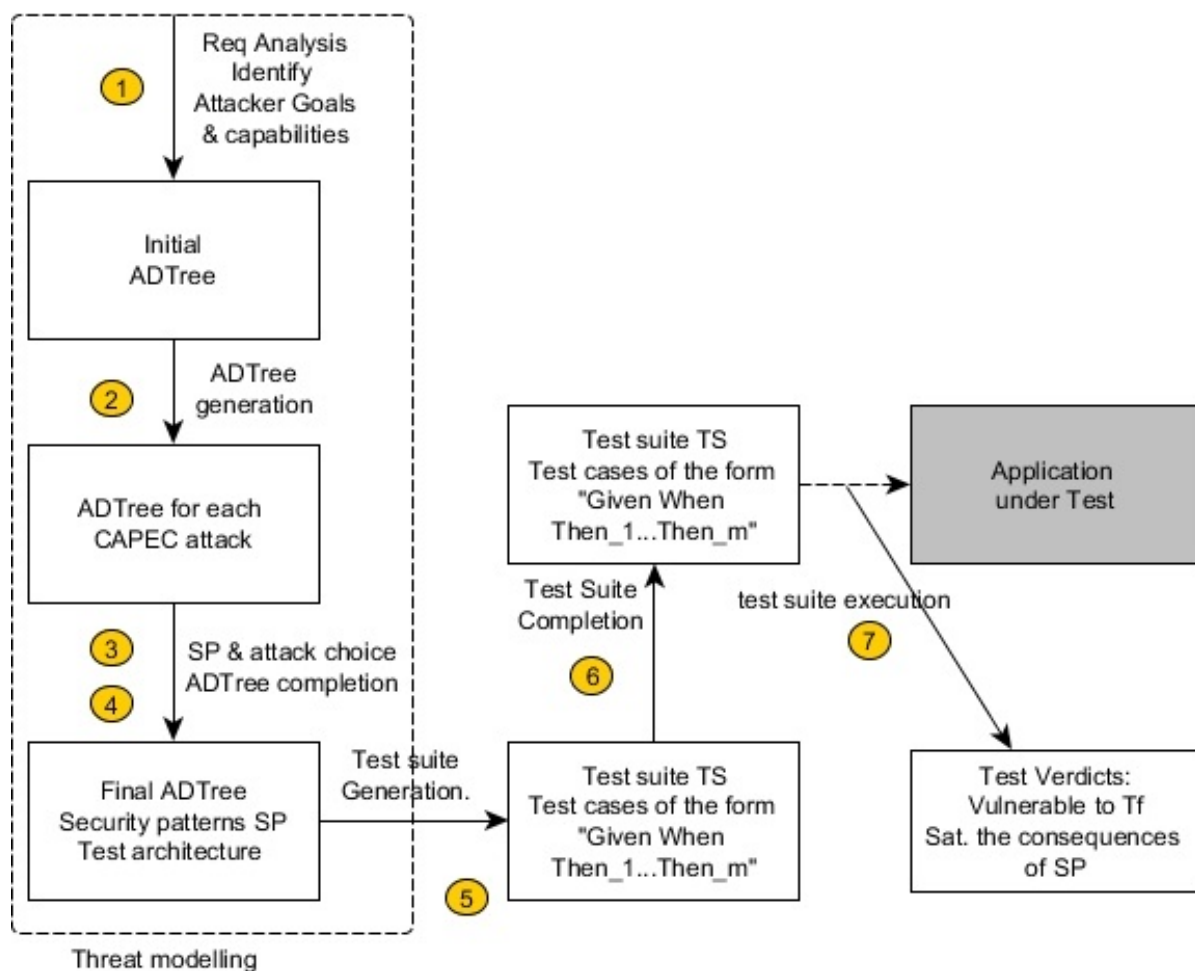


Fig. 5.2 Test cases generation and execution steps

5.4 An approach for guiding developers devise more secure applications

The four first steps take place in the Threat modeling stage, the three last ones in the test generation and execution.

5.4.1 Threat modelling

Step 1: Initial ADTree design

In this step, the developer initially analyzes and models the threats related to the application context, resources, technologies, etc. Threat modeling aims at identifying and describing the attackers goals and capabilities as well as the potential security problems to which the application can be confronted. In order to help developers in this task, several threat modeling approaches was proposed in the last decade, e.g., DREAD [76] or STRIDE [76, 43].

During the Threat modeling phase, we assume the developer initially devises a first ADTree T whose root node represents the attacker's main goal, which may be refined with children nodes. We assume here that T has leaves labeled by attacks (CAPEC-id) available in the data-store. Otherwise, a semantic alignment may be required to substitute the node descriptions.

Figure 5.3 illustrates an ADTree example whose goal, expressed by the root node, is to inject malicious code into an application. The goal is disjunctively refined with two children labeled by two attacks “CAPEC-66: *SQL Injection*” and “CAPEC-244: *Cross-Site Scripting via Encoded URI Schemes*”.

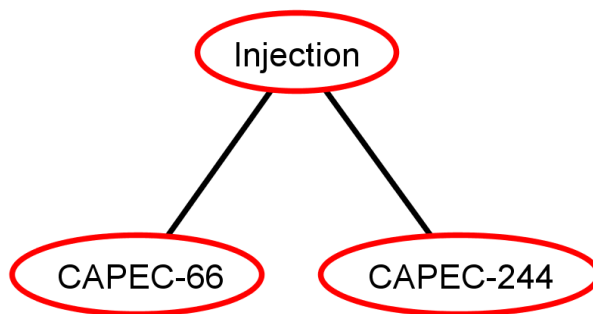


Fig. 5.3 Initial ADTree example

Technically, the developer can employ the tool ADTool³ for editing the initial ADTree, which is exported as an XML file given to the next step.

Step 2: ADTree generation

The ADTree T is often abstract. As developers are often not expert in security, attacks are not detailed, no or few countermeasures are given. This step aims at guiding developers to refine the ADTree T .

For each node of the ADTree T labeled with an attack Att , an ADTree $T(Att)$ is automatically generated from the data-store TDS , which express the sub-attacks, the sequences of attack steps, the techniques and the set of security patterns allowing to counter each attack step. We reuse the technique proposed in section 4.5.6.

We developed a tool taking as input the XML file of the ADTree T and generating a set of ADTrees. An XML file is automatically generated for each ADTree $T(Att)$, which can be loaded under ADTool in order to be visualized, analyzed or edited.

For instance, Figure 5.4 depicts the ADTree corresponding to the node CAPEC-66 in the ADTree obtained in Step 1. Each lowest attack step node has a defense node expressing pattern combinations.

The ADTrees obtained in Step 2 describe all the possible combinations of the security patterns related to an attack step. For instance, for the Step 2.1, the security pattern “Application Firewall” can be substituted with “Input Guard” or one of the two security patterns (“Output Gaurd”, “Comparator Checked Fault Tolerant System”). “Output Gaurd” can be substituted by “Comparator Checked Fault Tolerant System”. This is explained by the fact that a security mechanism can be implemented following different strategies or using different techniques.

³<http://satoss.uni.lu/members/piotr/adtool/>

5.4 An approach for guiding developers devise more secure applications

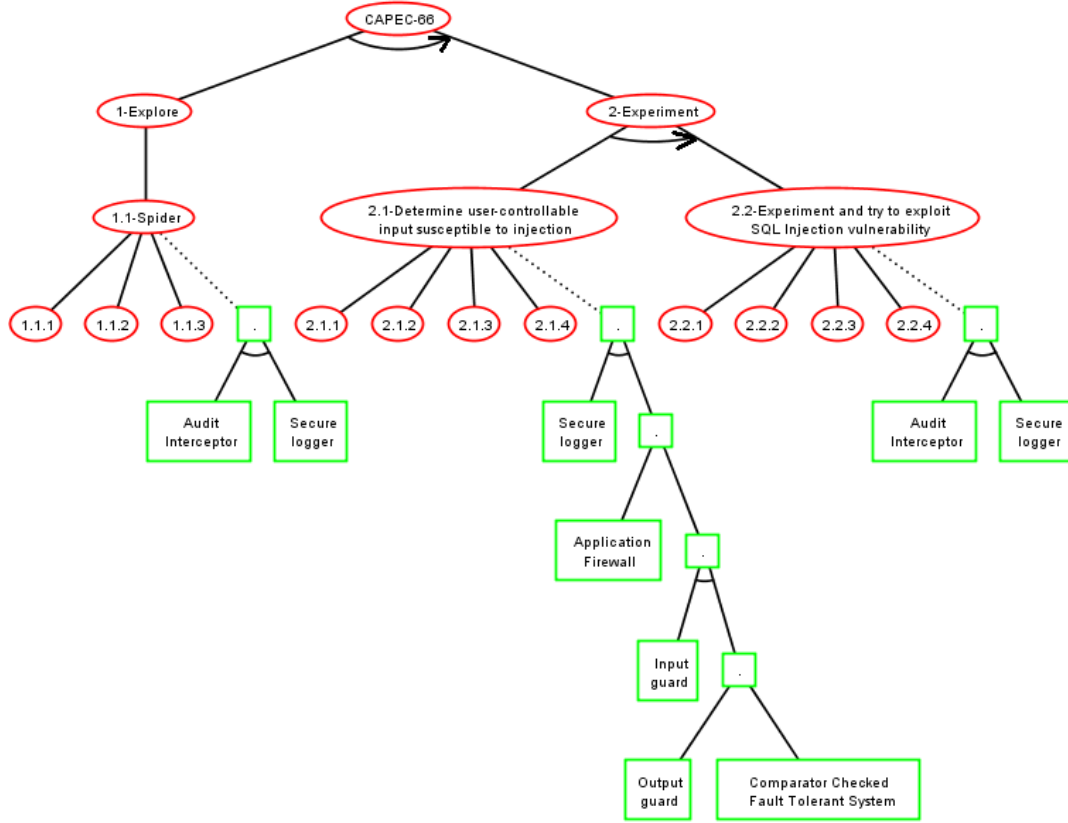


Fig. 5.4 CAPEC-66 ADTree

Step 3: Security pattern choice

In this step, the developer has to choose the security patterns with regard to the application context. As a result, each lowest node labeled by an attack step has to be related to a conjunction of security patterns. Technically, each ADTree $T(Att)$ can be loaded under ADTool and edited by the developer with regard to his/her preferences. This step yields ADTrees that are formalized with specific ADTerms:

Proposition 5.1. *An ADTree $T(Att)$ achieved by the steps 2 and 3 has an ADTerm $\iota(T(Att))$ having one of these forms:*

1. $\forall^P(t_1, \dots, t_n)$ with $t_i (1 \leq i \leq n)$ an ADTerm also having a form given in 1 or 2;
2. $\overrightarrow{\wedge^b}(t_1, \dots, t_n)$ with $t_i (1 \leq i \leq n)$ an ADTerm having the form given in 2 or 3;

3. $c^p(st, sp)$, with st and $ADTerm$ expressing an attack step and sp an $ADTerm$ expressing a security patterns combination.

The first $ADTerm$ expresses the description of an attack with more concrete ones. The second one represents sequences of attack steps. The last expression is composed of an attack step st refined with techniques, which can be prevented by a security patterns combination sp . In the remainder of this chapter, we call the last expression *Basic Attack Defense Step*, shortened as *BADStep*, which shall be particularly useful in the generation of GWT test case stubs.

Definition 5.2 (Basic Attack Defence Step (BADStep)). *A BADStep b is an $ADTerm$ of the form $c^p(st, sp)$, where st is an $ADTerm$ modeling an attack step and sp an $ADTerm$ of the form sp_1 or $\wedge^o(sp_1, \dots, sp_m)$ modeling a security pattern conjunction.*

$$defense(b) = \{sp_1 \mid b = c^p(st, sp_1)\} \cup \{sp_1, \dots, sp_m \mid b = c^p(st, \wedge^o(sp_1, \dots, sp_m))\}$$

Step 4: Final ADTree generation

In this step, each node labeled by an attack Att of the initial $ADTree$ T is substituted by an $ADTree$ $T(Att)$ given by Step 3. This can be done by substituting every term Att in the $ADTerm$ $\iota(T)$ by $\iota(T(Att))$. We denote T_f the resulting $ADTree$. It depicts a logical breakdown of the various options available to adversary and the defenses, materialized with security patterns, which have to be inserted in the application model.

In this step, we also extract from the data-store a description of the test architecture required to run the attack steps given by T_f on the application under test and to observe its reactions.

5.4.2 Test generation and execution

Step 5: Test suite generation

The semantics of an ADTree can be defined in terms of attack-defense scenarios. In general terms, a scenario is a minimal combination of events leading to the root attack, minimal in the sense that, if any event is omitted from the attack scenario, then the root goal will not be achieved. The semantics of an ADTree T_f , i.e., its scenario set, can be extracted from its ADTerm $\iota(T_f)$:

Definition 5.3 (Attack scenarios). *Let T_f be an ADTree and $\iota(T_f)$ be its ADTerm. The set of attack scenarios of T_f , denoted $SC(T_f)$ is the set of clauses of the disjunctive form of $\iota(T_f)$ over $BADStep(T_f)$.*

An attack scenario s of $SC(T_f)$ is an ADterm over $BADStep(s)$. $BADStep(s)$ denotes the set of BADSteps of s . We also denote $SP(s)$ the security patterns set found in s : $SP(s) = \{sp \mid \exists b \in BADStep(s) : sp \in defense(b)\}$. By extension, $BADStep(T_f)$ stands for the set of BADSteps found in $\iota(T_f)$; $SP(T_f)$ is the security patterns set of $\iota(T_f)$, found in all its scenarios.

Let's consider a security scenario $s \in SC(T_f)$. Given a BADStep $b = c^p(st, sp) \in BADStep(s)$, the approach generates the GWT test case $TC(b)$, which aims at checking whether the application under test is vulnerable to the attack step st and whether the consequences of the security pattern in $defense(b)$ can be detected from the behavior of the application. $TC(b)$ is assembled from the data-store by means of the following steps:

1. as illustrated in Figure 5.1, the data-store provides for each attack step st (with the relations $testG$, $testW$ and $testT$) one Given section, one When and one Then section. Each section is related to one procedure. The Then section aims at asserting the success of the attack step st on the application;
2. the data-store provides from the security patterns in $defense(b)$ a set of additional Then sections, each related to one procedure. These then sections aim at checking whether the

security pattern consequences can be observed in the application behavior (e.g., execution traces, graphical user interfaces, etc.);

3. all these sections are assembled to make up a GWT test case stub $TC(b)$.

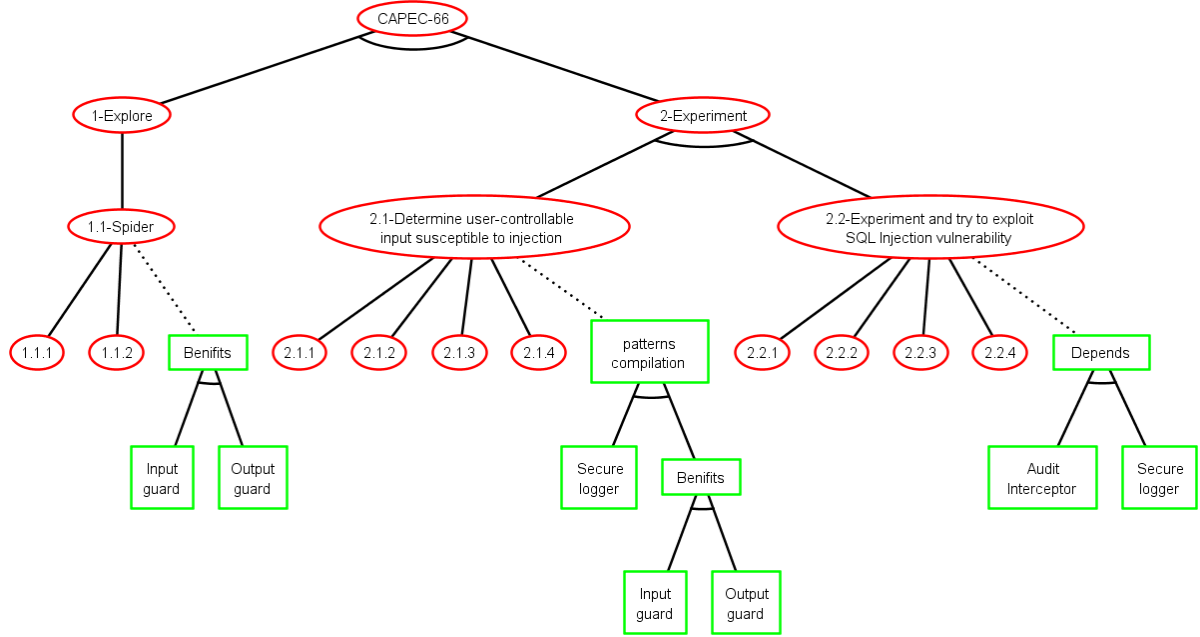


Fig. 5.5 CAPEC-66 ADTree after patterns choice

By applying theses steps on all the scenarios of $SC(T_f)$, we obtain the test suite TS with $TS = \{TC(b) | b = c^p(st, sp) \in BADstep(s) \text{ and } s \in SC(T_f)\}$.

```

Feature: CAPEC-66: SQL Injection
#1. Explore
Scenario: Step1.1 Survey application
#The attacker first takes an inventory of the functionality exposed by the application.
Given a new scanning session
When spider the application
Then the application is spidered
#assertions for security pattern testing
Then Output Guard security pattern is present
Then Input Guard security pattern is present

```

Fig. 5.6 A GWT test case Example

We implemented these steps to yield GWT test case stubs compatible with the Cucumber framework ⁴, which supports a large number of languages. Figure 5.6 shows a test case stub

⁴<https://cucumber.io/>

5.4 An approach for guiding developers devise more secure applications

example obtained with our tool from the first step of the attack CAPEC-66, depicted in Figure 5.5. The test case lists the Given When Then sections in a readable manner. Every section is associated to a generic procedure stored into an other file.

```
@When("^spider the application$")
public void theApplicationIsSpidered() {
    // Try one of the following techniques :
    // 1. Spider web sites for all available links
    // 2. Sniff network communications with application using a utility such
    //    as WireShark.

    getSpider().setMaxDepth(10);
    getSpider().setThreadCount(10);
    url = "http://localhost:8080";
    try {
        spider(url);
    } catch (InterruptedException e) {...}
    waitForSpiderToComplete();
}
```

Fig. 5.7 The procedure related to the When section of Figure 5.6

The procedure related to the When section is given in Figure 5.7. The comments are provided by the data-store and are extracted from the CAPEC base. This procedure includes a generic block of code which calls the penetration testing tool “ZAP proxy”. The “getSpider()” method, which belongs to the ZAP proxy API, calls a Web application crawler to cover the application under test in parallel (ten threads). The depth is the number of levels into the application that ZAP proxy will reach when looking for URLs. In this example, the root URL is “http://localhost:8080”.

Figure 5.8 illustrates an example of procedure for the Then section related to the pattern “Input Guard”. If an application is conceived with “Input Guard”, then the receipt of unexpected inputs should bring it to a quiescent state (observed with the HTTP status 503, 408) or it should return messages showing that errors have been detected. Furthermore, the fact that the application stops its execution or crashes (which can be observed with the HTTP status 500) is considered as a correct consequence for this pattern.


```
@Then("Input Guard security pattern is present")
public void assinputguard(){
    // in order to test the presence of input guard pattern
    //try to assert one of the following consequences :
    //no output or generic error message
    assertThat(app.getdriver().getPageSource(), anyOf(equals(""),
    containsString("error"),containsString("forbidden"),
    containsString("unauthorized")));
    //HTTP status(503, 408 for quiescent state, or for Unauthorized access)
    assertThat(con.getResponseCode(),anyOf(is(200),is(503),is(408)
    ,is(401),is(403),is(405),is(409)));
}
```

Fig. 5.8 The procedure related to the Then section (Input Guard) of Figure 5.6

Step 6: Test case stub completion

The developer has to complete the procedures related to the previous GWT test case stubs. We believe that the separation of the test case into sections and its link to the ADTree T_f (association among, steps, security patterns and procedures) make this step easier. In addition, the Generic procedures, composed of comments or blocks of code should make the developer more effective in the test case writing.

Step 7: Test suite execution

Once the GWT test case stubs are completed, these can be executed on the application under test denoted I . The test architecture allowing the experimentation of I is described in the report provided in Step 4.

After the execution of one test case $TC(b)$ on I , a test report is obtained. Figure 5.9 depicts an example of report obtained after the execution of the test case of Figure 5.6. This report shows that the second Then section has failed. This means that the consequence of the security pattern “Output Guard” is not detected from the application behavior.

To formalize these reports and to provide a variety of test verdicts, we here assume that the test cases are correctly developed with assertions in Then sections. As a test case $TC(b)$ is here

5.4 An approach for guiding developers devise more secure applications

```
<testcase classname="CAPEC-66: SQL Injection" name="Step1.1 Survey application" time="7.719405">
Given a new scanning session.....passed
When spider the application .....passed
Then the application is spidered.....passed
Then Output Guard security pattern is present.....failed
StackTrace:
java.lang.AssertionError:
Expected: is <true>
but: was <false>
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:20)
    at org.hamcrest.MatcherAssert.assertThat(MatcherAssert.java:8)
    at net.continuumsecurity.steps.AppScanningSteps.outputValide(AppScanningSteps.java:296)
    at *.Then output is validated(capec244.feature:14)
Then Input Guard security pattern is present.....skipped
```

Fig. 5.9 Test case Figure 5.6 result

composed of more than one assertion, its execution denoted $TC(b)||I$, may provide different sets of verdicts messages of the form :

- $\{Pass_{st}\}$ means that I is vulnerable to the attack step st although all the consequences of the security patterns are detected;
- $\{Fail_{st}\}$ means that I does not appear to be vulnerable to the attack step st and that the consequences of the security patterns are detected;
- $\{Pass_{st}, Fail_{sp_1}, \dots, Fail_{sp_k}\}$ shows I is vulnerable to the attack step st and that the consequences of some security patterns sp_1, \dots, sp_k are not detected;
- $\{Fail_{st}, Fail_{sp_1}, \dots, Fail_{sp_k}\}$ means I does not appear to be vulnerable to the attack step st but the consequences of the security patterns sp_1, \dots, sp_k are not detected.

Definition 5.4 (Test verdict sets). *Let I be an application under test and T_f be an ADTree. Let also $b = c^p(st, sp)$ be a BADStep of $BADstep(T_f)$ with $defense(b) = \{sp_1, \dots, sp_m\} (m > 0)$ and $TC(b) \in TS$ be test case.*

F stands for the lower set $P(\{Fail_{sp_i} | sp_i \in defense(b)\}) \setminus \phi$

The execution of $TC(b)$ on I leads to a verdict sets, which can be:

- $VUL = \{Pass_{st}\};$
- $NVUL = \{Fail_{st}\};$

- $VUL/VIOLATE = \{Pass_{st}\} \times F$;
- $NVUL/VIOLATE = \{Fail_{st}\} \times F$.

Now, we can define the verdicts obtained after the execution of one test case :

Definition 5.5 (Local test verdict). *Let I be an application under test, T_f an ADTree and $TC(b)$ a test case. The execution of $TC(b)$ on I is denoted $Verdict(TC(b)||I)$:*

$Verdict(TC(b)||I) = V$ with $V \in \{VUL, NVUL, VUL/VIOLATE, NVUL/VIOLATE\}$, iff $TC(b)||I$ returns a verdict set of V .

Subsequently, we define final test verdicts with regard to the ADTree T_f . These verdicts are given with the predicates $Vulnerable(T_f)$, $Unsat^c(SP(T_f))$ returning boolean values. $Vulnerable(b)$ is firstly defined to later apply a substitution $\sigma : BADStep(s) \rightarrow \{true, false\}$ on an attack-defense scenario s . The substitution σ maps each BADStep b of s onto the corresponding evaluation of $Vulnerable(b)$. A scenario holds if the evaluation ($eval(s\sigma)$) of applying the substitution σ to s (i.e. replacing every BADStep term b_i with the evaluation of $Vulnerable(b_i)$) returns true. When a scenario of T_f holds, then the threat modeled by T_f can be achieved on I . This is defined with the predicate $Vulnerable(T_f)$. $Unsat^c(SP(T_f))$ expresses whether the security pattern consequences are detected.

Definition 5.6 (Final test verdicts). *Let I be an application under test, T_f be an ADTree, and $s \in SC(T_f)$ a test scenario, with $BADStep(s) = \{b_1, \dots, b_n\}$.*

1. $Vulnerable(b) = true$ with $b \in BADStep(s)$, if $Verdict(TC(b)||I) \in \{VUL, VUL/VIOLATE\}$;
 $Vulnerable(b) = false$ otherwise;
2. $\sigma : BADStep(s) \rightarrow \{true, false\}$ is a substitution $\{b_1 \rightarrow (Vulnerable(b_1)), \dots, b_n \rightarrow Vulnerable(b_n)\}$;
3. $Vulnerable(T_f) = true$ if $\exists s \in SC(T_f) : eval(s\sigma)$ returns true, $Vulnerable(T_f) = false$ otherwise;

4. $Unsat^c(SP(T_f)) = true$ if $\exists b \in BADStep(T_f), Verdict(TC(b)||I) \in \{VUL/VIOLATE, NVUL/VIOLATE\}; Unsat^c(SP(T_f)) = false$ otherwise.

With regard to these verdicts, a set of corrective actions can be applied on the application model or code. Table 5.1 resumes some correctives for each combination of the values of $Vulnerable(T_f)$ and $Unsat^c(SP(T_f))$.

Table 5.1 Verdict Summary and solutions

$Vulnerable(T_f)$	$Unsat^c(SP(T_f))$	Corrective actions
false	false	no issue detected
true	false	At least one attack-defense scenario is successfully applied on the application. Fix the pattern instantiation or implementation. Or the chosen patterns are inconvenient, choose other patterns.
false	true	Some pattern consequences are not detected from the application behaviour. Check the pattern implementations. A pattern may be incorrectly implemented or another pattern conceals the consequences of the former.
true	true	The chosen security patterns are useless or incorrectly implemented. Check and fix the security patterns choice, models or implementation.

5.5 Evaluation

We empirically studied two scenarios on 24 participants to assess whether developers can take profit of our method. The duration of each scenario was set at most to one hour and half.

The participants are third to fourth year computer science undergraduate students; most of them have good skills in the design, development and test of Web applications (PHP, Javascript). They have some knowledge about classical attacks and are used to handle design patterns, but not security patterns.

The participants were given the task of choosing security pattern combinations to prevent two attacks, CAPEC 244: Cross-Site Scripting via Encoded URI Schemes and CAPEC 66:

SQL Injection, on two deliberately vulnerable Web applications, “*RopeyTasks*”⁵ and “*The Bodgeit Store*”⁶. We also asked the participants to write test cases with the tool Selenium in order to: show that both Web applications are vulnerable to the two attacks, test whether the application behaviors include at least one consequence of the security pattern “Input Guard” and at least one consequence of the security pattern “Output Guard”. As this last aspect was considered as difficult for beginners, we expected some assertions but not all the assertions.

In the first scenario, denoted Part 1, we supplied these documents to the students: the CAPEC base, two concrete examples detailing how to manually perform each attack along with the expected outcomes, and the security pattern catalog published by Yskout et al. [116], composed of 36 patterns. The participants had to: read the intents and consequences of the patterns, follow our examples of attacks and read the CAPEC base to write concrete test cases. In the second scenario, denoted Part 2, we supplied additional documents for the two attacks: the ADTrees of the two attacks (Figure 4.23 is one if them) and the generated *GWT* test case stubs (stored in Eclipse projects) for each step of the two attacks.

At the end of each scenario, the students were invited to fill in a form listing these questions:

- Q1: Was it difficult to choose security patterns?
- Q2: Was it difficult to use the CAPEC documentation (in Part 1) / our ADTrees (in Part 2)?
- Q3: Was it difficult to use the security pattern documents (in Part 1) / our ADTrees (in Part 2)?
- Q4: What was your time spent for choosing security patterns?
- Q5: How confident are you in your pattern choice?
- Q6: Provide your chosen security patterns

⁵<https://github.com/continuumsecurity/RopeyTasks>

⁶<https://github.com/psiinon/bodgeit>

- Q7: Was it easy to write test cases?
- Q8: What was the time you spent for writing test cases?
- Q9: How confident are you about the the test cases you wrote?
- Q10: Provide two test cases (or suites).

These questions was asked in order to evaluate the following criteria:

- C1: Comprehensibility: does our method ease the choice of the security patterns and the test case implementation?
- C2: Accuracy: are the chosen patterns and the given test cases correct?
- C3: Efficiency: does the use of our method reduces the time needed to choose patterns and to write tests?

5.5.1 Experiment Results

With the forms returned by the participants (available in [84]), we extracted the following results.

Firstly, Figure 5.10 illustrates the percentages of answers of the questions Q1 to Q3. For these, we proposed this four-valued scale: easy, fairly easy, difficult, very difficult. Similarly, we collected the answers of Question Q7 (yet on this four-valued scale). Figure 5.11 depicts the distribution of the participant opinions.

We collected the time spent by the participants for choosing patterns and writing test cases (in Part 1 and 2 of the experimentation). In summary, for the security pattern choice (Question Q4), response times varied between 15 and 60 minutes for Part 1, and between 4 and 30 minutes for Part 2. For the test case writing, the participants spent between 15 and 70 minutes in Part 1, while they took between 20 minutes and 86 minutes in Part 2.

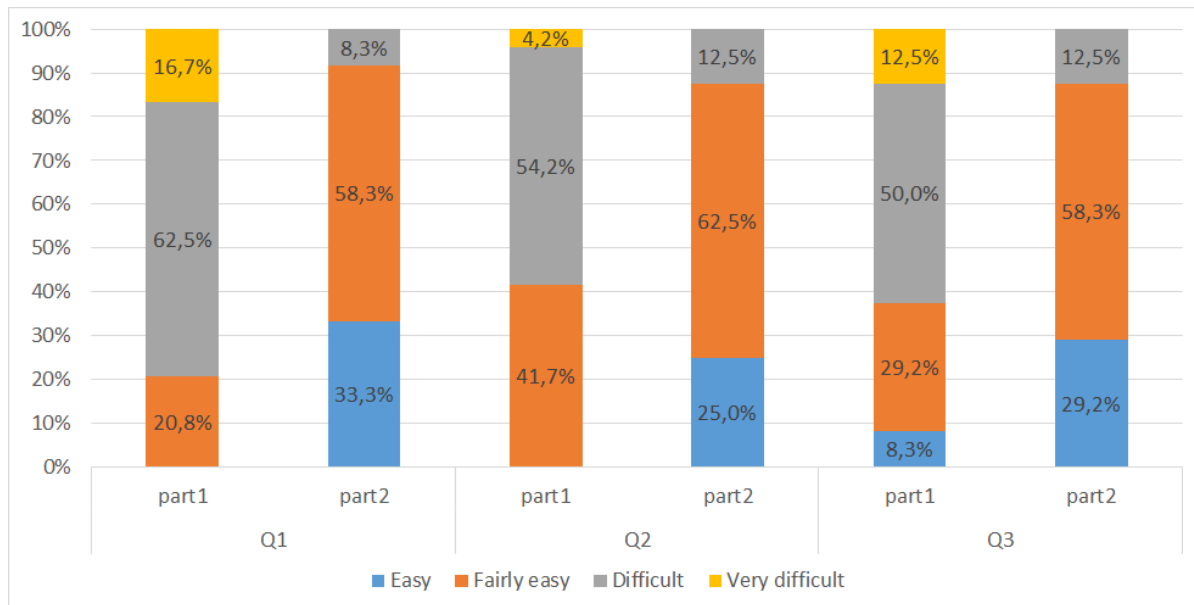


Fig. 5.10 Response rates for Q1 to Q3

The levels of confidence of the participants is estimated with Questions Q5 and Q9. The possible answers were for both scenarios: *very sure*, *sure*, *fairly sure*, *not sure*. Figure 5.12 depicts the percentages of answers of Question Q5 dealing with their confidences about their chosen set of security patterns and Figure 5.13 depicts the percentages of answers of Question Q9 dealing with their confidences about their test cases.

We analyzed the security pattern combinations provided by the participants in Question Q6. We organized these responses into four categories (ordered from the more to the less accurate):

1. **Correct:** several pattern combinations were accurate. When a participant gives one of these combinations, its response belongs to this category;
2. **Correct+Additional:** this category includes the responses composed of a correct pattern combination, completed with some other patterns;
3. **Missing:** we gather in this category, the incomplete pattern combinations without additional patterns;

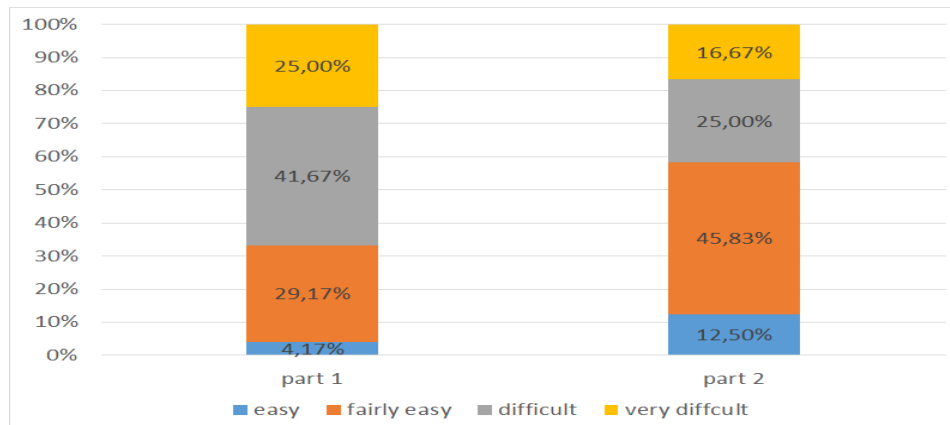


Fig. 5.11 Response rates for Q7

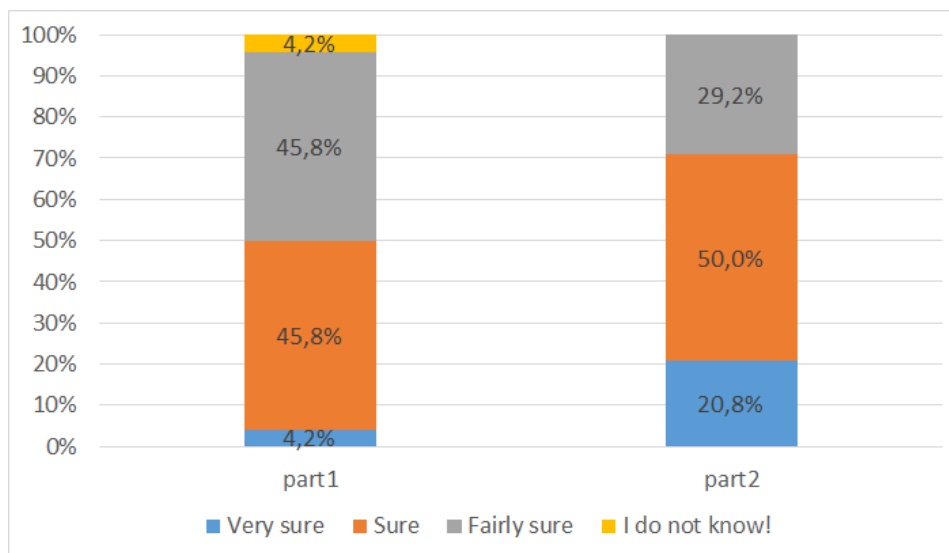


Fig. 5.12 Response rates for Q5

4. Missing+Additional: this category holds the worst responses, composed of unexpected patterns eventually accompanied with some expected ones.

With these categories, we obtained the bar charts of Figure 5.14, which gives the number of responses per category and per experiment scenario.

We finally analyzed the test cases given by the participants and evaluated their correctness with regard to four aspects:

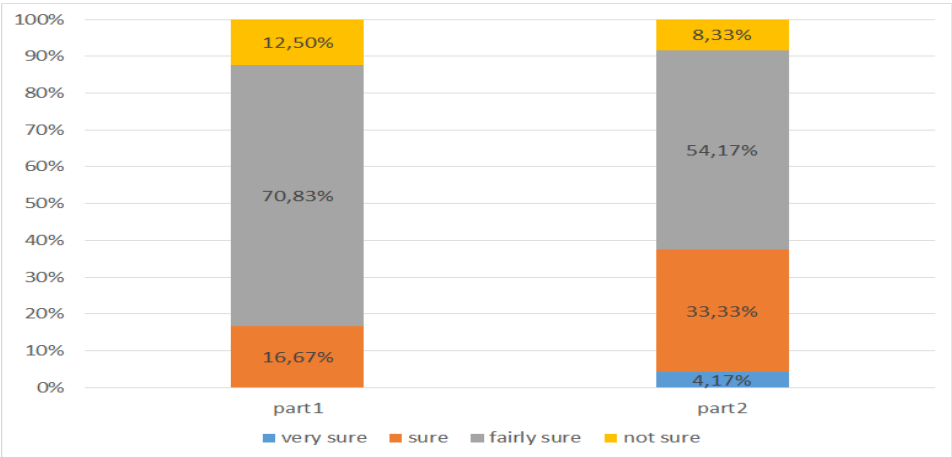


Fig. 5.13 Response rates for Q9

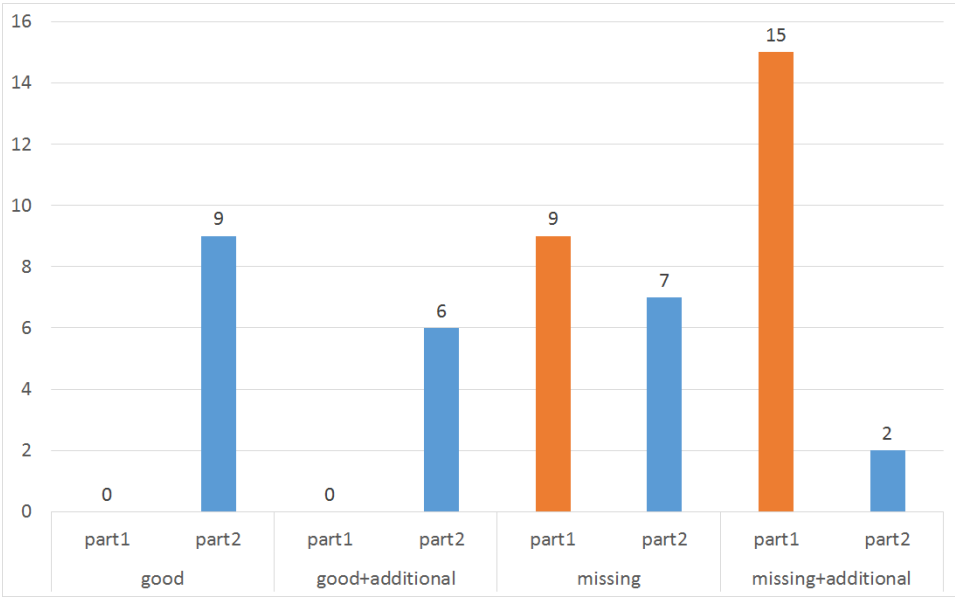


Fig. 5.14 Pattern choice correctness (Q6)

1. one test case or more show that the application is vulnerable to the attack CAPEC 66 (SQL injection);
2. one test case or more show that the application is vulnerable to the attack CAPEC 244 (SQL Cross site injection);
3. one test case or more detects that the application behavior does not include the consequences of the pattern “Input Guard”;

4. one test case or more detects that the application behavior does not include the consequences of the pattern “Output Guard”.

Figure 5.15 shows the number of participants per experiment scenario, who meet these aspects.

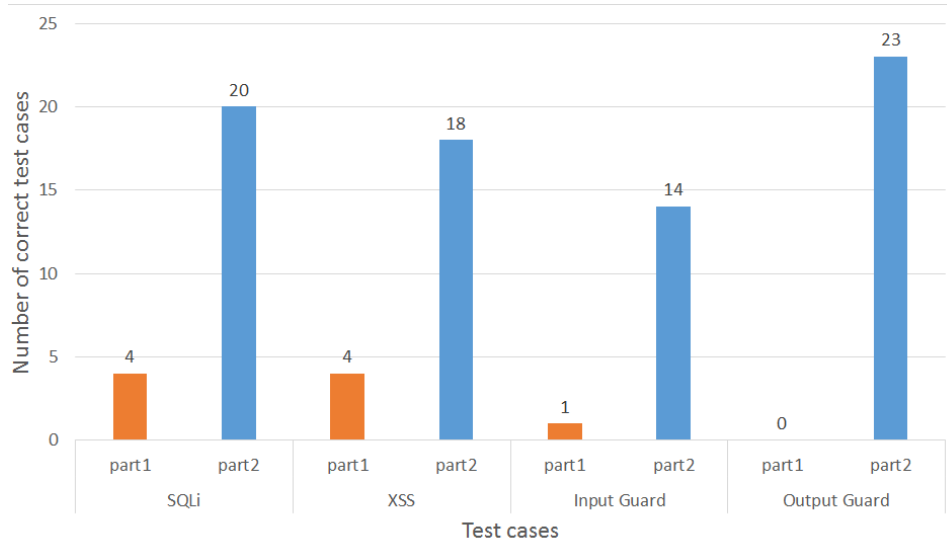


Fig. 5.15 Test case correctness (Q10)

5.5.2 Result interpretation

C1: Comprehensibility

We chose this criteria in order to evaluate how our methodology makes the patterns choice and the test implementation easier. Figure 5.10 shows that 33% of the participants estimated that the pattern choice was easy with our classification and ADTrees (Q1). In contrast, no participant found that the choice was easy when using only the security pattern catalog. The rate of “Easy” “Fairly Easy” increased by 70,8% between Part 1 and Part 2. With Question Q2, 41,7% of the participants found “Fairly easy” the use of the CAPEC base, whereas 87,5% esteemed our ADTrees “Easy” and “Fairly Easy” to use. Similarly, only 37,5% of the participants considered “Easy” and “Fairly easy” the reading of the pattern catalog. This rate reaches 87,5% with

our ADTrees. Consequently, Figure 5.10 shows that our generated ADTrees make the pattern choice easier and that they are simpler to interpret than the security pattern catalog. In addition, we observed that the confidence of the participants on their responses increased by 20,8%.

As for test cases, Figure 5.11 shows that 66,7% of the participants found that generating tests is “*difficult*” in the first part against 41,7% in the second part. Additionally, 4,17% of them found the test case development “*very easy*” in the first part against 12,5% in the second part. Hence one quarter of the students found easier the test writing with our test case stubs. After discussion with the participants, it turned out that the test case structure with the GWT template made test cases more readable and that the link of the test case sections with the attack steps of the ADTerms helped student understand what to develop.

Furthermore, Figure 5.13 shows that the confidence level of the participants about their test cases increases by 20,83%.

As a whole, we can conclude that the participants found their tasks easier with our ADTrees and GWT test case stubs thanks to the choice of modeling threats and counter-measures with graphical trees, and of organizing test case stubs w.r.t. attack steps, which provides readability and re-usability.

C2: Accuracy

Figure 5.14 reveals that no participant gave a correct pattern combination in Part 1. In contrast, when they used our ADTrees, the number of correct responses rises to 15 out of 24 (62,5%). Furthermore, the category of responses “Missing+Additional” (worst responses) is strongly reduced (62,5 % with Part 1 to 8% with Part 2). Consequently, the pattern choice is significantly more accurate with our ADTrees. Nonetheless, despite the use of our ADTrees, the number of participants that gave incomplete pattern combinations remains around in the same range (9 in Part 1, 7 in Part 2). More efforts are required to help designers not forget patterns in ADTrees.

Figure 5.15 depicts the results about the test case correctness. The columns “SQLi” and “XSS” provide the number of test cases allowing to reveal that both attacks can be successfully executed on the applications. In Part 1, few participants developed complete test cases. We indeed observed that assertions were missing in most of them. If we leave aside assertions, the number of test cases running the attacks rises to 14. The number of correct test cases strongly increases in Part 2 thanks to the comments the participants found in the procedures. The columns “Input Guard” and “Output Guard” give the number of Then sections (and procedures) allowing to show that the consequences of these security patterns are not detected from the application behaviors. This task was much more difficult for the students as they are not yet expert in security pattern. Hence, it is not surprising to see that only one student was able to write at least an assertion showing that the “Input Guard” consequences are not present. The number of correct Then sections rises to 14 in Part 2 thanks to the comments found in the procedures. With the pattern “Output guard”, the number of correct Then sections rises from 0 to 23 (almost all the participants) in Part 2. It worth noting here that this overly good result is due to the provided Generic procedure, which were composed of some blocks of code.

Consequently, we can conclude that the test cases given by the participants are more correct in Part 2, thanks to the amount of information (comments, blocks of code) found in the procedures.

C3: Efficiency

This criteria addresses the participant efficiency in terms of time spent for choosing security patterns and writing test cases. The average time spent by the participants for choosing patterns is equal to 32 minutes in the first scenario (Part 1). This time delay decreases to 14 minutes when the participants employed our ADTrees. Furthermore, no participants went over 30 minutes for choosing patterns in Part 2 (in contrast with 1 hour for Part 1). Hence, ADTrees, composed of security pattern combinations, make the participants more efficient in the patterns

choice. This is not really surprising as they only have to understand the ADTree terminology to deduce correct pattern combinations.

On average, the participants took 46 minutes for writing test cases from scratch and 60 minutes with the use of our method. We remark that they spent more time with the use of our method. This can be explained when we focus on the test case accuracy along side. Indeed, most of the test cases are not complete in Part 1, whereas almost all the test cases are correct in Part 2 (more assertions, etc.). They also took more time for choosing patterns in Part 1 than in Part 2, leaving less time for writing test cases. As most of them discovered for the first time the framework Cucumber, we also believe that they took the required time for understanding how it works.

As a consequence, these results show that the participants were more efficient for choosing patterns with ADTrees. We prefer not to conclude on the test case writing efficiency with this experimentation.

5.6 Chapter conclusion

We presented in this chapter a method taking advantage of data acquisition for guiding developers to devise more secure applications from the Threat modeling to testing stage. ADTrees and test case stubs are automatically generated allowing to check whether an application is vulnerable to attacks and whether security patterns consequences are detected from the application behavior.

In order to enhance the Comprehensibility, the Usefulness and the Accuracy of the method we take advantage of three notions:

1. the visual aspect provided by ADTrees in order to educate and help developers understand attacks, their execution flows and the security patterns related to each attack step;

2. the guidelines (comments) provided in generic procedures in addition to code blocks.
This considerably helps developers in the test cases completion. The richness of these guidelines is obtained thanks to the data store;
3. the automatic generation of attack scenarios, test cases from ADtrees and verdicts by means of a strict method.

We built a data-store gathering 215 attacks, 209 steps, 448 attack techniques, 217 counter-measure, 26 security patterns 43 security patterns consequences, 209 GWT test cases and 632 generic procedures covering a plethora of security problems for Web application context. The data-store and the tools for the generation of ADTrees and test cases are available in [84].

We evaluated our method on 24 participants to assess the criteria Comprehensibility, Efficiency, and Accuracy. The results show that, with our method, the participants are more accurate and have a better comprehension of attacks and security patterns. In addition, they felt more confident about their chosen security patterns and the test cases they implemented. This shows that our method and tools can be helpful in the software life cycle in order to integrate the notion of security patterns and to perform security testing.

The work presented in this chapter has been published in [91].

Chapter 6

Conclusions and perspectives

6.1 Summary of the achievements

We presented in Chapter 2 a set of notions, documents, models, tools along with an insight of the work proposed in the literature and related to this thesis manuscript.

The latter tackles the difficulty to devise secure applications all over the life cycle stages and proposes a set of approaches in order to help developers in the choice, the use and the testing of security solutions. These are provided by security patterns, which are abstract, generic and reusable solutions and “*relate countermeasures (stated in the solution) to threats and attacks in a given context*” [96]. The main objectives of the thesis are to :

1. perform a systematic verification of the instantiation of a security pattern set and their effectiveness against a security weakness;
2. establish security pattern classifications by means of a knowledge base to help designer/developer choose patterns;
3. help developers write threat models and concrete security test cases, again by means of a knowledge base.

Conclusions and perspectives

More precisely, in Chapter 3, we tackled the first objective and we introduced a method to help developers devise secure application models by assessing the instantiation of a security pattern set in a UML model to cure a given vulnerability set. We detailed the approach through a case study and we showed that a security pattern cannot be a “silver bullet” against a vulnerability and often has to be completed with other patterns. We also emphasized that it is difficult to choose the appropriate security pattern facing a security problem.

In Chapter 4, we introduced three methods for devising three multi-attribute classifications of security patterns. Three data-stores are built to associate security patterns, weaknesses, attacks and security principles. Security pattern classifications are automatically extracted from these data-stores, such that a weakness or an attack can be hindered by a set of inter-related security patterns. To enhance the readability of the classifications, we expose them graphically under the form of Security Activity Graphs (SAGs) and Attack defense trees (ADTrees). These visual supports make the security pattern choice easier. We also believe that they should promote the use of the security patterns in the Industry.

Even with security pattern classifications, developers still lack of guidance for writing and executing security test cases. Based on the data-stores developed in Chapter 4, we present in Chapter 5 a method to help developers in the Threat modeling and the writing and execution of security test cases. The purposes of the generated test cases are to assess whether an attack can be successfully executed on an application and to check whether security pattern consequences can be detected in the application’s behavior. We evaluated the method on 24 participants to assess its usefulness in terms of 3 criteria : 1) Comprehensibility; 2) Efficiency; 3) Accuracy. The results show that the participants are more accurate in the security pattern choice and write more accurate test cases. In addition, they felt more confident about their work, they better understand the functioning of the patterns and of the attacks. We concluded that the method is relevant to integrate security patterns throughout the application life cycle (Threat modeling, Test case generation, execution, etc.).

6.2 Towards a complete toolkit

The works introduced in this thesis are all accompanied with in a set of prototype tools. These are based on existing research tools and public resources, which allow to automate many steps of our approaches. However, our tools still require some manual steps. These manual steps often needed to associate together abstract data or notions with other data or formal expressions.

To overcome these limitations we present in this section some perspectives in order to devise a more complete toolkit.

6.2.1 Knowledge base completion

In Chapter 4, we constituted a knowledge base associating attacks, weaknesses, security patterns.etc. This is far from being exhaustive. A first perspective is to produce a larger base integrating a bigger number of security patterns, attacks, weaknesses, etc. The completion of the knowledge base requires manual steps. It could be interesting to investigate whether text mining techniques would help partially automate them. A bigger knowledge base gives a larger insight on security problems, solutions and application contexts.

The relations among patterns given in [116] are actually binary. It would be fruitful to provide relations among several patterns with regard to more elements (Specialization, Containment, etc.). In addition, the notion of composite patterns (i.e., the combination of a set of patterns to produce a new pattern) is not considered in this thesis manuscript.

In the security testing method introduced in Chapter 5, the generated test cases are implemented with generic procedures. These generic procedures are completed with comment. We completed some of them with code blocks calling penetration testing tools. We observed that a bigger number of generic procedures can be completed with code blocks thanks to the use of penetration testing tools. This would allow to :

Conclusions and perspectives

1. enlarge the use of the method on other application contexts (e.g., mobile, cloud, databases, networks, etc.);
2. educate developers about security testing with well-known tools ;
3. enhance the usability of the testing method introduced in Chapter 5 with a bigger number of generic GWT procedures.

6.2.2 Documentation Generation

In Chapters 4 and 5, we constituted a knowledge base associating several security notions. The structure of this base allows automated extractions of security pattern classifications. Currently, the extracted classifications are presented in tabular form, which limits its readability. We proposed to expose these classifications graphically with SAGs and ADTrees. With the growing number of test cases, test architectures, application contexts, etc. It appears important to provide a complete documentation allowing developers to better understand the notions gathered in the databases and the appropriate testing tools to use.

A perspective is to ease the navigation and the readability of the classifications by generating a documentation from the knowledge base in the form of Web pages. For instance, the presentation of an attack could expose the information about the attack, its related security patterns, its ADTree, test cases, tools to implement tests, etc.

In addition, we could complete the CAPEC and CWE databases with security patterns, SAGs and ADtrees. For instance, the current schema of the CAPEC database already provides a set of sections (e.g., Relevant Security Patterns, Non-Recommended Design Patterns, etc.), which relate CAPEC attacks and patterns. This illustrates the importance given to security patterns for documenting attacks. However, these sections are not yet fulfilled with information. We believe that our approach could be a first milestone to enrich the CAPEC database with security patterns, test cases, ADTrees, etc.

6.2.3 Runtime verification of security pattern properties in application traces

In the approach introduced in Chapter 3, we expressed the behavioral properties of security patterns with LTL formulas. A formula expresses an event sequence, which characterizes a behavioral property of a pattern. In Section 3.6, we detailed some limitations of the method, which are often related to the difficulty of writing LTL properties by developers. We introduced some LTL patterns (e.g., LTL response pattern in Figure 3.11) and an example of their use in order to express a security pattern strong point.

A perspective of this manuscript is to combine runtime verification and the methods proposed in Chapters 3, 5 to check whether security pattern behavioral properties hold in application traces. In the method introduced in Chapter 5, we test whether the consequences of a security pattern can be detected in an application behavior. But two security patterns can have the same consequences. We want here to complete the testing method proposed in Chapter 5 to determine with more accuracy which pattern has been used as follows. Given a security pattern Sp , an attack Att and an application A :

1. A is tested with Att by means of the method presented in Chapter 5 and traces of A are collected;
2. the behavioral properties of Sp are expressed with a set of generic LTL properties $P(Sp)$. These are manually derived with regard to the Sections “Participants”, “Collaborations” and “Dynamics” of the security pattern documentation. The LTL behavioral patterns¹ could help in this task;
3. Lemieux et al. [54] introduced the tool TEXADA², which dynamically mines LTL properties from application traces. It allows to enumerate all the instances of an LTL

¹<http://patterns.projects.cs.ksu.edu/documentation/patterns.shtml>

²<https://bitbucket.org/bestchai/texada/overview>

Conclusions and perspectives

property over the traces of an application. All the instances of a property $p \in P(Sp)$ in the traces obtained in step 1 are listed. This set of instances is denoted $I(p)$. The developer does not write LTL properties. He/she choose which property p of a pattern Sp he/she wants to check;

4. because of the generic nature of the LTL properties $P(Sp)$, the instances $I(p)$ do not all semantically correspond to Sp . We consider that an application satisfies a pattern property p if $\exists i \in I(p) : semcor(i, p)$ with $semcor(i, p)$ is true if i corresponds semantically to p .

It remains to define $semcor(i, p)$ and, in particular, to find a way to evaluate it. The generic and the textual nature of security patterns exposes the approach to some semantic challenges, we expose some of them in the next section.

The documents about security patterns, weaknesses, attacks, principles, etc. are often composed of texts. The information they provide are supplied by multiple authors who have different purposes and skills. It results in an heterogeneous documentation. Some works introduced the notion of “Security patterns language” [34, 36, 95]. Scumacher et al. [95] defined a pattern language as “*a network of tightly-interwoven patterns that defines a process for resolving a set of related, interdependent software development problems systematically*”. However, all the security patterns are not organized in this way. The current security pattern documentation lacks of standards to represent them. The proposed solutions are often very informal. Hence the need of a strict documentation that concretizes patterns in specific contexts emerges.

In the CAPEC and CWE bases, we are faced to the same challenge, which is often due to the multitude of the authors that contribute in fulfilling the database. Hence, the same security notion (e.g., security principle, mitigation, countermeasure, etc.) can be expressed differently, which increases the difficulties encountered by security designers and developers. We proposed the use of text mining techniques to group countermeasures. These techniques could be used to leverage some semantic challenges and automate more tasks.

6.3 Security pattern landscape

The growing number of security patterns makes them covering a plethora of security notions, problems, contexts, etc. However, their abstract nature makes their integration in more technical tasks very difficult. In this section, we present some perspectives for integrating security patterns in well-known security methods all over the application life.

6.3.1 Threat management with security patterns

The integration of security patterns in the threat modeling stage has been tackled in some papers [104, 103, 105, 34] and in Chapter 5, we presented how developers can be guided in the use of security patterns. We supposed that developers know the set of threats to which an application could be exposed. However, this is a strong assumption. STRIDE³ a threat modeling method that could help developers. It is provided with a plethora of tools and proved its effectiveness in both academic and industrial contexts. One of these tools, “*Microsoft Threat Analysis & Modeling*”⁴ allows threat modeling along with the architecture and the design of an application or a system. The application functional requirements are decomposed into components (e.g., roles, data, uses cases) that can be contextualized in many types of technologies. Then, with regard to these functional requirements, technical choices and a library of attacks, a set of threats is given and explained to the developer. Each threat is expressed with a factor and a set of possible attacks. A threat tree is automatically generated, which organizes hierarchically threats, attacks and the potential countermeasures. Figure 6.1 draws an example of Injection threat tree generated with this tool.

The hierarchy among all the attacks, threats and countermeasures is stored in an attack library and the user can upload his/her own one. An attack library is an xml file storing a hierarchical organization of attacks and countermeasures. In the data-stores presented in

³[https://msdn.microsoft.com/en-us/library/ee823878\(v=cs.20\).aspx](https://msdn.microsoft.com/en-us/library/ee823878(v=cs.20).aspx)

⁴<https://www.microsoft.com/en-us/download/details.aspx?id=14719>

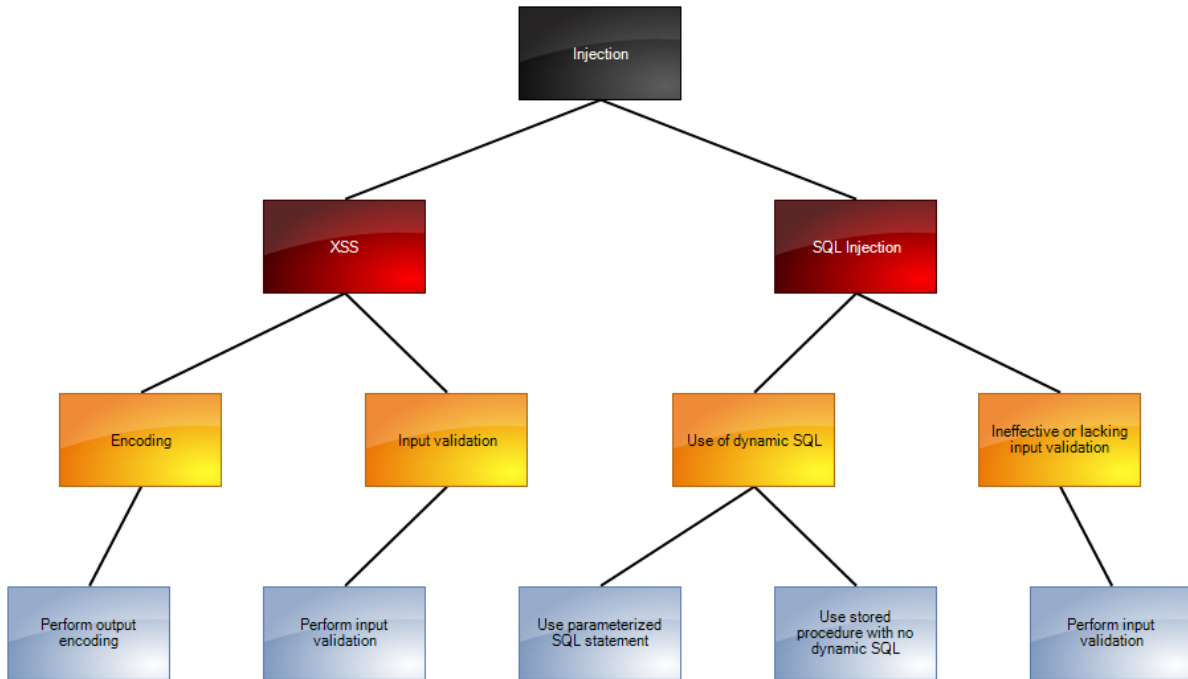


Fig. 6.1 Injection Threat Tree

Chapter 4, attacks, weaknesses, patterns are already hierarchically organized and our tools allow different kinds of extraction. Hence, it could be fruitful to connect our knowledge base with STRIDE threat model. This could be done by automatically generating these XML attack libraries from our base. Such a process would bring security patterns in STRIDE threat management (threat modeling and analysis) and more generally in threat modeling with few efforts. This approach could help developers identify threats (with regard to the functional choices) and devise the initial ADTree in the first step of the method introduced in Chapter 5.

6.3.2 Security reference architectures

Building a complex architecture of a system or a software (Functional and non functional requirements) is a hard task. If the application architecture is not well defined, it is almost impossible to determine which security solutions are required. To design application architectures, Fernandez et al. define a reference architecture “RA” as “*an abstract architecture that describes functionality without getting into implementation details*” [27]. “A ‘RA’ provides a

template-like solution that can be instantiated into a specific software architecture by adding implementation-oriented aspects” [26]. “RAs” prove their usefulness in both academical and professional contexts in order to understand, describe and build complex systems [27, 10, 61]. Security reference architectures “SRAs” are special “RAs” improved with security mechanisms, which are added in appropriate places in the “RA” [27]. In the literature, some “SRAs” have been proposed [18, 38, 27]. Fernandez et al. proposed in [27] a methodology for building a security reference architecture for cloud systems. They describe the “SRA” with UML schemes use patterns to build it. They indeed showed that it is possible to use security patterns to build “SRAs”.

Steel et al. presented in [100] a set of interrelated security patterns distributed over four component and logical tiers (Web tier, Business tier, Web service tier, Identity tier) to ensure an end to end security of web based application architectures. With regard to these works and our knowledge base, we believe that it would be possible to propose a supervised method to build “SRAs” with security patterns. “SRAs” can considerably help understand complex systems or applications and securing them with patterns. The ability of “RAs” of enhancing the visibility on the overall application is combined with the ability of security patterns in giving reusable solutions to build “SRAs”.

Furthermore, the approaches proposed in [26, 27] point that a “SRA” considerably help in the evaluation of the overall security of an application or system. They defined a set of misuse cases. They expressed, through an example (Publish a Malicious VM Image), the misuse cases with sequence diagrams. The approach we presented in Chapter 5 could be combined with “SRAs” building method to evaluate the overall application or system security in different application contexts.

6.4 Publications

The works presented in this thesis manuscript have been published in the following national and international journals and conferences proceedings:

1. Regainia, L., Bouhours, C., and Salva, S. (2015). Une démarche pour l'assistance à l'utilisation des patrons de sécurité. In 4ème Conférence en Ingénierie du Logiciel (CIEL);
2. Regainia, L., Bouhours, C., and Salva, S. (2016a). Systematic approach to assist designers in security pattern integration. In Second International Conference on Advances and Trends in Software Engineering (SOFTENG), Lisbon, Portugal;
3. Regainia, L., Salva, S., and Bouhours, C. (2016b). A classification methodology for security patterns to help fix software weaknesses. In 13th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2016, Agadir, Morocco, November 29 - December 2, 2016, pages 1–8;
4. Regainia, L., Salva, S., and Bouhours, C. (2016c). Une démarche pour l'assistance à l'utilisation des patrons de sécurité. *Technique et Science Informatiques*, 35(6):641–663;
5. Regainia, L. and Salva, S. (2017a). A methodology of security pattern classification and of attack-defense tree generation. In Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017, Porto, Portugal, February 19-21, 2017, pages 136–146;
6. Regainia, L., Bouhours, C., and Salva, S. (2017b). Un data-store pour la génération de cas de test. In 16èmes journées AFADL Approches Formelles dans l'Assistance au Développement de Logiciels GDR GPL DU 13 AU 16 JUIN 2017, MONTPELLIER;

7. Salva, S. and Regainia, L. (2017c). Using data integration to help design more secure applications. In Proceedings of the 12th International Conference on Risks and Security of Internet and Systems, Dinard, France;
8. Salva, S. and Regainia, L. (2017d). Using data integration for security testing. In IFIP International Conference on Testing Software and Systems, pages 178–194. Springer;

References

- [1] Aderhold, M., Cuéllar, J., Mantel, H., and Sudbrock, H. (2010). Exemplary formalization of secure coding guidelines. Technical report, Technical Report TUD-CS-2010-0060, TU Darmstadt, Germany.
- [2] Ahmed, T. and Tripathi, A. R. (2003). Static verification of security requirements in role based CSCW systems. *Proceedings of the eighth ACM symposium on Access control models and technologies - SACMAT '03*, page 196.
- [3] Al-Obeidallah, M. G., Petridis, M., and Kapetanakis, S. (2016). A survey on design pattern detection approaches. *International Journal of Software Engineering (IJSE)*, 7(3).
- [4] Alvi, A. K. and Zulkernine, M. (2011). A Natural Classification Scheme for Software Security Patterns. *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 113–120.
- [5] Alvi, Aleem, K. and Zulkernine, M. (2012). A Comparative Study of Software Security Pattern Classifications. *2012 Seventh International Conference on Availability, Reliability and Security*, pages 582–589.
- [6] Anand, P., Ryoo, J., and Kazman, R. (2014). Vulnerability-Based Security Pattern Categorization in Search of Missing Patterns. *2014 Ninth International Conference on Availability, Reliability and Security*, pages 476–483.
- [7] Arcelli Fontana, F. and Zanoni, M. (2011). A tool for design pattern detection and software architecture reconstruction. *Information Sciences*, 181(7):1306–1324.
- [8] Ardi, S., Byers, D., and Shahmehri, N. (2006). Towards a structured unified process for software security. In *Proceedings of the 2006 international workshop on Software engineering for secure systems*, pages 3–10. ACM.
- [9] Armando, A., Carbone, R., and Compagna, L. (2007). Ltl model checking for security protocols. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 385–396.
- [10] Avgeriou, P. (2003). Describing, instantiating and evaluating a reference architecture: A case study. *Enterprise Architect Journal*, 24.
- [11] Ballis, D., Baruzzo, a., and Comini, M. (2008). A Rule-based Method to Match Software Patterns Against UML Models. *Electronic Notes in Theoretical Computer Science*, 219:51–66.

References

- [12] Balser, M., Bäumler, S., and Knapp, A. (2004). Interactive verification of UML state machines. *Formal Methods and Software Engineering*, pages 434–448.
- [13] Bouhours, C. (2010). *Détection, Explications et Restructuration de défauts de conception : les patrons abîmés*. PhD thesis, l'Université Toulouse III – Paul Sabatier.
- [14] Bouhours, C., Leblanc, H., Percebois, C., and Millan, T. (2010). Detection of generic micro-architectures on models. In *Proceedings of PATTERNS 2010, The Second International Conferences on Pervasive Patterns and Applications*, pages 34–41, Lisbon, Portugal.
- [15] Buldas, A., Laud, P., Priisalu, J., Saarepera, M., and Willemson, J. (2006). Rational choice of security measures via multi-parameter attack trees. In *International Workshop on Critical Information Infrastructures Security*, pages 235–248. Springer.
- [16] Buschmann, F. (1996). *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley Series in Software Design Patterns. Wiley.
- [17] Byers, D., Ardi, S., Shahmehri, N., and Duma, C. (2006). Modeling software vulnerabilities with vulnerability cause graphs. In *Proceedings of the International Conference on Software Maintenance (ICSM06)*, pages 411–422.
- [18] Campbell, R. H., Montanari, M., and Farivar, R. (2012). A middleware for assured clouds. *Journal of Internet Services and Applications*, 3(1):87–94.
- [19] Corbett, J., Dwyer, M., and Hatcliff, J. (2000). A language framework for expressing checkable properties of dynamic software. *SPIN Model Checking and Software Verification*, pages 205–223.
- [20] Cortier, V., Delaitre, J., and Delaune, S. (2007). Safely composing security protocols. Research Report RR-6234, INRIA.
- [21] Cuppens, F., Cuppens-Boulahia, N., and Sans, T. (2005). Nomad: a security model with non atomic actions and deadlines. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 186–196.
- [22] Delessy, N. A. (2008). *A PATTERN-DRIVEN PROCESS FOR SECURE SERVICE-ORIENTED APPLICATIONS* by. PhD thesis.
- [23] Dreier, J., Giustolisi, R., Kassem, A., Lafourcade, P., and Lenzini, G. (2015). A framework for analyzing verifiability in traditional and electronic exams. In *Information Security Practice and Experience - 11th International Conference, ISPEC 2015, Beijing, China, May 5-8, 2015. Proceedings*, pages 514–529.
- [24] Edge, K. S., Dalton, G. C., Raines, R. A., and Mills, R. F. (2006). Using attack and protection trees to analyze threats and defenses to homeland security. In *Military Communications Conference, 2006. MILCOM 2006. IEEE*, pages 1–7. IEEE.
- [25] Felderer, M., Büchler, M., Johns, M., Brucker, A. D., Breu, R., and Pretschner, A. (2016). Chapter one-security testing: A survey. *Advances in Computers*, 101:1–51.
- [26] Fernandez, E. B. and Monge, R. (2014). A security reference architecture for cloud systems. In *Proceedings of the WICSA 2014 Companion Volume*, page 3. ACM.

-
- [27] Fernandez, E. B., Monge, R., and Hashizume, K. (2016). Building a security reference architecture for cloud systems. *Requirements Engineering*, 21(2):225–249.
- [28] Fernandez, E. B., Washizaki, H., Yoshioka, N., Kubo, A., and Fukazawa, Y. (2008). Classifying security patterns. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4976 LNCS, pages 342–347.
- [29] FERRO CASTRO, B. (1969). Pattern-oriented software architecture: A system of patterns. *Computación y Sistemas*, 1(002).
- [30] Flechais, I., Mascolo, C., and Sasse, M. A. (2007). Integrating security and usability into the requirements and design process. *Int. J. Electron. Secur. Digit. Forensic*, 1(1):12–26.
- [31] Gadyatskaya, O., Jhawar, R., Kordy, P., Lounis, K., Mauw, S., and Trujillo-Rasua, R. (2016). Attack trees for practical security assessment: ranking of attack scenarios with adtool 2.0. In *International Conference on Quantitative Evaluation of Systems*, pages 159–162. Springer.
- [32] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). Design patterns: elements of.
- [33] Gerth, R., Peled, D., Vardi, M. Y., and Wolper, P. (1996). Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification, Testing and Verification XV*, pages 3–18. Springer.
- [34] Hafiz, M. (2013). A pattern language for developing privacy enhancing technologies. *Software: Practice and Experience*, 43(7):769–787.
- [35] Hafiz, M. (2015). Security pattern catalog.
- [36] Hafiz, M., Adamczyk, P., and Johnson, R. E. (2007). Organizing security patterns. *IEEE software*, 24(4).
- [37] Hafiz, M. and Johnson, R. E. (2006). Security Patterns and their Classification Schemes. pages 1–25.
- [38] Hafner, M., Memon, M., and Breu, R. (2009). Seaas-a reference architecture for security services in soa. *J. UCS*, 15(15):2916–2936.
- [39] Hamid, B., Geisel, J., Ziani, A., Bruel, J., and Pérez, J. (2013). Model-driven engineering for trusted embedded systems based on security and dependability patterns. In *SDL 2013: Model-Driven Dependability Engineering - 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings*, pages 72–90.
- [40] Hansman, S. and Hunt, R. (2005). A taxonomy of network and computer attacks. *Computers & Security*, 24(1):31–43.
- [41] Heuzeroth, D., Holl, T., Hogstrom, G., and Lowe, W. (2003). Automatic design pattern detection. *MHS2003. Proceedings of 2003 International Symposium on Micromechatronics and Human Science (IEEE Cat. No.03TH8717)*, pages 94–103.

References

- [42] Holzmann, G. (2003). *Spin Model Checker, the: Primer and Reference Manual*. Addison-Wesley Professional, first edition.
- [43] Ingalsbe, J. A., Kunimatsu, L., Baeten, T., and Mead, N. R. (2008). Threat modeling: diving into the deep end. *IEEE software*, 25(1).
- [44] Jhawar, R., Kordy, B., Mauw, S., Radomirović, S., and Trujillo-Rasua, R. (2015). Attack trees with sequential conjunction. In *IFIP International Information Security Conference*, pages 339–353. Springer.
- [45] Jürgenson, A. and Willemson, J. (2009). Serial model for attack tree computations. In *International Conference on Information Security and Cryptology*, pages 118–128. Springer.
- [46] Kaczor, O., Guéhéneuc, Y. G., and Hamel, S. (2010). Identification of design motifs with pattern matching algorithms. *Information and Software Technology*, 52(2):152–168.
- [47] Kienzle, D., Elder, M., Tyree, D., and Edwards-Hewitt, J. (2002). Security patterns repository version 1.0. *DARPA, Washington DC*.
- [48] Konrad, S., Cheng, B. H., Campbell, L. a., and Wassermann, R. (2003). Using Security Patterns to Model and Analyze Security Requirements. *2nd International Workshop on Requirements Engineering for High Assurance Systems*, pages 13–22.
- [49] Kordy, B., Kordy, P., Mauw, S., and Schweitzer, P. (2013). Adtool: security analysis with attack–defense trees. In *International Conference on Quantitative Evaluation of Systems*, pages 173–176. Springer.
- [50] Kordy, B., Mauw, S., Radomirović, S., and Schweitzer, P. (2010). Foundations of attack–defense trees. In *International Workshop on Formal Aspects in Security and Trust*, pages 80–95. Springer.
- [51] Kordy, B., Mauw, S., Radomirović, S., and Schweitzer, P. (2012). Attack–defense trees. *Journal of Logic and Computation*, page exs029.
- [52] Kordy, B., Piètre-Cambacédès, L., and Schweitzer, P. (2014). Dag-based attack and defense modeling: Don’t miss the forest for the attack trees. *Computer Science Review*, 13–14:1 – 38.
- [53] Laverdiere, M., Mourad, A., Hanna, A., and Debbabi, M. (2006). Security design patterns: Survey and evaluation. In *Electrical and Computer Engineering, 2006. CCECE’06. Canadian Conference on*, pages 1605–1608. IEEE.
- [54] Lemieux, C., Park, D., and Beschastnikh, I. (2015). General LTL specification mining. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, Lincoln, NE, USA.
- [55] Liu, Y. and Man, H. (2005). Network vulnerability assessment using bayesian networks. In *Defense and Security*, pages 61–71. International Society for Optics and Photonics.
- [56] Mallouli, W., Mammar, A., and Cavalli, A. (2009a). A formal framework to integrate timed security rules within a tefsm-based system specification. In *Software Engineering Conference, 2009. APSEC’09. Asia-Pacific*, pages 489–496. IEEE.

- [57] Mallouli, W., Mammar, A., and Cavalli, A. R. (2009b). A formal framework to integrate timed security rules within a tefsm-based system specification. In *16th Asia-Pacific Software Engineering Conference, (ASPEC'09)*.
- [58] Marback, A., Do, H., He, K., Kondamarri, S., and Xu, D. (2009). Security test generation using threat trees. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 62–69. IEEE.
- [59] Marback, A., Do, H., He, K., Kondamarri, S., and Xu, D. (2013). A threat model-based approach to security testing. *Software: Practice and Experience*, 43(2):241–258.
- [60] Meadows, C. (1998). A representation of protocol attacks for risk assessment. In *Proceedings of the DIMACS Workshop on Network Threats*, pages 1–10.
- [61] Medvidovic, N. and Taylor, R. N. (2010). Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 471–472. ACM.
- [62] Meier, J. (2006). Web application security engineering. *Security & Privacy, IEEE*, 4(4):16–24.
- [63] Merz, S. and Rauh, C. (2002). Model checking timed uml state machines and collaborations. In *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002)*, pages 395–414.
- [64] Meszaros, G. and Doble, J. (1997). A pattern language for pattern writing. In *Proceedings of International Conference on Pattern languages of program design (1997)*, volume 131, page 164.
- [65] Miede, A., Nedyalkov, N., Gottron, C., König, A., Repp, N., and Steinmetz, R. (2010). A generic metamodel for it security attack modeling for distributed systems. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 430–437. IEEE.
- [66] Millan, T., Sabatier, L., Le Thi, T. T., Bazex, P., and Percebois, C. (2009). An ocl extension for checking and transforming uml models. In *proceedings of the 8th International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS)*, pages 144–150, <http://www.wseas.org/>. WSEAS Press. (Invited speaker).
- [67] Mitre corporation (2017a). Common attack pattern enumeration and classification, <https://capec.mitre.org/>.
- [68] Mitre corporation (2017b). Common vulnerabilities and exposures, <http://cve.mitre.org/>.
- [69] Mitre corporation (2017c). Common weakness enumeration, <https://cwe.mitre.org/>.
- [70] Morais, A., Martins, E., Cavalli, A., and Jimenez, W. (2009). Security protocol testing using attack trees. In *Computational Science and Engineering, 2009. CSE'09. International Conference on*, volume 2, pages 690–697. IEEE.
- [71] Motii, A., Hamid, B., Lanusse, A., and Bruel, J.-M. (2015). Guiding the selection of security patterns based on security requirements and pattern classification. In *Proceedings of the 20th European Conference on Pattern Languages of Programs*, page 10. ACM.

References

- [72] Motii, A., Hamid, B., Lanusse, A., and Bruel, J.-M. (2016a). Guiding the selection of security patterns for real-time systems. In *Engineering of Complex Computer Systems (ICECCS), 2016 21st International Conference on*, pages 155–164. IEEE.
- [73] Motii, A., Hamid, B., Lanusse, A., and Bruel, J.-M. (2016b). Towards the integration of security patterns in uml component-based applications. In *PAME/VOLT@ MODELS*, pages 2–6.
- [74] Mouratidis, H., Giorgini, P., and Manson, G. (2005). When security meets software engineering. *Inf. Syst.*, 30(8):609–629.
- [75] Murtagh, F. and Legendre, P. (2014). Ward’s hierarchical agglomerative clustering method: Which algorithms implement ward’s criterion? *Journal of Classification*, 31(3):274–295.
- [76] OWASP (2003). Owasp testing guide v3.0 project. In http://www.owasp.org/index.php/Category:OWASP_Testing_Project#OWASP_Testing_Guide_v3.
- [77] Peine, H., Jawurek, M., and Mandel, S. (2008). Security goal indicator trees: A model of software features that supports efficient security inspection. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 9–18. IEEE.
- [78] Potter, B. and McGraw, G. (2004). Software security testing. *IEEE Security & Privacy*, 2(5):81–85.
- [79] Qin, X. and Lee, W. (2004). Attack plan recognition and prediction using causal networks. In *Computer Security Applications Conference, 2004. 20th Annual*, pages 370–379. IEEE.
- [80] Ray, I. and Poolsapassit, N. (2005). Using attack trees to identify malicious attacks from authorized insiders. In *European Symposium on Research in Computer Security*, pages 231–246. Springer.
- [81] Regainia, L., Bouhours, C., and Salva, S. (2015). Une démarche pour l’assistance à l’utilisation des patrons de sécurité. In *4ème Conférence en Ingénierie du Logiciel (CIEL)*.
- [82] Regainia, L., Bouhours, C., and Salva, S. (2016a). Systematic approach to assist designers in security pattern integration. In *Second International Conference on Advances and Trends in Software Engineering (SOFTENG)*, Lisbon, Portugal.
- [83] Regainia, L. and Salva, S. (2017a). A methodology of security pattern classification and of attack-defense tree generation. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, ICISSP 2017, Porto, Portugal, February 19-21, 2017.*, pages 136–146.
- [84] Regainia, L. and Salva, S. (2017b). Security pattern classification, companion site <http://regainia.com/research/companion.html>.
- [85] Regainia, L., Salva, S., and Bouhours, C. (2016b). A classification methodology for security patterns to help fix software weaknesses. In *13th IEEE/ACS International Conference of Computer Systems and Applications, AICCSA 2016, Agadir, Morocco, November 29 - December 2, 2016*, pages 1–8.

-
- [86] Regainia, L., Salva, S., and Bouhours, C. (2016c). Une démarche pour l'assistance à l'utilisation des patrons de sécurité. *Technique et Science Informatiques*, 35(6):641–663.
- [87] Rocky, S. (2017). Security patterns repository.
- [88] Rojas, J. M., Fraser, G., and Arcuri, A. (2015). Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 338–349. ACM.
- [89] Roy, A., Kim, D. S., and Trivedi, K. S. (2010). Act: Attack countermeasure trees for information assurance analysis. In *INFOCOM IEEE Conference on Computer Communications Workshops, 2010*, pages 1–2. IEEE.
- [90] Saltzer, J. H. and Schroeder, M. D. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308.
- [91] Salva, S. and Regainia, L. (2017a). Using data integration for security testing. In *IFIP International Conference on Testing Software and Systems*, pages 178–194. Springer.
- [92] Salva, S. and Regainia, L. (2017b). Using data integration to help design more secure applications. In *Proceedings of the 12th International Conference on Risks and Security of Internet and Systems*, Dinard, France.
- [93] Sanjeevan, K., Cruellas, J., Canals, A., Millan, T., Chiorean, D., Mullet, M., and Robert, M. (2001). Neptune-an integrated uml toolset and methodology. In *Information Technology Interfaces, 2001. ITI 2001. Proceedings of the 23rd International Conference on*, pages A11–A12. IEEE.
- [94] Scambray, J. and Olson, E. (2003). *Improving Web Application Security*.
- [95] Schumacher, M. and Fernandez-Buglioni, E. (2006). *Security Patterns: Integrating security and systems engineering*.
- [96] Schumacher, M. and Roedig, U. (2001). Security Engineering with Patterns. *Engineering*, 2754:1–208.
- [97] Serkan, z. (2017). Cve details security vulnerability datasource.
- [98] Shahmehri, N., Mammar, A., De Oca, E. M., Byers, D., Cavalli, A., Ardi, S., and Jimenez, W. (2012). An advanced approach for modeling and detecting software vulnerabilities. *Information and Software Technology*, 54(9):997–1013.
- [99] Sommestad, T., Ekstedt, M., and Johnson, P. (2009). Cyber security risks assessment with bayesian defense graphs and architectural models. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE.
- [100] Steel, C., Nagappan, R., and Lai, R. (2006). *Core security patterns*.
- [101] Stytz, M. R. (2004). Considering Defense in Depth for Software Applications. *IEEE Security and Privacy*, 2(1):72–75.

References

- [102] Tøndel, I. A., Jensen, J., and Røstad, L. (2010). Combining misuse cases with attack trees and security activity models. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 438–445. IEEE.
- [103] Uzunov, A. V. and Fernandez, E. B. (2014a). An extensible pattern-based library and taxonomy of security threats for distributed systems. *Computer Standards and Interfaces*, 36(4):734–747.
- [104] Uzunov, A. V. and Fernandez, E. B. (2014b). An extensible pattern-based library and taxonomy of security threats for distributed systems. *Computer Standards & Interfaces*, 36(4):734–747.
- [105] Uzunov, A. V., Fernandez, E. B., and Falkner, K. (2012). Securing distributed systems using patterns: A survey. *Computers & Security*, 31(5):681–703.
- [106] Viega, J. and McGraw, G. (2001). *Building Secure Software: How to Avoid Security Problems the Right Way, Portable Documents*. Pearson Education.
- [107] Vlissides, J., Helm, R., Johnson, R., and Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11.
- [108] Wassermann, R. and Cheng, B. H. (2003). Security patterns. In *Michigan State University, PLoP Conf*. Citeseer.
- [109] Weiss, J. D. (1991). A system security engineering process. In *Proceedings of the 14th National Computer Security Conference*, volume 249, pages 572–581.
- [110] Wiesauer, A. and Sametinger, J. (2009). A security design pattern taxonomy based on attack patterns. In *International Joint Conference on e-Business and Telecommunications*, pages 387–394.
- [111] Willett, P. (1988). Recent trends in hierarchic document clustering: a critical review. *Information Processing & Management*, 24(5):577–597.
- [112] Yoder, J. and Barcalow, J. (1998). Architectural patterns for enabling application security. *Urbana*, 51:61801.
- [113] Yoon, K. P. and Hwang, C.-L. (1995). *Multiple attribute decision making: an introduction*, volume 104. Sage publications.
- [114] Yoshioka, N., Washizaki, H., and Maruyama, K. (2008). A survey on security patterns. *Progress in Informatics*, 5:35–47.
- [115] Yoshizawa, M., Kobashi, T., Washizaki, H., Fukazawa, Y., Okubo, T., Kaiya, H., and Yoshioka, N. (2014). Verifying implementation of security design patterns using a test template. In *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, pages 178–183. IEEE.
- [116] Yskout, K., Heyman, T., Scandariato, R., and Joosen, W. (2006). A system of security patterns.

- [117] Yskout, K., Scandariato, R., and Joosen, W. (2012). Does organizing security patterns focus architectural choices? *Proceedings - International Conference on Software Engineering*, pages 617–627.
- [118] Zaïane, O. R. (1999). Introduction to data mining.
- [119] Ziani, A., Hamid, B., Geisel, J., and Bruel, J. (2013). A model-based repository of security and dependability patterns for trusted RCES. In *IEEE 14th International Conference on Information Reuse & Integration, IRI 2013, San Francisco, CA, USA, August 14-16, 2013*, pages 448–457.

Abstract

Assisting in the development and testing of secure applications

by Loukmen REGAINIA

Ensuring the security of an application through its life cycle is a tedious task. The choice, the implementation and the evaluation of security solutions is difficult and error prone. Security skills are not common in development teams. To overcome the lack of security skills, developers and designers are provided with a plethora of documents about security problems and solutions (i.e, vulnerabilities, attacks, security principles, security patterns, etc.). Abstract and informal, these documents are provided by different sources, and their number is constantly growing. Developers are drown in a sea of documentation, which inhibits their capacity to design, implement, and the evaluate the overall application security. This thesis tackles these issues and presents a set of approaches to help designers in the choice, the implementation and the evaluation of security solutions required to overcome security problems. The problems are materialized by weaknesses, vulnerabilities, attacks, etc. and security solutions are given by security patterns.

This thesis first introduces a method to guide designers implement security patterns and assess their effectiveness against vulnerabilities. Then, we present three methods associating security patterns, attacks, weaknesses, etc. in a knowledge base. This allows automated extraction of classifications and help designers quickly and accurately select security patterns required to cure a weakness or to overcome an attack. Based on this knowledge base, we detail a method to help designers in threat modeling and security test generation and execution. The method is evaluated and results show that the method enhances the comprehensibility and the accuracy of developers in the security solutions choice, threat modeling and in the writing of security test cases.

Keywords. Security patterns, weaknesses, attacks, principles, life cycle, model checking, security testing.