

SPIM

Thèse de Doctorat



école doctorale **sciences pour l'ingénieur et microtechniques**
UNIVERSITÉ DE FRANCHE-COMTÉ

N° X X X

THÈSE présentée par

ALEXANDRE VERNOTTE

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Pattern-Driven and Model-Based Vulnerability Testing for Web Applications

Approche de test à partir de patterns et modèles pour les vulnérabilités des
applications Web

Unité de Recherche :

Institut Femto-ST - UMR CNRS 6174

Soutenue publiquement le 29 Octobre 2015 devant le Jury composé de :

OLGA KOUCHNARENKO	Présidente	Professeur à l'Université de Franche-Comté, France
ROLAND GROZ	Rapporteur	Professeur à l'Institut Polytechnique de Grenoble, France
FRANZ WOTAWA	Rapporteur	Professeur à Technische Universität Graz, Autriche
FREDRIK SEEHUSEN	Examineur	Senior Researcher à SINTEF ICT, Norvège
BRUNO LEGEARD	Directeur de thèse	Professeur à l'Université de Franche-Comté, France
FABIEN PEUREUX	Co-encadrant de thèse	Maître de Conférence à l'Université de Franche-Comté, France

ACKNOWLEDGEMENTS

En premier lieu, je remercie Messieurs Bruno Legeard et Fabien Peureux pour avoir supervisé mes travaux de thèse. Bruno Legeard d'abord, pour m'avoir transmis sa grande vision du monde de la recherche, pour ses inombrables conseils, et pour sa disponibilité. Fabien Peureux ensuite, pour m'avoir assisté de multiples façons pendant ces trois années, principalement pour lutter tant que faire se pouvait contre ma fâcheuse tendance à être tête en l'air. Cela va sans dire, son aide inestimable au quotidien a constitué un facteur important pour la réussite de ma thèse, et je lui en suis infiniment reconnaissant.

Je remercie sincèrement Madame Olga Kouchnarenko pour m'avoir fait l'honneur de présider mon jury de thèse.

As well, I warmly thank Franz Wotawa and Roland Groz for accepting to report on my research, and Fredrik Seehusen for being an examiner of my PhD.

Je remercie également Monsieur Franck Lebeau pour avoir été moteur lors du démarrage de mes travaux, m'ayant ainsi permis de minimiser les premiers moments difficiles du thésard tentant d'appivoiser son domaine de recherche.

Je tiens également à remercier Madame Elizabeta Fournernet pour sa disponibilité et son efficacité quant à la relecture de mon manuscrit de thèse.

Enfin, merci aux accolytes du 427C, Julien et Romain, ainsi qu'aux "sympathisants" du bureau, JM, Alban, Hadrien, Maria et j'en passe, pour leur bonne humeur, leur humour certainement criticable mais néanmoins appréciable, et pour m'avoir offert une raison supplémentaire de me diriger chaque jour avec plaisir vers le DISC.

LIST OF ORIGINAL PUBLICATIONS

- [1] Alexandre Vernotte, Cornel Botea, Bruno Legeard, Arthur Molnar, and Fabien Peureux. Risk-driven vulnerability testing: Results from ehealth experiments using patterns and model-based approach. In *RISK'15, 3rd Int. Workshop on Risk Assessment and Risk-Driven Testing*, jun 2015.
- [2] Mark Utting, Fabrice Bouquet, Elizabeta Fourneret, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Chapter: Recent advances in model-based testing. In *Advances in Computers*. Elsevier, 2015. To appear.
- [3] Bruno Legeard, Fabien Peureux, Martin Schneider, Fredrik Seehusen, and Alexandre Vernotte. The PMVT approach: a RASEN innovation for security pattern and model-based vulnerability testing. White paper, RASEN FP7 EU founded Research Project, April 2015. Available at <http://www.rasenproject.eu> (Accessed: 2015-05-04).
- [4] Cornel Botea and Alexandre Vernotte. The RASEN FP7 project innovations. Exhibition at the Cyber Security & Privacy Innovation Forum 2015, Brussels, Belgium, April 2015.
- [5] Alexandre Vernotte, Bruno Legeard, and Fabien Peureux. A pattern-driven and model-based test generation toolchain for Web vulnerability. In *Demo Session of ESSoS'15, the Int. Symposium on Engineering Secure Software and Systems*, Milan, Italy, March 2015.
- [6] Alexandre Vernotte, Frédéric Dadeau, Franck Lebeau, Bruno Legeard, Fabien Peureux, and François Piat. Efficient detection of multi-step cross-site scripting vulnerabilities. In *Information Systems Security - 10th Int. Conf., ICISS 2014, Hyderabad, India, December 16-20, 2014, Proceedings*, volume 8080 of *LNCS*, pages 358–377. Springer, 2014.
- [7] Julien Botella, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Risk-based vulnerability testing using security test patterns. In *ISoLA'14, 6th Int. Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, volume 8803, pages 337–352, Corfu, Greece, oct 2014. Springer.
- [8] Alexandre Vernotte, Bruno Legeard, and Fabien Peureux. Test de vulnérabilité Web à base de patterns et de modèles. In *6-èmes journées nationales du GDR CNRS du Génie de la Programmation et du Logiciel*, pages 123–124, Paris, France, June 2014.
- [9] Alain Ribault, Bruno Legeard, and Alexandre Vernotte. Les tests et le développement sécurisé peuvent-ils réduire les failles d'un système et les risques de hacking ? *Magazine Programmez*, 173:12–13, April 2014.
- [10] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Génération de tests de vulnérabilité web à partir de modèles. In *AFADL'13, 12èmes journées Francophones sur les Approches Formelles dans l'Assistance Au Développement de Logiciels*, pages 49–63, Nancy, France, April 2013.

- [11] Franck Lebeau, Bruno Legeard, Fabien Peureux, and Alexandre Vernotte. Model-based vulnerability testing for web applications. In *SECTEST'13, 4th Int. Workshop on Security Testing. In conjunction with ICST'13, 6th IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 445 – 452, Luxembourg, Luxembourg, mar 2013. IEEE Computer Society Press.
- [12] Bruno Legeard and Alexandre Vernotte. Active testing techniques. Tutorial talk at ICST'13, 6th IEEE Int. Conf. on Software Testing, Verification and Validation, Luxembourg, Luxembourg, March 2013.
- [13] Alexandre Vernotte. Research questions for model-based vulnerability testing of web applications. In *ICST'13 PhD Symposium, held during the 6th IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 505–506, Luxembourg, Luxembourg, mar 2013. IEEE Computer Society Press.

CONTENTS

1	Overview of Research	1
1.1	Cybersecurity threats	1
1.2	Motivation and Research Objectives	2
1.3	Research Scope	4
1.4	Contributions of the Thesis	5
1.4.1	Domain-Specific Modeling Language	6
1.4.2	Vulnerability Test Pattern Formalization	7
1.4.3	Pattern-driven and Model-based Vulnerability Testing process	8
1.4.4	toolchain	8
1.4.5	Experimentation	9
1.4.6	Contribution within the FP7 RASEN Project	9
1.5	Dissertation Structure	10
2	Presentation of the targeted Vulnerabilities	11
2.1	Cross-Site Scripting	12
2.2	SQL Injections	15
2.3	Cross-Site Request Forgery	18
2.4	Privilege Escalation	21
3	State of the Art	25
3.1	Static Application Security Testing	26
3.2	Dynamic Application Security Testing	27
3.2.1	Automated Penetration Testing: Web application Scanners	28
3.2.2	Manual/Tool-aided Penetration Testing: Ethical Hacking	29
3.3	Model-Based Vulnerability Testing	31
3.3.1	Pattern-Based and Attack-Model Based Approaches	32
3.3.2	Model-Checking Approaches	34
3.3.3	Fuzzing Approaches	35
3.4	Synthesis	36

4	Background	39
4.1	CertifyIt: Software Testing based on UML and OCL	39
4.1.1	Modeling Notation	41
4.1.1.1	Class Diagrams: Static Structure	41
4.1.1.2	Object Diagrams: Initial state	42
4.1.1.3	State-machine Diagrams: Dynamic structure	43
4.1.1.4	Object Constraint Language: Constraints and Actions	44
4.1.2	Test Selection	45
4.1.2.1	Functional Behavioral Testing	45
4.1.2.2	Test Purposes in CertifyIt	46
4.1.3	Test Generation	50
4.1.4	Test Concretization, Execution and Verdict Assignment	51
4.2	CertifyIt for Vulnerability Testing	52
5	PMVT: Process and Modeling Notation	55
5.1	PMVT Process	55
5.2	Running Example: Cuiteur	56
5.3	Web application Modeling Notation	60
5.3.1	DASTML: a Dedicated Language for Vulnerability Testing of Web applications	60
5.3.2	PMVT with UML4MBT	65
5.3.2.1	Generic Class Diagram: PMVT Metamodel	65
5.3.2.2	Specific Class Diagram: Application-dependent Information	70
5.3.2.3	Object Diagram: Structure Content	72
5.3.2.4	State-machine: Dynamic Structure	73
5.3.2.5	Modeling Concepts	74
5.3.3	From DASTML to UML4MBT	78
5.4	Synthesis	80
6	PMVT: Formalization of Vulnerability Test Patterns	81
6.1	Augmented Test Purpose Language for PMVT	82
6.2	SMA Test purposes for Web application Vulnerability Testing	85
6.2.1	Cross-Site Scripting	85
6.2.2	SQL Injections	88
6.2.3	Cross-Site Request Forgeries	94
6.2.4	Privilege Escalation	96

6.3	Synthesis	100
7	toolchain	101
7.1	Implementation of the PMVT Approach	101
7.1.1	Modeling Activity	102
7.1.2	Test Purpose Activity	104
7.1.3	Test Generation Activity	105
7.1.4	Test Concretization and Execution Activities	106
7.2	PMVT tools for Concrete Vulnerability Testing	107
7.2.1	PMVT Test Publisher	108
7.2.2	Executable Test Scripts	109
7.2.3	CSRF Web Server	109
7.2.4	Web Page Comparator	110
7.3	RASEN: Test Selection from Risk Assessment	111
8	Evaluation	115
8.1	Case Studies	115
8.1.1	WackoPicko	116
8.1.2	Bookshop	117
8.1.3	Stud-E	118
8.1.4	Medipedia Web Portal	119
8.2	Web application Vulnerability Scanners	120
8.2.1	IBM AppScan	121
8.2.2	IronWasp	122
8.3	Experiment Results	122
8.3.1	Experimental Setup	122
8.3.2	Dummy Web applications	123
8.3.3	Medipedia	126
8.3.4	Stud-E	127
8.4	Threats to Validity	128
8.5	Discussion	129
9	Conclusion	133
9.1	Summary	133
9.2	Future works	134
9.2.1	Improvements of the PMVT Approach	135

9.2.1.1	Model Inference	135
9.2.1.2	Composition with Scanners	137
9.2.1.3	Verdict Assignment	137
9.2.2	Evolutions of the PMVT Approach	138
9.2.2.1	Extension to Address Insecure Direct Object References .	138
9.2.2.2	Online MBT	139

A Appendix**147**

LIST OF FIGURES

2.1	Stored XSS typical workflow	13
2.2	XSS usage example: server-side and client-side code	14
2.3	SQL Injection: Typical Workflow	15
2.4	Authentication: Server-side and Client-side	16
2.5	CSRF: Typical Workflow	19
3.1	Web application Vulnerability Testing Overview	26
4.1	Smartesting CertifyIt Process	40
4.2	OCL4MBT expression: D/CC coverage	46
4.3	Test Purposes workflow for the production of test targets	47
5.1	PMVT: General Process	56
5.2	Cuiteur: Login and Registration System	57
5.3	Cuiteur: Home Page	58
5.4	Cuiteur: Profile Page	59
5.5	Cuiteur: User Search Page	59
5.6	Syntax of the DAST Modeling Language	61
5.7	From DASTML to UML4MBT	65
5.8	PMVT: Generic Class Diagram	66
5.9	PMVT Class Diagram: Specific Entities of Cuiteur	71
5.10	PMVT Object Diagram of Cuiteur	72
5.11	PMVT State-Machine of Cuiteur	74
5.12	Action Handling: Nominal and Attack Traces	76
6.1	Generic Vulnerability Test Pattern	82
6.2	Abstract XSS Attack Trace on Cuiteur	88
6.3	Error-based and Time-Based Abstract SQL Injection Attack Traces on Cuiteur	93
6.4	Abstract SQL Injection Attack Traces on Cuiteur	94
6.5	Abstract CSRF Attack Trace on Cuiteur	96
6.6	Abstract Privilege Escalation Attack Traces on Cuiteur	99

7.1	Overview of the PMVT toolchain	102
7.2	PMVT: Modeling Environment	103
7.3	PMVT: Test Purposes Editor	104
7.4	PMVT: Test Purposes Catalog	105
7.5	PMVT: Test Generation Environment	106
7.6	PMVT: Test Concretization and Execution Environment	107
7.7	RASEN: PMVT guided by Risk Assessment	112
8.1	Medipedia Services Architecture	121
9.1	PMVT Model Inference Process Using a Proxy	135
9.2	PMVT Model Inference Process Using Selenium IDE	136
A.1	Syntax of the Test Purpose Language	148
A.2	Test Pattern for multistep XSS attacks	149
A.3	Test Pattern of Error-Based SQL Injection attacks	149
A.4	Test Pattern for Time-Based SQL Injection attacks	150
A.5	Test Pattern for Boolean-Based SQL Injection attacks	150
A.6	Test Pattern for Cross-Site Request Forgery attacks	151
A.7	Test Pattern of Privilege Escalation attacks	151

LIST OF TABLES

6.1	Test Purpose for Cross-Site Scripting	87
6.2	Test Purpose for Error-based SQL Injections	89
6.3	Test purpose for Time Delay SQL Injections	91
6.4	Test purpose for Boolean-based SQL Injections	92
6.5	Test Purpose for Cross-Site Request Forgeries	95
6.6	Test Purpose for Privilege Escalation of Pages	97
6.7	Test Purpose for Privilege Escalation of Actions	98
8.1	Wackopicko: Experiment Results	124
8.2	Cuiteur: Experiment Results	125
8.3	Bookshop: Experiment Results	126
8.4	Medipedia: Experiment Results	127
8.5	Stud-e: Experiment Results	128
9.1	Test Purpose combining PMVT and Scanners for All Injections	137
9.2	Test Purpose for Insecure Direct Object References in Pages	139

OVERVIEW OF RESEARCH

Contents

1.1 Cybersecurity threats	1
1.2 Motivation and Research Objectives	2
1.3 Research Scope	4
1.4 Contributions of the Thesis	5
1.4.1 Domain-Specific Modeling Language	6
1.4.2 Vulnerability Test Pattern Formalization	7
1.4.3 Pattern-driven and Model-based Vulnerability Testing process	8
1.4.4 toolchain	8
1.4.5 Experimentation	9
1.4.6 Contribution within the FP7 RASEN Project	9
1.5 Dissertation Structure	10

This thesis was conducted within the VESONTIO team, Département Informatique des Systèmes Complexes (DISC) of the Institut Femto-ST¹ of the Université de Franche-Comté (Besançon - France), between October 2012 and October 2015. The work presented in this document has been produced as part of the EU FP7 research project RASEN².

1.1/ CYBERSECURITY THREATS

In recent years, the increase of attacks over the information and communication systems has been mainly focused on the vulnerabilities present in Web applications [65, 16]. Based on the current state of the art on security (see Section 3) and on security reports such as OWASP Top Ten 2013 [76], CWE/SANS 25 [52] and WhiteHat Website Security Statistic Report 2014 [75], Web applications are the most popular targets when speaking of cyberattacks. The fact that modern society's reliance on the Web keeps increasing foregrounds the challenges of IT security, particularly in terms of data privacy, data integrity and service availability. Economically, the digital revolution has resulted in the fast growth of a new buoyant market, greatly welcomed especially in these times of crisis. In France for instance, digital economy in 2014 represented 5.2% of its Gross Domestic

¹<http://www.femto-st.fr/> [Last visited: August 2015]

²<http://www.rasenproject.eu> [Last visited: August 2015]

Product (GDP) and 3.7% of employment³. With financial gain in sight, it is no surprise that organized crime has become the most frequently seen threat actor for Web application attacks [65].

Risks of security breaches inside Web applications have increased over the past ten years. Building a Web application today involves combining an entire mosaic of technologies, both on the client side and the server side. Web applications are becoming more and more complex and ubiquitous, and the need for security of such systems has never been this alarming. As a consequence, a significant growth has been observed in application-level vulnerabilities, with thousands of vulnerabilities detected and disclosed annually in public databases such as MITRE CVE - Common Vulnerabilities and Exposures [52]. The most common vulnerabilities found on these databases especially emphasize the lack of resistance to code injection, but other vulnerability kinds based on the logic of applications are also well represented. Often referred as the **big 4** of the internet, Google, Apple, Facebook and Amazon have all been suffering recently from vulnerabilities inside their services. The iCloud scandal⁴ from 2014, where thousands of people had their private data disclosed, is a vivid example. Another one is the stored Cross-Site Scripting vulnerability inside Facebook Chat/Messenger that allowed hackers to access private messages of other users⁵.

The next section foregrounds the motivation of the thesis by showing that the need for improved security is far from being met, as current vulnerability testing techniques are unstructured, time consuming, and lack precision and accuracy.

1.2/ MOTIVATION AND RESEARCH OBJECTIVES: A PRECISE AND ACCURATE VULNERABILITY TESTING TECHNIQUE

Current techniques to test / counteract / eliminate vulnerabilities are not precise and accurate enough. Indeed, making the Internet a safer place and improving the confidence of users in their ability to use Web applications for actions like purchases and banking is a great challenge.

A widespread “quickfix-style” solution that provides an additional security layer to current Web applications is client-side and server-side **prevention**. Taking the form of a browser functionality (client-side) or an application firewall (server-side), these mechanisms act as guardians that analyze traffic and sanitize or reject any input/request considered as potentially malicious. However, they lack completeness since they miss a lot of vulnerabilities. The complexity of Cross-Site scripting with its near-to-infinite number of variants is a vivid example⁶. In addition, they have poor knowledge of the applications they protect, which is necessary to efficiently filter out malicious requests without encroaching nominal user interactions.

Another technique strongly advised by security consortia⁷ is **defensive programming**: it consists of a set of good practices and habits one should follow during application development. The underlying idea is to never assume anything good concerning user be-

³http://www.justice.gouv.fr/include_htm/pub/rap_cybercriminalite.pdf [Last visited: August 2015]

⁴<http://www.bbc.com/news/technology-29237469> [Last visited: August 2015]

⁵<http://www.breaksec.com/?p=6129> [Last visited: August 2015]

⁶http://www.mediafire.com/view/7a57hv5z25s58lh/WAF_Bypassing_By_RAFAJBALLOCH.pdf Last visited: August 2015]

⁷https://www.owasp.org/index.php/Secure_Coding_Principles [Last visited: August 2015]

haviors and supplied inputs, by writing programs that check their own operations with assertions to restrict inputs and behaviors solely to the ones that are intended. In addition, Web application frameworks nowadays (e.g., Symfony⁸, Django⁹, Rails¹⁰) provide assistance to defensive programming with built-in security features regarding input validation, form manipulation, authentication and authorization mechanisms. The OWASP consortium also provide guidance on how to code in a secure manner.

However, designing an application following defensive programming rules is not enough in practice to prevent vulnerability proliferation because it is subjected to human errors, which by nature are inevitable, and the rising complexity of Web applications hardens the task even more.

Vulnerability proliferation can be overcome by deploying an **application-level vulnerability testing** campaign: it consists of a set of intrusive test cases, each targeting a certain vulnerability type. It may be done manually, automatically, or somewhere in the middle; it can rely on the source code (white-box), on the running application as a whole (black-box) or again, somewhere between the two of them (gray-box). Each technique possesses advantages and drawbacks, be it about its efficiency or its deploying cost (see Chapter 3 for more information on the state of the art about vulnerability testing).

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, for example in **Penetration Testing**. 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. Moreover, such companies work in time boxes, and often have to reduce their detection scope accordingly. Finally, **Web application vulnerability scanners** can be used to semi-automate the detection of vulnerabilities, but they lack precision and accuracy since they have no knowledge of the application's logic. Thereby, they often generate false positive and false negative results, and human investigation is often required [22, 30].

Model-Based Testing (MBT) [68] is a software testing approach in which both test cases and expected results are automatically derived from an abstract model of the system under test (SUT). More precisely, MBT techniques derive abstract test cases (including stimuli and expected outputs) from an MBT model, which formalizes the behavioral aspects of the SUT in the context of its environment and at a given level of abstraction. The test cases generated from such models allow the validation of the functional aspects of the SUT by comparing back-to-back the results observed on the SUT with those specified by the MBT model. 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 [20].

The strong lack of accuracy and precision in current vulnerability testing techniques represents the motivation of this thesis, to which we respond by proposing a Model-Based Vulnerability Testing approach dedicated to Web application vulnerabilities. This translates in the two following research objectives (RO₁ and RO₂):

⁸<https://symfony.com/> Last visited: August 2015]

⁹<https://www.djangoproject.com/> Last visited: August 2015]

¹⁰<http://rubyonrails.org/> Last visited: August 2015]

RO₁: DESIGN AN APPROACH THAT RELIES ON PATTERNS AND MODELS FOR BETTER ACCURACY AND PRECISION OF VULNERABILITY TESTING OF WEB APPLICATIONS

Vulnerability test patterns are used to describe testing procedures for each class of vulnerabilities [62]. Several security consortia, such as Mitre, CAPEC, and OWASP have all issued test patterns for the detection of specific vulnerabilities. Research projects have as well been focused on designing vulnerability test patterns, like the ITEA2 DIAMOND project [70]. However, such patterns are informal and applying them remains manual. Using vulnerability test patterns for automated testing still remains a challenge.

Our objective is to formalize vulnerability test patterns into a machine-readable language that would enable to automatically generate test cases dedicated to the detection of the targeted vulnerability. To achieve this, we propose to combine formalized test patterns with a model of the Web application under test, in order to design a Model-Based Vulnerability Testing technique for the detection of Web application vulnerabilities. This way, we combine the accuracy of patterns and the precision of the model to improve the overall detection of vulnerabilities.

RO₂: DESIGN AN APPROACH THAT IS USABLE, SCALABLE, EFFICIENT, AND WITH A GOOD LEVEL OF AUTOMATION

Model-Based Testing techniques generally suffer of the high cost associated to the design of models and test case concretization. Indeed, test cases generated from a model are abstract, since the model is an abstraction of the system. It is the responsibility of the test engineer to design the model and translate all test cases into executable scripts.

Moreover, Web application development is a fast-paced activity where time is essential. Therefore, there is a strong need for a usable, scalable and efficient testing approach, and automated as much as possible.

Our objective is to automate most activities of such approach for vulnerability testing purposes, by narrowing the information contained in the model and generify formalized test patterns. In addition, MBT techniques have proven to be scalable and efficient testing approaches deployed for critical and complex systems.

These two RO raise 3 research questions, which we express in the next section.

1.3/ RESEARCH SCOPE

The purpose of this thesis is to investigate the main problem **What is the effectiveness and efficiency of pattern-based and model-based approaches such as the one we are presenting for detecting Web vulnerabilities?** This research challenge can be broken down to 3 research questions that are expressed and developed below. We refer to these questions in Section 8.5 to evaluate the experimentation results.

RQ1 To what extent test patterns applied to a model of the Web application under test improves the accuracy and precision of vulnerability detection?

On the one hand, penetration testing is becoming more and more difficult as Web sites are growing in size and complexity. For instance, the OWASP foundation recommends

that each user input of a Web application should be tested for XSS using a list of a hundred XSS attack vectors¹¹. Considering a basic Web form composed of ten fields, a thorough validation against XSS attacks would represent approximately 1000 tests, which is clearly not realistic with manual penetration testing.

On the other hand, current automated vulnerability discovery techniques such as Web application scanners (e.g., AppScan, Acunetix, and so forth) can test for a large percentage of technical vulnerabilities. However, they cannot test 100 percent of them since they usually have issues when dealing with specific cases that require *intelligence*, like infinite Web sites with random URL-based session IDs, or automated form submission. Moreover, automated techniques also have issues to establish a verdict from a test case execution, and as a result they often produce many false positives.

One research axis of this thesis is to determine if a pattern-driven and model-based testing approach can ally advantages of both manual and automated techniques, by modeling certain functional aspects of a System Under Test (SUT), especially navigational information, to generate test cases that current techniques struggle to obtain.

RQ2 To what extent is it possible to provide generic test patterns for Web application vulnerabilities?

A vulnerability test pattern is a formalization of a generic test procedure that aims to test for a certain vulnerability. In the case of XSS/SQL Injections, it can be as simple as *insert this malicious vector in this area to observe this behavior*. Or it may involve more complex contrivances like forging a Web-form to highlight a CSRF (Cross-Site Request Forgery) vulnerability, or browse-forcing parts of the Web application that are supposed to be protected in the case of Privilege Escalation.

Each vulnerability type has a rather unique test pattern, and another aspect of this thesis is about establishing if test patterns can be represented as test purposes and applied on a model and interpreted by automated test generation engines.

RQ3 To what extent such Web application vulnerability testing process (based on patterns and models) may be automated?

This is one of the most attractive properties of MBT in general: its capacity for automating test generation and execution. However, MBT techniques generally require human intervention to provide a model of the system under test, and adapt the generated abstract test cases to make them executable on the real system. Given the specificity of our context, which concerns vulnerability testing for Web applications, we need to find out the level of automation that can be reached for the major steps of such process, namely model design, the application of generic patterns on these models, test generation, test execution, and verdict assignment and reporting.

1.4/ CONTRIBUTIONS OF THE THESIS

This thesis proposes a novel and innovative security testing approach, called **PMVT** for Pattern-driven and Model-based Vulnerability Testing, to address Web application vulnerabilities. The PMVT approach aims to improve the **accuracy** and **precision** of Web application vulnerability detection by proposing a test generation and execution technique

¹¹[https://www.owasp.org/index.php/XSS_\(Cross_Site_Scripting\)_Prevention_Cheat_Sheet](https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet) [Last visited: August 2015]

driven by automated **vulnerability test patterns** composed with a **model** (called in the sequel PMVT model) of the system under test. Patterns describe generic test scenarios that assess the robustness of the Web application w.r.t. a given kind of vulnerability. Test generation is achieved by relying on the information contained in PMVT models, 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, the PMVT approach increases the efficiency of penetration testers when detecting technical as well as logical vulnerabilities.

To summarize, the main 5 contributions of this thesis are:

- The design of a Web application modeling language, called DASTML, dedicated to vulnerability testing,
- The formalization of textual vulnerability test patterns into operational generic test purposes and a contribution to the extension of the test purpose language,
- The creation of a model-based testing process, called PMVT, which composes formalized generic patterns and PMVT models for the computation of vulnerability test cases,
- The engineering of a prototype toolchain that implements the PMVT approach and instruments its activities,
- The experimentation of the PMVT toolchain on 5 case studies to evaluate the effectiveness and efficiency of the PMVT approach.

The following sections of this chapter present each contribution of the thesis. Then we provide a short description of the research context, the European FP7 research project RASEN¹², and the story of how the PMVT approach was designed to be one of the key elements of a security assessment tool suite.

1.4.1/ DOMAIN-SPECIFIC MODELING LANGUAGE

The PMVT process is an extension of MBT and as such, the modeling activity consists of designing a PMVT model of the Web application under test that will be used to automatically compute abstract vulnerability test cases. This activity is based on CertifyIt [45], an MBT toolchain designed by Smartesting¹³, which requires a model designed using the UML4MBT notation [68, 13], a subset of UML / OCL introduced in Section 4.1.

One of the main constraints in MBT is the design of models, which is known to be time consuming. To address this issue, a Domain Specific Modeling Language (DSML) has been developed, dubbed **DASTML** for Dynamic Application Security Testing Modeling Language. It allows the modeling of 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 modeling environment.

¹²<http://www.rasenproject.eu> [Last visited: August 2015]

¹³<http://smartesting.com/> [Last visited: August 2015]

More precisely, DASTML is composed of 4 *entities* that match either an element or a concept related to Web applications. *Page entities* represent the different pages that compose the Web application under test. *Navigation entities* are 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 entities* are similar to navigation entities, except that they alter the internal state of the application and may carry data (e.g, in case of a Web form or a parameterized URL). *Data entities* represent user inputs and are each composed of a key and a value.

Entities interact with each other based on multiple *relationships*. For example, pages where each user input is rendered back are known thanks to a relation between inputs and pages entities.

The information contained in PMVT models is crucial, not only for conducting attacks, but also for assessing whether these attacks succeeded or failed. An in-depth breakdown of the metamodel ¹⁴ of DASTML and a thorough description of each entity is given in Section 5.3.

1.4.2/ VULNERABILITY TEST PATTERN FORMALIZATION

A Vulnerability Test Pattern is a normalized textual document describing the testing objective and procedure to detect a particular flaw in systems of a similar nature (e.g., Web applications). As such, there are as many patterns as there are types of application-level flaws. The approach presented in this thesis is based on pattern catalogues provided by dedicated organizations, for instance OWASP and SANS¹⁵, as well as research projects such as the ITEA2 research project DIAMONDS¹⁶ [70]. Nonetheless, Vulnerability Test Patterns are still textual and cannot be processed automatically.

One motivation of this thesis is centered on using textual patterns as starting point and formalizing them into operational *test purposes*, in order to automate testing strategies' implementation and execution. A *test purpose* is a high-level expression that formalizes a testing objective to drive the automated test generation on the PMVT model. Test purposes are written in a dedicated language, called *Smartesting Test Purpose Language*. It has been originally designed to drive model-based test generation for security components, typically Smart card applications and cryptographic components [11]. This language has been augmented in order to make test purposes generic, and thus enable the translation of vulnerability test patterns. Thereby, within the context of vulnerability testing, a test purpose formalizes a given textual pattern in order to drive the vulnerability test generation on the PMVT 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 of a test purpose is a request, such as calling an operation, activating a behavior or reaching a state.

As a result, multiple test purposes have been designed to tackle major vulnerabilities from the OWASP Top 10, that is to say Cross-Site Scripting, SQL Injection, Cross-Site Request Forgeries, and Privilege Escalation. A detailed presentation of each vulnerability lies in Chapter 2, and their associated test purposes are depicted in Chapter 6.

¹⁴<http://dictionary.reference.com/browse/metamodel> Last visited[August 2015]

¹⁵<https://www.sans.org/> [Last visited: August 2015]

¹⁶<http://www.itea2-diamonds.org> [Last visited: August 2015]

1.4.3/ PATTERN-DRIVEN AND MODEL-BASED VULNERABILITY TESTING PROCESS

Both test purposes and PMVT models are integrated in a model-based testing process dedicated to the generation of vulnerability test cases for Web applications.

The PMVT process is a composition of 4 activities:

The first activity concerns the modeling of the Web application using the DASTML notation. This is done by exploring the application and collecting relevant information in the model, such as the various pages or states, the possible user interactions within these pages, as well as user inputs in Web forms, anchors, cookies and HTTP headers.

The second activity consists of the design and / or selection of **Test Purposes** to cover specific vulnerabilities. PMVT test purposes are generic and can be applied on any Web application that has been modeled using the DASTML notation. It is thus possible to select existing test purposes, such as the ones presented in this thesis, or design new ones tailored to another vulnerability.

The third activity is about applying the selected test purposes on the model to automatically generate vulnerability abstract test cases. The test generation engine uses test purposes as guidance to animate the model for the computation of attack traces.

Finally, the last activity involves the concretization of the generated test cases and their automated execution on the real application, along with an automated verdict assignment.

Each activity of the process is described in Section 5.1.

1.4.4/ TOOLCHAIN

A prototype toolchain that supports the PMVT process has been developed as part of this thesis to conduct experiments, and thus assess the validity of the approach. It is built on top of the model-based testing software *CertifyIt* [8, 45], provided by the company Smartesting¹⁷. *CertifyIt* is a test generator that takes as input a test model, written with UML4MBT, which represents the behavior of the SUT to compute abstract test cases.

The PMVT toolchain embeds several tools to enable test engineers to complete each activity of the process. It relies on IBM Rational Software Architect¹⁸ for the modeling and test purpose design/selection activities. The RSA modeler that initially allows UML modeling is augmented with *CertifyIt* plugins that enable the design of UML4MBT models and test purposes. An additional plugin, developed as part of this thesis, allows the creation of DASTML model as well as their transformation in a UML4MBT model.

The *CertifyIt* test generation engine is responsible for composing test purposes and models to compute abstract vulnerability test cases. A second dedicated PMVT algorithm, plugged to the test generation engine, exports the abstract test cases in JUnit test scripts integrated to a Mavenized¹⁹ Java project.

Each JUnit test script contains a sequence of Selenium²⁰ primitives (e.g., load a Web page, click on a link, fill a form field, etc.) that emulate a headless Web browser and reproduce user actions to conduct attacks. Note that test engineers must provide a table that matches each abstract data from the model and the concrete data from the real

¹⁷<http://www.smartesting.com> [Last visited: August 2015]

¹⁸<https://www.ibm.com/developerworks/downloads/r/architect/> [Last visited: August 2015]

¹⁹<http://maven.apache.org/> [Last visited: August 2015]

²⁰<http://www.seleniumhq.org/> [Last visited: August 2015]

application, and in some cases may have to adapt Selenium primitives.

1.4.5/ EXPERIMENTATION

Experiments have been conducted on 5 Web applications. Three of them are dummy applications that have been made vulnerable for experimentation purposes: (i) **Wackopicko** is an open-source photo-sharing platform developed by Adam Doupé [5], (ii) **Cuiteur** and (iii) **Bookshop** are clones of Twitter and Amazon respectively, and have been initially developed at the Institut FEMTO-ST for a Web development course. The last two Web applications are real-life systems, both online at the moment of writing, with several thousands of users: (iv) a Romanian eHealth portal called **Medipedia** (which is also used for experimentation as part of the FP7 RASEN project, see Section 1.4.6 for more information), and (v) a French eLearning portal called **stud-E** (due to confidentiality issues, we chose to change its name).

To evaluate the robustness of the real-life case studies and assess the efficiency and effectiveness of the PMVT approach, test cases were generated to tackle 4 of the most commonly exploited Web application Vulnerabilities: Cross-Site Scripting, SQL Injections, Cross-Site Request Forgeries, and Privilege Escalation vulnerabilities (see Chapter 2 for a presentation of each vulnerability type).

Chapter 7 provides a full description of the toolchain and Chapter 8 presents and discusses experimentation results of the PMVT toolchain on the 5 Web applications.

In the following section, we introduce the European research project RASEN in which this work has been conducted.

1.4.6/ CONTRIBUTION WITHIN THE FP7 RASEN PROJECT

The PMVT approach presented in this document has been designed in the context of the EU FP7 research project RASEN²¹. This collaborative research project, driven by SINTEF ICT, has been underway since October 2012 and will complete in October 2015. It addresses risk assessment, legal compliance, and testing within the area of cybersecurity. To achieve this goal, the project brings together 8 partners from 4 European countries (Germany, Norway, France, and Romania). The partners fill one or more of the 3 possible roles in the RASEN project:

- **Research Partners:** Fraunhofer FOKUS, Université de Franche-Comté / Femto-ST, SINTEF, University of Oslo.
- **Technology Providers:** Smartesting, Software AG.
- **Use Case Providers:** Evry, Info World, Software AG.

The RASEN project develops an approach that allows organizations to conduct security risk assessments for large-scale networked systems and verify the assessment by means of security testing.

²¹<http://www.rasenproject.eu> [Last visited: August 2015]

The RASEN project addresses cybersecurity issues by making interrelated and synergistic risk management activity and security testing. On the one hand, it provides support for deriving test cases from risk assessment results, and on the other hand, it proposes to use the related test results to verify or update the risk picture and therefore risk assessment.

In a more concrete way, the RASEN project addresses these challenges by means of 3 innovations. The *first innovation* [60] is a method for risk-based security testing and legal compliance assessment, which provides a unified approach toward cybersecurity that addresses both technical and non-technical issues across different levels in the organizational hierarchy. The *second innovation* [69] is a tool for risk management, called RACOMAT, which combines risk-based security testing with test-based risk assessment into a unified iterative process. The *third innovation* [46] is an application-specific risk-based security testing approach that derives test cases from risk assessment results and test patterns.

The PMVT test generation technique, based on PMVT models and driven by test patterns, is the keystone of the third innovation, which combines the PMVT approach with the CORAS method[49], a risk assessment technique based on models. The principle of this combination is to use the CORAS risk assessment results to guide the PMVT test generation process by selecting and prioritizing test patterns related to the identified risks. It also provide traceability between risks and generated test cases to get feedback from test execution and to highlight which risks revealed to be real vulnerabilities.

1.5/ DISSERTATION STRUCTURE

This thesis dissertation is organized as follows.

The first 4 chapters lay down the context, motivation, and background of the thesis. We expose the research objectives, the research questions and the contributions in Chapter 1. Then, Chapter 2 defines the 4 targeted vulnerability types that are tackled by the proposed approach. Subsequently, Chapter 3 is dedicated to the state of the art and practice about Web application security testing. Finally, Chapter 4 presents the background work on top of which the PMVT approach is built.

In the next 3 chapters, we detail the technical contributions of the thesis. Chapter 5 describes the overall PMVT Process and the modeling notation. Chapter 6 the test pattern formalization into a machine-readable language and its instantiation to address the 4 vulnerability types. We present the prototype toolchain that supports the PMVT process in Chapter 7, along with its general work-flow.

Chapter 8 focuses on experimentation and evaluation. We describe the 5 case studies we included to demonstrate the validity of the experimentation results, which we present and discuss in depth in this chapter.

This document concludes with Chapter 9. We summarize the work completed during this thesis and discuss future works based on what has been achieved.

PRESENTATION OF THE TARGETED VULNERABILITIES

Contents

2.1 Cross-Site Scripting	12
2.2 SQL Injections	15
2.3 Cross-Site Request Forgery	18
2.4 Privilege Escalation	21

Web applications present all the risks that threaten normal applications: compromising, information leak, reputation damage, information and money loss. A standard Web application relies on 3 artifacts: the client, the Web server, and the storage unit. In general, the client is a Web browser (e.g., Firefox, Chrome, IE), manually handled by a physical person. The Web server (e.g., Apache, IIS, nginx) receives requests sent by the client and issues responses. The communication between them takes place using the Hypertext Transfer Protocol (HTTP). The Web server's end goal is to deliver Web pages to the client. The storage back-end is usually a database (e.g., MySQL, PostgreSQL, MongoDB). Its purpose is to store data that can be retrieved later, at will.

A typical work-flow involving the 3 artifacts would consists of the client sending an HTTP request to the Web server that contains data meant to be stored, for instance a blog article. The Web server receives and processes the request, passing the attached data to the storage back-end for storage. Some time later, in another context, the client issues a new HTTP request, whose purpose is to receive the previously sent data. The Web server processes the request, queries the database to retrieve the relevant data, uses it to compute an HTTP response and sends it to the client.

All these components can present different vulnerabilities and security issues, and may have different behaviors that will impact the existence and exploitability of vulnerability. For instance, if a careless user clicks on a malicious link and downloads a Trojan on his computer, or installs a malicious extension on his Web browser, the hacker behind the trap can steal the victim's credentials and compromise any system that the victim has access. As another example, if the database of a Web application has not been updated to the last version, and if the form fields of this application are not protected against SQL Injections, by sending the correct sequence of attack vectors a miscreant can get access to the root shell of the Web server.

The purpose of this section is to lay down all the vulnerability types that are tackled by the PMVT approach. These vulnerability types have been chosen specifically based on their

high severity and prevalence, and their complex detectability. Each type is provided with explanations on its workings and is illustrated with a usage example. We start with Cross-Site Scripting, a threatening vulnerability because of its almost-infinite attack vector list. It continues with the most prevalent vulnerability of all according to the OWASP TOP 10, SQL Injections. Then it highlights Cross-Site Request Forgeries, a direct consequence of the statelessness simulation of the Web by browsers. Finally, we end this section with Privilege Escalation vulnerabilities, which are closely related to the logic of Web applications, thus complexifying their detection.

2.1/ CROSS-SITE SCRIPTING

It is quite common for one to receive emails, sometimes from trusted acquaintances, requiring to click on a given link leading to a legitimate Website (e.g., by advertising a sales promotion). However, if one happens to click on the link he/she might become victim of session stealing, or get redirected to a malicious Website.

This popular hacking phenomena is known as Cross-Site Scripting (XSS). XSS represents a great part of the most prevalent and dangerous cyber-attacks against Web applications reported during the last decade; see, for example, OWASP Top Ten 2013 [76], CWE/SANS 25 [52] and WhiteHat Website Security Statistic Report 2014 [75]. In the latter, XSS appears to represent between 35% and 67% (depending on the programming language) 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 vulnerabilities¹. XSS was first disclosed in 1996 during the debut of the Internet, when the first e-stores came online, when the most enhanced Web applications were using HTML frames, and most importantly when Javascript was released. This dynamic programming language revolutionized the Web as people knew it, and developers were suddenly able to create dynamic and interactive applications with floating menus, pop-ups, and content altering without the need to refresh the current page. It also received a great welcome from the hacker community who saw in this language new ways to tamper with Web applications.

An XSS vulnerability occurs each time an application stores (with more or less persistence) a user input and uses it to compute an output without proper sanitation. When injecting a malicious input, such as a piece of code, it is therefore possible to have it executed by Web browsers. Indeed, if this data is displayed to all the visitors of a Website (e.g., in a reply of a forum post), then the code is executed on each visitor's browser, potentially causing severe damages (and not necessarily visible). XSS is easy to put into practice and presents a great number of variants; it is also an entry point for many exploits. For example, attackers can steal session credentials and hijack an active session, access sensitive or restricted information, or spy on a user's Web browsing habits.

XSS vulnerabilities can be classified into 4 categories²:

When the untrusted injected data (the attack vector) is directly displayed/executed after being injected, we speak of **reflected XSS**, and the response containing the attack vector

¹<https://developers.google.com/google-test-automation-conference/2013/presentations>[Last visited: August 2015]

²https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_%28XSS%29 [Last visited: February 2014]

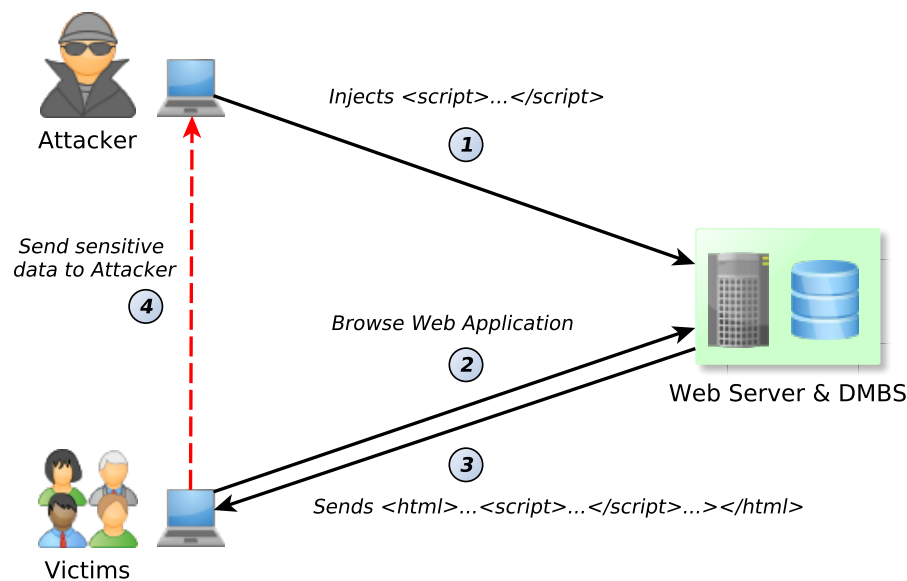


Figure 2.1 – Stored XSS typical workflow

is immediately produced and sent back to the user.

On the opposite, in **stored XSS** vulnerabilities the attack vector is stored (no matter how) by the application and retrieved later in another context.

A special case of XSS, called **multi-step XSS**, requires a user to perform several actions on the applications (mainly navigation steps) to display/execute the attack vector. In this thesis, we highlight two types of multi-step XSS: multi-step in a testing, and multi-step in the conducting of the attack. The first type concerns XSS attacks, reflected or stored, that are not observable in the immediate server response, but take place in another part of the application, or in another context (e.g., requiring to authenticate with another user role). The second type encompasses XSS attacks that require several steps between the injection and the actual storage of the payload in the DMBS of the application. Notice that these two groups are not exclusive, and certain complex XSS attacks can be of both types.

The last category, named **DOM-based XSS**, works at the level of the victim's browser by injecting the attack vector in the URL, replacing a parameter used by a local script to modify the DOM³ of a Web page.

The difficulty of handling XSS issues is mainly due to the increasing complexity of applications' logic: developers need to think of a systematic protection of the displayed data, which is an error-prone exercise as a given user input may be subsequently displayed in a large variety of places in the application.

STORED XSS: USAGE EXAMPLE

As an example for stored XSS, we consider a forum board where users can hold conversations in the form of posted messages. Creating a new topic or posting a reply is done through a specific page containing an HTML form with input fields. Upon submis-

³<http://www.w3.org/TR/WD-DOM/introduction.html> [Last visited: August 2015]

sion, either a new discussion thread is created or the posted message is appended to the thread. We also consider a miscreant that has detected a stored XSS vulnerability in the “body” input field of the form responsible for new messages. The server-side and client-side source code of this page is shown in Figure 2.2.

A typical stored XSS scenario is depicted in figure 2.1. On step ①, the attacker sets the trap by injecting a malicious script inside the “body” field input of the message posting form:

```
<SCRIPT>
document.location='http://evil.com/store-sid.php?sid='+document.cookie
</SCRIPT>
```

This script, which will be referred as {payload} from now on, redirects browsers to the URL “evil.com” using the document.location method. It also appends the content of the document.cookie variable, which contains the session ID of the victim.

<pre><?php // DB query for message posts foreach(\$msg in \$messages) { echo('<div><p>'. \$msg['title']. ' </p>', '<p>'. \$msg['body']. ' </p>', '<div/>'); } ?></pre>	<pre><html> [...] <div> <p>You will never believe this</p> <p> {payload} </p></div> [...] </html></pre>
--	---

(a) server-side code

(b) client-side code

Figure 2.2 – XSS usage example: server-side and client-side code

User inputs are not sanitized by the server, which sends the raw data to the DBMS for storage. Then, in step ②, a curious user decides to visit the topic created by the attacker. The Webserver makes a request to the DBMS, retrieves the payload, inserts it on the output document, and sends the response to the user (as shown on step ③). Code fragments from the server-side and client-side, as depicted in figure 2.2, show that both input values *title* and *body* are not sanitized and rendered back as is. Step ④, the victim's browser interprets the DOM and executes the malicious script, resulting in the disclosure of their session ID. The attacker can therefore spoof requests using a received ID and impersonate victims.

COMPLEXITY OF TESTING

The 1st, 2nd and 4th categories of XSS are usually well-identified and easily detected by current vulnerability detection techniques (e.g., Web application vulnerability scanners, see Section 3.2.1).

However, the 3rd category, which concerns multi-step XSS, represents a challenging issue [22]. Indeed, the result of an attack cannot be seen immediately, and the applications' logic must be taken into account to know in which part of the application and in which context a given user input is supposed to be sent back to the client.

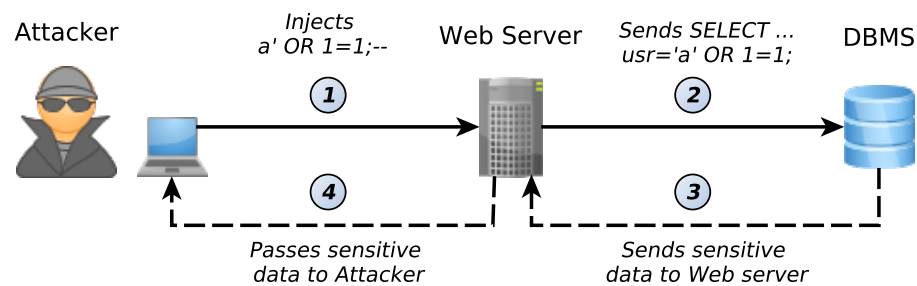


Figure 2.3 – SQL Injection: Typical Workflow

2.2/ SQL INJECTIONS

Injections (such as SQL, but also OS, LDAP⁴, and so on) are considered the most threatening and prevalent vulnerability type. It has always been on top of the OWASP TOP 10 since its first version, published in 2003. Indeed, injection vulnerabilities can be extremely severe: sensitive data read, removal or corruption, authentication bypass, identity spoof, denial of access, and in some cases host takeover. They are also very common since they may be present each time data with inadequate validation is interpreted. Last but not least, it is quite straightforward for an attacker to conduct an injection vulnerability. During this thesis, we focused on SQL Injections.

SQL is a special-purpose programming language initially developed by IBM in the early 1970s. There are many varieties of SQL, but today the most commonly used are based on the ISO/IEC 9075:2011 standard⁵. The most commonly used variants are MySQL, Oracle, PostgreSQL, Microsoft SQL, etc. Each variant comes with subtleties in its syntax, giving plenty of opportunities for hackers to find targeted injector vectors that might be missed by generic protection mechanisms.

SQL Injections were mentioned for the first time in 1998 by Rain.Forest.Puppy⁶ (a hacker, security consultant, and author of the RFPOLICY, a method of contacting vendors about security vulnerabilities found in their products) in a Phrack article⁷. His conclusion about SQL Injections became a popular quotation: “don’t assume user’s input is OK for SQL queries”. Even today, whereas SQL Injections have been around for almost 20 years and a lot of protection mechanisms have been designed, recent reports show that the number of exploits from SQL Injections is still alarming. In a report from 2014 [48], Ponemon Institute LLC expresses that 65% of organizations participating in the study experienced an SQL Injection attack that successfully evaded their perimeter defenses in the last 12 months.

Very much like XSS, SQL Injection attacks exploit the trust applications have in their users. They take place when data coming from an untrusted source enters an application

⁴<https://tools.ietf.org/html/rfc4510> [Last visited: August 2015]

⁵<http://www.iso.org/iso/home/search.htm?qt=9075&sort=rel&type=simple&published=on> [Last visited: August 2015]

⁶<http://lists.jammed.com/ISN/2001/10/0032.html> [Last visited: August 2015]

⁷<http://www.phrack.org/archives/issues/54/8.txt> [Last visited: August 2015]

(e.g., through a form input field), and is used to construct an SQL query, for instance untrusted data injected into data-plane input and sent to an SQL interpreter. By supplying an SQL code fragment instead of a nominal value as input, with respect to the syntax of the initial SQL query, hackers can alter the semantic of the request. As a consequence, the database server is tricked into running an arbitrary, unauthorized, unintended SQL query that implies unwanted effects on the integrity of its data.

Examples of SQL Injection exploits, such as the one presented in Figure 2.3 and discussed later in the next section, are anything but rare⁸. In 2011, various Websites owned by Sony (sonymusics, sonypictures) were compromised by LulzSec, a black hat computer hacker group. They managed to dump the entire database and disclosed its content to the Internet⁹. In December of 2014 archos.com, the Website of the French smartphone maker Archos, was compromised by an SQL Injection attack conducted by a hacking group known as “Focus”¹⁰. They claimed to have dumped 100,000 customer records.

SQL INJECTION: USAGE EXAMPLE

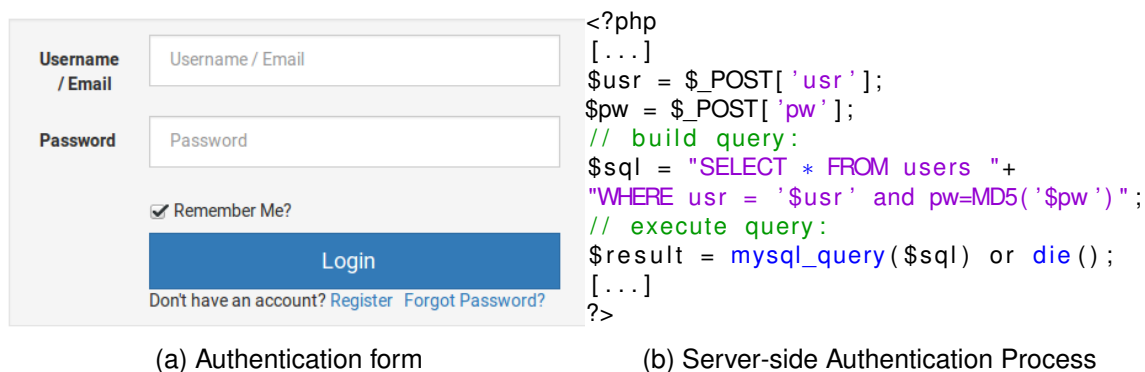


Figure 2.4 – Authentication: Server-side and Client-side

To illustrate SQL Injections we consider an eLearning Web portal, where users have access to various courses. They can download course material, do exercises and quizzes, pass exams, send messages to their teachers and fellow students, and so on. To get access to the platform, students must authenticate to the portal by providing their credentials (username and password).

We present below the syntax of the SQL query, whose purpose is to verify whether a user entry matches the provided username and password:

```
SELECT * FROM users WHERE usr= '$usr' and pw=MD5( '$pw' );
```

When students provide their credentials using the authentication form (see Figure 2.4a), for instance “john” as username and “@doedoe1!” as password, the server configures the query by replacing \$usr and \$pw by their corresponding value and submits the crafted request to the DBMS for interpretation (see Figure 2.4b). The problem in the server

⁸<http://codecurmudgeon.com/wp/sql-injection-hall-of-shame/> [Last visited: August 2015]

⁹<http://www.pcmag.com/article2/0,2817,2386362,00.asp> [Last visited: August 2015]

¹⁰<http://www.scmagazineuk.com/up-to-100k-archos-customers-compromised-by-sql-injection-attack/article/395642/> [Last visited: August 2015]

implementation comes from the lack of input sanitation: the received values are passed on raw to the DBMS. As result, attackers can supply an SQL fragment as input and alter the initial semantics of the query. A general scenario for SQL Injection is depicted in Figure 2.3.

For example, if a miscreant injects `a' or 1=1;--` as login and a random value as password (step ①), it changes the query:

```
SELECT * FROM users WHERE usr='a' or 1=1;-- and pw=MD5('$pw');
```

The SQL fragment injected by the miscreant is composed of 3 distinct parts:

- `a'`: the purpose of the first part is to respect the SQL syntax imposed by the preamble of the query. It provides a value, a, and closes the Boolean equality test with a simple quote;
- `or 1=1`: this is the part that modifies the semantic of the query by changing the comparison predicate of the “WHERE” clause. Because `1=1` is always true, when coupled with a “OR” operator it cancels “WHERE” clause.
- `;-`: this last part closes the query and comment the rest of the initial request, to preserve the syntactic correctness of the request.

The conduction of this attack is depicted in Figure 2.3. Upon reception of the credentials, the Webserver configures the SQL query and passes it on the DBMS (step ② in the figure). The injection of this particular payload forces the DBMS to retrieve the whole content of the “users” table (step ③ in the figure). As a result, depending on how the server treats the resulting data, the miscreant may be able to bypass the authentication form and log in using the credentials of the first users entry (step ④ in the figure). It often corresponds to the first user registered, therefore the probabilities that this user holds extensive permissions (e.g., a teacher or worse, an administrator) on the application are high.

COMPLEXITY OF TESTING

Detecting SQL Injection vulnerabilities, like Cross-Site Scripting, can be performed by injecting attack vectors through user input and analyzing the server’s response. In some cases, it may be very simple to fingerprint a database to unveil a vulnerability; by supplying as input a certain attack vector that will affect the syntactic correctness of the initial SQL query, it is possible to make the execution of the request fail on the back end database, which will generate an SQL exception. The evidence of an SQL exception error is often a manifestation of a vulnerability that can be exploited. Most automated tools are able to detect these vulnerabilities with a high level of confidence. In other cases though, it can be much more complex: when the back end database does not generate exception errors. There is no technical telltale on whether the request was interpreted, and one must resort to other techniques while being “blindfolded”. These vulnerabilities are referred as **Blind SQL Injections**.

When it comes to Blind SQL Injections, fingerprinting techniques consists of performing several attacks directed toward the same input, each one with a different goal, to seek for variations in the server’s responses. For example, time-delay techniques compare the

server's response time between two injections. One attack vector will compromise the syntax of the query, resulting in a fast response from the server since it does not involve any search over the database. The second attack vector, on the other end, will alter the request to return as many entries as possible: the end-goal is to force processing time. Observation is done by comparing response times; a big enough time gap is a good indication of the presence of an SQL Injection vulnerability. Another technique compares the nature of the responses, and generally involves 3 injection vectors. The first one is a nominal input that triggers the intended behavior of the application. The second one tries to change the semantic of the request to return as much entries as possible. The third vector is the opposite: it tries to make the request return no entry. A difference in the outputs is an indicator of the presence of a vulnerability. The detection of Blind SQL vulnerabilities is a source of trouble for automated tools because identifying variations in responses often implies understanding the workings and the logic of a Web application, thus requiring human intervention.

Along the lines of XSS, SQL Injections can be *multi-step* (also mentioned as "second order SQL Injections"), when injections are not immediately sent to the DBMS, but require specific user actions from the injection page to reach the actual execution of the query. Automated tools struggle to detect these vulnerabilities because, once again, they are generic and hence, not aware of the logic of Web application [7].

2.3/ CROSS-SITE REQUEST FORGERY

Cross-Site Request Forgery (CSRF) has often been an underestimated vulnerability, but it is well present in the OWASP TOP 10 since 2007, being ranked 5th in 2007 and 2010, and 8th in 2013. Although protecting Web applications against this vulnerability is straightforward, many CSRF attacks are still reported on a regular basis. The fact is that CSRF attacks remain in the shadow of the most common, highest profile vulnerabilities like Cross-Site Scripting and SQL Injections. As opposed to Cross-Site Scripting attacks that exploit the trust a user (or rather his Web-browser) has in a Web application, Cross-Site Request Forgery attacks exploit the trust a Web application has in its users. However, exploiting a CSRF vulnerability needs only basic knowledge of the targeted Web application to quickly identify relevant actions and victims to trick (e.g., by using phishing). During the last decade, lots of Web applications have been compromised by CSRF attacks. The most explicit example is the "Samy Worm" [42], which in fact is a combination of an XSS vulnerability with a CSRF vulnerability. This worm infected millions of Myspace accounts in less than a day. Several other major Web applications have been compromised like Ebay, Youtube and INGDirect.

Actions of a Web application (GET and POST requests) are usually linked to specific URLs (e.g., <http://bank.com/transfer.php?amount=10000&receiver=42982875983>). A direct request to such a URL allows its associated action to be performed. A CSRF attack consists of tricking a victim into making a specific request through his/her browser, that will ultimately lead to unwanted consequences on a trusted Web application. It is qualified as malicious because it indirectly impersonates a user to perform actions only he/she or a restricted group of users is allowed to do, and without him/her knowing. A typical action would be to modify a user's contact email, his/her password, or add items to a shopping cart and even activate the payment if the user has stored his/her credit card's info. That is, CSRF attacks target actions that modify the internal state of the Web

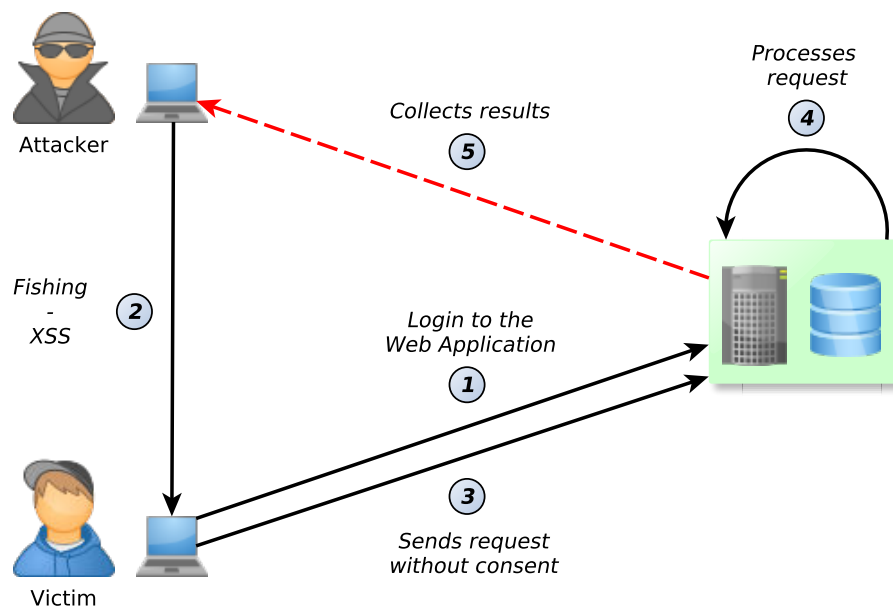


Figure 2.5 – CSRF: Typical Workflow

application, but they can also be conducted to access sensitive data. As shown by the previous example, major targets are social networks, Webmail clients, banking Websites, etc.

CSRF attacks are possible when the targeted Web application does not check whether an incoming request is really originating from the user owning the active session. More precisely, there are 3 different ways for a user to send a GET or POST request¹¹ to a Web application: (i) from the application, by clicking on a link or submitting a form, (ii) from the outside, by clicking on a link on a Website that points the URL responsible for the request, and (iii) by manually typing the URL in the browser's navigation bar. If the Web application cannot determine how the request has been made, then it is vulnerable to CSRF.

CSRF: USAGE EXAMPLE

We consider in this example a banking Web portal where users can manage their account: check their balance, go through recent transactions, and transfer money to other accounts. One common mistake made by the developers of such portal is the absence of URL rewriting, resulting in the disclosure of sensitive actions. For example, the URL that triggers a money transfer from the current session to the account n° 28728472647 has the following structure:

```
https://www.bank.com/money_transfer.asp?amount=1000&target_acc=28728472647
```

It should be noted that actions being displayed in URLs is not a prerequisite for the presence of a CSRF vulnerability. Hackers can record traffic between their browser and Web application to deduce URLs and key values.

¹¹http://www.w3schools.com/tags/ref_httpmethods.asp [Last visited: August 2015]

A hacker has detected the vulnerability and decided to deploy a scam in order to steal money from other accounts, by forcing clients to make the transfer themselves without their consent.

In step ① (as shown on Figure 2.5), the future victim authenticates to the vulnerable Web portal. Although authentication is not always mandatory to exploit a CSRF vulnerability, sensitive actions with consequences on a system are often session-related.

In step ②, the attacker tries to trick the victim into issuing the request, for instance with fishing; it can take the form of an email containing a direct link, or it can be embedded in an XSS vulnerability.

In step ③, the victim has been tricked (e.g., clicked on the link), resulting in an unwanted request toward the vulnerable Web application. Because the victim has a running session on the application, his browser appends the corresponding credentials to the request.

In step ④, the banking portal receives the request along with credentials. Since it has no protection against CSRF, the action is performed.

In step ⑤ the attacker receives the money he/she stole from the victim without his/her consent.

COMPLEXITY OF TESTING

CSRF mitigation can be easily implemented. The preferred protection mechanism against CSRF is the **synchronizer token pattern**. It consists of generating a unique randomly-generated token, which is inserted into sensitive URLs (as a key/value parameter) and Web forms (as a hidden field). Users who click on a link or submit a Web form will therefore send the token along with their input data. In this way, servers check for the good reception of the token. If the received token is missing or if it is different from the one that was generated and sent with the last server response, then the incoming request is dismissed. This mechanism ensures that requests are made from within the Web application GUI.

Another protection mechanism is the challenge-response method, which involves a verification step by the user before an action is completed. However, many Web application developers ignore these protection mechanisms, and a detection phase is required.

Automated, universal CSRF detection is a tough challenge, especially if it is based on request/response analysis: for instance, the presence of tokens in requests and responses does not ensure that they are processed by the server. Hence, the most reliable method to check whether a Web application is vulnerable to CSRF is to actually tamper with it, if possible in a harmless way.

Nevertheless, a common test scenario for CSRF detection consists of first performing the action under test in a nominal way, following the application's intended behavior, and saving the server's response for comparison. In a second step, it implies simulating a user caught in a fishing scam, for example by clicking on a link whose consequence is to send a direct request to the server to perform the action under test. The user must be authenticated to the Web application beforehand. Then, verdict is assigned by comparing the two responses. Another test scenario consists of swapping tokens between two separate user sessions and observe whether the server performed the actions or rejected them. However, this technique only shows if tokens are taken into account and does not ensure the presence of a CSRF vulnerability, since the presence of tokens may only be a decoy.

Although efficient, these scenarios are not straightforward, especially if conducted au-

tomatically. First, they require the ability to authenticate as a user with permissions to perform the action under test. Being able to provide valid credentials and successfully authenticate to a Web application is not always as straightforward as one thinks for automated scanners. Second, it means navigating to the page displaying the action, which requires some knowledge of the application's logic. Third, automatic verdict assignment is often complex because there may be no direct telltale indicating that an action has been completed or rejected.

2.4/ PRIVILEGE ESCALATION

The 3 previous vulnerability types addressed in this chapter, XSS, SQL Injection and CSRF, are all technical flaws. While there are cases where understanding the application under test is mandatory to conduct vulnerability testing, the detection process is often mechanical and can be performed the same way on most systems. On the contrary to these vulnerability types, Privilege Escalation is a logical vulnerability, meaning it is closely linked to the logic of applications. Privilege Escalation is part of a greater vulnerability type known as *Missing Function Level Access Control*, which is the 7th most highly ranked Web security risk, according to OWASP.

Whereas authenticating to a Website is probably one of the most frequent tasks performed by users multiple times every day, in Web applications with different user roles, an authentication mechanism is not enough to handle the delivery of content tailored to the user and its associated role. In these cases, users must have the **authorization** to access privileged functions. Authorization determines whether the authenticated user can perform an action and whether the resources can be accessed, depending on its "role". Roles management is usually handled with the help of an Access Control List (ACL), which is a list of Access Control Entries (ACE). An ACE is a combination of two values: a user or a role, and a type of resource or a specific resource. In other words, an ACE allows a certain user to access a certain resource.

However, the main problem is that Web application development languages do not embed built-in support for authorization policies specification. Therefore, access control mechanisms are implemented by Web developers, potentially making them flawed by design. A common error is to rely on the GUI to restrict user actions. It consists of hiding/disabling direct links to restricted areas to users that don't have the sufficient privileges. In this scenario, all it takes for one to access a function is its URL. Finding URLs to restricted functions is just a matter of time (URL bruteforcing, social engineering, etc.).

In other cases, the authorization scheme is extremely complex that makes it prone to human errors and at risk of not being restrictive enough. Indeed, reports of vulnerabilities related to permissions, privileges and access control are very frequent. Moreover, they are often associated with a high CVSS Severity and their victims are first class companies such as IBM¹², Cisco¹³, or Samsung¹⁴.

¹²<https://Web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0160> [Last visited: August 2015]

¹³<https://Web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-0713> [Last visited: August 2015]

¹⁴<https://Web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2015-3435> [Last visited: August 2015]

PRIVILEGE ESCALATION: USAGE EXAMPLE

Because this vulnerability type encompasses logical concepts, there is no typical workflow. Attackers will fingerprint an application and gather relevant information in order to understand its logic, the various user roles that exist, and restricted sensitive functions. As an example, we consider again the forum board that served as example in Section 2.1. As mentioned before, users can engage conversation, create topics and post messages.

With a misspelled word or an erroneous information, sometimes comes the need to modify a message. Rather than deleting the whole message and creating a new one, which may become out of context if other users have posted replies since, a popular function is “message edition”. A click on the edit button attached to one of its message will lead the author to the edition page, containing a form filled with the initial message and a submit button to save modifications.

Consider for instance an attacker who wants to edit messages from other users (e.g., for political purposes). His first reflex is to analyze how the Web server constructs URLs. In this case, the server has no URL rewriting and pages are visible in plain-text:

```
https://www.unsecure-forum.com/edit_post.php?id=1984729847
```

Based on this URL pattern the attacker fingerprints the Web server structure, looking for admin functions. He tries prefixing pages with “admin_”, “admin/”, etc. He finally gets a different error message when trying to access a restricted folder and discovers the “backend/” folder. He pursues with pages discovery and finds out that the “edit_post” page has a duplicate in the backend directory, which lacks of proper access control:

```
https://www.unsecure-forum.com/backend/edit_post.php?id=1984729847
```

The attacker gained access to an admin function and now has the ability to edit any message in the database as long as he provides its *id*, which can be guessed easily by analyzing features such as “reply” or “quotation”.

COMPLEXITY OF TESTING

The detection of the failure of an application to restrict access to unauthorized users is relatively easy if the application’s logic is well understood. Indeed, testing for such flaws consists of generating a matrix that enumerates all user roles as well as all resources. The design of the matrix can be done with the aid of the application developers, or simply by manually crawling the application and making educated guesses. For instance, it is useful to find out if the back end of the Web application under test is based on a Content Management System (e.g., Wordpress has six roles, from subscriber to super-admin).

The problem of this technique lies in its cost. First, it requires a consequent amount of manpower to design the matrix and confront each function with all the user roles. However, using spider tools can facilitate the process as they provide automation for the execution part. One has to provide credentials for each user role and then let operate the spider tool. Nonetheless, the verdict assignment activity is still done manually and going through the logs of a spider tool can be cumbersome. Some scanners however, such as AppScan¹⁵ or IronWasp, use the GUI as test Oracle: they perform a first crawl using a

¹⁵http://security.media-solutions.de/download/whitepaper_watchfire/testing_privilege_escalation.pdf [Last visited: August 2015]

set of credentials, and then perform a second exploration using another set of credentials, with a lower authorization level. Finally, they try to access URLs they found during the first crawl but did not find during the second crawl. If the server response is similar to the one obtained during the first crawl, they report it as a vulnerability.

STATE OF THE ART

Contents

3.1 Static Application Security Testing	26
3.2 Dynamic Application Security Testing	27
3.2.1 Automated Penetration Testing: Web application Scanners	28
3.2.2 Manual/Tool-aided Penetration Testing: Ethical Hacking	29
3.3 Model-Based Vulnerability Testing	31
3.3.1 Pattern-Based and Attack-Model Based Approaches	32
3.3.2 Model-Checking Approaches	34
3.3.3 Fuzzing Approaches	35
3.4 Synthesis	36

As reported previously, Web applications are fast evolving software systems, on which society relies more and more. However, too many of them are plagued with vulnerabilities that, if exploited by malicious users, may lead to sensitive data disclosure, identity theft, money stealing, and so on. Making these systems safe and secure has therefore been a strong area of research from academics as well as industrial companies.

One solution consists of attack **prevention** by designing third-party defense mechanisms. These mechanisms may be installed on the server, such as Web application Firewalls (WAFs), and/or on the client's browser that examines incoming data and sanitizes or rejects anything considered malicious. For instance, lots of solutions have been elaborated to protect against injection vulnerabilities, based on Web proxies [44], reversed proxies [77], dynamic learning [9], data tainting [54], fast randomization technique [4], data/code separation [23], or pattern-based HTTP request/response analysis [50]. Vulnerability prevention has proven to be efficient in a variety of general cases, but since it is generic by definition and has no knowledge of the system it is trying to protect, it misses a lot of attacks. For instance, client-side filtering cannot handle stored XSS [47] because it is unable 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 being unaware of the severity of vulnerabilities and good practices that help enforcing security. Worse, it might invite them to solely rely on third party security tools like WAFs and encourage poor-secured Web applications to proliferate. Web application security should be pushed toward writing secure code rather than deploying extra defense layers.

Another solution is then **detection**, also known as Web application Security Testing. Related work on Web application vulnerability testing, as stated in Figure 3.1, can be classified in two categories: Static Application Security Testing (SAST), and Dynamic Applica-

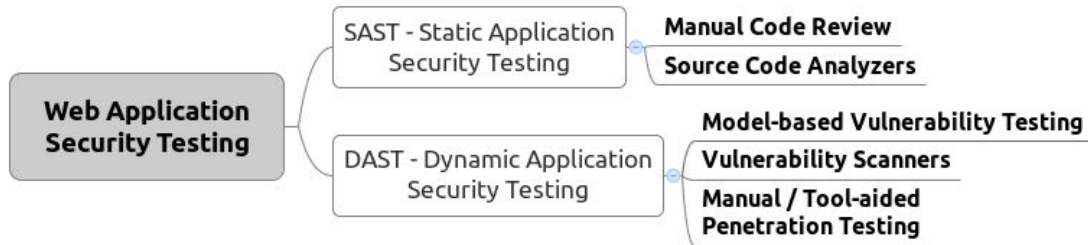


Figure 3.1 – Web application Vulnerability Testing Overview

tion Security Testing (DAST). Both categories feature manual and automated techniques. The first category encompasses the use of code-based techniques (mainly code review and analysis), while the second category consists of executing and stimulating the system in order to detect vulnerabilities (e.g., penetration testing). We discuss in the next sections the main techniques for both categories from the literature and the state of the practice. Note that we consecrate a section on the DAST category encompassing Model-based Vulnerability Testing techniques to point out their strengths and weaknesses, in order to further motivate the PMVT approach.

3.1/ STATIC APPLICATION SECURITY TESTING

A good majority of the techniques found in the literature propose to deal with Web application vulnerabilities using SAST techniques, most of them relying on taint analysis [57], which consists of keeping track of the values derived from user inputs. Although SAST techniques are not directly related to the work presented in this manuscript, we discuss the most relevant SAST papers in order to provide a self-contained state-of-the-art and motivate the design of a DAST technique.

Kieyzun et al. [43] propose a vulnerability detection technique addressing SQL Injections and XSS-1 (reflected) as well as XSS-2(stored), based on dynamic taint analysis. The main idea is to track the flow of tainted data in the application, and check whether tainted data can reach “sensitive sinks” (e.g., `mysql_query` function is a sensitive sink for SQL Injections). If an input reaches a sensitive sink, it gets modified by using a library of attack patterns, in order to test a vulnerability type. For XSS-2, a concrete+symbolic database is used and the program is run on two inputs (one from the attacker, the other from the victim). The algorithm tracks the flow of data from the attacker’s input through the database, and to a sensitive sink in the execution of the victim’s innocuous input. The symbolic database stores the taint sets of each given input parameter. Two observation methods for XSS: lenient and strict. The tool looks for differences between the output of an innocuous input and a potentially malicious input. Lenient reports a vulnerability when the output differs in script-inducing elements (e.g., `<script>` tags) or HTML elements (e.g., `href` tags). Strict seek differences only in script-inducing elements. The approach has been implemented as a tool called *Ardilla*¹. Its evaluation shows a good stored-XSS detection rate but a low code coverage.

¹see <http://groups.csail.mit.edu/pag/ardilla/>

Wassermann and Su [72] use string-taint analysis for cross-site scripting vulnerabilities detection. This technique combines the concepts of tainted information flow and string analysis. It detects XSS vulnerabilities due to unchecked untrusted data as well as those due to insufficiently-checked untrusted data. An analysis algorithm first translates programs into static single assignment form in order to encode data dependencies, then it constructs a context free grammar. Authors have established a policy that enumerates the possible manners an HTML document can invoke a browser's Javascript interpreter, by studying the Gecko layout engine as well as the W3C² recommendation. It states that no untrusted data should invoke the Javascript interpreter. XSS discovery is performed by analyzing the intersection between the Control-Flow Graph (CFG) of the program under test and the regular expressions that describe the policy: a non-empty intersection means that there is a vulnerability. Evaluation of the approach shows good results, unreported vulnerabilities have been discovered on *Claroline*³, although the approach does not work in a few cases.

Shar et al. [61] present an automated approach that not only detects XSS vulnerabilities, but also statically removes them from program source code. The detection phase relies on a taint-based analysis technique, that consists in extracting a CFG from program source code, that shows data dependence between input nodes and HTML output nodes. HTML output nodes produce an HTML response output, and an HTML output node *o* is defined as a potentially vulnerable output node if *o* is also an input node, or if *o* is data dependent or transitively data dependent on an input node *i*. The identification of potential vulnerable nodes has been implemented by tracking the flow of untrusted data between input nodes and HTML output nodes. They implemented a prototype-tool called *SaferXSS* and evaluated it against 5 Java-based open source applications. Results show that all the potential vulnerabilities identified by the tool were successfully removed.

Code analysis appears quite effective for detecting injection-type vulnerabilities. However, one main weakness is that program source code is not always available. Moreover these techniques are usually bound to one or a few specific programming languages, while there exists a tremendous number of languages to develop a Web application (PHP, ASP, JSP, Ruby, J2E, and so on). Several black-box techniques have been proposed regarding the detection of Web application vulnerabilities.

3.2/ DYNAMIC APPLICATION SECURITY TESTING

To overcome the issue of source code an/availability, especially for applications manipulating sensitive data (e.g., in the domain of banking), for which static testing cannot always be deployed, another research area, called Dynamic Application Security Testing (DAST) investigates "black-box" testing techniques. Its objective is to detect vulnerabilities and weaknesses by tampering with a running Application. In this section, we first discuss fully automated techniques, also viewed as point-and-shoot solutions. Second, we introduce manual/tool-aided penetration testing, the most popular approach for assessing the robustness of systems. Finally, we present current Model-Based Vulnerability Testing techniques and discuss them w.r.t. the research challenges we have identified.

²<http://www.w3.org/> [Last visited: August 2015]

³<http://www.claroline.net/>

3.2.1/ AUTOMATED PENETRATION TESTING: WEB APPLICATION SCANNERS

In the current state of the practice, automated penetration testing techniques are mainly implemented by a category of tools called Web application Scanners (WAS) [31, 7]. One can use WASs to test a Web application against major vulnerability types; There are many Web applications Scanners available on the market⁴.

These tools follow the same process composed of 3 steps:

1. **Exploration.** WASs explore the Web application under test using a **crawler**, i.e. a module that computes a map of the Web application. This map contains all the visited pages and the means to access each page: either the raw HTTP request of the page, or the User Interface (UI) link/button leading to the page.
2. **Testing.** Based on the previously-computed map, each page is then tested against various vulnerability types. At the HTTP level, each parameter of an HTTP request is fuzzed [64], whereas at the UI level, each user input field is fuzzed. Responses to attacks are stored, for further analysis.
3. **Analysis.** Based on the results obtained at the previous stage, each response from the server is analyzed to detect if the conducted attack has succeeded or failed.

Several kinds of **barriers** arise when it comes to use WASs [22, 7].

First, WASs have to handle **identification barriers** (e.g., user session, anti-CSRF tokens, etc.), embedded in Web applications in order to prevent identity abuse. These barriers arise at the HTTP level, but not at the UI level. Indeed, browsers handle them natively because they automatically append identification data to each HTTP request in order to simulate statelessness. If a WAS doesn't handle identification barriers, the server will reject any subsequent HTTP request. These barriers are encountered during the exploration and the testing phase, when WASs send requests to the server in order to respectively reach new pages or attack the application. It should be noted that most modern WASs are able to bypass identification barriers.

Second, WASs have to handle **logical data barriers**, meaning that the values of HTTP requests or the values of user inputs have to be set with relevant business-related values. These barriers arise at both HTTP and GUI levels. If a WAS does not handle these barriers, the server may reject HTTP requests containing irrelevant values (e.g., a transmitted parameter representing an email containing a simple string), and/or the client-side validation may reject such values in the input fields (e.g., input field representing an email filled with a simple string). In these cases, the request can be considered as malformed, and not even computed by the Web application. These barriers are encountered during the exploration and the testing phase, when the WAS sends requests to the server in order to respectively reach new pages or attack the application. In order to deal with this issue, one can define default values in the WAS, which may become a time-consuming work.

Third, WASs have to handle **logical work-flow barriers**, meaning that, in order to reach a particular page, WASs have to deal with a particular **sequence** of business-related interactions with the Web application under test. This barrier arises at both HTTP and GUI levels. It is mostly encountered during the testing phase, in which each page is

⁴For a list of open-source and commercial scanning tools maintained by OWASP, see https://www.owasp.org/index.php/Category:Vulnerability_Scanning_Tools [Last visited: August 2015].

tested. The problem is that each page is often tested *separately*, without consideration of the business-related flow.

To conclude, automated penetration testing techniques can achieve a first level of vulnerability detection by crawling Web applications and bombarding them with thousands of attack vectors. They can also point potential flaws to penetration testers in order to ease the discovery process. However, these tools produce a significant amount of false positives as well as false negatives, which makes them not reliable enough for critical systems like eBanking and eHealth Web portals, social applications, etc.

3.2.2/ MANUAL/TOOL-AIDED PENETRATION TESTING: ETHICAL HACKING

Penetration testing, also viewed as ethical hacking, is probably the most popular of all software security testing techniques [3, 66]. Companies and software owners heavily rely on penetration testing to assess the robustness of their products because it does not imply any change in their development life-cycle. This activity usually intervenes once development is completed, just before release.

Penetration Testers (aka pentesters) are white-hat hackers. As such, their process is similar to black-hats: they analyze systems to find breaches, flaws or backdoors, and if their search is successful they try to make their way into the systems' back end (e.g., DBMSs) and retrieve sensitive data. But they differ in their intention and what they do with their findings. Black-hat hackers' end-game is personal gain: they steal data in order to make money (e.g., by selling the data, or by threatening owners to disclose the stolen data and asking for a ransom), they also deny access to service for political reasons. On the contrary, white-hat hackers motivation is to make systems and network safer. They don't disclose stolen data but use it as a proof of poor security to alert owners in order to make them consider the issue and fix their product.

A theoretical workflow for penetration testing has been proposed by Engebretson [26], which can be broken down into 4 phases:

- i - Reconnaissance:** Information is naturally disclosed everywhere: on companies' Website, jobsites, social media, google. Penetration testers gather intelligence on their target: information about companies structure, about hosts. The objective is to use this information to plan tailored attacks. They often tackle the issues of using an information for social engineering attacks or discovering the build version of a targeted Web server. For Web applications, it also includes understanding the site map, its features, what users can and cannot do, the various user roles, and the kind of data stored in the DBMS.
- ii - Scanning:** Pentesters make use of the information they gathered during reconnaissance, and by relying on their experience start scanning their target for vulnerabilities and breaches. This process can be done manually or with the aid of technical tools, such as vulnerability scanners, which speed the process and at the same time harvest further preliminary data.
- iii - Gaining access:** When a breach has been found, next step is to exploit this breach in order to gain a privileged access to the target. For example, it may consists of obtaining admin credentials

iv - Maintaining access: With a direct privileged access to the targeted system (e.g., with administrator credentials), it is now possible to perform the “mission”: stealing data. In data stealing, pentesters rarely operate in a single shot, and there may be time constraints that force them to maintain their access open. For instance, it may consist of installing a backdoor to be able to connect to the target anytime.

Such process is often associated with a full report about the activities of the pentester(s). This document is sent to the system’s owners and serves as proof for lack of security. It should be noted that completing all 4 phases is not mandatory for assessing the robustness of a system but the further a pentester goes, the more the impact will be on the system’s owners to encourage them securing their product.

To achieve their ends, pentesters heavily rely on the use of tools to automate the detection and exploit process as much as possible. Main pentesting assisting tools are **intrusive proxies** such as Burp⁵ or Zap⁶). Each of these tools provides similar features, and may be very efficient in pentester expert hands. As proxies, they can **sniff** HTTP messages exchanged between browsers and a server. Moreover, these tools can record/replay each sniffed exchange. Using these features, pentesters can exactly know the exchanged data. Moreover, as an **intrusive** tool, transiting HTTP messages can be intercepted and modified on-the-fly with attack vectors. Thanks to this capability, pentesters can on the one hand test the robustness of the server-side part of Web applications (e.g., by modifying HTTP messages transiting to the server), and on the other hand test the robustness of the client-side part of Web applications (e.g., by modifying HTTP messages transiting to the browser). These tools are also able to replay recorded HTTP messages, with or without modification, and **spider** Web applications (i.e. explore a Web application and create a map as any Web application Scanner can do). Finally, intrusive proxies can usually be extended with plug-ins, depending on the community, for example dedicated to automated **fuzzing** and automated **injection**.

The problem of these tools is that they work at HTTP level, which is not suitable for many modern applications [35]. Indeed, most critical systems embed protection mechanisms to force users to interact with the GUI, which makes the crafting of relevant HTTP requests very complex. These mechanisms can be spread on both client-side and server-side parts of the application. For instance, when a Web server generates a new page, each user action (hyperlinks, from submit, ...) is associated to a dynamically-generated control parameter. When users interact with the application through the GUI, the Javascript code of the client-side part of the Web application crafts the correct HTTP request with the relevant control parameters. Hence, each request to the server embeds control parameters, dynamically generated on each page. Without the knowledge of the Javascript code’s behavior and the knowledge of the control parameter, crafting a correct HTTP request is merely impossible.

Therefore, when the use of tool is not a viable option, penetration testing turns to manual testing. In this context, the know-how and experience of the pentester is even more crucial. For each kind of vulnerability, pentesters rely on informal test patterns that provide the procedure to test the application, in order to look for vulnerabilities. They typically design code scripts (e.g., using python) to automate parts of the detection and exploit process. For example, a script can help to automate the iteration over a list of attack vectors for SQL Injection discovery.

⁵<https://portswigger.net/burp/> [Last visited: August 2015]

⁶<https://www.owasp.org/index.php/ZAP> [Last visited: August 2015]

The main limitation of manual penetration testing is its **lack of exhaustiveness** [34], as pentesters only focus on a sub-part of the application considered the most “at risk”. A Penetration testing campaign is often submitted to a limited budget, which makes pentesters work within a restricted time box, thus forcing them to make compromises about their test process. It means having to focus on major vulnerability types and conduct attacks only on the major parts of Web applications (based on their understanding of the applications, and/or following a risk assessment), letting aside parts of applications that, even if considered less critical, may be vulnerable and represent an entry-point for hackers to compromise the entire system.

Moreover, this technique can be viewed as a **one shoot solution** [55]. Therefore, for long-life Web applications with a constant evolution in their features and the way they deliver information, the cost of conducting a new penetration testing campaign will be close to the initial one, as the reusable material in this process is very thin.

Lastly, penetration testing is a **late life-cycle paradigm** [3] and as a result uncovers vulnerability too late, often when the final product development is considered completed. Hence, when both time and budget are severely constrained, options for fixing breaches are limited. These constraints lead companies to overlook security warnings and flaws in their products, because they assessed that the fixing-cost is superior to the probability of an attacker exploiting the flaw associated with the criticality of the assets at risk.

3.3/ MODEL-BASED VULNERABILITY TESTING

Model-Based Testing (MBT) [68] refers to the process and techniques for the automatic derivation of test cases from models. 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.

More recently, MBT has been extended to address security testing. A taxonomy about the current state of the art of Model-Based Security Testing has been created in [28]. Authors considered more than a hundred approaches and categorized them depending on several factors, such as model type (Threat model, Attack model), test selection criterion (structural coverage, fault-based, etc.), and maturity of evaluated system (prototype, premature system, production system).

Model-Based Security Testing has proven its efficiency when it comes to test documented security properties [27, 40], such as access control policies [53]. However, a large part of what we call security is implied and usually not explicitly specified in a document. This part is referred as Security Vulnerability Testing, which consists of defining, identifying, quantifying and prioritizing the security holes (vulnerabilities) in a system, network, or application software. Whereas Functional Security Testing is about “verifying that a given security property or policy is met”, Security Vulnerability testing is more about “verifying that users cannot use a system in a way it was not intended to be used” [67].

While model-based vulnerability testing may help conduct tests at every infrastructure layer (networks, systems, applications), most papers focus on application-level vulnerabilities, typically for Web applications. In this section, we provide an overview on Model-Based Vulnerability Testing, which can be grouped into 3 main families: pattern-based and attack-model based, model-checking, and fuzzing approaches.

3.3.1/ PATTERN-BASED AND ATTACK-MODEL BASED APPROACHES

The majority of MBT papers have chosen not to represent the behavior of the SUT, but rather the behavior of attackers. Whether these are referred to as *attack-models* or *patterns*, the idea is to model how malicious individuals would conduct an attack, step by step.

Blome *et al.* [10] describe a model-based vulnerability testing tool called VERA, standing for ‘VERA Executes the Right Attacks’. It is based on the fact that usually the presence of a vulnerability is a prerequisite to deploy an attack, but actually exploiting this vulnerability is time-consuming. This approach relies on attacker models, that can be seen as extensions of Mealy machines. These models, if combined with the back-end of the approach, can provide fully automated test scripts. The back-end of the approach is composed of (i) an instantiation library, which is basically a list of nominal and malicious vectors, (ii) a configuration file that contains system-specific information (cookie data such as the session ID, target URL, specific headers, etc...), and (iii) an XML file describing the attacker model to be tested. This approach can address a large variety of vulnerabilities like code injection, source disclosure, file enumeration, RFI, CSRF, among others. However, advanced vulnerability types like multi-step XSS and second order SQL Injections are not addressed in this paper. It would require to model a complex heuristic to inject and observe this particular vulnerability type.

Bozic and Wotawa [15] present a model-based testing approach relying on attack patterns to detect Web application vulnerabilities, namely SQL Injections and XSS. An attack pattern is a specification of a malicious attack. Represented by a UML state machine, it specifies the goal, conditions, individual actions and post-conditions of the represented attack. Test cases are computed and executed by branching through the states of the state machine and executing the corresponding methods of the SUT. Concretizing vulnerability test cases with malicious inputs is either done by supplying a static list (for SQL Injections), or by dynamic generation based on combinatorial testing (for XSS, more in-depth details are found in [14]). This approach has been implemented as a toolchain using several existing tools, such as Yakindu⁷ for the state machine modeling, Eclipse⁸ to encapsulate the entire system, and WebScarab⁹ for the interpretation of communication between the Web application and clients, and for manual submission of attacks. Experiments have been conducted on 3 vulnerable applications (DVWA, Mutillidae, and Bodgelt) and one real life application (WordPress Anchor). SQLI and XSS vulnerabilities were found on Mutillidae and DVWA, on various security levels. On the contrary, no vulnerability was found on WordPress Anchor because an administrator needs to approve each post submitted by users. Therefore, it requires a more detailed model of the attack. This technique, however, only addresses single-step XSS and SQL Injections, and there is no mention on how to treat cases that involve user actions between injection and observation. In addition, interactions with Web applications are based on HttpClient, a java library that provides basic functionality for accessing resources via HTTP but does not provide the full flexibility or functionality needed by many applications. As a consequence, client-side code is not taken into account, which represents a handicap to the deployment of this technique on modern applications with a strong reliance on Javascript and Ajax.

Wei *et al.* [73] focus on penetration test case inputs and propose a model based pen-

⁷<http://statecharts.org/> [Last visited: August 2015]

⁸<https://eclipse.org/> [Last visited: August 2015]

⁹https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project [Last visited: August 2015]

etration test method for SQL Injections. First, they provide attack models using the Security Goal Model notation, which is a modeling method used to describe vulnerabilities, security properties, attacks, and so on. Models are generic and describe goals in a top-down fashion. A typical goal is for instance “steal system information”, and is modeled as two sub-parts: error-message utilizing and blind injection. Hence, each top-down path in a model represents an attack process that realizes a certain attack goal. Each top-down successful attack process represents the attack scheme, defined as a triple $\langle OBJ, INP, OUT \rangle$, **OBJ** being the attack goal, **INP** being the attack input, and **OUT** being the vulnerable response of the Web application. To perform an actual attack, one must instantiate the test case model according to the fingerprint of the Web application and use certain coverage criteria to generate executable test cases. Authors created an automated Web application SQL Injection vulnerability penetration test tool called NKSI scan: it applies the widely used “crawling - attack – analysis” method to detect the SQL Injection vulnerability in subject applications. They compared their tool technique with popular scanners IBM AppScan and Acunetix. Results show that NKSI was able to discover more flaws than those two scanners. Nonetheless, this technique still makes use of a crawler to identify injection points. It therefore may be unable to visit parts of Web applications and cannot address second order SQL vulnerabilities.

In [78], authors present an approach to automate the generation of executable security tests from Threat Model-Implementation Description (TMID) specifications, which consist of threat models represented as Predicate/Transition (PrT) nets and a Model-Implementation Mapping (MIM) description. A threat model describes how a malicious individual may trigger the system under test to violate a security goal. A MIM description maps the individual elements of a threat model to their implementation constructs. Abstract test cases (i.e. complete attack paths) are computed in two steps. First a reachability graph is generated from the threat net. It represents all states and state transitions reachable from the initial marking. Then the reachability graph is transformed to a transition tree containing complete attack paths by repeatedly expanding the leaf nodes that are involved in attack paths but do not result from firings of attack transitions. Concrete test cases are derived by automatically composing the attack paths and the MIM description. The approach has been implemented in ISTA, a framework for automated test code generation from Predicate/Transition nets, and experiments have been conducted on two real-world systems. It shows good results with most vulnerabilities being found (90%), whether they are Web-related vulnerabilities (XSS, SQLi, CSRF, etc.) or protocol-based vulnerabilities (FTP).

Salva *et. al.* [58] present a Model-Based Data Testing approach for Android applications that automatically generates test cases from intent-based vulnerabilities, using vulnerability patterns. It specifically targets the Android Intent Messaging mechanism, whose objective is to allow sharing of actions and data between components using content-providers, in order to perform operations. The concern is that attackers may exploit this mechanism to pass on payloads from component to component, infecting the whole system and making their attack more severe. This approach therefore searches for data vulnerabilities inside components. The automated generation of test cases relies on 3 artifacts: vulnerability patterns, class diagrams, and specifications. Vulnerability patterns are specialized Input-Output Symbolic Transition Systems, which allow formal expression of intent-based vulnerabilities. A pattern formally exhibits intent-based vulnerabilities and helps to define test verdicts. Class diagrams are partially generated from the decompiled Android application under test, and represent Android components with their types

and their relationships. They typically provide the Activities (these are Android components that display screens to let users interact with programs) or Services composed with content-providers. Specifications are generated from the Android manifest. They express the behavior of components after the receipt of intents combined with content-provider requests. Test case generation is performed by composing the 3 artifacts. This method has been implemented in a tool called APSET, and has been experimented on several real life applications. Results support the effectiveness of the tool, finding vulnerabilities in popular Android applications such as YouTube and Maps.

3.3.2/ MODEL-CHECKING APPROACHES

Test cases can also be obtained by using a model-checker. Given a Website specification/model, a typical model-checking approach will inject faults into the model and use a model-checker to generate attack traces. Various techniques have been proposed to detect technical vulnerabilities (XSS, SQLi, CSRF, etc.) as well as logical vulnerabilities (authentication bypass, insecure direct object references, etc.).

Buchler *et al.* [17] represent the SUT using a secure AVANTSSAR Specification Language (ASLan++) model, where all traces fulfill the specified security properties. A library of fault injection operators has been developed. The goal is to apply a fault injection operator to the model, and use a model checker to report any violated security goal. If a security goal has indeed been violated, the reported trace then constitutes an Abstract Attack Trace (AAT). The attack traces are translated into concrete test cases by using a 2-step mapping: the first step is to translate an AAT into WAAL (Web application Abstract Language) actions, the second step is to translate WAAL actions into executable code. An attack may be conducted in a fully automated fashion, at the browser level. In some specific cases (disabled input elements, etc.), a test expert may be required to craft HTTP level requests in order to recover from the error. So far, this approach has only addressed first-order XSS. Moreover, the creation of a formalized representation of Web applications may become tedious and time-consuming as the size and complexity of the Web application under test rise.

Rocchetto *et al.* [56] present a formal model-based analysis technique for automatic detection of CSRF during the design-phase. It is based on the ASLan++ language to define the several entities involved (client, server) and their interactions. The client is used as an oracle by the attacker, and the model is centered around the Web server and extends the work of Dolev-Yao (usually used for security protocol analysis). The design of models relies on the extraction of a specification. Once designed, models are submitted to the AVANTSSAR platform, which, when a CSRF is found, returns an abstract attack trace reporting the list of steps an attacker would follow in order to exploit the vulnerability. This technique takes into account that the Web server may have some CSRF protection in place, and will try to bypass it. It will typically look for CSRF token-related flaws, for instance if the tokens are unique for each client, and for each client/server interaction. If no attack trace is produced, the specification is considered safe regarding CSRF. Authors assume that attackers can listen to the network and build their attack upon the transactions between a client and the server. However, this technique is not about testing the final product itself but rather the design of the application, relying on a specification or on how developers are planning to proceed. Therefore, testing the specification itself does not ensure that the implementation is protected.

Felmetsger *et. al.* [29] present advances toward the automated detection of application logic vulnerabilities, combining dynamic execution and model checking in a novel way. Dynamic execution allows for the inference of specifications that capture a Web application's logic, by collecting likely invariants. A likely invariant is derived by analyzing the dynamic execution traces of the Web application during normal operation, and captures constraints on the values of variables at different program points, as well as relationships between variables. The intuition is that the observed, normal behavior allows one to model properties that are likely intended by the programmer. Model checking is used with symbolic inputs to analyze the inferred specifications with respect to the Web application's code, and to identify invariants that are part of a true program specification. A vulnerability is therefore any violation of such an invariant. This technique has been implemented in a tool called Waler (Web application Logic Errors AnalyzeR), which targets servlet-based Web applications written in Java. Up to now, Waler detects a restricted set of logic flaws and is currently limited to servlet-based Web applications, but was still able to find previously-undetected vulnerabilities in real-life applications while producing a low number of false positives.

3.3.3/ FUZZING APPROACHES

Fuzzing is extensively used in vulnerability testing [41] to introduce malformed data or mutate nominal values to trigger flawed code in applications. Fuzzing techniques are usually very cheap to deploy, don't suffer from false positives, but lack an expected-result model and therefore rely on crashes and fails to assign a verdict. Two main fuzzing techniques exist: mutation-based and generation-based. Mutation fuzzing consist of altering a sample file or data following specific heuristics, while generation-based fuzzers take the input specification and generate test cases from it. Fuzzing may be used for crafting malicious input data [24], or crafting erroneous communication messages [71].

The approach presented by Duchene [24] consists of modeling the attacker's behavior, and driving this model by a genetic algorithm that evolves SUT input sequences. It requires a state-aware model of the SUT, either derived from an ASLan++ description or inferred from traces of valid/expected SUT execution. This model is then annotated using input taint data-flow analysis, to spot possible reflections. Concrete SUT inputs are generated with respect to an Attack Input Grammar that produces fuzzed values for reflected SUT input parameters. The fitness function of the genetic algorithm depends on the obtained SUT output following the injection of a concrete SUT input. It computes the veracity of an input by looking for correlations, using the string distance between a given input parameter value and a sub-string of the output. Two genetic operators are used: mutation and cross-over. It is an efficient technique for detecting XSS, as it goes beyond the classical XSS evasion filters that may not be exhaustive. Such a technique also tackles multi-step XSS discovery by using a more complex string matching algorithm to generate an annotated FSM, in order to inspect the SUT to find the possibilities of XSS at certain places. The downside of this approach is that it requires a great effort from test engineers to deploy the whole process: the model inference process needs to be rightly tuned. Also, it cannot handle modern Web applications, for instance client-side oriented applications (which rely on Javascript and Ajax).

A model-based behavioral fuzzing approach has been designed by Wang *et. al.* [71] to discover vulnerabilities of Database Management Systems (DBMS). A DBMS is defined

by a format rule that specifies packet format and a behavior rule that specifies its semantics and functionality. This approach is based on two main artifacts. The first artifact is a behavioral model, which includes fuzzing patterns and behavioral sequences. This is obtained from a behavior analysis of DBMS (protocol format analysis, attack surface analysis, etc.). A fuzzing pattern expresses the data structure of packets, the needs of security testing, and the design strategy for vulnerability discovery. A behavioral sequence defines the message transfer order between client and DBMS. The second artifact is a DBMS Fuzzer composed of a test instance (a detailed test script based on fuzzing patterns), and a finite state machine model EXT-NSFSM used for semi-valid test case generation based on behavioral sequences and test instances.

Authors describe a general framework for behavioral fuzzing that has been implemented and on which they carried on experiments. It allows for the generation of thousands of fuzzing instances, and despite a few errors of analysis and script, the tool was able to discover buffer overflow vulnerabilities, 10 of which were not released yet.

3.4/ SYNTHESIS

Two different strategies exist for improving security of Web applications: prevention and detection. Prevention is about designing security mechanisms and functionalities to protect applications against attacks. Detection is about testing a Web application for vulnerabilities to fix them before putting the final product online. When it comes to security testing of Web applications, Two main techniques are proposed in the literature and the state-of-the-practice: SAST and DAST.

SAST designates white-box techniques: they analyze source code, either manually or automatically, to detect faults in the implementation of the application that may lead to a vulnerability. However, SAST techniques suffer from several weaknesses. First, they are usually bound to a single programming language, limiting their testing scope to Web applications that use the same language. Second, they produce a fair amount of false positives because they don't run the application, and consequently report suspicions of vulnerabilities rather than actual vulnerabilities. Third, Web application source code is often unavailable.

As opposed to SAST, DAST designates black-box techniques: they probe Web applications for security vulnerabilities, without access to source code, by mimicking attacks from hackers.

The probing can be done manually by ethical hackers and security test engineers. We then speak of manual penetration testing. These experts rely on their experience and know-how, potentially aided with specific tools such as spiders tools, to conduct attacks in order to penetrate and compromise the application. This is the most widespread technique within the industry. Even so, there are several drawbacks inherent to manual penetration testing. First, manual penetration testing lacks exhaustiveness, as test engineers have to work within restricted time-boxes often due to a limited budget. Second, this technique can be viewed as a one shoot solution, which is not adapted to Web applications with a constant evolution in their features. Third, penetration testing is a late life-cycle paradigm that uncovers vulnerabilities too late, when options for fixing them are limited.

The probing can also be done automatically, using Web application vulnerability scanners. These tools are point-and-shoot solutions that bombard the Web application under test with thousands of malicious requests carrying payloads. These tools can test for a major-

ity of vulnerabilities with very little cost in human resources and time. However, scanners are meant to be generic and suitable for any Web application, which means they lack the necessary behavioral knowledge of the application to detect complex vulnerabilities. Indeed, they don't take the logic of applications into account, which leads to incomplete crawling results, and an incapacity to perform sophisticated attacks. Moreover, the detection rate strongly varies between scanners, and users may have to use several scanners in order to get a trustable feedback about the security status of their application(s).

A new type of approaches that extend MBT for security testing has been the object of many research studies and has attracted the interest of a large group of scientists in recent years. These approaches rely on a variety of techniques to compute black-box test cases, such as attack patterns, fuzzing, and model-checking. They have shown promising results, having better detection rates than scanners and being less time consuming than manual penetration testing. However, Model-based Vulnerability Testing approaches usually focus on only one or two vulnerability types, from the same category (e.g., XSS and SQLi are both malicious data injections). In addition, test concretization is often considered as an accessory and kept minimal. Consequently, these approaches have issues to address scalability and thus, modern Web applications heavily relying on Javascript and JQuery¹⁰.

With PMVT, we aim at advancing the state-of-the-art of Model-based Vulnerability Testing as well as of security testing, by proposing a model-based approach that can generate tests for 4 vulnerability types that are very different and concerns various aspects of Web applications (i.e., structure, logic, etc.). Test concretization has been addressed with care to enable the computation of executable test scripts on any Web application, regardless of their back-end and the technologies they rely on. The objective of PMVT is to conduct sophisticated attacks to unveil complex vulnerabilities that scanners fail to detect, with a good level of automation that makes it a better choice than manual penetration testing in terms of time and resources costs.

¹⁰<https://jquery.com/> [Last visited: August 2015]

BACKGROUND

Contents

4.1 Certifylt: Software Testing based on UML and OCL	39
4.1.1 Modeling Notation	41
4.1.2 Test Selection	45
4.1.3 Test Generation	50
4.1.4 Test Concretization, Execution and Verdict Assignment	51
4.2 Certifylt for Vulnerability Testing	52

The work presented in this manuscript registers as a model-based software vulnerability testing approach, which has been developed on top of an existing toolchain called Certifylt [45]. This chapter centers on describing the inner workings of Certifylt in order to get a better understanding of how it served as a basis for the PMVT approach. The chapter ends with an explanation on why Certifylt is a good candidate for vulnerability testing of Web applications but needs to be augmented and adapted, and on how the PMVT approach can fill this gap.

4.1/ CERTIFYLT: SOFTWARE TESTING BASED ON UML AND OCL

Certifylt [45] is a Model-Based Testing toolchain, which is a fruit of the collaboration between the Institut Femto-ST¹ and a software testing company called Smartesting².

Smartesting is a model-based testing technology provider focused on technical areas such as IT systems, electronic transactions, security components, embedded systems and distributed systems. For this, Smartesting develops the Certifylt technology that allows automatic test generation from requirements to test cases using a model-driven approach. Development of the Smartesting core technologies began in the mid 1990s at the Université de Franche-Comté in Besançon (France), with the goal of reducing the cost of software validation using automated test generation techniques. Smartesting (renamed Smartesting Solutions & Services SAS since January 2015) was incorporated at the beginning of 2003 as a spin-off of the Computer Science Lab of the Institut Femto-ST (Université de Franche-Comté / CNRS / INRIA).

Certifylt is a fully equipped toolchain that allows the automated generation of functional test cases from a behavioral model of the SUT. Behavioral models that are designed as

¹<http://www.femto-st.fr/> [Last visited: August 2015]

²<http://www.smartesting.com> [Last visited: August 2015]

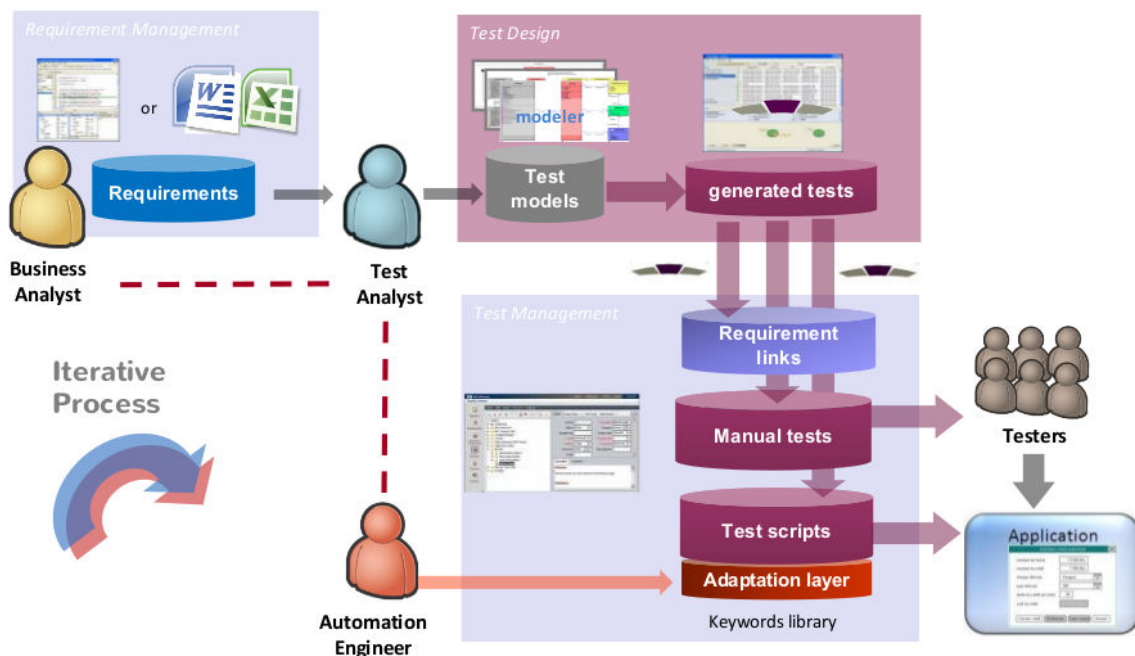


Figure 4.1 – Smartesting CertifyIt Process

part of the CertifyIt methodology use the UML notation and can be seen as the combination of 2 representations. On the one hand, the static representation (i.e. class diagrams) contains all the points of control and observation of the SUT, along with its state variables and their domain of definition. The initial state of the SUT is defined by instantiating these class diagrams using an object diagram. On the other hand, the dynamic representation (i.e. state-machines) contains the expected behavior of the SUT, which describes the evolution of its state variables defined in the static representation. Moreover, UML is associated with the Object Constraint Language (OCL) [36], which allows expected behavior to be formalized using OCL expressions, thus increasing the precision of models.

Test selection consists of deriving test targets to cover the test objective. CertifyIt can generate test cases for 2 types of characteristics: functional behavioral testing [12] and functional security testing [32]. Functional behavioral testing is about achieving structural coverage of the behavioral model, which combines decision-condition coverage for OCL expressions, and transition-based coverage for state-machines. Functional security testing relies on the formalization of test scenarios extracted from the test objective using high-level expressions called *test purposes*, in order to derive test targets from the model that are tailored to the test objective. Then, for each test target, an automated theorem prover is used to search for a path from the initial state to the targeted state, and data values that satisfy all the constraints along that path. The approach supports traceability between test cases and system requirements by using identifiers that refer to the expression of potentially informal and external requirements, and linking identifiers to model animation-related actions such as operation postconditions.

CertifyIt allows the publication of generated abstract test cases in a standardized XML format. This structured document is then used as a base for adapters that translate abstract test cases into executable scripts (e.g., JUnit³), or in a test repository (e.g., HP

³<http://junit.org/> [Last visited: August 2015]

Quality Center platform⁴).

We detail in the next subsections the activities that compose a typical MBT process using CertifyIt, as depicted in Figure 4.1 First, we present the modeling notation, based on UML and OCL. Second, we explain how test cases are selected and translated into test targets, depending on the test objective. Third, we describe the automated generation mechanism of abstract test cases from test targets. Finally, we express how abstract test cases are transformed into test scripts, which can be executed on the real system. Notice that we do not elaborate on the traceability between requirements and test cases, as it is natively handled by CertifyIt and is out of the scope of this thesis.

4.1.1/ MODELING NOTATION

CertifyIt uses a subset of UML2 [37], called UML4MBT [13], as notation for MBT purposes. UML is a standard graphical language, in a diagrammatic form, designed for writing software development plan while remaining a simple notation that is easy to write and understand. It can be used to view, specify, construct and document software system artifacts. Its large expressiveness makes it suitable for the modeling of various types of systems (Web applications, embedded systems, ...).

Because MBT models are bound to be processed in order to generate test cases, reducing the UML notation expressiveness as much as possible implies simpler processing and better test generation time. Thus, UML4MBT is based only on 3 diagrams: UML class diagrams (to model the control points and observation of the SUT), object diagrams (to define test data), and state-machines (to model dynamic behavior of the SUT). It also makes use of a subset of the OCL language [36], called OCL4MBT, in order to add constraints to operations and state-machine transitions. In addition, OCL4MBT is also used as an action language, i.e. a language that expresses state changes in the model, based on UML state machines and class diagrams [8].

These 3 diagrams, completed with OCL4MBT constraints, are sufficient to design comprehensive, precise and interpretable models to test finite state systems with the CertifyIt tool. The next subsections describe each modeling artifact of the UML4MBT and OCL4MBT notations.

4.1.1.1/ CLASS DIAGRAMS: STATIC STRUCTURE

Class diagrams express the static structure of the model in terms of classes and relationships between those classes. It provides an abstraction of the objects of the SUT and their relationships / dependencies.

UML4MBT *Class diagrams* modeling relies on a restricted set of UML elements:

- i - Classes:** They define a set of objects of the SUT with semantics and common properties. These properties are expressed through class attributes and class operations. Object-oriented properties such as inheritance are not implemented.

⁴<http://www8.hp.com/us/en/software-solutions/quality-center-quality-management/> [Last visited: August 2015]

- ii - **Reflexive and binary associations:** They represent a structural relationship between 2 objects of the SUT. Associations are always binary, defined between 2 classes. They may be reflective and link 2 objects of the same class. The available multiplicities are $0..1$, m , $n..*$ and $n..m$ with m and n integers such as $0 \leq n < m$. Association classes are not implemented.
- iii - **Class attributes:** They define the state variables of the SUT. They support primitive types (Boolean and bounded Integers) and enumeration types. Object types are represented using a 1-1 association between the 2 classes rather than attributes. The String type is not available in UML4MBT. Strings are abstracted into enumeration literals.
- iv - **Operations:** They model the actions supported by an object. Class operations are used to model the methods that can be applied to instances of the owner class. They formalize activated actions on the system. An operation can be defined with parameters (input and output) that can be typed as Boolean, Integer, Enumeration Literal or Object. OCL preconditions and postconditions can be used to formalize the behavior of an operation. Moreover, operations are also used to observe results.
- v - **Enumeration classes:** They are composed only of literals. Enumerations are used to model static types like strings (SUT's output messages, user inputs, ...). For example, the following user passwords "cheval" and "erf58RO_\$!!" are respectively abstracted as literals USER_PASSWORD_1 and USER_PASSWORD_2, contained in an enumeration class called USER_PASSWORDS.

There is a close correlation between class diagrams and object diagrams: an object diagram is an instantiation of a given class diagram. In this sense, a class describes a set of objects and an association describes a set of links. Objects are instances of classes, links are instances of associations. *Object diagrams* in the CertifyIt approach are used to characterize both the initial state of the SUT and the objects that can be operated during a test scenario.

4.1.1.2/ OBJECT DIAGRAMS: INITIAL STATE

A *class diagram* is a graphical representation of a structural object-oriented program that can be instantiated. To model this instantiation, UML proposes object diagrams. In this diagram it is thus possible to define objects that are in fact instances of a class, and links between objects that are instances of associations.

Object diagrams serve 2 purposes in UML4MBT. On the one hand, they represent the initial state of the SUT. Objects and their dependencies are modeled in their state before interacting with the SUT. On the other hand, Object diagrams contain all objects when running the model because creation and deletion of objects are not allowed in the UML4MBT notation. Instead, these 2 actions are respectively simulated by creation and deletion of links between objects in the model. This way, object diagrams represent the initial state of the system and any objects to be used during its animation.

Object diagrams can be composed of the following elements:

- i - **Objects (class instances):** They are the concrete objects of the SUT that are used in the generated tests. They further represent the initial state of the SUT.

- ii - **Slots (*attribute instances*)**: They define the value of a attribute (state variable) at a given point. Slots must be valued.
- iii - **Links (*association instances*)**: They define the dependencies between objects in the initial state of the SUT.

It should be noted that in UML4MBT, class diagrams typically has a specific class, called **class under test**, which represents the SUT. This class may only be instantiated once, and this instance represents the overall system.

4.1.1.3/ STATE-MACHINE DIAGRAMS: DYNAMIC STRUCTURE

State-machine diagrams, also called Statecharts, provide an abstract description of the dynamic of one or more UML entities, in the form of a finite state automaton (UML formalism is a variant of Harel's statecharts [59]). Typically, state-machines describe classes behavior. They supplement a class diagram by specifying a system's behavior and state changes. They have been chosen in UML4MBT instead of other dynamic diagrams, such as use-case diagrams, mainly because they allow the specification of richer behaviors and provide better support for loops and alternative paths.

In UML4MBT, state-machines define the behavior of the SUT. Indeed, the SUT is modeled as a UML class, and the state-machine is contained by this class. In this sense, state-machines can only be associated to the class under test. Test cases are directly related to the possible execution traces from the state-machine.

A state-machine is a type of controller: it consists of a set of nodes connected by transitions. A node represents the SUT's state at a given point, and a transition expresses a SUT state change. A transition is triggered by an event of different types (method call, a signal, a timeout).

The following UML elements can be used in UML4MBT state-machines:

- i - **Initial State**: Designates the starting point of the state-machine. It must be unique.
- ii - **Final State**: Designates the end of an execution of the state-machine. There may multiple final states.
- iii - **Simple State**: Basic state, represents the SUT at a given point.
- iv - **Choice State**: Designates a point of choice. It heads the execution to different states depending on how is evaluated its precondition (onEntry).
- v - **Composite State**: Designates a state that contains a sequence of sub-states.
- vi - **External Transition**: Links 2 different states (a source state and a target state) together.
- vii - **Internal Transition**: Used when the firing of the transition does not lead to a state change.

4.1.1.4/ OBJECT CONSTRAINT LANGUAGE: CONSTRAINTS AND ACTIONS

A UML diagram on itself cannot model all the important aspects of a system. The need for example to define constraints associated with objects of a model lead to the use of a language to express conditions and constraints, called Object Constraint Language (OCL) [36]. OCL is a formal language for defining constraints on UML models, for instance preconditions and postconditions on operations, guards constraints on state-machine transitions, and invariant expressions on classes. These constraints, in the form of Boolean expressions, can specify invariant conditions that a model must verify, or specify requests to objects defined in a model. Thereby, OCL clarifies models. The notation, between natural language and mathematical language, was developed by the IBM Insurance division, and comes from the Syntropy method [18].

A subset of OCL, called OCL4MBT, has been defined within the CertifyIt approach for MBT purposes. The language serves 2 purposes within the approach. On the one hand, it is used to improve the precision of behavioral models by adding constraints to operation call (i.e. as a precondition) and transition triggers (i.e. as a guard). On the other hand, it is used to formalize the behavior of the SUT, in the form of actions that change the SUT's state. However, OCL is not an action language (nor procedural), but a declarative language. It only allows the description of constraints on elements of models. This problem is addressed in CertifyIt by providing a double interpretation of the language.

Thus, OCL is interpreted as a constraint language in operation preconditions, to express the execution condition of the operation, and in transition guards, to express the condition of activation of the transition. It is interpreted as an action language in operation postconditions to express the behavior of the operation, and in transition effects to express the related behavior of the state change expressed by the transition. As an example, the OCL expression `self.attribute=value` will be interpreted differently depending on whether the context is passive (preconditions and guards) or active (postconditions and effects). In a passive context, the expression is interpreted and evaluated as a standard Boolean expression. In an active context, it is interpreted as an assignment of the object attribute to the given value. Notice that OCL easily lends itself to this particular second interpretation; the context of an expression is deterministic and is comprehensive in the 2 cases. This particular but non-ambiguous interpretation of OCL makes it possible to use OCL as an action language for UML test generation models. The 2 own interpretations (passive and active) are exhaustively defined in [13].

Moreover, OCL4MBT is interpreted sequentially as opposed to OCL. Indeed, OCL is a constraint language without side effect: an OCL expression cannot change the state of the system. Thus, a variable used in an operation exists in the state before and the state after, and has no intermediate value during the execution of the operation. For variables in a postcondition, OCL allows reference to both its value at the beginning of the operation and at its end: the keyword `@pre` is used to refer to its initial value. However, because of the double interpretation of the language and the absence of real postconditions, this property has not been integrated to OCL4MBT. Interpretation of OCL4MBT constraints is therefore sequential, a variable may change of value several times during an operation processing and the final value of the variable, i.e. its value at the end of the operation processing, is the one that has been affected to it last.

Next sections present how test selection is performed from UML4MBT models within CertifyIt, how abstract test cases are generated, and how they are transformed into exe-

cutable scripts.

4.1.2/ TEST SELECTION

MBT makes use of test selection criteria to select the tests to be derived from the model. In CertifyIt, test selection is done differently depending on the tested characteristics.

For functional behavioral testing, test selection is built-in: the test generation engine derives test targets until all behaviors have been activated. For functional security testing, test selection is done by formalizing the test scenarios extracted from the test objective using high-level expressions called **test purposes**, to drive the test generation engine through the test model in order to derive test targets and generate test cases. We present each technique in the next sections, with an emphasis on test purposes since the PMVT approach relies on an augmented version of the test purpose language for the formalization of vulnerability test patterns.

4.1.2.1/ FUNCTIONAL BEHAVIORAL TESTING

Test selection for functional testing is built-in in Certify-It. It directly relies on the information captured by the test model. The goal is to activate all behaviors with one or several test cases.

A behavior is a sequence of guarded substitutions that define a state of the SUT. A guarded substitution is the assignment of a variable or set of variables of the SUT in a specific condition.

Structural behavioral coverage of the model exercises the functionalities of the system by directly activating and covering the corresponding operations. In the case of a UML4MBT model, a behavior can be:

1. An execution branch in an operation,
2. A transition, internal or external, of the state-machine,
3. An input or output action of a state of the state-machine.

As behaviors, the generated targets are sequences of guarded substitutions. The test target production rule in CertifyIt uses the *Decision / Condition Coverage* (D/CC) criterion.

To illustrate this criterion, consider an OCL4MBT expression (see Figure 4.2) defined as the postcondition of an operation of the SUT, and therefore interpreted as an action language.

Three behaviors can be extracted from this operation:

$$B_1 = [I \vee J](a = 3); (c = 4)$$

$$B_2 = [\neg(I \vee J)](b = 5); [X \vee Y](a = 1)(c = 4)$$

$$B_3 = [\neg(I \vee J)](b = 5); [\neg(X \vee Y)](b = 2)(c = 4)$$

Satisfying the D/CC criterion requires the production of 5 test targets:

$$D/CC(B_1) = [I](a = 3); (c = 4), [J](a = 3); (c = 4)$$

```

1 if I or J then
    a = 3
3 else
    b = 5 and
5     if X or Y then
        a = 1
7     else
        b = 2
9     endif
    endif and
11 c = 4

```

Figure 4.2 – OCL4MBT expression: D/CC coverage

$$\begin{aligned}
 D/CC(B_2) &= \left\{ \begin{array}{l} [\neg(I \vee J)](b = 5); [X](a = 1)(c = 4) \\ [\neg(I \vee J)](b = 5); [Y](a = 1)(c = 4) \end{array} \right. \\
 D/CC(B_3) &= \left\{ \begin{array}{l} [\neg(I \vee J)](b = 5) \\ [\neg(X \vee Y)](b = 2)(c = 4) \end{array} \right.
 \end{aligned}$$

For each test target, the test generation engine tries to find a path that reaches the target to compute an abstract test case.

4.1.2.2/ TEST PURPOSES IN CERTIFYIT

Recently, a new test selection technique has been introduced in CertifyIt to generate test cases for security components [11], typically Smart card applications and cryptographic components, by relying on operational **test purposes**.

When the tested characteristics of the SUT are security functionalities, the general test objective is to trigger each functionality of the system, in one or several contexts, to assess that security is properly enforced. A common test design technique to perform security testing consists of *invalid testing*: testing using input values that should be rejected by the component or system. When done manually, security test engineers extract invalid test scenarios from the test objective (that specifies each security functionality and their purpose), in which some security properties might be violated by an erroneous implementation. Then, they translate each scenario in one or several test cases. CertifyIt enables to automate the derivation of test targets from test scenarios by formalizing these scenarios using operational test purposes.

Test purposes enable to formalize one or several test scenarios extracted from the test objective, using regular expressions, by relying on the information captured in UML4MBT models. The main principle is to formalize test scenarios in terms of model states to reach and model operations to call. Moreover, test purposes can specify several contexts within which states should be reached and operations should be called, making it possible to formalize several test scenarios from a unique expression. Test purposes enable to drive the automated test generation on UML4MBT models in order to derive test targets tailored to the test objective.

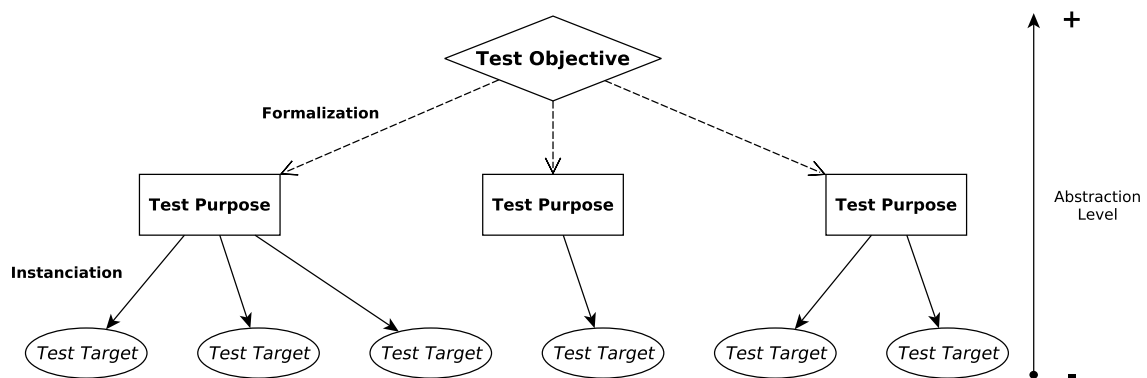


Figure 4.3 – Test Purposes workflow for the production of test targets

As stated in Figure 4.3, test purposes can be seen as an intermediary between the test objective and the test cases it implies. It is a 2-step process. The first step, which concerns the formalization of test scenarios, is manual and is the responsibility of test engineers. The second step, which concerns the derivation of test targets and the generation of test cases, is automated and entrusted to the test generation engine.

Test purposes are high-level textual expressions, based on a regular expression. They are defined using a dedicated notation called **Test Purpose Language** [11]. This is a textual language oriented toward test engineers: its semantics is based on regular expressions (see [39] and [32] for more details). In addition, it was designed to stay close to natural language in order to keep the formalized test scenario easily understandable, without prior test purpose language knowledge. Note that during this thesis, we have augmented the expressivity of the test purpose Language to enable the translation of vulnerability test patterns without information loss. The resulting grammar of the language is the content of Section 6.1.

A typical test purpose is composed of 2 main entities: iterators and stages. Stages define animation steps (states to reach and operations to call) that the test generation engine must activate. Iterators specify the various contexts within which stages must be activated. Thus, a typical test purpose has the following construction:

```
for_each Contexts
  activate stage1
  activate stage2
  activate stage3
...
```

We describe the creation of stages and iterators in the following sections.

STAGE CREATION

A test scenario often implies several steps. In test purposes, each test step is expressed with a *stage*. A test purpose is therefore composed of one or several stages, depending on the test scenario.

A stage is a directive towards the test generation engine that must be completed in order

to proceed to the next stage, if any, or to generate a test case, otherwise. A directive can request the call of one or multiple operations, the activation of behaviors or the reach of states. It can also specify restrictions, for instance prohibit the call to a specific operation. What makes stages powerful is that they specify the end-goal and the means to reach it, but the actual path (i.e. the exact sequence of actions) is computed by the test generation engine by animating the model.

Test purpose stages can contain up to 3 parts, thus:

- **The “CONTROL” part:** It defines the type of model entities (e.g., operations, states, behaviors) that can be used during the stage.
- **The “RESTRICTION” part:** It constrains the number of model entities that can be solicited in the current stage. This part is optional.
- **The “TARGET” part:** It defines the objective that must be reached at the end of the stage. This part is also optional.

A stage is always composed as follows:

```
use CONTROL RESTRICTION TARGET
```

If a test purpose contains multiple stages, then stage definitions are separated using the **then** keyword:

```
use CONTROL1 RESTRICTION1 TARGET1
then use CONTROL2 RESTRICTION2 TARGET2
then use...
```

The “CONTROL” part The “CONTROL” part specifies the type of model entities to be used in order to complete the stage. Several ways exist to select each of these model entities. It can be an operation chosen from a list of operations of the system:

- *any_operation*: Any system operation can be used during the current stage.
- *operation1*: Only the named operation can be used during the current stage.
- *operation1 or operation2 or ...*: Only the named operations can be used during the current stage.
- *any_operation_but operation1 or operation2 or ...*: The specified operations cannot be used during the current stage.

It can also be a behavior from a list of behaviors of the system:

- *any_behavior_to_cover*: Any behavior of the system can be used during this stage.
- *behavior_with_tags {REQ: req1, AIM : aim1}*: Only the behaviors of the system covering at least the specified tags can be used during this stage.
- *behavior_without_tags {REQ: req1, AIM : aim1}*: Only the behaviors of the system not covering the specified tags can be used during this stage.

- As for operations, it is possible to specify a list of behaviors, by separating each behavior with the **or** keyword.

An operation call can also be specified directly:

- *instance.operation1(value1, _)*: Only the designated operation from the specified instance can be called and its first parameter must take the value “value1”, the second parameter can take any value.
- As for operations and behaviors, a list of calls can be specified by separating each call with the **or** keyword.

The “RESTRICTION” part The length of each stage, in term of the number of animation steps, can be set. This part is optional. It is expressed as follows:

- *any_number_of_times*: The stage can involve as many animation steps (e.g., operation calls) as needed, zero included, to verify the condition contained its “TARGET” part.
- *at_least_once*: The stage can involve as many animation steps (e.g., operation calls) as needed, zero excluded, to verify the condition contained in its “TARGET” part.
- *i times*: The stage must involve exactly *i* animation steps.
- if no restriction part is specified, the stage should involve exactly one animation step (equivalent to the restriction *1 times*).

The “TARGET” part The TARGET part expresses a condition that must be verified at the end of the currently computed stage. It can be a state of the SUT or a behavior that must be activated by the last animation step of the stage. The definition of a system state that must be reached at the end of a stage is introduced by the **to_reach** keyword followed by an OCL constraint on a specified model instance, as follows:

```
use any_operation to_reach "self.status = ALL_STATUS::OK" on_instance sut
```

The example above represents a complete test purpose stage. It specifies that the test generation engine may use any of the operations from the model (but only one call from only one operation is allowed) to satisfy the OCL constraint, in the context of the *sut* instance.

ITERATOR CREATION

In order to create various contexts where states must be reached, or to activate several behaviors in a specific context, the language allows the creation of iterators. This enables to formalize several test scenarios from the same test purpose expression. Iterators are separated by a comma, and are expressed as follows:

```
for_each TYPE $Varname from DATA
```


Iterators can be created for many model entities: operations, behaviors, calls, enumeration literals, instances, and integer. Moreover, keywords used in the definition of stages to give instructions and restrictions to the test generation engine, such as *any_operation* or *behavior_without_tags*, can also be used to refine the set of model entities that will be iterated over.

Here is an example of an iterator over operations:

```
for_each operation $OP from any_operation_but op1 or op2 or ...
```

This iterator runs through all operations except *operation1* and *operation2* and for each allowed operation, makes the \$OP variable points to the current operation, and continue to the processing of the stages.

The value of variables defined by iterators can be used in test purpose stages. On the one hand, variables may be introduced in the CONTROL part as follows:

```
for_each operation $OP from any_operation_but op1 or op2
use $OP to_reach "self.status = STATUS::OK" on_instance INST
```

In this example, \$OP points to an operation of the model, obtained through an iterator, which must be used in order to satisfy the given OCL expression, which requires that the *status* attribute from the Class instance *inst* is valued with the enumeration literal "OK" from the enumeration "STATUS".

On the other hand, variables can also define the context of an OCL expression:

```
for_each instance $inst from inst1 or inst2 or inst3
use any_operation to_reach "self.status = STATUS::OK" on_instance $inst
```

In this stage, the OCL expression that represents the TARGET part is evaluated in the context of the instance contained in the \$inst variable, obtained through an iterator.

In the next section, we describe how test cases are generated from the derived test targets, depending on the tested characteristics (behavioral or security).

4.1.3/ TEST GENERATION

Test targets are derived from the model and transmitted to the test generation engine. For functional behavioral testing, there are as many test targets as there are behaviors to activate. For functional security testing, test targets are derived by unfolding each test purpose: the number of derived test targets, for a given test purpose, is equal to the number of possible combined iterator values. For instance, a test purpose with 2 iterators, the first with 3 elements [A, B, C] and the second with 2 elements [1, 2], leads to the derivation of 6 test targets ((A, 1); (A, 2); (B, 1); (B, 2); (C, 1); (C, 2)).

Once test targets have been derived, the test generation engine uses symbolic state exploration of the model (see [19] for more technical details) to cover each test target. A test case is produced for each reachable target. Test cases are sequences of model operation calls with parameters aiming to activate their corresponding test target from the initial state defined in the model.

The test generation engine can conclude on the reachability of the test targets in 3 ways:

- 1 - **Reachable:** There is at least one path in the model that activates the behavior /

trigger the security function.

- 2 - Unreachable:** There is no way to activate the behavior, or the evaluation of an OCL expression is undefined.
- 3 - Undetermined:** A test target is labeled as *undetermined* in the following 2 cases: (i) the status is temporary, the test target has not been processed yet (i.e. the test target is in its original state), or (ii) the status is final, the test generation engine could not find a path that reaches the test target within the allocated generation time.

Each time a path that reaches a given test target is found by the test generation engine, then this path constitutes a test case that covers the test target. Test cases are composed of 3 parts:

- 1 - The preamble:** It represents the sequence of operations calls, from the initial state to the state that allows to activate the behavior (for behavioral testing) or trigger the security function (for security testing) defined by the test target.
- 2 - The body:** It represents the effective activation of the behavior / triggering of the security function associated with the test target.
- 3 - The postamble *Optional*:** It represents the sequence of operations calls that brings the system back to its initial state. The postamble can be seen as a specific test target and thus can also be labelled as reachable, unreachable or undetermined. In this thesis, the postamble's role was to restore the initial state of the database.

Note that test cases embed both input data and intended results, which are obtained during the animation of the test model.

Next section describes the mechanism that enables to transform abstract test cases in test scripts, which are executable on the real system.

4.1.4/ TEST CONCRETIZATION, EXECUTION AND VERDICT ASSIGNMENT

Test cases computed with the CertifyIt technology are *abstract* by definition, since they have been derived from an abstracted view (an UML4MBT model) of the SUT. As a consequence, such test cases are not directly executable and they must be *concretized* for a validation of the real system.

The concretization activity consists of writing an adapter (i.e. a conformity table) between the abstract data from the model and their corresponding concrete data from the SUT. Abstract test cases are then transformed into test scripts that are executable on the SUT. This activity is therefore mandatory for test execution and verdict assignment. It enables to run test cases from the model on the real system to compare the real behavior of the system with what is expected and has been modeled.

Practically speaking, test cases concretization in CertifyIt consists of publishing abstract test cases, as well as additional information, in a standardized XML format. CertifyIt supports test publisher plugins, which use the XML file as a basis for the generation of executable test scripts in a given language. As an example, the most popular CertifyIt test

publisher plugin writes JUnit3 test cases. Test management is also targeted by publishing the generated tests in a format supported by test management and execution tools such as HP Quality Center⁵, and TestLink⁶.

4.2/ CERTIFYIT FOR VULNERABILITY TESTING

The CertifyIt approach had been initially designed to compute functional test cases by relying on a test model that captures both the input and output data for each operation call. Recently, a novel test selection technique has been introduced to compute functional security test cases by composing the test model with operational test purposes. Test purposes allow to formalize test scenarios by specifying a sequence of high-level steps called stages, which can be applied on several objects from the model, and in several contexts.

Such approach, when using a composition of test purposes and models, constitutes a relevant candidate for vulnerability testing of Web applications. On the one hand, the use of UML4MBT models enables to represent not only the structure of Web applications, but also their logic features and constraints as well as the various user roles and their corresponding privileges. On the other hand, test purposes can formalize test scenarios related to vulnerability testing, by translating informal vulnerability test patterns from catalogues such as MITRE and OWASP. It is then possible to use test purposes to drive the CertifyIt test generation engine through the model, in order to compute attack traces in the form of test cases (i.e. a sequence of operation calls). In addition, using CertifyIt for vulnerability testing implies automated test generation, and automated test execution and verdict assignment.

Although it is technically possible, using the CertifyIt approach in its current state for vulnerability testing would not be efficient nor effective.

First, the current *Test Purpose Language* lacks genericity. Indeed, functional security testing relies on a specification about the security properties of the SUT, and consequently the general test objective is tailored to these security properties and to the SUT. Vulnerability testing, on the other hand, relies on vulnerability test patterns. Such patterns are not linked to a particular application: they are defined at a higher level to be applicable on any Web application. Test purposes allow a certain freedom in the formalism of test scenarios, but they are too intricately bound to the associated model. It would require test engineers to specify each context with named objects from the test model. Since these objects vary from one model to another, test purposes would have to be constantly adapted to each Web application.

Second, the UML4MBT notation is too expressive for Web application vulnerability testing. It has been created to generate tests for a variety of systems, and therefore do not provide a predefined canvas for the modeling of Web applications. However, the genericity of test purposes can only be achieved if the elements and data types they interact with remains the same, regardless of the Web application that was modeled. Lastly, creating a UML4MBT model implies being familiar with the UML notation, the OCL notation, and the particularities of UML4MBT (especially the passive and active interpretations of OCL4MBT expressions). Web application development and Web application vulnerability

⁵<http://www8.hp.com/us/en/software-solutions/quality-center-quality-management/> [Last visited: August 2015]

⁶<http://testlink.org/> [Last visited: August 2015]

testing are fast-paced activities with strong time constraints, and the actors responsible of the vulnerability testing part of the development are usually not familiar with model-based techniques. Therefore, the trade-off for the CertifyIt approach for vulnerability testing purposes is not acceptable.

Based on the constraints and needs of penetration testers and vulnerability test engineers, an efficient MBT technique for Web application vulnerability testing should:

- Not involve new complicated programming languages and notations;
- Be fast and easy to deploy on any Web application;
- Reusable from one testing project to another as much as possible;
- Not bound to only one vulnerability category (e.g., technical, logical).

The PMVT approach that we designed during this thesis is built on top of CertifyIt and provide additions and adaptations to enable the generation of vulnerability test cases, which is not currently supported.

First, we propose to represent Web applications using a simple dedicated textual language that is straightforward to master. It provides a unified methodology on how to model Web applications strictly for Vulnerability testing purposes. Such model is then automatically transformed in an UML4MBT instance, therefore saving test engineers from having to learn a panel of new notations and concepts.

Second, we augmented the *Test Purpose Language* to make it generic w.r.t. the textual modeling language and to allow the computation of sophisticated attacks, which involve to take relationships between model entities into account and/or are linked to the business logic of the Web application. Consequently, test purposes are reusable from one testing project to another, without any required adaptation, and can lead to the computation of vulnerability test cases for many vulnerability types. Moreover, test engineers who are not interested in creating new test purposes to tackle other vulnerability types can rely on the existing catalog we created during this thesis.

In the next two chapters, we present PMVT and its general process, along with a full description of the dedicated textual notation and the augmented *Test Purpose Language*.

PMVT APPROACH: PROCESS AND MODELING NOTATION

Contents

5.1 PMVT Process	55
5.2 Running Example: Cuiteur	56
5.3 Web application Modeling Notation	60
5.3.1 DASTML: a Dedicated Language for Vulnerability Testing of Web applications	60
5.3.2 PMVT with UML4MBT	65
5.3.3 From DASTML to UML4MBT	78
5.4 Synthesis	80

This chapter introduces a Pattern-driven and Model-based Vulnerability Testing (PMVT) approach, as a generic solution for Web application vulnerability testing. PMVT is composed of several artifacts whose objective is to capture the logic of a concept, a technique or a system that plays a role in Web application security and vulnerability discovery.

We first describe the principles of the approach and the inner workings of the PMVT process. Then, we present the running example that helps illustrate each activity of the process. The chapter continues by giving information on the PMVT modeling notation, a domain-specific modeling language we designed called DASTML, which allows the textual modeling of Web applications in a simple manner. It captures structural and behavioral aspects of Web applications by relying on a restricted set of keywords. Test generation has been made possible by transforming DASTML in UML4MBT and sending the result to CertifyIt.

5.1/ PMVT PROCESS

The PMVT process, depicted in Figure 5.1, is composed of the 4 following activities:

- ① The **Modeling** activity aims to design a test model that captures the behavioral aspects of the SUT in order to generate consistent sequences of stimuli, from a behavioral and structural point of view.

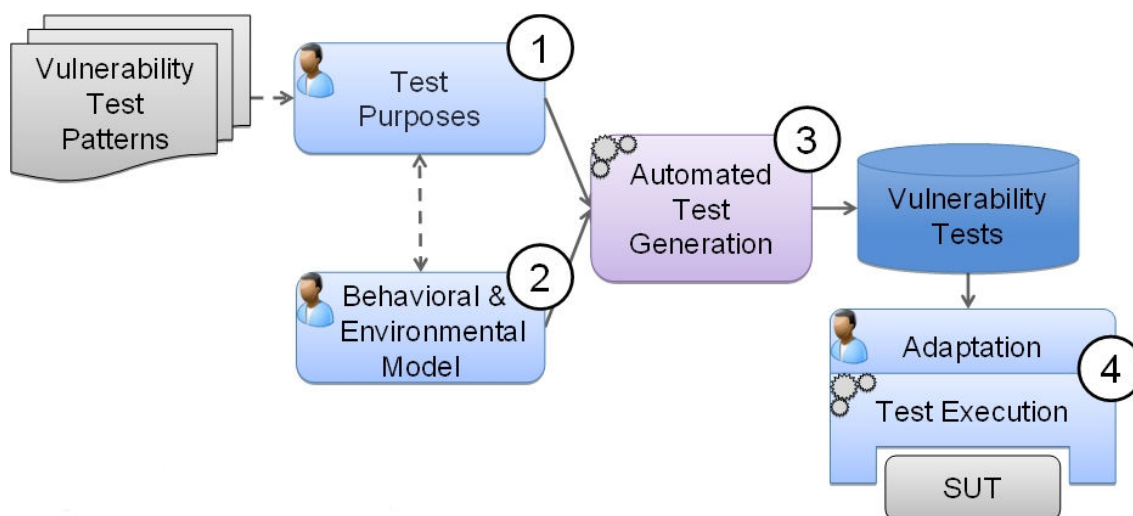


Figure 5.1 – PMVT: General Process

- ② The **Test Purposes** activity consists of formalizing an operational test procedure from vulnerability test patterns that the generated test cases have to cover. Test purposes are generic and can be used on any Web application. A catalog of test purposes has been designed during this thesis to tackle several high-profile vulnerabilities that test engineers can import into their testing project as is.
- ③ The **Test Generation** activity consists of applying the test purposes on the model, using the CertifyIt test generation engine, to automatically produce vulnerability abstract test cases.
- ④ The **Concretization, Test Execution and Verdict Assignment** activity aims to (i) translate abstract test cases into executable test scripts, (ii) automatically execute these scripts on the SUT, and (iii) automatically compare SUT's responses with expected results in order to assign a test verdict and detect vulnerabilities.

All these activities are supported by a dedicated toolchain, based on an existing model-based testing software named *CertifyIt* [45]. The components of the PMVT toolchain are described in Chapter 7.

In the following section, we describe the running example whose purpose is to illustrate the deployment of each PMVT artifact.

5.2/ RUNNING EXAMPLE: CUITEUR

Cuiteur is a simplified copycat of Twitter¹. It has been created for educational purposes at the University of Franche-Comté: students were asked to develop a similar Web application based on the course they were given. It is a PHP / MySQL web site, run on an Apache Server. The Web application has been fully developed from scratch, and thus

¹<http://www.twitter.com> [Last visited: August 2015]

does not rely on any framework (e.g. Symfony, Zend, and so on) nor any third-party library.

Users must register and authenticate to access the application. Once logged in, they can read and post short messages called **cuits**, of a maximum length of 255 characters. Messages are public, and an authenticated user can view all the **cuits** from any other user. Users can follow other users to automatically retrieve and display the content they post. They can repost (**recuit**) other users' **cuits**. They can also make mention of a certain user in a cuit by typing the character '@' followed by the user's nickname (e.g., @yoda). Hashtags can also be inserted in cuits, by typing the character '#' followed by the hashtag name (e.g., #XSS).

Here are the various features of the application:

- **Authentication and registration system.** As opposed to Twitter, Users of Cuiteur must be logged in to access the application's features, and this requires a registration. To do so, users should provide a valid email address, a valid password, plus their name and birthday. A valid new registration results in automatically logging in the user.

The figure consists of two screenshots of the Cuiteur web application. Screenshot (a) shows the login page with the title 'To access Cuiteur, you must log in:'. It features two input fields labeled 'Login' and 'Password', a 'Log In' button, and links for 'No Account? Register!' and 'Not sure about registering? Visit our presentation page and get overwhelmed by Cuiteur's amazing features.' Screenshot (b) shows the registration page with the title 'To register, please provide the following information:'. It features five input fields labeled 'Choose a nickname', 'Choose a password', 'Password confirmation', 'Name', and 'Email address', and a 'Register' button.

(a) Log In

(b) Registration

Figure 5.2 – Cuiteur: Login and Registration System

- **Cuits posting system.** This is the core feature of Cuiteur. Users can post **cuits** from the home page, and retrieve their cuit feed that consists of the last cuits from their subscription(s). Users can **recuit** (repost) posted messages from other Users and reply to a cuit. It is also possible to create trends with the use of hashtags. The more active trends are featured on the home page.
- **Account info.** This page allows users to see and modify their account information. It is not accessible unless the user is logged in. The first form on the page allows to modify personal information: (i) firstname and lastname, (ii) birthday, (iii) City, (iv) Mini-biography. The second form concerns personal email address and website. The last form on the page allows to modify connection information: (i) password and confirmation, (ii) profile picture, (iii) display or hide photo.
- **User search.** The search page displays a text input, allowing users to search for other users. The result is a set of mini user profiles, providing a link to the user's complete profile page, and keys information such as the number of followers, subscriptions, posts, mentions.



Figure 5.3 – Cuiteur: Home Page

- **Administrator Panel.** A restricted set of Users has access to the administrator panel. To get access, authorized users need to re-authenticate by providing their account password. A successful authentication redirects users to the panel where they can edit and delete user accounts. Moreover, authenticated administrators can go back to their regular user environment while preserving administration rights, and therefore can edit and delete cuits from any user.

Cuiteur has been made permissive to various vulnerability kinds on purpose. We detail each vulnerability below:

- **CT-RXSS-01: Reflected XSS.** On the user search page, the content of the “search” field is rendered back on the next page without sanitization.
- **CT-SXSS-02: Stored XSS.** On the profile page, the content of the “usBio” field is stored in the database and rendered back to clients without proper sanitization.
- **CT-MSXSS-03: Multi-step XSS.** On the authenticated home page, the content of the “txtMessage” field (used for post **cuits**) is stored in the database and displayed without proper sanitization. However, users must first submit the message for preview, and then validate or dismiss it.
- **CT-JQXSS-04: Stored XSS behind JQuery.** On the authenticated home page, the content of the “txtMessage” field (used for post **cuits**) is stored in the database and displayed unsanitized.

Account parameters

On this page, you can add, delete, and modify information about yourself and your account.

Personal Information

Name: Yoda

Birthday: 1 janvier 1916

City: Dagobah

Mini-bio: When nine hundreds years like me you will be, less in shape your body will be.

Submit

Account Information

Email Address: yoda@nowhere.com

Website: http://en.wikipedia.org/wiki/Yoda


Submit

Figure 5.4 – Cuiteur: Profile Page

Search for Users

chuck Search

Search Results

 **nono** Chuck Norris
5 Cuits - 3 Mentions - 1 Follower - 3 Subscriptions

follow

Submit

Figure 5.5 – Cuiteur: User Search Page

- **CT-SQLI-05: SQL injection.** On the profile page, the content of the “usBio” field is stored in the database without proper sanitization.
- **CT-SQLI-06: SQL injection.** On the user search page, the content of the “search” field is not sanitized against SQL Injections.
- **CT-CSRF-07: CSRF.** All forms of the Cuiteur application are vulnerable to CSRF.
- **CT-PE-08: Privilege escalation.** The access restriction to the “delete user” action, supposedly restricted to administrators, is not enforced.

We use this running example in the next sections to illustrate the deployment of the PMVT approach and show how each artifact is put into practice on a simple use case. To that end, we have defined a restricted subset of the application to keep the description of the PMVT artifacts simple. The subset only features the authentication and cuits posting systems, and the administration home page with links to delete given users.

5.3/ WEB APPLICATION MODELING NOTATION

As for every MBT approach, the modeling activity consists of designing a test model that will be used to automatically generate abstract test cases. The PMVT approach, built on top of CertifyIt, requires a UML4MBT model. However, as stated in Chapter 5, UML4MBT is not usable as is. Therefore, we propose a simpler notation that is dedicated to Web application vulnerability testing. We created a textual domain-specific modeling language, called DASTML, which allows to represent the structure and logic of Web applications using a restricted set of keywords. Then, DASTML models are transformed into UML4MBT models, based on a set of transformation rules, in order to generate abstract test cases using CertifyIt.

In the following subsections, we first present the DASTML notation along with its entities and relationships. We then explain the content of a UML4MBT model adapted to meet PMVT's needs. We finally describe the transformation rules that enable to transform DASTML models into UML4MBT models.

5.3.1/ DASTML: A DEDICATED LANGUAGE FOR VULNERABILITY TESTING OF WEB APPLICATIONS

To ease and accelerate the modeling activity, which is known to be time consuming, we have developed a Domain Specific Modeling Language (DSML), called *DASTML* for Dynamic Application Security Testing Modeling Language, to represent the structural and behavioral properties of Web applications. 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 plugin integrated to the CertifyIt modeling environment.

Web applications share a common structure. A typical Web application is a set of pages (in the case of a standard Web application), or a set of states (in the case of a Rich Internet Application). Each page or state contains possible interactions with users: anchors, forms, buttons, etc. A click on an anchor, a form submission, or a click on a button may take the user to another page / state. User interactions may require special privileges to be executed, and sometimes require users to provide data (e.g., Web Forms). Input data may be rendered back in another page, in another context or session.

DASTML enables to represent this information, and only this information. It is sufficient to generate vulnerability test cases for the 4 vulnerability types described in Chapter 2.

Note that the objective of DASTML is to provide a simple way to capture the structure of Web applications. All the model artifacts that are necessary to perform attacks (injections operations, verdict assignment operations, etc.) are automatically generated and integrated to the target UML4MBT model (see Section 5.3.2). This way, PMVT users don't

have to think about **how** to conduct attacks, but only **where** to conduct attacks, by solely representing the relevant parts of the Web application that need to be tested.

The language grammar is shown on Figure 5.6. We provide a short description of each DASTML entities in the next paragraphs. Then, we illustrate this language by discussing the DASTML model of Cuiteur.

```

model ::= PAGES { ( page ) * }
page  ::= ident init? { action_list? navigation_list? restriction_list? }
init   ::= : INIT
action_list ::= ACTIONS { action ( , action ) * }
navigation_list ::= NAVIGATIONS { navigation ( , navigation ) * }
restriction_list ::= RESTRICTIONS { restriction ( , restriction ) * }
navigation ::= ident nav
nav        ::= → ident
action     ::= ident session_type? data_list? nav?
session_type ::= : restriction
data_list  ::= ( data ( , data ) ? )
data       ::= ident value? output_usage?
value      ::= = ident
output_usage ::= ⇒ ident
            | ⇒ ident ( , ident ) ?
restriction ::= ANONYMOUS | USERS | ADMINS
ident       ::= _ 'A'..'Z' (( '_' ) ? 'A'..'Z' | '0'..'9') * _

```

Figure 5.6 – Syntax of the DAST Modeling Language

Page A page is an output from the server, in the form of a HTML document (or alike format). In PMVT, we differentiate pages based on the technique presented in [21]. Hence, a page is considered unique if its set of control points (actions and navigation links) is unique, or if its set of outputted inputs is unique. A Page instance may contain *Navigations*, *Actions*, and *Restrictions*.

```

"PAGE_NAME" {
  ACTIONS {...}
  NAVIGATIONS {...}
  RESTRICTIONS {...}
}

```

In addition, we represent the initial page using a suffix, `:INIT`, appended to the page name (e.g., `"PAGE_NAME":INIT`). The purpose of this information is to establish the initial state of the model, which is mandatory for UML4MBT models.

Restrictions To represent the different areas of a Web application, each modeled page has restrictions. A restriction set defines which user roles has access to the page. As such, it is implied that all actions owned by the page are submitted to the same restrictions. As of now, users roles have been hardcoded in the language. We consider 3 possible user roles: *ANONYMOUS*, *USERS*, and *ADMINS*.

```
"PAGE_NAME" {
    RESTRICTIONS {USERS, ADMINS}
    ...
}
```

Navigations Navigation entities represent mechanisms that take users from one Page to another. Navigations represent anchors, buttons, element hovering, any action that will trigger an HTTP request to the server that ask for another page. These entities enable to create a dynamic map of the pages of the Web application.

```
"NAVIGATIONS" {
    "NAV_NAME"
        → "TARGET_PAGE",
    ...
}
```

Actions As opposed to Navigations, Actions represent user interactions that can modify the internal state of the application, and/or carry data.

It can be the filling of a Web form, a click on a button, the hovering of a graphical control element, and so on. For example, consider the action of login to a Web application. It generally consists of a form composed of 2 fields (login and password) and a submit button. The submission of the form will create a user session, and will provide access to authenticated area.

```
"ACTIONS" {
    "ACTION_NAME"
        → "TARGET_PAGE",
    ...
}
```

Actions may carry data, i.e. user inputs. Depending on the nature of the action, user inputs can be form fields, URL parameters, cookie variables, etc. We represent user inputs as a key-value pair inside the owning action, thus:

```
"ACTION_NAME"
    ("INPUT_NAME" = "INPUT_VALUE",
    ("INPUT2_NAME" = "INPUT2_VALUE")
        → "TARGET_PAGE",
    ...
```

Sometimes, the value of a user input is rendered back by the application in other pages. The detection of XSS vulnerabilities becomes complex when the output pages are in another part of the Web application, possibly only accessible with another session type. Typically, scanners struggle with this issue and may miss vulnerabilities. In PMVT, we represent input values resurgences with a double arrow (e.g., \Rightarrow), which points to the set of pages that output the input value, such as:

```
"INPUT_NAME" = "INPUT_VALUE"  $\Rightarrow$  {"PAGE1", "PAGE2"}
```

Finally, actions are responsible for session type changes. In the initial state of the Web application, it is implied that no authentication has occurred, therefore the initial session

type is defined as *ANONYMOUS*. An action that leads to a session type change, such as login or registration actions, is represented using a suffix that corresponds to the new session type. For instance, the action "LOGIN":USERS implies a session type change, from *ANONYMOUS* to *USERS*.

DASTML ILLUSTRATED: REPRESENTATION OF CUITEUR

The DASTML model of Cuiteur is as follows:

```

1  PAGES {
2    "HOME":INIT {
3      ACTIONS {
4        "LOGIN":USERS ("ACCOUNT_LOGIN" = "A_LOGIN_2",
5          "ACCOUNT_PASSWORD" = "A_PASSWORD_2")
6        -> "HOME_LOGGED_IN",
7        "LOGIN":ADMINS ("ACCOUNT_LOGIN" = "A_LOGIN_1",
8          "ACCOUNT_PASSWORD" = "A_PASSWORD_1")
9        -> "HOME_LOGGED_IN"
10     }
11    NAVIGATIONS {
12      "GOTO_REGISTRATION"
13      -> "REGISTRATION"
14    }
15    RESTRICTIONS {ANONYMOUS}
16  }
17
18  "HOME_LOGGED_IN" {
19    ACTIONS {
20      "POST_CUIT" ("CUIT_POST" = "CUIT_POST_1" => "HOME_LOGGED_IN")
21      -> "HOME_LOGGED_IN",
22      "LOGOUT":ANONYMOUS
23      -> "HOME"
24    }
25    NAVIGATIONS {
26      "GOTO_PROFILE"
27      -> "PROFILE"
28    }
29    RESTRICTIONS {USERS, ADMINS}
30  }
31
32  "REGISTRATION" {
33    ACTIONS {
34      "REGISTER":USERS ("ACCOUNT_LOGIN" = "A_LOGIN_3",
35        "ACCOUNT_PASSWORD" = "A_PASSWORD_3",
36        "ACCOUNT_PASSWORD_CONF" = "A_PASSWORD_CONF_3",
37        "ACCOUNT_NAME" = "A_NAME_3" => {"HOME_LOGGED_IN", "PROFILE"},
38        "ACCOUNT_EMAIL" = "A_EMAIL_3" => "PROFILE")
39      -> "HOME_LOGGED_IN"
40    }
41    RESTRICTIONS {ANONYMOUS}
42  }
43
44  "PROFILE" {
45    NAVIGATIONS {
46      "GOTO_HOME_LOGGED_IN"
47      -> "HOME_LOGGED_IN"
48    }
49    RESTRICTIONS {USERS, ADMINS}

```

```

51 }
52
53 "ADMIN_PANEL_LOGIN" {
54   ACTIONS {
55     "ADMIN_LOGIN" ("ACCOUNT_PASSWORD" = "A_PASSWORD_1")
56       -> "ADMIN_PANEL"
57   }
58   NAVIGATIONS {
59     "GOTO_HOME_LOGGED_IN"
60       -> "HOME_LOGGED_IN"
61   }
62   RESTRICTIONS {USERS, ADMINS}
63 }
64
65 "ADMIN_PANEL" {
66   ACTIONS {
67     "DELETE_USER" ("ACCOUNT_LOGIN" = "A_LOGIN_2"),
68     "ADMIN_LOGOUT" :USERS -> "HOME_LOGGED_IN"
69   }
70   NAVIGATIONS {
71     "GOTO_HOME_LOGGED_IN"
72       -> "HOME_LOGGED_IN"
73   }
74   RESTRICTIONS {ADMINS}
75 }
76 }

```

In this model, we represented 6 pages. The “HOME” page is the initial page of the Web application (i.e. the page sent to users when they send an HTTP request to the root URL). The page features 2 actions, both are called “LOGIN” and are about authenticating users. The first has a *USERS* suffix, which means that if a user provides the credentials *A_LOGIN_2* / *A_PASSWORD_2*, he/she will authenticate as a standard user. The second action has an *ADMINS* suffix and authenticates users as administrators, given the right credentials. From the “HOME” page, it is possible to access the “REGISTRATION” page by triggering the navigation link called “GOTO_REGISTRATION”. This page is only accessible to *ANONYMOUS* users, meaning that once users have logged in, they cannot access this page anymore, unless they trigger the “LOGOUT” action (which has an *ANONYMOUS* suffix) from the “HOME_LOGGED_IN” page.

The “REGISTRATION” page has one action, “REGISTER”, which contains 5 valued user inputs. When users trigger the action, they will be taken to the “HOME_LOGGED_IN” page, and will be authenticated with the credentials they provided for registration. Moreover, the value of the “ACCOUNT_EMAIL” input is rendered back on the “PROFILE” page. Likewise, the “ACCOUNT_NAME” input is rendered back on 2 other pages: “HOME_LOGGED_IN” and “PROFILE”.

In the next section, we detail how DASTML models are represented using UML4MBT, and we describe the additional material, automatically generated, which enable to generate vulnerability test cases.

5.3.2/ PMVT WITH UML4MBT

Once designed, DASTML models are transformed in UML4MBT as 4 diagrams (2 class diagrams, one object diagram and one state-machine) in order to generate vulnerability test cases using CertifyIt.

To ensure the genericity of test purposes and make possible the transformation of DASTML models in UML4MBT models, we have specialized UML4MBT to meet PMVT's need. In the original use of UML4MBT (i.e. for functional testing), class diagrams represent the SUT, object diagrams represent its initial state, and state-machines represent its dynamic. PMVT, on the other hand, relies on 2 class diagrams:

- A generic class diagram that captures the general structure of Web applications. It can be seen as the UML4MBT equivalent to the DASTML grammar.
- A specific class-diagram that captures operations and datatypes related to the Web application under test.

Then, an object diagram instantiates both class diagrams to represent the application and its initial state, and a state-machine represents its pages map.

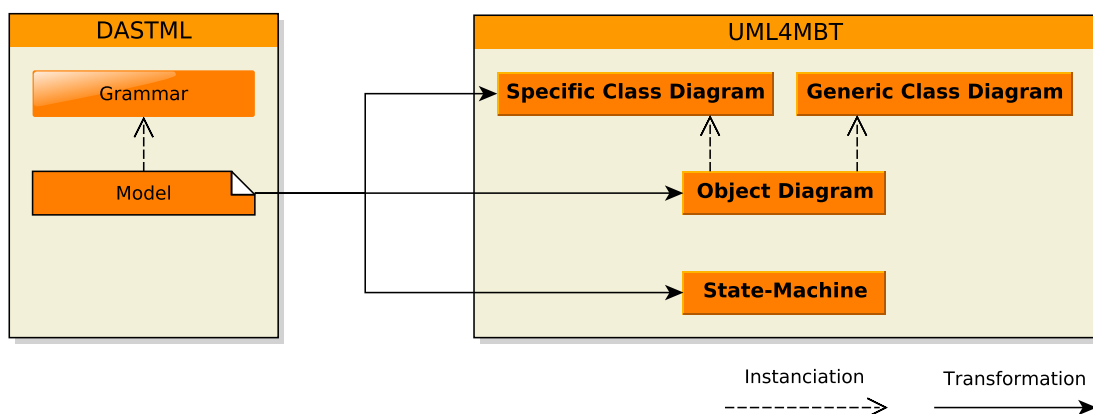


Figure 5.7 – From DASTML to UML4MBT

Therefore, as stated in Figure 5.7, DASTML models are transformed in a UML4MBT specific class diagram that connects to the generic class diagram, an object diagram that instantiates both class diagrams, and a state-machine contained in the main class of the specific class diagram.

In the next subsections we detail each diagram, with a strong emphasis on the generic class diagram as it constitutes the core of PMVT models. We end this section with modeling concepts that were created as part of the PMVT notation to enable the computation of sophisticated attacks.

5.3.2.1/ GENERIC CLASS DIAGRAM: PMVT METAMODEL

We propose to use a UML4MBT class diagram to describe the generic structure of Web applications. This class diagram is generic and can be applied to any Web application. In

some extent, it can be qualified as metamodel for Web applications structure, and is the UML4MBT equivalent to the DASTML grammar.

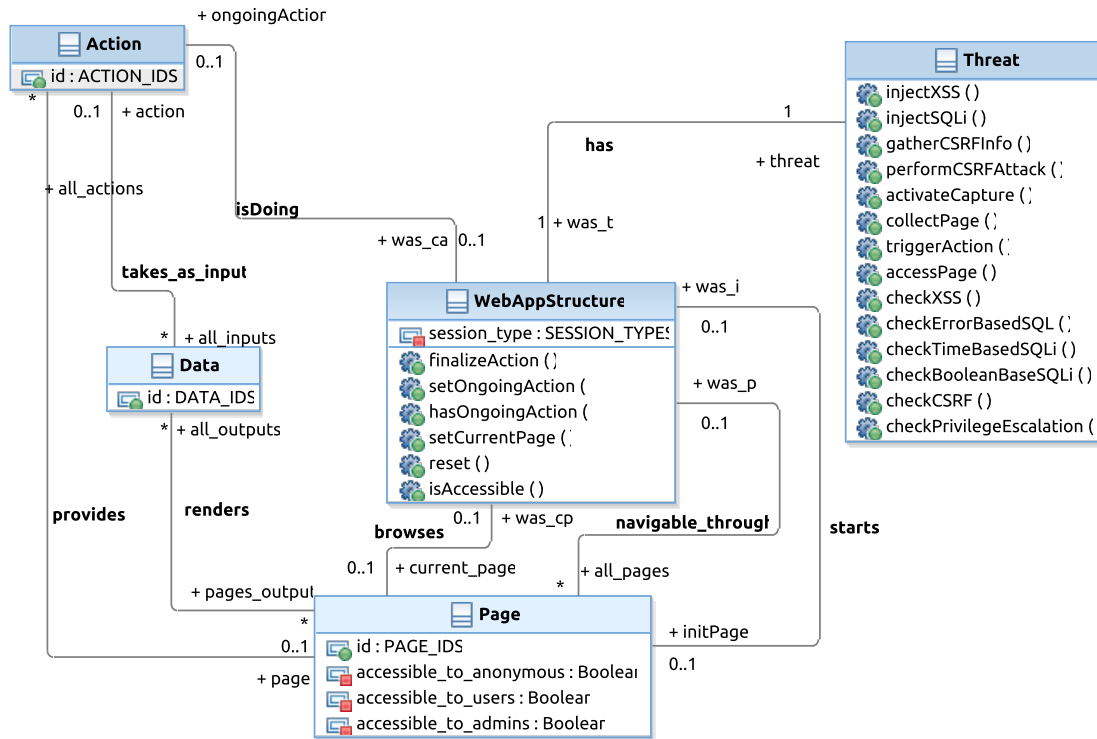


Figure 5.8 – PMVT: Generic Class Diagram

Figure 5.8 shows the generic class diagram as used in PMVT. Note that this representation of Web applications only allows the simulation of a single user - system interaction. Therefore, it can be seen as a model of an application viewed by a particular user. We describe each entity in the next paragraphs.

Datatypes There are 4 generic datatypes that serves for the identification of class instances, and their values are used in OCL expressions to refer to a particular object. These datatypes are represented as Enumerations: (i) *PAGE_IDS* contains all the Page identifiers, (ii) *ACTION_IDS* contains all the Action identifiers, (iii) *DATA_IDS* contains all the Data identifiers, which represent user inputs, and (iv) *SESSION_TYPES* contains all the user roles identifiers.

The use of such datatypes is necessary for the identification of a particular Class instance among the object diagram. Indeed, we cannot refer directly to the instance name because String manipulation is not implemented in CertifyIt. Therefore, every object in a PMVT test model can be identified using enumeration literals, which are affected to the *id* slot of *Pages*, *Actions* and *Data* instances. Note that enumeration literals from these datatypes are application-dependent and must be defined in the specific class diagram.

In contrast, session types are hard-coded in the generic PMVT class diagram as 3 roles defined as enumeration literals: (i) *ANONYMOUS* which represents non-authenticated

users, (ii) *USERS* which represents standard authenticated users without any administration rights, and (iii) *ADMINS* which represents users that have administrator rights.

WebAppStructure This class is the core of the static structure of the WAUT. It contains all the pages of the Web application under test (via the *navigable_through* association), and is linked to the *Threat* class with the *has* association.

The *WebAppStructure* class has one attribute named *session_type*, of type *SESSION_TYPES*, which defines the current status of the active session (anonymous, authenticated, admin). Section 5.3.2.5 provide more information on session handling in PMVT. Moreover, the *WebAppStructure* class contains additional information about the current state of the application, thanks to 2 associations: (i) the *browses* association is linked to the *Page* class and represents the current page displayed to the user, and (ii) the *isDoing* association is linked to the *Action* class and represents the current ongoing Action that the user is doing. It is also linked to the *Page* class with the *starts* association, which symbolizes the initial page displayed to the user (as represented by the *:INIT* suffix in DASTML).

Several operations are owned by this class, which we can separate in 2 categories. The first category contains private operations that cannot be directly called by the test generation engine but are used in OCL pre and postconditions. Their purpose is to factorize OCL expressions that are extensively used (e.g., to change the current page):

- **setOngoingAction(*ACTION_IDS newActionId*)**. This operation enables to change the current ongoing action. It takes one parameter, *newActionId*, which is an enumeration literal of type *ACTION_IDS* and contains the new ongoing action *id*.

This operation contains an OCL postcondition that uses the value contained in *newActionId* to retrieve the corresponding *Action* instance and creates a *is_ongoing* link between that instance and the *waut* instance:

```
POST: self.ongoingAction =
self.current_page.all_actions->any(a:Action|a.id=newActionId)
```

- **result ← hasOngoingAction()**. This is a Boolean operation, with one return parameter called *result*, which purpose is to check whether there is an action ongoing. This check is done using an OCL postcondition that puts the result of the Boolean expression in the *result* return parameter:

```
POST: result = not self.ongoingAction.ocIsUndefined()
```

- **setCurrentPage(*PAGE_IDS newPageId*)**. This operation enables to change the current page displayed to the user. It takes one parameter, *newPageId*, which is an enumeration literal of type *PAGE_IDS* and contains the new page *id*.

This operation contains an OCL postcondition that uses the value contained in *newPageId* to retrieve the corresponding *Page* instance and creates a *browses* link between that instance and the *waut* instance:

```
POST: self.current_page = self.all_pages->any(p:Page|p.id=newPageId)
```

- **isAccessible(*SESSION_types session, PAGE_IDS pageId*)**. This a Boolean operation which purpose is to check whether a given session type can access a given page. It takes 2 parameters: *session*, valued with a literal from *SESSION_TYPES*

and *pageId*, valued with a literal from *PAGE_IDS*.

This operation contains an OCL postcondition that compares back to back the value of the Boolean attributes of the page (i.e. its restrictions) with the given session type:

```
POST: let page = self.all_pages->any(p:Page|p.id=pageId) in
if (
    (page.accessible_to_anonymous and
     session_type = SESSION_TYPES::LIT_ANONYMOUS)
or
    (page.accessible_to_users and
     session_type = SESSION_TYPES::LIT_USERS)
or
    (page.accessible_to_admins and
     session_type = SESSION_TYPES::LIT_ADMINS)
) then result = true
else result = false
endif
```

For instance, if the page is accessible to authenticated users (*page.accessible_to_users* = true and the value of *session_type* is *LIT_USERS*, this operation will return *true*.

The second category contains operations that can be called by the test generation engine and their purpose is to enable sophisticated attacks:

- **finalizeAction()**. The *finalizeAction* operation serves a purpose in the conducting of attacks (as detailed in Section 5.3.2.5). Its semantics is about submitting forms and clicking on links, in order to make sure that a system's unattended behavior (e.g., crash or error message) is because of the attack, and not because of any side effects (e.g., validation mechanisms of other fields in case of a Web form).
- **reset()**. The purpose of this operation is to put the system back in its initial state. It is used for attacks that require to conduct a given sequence of steps several times and compare the different outcomes (e.g., for Time-Based and Boolean-Based SQL Injections). The OCL postcondition responsible of reinitializing the model contains 3 statement: first we remove any ongoing action, second we initialize the session, and third we define the initial page as the current page:

```
POST: let was = self.webAppStructure in
was.ongoingAction.oclIsUndefined() and
self.session_type = SESSION_TYPES::LIT_ANONYMOUS and
was.current_page = self.initPage
```

Threat This class models the potential attacks that can be applied to the WAUT. It contains 3 types of operations, all required to conduct attacks. First, there are information collecting operations:

- **gatherCSRFInfo()**. Used for CSRF attacks to gather information about the user action under test, namely the form or link used to submit data. Because this operation must be called between the data filling and data submission, it contains the following OCL precondition:

<pre>PRE: self.was_t.hasOngoingAction()</pre>

This condition means that this operation cannot be called unless there is an action ongoing. For more information about action handling in PMVT, report to Section 5.3.2.5.

- **collectPage.** Used for Privilege Escalation attacks to gather information about the page under test, that is to say its content, for later comparison with the result of the attack.
- **activateCapture.** Dedicated to Privilege Escalation attacks that focus on triggering an access-controlled function.

Second, there are attack operations:

- **injectXSS().** Removes the nominal value from the user input under test and replaces it with an XSS vector.
- **injectErrorBasedSQLi() / injectTimeBasedSQLi() / injectBooleanBasedSQLi().** Removes the nominal value from the user input under test and replaces it with an SQL vector.
- **performCSRFAttack().** Makes use of the information gathered to replicate the form or link on an external server and submit it.
- **triggerAction() / accessPage().** Used for Privilege Escalation attacks to browse-force a page that is supposed to have a restricted access.

Finally, the *Threat* class contains all the verdict assignment operations:

- **checkXSS().** Analyzes the current page to check whether the injected vector has been sanitized, removed, or used verbatim.
- **checkErrorBasedSQLi() / checkTimedBasedSQLi() / checkBooleanBasedSQLi().** The error-based SQLi operation checks whether the application displayed a DBMS error message, the time-based SQLi operation measures the response time between 2 injections, and the Boolean-based SQLi operation compares the results of 3 SQL Injections.
- **checkCSRF().** Compares the response page from the nominal completion of the action under test with the attack response page to check whether the action was completed by the WAUT, even if the data involved in the action were submitted from an outside server.
- **checkPrivilegeEscalation().** Compare the response page obtained through a nominal workflow with the response page obtained through the attack.

Note that all these operations (except for *gatherCSRFInfo()*) does not contain OCL pre and postconditions, and have no effect on the model: their purpose is to automate the test oracle. Therefore, the test generation engine cannot automatically call them to compute attack traces since they do not modify the state of the model. Test purposes are responsible for calling these operations, which is fully described in Section 6.1.

Page This class represents the various pages of the WAUT as static elements. It contains 4 attributes:

- **id: PAGE_IDS:** Contains the identifier of the page.
- **accessible_to_anonymous: Boolean:** Defines if the page is accessible to non-authenticated users.
- **accessible_to_users: Boolean:** Defines if the page is accessible to authenticated users.
- **accessible_to_admins: Boolean:** Defines if the page is accessible to authenticated users with administrator rights.

A *Page* instance owns Actions (e.g., web forms or anything that modifies the WAUT state variables) through the *provides* association, and is potentially connected to *Data* instances (user inputs) if this page uses their value as output, through the *renders* association. Section 5.3.2.5 details the page handling and session handling mechanisms in the PMVT notation.

Action We consider *Action* any user interaction that will have a consequence on the WAUT. It can be the filling of a Web form, a click on a button, the hovering of a graphical control element, and so on.

The *Action* class owns an attribute, called *id* for identifier, which can be set with literals from the *ACTION_IDS* enumeration.

Moreover, the class is connected to the *Data* class with the *takes_as_input* association, which symbolizes the user inputs involved in the completion of the action. It is also connected to the *WebAppStructure* class with the *is_doing* association, and with the *Page* class with the *provides* association.

Data The *Data* class represents user-supplied inputs. This class has one attribute, called *id* for identifier, which can be set with literals from the *DATA_IDS* enumeration.

Data instances are owned by an action through the *takes_as_input* association, and own output pages through the *renders* association.

5.3.2.2/ SPECIFIC CLASS DIAGRAM: APPLICATION-DEPENDENT INFORMATION

To complete the transformation of DASTML models, PMVT uses a specific class diagram to represent application-dependent information such as navigational and logical operations, as well as datatypes and values. Application-dependent operations are contained in a class that represent the overall Web application, and datatypes are represented with Enumerations. We present these entities in the next paragraphs.

WAUT The *WAUT* class, whose name stands for Web application Under Test, represents the overall Web application, in a logical point of view. As such, it contains all logical operations, e.g. *LOGIN(...)*, *REGISTER(...)*, or *BUYITEM(...)*, and navigational operations such as *GOTO_PROFILE()* or *GOTO_REGISTRATION()*. These operations are

application-dependent and correspond to actions and navigations of the DASTML model. The class is linked to the *WebAppStructure* class (from the generic class diagram).

Specific Datatypes and values First, the specific class diagram enables to specify all values from the generic datatypes *PAGE_IDS*, *ACTION_IDS* and *DATA_IDS*. Second, it contains all specific datatypes and their possible values. A specific datatype represents a category of data, such as login credentials, passwords, birthdates, pictures identifiers, and so on. These data are used to provide nominal data to Web forms and URLs.

PMVT SPECIFIC CLASS DIAGRAM OF CUITEUR

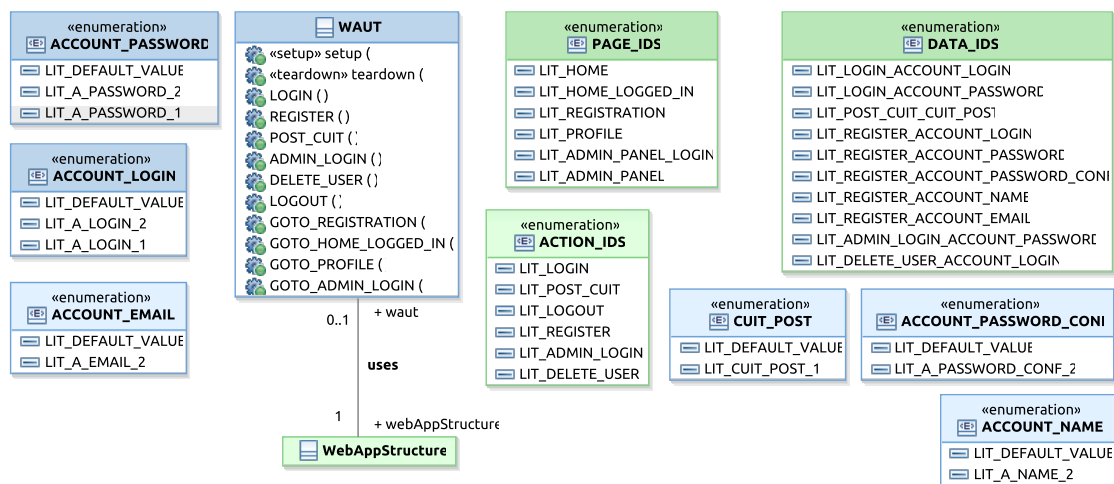


Figure 5.9 – PMVT Class Diagram: Specific Entities of Cuiteur

Test engineers must define the specific entities of the WAUT in a specific class diagram, thus:

- Logical datatypes (e.g., email accounts, genders, etc) along with values.
- Literals for generic enumerations *PAGE_IDS*, *ACTION_IDS* and *DATA_IDS*.
- Logical operations, such as *Login()* or *Register()*.
- Navigational operations, such as *GOTO_PROFILE_PAGE()*.

The specific entities of the Cuiteur Web application are shown in blue in Figure 5.9. As precised in Section 5.2, we only use a subset of *Cuiteur* to illustrate the PMVT notation. As such, we have modeled 6 logical operations: *LOGIN*, *REGISTER*, *POST_CUIT*, *ADMIN_LOGIN*, *DELETE_USER* and *LOGOUT*, as well as 4 navigational operations: *GOTO_REGISTRATION*, *GOTO_HOME_LOGGED_IN*, *GOTO_PROFILE*, and *GOTO_ADMIN_LOGIN*. These 10 operations are sufficient to authenticate to the application, register, post a cuit, access the admin panel, and delete users. Accordingly, we have created identifiers for 6 pages, 6 actions, and 10 user inputs (2 for

login, 5 for registration, 1 to authenticate as an admin, and 1 for user deletion). Finally, we modeled the necessary abstract logical values to interact with the system. Datatypes for the registration form (ACCOUNT_[...]), content for cuits (CUIT_POST), and credentials to authenticate.

5.3.2.3/ OBJECT DIAGRAM: STRUCTURE CONTENT

Very much like with CertifyIt, object diagrams in PMVT represent the initial state of the WAUT. It is an instantiation of both class diagrams. Test engineers must instantiate several entities from the class diagrams, which can be separated into 3 categories:

- 1 - Web application Backbone:** The first category of entities is what makes the WAUT a system as a whole. This consists of instantiating the system classes: *waut*, *WebAppStructure*, *Threat*.
- 2 - Web application Structure:** The second category concerns the content of the WAUT. Test engineers must define each page (by instantiating the *Page* class), user (by instantiating the *User* class), user action (by instantiating the *Action* class), and user input (by instantiating the *Data* class) of the WAUT.
- 3 - Relations between entities:** The last category is about the relationships between the instances. Test engineers must instantiate associations to link the various instances, in order to: define the initial page of the application, link user inputs to their corresponding user action, link user actions to their corresponding page, and link user inputs to their output page (i.e. the pages that output them).

PMVT OBJECT DIAGRAM OF CUITEUR

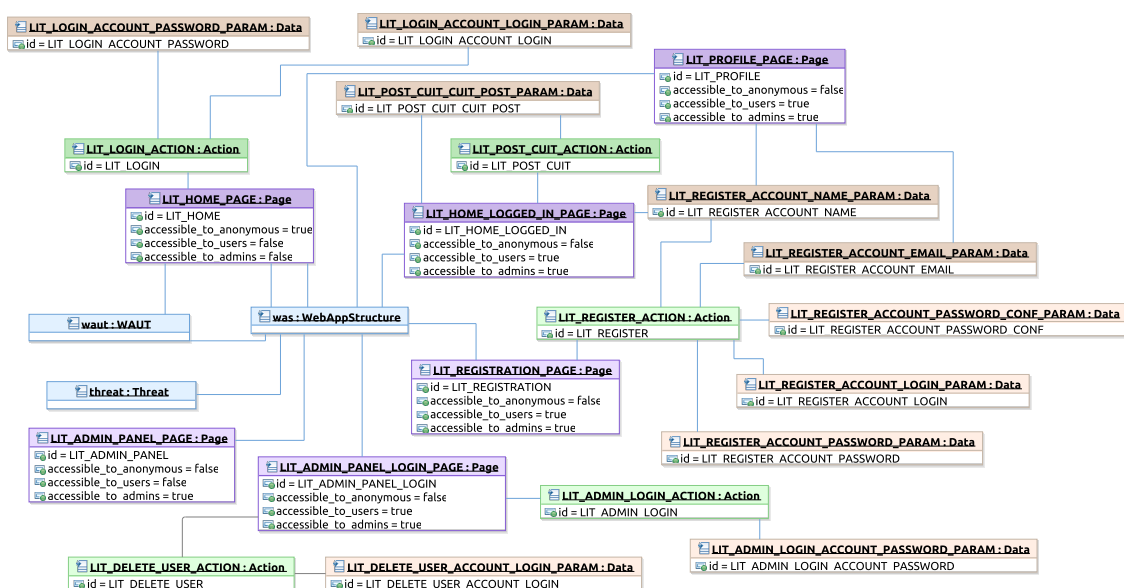


Figure 5.10 – PMVT Object Diagram of Cuiteur

Figure 5.10 shows the PMVT object diagram for Cuiteur. Elements in light blue are instances of the classes that represents the backbone of the application: *WAUT*, *WebAppStructure*, and *Threat*. The *session_type* slot of the *waut* instance has been set to *anonymous*, since the initial state represents the system before any interaction has occurred. Elements in purple are Page instances. Six pages are modeled: the *Login* page, the *Home* page, the *Registration* page, the *Profile* page, the *Admin Login* page, and the *Admin Panel* page. Each instance has a identifier, and its restriction slots are set. For instance, the *Profile* page has an id, *LIT_PROFILE*, and is accessible by users and admins, but not by non-authenticated users. Moreover, the login page is connected to the *waut* instance, which means it is the initial page. As well, this same page is connected to the *WebAppStructure* instance, which means it is the current page.

Elements in green are Action instances. The object diagram for Cuiteur contains 6 actions: *LOGIN*, *REGISTER*, *POST_CUIT*, *ADMIN_LOGIN*, *DELETE_USER*, and *LOGOUT*. Each action has an identifier and is linked to a *Page* instance. For example, the *POST_CUIT* action is owned by the *HOME_LOGGED_IN* instance, which means that on the real system users can post cuit from the home page, once they are logged in.

Elements in orange are Data instances. Each instance has an identifier and is owned by an *Action* instance. For example, the *LOGIN_ACCOUNT_PASSWORD* data is owned by the *LOGIN* action. Moreover, some *Data* instances are linked to output pages such as the *REGISTER_ACCOUNT_EMAIL* param, which is linked to the *Profile* page, meaning that on the real system this page outputs the value of the user input.

This object diagram defines the initial state of Cuiteur as well as its static structure. The definition of its dynamic structure is the object of the next section.

5.3.2.4/ STATE-MACHINE: DYNAMIC STRUCTURE

State-machines in PMVT represent the dynamic of the WAUT, that is to say its map. It contains the various workflows a user can conduct on the application. States of the state-machine represent pages, and transitions between states represent user behavioral and navigational interactions with the WAUT.

Transitions Transitions are user actions that lead to a change of page. Each transition is linked to an operation from the WAUT, which when called by the test generation engine, triggers the firing of its associated transition automatically. In addition, each state has a transition linked to the *reset* operation that leads to the initial page. Consequently, it is possible to reset the model to its initial state, not matter what its current state is.

State States represent unique pages of the WAUT. States are also responsible for the change of current page: when a transition fires, its target state executes the content of its *onEntry* action. More information about page handling is provided in Section 5.3.2.5.

PMVT STATE-MACHINE OF CUITEUR

The PMVT state-machine of Cuiteur is shown in Figure 5.11. It is composed of 6 states, each one representing a unique page. The initial start point of the state-machine leads to the *HOME* state, matching what has been designed in the object diagram. Then, each

- **PAGE_IDS literal:** It is the identifier of the page, contained in an attribute of the *Page* class, named *id*.
- **State-machine state:** Each state of the state-machine represents a page as a dynamic element. Pages are connected to each other with transitions, which are associated to behavioral and navigational operations from the model. This information is used to define the server response (target page) that results from the completion of a user interaction.

When an operation from the model is called, its associated transitions may lead to a new state (i.e. another page). Therefore, each state of the diagram contains an OCL expression in its *onEntry* action which purpose is to replace the current *browses* link between the *WebAppStructure* instance and a given *Page* instance, with a new link involving the page represented by the state:

```
ENTRY: self.webAppStructure.setCurrentPage(PAGE_IDS::LIT_SOME_PAGE)
```

In this expression, the current page displayed to the user is now the *Page* instance which identifier is *LIT_SOME_PAGE*.

Action Handling and Attack Process In PMVT, Actions represent user interactions with the WAUT that can carry data (e.g., Web form, anchors with parameters) and are involved in an attack process. An example is the action of login to the Web application, which consists of a form, usually composed of 2 fields (login and password) and a submit button. A user action is represented by 4 elements: an *Action* instance, an identifier from the *ACTION_IDS* enumeration, an operation owned by the *WAUT* class, and a state-machine transition.

- **Action instance.** It is connected to a *Page* instance and may own one or several *Data* instances. This information is used to know which action is ongoing.
- **ACTION_IDS literal.** This is an enumeration literal that identifies an action. The *Action* class has a attribute called *id*, to which an *ACTION_IDS* literal is affected.
- **WAUT operation.** A WAUT operation is linked to each *Action* instance, and a call to this operation results in the creation of a link (from the *is_ongoing* Association) between the *waut* instance and the corresponding *Action* instance.
- **State-machine transition.** The call to an action operation triggers the firing of a state-machine transition, which may lead to a state change, and therefore a change of current page.

The completion of an action in a nominal way (e.g., filling the 2 fields and clicking on the submit button in the case of the login form) is a process divided in 2 steps:

- 1 - **Nominal data filling:** The first step is about providing values to all the user inputs involved in the completion of the action, if any.
- 2 - **Data submission:** The second step consists of submitting the data (e.g., clicking on the submit button in the case of the login form). Data submission is represented by a single operation in the model, called *finalizeAction()*, in the context of the *WebAppStructure* class. This “finalization” is action-dependent and its is handled during test concretization.

The objective of separating data from data submission is to allow attacks that concern user inputs, such as Injections and Cross-Site Scripting. Indeed, the result of an attack can be clouded by data validation mechanisms from other inputs, especially in the case of Web forms.

Consider again the login form. If, when trying to test the login field for SQL Injections, the password is left empty, the submission may fail because there is a validation mechanism that checks (client-side or server-side) if all fields have been filled. Moreover, it is usually not enough to provide random data as validation mechanism may force a specific data format. As another example, consider a registration form that asks users to provide personal information such as their firstname, lastname and date of birth in a specific format (e.g., MMDDYYYY). An attack towards the firstname field may fail because the random data inserted in the birthdate field is malformed: a protection mechanism may be in place to verify that the data entered matches a valid date of birth. By splitting the Action process, we ensure that all user inputs have been assigned with valid data.

This 2-steps action processing is made possible thanks to OCL operation preconditions and postconditions. In order to force the test generation engine to call the `finalizeAction()` when an action is ongoing, all operations from the model related to the completion of an action are constrained as follows:

```
PRE: not (self.webAppStructure.hasOngoingAction())
POST: self.webAppStructure.ongoingAction = ACTION_IDS::SOME_ACTION
```

The OCL precondition means that for the operation to be callable, there must be no ongoing action. The OCL postcondition, which must be processed as an action language (see Section 4.1 for more information on the double interpretation of the OCL language), defines a new action as ongoing depending of the operation nature (`SOME_ACTION` must be replaced by the action identifier linked to the operation). Contrariwise, the OCL precondition of the `finalizeAction()` is the negation of the OCL precondition above:

```
PRE: self.webAppStructure.hasOngoingAction()
POST: self.webAppStructure.ongoingAction.OclIsUndefined()
```

As well, the postcondition of the `finalizeAction()` removes the existing link between the *WebAppStructure* instance and the action instance that was ongoing.

Thereby, when an operation that represents an action is called, no other operation from the model is callable until the `finalizeAction()` has been called, except for attack operations. This specific representation enables the automated computation of attack traces and ensure precision of the test verdict.

- | | |
|--|---|
| <ol style="list-style-type: none"> 1. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1) 2. was.finalizeAction() 3. waut.someOperation() 4. ... | <ol style="list-style-type: none"> 1. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1) 2. threat.injectSQLi(LIT_A_LOGIN) 3. was.finalizeAction() 4. threat.checkErrorBasedSQLi() |
| (a) Nominal trace | (b) Attack Trace |

Figure 5.12 – Action Handling: Nominal and Attack Traces

Consider the login example we introduced above. Figure 5.12 shows 2 traces involving the *LOGIN* action, one nominal and one aiming to attack the login field with SQL Injec-

tions.

In the nominal trace (Figure 5.12a), we first call the *LOGIN()* operation. Then, on line 2, the call to *finalizeAction()* is the only possibility the test generation engine has in order to go further on the model. Indeed, the effect of the *LOGIN()* operation is to create an *is_ongoing* link between the *WebAppStructure* instance and the *LOGIN* Action instance. At this point, no other operation from the model can be called because of their precondition, which requires that no action must be ongoing. Finally, on line 3, the *LOGIN* action has been finalized, which means the *is_ongoing* link has been deleted. Other operations from the model are now callable, and the exploration of the model can be resumed.

The attack trace (Figure 5.12b) is computed in a similar fashion. We first call the *LOGIN()* operation, but before finalizing the action, we make a call to the malicious operation *injectSQLi(...)* that represents the injection, and consists of replacing the nominal value from the login field with an SQL attack vector. Attack operations have no incidence on the model and are not constrained, which means they can be called regardless of the model state but the test generation engine does not automatically solicit them since they would only make traces bigger (the test generation engine is configured to produce the smallest trace possible). Once the malicious operation has been called, the *LOGIN* action can be finalized and the exploration of the model can be resumed.

Session Handling The PMVT modeling notation enables to represent user sessions, along with page access restrictions depending on the session type and the page accessibility (see Section 5.3.2.1 for more information on how page restriction are defined). Access restriction check is performed in transition guard. Indeed, each transition is guarded with the following OCL expression:

```
GUARD: self.webAppStructure.isAccessible(self.webAppStructure.session_type, PAGE_IDS::LIT_PROFILE)
```

In this example, the called operation implies a change of a page, leading to the *LIT_PROFILE* page. Its associated transition is guarded accordingly: it uses the private operation *is_accessible* to check whether the page is accessible considering the current session type. If the guard is evaluated to **false**, then the transition cannot be fired, and its association cannot be called. This is how access restriction is enforced in PMVT.

The authentication process is represented as an action, and as such it is composed of the same entities: an Action instance, a *WAUT* operation, an id from the *ACTION_IDS* enumeration, and a state-machine transition triggered an *waut* operation call. All these entities are called "LOGIN".

Similar to restriction checks, session type modification is also performed in transitions. Each operation that leads to a change of a session type (either upgrade or downgrade) has one or several transitions, which *effect* is defined as follows:

```
EFFECT: self.webAppStructure.session_type = SESSION_TYPES::LIT_USERS
```

This OCL expression changes the session type to *LIT_USERS*, which symbolizes either a non-authenticated user that login to the application, or an admin that exits the admin panel.

5.3.3/ FROM DASTML TO UML4MBT

The objective when designing a DASTML model is to generate vulnerability test cases. Therefore, this model has to be translated in a UML4MBT instance, compliant with its PMVT subset, in order to be processed by the test generation engine. This is done by a dedicated algorithm that parses the DASTML model using ANTLR² and creates the corresponding UML4MBT entities.

In the next paragraphs, we explain how each entity is transformed into one or several UML4MBT entities. Note that several of these entities are automatically created at the start of a transformation, namely the creation of the generic PMVT class diagram presented in Section 4.1.1.1, and the instantiation of the classes corresponding to the backbone structure of the WAUT (the WAUT class, the WebAppStructure class, and the Threat class).

“PAGE” {...} The declaration of a DASTML page leads to the creation of the following UML4MBT PMVT entities:

- An instance of the *Page* class,
- A state in the state-machine,
- An Enumeration Literal called “PAGE” in the *PAGE_IDS* enumeration,
- A slot (instance of the attribute *id* of the *Page* class) owned by the *Page* instance and valued with the enumeration literal “PAGE”,
- A link (from the *navigable_through* association) between the *Page* instance and the *WebAppStructure* instance.

“PAGE”:INIT Appended to a “PAGE”, the *:INIT* suffix means this page is the initial page. As a consequence, a link from the association *starts* is created between this *Page* instance and the *waut* instance.

RESTRICTED_TO This section is owned by a page and defines its access restrictions, which consists of assigning values to the page accessibility slots, depending on the content of the *RESTRICTED_TO* entity. For instance, if a page has the following restrictions:

RESTRICTED_TO{USERS, ADMINS}

Then the *Page* instance will be set as follows:

accessible_to_anonymous = FALSE

accessible_to_users = TRUE

accessible_to_admins = TRUE

Note that if the *RESTRICTED_TO* keyword is not specified, then all the accessibility slots will be set to *TRUE*.

²<http://www.antlr.org/> [Last visited: August 2015]

“ACTION” (...) → “TARGET_PAGE” The declaration of a DASTML action leads to the creation of the following UML4MBT PMVT entities:

- An instance of the *Action* class,
- An Enumeration literal called “ACTION” in the *ACTION_IDS* enumeration,
- A slot (instance of the attribute *id* of the *Action* class) owned by the *Action* instance and valued with the enumeration literal “ACTION”,
- An operation owned by the *WAUT* class along with its behavior written in OCL (as explained in Section 5.3.2.5).

Moreover, if the completion of the action involves a change of a page, as declared with a right directed arrow, then a state-machine transition is created between the state associated with the *Page* that owns the action and the target state associated to “TARGET_PAGE”. This transition is guarded according to the target page access restrictions.

“ACTION”:(ANONYMOUS | USERS | ADMINS) As explained in Section 5.3.2.5, operations have the ability to change the session type (such as login or register operation). With DASTML, this is simply done with the use of a suffix. As a consequence, the effect of the transition associated with the given operation is defined to change the session type accordingly.

“NAVIGATION” → “TARGET_PAGE” The declaration of a DASTML action leads to the creation of the following UML4MBT PMVT entities:

- An operation owned by the *WAUT* class along with its behavior written in OCL (as explained in Section 5.3.2.5),
- A transition between the state associated with the page that owns the navigation and the target state associated to “TARGET_PAGE”, guarded according to the access restrictions of the target page.

“USER_INPUT” = “VALUE” ⇒ {“PAGE1”, “PAGE2”} User inputs are owned by an action. A user input is associated with a value, and with possible output pages. The declaration of a DAST user input leads to the creation of the following UML4MBT PMVT entities:

- An instance of the *Data* class,
- An enumeration literal called “USER_INPUT” contained in the *DATA_IDS* enumeration,
- A slot (instance of the attribute *id* of the *Data* class, owned by the *Data* instance and valued with the enumeration literal “USER_INPUT”,
- A operation parameter in the *WAUT* operation associated to the *Action* instance that owns the user input.

If the user input has output pages (i.e. the pages that use the value of the input in their output), as declared with a right directed double arrow, then for each specified page a *renders* link is created between the *Data* instance and the corresponding *Page* instance.

5.4/ SYNTHESIS

PMVT is a Model-Based Vulnerability Testing approach that composes formalized generic test patterns with a model of the Web application under test, to automatically generate test cases.

We have designed a dedicated notation to represent Web applications. It is a domain-specific modeling language called DASTML. It allows the modeling of the structure and behaviors of Web applications: the various user roles, 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 the necessary information to generate vulnerability test cases. In order to use the CertifyIt technology to automatically generate test cases, DASTML models are automatically translated in UML4MBT as 4 diagrams, thanks to a set of transformation rules.

In the next chapter, we first present the additions we made to the test purpose language in order to generify vulnerability test purposes w.r.t. PMVT models. Then, we describe and illustrate each of the 7 test purposes we designed to address the 4 vulnerability types detailed in Chapter 2.

PMVT APPROACH: FORMALIZATION OF VULNERABILITY TEST PATTERNS

As the need for security has never been stronger, research in this domain has increased significantly to address the issue. Consequently, a large amount of knowledge has been collected, and various techniques have been designed to test for security breaches and vulnerabilities in systems. However, all this knowledge is scattered and takes multiple forms. The fact that no common standard has been imposed by the academia and research industry constitutes an obstacle to the dissemination of security testing best practices. Recently, there has been an impulse on the design of test patterns and their adaptation to security purposes [70, 62].

Vulnerability test patterns (vTPs) are a form of design pattern, which is a notion coming from building construction [63]. A design pattern is a textual document that describes a recurrent problem and provide a reusable and optimized solution “that you can use a million times over, without ever doing it the same way twice” [1]. First mention of design patterns in software development can be traced back to the Gang-of-4 [33] and their work on showing that relying on patterns for the design of object-oriented software reduces the complexity of software products, by making their architecture and source code easier to understand, while improving their quality and decreasing the cost of development.

VTPs are the initial artifacts of the PMVT approach. A vTP expresses the testing needs and procedure allowing the detection of a particular flaw in Web applications, using informal textual descriptions. There are as many vTP as there are types of application-level flaws. The vTPs that have been designed as part of this thesis are an aggregation of several sources, such as security consortia (CAPEC, OWASP) and research projects. For instance, the ITEA2 DIAMONDS¹ research project has already studied vTPs, and provide a first definition as well as a first listing of vTPs [70]. The characteristics of a vTP are introduced in Figure 6.1.

This chapter explains how PMVT relies on a formalization of vTPs to automate the generation of vulnerability test cases. First, we detail the additions we made to the test purpose language to generify test purposes and enable the formalization of complex and sophisticated attacks. Then, we present the generic test purposes we designed to address the 4 vulnerability types, Cross-Site Scripting, SQL Injections, Cross-Site Request Forgery, and Privilege Escalation.

¹<http://www.itea2-diamonds.org> [Last visited: August 2015]

Name	identifies the pattern
Description	specifies its usage contexts
Objective(s)	specifies the addressed testing objectives
Prerequisites	specifies the conditions and knowledge required for a right execution
Procedure	specifies the <i>modus operandi</i> to put the pattern in practice
Observations and oracle	specifies which information has to be monitored in order to identify the presence of an application-level vulnerability
Variants	specifies some alternatives regarding the means in use, or the attack vector, or what is observed
Known Issue(s)	specifies any limitation or problem (e.g., technical) limiting its usage
Affiliated vTP	lists its related vTPs
References	relates to public resources dealing with application-level vulnerability issues, such as CVE, CWE, OWASP, etc.

Figure 6.1 – Generic Vulnerability Test Pattern

6.1/ AUGMENTED TEST PURPOSE LANGUAGE FOR PMVT

A vTP is the expression of the essence of a well-understood solution to a recurring software vulnerability testing problem. It can be represented as a table containing informal information about the problem. However, the semi-formal way does not allow automated test case generation and execution. With PMVT, we formalize the pattern part that concerns the test procedure with one or several operational *test purposes*, in order to automatically produce the corresponding test cases with model based testing.

A vulnerability test objective aims to be generic in order to be applied on several models to generate test sequences. However, current test purposes contain information coming directly from the current model, which makes them reliant on it. To avoid any dependency, several additions were made to the language to allow the translation of vTPs in operational test purposes while improving their genericity. Namely, these contributions to the language are:

- Introduction of Keyword lists that refer to elements from the model, in order to externalize data manipulation;
- Improvement of *for_each* statements to iterate the results of an OCL expression;
- Addition of variable usage for nested iterators on a set of instances, in order to use the instance obtained from the outer iterator as context for the OCL expression of the inner iterator;
- Addition of variable usage in OCL expressions throughout a test purpose;
- Introduction of stage loops for cases where one or several stages must be activated more than once.
- The creation of a test purpose catalog that allows automatic import/export of existing test purposes from on PMVT test project to another.

These additions enable to represent generic test purposes with the sufficient expressiveness to formalize vTPs, in order to tackle the 4 vulnerability types we presented in Chapter 2.

In the next sections, we define each addition that were made to the language with an explanation on how they increase the genericity of the language and/or enable to compute more sophisticated attack traces.

Keyword Lists A keywords mechanism has been introduced, which consists of using specific arguments, called *keywords*, in test purposes to represent generic artifacts of a PMVT model. They can represent behaviors, calls, instances, integers, literals, operations, or a state regarding a specific instance of the model. Test engineers only have to link keywords with the specific elements of the current PMVT model.

Keywords are contained in lists, and a list may only contain keywords that point to elements of the same nature (behaviors, instances, literals, etc.). Keywords lists can be used both in the iteration and stage phases to replace any of this model information preceded by the character “#”.

For instance, consider an Enumeration (e.g., the PMVT ACTION_IDS enumeration). A keyword list enables to only apply test purposes to literal of the enumeration that share the same properties or restrictions (e.g., selecting only keywords that points to user actions worth testing for CSRF, and excluding unnecessary actions that represent for instance search forms).

```
for_each literal $lit from #KEYWORD_LIST
```

In this scenario, the iterator goes through all the keywords from *#KEYWORD_LIST*, each keyword pointing to a certain enumeration literal.

As another example, consider a test purpose stage that requires to restrict the test generation engine to call an operation from a restricted set, or prohibit the call to a given set of operations. This is done as follows:

```
use any_operation #RELEVANT_OPS to_reach OCL_EXPR1 on_instance $inst1
use any_operation_but #UNWANTED_OPS to_reach OCL_EXPR2 on_instance $inst2
```

The first state expresses to only use any operation that have a corresponding keyword in *#RELEVANT_OPS*. Contrariwise, the second stage expresses to use any operation, except the ones that have a corresponding keyword in *#UNWANTED_OPS*. Note that the latter construction has been extensively employed in PMVT test purposes to restrict the generator to solely call navigational and behavioral operations during the computing of test cases. It is described in details in Section 6.2.

Iterating the Result of an OCL Expression Keywords lists provide a first level of genericity to test purposes. The use of such lists is necessary when the objects they contain must be selected manually (e.g., for CSRF testing). However, when the keywords from a list can be deduced based on the information from the model, it is therefore possible to extract their corresponding element automatically. Hence, we augmented the language to make it possible to iterate the results of an OCL expression. It is constructed as follows:

```
for_each instance $inst from “self.all_users->select(u:User|u.bought_items = 2)” on_instance shop
```

First the OCL expression is evaluated, in the context of the *shop* instance. The expression returns all User instances that have bought exactly 2 items. Then, the results are transmitted to the iterator to be used in the stage phase.

This construction preserves the genericity property of test purposes and automates the test data selection meant to be used for test generation.

Variable usage in nested *for_each* loops Certain types of attack require to consider several datatypes as well as the relationships between them. (e.g., testing for multi-step XSS implies, for a given page, to retrieve all the user inputs that are rendered back on this page). We have thus added variable usage between *for_each* loops. In cases where the outer loop iterates instances and the inner loop iterates the results of an OCL expression, it is possible to use the instance from the first loop as the OCL context for the second loop:

```
for_each instance $inst1 from #INST_LIST
for_each instance $inst2 from "self.all_items" on_instance $inst1
```

In this example, the outer *for_each* iterates a list of instance. The inner *for_each* is reliant on the value coming from its parent as it uses it for defining the context of its OCL expression. Thereby, the *self* variable from the OCL expression corresponds to *\$inst1*.

Usage of data-dependent nested loops is for instance necessary to compute abstract test cases for multi-step XSS, as it avoids the production of unreachable targets.

Variable usage in OCL expressions In more sophisticated attacks, data dependency goes beyond their selection and must be carried throughout the test purpose. For instance, Privilege Escalation attacks involve session types, pages, and their relations, in order to test that access control policies are not flawed. In these cases, we need to use the value from the iterator to configure OCL expressions in order to make test purposes more precise and avoid the submission of irrelevant or unreachable test targets to the test generation engine.

Variables can be used in the iteration phase in cases of nested *for_each* statements, thus:

```
for_each literal $lit from #LITERAL_LIST
for_each instance $INST from "self._insts->select(u:User|u.bought_items = 2)" on_instance shop
```

Moreover, variables can be used in OCL expressions from the restriction part of stages:

```
use any_operation to_reach "self.status = STATUS::$LIT" on_instance sut
```

This stage expresses that any operation from the model can be used, the goal is that the *status* attribute from the WAUT is valued with the content of *\$LIT*, which contains an enumeration literal from the enumeration "STATUS".

Stage Loops In some cases, it is necessary to reproduce the exact same set of steps several times, in order to conduct an attack. This is the case notably for *time-based* and *boolean-based* SQL Injections, which require to inject several vectors in the same user input and compare the results.

To make the design of such test purpose simpler while reducing test generation time, we have introduced the notion of stage loops in the test purpose language. Stage loops

are defined using the declaration keyword `repeat`, followed by a integer and the keyword `times`, expressing the number of loop to accomplish:

```
repeat 3 times
  use CONTROL1 RESTRICTION1 TARGET1
  then use CONTROL2 RESTRICTION2 TARGET2
  then ...
end_repeat
```

In this sequence, we declare that the 3 stages enclosed in the loop must be repeated 3 times.

Test Purpose Catalog Test purposes are stored in a Test Purpose catalog (in XML format), with a reference to the pattern they belong to. This allows test engineers to easily select relevant test purposes depending on the test objective, motivated by a test selection criteria. In the RASEN project for instance, test purpose selection is conducted based on a risk assessment of the WAUT. Regarding the information present in the CORAS diagram, the corresponding vTPs are chosen and the corresponding test purposes are selected for test generation.

Next section describes the test purposes we designed during this thesis to tackle the 4 vulnerability types we presented in Chapter 2, namely Cross-Site Scripting, SQL Injections, CSRF, and Privilege Escalation.

6.2/ SMA TEST PURPOSES FOR WEB APPLICATION VULNERABILITY TESTING

We present in this section the generic test purposes we have designed during this thesis to tackle the 4 vulnerabilities we have described in Chapter 2. Some vulnerabilities required the design of several test purposes when the implementation of multiple attack subcategories was necessary for efficient testing (e.g., for SQL Injections and Privilege Escalation).

For each test purpose, we first present the vTP that we used to design it. Then, we describe its workings by going through each of its steps. Finally, we illustrate the application of such test purpose on the PMVT model of Cuiteur (as depicted in Section 5.3.2) by describing one of the attack traces it helped produce.

6.2.1/ CROSS-SITE SCRIPTING

As we mentioned in Section 2.1, XSS vulnerabilities consists of an attacker injecting an hostile browser executable code (i.e. Javascript, VBScript) into Web pages through user inputs, typically Web forms, or through parameters which value can be modified by clients, such as cookie values. This vulnerability type is one of the consequences of the lack of proper user-supplied input data analysis from the WAUT.

Most vulnerability testing techniques tackle XSS as a one shot kind of attack, and usually proceed within 3 steps: (i) locate a user-supplied input, (ii) inject an XSS vector, and (iii) analyze the server response. Typically, efforts are primarily focused on finding new attack

vectors rather than trying to cover all XSS types. As such, there is very little work in the literature on how to address multi-step XSS vulnerabilities [5].

With PMVT it is possible to tackle all XSS types at once, by applying the testing strategy depicted in Figure A.2. The testing technique represented in this vTP is quite similar to other techniques that we referred to in the above paragraph. However, we make use of the information that we captured in the model: links between user-supplied inputs and the pages of the WAUT that use them to compute an output. Thereby, for the testing of a particular user input, the PMVT attack procedure for XSS is as follows:

- i - Locate the user input:** Following proper user interactions, the WAUT is put in a state where the current page is the page where the user input can be provided. It can be a form field, a parameter in the href attribute of an anchor, a cookie value, etc.
- ii - Fill nominal values:** Often, the user input under test is part of a form or URL, which contains multiple parameters. These parameters need to be assigned with relevant values to prevent any data validation functions (e.g., some parameter must not be left empty, or must be only assigned with specific a datatype) to block the submission of the form/request.
- iii - Replace the input with an attack vector:** Here, the initial nominal content of the user input under test is erased, and replaced with an attack vector.
- iv - Submit the crafted request:** Once the attack vector has been inserted, the crafted request is submitted. Depending on the type of user input, it means submitting the form, or clicking on the link.
- v - Locate an output point:** Instead of simply waiting for the next server response, the information contained in the PMVT model enables to determine which pages use the user input under test as part of their output. The WAUT state is changed accordingly in order to display one of these pages.
- vi - Analyze the result:** The content of the page is analyzed to assess whether the attack vector has been inserted verbatim in the page. If it has not undergone any modification, the WAUT is considered vulnerable to XSS, from this particular user input, and on this particular page.

This test procedure has been translated into a test purpose in order to give each instruction to the test generation engine from Smartesting. The test purpose for multi-step XSS is shown in Table 6.1

First Phase The first 3 lines of the test purpose for XSS compose the first phase. Because this is about XSS, the first *for_each* statement selects all the page that are using at least one user input as output. The selection is done using the OCL expression `Pages.allInstances()->select(p:Page|not(p.all_outputs->isEmpty()))` executed from the context of the WAUT instance. This OCL expression can be split as follows: From all the pages (`Pages.allInstances()`), we select all the pages (`->select(p:Page|)`) that are linked to one or more Data instances (`not(p.all_outputs->isEmpty())`). The result of the OCL expression is a set of *Page* instances.

```

1  for_each instance $page from
2  "self.all_pages->select(p:Page|not(p.all_outputs->isEmpty()))" on_instance was,
3  for_each instance $param from "self.all_outputs" on_instance $page,
4  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
5  "WebAppStructure.allInstances()->any(true).
6    ongoingAction.all_inputs->exists(d:Data|d=self))"
7  on_instance $param
8  then use threat.injectXSS($param)
9  then use was.finalizeAction()
10 then use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
11 "self.was_p.current_page = self and
12 self.was_p.ongoingAction.ocllsUndefined()" on_instance $page
13 then use threat.checkXSS()

```

Table 6.1 – Test Purpose for Cross-Site Scripting

The next *for_each* statement selects all the Data instances that are linked to the Page instance contained in \$page, i.e. all the user inputs that \$page uses to compute its output. Here, the selection is done using the OCL expression `self.all_outputs` from the context of \$page. Therefore, the second stage of the test purpose handles 2 elements, a user input and one of the pages that outputs it.

Second Phase The second phase starts on lines 4-5-6 by putting the WAUT in a state where the displayed page to the user is the injection page, and where the action containing \$param is ongoing, meaning all other fields (in the case of a form) or parameters (in the case of a link) have been filled with nominal values, ready to be submitted. In the context of the selected user input (`on_instance $param`), the test purpose tells the test generation engine to satisfy the OCL expression `WAUT.allInstances()->any(true).webAppStructure.ongoingAction.all_inputs->exists(d:data|d=self)`. First, we retrieve the instance of the WAUT (`WAUT.allInstances()->any(true)`). Second, we navigate in the model until we reach the ongoing action (`webAppStructure.ongoingAction`). Third, we check that the user input \$param is contained in the action (`all_inputs->exists(d:data|data = self)`). To satisfy this expression, the test generation engine must animate the model, and for this we provide the instructions `use any_operation_but #UNWANTED_OPS any_number_of_times`, which means that any behavioral or navigational operation from the WAUT can be called, as many times as needed, in order to find the right WAUT state. Indeed, each test purpose we designed during this thesis possess a keyword list, named `#UNWANTED_OPS`, which contains all the operations from the *Threat* and *WebAppStructure* classes. These operations are not meant to be called during the computation of navigational and behavioral steps, therefore we exclude them to When the right state is found, it is then possible to complete XSS injection. The actual injection is performed in line 7 by calling the operation `threat.injectXSS($param)`, which targets the user input \$param.

Next lines concern verdict assignment. The goal is to put the WAUT in a state where the displayed page is the one that outputs the user input (\$page) in order to analyze its content. This is done by satisfying the OCL expression in line 9, defined in the context

of page. The first part of the expression is about verifying that no action is pending, and the second part specifies that the current page is *self*, i.e. *\$page*. Again, the test generator engine may use any behavioral or navigational operation of the model, as much as necessary. The last line of the test purpose is a call to the operation `threat.checkXSS()` which scraps the page content to look for the injected vector.

Next paragraph provides an example on Cuiteur that shows how the test purpose for XSS is unfolded and applied to one particular user input and output page.

APPLYING THE XSS TEST PURPOSE ON CUITEUR

In the running example presented in Section 5.2, we modeled 6 pages and 10 user inputs. Three of these inputs are rendered back on other pages. Accordingly, the test generation engine computes 4 test targets by applying the XSS test purpose on the DASTML model of Cuiteur:

1. CUIT_POST \Rightarrow HOME_LOGGED_IN
2. ACCOUNT_NAME \Rightarrow HOME_LOGGED_IN
3. ACCOUNT_NAME \Rightarrow PROFILE
4. ACCOUNT_EMAIL \Rightarrow PROFILE

Figure 6.2 shows the generated abstract XSS attack trace for the email field on the register page and its resurgence on the profile page. It consists of: ① accessing Cuiteur's home page, ② clicking on the link that points to the registration page, ③ filling the registration form, ④ replacing the nominal value of the email field with an XSS vector, ⑤ submitting the form and being taken to the home logged in page, ⑥ clicking on the link that points to the profile page, and ⑦ analyzing the content of the profile page.

6.2.2/ SQL INJECTIONS

Like XSS, SQL Injections are another consequence of poor input data validation. This class of vulnerability exploits the trust a Web application has in its users by triggering unwanted interactions between the application and its database. This is done by injecting SQL fragments through user inputs, such as form fields or cookie variables, to alter the semantic of hardcoded SQL queries. Of course, SQL Injections are only possible when

1. `waut.setup()`
2. `waut.GOTO_REGISTRATION()`
3. `waut.REGISTER(LIT_A_LOGIN_2, LIT_A_PASSWORD_2, LIT_A_PASSWORD_CONF_2, LIT_A_NAME_2, LIT_A_EMAIL_2)`
4. `threat.injectXSS(LIT_REGISTER_ACCOUNT_EMAIL_PARAM)`
5. `was.finalizeAction()`
6. `waut.goToProfile()`
7. `threat.checkXSS()`
8. `waut.teardown()`

Figure 6.2 – Abstract XSS Attack Trace on Cuiteur


```

1  for_each literal $param from #DATA
2  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
3  "not(self.ongoingAction.ocllsUndefined()) and
4      "self.ongoingAction.all_inputs->exists(d:Data|d.id=DATA_IDS::$param)"
5  on_instance was
6  then use threat.injectSQLi($param)
7  then use was.finalizeAction()
8  then use threat.checkErrorBasedSQLi()

```

Table 6.2 – Test Purpose for Error-based SQL Injections

the value contained in the user input that lacks sanitization is used by the application to configure an SQL query.

However, the discovery of SQL Injections is much more complex than XSS. Indeed, XSS targets Web browsers and therefore happens on the client-side, where it is easily possible to assess the existence of a vulnerability. On the contrary, SQL Injections affect the database of the WAUT, to which users (and test engineers) don't have direct access. Moreover, in many cases the database is installed on another server. For these reasons, probing the database is out of question.

With PMVT we follow the same verdict assignment process as penetration testers, which consists of "taking what the WAUT gives you". The amount of information about the database that is leaked by the WAUT varies a lot. Hence, we cannot tackle SQL Injections with only one test purpose but with a set of 3 test purposes, each one implementing a dedicated injection and observation style.

Error-Based SQL Injections This is the best case scenario for a hacker / test engineer. Error-based SQL Injections means that syntax error messages from the database (e.g., "You have an error in your SQL syntax" for MYSQL) are displayed to end-users. It can be default error messages from the database but also custom ones, designed for development purposes.

Consequently, the main objective of error-based SQL Injections, when limited to vulnerability discovery only (see the OWASP Testing Guide about standard SQL Injections² for more information), is about breaking the syntactic correctness of the initial query to generate an error message. The reception of an error message is a strong indicator of the presence of a vulnerability, because it means we were able to tamper with the query.

Table 6.2 shows the test purpose for Standard SQL Injections. There is only one iterator for the first phase, which receives user input identifiers. Indeed, we want to test every user input regardless of their possible resurgence. The **for_each** iterates a keyword list, called #SQLI_VULN_PARAMETERS. This list is created automatically by the algorithm responsible of the translation of DASTML models into UML4MBT models, it adds by default all the Data identifiers to the list. The objective is to let the test engineer decide whether one or more user inputs are not worse testing.

²https://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OTG-INPVAL-005%29#Standard_SQL_Injection_Testing [Last visited: August 2015]

The second phase starts on lines 2 to 5. The test purpose instructs the test generation engine to satisfy, in the context of the WAUT instance, an OCL expression composed of 2 sub-expressions. The first sub expression imposes that an action is ongoing to ensure that all other fields that are part of the same request have been properly set. The second sub-expression imposes that the ongoing action must involve the Data instance which identifier is contained in \$param. This way, we know that we are on the right state to inject the user input .

The injection is performed on line 6, with the dedicated operation `threat.injectSQLi()`. Then, we submit the data by calling the *finalize* operation on line 7, and result is assigned on line 8 with the operation `threat.checkErrorBasedSQLi()`.

Time Delay SQL Injections When error messages from the database are not passed on to end-users, another solution for the detection of SQL Injection vulnerabilities is to conduct a temporal differential analysis between several injections. This is performed in PMVT with the injection of 2 vectors. The role of the first vector is to disrupt the syntax of the SQL query in order to cause an immediate response from the database, i.e. with little latency, thus:

```
SELECT * FROM products WHERE name LIKE ' ';
```

In this example, we have injected a single quote, which effect was to disrupt the syntactic correctness of the query.

The role of the second vector is to alter the initial query to generate delay while being processed by the database. It can be done by modifying the query to make the database returns as much data as possible, or by injecting built-in methods such as *sleep(10)*, which stalls the database for 10 seconds:

```
SELECT * FROM products WHERE name LIKE '1' or sleep(10)#';
```

The objective is to observe a variation in response time from the WAUT between the 2 injections.

The test purpose for time delay SQL Injections is depicted in Table 6.3. This test purpose has a similar logic as the one for error-based injections. The iterator in the first phase collects all user input identifiers from a keywords list, which contains only the identifiers that are intended to be tested for SQL Injections.

The second phase is composed of a stage sequence meant to be executed 2 times under the same conditions, one execution dedicated to each injection. During this repeated sequence, the test purpose instructs the test generation engine to drive the model in a state where the current page is the page displaying the user input, then a call to `threat.injectSQLi()` perform the attack by replacing the nominal value with an attack vector, to finally submit the data by calling the `was.finalizeAction()`. Note that the sequence starts with a call to `sut.reset()`, which goal is to reset the WAUT in order to perform another injection within the same conditions. Note that the selection of appropriate SQL attack vectors is handled by the test harness, which is automatically generated during test concretisation. Once the sequence as been executed 2 times, we assess the injections results by calling the `threat.checkTBSQLi()`.

Boolean-Based SQL Injections Another technique for Blind SQL Injections is to perform several attacks and conduct a differential analysis between the server responses.

```

1  for_each literal $param from #DATA
2  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
3  "not(self.ongoingAction.ocllsUndefined()) and
4  "self.ongoingAction.all_inputs->exists(d:Data|d.id=DATA_IDS::$param)"
5  on_instance was
6  then use was.finalizeAction()
7  repeat 2 times
8    then use was.reset()
9    use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
10   "not(self.ongoingAction.ocllsUndefined()) and
11   "self.ongoingAction.all_inputs->exists(d:Data|d.id=DATA_IDS::$param)"
12   on_instance was
13   then use threat.injectSQLi($param)
14   then use was.finalizeAction()
15 end_repeat
16 then use threat.checkTimeDelaySQLi()

```

Table 6.3 – Test purpose for Time Delay SQL Injections

In PMVT, the vTP we relied on to create this test purpose has been designed following the testing strategy³ proposed by IBM and implemented in its scanning tool, AppScan. Indeed, by injecting SQL fragments that will cause singular changes to the initial SQL query, the objective is to observe a difference of behavior from the WAUT.

Consider a Web application with a search page containing a text field. The content of this field (*\$inputvalue*) is sent to the database in order to configure the following SQL query:

SELECT * FROM products WHERE name LIKE '\$inputvalue';

The response contains the product entries whose name is close to the content of the search field. The result is sent to the user, in the form of a Web page that lists the content.

A Boolean-Based SQL Injection is composed of 4 injections, as follows:

- 1 - Nominal Injection:** This is the intended interaction with the WAUT. The server response is used as “control group”, its objective is to compare the nominal behavior of the WAUT with its behavior when receiving SQL fragments as input.
- 2 - AND TRUE:** The objective is to inject an SQL fragment that has a positive logic and does not change the overall logic of the query, such as:

SELECT * FROM products WHERE name LIKE 'NOM' AND 1=1;
--

Based on the monotone law of identity for \wedge , since the boolean sub expression $1=1$ is always true and because it is tied to a conjunction, then the result of the expression depends on the other sub-expression *EXPR* of the conjunction:

$$EXPR \wedge 1 = EXPR$$

- 3 - AND FALSE:** The objective is to inject an SQL fragment that has a negative logic and changes the overall logic of the query, such as:

³<http://www-01.ibm.com/support/docview.wss?uid=swg21659226> [Last visited: August 2015]

```

1  for_each_literal $param from #DATA,
2  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
3  "not(self.ongoingAction.ocllsUndefined()) and
4    "self.ongoingAction.all_inputs->exists(d:Data|d.id=DATA_IDS::$param)"
5  on_instance was
6  then use was.finalizeAction()
7  repeat 3 times
8    use was.reset()
9    then use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
10   "not(self.ongoingAction.ocllsUndefined()) and
11     "self.ongoingAction.all_inputs->exists(d:Data|d.id=DATA_IDS::$param)"
12   on_instance was
13   then use threat.injectSQLi($param)
14   then use was.finalizeAction()
15 end_repeat
16 then use threat.checkBooleanBasedSQLi()

```

Table 6.4 – Test purpose for Boolean-based SQL Injections

SELECT * FROM products WHERE name LIKE 'NOM' AND 1=2;
--

Based on the monotone law of identity for \wedge , since the boolean sub-expression $1=2$ is always false and because it is tied to a conjunction, then the result of the expression is always false:

$$EXPR \wedge 0 = 0$$

- 4 - OR FALSE:** This injection is similar to the *AND TRUE* injection, and is mainly use to rule out the possibility of SQL Injections by reinforcing the verdict:

SELECT * FROM products WHERE name LIKE 'NOM' OR 1=2;

Based on the monotone law of identity for \vee , since the boolean sub-expression $1=2$ is always false and because it is tied to a disjunction, then the result of the expression depends on the other sub-expression $EXPR$ of the conjunction:

$$EXPR \vee 0 = EXPR$$

Verdict is assigned by comparing the responses from the server.

If all responses are equivalents, we can conclude that SQL Injections are not possible:

$$NOMINAL = ANDTRUE = ANDFALSE = ORFALSE \Rightarrow \neg(Inj_{input})$$

However, if the results from the nominal and *AND TRUE* injections are equivalents, but there is a difference in the responses between the *AND TRUE* and *AND FALSE* injections, we can conclude that there is a strong possibility that the injected user input is vulnerable to SQL Injections:

$$(NOMINAL = ANDTRUE) \wedge (ANDTRUE \neq ANDFALSE) \Rightarrow Inj_{input}$$

This attack has been translated to a test purpose, as showed in Table 6.4. First phase consists of collecting all the user input identifiers that are intended to be tested for SQL Injections, and assigning them one after another to $\$param$ to compute attack traces.

Second phase is composed of 2 main sequences. The first sequence consists of sending nominal values, and collecting the resulting page. First, the test purpose proposes in

line 2 to use any behavioral or navigational operation, as many times as necessary, to satisfy the OCL expression defined in lines 3-4. This expression requires, on the one hand, that an action must be ongoing, and on the other hand that this action involves the user input whose identifier is \$param. Satisfying this expression will take the hypothetical user to the injection page, with all fields filled (in the case of a Web form). Then in line 6, the test purpose instructs to call the `finalizeAction()` method, in order to submit the form / click on the link.

The second sequence is responsible for the completion of the 3 SQL Injections. Since the protocol is the same for each injection, we use the *repeat* keyword to simplify the test purpose and save test generation time. Note that the selection of appropriate SQL attack vectors is handled by the test harness, which is automatically generated during test concretisation. Therefore, each attack starts by calling the `was.reset()` operation, in order to put the model back to its initial state. Then in lines 9 to 12, and similarly to the nominal sequence, the second step is to put the model in a state where the current ongoing action is this action involving the user input under test (\$param). Then, the injection is performed in line 13, and the newly crafted request is submitted to the server in line 14.

Once the attack sequence has been executed 3 times, the `threat.checkBlindSQLi()` operation is called in line 16 to compare all 4 responses and assign a verdict.

APPLYING THE SQLi TEST PURPOSES ON CUIEUR

In the running example presented in Section 5.2, we modeled 10 user inputs. Accordingly, the test generation engine computes 10 test targets per SQLi test purpose (30 test targets in total) by applying each test purpose on the DASTML model of Cuietur.

	1. <code>waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)</code>	
	2. <code>was.finalizeAction()</code>	
	3. <code>waut.POST_CUIT(LIT_CUIT_POST_1)</code>	
	4. <code>threat.injectSQLi(LIT_POST_CUIT_CUIT_POST)</code>	
	5. <code>was.finalizeAction()</code>	
	6. <code>was.reset()</code>	
1. <code>waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)</code>	7. <code>waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)</code>	
2. <code>was.finalizeAction()</code>	8. <code>was.finalizeAction()</code>	
3. <code>waut.POST_CUIT(LIT_CUIT_POST_1)</code>	9. <code>waut.POST_CUIT(LIT_CUIT_POST_1)</code>	
4. <code>threat.injectSQLi(LIT_POST_CUIT_CUIT_POST)</code>	10. <code>threat.injectSQLi(LIT_POST_CUIT_CUIT_POST)</code>	
5. <code>was.finalizeAction()</code>	11. <code>was.finalizeAction()</code>	
6. <code>threat.checkErrorBasedSQLi()</code>	12. <code>threat.checkTBSQLi()</code>	
(a) Error-Based	(b) Time-Based	

Figure 6.3 – Error-based and Time-Based Abstract SQL Injection Attack Traces on Cuietur

1. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)	12. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)
2. was.finalizeAction()	13. was.finalizeAction()
3. waut.POST_CUIT(LIT_CUIT_POST_1)	14. waut.POST_CUIT(LIT_CUIT_POST_1)
4. was.finalizeAction()	15. threat.injectSQLi(LIT_POST_CUIT_CUIT_POST)
5. was.reset()	16. was.finalizeAction()
6. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)	17. was.reset()
7. was.finalizeAction()	18. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)
8. waut.POST_CUIT(LIT_CUIT_POST_1)	19. was.finalizeAction()
9. threat.injectSQLi(LIT_POST_CUIT_CUIT_POST)	20. waut.POST_CUIT(LIT_CUIT_POST_1)
10. was.finalizeAction()	21. threat.injectSQLi(LIT_POST_CUIT_CUIT_POST)
11. was.reset()	22. was.finalizeAction()
	23. threat.checkBlindSQLi()

(a) Boolean-Based

Figure 6.4 – Abstract SQL Injection Attack Traces on Cuiteur

Figures 6.3 and 6.4 show the generated abstract SQL Injection attack traces for the `post_cuit` field on the home logged in page: The attack trace for the error-based SQL Injection is depicted in Sub-figure 6.3a, the one for time-based injection in Sub-figure 6.3b, and the one for Boolean-based injection in Sub-figure 6.4a.

The error-based SQL Injection attack trace consists of 6 steps: ① filling the login form, ② submitting the login form, ③ filling the `post_cuit` form with a nominal value, ④ replacing the nominal value with an SQLi vector, ⑤ submitting the form and being taken to the home logged in page, and ⑥ analyzing the content of the home logged in page.

Attack traces for time-based and boolean-based SQL Injection consists of respectively 12 and 23 steps. Notice that we do not describe them since they are of a similar logic than the attack trace for error-based SQL Injection.

6.2.3/ CROSS-SITE REQUEST FORGERIES

A CSRF attack consists of tricking a victim into making a specific request through his/her browser, that will ultimately lead to unwanted consequences on a trusted Web application. It is qualified as malicious because it indirectly impersonates a user to perform actions only him/her or a restricted group of users is allowed to do, and without him/her knowing. It is due to the fact that browsers automatically append user credentials (session data) to each request made towards a Web application where a user session has been started. These attacks are made possible when the targeted Web application does not check whether an incoming request is really originating from the user owning the active session.

The proposed vTP consists of conducting an actual CSRF attack by cloning the action being tested on an external server, to assess whether this action can be triggered from outside the application. The logic is similar to BURP's CSRF PoC⁴, and goes as follows:

- 1 - Nominal Action:** The objective is to follow the intended behavior of the application and perform the action from inside, using the GUI.

⁴<https://support.portswigger.net/customer/portal/articles/1965674-using-burp-to-test-for-cross-site-request-forgery-csrf-> [Last visited: August 2015]


```

1  for_each literal $action from #CSRF_(SESSION_TYPE)_ACTIONS
2  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
3  "not(self.ongoingAction.ocllsUndefined()) and self.ongoingAction.id=ACTION_IDS::$action
4  "and self.session_type = (SESSION_TYPE)" on_instance was
5  then use threat.gatherCSRFInfo()
6  then use was.finalizeAction()
7  then use was.reset()
8  then use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
9  "not(self.ongoingAction.ocllsUndefined()) and
10 "and self.session_type = (SESSION_TYPE)" on_instance was
11 then use was.finalizeAction()
12 then use threat.performCSRFAttack()
13 then use threat.checkCSRF()

```

Table 6.5 – Test Purpose for Cross-Site Request Forgeries

- 2 - Information collect:** The link / form being responsible for the triggering of the action is extracted, as well as the output page, for later comparison.
- 3 - Reset:** The application is reinitialized and the current user session is closed.
- 4 - Login:** The user authenticates to the application, to open a new session.
- 5 - External Action:** The action is submitted from an external Web server, using the same browser. This is done using a java program that starts a local Web server, which takes as input the data gathered during information collect. The server recreates the form or link based on the received data, and sends the result to the user, in the form of an interactive Web page.
- 6 - Result Comparison:** The results from the nominal and the external actions are compared back to back. If both results are similar, then the tested action is considered vulnerable to CSRF attacks.

This vTP has been translated into a test purpose depicted in Table 6.5. In the first phase, we collect all the actions that are part of the test objective regarding CSRF. Each action will be affected to the \$action variable to configure the second phase of the test purpose.

The second phase starts by triggering the action as intended by the application. This is performed by satisfying the OCL expression on line 3-5 that requires to put the model in a state where the action under test is ongoing and the user is properly logged in. Then in line 6, the `threat.gatherCSRFInfo()` operation is called to retrieve the Web form or link that is used to submit the action. On line 7, we finalize the action and then reset the application on line 8.

The attack sequence starts in line 9-11 by instructing the test generation engine to satisfy an OCL expression that expresses that a new user session should be started, with the same privileges as during the nominal sequence, and that no action should be ongoing (i.e. the login form has been submitted). This is done with an OCL expression, that can be satisfied using any behavioral or navigation operation from the model, as many times as necessary. Finally, the CSRF Attack is performed in line 13, and the 2 results are compared on line 14, by calling the `threat.checkCSRF()` operation.

APPLYING THE CSRF TEST PURPOSE ON CUITEUR

In the running example presented in Section 5.2, 4 actions were modeled (login, registration, post of cuits, logout). Accordingly, the test generation engine computes 4 test targets, one per action.

```

1. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)
2. was.finalizeAction()
3. waut.POST_CUIT(LIT_CUIT_POST_1)
4. threat.gatherCSRFInfo()
5. was.finalizeAction()
6. was.reset()
7. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)
8. was.finalizeAction()
9. threat.performCSRFAttack()
10. threat.checkCSRF()

```

Figure 6.5 – Abstract CSRF Attack Trace on Cuiteur

Figure 6.5 shows the generated abstract CSRF attack trace for the *post_cuit* field on the *HOME_LOGGED_IN* page. It consists of: ① filling the login form, ② submitting the login form, ③ filling the *post_cuit* form with a nominal value, ④ gathering info, namely the Web form, ⑤ submitting the form and being taken to the home logged in page, ⑥ resetting the application, ⑦ filling the login form again to open a session, ⑧ submitting the login form, ⑨ conducting the attack, and ⑩ comparing the 2 responses for verdict assignment.

6.2.4/ PRIVILEGE ESCALATION

Applications do not always protect application functions properly. As anyone with network access to a Web application can send a request to it, such application should verify action level access rights for all incoming requests. When designing a Web application frontend, developers must build restrictions that define which users can see various links, buttons, forms, and pages. Although developers usually manage to restrict Web interface, they often forget to put access controls in the business logic that actually performs business actions: sensitive actions are hidden but the application fail to enforce sufficient authorization for these actions. If checks are not performed and enforced, malicious users may be able to penetrate critical areas without the proper authorization.

The method we implemented in PMVT to test for Privilege Escalation is called *Forced Browsing*. The objective is to obtain a direct URL to trigger a action or access a page of the Web application that is supposed to be available only to users with sufficient rights. The underlying idea is that developers may have hidden the access to such action or page in the GUI but forgot to enforce the restriction in the action's code. Therefore, the vTP for Privilege Escalation consists of the following steps:

1. Access the page / Trigger the action as intended, from the GUI, with a session that

```

1  for_each literal $session from #SESSION_TYPES,
2  for_each instance $page from
3  "self.all_pages->select(p:Page|not(self.isAccessible(SESSION_TYPES::$role,p.id)))"
4  on_instance was,
5  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
6  "self.was_p.current_page = self and self.was_p.ongoingAction.ocllsUndefined()"
7  on_instance $page
8  then use threat.collectPage()
9  then use was.reset()
10 then use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
11 "self.ongoingAction.ocllsUndefined() and
12 "and self.session_type = SESSION_TYPES::$role"
13 on_instance was
14 then use threat.accessPage()
15 then use threat.checkPrivilegeEscalation()

```

Table 6.6 – Test Purpose for Privilege Escalation of Pages

carries the sufficient rights.

2. Save the direct URL that point to that page / action.
3. Save the output result for later comparison.
4. Logout from the Web application, or change the session state (from admin to regular user, for instance).
5. Access the URL directly, and save the output result.
6. Compare the 2 outputs.

If the output results are equivalent, it constitutes an indicator that we were able to access the restricted page or action.

The vTP described above has been formalized in 2 test purposes: one for pages and one for actions.

Privilege Escalation: Pages The first phase of the test purpose for Privilege Escalation of pages, as shown in Table 6.6, is composed of 2 nested *for_each*. The first iterator retrieves all the possible session types, and the second iterator retrieves all the pages that are not accessible to the currently iterated session type. To do this, we use the private operation *isAccessible* (as described in Section 5.3.2.1) that defines whether a given session type can access to a given page.

In the second phase, the test purpose first instructs the test generation engine to satisfy an OCL expression that requires to put the model in a state where the current page is *\$page*, and no action is ongoing. Then, we collect the relevant information using the *collectPage* operation. The nominal part of the vTP is done at this point, the next step is then to reset the system, and start the attack part. This is done by instructing the test generation engine to satisfy an OCL expression, which is evaluated to true when the

current session type of the WAUT is the session type from the iterator. Once the system is in the right state, the restricted page is visited using the *accessPage* operation. The last step is verdict assignment, thanks to the *checkPrivilegeEscalation* operation.

Privilege Escalation: Actions The test purpose for privilege escalation of restricted actions, depicted in Table 6.7, shares a similar structure with the one for pages.

```

1  for_each literal $session from #SESSION_TYPES,
2  for_each instance $action from
3  "self.all_pages->select(p:Page|not(self.isAccessible(SESSION_TYPES::$role,p.id)))
4  ->collect(p:Page|p.all_actions" on_instance was,
5  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
6  "self.was_ca.ongoingAction = self" on_instance $action
7  then use threat.activateCapture()
8  then use was.finalize()
9  then use threat.collectPage()
10 then use was.reset()
11 then use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
12 "self.ongoingAction.ocllsUndefined() and
13 "and self.session_type = SESSION_TYPES::$role"
14 on_instance was
15 then use threat.triggerAction()
16 then use threat.checkPrivilegeEscalation()

```

Table 6.7 – Test Purpose for Privilege Escalation of Actions

In the first phase, the outer loop retrieves all possible session types and for each session type, the inner loop retrieves all the actions that cannot be triggered by users under this session type.

The second phase starts by requesting to put the model in a state where the iterated action is ongoing, which means the current page is the page owning this action. In line 8, the *activateCapture* is for concretization purposes: it tells the test harness to start capturing the outgoing request made by the test script, in order to collect relevant information (target URL, parameters, etc.). Then, we submit the action, collect the page result, and reset the application in lines 8-10.

The attack sequence first requests to put the model in a state where the current session type of the WAUT corresponds to the one from the iterator, and where no action is ongoing (meaning the authentication credentials has been submitted). In line 14, we try to trigger the action by calling the *triggerAction* operation, using the information collected by the *activateCapture* operation. Finally, we compare the 2 outputs from the server by calling the *checkPrivilegeEscalation* operation.

APPLYING THE TEST PURPOSES ON CUITEUR

There are 6 pages on Cuiteur and 3 different session types. Based on the access restrictions of these 6 pages, the test generation engines computes 5 test targets:

- 1 - **ANONYMOUS:** to access the *HOME_LOGGED_IN* page.

- 2 - ANONYMOUS:** to access the *PROFILE* page.
- 3 - ANONYMOUS:** to access the *ADMIN_LOGIN* page.
- 4 - ANONYMOUS:** to access the *ADMIN_PANEL* page.
- 5 - USERS:** to access the *ADMIN_PANEL* page.

Similarly, there are 5 actions on the PMVT model of Cuiteur, which leads to the computation of 3 test targets:

- 1 - ANONYMOUS:** to trigger the *POST_CUIT* page.
- 2 - ANONYMOUS:** to trigger the *DELETE_USER* page.
- 3 - USERS:** to trigger the *DELETE_USER* page.

For Privilege Escalation of pages, consider the test target $n^{\circ} 5$, which consists of being authenticated as a regular user and directly request the *ADMIN_PANEL* page. Its corresponding attack trace is shown in Figure 6.6a. The first 4 steps are nominal actions: login as user with admin privileges, reach the admin login page, authenticate, and access the admin panel. At this point, we collect information about the page content and its URL. Then, the application is reset. Finally, we authenticate as a regular user, try to access the admin panel directly, and observe the result.

Test target $n^{\circ} 3$ consists of being authenticated as a regular user and directly trigger the *DELETE_USER* action, whose attack trace is displayed in Figure cuitur:priv:trace:action. The first 4 steps are similar to the attack trace for the *ADMIN_PANEL* page. Then, we activate the information capture, trigger the action and collect the output result. The application is then reset/ Finally, we authenticate as a regular user, try to directly trigger the action, and observe the results.

1. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)	1. waut.LOGIN(LIT_A_LOGIN_1, LIT_A_PASSWORD_1)
2. was.finalizeAction()	2. was.finalizeAction()
3. waut.GOTO_ADMIN_LOGIN()	3. waut.GOTO_ADMIN_LOGIN()
4. waut.ADMIN_LOGIN(LIT_A_PASSWORD_1)	4. waut.ADMIN_LOGIN(LIT_A_PASSWORD_1)
5. was.finalizeAction()	5. was.finalizeAction()
6. threat.collectPage()	6. threat.activateCapture()
7. was.reset()	7. waut.DELETE_USER(LIT_A_USER_1)
8. waut.LOGIN(LIT_A_LOGIN_2, LIT_A_PASSWORD_2)	8. threat.collectPage()
9. was.finalizeAction()	9. was.reset()
10. threat.accessPage()	10. waut.LOGIN(LIT_A_LOGIN_2, LIT_A_PASSWORD_2)
11. threat.checkPrivilegeEscalation()	11. was.finalizeAction()
	12. threat.triggerAction()
	13. threat.checkPrivilegeEscalation()

(a) Page-Based

(b) Action-Based

Figure 6.6 – Abstract Privilege Escalation Attack Traces on Cuiteur

6.3/ SYNTHESIS

In PMVT, precise and accurate test generation is made possible thanks to test purposes. A test purpose is a high-level textual expression that formalize test scenarios, based on regular expressions, to drive the test generation process through the model in order to derive test targets and generate test cases.

However, test purposes in their original form are too reliant on the model on which they are applied. Therefore, we have augmented the test purpose language with 6 additions to improve the genericity of test purposes, and make them reliant on the UML4MBT generic class diagram. Since all PMVT test projects share this generic class diagram, test purposes are therefore usable as is for any Web application. Using the augmented test purpose language, 7 test purposes were to address the 4 vulnerability types described in Chapter 2.

In the next chapter, we present the toolchain that was developed to support the PMVT process along with a description of all the tools it relies on, and all the mechanisms that were designed to automate the test execution.

Contents

7.1 Implementation of the PMVT Approach	101
7.1.1 Modeling Activity	102
7.1.2 Test Purpose Activity	104
7.1.3 Test Generation Activity	105
7.1.4 Test Concretization and Execution Activities	106
7.2 PMVT tools for Concrete Vulnerability Testing	107
7.2.1 PMVT Test Publisher	108
7.2.2 Executable Test Scripts	109
7.2.3 CSRF Web Server	109
7.2.4 Web Page Comparator	110
7.3 RASEN: Test Selection from Risk Assessment	111

We present in this chapter the technical details of the PMVT approach. First, we illustrate the implementation of the PMVT process using a dedicated toolchain. We describe each activity along with the tools involved. Then, we expose the various mechanisms we designed to automate the test execution, and especially the conducting of attacks regarding the four vulnerability types presented in Chapter 2.

7.1/ IMPLEMENTATION OF THE PMVT APPROACH

A toolchain has been created for the deployment of the PMVT approach to conduct experiments, in order to assess its accuracy and precision (see RO1) as well as its usability, scalability and capacity of automation (see RO2). This toolchain is composed of a set of tools, each one having a role in one or several activities of the approach:

- **IBM Rational Software Architect.** An eclipse-based graphical modeling tool created by IBM. The CertifyIt MBT methodology relies on this tool with two eclipse plugins to enable the creation UML4MBT models, their animation, and their exportation for test generation. In a PMVT deployment, this tool is used during the modeling and test selection activities. First, it is used for the translation of DASTML models into UML4MBT models, thanks to a dedicated plugin. Second, it allows advanced users to make adjustments to the translated UML4MBT models. Third, it enables user to import, create, and select test purposes based on their test objectives.

- **Certifylt Test Generation Engine.** This tool takes as input a UML4MBT model, derives test targets; computes test cases from these test targets, and exports the results. It is used in the PMVT process during the test generation activity to unfold the test purposes, compute the corresponding attack traces by relying on the model, and translate the attack traces in JUnit4 Test scripts.
- **Integrated Development Environment (optional).** For a better management of the generated test scripts, we relied on an Integrated Development Environment (IDE) for the test concretization, execution, and verdict assignment activities. During this thesis, we have exclusively worked with IntelliJ Idea. However, it is possible to complete these activities with another IDE such as Eclipse or Netbeans, or without an IDE.

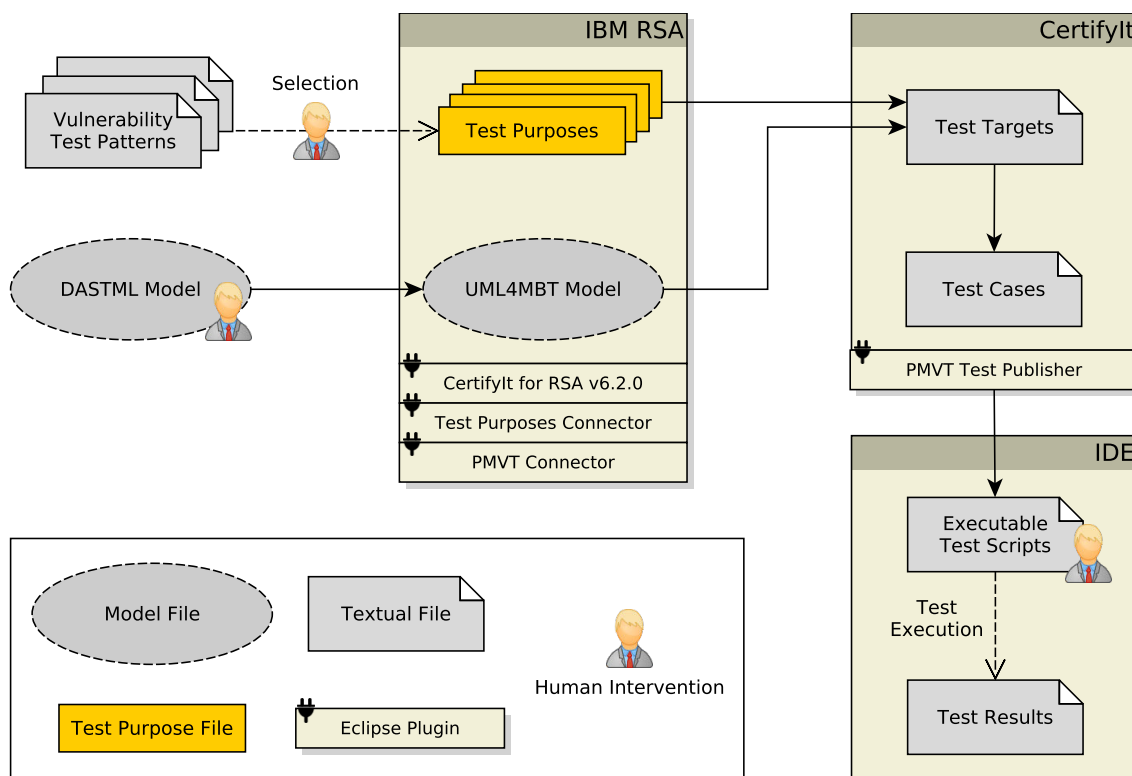


Figure 7.1 – Overview of the PMVT toolchain

We demonstrate the implementation of each PMVT activity and the use of the associated tools in the following subsections.

7.1.1/ MODELING ACTIVITY

The first action in a PMVT process is Web application modeling. We have designed a domain-specific modeling language called DASTML (see Chapter 5 for more information about the language), which enables to create textual PMVT models. Therefore, any text editor can be used for model design.

Designing a PMVT DASTML model does not require any testing skills, and consists of doing a manual crawling of the Web application under test to gather relevant information

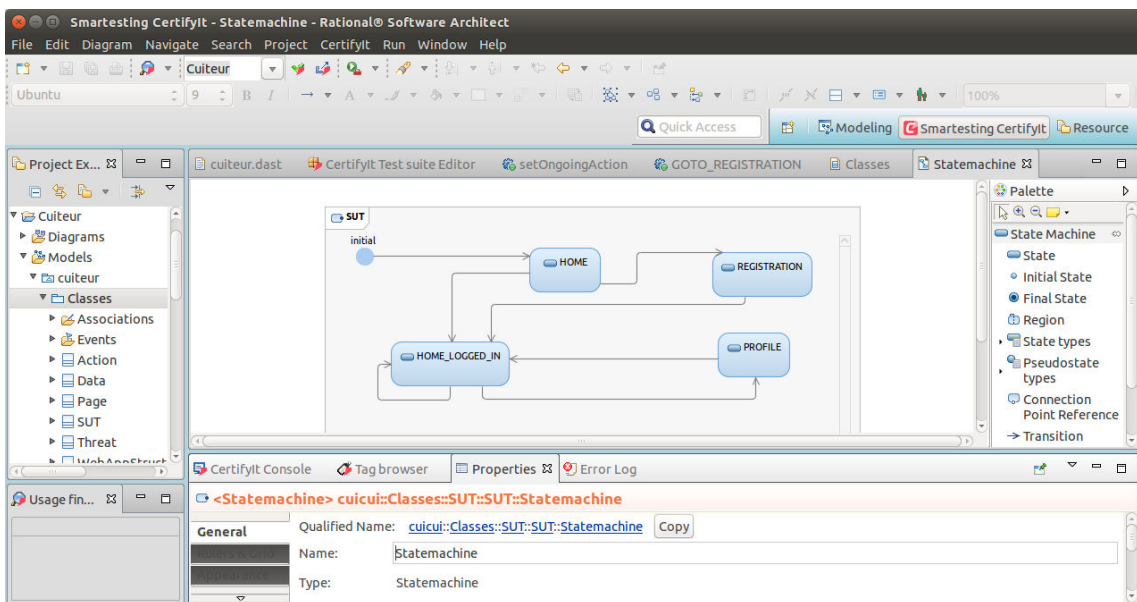


Figure 7.2 – PMVT: Modeling Environment

that will be used for vulnerability detection purposes.

The first step of this manual process is about creating a “map” of the Web application, that is to say the *page flow*. This can be done by clicking on links and filling Web forms, the goal is to identify each unique page that requires testing (e.g., because it provides user interactions that are not purely navigational) or would be part of an attack process (e.g., a result page).

Then, the second step is to do reconnaissance on each page that has been modeled to gather information in order to design a DASTML Model, in a text file. Once the model is complete, or a first subset of the Web application has been modeled, the file can be processed by RSA to translate its content in an UML4MBT model.

Figure 7.2 shows the RSA interface, extended with several plugins.

The first plugin, *Smartesting Certifylt for RSA v5.2.0*, enables to create UML4MBT models, animate them, check their compliancy with the UML4MBT metamodel, and export them in a standardized XML file. It is used as is in PMVT to manage generated UML4MBT entities and potentially make adjustments in special cases where experimented test engineers need to go outside the PMVT realm, for instance to create new test purposes. It is also possible to animate PMVT models to check whether they are accurate regarding the Web application under test.

The second plugin, *Test Purposes connector for RSA v1.3*, provides a graphical interface to manage test purposes: creation, import / export, validity check. It is more discussed in the next section.

The third plugin, *PMVT Connector for RSA v1.2*, is responsible for the translation of DASTML models in UML4MBT models. DASTML models are parsed using an ANTLR3 grammar, which creates an Abstract Syntax Tree (AST) out of it. Then, a dedicated algorithm visits the AST to create the corresponding UML4MBT entities on the fly. This is delivered to users of the solution in two ways.

First, test engineers can choose to create a new modeling project. An eclipse wizard has been created for PMVT purposes and provides two inputs: a DASTML model file, and

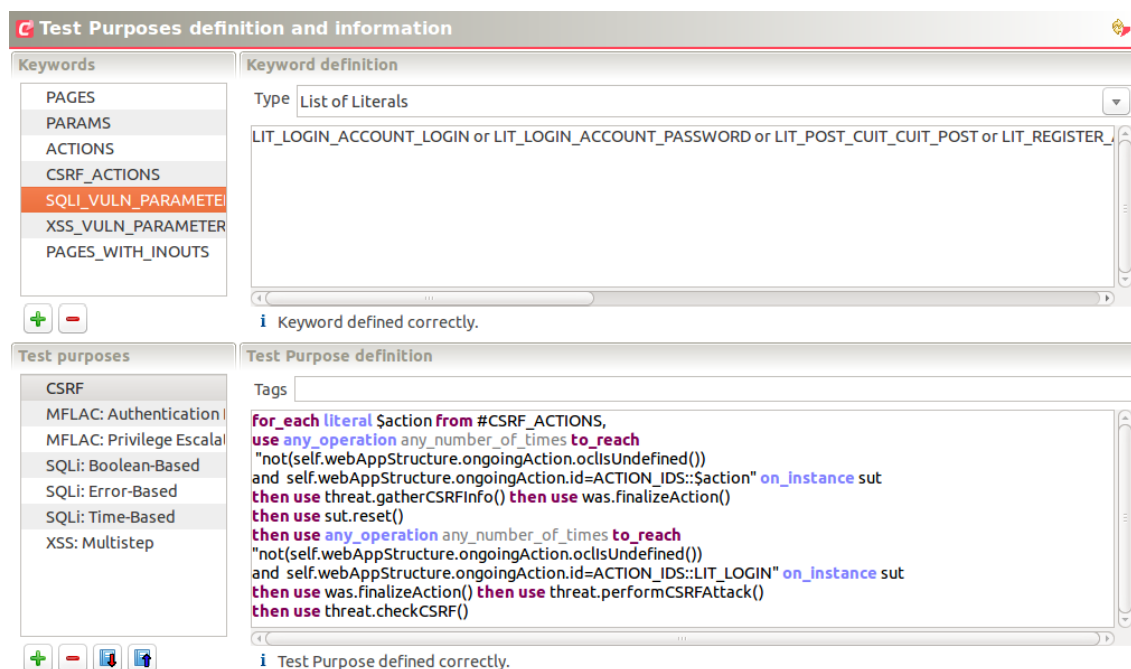


Figure 7.3 – PMVT: Test Purposes Editor

a test purpose catalog. From these inputs, the wizards automatically creates the corresponding UML4MBT models, and integrates the test purposes from the catalog. Second, it is possible to import a DASTML model in an existing project. However, model evolution is not yet supported at this point and test engineers must remove any previously generated entity before importing a new DASTML model.

Next step of the PMVT process is the selection and / or creation of test purposes, as described in the next section.

7.1.2/ TEST PURPOSE ACTIVITY

Once the model has been designed and translated in an UML4MBT instance, test engineers have to proceed to test selection. This is done in PMVT using test purposes (described in Section 6.1). The design and selection of test purposes is also done through RSA. A graphical interface has been created, which is depicted on Figure 7.3.

The upper part of the interface concerns keywords lists. Such list can contain a set of Enumeration literals, instances, states, behaviors, etc. Lists content is then used by test purposes iterators to compute test cases based on a refined set of model elements, therefore allowing test engineers to generate precise test suites that only address the selected elements. Note that all the visible keywords lists visible on the figure have been generated automatically.

The lower part of the interface is the test purpose editor. The left side lists all the created or imported test purposes, and on the right side is the actual editor. It is shipped with test purposes and OCL expressions on-the-fly syntax check.

Users can add and remove test purposes, and export their test purposes as a catalog in an XML file. This way, they can import them back into another project.

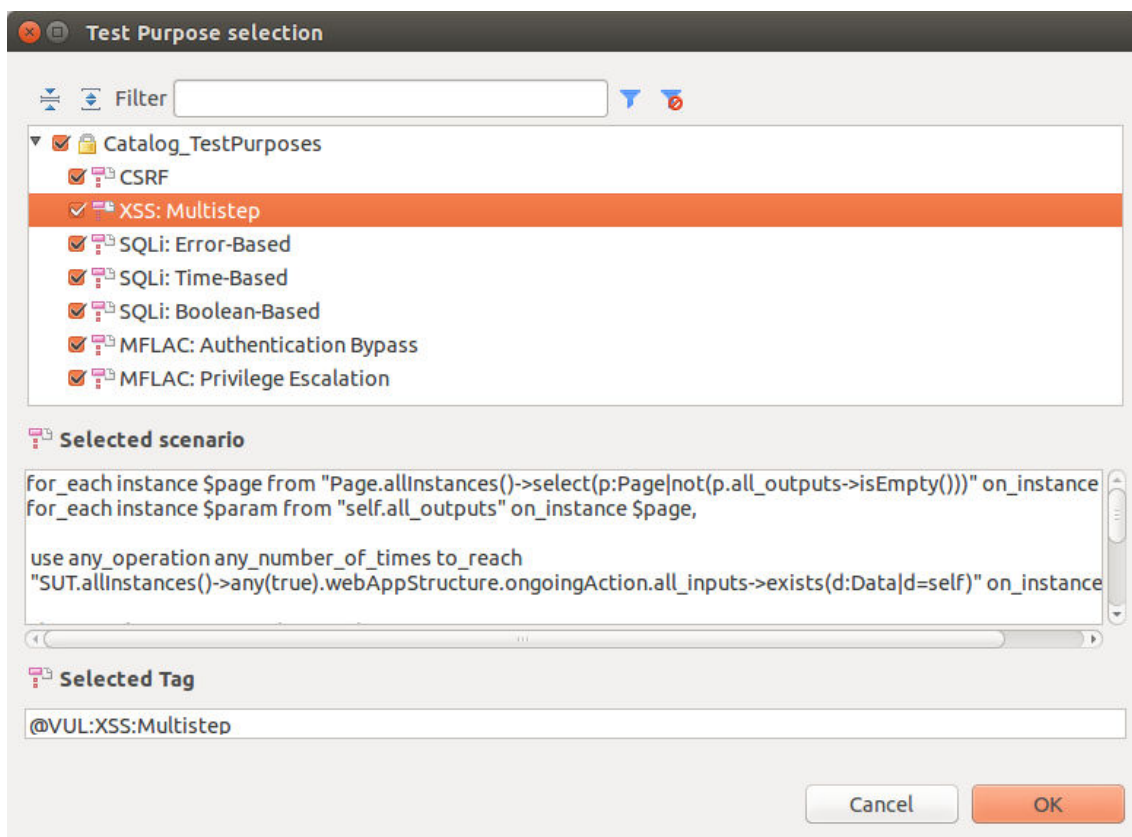


Figure 7.4 – PMVT: Test Purposes Catalog

As part of PMVT, we provide to users of the approach a predefined catalog containing the existing test purposes we presented in Section 6.2. Users can import all or part of this catalog. The test purposes stored this catalog are usable as is to test any Web application, without the need for adaptation. Figure 7.4 shows the interface for importing test purposes from the catalog in a PMVT test project. Notice that for the existing test purposes to be usable, the associated model must be compliant with the PMVT modeling notation.

When the model and the test purposes have been defined, test engineers can check the validity of their PMVT project. If it has been successfully checked, the next step is to export the PMVT project in a standardized XML file and pass it on to the test generation engine.

7.1.3/ TEST GENERATION ACTIVITY

The test generation engine that is used as part of the PMVT process is *CertifyIt 5.2.0*. This tool generates abstract test cases by finding paths in the PMVT model to reach the test targets that have been derived from the test purposes. Figure 7.5 shows its interface. The left part lists all the test targets that the tool extracted from the model. If a test target is successfully reached, a test case is produced and all its steps are displayed on the right part of the tool.

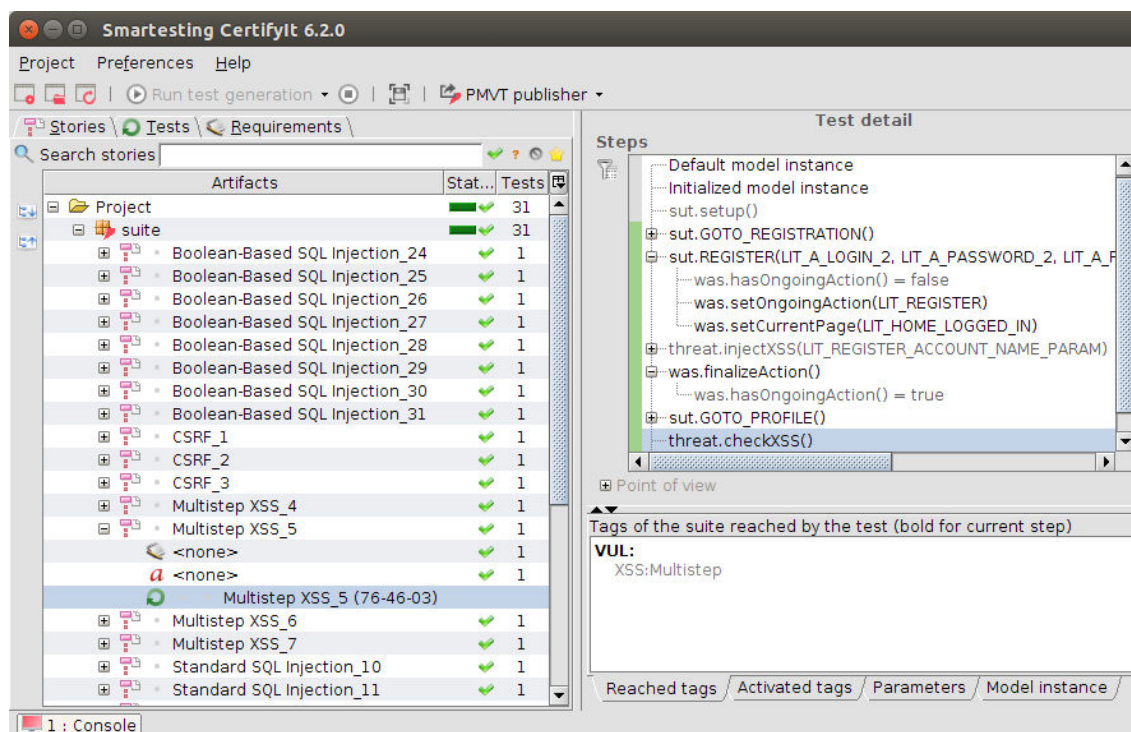


Figure 7.5 – PMVT: Test Generation Environment

CertifyIt interprets test purposes using a two-step process.

First, during the importation of a PMVT project, the test generation **unfolds** the test purposes: if a test purpose contains iterators, then the test generation engine creates a test target for each element of the list linked to the iterator.

Second, the test generation engine tries to reach each test target by animating the model.

When test generation is complete, test engineers in general export the generated abstract test cases in executable test scripts, which we detail in the next section.

7.1.4/ TEST CONCRETIZATION AND EXECUTION ACTIVITIES

Abstract test cases are not executable as is on the real system, therefore they must be concretized (adapted) and exported as test scripts. Test engineers must implement each operation from the model, provide concrete data to match each abstract data, and may need to develop a test harness in order to simplify test execution.

In the context of PMVT, generated abstract test cases are exported as JUnit test scripts. These scripts allow to automatically interact with the Web application through its GUI, as a normal user would do. To accomplish this programmatically, PMVT test scripts use libraries such as Selenium¹ or HTMLUnit², which enable to operate a headless browser using primitives (i.e. methods such as filling form fields and clicking buttons / anchors).

Test concretization activity represents a considerable amount of work to obtain executable test scripts. A PMVT test publisher has therefore been designed in order to

¹<http://www.seleniumhq.org/> [Last visited: August 2015]

²<http://htmlunit.sourceforge.net/> [Last visited: August 2015]

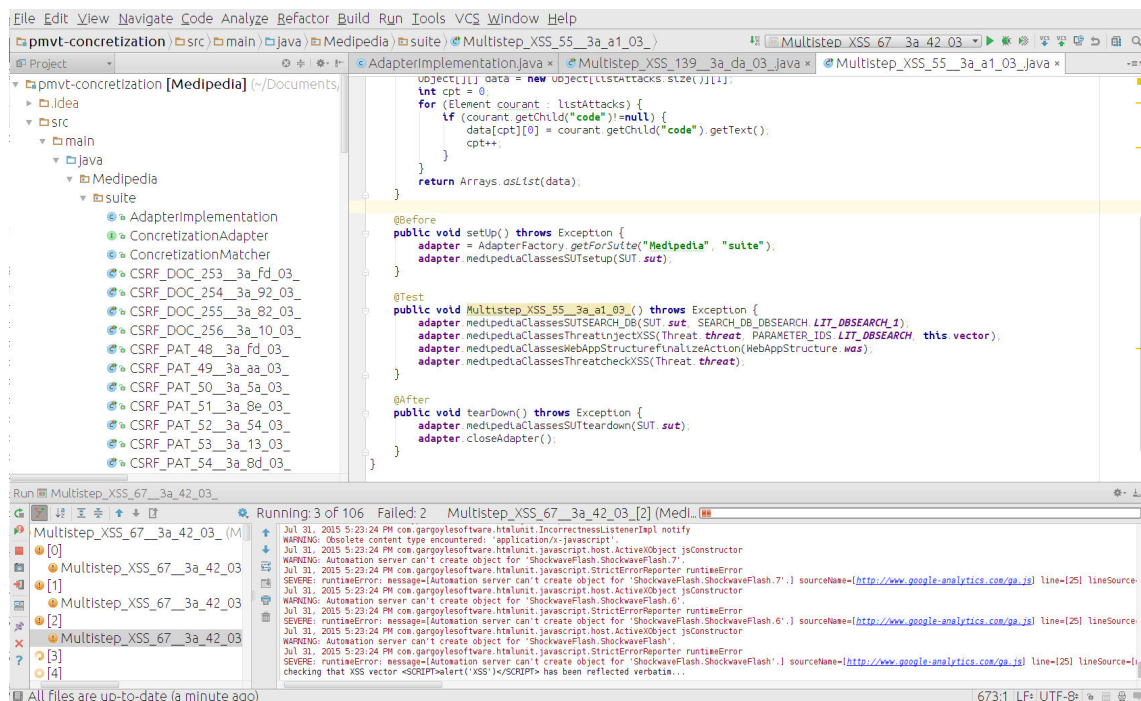


Figure 7.6 – PMVT: Test Concretization and Execution Environment

automate most of the process.

When all abstract data have been matched with concrete data and all methods have been implemented, the obtained test scripts can be executed on the real Web application. Test execution and verdict assignment are fully automated, thanks to the test harness generated by the PMVT publisher. Test engineers execute the test suite, and collect the results.

Both the test concretization and execution activities are performed preferably using an IDE (in our case, IntelliJ Idea), as depicted in figure 7.6. Such tool provides a graphical view of the test results, and an “export” function to store the results, for instance in an XML file. This file can be used to generate a textual report, although it is out of the scope of this thesis.

In the next section, we present the various tools and mechanisms we relied on to concretize and execute the attack traces generated with the PMVT approach.

7.2/ PMVT TOOLS FOR CONCRETE VULNERABILITY TESTING

Several tools and mechanisms were designed and/or utilized during this thesis to simplify the concretization of abstract test cases and automate the execution of test scripts. We describe these tools in the next sections. We start with the PMVT Test Publisher, which enabled to save considerable amounts of time by inferring most of the test concretization process. We continue with the additions we brought to existing java libraries that enable to browse Web applications programmatically, in order to conduct sophisticated attacks.

Then, we describe the Web server that allowed PMVT to simulate real-life CSRF attacks for efficient CSRF testing. Lastly, we present the technique we implemented for smart Web page comparison, which is extensively used to establish verdict assignments for blind SQL Injections, CSRF, and Privilege Escalations.

7.2.1/ PMVT TEST PUBLISHER

The concretization activity in most MBT techniques is considered as a tedious and time-consuming task. In this section, we discuss the PMVT test publisher that was designed to automate most of the concretization activity. It parses the XML file containing all the generated abstract test cases and computes JUnit test scripts accordingly. Such test scripts are then integrated in a test harness that centralizes data and automates their execution.

Test engineers must provide concrete values to each abstract data from the model, and implement each operation from the WAUT:

- Navigational and logical operations, using a library providing a programmatically-controlled browser, such as Selenium.
- Attack and observation operations, using predefined attack scripts or from scratch.

To ease this consequent activity, a PMVT test publisher has been created. It exports all generated test cases as JUnit test scripts and generates a complete test harness that automates most of the concretization activity, as follows:

- Implementation of operations from the model is centralized. All operations from the model are exported as Java methods in a separate class file, and JUnit test scripts simply call these methods.
- All attack and observation methods are implemented.
- All navigational and logical methods are implemented with generic enhanced Selenium primitives (See next section for more information about how Selenium is used in PMVT).
- Abstract data matching with concrete values is centralized. Manipulated data are exported as static String variables and defined in a separated class file. Methods simply call these variable to obtain the concrete value.
- All tests scripts and tools are structure in a Mavenized Java project, which automates test execution.

Thereby, test engineers can focus on matching abstract data with concrete values and possibly edit operations that have specific behavior. The PMVT Publisher has shown great results and considerably reduced concretization design time by more than 60% compared to the built-in JUnit publisher in CertifyIt.

7.2.2/ EXECUTABLE TEST SCRIPTS

Vulnerability test cases that have been generated as part of a PMVT process are translated in JUnit test scripts. Test scripts must trigger the real Web application the same way a standard user does, by interacting with the GUI. It means clicking on links and buttons, filling form fields, etc. In addition, test scripts must have access to the emitted HTTP requests as well as the HTTP responses from the server, in order to conduct attacks. For instance, Cross-Site Scripting and SQL Injections attacks are performed by intercepting requests containing nominal values, and replacing the initial value of the parameter under test with an attack vector (see Section 5.12a for more details about user actions and attacks handling).

This is done programmatically in PMVT using a combination of Java libraries:

- **Web Browser Driver:** In order to stay close to the GUI, access any Web application the same way users do, while having the ability to conduct attacks, we used Selenium. This Java library allows to control a Web browser programmatically through a set of commands, such as *click()*, *sendKeys()*, *get(URL)*, and so on. Selenium can control several browsers, such as Firefox, Chrome, and Internet Explorer.
- **Headless Browser:** Selenium provides program control over several browsers. However, instantiating a full graphical browser for vulnerability testing purposes is too time and memory consuming. To ensure fast execution of the test scripts as well as browser-like support of Web applications, we made selenium controls PhantomJS³, a headless javascript Web browser based on Web kit⁴ (Webkit is the layout engine that powers Safari and Chrome).
- **HTTP interceptor:** Requests and Responses capture is not implemented in Selenium. Therefore, we rely on Browsermob-proxy⁵. This java library enables to create a proxy programmatically, and provide methods to intercept requests / responses for observation, modification, or rejecting.

In order to efficiently combine these libraries, we extended the Selenium main class and integrated PhantomJS and Browsermob-proxy to it. This Web driver, called PMVT Web driver, enables to write simple and powerful test scripts that can browse any Web applications accessible using the regular Safari. Moreover, we have implemented necessary features for attack purposes, such as a page load timer (which is not implemented in Selenium) that was used for verdict assignment of Time Delay SQL Injections. The source code of this Web driver is available on Github⁶.

7.2.3/ CSRF WEB SERVER

The testing procedure for CSRF detection (see Section 6.2.3) involves the use of a Web Server, which acts as an outside Web application from which a hacker would trick its victims. The objective is to reproduce actions that can only be performed from within the WAUT User Interface, such as button / anchor clicks and form submissions. If the

³<http://phantomjs.org/> [Last visited: August 2015]

⁴<http://www.webkit.org/> [Last visited: August 2015]

⁵<http://bmp.lightbody.net/> [Last visited: August 2015]

⁶<https://github.com/Alex-Vernotte/PMVT> [Last visited: August 2015]

response from the WAUT is identical to its typical response when the action is performed following a nominal workflow, we can suspect the presence of a CSRF vulnerability.

To perform this task, we created a Java program that starts a local Web server along with a HTTP request handler. Its purpose is to reproduce a complete Web form based on the data it receives. Two parameters are mandatory: *form_method*, which specify the type of request to send (GET or POST), and *form_action*, which specify the target URL to which send the data. All other parameters are considered as parameters from the original form, and are included in the crafted form.

For instance, consider the URL below that points to the CSRF Server with several input parameters:

```
http://csrf_server:8045/?form_method=GET
&form_action=http://www.waut.dev/index.php&login=toto&password=superpw
```

This URL leads to the creation of following Web form:

```
2 <form method='GET' action='http://www.waut.dev/index.php'>
  <input type='text' name='login' value='toto' />
  <input type='text' name='password' value='superpw' />
4  <input type='submit' name='submit' />
  </form>
```

This technique makes it possible to simulate a CSRF attack by relying on an external server to submit the form. Note that this technique can be used to test Web forms, but also parameterized anchors and buttons.

7.2.4/ WEB PAGE COMPARATOR

Test verdict assignment is the act of defining whether a test has succeeded or failed upon execution. In order to assign a verdict, a test case contains assertions to compare test execution data and expected data, with the latter obtained using an *oracle*. When testing functional or security functionalities, the oracle is usually deduced from the specification of the system. On the contrary, for vulnerability testing the oracle is inherent to the vulnerability type under test. In Web application vulnerability testing, verdict assignment is performed through analysis and/or comparison between outputs (i.e. Web pages).

However, as we observed during early experiments, when comparing two Web pages one cannot simply confront the two raw outputs: there is too much transient noise in a Web page. Indeed, a lot of content in an output from the server is generated randomly, and / or include random data. For instance, a Web applications often have a dedicated frame for advertising to help owner(s) earn money based on traffic. These ads are usually loaded randomly and are therefore different from one client request to another.

For effective Web page comparison, it is mandatory to strip Web pages of random content or better yet, extract static content only.

During this thesis, we designed a tool capable of comparing two Web pages, depending of four comparison criteria:

- **Page content.** Full content comparison between two pages;

- **Full URL.** Only the URL of the two pages are compared;
- **Abstracted URL.** Because URLs can also include random parameter values, this technique removes these values to only keep the base URL and the parameters name.
- **Control points.** This technique consists of extracting HTML elements that allow user interactions, such as anchors, web forms, and buttons.

In practice, we almost entirely relied on the Control point comparison criterion during experiments. We detail this technique in the next section.

CONTROL POINTS COMPARISON

This Web page comparison technique is based on the state-aware vulnerability scanner in [21]. It consists of only taking into account elements from the page that, if triggered, tell the browser how to create a subsequent request. The idea is to reduce Web pages to a set of possible users interactions. Following this logic, two pages that contain the same set of user interactions are considered equal. Contrariwise, two pages that look similar may in fact considered different because one of the page has an additional anchor toward the administration panel, for instance. This comparison technique is loose enough to strip off pages from random content, such as advertising, but strong enough to detect the one user interaction that cannot be found in both pages.

Therefore, when comparing two Web pages, we only take into account a very restricted set of HTML elements:

- Anchors (`< a > .. < /a >` in HTML)
- Buttons (`< button../ >` in HTML)
- Web forms and all their possible inputs (inputs of any kind, select, textarea)
- Javascript Events (e.g., `onClick`, `onFocus`, `onSubmit`, and so on)

We have designed an algorithm, in Java, which takes as input two HTML documents and extract all control points from both pages, in order to compare back to back the nature of the control points. The source code has been put on Github⁷. This algorithm allowed us to test for CSRF, Time-based SQL Injections, and Privilege Escalation vulnerabilities.

7.3/ RASEN: TEST SELECTION FROM RISK ASSESSMENT

The RASEN approach proposes to extend the PMVT technique by relying on risk assessment results to guide the selection of test cases. VTPs are affiliated to one type of risk, and depending on the identified risks and their severity and likelihood, corresponding test patterns are selected and prioritized to generate relevant test cases. The adaptation of such technique for risk-based vulnerability testing defines novel features and perspectives

⁷<https://github.com/Alex-Vernotte/PMVT> [Last visited: August 2015]

in this research domain. Risk and requirements-driven testing was originally introduced by James Bach in [6], where he underlined the creative aspects of software testing to manage stated and unstated requirements depending on risks associated with the SUT. This means going beyond having just one test for each stated requirement and implies abilities from the Quality Assurance team to recognize potential risks for the SUT. Risk may be defined as a Unwanted incident, which may occur at a given likelihood and impact an asset with a given consequence. Risk-driven test process management focuses on risk assessment and test prioritization based on requirements.

This approach influences the entire MBT process by driving the development of test generation artifacts: (i) MBT models have to precisely capture risk aspects besides functional features, and (ii) the test selection strategies applied on MBT models have to be specified to cover risk and its related priorities.

The Figure 7.7 introduces the PMVT process, which implements these features to enable risk-driven MBT approach.

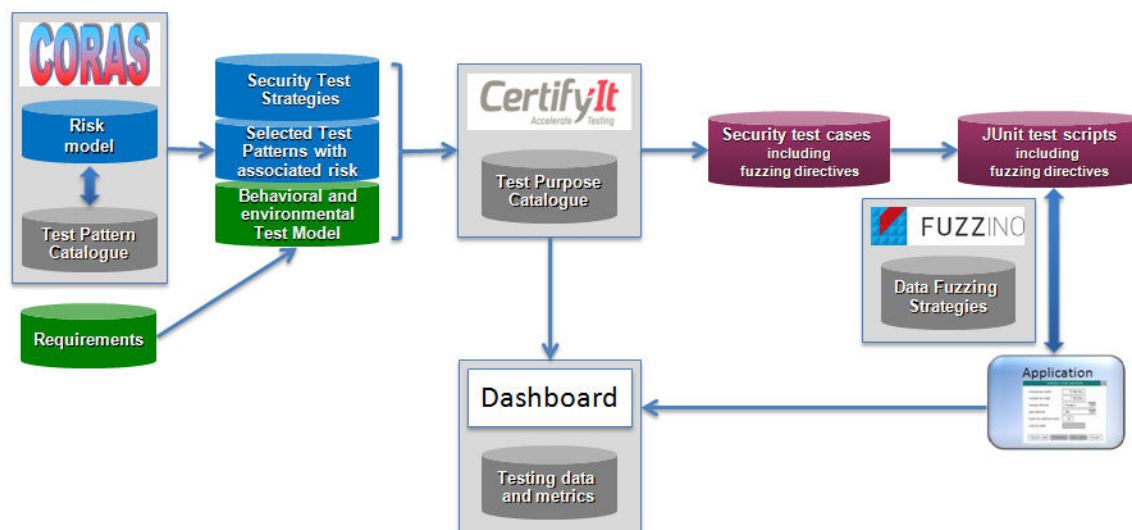


Figure 7.7 – RASEN: PMVT guided by Risk Assessment

The process starts with a risk model as a result from risk assessment. Risk may be defined as a unwanted incident which may occur at a given likelihood and impact an asset with a given consequence. Especially for complex systems, there are not sufficient resources to test all vulnerabilities and threat scenarios identified during risk analysis. Each threat scenario of the CORAS risk model is linked to a security test pattern defining the testing procedure to detect the threat in the application under test.

Hence, a CORAS risk model (in relation with associated generic security test pattern and vulnerability catalogues) enables to select security test purposes and to prioritize them with respect to risk estimation. CORAS is a model-driven method for risk analysis featuring a tool-supported modeling language specially designed to document risks and their causes. Risk model elements such as vulnerabilities are linked to dedicated security test patterns.

The PMVT process ensures the traceability between the test case, the verdict of the execution and the targeted vulnerabilities identified during risk assessment. To manage

traceability between risk assessment and the generated test cases, the test model also embeds all the related information about prioritization and test procedure, which are inherited from the CORAS model and the test pattern. The results of the security testing metric functions help to characterize the security risks of the system under test. Hence, it is possible to improve a risk assessment based upon the results since these functions can be used, for instance, to calculate and update the likelihood values regarding the exploitability of some threat scenarios.

Contents

8.1 Case Studies	115
8.1.1 WackoPicko	116
8.1.2 Bookshop	117
8.1.3 Stud-E	118
8.1.4 Medipedia Web Portal	119
8.2 Web application Vulnerability Scanners	120
8.2.1 IBM AppScan	121
8.2.2 IronWasp	122
8.3 Experiment Results	122
8.3.1 Experimental Setup	122
8.3.2 Dummy Web applications	123
8.3.3 Medipedia	126
8.3.4 Stud-E	127
8.4 Threats to Validity	128
8.5 Discussion	129

In this chapter, we present the experimentation results obtained by applying the PMVT toolchain on 5 case studies and comparing its results with those obtained using 2 vulnerability scanners. First, we describe the five case studies, along with their eventual implementation specificities that can harden vulnerability detection. Second, we provide information on the 2 scanners that were integrated in the experimentation. Third, we present the obtained results on each use case, and for each tool. Finally, we analyze and discuss these results and provide answers to the Research Questions introduced in Section 1.3.

8.1/ CASE STUDIES

We detail in this section the case studies we relied on to conduct experiments with PMVT and security scanners.

8.1.1/ WACKOPICKO

Wackopicko is a realistic Web application that has been made vulnerable deliberately. It has been created by [22] for experimentation purposes, motivated by the fact that existing vulnerable applications were either educational tools and therefore not realistic enough for thorough experiments, or known well enough so existing techniques may be optimized to perform well on them. It is a PHP / MySQL web site, run on an Apache Server.

The application is a photo sharing and photo-purchasing site. Users can upload photos, browse other user's photos, comment on photos, and purchase the rights to a high-quality version of a photo. It has several features :

- **Authentication.** WackoPicko has a user registration system. Users have to register and then log in to access WackoPicko's restricted features.
- **Pictures.** Users can upload pictures, and other users can comment on them as well as purchase the right to a high-quality version. It follows a classic e-commerce process, with a shopping cart filled with items, possible discount coupons, and a shopping summary page. Once purchased, a link to a high-quality version is sent to the user.
- **Search.** A search toolbar lies at the top of every page, to search for particular pictures.
- **Guestbook.** Visitors can provide feedback by submitting a form containing a *name* and a *comment* field.
- **Admin Area.** Administrators have access to a special area, with a dedicated authentication mechanism. Once logged in, they have access to sensible actions such as creating or deleting users.

The web site contains 16 vulnerabilities that are representative of the current landscape, according to well-known organizations such as OWASP[76]. For the sake of this experiment, we only present those that are tackled by the PMVT approach:

- **WP-RXSS-01: Reflected XSS.** There is an XSS vulnerability on the search page which is accessible without having to be authenticated.
- **WP-SXSS-02: Stored XSS.** The *comment* field in the guestbook page is not properly escaped.
- **WP-RSqli-03: Reflected SQL Injection.** An SQL Injection is possible through the *username* field of the authentication form.
- **WP-JSXSS-04: Reflected XSS behind JavaScript.** There is a form on WackoPicko's home page that checks if a file is in the proper format for upload. It has a *name* parameter that is echoed back unsanitized upon a successful check.
- **WP-SSqli-05: Stored SQL Injection.** On the registration page, the supplied value for the *firstname* field is used unsanitized on an SQL request in another page.
- **WP-MSXSS-06: Multi-Step Stored XSS.** On the picture page, the *comment* field is vulnerable to XSS. It is used unsanitized when displayed along the picture. It

is qualified as multi-step because the comment has to be previewed and validated upon submission: only the preview is sanitized and a typical vulnerability test case has to go all the way to the validation.

- **WP-FLRXSS-07: Reflected XSS Behind Flash.** There is a Flash form on the home page, asking users for their favorite color. The *value* parameter is echoed back without sanitization.

8.1.2/ BOOKSHOP

Similar to Cuiteur, bookshop has been initially created for educational purposes: students were asked to develop a similar Web application based on the course they were given. It is a PHP / MySQL web site, run on an Apache Server. Notice that one particularity of this application lies in its design: it forces users to rely on the GUI by generating a token on each page, and every user action must carry this token to the next page in order to access it. If the token received on the target page matches the token generated on the source page, access is granted and a new token is generated. If the tokens don't match, the user is automatically logged out. This mechanism is implemented in sensitive Web applications, such as Banking portals. The goal is to evaluate whether scanners and the PMVT approach are able to test such application.

The web site mimics an e-commerce platform that sells books. Once registered, a user can search for books, put books into his/her cart, complete the ordering process, create a wish-list, and display other users' wish-list.

- **Authentication and registration system.** To be able to shop on Bookshop users should be logged in to the system, and this requires a registration. To do so, users should provide a valid email address, a valid password, plus their name and birthday. A valid new registration implies that the user is automatically logged in.
- **Account info.** This page allows users to see and modify their account and shipping information. It is not accessible unless the user is logged in. The first form on the page allows to modify connection information: (i) firstname and lastname, (ii) birthday, (iii) email address, (iv) password and confirmation. The second form on the page allows to modify shipping information: (i) address, (ii) zipcode, (iii) city, (iv) country. Users can't make an order if any of these data is missing.
- **Book search.** The search page displays a text input, allowing users to search for a particular kind of book. Search can be performed either by providing an author's name or by providing a book title fragment. Each result entry can then be added to the user's cart. For authenticated users, it is also possible to add these items to a wish-list.
- **Friends' wish-list search.** On this page users can search for other users' wish-list. When a valid email address is provided, the wish-list of the corresponding user is displayed, listing each book that has not been offered yet. The action here is to add one or more books to the cart.
- **Cart management.** On the cart page, users can modify the quantity of each book, remove them from the cart, and validate the cart. The validation process takes

users to a Paypal sandbox¹, where they can pay-pretend for their order.

- **Administrator panel.** Administrators have access to a special area, with a dedicated authentication mechanism. The administrator panel provides several actions, such as the creation and deletion of users, and the creation and deletion of administrators.

Bookshop has been made vulnerable to various vulnerability kinds on purpose. We detail each vulnerability below:

- **BS-RXSS-01: Reflected XSS.** On the search book page, the search field is vulnerable to reflected XSS. There is no sanitization and malicious scripts can be injected and will be executed in the output page.
- **BS-SXSS-02: Stored XSS.** On the account page, the *name* field is vulnerable to stored XSS. There is no sanitization and malicious scripts can be injected and will be executed every time a user perform a request to the infected profile view page.
- **BS-SXSS-03: Stored XSS.** On the account page, the *city* field is vulnerable to stored XSS. There is no sanitization and malicious scripts can be injected and will be executed every time a user perform a request to the infected profile view page.
- **BS-SQLI-04: SQL Injection.** On the friend search page, the *search* field is vulnerable to SQL Injection. There is no sanitization and a crafted SQL statement can alter the initial SQL request and collect/modify/delete sensitive data.
- **BS-SQLI-05: SQL Injection.** On the account page, the *email* field is vulnerable to SQL Injections. There is no sanitization and a crafted SQL statement can alter the initial SQL request and collect/modify/delete sensitive data.
- **BS-CSRF-06: CSRF.** On the admin edit client page, the form is vulnerable to CSRF. An attacker may trick a user into making this request without his consent.
- **BS-PE-07: Privilege Escalation.** The create administrator action is vulnerable to Privilege Escalation. An authenticated user can send a crafted request to create a new administrator account.

8.1.3/ STUD-E

This case study is an e-learning Web-based application, named *stud-e*, that is currently used by more than 15.000 users. It provides 3 main profiles: *students*, *teachers*, and *administrators*. Students can access and download material of their courses, practice quiz and exercises, can participate to their exams and review their scores, can 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's inscriptions, teachers management, privilege definition and parameter settings.

The application has been under development for 10 years, and several releases have been produced and used. Both server-side and client-side parts of the application use

¹<https://developer.paypal.com/developer/accounts/> [Last visited: August 2015]

programming languages. The server uses PHP to (i) manage user sessions and identifications, (ii) produce the HTML pages, (iii) interact with a MySQL database. The client-side part of the application uses Javascript and CSS for user interface matters. This application is of the kind *infinite-urls*, meaning that every single produced page is accessed through a unique time-stamped identifier. There is also a custom URL-rewriting mechanism.

It is important to note that some effort has been put on security-related matters:

- all non-user data are encrypted (e.g. session data are encrypted, actions are encrypted and thus do not appear in HTTP messages, database entries such as identifiers or keys are encrypted and hence do not appear in HTTP messages, ...),
- during page production, the server sanitizes every single piece of information retrieved from the database,
- user input validation occurs at client-side and server-side.

8.1.4/ MEDIPEDIA WEB PORTAL

In this section we detail the *Medipedia* Web portal by introducing its main features and the identified security and legal risks. This case study is brought by *Info World* in the context of the European FP7 RASEN Project.

Presentation of the Medipedia Web application The *Medipedia* Web portal is a complex eHealth system that is accessible by creating a free user profile. The system acts as a Web portal that provides articles and news that are relevant for the prevention, treatment and control of diseases commonly occurring in the Romanian population. Users can access the portal via one of the 3 roles: public user, registered user and medical personnel. Each role has clearly defined rights and limitations. Currently, the *Medipedia* community includes over 40.000 registered users as well as over 150.000 weekly visitors, a linear increase of around 20% from 2014.

Everyone can access the public section that includes the articles section together with medicine brochures, suppliers and a medical forum where they can interact with other users or registered physicians. In addition, registered users can access their electronic health record that acts as the hub of their electronic healthcare data. As it includes sensitive personal data, access to the health records are provided based on a signed agreement and data is secured via the user's account. Users can provide other users, such as family members or physicians registered within the platform access to some or all of their stored data, on the basis of a signed agreement. Moreover, granting temporary access rights to registered physicians is also possible for the purposes of gaining a second opinion. In this way, physicians can access the patient's medical history in a centralized manner during the patient encounter, with no time or location constraints.

Medical data can be added to a user's record in two ways. First of all, organizations that are interconnected with *Medipedia* can add the results of medical or laboratory examinations to the user's account automatically and securely, while preserving the user's right to privacy. Currently, the *Medipedia* platform is integrated with *Medcenter*, a nationwide Romanian network of over 40 clinics that provides consultations and laboratory analyses.

Medical results obtained outside of the Medcenter network can be added by registered users manually.

Technical and legal aspects The Medipedia portal is part of Info World's portfolio for the management of care and financial processes in clinical units. Together with its systems for hospital, laboratory, pharmacy and imaging management, Info World solutions cover the entire spectrum of requirements for patient management and care. From a technical standpoint, all Info World products are Health Level 7 (HL7) v3 compliant [2] in order to ensure interoperability between the company's products as well as facilitate integration with components provided by other vendors. In order to facilitate interoperability, reduce software redundancy the HL7 group have defined a number of components that facilitate the management of patients, patient encounters and medical records [25].

The Medipedia platform employs several such components that were developed in-house, but according to HL7 interoperability specifications. A detailed description of these services in a relevant context for Medipedia is available at [74]. The Entity Identification Services (EIS), which manages the involved entities such as external organizations, clinical locations, practitioners and registered users. All medical documents are attached to the account of a user from the EIS service. The management of medical records is handled by the Resource Location and Reporting Service, that implements Integrating the Healthcare Enterprise (IHE)'s XDS Cross Enterprise Document Exchange profile and understands HL7 v3, CDA v2 and DICOM messages [38]. Medical documents uploaded by the user or transferred to their profile by the system are stored within this service. At the base of these services we find the Enterprise Vocabulary Service, which provides common and machine translatable clinical vocabulary services and is employed within all medical documents.

These services can be accessed after authentication and authorization by the Security Service, which uses a role-based access control model. The architecture of the security services itself is compliant with the AuditTrail and Node Authentication, Cross Enterprise User Authentication and Document Digital Signature profiles of the IHE as detailed within [38]. This ensures the confidentiality, integrity and availability of the system, together with recording detailed audit logs that allow post-mortem analysis, with important significance in tracing healthcare decisions. The services environment for Medipedia, including the security components is illustrated in Figure 8.1. Physically, the platform is currently deployed in a secured data-centre that is off-site from the company locations.

8.2/ WEB APPLICATION VULNERABILITY SCANNERS

To evaluate the precision and accuracy of the PMVT approach, we have confronted it with Web application vulnerability scanners. We have selected two scanners, one commercial and one open-source, based on their crawling capacity and vulnerability detection rates, as specified in the dedicated website SecToolMarket².

We present both scanners along with their characteristics in the next sections.

²<http://www.sectoolmarket.com/> [Last visited: August 2015]

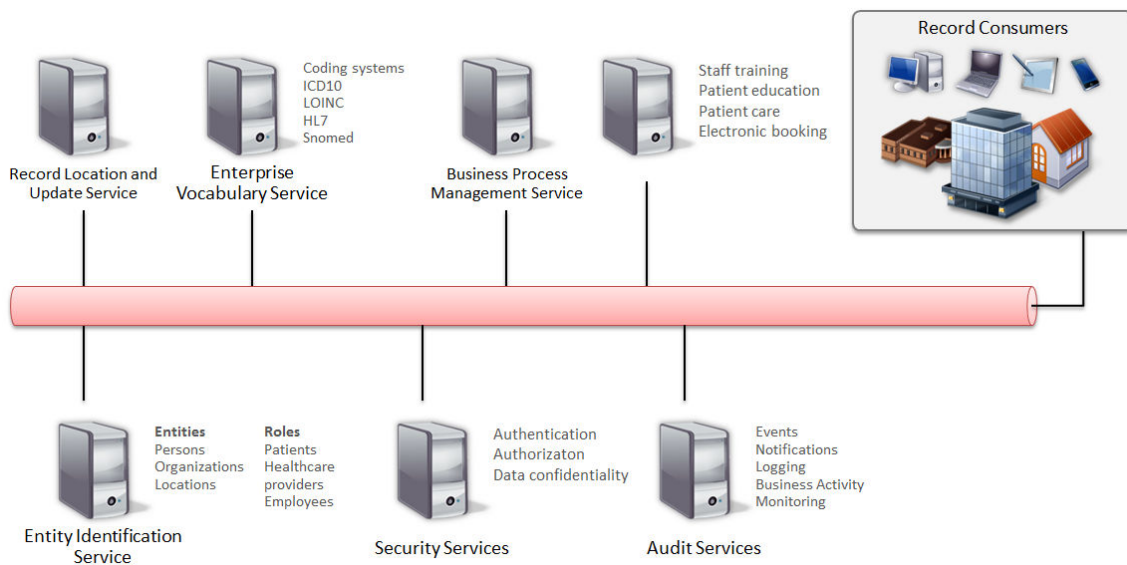


Figure 8.1 – Medipedia Services Architecture

8.2.1/ IBM APPSCAN

IBM AppScan³ is a Web vulnerability scanner originally developed by Israeli Software Company Sanctum whose first release came out in 1998. Since 2007 and following multiple redemptions of the product, AppScan development and maintenance continued with IBM. AppScan enables to verify the presence of security vulnerabilities in Web applications during the development process. The product learns the behavior of each tested application, whether an external application or developed internally, and then test all its functions for the most common vulnerabilities, such as XSS, SQL Injections, CSRF, as well more complex and specific vulnerabilities that are linked to a particular Web framework (e.g., Symfony, Zend, etc.) or Content Management System (e.g., Wordpress⁴, Joomla⁵, etc.). As such, AppScan bombards each user input of the Web application with thousands of attack vectors.

To access parts of the application that require authentication, AppScan asks users to record a login session. AppScan then replays the session before each attack. If the automated crawling module failed at discovering parts of the application, AppScan provides a manual crawling module, consisting of a Web browser. Test engineers have to navigate through the Web application, and trigger the functionalities they want to be tested.

AppScan allows the testing of type 1 multi-step XSS by keeping records of each injected XSS vector, and then crawling the application a second time to look for these vectors. It also allows the testing of type 2 multi-step XSS with manually defined multi-step sequences: it consists of a sequence of actions the scanner must perform in order to reach parts of the application that require this exact sequence to be accessible, or before observing the result of an attack.

Privilege Escalation can be tested as well. It requires two scans: one configured with login credentials leading to high privileges, and another one with credentials leading to less privileges. The scanner then tries to force-browse URLs it could access during the

³<http://www-03.ibm.com/software/products/en/appscan> [Last visited: August 2015]

⁴<https://www.wordpress.com> [Last visited: August 2015]

⁵<http://www.joomla.org/> [Last visited: August 2015]

first scan through the GUI but could not during the second. If the server response from the force-browsed URLs is equal to the server responses from the first scan, AppScan reports a potential vulnerability.

AppScan generates post-evaluation reports in Word and PDF format containing the list of security vulnerabilities discovered by the application. Note that the price of the software is non-negligible (\$ 9,500 to over \$ 36,000 depending on the license longevity and the software version), making it hardly accessible for smaller companies or independent developers.

8.2.2/ IRONWASP

IronWasp⁶ has been created and is maintained by Lavakumar Kuppan. This Open Source software allow users to scan Web applications in order to detect vulnerabilities. Unlike other scanners, such as AppScan, which aims at being relatively simple to use for people who have no specific expertise in penetration testing, IronWasp allows its users to perform automatic detection scans as well as develop Python or Ruby scripts to extend the original behavior of the scanner. However, since our objective is to compare the accuracy and precision of vulnerabilities detection between PMVT and point-and-shoot scanners, only the built-in modules of IronWasp were taken into account.

As with AppScan, it is possible to record a login session to access the authenticated area. It is also possible to manually crawl through the different pages of the application to focus on parts that were not discovered during automated crawling. In addition, IronWasp relies on a similar testing method for the detection of privilege escalations.

8.3/ EXPERIMENT RESULTS

In this section, we present and discuss the experiments results from the PMVT toolchain and the two vulnerability scanners.

8.3.1/ EXPERIMENTAL SETUP

All experiments have been conducted on a Windows 7 machine, with 8GB of RAM. The 3 dummy Web applications were powered by a local Web server directly installed on the machine (using Xampp⁷. Contrariwise, Stud-e was installed in a dedicated Apache/PHP/MySQL server accessible through Femto-ST intranet. Medipedia was installed on a windows Server 2008 R2, with IIS as Web server and SQL Server as DBMS.

We based our study on two factors: **vulnerability detection rates** and **time spent** (for PMVT, the time spent on each of its activities, and for scanners, the time spent on configuration and execution). Time spent on vulnerability scanners encompasses configuration, crawling, and 2 or 3 scans (one for each user role). Time spent on PMVT encompasses model design (designated by the letter **M**), test generation (designated by the letter **G**), test concretization (designated by the letter **C**), and test execution (designated by the letter **E**).

⁶<https://ironwasp.org/> [Last visited: August 2015]

⁷<https://www.apachefriends.org/index.html>

We used a detected (✓) / undetected (✗) metric for the detection of each featured vulnerability in dummy applications. However, since vulnerabilities present in real-life case studies were not known in advance, metrics for these case studies are presented as a set of 3 values:

Total number of vulnerabilities detected (True positives / False positives)

We define as *total number of vulnerabilities detected* the number of reported vulnerabilities per tool, for one vulnerability type. *True positives* are vulnerabilities that have been detected whose presence was confirmed after manual investigation. Contrariwise, *false positives* are detected vulnerabilities that revealed to be non-existent after manual investigation.

For the two scanners, we separated the results depending on the crawling mode: *Automated* for automated crawling and scanning, and *Manual* for Manual crawling and/or additional configurations (e.g., multi-step sequences in Appscan).

8.3.2/ DUMMY WEB APPLICATIONS

The objective of the 3 dummy applications is to compare PMVT and scanners within a controlled environment. Vulnerabilities present in these applications are known and it allowed us to not only evaluate the false positives rate of the approaches, but also their false negatives rate.

Table 8.1 presents experimentation results on *Wackopicko*. This application served a specific purpose in the evaluation: we used *Wackopicko* to evaluate how scanners perform today compared to when the website was designed as part of [22], in order to assess whether they have improved their crawling capacity and detection rates since.

As given in the table, the prototype implementation of the PMVT approach had the best detection rate, finding all vulnerabilities except the XSS behind Flash. This kind of vulnerability is not taken into account in PMVT, since the current version of Selenium cannot interact with Flash objects.

We met some problem with IronWasp on Wackopicko: we were unable to record a login sequence, nor create a login handling session. The scanner was thereby limited to the anonymous section of the Web application. It did find the vulnerabilities present in this area, but was not able to access the authenticated area, even when we involved its manual crawling module.

AppScan performed well, finding most of the vulnerabilities in automated mode. However, it could not detect the multi-step XSS vulnerability, even when using its manual crawling module. We decided to define a multi-step operation sequence starting at the injection point and ending at the storing point, but not change in the results was detected.

Regarding the time spent, scanners provided results at least five times faster than the PMVT approach. Indeed, it took 1 hour to design the PMVT model of Wackopicko, 45 minutes to generate test cases, 3 hours to concretize them, and 15 minutes to execute them.

Results of the experiment conducted on *Cuiteur* are displayed in Table 8.2. The prototype implementation of the PMVT process gave again the best results: almost all vulnerabilities were detected.

AppScan gave good results as well, and was able to detect most injection vulnerabilities, except for the multi-step XSS and the stored XSS behind JQuery. Using the manual crawl-

	IronWasp		AppScan		PMT
	Auto.	Manual	Auto.	Manual	
WP-RXSS-01	✓	✓	✓	✓	✓
WP-SXSS-02	✓	✓	✓	✓	✓
WP-RSQLI-03	✓	✓	✓	✓	✓
WP-JSXSS-04	✗	✗	✓	✓	✓
WP-SSQLI-05	✗	✗	✓	✓	✓
WP-MSXSS-06	✗	✗	✗	✗	✓
WP-FLRXSS-07	✗	✗	✗	✗	✗
Time Spent	0h20	1h15	0h35	1h	5h
Time Spent	0h20	1h15	0h35	1h	M / G / C / E 1h / 0h45 / 3h / 0h15

Table 8.1 – Wackopicko: Experiment Results

ing module provided new URLs to the tool, but it did not change the results. Therefore, we had to define a multi-step sequence containing the submission of a cuit for preview and its final submission. With this sequence, AppScan was able to detect the multi-step XSS vulnerability. However, even with this technique, the stored XSS vulnerability behind JQuery could not be detected.

We were able this time to record a login sequence with IronWasp, but the scanner performed poorly in automated scanning mode and was not able to detect any vulnerability. Thus, we relied on its manual crawling module to explore the application and feed URL to the scanner. It was then able to detect all XSS and SQL Injections, except for the XSS vulnerability behind JQuery. However, the admin area could not be accessed with IronWasp since users have to be logged in as standard users, then authenticate as admins on another page, which is not supported by IronWasp.

Note that none of the techniques were able to detect the Privilege Escalation vulnerability. This is due to the implementation of the *delete user* page, i.e. once the user is deleted the page redirects to the previous page. Indeed, when admin triggers this action they are redirected to the admin panel page, but when normal users try to force-browse the delete user page, the action is triggered but they are redirected to the home page of Cuiteur. Therefore, output pages between admin and standard users are different, even though the action was indeed triggered in both cases. This means the verdict assignment technique implemented by all approaches is partial, and a new strategy should be investigated.

	IronWasp		AppScan		PMVT
	Auto.	Manual	Auto.	Manual	
<i>CT-RXSS-01</i>	✗	✓	✓	✓	✓
<i>CT-SXSS-02</i>	✗	✓	✓	✓	✓
<i>CT-MSXSS-03</i>	✗	✗	✗	✓	✓
<i>CT-JQXSS-04</i>	✗	✗	✗	✗	✓
<i>CT-SQLI-05</i>	✗	✓	✓	✓	✓
<i>CT-SQLI-06</i>	✗	✓	✓	✓	✓
<i>CT-CSRF-07</i>	✗	✗	✗	✗	✓
<i>CT-PE-08</i>	✗	✗	✗	✗	✗
Time Spent	0h20	1h	0h20	1h30	M / G / C / E 1h30 / 1h / 4h / 0h30

Table 8.2 – Cuiteur: Experiment Results

Interestingly, none of the scanners were able to see past the decoy token that Cuiteur inserts in all its forms. It means that scanners only check for the presence of a token, but do not actually conduct a real attack. This appears as a major weakness of these tools. As with Wackopicko, scanners were at least five times faster than the PMVT approach. Indeed, it took 1h30 to design the PMVT model of Cuiteur, 1h to generate test cases, 4h to concretize them, and 30min to execute them.

Table 8.3 shows the results of the experiment conducted on *Bookshop*. The prototype implementation of the PMVT process obtained the best results by detecting all the featured vulnerabilities. It took 1h30 to design the PMVT model, 1h to generate test cases, 3h to concretize them, and 30min to execute them.

Contrariwise, scanners were unable to handle the dynamic token generated for each page. Thus, they were unable to automatically crawl the Web application, and did not detect any vulnerability. When relying on their manual crawling module, their detection rate did not improve. This is due to their attack process: scanners reuse HTTP requests emitted during the crawling phase, and re-send them with attack vectors. The presence of a dynamic token prevents them from successfully conducting attacks.

	IronWasp		AppScan		PMVT
	Auto.	Manual	Auto.	Manual	
<i>BS-RXSS-01</i>	✗	✗	✗	✗	✓
<i>BS-SXSS-02</i>	✗	✗	✗	✗	✓
<i>BS-SXSS-03</i>	✗	✗	✗	✗	✓
<i>BS-SQLI-04</i>	✗	✗	✗	✗	✓
<i>BS-SQLI-05</i>	✗	✗	✗	✗	✓
<i>BS-CSRF-06</i>	✗	✗	✗	✗	✓
<i>BS-PE-07</i>	✗	✗	✗	✗	✓
Time Spent	0h30	2h	0h45	2h30	M / G / C / E 1h30 / 1h / 3h / 0h30

Table 8.3 – Bookshop: Experiment Results

8.3.3/ MEDIPEDIA

Medipedia is a large real-life website with many different pages and possible user interactions. The experiment results from the scanners and PMVT have been aggregated in Table 8.4.

As for Wackopicko, we were unable to record a login sequence in IronWasp, nor define a login handling session. Therefore, the scanner could not access the authenticated area of Medipedia, which represents a consequent part of the application. Nonetheless, we were able to use the automated crawler to find 3 Reflected XSS vulnerabilities. We tried to manually crawl the authenticated area, but it did not improve the results.

AppScan obtained the highest detection rate, with 8 Reflected XSS vulnerabilities. Moreover, its automated crawling module was able to visit most parts of the application, therefore we did not have to perform manual crawling.

PMVT obtained good results, but missed on 3 Reflected XSS vulnerabilities. This is because of the places of injection: each undetected vulnerability resulted from the injection of HTTP headers or hidden tokens sent as HTTP request parameters. We did not represent these inputs during the modeling activity and therefore did not test them.

Note that both AppScan and PMVT had false positives for CSRF. Indeed, Medipedia produces the same output regardless of the genuineness of requests. It duped PMVT's verdict assignment, which relies on Web page comparison, and it probably duped AppScan as well for the same reason. These results show that, again, Web page comparison

	IronWasp		AppScan		PMVT
	Auto.	Manual	Auto.	Manual	
<i>Refl. XSS</i>	3 (3/0)	3 (3/0)	8 (8/0)	N/A	5 (5/0)
<i>M.Step XSS</i>	0	0	0	N/A	0
<i>SQL Inj.</i>	0	0	0	N/A	0
<i>Blind SQL Inj.</i>	0	0	0	N/A	0
<i>CSRF</i>	0	0	25 (0/25)	N/A	8 (0/8)
<i>Priv. Escal</i>	0	0	0	N/A	0
Time Spent	3h30	4h	13h	N/A	M / G / C / E 2h / 6h / 8h / 1h

Table 8.4 – Medipedia: Experiment Results

is not always reliable, which foregrounds the need for designing a new verdict assignment strategy.

The time spent by AppScan and PMVT are very close. Although AppScan does not involve human intervention once the manual crawling activity is completed, it took 30 minutes to configure it and the scan activity lasted 12 hours and a half. PMVT, on the other hand, requires human intervention for 60% of the process: 2 hours for model design, 6 hours for test generation, 8 hours for test concretization, and 1 hour and a half for text execution.

8.3.4/ STUD-E

As previously stated in Section 8.1.3, *Stud-E* is a complete custom Web application, developed in PHP. It is a single-URL website: the nature of the server response depends on the HTTP parameters present in the request.

Results from the scanners and the PMVT toolchain are presented in Table 8.5. Vulnerability scanners were unable to automatically crawl the application: the presence of a frame-set and the absence of unique URLs is the most probable cause for their incapacity. IronWasp could not detect any vulnerability, even when supplied with manually crawled URLs. On the other hand, AppScan was able to use results from manual crawling to scan the application. The scanner obtained the best results with the detection of 9 Reflected XSS, 2 multi-step XSS, and 16 blind SQL Injections. However, it produced two false positives for CSRF, reinforcing the fact that its technique for CSRF is flawed.

As for Medipedia, PMVT obtained good results but missed on several XSS and SQL

	IronWasp		AppScan		PMVT
	Auto.	Manual	Auto.	Manual	
<i>Refl. XSS</i>	0	0	0	9 (9/0)	2 (2/0)
<i>M.Step XSS</i>	0	0	0	2 (2/0)	3 (3/0)
<i>SQL Inj.</i>	0	0	0	0	0
<i>Blind SQL Inj.</i>	0	0	0	16 (16/0)	9 (9/0)
<i>CSRF</i>	0	0	0	3 (0/3)	0
<i>Priv. Escal</i>	0	0	0	N/A	0
Time Spent	0h30	5h	1h30	19h00	M / G / C / E 2h / 7h / 9h / 2h

Table 8.5 – Stud-e: Experiment Results

Injects, because of non-handled injection points. Nonetheless, PMVT was able to detect a complex multi-step XSS spread on different user sessions: the injection was performed using an administrator role, and the vulnerability could only be observed when logged in as a student. None of the scanners were able to detect this vulnerability. PMVT relied on the information captured in the model to determine where the input is rendered back, which lead to a session change. This result strongly supports the use of PMVT models for the detection of complex vulnerabilities.

The time spent on AppScan and PMVT are very close. Although AppScan does not involve human intervention once the manual crawling activity is completed, it took one hour to configure it and the scan activity lasted 18 hours. PMVT, on the other hand, required human intervention during 70% of the process: 2 hours for model design, 7 hours for test generation, 9 hours for test concretization, and 2 hours for text execution.

8.4/ THREATS TO VALIDITY

These experiments, like any others, have some limitations.

First, since we personally conducted all experiments, there is a difference between our expertise of PMVT and our expertise of scanners: we had to learn how to properly configure and execute these tools in order to use them efficiently. However, we don't claim to have become experts on using scanners, and the eventuality that we missed a few configuration settings during the experiments exists. Nonetheless, to mitigate this issue,

we worked with penetration testers from a french company called NBS System⁸ during the early days of PMVT to get professional insights about manual penetration testing in general and how to use scanners efficiently.

Second, we knew in advance the location of all vulnerabilities in the 3 dummy applications. Even though we modeled these applications thoroughly, model creation may have been somehow influenced by this knowledge. As such, it is possible that parts of the dummy applications were unconsciously looked off during model creation if they did not feature any vulnerability.

Third, we have only reported results that concerned the 4 vulnerability types addressed by PMVT. Scanners found a few vulnerabilities on Medipedia and Stud-E that were not reported because out of scope. Therefore, we depicted a somewhat reductive image of scanners, and readers should acknowledge that these tools can test for a greater set of vulnerability types [51].

8.5/ DISCUSSION

We have evaluated the toolchain that supports the PMVT process on 5 Web applications, and compared its results with those obtained using AppScan and IronWasp. We discuss these results by responding to the research questions we defined in Section 1.3.

RQ1 To what extent test patterns applied to a model of the Web application under test improves the accuracy and precision of vulnerability detection?

Overall, The PMVT approach obtained the best results during the experimentation. First, we were able to conduct successful attacks on each Web application, as opposed to scanners who had issues to crawl Stud-E and were unable to crawl Bookshop. Second, PMVT enables to detect complex attacks such as type 2 multi-step XSS, which are present in Wackopicko and Cuiteur and were not detected by scanners. Moreover, we were able to detect type 1 multi-step XSS that are spread across several user sessions (e.g., injection as a standard user, resurgence as an admin), which scanners missed. Third, CSRF detection in PMVT is more efficient than the one implemented in scanners. We were able to dupe AppScan and IronWasp into missing on vulnerabilities by introducing decoy tokens in Web forms, which we detected with PMVT.

However, scanners had better results at finding “simple” vulnerabilities. We qualify as simple the vulnerabilities that only require understanding of the application’s logic to reach the page on which the attack is performed. This is the case for technical vulnerabilities such as Reflected XSS and SQL Injections, for instance. Once the targeted page has been reached, the attack process is purely technical: (i) inject a payload, (ii) submit the data, and (iii) analyze the immediate response from the server.

During this thesis, we have decided to create all the attack mechanisms from scratch, in order to have full control over the entire toolchain and have the ability to easily tweak / enhance / change any attack process that showed weaknesses. As such, we have solely relied on the information captured in the model to generate test cases, which means that any user input that has not been modeled is not tested. It is the reason why PMVT underperformed compared to scanners for the detection of Reflected XSS and SQL Injections. On the other hand, scanners were able to detect more of these vulnerabilities because

⁸<https://www.nbs-system.com/> [Last visited: August 2015]

they systematically conduct injections in every user input of a page. In PMVT, only the modeled inputs are injected, and we did not model every possible input during experimentation. For instance, Stud-E contains many hidden fields containing tokens, which are inserted in every page of the application. We missed them during the modeling activity, and therefore did not test for them. Scanners revealed that a few of these hidden fields were vulnerable to XSS and SQL Injections, due to an incomplete sanitization. Nonetheless, such exhaustive injection technique could be integrated in PMVT, without unbearable cost. We discuss this as a possible future work in Section 9.2.1.2.

PMVT composes test patterns and a model of the Web application, which improves the accuracy and precision of vulnerability detection by finding complex vulnerabilities missed by scanners, and testing Web applications that scanners are unable to crawl.

RQ2 To what extent is it possible to provide generic test patterns for Web application vulnerabilities?

Thanks to the modeling notation and the additions we brought to the test purpose language, we were able to design seven generic test purposes based on vulnerability test patterns, to detect four types of vulnerabilities. We successfully applied these test purpose on each model of the five Web applications present in the experimentation, without modification. Test cases generated from the test purposes enabled to detect many instances of each vulnerability type.

Indeed, PMVT relies on a UML4MBT class diagram to represent the metamodel of the DASTML language. This class diagram is therefore unique and not dependant of the target Web application. With this specific notation, we provide means for test purposes to be generic: test purposes only use elements from the UML4MBT class diagram and generic instances (instances of classes such as WebAppStructure, WAUT, Threat).

PMVT makes it possible to design generic test patterns to generate vulnerability test cases, for the detection of 4 vulnerability types.

RQ3 To what extent such Web application vulnerability testing process (based on patterns and models) may be automated?

We were able to automate most of the PMVT process:

- Test purposes are generic; which means they have to be designed once and can be used as is afterwards, for any project. Therefore, we consider the incorporation of test purposes in a PMVT test project as automatic.
- Test selection and generation are automated; thanks to the additions PMVT brought to the initial CertifyIt process to enable the CertifyIt test generation engine to compute test cases from PMVT models and test purposes.
- Test concretization is 90% automated; the PMVT test publisher generates a complete Mavenized Java project containing a test harness. Each operations from the model is translated in an executable script with Selenium primitives. The only task remaining manual is the matching between abstract and concrete data.
- Test execution and Verdict assignment are automated; thanks to the PMVT test publisher, which relies on JUnit and includes dedicated observation techniques in the test harness to assess the existence of vulnerabilities.

The only activity that stayed fully manual is the design of PMVT models. However, thanks to the DASTML language, we were able to strongly ease this activity. The creation of DASTML models is straightforward, close to the real application, and has proven to be 3 times faster than directly designing PMVT-compliant UML4MBT models.

PMVT is a semi-automated MBT approach dedicated to vulnerability testing.

CONCLUSION

Contents

9.1 Summary	133
9.2 Future works	134
9.2.1 Improvements of the PMVT Approach	135
9.2.2 Evolutions of the PMVT Approach	138

This chapter aims to summarize the work that was produced during this thesis and proposes five research perspectives to present possible extensions and continuations of this work.

9.1/ SUMMARY

Current techniques to test / counteract / eliminate vulnerabilities are not precise and accurate enough. On the one hand, manual penetration testing is the most used technique that enable to detect many vulnerabilities. However, such technique rely on the experience and know-how of penetration testers, who mostly proceed manually by writing specific attack scripts tailored to a given Web application. This makes the dissemination of their techniques very difficult, and the impact of this knowledge very low. Moreover, penetration testers often have to work in time boxes, and as a consequence have to reduce their detection scope accordingly. On the other hand, Web application vulnerability scanners are able to the detect major vulnerabilities, but they lack precision and accuracy since they have no knowledge of the application's logic. Consequently, they often generate false positive and false negative results, and deep human investigation is often required.

In this thesis, we introduced an approach called Pattern-driven and Model-based Vulnerability Testing (PMVT) based on vulnerability test patterns and models for the detection of Web application vulnerabilities. This original approach improves the precision and accuracy of vulnerability testing, by producing executable test scripts faster than if written manually, and by enabling to conduct complex and sophisticated attacks to detect vulnerabilities that scanners fail to find, thanks to patterns and models.

Models in PMVT are designed using a dedicated domain-specific modeling language, called DASTML. It allows the modeling of the global structure of the targeted 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 algorithm.

Test purposes represent generic vulnerability test patterns, i.e. independent from the targeted Web application. They define abstract vulnerability detection scenarios that drive the test generation process through the model, by reasoning in terms of states to reach and operations to call.

The proposed approach thus consists of instantiating the abstract scenarios regarding the Web application model in order to automatically generate test sequences, which enable to reveal potential vulnerabilities w.r.t. the test pattern objective.

To experiment and evaluate the PMVT approach, a full automated toolchain, from model design to test execution, has been developed. It has been built on top of an existing MBT tooling, called CertifyIt. As such, PMVT extends the existing artifacts of CertifyIt in order to generate vulnerability test cases: its initial test purpose language has been augmented in order to formalize generic vulnerability test patterns, and its modeling notation, UML4MBT, has been specialized to meet PMVT's need. The PMVT toolchain enables to test for four of the most critical and widespread Web applications vulnerabilities, that is to say Cross-Site Scripting, SQL Injections, Cross-Site Request Forgeries, and Privilege Escalations.

A thorough experimentation on two real-life Web portals (Medipedia and Stud-E) and three dummy applications (Wackopicko, Cuiteur and Bookshop) enabled to evaluate the PMVT approach, as we were able to detect many vulnerabilities in each case studies. A comparison with existing automated testing solutions (i.e., vulnerability scanners) showed its effectiveness to generate more accurate vulnerability test cases and conduct complex and sophisticated attacks that current automated solutions are unable to perform.

These benefits directly stem from the combination of PMVT models, capturing the logical and structural aspects of the application under test, and the test purposes, 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 vulnerability regression tests on updated or corrected further versions of the Web application under test.

Furthermore, we have laid the foundations for the extension of MBT to address vulnerability testing. In the next section, we present five future works to improve and extend the PMVT approach.

9.2/ FUTURE WORKS

As showed by the experiment results (see Chapter 8), the PMVT approach provides means to test for complex vulnerabilities on custom-made Web applications. Besides these research results, the experiments showed possible improvements of the method and the toolchain. Therefore, we propose in the next sections three improvements of the toolchain to increase its effectiveness and efficiency, and two extensions to the process in order to address other vulnerability types.

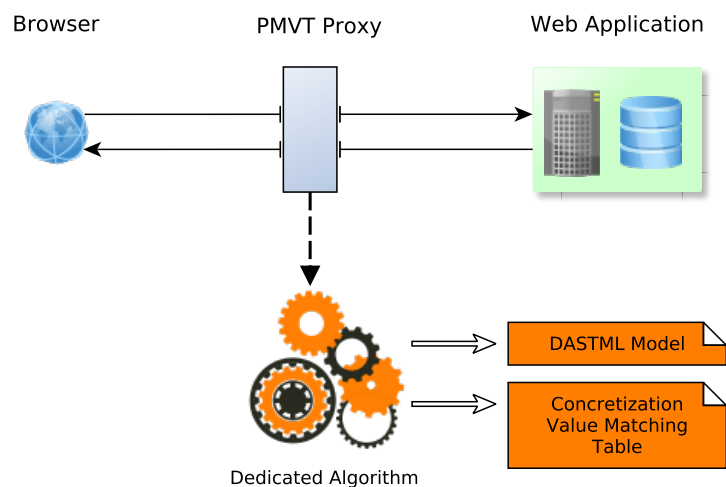


Figure 9.1 – PMVT Model Inference Process Using a Proxy

9.2.1/ IMPROVEMENTS OF THE PMVT APPROACH

The time and resource cost related to the deployment of the approach are still higher than scanners. As well, the verdict assignment strategy that was employed for CSRF and Privilege Escalations is flawed, as it has lead to several false positives and false negatives on Cuiteur and Medipedia. To overcome these problems we propose three possible ways of improving the current toolchain in terms of effectiveness and efficiency: model inference, composing PMVT and scanners, and a verdict assignment strategy based on database content analysis. We detail these improvements in the next sections.

9.2.1.1/ MODEL INFERENCE

PMVT inherits MBT's weakness regarding deployment cost: model design and test concretization are both time consuming and tedious tasks. During this thesis, we reached a first level of simplification by introducing the dedicated DASTML modeling language, which allows to capture Web applications structure and logic in a much simpler way than with UML4MBT. In addition, the PMVT test publisher infers most of the concretization activity by creating a complete test harness in which model operations have been implemented using Selenium primitives.

A possible second level of simplification concerns the capture of user traces. The actual practice to create a DASTML model is to manually visit the Web application under test, and then manually report the relevant information in the model. The second part of this practice could be semi-automated by recording the interactions between the browser and the Web application to infer the model. The recording of browser/application interactions may be done in two ways.

The *first solution*, as depicted in Figure 9.1, is to insert a proxy (e.g., using libraries such as *Browsermob Proxy*) between the browser and the Web application to record all HTTP requests and responses. Then, a dedicated algorithm, using a similar approach than the one proposed in [5] for the identification of unique pages (or states), would analyze the traces and infer a model the same way it is manually done currently. The underlying idea

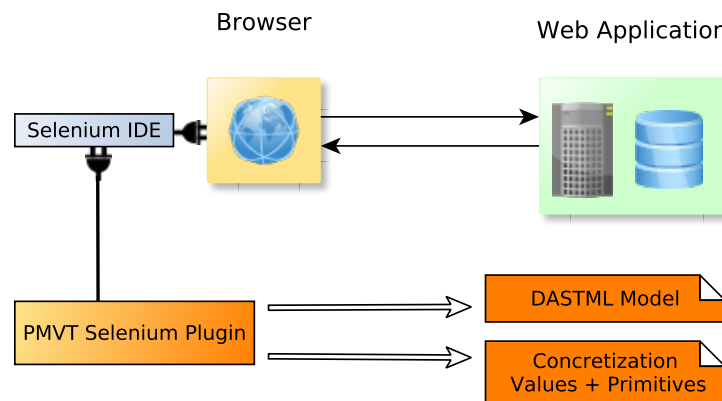


Figure 9.2 – PMVT Model Inference Process Using Selenium IDE

is to manually crawl Web applications using a Web browser, similarly to scanners' manual crawling module, to automatically infer PMVT models.

Capturing the logical aspects of Web applications (e.g., user inputs resurgence, user roles, multi-step actions, etc.) requires human intelligence, and therefore would be done manually through the browser by developing a browser extension. This extension would provide features to allow test engineers to define multi-step actions (e.g., with a recording mechanism), characterize the current session type, and link page content to user inputs (for instance by highlighting a text fragment and, with a right-click, select a user input from a list created automatically from the intercepted HTTP requests).

In addition, the use of a proxy would enable to infer the concretization value matching table (i.e., mapping between abstract data from the model and concrete data from the real application), since all abstracted data would be automatically computed from concrete data.

The *second solution*, as depicted in Figure 9.2, is to build a PMVT model inference plugin on top of Selenium IDE ¹. Selenium IDE is a Firefox extension that allows test engineers to record sequences of actions on the Web application under test, such as button clicks, form fields filling, URL loading, and so on. Each recording constitutes a test case that can be automatically replayed. An export function enables to generate program test cases, for instance JUnit test cases, filled with Selenium primitives w.r.t. the sequence of actions. Moreover, Selenium IDE has a plugin system that allows for extension and customization. The objective would be to create a Selenium IDE plugin that uses records to infer PMVT models.

As for the first solution, the logical aspects of Web applications would be captured through the plugin by creating features to create multi-step actions (e.g., by grouping a fragment of a sequence of actions and marking it as multi-step), characterizing the current session type and linking page content to user inputs.

With this solution, it would not only enable to infer the concretization value matching table, but it would also allow to infer all the Selenium primitives that are necessary for test execution. This would considerably improve the automation of the test concretization activity, which currently relies on general assumptions about how PMVT actions translate best in real user actions.

¹<http://www.seleniumhq.org/projects/ide/> [Last visited: August 2015]

```

1  for_each literal $pagelit from "self.all_pages->select(p:Page|p.id)" on_instance was
2  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
3  "not(self.ongoingAction.ocllsUndefined() and
4      "self.current_page.id = PAGE_IDS::$pagelit)"
5  on_instance was
6  then use threat.execScannerOnPage()

```

Table 9.1 – Test Purpose combining PMVT and Scanners for All Injections

9.2.1.2/ COMPOSITION WITH SCANNERS

Experiment results showed that scanners are efficient at finding “simple” vulnerabilities, because they systematically conduct injections in every user input of a page. PMVT failed to detect some of these vulnerabilities because only the modeled inputs are injected, and not every input was modeled during experiments.

An interesting improvement of the PMVT process would consist of composing the navigation capacities of PMVT and the exhaustive testing capacities of scanners. If we take a look back at the structure of test cases generated by CertifyIt (see Section 4.1.3), we see that it is composed of two mandatory parts (we don’t consider the postamble since it is optional and has not been used in PMVT test cases): (i) the preamble, which represents all the interactions prior to the attack, and (ii) the body, which concerns the attack. Following this logic, PMVT would be responsible for the preamble of the test cases by relying on the information captured in the model. The body of test cases, which concerns the actual attack and verdict assignment, would be delegated to the attack module of a scanner.

Furthermore, such composition would extend the testing capacity of PMVT. Indeed, many injection-related vulnerabilities (such as Command Injections and LDAP Injections) share the same attack process with Reflected XSS and SQL Injections. It would be then possible to address all these vulnerability types at once, using a single test purpose.

A theoretical test purpose that illustrates the composition of PMVT and scanners to address all injections is depicted in Table 9.1. It consists of iterating the pages of the application, and for each page, use the model to navigate from the root URL to the page, and then delegate attacks to a scanner.

9.2.1.3/ VERDICT ASSIGNMENT

While verdict assignment for XSS and SQL Injections showed no weakness during experiments, several false positives were observed concerning CSRF and Privilege Escalations.

The verdict assignment strategy used to assess the presence of these two vulnerabilities consists of Web page comparison (as stated in Chapter 7). The problem of this technique is that it only relies on the information that the Web server is willing to share. It is an elaborated assumption that Web applications respond differently whether their functions have been successfully triggered or not, or that the server response will be identical regardless of the type of user that triggered the functions. For instance, Medipedia computes the same output whether form submissions have been performed or not, which lead to false positives when testing for CSRF. As well, Cuiteur redirects its administrators to the

administrator panel page once they modified a user account. If the user who triggered the function is not logged in as administrator, Cuiteur redirects him further, to the home page. It lead to false negatives when testing for Privilege Escalations: the function had been triggered but the result pages were different. Therefore, a new verdict assignment strategy must be designed.

A solution would consists of monitoring the database to collect and compare changes between a nominal usage of a function and an attack. Consider for instance an administrator panel with a function to create administrator accounts (e.g., a Web form). Testing this function for CSRF would first require to get a snapshot of the database prior to triggering the function in a nominal way (e.g., by an authenticated administrator through the GUI of the application). Then, another snapshot of the database is taken just after the function triggering, to extract the delta between the two snapshots. The same database snapshot acquisition process must be applied during the conduction of the CSRF attack. Finally, verdict assignment is performed by comparing the two deltas (after stripping out ids, timestamps and other data with temporal or random properties).

9.2.2/ EVOLUTIONS OF THE PMVT APPROACH

We propose two evolutions of the PMVT approach to increase its vulnerability coverage. The first concerns an extension of DASTML and the test purpose language to tackle Insecure Direct Object References. The second is about converting PMVT into an online MBT process.

9.2.2.1/ EXTENSION TO ADDRESS INSECURE DIRECT OBJECT REFERENCES

So far, the PMVT approach is able to generate vulnerability test cases for four vulnerabilities: XSS, SQL Injections, CSRF, and Privilege Escalation. We believe that such approach could address much more vulnerability types by making additional extensions to the test purpose language and DASTML.

A possible extension would be to address Insecure Direct Object References. Along the lines of Privilege Escalations, this vulnerability type is due to an Access Control problem. It implies that users can access resources that are restricted to a single user (e.g., an account panel), for example by modifying the URL parameter that points to the ID of the account. The test scenario to address this vulnerability type is also very close to the one we relied on to generate Privilege Escalations test case: first access the resource as intended, then login as another user and try to access the same resource. If the resource is retrieved, then the Web application is vulnerable.

We have designed an hypothetical test purpose to address this vulnerability type, which is depicted in Table 9.2. It first consists of collecting a user, all the pages that qualify as user-specific, and all the other users. Then, the first part (lines 6-9) is about accessing the page when connected as the first user. The second part (lines 10-12) is about trying to access the same page when connected with another user. Finally, verdict assignment (line 13) is performed by comparing the two page outputs.

However, we could not integrate this test purpose into this thesis, because it requires several evolutions of the test purpose language and DASTML. First, it is not possible to use variables that contain instances in OCL expressions (it has been implemented only

```

1  for_each instance $user from "User.allInstances()" on_instance was,
2  for_each instance $page from "self.all_pages->select(p:Page|p.is_user_specific)"
3  on_instance was,
4  for_each instance $otherUser from "self.all_users->excluding($user)"
5  on_instance was,
6  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
7  "self.current_page = $page and self.current_user = $user" on_instance was
8  then use threat.collectPage()
9  then use was.reset()
10 then use any_operation_but #UNWANTED_OPS any_number_of_times
11 to_reach "self.current_page = $page and "and self.current_user = $otherUser"
12 on_instance was
13 then use threat.checkIDOR()

```

Table 9.2 – Test Purpose for Insecure Direct Object References in Pages

for literal values). Second, it is not possible to nest three *for_each*. The language is still considered as a proof-of-concept and can only nest two iterators at the moment. Third, DASTML does not allow to model users, nor to specify that pages are user-specific. The creation of users as part of DASTML could be done by declaring all users prior to the pages. Then, for login operations, use a user as parameter instead of credentials. Concerning user-specific pages, such property could be defined as a special keyword in their restrictions section. It could also be defined with the creation of a new section inside pages declaration. Indeed, there are many possibilities, and a thorough experiment would be necessary to decide which notation fits best.

9.2.2.2/ ONLINE MBT

PMVT is an offline MBT technique, which consists of generating test cases prior to their execution on the application. An interesting future work would be to transform PMVT into an online MBT technique.

Online MBT (or on-the-fly MBT) combines test generation and test execution: MBT tools interact directly with the SUT and test it dynamically. Such process consists of generating only a single test step from the model at a time and immediately execute it on the system under test. The test stops either if the stop conditions are satisfied (timeout, coverage condition satisfied etc.) or if an unexpected behavior has been detected. The main advantage of online MBT is indeed that the current execution results may influence the path to take through the model for the next execution step, as well as produce instant results (including feedback for modelers) both for test generation and execution. In component and embedded (cyber-physical) systems, online MBT is commonly used to detect operational faults.

Implementing PMVT as an online MBT technique would allow to further improve the precision of vulnerability test cases, perform more sophisticated attacks, and react on vulnerability discovery (e.g., to exploit it).

First, an online approach could improve the detection of injection-type vulnerabilities (XSS and SQLi), while reducing execution time. By injecting user inputs with supposedly illegal

characters that are commonly used in SQL Injections and/or XSS vectors, it would be possible to assess which characters are sanitized and which characters are not. Based on these ideas, the objective would be to refine attack vector lists by excluding vectors that don't rely on characters that were identified as sanitized. Second, an online approach could take the eventual hierarchy between vulnerabilities into account to perform complex and sophisticated attacks. Consider a Web application that protect its users against CSRF by checking the origin of their requests. The current attack process implemented in PMVT would not be able to test for CSRF since it performs the attack from an external Web server. However, if one of the user inputs is vulnerable to XSS, then it would be possible to use this vulnerability to conduct a CSRF attack from inside the Web application (see the SAMY worm² for an example of an XSS+CSRF attack). Therefore, whenever an XSS vulnerability is found, the test generation engine would react on it and proceed to the generation of XSS+CSRF test cases.

Third, online MBT would allow to go further than coupling vulnerabilities. It would enable to perform each four phases of manual penetration testing (as stated in Section 3.2.2, from reconnaissance to exploitation and maintaining access).

However, a suitable and scalable online testing process implies that test generation cannot be slower than intended by the test execution framework. The required synchronization between test generation and test execution must be ensured by implementing efficient test generation algorithms, in order to produce results in a given lapse of time. Furthermore, an online testing process should ensure the repeatability of the test generation activity for the same version of the Web application and from the same PMVT model, and should result in the same generated tests and the same test execution results.

²<http://namb.la/popular/> [Last visited: August 2015]

BIBLIOGRAPHY

- [1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure)*. Oxford University Press, 1977.
- [2] P. K. Andrikopoulos and Petros Belsis. Towards effective organization of medical data. In *Proc. of the 17th Panhellenic Conf. on Informatics, PCI '13*, pages 305–310, New York, NY, USA, 2013. ACM.
- [3] Brad Arkin, Scott Stender, and Gary McGraw. Software penetration testing. *IEEE Security & Privacy*, 3(1):84–87, 2005.
- [4] Elias Athanasopoulos, Vasilis Pappas, Antonis Krithinakis, Spyros Ligouras, Evangelos P. Markatos, and Thomas Karagiannis. xJS: practical XSS prevention for web application development. In *Proc. of the USENIX Conf. on Web application development (WebApps'10)*, pages 147–158, Boston, MA, USA, June 2010. USENIX Association.
- [5] Andrew Austin and Laurie Williams. One Technique is Not Enough: A Comparison of Vulnerability Discovery Techniques. In *Proc. of the 5th Int. Symposium on Empirical Software Engineering and Measurement (ESEM '11)*, pages 97–106, Banff, Alberta, Canada, September 2011. IEEE CS.
- [6] James Bach. Risk and Requirements-Based Testing. *Computer*, 32(6):113–114, June 1999. IEEE Press.
- [7] Jason Bau, Elie Bursztein, Divij Gupta, and John Mitchell. State of the Art: Automated Black-Box Web Application Vulnerability Testing. In *Proc. of the 31st Int. Symp. on Security and Privacy (SP'10)*, pages 332–345, Oakland, USA, May 2010. IEEE CS.
- [8] Eddy Bernard, Fabrice Bouquet, Amandine Charbonnier, Bruno Legeard, Fabien Peureux, Mark Utting, and Eric Torreborre. Model-based testing from uml models. In *Proc. of the Int. Workshop on Model-Based Testing (MBT'06)*, volume 94 of *LNI*, pages 223–230, Dresden, Germany, October 2006. GI.
- [9] Prithvi Bisht and VN Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 23–43. Springer, 2008.
- [10] Abian Blome, Martin Ochoa, Keqin Li, M. Peroli, and Mohammad T. Dashti. Vera: A flexible model-based vulnerability testing tool. In *Proc. of the 6th Int. Conf. on Software Testing, Verification and Validation (ICST'13)*, pages 471–478, Luxembourg, March 2013. IEEE CS.
- [11] Julien Botella, Fabrice Bouquet, Jean-Francois Capuron, Franck Lebeau, Bruno Legeard, and Florence Schadle. Model-Based Testing of Cryptographic Components –

- Lessons Learned from Experience. In *Proc. of the 6th Int. Conf. on Software Testing, Verification and Validation (ICST'13)*, pages 192–201, Luxembourg, March 2013. IEEE CS.
- [12] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, and Fabien Peureux. A test generation solution to automate software testing. In *Proc. of the 3rd Int. Workshop on Automation of Software Test (AST'08)*, pages 45–48, Leipzig, Germany, May 2008. ACM Press.
- [13] Fabrice Bouquet, Christophe Grandpierre, Bruno Legeard, Fabien Peureux, Nicolas Vacelet, and Mark Utting. A subset of precise UML for model-based testing. In *3rd int. Workshop on Advances in Model Based Testing (A-MOST'07)*, pages 95–104, London, United Kingdom, July 2007. ACM Press.
- [14] Josip Bozic, Dimitris E Simos, and Franz Wotawa. Attack pattern-based combinatorial testing. In *Proc. of the 9th Int. Workshop on Automation of Software Test*, pages 1–7. ACM, 2014.
- [15] Josip Bozic and Franz Wotawa. Security testing based on attack patterns. In *IEEE 7th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW'14)*, pages 4–11. IEEE, 2014.
- [16] Phillip W. Brunst. Terrorism and the internet: New threats posed by cyberterrorism and terrorist use of the internet. In *Wade, Marianne/Maljevic, Almir (Eds.): A War on Terror? The European Stance on a New Threat, Changing Laws and Human Rights Implications*, pages 51–78. Springer, 2009.
- [17] Matthias Buchler, Johan Oudinet, and Alexander Pretschner. Semi-Automatic Security Testing of Web Applications from a Secure Model. In *Proc. of the 6th Int. Conf. on Software Security and Reliability (SERE'12)*, pages 253–262, Gaithersburg, MD, USA, June 2012. IEEE CS.
- [18] Steve Cook and John Daniels. *Designing object systems*, volume 135. Prentice Hall Englewood Cliffs, 1994.
- [19] Frédéric Dadeau, Fabien Peureux, Bruno Legeard, Régis Tissot, Jacques Julliand, Pierre-Alain Masson, and Fabrice Bouquet. Test generation using symbolic animation of models. In *Model-Based Testing for Embedded Systems*, Series on Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 195–218. CRC Press, 2011.
- [20] Arilo C. Dias-Neto and Guilherme H. Travassos. A Picture from the Model-Based Testing Area: Concepts, Techniques, and Challenges. *Advances in Computers*, 80:45–120, July 2010. ISSN: 0065-2458.
- [21] Adam Doupé, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Enemy of the state: A state-aware black-box web vulnerability scanner. In *Proc. of the 21st USENIX Security Symposium*, pages 523–538, 2012.
- [22] Adam Doupé, Marko Cova, and Giovanni Vigna. Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In *Proc. of the 7th Int. Conf. on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)*, pages 111–131, Bonn, Germany, July 2010. Springer.

- [23] Adam Doupé, Weidong Cui, Mariusz H. Jakubowski, Marcus Peinado, Christopher Kruegel, and Giovanni Vigna. deDacota: toward preventing server-side XSS via automatic code and data separation. In *Proc. of the 20th ACM SIGSAC Conf. on Computer and Communications Security (CCS'2013)*, pages 1205–1216, Berlin, Germany, 2013. ACM.
- [24] Fabien Duchene. *Detection of Web Vulnerabilities via Model Inference assisted Evolutionary Fuzzing*. PhD thesis, Grenoble University, 2014.
- [25] Marco Eichelberg, Thomas Aden, Jörg Riesmeier, Asuman Dogac, and Gokce B. Laleci. A survey and analysis of electronic healthcare record standards. *ACM Comput. Surv.*, 37(4):277–315, December 2005.
- [26] Patrick Engebretson. *The basics of hacking and penetration testing: ethical hacking and penetration testing made easy*. Elsevier, 2013.
- [27] Michael Felderer, Berthold Agreiter, Philipp Zech, and Ruth Breu. A classification for model-based security testing. *Advances in System Testing and Validation Lifecycle (VALID 2011)*, pages 109–114, 2011.
- [28] Michael Felderer, Philipp Zech, Ruth Breu, Matthias Büchler, and Alexander Pretschner. Model-based security testing: a taxonomy and systematic classification. *Software Testing, Verification and Reliability (STVR)*, 2015.
- [29] Viktoria Felmetzger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *USENIX Security Symposium*, pages 143–160, 2010.
- [30] Matthew Finifter and David Wagner. Exploring the relationship between web application development tools and security. In *Proc. of the 2nd USENIX Conf. on Web Application Development (WebApps'11)*, pages 99–111, Portland, OR, USA, June 2011. USENIX Association.
- [31] Elizabeth Fong and Vadim Okun. Web application scanners: definitions and functions. In *Proc. of the 40th Annual Hawaii Int. Conf. on System Sciences (HICSS'07)*, page 280b, Waikoloa, HI, USA, January 2007. IEEE CS.
- [32] Elizabetha Fournoret, Martin Ochoa, Fabrice Bouquet, Julien Botella, Jan Jürjens, and Parvaneh Yousefi. Model-based security verification and testing for smart-cards. In *Proc. of the 6th Int. Conf. on Availability, Reliability and Security (ARES'11)*, pages 272–279, Vienna, Austria, August 2011. IEEE CS.
- [33] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [34] William GJ Halfond, Shauvik Roy Choudhary, and Alessandro Orso. Improving penetration testing through static and dynamic analysis. *Software Testing, Verification and Reliability (STVR)*, 21(3):195–214, 2011.
- [35] William GJ Halfond and Alessandro Orso. Improving test case generation for web applications using automated interface discovery. In *Proc. of the the 6th joint meeting of the European software engineering Conf. and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 145–154. ACM, 2007.

- [36] Object Management Group inc. Object constraint language (ocl) 2.4. <http://www.omg.org/spec/OCL/>, February 2014.
- [37] Object Management Group inc. Unified modeling language (uml) 2.5. <http://www.omg.org/spec/UML/>, June 2015.
- [38] IHE International. IHE IT Infrastructure White Paper - HIE Security and Privacy through IHE Profiles. http://www.ihe.net/Technical_Framework/upload/IHE_ITI_Whitepaper_Security_and_Privacy_of_HIE_2008-08-22-2.pdf, August 2008. Last visited: August 2015.
- [39] Jacques Julliand, Pierre-Alain Masson, and Regis Tissot. Generating security tests in addition to functional tests. In *Proc. of the 3rd Int. workshop on Automation of software test*, pages 41–44, Leipzig, Germany, May 2008. ACM.
- [40] Jan Jürjens. Model-based Security Testing Using UMLsec: A Case Study. *The Journal of Electronic Notes in Theoretical Computer Science (ENTCS)*, 220(1):93–104, December 2008.
- [41] Rauli Kaksonen and Ari Takanen. [talk] test coverage in model-based fuzz testing. In *Model Based Testing User Conference, Tallinn/Estonia*, page Invited talk. ETSI, 2012.
- [42] Samy Kamkar. The myspace worm. http://www.owasp.org/images/7/79/OWASP-WASCApSec2007SanJose_SamyWorm.ppt, 2007. Last visited: August 2015.
- [43] Adam Kiežun, Philip J. Guo, Karthick Jayaraman, and Michael D. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proc. of the 31st Int. Conf. on Software Engineering (ICSE'09)*, pages 199–209, Vancouver, Canada, May 2009. IEEE.
- [44] Engin Kirda, Nenad Jovanovic, Christopher Kruegel, and Giovanni Vigna. Client-side cross-site scripting protection. *Computers & Security*, 28(7):592–604, 2009.
- [45] Bruno Legeard and Arnaud Bouzy. Smartesting CertifyIt: Model-Based Testing for Enterprise IT. In *Proc. of the 6th Int. Conf. on Software Testing, Verification and Validation (ICST'13)*, pages 391–397, Luxembourg, March 2013. IEEE CS.
- [46] Bruno Legeard, Fabien Peureux, Martin Schneider, Fredrik Seehusen, and Alexandre Vernotte. The PMVT approach: a RASEN innovation for security pattern and model-based vulnerability testing. White paper, RASEN FP7 EU founded Research Project, April 2015. Available at <http://www.rasenproject.eu/downloads/752/> (Accessed: 2015-05-01).
- [47] Sebastian Lekies, Ben Stock, and Martin Johns. A tale of the weaknesses of current client-side xss filtering. In *Briefings of BlackHat USA 2014*, October 2014. Available at <https://www.blackhat.com/us-14/archives.html> [Last visited: August 2015].
- [48] Ponemon Institute LLC. The sql injection threat study. <http://www.dbnetworks.com/pdf/ponemon-the-SQL-injection-threat-study.pdf>, April 2014. Last visited: August 2015.
- [49] Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen. *Model-Driven Risk Analysis: The CORAS Approach*. Springer Publishing Company, Incorporated, 1st edition, 2010.

- [50] R. P. Mahapatra, Ruchika Saini, and Neha Saini. A pattern based approach to secure web applications from XSS attacks. *Int. Journal of Computer Technology and Electronics Engineering (IJCTEE)*, 2(3), June 2012.
- [51] Kinnaird McQuade. Open source web vulnerability scanners: The cost effective choice? In *Proc. of the Conf. for Information Systems Applied Research ISSN*, volume 2167, page 1508, 2014.
- [52] MITRE. Common weakness enumeration. <http://cwe.mitre.org/>, October 2013. Last visited: August 2015.
- [53] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In *Model Driven Engineering Languages and Systems*, pages 537–552. Springer, 2008.
- [54] Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proc. of the Network and Distributed System Security Symposium (NDSS'07)*, pages 1–12, San Diego, CA, USA, February 2007. The Internet Society.
- [55] Chiem Trieu Phong and Wei Qi Yan. An overview of penetration testing. *Int. Journal of Digital Crime and Forensics (IJDCF)*, 6(4):50–74, 2014.
- [56] Marco Rocchetto, Martín Ochoa, and Mohammad Torabi Dashti. Model-based detection of csrf. In *ICT Systems Security and Privacy Protection*, pages 30–43. Springer, 2014.
- [57] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *Journal on Selected Areas in Communications archive*, 21(1):5–19, September 2006.
- [58] Sébastien Salva and Stassia R Zafimiharisoa. Data vulnerability detection by security testing for android applications. In *Information Security for South Africa, 2013*, pages 1–8. IEEE, 2013.
- [59] Steve Schneider. *The B-Method: an introduction*, volume 200. Palgrave, 2001.
- [60] Fredrik Seehusen, Jürgen Großmann, and Samson Yoseph Esayas. The rasen method for risk-based security testing and legal compliance assessment. White paper, RASEN FP7 EU founded Research Project, April 2015. Available at <http://www.rasenproject.eu/downloads/748/>.
- [61] Lwin Khin Shar and Hee Beng Kuan Tan. Automated removal of cross site scripting vulnerabilities in web applications. *Information and Software Technology*, 54(5):467–478, May 2012.
- [62] Ben Smith and Laurie Williams. On the Effective Use of Security Test Patterns. In *Proc. of the 6th Int. Conf. on Software Security and Reliability (SERE'12)*, pages 108–117, Washington, DC, USA, June 2012. IEEE CS.
- [63] Benjamin H Smith. *Empirically Developing a Software Security Test Pattern Catalog Using a Grounded Theory Approach*. PhD thesis, North Carolina State University, 2012.

- [64] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.
- [65] Verizon Enterprise ®. 2015 data breach investigations report. Technical report, 2015. http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigation-report-2015_en_xg.pdf Last visited: August 2015.
- [66] Herbert H. Thompson. Application penetration testing. *IEEE Security & Privacy*, 3(1):66–69, 2005.
- [67] Gu Tian-yang, Shi Yin-sheng, and Fang You-yuan. Research on software security testing. *World Academy of science, engineering and Technology*, 70:647–651, 2010.
- [68] Mark Utting and Bruno Legeard. *Practical Model-Based Testing - A tools approach*. Morgan Kaufmann, San Francisco, CA, USA, 2006.
- [69] Johannes Viehmann. The racomat tool – risk assessment combined with automated testing. White paper, RASEN FP7 EU founded Research Project, April 2015. Available at <http://www.rasenproject.eu/downloads/750/>.
- [70] A.-G. Vouffo Feudjio. 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: August 2015].
- [71] Jiajie Wang, Tao Guo, Puhan Zhang, and Qixue Xiao. A model-based behavioral fuzzing approach for network service. In *Proc. of the 3rd Int. Conf. on Instrumentation, Measurement, Computer, Communication and Control (IMCCC'13)*, pages 1129–1134. IEEE, 2013.
- [72] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *Proc. of the 30th Int. Conf. on Software Engineering (ICSE'08)*, pages 171–180, Leipzig, Germany, May 2008. IEEE.
- [73] Tian Wei, Yang Ju-Feng, Xu Jing, and Si Guan-Nan. Attack model based penetration test for sql injection vulnerability. In *IEEE 36th Annual Computer Software and Applications Conf. Workshops (COMPSACW'12)*, pages 589–594. IEEE, 2012.
- [74] Frank Werner. RASEN Deliverable D2.1.1 - Use Case Scenarios Definition. <http://www.rasenproject.eu/downloads/723/>, October 2013. Last visited: March 2015.
- [75] Whitehat. Website security statistics report. <http://info.whitehatsec.com/rs/whitehatsecurity/images/statsreport2014-20140410.pdf>, October 2014. Last visited: August 2015.
- [76] Dave Wichers. Owasp top 10. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project, October 2013. Last visited: August 2015.
- [77] Peter Wurzinger, Christian Platzter, Christian Ludl, Engin Kirda, and Christopher Kruegel. SWAP: mitigating XSS attacks using a reverse proxy. In *Proc. of the 5th Int. Workshop on Software Engineering for Secure Systems (SESS'09)*, pages 33–39, Vancouver, Canada, May 2009. IEEE.
- [78] Dianxiang Xu, Manghui Tu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Weifeng Xu. Automated security test generation with formal threat models. *IEEE Transactions on Dependable and Secure Computing*, 9(4):526–540, 2012.

A

APPENDIX

```

test_purpose ::= ( quantifier_list 1 )? seq
quantifier_list ::= quantifier ( 1 quantifier ) *
quantifier ::= for each behavior var from behavior_choice
| for each operation var from operation_choice
| for each literal var from literal_choice
| for each instance var from instance_choice
| for each integer var from integer_choice
| for each call var from call_choice
operation_choice ::= any operation
| operation_list
| any operation but operation_list
call_choice ::= call_list
behavior_choice ::= any behavior to cover
| behavior_list
| any behavior but behavior_list
literal_choice ::= <identifier> (or <identifier>)*
instance_choice ::= instance (or instance)*
integer_choice ::= { <number> (, <number>)* }
var ::= $ <identifier>
state ::= ocl_constraint on instance instance
ocl_constraint ::= <string>
instance ::= <identifier>
seq ::= bloc (then bloc)*
bloc ::= use control restriction? target?
restriction ::= at least once
| any number of time
| <number> times
| var times
target ::= to_reach state
| to activate behavior
| to activate var
control ::= operation_choice
| behavior_choice
| var
| call_choice
call_list ::= call (or call)*
call ::= instance_operation(parameter_list)
operation_list ::= operation (or operation)*
operation ::= <identifier>
parameter_list ::= (parameter (, parameter)*)?
parameter ::= free_value
| <identifier>
| <number>
| var
behavior_list ::= behavior (or behavior)*
behavior ::= behavior with tag tag_list
| behavior without tag tag_list
tag_list ::= { tag (, tag)* }
tag ::= @REQ: <identifier>
| @AIM: <identifier>

```

Figure A.1 – Syntax of the Test Purpose Language

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	<ol style="list-style-type: none"> 1. Identify a sensible user input; 2. If there are other inputs (in the case of a Web form), supply nominal value to each input; 3. Inject an attack vector (for instance <code><script>alert(xss)</script></code>) into the sensible user input.
Observation	<ol style="list-style-type: none"> 1. Go to a page echoing the user input; 2. Check if the vector has been rendered verbatim.
Oracle	
Variant(s)	<ul style="list-style-type: none"> - 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)	XSS Cheat Sheet Web Application Firewalls (WAF) filter messages sent 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)

Figure A.2 – Test Pattern for multistep XSS attacks

Name	Error-Based SQL Injection
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)	Detect if a user input can embed attack vector enabling SQL Injections.
Prerequisites	N/A
Procedure	Based on attack pattern CAPEC-66 <ol style="list-style-type: none"> 1. Use the application, client or web browser to inject SQL constructs input (e.g., ' or ") through text fields or through HTTP GET parameters, to corrupt the syntactic correctness of the SQL query. 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-freeform fields, etc.
Observation	Scrap the resulting page for a DBMS syntax error message.
Oracle	
Variant(s)	SQL Injection Cheat Sheet
Known Issue(s)	Web Application Firewalls (WAF) filter messages sent to the server (black list, clac regEx, ...); variants allows to overcome these filters
Affiliated vTP	Time-Based SQL Injections, Boolean-Based SQL Injections
Reference(s)	CAPEC: http://capec.mitre.org/data/definitions/66.html WASC: http://projects.webappsec.org/w/page/13246963/SQL%20Injection OWASP: https://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OTG-INPVAL-005%29#Error_based_Exploitation_technique

Figure A.3 – Test Pattern of Error-Based SQL Injection attacks

Name	Time-Based SQL Injection
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)	Detect if a user input can embed attack vector enabling SQL Injections.
Prerequisites	N/A
Procedure	Based on OWASP Testing guide v4: <ol style="list-style-type: none"> 1. Use the application, client or web browser to inject an SQL fragment input through text fields or through HTTP GET parameters, with the objective to corrupt the query and get a fast response from the DBMS 2. Use the application, client or web browser to inject a second SQL fragment input through text fields or through HTTP GET parameters, with the objective to delay the response from the DBMS as much as possible (e.g., using built-in functions such as MySQL sleep())
Observation	Compare the two response times. A notable difference between response times is a strong indicator of the presence of SQL Injections
Oracle	
Variant(s)	SQL Injection Cheat Sheet
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	Error-Based SQL Injections, Boolean-Based SQL Injections
Reference(s)	CAPEC: http://capec.mitre.org/data/definitions/7.html WASC: http://projects.webappsec.org/w/page/13246963/SQL%20Injection OWASP: https://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OTG-INPVAL-005%29#Time_delay_Exploitation_technique

Figure A.4 – Test Pattern for Time-Based SQL Injection attacks

Name	Boolean-Based SQL Injection
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)	Detect if a user input can embed attack vector enabling SQL Injections.
Prerequisites	N/A
Procedure	Based on IBM AppScan Test Pattern: <ol style="list-style-type: none"> i - CTRL: Use the application, client or web browser to trigger the function containing the user input as intended. ii - &TRUE: Use the application, client or web browser to inject SQL constructs input with a positive logic (e.g., AND 1=1). iii - &FALSE: Use the application, client or web browser to inject SQL constructs input with a negative logic (e.g., AND 1=2). iv - FALSE: Use the application, client or web browser to inject SQL constructs input that has no logic influence (e.g., OR 1=2).
Observation	Compare the control points of the resulting pages. If CTRL ≈ &TRUE ≈ FALSE and &TRUE ≠ &FALSE then the application is vulnerable.
Oracle	
Variant(s)	SQL Injection Cheat Sheet
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	Time-Based SQL Injections, Error-Based SQL Injections
Reference(s)	CAPEC: https://capec.mitre.org/data/definitions/7.html WASC: http://projects.webappsec.org/w/page/13246963/SQL%20Injection OWASP: https://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OTG-INPVAL-005%29#Boolean_Exploitation_Technique IBM: http://www-01.ibm.com/support/docview.wss?uid=swg21659226

Figure A.5 – Test Pattern for Boolean-Based SQL Injection attacks

Name	Cross-Site Request Forgery
Description	The web application does not, or cannot, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.
Objective(s)	Detect if requests can be sent from outside the application.
Prerequisites	N/A
Procedure	<ol style="list-style-type: none"> 1. Authenticate as a potential victim on the application. 2. Use the application, client or web browser to trigger the tested function as intended, and store the resulting page. 3. Logout, and re-authenticate as a potential victim on the application. 4. Make a request to an external application. 5. From the external application, send a request that triggers the function on the application under test, and store the resulting page.
Observation	Compare the control-points of the two resulting pages. If the two sets of control points are equal, the tested function is considered vulnerable to CSRF.
Oracle	
Variant(s)	
Known Issue(s)	
Affiliated vTP	
Reference(s)	CAPEC: http://capec.mitre.org/data/definitions/62.html WASC: http://projects.webappsec.org/w/page/13246919/Cross%20Site%20Request%20Forgery OWASP: https://www.owasp.org/index.php/Testing_for_CSRF_%28OTG-SESS-005%29

Figure A.6 – Test Pattern for Cross-Site Request Forgery attacks

Name	Privilege Escalation
Description	The software does not perform a rigorous authorization check for functionalities and pages that requires a provable user identity.
Objective(s)	Detect if a page or function can be requested without proper authorization (e.g., without being authenticated, or authenticated with low-privilege credentials).
Prerequisites	N/A
Procedure	Based on attack pattern from OWASP Testing Guide v4: <ol style="list-style-type: none"> 1. Authenticate as a potential victim on the application. 2. Use the application, client or web browser to access the tested page / trigger the tested function as intended, and store the resulting page. 3. Logout and re-authenticate as a potential victim on the application. 4. Force-browse the request that retrieves the tested page / triggers the tested function, without relying on the GUI, and store the resulting page.
Observation	Compare the control points of the two resulting pages. If the two sets of control points are equal, the tested function is considered vulnerable to Privilege Escalation.
Oracle	
Variant(s)	
Known Issue(s)	
Affiliated vTP	
Reference(s)	CAPEC: http://capec.mitre.org/data/definitions/233.html OWASP: https://www.owasp.org/index.php/Testing_for_Privilege_escalation_%28OTG-AUTHZ-003%29

Figure A.7 – Test Pattern of Privilege Escalation attacks

Abstract:

This thesis proposes an original approach, dubbed PMVT for Pattern-driven and Model-based Vulnerability Testing, which aims to improve the capability for detecting four high-profile vulnerability types, Cross-Site Scripting, SQL Injections, CSRF and Privilege Escalations, and reduce false positives and false negatives verdicts. PMVT relies on the use of a behavioral model of the application, capturing its functional aspects, and a set of vulnerability test patterns that address vulnerabilities in a generic way.

By adapting existing MBT technologies, an integrated toolchain that supports PMVT automates the detection of the four vulnerability types in Web applications. This prototype has been experimented and evaluated on two real-life Web applications that are currently used by tens of thousands users. Experiments have highlighted the effectiveness and efficiency of PMVT and shown a strong improvement of vulnerability detection capabilities w.r.t. available automated Web application scanners for these kind of vulnerabilities.

Keywords: Vulnerability Testing, Model-Based Testing, Vulnerability Test Patterns, Web applications, Cross-Site Scripting, SQL Injections, Cross-Site Request Forgery, Privilege Escalation

Résumé :

Cette thèse propose une approche originale de test de vulnérabilité Web à partir de modèles et dirigée par des patterns de tests, nommée PMVT. Son objectif est d'améliorer la capacité de détection de quatre types de vulnérabilité majeurs, Cross-Site Scripting, Injections SQL, Cross-Site Request Forgery, et Privilege Escalation. PMVT repose sur l'utilisation d'un modèle comportemental de l'application Web, capturant ses aspects fonctionnels, et sur un ensemble de patterns de test de vulnérabilité qui adressent un type de vulnérabilité de manière générique, quelque soit le type de l'application Web sous test.

Par l'adaptation de technologies MBT existantes, nous avons développé une chaîne outillée complète automatisant la détection des quatre types de vulnérabilité. Ce prototype a été expérimenté et évalué sur deux applications réelles, actuellement utilisées par plusieurs dizaines de milliers d'utilisateurs. Les résultats d'expérimentation démontrent la pertinence et de l'efficacité de PMVT, notamment en améliorant de façon significative la capacité de détection de vulnérabilités vis à vis des scanners automatiques d'applications Web existants.

Mots-clés : Test de Vulnérabilité, Test à partir de Modèles, Patterns de Test de Vulnérabilité, Applications Web, Cross-Site Scripting, Injections SQL, Cross-Site Request Forgery, Privilege Escalation

