Efficient Detection of Multi-step Cross-Site Scripting Vulnerabilities

Alexandre Vernotte[1], Frédéric Dadeau[1,2], Franck Lebeau[3], Bruno Legeard[1,4], Fabien Peureux[1], and François Piat[1]

[1] Institut FEMTO-ST, UMR CNRS 6174 – Route de Gray, 25030 Besançon, France
{avernott,fdadeau,blegeard,fpeureux,fpiat}@femto-st.fr

[2] INRIA Nancy Grand Est – BP 239, 54506 Vandoeuvre-lès-Nancy, France
frederic.dadeau@inria.fr

[3] Erdil – 9, Avenue des Montboucons, 25000 Besançon, France
franck.lebeau@erdil.com

[4] Smartesting R&D Center – 2G, Avenue des Montboucons, 25000 Besançon, France,
bruno.legeard@smartesting.com

Abstract. Cross-Site Scripting (XSS) vulnerability is one of the most critical breaches that may compromise the security of Web applications. Reflected XSS is usually easy to detect as the attack vector is immediately executed, and classical Web application scanners are commonly efficient to detect it. However, they are less efficient to discover multi-step XSS, which requires behavioral knowledge to be detected. In this paper, we propose a Pattern-driven and Model-based Vulnerability Testing approach (PMVT) to improve the capability of multi-step XSS detection. This approach relies on generic vulnerability test patterns, which are applied on a behavioral model of the application under test, in order to generate vulnerability test cases. A toolchain, adapted from an existing Model-Based Testing tool, has been developed to implement this approach. This prototype has been experimented and validated on real-life Web applications, showing a strong improvement of detection ability w.r.t. Web application scanners for this kind of vulnerabilities.

Keywords: Vulnerability Testing, Model-Based Testing, Vulnerability Test Patterns, Web Applications, Multi-step Cross-Site Scripting.

1 Introduction

Code injection security attacks, and more particularly cross-site scripting (XSS), are part of the most prevalent and dangerous cyber-attacks against Web applications reported these last years; see, for example, OWASP Top Ten 2013 [29], CWE/SANS 25 [20] and WhiteHat Website Security Statistic Report 2013 [28]. In this latter, XSS appears to represent 43% of all the serious vulnerabilities discovered in a large panel of Web applications. As another example, Claudio Criscione reports at GTAC 2013 that nearly 60% of security bugs detected in Google software are XSS vulnerabilities[5].

[5] https://developers.google.com/google-test-automation-conference/2013/presentations#Day2Presentation7 [Last visited: July 2014]

An XSS vulnerability occurs each time an application stores (with more or less persistence) a user input and displays it into a Web browser without proper sanitization (without removing or replacing any character that may contribute to an unwanted behavior). Therefore, it is possible to inject a piece of code and see this code executed by the Web browser, potentially causing severe damage to visitors (often without them knowing). XSS attacks is easy to put into practice, and presents a great number of variants. It is also an entry point for many exploits (session hijacking, credentials stealing, etc.). The difficulty of handling XSS issues is mainly due to the complexity of the application logics. Indeed, developers need to think about a systematic protection of the displayed data, what is an error-prone exercise, since a given user input may be subsequently displayed in a large variety of places in the application. It is thus mandatory to detect XSS-related issues at the earliest, by performing vulnerability testing at the application level. XSS vulnerabilities can be classified into four categories[6]:

- **(i) DOM-based XSS** when the injected data stay within the browser (and modify the DOM "environment"),
- **(ii) Reflected XSS** when the untrusted injected data are directly displayed/executed right after being injected,
- **(iii) Stored XSS** when the injected data is stored by the application and retrieved later in another context (e.g., in a user's profile),
- **(iv) Multi-step XSS** (a special breed of stored XSS) when it requires that the user performs several actions on the applications (mainly navigation steps) to display/execute the attack vector.

While the first three categories are usually well-identified and easily detected by automated penetration testing tools, such as Web application vulnerability scanners [2], the last one remains a challenging issue [10]. On the one hand, manual vulnerability testing is becoming more and more difficult as Websites are growing in size and complexity: indeed, as the result of an attack cannot be seen immediately, the penetration tester has to dig into the application logics to understand where a given user input is supposed to be sent back to the client. On the other hand, current automated vulnerability discovery techniques can test for a large percentage of technical vulnerabilities, but are often limited in accessing large parts of the Web application, because they lack any knowledge about the functional behaviour and the business logics of the application.

Recently, vulnerability test patterns have been introduced to describe a testing procedure for each class of vulnerabilities [25]. However, such a process remains manual, and using vulnerability test patterns for testing automation is still a challenge. In addition, the current automated vulnerability testing tools (i.e. Web application scanners) often display false positive and false negative results, raising alarms when there is no error or missing potential weaknesses, respectively. Hence, it is the cause of a useless and costly waste of time.

[6] `https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting\_(XSS)`[Last visited: July 2014]

The approach presented in this paper aims to improve the accuracy and precision of multi-step XSS testing, by proposing a testing approach driven by automated *vulnerability test patterns* composed with a *behavioral model* of the system under test. These patterns describe generic test scenarios that assess the robustness of the Web application w.r.t. a given kind of vulnerability. To achieve that, it relies on the information contained in the behavioral model, especially the location of the possible user inputs and their associated resurgences, to check that user inputs are correctly sanitized before being displayed on a Web page. As a major result, this approach increases the efficiency of penetration testers for detecting vulnerabilities such as multi-step XSS. The main contribution of this paper relates to the proposal of a pattern-driven and model-based approach to generate vulnerability tests for Web applications. More precisely, this concerns:

– The formalization of vulnerability test patterns using generic test purposes to drive the test generation engine, including a combinatorial unfolding of untrusted injected data taken from standard databases, such as the OWASP collection of attack vectors[7].
– The separation of the behavioral model for Web application vulnerability testing between a generic part (whatever the application under test is) and an ad-hoc part, which is specific to the targeted application under test.
– The full automation of the testing process, including test generation, test execution and verdict assignment.

The paper is organized as follows. Section 2 introduces the principles of XSS-based attacks, and illustrates them on a running example of a vulnerable Web application named WackoPicko. Section 3 describes the contribution of the paper, namely our pattern-driven and model-based vulnerability testing approach. It especially defines the content of the behavioral model and the expressiveness of the test pattern language, which are the key artefacts of the approach. Experience reports are provided and experimental results are discussed in Sect. 4. Finally, the related work is presented in Sect. 5, while conclusion and future works are given in Sect. 6.

2 Challenges of Detecting Multi-Step Cross-Site Scripting Vulnerabilities

This section introduces the challenge of detecting multi-step XSS vulnerabilities, and illustrates this issue on a running example of a vulnerable Web application. More precisely, this section aims to explain and exemplify the difficulties faced by mainstream automated penetration testing tools (i.e. commercial or open-source Web scanners) for accurately detecting multi-step XSS vulnerabilities. Based on these conclusions, we finally expose the research questions we are addressing to efficiently detect such multi-step XSS vulnerabilities.

[7] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet` [Last visited: July 2014]

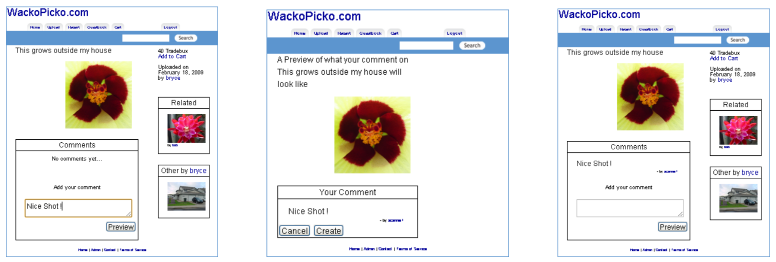
2.1 Running Example: the WackoPicko Web Application

First of all, in order to illustrate multi-step XSS vulnerability and to evaluate the accuracy and precision of our approach, we use the Web application called WackoPicko[8], which is a deliberately-unsecured Web application developed by Adam Doupé [10]. The objective of this test bed, developed using PHP/MySQL, is to provide a realistic but vulnerable environment. Like education-oriented vulnerable Web applications such as DVWA (Damn Vulnerable Web Application[9]) or WebGoat[10], WackoPicko can aid security professionals to learn, improve or test their skill in vulnerability discovery on a realistic Web application, with nowadays features (e.g., posts, comments) and realistic workflows. It can also be used to test Web security testing tools, like vulnerability scanners for instance.

Basically, WackoPicko allows users to authenticate themselves, share pictures, comment pictures, and possibly buy pictures. WackoPicko presents realistic features (authentication, shopping, ...) that can be found in many Websites, along with more complex workflows (e.g., uploading a picture, commenting the picture). It embeds several vulnerabilities, notably SQL Injection, Cross-Site Scripting, Cross-Site Request Forgery and Local/Remote File Inclusion, which are ranked by the OWASP project among the most frequently used attacks.

2.2 Multi-step XSS Principles and Illustration

The main characteristic of a multi-step vulnerability is that the attack vector is injected in one page, saved (e.g., in a database), and then echoed later in another page or another application. Hence, detecting such a vulnerability involves being able to perform a sequence of actions starting from attack vector injection until vulnerability checking. For instance, using the WackoPicko example, such a sequence appears when a user adds a comment to a picture. The corresponding workflow, depicted in Fig. 1, is now described.



(a) Comment setting (b) Comment preview (c) Comment display

Fig. 1. Nominal workflow of picture comment using WackoPicko

[8] `https://github.com/adamdoupe/WackoPicko` [Last visited: July 2014]
[9] `http://www.dvwa.co.uk/` [Last visited: July 2014]
[10] `https://www.owasp.org/index.php/Category:OWASP_WebGoat_Project` [Last visited: July 2014]

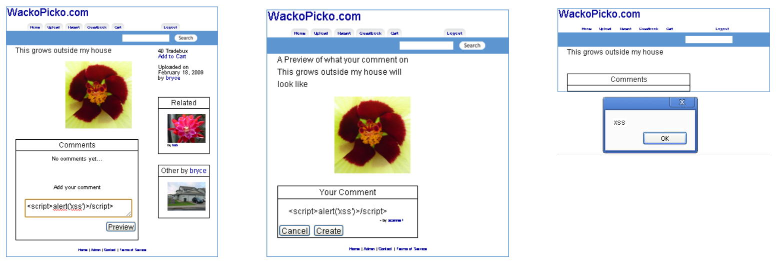
0. Prerequisites. This preliminary step consists in logging the user on the application, and browsing the application until viewing a particular picture.

1. Setting a comment. In this step (see Fig. 1(a)), the user sets his new comment in the text area, and clicks the *Preview* button. By clicking the button, the client (i.e. the browser) sends a *POST* request to the Web server.

2. Preview of the comment. This step (see Fig. 1(b)), consists of visualizing the comment before validation by the user. When the server receives the *POST* request, it stores the new comment in the *comments_preview* table of its database. Then, the server sends back to the browser a new page that displays a preview of the comment. The user may accept or reject its comment with the respecting *Create* and *Cancel* buttons. By clicking the *Create* button, the browser sends a *POST* request to the server.

3. Displaying the comment. The final step (see Fig. 1(c)) consists in displaying a validated comment. When the server receives the previous *POST* request, it concretely relates the comment to the picture, making this comment available every time the picture page is displayed.

A malicious attack can consist of injecting a piece of code (for instance the vector `<script>alert("XSS")</script>`) in the text area, previewing, creating, and visualizing the result. What makes this attack a multi-step XSS attack is the fact that only the picture page is vulnerable to XSS: the injected vector is properly sanitized on the comment preview page and it thus requires an extra step from the user (validating the comment) to detect the vulnerability. The corresponding workflow, depicted in Fig. 2, shows that the attack vector injected as picture comment (see Fig. 2(a)) is next interpreted as Javascript code (and not as a harmless string) and the alert window is displayed (see Fig. 2(c)).


WackoPicko.com
This grows outside my house
Comments
No comments yet...
Add your comment
<script>alert('xss')</script>
Preview
Related
WackoPicko.com
A Preview of what your comment on This grows outside my house will look like
Your Comment
<script>alert('xss')</script>
Cancel
Create
WackoPicko.com
This grows outside my house
Comments
XSS
OK

(a) Comment setting (b) Comment preview (c) Comment display

Fig. 2. Multi-step XSS attack workflow of picture comment using WackoPicko

It should be noted that the comment preview page is not vulnerable to XSS attack (see Fig. 2(b)): there is no alert message since the attack vector is treated as a standard string in which special characters are encoded. Indeed, the source code of the page embeds the harmless HTML-encoded attack vector (`&lt;script&gt;alert("XSS")&lt;/script&gt;`). This prevents the `<script>` tag from being interpreted by the browser.

2.3 Research Questions

As illustrated in the previous section, whereas it is mostly easy to automatically detect reflected XSS, multi-step XSS are far more difficult to discover. Indeed, the untrusted data are not immediately displayed/executed after they are injected, and several navigation steps to display/execute the attack vector are required to record the breach. Current vulnerability detection techniques highly struggle with this problem, mostly because it requires knowledge of the logic of the application under test to navigate from an injection point to its output page. Hence, within our work, we aim to address the following research questions:

RQ1 To what extend does the knowledge of the business logic of the application help to increase the accuracy of the detection for multi-step XSS?
RQ2 To what extend is it possible to automatize generic test patterns dedicated to such Web application vulnerabilities?
RQ3 To what extend test execution and verdict assignment can be fully automated?
RQ4 To what extend is it possible to improve the overall efficiency of the process with respect to manual penetration testing activities and state of the practice by means of automated penetration testing techniques and tools?

To achieve this goal, the proposed testing approach consists to combine formalized test patterns with a behavioral model focused on the business logic for vulnerability testing of Web applications. Formalized test patterns provide penetration testing scenarios, and the model provides the minimal but necessary required information, namely: states/pages and transitions/navigation combined with logical application data and dataflow information. The next section introduces this testing approach, called Pattern-driven and Model-based Vulnerability Testing (PMVT).

3 Pattern-driven and Model-based Vulnerability Testing for Multi-step XSS

This section introduces a Pattern-driven and Model-based Vulnerability Testing (PMVT) approach, which is a generic solution for Web application vulnerability testing. We first describe the principles of the approach, before giving information on the different artefacts that it involves, namely a behavioral model and test purposes implementing a vulnerability test pattern.

3.1 Principles of the PMVT Approach

The PMVT process, depicted in Fig. 3, is composed of four activities:

① The *Test Purposes* design activity consists of formalizing a test procedure from vulnerability test patterns that the generated test cases have to cover. These Test Purposes can be generic to be applied for a category of application. We show later that the Test Purpose for multi-step XSS is generic whatever the Web application is.

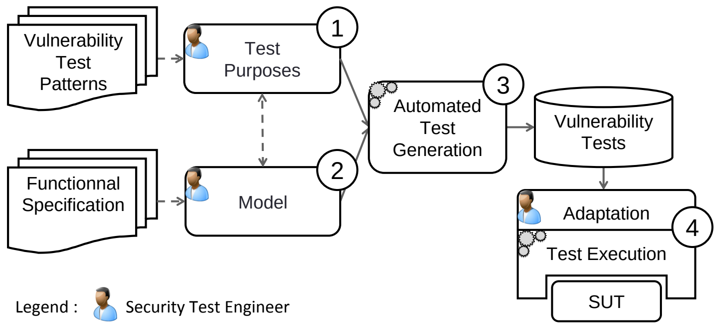

Vulnerability Test Patterns
Test Purposes
1
Functionnal Specification
Model
2
Automated Test Generation
3
Vulnerability Tests
Adaptation
Test Execution
4
SUT
Legend : Security Test Engineer

Fig. 3. Pattern-driven and Model-based Vulnerability Test process

② The *Modeling* activity aims to design a test model that captures the behavioral aspects of the application under test to generate consistent (from a functional point of view) sequences of stimuli.

③ The *Test Generation* activity consists of automatically producing abstract test cases, including expected results, from the artifacts defined during the two previous activities.

④ The *Adaptation, Test Execution and Observation* activity aims to (*i*) translate the generated abstract test cases into executable scripts, (*ii*) to execute these scripts on the application under test, (*iii*) to observe the responses and to compare them to the expected results in order to assign the test verdict and automate the detection of vulnerabilities.

All these activities are supported by a dedicated toolchain, based on an existing Model-Based Testing (MBT) software named *CertifyIt* [18] provided by the company Smartesting[11]. CertifyIt is a test generator that takes as input a test model, written with a subset of UML (called UML4MBT [3, 8]), capturing the behavior of the application under test. A UML4MBT model consists of (*i*) UML class diagrams to represent the static view of the system (with classes, associations, enumerations, class attributes and operations), (*ii*) UML object diagrams to define the data and entities (used to compute test cases) that exist at the initial state, and (*iii*) statechart diagrams (annotated with OCL constraints) to specify the dynamic view of the application under test. Such UML4MBT models have a precise and unambiguous meaning, so that those models can be understood and computed by the *CertifyIt* technology. This precise meaning makes it possible to simulate the execution of the models and to automatically generate test cases by applying the strategies given by the test purposes. Each generated test case is typically an abstract sequence of high-level actions from the UML4MBT models. These generated test cases contain the sequence of stimuli to be executed, but also the expected results (to perform the observation activity and automate the verdict assignment), obtained by resolving the associated OCL constraints. The next sections describe each of the activities and illustrate them using the WackoPicko running example.

[11] `http://www.smartesting.com` [Last visited: July 2014]

3.2 Formalizing Vulnerability Test Patterns into Test Purposes

A Vulnerability Test Patterns (vTP) is a normalized textual document describing the testing objectives and procedures to detect a particular flaw in a Web application. Hence, there are as much vTP as there are types of application-level flaws. Our approach is based on the vTP provided during the ITEA2 research project DIAMONDS[12] [26]. For instance, Fig. 4 presents an excerpt of the vTP defined for the multi-step XSS vulnerability. At this stage, Vulnerability Test Patterns are still textual. The PMVT approach takes such textual vTP as starting point by translating them into formal directives, called Test Purposes, in order to be able to automate testing strategy implementation and execution.

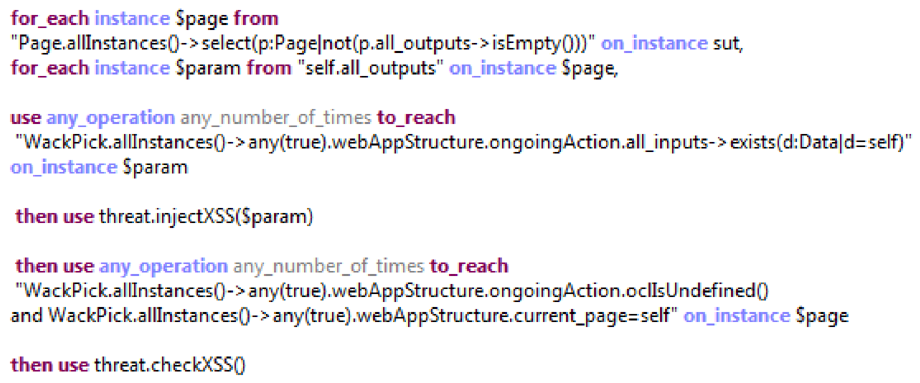
Name	multi-step XSS
Description	This pattern can be used on an application that does not check user inputs. An XSS attack can redirect users to a malicious site, or can steal user's private information (cookies, session, ...).
Objective(s)	Detect if a user input can embed attack vector enabling an XSS attack.
Prerequisites	N/A
Procedure	Identify a sensible user input, inject the attack vector *<script>alert(xss)</script>*.
Observation/ Oracle	Go to a page echoing the user input, check if a message box 'xss' appears.
Variant(s)	- attack vector variants: character encoding, Hex-transformation, comments insertion - procedure variants: attack can be applied at the HTTP level; the attack vector is injected in the parameters of the HTTP messages sent to the server, and we have to check if the attack vector is in the response message from the server
Known Issue(s)	Web Application Firewalls (WAF) filter messages send to the server (black list, clac regEx, ...); variants allows to overcome these filters
Affiliated vTP	Stored XSS
Reference(s)	CAPEC: `http://capec.mitre.org/data/definitions/86.html` WASC: `http://projects.Webappsec.org/w/page/13246920/CrossSiteScripting` OWASP: `https://www.owasp.org/index.php/Cross-site\_Scripting\_(XSS)`

Fig. 4. Vulnerability Test Pattern of multi-step XSS attack

A *test purpose* is a high-level expression that formalizes a testing objective to drive the automated test generation on the behavioral model. It has been originally designed to drive model-based test generation for security components, typically Smart card applications and cryptographic components [6]. Within PMVT context, a test purpose formalizes a given vTP in order to drive the vulnerability test generation on the behavioral model. Basically, such a test purpose is a sequence of significant *steps* that has to be exercised by the test case scenario in order to assess the robustness of the application under test w.r.t. the related vulnerability. Each step takes the form of a set of operations/behaviors to execute, or specific state to reach.

Figure 5 shows the WackoPicko test purpose formalizing the vTP presented in Fig. 4. This automatically generated test purpose specifies that, for all sensible pages echoing user inputs and for each user input of a given page, a test has to perform the following actions: (i) use any operation to reach a page showing

[12] `http://www.itea2-diamonds.org` [Last visited: July 2014]



```
for_each instance $page from
"Page.allInstances()->select(p:Page|not(p.all_outputs->isEmpty()))" on_instance sut,
for_each instance $param from "self.all_outputs" on_instance $page,

use any_operation any_number_of_times to_reach
 "WackPick.allInstances()->any(true).webAppStructure.ongoingAction.all_inputs->exists(d:Data|d=self)"
on_instance $param

 then use threat.injectXSS($param)

 then use any_operation any_number_of_times to_reach
 "WackPick.allInstances()->any(true).webAppStructure.ongoingAction.oclIsUndefined()
and WackPick.allInstances()->any(true).webAppStructure.current_page=self" on_instance $page

then use threat.checkXSS()
```

Fig. 5. Test purpose formalizing the vTP of multi-step XSS attack (Figure 4)

the XSS-sensitive user input, (*ii*) inject an attack vector in this user input, (*iii*) use any operation to reach a page echoing the user input, and (*iv*) check if the attack succeeded. It should be underlined that the structure of this test purpose, addressing multi-step XSS vulnerability, is fully generic. Moreover, since pages and user inputs are automatically retrieved from OCL constraints from the UML4MBT test model, this automated generation of test purpose can therefore be applied for any Web application.

3.3 Test Model Specification

As for every Model-Based Testing (MBT) approach, the modeling activity consists of designing a test model that will be used to automatically generate abstract test cases. Our approach, based on Smartesting technology, requires a model designed using the UML4MBT notation. To ease and accelerate this modeling activity, which is known to be time consuming, we have developed a Domain Specific Modeling Language (DSML), called *DASTML*, that allows to model the global structure of a Web application: the available pages, the available actions on each page, and the user inputs of each action potentially used to inject attack vectors. It solely represents all the structural entities necessary to generate vulnerability test cases. The transformation of a DASTML instantiation into a valid UML4MBT model is automatically performed by a dedicated plug-in integrated to the Smartesting modeling environment. The DASTML Domain Specific Modeling Language is composed of four entities:

Page. Page entities represent the different pages that compose the Web application under test. We follow the comparison technique proposed in [11], meaning that we may consider two physical pages as the same if they exactly provide the same action and navigation entities. On the contrary, we may consider a single physical page as two distinct pages if there is at some point a variation in the action and navigation entities. We also distinguish the initial page from the others by using a boolean attribute *is_init*.

Navigation. Navigation entity is typically a link or a button that takes the user to another page, without altering the internal state of the application nor triggering any function or service of the application.

Action. Action entities have pretty much the same form as navigation entities, but there are two main differences. First, an action entity may carry data (in case of a Web form for instance). Second, an action entity can alter the internal state of the application (e.g., any user interaction that has modify the database is considered as an action). In addition, the *is_auth* attribute allows to distinguish authentication actions from the others. This way, we can easily refer to it when the attacker has to log on the Web application.

Data. Data entity, defining any user input, is composed of a key and a value.

The metamodel of DASTML is depicted in Fig. 6. Entities interact with each other based on multiple relations. *Navigate_to* and *navigate_from* provide the source page and the target page of a navigation entity. Identically, *has_action* and *sends_users_to* provide the source page and the target page of an action entity. An action may be associated to one or more data (in case of a Web form for instance), with relation *has_data*. Data have a (*reflects*) relation to link them to one or more output page (in this way, for each user input, the page where it is rendered back is known, what is crucial for XSS vulnerability testing).

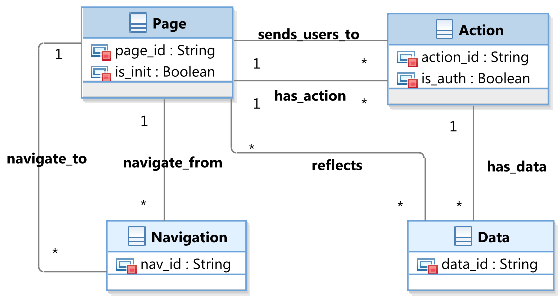


Fig. 6. Metamodel of the DASTML Domain Specific Modeling Language

The code fragment, introduced in Fig. 7, is an instantiation of DASTML to model the WackoPicko running example. In this DASTML model, the entry point is the "HOME" page, where users can navigate to the "LOGIN" page. There, users can authenticate themselves (see the *:auth* suffix on the "LOGIN" action), and if valid credentials are provided, they reach the "HOME_LOGGED_IN" page. At some point, users may visit a picture page. This page has a "COMMENT_PICTURE" action that requires a user input called "CP_CONTENT", which abstract value is "CONTENT1". This input is rendered back on two pages: "PICTURE_CONFIRM_COMMENT" and "PICTURE". Finally, the completion of the action redirects users to the "PICTURE_CONFIRM_COMMENT".

```
PAGES {
  "HOME" : init  {
    NAVIGATIONS {
      "GO_TO_LOGIN" -> "LOGIN"
    }
  }

  "LOGIN" {
    ACTIONS {
      "LOGIN" : auth  ("USERNAME" = "USER1",
                       "PASSWORD" = "PWD1")
        -> "HOME_LOGGED_IN"
    }
  }

  "HOME_LOGGED_IN" {
    NAVIGATIONS {
      "GOTO_RECENT_PICTURES"
        -> "RECENT_PICTURES",
      "GOTO_GUESTBOOK"
        -> "GUESTBOOK"
    }
  }

  "RECENT_PICTURES" {
    NAVIGATIONS {
      "SHOW_PICTURE" -> "PICTURE"
    }
  }

  "PICTURE" {
    ACTIONS {
      "COMMENT_PICTURE" ("CP_CONTENT" = "
    CONTENT1"
         => {"PICTURE_CONFIRM_COMMENT","
    PICTURE"})
       -> "PICTURE_CONFIRM_COMMENT",
    }
  }
  "PICTURE_CONFIRM_COMMENT" {
    ACTIONS {
      "CONFIRM_COMMENT" -> "PICTURE"
    }
  }
}
```

Fig. 7. DASTML instantiation for the WackoPicko application

3.4 Test Generation

The *test generation* activity, which aims to produce test cases from both the behavioral model and the test purpose, is composed of three phases.

The first phase aims to transform the model and the test purposes into elements computable by the test case generator *CertifyIt*. Notably, test purposes are transformed into *test targets*, which a sequence of *intermediate objectives* used by the test generation engine [7]. The sequence of steps of a test purpose is mapped to a sequence of intermediate objectives of a test target. Furthermore, this first phase unfolds the combination of values between iterators of test purposes, such that one test purpose produces as many test targets as possible combinations.

The second phase consists to automatically derive *abstract test cases* by computing the test targets on the behavioral model. This phase is computed by the test case generator *CertifyIt*. An abstract test case is a sequence of completely valuated *operation calls* (i.e. all parameters are instantiated). An operation call represents either a stimulation or an observation of the application under test. Each test target automatically produces one test case verifying both the sequence of intermediate objectives and the model constraints. Note that an intermediate objective (i.e. a test purpose step) can be translated into several operation calls.

Finally, the third phase allows to export the abstract test cases into the execution environment. Within PMVT approach, this consists of (*i*) automatically creating a JUnit test suite, in which each abstract test case is exported as a JUnit test case, and (*ii*) automatically creating an interface, which defines the prototype of each operation of the application under test. The implementation of these operations, which aims at linking abstract keywords/operations to concrete actions, is in charge of the test automation engineer (see next subsection).

Figure 8 presents an abstract test case for the WackoPicko example, generated from the multi-step XSS attack test purpose introduced in Fig. 5, and the test model derived from the DASTML instantiation presented in Fig. 7.

```
1    sut.goToLogin()
2    sut.login(LOGIN_1,PWD_1)
3    was.finalizeAction()
4    sut.checkPage() = HOME_LOGGED_IN
5    sut.goToRecentPictures()
6    sut.checkPage() = RECENT_PICTURES
7    sut.goToPicture(PICTURE_ID_1)
8    sut.checkPage() = PICTURE
9    sut.submitComment(P_COMMENT_CONTENT_1)
10   threat.injectXSS(PICTURE_COMMENT)
11   was.finalizeAction()
12   sut.checkXSS()
13   sut.checkPage() = PICTURE_COMMENT_PREVIEW
14   sut.validateComment())
15   sut.checkXSS()
16   sut.checkPage(PICTURE)
```

Fig. 8. Abstract test case for the WackoPicko application

Basically, this test case consists to (*i*) log on the application using valid credentials (steps #1, #2, #3 and #4), (*ii*) browse the application to a picture (steps #5, #6, #7 and #8), (*iii*) submit a comment with an attack vector on a given user input (steps #9, #10, #11, #12 and #13), (*iv*) browse to a page echoing the injected data and check if there exists an application-level flaw (steps #14, #15 and #16), using the *checkXSS()* observation that allows to assign a verdict to the test case.

3.5 Adaptation and Test Execution

During the modeling activity, all data used by the application (pages, user input, attack vector, etc.) are modeled at an abstract level. As a consequence, the test cases are abstract and cannot thus be executed as they are. The gap between stimuli (resp. keywords) of the abstract test cases and the concrete API (resp. data) of the application under test must be bridged. To achieve that, the test case generator *CertifyIt* generates a file containing the signature of each operation. The test automation engineer is in charge of implementing each operation of this interface. Since Web applications become richer and richer (notably due to more and more complex client-side behaviors), actions and observations of the application are executed on the client-side Web GUI by using the HtmlUnit framework[13]. The different attack vector variants are extracted from the OWASP XSS Filter evasion cheat sheet[14], which provides about one hundred variants for XSS detection. This way, our approach only focuses on producing abstract vulnerability test cases, and each one is concretized and automatically executed with each possible attack vector variant.

Finally, regarding the test verdict assignment, we introduce the following terminology: *Attack-pass* when the complete execution of a test reveals that the application contains a breach, *Attack-fail* when the failure of the execution of the last step reveals that the application is robust to the attack.

[13] `http://htmlunit.sourceforge.net/` [Last visited: July 2014]

[14] `https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet` [Last visited: July 2014]

4 Experimental Results on Real-Life Applications

The execution of the hundred test cases generated for the vulnerable WackoPicko example (derived from the abstract test case introduced in Fig. 8 and using the hundred OWASP XSS attack vector variants), has shown that 80% of the executed test cases are attack-pass. The remaining 20% have been run with variants designed to unveil a particular XSS vulnerability, for which WackoPicko example is not sensitive. Hence, these results fit the manual experiments we conducted on WackoPicko and gave a first validation of our approach, i.e. our approach is suitable for effective detection of multi-step XSS vulnerabilities on early 2000's simple Web applications. However, to complete and confirm these first results, further experimentations have been conducted to do comparison with other techniques like vulnerability scanners and penetration testing.

4.1 Overview of the *stud-e* Web Application

We notably applied the PMVT approach on a real-life Web application, named *stud-e*, and conducted this industrial use case in partnership with the development team. This case study is an e-learning Web-based application that is currently used by more than fifteen thousands users per year in France. It provides three profiles: *students*, *teachers*, and *administrators*. Students can access and download material of their courses, practice quizzes and exercises, participate to their exams and review their scores, interact with their teachers through embedded emails and forums. Teachers can grant course material, elaborate quiz and exercises, manage their courses, group courses into modules, define exams, give scores to exams, tutor their students. Administrators are in charge of student registrations, teacher management, privilege definition and parameter settings. This application uses *infinite-urls*, meaning that every page is accessed through a unique timestamped identifier. It also uses a custom url-rewriting mechanism. A lot of effort has been put into security-related matters: all non-user data are encrypted (e.g., session data, database keys, etc.), data retrieved from the database are sanitized, and user input validation occurs both at client-side and server-side. That is why *Stud-e* is representative of an important class regrouping sensitive Web applications (e.g., banking area) that emphasizes security protection and encryption, which is a struggle for current vulnerability detection techniques.

Because the application uses infinite-urls, exhaustive testing of the Website based on its url is impossible. Hence, we applied a risk assessment approach to identify potential threats, business assets, vulnerabilities, and attack scenarios. These pieces of information were gathered while interviewing real users about the attack scenarios they feared the most. As a result, two possible attack scenarios arise. The first attack scenario focuses on a *dishonest teacher* who wants to steel/suppress educational material, for its own needs or for revenge. In this first case, the threat is a dishonest teacher, the targeted business assets are the educational materials. The second scenario focuses on a *dishonest student* who wants to cheat by influencing its scores. In this second case, the threat is a dishonest student, the targeted business assets are the exams and related scores.

Both attack scenarios are particularly complex. For instance, the *dishonest teacher* scenario involves browsing 9 pages and performing 38 user actions (clicks, field filling, etc.), and features a great number of intermediate pages and actions. In this scenario, the test engineer has to browse 6 pages and perform 8 user actions between the injection page and the observation page. These features make the detection of multi-step XSS vulnerabilities very hard for current techniques.

4.2 Experimental Results

We were able to successfully apply the PMVT approach to *stud-e* despite all its security features. It took approximately 3 hours to produce the DASTML instantiation of the Web application including pages, actions and data. The test generator computed the 14 expected vulnerability test cases in 15 minutes. Five (5) more hours were spent for adaptation activity to write the HtmlUnit implementation of the operations corresponding to the abstract actions. Finally, it took about 2 seconds to execute one test with a particular variant. Hence, it requires approximately 50 minutes, to execute the entire set of tests (2s $\times$ 14 tests $\times$ 106 variants = 2968 s).

Two vulnerabilities has been discovered. The first one was introduced for the sake of this study, whereas the second concerned a unintended multi-step XSS vulnerability. Tests executions did not produce any false positive, thanks to the short risk assessment phase and the precision of the test targets. They did not produce any false negative either, even though 20% of executions were marked as *attack-fail*: it means that *stud-e* is robust to the variants used in each *attack-failed* test execution. It takes the entire variants list to assess the presence of a certain vulnerability. Compared to the two identified attack scenarios, the sequence embedding the second multi-step XSS vulnerability was both shorter and simpler (only one profile was involved). This discovery led to a update of the source code of the Web application. Notice that this discovery was due to the systematic identification of user input fields and their respecting echoing page, which produces test cases with many relevant checks all along the test case.

4.3 Comparison Studies

To do comparison with other approaches, we conducted two vulnerability detection campaigns on the *stud-e* application: one using Web Application Scanners (WAS), one following a penetration testing protocol. Experiments with five WASs (IBM AppScan, NTOSpider, w3af, skipfish, and arachni) showed that these tools are not suitable for this kind of Web application. Most of them (w3af, skipfish, arachni and NTOSpider) were not able to authenticate to the application.

The protection mechanism of *stud-e*, which we described earlier, constitutes a solid barrier for scanners since their modus operandi relies on storing all found URL to fuzz each of them without respect of logical workflow (aside from the authentication process).

An additional protection mechanism, which makes *stud-e* almost impossible to crawl, is the use of a frame set. If the request does not originate from the frame that contains the link or the form responsible for the request, the server refuses the request and the user gets thrown back to the authentication page. Hence, only IBM AppScan was able to get past through the authentication page and access the authenticated area. However, we had to define a "multi-step operation sequence" in order to reach an injection page. No XSS vulnerability has been found during the scan.

Experiments with penetration testing were not straightforward. The use of tools (like intrusive proxies) demonstrates to be inefficient, mainly for two reasons. Firstly, none of these tools are able to replay a full test sequence. Their replay feature only allows to replay one HTTP request to the server, and this is not relevant for the purpose of detecting multi-step XSS vulnerabilities. Secondly, they work at the HTTP level, which is not suited for *stud-e*. Indeed, this application embeds a protection mechanism, which makes the crafting of relevant HTTP requests very difficult. Each request to the server embeds control parameters, dynamically generated on each page. Without the knowledge of the Javascript code behavior and the knowledge of the control parameter, crafting a correct HTTP request is merely impossible.

Hence, after failing at using intrusive proxies, we finally execute the tests by hand. For the *dishonest student* attack scenario, it took approximatively 1 minute to execute the entire scenario. Knowing that this scenario has three tests, and that each test must be executed 106 times (because of the 106 attack vector variants), the total execution time required to test the scenario is approximately 5 hours (1 min $\times$ 3 tests $\times$ 106 variants = 318 min). For the *dishonest teacher* attack scenario, it took approximatively one minute and a half to execute the entire scenario. Knowing that this scenario has 11 tests, and that each test must also be executed 106 times, the total execution time required to test the scenario is approximatively 29 hours (1.5 min $\times$ 11 tests $\times$ 106 variants = 1749 min to compute all the execution configurations).

4.4 Experimentation Summary

After a short risk assessment phase that lead to identify two threat scenarios, the PMVT approach has been experimented with a focus on them. It successfully detected 2 multi-step XSS vulnerabilities (now corrected by the development team) on a large and real-life Web application. We spent 10 hours to deploy the whole process. In comparison with a manual penetration testing approach, we have shown the efficiently of PMVT, which makes it possible to save about 19 hours in regard to manual testing attempt. The experiments using 5 Web Application Scanners have also shown that, due to specific characteristics of the *stud-e* application (defensive programming), no scanner succeeded to find any of the vulnerabilities. To conclude, these encouraging experimental results enable to successfully validate the relevance and efficiency of our approach.

5 Related Work

Due to the prevalence of XSS vulnerabilities, many research directions are investigated to prevent XSS exploits or to detect XSS flaws.

Examples of *prevention* are defense mechanisms installed on the server (Web application firewalls for instance) and/or on the client's browser that examines incoming data and sanitizes anything considered malicious. Lots of solutions have been elaborated to protect against XSS, based on Web proxies [16], reversed proxies [30], dynamic learning [4], data tainting [21], fast randomization technique [1], data/code separation [12], or pattern-based HTTP request/response analysis [19]. XSS prevention is efficient against multi-step XSS vulnerabilities because it is enough to scan user inputs to spot malicious vectors. But it comes with another challenge, which is the capacity of identifying script code as being malicious. Again, it takes some knowledge of the application's behavior to separate harmless scripts sent by the server from malicious scripts injected by miscreants. In addition, it does not solve the main problem of developers who are unaware of the severity of XSS and good practices that help enforcing security. Worse, it might invite them to solely rely on third party security tools like Web Application Firewalls and foster poor-secured Web applications to proliferate.

Contrary to prevention approach, *detection* is an offensive strategy. It is a testing activity consisting of impersonating a hacker and performing attack scenarios using manual, tool-based (intrusive proxies, ...) or automated techniques (Web Application Vulnerability Scanners, ...), in a harmless way (without compromising the application or the server where it is hosted). Usually, XSS detection is done post-development, by a third-party security organization. It can also be done prior to the application's deployment, and therefore may be seen as an acceptance test criterion. Related work on XSS detection can be classified into two categories: static analysis security testing (SAST), or dynamic analysis security testing (DAST). The first category encompasses the use of code-based techniques while the second category consists of executing and stimulating the system in order to detect vulnerabilities.

A majority of the techniques found in the literature propose to deal with XSS using SAST techniques. Kieyzun et al. [15] propose a vulnerability detection technique addressing SQL injections (SQLi) and XSS-1 (reflected) as well as XSS-2 (stored) vulnerabilities, based on dynamic taint analysis. Wassermann and Su [27] use string-taint analysis for cross-site scripting vulnerabilities detection. This technique combines the concepts of tainted information flow and string analysis. Shar et al. [23] present an automated approach that not only detects XSS vulnerabilities using a combination of the concepts of tainted information flow and string analysis, but also statically removes them from program source code. The same authors designed another approach [24] that aims to build SQLi and XSS detectors by statically collecting predefined input sanitization code attributes. All these approaches are SAST techniques, meaning that program source code has to be disclosed one way or another. The underlying concept behind each is taint analysis [22] which consists of keeping track of the values derived from user inputs throughout the application internals.

Although code analysis appears quite effective for detecting multi-step XSS, a major problem is that program source code is not always available. Moreover, these techniques are bound to a specific programming language, while there exists a tremendous number of languages to develop a Web application (PHP, .NET, JSP, Ruby, J2E, and so on). Hence, several dynamic application security testing (DAST) techniques have been proposed regarding the detection of vulnerabilities such as XSS.

In [17], Korscheck proposes a workflow-based approach to deal with multi-step XSS vulnerabilities, by using manually recorded traces to model a Web application and then injecting malicious data by replaying the traces. User traces reduce the test design cost while still carrying enough information to handles logical barriers, but it hardly handles the Web application evolution.

In [14], the authors present a multi-agent black-box technique to detect stored-XSS vulnerabilities in Web forms. It is composed of a Web page agent parser (i.e. a crawler), a script injection agent to perform the attacks, and a verification agent to assign a verdict. This approach solely relies on an automatic Web crawler, which may miss consequent parts of the Web application, and therefore miss potentially vulnerable injection points.

Blome et al. [5] propose a model-based vulnerability testing that relies on attacker models, which can be seen as an extension of Mealy machines. The approach is based on a list of nominal and attack vectors, a configuration file that contains system-specific information, and an XML file, describing the attacker model. This approach addresses lots of vulnerabilities but multi-step XSS is not addressed. It would imply to model a complex heuristic to inject and observe this particular vulnerability type. Also, attacker models are specific to one Web application, and it requires great effort from test engineers to design these artifacts for every test campaign.

The approach presented in [13] consists of modeling the attacker behavior. It also requires a state-aware model of the application under test, annotated using input taint data-flow analysis, to spot possible reflections. Concrete application inputs are generated with respect to an Attack Input Grammar, which produces fuzzed values for reflected application input parameters. This technique tackles multi-step XSS detection. However, it requires a great effort from test engineers to deploy the approach: the model inference process needs to be rightly tuned. Also, it cannot handle client-side oriented applications (using Ajax).

Buchler et al. [9] formalize the application under test using a secure ASLan++ model, where all traces fulfill the specified security properties. The goal is to apply a fault injection operator to the model, and use a model checker to report any violated security goal. For each violated security goal corresponds an abstract attack trace which is concretized semi-automatically using a pivot language. This approach has been able to find reflected XSS vulnerabilities, but has not been used to discover multi-step XSS. Also, having test engineers provide a formalized representation of a Web application is something we consider highly handicapping.

6 Conclusion and Future Works

This paper introduced an original approach, called Pattern-driven and Model-based Vulnerability Testing (PMVT), for the detection of Web application vulnerabilities. This approach is based on generic test patterns (i.e. independent from the Web application under test) and a behavioral models of the application under test. The behavioral model describes the functional and behavioral aspects of the Web application. The generic test patterns define abstract vulnerability scenarios that drive the test generation process. The proposed approach thus consists of instantiating the abstract scenarios on the behavioral model in order to automatically generate test cases, which target the vulnerability described in the initial test pattern. To experiment and evaluate the PMVT approach, a full automated toolchain, from modeling to test execution, has been developed and experimented, using real-life Web applications, to detect multi-step cross-site scripting vulnerabilities, which are nowadays one of the most critical and widespread Web application attacks.

A thorough experimentation on a real-life e-learning Web application has been conducted to validate the approach, and a comparison with existing automated testing solution, such as vulnerability scanners, has shown its effectiveness to generate more accurate vulnerability test cases and to avoid the generation of false positive and false negative results. These benefits directly stem from the combination of the behavioral model, capturing the logical aspects of the application under test, and the test patterns, driving with precision the test generation process. Moreover, the automation of the test generation and test execution makes it possible to adopt an iterative testing approach and is particularly efficient to manage security regression tests on updated or corrected further versions of the application under test.

Besides these research results, the experiments showed possible improvements of the method and the toolchain. The main drawback of our approach echoes the one of traditional MBT process. Indeed, although we reached a first level of simplification using the dedicated DASTML Domain Specific Modeling Language, the needed effort to design the model is still high. We are working on to integrate another simplification level by using user traces (as proposed in [17]) to infer the model: users would browse a Web Application and record their actions, then an algorithm would translate the results into a DASTML instantiation. This improvement may also automate the adaptation of the generated abstract test cases since the user traces could naturally provide the link between the abstract stimuli/data of the model and the corresponding concrete ones. We are also investigating the extension of the approach in order to address more vulnerability classes, both technical (such as cross-site request forgery, file disclosure and file injection) and logical (such as integrity of data over applications business processes). This extension requires to define generic test patterns ensuring the automated coverage of these vulnerabilities. Finally, another research direction aims at experimenting and extending the current approach to address Web applications on mobile devices.

Acknowledgment

This work is supported by the French FSN project DAST (see the project Website at `dast.univ-fcomte.fr` [Last visited: July 2014]).

References

1. Athanasopoulos, E., Pappas, V., Krithinakis, A., Ligouras, S., Markatos, E.P., Karagiannis, T.: xJS: practical XSS prevention for web application development. In: Proc. of the USENIX conference on Web application development (WebApps'10). pp. 147–158. USENIX Association, Boston, MA, USA (June 2010)
2. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the Art: Automated Black-Box Web Application Vulnerability Testing. In: Proc. of the 31^{st} Int. Symp. on Security and Privacy (SP'10). pp. 332–345. IEEE CS, Oakland, USA (May 2010)
3. Bernard, E., Bouquet, F., Charbonnier, A., Legeard, B., Peureux, F., Utting, M., Torreborre, E.: Model-based testing from UML models. In: Proc. of the Int. Workshop on Model-Based Testing (MBT'06). LNCS, vol. 94, pp. 223–230. Springer, Dresden, Germany (October 2006)
4. Bisht, P., Venkatakrishnan, V.: XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 23–43. Springer (2008)
5. Blome, A., Ochoa, M., Li, K., Peroli, M., Dashti, M.: Vera: A flexible model-based vulnerability testing tool. In: 6^{th} Int. Conference on Software Testing, Verification and Validation (ICST'13). pp. 471–478. IEEE CS, Luxembourg (March 2013)
6. Botella, J., Bouquet, F., Capuron, J.F., Lebeau, F., Legeard, B., Schadle, F.: Model-Based Testing of Cryptographic Components – Lessons Learned from Experience. In: Proc. of the 6^{th} Int. Conference on Software Testing, Verification and Validation (ICST'13). pp. 192–201. IEEE CS, Luxembourg (March 2013)
7. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F.: A test generation solution to automate software testing. In: Proc. of the 3^{rd} Int. Workshop on Automation of Software Test (AST'08). pp. 45–48. ACM Press, Leipzig, Germany (May 2008)
8. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: A subset of precise UML for model-based testing. In: Proc. of the 3^{rd} Int. Workshop on Advances in Model-Based Testing (AMOST'07). pp. 95–104. ACM Press, London, UK (July 2007)
9. Buchler, M., Oudinet, J., Pretschner, A.: Semi-Automatic Security Testing of Web Applications from a Secure Model. In: 6^{th} Int. Conference on Software Security and Reliability (SERE'12). pp. 253–262. IEEE, Gaithersburg, MD, USA (June 2012)
10. Doupé, A., Cova, M., Vigna, G.: Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Proc. of the 7^{th} Int. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10). pp. 111–131. Springer, Bonn, Germany (July 2010)
11. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the State: A State-aware Black-box Web Vulnerability Scanner. In: Proc. of the 21^{st} USENIX Conference on Security Symposium (Security'12). pp. 523–537. USENIX Association, Bellevue, WA, USA (Aug 2012)
12. Doupé, A., Cui, W., Jakubowski, M.H., Peinado, M., Kruegel, C., Vigna, G.: deDacota: toward preventing server-side XSS via automatic code and data separation. In: Proc. of the 20^{th} ACM SIGSAC Conference on Computer and Cummunications Security (CCS'2013). pp. 1205–1216. ACM, Berlin, Germany (2013)

13. Duchene, F., Groz, R., Rawat, S., Richier, J.L.: XSS Vulnerability Detection Using Model Inference Assisted Evolutionary Fuzzing. In: Proc. of the 5^{th} Int. Conference on Software Testing, Verification and Validation (ICST'12). pp. 815–817. IEEE CS, Montreal, Canada (April 2012)
14. Gálan, E.C., Alcaide, A., Orfila, A., Alís, J.B.: A multi-agent scanner to detect stored-XSS vulnerabilities. In: 5^{th} Int. Conference for Internet Technology and Secured Transactions (ICITST'10). pp. 1–6. IEEE, London, UK (November 2010)
15. Kieżun, A., Guo, P.J., Jayaraman, K., Ernst, M.D.: Automatic creation of SQL injection and cross-site scripting attacks. In: 31^{st} Int. Conference on Software Engineering (ICSE'09). pp. 199–209. IEEE, Vancouver, Canada (May 2009)
16. Kirda, E., Jovanovic, N., Kruegel, C., Vigna, G.: Client-side cross-site scripting protection. Computers & Security 28(7), 592–604 (2009)
17. Korscheck, C.: Automatic Detection of Second-Order Cross Site Scripting Vulnerabilities. Diploma thesis, Wilhelm-Schickard-Institut für Informatik, Universität auf Tübingen (December 2010)
18. Legeard, B., Bouzy, A.: Smartesting CertifyIt: Model-Based Testing for Enterprise IT. In: Proc. of the 6^{th} Int. Conference on Software Testing, Verification and Validation (ICST'13). pp. 391–397. IEEE CS, Luxembourg (March 2013)
19. Mahapatra, R.P., Saini, R., Saini, N.: A pattern based approach to secure web applications from XSS attacks. Int. Journal of Computer Technology and Electronics Engineering (IJCTEE) 2(3) (June 2012)
20. MITRE: Common weakness enumeration. `http://cwe.mitre.org/` (Oct 2013), last visited: February 2014
21. Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., Vigna, G.: Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In: Proc. of the Network and Distributed System Security Symposium (NDSS'07). pp. 1–12. The Internet Society, San Diego, CA, USA (February 2007)
22. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. Journal on Selected Areas in Communications archive 21(1), 5–19 (September 2006)
23. Shar, L.K., Tan, H.B.K.: Automated removal of cross site scripting vulnerabilities in web applications. Information and Software Technology 54(5), 467–478 (May 2012)
24. Shar, L.K., Tan, H.B.K.: Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. Information and Software Technology 55(10), 1767–1780 (October 2013)
25. Smith, B., Williams, L.: On the Effective Use of Security Test Patterns. In: Proc. of the 6^{th} Int. Conference on Software Security and Reliability (SERE'12). pp. 108–117. IEEE CS, Washington, DC, USA (June 2012)
26. Vouffo Feudjio, A.G.: Initial Security Test Pattern Catalog. Public Deliverable D3.WP4.T1, Diamonds Project, Berlin, Germany (June 2012), `http://publica.fraunhofer.de/documents/N-212439.html` [Last visited: February 2014]
27. Wassermann, G., Su, Z.: Static detection of cross-site scripting vulnerabilities. In: Proc. of the 30^{th} Int. Conference on Software Engineering (ICSE'08). pp. 171–180. IEEE, Leipzig, Germany (May 2008)
28. Whitehat: Website security statistics report. `https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf` (October 2013), last visited: February 2014
29. Wichers, D.: Owasp top 10. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project` (Oct 2013), last visited: February 2014
30. Wurzinger, P., Platzer, C., Ludl, C., Kirda, E., Kruegel, C.: SWAP: mitigating XSS attacks using a reverse proxy. In: 5^{th} Int. Workshop on Software Engineering for Secure Systems (SESS'09). pp. 33–39. IEEE, Vancouver, Canada (May 2009)