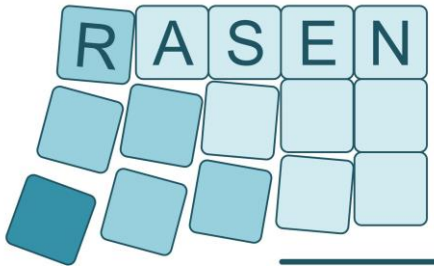


Compositional Risk Assessment and Security Testing of Networked Systems

Deliverable D4.2.3

Techniques for Compositional Risk-Based Security Testing v.3

Project title:	RASEN
Project number:	316853
Call identifier:	FP7-ICT-2011-8
Objective:	ICT-8-1.4 Trustworthy ICT
Funding scheme:	STREP – Small or medium scale focused research project

Work package:	WP4
Deliverable number:	D4.2.3
Nature of deliverable:	Report
Dissemination level:	PU
Internal version number:	1.0
Contractual delivery date:	2015-09-30
Actual delivery date:	2015-09-30
Responsible partner:	Fraunhofer

Contributors

Editor	Jürgen Großmann (FOKUS)
Contributors	Fredrik Seehusen (SINTEF), Fabien Peureux (UFC), Alexandre Vernotte (UFC), Martin Schneider (FOKUS), Johannes Viehmann (FOKUS)
Quality assurors	Frank Werner (SAG), Samson Yoseph Esayas (UiO)

Version history

Version	Date	Description
0.1	15-06-06	ToC proposition
0.2	15-06-06	Initial content Fraunhofer FOKUS
0.3	15-09-09	Input Testing Dashboard
0.4	15-09-17	API and implementation guidelines of Dashboard
0.5	15-09-17	Test purpose chapter finalized – overall review
1.0	15-09-25	Final version

Abstract

Work package 4 has developed a framework for security testing guided by risk assessment. This framework, starting from security test patterns and test generation models, allows for a compositional security testing approach that is able to deal with large-scale networked systems. This deliverable is the final part of a series of three deliverables (D4.2.1, D4.2.2, D4.2.3) that document how the RASEN approach for risk-based security testing has been evolved through continuous and iterative updates. It provides the final update for the RASEN approach of formalizing test patterns using the Test Purpose Language, and it introduces the RASEN Testing Dash Board for Test Result Aggregation.

Keywords

Security testing, risk-based security testing, Test Purpose Language, fuzzing on security models, security testing metrics, large-scale networked systems, test selection, test prioritization

Executive Summary

The overall objective of RASEN WP4 is to develop techniques for the use of risk assessment as guidance and basis for security testing, and to develop an approach that supports a systematic aggregation of security testing results by means of security testing metrics. This comprises the development of a tool-based integrated process for guiding security testing by means of reasonable risk coverage and probability metrics. This deliverable is the third and final part of a series of three deliverables that define the overall RASEN approach for risk-based security testing. The earlier deliverables have introduced approaches for risk-based test identification and selection, the notion of test pattern, new fuzz testing techniques and the RASEN approach for pattern-driven and model-based vulnerability testing (PMVT). This deliverables updates the PMVT approach by showing the formalization and operationalization of test patterns using the Test Purpose Language. Moreover, it introduces metrics that classify test results at the testing level and show their implementation by the RASEN Testing Dashboard. The RASEN Testing Dashboard allows for a concise visualization of metric results.

Table of contents

1 Introduction

The overall objective of RASEN WP4 is to develop techniques for risk-based security testing. Risk assessment is used to provide guidance and yield as basis for security testing and to develop an approach that supports a systematic aggregation of security testing results. The objective includes the development of a tool-based integrated process for guiding security testing by means of reasonable risk coverage and probability metrics.

This deliverable is the third and final deliverable in a series of three deliverables that presents techniques to address compositional security testing guided by risk assessment. Risk assessment is used to provide a systematic guidance for planning, structuring and organizing the security testing process. The overall RASEN approach for risk-based security testing, that defines the Innovation 3 of the project, has been described and detailed in the previous deliverable of this series [2]. The process is recalled in Figure 1.

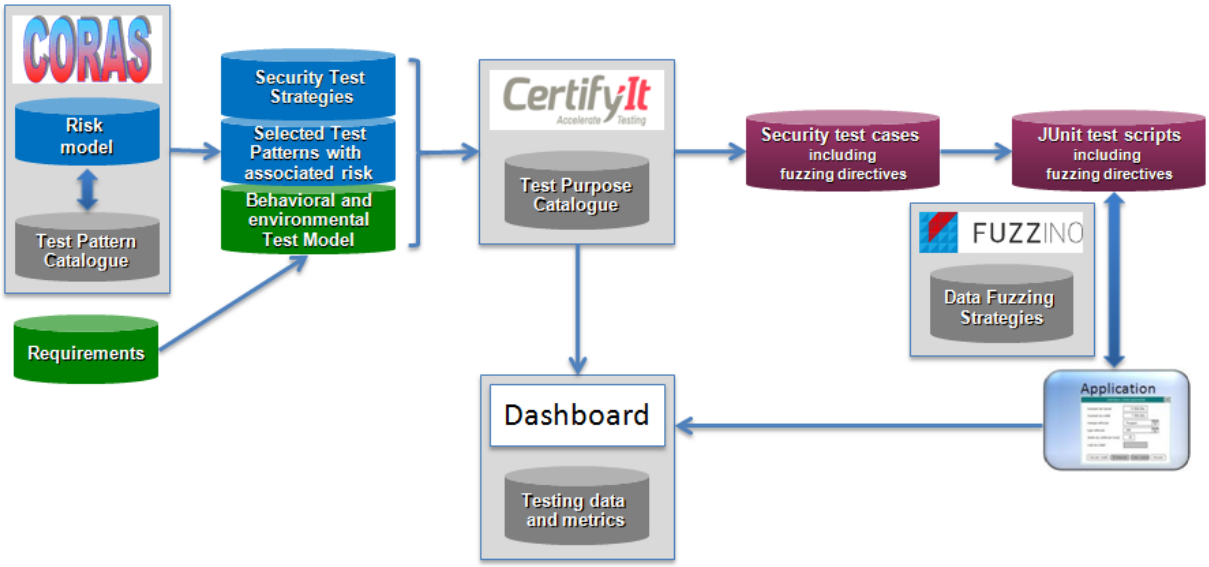


Figure 1 – Overall process of security testing based on risk assessment results

This process starts with a risk model, a result obtained from the risk assessment that is created by using the *CORAS* method from SINTEF. This risk model allows identifying potential threat scenarios and vulnerabilities, and is used for the identification and prioritization of appropriate security test patterns. Based on the selected security test patterns, test cases are generated by combining information from the risk model, a test model and test generation techniques. The latter are composed of test purposes (formalizing the security test patterns) developed by UFC for Smartesting *CertifyIt* and fuzzing techniques implemented by the Fraunhofer FOKUS's fuzzing library *Fuzzino*. Finally, test scripts are generated, compiled and executed against the application under test, and related test results are gathered and displayed in a Dashboard that provides various security testing metrics.

This deliverable focuses on results from the task dealing with automating test execution based on risk assessment in a compositional way and the task to develop metrics and a dashboard for security testing results based on risk assessment. In this way, Section 2 provides an update for the RASEN approach of formalizing test pattern using the Test Purpose Language. Section 3 shows metrics that classify test results at the testing level and their implementation by the RASEN Testing Dashboard that allows for a concise visualization of test results and test metric results.

2 Formalizing Test Patterns with Test Purpose Language

Security test patterns, based on prioritized vulnerabilities from the *CORAS* model, provide a starting point for security test case generation by giving information on how appropriate security test cases can be created from risk analysis results. The security test patterns express in a textual manner the testing procedures to detect Web application threats. This way, they propose solutions to improve testing automation (data vector libraries, test metrics to complete test coverage criteria, etc.). Therefore they are imported from the risk model elements and then formalized to drive and automate the test generation process. To enable automation, the test generation tool *CertifyIt* proposes a catalogue of generic test purposes that formalize test patterns. To summarize, a test purpose formalizes the intention of a given security test pattern and thus allows to automatically generate the expected test cases with model-based testing techniques.

A test purpose is a high-level expression that formalizes a testing objective to drive the automated test generation on the test model. As introduced in deliverable D4.2.1 [1], such a test purpose can be seen as a partial algorithm (some steps may be not explicit) defining a sequence of significant steps that has to be executed by the test case scenario. Each step takes the form of a set of operations or behaviors to be covered, or specific state to be reached on the test model in order to assess the robustness of the application under test with respect to the related vulnerability to be tested.

A typical test purpose is composed of two main entities: iterators and stages.

- Stages define execution steps (in terms of states to be reached and operations to be executed) 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 construction introduced inListing 1.

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

Listing 1 – Test purpose construction

A first version of the syntax and examples of practical use of the test purpose language is described in the Deliverable D4.2.1 [1] and its grammar is recalled in Figure 2. However, to make generic test purposes and to formalize complex and sophisticated attacks (required to conduct the RASEN case-studies and thus to validate the proposed approach), this initial version has been extended. The next subsections respectively detail these additions and introduce the generic test purposes that formalize the four vulnerabilities identified during risk assessment of the RASEN case-studies and targeted by test generation (namely Cross-Site Scripting, SQL Injections, CSRF, and Privilege Escalation).

2.1 Extension of the Test Purpose Language

Within RASEN vulnerability testing approach, a test purpose formalizes the expression of the essence of a well-understood solution to a recurring software vulnerability testing problem and how it can be solved. To reach this goal, a test purpose captures in a generic way the test pattern part that concerns the test intention with one or several operational test purposes, in order to automatically produce the corresponding test cases with model-based testing.

Such a test purpose aims to be generic (meaning that it can be applied without update whatever the test model is) in order to be applied on several models to generate test sequences. However, current test purposes contain information coming directly from the current test model, which makes them reliant on it. To avoid any dependence, several additions were made to the test purpose language to allow and improve their genericity.

Namely, these contributions to the language are the following:

- Creation of the lists of keywords, referring to model entities, to externalize the use of data;
- 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, 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 so that one or several stages can be activated more than once;
- Creation of a test purpose catalogue that allows automatic import/export of existing test purposes from one testing project to another.

```
test_purpose      ::=  (quantifier_list, )?seq
quantifier_list   ::=  quantifier(,quantifier)*
quantifier        ::=  for_each_behaviorvarfrombehavior_choicefor_
                  |    each_operationvarfromoperation_choice
                  |    for_each_literalvarfromliteral_choice
                  |    for_each_instancevarfrominstance_choicefo
                  |    r_each_integervarfrominteger_choicefor_ea
                  |    ch_callvarfromcall_choiceany_operation
operation_choice  ::=
                  |    operation_list
                  |    any_operation_butoperation_list
call_choice       ::=  call_list
behavior_choice   ::=  any_behavior_to_cover
                  |    behavior_list
                  |    any_behavior_butbehavior_list
literal_choice    ::=  <identifier>(or<identifier>)*
instance_choice   ::=  instance(orinstance)*
integer_choice    ::=  {<number>(,<number>)*}
var               ::=  $<identifier>
state             ::=  ocl_constrainton_instanceinstance
ocl_constraint    ::=  <string>
instance          ::=  <identifier>
seq               ::=  bloc(thenbloc)*
bloc              ::=  usecontrolrestriction?target?
restriction       ::=  at_least_once
                  |    any_number_of_time
                  |    <number>timesv
                  |    artimesto_reach
target            ::=  state
                  |    to_activatebehavior
                  |    to_activatevarop
control           ::=  eration_choice
                  |    behavior_choice
                  |    varcall_choi
                  |    ce
call_list         ::=  call(orcall)*
call              ::=  instance.operation(parameter_list)
operation_list    ::=  operation(oroperation)*
operation         ::=  <identifier>
parameter_list    ::=  (parameter(,parameter)*)?
parameter         ::=  free_value
                  |    <identifier>
                  |    <number>
                  |    var
behavior_list     ::=  behavior(orbehavior)*
behavior          ::=  behavior_with_tagtag_list
                  |    behavior_without_tagtag_list
tag_list          ::=  {tag(,tag)*}
tag               ::=  @REQ:<identifier>
                  |    @AIM:<identifier>
```

Figure 2 – Grammar of the test purpose language

The next sections introduce each of these additions that enable a sufficient expressiveness to formalize generic test purposes targeting the four vulnerability types handled during the RASEN case studies.

2.1.1 Keyword Lists

The keywords mechanism, which has initially been introduced in [1], consists of using specific arguments, called keywords, in test purposes to represent generic artifacts of a test 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 test 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, considering an enumeration, a keyword list enables to only apply test purposes to literals of the enumeration that share the same properties or restrictions (e.g., selecting only keywords that point to user actions and excluding unnecessary actions that represent for instance search forms).

In theListing 2, the iterator `for_each` goes through all the keywords from the #KEYWORD_LIST, each keyword pointing to a certain enumeration literal.

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

Listing 2 – Literal iteration construction

As introduced in Listing 3, a test purpose stage can require 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:

```
useany_operation #RELEVANT_OPS to_reach OCL_EXPR1 on_instance$inst1
useany_operation_but #UNWANTED_OPS to_reach OCL_EXPR2 on_instance $inst2
```

Listing 3 – Operation call construction

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`.

2.1.2 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. However, when the keywords from a list can be deduced based on the information from the model, it is thus possible to extract their corresponding element automatically. Hence the language has been extended to iterate the results of an OCL expression. It is constructed as shown in Listing 4.

```
for_each  instance $instfrom  "self.all_users->select(u:User|u.att1= 2)" on_instance User1
```

Listing 4 – OCL result iteration construction

First the OCL expression is evaluated, in the context of the `User1` instance. The expression returns all User instances such that `att1` is equal to 2. Then, the results are transmitted to the iterator to be used in the stage phase. This construction preserves the generic features of test purposes and automates the test data selection to be used for test generation.

2.1.3 Variable Usage in Nested "for_each"Loops

Certain types of attack require to consider several data types 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). To meet this need, variable usage between `for_each` loops has been implemented. In case 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 as described in Listing 5.

```
for_eachinstance $inst1from#INST_LIST
for_eachinstance $inst2from"self.all_items" on_instance $inst1
```

Listing 5 – Variable usage in nested iteration construction

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.

2.1.4 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, it needs 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. As introduced in Listing 6, 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 $instfrom  "self.all_users->select(u:User|u.att1= 2)" on_instance User1
```

Listing 6 – Variable usage in OCL expressions within nested iteration

Moreover, variables can also be used in OCL expressions from the restriction part of stages as shown in Listing 7

```
useany_operationto_reach  "self.status = STATUS::$lit" on_instance SUT
```

Listing 7 – Variable usage in OCL expressions within stages

This stage expresses that any operation from the model can be used, with the goal that the status attribute from the system under test is valuated with the content of `$lit`, which contains an enumeration literal from the enumeration `STATUS`.

2.1.5 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 especially for time-based and Boolean-based SQL injections, which require the injection of 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, the notion of stage loops has been introduced in the test purpose language. As introduced in Listing 8, 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 …
thenuse...
end_repeat
```

Listing 8 – Stage loop construction

In this sequence, the stages "`use`..." enclosed in the loop must be repeated three times.

2.1.6 Test Purpose Catalog

Test purposes are stored in a test purpose catalogue (in XML format), with a reference to the pattern it belongs to. Within the RASEN project, test purpose selection is directly conducted based on a risk assessment: regarding the information present in the CORAS model, the corresponding test purposes are chosen for test generation. It should be noted that test engineers can also manually select relevant test purposes to be applied, depending on the test objective or motivated by a test selection criteria.

2.2 Vulnerability Test Purposes

This section introduces the generic test purposes designed during the RASEN project to tackle the four vulnerabilities targeted during the conducted case studies (Cross-Site Scripting, SQL Injections, CSRF, and Privilege Escalation). Some vulnerability 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 test purpose used to design it, and describe next its functionality by going through each of its steps.

2.2.1 Cross-Site Scripting

Cross-Site Scripting vulnerability (XSS for short) consists of an attacker injecting a hostile browser executable code (e.g., 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 due to the lack of proper user-supplied input data analysis from the web application under test.

As stated in the XSS test pattern, it is possible to perform XSS attack by applying the following testing strategy that is composed of three steps: (i) locate a user-supplied input, (ii) inject an XSS vector, and (iii) analyze the server response. However, to tackle all XSS types at once, the XSS test purpose makes use of the structure information that is specified in the model: links between user-supplied inputs and the pages of the Web application under test that use them to compute an output. Thereby, for the testing of a particular user input, the test purpose for XSS proceeds as follows:

1. **Locate the user input**: Following proper user interactions, the Web application 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.
2. **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 a specific data type) to block the submission of the form/request.
3. **Replace input with attack vector**: Here, the initial nominal content of the user input under test is erased and an attack vector is injected instead.
4. **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.
5. **Locate an output point**: Instead of simply waiting for the next server response, the test model is used to determine which page uses the user input under test to compute its output, and the Web application state is changed such that it displays the page.

6. **Analyze the result**: The content of the page is then analyzed to assess whether the attack vector has been inserted in the page. If it has not undergone any modification, it can be concluded that the Web application is 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 Listing 9.

```
1   for_eachinstance$pagefrom
2   "self.all_pages->select(p:Page|not(p.all_outputs->isEmpty())" on_instancewas,
3   for_eachinstance$paramfrom"self.all_outputs"on_instance$page,
4   useany_operation_but#UNWANTED_OPSany_number_of_timesto_reach
5   "WebAppStructure.allInstances()->any(true).
6           ongoingAction.all_inputs->exists(d:Data|d=self))"on_instance$param
7   thenusethreat.injectXSS($param)
8   thenusewas.finalizeAction()
9   thenuseany_operation_but#UNWANTED_OPSany_number_of_timesto_reach
10  "self.was_p.current_page=selfand
11  self.was_p.ongoingAction.oclIsUndefined()" on_instance$page
12  thenusethreat.checkXSS()
```

Listing 9 – Test purpose for cross-site scripting

The first three lines of the test purpose for XSS compose the first phase. Because this is about XSS, the first `for_each` statement selects all the pages 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 SUT instance, that defines the Web application under test. This OCL expression can be split as follows: from all the pages "`Pages.allInstances()`", all the pages "`->select(p:Page|`" that are linked to one or more data instances "`not(p.all_outputs->isEmpty())`" are selected. The result of the OCL expression is a set of page instances.

Afterwards the `for_each` statement selects all the data instances linked to the page instance contained in $page, i.e. all the user inputs that $page uses to compute its output. Here, the selection is done using the OCL expression "`self.all_ouputs`" from the context of `$page`. Therefore, the second stage of the test purpose handles two elements, a user input and one of the pages that outputs it.

The second phase starts on lines 4-5-6 by putting the Web application in a state where the page displayed 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 $param`"), the test purpose tells the test generation engine to satisfy the OCL expression "`WebAppStructure.allInstances()->any(true).ongoingAction.all_inputs->exists(d:data|d=self)`". First, the instance of the Web application under test is retrieved "`WebAppStructure.allInstances()->any(true)`". Second, we navigate in the model until the ongoing action "`ongoingAction`" is reached. Third, one check that the user input `$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 by executing the instructions "`use any_operation_but #UNWANTED_OPS any_number_of_times`", which means that any behavioral or navigational operation from the Web application can be called, as many times as needed, in order to find the right state. Indeed, each designed test purpose possesses a keyword list, named `#UNWANTED_OPS`, which contains all the operations from the `WebAppStructure` classes except those to exercise an attack. Those operations are not meant to be called during the computation of navigational and behavioral steps, therefore they are excluded to find the right state,

but they are used to complete XSS injections in line 7 by calling the operation "`threat.injectXSS($param)`", which targets the user input `$param`.

Lines there after (8 to 12) handle verdict assignment. The goal is to put the Web application such that it displays the page (`$page`) that outputs the user input and analyze its content. This is done by satisfying the OCL expression in lines 10 and 11, defined in the context of `$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 equal to `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 scans the page content to look for the injected vector.

2.2.2 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 the value contained in the user input lacks sanitization used by the application to configure a 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 Web application, to which users (and test engineers) do not have direct access. Moreover, in many cases the database is installed on another server. For these reasons, probing the database is out of question.

Test purpose approach follows the same verdict assignment process as penetration testers, which consists of "taking what the Web application gives you". The amount of information about the database that is leaked by the application varies a lot. Hence, SQL injections cannot be tackled with only one test purpose but with a set of three test purposes, each one implementing a dedicated SQL injection.

2.2.2.1 Error-Based SQL Injections

This is the best case scenario for a hacker / test engineer. Error-based SQL injection 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[1], 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 temper with the query.

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

Listing 10 – Test purpose for error-based SQL injection

Listing 10 shows the test purpose for Standard SQL injections. There is only one iterator for the first phase (line 1) that receives user input identifiers, since all user inputs must be tested regardless of their possible resurgence. To perform such a testing coverage, the `for_each` iterates a keyword list,

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

called `#SQLI_VULN_PARAMETERS`, that lists all the SQL injections vulnerable parameters referenced in the test model.

The second phase starts on lines 2 to 5: the test purpose instructs the test generation engine to satisfy, in the context of the Web application instance, an OCL expression composed of two 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, it enables to reach the right state to inject the user input.

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

2.2.2.2 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 with the injection of two 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, such as

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

that uses a single quote, which effect is 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 Web application between the two injections. The test purpose for time delay SQL injections is depicted in Listing 11.

```
1  for_each literal $param from #DATA
2  use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
3  "not(self.ongoingAction.oclIsUndefined()and
4          "self.ongoingAction.all_inputs->exists(d:Data|d.id=DATA_IDS::$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.oclIsUndefined()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()
```

Listing 11 – Test purpose for time delay SQL injection

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 two 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()`" performs 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 Web application in order to perform another injection within the same conditions. Once the sequence has been executed two times, the injections results are assessed by calling the "`threat.checkTBSQLI()`".

2.2.2.3 Boolean-Based SQL Injections

Another technique for Blind SQL injections is to perform several attacks and conduct a differential analysis between the server responses. The test pattern we rely on to create this test purpose has been designed following the testing strategy[2] 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 Web application under test.

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 therefore composed of four injections, as follows:

1. **Nominal Injection**: This is the intended interaction with the Web application. The server response is used as "control group" and its objective is to compare the nominal behavior of the application with its behavior when receiving SQL fragments as input.

2. **AND TRUE**: The objective is to inject an SQL fragment that is always evaluated to true and does not change the overall value of the query, such as:

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

Based on the monotone law of identity for AND, since the Boolean sub expression`1=1` is always true and because it is tied to a conjunction, the result of the expression depends on the other sub-expression of the conjunction

3. **AND FALSE**: The objective is to inject an SQL fragment that that is always evaluated to false and changes the overall value of the query, such as:

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

Based on the monotone law of identity for AND, 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.

4. **OR FALSE**: This injection is similar to the AND TRUE injection, and is mainly used 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 OR, 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 of the disjunction.

[2] http://www-01.ibm.com/support/docview.wss?uid=swg21659226 [Last visited: September 2015]

Verdict is assigned by comparing the responses from the server. If all responses are equivalents, it can be assumed that SQL injections are not possible. 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, it can be assumed that there is a strong possibility that the injected user input is vulnerable to SQL injections.

This attack has been translated into a test purpose, as shown in Listing 12.

```
1   for_each literal $param from #DATA
2   use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
3   "not(self.ongoingAction.oclIsUndefined() and
4           "self.ongoingAction.all_inputs->exists(d:Data|d.id=DATA_IDS::$param)"
5   on_instance was
6   then use was.finalizeAction()
7   repeat 3 times
8       then use was.reset()
9       use any_operation_but #UNWANTED_OPS any_number_of_times to_reach
10      "not(self.ongoingAction.oclIsUndefined() 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()
```

Listing 12 – Test purpose for Boolean-based SQL injection

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 two main sequences. The first one consists of sending nominal values and collecting the resulting page. First, the test purpose in line 2 proposes 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, the test purpose instructs to use any behavioral or navigational operation, as many times as necessary, to satisfy the OCL expression defined in line 8. Satisfying this expression means finalizing the ongoing action, which implies the submission of the form, or click on the link.

The second sequence is responsible for the completion of the three SQL injections. Since the protocol is the same for each injection, the repeat keyword is used to simplify the test purpose and save test generation time.

Therefore, each attack starts by calling the "`was.reset()`" operation, in order to put the test model back to its initial state. Then, 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 crafted request is submitted to the server in line 14.

Once the attack sequence has been executed three times, a call to the "`threat.checkBlindSQLI()`" operation in line 16 compares the responses to assign a verdict.

2.2.3 Cross-Site Request Forgeries

A Cross-Site Request Forgery attack (CSRF for short) consists of tricking a victim into making a specific request through his browser that will ultimately lead to unwanted consequences on a trusted Web application. It is qualified as malicious because it indirectly impersonate a user to perform actions only him or a restricted group of users is allowed to do, and without him 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 test pattern strategy in use 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 PoC[3], 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.
2. **Information collect**: The link / form being responsible for the triggering of the action is retrieved, the output page for later comparison is also collected.
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. To do this, a dedicated java program 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. If both results are similar, it can be concluded that CSRF attacks are possible.

This strategy has been translated into the test purpose described in Listing 13.

```
1   for_eachliteral$actionfrom#CSRF_(SESSION_TYPE)_ACTIONS
2   useany_operation_but#UNWANTED_OPSany_number_of_timesto_reach
3   "not(self.ongoingAction.oclIsUndefined()andself.ongoingAction.id=ACTION_IDS::$action
4   "andself.session_type=(SESSION_TYPE))"on_instancewas
5   thenusethreat.gatherCSRFInfo()
6   thenusewas.finalizeAction()
7   thenusewas.reset()
8   thenuseany_operation_but#UNWANTED_OPSany_number_of_timesto_reach
9   "not(self.ongoingAction.oclIsUndefined()and
10  andself.session_type=(SESSION_TYPE))"on_instancewas
11  thenusewas.finalizeAction()
12  thenusethreat.performCSRFAttack()
13  thenusethreat.checkCSRF()
```

Listing 13 – Test purpose for cross-site request forgery

In the first phase, all the actions that are part of the test objective regarding CSRF are collected. 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 in line 3-4 that requires to put the model in a state where the action is ongoing. Then in line 5, The "`threat.gatherCSRFInfo()`" operation is called to retrieve the Web form or link that is used to submit the action. In line 6, the actions are finalized and then the application is reset in line 7. The attack sequence starts in line 8-10 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 12, and the two results are compared in line 13, by calling the "`threat.checkCSRF()`"operation.

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

2.2.4 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 front-end, 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 strategy implemented to test Privilege Escalation is called Forced Browsing. The objective is to obtain a direct URL to trigger an 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 actions or pages in the GUI but forgot to enforce the restriction in the actions' code. Thus, the test pattern strategy for Privilege Escalation consists of the following steps:

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

If the output results are equivalent, it constitutes an indicator that the restricted page or action can be accessed. This strategy has been formalized in two test purposes: the first one for pages and the second one for actions.

2.2.4.1 Privilege Escalation of Pages

As shown in Listing 14, the first phase of the test purpose for Privilege Escalation of pages, is composed of two 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, a dedicated private operation "`isAccessible()`" of the test model is used to define 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, the relevant information is collected using the "`collectPage()`" operation. 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 Web application under test is the session type from the iterator. Once the system is in the right state, the restricted page is accessed using the "`accessPage()`" operation. The last step is verdict assignment, thanks to the "`checkPrivilegeEscalation()`" operation.

```
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.oclIsUndefined()"
7  on_instance $page
8  then use threat.collectPage()
9  then use was.reset()
```

```
10  thenuseany_operation_but#UNWANTED_OPSany_number_of_timesto_reach
11  "self.ongoingAction.oclIsUndefined()and
12  "andself.session_type=SESSION_TYPES::$role"
13  on_instancewas
14  thenusethreat.accessPage()
15  thenusethreat.checkPrivilegeEscalation()
```

Listing 14 – Test purpose for privilege escalation of pages

2.2.4.2 Privilege Escalation of Action

The test purpose for privilege escalation of restricted actions, introduced in Listing 15, shares a similar structure with the one for pages.

```
1   for_eachliteral$sessionfrom#SESSION_TYPES,
2   for_eachinstance$actionfrom
3   "self.all_pages->select(p:Page|not(self.isAccessible(SESSION_TYPES::$role,p.id)))
4   ->collect(p:Page|p.all_actions" on_instancewas,
5   useany_operation_but#UNWANTED_OPSany_number_of_timesto_reach
6   "self.was_ca.ongoingAction=self"on_instance$action
7   thenusethreat.activateCapture()
8   thenusewas.finalize()
9   thenusethreat.collectPage()
10  thenusewas.reset()
11  thenuseany_operation_but#UNWANTED_OPSany_number_of_timesto_reach
12  "self.ongoingAction.oclIsUndefined()and
13  "andself.session_type=SESSION_TYPES::$role"on_instancewas
14  thenusethreat.triggerAction()
15  thenusethreat.checkPrivilegeEscalation()
```

Listing 15 – 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 7, 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 (targeted URL, parameters, etc.). Then, the action is submitted, the page result collected, and the application reset in lines 8-10.

The attack sequence first requests to put the model in a state where the current session type of the Web application corresponds to the one from the iterator, and where no action is on-going (meaning the authentication credentials has been submitted). Line 14 tries to trigger the action by calling the "`triggerAction()`" operation, using the information collected by the "`activateCapture()`" operation. Finally, the two outputs from the server are compared by calling the "`checkPrivilegeEscalation()`" operation.

2.3 Synthesis

This section describes the update regarding test purpose language expressiveness. These updates allow on the one hand to make the vulnerability test purposes generic, and on the other hand to make them more efficient in detecting vulnerabilities. More precisely, the section details the test purposes of the four vulnerabilities that have been mostly targeted during the RASEN case studies: Cross-Site Scripting, SQL Injections (error-based, time-based and Boolean-based), Cross-Site Request Forgeries and Privilege Escalation (page-based and action-based).

Each of these test purposes allows producing one or several abstract test cases verifying the test purpose specification and the behavioral test model constraints. Such a test case takes the form of a sequence of steps, where a step corresponds to an operation call representing either an action or an observation of the system under test. It also embeds the security test strategies (from security test patterns) that is next used to apply data fuzzing strategies on attack vectors during test scripts generation and execution, as described in the testing process illustration depicted in Figure 1.

The next and last phase of the testing process consists of exporting and executing the test cases in the execution environment in order to provide test results. In the present case, it consists of creating a JUnit test suite, where each abstract fuzzed test case is exported as a JUnit test case, and creating an interface. This interface defines the prototype of each operation of the application and links the abstract structures / data of the test cases to the concrete ones. Since this process ensures the traceability between the verdict of the test case execution and the targeted vulnerabilities identified during risk assessment, the test results can be gathered and processed to provide testing metrics that help engineers to complement the risk picture of the system under test. The next section introduces the test result aggregation that makes it possible to deliver such relevant and useful testing metrics.

3 Security Test Result Aggregation

Security test result aggregation is the process of summarizing test results in a meaningful way. Within the RASEN context, testing metrics is a concept to transfer the information from security testing to risk assessment. Test metrics in general can serve as an important indicator of the efficiency and effectiveness of a software testing process. In RASEN test result aggregation is specified on basis of metrics that use the information contained in the RASEN Test Result Exchange Format (see Deliverables D5.4.2 and D5.4.3). The aggregation is processed by the RASEN Testing Dashboard that is described in Section 3.5. The aggregation results are propagated via the RASEN Aggregated Test Result Exchange Format so that they can be processed in Security Risk Assessment Tools according to the integration scenarios defined in Deliverable D5.4.3. The Sections 3.1, 3.2, and 3.3 specify a set of testing metrics that could be used for test aggregation. The definitions show ID, Name and Description of the metric. The Metric Description uses references to items from the RASEN conceptual model and the RASEN Exchange Format. These references are denoted with a starting backslash (e.g. \testItem). Section 3.5 shows the implementation of the test metrics and the process of test aggregation by means of the RASEN Testing Dashboard.

3.1 List Up Metrics

List up metrics are the most basic kind of a testing metrics. Applying their functions does nothing but listing up a summary of the most important test results in a format specified by the metric. The results are used as documentation in the risk graphs. Additionally, list up metrics can be used to identify any unexpected incidents. These can be suggested as potential new unwanted incidents to the risk analysts. Table 1 specifies a set of simple list up metrics.

ID	Name	Description
LU1	# of specified test cases	counts up all specified test cases for a certain \testItem, /testCoverageItem. The # of specified test cases is usually an indicator for the intended coverage of the \testItem or \testCoverageItemwith test cases.
LU2	# executed test cases	counts up all specified test cases for a certain \testItem, /testCoverageItem. The # of executed test cases is usually an indicator for the actual coverage of the \testItem or \testCoverageItemwith test cases.
LU3	# of passed test cases	counts up all test cases for a certain \testItem, \testCoverageItemthat have been executed and passed. The # of passed test cases is usually an indicator for a lower probability of the existence of errors or vulnerabilities in covered functions of the \testItem.
LU4	# failed test cases	counts up all test cases for a certain \testItem, \testCoverageItemthat have been executed and failed. The # of failed test cases is usually an indicator for the existence of vulnerabilities in the \testItem.
LU5	# inconc test cases	counts up all test cases for a certain \testItem, \testCoverageItemthat have been executed and shows an inconclusive result. The # of inconclusive test cases is usually an indicator for open issues that need to be resolved manually.
LU6	# of error test cases	counts up all test cases for a certain \testItem, \testCoverageItemthat have been executed and shows an erroneous result. Erroneous results are caused by errors in the test system and not by errors in the \testItem. The # of erroneous test cases is usually an indicator for the quality of the test system or its connection with the \testItem.

LU7	# incidents	counts up all incidents that occur during the execution of test for a \testItem, \testCoverageItem(an incident is indicated through test case that result to fail and error).
LU8	# errors	counts up all errors that occur during the execution of test for a \testItem, /testCoverageItem.
LU9	Fail/pass ratio	ratio of # of failed test cases to # passed test cases. The ratio is usually an indicator for the effectiveness of the test cases and the stability of the software.
LU10	Test execution stats: executed/specified ratio	ratio of # of executed test cases to # specified test cases. The ratio is usually an indicator for the status of the test execution and the status of the test implementation.
LU13	Vulnerability discovery rate	ratio of total # of vulnerabilities discovered for \testItem, \testCoverageItem to # of test cases.
LU14	Vulnerability density	# of vulnerabilities / total size of the system (e.g. loc, Mbyte of binary, Mbyte of source code).

Table 1 – List up metrics

3.2 Coverage Metrics

This kind of metric tries to calculate how complete the testing was. Such metrics measure for example, how much of the potential input value space has actually been created as test data or how much of the code of the system under test has in fact been executed during the testing process. Coverage metrics are widely used for all kinds of testing and there is a large amount of literature on that subject [13][12][11].

Coverage metrics are typically used as an indicator for the overall test quality. Results can be used for documentation purpose within the risk analysis. Eventually coverage of negative tests might be an indicator for the likelihood that some vulnerability exists at all. Table 2 specifies a set of coverage metrics by denoting the ID, a name, a description of the metric and by showing references to the list up metrics from Table 1 that could be used to detail the respective coverage statements.

ID	Name	Description	Combinations
C1	Requirements or specification coverage	percentage of requirements/features/specification elements that are addressed by test cases/test procedures. Can be used as an indicator for the completeness of testing.	LU1 -LU10, E1, E2
C2	Attack surface coverage	percentage of the attack surface elements that are addressed by test cases/test procedures. Can be used as an indicator for the completeness of testing. Can be extended by counting up # of test cases/ resources used for each interface item of the attack surface or by differentiating the vulnerabilities (C3) and the respective attack vector (see C4).	LU1 -LU10, E1, E2
C3	Known/expected vulnerability coverage	percentage of the known/expected vulnerabilities that are addressed by test cases/test procedures. Can be extended by counting up # of test cases/ resources used for each known/expected vulnerability. Can be weighted with factors estimating severity, probability and detectability.	LU1 -LU10, E1, E2

C4	Attack vector (threat scenario) coverage	counts up # of test cases/resources used for each attack vector class/partition. Classes/partitions are derived by classification/equivalence partitioning of a given attack vector. Each partition can be weighted with factors estimating factors estimating severity, probability and detectability.	LU1 -LU10, E1, E2
C5	Attack path coverage	percentage of the known/expected attack paths. Can be extended by counting up # of test cases/resources used to test for vulnerabilities in the attack paths that lead to an unwanted incident.	LU1 -LU10, E1, E2, C2-C4
C6	Vulnerability coverage efficiency of RA model: # vulnerabilities not anticipated by RA/ (# vulnerabilities anticipated by RA + # vulnerabilities not anticipated by RA)	ratio of # incidents not covered by RA to (# vulnerabilities covered by RA + # vulnerabilities not covered by RA) indicates whether the actual RA covers the relevant vulnerabilities of the system.	

Table 2 – Coverage metrics

The metric C4(Attack Vector Coverage) is a variant of a metric that measures the coverage of the input space equivalence partitioning. Equivalence partitioning is a software testing technique that divides the input data or test scenarios into partitions. The data or scenarios within one partition are considered to be equivalent with respect to the given testing problem, thus it is expected that the results do not to differ for any of the data or scenarios in one partition. In theory, equivalence partitioning requires only one test case for each partition to evaluate the software properties for the related partition.

In the case of security testing, we propose to define partitions on basis of the attack vector for a given vulnerability. The attack vector itself comprises all possible attacks to exploit a given vulnerability. By means of decomposition we have tried to identify attack vector classifications to distinguish attack vector partitions with equivalent attack vectors. Examples for vulnerabilities, the related attack vector and the proposed attack vector classifications could be found in Table 3.

Vulnerability	**Attack vector**	**Attack vector classification**
CWE-89: Improper Neutralization of Special Elements used in an SQL Command	SQL injection	1) Union exploitation 2) Boolean exploitation a) Force usage of logical operations for invalidating values b) Force usage of big numbers for invalidating values 3) Error-based exploitation 4) Out of band exploitation 5) Time delay exploitation a) Use of escaping mechanism b) Randomcase 6) Stored procedure exploitation 7) Obfuscation of the payload 8) Stacked queries exploitation

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	XSS attack	1) Server XSS 2) Client XSS
CWE-287: Improper Authentication	Exploit weak authentication mechanism	1) Exploit use of default credentials 2) Exploit weak lock out mechanism 3) Bypassing authentication schema a) Direct page request (forced browsing) b) Parameter modification c) Session ID prediction d) SQL injection 4) Exploit remember password functionality 5) Exploit weak password policy 6) Exploit weak security question/answer 7) Exploit weak password change or reset functionalities 8) Exploit weaker authentication in alternative channel
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')	Code injection	1) Direct injection 2) Indirect injection

Table 3 – Attack vector classification examples

3.3 Efficiency Metrics

Efficiency metrics are used to calculate how much effort has been spent for testing. These metrics are especially interesting for the case that with the testing effort spend so far no fault or unwanted incident has been triggered. The idea is that using the same attack strategy, which was used for testing, an attacker will probably have to spend even more resources in order to trigger an unwanted incident.

The result of an efficiency metrics for security testing is an indicator for the costs of related threat scenario. Taking the resources and the calculation power potential attackers have in relation to these costs might be a good indicator for the likelihood that the threat scenario will be exploited successfully within a given time period. Table 4 shows a set of efficiency metrics for testing.

ID	Name	Description
E1	Test case/procedure preparation complexity: Effort per test case/procedure	sums up the efforts spent for specifying and implementing a test case or a test procedure. The efforts spent could be used as an indicator for the complexity of the testing problem and thus of the detectability of the addressed vulnerability.

E2	Test case/procedure execution time	sums up the efforts/time spent for executing a test case or a test procedure. The execution time might be an indicator for the complexity of the testing problem and thus of the detectability of the addressed vulnerability.
E3	Size of software tested /resources used ratio	ratio of size of software tested to resources used might be used as an indicator of test efficiency or sufficiency of tests.

Table 4 – Efficiency metrics

3.4 Process/Progress Related Metrics

Process/progress related metrics are used to measure the progress of the test process and the respective quality improvements of the test item over time. Table 5 shows two process/progress related testing metrics.

ID	Name	Description
P1	Vulnerability discovery rate increase/decrease	Compares vulnerability discovery rates over time. Can be used as an indicator if additional test effort leads to the identification of more vulnerabilities/failures.
P2	Test case/procedure preparation complexity increase/decrease	Compares test case/procedure preparation complexity over time. Can be used as an indicator if additional test effort would lead a number of reasonable new test cases.

Table 5 – Process/progress related metrics

3.5 The RASEN Testing Dashboard

The Testing Dashboard is designed to realize and visualize the security testing metrics defined in Chapter 3 and to provide an exporter for aggregated test reports[8].

3.5.1 Principles

A RASEN security testing metric refers to multiple parts of security models like risk model and test report. The RASEN Testing Dashboard manages the referenced models for such metrics, generates metrics and measurements for the security elements of interest, visualizes them in different views and provides an exporter for aggregated results.

3.5.2 Architecture

The Testing Dashboard is designed as a Java plugin for the Eclipse environment. It consists of a risk test model analyzer, a metric generator and two different views embedded in the Eclipse workbench (see Figure 3): a dashboard metric table view and a metric chart view. The risk test model analyzer processes registered models for metric generation and visualization. The GUI part of the Testing Dashboard provides different user interaction options, including selecting elements and metrics for the analysis, and setting different parameters for their visualization. Selected elements and their metrics can be exported as aggregated test reports.

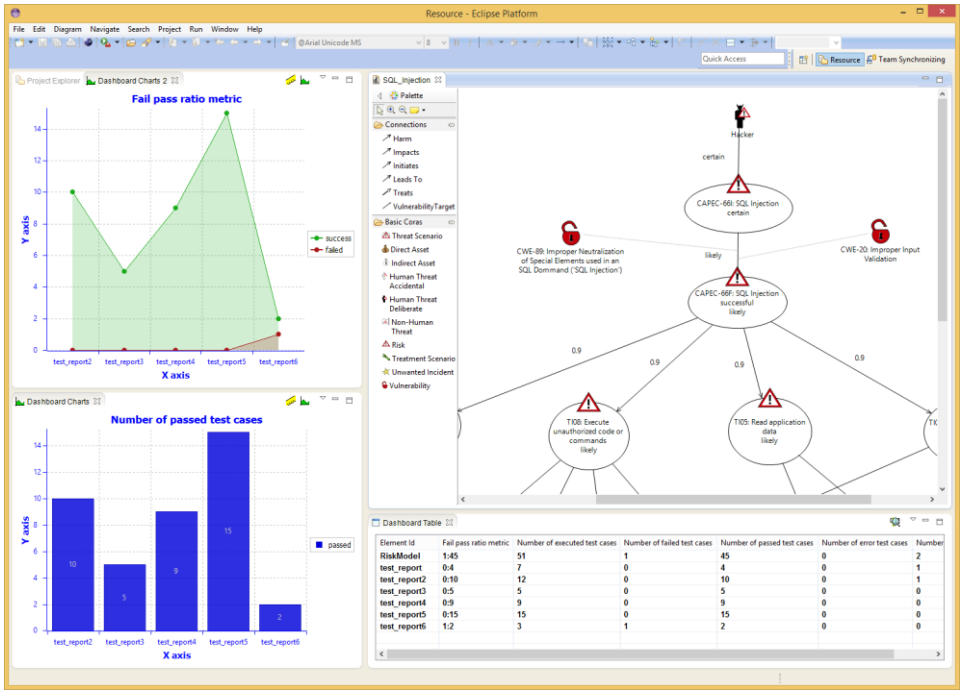


Figure 3 – RASEN testing dashboard embedded in Eclipse

The dashboard architecture consists of the models to be analyzed and the aggregated report as the analysis result in the bottom level, the analyzer and the test metric generator in the middle, and the GUI part on the top level (see Figure 4).

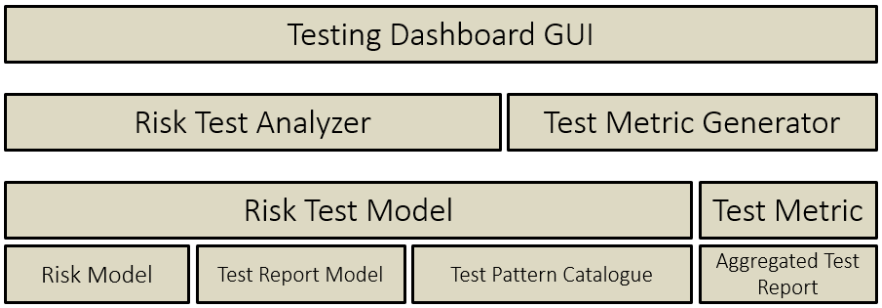


Figure 4 – RASEN testing dashboard architecture

The central part of RASEN Testing Dashboard is the analyzer component compositing and working on the associated risk test model, and the test metric generator processing the security metrics with help of the analyzer. The analyzer collects all data from the model needed for single analysis requests from the GUI or the test metric generator. The risk test model is a unified model of all registered models and will be used by the analyzer. Such single model can be dynamically added to and removed from the dashboard. Each model part, namely a risk model, a test report model and also a test pattern catalogue must be defined in the exchange format [7], an XML Ecore format, and will be loaded by an EMF loader. The risk test analyzer links such loaded EMF models together to the risk test model. To identify a relation between a risk model and a test report model, for instance, these models contain reference elements with only an identifier attribute for the related element in the related model. These reference elements are required for tracking relationships between test results, metrics and the risk test model.

The metric generator realizes the security testing metrics and enables the export into the aggregated test report format. The test metric format and the aggregated test report format is in detail described in [8]. The risk test analyzer will be used to resolve inquiries for the risk test model to create the metric and measurement values. Such values are used for visualization in the GUI as well as for the

aggregated test report model. The exporter uses an appropriated EMF model adapter to export such report model into an XML Ecore file, formatted to the according XML Schema specification.

Finally, the dashboard GUI on the top level represents the user interaction part of the dashboard. It consists of two views and GUI elements for user configurations and will be described in the next chapter in detail.

3.5.3 GUI

The dashboard GUI as the user interaction part is integrated in the Eclipse workbench and its views are realized as Eclipse SWT view elements. The Testing Dashboard uses model parts, which are dynamically registered, as described before. To register, the user can select files of such models in the project explorer and also in the Java package explorer. By right-click on one or more selected files he can select the option “Register Model for Testing Dashboard” in the popup menu dialog (see Figure 5) for registration. The analyzer will try to load and register all selected files. If the loading fails, Eclipse will open an information dialog and continue with the next file. If the loading succeeded, the risk test model will be immediately updated, and also all dashboard views and shown metric values.

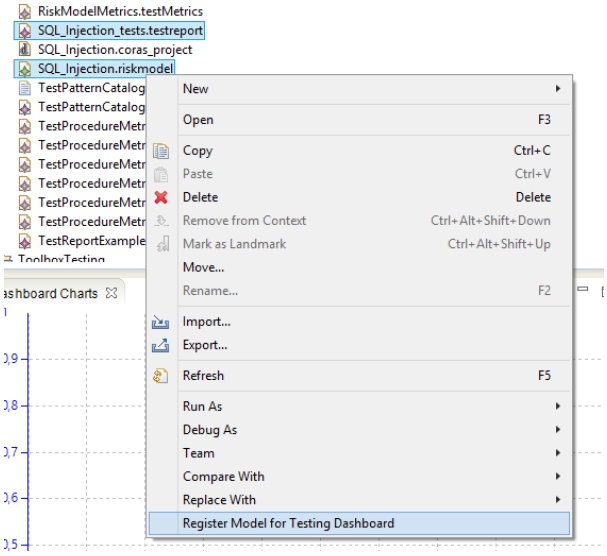
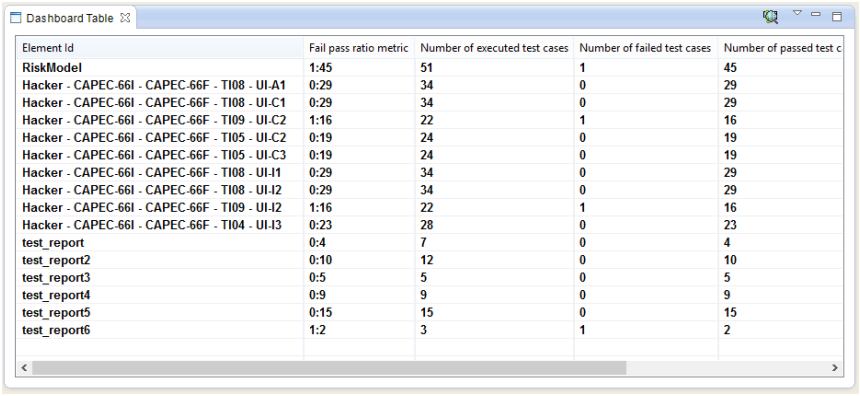


Figure 5 – Register model dialog

Only models in the exchange format defined in [7] are valid for the registration. Additionally, risk models in the CORAS risk format [16] can also be directly registered. This is realized by an internal transforming into the risk model exchange format. The registered models will be saved as property values in the Eclipse storage so that all models will be loaded with the start of a new Eclipse dashboard session.

The first view of the dashboard GUI is the dashboard metric table (see Figure 6). The view can be found in Eclipse in Window -> Show View ->Other in the “Other” folder. The view lists categorized elements and their metric values in a table format. Each line shows an element followed by the metric values of all registered metric types. By selecting the option “select displaying object(s)...” (see Figure 7) the user can choose one or more categories for displaying elements in the dashboard. A category can be “unwanted incidents” that displays all unwanted incidents of registered risk models and all metrics related to each incident, or can be “test procedure” that displays test procedures of registered test reports and all metrics related to them.



Element Id	Fail pass ratio metric	Number of executed test cases	Number of failed test cases	Number of passed test c
RiskModel	1:45	51	1	45
Hacker - CAPEC-66I - CAPEC-66F - TI08 - UI-A1	0:29	34	0	29
Hacker - CAPEC-66I - CAPEC-66F - TI08 - UI-C1	0:29	34	0	29
Hacker - CAPEC-66I - CAPEC-66F - TI09 - UI-C2	1:16	22	1	16
Hacker - CAPEC-66I - CAPEC-66F - TI05 - UI-C2	0:19	24	0	19
Hacker - CAPEC-66I - CAPEC-66F - TI05 - UI-C3	0:19	24	0	19
Hacker - CAPEC-66I - CAPEC-66F - TI08 - UI-I1	0:29	34	0	29
Hacker - CAPEC-66I - CAPEC-66F - TI08 - UI-I2	0:29	34	0	29
Hacker - CAPEC-66I - CAPEC-66F - TI09 - UI-I2	1:16	22	1	16
Hacker - CAPEC-66I - CAPEC-66F - TI04 - UI-I3	0:23	28	0	23
test_report	0:4	7	0	4
test_report2	0:10	12	0	10
test_report3	0:5	5	0	5
test_report4	0:9	9	0	9
test_report5	0:15	15	0	15
test_report6	1:2	3	1	2

Figure 6 – Dashboard table view

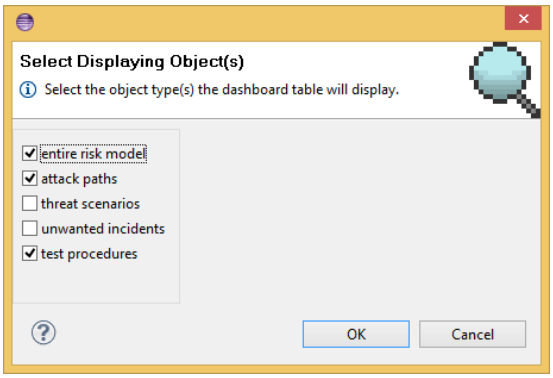


Figure 7 – Select displaying object category dialog

The user can also remove registered models or clear the whole dashboard by selecting the appropriate option in the option menu. Finally, all shown elements and metrics can be exported into a XML file. The exported format conforms to the aggregated test report format.

The second view is the metric chart view (see Figure 8) and can be found in Eclipse in Window -> Show View ->Other in the “Other” folder.

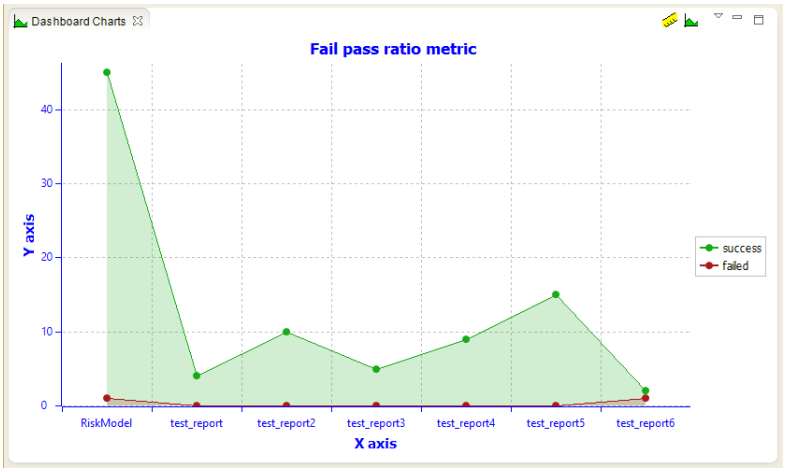


Figure 8 – Dashboard charts view

The chart view visualizes elements and their metric result in a chart diagram. The elements are selected by the user in the dashboard table view or in the CORAS diagram editor. The metric type for the charts is also selectable by users in the “select metric” dialog (see Figure 9). He can choose one of

all registered metrics in the dashboard. The user has also the option to switch between different chart types, namely: a bar chart, a stacked bar chart, a line chart and an area chart.

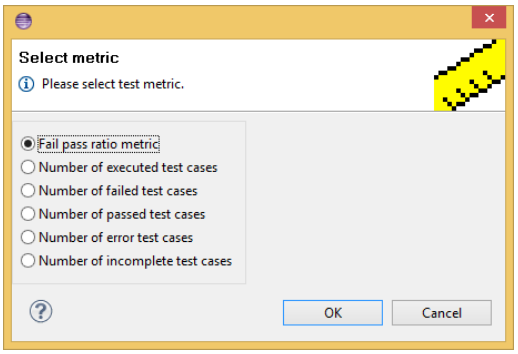


Figure 9– Select metric for dashboard charts

Finally, all visualized elements in chart diagram can also be exported in the same way as in the dashboard table view.

3.5.4 API and Implementation Guidelines

The RASEN security dashboard is designed to create new metrics and measurements with customizable visualizations. The risk test model can be used to evaluate extensive risk inquiries such as determining a risk value for a risk element or estimate the test effectiveness and test effort level by considering related test patterns and test reports for the risk element. This can be inevitable when complex metrics such as efficiency metrics are added to the dashboard.

The dashboard metric generator realizes the most of list up metrics in the current state. Other metrics, particularly single coverage metrics and efficiency metrics, need complex analytical methods on the risk test model for realization.

The underlying metric and measurement model handling intends to extend it for new metrics and measurement types for further development (see Figure 10). The test metric and measurement model is generated as an EMF Ecore model. A test metric model handler is available to create such models in a comfortable way. Customized metric and measurement classes are using this handler and have a small interface for specific calculations and for inquiry handling on the risk test model. These customized classes can be used to develop new metric types. To add the new metrics to the dashboard, the plug-in has to realize a *TestMetricFactory* class that provides all such new metrics, and register this factory to the provided extension point. The testing dashboard will dynamically load all such registered metric factories and provides the metrics for further handling of the metric generator and the dashboard GUI. To show a metric in the chart diagram, a specific visualization has to be specified, realized by a customized visualization class with different customization options like naming, coloring and displaying input values. A metric can only be displayed in the chart diagram with an according visualization.

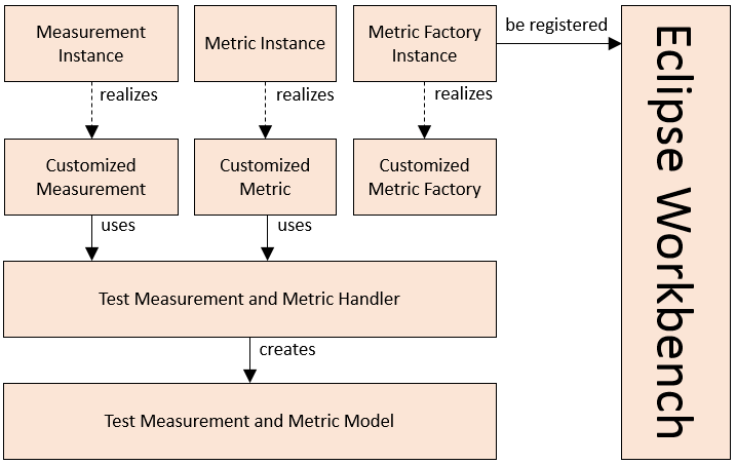


Figure 10 – Overview of the test measurement and metric API

4 Summary

The overall objective of RASEN WP4 is to develop techniques to use risk assessment as guidance and basis for security testing, and to develop an approach that supports a systematic aggregation of security testing results by means of security testing metrics. The objective includes the development of a tool-based integrated process for guiding security testing by means of reasonable risk coverage and probability metrics. This deliverable is the last deliverable of a series of three deliverables that describes the RASEN tools and techniques for risk-based security testing.

Deliverable D4.2.1 has introduced a technique for risk-based test identification and prioritization and an approach for test identification and generation based on the notion of security test pattern and their formal representation by means of a test purpose language.

Deliverable D4.2.2 has updated the technique for risk-based test identification and prioritization with an approach for test identification, prioritization, selection and test case derivation based on CAPEC attack patterns. The approach describes how a test procedure can be derived in three steps by (1) generating generic risk models from CAPEC attack patterns, (2) adapting them to the target of the risk assessment and (3) deriving from the target-specific risk model a test procedure consisting of select and prioritized test scenarios. Based on these test scenarios, test patterns may be selected as starting point for test case derivation. They provide refined techniques for test generation (stimulation strategies) and test verdict arbitration (observation strategies). The actual test case generation starts by instantiating a security test pattern, employing security test purposes for test sequence generation and fuzzing techniques for actual security test case generation.

First, this deliverable has described the update developed to improve and extend the test purpose language expressiveness. These updates enable providing vulnerability test purposes formalizing complex and sophisticated attacks, and on the other hand to make them more efficient to detect vulnerabilities. More precisely, we have detailed the test purposes of the four types of vulnerability that have been targeted within the RASEN case studies: Cross-Site Scripting, SQL Injections (error-based, time-based and Boolean-based), Cross-Site Request Forgeries and Privilege Escalation (page-based and action-based).

Second, this deliverable has introduced the notion of test result aggregation and a testing dashboard in order to provide an overview of test progress and results. The RASEN Testing Dashboard is the implementation of a set of testing metrics that provide a problem specific view on the results of security testing. The RASEN Testing Dashboard on one hand allows the visualization of test results and their problem specific aggregation by testing metrics both in the context of testing as well as in the context of the initial risk assessment. On the other hand the RASEN Testing Dashboard allows exporting aggregated test results (i.e. test results and test metric results) for further processing by other tools.

All of the above mentioned achievements provide a set of techniques that completely support the risk-based security testing process, beginning with risk assessment, followed by test identification, test prioritization, test selection, test generation, test execution and finally test result aggregation, test reporting and visualization (see Figure 1on page 6).

References

[1] RASEN Deliverable D4.2.1, Techniques for Compositional Risk-Based Security Testing v.1, 2013

[2] RASEN Deliverable D4.2.2, Techniques for Compositional Risk-Based Security Testing v.2, 2014

[3] RASEN Deliverable D5.3.1, Methodologies for legal, compositional, and continuous risk assessment and security testing v.1, 2013

[4] RASEN Deliverable D5.3.2,Methodologies for legal, compositional, and continuous risk assessment and security testingv.2, 2014

[5] RASEN Deliverable D5.3.3,Methodologies for legal, compositional, and continuous risk assessment and security testingv.3, 2015

[6] RASEN Deliverable D5.4.1, A toolbox for risk assessment and security testing v.1, 2013

[7] RASEN Deliverable D5.4.2, A toolbox for risk assessment and security testing v.2, 2014

[8] RASEN Deliverable D5.4.3, A toolbox for risk assessment and security testingv.3, 2015

[9] F. Seehusen: A Technique for Risk-Based Test Procedure Identification, Prioritization and Selection. In Proc. of the 6th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, to appear.

[10] W.e Jansen: Directions in Security Metrics Research, NISTIR 7564, Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg 2009

[11] M. W. Whalen, A. Rajan, M. P. E. Heimdahl, S. P. Miller: Coverage Metrics for Requirements-based Testing. In Proc. of the 2006 International Symposium on Software Testing and Analysis, pp. 25-36, ACM New York

[12] P. E. Ammann, P. E. Black: A Specification-Based Coverage Metric to Evaluate Test Sets. International Journal of Reliability, Quality & Safety Engineering. Dec 2001, Vol. 8 Issue 4, pp. 275-299, World Scientic Publishing 2001

[13] J. J. Chilenski, S. P. Miller: Applicability of modified condition/decision coverage to software testing. Software Engineering Journal, Volume 9, Issue 5, September 1994, pp. 193 – 200, Institution of Electrical Engineers 1994

[14] International Organization for Standardization/: ISO/IEC 29119-1 Systems and software engineering—Software testing—Part 1: Concepts and definitions (2013)

[15] Open Web Application Security Project: Top 10 2013 (2013). [ONLINE] Available at: https://www.owasp.org/index.php/Top_10_2013 [Accessed 8 September 2013]

[16] M. S. Lund, B. Solhaug, K. Stølen: Model-Driven Risk Analysis - The CORAS Approach, Springer, 2011