# A Pattern-driven and Model-Based Test Generation Toolchain for Web Vulnerability

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

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

**Abstract.** The purpose of this demonstration is to present a tooled Pattern-driven and Model-based Vulnerability Testing approach (PMVT for short) to improve the capability of detection of various vulnerability types such as injections (Cross-Site Scripting, SQL injections, etc.). This approach relies on generic vulnerability test patterns, which are applied on a behavioral model of the application under test, in order to generate vulnerability test cases. Hence, we propose to demonstrate the toolchain especially regarding XSS and SQLI vulnerabilities. This prototype has been experimented and validated on real-life Web applications, showing a strong improvement of detection ability w.r.t. Web application scanners for these vulnerabilities. This work is partially supported by the FP7 European project RASEN (http://www.rasenproject.eu).

## 1 Context

Based on the current state of the art on security and on all the security reports like OWASP Top Ten 2013 CWE/SANS 25 and WhiteHat Website Security Statistic Report 2013, Web applications are the most popular targets when speaking of cyber-attacks. The mosaic of technologies used in current Web applications increases the risk of security breaches. This situation has led to significant growth in application-level vulnerabilities.

Application-level vulnerability testing is first performed by developers, but they often lack the sufficient in-depth knowledge in recent vulnerabilities and related exploits. This kind of tests can also be achieved by companies specialized in security testing, in penetration testing for instance. But they mainly use manual approaches, making the dissemination of their techniques very difficult, and the impact of this knowledge very low. Finally, Web application vulnerability scanners can be used to automate the detection of vulnerabilities, but since they often generate many false positive and false negative results, human investigation is also required [2].

The approach we propose, called Pattern-driven and Model-based Vulnerability Testing (PMVT for short) aims to improve the accuracy and precision of vulnerability testing, by proposing an automated testing approach driven by *vulnerability test patterns* composed with a *behavioral model* of the system under test (SUT). The next sections respectively introduce the principles of the PMVT approach and describe the architecture of the toolchain implementing it.

## 2 Principles of the PMVT Approach

The PMVT process, shown in Fig. 1, is composed of the four following activities.

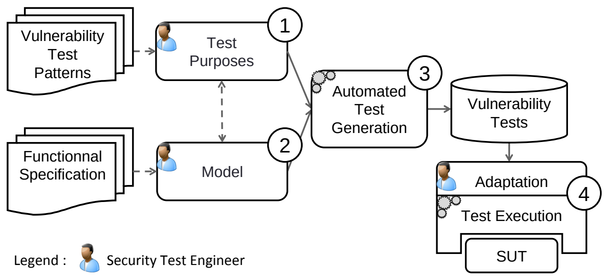

Vulnerability Test Patterns
Test Purposes
1
Automated Test Generation
3
Vulnerability Tests
Functionnal Specification
Model
2
Adaptation
4
Test Execution
SUT
Legend : Security Test Engineer

**Fig. 1.** Pattern-driven and Model-based Vulnerability Test Process

① **The *Test Purpose* design activity.** It consists of formalizing a test procedure from vulnerability test patterns (vTP) that the generated test cases have to cover. Each procedure, targeting a dedicated vulnerability, is given by a test purposes, which is a high-level expression. It formalizes a testing objective and drives the automated test generation on the behavioral model of SUT.

② **The *Modeling* activity.** As for every Model-Based Testing (MBT) approach, the modeling activity consists of designing a test model that will be used to automatically generate abstract test cases. To ease and accelerate this modeling activity, we have developed a Domain Specific Modeling Language (DSML), called *DASTML*, that allows to model the global structure of a Web application.

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

④ **The *Adaptation, Test Execution and Observation* activity.** During the modeling activity, all data used by the application are modeled at an abstract level. As a consequence, the test cases are abstract and cannot thus be executed as they are. To bridge the gap, this activity mainly consists of linking the abstract data to concretes one in order to provide executable test scripts. It should be underlined that these test scripts embed the test sequence and the observation procedure in order to automate the verdict assignment.

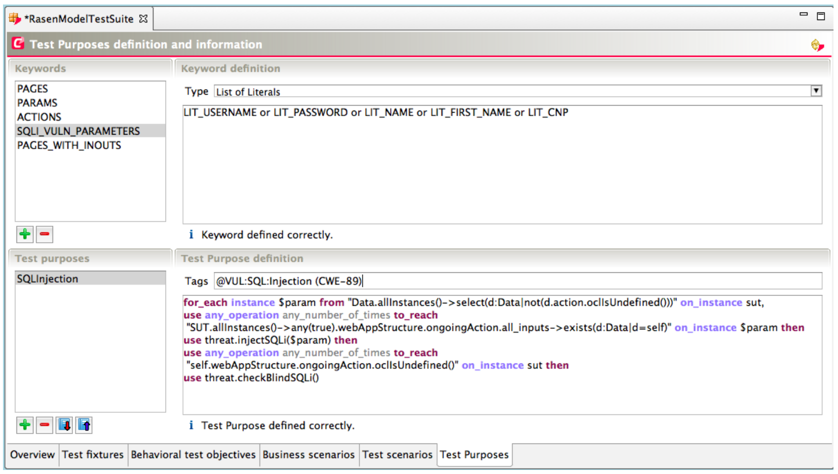
Basically, the main contribution (that we aim to illustrate and emphasize during demonstration) of this tooled approach concerns [4]:

– The formalization of vulnerability test patterns using generic test purposes to drive the test generation engine.
– The use of a DSML to ease and accelerate the functional modeling of the Web application under test.
– The full automation of the testing process, including test generation, test execution and verdict assignment.

## 3 Overview of the PMVT toolchain

All the PMVT activities are supported by a dedicated toolchain, based on an existing Model-Based Testing (MBT) software named *CertifyIt* [3] provided by the company Smartesting[3]. CertifyIt is a test generator that takes as input a test model (written with a subset of UML called UML4MBT [1]), capturing the behavior of the SUT. This model is managed using a dedicated plugin in IBM Rational Software Architect[4].

The process supported by the toolchain takes as input one or several vulnerability test purpose (vTP) and a UML4MBT model of the SUT. A test purpose formalizes a given vulnerability in relation with associated generic vulnerability test pattern catalogues. The demonstration will address Cross-Site Scripting (XSS) and SQL injections (SQLI), but the security test pattern catalogue is currently being extended to address the OWASP Top 10 weaknesses. Basically, such a test purpose is a sequence of significant *steps* that has to be exercised by the test case scenario in order to assess the robustness of the application under test w.r.t. the related vulnerability. Each step takes the form of a set of operations to execute, or specific state to be reached on the UML4MBT behavioral model. For instance, Fig. 2 shows the test purpose formalizing the vTP for SQLI.



Keywords: PAGES, PARAMS, ACTIONS, SQLI_VULN_PARAMETERS, PAGES_WITH_INOUTS

Type: List of Literals

```
LIT_USERNAME or LIT_PASSWORD or LIT_NAME or LIT_FIRST_NAME or LIT_CNP
```

Keyword defined correctly.

Test purposes: SQLInjection

Tags: @VUL:SQL:Injection (CWE-89)

```
for_each instance $param from "Data.allInstances()->select(d:Data|not(d.action.oclIsUndefined()))" on_instance sut,
use any_operation any_number_of_times to_reach
 "SUT.allInstances()->any(true).webAppStructure.ongoingAction.all_inputs->exists(d:Data|d=self)" on_instance $param then
use threat.injectSQLi($param) then
use any_operation any_number_of_times to_reach
 "self.webAppStructure.ongoingAction.oclIsUndefined()" on_instance sut then
use threat.checkBlindSQLi()
```

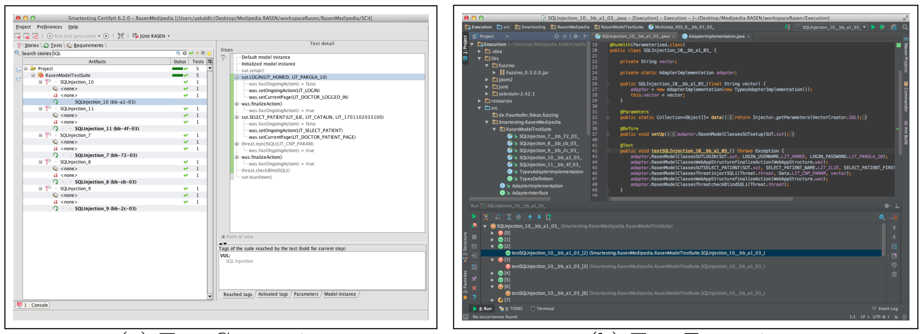
Test Purpose defined correctly.

**Fig. 2.** SQLI Vulnerability Test Purpose

During test generation, this test purpose drives the vulnerability test generation on the UML4MBT behavioral model. This model is designed using a dedicated DSML (called DASTML) that allows to describe all the structural entities and behaviors, of the Web application under test, which are needed to generate vulnerability test cases: pages, available actions on each page, etc.

[3] `http://www.smartesting.com` [October 2014]
[4] `http://www-03.ibm.com/software/products/en/ratisoftarch` [October 2014]

At this stage, the test generator automatically combines model and test purposes to compute abstract test cases, as shown in Fig. 3(a), regarding SQLI. To automate test execution, the test engineer has to match each abstract value with a concrete value. This consists of automatically creating a JUnit test suite, in which each abstract test case is exported as a JUnit test case. The test case generator also produces a file containing the signature of each operation that the test engineer has to implement using a Web navigation library such as HtmlUnit or Selenium. Once this task is completed, test cases can be executed and verdict is assigned automatically, as shown in Fig 3(b).


Smartesting CertifyIt 6.2.0 – RasenMedipedia [/Users/yakaldir/Desktop/Medipedia RASEN/workspaceRasen/RasenMedipedia/SCit]
Project
Preferences
Help
Run test generation
JUnit RASEN
Stories
Tests
Requirements
Search stories SQL
Test detail
Steps
Point of view
Reached tags
Activated tags
Parameters
Model instance
1 : Console
SQLInjection_10__bb_a1_03_.java – [Execution] – Execution – [~/Desktop/Medipedia RASEN/workspaceRasen/Execution]
Project
SQLInjection_10__bb_a1_03_.java
AdapterImplementation.java
4: Run
6: TODO
Terminal
Event Log
No occurrences found

| Artifacts | Status | Tests |
|---|---|---|
| Project | ✓ | 5 |
| RasenModelTestSuite | ✓ | 5 |
| SQLInjection_10 | ✓ | 1 |
| <none> | ✓ | 1 |
| <none> | ✓ | 1 |
| SQLInjection_10 (bb-a1-03) | | |
| SQLInjection_11 | ✓ | 1 |
| <none> | ✓ | 1 |
| <none> | ✓ | 1 |
| SQLInjection_11 (bb-4f-03) | | |
| SQLInjection_7 | ✓ | 1 |
| <none> | ✓ | 1 |
| <none> | ✓ | 1 |
| SQLInjection_7 (bb-72-03) | | |
| SQLInjection_8 | ✓ | 1 |
| <none> | ✓ | 1 |
| <none> | ✓ | 1 |
| SQLInjection_8 (bb-cb-03) | | |
| SQLInjection_9 | ✓ | 1 |
| <none> | ✓ | 1 |
| <none> | ✓ | 1 |
| SQLInjection_9 (bb-2c-03) | | |

```
Default model instance
Initialized model instance
sut.setup()
sut.LOGIN(LIT_HOMED, LIT_PAROLA_10)
    was.hasOngoingAction() = false
    was.setOngoingAction(LIT_LOGIN)
    was.setCurrentPage(LIT_DOCTOR_LOGGED_IN)
was.finalizeAction()
    was.hasOngoingAction() = true
sut.SELECT_PATIENT(LIT_ILIE, LIT_CATALIN, LIT_1701102033100)
    was.hasOngoingAction() = false
    was.setOngoingAction(LIT_SELECT_PATIENT)
    was.setCurrentPage(LIT_DOCTOR_PATIENT_PAGE)
threat.injectSQLi(LIT_CNP_PARAM)
    was.hasOngoingAction() = true
was.finalizeAction()
    was.hasOngoingAction() = true
threat.checkBlindSQLi()
sut.teardown()
```

Tags of the suite reached by the test (bold for current step)

**VUL:**
SQL:Injection

```
Execution (~/Desktop/Medipedia RASEN/work
  .idea
  libs
    fuzzino
      Fuzzino_0.3.0.0.jar
    jdom2
    junit
    selenium-2.42.1
  resources
  src
    de.fraunhofer.fokus.fuzzing
    Smartesting.RasenMedipedia
      RasenModelTestSuite
        SQLInjection_7__bb_72_03_
        SQLInjection_8__bb_cb_03_
        SQLInjection_9__bb_2c_03_
        SQLInjection_10__bb_a1_03_
        SQLInjection_11__bb_4f_03_
        TypesAdapterImplementation
        TypesDefinition
      AdapterImplementation
      AdapterInterface
```

```java
@RunWith(Parameterized.class)
public class SQLInjection_10__bb_a1_03_ {

    private String vector;

    private static AdapterImplementation adapter;

    public SQLInjection_10__bb_a1_03_(final String vector) {
        adapter = new AdapterImplementation(new TypesAdapterImplementation());
        this.vector = vector;
    }

    @Parameters
    public static Collection<Object[]> data() { return Injector.getParameters(VectorCreator.SQL); }

    @Before
    public void setUp() { adapter.RasenModelClassesSUTsetup(SUT.sut); }

    @Test
    public void testSQLInjection_10__bb_a1_03_() throws Exception {
        adapter.RasenModelClassesSUTLOGIN(SUT.sut, LOGIN_USERNAME.LIT_HOMED, LOGIN_PASSWORD.LIT_PAROLA_10);
        adapter.RasenModelClassesWebAppStructurefinalizeAction(WebAppStructure.was);
        adapter.RasenModelClassesSUTSELECT_PATIENT(SUT.sut, SELECT_PATIENT_NAME.LIT_ILIE, SELECT_PATIENT_FIRST
        adapter.RasenModelClassesThreatinjectSQLi(Threat.threat, Data.LIT_CNP_PARAM, vector);
        adapter.RasenModelClassesWebAppStructurefinalizeAction(WebAppStructure.was);
        adapter.RasenModelClassesThreatcheckBlindSQLi(Threat.threat);
    }
```

```
Run SQLInjection_10__bb_a1_03_
SQLInjection_10__bb_a1_03_ (Smartesting.RasenMedipedia.RasenModelTestSuite)
  [0]
  [1]
  [2]
    testSQLInjection_10__bb_a1_03_[2] (Smartesting.RasenMedipedia.RasenModelTestSuite.SQLInjection_10__bb_a1_03_)
  [3]
    testSQLInjection_10__bb_a1_03_[3] (Smartesting.RasenMedipedia.RasenModelTestSuite.SQLInjection_10__bb_a1_03_)
  [4]
  [5]
  [6]
    testSQLInjection_10__bb_a1_03_[6] (Smartesting.RasenMedipedia.RasenModelTestSuite.SQLInjection_10__bb_a1_03_)
  [7]
```

(a) Test Generation (b) Test Execution

**Fig. 3.** Test Case Generation and Execution using PMVT Approach

In order to illustrate the PMVT approach during Demonstration Session, we will use the toy example WackoPicko[5]. This deliberately-unsecured Web application allows users to authenticate themselves, share comment and buy pictures. To keep it simple, we will focus on Cross-Site Scripting vulnerability: a malicious user injects a script and forces victims visiting the page to execute them.

## References

1. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F.: A test generation solution to automate software testing. In: Proc. of the 3rd Int. Workshop on Automation of Software Test (AST'08). pp. 45–48. ACM Press, Leipzig, Germany (May 2008)
2. Doupé, A., Cova, M., Vigna, G.: Why Johnny can't pentest: an analysis of black-box web vulnerability scanners. In: Proc. of the 7th Int. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10). pp. 111–131. Springer, Bonn, Germany (July 2010)
3. Legeard, B., Bouzy, A.: Smartesting CertifyIt: Model-Based Testing for Enterprise IT. In: Proc. of the 6th Int. Conference on Software Testing, Verification and Validation (ICST'13). pp. 391–397. IEEE CS, Luxembourg (March 2013)
4. Vernotte, A., Dadeau, F., Lebeau, F., Legeard, B., Peureux, F., Piat, F.: Efficient detection of multi-step cross-site scripting vulnerabilities. In: Proceedings of the 10th International Conference on Information Systems Security (ICISS'14). LNCS, vol. 8880, pp. 359–378. Springer, Hyderabad, India (Dec 2014)

[5] `https://github.com/adamdoupe/WackoPicko` [October 2014]