Une démarche pour l'assistance à l'utilisation des patrons de sécurité

Loukmen REGAINIA[1], Cédric BOUHOURS[1]et Sébastien SALVA[1]
[1] Limos, Université d'Auvergne, France
[loukmen.regainia,cedric.bouhours,sebastien.salva]@udamail.fr

Abstract

La sécurité des applications est critique et primordiale pour la préservation des données personnelles et elle doit donc être prise en compte dès les premières phases du cycle de vie d'une application. Pour cela, une possibilité est de profiter des patrons de sécurité, qui offrent les lignes directrices pour le développement d'une application sûre et de haute qualité. Néanmoins le choix du bon patron et son utilisation pour palier à un problème de sécurité restent difficiles pour un développeur non expert dans leur maniement. Nous proposons dans ce papier une démarche d'assistance aux développeurs permettant de vérifier si un modèle UML composé de patrons de sécurité soulève des vulnérabilités. Notre approche est basée sur une liste de patrons de sécurité et, pour chaque patron, sur une liste de propriétés « génériques » de vulnérabilités. Ainsi, à partir d'un modèle UML composé de patrons de sécurité, notre approche vise à vérifier si un patron de sécurité peut être une garantie à l'absence de vulnérabilités dans une application. Le développeur peut ainsi savoir si son modèle UML comporte des failles qui se retrouveront dans son implémentation malgré l'utilisation de patrons de sécurité. De plus, notre approche peut également montrer que le modèle UML est mal conçu ou qu'un ou des patrons de sécurité ont été abimés lors de leur utilisation.

1 Introduction

La grande expansion des applications web complexes augmente considérablement le nombre de vulnérabilités auxquelles il faut faire face. Une vulnérabilité est définie par « l'Open Web Application Security Project » (OWASP) comme une faille ou une faiblesse, qui permet, à un tiers non autorisé, de l'exploiter pour effectuer diverses opérations non autorisées, dans le but de nuire à l'application. Elle est souvent causée par une erreur de conception, d'implémentation, ou la négligence des bonnes pratiques de conception [11]. De ce fait, il est important de prendre en compte ce risque dès les premières phases du cycle de vie d'une application. Cependant, il est souvent difficile pour les équipes de développement de respecter au quotidien les bonnes pratiques de développement, notamment dans les grands projets. Nos travaux s'inscrivent dans cette problématique dont l'une des solutions est l'assistance à l'utilisation des patrons de sécurité.

Un patron de sécurité présente une solution générique à une problématique de sécurité récurrente. Chaque patron de sécurité est caractérisé par une intention [1], et est décrit avec un ensemble d'éléments facilitant son utilisation (motivation, contraintes d'intégration…). Chaque patron est présenté avec un haut niveau d'abstraction ce qui permet son utilisation indépendamment du contexte de l'application. Et ainsi, de le contextualiser afin de rependre aux spécifications statiques et comportementales d'une application [13]. Néanmoins, ce haut niveau d'abstraction impose aux concepteurs de bien comprendre le patron pour bien réussir sa contextualisation. Ce qui correspond à l'adaptation du patron de sécurité

aux spécifications d'une application, afin de palier à des erreurs de conception qui peuvent mettre en danger la sécurité de cette dernière.

Ainsi, d'une part, une maîtrise de son utilisation s'impose afin d'éviter d'induire de nouvelles erreurs de conception [2]. En effet, une mauvaise utilisation d'un patron de sécurité le dégrade, faisant ainsi perdre son efficacité face à la problématique à laquelle il est censé apporter une solution. D'autre part, même si chaque patron est lié, dans la littérature, à une liste de vulnérabilités [3] [4], il est difficile de garantir à un concepteur que l'utilisation du patron garantit l'absence de vulnérabilités, ce qui rend difficile le choix du bon patron de sécurité.

Dans ce papier, nous présentons une démarche visant à vérifier si un patron de sécurité peut être une garantie à l'absence de vulnérabilités dans une application. Dans une première partie, nous présentons les travaux liés, puis dans un deuxième temps nous présentons notre démarche. Enfin nous détaillons notre démarche à travers un exemple.

2 Contexte

Vue le grand nombre des patrons de sécurité présentés dans la littérature, le choix du bon patron de sécurité est une tâche difficile. Afin de faciliter cette tâche, plusieurs travaux présentent des méthodologies visant à classer et à organiser les patrons de sécurité. Basé sur la méthode de gestion des risques STRIDE, Munawar et al. présentent un catalogue de 97 patrons de sécurité. Ainsi, ils ont proposé une classification des patrons de sécurité par couches de système (Noyau, Périmètre, Extérieur) et par famille de risques de sécurité [4][5]. C'est sur ce type de catalogues de patrons de sécurité que nous allons baser notre approche afin d'identifier les patrons de sécurité et le but donc est d'associer ces patrons de sécurité à une liste de vulnérabilités.

Une partie très importante des patrons de sécurité est leur aspect comportemental. Afin de permettre de modéliser et de vérifier cet aspect on a souvent recours aux méthodes formelles [6] [7], notamment la logique temporelle LTL. Elle est largement utilisée dans la vérification des propriétés de sécurité dans un système. Tanvir, et al. ont présenté en 2003 une méthodologie de vérification de l'impact de l'utilisation du patron RBAC (Role based Access control) dans un système CSCW (Computer Supported Cooperative Work) [6]. Ils ont montré ainsi comment on peut vérifier formellement si une application répond aux exigences de sécurité.

Ces méthodes formelles sont également utilisées dans la vérification de la satisfiabilité des comportements souhaités dans les modèles exprimés en UML. En 2003, Konrad, et al. ont présenté les difficultés rencontrées lors de l'utilisation des patrons de sécurité. Ils ont proposé une méthodologie basée sur la logique temporelle pour vérifier les conditions et les conséquences de l'utilisation des patrons de sécurité. Dans une petite application de e-commerce, ils ont identifié explicitement quels principes de sécurité sont adressés par un patron de sécurité, et présentent ainsi une démarche exemplaire pour la bonne application d'un patron de sécurité [7].

En complément à ces travaux qui expriment explicitement les principes de sécurité, et les propriétés des patrons de sécurité pour un système spécifique, dans notre travail nous avons proposé une méthode qui permet de vérifier la bonne utilisation des patrons de sécurité qui peuvent être plus génériques que le patron de sécurité RBAC étudié par Tanvir, et al. [6]. Nous avons choisis une formalisation des patrons de sécurité, en logique linéaire temporelle (LTL) dans un format générique, ce qui permettra leur réutilisation, indépendamment du contexte de l'application, afin de vérifier la bonne utilisation des patrons de sécurité.

La méthode proposée permet aussi la vérification de l'absence des vulnérabilités de sécurité après l'utilisation d'un patron de sécurité. En complément au travail de Konrad et al. [7], où on a vérifié les propriétés de sécurité adressé par un patron de sécurité, dans notre travail nous essayons de lier formellement les patrons de sécurité aux vulnérabilités. Pour ce faire, nous avons formalisé les propriétés de vulnérabilité en LTL dans un format générique qui permet la vérification de leur absence

indépendamment du contexte de l'application. Dans notre démarche, nous montrons un exemple de la contextualisation de ces propriétés génériques au contexte d'une application modélisée en UML. Cela permet de vérifier la bonne application du patron de sécurité, et le niveau de sécurité apporté par l'utilisation de ce patron de sécurité.

3 Présentation de la démarche

Le but de notre démarche est de chercher la corrélation entre l'utilisation d'un patron de sécurité, et l'absence d'une vulnérabilité liée, dans la littérature, à ce même patron. En d'autres termes nous cherchons à vérifier si un patron de sécurité est correctement appliqué dans un modèle UML, et cela, en vérifiant la présence des points forts du patron de sécurité. Les points forts d'un patron expriment les critères d'architecture et les facteurs de qualité apportés par son utilisation. Extraits depuis la spécification textuelle du patron de sécurité, ils explicitent en quoi un patron est la meilleure solution connue à un type de problème [13].

Notre démarche vise aussi à vérifier l'absence d'une vulnérabilité dans un modèle UML intégrant un patron de sécurité, lié dans la littérature à ce patron de sécurité. Ce qui permettra de vérifier l'impact de l'utilisation des patrons de sécurité, ainsi que le risque d'induire de nouvelles vulnérabilités suite à l'utilisation de plusieurs patrons de sécurité dans une même application.

Pour ce faire, nous avons mis en place une méthodologie vérifiant la satisfaction de propriétés de vulnérabilité sur la spécification d'un modèle UML contenant l'application et le patron de sécurité (Figure 1). Nous avons basé notre démarche sur une base de patrons de sécurité (B.P), et une base de vulnérabilités (B.V). Nous avons présenté les propriétés génériques qui définissent le patron de sécurité (les points forts du patron de sécurité), et de la vulnérabilité en LTL (Logique Temporelle Linéaire).

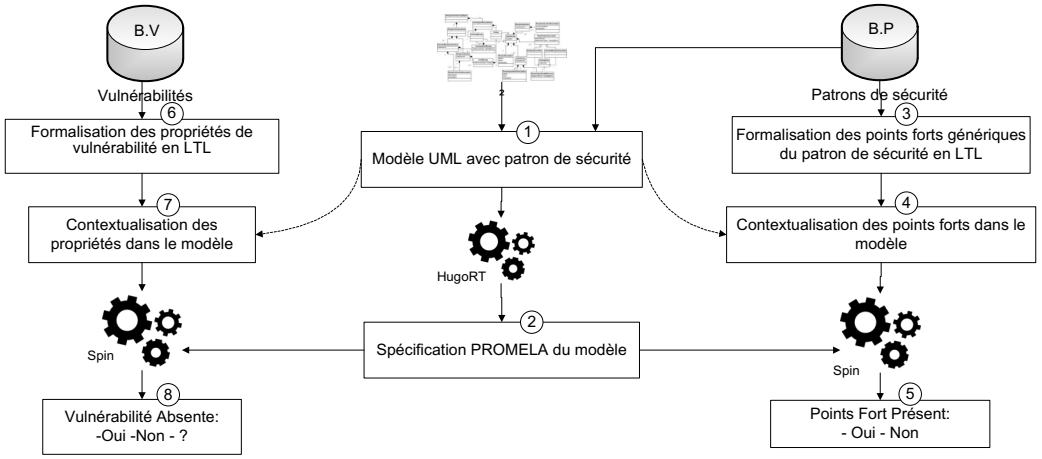

Figure 1 : Schématisation de la méthodologie

Dans notre démarche nous supposons qu'un concepteur choisis un patron de sécurité pour palier à une liste de vulnérabilités. Mené du modèle UML de son application (*app*), le concepteur contextualise le patron de sécurité sur l'application *app* en adaptant les spécifications structurelles et comportementales du patron de sécurité sur le modèle UML de l'application *app*①. Le résultat de cette étape est un modèle UML qui caractérise l'application avec le patron de sécurité (*app-sec)*. Etant présentée en UML, l'application *app-sec* ne permet pas la vérification des propriétés exprimées en LTL. Afin de la permettre nous avons proposé la transformation du modèle UML de *app-sec* en une spécification PROMELA (PROtocol Meta LAnguage)②, et cela moyennant l'outil HugoRT[14]. Suite

à cette étape *app-sec* est présentée avec automate à états finis exprimant tous les états que l'application peut prendre.

Il est nécessaire pour le concepteur, avant de vérifier l'absence des vulnérabilités, de s'assurer que le patron a été correctement contextualisé dans *app-sec*. Pour ce faire, le concepteur doit s'assurer que l'application *app-sec* satisfait les points forts du patron de sécurité. Ainsi, le concepteur choisis les points forts génériques du patron de sécurité utilisé, exprimés en LTL③, et il contextualise ces points forts sur le modèle de l'application *app-sec*④. Ensuite, grâce au solveur de logique temporelle Spin [6], le concepteur peut vérifier la présence des points forts du patron de sécurité⑤. Si l'un des points forts du patron de sécurité est absent dans l'application le concepteur doit revoir l'étape① de la démarche.

Une fois la vérification de la bonne intégration du patron de sécurité est effectuée, le concepteur peut entamer la dernière étape consistant à vérifier si le patron protège ou non d'une vulnérabilité. Nous utilisons les bases de vulnérabilités (OWASP, CWE, WASC) pour définir des propriétés qui permettent l'exploitation de la vulnérabilité (mauvaise gestion, manque de vérification,…) en une liste de formules de logique temporelle. Le concepteur choisis ainsi les propriétés génériques de la vulnérabilité qu'il veut vérifier⑥, ensuite il les contextualise sur le modèle UML de *app-sec*⑦ puis, à l'aide de Spin, leur absence est vérifiée dans la spécification PROMELA de *app-sec*⑧. Nous cherchons ainsi l'existence d'un contre-exemple pour chaque propriété de la vulnérabilité.

4 Illustration

Dans cette section, nous décrivons plus en détails les étapes de la démarche à travers un exemple de fragment d'application web. Dans cette application un « Client » saisit des données destinées à être traitées au niveau d'un objet « Target » dont on suppose qu'il accède à une base de données de type SQL.

Figure 2 : Exemple d'application (*app*)

Ce type d'applications est souvent exposé aux risques liés aux entrées « passing illegal data » [15], notamment les attaques de type « Injection». Pour renforcer cette application, nous avons choisi, comme illustré dans la figure 3, le patron de sécurité « Intercepting Validator » qui est lié dans la littérature à ce type de failles [10].

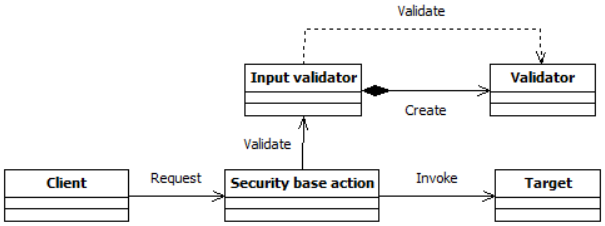

Figure 3 : Patron de sécurité « Intercepting Validator »

Intercepting Validator est un patron de sécurité, dont l'intention est de « vérifier toutes les entrées de l'utilisateur avant de les utiliser comme paramètres pour interagir avec une partie de l'application» et dont les points forts peuvent se résumer ainsi [16]:

1. Un validateur pour chaque type de donnée.
2. Un seul mécanisme pour tous les types de données.

3. Séparer la validation des entrées de la présentation.

L'utilisation du patron de sécurité Intercepting Validator est liée aux vulnérabilités de type injection [15], à savoir « *CWE-77: ('Command Injection') », « CWE-90: ('LDAP Injection')», «CWE-91: XML Injection (aka Blind XPath Injection)», «CWE-88: Argument Injection or Modification», «CWE-79: ('Cross-site Scripting')», « CWE-99: Improper Control of Resource Identifiers ('Resource Injection') »…* [12]

La vulnérabilité prise en exemple ici est la vulnérabilité «*CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')»* [12], qui est la cause principale des attaques de type Injection SQL (Top 1 OWASP 2013). Cette vulnérabilité est définie par les propriétés génériques:

1. Aucune validation des entrées.
2. Mauvaise validation des entrées.
3. Escalade des privilèges.
4. Utilisation des informations retournées dans les messages d'erreurs.

Ce qui définit les éléments auxquels il faut faire attention pour ne pas permettre à un attaquant d'exploiter cette vulnérabilité.

Nous avons contextualisé le patron de sécurité intercepting Validator sur l'application *app* et cela en adaptant le patron de sécurité aux spécifications de l'application *app,* à savoir, le client, target, ainsi qu'un validateur pour les commandes de type SQL.

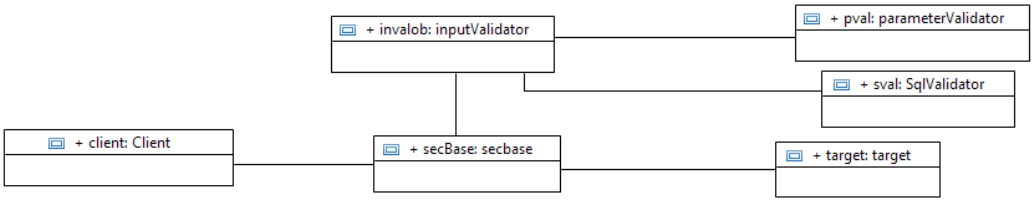

Figure 4 : la contextualisation du patron de sécurité (*app-sec*)

4.1 Vérification de la bonne intégration du patron

Afin de pouvoir vérifier la bonne contextualisation du patron « Intercepting Validator » nous avons formalisé ses points forts génériques (Un validateur pour chaque type de donnée, Un seul mécanisme pour tous les types de données, Séparer la validation des entrées de la présentation.) (Tableau 1).

Exemple : «1. Un validateur pour chaque type de donnée » ce point fort générique est présenté en LTL avec la formule :

$$F1:\square\ (Clientinput(Data) \rightarrow !\ Validate(Data)\ U\ createValidator(data.type))$$

Ce qui signifie : « Pour chaque donnée entrée par le client, ne pas valider cette donnée jusqu'à la création du validateur qui correspond au type de cette donnée ».

Etant présentée dans son format générique, cette formule définie le point fort indépendamment du contexte de l'application. Pour permettre la vérification de la présence du point fort dans l'application, nous l'avons contextualisé sur le modèle Exemple :

$$F1':\square\ (client.inState(input) \rightarrow (!\ secBase.inState(WaintingValidation)\ U\ invalob.inState(validatorsCreated)))$$

Où client, secBase, et invalob sont des objets du modèle de l'application, et input, WaintingValidation, et validatorsCreated sont des états que ces objets peuvent prendre au cours du comportement de l'application.

La contextualisation des points forts du patron de sécurité dans le modèle implique le choix des éléments dans le modèle de l'application qui correspondent aux faits exprimés dans la forme générique de la formule. Par exemple le fait *Clientinput(Data)* dans la formule F1 correspond dans le modèle à l'état *input* de la classe client de notre modèle d'application.

Après avoir transformé le modèle UML de *app-sec* en une spécification PROMELA, nous nous assurons de la présence des points forts dans le modèle *app-sec* à l'aide de l'outil Spin. En d'autres termes, nous vérifions que l'automate correspondant à *app-sec* ne finit jamais dans un état qui correspond à un contrexemple de la formule qui caractérise le point fort.

Point fort	***Points forts génériques LTL***	***Points forts contextualisés sur app-sec***	***Satisfiabilité***
1	□ (Clientinput(Data) → !Validate(Data) U createValidator(data.type))	□ (client.inState(input) → (! secBase.inState(WaintingValidation) U invalob.inState(validatorsCreated)))	Oui
2	□(inputValdiator.isUnique)	□ (secBase.isUnique and Inval.isUnique)	Oui
3	□ (clientInput(data) and ! ServerValidate (data) and ◊ ServerValidate (data)) → (! returnGeneric (message) U ServerValidate (data))	□ ((client.inState(input) and ! secBase.inState(nonvalid) and ◊ secBase.inState(nonvalid)) → ! client.inState(genmessa) U secBase.inState(nonvalid))	Oui

Tableau 1 : Propriétés du patron de sécurité « InterceptingValidator »

Dans le cas où l'un des points forts du patron de sécurité est absent, il est nécessaire pour le concepteur de revoir l'étape de contextualisation du patron de sécurité. L'absence d'un point fort est généralement causée par une erreur de conception lors de la contextualisation du patron de sécurité.

4.2 Recherche de corrélations

Dans le but de vérifier l'absence de la vulnérabilité dans l'application, nous formalisons la liste des propriétés génériques de la vulnérabilité CWE-89 en une liste de formules LTL (Aucune validation des entrées, Mauvaise validation des entrées, Utilisation des informations retournées dans les messages d'erreurs, Escalade des privilèges.). À l'aide de Spin, nous vérifions si la spécification PROMELA de *app-sec* contient les propriétés de la vulnérabilité. Contrairement à la vérification de la présence des points forts, dans cette étape nous vérifions l'absence des vulnérabilités. En d'autres termes, nous vérifions que l'automate correspondant à l'application finit toujours dans un état qui correspond à un contrexemple de la formule qui définit la propriété de la vulnérabilité. Exemple : « 1. Aucune ou mauvaise validation des entrées. »

$$F2: \square \textit{ (clientInput (data)} \rightarrow \Diamond \textit{ invokeTarget (data))}$$

Ce qui signifie : « pour chaque donnée (*data)* entrée par le client la classe cible (*Target)* est invoquée pour traiter cette donnée ».

Etant présentée dans son format générique, cette formule définie la propriété de la vulnérabilité indépendamment du contexte de l'application. Pour permettre la vérification de l'absence de la vulnérabilité dans l'application, nous l'avons contextualisé sur le modèle Exemple :

$$F2': \square \textit{ (client.inState(input)} \rightarrow \Diamond \textit{ target.inState(targetcalculate))}$$

Où client, et target sont des objets du modèle de l'application, et input et targetcalculate sont des états que ces objets peuvent prendre au cours du comportement de l'application.

Dans le tableau 2, la contextualisation de certaines propriétés de vulnérabilité est notée avec des « **?** ». La propriété 3 « escalade de privilèges »de la vulnérabilité CWE-89 n'est pas vérifiable car le patron de sécurité « Intercepting Validator », le seul patron intégré dans l'exemple, ne gère pas la problématique de l'escalade des privilèges. Nous montrons ainsi qu'un patron de sécurité seul peut ne pas couvrir toutes les propriétés d'une vulnérabilité.

Propriétés	***Propriétés générique de la vulnérabilité LTL***	***Propriétés de la vulnérabilité contextualisée sur le modèle***	***Vulnérable***
1	□(clientInput(data) → ◊ invokeTarget(data))	□ (client.inState(input) → ◊ target.inState(targetcalculate))	Non
2	□ (clientInput(data) → □ (!Valid(data) → ◊ invokeTarget(data)))	□ (client.inState(input) → □ (! secBase.inState(nonvalid) → ◊ (target.inState(targetcalculate))))	Non
3	□(clientInput(data) andclient.right(Min)→ ◊ client.right(Max))	**?**	**?**
4	□ (! valid(data) → ◊ (! genMessage))	□ (secBase.inState(nonvalid) → ◊ (! client.inState(genmessa)))	Oui

Tableau 2 Propriétés de la Vulnérabilité CWE-89

Le patron de sécurité « Intercepting Validator » offre un mécanisme pour la validation d'une large variété de types de données que l'application peut utiliser (SQL, XML, LDAP,…). Le tableau 2 montre qu'il protège des problèmes de mauvaise validation des entrées, cause principale de failles de type « Injection SQL », mais qu'il ne protège pas de l'exploitation d'informations dans les messages retournés « Output information disclosure », et les failles de type « Escalade de Privilèges ». Ce dernier type de vulnérabilité est lié au patron de sécurité «Least Privilege» [9]. Ainsi, lors de la présence d'une propriété de vulnérabilité, on a souvent recours à l'utilisation d'autres patrons de sécurité.

5 Conclusion

Dans cet article, nous avons présenté une démarche permettant de vérifier la présence des points forts d'un patron de sécurité dans un modèle UML, et l'absence des propriétés d'une vulnérabilité liée dans la littérature à ce patron de sécurité. A travers un exemple d'utilisation du patron de sécurité « Intercepting Validator », nous avons montré que malgré la présence des points forts de ce patron de sécurité dans l'application (tableau 1), son utilisation ne protège pas de tous les éléments qui peuvent être exploités dans une attaque de type Injection SQL (tableau 2). Un type d'attaque lié dans la littérature à ce patron de sécurité. A terme, nous souhaitons associer un ensemble de collaborations entre une liste de patrons de sécurité et une famille de vulnérabilités.

Nous avons également présenté à travers cet exemple une formalisation dans un format générique des propriétés d'un patron de sécurité, et d'une vulnérabilité afin de permettre leur réutilisation dans d'autres types d'applications. Tout ceci s'inscrit dans une volonté de faciliter, à la fois, le choix des patrons de sécurité, en identifiant quels éléments de sécurité le patron peut couvrir, et la bonne utilisation du patron de sécurité avec une vérification formelle de la présence de ses points forts dans une application.

La présentation générique des propriétés d'un patron de sécurité et d'une vulnérabilité, permet leur utilisation indépendamment du contexte particulier de l'application. En se basant sur une base contenant les propriétés génériques qui caractérisent les patrons de sécurité et les vulnérabilités, nos prochains

travaux consisteront, en premier lieu, à mettre le lien formel entre les patrons de sécurité et la liste des vulnérabilités que chaque patron de sécurité peut couvrir. Puis en deuxième lieu, la génération des points forts du patron de sécurité à utiliser depuis une liste de vulnérabilités.

Dans le plus long terme le but est de décharger le concepteur de la manipulation de la logique temporelle et cela en automatisant la contextualisation des points forts et des propriétés de vulnérabilité dans un framework. Après une étape de validation par expérimentation, ce Framework facilitera au concepteur le développement d'une application plus sûre, grâce à l'utilisation de patrons de sécurité. Et cela en lui facilitant la tâche de l'utilisation des patrons de sécurité.

Références

[1] C. Dougherty, K. Sayre, and R. Seacord, "Secure design patterns", *Technical report, CMU/SEI-2009-TR-010 ESC-TR-2009-01*, 2009.

[2] M. Schumacher and U. Roedig, "Security Engineering with Patterns", *ISBN 3540407316*, 2001.

[3] Dangler, Jeremiah Y., "Categorization of Security Design Patterns". *Electronic Theses and Dissertations.Paper 1119*, 2013. http://dc.etsu.edu/etd/1119.

[4] D. M. Kienzle, M. C. Elder, D. Tyree, and J. Edwards-Hewitt, "Security patterns repository version 1.0," *DARPA, Washingt. DC*, 2002.

[5] M. Hafiz, P. Adamczyk, and R. E. Johnson, "Organizing security patterns," IEEE Softw., vol. 24, pp. 52–60, 2007.

[6] T. Ahmed and A. R. Tripathi, "Static verification of security requirements in role based CSCW systems," Proc. eighth ACM Symp. Access Control Model.Technol. - SACMAT '03, p. 196, 2003.

[7] S. Konrad, B. H. C. Cheng, L. a. Campbell, and R. Wassermann, "Using Security Patterns to Model and Analyze Security Requirements," 2nd Int. Work. Requir.Eng. High Assur.Syst., pp. 13–22, 2003.

[8] W. Tian, J. F. Yang, J. Xu, and G. N. Si, "Attack model based penetration test for SQL injection vulnerability," Proc. - Int. Comput.Softw. Appl. Conf., pp. 589–594, 2012.

[9] M. Schumacher, "Security Patterns," Informatik-Spektrum, vol. 25, pp. 220–223, 2002.

[10] Chris Steel, Ramesh Nagappan and Ray Lai '' Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management", *ISBN 013146307*, 2005.

[11] Open Web Application Security project, https://www.owasp.org

[12] Common Weakness Enumeration http://cwe.mitre.org/

[13] Cédric Bouhours, Hervé Leblanc and Christian Percebois, "Bad smells in design and design patterns", in: Journal of Object Technology, ETH S.F.I.T., Vol. 8, Num. 3, pages 43-63, 2009.

[14] M. Balser, S. Bäumler, and A. Knapp, “Interactive verification of UML state machines,” Form. Methods Softw. Eng., pp. 434–448, 2004.

[15] A. V. Uzunov and E. B. Fernandez, “An extensible pattern-based library and taxonomy of security threats for distributed systems,” *Comput. Stand. Interfaces*, vol. 36, no. 4, pp. 734–747, 2014.

[16] C. Steel, R. Nagappan, and R. Lai. Core security patterns: Best practices and strategies for J2EE(TM), Web services, and identity management. Prentice Hall PTR, October 2005.