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

Loukmen Regainia, Cédric Bouhours, Sébastien Salva

▶ **To cite this version:**

Loukmen Regainia, Cédric Bouhours, Sébastien Salva. Une démarche pour l?assistance à l'utilisation des patrons de sécurité. 4ème Conférence en IngénieriE du Logiciel (CIEL), Jun 2015, Bordeaux, France. hal-02019695

HAL Id: hal-02019695
https://uca.hal.science/hal-02019695

Submitted on 14 Feb 2019

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.

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. Elle est souvent causée par une erreur de conception ou d'implémentation [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 de le contextualiser aux spécifications statiques et comportementales d'un modèle UML [13]. Néanmoins, ce haut niveau d'abstraction impose aux concepteurs une maîtrise de son utilisation afin d'éviter qu'il n'induise de nouvelles erreurs de conception [2]. En effet, une mauvaise utilisation du 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.

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

Afin de faciliter le choix et la bonne utilisation d'un patron de sécurité, plusieurs travaux présentent des méthodologies visant à classer et à organiser les patrons de sécurité. Munawar, et al. présentent un catalogue de 97 patrons de sécurité classés par couches de système et par propriétés de sécurité [4][5]. C'est sur ce catalogue que nous allons baser notre approche afin d'identifier les patrons de sécurité et leurs utilisations.

Le recours aux méthodes formelles, notamment la logique temporelle LTL 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 (Rolebased Access control) dans un système CSCW (Computer SupportedCooperativeWork) [6]. Ces méthodes formelles sont aussi utilisées dans la vérification de la satisfiabilité des comportements souhaités dans un modèle UML. En 2003, Conrad, et al.ont présenté les difficultés rencontrées lors de l'utilisation des patrons de sécurité et ont proposé une méthodologie basée sur la logique temporelle pour vérifier 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, le but de notre travail est de présenter ces propriétés dans un format générique indépendamment du contexte de l'application, ce qui permettra leur réutilisation. Dans notre démarche, nous proposons la contextualisation de ces propriétés surle modèle UML de l'application. Cela permet de vérifier le niveau de sécurité apporté par le patron de sécurité dans tous les états que l'application peut prendre.

3 Présentation de la démarche

Le but de notre démarche est de chercher la corrélation entre l'utilisation d'un patron de sécurité, et une vulnérabilité liée, dans la littérature, à ce même patron. En d'autres termes nous cherchons à vérifier si un modèle UML contenant un patron de sécurité correctement appliqué ne présente pas de vulnérabilité. Pour ce faire, nous avons mis en place une méthodologie vérifiant la satisfaction de propriétés de vulnérabilité sur la spécification du modèle UML. Les propriétés de vulnérabilité ont été formalisées en LTL, le modèle UML a été spécifié en PROMELA grâce à l'outil HugoRT[14], ce qui permet de le manipuler comme un automate à états finis exprimant tous les états que l'application peut prendre dans le temps. Ainsi, en exprimant les vulnérabilités en LTL, il est possible de vérifier leur satisfaction grâce à l'outil Spin [6].

Comme illustré dans la figure 1, c'est sur une base de patrons de sécurité, une base de vulnérabilités et un modèle UML que nous axons la démarche. Le modèle UML exprime les aspects statiques et comportementaux d'une éventuelle application. Après avoir injectéun patron dans le modèle①, nous le transformons, à l'aide de l'outil HugoRT, en spécification PROMELA (PROtocolMEtaLAnguage)②.

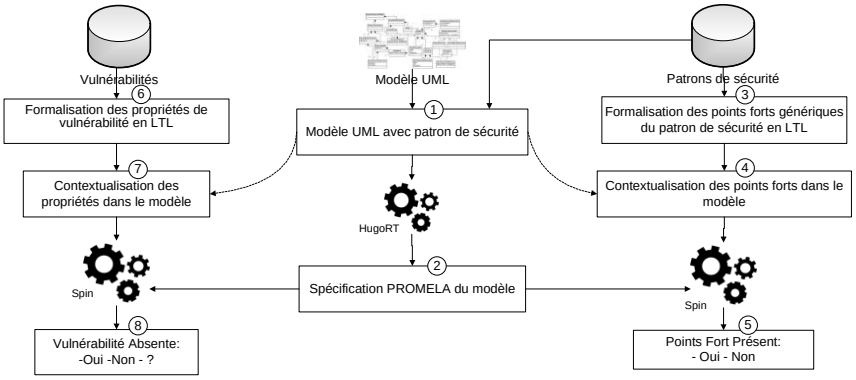

Figure 1 :Schématisation de la méthodologie

Cette spécification ainsi générée, il est nécessaire, avant de chercher d'éventuelles corrélations entre le patron et des vulnérabilités, de s'assurer que le patron a été correctement intégré. Pour ce faire, nous utilisons les points forts du patron choisi. « Les points forts » d'un patron expriment les critères d'architecture et les facteurs de qualité apportés par son utilisation. Ils explicitent en quoi un patron est la meilleure solution connue à un type de problème [13]. En les formalisant en LTL③, et en les contextualisant sur le modèle UML④, il devient possible de s'assurer qu'ils sont tous présents⑤, grâce à Spin, un solveur de logique temporelle[6].
Une fois la vérification de la bonne intégration effectuée, nous pouvons entamer la dernière étape consistant à vérifier si le patron protège ou non des vulnérabilités. Nous utilisons les bases de vulnérabilités (OWASP, CWE, WASC) pour définir des propriétés qui permettent leur exploitation (mauvaise gestion, manque de vérification,…) en une liste de formules de logique temporelle⑥. Ces propriétés sont ensuite contextualisées sur le modèle UML⑦ puis, à l'aide de Spin, leur absence est vérifiéedans la spécification PROMELA du modèle⑧. 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 d'application. 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. Ce type d'applications est souvent exposé à des attaques de type « Injection SQL ». Pour renforcer cette application, nous avons choisi, comme illustré dans la figure 2, d'intégrer le patron de sécurité « InterceptingValidator » qui est conseillé dans la littérature pour palier à ce type de failles [10].

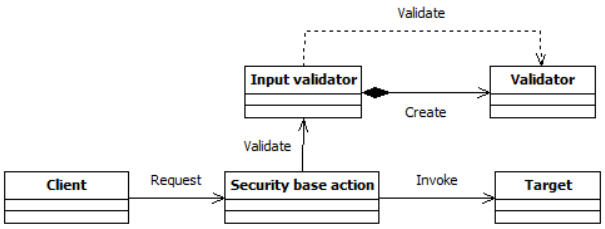

Figure 2 : Patron de sécurité « InterceptingValidator »

InterceptingValidator est un patron de sécurité, dont l'intention est de« vérifier les entrées de l'utilisateur avant de les utiliser » et dont les points forts peuvent se résumer ainsi :

1. Un validateur pour chaque type de donnée.
2. Un seul mécanisme pour tout type de données.
3. Séparer la validation de la présentation.

La vulnérabilité que nous avons étudié est la vulnérabilité «*CWE-89: ImproperNeutralization of SpecialElementsused 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 [3]:

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

4.1 Vérification de la bonne intégration du patron

Afin de vérifier la bonne intégration du patron « InterceptingValidator » nous devons formaliser ses points forts (tableau 1).Ensuite, à l'aide de l'outil Spin nous nous assurons de la présence des points forts dans le modèle exprimé en PROMELA.En d'autres termes, nous vérifionsque l'automate correspondant à l'application finit toujours dans un état qui satisfait la formule qui caractérise le point fort. Exemple : «1. Un validateur est créé pour chaque type de donnée » cette propriété générique est présentée en LTL dans la formule :

F1:□ *(Clientinput(Data) → ! Validate(Data) Until 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 ».

Point fort	***Point fort LTL***	***Point fort contextualisé sur le modèle***	***Satisfiabilité***
1	□ (Clientinput(Data) → !Validate(Data) Until createValidator(data.type))	G(client.inState(input) implies (not secBase.inState(WaintingValidation) U invalob.inState(validatorsCreated)))	Oui
2	□(inputValdiator.isUnique)	G (secBase.isUnique and Inval.isUnique)	Oui
3	□ (clientInput(data) and ! ServerValidate (data) and ◊ServerValidate (data)) → (! returnGeneric (message) UntilServerValidate (data))	G ((client.inState(input) and not secBase.inState(nonvalid) and F secBase.inState(nonvalid)) implies not client.inState(genmessa) U secBase.inState(nonvalid))	Oui

Tableau 1 Propriétés du patron de sécurité « InterceptingValidator » (« client », « secBase », et « Inval » sont des objets du modèle conçu)

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(tableau 2).À l'aide de Spin, nous vérifions si le modèle avec le patron 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 le déroulement infini de l'automate caractérisant l'application finit toujours avec un état qui satisfait la négation de la formule qui définit la propriété de la vulnérabilité.

La propriété « escalade de privilèges »de la vulnérabilité CWE-89 n'est pas vérifiable car le patron de sécurité « InterceptingValidator »,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 → ◊ invokeTarget(data))	G (client.inState(input) implies target.inState(targetcalculate))	Non
2	□ (clientInput(data) → □ (!Valid(data) → ◊ invokeTarget(data)))	G (client.inState(input) implies (not secBase.inState(nonvalid) implies (target.inState(targetcalculate))))	Non
3	□(clientInput(data) andclient.right(Min)→ ◊ client.right(Max))	**?**	**?**
4	□ (! valid(data) → ◊(!genMessage))	G (secBase.inState(nonvalid) implies (not client.inState(genmessa)))	Oui

Tableau 2 Propriétés de la Vulnérabilité CWE-89

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 « InterceptingValidator », 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).

Le patron de sécurité « InterceptingValidator » offre un mécanisme pour la validation d'une large variété de types de données que l'application peut utiliser (SQL, XML, LDAP,…). Nous avons montré 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, et lesfailles detype « Escalade de Privilèges ». Ce type de vulnérabilité est lié au patron de sécurité «Least Privilege»[9]. 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 à automatiser la tâche de la contextualisation et la vérification de ces propriétés. Cela est possible avec une analyse dont le but est d'extraire les éléments de l'application (états, transitions, interactions, …) qui décrivent les comportements apportés par le patron de sécurité. L'extraction de ceséléments permettra d'automatiser la contextualisation des points forts et des propriétés de vulnérabilité. Ainsi, le Framework à développer déchargera le concepteur de la manipulation des formules LTL. Après une étape de validation par expérimentation, ce Framework facilitera au développeur la conception d'une application plus sûre, grâce à l'utilisationde 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] CédricBouhours, 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.