Génération de tests de vulnérabilité Web à partir de modèles

Franck Lebeau [1], **Bruno Legeard** [1,2], **Fabien Peureux** [1], **Alexandre Vernotte** [1]

1. *FEMTO-ST Institute / DISC department - UMR CNRS 6174*
   16, route de Gray, 25030 Besançon, France.
   {flebeau,blegeard,fpeureux,avernott}@femto-st.fr

2. *Smartesting R&D Center*
   18, rue Alain Savary, 25000 Besancon, France.
   bruno.legeard@smartesting.com

RÉSUMÉ. *Cet article concerne la génération automatique de tests de vulnérabilité à partir de modèles pour applications Web. Les technologies de génération de tests à partir de modèles sont aujourd'hui principalement mises en œuvre dans le contexte du test fonctionnel. L'application de ces techniques au test de vulnérabilité sur applications Web en renouvelle les problématiques de recherche, tant au niveau de la modélisation, du pilotage de la génération des tests qu'au niveau de l'assignation du verdict de test. Cet article présente une approche originale pour la génération de tests de vulnérabilité fondée sur une modélisation du comportement fonctionnel de l'application sous test et sur la formalisation des attaques à tester sous la forme de schémas de tests. Ces schémas définissent ainsi la stratégie de génération des tests de vulnérabilité. Cette approche est illustrée sur un exemple de site Web vulnérable (l'application DVWA).*

ABSTRACT. *This paper deals with an original approach to automate Model-Based Vulnerability Testing (MBVT) for Web applications. Today, Model-Based Testing techniques are mostly used to address functional features. The adaptation of such techniques for vulnerability testing defines novel issues in this research domain. In this paper, we describe the principles of our approach, which is based on a mixed modelling of the application under test: the specification indeed captures the behavioral aspects of the Web application, and includes vulnerability test purposes to drive the test generation algorithm. Finally, this MBVT approach is illustrated with the widely-used DVWA example.*

MOTS-CLÉS : *Test de vulnérabilité, test à partir de modèles, applications Web, exemple DVWA*

KEYWORDS: *Vulnerability testing, model-based testing, Web applications, DVWA example*

DOI:10.3166/AFADL.17.12.1-?? © 2012 Lavoisier

Approches Formelles dans l'Assistance au Développement de Logiciels – n° 12/2012, 1-??

1. Introduction

La croissance permanente des usages de l'internet et le développement des applications Web met au premier plan les enjeux de la sécurité informatique, particulièrement en termes de confidentialité des données, de leur intégrité et de la disponibilité des services associés.
Ainsi, dans le baromètre annuel des préoccupations des Directeurs des Systèmes d'Information [1], pour 72% d'entre eux, la sécurité informatique et la protection des données constituent leur préoccupation première. Cette croissance du risque provient en particulier de la richesse des technologies mises en œuvre dans les applications Web actuelles (par exemple avec HTML5) qui accroissent les risques de failles de sécurité. Ce contexte a conduit à un développement important des vulnérabilités applicatives, avec plusieurs milliers de vulnérabilités détectées et divulguées chaque année au sein de bases telle que celle du MITRE CVE - Common Vulnerabilities and Exposures [2]. Les vulnérabilités les plus fréquentes relevées sur ces bases concernent en particulier le manque de résistance face aux injections de code de type Injection SQL ou de type Cross-Site Scripting (XSS), qui possèdent de très nombreuses variantes. Elles se situent en tête des attaques recensées sur les applications Web.
Le test de vulnérabilité au niveau applicatif est d'abord réalisé par les développeurs, mais ils manquent souvent de connaissances suffisamment approfondies sur les vulnérabilités récentes et les exploitations réalisées. Ce type de test peut aussi être réalisé par des sociétés spécialisées dans le test de sécurité, le test d'intrusion par exemple. Ces sociétés assurent une veille permanente sur les vulnérabilités découvertes et l'évolution des attaques, mais en pratique utilisent des méthodes très manuelles qui rendent difficile une diffusion large de leur technique et diminue l'impact de ce savoir-faire.
Le travail présenté dans cette contribution vise à automatiser le test de vulnérabilité d'applications Web, pour étendre la capacité de tests de ces applications, augmenter la détection de vulnérabilités, et ainsi améliorer le niveau global de sécurité. L'approche étudiée, fondée sur les techniques de génération de tests à partir de modèle, vise ainsi à capturer des patrons de test de vulnérabilité pour en automatiser la mise en œuvre sur des applications Web.

Les contributions originales du travail présenté dans cet article se situent :

– dans la prise en compte des comportements applicatifs pour la génération de tests de vulnérabilité, ce qui permet une exploration et des tests plus approfondis ;

– dans la capture des patrons de test de vulnérabilité sous forme de schéma de tests utilisés pour guider la génération automatique de tests à partir de modèle ;

– dans la modélisation pour le test de vulnérabilité qui combine modélisation du système sous test et modélisation des actions de l'environnement.

---

1. Baromètre CIO 2012, enquête réalisée par l'institut CSC, le magazine 01 Informatique et la radio BFM http://assets1.csc.com/fr/downloads/Barometre_CIO_2012_Tout_se_transforme_OK.pdf
2. Site Web de la base de vulnérabilités MITRE CVE - http://cve.mitre.org

Dans la suite de cet article, nous présentons notre approche à partir de l'exemple du test de vulnérabilité de type Cross-Site Scripting (XSS), en fournissant la modélisation et les schémas de test utilisés, pour montrer comment notre outillage de génération de tests produit des vecteurs de test de vulnérabilité exécutables automatiquement sur l'application cible. Cette présentation s'appuie sur un exemple fil rouge : l'application Web DVWA - Damn Vulnerable Web Application - qui constitue une application de démonstration de vulnérabilités classiques. Nous positionnons ces travaux dans l'état de l'art puis proposons une conclusion et des perspectives à ce travail de recherche.

2. Génération de tests de vulnérabilité à partir de modèle

Nous proposons de revisiter et d'adapter l'approche de génération de tests à partir de modèles (dite de Model-Based Testing - MBT) afin de l'appliquer au contexte du test de vulnérabilité sur applications Web (dit de Model-Based Vulnerability Testing - MBVT). Dans un premier temps, nous présentons les spécificités de l'approche MBVT proposée. Dans un second temps, nous introduisons l'exemple fil rouge utilisé dans la suite du document pour illustrer cette approche, et précisons le périmètre de l'expérimentation menée sur cet exemple.

2.1. Principes de l'approche MBVT

Les techniques de génération de tests à partir de modèles, plus communément appelées *MBT* pour *Model-Based Testing* (Utting, Legeard, 2006), ont connu depuis plusieurs années un intérêt et un essor important. Cette attractivité, de la part du milieu scientifique comme industriel, s'explique notamment par la complexité toujours croissante des systèmes et le besoin récurrent en termes de qualité et de fiabilité.
Dans l'approche MBT, le modèle du système sous test (SUT - System Under Test), qui constitue le référentiel d'entrée du processus, permet de dériver des séquences de stimuli et joue le rôle d'oracle de test en prédisant les résultats attendus. Ce modèle représente le comportement fonctionnel attendu des services offerts par le SUT, indépendamment de la manière dont ces services ont été implémentés. Les cas de test générés à partir de tels modèles permettent ainsi de valider la cohérence comportementale du SUT en comparant dos-à-dos les réactions observées sur le système développé avec celles décrites par le modèle.
Les principaux avantages de l'approche MBT résident donc dans sa capacité à automatiser et à augmenter la productivité durant la phase de test, tout en favorisant la maîtrise de la qualité et la pertinence des tests en garantissant une couverture donnée des fonctionnalités et des combinaisons de comportements (stimuli) testées. Cependant, si cette technique permet de couvrir les exigences fonctionnelles spécifiées dans le modèle comportemental du système à tester, elle se limite aussi à cet aspect : ce qui n'est pas modélisé ne sera donc pas testé.

L'application du MBT au test de vulnérabilité conduit à faire évoluer à la fois la modélisation pour la génération des tests et le pilotage de la génération. Dans l'ap-

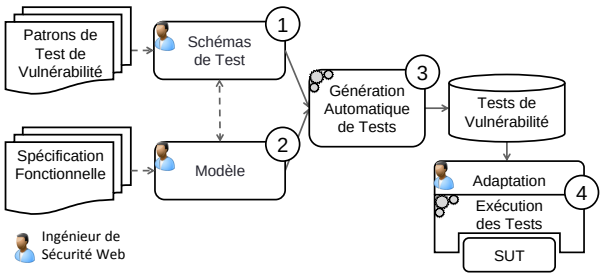

Figure 1. Processus MBVT

proche MBT appliquée au test fonctionnel, il s'agit de générer des tests positifs [3], permettant la validation de l'application par rapport aux exigences fonctionnelles. Le test de vulnérabilité correspond à des tests négatifs [4], typiquement des scénarios d'attaque cherchant à obtenir des données de façon non autorisée par le SUT. L'approche globale de génération de tests de vulnérabilité que nous proposons est présentée dans la figure 1. Elle se compose des quatre activités successives suivantes :

① L'activité de définition des *objectifs de test de vulnérabilité* consiste à formaliser, par des schémas de test, les patrons de test de vulnérabilité que les tests générés doivent couvrir.

② L'activité de *modélisation* consiste à définir un modèle qui permet de capturer les aspects nécessaires pour calculer des séquences de stimuli cohérentes (d'un point de vue fonctionnel) sur le système réel.

③ L'activité de *génération des cas de test* consiste à calculer automatiquement des cas de test abstraits sur la base des artefacts produits lors des deux premières étapes.

④ L'activité de *concrétisation, d'exécution des cas de test et de dépouillement* consiste respectivement à (i) dériver des cas de test exécutables à partir des tests abstraits obtenus lors de l'étape de génération, (ii) d'exécuter ces tests sur l'application sous test, et (iii) d'observer les réactions du système et de les comparer au résultat attendu afin d'assigner un verdict et d'automatiser la détection des failles de sécurité.

La réalisation de ces activités MBVT est supportée par une chaîne outillée qui s'appuie sur une chaîne MBT existante basée sur le logiciel *CertifyIt* (Bouquet *et al.*, 2008) commercialisé par la société Smartesting [5]. Ce logiciel est un générateur de tests qui prend en entrée un modèle de spécification, exprimé dans un sous-ensemble du langage UML (nommé UML4MBT (Bouquet *et al.*, 2007)), qui représente le comportement du système à tester. Concrètement, un modèle UML4MBT contient : (i) le

3. Nous appelons "cas de test positif" un cas de test qui vérifie si une séquence d'actions prévues se déroule comme attendu au niveau des spécifications. Lorsque le cas de test positif réussit, il met en évidence que le scénario testé est correctement implémenté.

4. Par "cas de test négatif", nous entendons un cas qui vise un usage non prévu du système; lorsque le cas de test négatif réussit, il met en évidence un problème au niveau du SUT.

5. http://www.smartesting.com

diagramme de classes modélisant les interfaces pour le test et les données logiques de test, (ii) le diagramme d'objets modélisant les entités manipulées par les cas de test et l'état de ces entités dans l'état initial du SUT, et (iii) un diagramme d'états-transitions, annoté de contraintes exprimées dans un sous-ensemble du langage OCL, spécifiant le comportement dynamique à tester.
CertifyIt permet d'interpréter le modèle UML4MBT et d'automatiser, par application de stratégies prédéfinies, la génération des cas de test et des résultats attendus. Les tests générés prennent ainsi la forme de séquences d'appels d'opérations abstraites (stimuli) agrémentées des résultats attendus sur le système (observation).

La section 3 présente de façon détaillée chacune de ces étapes et les illustre par un exemple pédagogique qui est introduit dans la prochaine sous-section.

2.2. Présentation de l'exemple DVWA

Afin d'évaluer l'efficacité et l'efficience de notre approche, nous l'avons appliquée à une application Web nommée DVWA [6]. L'objectif de cette application en logiciel libre, basée sur PHP/MySQL, est purement pédagogique puisqu'elle a été rendue vulnérable volontairement. Elle est conçue pour permettre aux professionnels de la sécurité, aux développeurs Web et aux enseignants/élèves d'apprendre, d'améliorer et de tester leurs compétences en matière de sécurité Web. DVWA peut également être utilisée pour tester l'efficacité d'outils de test de vulnérabilité. Au sein de l'application DVWA, on trouve en particulier les vulnérabilités de types Injection SQL et Blind Injection SQL, et Reflected et Stored XSS. Il s'agit de vulnérabilités couramment exploitées lors d'attaques sur les applications Web [7].
Chaque vulnérabilité dispose d'une entrée dans un menu dédié, qui mène vers une page exploitable. DVWA dispose de trois niveaux de sécurité : faible, moyen et haut. Chaque niveau de sécurité comporte différents moyens de protection : le niveau faible n'est pas protégé, le niveau moyen est une version améliorée mais toujours vulnérable, et le niveau haut est une version sécurisée. Les utilisateurs peuvent choisir le niveau de sécurité avec lequel ils veulent travailler. Il est également possible de visualiser et de comparer le code source associé à chaque niveau.

2.3. Périmètre et objectif des expérimentations

Notre approche permet de couvrir ces quatre types de vulnérabilité (Injection SQL, Blind Injection SQL, Reflected XSS, Stored XSS). Pour des raisons de place, nous concentrons notre présentation sur le type Reflected XSS (RXSS). C'est l'une des vulnérabilités majeures du fait de la fréquence de son exploitation et de l'impact potentiel du risque créé (par exemple en conduisant à des usurpations d'identité).

---

6. Damn Vulnerable Web Application - http://www.dvwa.co.uk/

7. Ces vulnérabilités figurent parmi les trois types d'attaque les plus fréquentes recensées par le site OWASP - Open Web Application Security Project - cf www.owasp.org.

De manière générale, les attaques XSS sont dirigées vers le poste client. Un attaquant injecte un vecteur malicieux (contenant un script de code pouvant être interprété et exécuté par le navigateur, typiquement en JavaScript) dans l'application par l'intermédiaire d'une entrée d'utilisateur pour qu'une victime charge ce vecteur et l'interprète. Ce genre d'attaque peut se produire dès lors que l'application Web utilise une entrée utilisateur pour construire une sortie. C'est le manque de validation des sorties qui rend une application vulnérable. Une attaque XSS est soit réfléchie (Reflected), ce qui signifie que le vecteur est retourné immédiatement à l'utilisateur, ou stockée (Stored), ce qui signifie que le vecteur a été enregistré dans une base de données et peut être retourné plus tard, dans un autre contexte. Notre présentation porte sur les vulnérabilités aux attaques XSS de type réfléchie (RXSS).
Pour chaque niveau de sécurité, notre objectif est de mettre en œuvre le processus MBVT de manière à générer et exécuter des tests pour ce type de vulnérabilité.

3. Détail de notre approche

Dans cette section, nous détaillons chacune des activités importantes du processus MBVT. Pour chaque activité, nous présentons ses objectifs, son fonctionnement, et nous illustrons nos propos sur l'exemple fil rouge DVWA pour le test de vulnérabilité de type RXSS.

3.1. Formalisation des Patrons de Test de Vulnérabilité en Schémas de Test

Les *Patrons de Test de Vulnérabilité (PTv)* sont les artéfacts initiaux de notre démarche. Un PTv exprime le besoin et la procédure de test permettant de mettre en évidence une faille applicative particulière sur une application Web. Il y a autant de PTv que de types de faille applicative. Les Patrons de Test de Vulnérabilité sont étudiés dans le projet de recherche ITEA2 Diamonds qui en a formalisé la définition et établi un premier catalogue [8]. Un Patron de Test de Vulnérabilité est caractérisé par : son *nom*, sa *description*, ses *objectifs* de test, ses *pré-requis* (qui précisent les conditions et connaissances nécessaires à son bon déroulement), sa *procédure* (qui précise son *modus operandi*), ses *observations* et son *oracle* (qui précisent quelles informations doivent être surveillées pour pouvoir conclure à la présence d'une faille applicative), ses *variants* (qui précisent les possibles alternatives tant au niveau des moyens mis en œuvres, des données malicieuses utilisées, que des informations observées), ses *problèmes connus* (qui précisent les limitations ou problèmes (techniques, par exemple) qui limitent son utilisation), ses *PTv associés*, et ses *références* (vers les ressources publiques traitant des problèmes de vulnérabilité applicative, telles que CVE, CWE, OWASP, CAPEC, ...).
La figure 2 illustre la notion de Patron de Test de Vulnérabilité, en présentant celui relatif aux attaques de type Reflected XSS.

8. http://www.itea2-diamonds.org

Nom	Reflected XSS
Description	Ce patron peut être utilisé sur une application qui ne filtre pas les données utilisateur. Une attaque de type XSS est utilisée pour rediriger l'utilisateur d'une application vulnérable vers un site malveillant, ou pour voler des données utilisateurs (cookies, session)
Objectif(s)	Détecter si une donnée utilisateur est susceptible de transporter des données malicieuse mettant en œuvre une attaque de type XSS
Pré-requis	N/A
Procédure	Identifier un champ utilisateurs sensible, et y injecter la donnée malicieuse *<script>alert(rxss)</script>*.
Observation & Oracle	Vérifier la présence d'une boite de dialogue affichant 'rxss'.
Variant(s)	- variants sur les données malicieuses: encodage des caractères, transformation Hexadécimale, insertion de commentaires - variant sur la procédure: l'attaque peut être mise en œuvre sur les messages HTTP transitant entre le client et le serveur; dans ce cas, on injecte la donnée malicieuse dans les paramètres de message HTTP, et on observe la présence de la donnée injectée dans le message HTTP de réponse du serveur
Problème(s) connu(s)	Les Web Application Firewall (WAF) permettent de filtrer les requêtes envoyées au serveur (black list, black regEx, ...); les variants servent à contourner ces filtres
PTv(s) associé(s)	Stored XSS, DOM XSS
Référence(s)	CAPEC: http://capec.mitre.org/data/definitions/86.html WASC: http://projects.Webappsec.org/w/page/13246920/Cross Site Scripting OWASP: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

Figure 2. PTv pour les attaques de type Reflected XSS

Pour ce PTv, la notion de variant sur les données malicieuses est prise en compte pendant l'activité de modélisation, et la notion de variant de procédure est prise en compte lors de l'activité d'exécution des tests. La procédure initiale est quand à elle formalisée par un Schéma de Test.

Les *Schémas de Test* sont des expressions de haut niveau qui formalisent une intention de test, et qui guide la génération de test à partir du modèle comportemental. Dans le contexte MBVT, nous utilisons les Schémas de Test pour formaliser les PTv. Les Schémas de Test guident la génération des tests sur le modèle et sont exploités par le générateur de tests CertifyIt comme une stratégie de génération.

Le langage des Schémas de Test utilisé se nomme *Smartesting Test Purpose Language* (Botella *et al.*, 2013). C'est un langage textuel basé sur le langage des expressions régulières (Julliand *et al.*, 2008). Il permet de formaliser des intentions de test de vulnérabilité sous forme d'états à atteindre et d'opérations à exécuter. Ce langage s'appuie sur l'utilisation de mots-clés, ce qui ajoute du sens sémantique, et permet d'avoir à la fois un fort niveau d'expressivité et une facilité de compréhension.

Un Schéma de Test est principalement une séquence d'*étapes* à atteindre (trois dernières lignes de la figure 3). Une étape est un ensemble d'opérations ou de comportements à utiliser, ou d'états à atteindre. La transformation de la séquence d'étapes en un test complet, basée sur les comportements et les contraintes du modèle, est à la charge de la technologie MBT (cf. section 3.3). De plus, au début de chaque Schéma de Test, l'ingénieur de sécurité Web peut définir des *itérateurs*. Les itérateurs sont utilisés dans les étapes, pour introduire une forme de variabilité du contexte d'exécution des étapes. La combinatoire de valeurs des itérateurs conduira à produire plusieurs tests (cf. section 3.3).

EXEMPLE (DVWA). — La figure 3 présente un Schéma de Test formalisant le PTv de la figure 1, appliqué au cas d'étude DVWA. Ce Schéma de Test précise que pour toutes

les pages sensibles du SUT, toutes les données malicieuses permettant de détecter une faille de type RXSS, et tous les niveaux de sécurité de DVWA, il faut faire les étapes suivantes : (i) utiliser n'importe quelle opération pour activer la page sensible avec le niveau de sécurité choisi, (ii) injecter la donnée malicieuse dans tous les champs utilisateur de la page sensible, (iii) vérifier si la page est sensible aux attaques de type RXSS. Les trois mots-clés *ALL_** sont des énumérations de valeurs, définies par l'ingénieur de sécurité Web, et permettent de maîtriser le nombre de tests produits.

```
0  for_each_literal $sensiblePage from ALL_SENSIBLE_PAGES,
1  for_each_literal $maliciousRxssDatum from ALL_MALICIOUS_RXSS_DATA,
2  for_each_literal $securityLvl from ALL_SECURITY_LEVELS,
3  use any_operation any_number_of_times to_reach (currentPage=$sensiblePage and
   security_level=$securityLvl) on_instance SUT
4  then use injectAllFields($sensiblePage, $maliciousRxssDatum)
5  then use checkRxssVulnerabilty()
```

Figure 3. Schéma de Test formalisant le PTv de la figure 2 sur DVWA

Nous utilisons un second Schéma de Test, proche de celui présenté, permettant de cibler précisément quels champs utilisateur vont être injectés. Ce Schéma de Test offre des moyens de contrôle à l'ingénieur de sécurité Web. Les modifications concernent: (i) l'ajout d'un itérateur pour cibler les champs injectables, et (ii) l'utilisation de l'opération *injectField* en lieu et place de l'opération *injectAllField* (l.4).

□

3.2. Modélisation

L'activité de modélisation produit un *modèle* à partir, d'une part, des spécifications fonctionnelles de l'application, et, d'autre part, des Schémas de Test qui vont être appliqués sur celui-ci (les mots-clés des Schémas de Test doivent être modélisés). Nous présentons ci-après les différents diagrammes UML utilisés (de classes, d'objets, d'états), et leur utilité respective dans le cadre de notre approche MBVT.
Le diagramme de classes spécifie l'aspect statique du modèle, en définissant les objets abstraits du SUT. Par rapport au contexte MBT, les similarités sont nombreuses. Les *classes* modélisent les objets métiers (on y trouve notamment la classe *SUT*, qui représente le système sous test, et qui porte les points de contrôle et d'observation). Les *associations* modélisent les relations entre les objets métiers. Les *énumérations* modélisent des ensembles de valeurs abstraites, et les *litéraux* modélisent ces valeurs. Les *attributs de classe* modélisent les caractéristiques évolutives des objets métiers. Les *opérations de classe* modélisent les points de contrôle et d'observation (PCOs) du SUT (on y trouve notamment les opérations de navigation entre pages). Toutefois, notre approche MBVT se distingue du contexte général MBT par (cf. figure 4):

– la présence des deux classes (*page* et *field*) et de leurs relations, qui modélisent respectivement la structure générale de l'application et les champs utilisateurs susceptibles d'être injectables avec des données malicieuses;

– la présence d'opérations supplémentaires, issues des Schémas de Test, qui modélisent des moyens de mise en œuvre et d'observation d'une attaque;

– la présence d'une énumération supplémentaire qui modélise les données malicieuses injectables dans les champs utilisateur.

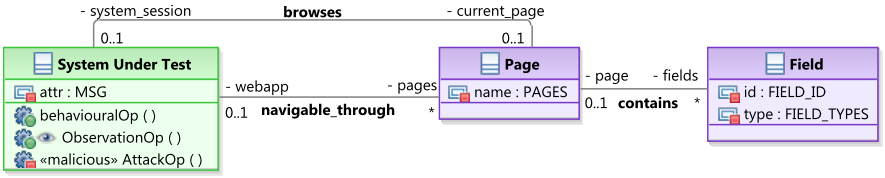

Figure 4. Diagramme de classes de la structure du SUT, dans le contexte MBVT

Le diagramme d'états UML spécifie graphiquement l'aspect dynamique du SUT, en modélisant les navigations entre les pages Web de l'application. Les *états* UML modélisent les pages Web, et les *transitions* UML modélisent les navigations possibles entre ces pages Web. Les *déclencheurs* des transitions sont les opérations UML de la classe *SUT*. Les *gardes* des transitions (spécifiées en OCL) définissent le contexte précis de franchissement. Les *effets* des transitions (spécifiés en OCL également) définissent précisément les modifications induites par le franchissement d'une transition. Le diagramme d'objets UML sert à modéliser l'état initial du SUT en instanciant les éléments du diagramme de classes. Ainsi, les *instances* UML modélisent les entités métiers présentes à l'état initial du SUT, et les *liens* UML modélisent les relations initiales entre ces entités. Dans notre approche, le diagramme d'objets modélise les pages Web de l'application et les champs utilisateur contenus dans ces pages.

EXEMPLE (DVWA). — La figure 5 présente le modèle de classes adapté pour l'exemple fil rouge DVWA (les classes *Page* et *Field* ne sont pas représentées).

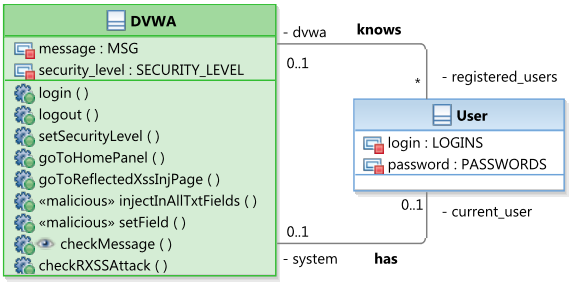

Figure 5. Diagramme de classes de DVWA

On remarque notamment que: (i) la classe User a été ajoutée afin de modéliser les utilisateurs potentiels de l'application; (ii) les attributs *message* et *security_level* modélisent respectivement les messages d'information de l'application et le niveau de sécurité de l'application; (iii) les cinq premières opérations modélisent la partie nécessaire et suffisante de l'application permettant d'atteindre la page que l'on souhaite tester avec le niveau de sécurité désiré; (iv) les opérations *injectAllFields* et *injectField*, mots-clés des Schémas de Test, modélisent l'injection d'une donnée malicieuse sur tout ou partie des champs utilisateur d'une page; (v) l'opération *checkMessage* modélise l'observation de l'attribut *message*; (vi) l'opération *checkRXSSAttack* modélise l'étape d'observation de l'attaque, et sert d'oracle.

De plus, concernant l'aspect statique du modèle, nous avons modélisé certains variants de données malicieuses par des littéraux : *RXSS_DUMMY* est un variant basique, *RXSS_COOKIE1* et *RXSS_COOKIE2* sont deux variants permettant de récupérer des informations sensibles, *RXSS_WAF_EVASION* est un variant permettant de contourner certains pare-feux applicatifs (cf. la section 3.4 pour leur concrétisation).
La figure 6 présente le diagramme d'états modélisant le comportement de DVWA. On remarque notamment que ce diagramme définit des précédences dans les navigations entre pages : il faut être authentifié pour accéder aux autres pages de l'application.

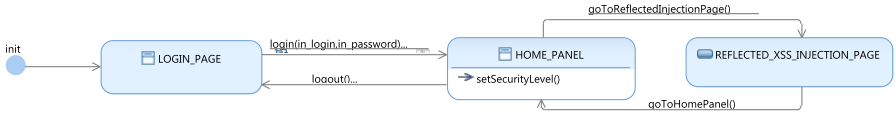

Figure 6. Diagramme d'états de DVWA

Finalement, la figure 7 présente l'état initial du modèle de DVWA. On y trouve : (i) un utilisateur avec ses informations de connexion; (ii) les pages et champs utilisateur de DVWA.

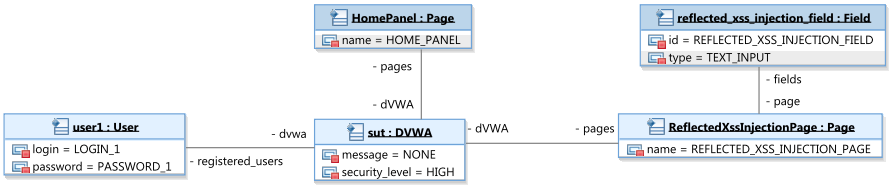

Figure 7. Diagramme d'objets de DVWA

□

3.3. Génération des tests

L'activité de génération des tests permet principalement de produire les tests à partir du modèle, en respectant les Schémas de Test. Trois phases sont identifiables. La première phase consiste à transformer le modèle et les Schémas de Test en éléments exploitables pour la technologie MBT de l'outil *CertifyIt*. Notamment, les Schémas de Test sont transformés en *cibles de test*, chaque cible de test étant une séquence d'*objectifs intermédiaires* interprétables par le générateur. Ainsi, la séquence d'étapes d'un Schéma de Test est traduite en une séquence d'objectifs intermédiaires de cible de test. De plus, cette première phase gère la combinatoire de valeurs entre les itérateurs des Schémas de Test, de sorte qu'un unique Schéma de Test produit autant de cibles de test que de combinaisons possibles de valeurs des itérateurs.
La seconde phase consiste à produire les *test abstraits* à partir des cibles de test. Cette phase est à la charge du générateur de test. Un cas de test abstrait est une séquence de *pas de test*, où un pas de test correspond à un appel d'opération complètement

valué. Un appel d'opération représente soit une stimulation soit une observation du SUT. Chaque cible de test produit un cas de test qui (i) vérifie la séquence d'objectifs intermédiaires, et (ii) vérifie les contraintes du modèle. Il est important de noter qu'un objectif intermédiaire (et par filiation, une étape d'un Schéma de Test) peut être transformé en un ensemble de pas de test. La figure 8 présente un test issu du Schéma de Test de la figure 3. On remarque notamment que les cinq premiers pas de test correspondent à la première étape du Schéma de Test.
Finalement, la troisième phase consiste à exporter les tests abstraits vers l'environnement d'exécution des tests. Dans notre cas, il s'agit de créer une suite de tests de type JUnit, où chaque test abstrait est transformé en un test JUnit. Dans le même temps une interface est créée. Elle spécifie chacune des opérations du SUT, et son implémentation dépend de l'ingénieur d'automatisation.

EXEMPLE (DVWA). — Nous disposons de deux Schémas de Test. Nous avons introduit quatre vecteurs malicieux dans l'objectif de tester une seule page contenant un seul champ. Les deux Schémas de Test produisent 12 cibles de test chacun, et chaque cible de test produit un test, pour un total de 24 tests abstraits.
La figure 8 présente un test abstrait généré. Il s'interprète de la manière suivante : (i) on s'authentifie auprès du système; (ii) on définit le niveau de sécurité; (iii) on se dirige vers la page cible; (iv) l'observation *checkMessage* permet de vérifier que la séquence de pas de test fonctionnel s'est déroulée correctement; (v) puis on injecte un vecteur malicieux; (vi) et finalement l'observation *checkRXSSAttack* vérifie si la faille applicative est présente. Cette dernière étape établit le verdict du test.
Concernant le Schéma de Test qui cible plus particulièrement un champ utilisateur, les tests diffèrent sur le pas de test #6, où c'est l'opération *injectField* qui est utilisée.

```
1  sut.login(LOGIN_1,PASSWORD_1)
2  sut.checkMessage() = WELCOME
3  sut.setSecurityLevel(MEDIUM)
4  sut.goToReflectedXssInjPage()
5  sut.checkMessage() = REFLECTED_XSS_MESSAGE
6  sut.injectAllFields(RXSS_DUMMY)
7  sut.checkRXSSAttack(RXSS_DUMMY)
```

Figure 8. Cas de test abstrait

□

3.4. Couche d'adaptation et exécution des tests

Au cours de l'activité de modélisation, chaque page, champ, vecteur malicieux, nom d'utilisateur, mot de passe, etc .. soit toutes les données manipulées par le système, sont représentées de manière abstraite. La suite de tests générée ne peut donc être exécutée en l'état, et il est nécessaire de créer un pont entre les tests abstraits et le système sous test. Cette jonction se fait par l'association de chaque valeur abstraite avec une valeur concrète qui peut être manipulée par le SUT : c'est l'activité de concrétisation. Afin de facilité la compréhension de notre approche, nous présentons une concrétisation de type $1 \leftrightarrow 1$, mais l'utilisation de tableaux de valeurs permettant des concrétisations multiples est aussi utilisée.

Les opérations de stimulation du SUT nécessitent aussi d'être concrétisées. L'outil de génération fournit une interface qui spécifie chaque opération, mais il est de la responsabilité de l'ingénieur de test d'implémenter cette interface pour pouvoir implémenter l'exécution automatique des tests. Puisqu'il s'agit de tester des applications Web, nous avons étudié deux techniques de concrétisation des tests :

– Au niveau *navigateur :* c'est une technique basée sur Selenium qui consiste à solliciter le système depuis un navigateur Web. Cette technique s'avère chronophage mais peut être utile pour les applications qui nécessitent l'exécution de scripts.

– Au niveau *HTTP :* il peut être préférable d'opérer à un niveau inférieur pour parcourir l'application Web. Le but ici est d'envoyer des requêtes HTTP et de stocker / analyser les réponses, en utilisant la bibliothèque Apache HTTPClient. Cette technique est très rapide et peut aider à contourner les restrictions HTML et Javascript.

La dernière activité du processus MBVT consiste à exécuter les tests concrétisés sur le système pour parvenir à un verdict. Nous introduisons une terminologie adaptée pour caractériser le résultat d'une exécution de test :

– *Attack-pass* : l'exécution complète d'un test met en évidence que l'application est faillible, à l'inverse du MBT, où il mettrait en évidence que le scénario testé est correctement implémenté;

– *Attack-fail* : l'échec de l'exécution d'un test sur la dernière étape met en évidence que l'application résiste à l'attaque, à l'inverse du MBT, où un tel échec mettrait en évidence un problème au niveau du SUT;

– *Inconclusive* : il n'est pas toujours certain qu'une vulnérabilité ait été découverte. Une anomalie s'est produite, mais aucune vulnérabilité connue n'a pu être observée.

EXEMPLE. — Nous avons utilisé quatre vecteurs malicieux pour tester la résistance du SUT contre les attaques RXSS. Dans le modèle, ces vecteurs sont représentés par des littéraux d'énumération, et pendant l'activité d'adaptation nous associons chacun d'eux à un vecteur réel. Voici les valeurs concrètes utilisées pour la stimulation :

```
1  RXSS_DUMMY ↔ <>
2  RXSS_COOKIE1 ↔ <script>alert(document.cookie)</script>
3  RXSS_COOKIE2 ↔ <img src="foo" onerror="javascript:alert(document.cookie)"/>
4  RXSS_WAF_EVASION ↔ <scr<script>ipt>alert(document.cookie)</script>
```

Nous avons concrétisé les opérations du SUT selon les deux techniques d'implémentation (Selenium et HTTPClient). Cependant, nous avons principalement utilisé l'approche au niveau HTTP puisqu'elle représente, vis-à-vis de l'exemple DVWA, un gain de temps considérable pour des résultats similaires. Suite à l'exécution de la suite de tests, 50% des tests ont été fléchés *attack-pass* : les deux premiers vecteurs en mode low, le troisième vecteur en mode low et medium, et le quatrième vecteur en mode medium uniquement. Ces résultats concordent avec ceux que nous avons obtenus suite à la mise en œuvre manuelle des attaques. Cette concordance permet une validation de notre approche sur le sous-ensemble de vulnérabilité que nous avons défini, dans le contexte de l'application DVWA.

□

4. État de l'art

Le domaine de l'automatisation du test de vulnérabilité d'applications Web est très actif, tant du point de vue des offres commerciales et des outils en logiciel libre, que du point de vue de la recherche académique. Ceci provient de la sensibilité accrue aux risques de sécurité pour des applications Web qui se multiplient dans tous les domaines du logiciel avec l'émergence du cloud.
On compte près d'une vingtaine d'outils commerciaux et plus d'une quarantaine d'outils en logiciel libre [9], automatisant le test dynamique de vulnérabilité d'applications Web (outils appelés "Web application vulnerability scanner"). On trouve par exemple les outils IBM Appscan et Acunetix Web Application Scanner qui ressortent en tête des essais comparatifs réalisés [10]. Ces outils partagent un même fonctionnement consistant en l'analyse de la navigation de l'application Web (extraction des pages, phase appelée "crawling"), puis à l'injection de vecteurs d'attaque pour détecter des vulnérabilités (Bau *et al.*, 2010). L'article de (Doupé *et al.*, 2010) met bien en évidence les faiblesses de ce type d'outils de test de vulnérabilité en mode boîte noire. D'une part le parcours de l'application est souvent partiel, dû à une méconnaissance applicative qui empêche d'accéder à certaines parties de l'application. Cela limite la détection des vulnérabilités aux pages les plus accessibles de l'application Web. D'autre part, l'analyse des résultats des injections des vecteurs d'attaque conduit à de nombreux faux positifs. Ainsi, des essais réalisés [10] mettent en évidence 30% de faux positifs avec l'outil IBM Appscan (le leader du marché). Cela corrobore nos propres analyses lors d'expérimentations avec ces outils. De même, dans une étude récente (Finifter, Wagner, 2011), ont montré sur un exemple applicatif, que l'analyse manuelle de code détectait 1,5 fois plus de vulnérabilités que le test boite noire automatisé.
Les approches de test de sécurité à partir de modèles visent à dépasser les limites des outils actuels, tant du point de vue de la couverture des vulnérabilités détectées (en se dotant de capacité plus "intelligente" de détection de vulnérabilités) que dans la réduction de la production de faux positifs au niveau du verdict (Schieferdecker *et al.*, 2012). On distingue trois types d'approches principales dans l'état de l'art scientifique en génération de tests de vulnérabilité à partir de modèles : à partir d'un modèle d'attaques, à partir d'un modèle du système et particulièrement de protocoles de sécurité, et des approches combinant les deux.
(Buchler *et al.*, 2012) et (Dadeau *et al.*, 2011) présentent une approche fondée sur la mutation d'une formalisation du protocole de sécurité, en s'appuyant sur des analyses symboliques du modèle. Ces mutations établissent des traces qui constituent, après traitement, des cas de test complexes susceptibles de mettre en évidences des failles de sécurité. (Schieferdecker, 2012) proposent une approche dite de "Model-based fuzzing" qui s'appuie sur des séquences d'actions sur le systèmes définies sous la forme de diagrammes de séquences UML. Ces traces sont modifiées (mutées) pour obtenir des test définis par des enchaînements non prévus (par exemple sur le protocole d'au-

9. Voir http://www.sectoolmarket.com/
10. Voir http://sectooladdict.blogspot.co.il/2012/07/2012-web-application-scanner-benchmark.html

thentification) pour essayer de détecter des failles. (Wang *et al.*, 2007) utilisent un modèle de menace, produit sous forme de diagrammes de séquences UML, qui caractérise des séquences d'actions que pourrait réaliser un attaquant. Ces modèles sont ensuite parcourus pour générer et automatiser les tests de vulnérabilité.
En synthèse, les approches de génération de tests de vulnérabilité à partir de modèles utilisent une modélisation soit du comportement du système sous test (par exemple d'un protocole de sécurité), soit du comportement de l'attaquant. Dans notre approche, nous mixons au sein de la modélisation la représentation des comportements applicatifs (pour la navigation) et la modélisation d'actions d'attaque et d'observation sur le système sous test. Cette mixité d'approche permet de couvrir en profondeur l'application par la connaissance des comportements applicatifs, mais aussi d'adapter finement la construction des vecteurs d'attaque aux spécificités de la page Web traitée.

5. Conclusion et travaux futurs

Nous avons présenté dans cet article une approche pour l'automatisation du test de vulnérabilité Web à partir de modèles. Cette approche s'appuie sur une modélisation mixte des comportements applicatifs et d'actions d'attaque, composée avec des Schémas de Test qui formalisent la procédure de test de vulnérabilité.
Cette approche est générale car la flexibilité de la répartition des informations entre la modélisation comportementale, les schémas de test et la couche d'adaptation, permet de prendre en compte des types de vulnérabilité variés tels que les attaques RXSS. La précision de la modélisation et des Schémas de Test permet d'être très ciblé sur un type de vulnérabilité particulier, et donc d'obtenir des bons résultats sur la précision des tests et des verdicts. C'est un apport vis-à-vis des outils actuels de test de vulnérabilité mais aussi par rapport aux approches de Fuzzing à partir de modèles.
Les limites principales de notre approche proviennent justement de cette précision : elle demande un effort de modélisation comportementale en articulation avec la formalisation des schémas et le développement de la couche d'adaptation. Pour réduire ces efforts, nous poursuivons plusieurs directions de recherche :

– rendre les schémas de test génériques pour chaque classe de vulnérabilité considérée, et donc permettre leur réutilisabilité d'un projet à un autre ;

– prédéfinir le modèle en fonction de la plateforme de développement Web utilisée pour l'application testée. Par exemple, pour une application de eCommerce développée avec la plateforme de logiciel libre Magento (leader des plateformes pour le eCommerce), une partie de la modélisation peut être prédéfinie et réutilisable ;

– en lien avec cette prédéfinition du modèle, la prédéfinition de la couche d'adaptation, pour une plateforme Web, peut être réalisée et rendue réutilisable.

Ces éléments sont un moyen pour faciliter le déploiement d'une approche de test de vulnérabilité à base de modèle. Ces directions de recherche constituent nos travaux en cours et à venir pour améliorer l'automatisation du test de vulnérabilité d'applications Web, en augmentant la couverture applicative et en diminuant les verdicts de faux positifs dont souffrent les outils actuels.

Remerciements

Ce travail est supporté par le projet FSN DAST (http://projects.femto-st.fr/dast) et le projet ITEA2 DIAMONDS (http://www.itea2-diamonds.org).

Bibliographie

Bau J., Bursztein E., Gupta D., Mitchell J. (2010, May). State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proceedings of the* 31^{th} *Int. Symposium on Security and Privacy (SP'10)*, p. 332–345. Oakland, CA, USA, IEEE Computer Society.

Botella J., Bouquet F., Capuron J.-F., Lebeau F., Legeard B., Schadle F. (2013, March). Model-based Testing of Cryptographic Components Lessons Learned from Experience. In *Int. Conf. on Software Testing, Verification and Validation (ICST'13).* Luxembourg, IEEE CS.

Bouquet F., Grandpierre C., Legeard B., Peureux F. (2008, May). A test generation solution to automate software testing. In *Proceedings of the* 3^{rd} *Int. Workshop on Automation of Software Test (AST'08)*, p. 45–48. Leipzig, Germany, ACM Press.

Bouquet F., Grandpierre C., Legeard B., Peureux F., Vacelet N., Utting M. (2007, July). A subset of precise UML for model-based testing. In *Proceedings of the* 3^{rd} *Int. Workshop on Advances in Model Based Testing (A-MOST'07)*, p. 95–104. London, UK, ACM Press.

Buchler M., Oudinet J., Pretschner A. (2012, June). Semi-Automatic Security Testing of Web Applications from a Secure Model. In *Proc. of the* 6^{th} *IEEE Int. Conf. on Software Security and Reliability (SERE'12)*, p. 253–262. Gaithersburg, MD, USA, IEEE Computer Society.

Dadeau F., P-C.Héam, Kheddam R. (2011, March). Mutation-Based Test Generation from Security Protocols in HLPSL. In *Proc. of the* 4^{th} *IEEE Int. Conf. on Software Testing, Verification and Validation (ICST'11)*, p. 240–248. Berlin, Germany, IEEE Computer Society.

Doupé A., Cova M., Vigna G. (2010, July). Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In *Int. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'10)*, p. 111–131. Bonn, Germany, Springer.

Finifter M., Wagner D. (2011, June). Exploring the relationship between web application development tools and security. In *Proc. of the* 2^{nd} *USENIX Conference on Web Application Development (WebApps'11)*, p. 99–111. Portland, OR, USA, USENIX Association.

Julliand J., Masson P.-A., Tissot R. (2008, May). Generating security tests in addition to functional tests. In *Proceedings of the* 3^{rd} *International Workshop on Automation of Software Test (AST'08)*, p. 41–44. Leipzig, Germany, ACM Press.

Schieferdecker I. (2012, April). *Model-Based Fuzzing for Security Testing.* Keynote talk at the 3^{rd} International Workshop on Security Testing (SECTEST'12), Montreal, Canada.

Schieferdecker I., Grossmann J., Schneider M. (2012, March). Model-based security testing. In *Proceedings of the* 7^{th} *International Workshop on Model-Based Testing (MBT'12)*, vol. 80, p. 1–12. Tallinn, Estonia, Open Publishing Association.

Utting M., Legeard B. (2006). *Practical model-based testing - a tools approach* (Morgan, Kaufmann, Eds.). San Francisco, CA, USA, Elsevier Science.

Wang L., Wong E., Xu D. (2007, May). A threat model driven approach for security testing. In *Proceedings of the* 3^{rd} *Int. Workshop on Software Engineering for Secure Systems (SESS'07).* Minneapolis, MN, USA, IEEE Computer Society.