A Systematic Approach to Assist Designers in Security Pattern Integration

Loukmen Regainia, Cédric Bouhours, Sébastien Salva

A Systematic Approach to Assist Designers in Security Pattern Integration

Loukmen Regainia*, Cédric Bouhours† and Sébastien Salva‡
LIMOS - UMR CNRS 6158
Auvergne University, France
Email: * loukmen.regainia@udamail.fr, † cedric.bouhours@udamail.fr, ‡ sebastien.salva@udamail.fr

***Abstract*—The last decade has witnessed significant contributions in software engineering to design more secure systems and applications. Software designers can now leverage specific patterns, called *security patterns* as reusable solutions to model more secure applications. But, despite the advantages offered by security patterns, these are rarely used in practice, because choosing and employing them for devising less vulnerable applications, is still a difficult and error-prone task. In this work, we propose an original approach to guide designers for checking whether a set of security patterns is correctly integrated into models and if vulnerabilities are yet exposed despite their use. 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, we check whether the vulnerabilities, which should be removed by the use of patterns, are not exposed in the model. We illustrate this approach on an example of Web application, the *Moodle education platform*.**

Keywords*—*Model; UML; Security Patterns; Verification

I. Introduction

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 on the other hand, devising them is a difficult task.

To solve this issue, a large set of papers and tools have been proposed to help the integration of security in the software engineering steps [1][2]. Among them, the pattern community proposed the notion of security patterns as reusable solutions to security issues in the modeling stage [3]. Specifically, a pattern represents a structure, a behavior, or some intents that have to be applied in models to meet security properties or to prevent threats (sometimes partially). At present, the security pattern base holds hundreds of available patterns, more and less detailed and compatible with each other. Many of them are indeed described with text only, their contextualization (a.k.a. instantiation) being left to designers. Furthermore, the impact of their composition is often unknown. Hence, the choice of the relevant patterns and their inclusion in models is yet onerous and error-prone, even for experts. This paper focuses on these difficulties and proposes an approach for helping designers to integrate the appropriate patterns and to design more secure applications. This approach corresponds to a sequential process, whose main benefits are twofold: measuring the integration quality of security patterns in models, and checking if these models are yet vulnerable to attacks despite the use of patterns.

More precisely, the designer initially chooses a list of vulnerabilities that must not be found in the application model and a set of security patterns that should correct these vulnerabilities or prevent attackers from exploiting them. The proposed formal method-based process firstly aims at helping the designer to integrate each pattern in the application model: we check whether the structural and behavioral properties of the pattern hold in the model by means of a method based on OCL (Object Constraint Language) queries and a verification technique. The former tries to locate the pattern shape in the model and returns a coefficient of disclosure. The verification technique checks whether some behavioral properties of the security pattern, expressed with LTL (Linear Temporal Logic) formulas, hold in the model. Then, quality metrics are computed to evaluate the integration quality of each pattern and of all the patterns in the model. If these metrics are low, the model should be revisited. In a second stage, the process also checks whether each vulnerability can be detected in the model by means of a verification technique. Actually, despite the use of security patterns, a vulnerability may be still present on account of several reasons, e.g., the use of an incomplete or wrong list of security patterns, or the composition of several patterns in the model that may induce a vulnerability. Vulnerabilities are formally expressed with LTL properties that are given to a model-checker. If any property is satisfied in the model, the designer is then warned that the latter still includes vulnerabilities and requires modifications.

We illustrate the benefits of this approach by applying it on a case study, based on Moodle education platform and on the vulnerability *Code injection*, which is a well known flaw allowing to inject code that is then interpreted by the application [4].

The paper is structured as follows: we briefly present the background and motivations in Section II. The approach is described in Section III. Its illustration on a Web application example is given in Section IV. Finally, we draw conclusions and perspectives for future work in Section V.

II. Background

As the number of available security patterns is continuously growing, choosing the appropriate ones, to design more secure applications, is more and more tedious and error-prone. To help designers in this task, several papers proposed classifications and taxonomies to organize security patterns. Based on the STRIDE threat management methodology, Munawar et

al. presented an organization of 97 security patterns on three architectural layers (Core, Exterior, Perimeter) [5]. In addition, Alvi et al. proposed in [6] a natural scheme for classifying security patterns. They associated their classification to the phases of software life cycle, i.e., security objectives in the requirement phase, security properties in the design phase, and attack patterns in the implementation phase.

The above papers provide classifications to help designers in the choices of security patterns. But, they do not help check whether patterns are correctly included in models. They also do not ensure that models are indeed more secure.

For the first point, Konrad et al. introduced in [7] a security pattern template to ease pattern integration. They exposed the difficulties related to the lack of comprehensive and formal description and proposed a template composed of Unified Modeling Language (UML) diagrams and LTL properties. Actually, we assume in our approach, that security patterns are indeed described with this kind of template.

In the last decade, several papers also proposed methods to check if UML models meet security requirements, the latter being usually expressed with LTL properties [8]-[9]. For instance, Tanvir, et al. proposed an approach to verify the impact of using Role Based Access Control (RBAC) in a Computer Supported Cooperative Work (CSCW) [8]. Thereby, they showed how to formally check if an application meets security requirements.

Our approach proposes to consider both above aspects, i.e., pattern integration assistance and verification of security properties on UML models, inside a whole process. We check pattern integration in UML models by considering their structural and behavioral properties. We also define and assess their integration quality with metrics. If these are low, the designer is warned that the UML diagrams of the application should be revisited. Afterwards, we also check, by means of model-checking, that the chosen security patterns actually remove undesired vulnerabilities from the UML models. This step aims to attest that the chosen patterns are effective against undesired vulnerabilities, or to warn designers to chose other patterns.

III. Model securing with pattern integration aided by formal techniques

In this section, we present our approach to assist designers to integrate security patterns in models for devising more secure applications, as illustrated in Figure 1. In this paper, we assume having a UML model M of the application, a vulnerability set V, which must not be exploited by attackers in the application, and a security pattern set $Sp = \{Sp_1, ..., Sp_k\}$, which should prevent the vulnerabilities of $V = \{V_1, ..., V_l\}$ from being exploited. We also assume having a base of generic formal properties P_{V_i} and P_{Sp_j} describing V_i and Sp_j. The approach illustrated in Figure 2 aims at checking whether each pattern Sp_j is correctly integrated on M and if M still has the vulnerability V_i despite the use of Sp.

As the patterns are described in a generic, abstract form, the first step for the designer is to instantiate every pattern

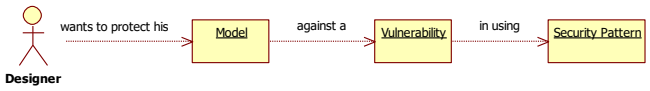

Figure. 1. Context of the proposed approach

to the context of his model. Instantiating a pattern consists in adapting each constituting element to the specific context of a model. It is a complex step, which requires expertise in order to not damage the pattern. Hence, this step has to be verified:

1) Given a pattern Sp_j, we check if the structural and behavioral properties of Sp_j hold in the model M. For the structural properties, we use the approach and tool we developed in [10]. Given the pattern Sp_j and its generic description, the tool automatically derives a set of OCL queries encoding the structure of the pattern. Then, we call an executor of procedural OCL queries [11]. After the execution, all micro-architectures, subsets of the model, looking like the pattern are listed and a coefficient of disclosure C_{Sp_j}, ranging between 0 and 1, is given. The one with the highest coefficient is taken into account to locate where the pattern has been integrated in the model. If the designer does not agree, two cases are possible. On one hand, the pattern instantiation is incorrect and should be modified. On the other hand, the context is specific enough to justify a change in the structure. In any case, if the designer does not consider that the pattern is structurally integrated, the next step cannot be reached.
2) This step consists in checking whether some behavioral properties of Sp_j hold in M. We provide, with the pattern, a set of generic behavioral properties P_{Sp_j}, described in LTL. These rules describe the sequential message exchanges between the methods and the temporal states of the objects. These properties are generic and should be manually instantiated by the designer. Once instantiated, M is automatically translated into a Promela (PROtocol MEta LAnguage) specification with the HugoRT tool [12] and we check if this specification satisfies the previous LTL formulas with the model-checker Spin [13]. This indicates whether the pattern integration into the model respects behavior imposed by the pattern.
3) Quality metrics are now computed to measure the integration quality of the pattern Sp_j of Sp into the model M with regard to the coefficient of disclosure C_{Sp_j} and the LTL property set P_{Sp_j} provided with Sp_j. Intuitively, the closer C_{Sp_j} is to 1 and the larger the set P_{Sp_j}, the more accurate the estimation of the pattern integration is. We define the mapping $m : P_{Sp} \rightarrow \{0, 1\}$ by $m(p) = 1$ iff $M \models p$, and $m(p) = 0$ otherwise. The estimation range of a pattern Sp_j integration is between 0 and $n = Card(P_{Sp_j}) + 1$ and this first integration metric is defined as:

$$0 \leq m(Sp_j) = \sum_{p_i \in P_{Sp_j}} m(p_i) + C_{Sp_j} \leq n$$

Afterwards, we compute the overall integration quality of the security pattern set $Sp = \{Sp_1, ..., Sp_k\}$ using an utility function U. With this aim, we take *Simple Additive Weighting* (SAW) [14] which allows to adjust the integration estimations of each pattern having different ranges by a weight representing user preferences and priorities. In our case U is defined as:

$$U = 0 \leq \sum_{i=1}^{k} m(Sp_i)/(Card(P_{Sp_i}) + 1).w_i \leq 1$$

with $w_i \in \mathbb{R}_0^+$ and $\sum_{i=1}^{k} w_i = 1$, w_i being the weight of Sp_i to represent designer preferences. The closer U is to 1, the greater the estimation that security patterns $Sp_1, ..., Sp_k$ are correctly integrated into M. A good integration quality U can be reached with few properties. With a small number of properties, the pattern is not well documented / defined. In contrast, if the pattern integration is defined with large property sets and U is close to 0, this means that a mistake has been made during the instantiation or that the context is specific enough to justify a change in the structure of the pattern.

4) The last step of our approach starts when the designer considers that the pattern is integrated with a sufficient quality. For the vulnerability $V_i \in V$, we have a set of generic LTL properties P_{V_i} describing V_i, i.e., undesired behavior that should never happens. Once more, the designer has to instantiate these properties in accordance with the model M. The Spin tool is now called to check that the Promela specification of M will never satisfy the undesired behavior expressed by P_{V_i}. If the presence of a vulnerability is detected, a counter-example is returned by Spin to detail the origin of the violation of the property. Hence, guided by the counter-example, the designer has to ensure that the chosen patterns are effective against the vulnerability or that the combination of patterns does not induce flaws.

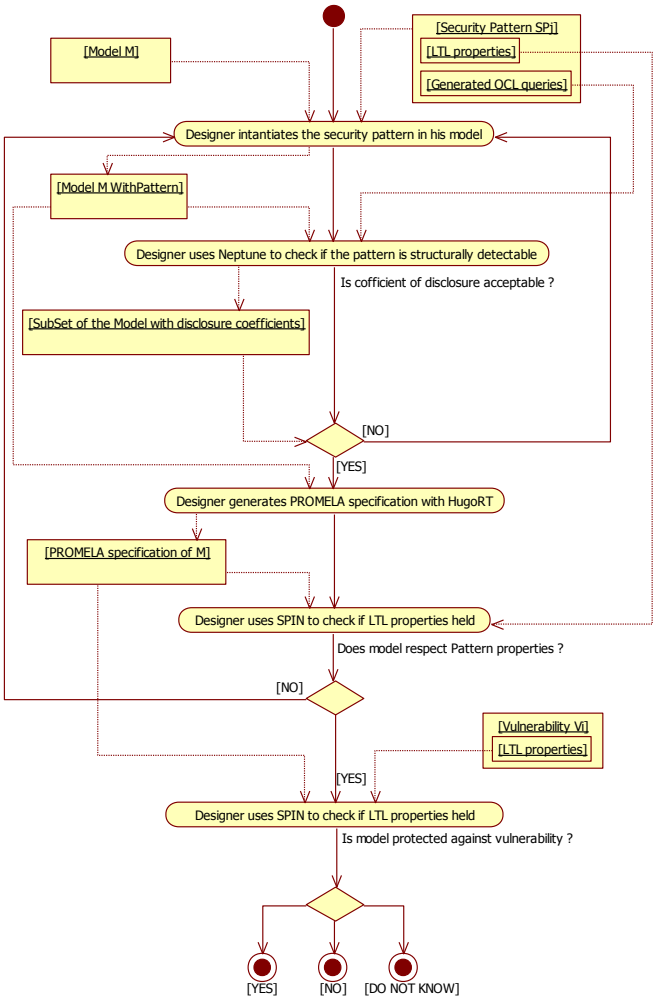

Figure. 2. An approach to assist designers to devise more secure applications

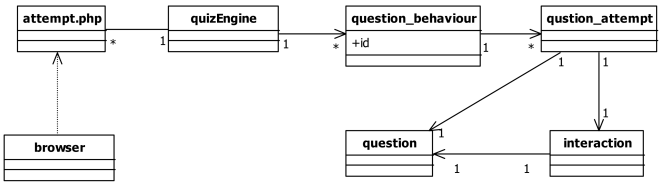

Figure. 3. Moodle Quiz engine class diagram

IV. Case Study

In this section, we describe, with more details, the steps of the proposed approach through a Web application example, the Moodle education platform [15], and precisely on its exam (quiz) functionality. The UML class diagram of this functionality is given in Figure 3 and its sequence diagram in Figure 4.

A user is identified with an *id* and accesses the service with a request of the form "GET /moodle/quiz/attempt.php?id=123". The identifier is used by the quizEngine as a parameter to get information about the user profile so that he/she has only one opportunity to pass an exam. An SQL database is used to get the user profile information and creates an exam session with regard to the requested id.

It is well-known that this kind of Web applications is usually exposed to threats related to *input ports 'passing illegal data'*, and especially injection attacks [4]. For instance, an attacker may exploit a vulnerability to pass an exam many times by spoofing or forging identities stored in database through SQL injection attacks. We have chosen to take as example here a familiar vulnerability called *CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')*, which is the main reason of SQL Injection attacks and one of the most recurrent vulnerabilities [16].

In order to secure the Web application, we have chosen to

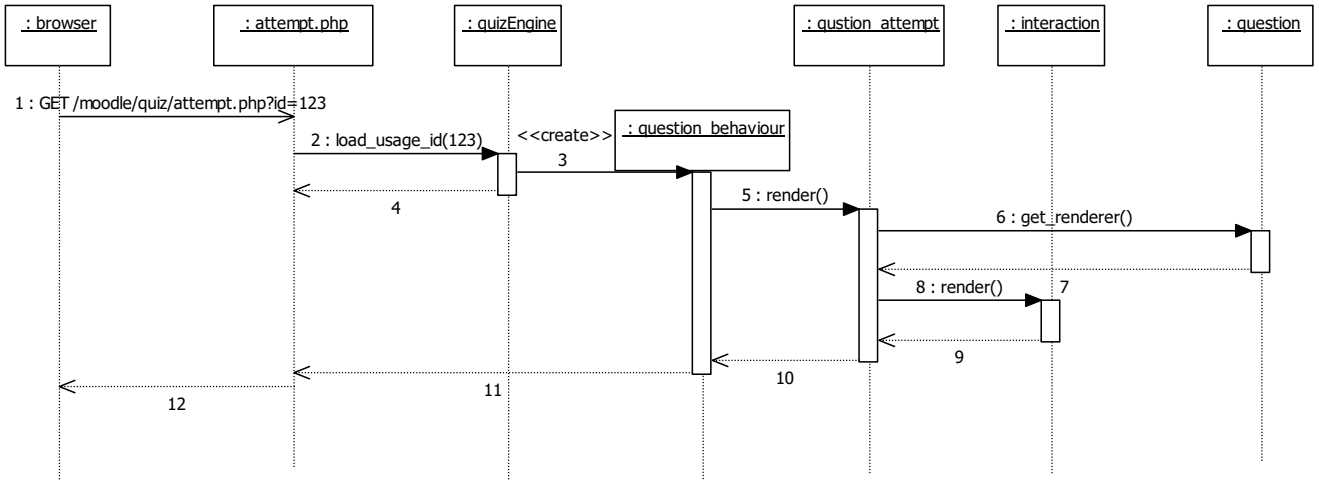

Figure. 4. Moodle Quiz engine sequence diagram

use the *Intercepting Validator* security pattern whose UML class diagram is illustrated in Figure 5. This pattern is indeed presented as a solution to prevent attackers from exploiting the above vulnerability [17]. The intent of this security pattern is to validate every user input request before using it as a parameter by a dynamically loadable validation logic [17].

According to the *Intercepting Validator* security pattern documentation, its behavior is highlighted most notably by the following properties:

1) a validation logic for every data-type used in the application,
2) a single mechanism to validate all data-types,
3) the separation of the validation logic from the presentation logic,

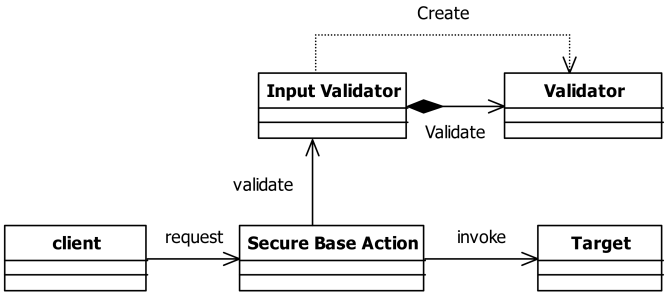

Figure. 5. Intercepting Validator class diagramm

A. Security pattern integration

We instantiated the *Intercepting Validator* security pattern on the model of the Moodle QuizEngine application by adapting its structural and behavioral properties in concordance with the diagrams in Figures 3 and 4. The pattern classes are firstly added between *attempt.php* and *quizEngine* in the QuizEngine class diagram to prevent from SQL Injections through *quizEngine*, which has access to the database. The resulting class diagram is depicted in Figure 6. The sequence diagram of the QuizEngine application is also extended to include the security pattern behavior. The resulting diagram, given in Figure 7, shows that the messages exchanged between *attempt.php* and *quizEngine* are now validated.

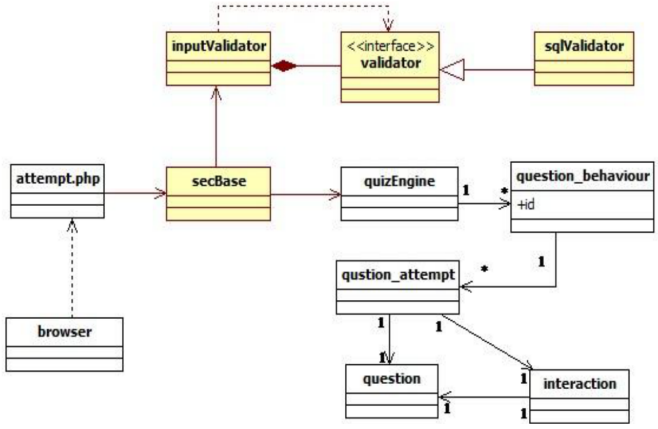

Figure. 6. Instantiation of the security pattern: class diagramm

B. Security pattern instantiation assessment

In this step, we check if the structural and behavioral properties of the security pattern hold in the application model (steps 1, 2 of the approach).

Firstly, we use the method we developed in [10] to extract the pattern structural properties, expressed with OCL queries. Then, we call the tool Neptune [11] to return a list of pairs $(v, coef)$ with v a vertex of the model that is also a vertex of the pattern and $coef$ a coefficient of disclosure. With the class diagram of Figure 6, the tool provides the class "SecBase"

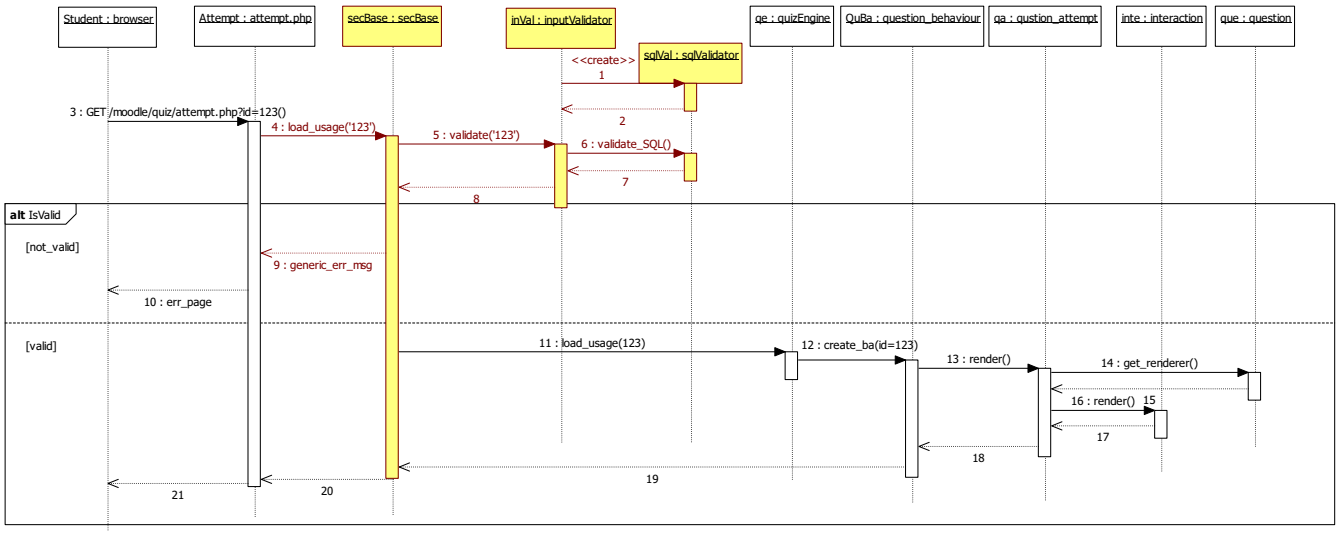

Figure. 7. Instantiation of the security pattern: sequence diagramm

and a coefficient equal to 1. This means that the pattern structure has been completely found only one time. With more complex diagrams, the designer may check if the tool has indeed recognized the pattern or has revealed a diagram part that looks like the pattern. He/she can change the class diagram if required.

For the behavioral properties, we assume having a set of generic LTL properties describing the *Intercepting Validator* pattern. We assume having three generic properties here, given in Table I col2 have to be manually instantiated to meet the QuizEngine application model. These ones are given in Table I col3.

For example, the property *"A validation logic for every data-type used in the application"* given above, is formalized with the LTL formula p_1, which means *"For every Client input, we do not validate data until the creation of the matching validator (SQL, XML, LDAP, etc.)"*.

$$p_1 : \Box\ (Clientinput(Data) \rightarrow !Validate(data)$$

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

This generic formula is instantiated with respect to the QuizEngine application context, found in the sequence diagram of Figure 7. The instantiation of the property implies the good choice of the events matching the facts addressed by the generic LTL formula. For example, the fact $Clientinput(Data)$ corresponds to the state input (the arrival of the request "GET /moodle/quiz/attempt.php?id=123") of the object *attempt.php*. The LTL formula becomes :

$$p_1 : \Box\ (attempt.inState(input) \rightarrow$$

$$!secBase.inState(WaitingVal)\ \mathcal{U}\ inVal.inState\ (valCreated))$$

To check whether the LTL formula p_1, p_2, p_3 hold in the model of Figure 7, we then performed the two following steps:

1) The UML diagram is translated into a Promela specification, with the HugoRT tool [12],
2) The Spin model-checker [13] is called to check whether the Promela specification satisfies the LTL formulas. Namely, Spin checks that the Promela specification never ends in a state corresponding to a counterexample of one of the properties p_1, p_2, p_3.

In this example, all the properties of Table I hold. The pattern integration quality can now be straightforwardly estimated by combining the results obtained from the previous steps. Here, the estimation of the pattern integration is given by the metrics $0 \leq m(Sp_1) = 1+1+1+1 \leq 4$ and $0 \leq U = 1 \leq 1$. The latter shows that the security pattern is well integrated compared to the number of available behavioral properties. In contrast, the metric $m(Sp)$ also reveals that the upper bound of the metric range (n) is low. This means that the number of behavioral properties is modest, and perhaps insufficient to ensure that the pattern is really correctly integrated.

C. Vulnerability exposure assessment

The last step aims at confirming that the security vulnerability is no longer exposed in the application model. We also assume having a set of LTL generic properties expressing behaviors that should never happen. For the CWE-89 vulnerability, taken as example in the paper, its documentation provides the following properties [4]: v_1: No input validation, v_2: Bad input validation, v_3: Privilege escalation, v_4: Remote information inference. This undesired behavior is formalized with the LTL formulas given in Table II col.2. These formulas also have to be manually instantiated with respect to the context of the application. The resulting formulas are given in Table II col.3. With the UML diagram of Figure 6, one can deduce that the event *clientInput* corresponds to the arrival of information from *attempt.php*, and *invokeTarget* corresponds

TABLE I
INTERCEPTING VALIDATOR LTL PROPERTIES

P	LTL generic form	LTL instantiated form LTL
p_1	$\Box(Clientinput(Data) \rightarrow !Validate(Data)\ \mathcal{U}\ createValidator(data.type))$	$\Box\ (attempt.inState(input)) \rightarrow (!\ secBase : inState\ (WaintingValidation)\ \mathcal{U}\ inVal.inState\ (validators\ Created)))$
p_2	$\Box(inputValdiator.isUnique)$	$\Box(secBase.isUnique \wedge inVal.isUnique)$
p_3	$\Box(clientInput(data) \wedge !ServerValidate(data) \wedge \Diamond ServerValidate(data)) \rightarrow (!returnGeneric(message)\ \mathcal{U}\ ServerValidate(data))$	$\Box((attempt.inState(input) \wedge !secBase.inState(nonvalid) \wedge \Diamond secBase.inState(nonvalid)) \rightarrow !attempt.inState(err\_page)\ \mathcal{U}\ secBase.inState(nonvalid))$

TABLE II
CWE-89 VULNERABILITY PROPERTIES

Vulnerability property	LTL generic formula	LTL instantiated formula	Sat
v_1	$\Box(clientInput(data) \rightarrow \Diamond invokeTarget(data))$	$\Box(attempt.inState(input) \rightarrow \Diamond quizEngine.inState(loadUsage))$	No
v_2	$\Box(clientInput(data) \rightarrow \Box(!Valid(data) \rightarrow \Diamond invokeTarget(data)))$	$\Box(attempt.inState(input) \rightarrow \Box(secBase.inState(nonvalid) \rightarrow \Diamond(quizEngine.inState(loadUsage))))$	No
v_3	$\Box(clientInput(data) \wedge client.right(Min) \rightarrow \Diamond client.right(Max))$	?	?
v_4	$\Box(!valid(data) \rightarrow \Diamond(!genMessage))$	$\Box(secBase.inState(nonvalid) \rightarrow \Diamond\ (!attempt.inState(err\_page)))$	Yes

to the use of the input data in the object *quizEngine*. With the sequence diagram of Figure 6, one deduces that the fact *invokeTarget(Data)* of the generic formulas has to be replaced by the state *loadUsage* of the object *attempt.php*.

For example, the property *No input validation* is formulated with the generic LTL formula v_1 in Table II, which intuitively means "for every client input (data) the target is eventually invoked with (data) as parameter". This formula reflects undesired behavior because client inputs always have to be validated before any invocation. The instantiation of the generic formula v_1 gives:

$$v_1 : \Box(attempt.inState(input)\ \rightarrow$$
$$\Diamond\ quizEngine.inState(loadUsage))$$

During the generic formula instantiation, we observed that the third property cannot be deduced. Indeed, this property, which is related to privilege escalation through SQL injection cannot be expressed with the events found in the diagram of Figures 6 and 7. This means that the application model does not include the required features for exposing this property. Here, the notion of level of access is indeed not represented.

We now call the Spin model-checker to check the absence of vulnerabilities in the QuizEngine application model. We obtain the results listed in Table II col.4. Spin detects the presence of the vulnerability property v_4, and provides a counter-example. This property means that in the case when the client input is not valid, the generic error messages (whose content is minimal) is not sent to the client. Thus, the client may get a more detailed error message with sensitive information about the application. This information may be used to deduce attack vectors with a remote information inference.

As a consequence, the QuizEngine application still exposes the CWE-89 vulnerability and this step reveals that the *Intercepting Validator* pattern is insufficient to not expose this vulnerability. After analysis of the counter-example, we deduced that the vulnerability was detected because there is no mechanism, expressed in the security pattern, to generate generic error messages in the case of invalid input messages. Indeed, the *Intercepting Validator* pattern validates and filters input messages only. Another security pattern is therefore required, for instance the *Exception Shielding* pattern [5].

In conclusion, all this process helped integrate the security pattern and showed that this one was not sufficient to secure the application against all the threats related to the CWE-89 vulnerability. Either a more appropriate pattern has to be taken, or another pattern has to be combined with *Intercepting Validator*.

V. CONCLUSION

This paper presents a model-based process for helping designers to devise more secure applications by checking the appropriate use and contextualization of security patterns within UML diagrams. In an initial step, we assume that the designer chooses a list of vulnerabilities that must not appear in the application and a list of security patterns, which should prevent these vulnerabilities from being exploited. The proposed approach then provides several automatic or manual steps to ensure whether security patterns are correctly integrated and if vulnerabilities are still exposed despite the use of security patterns. A coefficient of disclosure is computed for every pattern and assesses if its structure can be found in the application model. Then, a quality metric is

computed to estimate the integration quality of the patterns with regard to a set of available properties expressing the pattern behaviors. Theses metrics guide the designer in the correct pattern integration. The last step of the process aims at warning the designer if a vulnerability is still exposed. We have illustrated this approach with an example of Web application, which has to be protected against the CWE-89 vulnerability related to *Code Injection*.

In the near future, we intend to automate more this process to make its application easier to use for designers. Automation sounds typically not applicable to the entire process but to specific steps, e.g., the choice of the security patterns from vulnerabilities, or the instantiation of LTL properties from the application model. Hence, a designer having a very limited knowledge about security patterns and formal verification could improve the security of its applications anyway. To achieve such an automatic process, an initial step will consist of building an exhaustive base of formal vulnerabilities and security patterns providing the relationship between them.

VI. Acknowledgment

Research supported by the industrial chair on Digital Confidence http://confiance-numerique.clermont-universite.fr/index-en.html

References

[1] H. Mouratidis, P. Giorgini, and G. Manson, "When security meets software engineering," *Inf. Syst.*, vol. 30, no. 8, pp. 609–629, Dec. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.is.2004.06.002

[2] I. Flechais, C. Mascolo, and M. A. Sasse, "Integrating security and usability into the requirements and design process," *Int. J. Electron. Secur. Digit. Forensic*, vol. 1, no. 1, pp. 12–26, May 2007. [Online]. Available: http://dx.doi.org/10.1504/IJESDF.2007.013589

[3] N. Yoshioka, H. Washizaki, and K. Maruyama, "A survey on security patterns," *Progress in Informatics*, vol. 5, pp. 35–47, Mar. 2008.

[4] Common weakness enumeration. [Online]. Available: https://cwe.mitre.org/

[5] Security pattern catalog. [Online]. Available: http://www.munawarhafiz.com/securitypatterncatalog/

[6] A. K. Alvi and M. Zulkernine, "A Natural Classification Scheme for Software Security Patterns," *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pp. 113–120, 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6118361

[7] S. Konrad, B. H. Cheng, L. a. Campbell, and R. Wassermann, "Using Security Patterns to Model and Analyze Security Requirements," *2nd International Workshop on Requirements Engineering for High Assurance Systems*, pp. 13–22, 2003.

[8] T. Ahmed and A. R. Tripathi, "Static verification of security requirements in role based CSCW systems," *Proceedings of the eighth ACM symposium on Access control models and technologies - SACMAT '03*, p. 196, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?doid=775412.775438

[9] M. Al-lail, R. Abdunabi, R. B. France, and I. Ray, "An Approach to Analyzing Temporal Properties in UML Class Models," pp. 77–86, 2013.

[10] C. Bouhours, H. Leblanc, C. Percebois, and T. Millan, "Detection of generic micro-architectures on models," in *Proceedings of PATTERNS 2010, The Second International Conferences on Pervasive Patterns and Applications*, Lisbon, Portugal, 21st - 26th November 2010, pp. 34–41.

[11] T. Millan, L. Sabatier, T. T. Le Thi, P. Bazex, and C. Percebois, "An ocl extension for checking and transforming uml models," in *proceedings of the 8th International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS)*. http://www.wseas.org/: WSEAS Press, 2009, pp. 144–150, (Invited speaker).

[12] S. Merz and C. Rauh, "Model checking timed uml state machines and collaborations," in *7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002*, 2002, pp. 395–414.

[13] G. Holzmann, *Spin Model Checker, the: Primer and Reference Manual*, 1st ed. Addison-Wesley Professional, 2003.

[14] K. P. Yoon and C.-L. Hwang, "Multiple attribute decision making: An introduction (quantitative applications in the social sciences)," 1995.

[15] Overview of the moodle question engine. [Online]. Available: https://docs.moodle.org/dev/Overview_of_the_Moodle_question_engine

[16] OWASP, "Owasp testing guide v3.0 project," in *http://www.owasp.org/index.php/Category:OWASP_Testing_Project#OWASP_Testing_Guide_v3*, 2003.

[17] C. Steel, *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall PTR, 2005.