Model-Based Vulnerability Testing for Web Applications

Franck Lebeau*, Bruno Legeard*†, Fabien Peureux* and Alexandre Vernotte*
*FEMTO-ST Institute - DISC department - UMR CNRS 6174
16, route de Gray 25030 Besançon, FRANCE
Email: {flebeau, blegeard, fpeureux, avernott}@femto-st.fr
†Smartesting R&D Center - 18, rue Alain Savary, 25000 Besançon, FRANCE
Email: bruno.legeard@smartesting.com

***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 some 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.**

Keywords*—*Vulnerability testing, Model-Based Testing, Web applications, DVWA example.

I. INTRODUCTION

The continued growth of Internet usage as well as the development of Web applications foreground the challenges of It security, particularly in terms of data confidentiality, data integrity and service availability.
Thus, as stated in the annual barometer concerns of Information Technology Managers[1], for 72% of them, computer security and data protection are their primary concerns. This growth of risk arises from the mozaic of technologies used in current Web applications (eg HTML5), which increases the risk of security breaches.
This situation has led to significant growth in application-level vulnerabilities, with thousands of vulnerabilities detected and disclosed annually in public databases such as the MITRE CVE - Common Vulnerabilities and Exposures[2]. The most common vulnerabilities found on these databases especially emphasize the lack of resistance to code injection of the kind SQL Injection or Cross-Site Scripting (XSS), which have many variants. They appear in the top list of current web applications attacks.
Application-level vulnerability testing is first performed by developers, but they often lack the sufficient in-depth knowledge in recent vulnerabilities and related exploits. This kind of tests can also be achieved by companies specialized in security testing, in pen-testing (for penetration testing) as instance. These companies monitor the constant discovery of such vulnerabilities, as well as the constant evolution of attack techniques. But they mainly use manual approaches, making the dissemination of their techniques very difficult, and the impact of this knowledge very low.

The work presented in this paper aims to automate vulnerability testing of Web applications, in order to extend the testing capability of these applications, increase the detection of such vulnerabilities and improve the overall level of security. The approach studied, based on model-based test generation techniques, aims to capture vulnerability test patterns in order to automate their implementation on Web applications.

The original contributions of this paper are:

- The consideration of the application's functional behaviour to generate vulnerability test cases, which allows for a more thorough exploration and testing of the Web application;
- The capture of Vulnerability Test Patterns as test purposes used to drive the test generation engine through models;
- The modelling activity dedicated to vulnerability testing that combines modelling of the system under test and modelling of the environment activities.

In the remainder of this paper, we first present our approach following the example of vulnerability testing of Cross-Site Scripting (XSS). We provide modelling material and test purposes in order to show how our test generation tool chain computes vulnerability test cases, that are executable on the target application. This presentation then relies on an experiment conducted on a Web application named DVWA - Damn Vulnerable Web Application - which is a demonstration application featuring typical vulnerabilities. Finally, we position this work in the state of the art and offer a conclusion and perspectives.

II. MODEL-BASED VULNERABILITY TESTING

We propose to revisit and adapt the traditional approach of Model-Based Testing (MBT) in order to generate vulnerability test cases for Web applications. This adapted approach is called Model-Based Vulnerability Testing (MBVT). In this section, we firstly describe the specificities of the MBVT approach. Secondly, we introduce the running example, named DVWA, used in the rest of the paper to illustrate the MBVT approach, and finally, we define the scope of the experiments conducted on this example.

[1]Barometer CIO 2012, survey from the *CSC* institute, the *01 Informatique* magazine and the *BFM* radio http://assets1.csc.com/fr/downloads/Barometre_CIO_2012_Tout_se_transforme_OK.pdf
[2]Web site of MITRE CVE database - http://cve.mitre.org

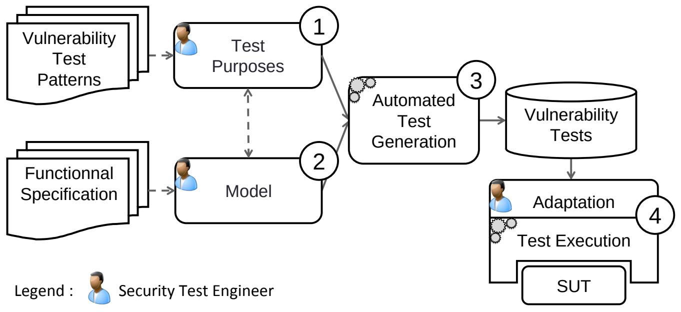

Vulnerability Test Patterns
Test Purposes
1
Automated Test Generation
3
Vulnerability Tests
Functionnal Specification
Model
2
Adaptation
4
Test Execution
SUT
Legend : Security Test Engineer

Fig. 1. Model-Based Vulnerability Test process

A. Principles of the MBVT approach

MBT [1] is an increasingly widely-used approach that has gained much interest in recent years, from academic as well as industrial domain, especially by increasing and mastering test coverage, including support for certification, and by providing the degree of automation needed for shortening the testing execution time. MBT refers to a particular approach of software testing techniques in which both test cases and expected results are automatically derived from a high-level model of the System Under Test (SUT). This high-level model, which defines the input of the MBT process, specifies the behaviours of the functions offered by the SUT, independently of how these functions have been implemented. The generated test cases from such models allow to validate the behavioural aspects of the SUT by comparing back-to-back the results observed on the SUT with those specified by the model. MBT aims thus to ensure that the final product conforms to the initial functional requirements. It promises higher quality and conformance to the respective functional requirements, at a reduced cost, through increased coverage (especially about stimuli combination) and increased automation of the testing process [2]. However, if this technique is used to cover the functional requirements specified in the behavioural model of the SUT, it is also limited to this scope, since what is not modeled will not be tested.

The use of MBT techniques to vulnerability testing requires to adapt both the modelling approach and the test generation computation. Within the traditional MBT process, which allows to generate functional test cases, positive test cases[3] are computed to validate the SUT in regards to its functional requirements. Within vulnerability testing approach, negative test cases[4] have to be produced: typically, attack scenarios to obtain data from the SUT in an unauthorized manner. The proposed process, to perform vulnerability testing, is depicted in Figure 1. This process is composed of the four following activities:

① the *Test Purposes* activity consists in formalizing test purposes from vulnerability test patterns that the generated test cases have to cover;

② the *Modelling* activity aims to define a model that captures the behavioural aspects of the SUT in order to generate consistent (from a functional point of view) sequences of stimuli;

③ the *Test Generation and Adaptation* activity consists in automatically producing abstract test cases from the artefacts defined during the two previous activities;

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

All these MBVT activities are supported by a dedicated toolchain, which is based on an existing MBT software named *CertifyIt* [3] provided by the company Smartesting [5].

This software is a test generator that takes as input a test model, written with a subset of UML (called UML4MBT [4]), which captures the behaviour of the SUT. More concretely, 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 list the concrete objects used to compute test cases and to define the initial state of the SUT, and (iii) state diagrams (annotated with OCL constraints) to specify the dynamic view of the SUT.

Such UML4MBT models have a precise and unambiguous meaning, so that the of those models can be understood and computed by the *CertifyIt* technology. OCL expressions indeed provide the expected level of formalization necessary for MBT modelling. This precise meaning makes it possible to simulate the execution of the models and to automatically generate test cases by applying predefined coverage strategies. Each generated test case is typically an abstract sequence of high-level actions from the UML4MBT models. These generated test sequences contain the sequence of stimuli to be executed, but also the expected results (to perform the observation activity), obtained by resolving the associated OCL constraints.

Section III describes, in a detail manner, each of these activities and illustrate them using a running example, which is introduced in the next subsection.

B. Presentation of the DVWA example

In order to evaluate the effectiveness and efficiency of our approach, we have applied it on a Web application called DVWA[6]. The objective of this open-source Web application test bed, based on PHP/MySQL, is to provide an aid for security professionals, web developers and teachers/students to learn, improve, and test their skills in vulnerability discovery. It can also be used to test web security testing tools, like vulnerability scanners for instance.

[3]We call ”positive test” a test case that checks whether a sequence of stimuli produces the expected effects with regards to the specifications. When a positive test is in success, it demonstrates that the tested scenario is implemented correctly.

[4]By ”negative test case” we define a test case targeting an unexpected use of the SUT. When a negative test case succeeds, it highlights a problem in the SUT.

[5]http://www.smartesting.com

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

DVWA embeds several vulnerabilities, notably vulnerabilities of the kind SQL Injection and Blind SQL Injection, and Reflected and Stored XSS. These vulnerabilities are commonly used to attack current Web applications [7].
Each vulnerability has a dedicated menu item leading to a dedicated page. DVWA also embeds three security levels: low, medium, and high. Each level carries different security protections: the lowest level has no protection at all, the medium level is a refined version but is still quite vulnerable, and the highest level is a properly secured version. Users can choose which level they want to work with by specifying it through the application. It is also possible to view and compare the source code of each security level.

C. Perimeter and objectives of experimentations

Our approach allows testing an application among the four types of attack mentioned earlier (Blind and Not Blind SQL Injection, Reflected/Stored XSS). To ease the understanding, we focus this presentation on the Reflected XSS attack (RXSS). It is one of the major breach because it is highly used and because its exploitation leads to severe risks (such as identity spoofing).
An RXSS attack targets end-users. This kind of attack happens when a user input field is used by the server to produce a response sent back to a end-user. A pirate injects malicious data (such as a script, typically written in JavaScript, which will be executed by an end-user browser) in the Web application through a user input field. Lack of user input validations leads to unsecured applications. An XSS attack is either Reflected (the response containing malicious data is immediately produced and sent back to the end-user) or Stored (the malicious data is saved in the application's database, and retrieved later, in another context). Our presentation deals with vulnerabilities of the kind Reflected XSS.
For each security level, our objective is to apply our MBVT approach in order to compute and execute test cases allowing discovery of RXSS vulnerabilities.

III. Details of our approach

In this section, we detail each main activity of the MBVT process. For each activity, we present its objectives as well as its behaviours, and we use the DVWA example to illustrate our statements.

A. Formalizing Vulnerability Test Patterns into Test Purposes

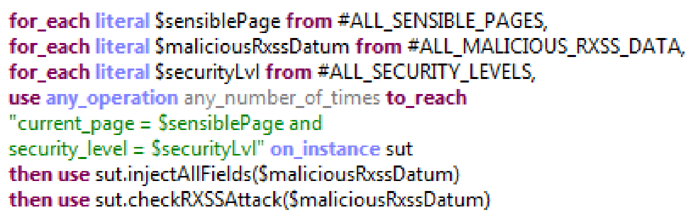
Vulnerability Test Patterns (vTP) are the initial artefacts of our approach. A vTP expresses the testing needs and procedures allowing the identification of a particular breach in a Web application. There is as much vTP as types of application-level breaches. The ITEA2 Diamonds research project has already studied vTP, and provide a first definition as well as a first listing of vTP [8]. The characteristics of a Vulnerability Test Pattern are: its *name*, its *description*, its *testing objectives* (precising the addressed testing objectives), its *prerequisites* (precising the conditions and knowledges required for a right execution), its *procedure* (precising its *modus operandi*), its *observations* and its *oracle* (precising which information have to be prone in order to identify the presence of an application-level breach), its *variants* (precising some alternatives regarding the means in use, or the malicious data, or what is observed), its *known issues* (precising any limitation or problem (eg., technical) limiting its usage), its *affiliated vTP* (listing its correlated vTP), its *references* (to public resources dealing with application-level vulnerability issues, such as CVE, CWE, OWASP, CAPEC, ...). Figure 2 presents the Vulnerability Test Pattern of Reflected XSS attack.

Name	Reflected XSS
Description	This pattern can be used on an application which doesn't checks user inputs. A Reflected XSS attack can redirect users to a malicious site, or can steal users' private information (cookies, session, ...).
Objective(s)	Detect if a user input can embed malicious datum enabling a Reflected XSS attack.
Prerequisites	N/A
Procedure	Identify a sensible user input field, inject the malicious datum *<script>alert(rxss)</script>*.
Observation & Oracle	Check if a message box 'rxss' appears.
Variant(s)	- malicious data variants: character encoding, Hex-transformation, comments insertion - procedure variants: attack can be applied at the HTTP level; in this case, malicious data are injected in the parameters of the HTTP messages send to the server, and we have to check if the malicious data is in the response message coming from the server
Known Issue(s)	Web Application Firewalls (WAF) filter messages send to the server (black list, clac regEx, ...); variants allows to overcomes 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. 2. vTP of Reflected XSS attacks

For this vTP, variants of malicious data are defined during the modelling activity, variants of the procedure are defined during the adaptation and execution activity. The initial procedure is defined in a test purpose. A *test purpose* is a high level expression that formalizes a test intention linked to a testing objective to drive the automated test generation on the behavioural model. In the MBVT context, we propose to use test purposes to formalize vTP. We propose test purposes as a mean to drive the automated test generation.
The test purpose language we are presenting is called *Smartesting Test Purpose Language* [5]. This is a textual language based on regular expressions [6], allowing the formalization of vulnerability test intention in terms of states to be reach and operations to be called. The language relies on combining keywords, to produce expressions that are both powerful and easy to read.
Basically, a test purpose is a sequence of important *stages* to reach (last three lines in Figure 3). A stage is a set of operations or behaviour to use, or/and a state to reach. Transforming the sequence of stages into a complete test case, based on the model behaviour and constraints, is left to the MBT technology (more details comes in the section III-C). Furthermore, at the beginning of a test purpose, the test engineer can define *iterators* (three first lines in Figure 3). Iterators are used in stages, in order to introduce context variations. Each combination of possible values of iterators produces a specific test case.

[7]These vulnerabilities appears in the top three most frequently used types of attack listed by the OWASP project - Open Web Application Security Project - cf www.owasp.org.
[8]http://www.itea2-diamonds.org



```
for_each literal $sensiblePage from #ALL_SENSIBLE_PAGES,
for_each literal $maliciousRxssDatum from #ALL_MALICIOUS_RXSS_DATA,
for_each literal $securityLvl from #ALL_SECURITY_LEVELS,
use any_operation any_number_of_times to_reach
"current_page = $sensiblePage and
security_level = $securityLvl" on_instance sut
then use sut.injectAllFields($maliciousRxssDatum)
then use sut.checkRXSSAttack($maliciousRxssDatum)
```

Fig. 3. test Purpose formalizing the vTP of Figure 2 on DVWA

Example (DVWA): Figure 3 shows the test purpose formalizing the vTP of Figure 2, applied on the DVWA case study. This schema precises that for all sensible web pages, for all malicious data enabling the detection of RXSS breach, and for all security levels of DVWA, it is required to do the followings: (i) use any operation to activate the sensible page with the required security level, (ii) inject the malicious data in all the user inputs of the page, (iii) check if the page is sensible to the RXSS attack. The three keywords *ALL_** are enumerations of values, defined by the security test engineer, allowing him/her to master the final amount of test cases.

We use a second test purpose, similar to the presented one, enabling to precisely target which user input fields have to be injected. This test purpose gives ways of control to the security engineer. Modifications are: (i) one added iterator targeting sensible fields, and (ii) the use of the operation *injectField* instead of the operation *injectAllFields*.

B. Modelling

The modelling activity produces a *model* based, on one hand, on the functional specifications of the application, and on the other hand on the test purposes which will be applied on it (keywords used in test purposes have to be modelled). We present in the following the used UML diagrams (classes, objects, state diagrams), and their respective usages in the context of our MBVT approach.

Class diagrams specify the static aspect of the model, by defining the abstract objects of the SUT. Class diagrams of our approach share many similarities with traditional MBT. *Classes* model business objects (notably, the *SUT* class models the system under test, and defines the points of control and observation). *Associations* model relations between business objects. *Enumerations* model sets of abstract values, and *literals* model each value. *Class attributes* model evolving characteristics of business objects. *Class operations* model points of control and observation of the SUT (we found here navigations between pages). Nevertheless, our MBVT approach differs from the traditional MBT by (see Figure 4):

- two additional classes (*page* and *field*) and their relations, which respectively model the general structure of the application and the user input fields potentially used to inject malicious data;
- some additional operations, coming from the test purposes, which model means to exercise and observe the attack
- one additional enumeration which model malicious data injected in user input fields.

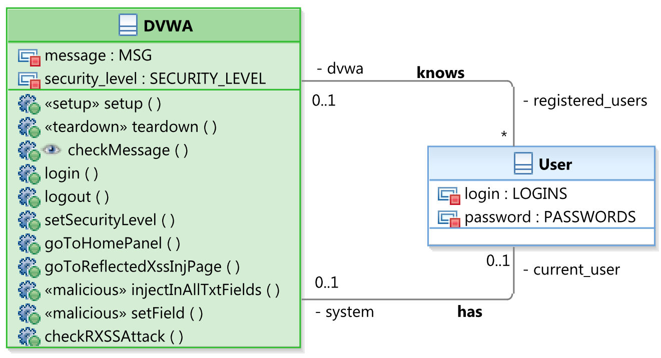


Fig. 5. Class diagram of DVWA

The UML state diagram graphically specifies the behavioural aspect of the SUT, modelling the navigations between pages of the Web application. *States* model Web pages, and *transitions* model the available navigations between these Web pages. *Triggers* of transitions are the UML operations of the SUT class. *Guards* of transitions (specified in OCL) precisely define the context of firing. *Effects* of transitions (also specified in OCL) precisely define the modifications induced by the execution of transitions.

The UML object diagram models the initial state of the SUT, by instantiating class diagram elements. Thus, *instances* model business entities available at the initial state of the SUT, and *links* model relations between these entities. In our MBVT approach, the object diagram model the Web pages of the application and the user input fields of these pages.

Example (DVWA): Figure 5 presents the class model of the DVWA example (*Page* and *Field* classes are not shown).

Note that: (i) the additional class *User* models the potential users of the application; (ii) class attributes *message* and *security_level* respectively model application feedbacks and security level; (iii) the five first operations model the necessary and sufficient functional subset of the application allowing the access to the tested pages with the relevant level of security; (iv) operations *injectAllFields* and *injectField*, which are keywords coming from test purposes, model the injection of malicious data on all or part of the user input fields of Web pages; (v) operation *checkMessage* models the observation of the *message* attribute; operation *checkRXSSAttack* models the observation of the attack, and serves as oracle.

Moreover, regarding the static aspect of the model, some enumeration literals model malicious data variants: *RXSS_DUMMY* is a basic variant, *RXSS_COOKIE1* and *RXSS_COOKIE2* are two variants allowing to retrieve private user information, and *RXSS_WAF_EVASION* models a variant allowing to bypass some web application firewall techniques (see section III-D for their translation into concrete values).

Figure 6 presents the state diagram, which models the behaviour of DVWA. It defines precedences between pages: identification is required before reaching any other page of the application. Finally, Figure 7 presents the initial state of the DVWA model. It specifies: (i) one user, with its credentials, and (ii) the pages and user input fields of DVWA.

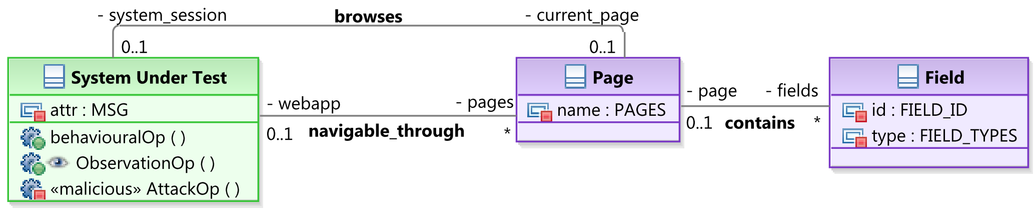


Fig. 4. Class diagram of the SUT structure, for our MBVT approach

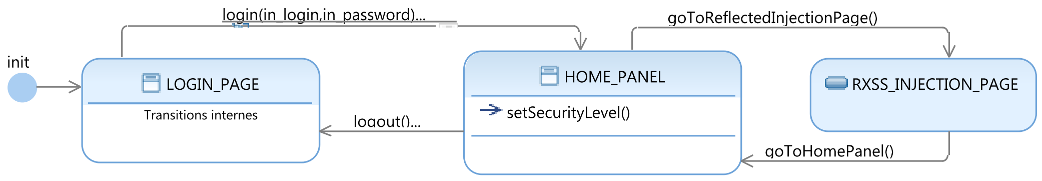


Fig. 6. State diagram of DVWA example

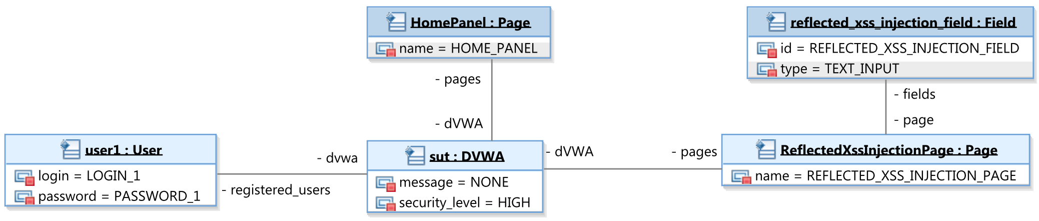


Fig. 7. Object diagram of DVWA

C. *Test generation*

The main purpose of the *test generation* activity is to produce test cases from both the model and the test purposes. Three phases compose this activity. The first phase transforms the model and the test purposes into elements usable by the *Smartesting CertifyIt* MBT tool. Notably, test purposes are transformed into *test targets*, a test target being a sequence of *intermediate objectives* used by the symbolic generator. Hence, the sequence of stages of a test purpose is mapped to a sequence of intermediate objectives of a test target. Furthermore, this first phase manages the combination of values between iterators of test purposes, such that one test purpose produces as many test targets as possible combinations of iterators values.

The second phase produces the *abstract test cases* from the test targets. This phase is left to the test case generator. An abstract test case is a sequence of *steps*, where a step corresponds to a completely valued operation call. An operation call represents either a stimulation or an observation of the SUT. Each test target produces one test case (i) verifying the sequence of intermediate objectives and (ii) verifying the model constraints. Note that an intermediate objective (and hence, a test purpose stage) can be transformed into several steps. Figure 8 presents a test cases obtained from the test purpose of Figure 3. The five first steps of this test case correspond to the first stage of the test purpose.

Finally, the third phase exports the abstract test cases into the execution environment. In our case, it consist on (i) creating a JUnit test suite, where each abstract test case is exported as a JUnit test case, and (ii) creating an interface. This interface defines the prototype of each operation of the SUT. The implementation of these operations is in charge of the test automation engineer.

Example (DVWA): We are using two test purposes and have defined four malicious data, in order to test one page with one user input field. Each test purpose produces 12 test targets, where each test target produces exactly one abstract test case, for a total amount of 24 abstract test cases.
Figure 8 presents one of the generated abstract test cases. It has to be interpreted this way: (i) it logs in the application with valid credentials; (ii) it sets the security level; (iii) it loads the targeted Web page; (iv) it verifies the correct execution of the functional part of the test case (using the *checkMessage* observation); (v) it injects the malicious datum; (vi) it verifies if there exists an application-level breach or not (using the *checkRXSSAttack* observation). This last step assigns the verdict of the test case.
Regarding the test purpose focusing on precise user input fields, test cases only differ at step #6, where the *injectField* operation replaces the *injectAllFields* operation.

```
sut.login(LOGIN_1,PASSWORD_1)
sut.checkMessage() = WELCOME
sut.setSecurityLevel(MEDIUM)
sut.goToReflectedXssInjPage()
sut.checkMessage() = REFLECTED_XSS_MESSAGE
sut.injectAllFields(RXSS_DUMMY)
sut.checkRXSSAttack(RXSS_DUMMY)
```

Fig. 8. Abstract test case example

D. Adaptation and test execution

During the modelling activity, each page, user input field, malicious datum, user credentials, etc. ... in summary, all data used by the application, are modelled in a abstract way. Hence, the test suite can't be executed as it is. The gap between abstract keywords used in abstract test cases and the real API of the SUT must be filled. To ease the understanding of our approach, we only present an adaptation of the kind 1↔1, but tables with multiple values are also used for a mapping of the kind 1↔*.
Stimuli must also be adapted. When exporting the abstract test cases, the MBT tool provides an interface defining the signature of each operation. The test automation engineer is in charge to implement the automated execution of each operation of this interface. Because we are testing Web applications, we have studied two ways of automation:

- at the *GUI* level: we stimulate and observe the application *via* the client-side GUI of the application. Even if this technique is time consuming, it could be necessary when the client-side part of the application embeds JavaScript scripts. For this technique, we use the Selenium framework.
- at the *HTTP* level: we stimulate and observe the application *via* HTTP messages send to (and received from) the server-side application. This technique is extremely fast and can be used to bypass HTML and JavaScript limitations. For this technique, we are using the Apache HTTPClient Java library.

The last but not the least activity of the MBVT is to execute the adapted test cases in order to produce a verdict. We introduce a new terminology fitting the characteristics of a test execution:

- *Attack-pass*: the complete execution of the test reveals that the application owns a breach, unlike in MBT where a complete execution of a test indicates a valid implementation;
- *Attack-fail*: the failure of the execution of the last step reveals that the application is robust to the attack, unlike MBT where such a failure indicates a invalid implementation;
- *Inconclusive*: in certain circumstances, it is not sure that a breach is discovered (eg., due to technical issues). An abnormal event happens, but no breach has been observed.

Example (DVWA): The model defines four malicious data dedicated to Reflected XSS attacks. These values are defined in an abstract way, and must be adapted. Each of them is mapped to a concrete value:

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

Operations of the SUT can be adapted in two ways: using Selenium or HTTPClient. However, we mainly use the HTTP-based approach (HTTPClient), because this techniques dramatically saves time on DVWA, for the same results. Based on the execution of the test suite, 50% of the test cases have been identified as *Attack-pass*: the two first malicious data with a low security level, the third malicious datum with the low and medium security level, and the fourth malicious datum with the medium security level. These results fit our manual experiments on DVWA. This concordance gives a first validation of our approach with regards to the addressed subset of vulnerabilities, and the DVWA context.

IV. State of the Art

The tool landscape in web application security testing is structured in two main classes of techniques:

1) Static Application Security Testing (SAST), which are white-box approaches that include source, byte and object code scanners and static analysis techniques;
2) Dynamic Application Security Testing (DAST), which includes black-box web applications scanners, fuzzing techniques and emerging model-based security testing approaches.

In practice these techniques are complementary, addressing different types of vulnerabilities. For example, SAST techniques are good to detect buffer overflow and other badly formatted string, but not so good to detect XSS or CSRF vulnerabilities. So, in this section, we focus on DAST techniques and provide a state of the art of emerging model-based security testing techniques.

Fuzzing techniques relate to the massive injection of invalid or atypical data (for example by randomly corrupting an XML file) generally by using a randomized approach [7]. Test execution results can therefore expose various invalid behaviours such as crash effects, failing built-in code assertions or memory leaks.

Web application vulnerability scanners aim to detect vulnerabilities by injecting attack vectors. These tools generally include three main components [8]: a crawler module to follow web links and URLs in the web application in order to retrieve injection points, an injection module which analyses web pages, input points to inject attack vectors (such as code injection), and an analysis module to determine possible vulnerabilities based on the system response after attack vector injection. As shown in a recent comprehensive study [9], corroborated by research papers [10], [11] and confirmed by our own experience with tools such as IBM AppScan[9], these tools suffer from two major weaknesses that highly decrease their practical usefulness:

[9]See www.ibm.com/software/awdtools/appscan/

- **Limitations in application discovery** As black-box web vulnerability scanners ignore any request that can change the state of the web application, they miss large parts of the application. Therefore, these tools test generally a small part of the web application due to the ignorance of the application behavioural "intelligence". Due to the growing complexity of the web applications, they have trouble dealing with specific issues such as infinite web sites with random URL-based session IDs or automated form submission.
- **Generation of many false positive results** The already-mentioned benchmark shows that a common drawback of these tools is the generation of false positives at a very important rate either for Reflected XSS, SQL injection or Remote File Inclusion vulnerabilities. The reason is that these tools use brute force mechanisms to fuzz the input data in order to trigger vulnerabilities and establish a verdict by comparison to a reference execution trace. Therefore, they lack precision to assign the verdict, as they do not compute the topology of the web application to precisely know where to observe.

These strong limitations of existing web vulnerability scanners lead to the key objectives of model-based vulnerability testing techniques: better accuracy in vulnerability detection, both by better covering the application (by capturing the behavioural intelligence) and by increasing the precision of the verdict assignment.

Model-based security testing are emerging techniques aiming to leverage model-based approaches for security testing [12]. This includes:

- **Model-based test generation from security protocol, access-control or security-oriented models** Various types of models of security aspects of the system under test have been considered as input to generate security test. For example, [13] proposes a method using security protocol mutation to infer security test cases. [14] develops a model-based security test generation approach from security models in UMLSec. [15] presents a methodology to exploit a model describing a Web application at the browser level to guide a penetration tester in finding attacks based on logical vulnerabilities.
- **Model-based fuzzing** This approach applies fuzzing operator in conjunction with models; For example, [16] proposes an approach that generates invalid message sequences instead of invalid input data by applying behavioural fuzzing operators to valid message sequences in form of UML sequence diagrams.
- **Model-based test generation from weakness or attack models** In these types of approaches, test cases are generated using threat, vulnerability or attacker models, which reflects the common steps needed to perform an attack, and the required associated data. For example, [17], threats to security policies are modeled with UML sequence diagrams, allowing to extract event sequences that should not occur during the system execution.

Our approach allows to generate vulnerability tests from a model that mixes functional behavioural features of the system under test and aspects that model the possible attacks, which is modelling aspects of the environment of the system. Moreover, contrary to functional MBT, the test generation process is driven by the vulnerability test patterns, so that the behavioral model is restricted to the only elements that are needed to compute the vulnerability test cases.

V. Conclusion and future works

Web application vulnerabilities fall into two categories: Technical vulnerabilities include cross-site scripting, injection flaws and buffer overflows. Logical vulnerabilities relate to the logic of the application to get it to do things it was never intended to do. They often are the result of faulty application logic. Logical vulnerabilities are specific to the functionality of particular web applications, and, thus, they are extremely difficult to characterize and identify. For example, an important security breach have been discovered and disclosed in 2012 in the Paypal payment module of Magento[10] eCommerce framework, due to the capability to falsify the payment amount after concluding the deal [18].

This paper proposes a Model-Based Vulnerability Testing approach from a behavioral model and test patterns, which aims to address both technical and logical vulnerabilities. Technical vulnerabilities are managed by the composition of a navigational behavioral model and related test patterns; and logical vulnerabilities may be address through more complete modeling and adequate patterns. The main drawback of model-based vulnerability testing echoes the one of MBT in general: the needed effort to design models, test purposes, and adapter. We are following several research directions to reduce this effort, which consist in identifying the reusability potential of the three artifacts from one project to another: test purposes can be made generic to their affiliated vulnerability type, model parts can be made generic to a web development framework (like Magento for e-commerce solutions) and also automatically generated, at least partially, using crawling techniques, and the adapter of those model parts can also be made generic to the associated framework.

Therefore, our future work leads in two main research directions: (1) extending the method by covering more vulnerability classes, both technical (such as CSRF, file disclosure and file injection) and logical (such as the integrity of data over applications business processes). We will also study (2) how the various MBVT artifacts may be made generic and reusable from one project to another project. In this context, we will focus on eCommerce applications, and more particularly eCommerce applications build on the top of the Magento framework. Indeed, eCommerce applications built with Magento have good properties because they rely to custom development and use of add-ons, both being well-known to introduce security vulnerabilities.

Acknowledgment

This work is supported by the French FSN project DAST (dast.deptinfo-st.univ-fcomte.fr) and the European ITEA2 project (http://www.itea2-diamonds.org).

[10]http://www.magentocommerce.com/

References

[1] M. Utting and B. Legeard, *Practical Model-Based Testing - A tools approach*, Morgan and Kaufmann, Eds. San Francisco, CA, USA: Elsevier Science, 2006.

[2] A. Dias-Neto and G. Travassos, “A Picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges,” *Advances in Computers*, vol. 80, pp. 45–120, July 2010, iSSN: 0065-2458.

[3] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux, “A test generation solution to automate software testing,” in *Proceedings of the 3^{rd} Int. Workshop on Automation of Software Test (AST’08)*. Leipzig, Germany: ACM Press, May 2008, pp. 45–48.

[4] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, “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)*. London, UK: ACM Press, July 2007, pp. 95–104.

[5] J. Botella, F. Bouquet, J.-F. Capuron, F. Lebeau, B. Legeard, and F. Schadle, “Model-based Testing of Cryptographic Components Lessons Learned from Experience,” in *Int. Conf. on Software Testing, Verification and Validation (ICST’13)*. Luxembourg: IEEE CS, March 2013.

[6] J. Julliand, P.-A. Masson, and R. Tissot, “Generating security tests in addition to functional tests,” in *Proceedings of the 3^{rd} International Workshop on Automation of Software Test (AST’08)*. Leipzig, Germany: ACM Press, May 2008, pp. 41–44.

[7] A. Takanen, J. DeMott, and C. Miller, *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, Inc., 2008.

[8] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, “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)*. Oakland, CA, USA: IEEE Computer Society, May 2010, pp. 332–345.

[9] “Web site on Price and Feature Comparison of Web Application Scanners,” http://www.sectoolmarket.com/, December 2012, last access January 2013.

[10] A. Doupé, M. Cova, and G. Vigna, “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)*. Bonn, Germany: Springer, July 2010, pp. 111–131.

[11] M. Finifter and D. Wagner, “Exploring the relationship between web application development tools and security,” in *Proc. of the 2^{nd} USENIX Conference on Web Application Development (WebApps’11)*. Portland, OR, USA: USENIX Association, June 2011, pp. 99–111.

[12] I. Schieferdecker, J. Grossmann, and M. Schneider, “Model-based security testing,” in *Proceedings of the 7^{th} International Workshop on Model-Based Testing (MBT’12)*, ser. EPTCS, vol. 80. Tallinn, Estonia: Open Publishing Association, March 2012, pp. 1–12.

[13] F. Dadeau, P-C.Héam, and R. Kheddam, “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)*. Berlin, Germany: IEEE Computer Society, March 2011, pp. 240–248.

[14] J. Jürjens, “Model-based Security Testing Using UMLsec: A Case Study,” *The Journal of Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 220, no. 1, pp. 93–104, December 2008.

[15] M. Buchler, J. Oudinet, and A. Pretschner, “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)*. Gaithersburg, MD, USA: IEEE Computer Society, June 2012, pp. 253–262.

[16] M. Schneider, “Model-based behavioural fuzzing,” in *Proceedings of the 9^{th} International Workshop on Systems Testing and Validation (STV’12)*, Paris, France, October 2012, pp. 39–47.

[17] L. Wang, E. Wong, and D. Xu, “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, May 2007.

[18] “NBS System - Vulnerability in Magento’s implementation of PayPal,” http://www.nbs-system.com/wp-content/uploads/Advisory_Magento_Paypal.pdf, 2012, last access January 2013.