Risk-Based Vulnerability Testing using Security Test Patterns

Julien Botella[1], Bruno Legeard[1,2], Fabien Peureux[1,2], and Alexandre Vernotte[2]

[1] Smartesting R&D Center - 2G, Avenue des Montboucons, 25000 Besançon, France,
`{botella, legeard, peureux}@smartesting.com`
[2] Institut FEMTO-ST, UMR CNRS 6174 - Route de Gray, 25030 Besançon, France
`{blegeard, fpeureux, avernott}@femto-st.fr`

Abstract. This paper introduces an original security testing approach guided by risk assessment, by means of risk coverage, to perform and automate vulnerability testing for Web applications. This approach, called Risk-Based Vulnerability Testing, adapts Model-Based Testing techniques, which are mostly used currently to address functional features. It also extends Model-Based Vulnerability Testing techniques by driving the testing process using security test patterns selected from risk assessment results. The adaptation of such techniques for Risk-Based Vulnerability Testing defines novel features in this research domain. In this paper, we describe the principles of our approach, which is based on a mixed modeling of the System Under Test: the model used for automated test generation captures some behavioral aspects of the Web applications, but also includes vulnerability test purposes to drive the test generation process.

Keywords: Risk-Based Testing, Security test pattern, Model-Based Testing, Web application vulnerability, CORAS, SQL Injection.

1 Introduction

Based on the current state of the art on security and on all the security reports like OWASP Top Ten 2013 [1], CWE/SANS 25 [2] and WhiteHat Website Security Statistic Report 2013 [3], Web applications are the most popular targets when speaking of cyber-attacks. The fact that modern society relies on the Web a little more everyday foregrounds the challenges of IT security, particularly in terms of data privacy, data integrity and service availability.

The mosaic of technologies used in current Web applications (e.g., HTML5 and JavaScript frameworks) 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 (SQLI) or Cross-Site Scripting (XSS), which have many variants. This kind of vulnerabilities indeed appears 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 penetration testing for 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. Finally, Web application vulnerability scanners can be used to automate the detection of vulnerabilities, but since they often generate many false positive and false negative results, human investigation is also required [4, 5].

This paper proposes a Risk-Based Vulnerability Testing (RBVT) approach in order to improve the overall level of Web application security by increasing the accuracy and precision of security testing according to the risk assessment. To achieve this goal, the approach consists to drive the test generation strategy using risk metrics and relevant vulnerability test patterns, which are directly related to risk assessment process of the System Under Test (SUT). This objective also includes the development of a tool supporting this RBVT process in order to automate the detection of such vulnerabilities, and therefore to get feedback about risk assessment. Hence the main contributions of the paper relate to the proposal of a risk-based and pattern-driven approach to generate vulnerability test cases for Web applications. More precisely, they are the following:

- Techniques addressing both risk-based test identification, by means of vulnerability test patterns, and test prioritization to drive the overall testing generation process.
- The extension of a test purpose language to drive the test generation engine through models in order to cover the targeted vulnerability test patterns.
- The full automation, ensuring risk traceability, of the test purpose selection (given by the risk model), test case execution and verdict assignment from risk assessment results.

The paper is organized as follows: Section 2 introduces the context and the principles of the RBVT approach. Section 3 details the use of the RBVT by describing the content of the testing input artefacts, the test pattern language and the risk-driven test generation, which is illustrated using a simple example of Web application, namely eCinema. Related work about vulnerability detection is discussed in Section 4. Finally, conclusion and future works are given in Section 5.

2 Context and principles of the RBVT approach

Model-Based Testing (MBT) [6] is a software testing approach in which both test cases and expected results are automatically derived from an abstract model of the SUT. MBT is usually performed to automate and rationalize functional black-box testing. It is a widely-used approach that has gained much interest in recent years, from academic as well as industrial domains, especially by increasing and mastering test coverage, including support for certification, and by providing the degree of automation needed for accelerating the test process [7].

More precisely, MBT techniques derive abstract test cases (including stimuli and expected outputs) from an abstract model, and enable the generation of executable tests from these abstract test cases. The abstract model, called test model, formalizes the behavioural aspects of the SUT in the context of its environment and at a given level of abstraction. It thus captures the control and observation points, the expected dynamic behaviour, the data associated with the tests, and finally the initial state of the SUT. The test cases generated from such models allow to validate the functional aspects of the SUT by comparing back-to-back the results observed on the SUT with those specified by the model. Therefore MBT aims to ensure that the final product conforms to the initial functional requirements. However, if these techniques are used to cover the functional requirements specified in the test model of the SUT, they are also limited to this scope since what is not modeled cannot be tested.

The proposed approach to perform vulnerability testing is based on MBT process, and is thus composed of the four activities depicted in Figure 1:

① the *Test Purposes* activity consists of formalizing test purposes from vulnerability test goals that the generated test cases have to cover;
② the *Modeling* activity aims to define a model that captures the behavioral aspects of the SUT in order to generate consistent (from a functional point of view) sequences of stimuli;
③ the *Test Generation* activity comprises the automated production of abstract test cases from the artefacts defined during the two previous activities;
④ the *Adaptation, Test Execution and Observation* activity aims (i) to 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.

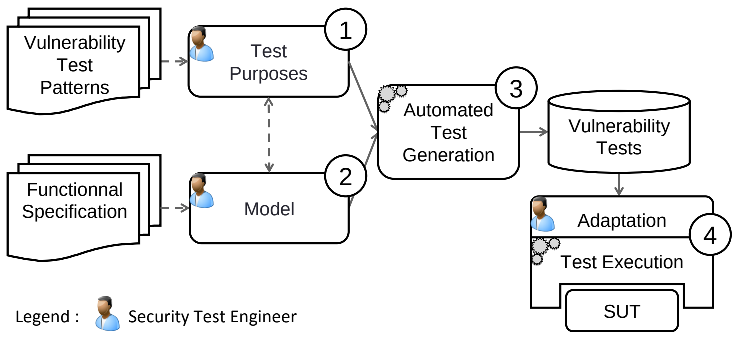

Vulnerability Test Patterns
Test Purposes
1
Automated Test Generation
3
Vulnerability Tests
Functionnal Specification
Model
2
Adaptation
Test Execution
4
SUT
Legend : Security Test Engineer

Fig. 1. Model-based vulnerability testing process

For a further description of each activities of this model-based vulnerability testing process, a detailed presentation can be found in [8].

All these activities are supported by a dedicated toolchain, which is based on an existing MBT software named *CertifyIt* [9] provided by the company Smartesting[3]. This software is a test generator that takes as input a test model, written with a subset of UML/OCL (called UML4MBT [10]), which captures the behavior of the SUT. Concretely, a UML4MBT test 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. OCL expressions provide the expected level of formalization necessary for model-based testing modeling since an operational interpretation of OCL postconditions makes it possible to determine its effect (this specific interpretation of OCL, called OCL4MBT [10], basically consists to interpret OCL equality as an assignment). That is why such UML4MBT test models have a precise and unambiguous meaning, so that these models can be understood and processed by the *CertifyIt* technology. This precise meaning makes it possible to simulate the execution of the test models and to automatically generate test cases by applying predefined coverage strategies or by applying test directives formalized by a dedicated test purpose language.

A *test purpose* is a high-level expression that formalizes a test intention linked to a test objective to drive the automated test generation on the test model. This is a textual language based on regular expressions, allowing the formalization of vulnerability test intention in terms of states to be reached and operations to be called. This test purpose language has been originally designed to drive model-based test generation for security components, typically Smart card applications and cryptographic components [11]. This test purpose language has been extended to be able to formalize typical vulnerability test patterns for Web applications in conjunction with generic and specific test models.

Each of such generated test cases is typically an abstract sequence of high-level actions (operations) specified in the UML4MBT test 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 OCL4MBT constraints. About this vulnerability testing process, it should be noted that, within the traditional MBT process that allows to generate functional test cases, positive test cases are computed to validate the SUT in regards to its functional requirements. 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. Within vulnerability testing approach, “negative test cases” have to be produced: typically, attack scenarios to obtain data from the SUT in an unauthorized manner. A negative test case thus targets an unexpected use of the SUT in order to show that the SUT allows something that it is not supposed to allow. In our approach, when a negative test case succeeds, it highlights a problem in the SUT.

[3] `http://www.smartesting.com`

We propose to drive this vulnerability testing process by risk assessment in order to perform and automate risk-based testing for Web applications. *Risk & requirements-based testing* was originally the title of an article from James Bach [12]. This article was underlining the creative aspects of software testing to manage stated and unstated requirements depending on risks associated with the SUT. Risk may be defined as the combination of the impact of the severity (consequence) and the likelihood (probability) of a hazardous failure of the SUT. A risk-based testing management method focuses on risk assessment and test prioritization based on requirements. Within MBT, this approach influences the entire testing process, and has the following impacts:

– Risk analysis drives the development and maintenance of test generation artefacts: the level of detail as well as the scope of test generation models are determined according to established priorities. This impacts the test models, which have to capture risk aspects besides functional features.
– During the test generation phase, test selection criteria applied on the test models are specified to cover risk and priorities for requirements coverage.

Therefore, MBT allow to implement risk-based testing in the modeling phase by adapting modeling effort to risk analysis and assessment, and in the test generation phase by adapting test selection criteria to risk-based test priorities. RBVT aims to integrate the existing model-based vulnerability testing approach with risk-based testing approach. Concretely, it consists somehow to drive the test generation regarding risk assessment results and using dedicated vulnerability test patterns. The RBVT overall testing process is depicted in Figure 2.

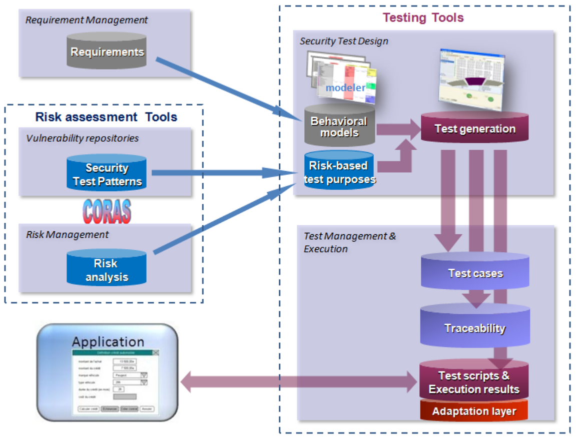


Fig. 2. Risk-Based Vulnerability Testing overall process

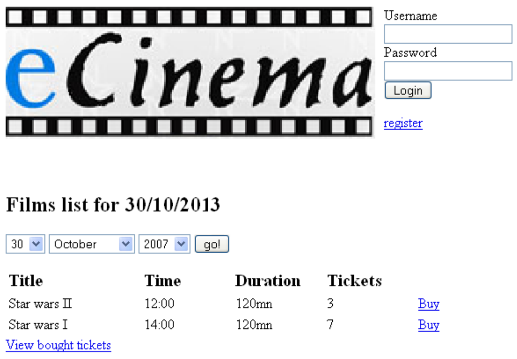
RBVT process starts by risk analysis, for example with an approach such as CORAS [13], which provides a customized language for threat and risk modeling. Security test patterns based on identified and prioritized vulnerabilities from the risk analysis provide a starting point for test case generation: they indeed link the risk analysis results and security testing goal by providing information how relevant vulnerability test cases can be derived from risk assessment.

In order to generate these expected vulnerability test cases, test cases are automatically derived from a formalization of the security test patterns using the test purpose language. Finally, the last step consists to export the abstract test cases into an execution environment, in which they are concretized using a dedicated adaptation layer to be executed. Moreover, this process makes it possible to manage the traceability between the targeted security test patterns (formalized with test purposes) and the associated generated test cases. This management is performed through the automated generation, during the test generation process, of a traceability matrix that links vulnerabilities to generated test cases. To support this RBVT process, the *CertifyIt* technology has been extended by the following developments:

– Import of the risk analysis results. It enables to select the related test purposes and to prioritize them regarding risk identification and estimation.
– Test purpose language extensions. On the one hand, the definition of keywords enables to provide generic Test Purposes related to security test patterns, and to help for maintenance and reuse. On the other hand, a mechanism to link a Test Purpose to a requirement identifier has been created to ensure the traceability through the all test generation process.
– Test Purpose catalogue import/export. It makes it possible to reuse and apply generic Test Purposes on several SUT.

3 Applying the RBVT approach

In this section, we detail each activity of the process introduced in Figure 2, including the features introduced at the end of the previous section. For each activity, we present its objectives as well as the tooling that automates it. The eCinema running example is used to illustrate our statements. Basically, eCinema is a simple Web application that allows a customer to buy tickets on line before to go to his favorite cinema. The welcome screen, depicted in Figure 3, displays the list of available movies and show times.



Films list for 30/10/2013

Title	Time	Duration	Tickets	
Star wars II	12:00	120mn	3	Buy
Star wars I	14:00	120mn	7	Buy

View bought tickets

Fig. 3. eCinema welcome screen

Before selecting tickets, a user should be logged to the system. This requires a registration. A registration is valid when a user gives a name (not already used) and a valid password. A valid new registration implies that the user is automatically logged in. When logged in, the user can buy tickets. If tickets are available, he can buy some of them and see his basket to verify his selection. When checking his selection, the user can delete tickets and then the number of available tickets for the session is automatically updated.

3.1 Selection and prioritization of vulnerabilities from risk analysis

The starting point of the process is the identification and prioritization of the vulnerabilities, which are defined by a risk analysis activity. These results are indeed used to drive the test generation strategy. Our approach is based on the CORAS risk assessment method [13]. CORAS is a model-driven method for risk analysis featuring a tool-supported modelling language especially designed to model risks that are common for a large number of systems. Such models serve as a basis to perform risk identification and prioritization. For example, Figure 4 shows an example of CORAS threat diagram describing SQL Injection vulnerability that can occur when a user is logging the eCinema Web application. In this context, due to the *insufficient user validation* threat, *SQL Injection successful* is a threat scenario and can lead to the unwanted incident defined by the *disclosure of confidential information*. The likelihood of the threat is considered as *possible* and its consequence *moderate*.

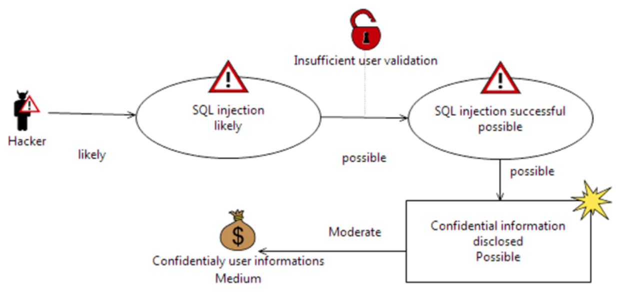

Insufficient user validation
Hacker
likely
SQL injection likely
possible
SQL injection successful possible
possible
Confidential information disclosed Possible
Moderate
Confidentialy user informations Medium

Fig. 4. CORAS model example for SQL Injection vulnerability

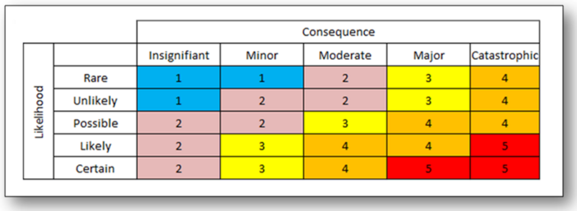
Each identified threat scenario is linked to a dedicated vulnerability test pattern (vTP). A vTP defines the testing procedure allowing the detection of the corresponding threat in a Web application. There are as much vTP as there are types of application-level breaches. The ITEA2 DIAMONDS[4] research project provided a first definition, as well as a first listing of vTP [14], which has been extended for test generation needs. Figure 5 presents an excerpt of the vulnerability test pattern defining the SQL Injection.

[4] `http://www.itea2-diamonds.org`

Name	SQL Injection
CWE-ID(s)	CWE-89
Description	The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.
Objective(s)	Based on attack pattern CAPEC-66 1. Use the application, client or Web browser to inject SQL constructs input through text fields or through HTTP GET parameters. 2. Use a possibly modified client application or Web application debugging tool such to submit SQL constructs for submitted values or to modify HTTP POST parameters, hidden fields, non-free form fields, etc. 3. Check for error messages, delays, disclosed values in the client application and new / modified / deleted values in the database. Detect if a user input can embed malicious datum enabling a Reflected XSS attack.
Test Data	SQL Injection Cheat Sheet
...	...
References	OWASP Top 10 (2013): A1-Injection, CAPEC-7: Blind SQL Injection, CAPEC-66: SQL Injection, OWASP Testing Guide: Testing for SQL Injection (OWASP-DV-005), OWASP: Automated Audit using SQLMap

Fig. 5. Vulnerability test pattern for SQL Injection

The vulnerability test patterns that have to be used by the test generation algorithm are gathered from the threat scenarios of each CORAS model related to the SUT. Moreover, likelihood and consequence are also collected from the CORAS model to assign a priority to the threat scenarios, and thus to prioritize them. Figure 6 shows the risk assessment matrix that enables to set such priority.



		Consequence				
		Insignifiant	Minor	Moderate	Major	Catastrophic
Likelihood	Rare	1	1	2	3	4
	Unlikely	1	2	2	3	4
	Possible	2	2	3	4	4
	Likely	2	3	4	4	5
	Certain	2	3	4	5	5

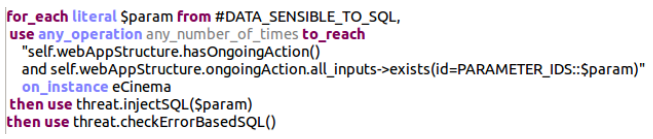
Fig. 6. Risk evaluation matrix

The assigned priority level (from 1 to 5) will be used during test case generation to select the coverage of the test purpose (priority 1 defines the lower coverage and so less generated test cases, whereas priority 5 defines the higher coverage and so more generated test cases). The CWE identifiers and the corresponding priority levels are then exported to *CertifyIt* in order to drive the test generation process, which is presented in the next subsections.

3.2 Formalizing Vulnerability Test Patterns into Test Purposes

In the test generation tool, dedicated and generic test purposes make it possible to formalize each vTP imported from risk assessment. A test purpose is a high level expression that formalizes a test intention linked to a test objective to drive the automated test generation on the test model. It allows the formalization of vulnerability test intention in terms of states to be reached 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 major *stages* to be reached. A stage is a set of operations or behaviors to use, or/and a state to reach. Transforming the sequence of stages into a complete test case, based on the test model, is left to the MBT technology (more details will be given in subsection 3.4). Furthermore, at the beginning of a test purpose, are defined *iterators* that are used in the stages in order to introduce context variations (the threat priority exported from CORAS model is then used to set a given level of variation combinations). Each combination of possible values of iterators produces a specific test case.

Figure 7 shows the instantiated test purpose formalizing the vTP of Figure 5. This schema precises that for all malicious data enabling the detection of SQL Injection and from all sensible Web pages, it is required to do the following actions: (i) use any operation to activate the sensible page, (ii) inject malicious data in all the user inputs of the page, (iii) check if the page is sensible to the attack. The keywords ALL_* define enumerations of values allowing to master the final amount of test cases regarding test priority.



```
for_each literal $param from #DATA_SENSIBLE_TO_SQL,
 use any_operation any_number_of_times to_reach
   "self.webAppStructure.hasOngoingAction()
   and self.webAppStructure.ongoingAction.all_inputs->exists(id=PARAMETER_IDS::$param)"
   on_instance eCinema
 then use threat.injectSQL($param)
then use threat.checkErrorBasedSQL()
```

Fig. 7. Test purpose formalizing the SQL Injection vTP (of Figure 5)

Finally, variants of malicious data are defined during the modeling activity, variants of the procedure are defined during the adaptation and execution activity. In order to generate tests from models, the test purposes is used in conjunction with the test model, which is introduced in the next subsection.

3.3 Modeling

As for every MBT approach, the modeling activity consists of designing a test model that will be used as basis to generate the abstract test cases. This model uses the UML notation to represent the Web application to be tested. We will see that some parts of the model are generic and re-usable for modeling any Web applications, while some other parts are specific to the Web application that is considered. We present in the following the used UML diagrams (classes, objects, statechart diagrams), and their respective use in the context of our approach.

Class diagrams specify the static aspect of the model, by defining in an abstract manner the structure and entities managed by the SUT. *Classes* model business objects. *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 describe here the navigation between pages). In the context of Web applications, the model presents some generic parts, shown in Figure 8, which are the same for all considered Web applications:

– four classes (*WebAppStructure*, *Page*, *Action* and *Data*) and their associations respectively model the general structure of the application, the available pages (or screens in case single-URL applications), the available actions on each page, and the user inputs of each action potentially used to inject an attack vector (i.e. malicious data to perform the attack). The login page of an application is modeled using:
  - a particular *'Login' Page*, modeling the application's login page;
  - a particular *'Login' Action*, modeling the sending of the form to the server
  - two particular *Data*, modeling the user's name and password

– the *Threat* class models the potential threats: its operations *injectXSS()* and *checkXSS()* model the means to exercise and observe the attack.

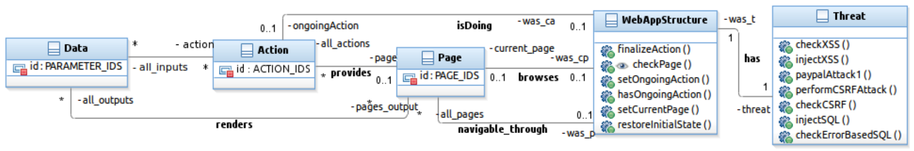


Fig. 8. Generic class diagram of the SUT structure

Figure 9 presents the class model of the eCinema example. These classes are eCinema specific classes, and are in addition to the generic classes presented in Figure 8. This class diagram displays the additional classes *ECinema* and *User* to respectively model the SUT and its potential users.

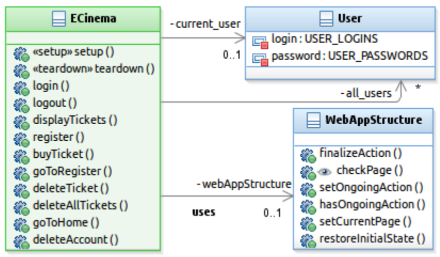


Fig. 9. eCinema-specific class diagram

The UML statechart diagram graphically represents the behavioral aspect of the SUT, modeling the navigation between pages in the Web applications. *States* model Web pages, and *transitions* model the available links between these Web pages (HTML links, form submissions, etc.). *Triggers* of transitions are the UML operations of the SUT class. *Guards* of transitions (specified using OCL4MBT) precisely define the execution context of the transition. Finally, the *effects* of the transitions (also specified using OCL4MBT) precisely describe its expected behavior that should be modeled for vulnerability test generation. Figure 10 presents the statechart diagram of the eCinema example.

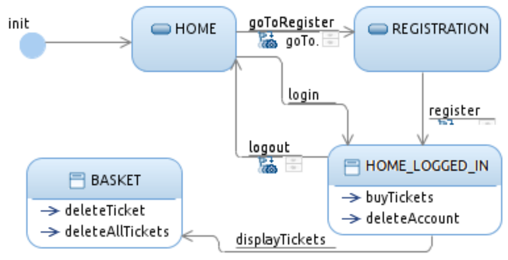


Fig. 10. eCinema statechart diagram

The UML object diagram models the initial state of the SUT by instantiating the class diagram: the *instances* model business entities available at the initial state, and the *links* instantiate the associations between these instances. In our approach, the object diagram models the Web pages and the user inputs of these pages. Figure 11 presents the initial state of the eCinema example. It specifies: (*a*) one user, with its credentials, and (*b*) the pages and user inputs of eCinema.

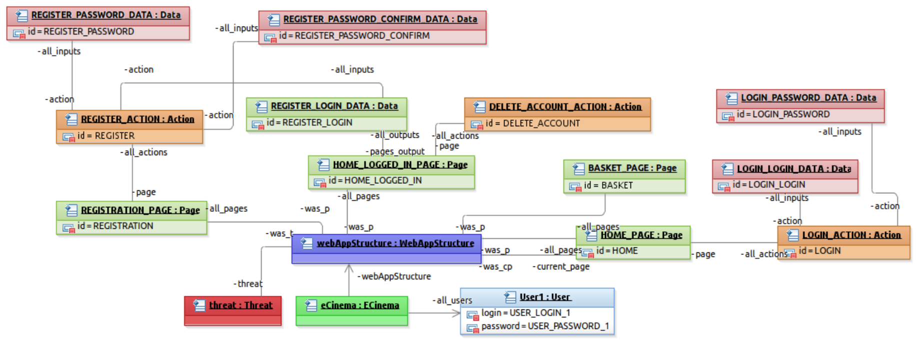


Fig. 11. eCinema object diagram for the initial state

These last two parts (namely, the statechart diagrams and the object diagrams) are necessarily specific to each considered application.

3.4 Test generation and execution

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 computable by the *CertifyIt* MBT tool. Notably, test purposes are transformed into *test targets*, which can be seen as 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 an amount of test targets depending of the priority level calculated during risk assessment. The generator respectively calculates the first N combinations such that $N = maxCombination/(5 - priority + 1)$ where $maxCombination$ and $priority$ respectively denote the maximum amount of possible combinations and the priority level. For example, if the priority is 5 (the higher), all the combinations of values between iterators are expanded.

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.

Finally, the third phase exports the abstract test cases into the execution environment. In our case, it consists of (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.

In the test model, all data used by the application (page, user input field, malicious datum, user credentials, etc.) are specified in an abstract way. Hence, the test suite cannot 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. Stimuli must also be adapted. When exporting abstract test cases, the MBT tool provides an interface defining each operation signature. The test automation engineer is in charge to implement the automated execution of each operation of this interface. Since we are testing Web applications, two ways of automation are proposed:

- 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, Selenium framework is used.
- 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.

4 Related work

Related work on vulnerability detection can be classified into two categories: static and dynamic analysis security testing. Static Application Security Testing (SAST) are white-box approaches including source, byte and object code scanners and static analysis techniques. Dynamic Application Security Testing (DAST) includes black-box web application 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 known to be efficient to detect buffer overflow and badly formatted string, but weak to detect SQLI, XSS or CSRF vulnerabilities. RBVT is a dynamic testing technique, so this section focuses on DAST techniques by providing a state of the art of emerging model-based security testing techniques.

Web application vulnerability scanners aim to detect vulnerabilities by injecting attack vectors. These tools generally include three main components [15]: a crawler module to follow Web links and URLs in the Web applications in order to retrieve injection points, an injection module which analyzes Web pages, input points to inject attack vectors (such as SQL Injection), and an analysis module to determine possible vulnerabilities based on the system response after attack vector injection. As shown in recent comprehensive studies [16, 17], corroborated by research papers [4, 5] and confirmed by our own experience with tools such as IBM AppScan[5], these tools suffer from two major weaknesses that highly decrease their practical usefulness:

– **Limitations in application discovery** As black-box Web vulnerability scanners ignore any request that can change the state of the Web applications, they miss large parts of the application. Therefore, these tools test generally a small part of the Web applications due to the ignorance of the application behavioral "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 applications 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 behavioral intelligence) and by increasing the precision of the verdict assignment.

[5] `http://www.ibm.com/software/awdtools/appscan/`

In this way, model-based security testing are emerging techniques aiming to leverage model-based approaches for security testing [18]. This includes:

- **Model-based test generation from security protocol, access-control or security-oriented models.** Various types of models of security aspects of the SUT have been considered as input to generate security test. For example, [19] proposes a method using security protocol mutation to infer security test cases. [20] develops a model-based security test generation approach from security models in UMLSec. [21] 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. *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 [22]. Test execution results can therefore expose various invalid behaviors such as crash effects, failing built-in code assertions or memory leaks. [23] proposes an approach that generates invalid message sequences instead of invalid input data by applying behavioral fuzzing operators to valid message UML sequence diagrams.
- **Model-based test generation from weakness or attack models.** Test cases are generated using threat, vulnerability or attacker models, which reflects the attack steps and the required associated data. For example, in [24], threats of security policies modeled with UML sequence diagrams allow to extract event sequences that should not occur during the system execution.

Complementary to these model-based techniques for security testing, our Risk-Based Vulnerability Testing approach is based on a model that captures functional behavioral features of the SUT, but also specifies the fields that allow possible attacks. This feature enables to generate more accurate test cases. Moreover, contrary to functional MBT, the proposed RBVT process is directly driven by the risk analysis (with CORAS) and the vulnerability test patterns, so that the behavioral model is restricted to the only elements that are needed to compute risk-based vulnerability test cases.

5 Conclusion and future works

This paper has introduced the RBVT approach, that integrates techniques addressing both risk-based test identification and test prioritization to drive the overall model-based testing generation process. System requirements are used to write the UML test model, while the CORAS security model (in relation with associated generic test pattern catalogue) enables to define selected test purposes and to prioritize them regarding risk assessment. The UML test model completed with selected test purposes defines the input of the test generation tool *CertifyIt*, which automatically derives abstract risk-based vulnerability test cases and next test scripts that can be executed on the SUT. The overall process ensures the traceability between the generated test cases and the targeted vulnerabilities identified during risk assessment.

To achieve and automate this process, we have developed and extended the existing MBT toolchain, based on *CertifyIt*, in order to manage the risk treatment by applying appropriate testing strategies regarding risk assessment. Concretely, the generation of test cases is driven by the risk assessment results, in terms of system perimeter, type of vulnerabilities and associated risk level.

The future work leads in three 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 (2) investigate methods to gather and aggregate test results, which could be used to automatically complement the risk assessment picture. This feature is indeed enabled by the risk traceability matrix from/to generated test cases and vulnerability objectives. Finally, we want to (3) study the scalability of the testing process to address large scale systems. To reach this goal, we propose to define and support a compositional testing approach by proposing a model composition strategy.

Acknowledgements

The evolution of Smartesting technology to support this risk-based vulnerability test generation is mostly developed within the European FP7 project RASEN[6], which aims to provide risk-based security testing techniques for large-scale networked systems.

References

1. Wichers, D.: Owasp top 10. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project` (October 2013) Last visited: February 2014.
2. MITRE: Common weakness enumeration. `http://cwe.mitre.org/` (October 2013) Last visited: February 2014.
3. Whitehat: Website security statistics report. `https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf` (October 2013) Last visited: February 2014.
4. Doupé, A., Cova, M., Vigna, G.: Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Proc. of the 7th Int. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10), Bonn, Germany, Springer (July 2010) 111–131
5. Finifter, M., Wagner, D.: Exploring the relationship between web application development tools and security. In: Proc. of the 2nd USENIX Conference on Web Application Development (WebApps'11), Portland, OR, USA, USENIX Association (June 2011) 99–111
6. Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach. Morgan Kaufmann, San Francisco, CA, USA (2006)
7. Dias-Neto, A., Travassos, G.: A Picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges. Advances in Computers **80** (July 2010) 45–120 ISSN: 0065-2458.

[6] `http://www.rasenproject.eu/`

8. Lebeau, F., Legeard, B., Peureux, F., Vernotte, A.: Model-Based Vulnerability Testing for Web Applications. In: Proc. of the 4^{th} Int. Workshop on Security Testing (SECTEST'13), Luxembourg, IEEE CS Press (March 2013) 445–452
9. 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), Leipzig, Germany, ACM Press (May 2008) 45–48
10. 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), London, UK, ACM Press (July 2007) 95–104
11. 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), Luxembourg, IEEE CS (March 2013) 192–201
12. Bach, J.: Risk and Requirements-Based Testing. Computer **32**(6) (June 1999) 113–114 IEEE Press.
13. Lund, M.S., Solhaug, B., Stølen, K.: Model-Driven Risk Analysis: The CORAS Approach. 1^{st} edn. Springer Publishing Company, Incorporated (2010)
14. 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].
15. 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), Oakland, CA, USA, IEEE CS (May 2010) 332–345
16. Allan, D.: Web application security: automated scanning versus manual penetration testing. IBM White Paper (2008) `ftp://ftp.software.ibm.com/software/rational/web/whitepapers/r_wp_autoscan.pdf` [Last visited: February 2014].
17. SecToolMarket: Price and Feature Comparison of Web Application Scanners. `http://www.sectoolmarket.com/` (February 2014) Last visited: February 2014.
18. Schieferdecker, I., Grossmann, J., Schneider, M.: Model-based security testing. In: Proc. of the 7^{th} Int. Workshop on Model-Based Testing (MBT'12). Volume 80 of EPTCS., Tallinn, Estonia, Open Publishing Association (March 2012) 1–12
19. Dadeau, F., P-C.Héam, Kheddam, R.: Mutation-Based Test Generation from Security Protocols in HLPSL. In: Proc. of the 4^{th} Int. Conf. on Software Testing, Verification and Validation, Berlin, Germany, IEEE CS (March 2011) 240–248
20. Jürjens, J.: Model-based Security Testing Using UMLsec: A Case Study. The Journal of Electronic Notes in Theoretical Computer Science (ENTCS) **220**(1) (December 2008) 93–104
21. Buchler, M., Oudinet, J., Pretschner, A.: Semi-Automatic Security Testing of Web Applications from a Secure Model. In: Proc. of the 6^{th} Int. Conference on Software Security and Reliability (SERE'12), Gaithersburg, MD, USA, IEEE CS (June 2012) 253–262
22. Takanen, A., DeMott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House, Inc., Norwood, MA, USA (2008)
23. Schneider, M., Großmann, J., Tcholtchev, N., Schieferdecker, I., Pietschker, A.: Behavioral Fuzzing Operators for UML Sequence Diagrams. In: Proc. of the 7^{th} Int. Workshop on System Analysis and Modeling (SAM'12). Volume 7744 of LNCS., Innsbruck, Austria, Springer (October 2012) 88–104
24. Wang, L., Wong, E., Xu, D.: A threat model driven approach for security testing. In: Proc. of the 3^{rd} Int. Workshop on Software Engineering for Secure Systems (SESS'07), Minneapolis, MN, USA, IEEE CS (May 2007)