Compositional Risk Assessment and Security Testing of Networked Systems

Deliverable D4.2.1

Techniques for Compositional Risk-Based Security Testing v.1

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:	D4.2.1
Nature of deliverable:	Report
Dissemination level:	PU
Internal version number:	1.0
Contractual delivery date:	2013-09-30
Actual delivery date:	2013-09-30
Responsible partner:	Fraunhofer

Contributors

Editor(s)	Martin Schneider (FOKUS)
Contributor(s)	Bruno Legeard (SMA), Fabien Peureux (SMA), Martin Schneider (FOKUS), Fredrik Seehusen (SINTEF)
Quality assuror(s)	Samson Esayas (UiO), Albert Zenkoff (SAG)

Version history

Version	Date	Description
0.1	13-06-06	TOC proposition
0.2	13-09-23	SMA contribution
0.3	13-09-06	SINTEF contribution
0.4	13-09-15	FOKUS contribution
0.5	13-09-24	First revision after internal review
0.6	13-09-27	SMA corrections
0.7	13-09-29	Minor corrections
1.0	13-09-30	Final version

Abstract

Work package 4 will develop a framework for security testing guided by risk assessment and compositional analysis. This framework, starting from security test patterns and test generation models, aims to propose a compositional security testing approach able to deal with large scale networks systems. This report provides the first results for how test cases can be derived from risk assessment results by means of risk-based test identification and prioritization, security test patterns and test case generation using security test patterns together with a test purpose language extended for security testing. This is based on the baseline defined in RASEN deliverable D4.1.1 and will be refined and complemented by the subsequent RASEN deliverables D4.2.2 and D4.2.3.

Keywords

Security testing, risk-based security testing, fuzzing on security models, security testing metrics, large-scale networked systems, risk-based test-identification, risk-based test prioritization, security test patterns

Executive Summary

The overall objective of RASEN WP4 is to develop techniques for how 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. The objective includes the development of a tool-based integrated process for guiding security testing deployment by means of reasonable risk coverage and probability metrics.

This document provides techniques for deriving test cases from risk assessment results. The starting point for the development of these techniques is defined by the RASEN deliverable D4.1.1 that provides the baseline for the works.

The description of the techniques for deriving test cases from risk assessment results covers the research task T4.1“Deriving test cases from risk assessment results, security test patterns and test generation models in a compositional way”. The research question relevant in this context is:

What are good methods and tools for deriving, selecting, and prioritizing security test cases from risk assessment results?

This deliverable is the first one of two deliverables that cover this question. It presents techniques for the parts of this research question for identification of security test cases based on risk assessment results, prioritization of security test cases based on risk assessment result, and deriving security test cases from risk assessment results.

Table of Contents

1 Introduction

The objective of RASEN WP4 is to develop techniques for how 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. The objective includes the development of a tool-based integrated process for guiding security testing deployment by means of reasonable risk coverage and probability metrics. In reaching the objectives, WP4 focus in particular on three more specific tasks.

The deliverable provides techniques regarding the first task T4.1 “Deriving test cases from risk assessment results, security test patterns and test generation models in a compositional way”. The presented techniques cover all parts of the related research question

What are good methods and tools for deriving, selecting, and prioritizing security test cases from risk assessment results?

The overall process for deriving test cases from risk assessment results is sketched in Figure 1. It starts on the left at the risk model as a result from the risk assessment. The risk model is used for risk-based test identification and prioritization what is presented in Section2. Especially for complex systems, there are not sufficient resources to test all vulnerabilities and threat scenarios identified during risk analysis. Hence, an identification and prioritization is required as basis for test case generation.

In the next step, security test patterns based on prioritized vulnerabilities from the risk model provide a starting point for test case derivation and is described in Sections3.1 and 3.2. Security test patterns provide a link between risk analysis and security testing by providing information how appropriate security test cases can be created from risk analysis results.

In order to generate security test cases, test sequences based on formalized security test patterns using a Test Purpose Language is described in Sections 3.3, 3.4, and 3.5 has to be generated. These test sequences provide the basis for actual security test case generation. Section 4 presents a way to specify test case derivation by applying security test strategies to a model. Actual security test cases are generated based on these test sequences (upper path), this is described in Section 5.1.These test sequence can also be used as intermediate step for test case generation using data and behavioral fuzzing techniques described in Section 5.2 using security test strategies as described in Section 4. Section 6 gives a summary of this document.

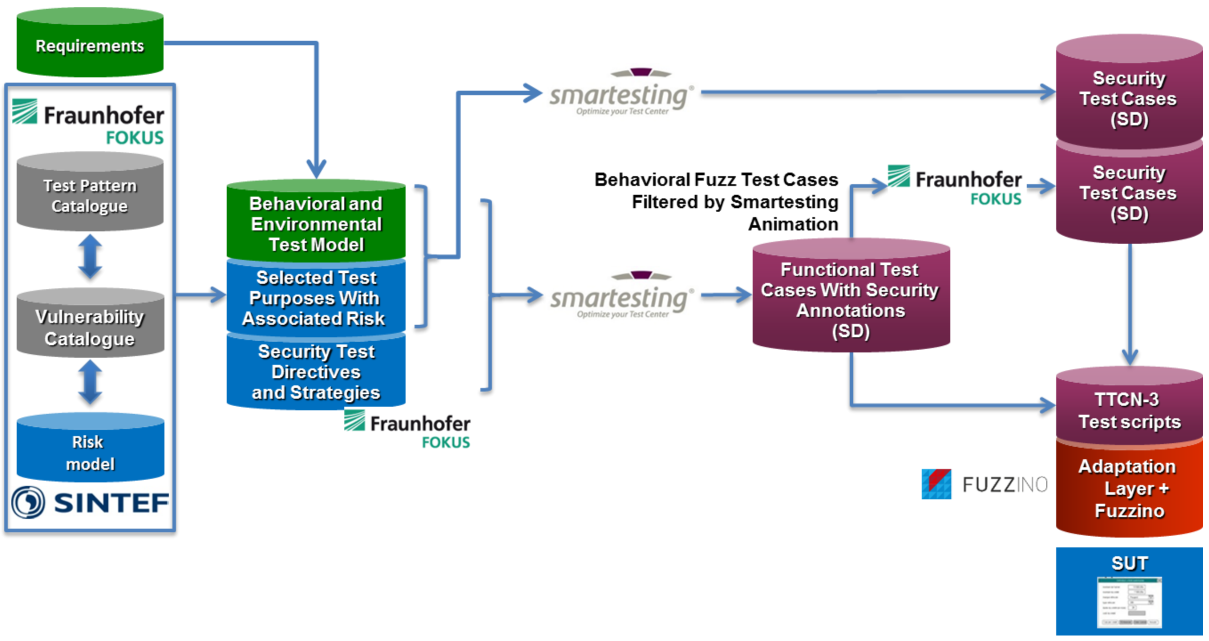

Requirements
Fraunhofer FOKUS
Test Pattern Catalogue
Vulnerability Catalogue
Risk model
SINTEF
Behavioral and Environmental Test Model
Selected Test Purposes With Associated Risk
Security Test Directives and Strategies
Fraunhofer FOKUS
smartesting Optimize your Test Center
smartesting Optimize your Test Center
Behavioral Fuzz Test Cases Filtered by Smartesting Animation
Fraunhofer FOKUS
Functional Test Cases With Security Annotations (SD)
Security Test Cases (SD)
Security Test Cases (SD)
TTCN-3 Test scripts
Adaptation Layer + Fuzzino
FUZZINO
SUT

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

2 Risk-based Test Identification and Prioritization

We present a technique for risk-based test procedure identification, prioritization, and selection. The technique takes a risk model in the form of a risk graph as input, and produces a list of prioritized selected test procedures as output. The prioritization can be automated. However, the risk graph may have to be annotated with information relevant for the prioritization in order to be accurate.

In this section, we define the notion of a *risk graph* based on the notion of a *weighed graph.* We then define a function for prioritizing and selecting test procedures based on a risk graph.

2.1 Risk Graphs

Weighted graphs

Before we define what is meant by a risk graph, we define a notion of a weighted graph. A weighted graph is a directed acyclic graph whose nodes and edges may be annotated by likelihood values. The nodes typically represent occurrences of events, and the likelihood value of a node specifies how likely it is that its associated event will occur. Edges represent causal relationships between nodes. Likelihood values of edges should be understood as *conditional likelihood values*.

In practice, there are many ways to specify likelihood values, e.g. as probabilities, frequencies, or intervals of these. Because of this, we parameterize our notion of a weighed graph by a notion of a *likelihood structure*. The advantage of this is that we do not have to specify separate rules for each possible way of specifying likelihoods.

Definition [Likelihood structure]: A likelihood structure *LS* is a tuple $(L,\oplus,\otimes,\mathbf{1},\mathbf{0})$ consisting of

- A set *L* known as the likelihood values of LS;
- Two elements **1** and **0** in *L* known as the minimum and maximum value of *L*, respectively.
- A binary operator $\oplus$ on *L*, known as the or-operator or the sum-operator.
- A binary operator $\otimes$ on *L*, known as the and-operator or the product-operator.

We denote by *LS.L*, *LS.**0***, *LS.**1***, *LS.*$\oplus$, and *LS.*$\otimes$, the likelihood values, the minimum value, the maximum value, the or-operator, or the and-operator of *LS*, respectively. We sometimes drop the LS. prefix when LS is clear from the context.

Example 1
To express likelihoods in terms of, say, probabilities, we have to instantiate the elements of the likelihood structure. Specifically, L will be defined the set of all real numbers between 0 and 1, **1** is defined by 1, **0** is defined by 0, $\oplus$ is defined by $a \oplus b = a + b - (a * b)$, and $\otimes$ is defined by * (multiplication). We will call this likelihood structure the statistically independent probability structure, and denote it by **P.** ■

Definition [Weighted graph] A weighted graph *G* over a likelihood structure *LS* is a tuple (Q, I, E, l) consisting of

- A set *Q* of states.
- A set of initial state I $\subseteq Q$
- A set of edges $E \subseteq Q \times Q$
- A function l$\in Q \cup E \rightarrow L$, assigning likelihood values to states and edges.

We denote by $G.Q, G.I, G.E$, and $G.l$, the states, initial states, edges, and likelihood assignment function of G. We sometimes just write Q, I, E, or l if G is clear from the context. We require that all weighted graphs be acyclic.

Example 2
Figure 2 shows an example of a weighted graph. This is actually a CORAS threat diagram (which can be seen a special case of a weighted graph) where nodes may be of different kinds, but this can be ignored. The example shows a graph with six nodes labeled by S1 to S6. The edges between the nodes are shown as arrows. None of the nodes have been given likelihood values, but all the edges are labeled by conditional likelihood values in the form of probabilities.

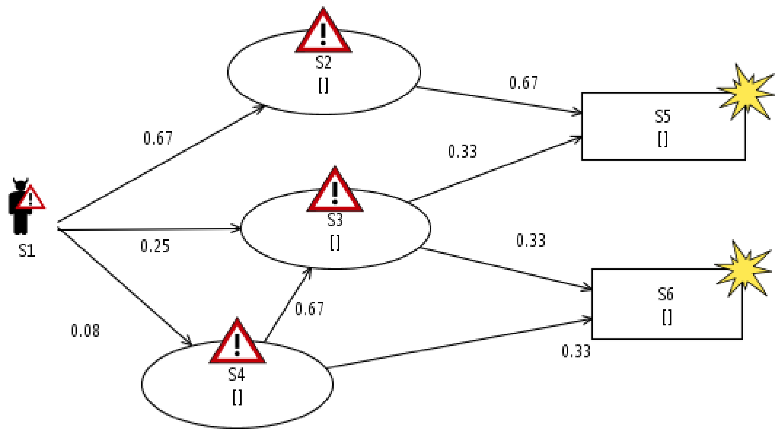


Figure 2 – Example of a weighted graph

■

If $e = (p, q)$ is an edge, then the source and target states of the edge are defined by $src(e) = p$ and $tar(e) = q$. If G is a weighted graph and p is a state in G, then we denote by $src(G, p)$, all the edge in G that have p as target, i.e.,

$$src(G,p) \coloneqq \{e \in G.E \mid tar(e) = p\}$$

To specify the test procedure prioritization function later, we need to be able to calculate the likelihood values of nodes in a graph based on the likelihood values of the edges. This is defined in the following.

Definition [Calculated likelihood of a state] Let G be a weighted graph over a likelihood structure LS where $LS.\oplus$ is commutative. Then the likelihood value of a state p in G, written $l[G,p]$, as calculated from the edges in G, is defined by

$$l[G,p] \coloneqq \begin{cases} \bigoplus_{e \in src(G,p)} l(e) \otimes l[G, src(e)] & if\ src(G,p) \neq \emptyset \\ \mathbf{1} & if\ src(G,p) = \emptyset \end{cases}$$

Note that the order in which the likelihood values are summed should not matter. Therefore we have required that $\oplus$ must be commutative.

Example 3
Figure 3 shows an example of a weighted graph in which the likelihood values of all the nodes have been calculated from the conditional likelihood values on the edges. For instance, node S2 has likelihood value 0.67, and node S5 has likelihood value 0.5;

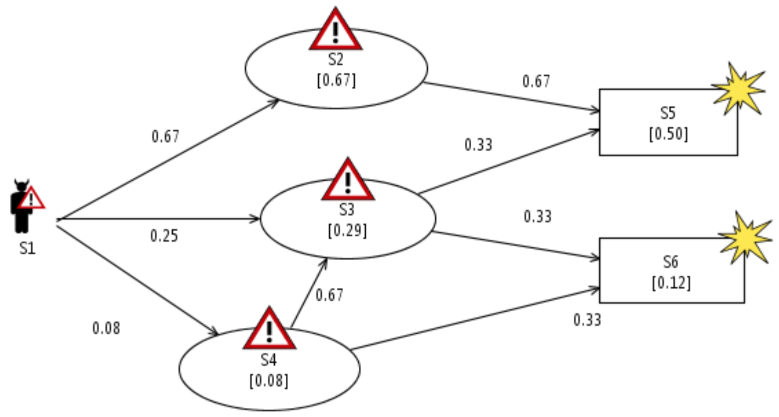


Figure 3 – Example of node likelihood calculation

■

Risk graphs
In this section we define the notion of a *risk graph*, which is basically just a weighted graph whose nodes may be assigned consequence values in addition to likelihood values. We parameterize risk graphs with a notion of a *risk structure.*

Definition [Risk structure] A risk structure RS is a tuple $(LS, C, R, \odot, rv)$ consisting of

- A likelihood structure LS;
- A set of consequence values C;
- A set of risk values R;
- An operator $\odot \in R \times R \to R$
- A risk value function $rv \in LS.L \times C \to R$ mapping likelihood values of LS and consequence values into risk values.

If RS is a risk structure, we denote by $RS.LS, RS.C, RS.R, RS.\odot, RS.rv$ the likelihood structure, the consequence values, the risk values, the risk value or-operator, and the risk value function of RS, respectively.

Definition [Risk graph] A risk graph G over a risk structure RS, is a tuple (Q, E, I, l, c), consisting of

- A weighted graph (Q, E, I, l) over $RS.LS$;
- A partial function $c \in Q \to RS.C$ assigning consequence values of RS to states;

If G is a risk graph, then we denote by $G.Q_c$, or just Q_c if G is clear from the context, the set of all states in G that have a consequence value assigned to it.

In a given risk graph, we refer to all nodes that have a likelihood and a consequence value as risks.

Example 4
We define a risk structure **R** as follows:

- The likelihood structure of **R** is **P** as defined in Example 1.
- The set of consequence values are defined by real values from 1 to 5.
- The set R is defined as the of all non-negative real numbers
- The operator $\odot$ is defined by $r \odot r' := r + r'$
- The risk value function is defined by $rv(l, c) := l * c$

Figure 4 shows a CORAS threat diagram which can be seen as an instance of a risk graph. In these kinds of diagrams, consequence values of nodes are specified by drawing an edge from the nodes to a special kind of node called an asset (illustrated by a money bag) and annotating the edge with a consequence value. Hence Figure 4 shows a risk graph in which node S5 has consequence value 2 and node S6 has consequence value 4. All other nodes in the graph have no consequence values.

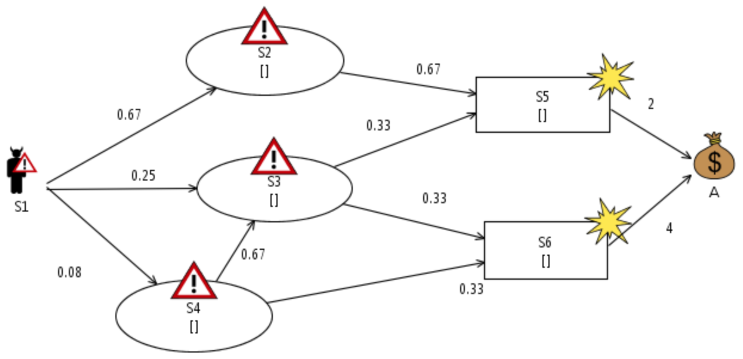


Figure 4 – Example of a risk graph

■

Before we continue, we define a helper function which will be needed later.

Definition [Weight replacement] If e is an edge in G and l is a likelihood value of the likelihood structure of G, then the risk graph obtained by replacing the likelihood value of e by l is denoted by $we(G, e, l)$.

2.2 Test Procedure Prioritization

In this section, we describe a function for prioritizing test procedures on a basis of risk graphs. We make the assumption that every edge in a risk graph is a potential test procedure. Intuitively, testing an edge e in a risk graph should be understood as testing the degree to which the system under test has any vulnerabilities that can be exploited in order to cause the event that is represented by the target node of e, given that the event represented by the source node of e has occurred.

The calculation of the priority is based on the likelihood and the consequence values of risks graphs in addition to an *uncertainty* estimate for each edge in the risk graph. This is an estimate of how uncertain we are about the correctness of the conditional likelihood value of the edge corresponding to a test procedure. If the uncertainty is very low, then there might not be a need for testing, since then the test results may not give any new information. Conversely, a high uncertainty suggests that the test procedure should be prioritized for testing.

The priority calculation a given edge (or test procedure) with a likelihood value l and an uncertainty estimate u in a risk graph G, is based on comparing the risks of two risk graphs: one risk graph where the conditional likelihood of the edge corresponding to the test procedure is *reduced* to a likelihood l_{min} specifying that the edge is less likely to lead up to something, and one risk graph where the conditional likelihood of the edge corresponding to the test procedure is *increased* to a likelihood l_{max} specifying that the edge is more likely lead up to something. More precisely, we assume that u is a likelihood value of the same likelihood structure as l, and that l_{max} is obtained by addition u to l, i.e. $l_{max} = l \oplus u$ and that l_{min} is obtained by effectively subtracting u from l. To formalize the latter, we assume that the $\oplus$ operator has an *inverse* unary operator denoted $\_^{\oplus}$, such that $a \oplus a^{\oplus} = \mathbf{0}$. We can then calculate l_{min} by $l_{min} = l \oplus u^{\oplus}$.

The priority is then the difference between the risk values of the nodes in the two risk graphs. To calculate this precisely, we make of a function $diff \in R \times R \rightarrow \mathbb{R}$ which calculates the difference between to risk values.

We are now ready to define the priority function precisely.

Definition [Priority] Given a function $diff \in R \times R \rightarrow \mathbb{R}$ which computes the difference of two risk values, and a function $u \in E \rightarrow L$ mapping edges to uncertainty values, the priority of an edge e in a risk graph G, denoted $p[G,e]$, is defined as follows

$$p[G,e] := \sum_{p \in G.Q_c} diff\big(rv(l[G_{max},p],c(p)), rv(l[G_{min},p],c(p))\big)$$

where $G_{min} = we(G,e,l(e) \oplus u(e)^{\oplus})$ and $G_{max} = we(G,e,l(e) \oplus u(e))$.

The function is lifted to a set of edges E such that $p[G,E]$ yields the sum of the priorities in E, i.e. $p[G,E] := \sum_{e \in E} p[G,e]$.

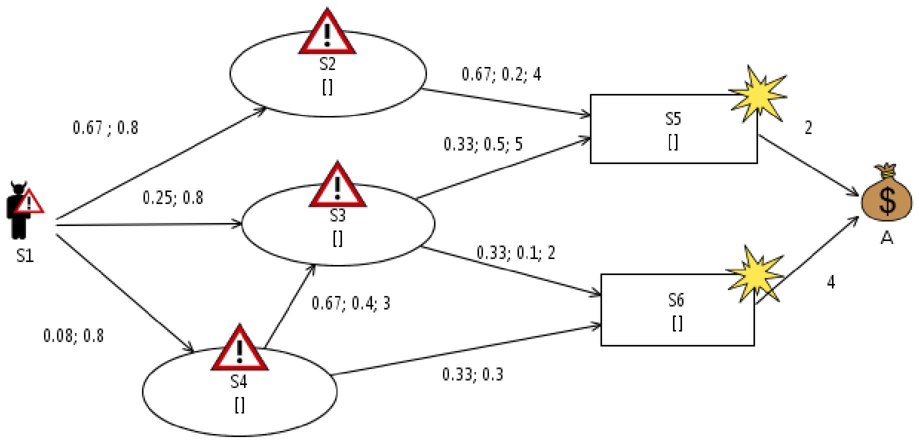


Figure 5 – Example of a risk graph annotated with uncertainty values

Example 5
In Figure 5, we have illustrated a risk graph annotated with uncertainty values. That is, the first values that appear after the first semicolon on each edge are understood as uncertainty values. For example, the edge going from S2 to S5 has uncertainty 0.2, and the edge going from S3 to S6 has uncertainty 0.1. Ignore the values appearing after the second semicolon for now (this will be explained in the next example). 4.

Assuming that we have the same risk structure **R** as in Example 4, and that the function $diff$ is defined by $diff(r,r') := |r - r'|$ (where r and r' are real numbers), the calculation of the priority of each edge in the risk graph of Figure 5 is shown in Table 1.

Source	Target	Priority
s1	s4	1.3697311594560002
s1	s3	0.9362988835200003
s1	s2	0.31988181983999997
s3	s5	0.10715257739999995
s2	s5	0.07997045495999999
s3	s6	0.07572037696
s4	s6	0.058160330879999944
s4	s3	0.013059113759999907

Table 1 – Calculated priority values of edges

■

2.3 Test Procedure Selection

In the previous section, we defined a function for prioritizing each edge in a risk graph under the assumption that each edge represented a potential *test procedure*. However, it is often not the case that every test procedure represented by a risk graph will be refined into concrete test cases. Thus, we will have to *select* those test procedures that will refined into test cases and executed. In this section we define a function for doing this.

In order to define the test selection function, we assume that each edge (or test procedure) is given an *effort value*, i.e. an estimate of the effort required to implement and execute the concrete test cases of the test procedure given the tools and expertise available. Furthermore, we also assume that we have a *max effort value* that estimates the total time available for test case refinement and execution. Given these estimate values, we defined a *valid selection* as any set of edges whose effort values sum up to a value that is less than the max effort value. This is precisely defined in the following.

Definition [Valid test procedure selection] Given a function $ef \in E \in \mathbb{R}$ mapping edges to estimated effort values and a real number max denoting the maximum total effort available for testing, we say that a set of edges E is an valid test procedure selection for a risk graph G, denoted $vs[G, E]$, if

$$E \subseteq G.E \wedge \left(\sum_{e \in E} ef(e) \right) \leq max$$

An optimal test procedure selection for a risk graph G is then considered optimal if it is a valid test procedure selection of G and there no other valid selection with a higher priority. This is formally defined in the following.

Definition [Optimal test procedure selection] Given a function $ef \in E \in \mathbb{R}$ mapping edges to estimated effort values and a real number max denoting the maximum total effort available for testing, we say that a set of edges E is an optimal test procedure selection for a risk graph G, denoted $os[G, E]$, if

$$vs[G, E] \wedge \forall E' : vs[G, E'] \Rightarrow p[G, E'] \leq p[G, E]$$

Example 6
In Figure 5 we have illustrated a risk graph with uncertainty and effort estimates. As in the Example 5, the first values that appear after the first semicolon on each edge are understood as uncertainty values. Furthermore, the first values that appear after the second semicolon on each edge are understood as effort value estimates. Thus we have for example that the edge from S1 to S5 has effort

estimate 4, and the edge from S3 to S6 has effort estimate 2. The edges that have no effort estimate are assumed to be excluded from the test procedure selection (we can think of these as having a higher effort estimate than the maximum total effort available).

Assuming the risk structure **R** (defined in Example 4) and the risk value difference function defined in Example 5, and a maximum total effort estimate of 7, and optimal test procedure selection of the risk graph in Figure 5, is shown in Table 2.

Source	Target	Priority
s3	s5	0.10715257739999995
s3	s6	0.07572037696

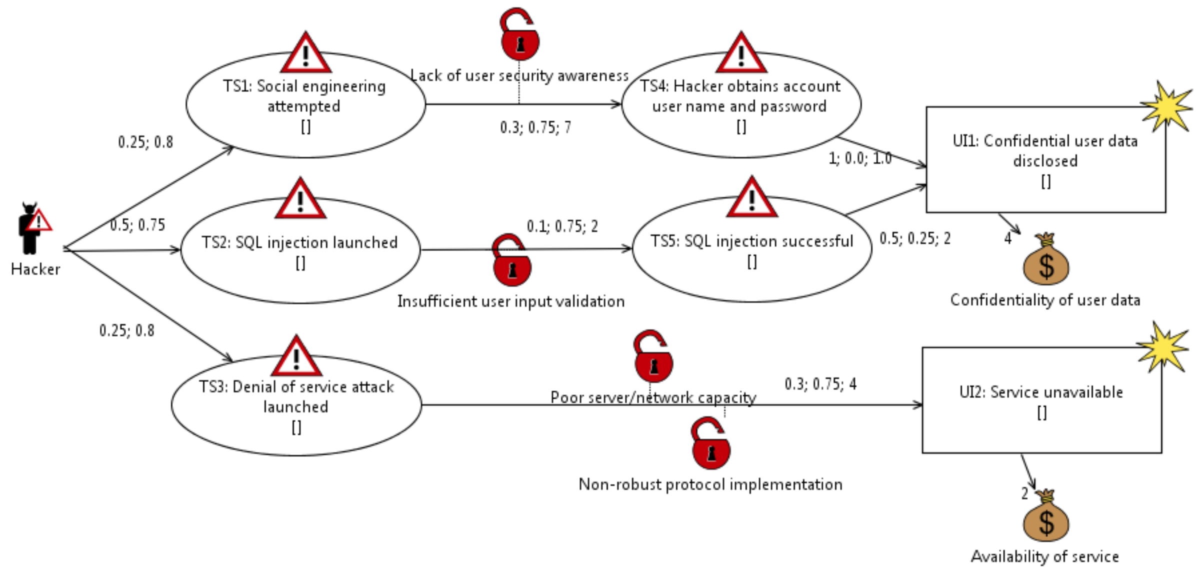
Table 2 – Optimal test procedure selection

2.4 A CORAS Diagram Example

In this section we give a more realistic example of test procedure prioritization and selection than we have seen so far. The basis for the example is the risk graph specified in Figure 6. The figure specified three different attacks that can be performed by a Hacker and that may lead to the unwanted incidents confidential user data disclosed or Service unavailable. These unwanted incidents have consequence values 4 and 2, respectively on a consequence scale from 1 to 5 where 5 is understood as the most severe consequence value. As in the last examples of the previous section, the edges are annotated by labels of the from $l;u;ef$ where l is understood a conditional likelihood estimate, u is understood as an uncertainty estimate, and ef is understood as an estimate of the effort that would be required in order to refine and execute the test procedure represented by an edge. The likelihood and uncertainty values are assumed to be probabilities, whereas the effort estimates are understood to be the number of days it will take to refine and execute the test procedures.

In Figure 6, we see that the uncertainty regarding where a hacker will launch an attack in the first place is fairly large (ranging from 0.8 to 0.75). But we also see that none of the outgoing edges from the hacker are given effort estimates, meaning that they will be excluded from test selection. The reason for this is that it may be difficult to test, using conventional testing means, whether an attack will be launched in the first place. Given that an attack is performed however, it is often possible to test whether the system has any vulnerabilities that can be exploited by the attack. In Figure 5, potential vulnerabilities are indicated by red open locks. Note however, that these do not affect the calculation of the priorities.

Assuming the risk structure **R** defined in Example 4 of Section 2.1, and the function $diff$ defined in Example 5 in 2.2, the test procedure prioritization on the basis of each edge in the risk graph of Figure 6 is shown in Table 3. Here the description of each test procedure correspond the English translation of each edge in the CORAS threat diagram. This translation can be automated.



TS1: Social engineering attempted []
Lack of user security awareness
TS4: Hacker obtains account user name and password []
0.3; 0.75; 7
0.25; 0.8
1; 0.0; 1.0
UI1: Confidential user data disclosed []
Hacker
0.5; 0.75
TS2: SQL injection launched []
0.1; 0.75; 2
TS5: SQL injection successful []
0.5; 0.25; 2
4
Insufficient user input validation
Confidentiality of user data
0.25; 0.8
TS3: Denial of service attack launched []
Poor server/network capacity
0.3; 0.75; 4
UI2: Service unavailable []
Non-robust protocol implementation
2
Availability of service

Figure 6 – Example of a CORAS risk diagram

Test procedure	Effort	Priority
Hacker initiates Social engineering attempted with likelihood 0.25.	N/A	0.702
SQL injection launched leads to SQL injection successful with conditional likelihood 0.1, due to vulnerability Insufficient user input validation.	2 days	0.6243749999999999
Social engineering attempted leads to Hacker obtains account user name and password with conditional likelihood 0.3, due to vulnerability Lack of user security awareness.	7 days	0.5118750000000001
Hacker initiates Denial of service attack launched with likelihood 0.25.	N/A	0.36
Denial of service attack launched leads to Service unavailable with conditional likelihood 0.3, due to vulnerabilities Poor server/network capacity and Non-robust protocol implementation.	4 days	0.26250000000000007
Hacker initiates SQL injection launched with likelihood 0.5.	N / A	0.06937499999999996
SQL injection successful leads to Confidential user data disclosed with conditional likelihood 0.5.	2 days	0.023124999999999995
Hacker obtains account user name and password leads to Confidential user data disclosed with conditional likelihood 1.0.	1 day	0.0

Table 3 – Example of prioritized test procedures

Given that we have a maximum of 7 days available for refining the test procedures into concrete test cases and executing them, an optimal test procedure selection is shown Table 4.

Test procedure	Effort	Priority
SQL injection launched leads to SQL injection successful with conditional likelihood 0.1, due to vulnerability Insufficient user input validation.	2 days	0.6243749999999999
Denial of service attack launched leads to Service unavailable with conditional likelihood 0.3, due to vulnerabilities Poor server/network capacity and Non-robust protocol implementation.	4 days	0.26250000000000007

Table 4 – Example of an optimal test procedure selection given 7 days available

3 Test Purpose Language for Test Pattern Formalization

3.1 Test Pattern Description

Security test patterns are used to describe solutions to various kinds of recurring problems in security testing [33]. Such a pattern is provided in a structured way comprising a set of fields which contain the different kinds of information. For the purpose of risk-based security testing, we adapt this structure. Some fields will be removed because we concentrate on security testing while others will be added or refined.

The following Table 5 shows on the left side the former structure of a security test pattern and on the right side the modification in the context of risk-based security testing and a short rationale.

Original Field	Change	Rationale
Pattern Name	no change	
Context	removed	Content is the same for all patterns in the context of the RASEN project.
Problem/Goal	replaced with CWE-ID and description of weakness	Facilitates mapping between a risk model and suitable security test patterns.
Solution	subdivided into several fields	Allows description of several test design techniques that are suitable for finding in the weakness in question in a systematic way.
Known Uses	removed/moved to solution	This field becomes part of the solution as test design technique.
Discussion	no change	
Related Patterns (optional)	replaced by Generalization of <pattern>	Enables a hierarchy of security test patterns for different purposes.
References (optional)	no change	

Table 5 – Overview of changes to security test pattern structure

In the following, we describe in detail the changes to the different fields and rationales for these changes:

- The field **Context** will be removed. In the context of the RASEN project, we focus on security test patterns that enable to generate security test cases. Therefore, the context of all patterns is restricted to the test pattern kind "behavioral" and the test pattern approach "prevention".
- The field **Problem/Goal** will be replaced with a CWE-ID and a **Weakness Description**. In the context of risk-based security testing, a risk model constitutes the starting point for security testing. The risk model provides a set of vulnerabilities that shall be assessed by means of security testing. In order to facilitate the mapping of vulnerabilities from the risk models to security test patterns that test for these vulnerabilities, such CWE-IDs can be used. Each security test pattern will reference exactly one CWE-ID.
- The field **Solution** will be subdivided into several fields. This is done in order to allow a more systematic description of how certain test design techniques can be employed in order to reveal a weakness and which metrics can be used to measure and assess the actual security testing when performed. This field and its substructure are detailed described in the following

subsection. In order to test for a certain weakness, several design techniques can be applied often. Therefore, the substructure can be contained several times within the field **Solution**, one for each test design technique.

- The field **Known Uses** is removed from the main structure of a test pattern and becomes part of the substructure of the field **Solution**.
- **Related Patterns** will be replaced in order to allow a hierarchy of patterns in terms of generalization by weaknesses and protocols. On one hand, the CWE database has a hierarchy of weaknesses where at the bottom of this hierarchy, weaknesses are more specific than at the top. On the other hand, a pattern can be specialized to a certain protocol in order to give more precise guidelines how to generate for a weakness in the context of a certain protocol. This hierarchy will be detailed in the subsequent subsection.

The following fields are added:

- According to the ISO 29119[1], a test coverage item is an “attribute or combination of attributes that is derived from one or more test conditions by using a test design technique that enables the measurement of the thoroughness of the test execution”. We use an informal **Test Coverage Item Description** of elements of the SUT, e.g. interfaces, as well as elements of the solution, e.g. test data. The test coverage items described by this field may act as measurands for metrics.
- The field **Metrics** provides appropriate test and coverage metrics. These metrics can also be used to specify a test completion criterion. The actual metrics will be developed within the Task T4.3 and presented within the deliverable D4.2.3. For that reason, this field is omitted in the patterns of this deliverable.
- **Test Data** describes how test data can be created or where test data libraries or generators can be obtained from.
- **Tools** references tools that can be used to generate and execute such test cases.

3.1.1 Solution of a Security Test Pattern

Additional to the solution in the form of a step by step-guide (originating either from the DIAMONDS Security Test Pattern Catalogue or from the Common Attack Pattern Enumeration [4] and Classification [8]), the field **Solution** of a pattern contains a substructure for each test design technique that is appropriate for finding the weakness in question. This substructure consists of the following fields:

- The **Test Design Technique** identifies in natural language the technique that is able to find the weakness in question, e.g. data fuzzing. However, how this test design technique should be used in order to generate appropriate test cases is determined by the next field.
- In order to specify how a certain test design technique shall be used, **Test Strategies** specify how to apply a test design technique. However, a first sketch of the concept of test strategies for security testing is given in Section4.
- In order to allow a reasonable selection and prioritizations of test design techniques that shall be used for test case generation, test **Effort** as well as test **Effectiveness** are added. They provide a qualitative estimation of the effort that is necessary to generate and execute test cases and how effective they are in finding a certain weakness. A scale with the values *low*, *medium*, and *high* gives an estimation of test effort and test effectiveness.

This structure provides information that facilitates automatic test case generation from a security test pattern. How test patterns are instantiated for test case generation is described in Section5.

3.1.2 Generalization of Pattern

The benefit of the generalization hierarchy of patterns is twofold. First, it allows to select security test patterns on basis of less specific vulnerability specification from the risk model. The hierarchy allows to performing security tests in a general way as illustrated by the pattern “Improper Input Validation” described below. This very general vulnerability in a risk model allows security testing in several ways. It can be less specific by remaining on this abstract level. However, this is not very efficient. When having more information about the system, e.g. that is uses a database to store and retrieve information or format string function to display information, specializations of this pattern can be used for making the security testing more efficient. Such information can be used to select more specific test patterns (e.g. patterns “

SQL Injection", "Uncontrolled Format String") even if a vulnerability in the risk model is not that specific. Secondly, patterns can be specialized for protocols or architectures. If, for example, a database abstraction layer is used to access a database, this may induce additional vulnerabilities, e.g. by providing an additional query language as in case of Hibernate, the Hibernate Query Language. This example is expressed by a specialization of the pattern "

SQL Injection” by the pattern “SQL Injection through a Database Abstraction Layer”.

3.1.3 Security Test Pattern Template

Pattern Name	*A meaningful name for the pattern, e.g. the name of the weakness.*	
CWE-ID(s)	*The IDs of a weakness from the Common Weakness Enumeration.*	
Weakness Description	*A high-level description of the weakness.*	
Solution	*How the weakness could be revealed manually.*	
	Test Design Technique	*Test design technique that is able to find the weakness.*
	Test Strategies	*Test strategies specific for a certain test design technique that shall be applied in order to generate test cases for the weakness in question.*
	Effort	*The effort to generate and execute such test cases on a scale with the values ‘low’, ‘medium’, and ‘high’-*
	Effectiveness	*How effective is the test design technique in finding such a weakness (how many test cases are necessary to find one weakness, how many weaknesses might be missed).*
Description of Test Coverage Items	*Informal description of items to be covered by test cases created on basis of a pattern.*	
Metrics	*Appropriate test and coverage metrics. These will be developed in Task T4.3. This field is omitted within this deliverable.*	
Discussion	*A short discussion on the pitfalls of applying the pattern and the potential impact it has on test design in general and on other patterns applicable to that same context in particular.*	
Test Data	*Actual or references to test data and test data generators.*	
Tools	*References to tools appropriate for test case generation and execution.*	
Generalization of	*References to other security test patterns that are specializing this pattern.*	
References	*References to OWASP Top 10 weaknesses CWE descriptions, related CAPEC attack patterns*	

Table 6 – Template of a security test pattern

3.2 Security Test Patterns

In order to extend the initial security test pattern catalogue compiled during the DIAMONDS project, we rely on the Top 10 weaknesses from the Open Web Application Security Project [27]. The project collects data from over 500,000 vulnerabilities and thousands of applications in order to determine the ten most important web application security weaknesses. These weaknesses constitute the starting point for the patterns. The OWASP Top 10 weaknesses are already related to weaknesses from the Common Weakness Enumeration CWE [10]. However, not all relevant CWE weaknesses are related to an OWASP Top 10 weaknesses. Therefore, the next step consists in identifying the relevant CWE weaknesses. This task is facilitated by the hierarchical structure of CWE database.

Consider for example the OWASP Top 10 weakness A1-Injection. This OWASP weakness is related by OWASP Top 10 with CWE-77 on Command Injection, CWE-89 on SQL Injection, and CWE-561 on Hibernate Injection. Following the hierarchy of CWE weaknesses, we find the weakness class CWE-20 “Improper Input Validation” that is the counterpart to the OWASP weakness. All children of CWE-20 form different security test patterns along a hierarchy as discussed in Section 3.1.2.

In the next step, each weakness is related to an attack pattern from the CAPEC database [1]. The related attack patterns forms a basis for the field Solution of pattern. Depending on the kind of an attack pattern, its description contains the Attack Execution Flow field that is used for a security test pattern. It is divided into three parts: **Explore**, **Experiment**, and **Exploit**.

The **Explore** part describes where to place an attack on an application. In testing, this is where stimuli are submitted to the system under test. This information is part of the **Test Coverage Item** that needs to be covered by the different test cases created from a security test pattern.

The **Experiment** part describes how to actually perform an attack by stimulating a system. This is actual testing and serves as **Solution** description for manual testing for a weakness.

The **Exploit** part describes how to exploit an actual vulnerability and may provide additional information for the **Solution** or for test data and tools.

This section provides first security test patterns based on the development presented in Section 3.1 and will be developed along the case studies from the RASEN project.

3.2.1 Improper Input Validation

Pattern Name	Improper Input Validation	
CWE-ID(s)	CWE-20	
Weakness Description	The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program.[17]	
Solution	This solution is based on the security test pattern “Detection of Vulnerability to Injection Attacks”. [4] 1. For each of the interfaces and user input fields from the identified test coverage items (see below) create an input element that includes code snippets likely to be interpreted by the SUT. For example, if the SUT is web-based, programming languages and other notations frequently used in that domain (JavaScript, JAVA…) will be used. Similarly, if the SUT involves interaction with a database, notations such as SQL may be used. The additional code snippets should be written in such a way that their interpretation by the SUT would trigger events that could easily be observed (automatically) by the test system. Example of such events include: • visual events: e.g. a pop-up window on the screen • recorded events: e.g. an entry in a logging file or similar • call-back events: e.g. an operation call on an interface provided by the test system, including some details as parameters 2. Use each of the input elements created at step 2 as input on the appropriate SUT interface, and for each of those • check that none of the observable events associated to an interpretation of the injected code is triggered	
	Test Design Technique	Data Fuzzing

	Test Strategies	*all*
	Effort	Low to medium: can be highly automated using fuzzing techniques or injection dictionaries, in particular if a model of the protocol already exists.
	Effectiveness	Low: Without any constraints, any kind of input that could possibly interpreted by the system under test has to be used as stimulus.
Description of Test Coverage Items	• All interfaces of the system under test that get input from the external world, including the kind of data potentially exchanged through those interfaces[4] • User input fields • Injection payloads	
Discussion	The level of test automation for this pattern will mainly depend on the mechanism for submitting input to the SUT and for evaluating potential events triggered by an interpretation of the added probe code.[4]	
Test Data	• Fuzzing library Fuzzino[6]	
Tools	• Fuzzing framework Peach [3] • Fuzzing framework Sulley [7]	
Generalization of	• • SQL Injection • Uncontrolled Format String	
References	• OWASP Top 10 (2013): A1-Injection[30] • CWE-20: Improper Input Validation[17] • CAPEC-152: Injection (Injecting Control Plane content through the Data Plane)[10]	

Table 7 – Security test pattern “Improper Input Validation”

3.2.2 SQL Injection

Pattern Name	SQL Injection	
CWE-ID(s)	CWE-89	
Weakness Description	The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.[18]	
Solution	Based on attack pattern CAPEC-66 [10] 1. Use the application, client or web browser to inject SQL constructs input through text fields or through HTTP GET parameters. 2. Use a possibly modified client application or web application debugging tool such to submit SQL constructs for submitted values or to modify HTTP POST parameters, hidden fields, non-freeform fields, etc. 3. Check for error messages, delays, disclosed values in the client application and new/modified/deleted values in the database.	
	Test Design Technique	Data fuzzing
	Test Strategies	SQL Injection
	Effort	Low to medium: can be highly automated using fuzzing techniques or SQL injection dictionaries.
	Effectiveness	Medium [18] to high, depending on detection capabilities by access to the affected database and to error messages
Description of Test Coverage Items	• Functionality that involves user input, e.g. dialogs, URLs of a web application, that might be used in a database query • User input fields • SQL injection payloads • Names of tables and rows of the database schema • Values of existing records	
Discussion	SQL injection is a task that could be rather trivial but also very complex. This depends on several factors. For instance, error messages resulting from incorrect SQL constructs caused by SQL injection are very helpful in deciding whether SQL injection is generally possible. In order to detect whether table data can be modified, it is helpful to have knowledge of the database management system (different systems have little differences in SQL syntax) and the database schema (modifying existing records may require knowledge in which tables they are stored). If SQL injection is possible, the extent of SQL injection can be assessed by trying to modify existing data which requires knowledge of existing values in the database tables. This enables to determine whether existing database entries can be read, modified or deleted.	
Test Data	• SQL Injection Cheat Sheet[5] • Fuzzing library Fuzzino[6]	
Testing Tools	• Fuzzing framework Sulley[7] • Sqlmap[2]	
Generalization of	SQL Injection through a Database Abstraction Layer	

References	• OWASP Top 10 (2013): A1-Injection[30] • CWE-89: SQL Injection[18] • CAPEC-7: Blind SQL Injection[9] • CAPEC-66: SQL Injection[10] • OWASP Testing Guide: Testing for SQL Injection (OWASP-DV-005)[27] • OWASP: Automated Audit using SQLMap[24]

Table 8 – Security test pattern "Improper Input Validation"

3.2.3 SQL Injection through a Database Abstraction Layer

Pattern Name	SQL Injection through a Database Abstraction Layer	
CWE-ID(s)	CWE-564, CWE-100	
Weakness Description	Using a database abstraction layer to execute a dynamic SQL or abstraction layer-specific statement built with user-controlled input can allow an attacker to modify the statement's meaning or to execute arbitrary SQL or abstraction layer-specific commands.[23]	
Solution	Based on attack patterns CAPEC-66 [10] and CAPEC-109 [13] 1. Use the application, client or web browser to inject SQL constructs or constructs specific to the database abstraction layer input through text fields or through HTTP GET parameters. 2. Use a possibly modified client application or web application debugging tool to submit SQL constructs or constructs specific to the database abstraction layer for submitted values or to modify HTTP POST parameters, hidden fields, non-freeform fields, etc. 3. Check for error messages, delays, disclosed values in the client application and new/modified/deleted values in the database.	
	Test Design Technique	Data Fuzzing
	Test Strategies	SQL Injection
	Effort	Medium to high
	Effectiveness	Medium
Description of Test Coverage Items	• Functionality that involves user input, e.g. dialogs, URLs of a web application, that might be used in a database query • User input fields • Database abstraction layer-specific injection payloads • Names of tables and rows of the database schema • Identifier of one record of each table	
Discussion	Using a database abstraction layer does not necessarily mean to be safe against SQL injections. A database abstraction layer may provide interfaces that can be used to avoid SQL injection vulnerabilities. However, such interfaces have to be used by the developer. Additionally, such a layer may provide its own query language (e.g. Hibernate provides HQL). Using such a query language may induce vulnerabilities to such a query language. Testing for vulnerabilities resulting from the inadequate usage of such a database abstraction layer requires testing for SQL injection vulnerabilities	

	injected through abstraction-layer specific queries. This may require knowledge of the abstraction layer-specific language and how SQL queries are constructed from it.
Test Data	depends on database abstraction layer
Testing Tools	
Generalization of	
References	• OWASP Top 10 (2013): A1-Injection[30] • OWASP Testing Guide: Testing for SQL Injection (OWASP-DV-005)[27] • CWE-564: SQL Injection: Hibernate[23] • OWASP Testing Guide: Testing for ORM Injection (OWASP-DV-007)[26]

Table 9 – Security test pattern "SQL Injection through a Database Abstraction Layer"

3.2.4 Uncontrolled Format String

Pattern Name	Uncontrolled Format String	
CWE-ID(s)	CWE-134	
Weakness Description	The software uses externally-controlled format strings in printf-style functions, which can lead to buffer overflows or data representation problems. [20]	
Solution	Based on attack pattern CAPEC-135 [14]: 1. Inject probe payload which contains formatting characters (%s, %d, %n, etc.) through input parameters. 2. Check if an abnormal message is received (e.g., with a partial dump of the memory) from the application which indicates that the format string was successfully manipulated.	
	Test Design Technique	Data fuzzing
	Test Strategies	Format String
	Effort	Low: can be highly automated using fuzzing techniques and/or format string attack dictionaries.
	Effectiveness	Medium [18] to high, depending on detection capabilities by access to error logs and error messages
Description of Test Coverage Items	• Functionality that involves user input, e.g. dialogs, URLs of a web application, that might be used in a format string function • User input fields, parameters, external variables • Format string attack payloads	
Discussion	An attacker includes formatting characters in a string input field on the target application. Most applications assume that users will provide static text and may respond unpredictably to the presence of formatting character. For example, in certain functions of the C programming languages such as printf, the formatting character %s will print the contents of a memory location expecting this location to identify a string and the formatting character %n prints the number of DWORD written in the memory.[14]	
Test Data	• Fuzzing library Fuzzino[6]	
Tools	• Fuzzing framework Sulley[7]	
Generalization of		
References	• OWASP Top 10 (2013): A1-Injection[30] • CWE-134: Uncontrolled Format String[20] • CAPEC-67: String Format Overflow in syslog()[11] • CAPEC-135: Format String Injection[14] • OWASP Testing Guide: Testing for Format String[28]	

Table 10 – Security test pattern “Uncontrolled Format String”

3.2.5 Reflection Attack Vulnerability in an Authentication Protocol

Pattern Name	Reflection Attack Vulnerability in an Authentication Protocol	
CWE-ID(s)	CWE-301	
Weakness Description	Simple authentication protocols are subject to reflection attacks if a malicious user can use the target machine to impersonate a trusted user.[21]	
Solution	Based on attack pattern CAPEC-90 [12]: 1. Open a connection to the target server and send it a challenge. 2. The server responds by returning the challenge encrypted with a shared secret as well as its own challenge to the attacker. Record the challenge from the server. 3. Initiate a second connection to the server and send it, as challenge, the challenge received from the server on the first connection. 4. The server treats this as just another handshake and responds by encrypting the challenge and issuing its own. 5. Record the encrypted challenge on the second connection and send it as response to the server on the first connection. 6. Check if authentication is successful on the first connection is successful.	
	Test Design Technique	
	Test Strategies	
	Effort	Low
	Effectiveness	High
Description of Test Coverage Items	• interfaces that provide a challenge-response-based authentication mechanism	
Discussion	The attack is possible if a challenge-response authentication is provided symmetrically and the server can be used to provide the response to its own challenge. In order to perform such a test, knowledge of the exchanged messages is required.	
Test Data		
Tools		
Generalization of		
References	• OWASP Top 10 (2013): A2-Broken Authentication and Session Management[31] • CWE-301: Reflection Attack in an Authentication Protocol[21]	

Table 11 – Security test pattern “Reflection Attack Vulnerability in an Authentication Protocol”

3.2.6 Missing Authentication for Critical Function

Pattern Name	Missing Authentication for Critical Function	
CWE-ID(s)	CWE-306	
Weakness Description	The software does not perform any authentication for functionality that requires a provable user identity.[22]	
Solution	1. Try to access each function that requires authentication. 2. Perform an authentication with valid credentials. 3. Perform a logout.	
	Test Design Technique	Behavioral Fuzzing
	Test Strategies	Remove Message: authentication message(s)
	Effort	Low
	Effectiveness	High
Description of Test Coverage Items	• Interfaces that provide functions • Authentication messages • Functions that require authentication	
Discussion	Missing access control on function level can be exploited if authentication is performed on client-side but not on server-side or if it is just missing.	
Test Data		
Tools		
Generalization of		
References	• OWASP Top 10 (2013): A7-Missing Function Level Access Control[32] • CWE-306: Missing Authentication for Critical Function[22]	

Table 12 – Security test pattern "Missing Authentication for Critical Function"

3.3 Introduction to Test Purpose Language

Within model-based testing, we propose to use a dedicated language, called Test Purpose language, in order to represent a test objective aiming to test a security property. This test objective is used by the test generator to drive the test cases generation. This language should also be close from natural language, in order to make it possible to understand the test objective easily, without prior test purpose language knowledge.

3.3.1 Test Purpose Language

A test purpose is a high-level expression that formalizes a test intention linked to a testing objective to drive the automated test generation on the behavioral model. In the RASEN context, we propose to use test purposes to formalize security test patterns. The test purpose language we are presenting is called Smartesting Test Purpose Language. This is a textual language based on regular expressions, allowing the formalization of security test intention in terms of states to be reach and operations to be called. The next subsections introduce this language.

3.3.1.1 Grammar

The Test Purpose Language is based on an ANTLR v3 grammar (see http://www.antlr.org/ for more details about this parser generator language). ANTLR is used to generate the lexer, and the parser needed for the various exploitations of the language is performed by Smartesting tools.

The Test Purpose language grammar, respecting the previous constraints, is presented here:

```
scheme
        : (quantifier_list COMMA)? seq EOF;

quantifier_list
        : quantifier (COMMA quantifier)*;

quantifier
        : FOR_EACH BEHAVIOR var FROM behaviour_choice
        | FOR_EACH OPERATION var FROM op_choice
        | FOR_EACH LITERAL var FROM literal_choice
        | FOR_EACH INSTANCE var FROM instance_choice
        | FOR_EACH INTEGER var FROM integer_choice
        | FOR_EACH CALL var FROM call_choice;

op_choice
        : ANY_OPERATION
        | ANY_OPERATION_BUT op_list
        | op_list;

call_choice
        : call_list;

behaviour_choice
        : ANY_BEHAVIOR_TO_COVER
        | ANY_BEHAVIOR_TO_COVER_BUT behaviour_list
        | behaviour_list;

literal_choice
        : IDENTIFIER (OR IDENTIFIER)*
        | keyword;

instance_choice
        : instance (OR instance)*
        | state
        | keyword;

integer_choice
        : CURLY_OPEN INT (COMMA INT)+ CURLY_CLOSE
        | keyword;

var
        : DOLLAR IDENTIFIER;

keyword
        : SHARP IDENTIFIER;

state
        : ocl_constraint ON_INSTANCE instance
        | keyword;

instance
        : IDENTIFIER;

ocl_constraint
        : STRING_LITERAL;

seq
        : bloc (THEN bloc)*;

block
        : USE control restriction? target?;

restriction
        : AT_LEAST_ONCE
        | ANY_NUMBER_OF_TIMES
        | INT TIMES
        | var TIMES;

target
        : TO_REACH state
        | TO_ACTIVATE behaviour
        | TO_ACTIVATE var;

control
        : op_choice
        | behaviour_choice
        | var
        | call_choice;

call_list
        : call (OR call)*
        | keyword;

op_list
        : operation (OR operation)*
        | keyword;

operation
        : IDENTIFIER;


call
        : instance '.' operation parameters;

parameters
        : PARENTHESIS_OPEN (parameter (COMMA parameter)*)? PARENTHESIS_CLOSE;

parameter
        : FREE_VALUE
        | IDENTIFIER
        | var
        | INT;
```

```
behaviour_list
            : behaviour (OR behaviour)*
            | keyword;

behaviour
            : BEHAVIOR_WITH_TAGS tag_list
            | BEHAVIOR_WITHOUT_TAGS tag_list;

tag_list
            : CURLY_OPEN tag (COMMA tag)* CURLY_CLOSE;

tag
            : REQ COLON IDENTIFIER
            | AIM COLON IDENTIFIER;

TIMES                             : 'times' ;
FOR_EACH                          : 'for_each' ;
BEHAVIOR                          : 'behavior' ;
OPERATION                         : 'operation' ;
INTEGER                           : 'integer' ;
CALL                              : 'call' ;
INSTANCE                          : 'instance' ;
LITERAL                           : 'literal' ;
FROM                              : 'from' ;
THEN                              : 'then' ;
USE                               : 'use' ;
TO_REACH                          : 'to_reach' ;
TO_ACTIVATE                       : 'to_activate' ;
ON_INSTANCE                       : 'on_instance' ;
ANY_OPERATION                     : 'any_operation' ;
ANY_OPERATION_BUT                 : 'any_operation_but' ;
OR                                : 'or' ;
ANY_BEHAVIOR_TO_COVER             : 'any_behavior_to_cover' ;
ANY_BEHAVIOR_TO_COVER_BUT         : 'any_behavior_to_cover_but' ;
BEHAVIOR_WITH_TAGS                : 'behavior_with_tags' ;
BEHAVIOR_WITHOUT_TAGS             : 'behavior_without_tags' ;
AT_LEAST_ONCE                     : 'at_least_once' ;
ANY_NUMBER_OF_TIMES               : 'any_number_of_times' ;
COMMA                                   : ',' ;
CURLY_OPEN                        : '{' ;
CURLY_CLOSE                       : '}' ;
BRACKET_OPEN                      : '[' ;
BRACKET_CLOSE                     : ']' ;
PARENTHESIS_OPEN                  : '(' ;
PARENTHESIS_CLOSE                 : ')' ;
COLON                             : ':' ;
DOLLAR                            : '$' ;
SHARP                             : '#' ;
REQ                               : 'REQ' ;
AIM                               : 'AIM' ;
FREE_VALUE                        : '_' ;
DOT                               : '.' ;
DOUBLE_DOT                        : '..';
fragment DIGIT                    : '0'..'9' ;
fragment IDENTIFIER_FIRST         : 'a'..'z' | 'A'..'Z' | '_' ;
fragment IDENTIFIER_BODY          : 'a'..'z' | 'A'..'Z' | DIGIT | '_' | '/';
IDENTIFIER                        : IDENTIFIER_FIRST IDENTIFIER_BODY* ;

INT
    : DIGIT+;

STRING_LITERAL
    :  '"'(~('\\'|'"'))*'"';

WHITESPACE: (' ' | '\t' | '\r' | '\n')+ { skip(); };

COMMENT
    : '/*' .* '*/' {$channel=HIDDEN;};

LINE_COMMENT
    : '//' ~('\n'|'\r')* '\r'? '\n' {$channel=HIDDEN;};
```

3.3.1.2 Semantics and Samples

The main idea consists in testing the same behavior in several contexts, and several behaviors in the same context. The language should enable to create several test objectives from a unique expression. To ease the understanding of the Test Purpose Language expressions, the operators of the Test Purpose Language are printed in **bold purple**.

3.3.1.2.1 Stage Creation

A test purpose Stage can contain the following parts:

The « CONTROL » part defining the system control points that can be used during the stage.

The « RESTRICTION » part constraining the number of system control points that can compose the current stage. This part is optional.

The « TARGET » part defining the objective that must be reached at the end of the stage. This one is also optional.

The Stage is always composed as follows:

use CONTROL RESTRICTION TARGET

Between each Stage, the operator **then** has a separator role.

3.3.1.2.1.1 The « CONTROL » Part

Several types of system control points can be used. Several ways exist to select each of those system control points:

- An operation chosen from a list of operations of the system:
  - **any_operation** (any system operation can be used during this stage)
  - **operation1** (only the named operation can be used during the stage)
  - **operation1 or operation2 or...** (only the named operations can be used during the stage)
  - **any_operation_but operation1 or operation2 or...** (the names operations cannot be used during the stage)

- A behavior from a list of behaviors of the system:
  - **any_behavior_to_cover** (any behavior of the system can be used during this stage)

- ○ **behavior_with_tags {REQ: req1, AIM : aim1}** (only the behaviors of the system covering at least the specified tags can be used during this stage)
  - ○ **behavior_without_tags {REQ: req1, AIM : aim1}** (only the behaviors of the system not covering the specified tags can be used during this stage)
  - ○ in the same way as for the operations, a list of behaviors separated by**or**can be specified.
  - ○ **any_behavior_but** beh1 **or** beh2 **...** (the specified behaviors cannot be used for this stage)

- An operation call can also be specified:
  - ○ **instance.operation1(value1, _)** (only the names operation, from the specified instance can be called and its first parameter must take the value “value1”, the second parameter can take any value.)
  - ○ in the same way as the operations, a list of calls separated by **or** can be specified.

3.3.1.2.1.2 The « RESTRICTION » part

The length of each stage in terms of control point numbers can be set. This is optional and is expressed as follows:

- **any_number_of_times** (the stage is optional, and can contain any number of control points)
- **at_least_once** (the stage must be composed of at least one control point)
- **i times** (the stage must use exactly i control points)
- if no restriction part is specified, it means that the stage is mandatory, and that is should contain exactly one control point.

3.3.1.2.1.3 The « TARGET » Part

The TARGET part expresses a condition, which must be respected at the end of the current stage. It can be a state of the system under test, or a behavior that must be activated by the last control point of the stage.

3.3.1.2.1.3.1 Behavior TARGET

In order to specify a behavior that must be activated, the « **to_activate** » keyword is used, followed by a specified behavior. It is specified by the tags it covers, or it must not cover, as follows:

use any_operation **to_activate behavior_with_tags {REQ: req1, AIM : aim1}**
use any_operation **to_activate behavior_without_tags {REQ: req1, AIM : aim1}**

3.3.1.2.1.3.2 State TARGET

To define a state of the system that must be reached at the end of a stage, it can be introduced by the « **to_reach** » keyword, followed by an OCL constraint on a specified model instance, as follows:

use any_operation **to_reach** ‘’self.status = ALL_STATUS ::OK’ ’**on_instance** *sut*

3.3.1.2.2 Iterator Creation

In order to create various contexts where the behavior can be activated, or to call several behaviors in a specific context, the language allows to create iterators. This enables to create several test objectives from the same Test Purpose expression.

Iterators are separated by a comma, and are expressed as follows:

for_each TYPE $Varname **from** VALEURS

Several kinds of iterators can be created:

- operation
- behavior
- call
- literal
- instance
- integer

Following are described samples for each of those iterators.

3.3.1.2.2.1 Iterate Over Operations

Iterate over each operations of the system under test:

for_each operation $OP **from** any_operation

Iterate over a list of operations:

for_each operation $OP **from** operation1 **or** operation 2 **or** ...

Iterate over operations of the system under test not in a list:

for_each operation $OP **from** any_operation_but operation1 **or** operation2 **or** ...

3.3.1.2.2.2 Iterate Over Behaviors

Iterate over each behavior to cover from the test suite:

for_each behavior $CPT **from** any_behavior_to_cover

Iterate over a dedicated list of behaviors:

for_each behavior $CPT **from** behavior_with_tags {AIM :a1, REQ : r1} **or** behavior_without_tags {REQ : r1}...

Iterate over each behavior to cover from a suite not in a list:

for_each behavior $CPT **from** any_behavior_to_cover_butbehavior_with_tags {AIM :a1, REQ : r1} **or** behavior_without_tags {REQ : r1}...

3.3.1.2.2.3 Iterate Over Calls

Iterate over a list of calls:

for_each call $CALL **from** inst1.op(_, Value1) **or** inst2.op1() **or** ...

3.3.1.2.2.4 Iterate Over Enumeration Literals

Iterate over each specified enumeration literals (all from the same enumeration):

for_each literal $LIT **from** value1 **or** value2 **or** ...

3.3.1.2.2.5 Iterate OverInstances

Iterate over each specified instances (all from the same class):

for_each instance $Inst **from** instance1 **or** instance2 **or** ...

For the RASEN project, an extension of the language has been developed. This allowsspecifying a set of instances, using OCL code. This code is evaluated regarding the initial state of the system, and gives the set of instances on which to iterate.

for_each instance $Inst **from** ‘’self.users->select(u : User | u.admin = true)’’ **on_instance** *sut*

3.3.1.2.2.6 Iterate Over Integers

Iterate over a specified list of integers:

for_each integer $I **from** {1, 3, 7}

3.3.1.2.3 Variable Usage

Variables defined by iterators can be used in the Test Purpose stages. If a variable is used several times on a Test Purpose, in the corresponding test objectives the variable value must be the same for each of its usage.

3.3.1.2.3.1 Operation Variable

It can be used in the CONTROLpart as follows:

use $OP **to_reach** ‘’self.status = ALL_STATUS ::OK’ ’**on_instance** *sut*

3.3.1.2.3.2 Behavior Variable

It can be used in the CONTROL part as follows:

use $CPT **to_reach** ‘’self.status = ALL_STATUS ::OK’’ **on_instance** *sut*

Also, it can be used in the TARGETpart as follows:

use any_operation **to_activate** $CPT

3.3.1.2.3.3 Call Variable

It can be used in the CONTROL part as follows:

use $CALL **to_reach** ‘’self.status = ALL_STATUS ::OK’’ **on_instance** *sut*

3.3.1.2.3.4 Enumeration Literal Variable

It can be used in an OCL state definition as follows:

use any_operation **to_reach** ‘’self.status = ALL_STATUS::$LIT" **on_instance** *sut*

It can also enable to set a call parameter in the CONTROLpart as follows:

use inst1.op1(_, $LIT) **to_reach** ‘’self.status = ALL_STATUS ::OK’’ **on_instance** *sut*

3.3.1.2.3.5 Instance Variable

It can be used in the TARGET part, as the instance on which the state must be respected:

> **use** any_operation **to_reach** ''self.status = ALL_STATUS::OK" **on_instance** *$Inst*

It can also enable to set a call parameter in the CONTROL part as follows:

> **use** $Inst.op1(_, Value1) **to_reach** ''self.status = ALL_STATUS ::OK'' **on_instance** *sut*

3.3.1.2.3.6 Integer Variable

It can enable to set a call parameter in the CONTROLpart as follows:

> **use** inst1.op1(_, $I) **to_reach** ''self.status = ALL_STATUS ::OK'' **on_instance** *sut*

It can also be used RESTRICTION part as follows:

> **use** any_operation $I times **to_reach** ''self.status = ALL_STATUS ::OK'' **on_instance** *sut*

3.3.2 Derivation of Test Objective from Test Purpose Definition

A Test Purpose is dedicated to the production of one or several test objectives. Test purposes are thus automatically transformed into test objectives, a test objective being a sequence of intermediate objectives that will be used by the test generator (Smartesting CertifyIt) to produce test cases. To address that, the sequence of stages of a test purpose is mapped to a sequence of intermediate objectives of a test target. Furthermore, this computation unfolds the combination of values between the iterators of the test purposes, such that one test purpose produces as many test objectives as possible combinations.

3.4 Extension of Test Purpose for Security Testing

During the RASEN project, several Smartesting CertifyIt extensions have to be developed. They are used at different levels of the expected RASEN overall process, as shown in the Figure 7.

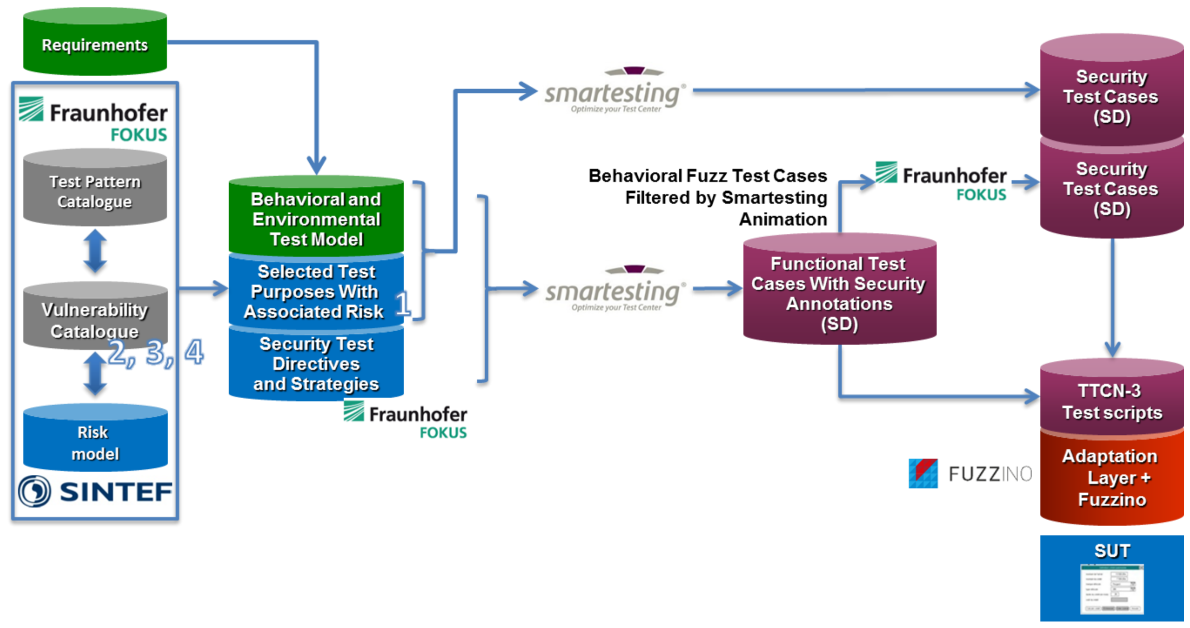

Requirements
Fraunhofer FOKUS
Test Pattern Catalogue
Vulnerability Catalogue
2, 3, 4
Risk model
SINTEF
Behavioral and Environmental Test Model
Selected Test Purposes With Associated Risk
1
Security Test Directives and Strategies
Fraunhofer FOKUS
smartesting
smartesting
Behavioral Fuzz Test Cases Filtered by Smartesting Animation
Fraunhofer FOKUS
Functional Test Cases With Security Annotations (SD)
Security Test Cases (SD)
Security Test Cases (SD)
TTCN-3 Test scripts
Adaptation Layer + Fuzzino
FUZZINO
SUT

Figure 7 – RASEN overall process

1. Test Purpose language extension, enabling sets creation evaluated on the initial state of the system.
2. Keyword creation to be used by Test Purposes. This mechanism enables to create generic Test Purposes, and to help for maintenance and reuse.
3. Capability to link a Test Purpose to a requirement identifier to ensure the traceability through the all test generation process.
4. Test Purpose catalogue import/export to reuse and apply Test Purposes on several systems under test.
5. Generation and Animation standalone Java API to enable fuzzed test sequences validation regarding the model.

Those features are detailed below.

3.4.1 Test Purpose Language Extension

In order to create security tests, we can need to iterate over instances of a class. The original Test Purpose language version only allows iterating upon a set of instances specified by their name. This mechanism makes it difficult to keep the Test Purpose generic. It is why we need to extend it to be able to select instances verifying a specific constraint. This selection is expressed by an OCL constraint and performed using an OCL constraint evaluation. This feature enables to automatically iterate over all the instances of a class, or restrict the iteration to only a set of instances respecting a condition (an attribute value, etc.). This is detailed in Section 3.3.1.2.2.5.

3.4.2 Keywords for Test Purposes

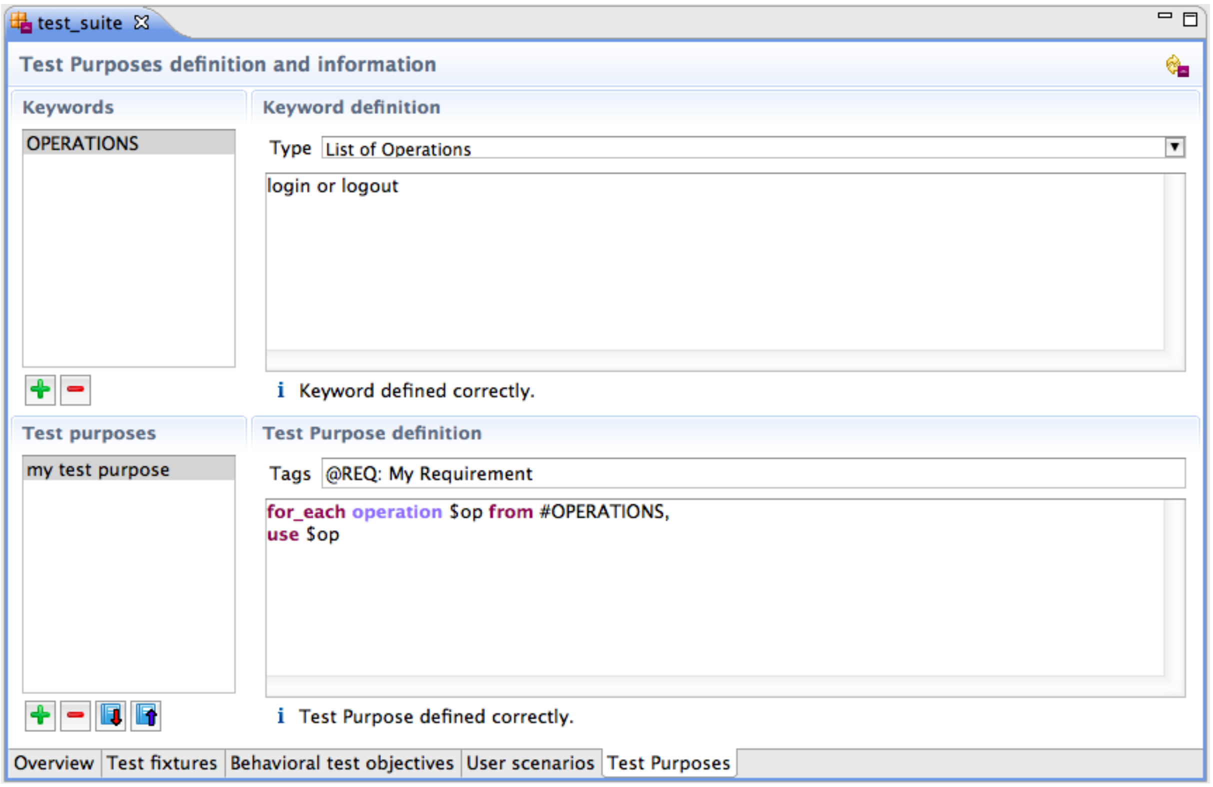
A security test objective aims to be generic, and so it can be applied on several models to generate test sequences. But a Test Purpose contains some information coming directly from the current model, which makes it reliant on it. To avoid this dependence, Keywords mechanism has been developed to ensure the test purposes to be generic. The idea is to use specific arguments, called keywords, in the

Test Purposes to represent generic artefacts of a model. For each model, engineers have only to link keywords with the specific element of the current model.

The keyword can represent and be linked to the following information coming from the model:

- A list of behaviours
- A list of calls
- A list of instances
- A list of integers
- A list of literals
- A list of operations
- A state regarding a specific instance of the model

The keywords can be used in the Test Purpose definition to replace any of this model information preceded by "#" as shown in Figure 8.



OPERATIONS

Type: List of Operations

login or logout

Keyword defined correctly.

my test purpose

Tags: @REQ: My Requirement

```
for_each operation $op from #OPERATIONS,
use $op
```

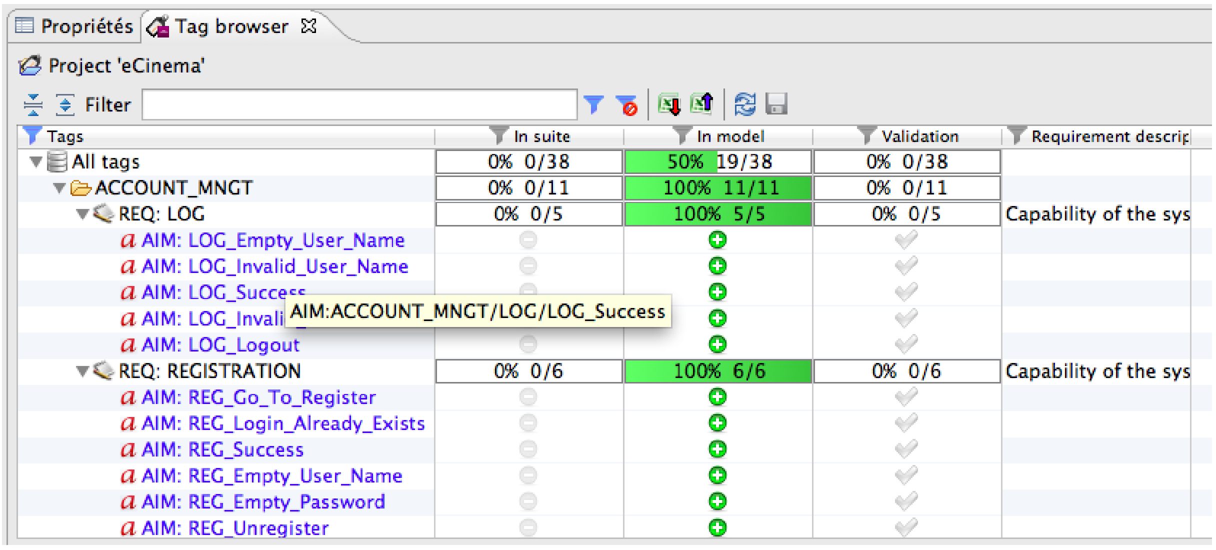
Test Purpose defined correctly.

Figure 8 – Keywords definition for Test Purposes

This mechanism allows to factor common model information to be re-used in several Test Purposes. A Test Purpose can be re-used as it in several test projects, the keyword definition making it adapted to the project model. It should also be noted that the use of keywords in the Test Purpose language, instead of concrete model element, allow to produce expressions that are both powerful (for the test generator) and easy to read (for the engineer).

3.4.3 Test Purpose to Requirement Traceability

Smartesting CertifyIt already makes it possible to manage the requirement traceability between the functional requirement and the associated generated test cases. To achieve this traceability, Smartesting approach proposes to tag the behavioural model with the requirement identifiers formalized using the ad-hoc tags "REQ" and "AIM". Such tags enable to associate a specific requirement with a model artefact, and to measure its coverage by the set of generated test cases. This approach makes it possible to automatically produce the traceability matrix at the same time as the generated test cases and the corresponding coverage rate. The Smartesting CertifyIt Tag browser, depicted in Figure 9, makes it possible to view the requirement coverage.


Propriétés | Tag browser

Project 'eCinema'

Filter

Tags	In suite	In model	Validation	Requirement descrip
All tags	0% 0/38	50% 19/38	0% 0/38	
ACCOUNT_MNGT	0% 0/11	100% 11/11	0% 0/11	
REQ: LOG	0% 0/5	100% 5/5	0% 0/5	Capability of the sys
AIM: LOG_Empty_User_Name	⊖	⊕	✓	
AIM: LOG_Invalid_User_Name	⊖	⊕	✓	
AIM: LOG_Success	⊖	⊕	✓	
AIM: LOG_Invali		⊕	✓	
AIM: LOG_Logout	⊖	⊕	✓	
REQ: REGISTRATION	0% 0/6	100% 6/6	0% 0/6	Capability of the sys
AIM: REG_Go_To_Register	⊖	⊕	✓	
AIM: REG_Login_Already_Exists	⊖	⊕	✓	
AIM: REG_Success	⊖	⊕	✓	
AIM: REG_Empty_User_Name	⊖	⊕	✓	
AIM: REG_Empty_Password	⊖	⊕	✓	
AIM: REG_Unregister	⊖	⊕	✓	

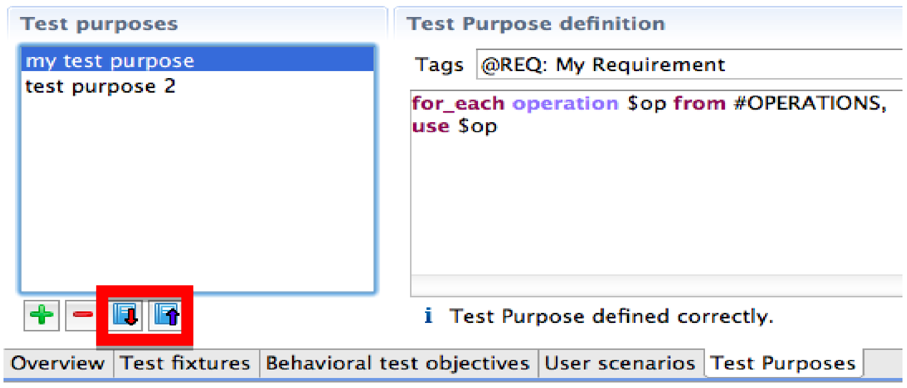
AIM:ACCOUNT_MNGT/LOG/LOG_Success

Figure 9 – Smartesting Tag browser

Within RASEN project, we propose to extend this mechanism by offering to link a requirement to a Test Purpose (as shown in the field "Tags" in the Test Purpose window of Figure 8). This mechanism allows the traceability link from a requirement specification to the corresponding generated test cases. The Smartesting Tag browser could then be used to view the requirement coverage regarding the test objectives coming from the security Test Purposes managed in the RASEN project.

3.4.4 Test Purpose Catalogue Import/Export

To re-apply a set of Test Purposes on different systems, the keyword mechanism allows creating generic Test Purposes. In addition to that, an import/export mechanism has been implemented during the RASEN project.



```
for_each operation $op from #OPERATIONS,
use $op
```

Test Purpose defined correctly.

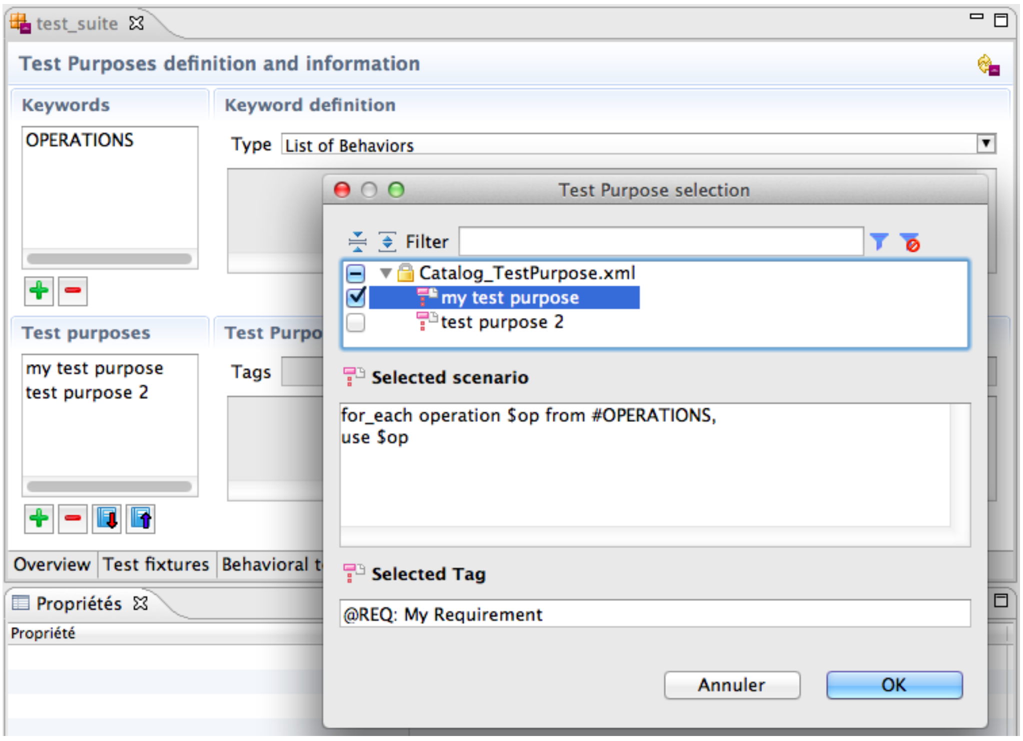
Figure 10 – Test Purpose import/export buttons

It exports the Test Purposes as an XML file, containing all the necessary information to import it later in another project. The xls file is constructed as follow:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <Category name="Catalog_TestPurpose.xml">
    <testpurpose name="my test purpose" scenario="for_each operation $op from #
        OPERATIONS,&#xA;use $op" tag="@REQ: My Requirement" />
    <testpurpose name="test purpose 2" scenario="for_each operation $op from #
        OPERATIONS,&#xA;use $op" tag="@REQ: seond one" />
  </Category>
</catalog>
```

Figure 11 – Test Purpose catalogue format

The import feature allows choosing the Test Purpose that must be imported as shown on the figure below:



Keywords: OPERATIONS

Type: List of Behaviors

Test purposes:
my test purpose
test purpose 2

Tags

Catalog_TestPurpose.xml
[✓] my test purpose
[ ] test purpose 2

Selected scenario

```
for_each operation $op from #OPERATIONS,
use $op
```

Selected Tag

@REQ: My Requirement

Figure 12 – Test Purpose import feature

The keyword feature, in addition to the import/export feature enables creating generic Test Purpose catalogues.

3.4.5 Standalone Animation API

Having generated test sequences from Test Purposes with the Smartesting CertifyIt tool, the test sequences can be used as an input for the fuzzing tools provided by Fraunhofer FOKUS. After being fuzzed, only the sequences that could make the system fail should be executed on the system under test.

A way to filter the test sequences consists in replaying them on the test model. A JAVA standalone API has been developed to reuse the Smartesting Animation engine. The test model can be loaded with this API, and the test animation regarding this model gives the information about the correctness of the sequence.

3.5 Formalization of Test Patterns with Test Purpose Language

A test pattern is the expression of the essence of a well-understood solution to a recurring software testing problem. It can be represented as a table, containing informal information about the problem.

Here is an example of a test pattern concerning the Multi-step XSS vulnerability. The Test Purpose language can be used to formalize the way to test it. In this example, it has to be added in the “Solution” part of the test pattern.

Pattern name	Multi-step XSS
Context	N/A
Problem/Goal	This pattern can be used on an application which doesn't check user inputs. An XSS attack can redirect users to a malicious site, or can steal user's private information (cookies, session, ...). The objective consists in Detecting if a user input can embed malicious datum enabling an XSS attack.
Solution	Identify a sensible user input field, inject the untrusted payload <script>alert(xss)</script> Observation: Go to a page echoing the user input, check if a message box 'xss' appears. Associated Test Purpose : **for_each** literal $param**from** #DATA_SENSIBLE_TO_XSS, **for_each** literal $page **from** #PAGES_SENSIBLE_TO_XSS, **use** **any_operation** any_number_of_times **to_reach** #PAGE_WITH_SENSIBLE_DATA_INPUT **then** **use** threat.injectXSS($param) **then** **use** **any_operation** any_number_of_times **to_reach** #PAGE_WITH_SENSIBLE_DATA_OUTPUT($page, $param) **then** **use** threat.checkXSS()
Known uses	Web Application Firewalls (WAF) filter messages sent to the server(blacklist, clacregEx, ...) ; variants allow to overcomethesefilters
Discussion	
Related patterns	Stored XSS
References	CAPEC: http://capec.mitre.org/data/definitions/86.html WASC: http://projects.Webappsec.org/w/page/13246920/CrossSiteScripting OWASP: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

Table 13 – Test pattern with Test Purpose definition

4 Security Test Strategies

Security test patterns specify in a general way how a recurring security problem can be solved. In the context of the RASEN project, this is restricted to security testing. However, the semi-formal way does not allow automatic test case generation and execution. For a significant number of patterns, there already exist test design techniques that are appropriate for the weaknesses in question. These techniques can be guided to narrow the scope of test cases created using a test design technique to a weakness. This reduces the number of test cases and thus, improves the efficiency when using it. In the context of security testing, the goal is usually to discover a vulnerability or to ensure that it does not exist.

A test strategy for security testing provides the information necessary for generating corresponding test cases. Test strategies are generic in the context of a certain test design technique. They specify a certain strategy that can be implemented by test case generators that support this test strategy. A test strategy is represented in a UML model by a stereotype. It is applied to an element the test strategy shall be applied to.

For data and behavioral fuzzing, we define security test strategies based on fuzzing heuristics provided the behavioral fuzz test generator and the data fuzzing library Fuzzino. Table 14 depicts some examples of test strategies.

Test Design Technique	Strategy	Description
Data Fuzzing	SQL Injection	generate test data that is able to discover SQL injection vulnerabilities
	Format String	generate test data that is able to discover format string vulnerabilities
Behavioral Fuzzing	Remove Message	remove a message from a sequence diagram
	Move Message	move a message to another position and/or sequence diagram, applied to a message within an Interaction

Table 14 – Test Strategies for Data and Behavioral Fuzzing (Examples)

Depending on the test strategy, it can require more information to specify how it shall be used for test case generation and to avoid test cases that are less capable in finding a vulnerability. This information can be appended to a test strategy using attributes of the corresponding stereotype. How such information can be specified is depicted in Table 15.

Strategy	**Attributes**	**Description**
SQL Injection	dbms : String	e.g. MySQL, MS-SQL, PostGreSQL, ...
	tables : String[*]	name of different tables within the database
	fields : String[*]	name of fields within tables
	ids : Integer[*]	IDs of existing records
Format String	*no further attributes*	
Remove Message	*no further attributes*	
Move Message	targetPosition : Message[*]	to which message shall a message be moved
	before : Boolean[*]	shall the message be moved before or after *targetPosition*

Table 15 – Attributes of Different Test Strategies (Examples)

By applying a test strategy to an element of a model, the information how test cases shall be generated from the model is carried by the model. The model is the only artifact that is required for test case generation. Test strategies can be applied automatically when a security test pattern is instantiated. The information for the attributes of a strategy has to be provided manually if it is not contained in the model. In case of standard protocols, the required information can be automatically obtained by a specialization of a pattern for this protocol that specified the required information. However, if no standard protocols are used or if some information, e.g. the used database management system, is not contained within a model, the corresponding values for the attributes of a test strategy has to be assigned manually.

5 Instantiating Test Patterns for Test Case Generation

5.1 Test Sequence Generation

5.1.1 Test Objectives Creation Strategy

In order to create test objectives, the strategy consists of creating all possible combinations (Cartesian product) from the values defined in the iterators, and to instantiate the stages part of the Test Purpose.

For example, a Test Purpose containing an iterator over 3 operations, another over 3 behaviors and one over 2 instances will produce 3*3*2 = 18 test objectives.

5.1.2 Test Objectives

Regarding the following Test Purpose:

```
for_each operation $OP from operation1 or operation 2,
for_each behavior $CPT from behavior_with_tags {AIM :a1, REQ : r1} or
          behavior_without_tags {REQ : r2},
use $OP to_reach "self.status = ALL_STATUS::OK" on_instance sut then
use any_operationany_number_of_times to_activate $CPT
```

The following test objectives will be produced:

Objective 1:

```
use operation1 to_reach "self.status = ALL_STATUS::OK" on_instancesutthen
useany_operationany_number_of_timesto_activatebehavior_with_tags {AIM :a1, REQ : r1}
```

Objective 2:

```
use operation2 to_reach "self.status = ALL_STATUS::OK" on_instance sut then
use any_operationany_number_of_times to_activate behavior_with_tags {AIM :a1, REQ : r1}
```

Objective 3:

```
use operation1 to_reach "self.status = ALL_STATUS::OK" on_instance sut then
use any_operation any_number_of_times to_activate behavior_without_tags {REQ : r2}
```

Objective 4:

```
use operation2 to_reach "self.status = ALL_STATUS::OK" on_instance sut then
use any_operation any_number_of_times to_activate behavior_without_tags {REQ : r2}
```

For each of those objectives, the Smartesting CertifyIt generation engine will produce, if reachable, a test.

5.2 Security Test Case Generation from Annotated Test Sequences

Test purposes can be employed to generate test sequences from a system model that cover the test coverage items described in a security test pattern. Referring to the test coverage items by a test purpose ensures that the resulting test sequences contain them. As discussed in Section4, test strategies are used to guide a test case generator by applying them to model elements that are relevant for test case generation. Based on these sequences annotated with security test strategies, actual security test cases can be generated. This process is different for data and behavioral fuzzing. In case of behavioral fuzzing, strategies are used for test case generation by applying the corresponding behavioral fuzzing operators to the already generated test sequences in form of UML sequence diagrams. This results in a set of sequence diagrams that are behavioral fuzz test cases.

In case of data fuzzing, all the test cases do not differ in the messages exchanged with the SUT but only in the values for arguments of these messages. Only a few arguments of these messages contain fuzz test data. If for each of the different fuzz test data to be used to stimulate the SUT, this would result in a large number of test cases that differ merely in these fuzz test data. Hence, all the test cases have much in common. This blows up the model that can be avoided by specifying the basic message sequence and where fuzz test data shall be inserted at test execution time by using security test strategies. Thus, the model is kept clean, and at test execution time this message sequence is submitted to the SUT. In each iteration, different fuzz test data is used for the message arguments marked to carry fuzz test data using security test strategies. The fuzz test data are obtained from the fuzz test data generation Fuzzino [6]. Fuzzino determines the fuzz test data to generate by evaluating the security test strategies applied to a message arguments, the type description of the message argument and possibly valid values if provided.

6 Summary

RASEN WP4 addresses compositional security testing guided by risk assessment. This deliverable presents first techniques for deriving test cases from risk assessment results using the baseline defined in RASEN deliverable D4.1.1 [33] as a starting point. It constitutes a first answer to the research question:

What are good methods and tools for deriving, selecting, and prioritizing security test cases from risk assessment results?

A technique for risk-based test identification and prioritization is presented in Section 2. An advanced security test pattern approach as an intermediate step between risk analysis and test case generation based on vulnerabilities from the risk model is described in Sections3.1 and 3.2. Test sequence generation based on formalizations using a Test Purpose Language is described in Section 3.3, 3.4, and 3.5. Section 4 presents a way to specify test case derivation by applying security test strategies to a model. How to generate actual security test cases on basis of these sequences is discussed in Sections5.1 and5.2.

The presented techniques show how risk-based security test case derivation can be done and provide the starting point for the subsequent deliverable D4.2.2.

References

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

[2] B. Damele A. G., M. Stampar: sqlmap – automatic SQL injection and database takeover tool (2013). [ONLINE] Available at: http://sqlmap.org/ [Accessed 13 September 2013]

[3] Deja vu Security: Peach fuzzer (2013). [ONLINE] Available at: http://peachfuzzer.com/ [Accessed 11 September 2013]

[4] DIAMONDS: Initial Security Test Patterns Catalogue. DIAMODS project deliverable D3.WP4.T1 (2012)

[5] FerruhMavituna: SQL Injection Cheat Sheet (2011). [ONLINE] Available at: http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/ [Accessed 11 September 2013]

[6] Fraunhofer FOKUS: Fuzzing library Fuzzino on Github (2013). [ONLINE] Available at: https://github.com/fraunhoferfokus/Fuzzino [Accessed 11 September 2013]

[7] Github: Sulley – a pure-python fully automated and unattended fuzzing framework (2013). [ONLINE] Available at: https://github.com/OpenRCE/sulley [Accessed 11 September 2013]

[8] MITRE: Common Attack Pattern Enumeration and Classification (2013). [ONLINE] Available at: http://capec.mitre.org/ [Accessed 11 September 2013]

[9] MITRE: Common Attack Pattern Enumeration and Classification – CAPEC-7: Blind SQL Injection (2013). [ONLINE] Availabe at: http://capec.mitre.org/data/definitions/7.html [Accessed 11 September 2013]

[10] MITRE: Common Attack Pattern Enumeration and Classification – CAPEC-66: SQL Injection (2013). [ONLINE] Available at: http://capec.mitre.org/data/definitions/66.html [Accessed 11 September 2013]

[11] MITRE: Common Attack Pattern Enumeration and Classification – CAPEC-67: String Format Overflow in syslog() (2013). [ONLINE] Available at: http://capec.mitre.org/data/definitions/67.html [Accessed 11 September 2013]

[12] MITRE: Common Attack Pattern Enumeration and Classification – CAPEC-90: Reflection Attack in Authentication Protocol (2013). [ONLINE] Available at: http://capec.mitre.org/data/definitions/90.html [Accessed 13 September 2013]

[13] MITRE: Common Attack Pattern Enumeration and Classification – CAPEC-109: Object Relational Mapping Injection (2013). [ONLINE] Available at: http://capec.mitre.org/data/definitions/109.html [Accessed 13 September 2013]

[14] MITRE: Common Attack Pattern Enumeration and Classification – CAPEC-135: Format String Injection (2013). [ONLINE] Available at: http://capec.mitre.org/data/definitions/135.html [Accessed 11 September 2013]

[15] MITRE: Common Attack Pattern Enumeration and Classification–CAPEC-152: Injection (Injecting control plane content through the data plane) (2013). [ONLINE] Available at: http://capec.mitre.org/data/definitions/152.html [Accessed 11 September 2013]

[16] MITRE: Common Weakness Enumeration (2013). [ONLINE] Available at: http://cwe.mitre.org/ [Accessed 8 September 2013]

[17] MITRE: Common Weakness Enumeration – CWE-20: Improper Input Validation (2013). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/20.html [Accessed 11 September 2013]

[18] MITRE: Common Weakness Enumeration – CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') (2013). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/89.html [Accessed 11 September 2013]

[19] MITRE: Common Weakness Enumeration – CWE-100: Technology-Specific Input Validation Problems (2013). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/100.html [Accessed 13 September 2013]

[20] MITRE: Common Weakness Enumeration – CWE-134: Uncontrolled Format String (2013). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/134.html [Accessed 11 September 2013]

[21] MITRE: Common Weakness Enumeration – CWE-301: Reflection Attack in an Authentication Protocol (2013). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/301.html [Accessed 13 September 2013]

[22] MITRE: Common Weakness Enumeration – CWE-306: Missing Authentication for Critical Function (2013). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/306.html [Accessed 13 September 2013]

[23] MITRE: Common Weakness Enumeration – CWE-564: SQL Injection: Hibernate (2013). [ONLINE] Available at: http://cwe.mitre.org/data/definitions/564.html [Accessed 13 September 2013]

[24] Open Web Application Security Project: Automated audit using SQLMap (2013). [ONLINE] Available at: https://www.owasp.org/index.php/Automated_Audit_using_SQLMap [Accessed 13 September 2013]

[25] Open Web Application Security Project: Testing Guide Project (2013). [ONLINE] Available at: http://www.owasp.org/index.php/OWASP_Testing_Project [Accessed 11 September 2013]

[26] Open Web Application Security Project: Testing Guide Project Testing for ORM Injection (OWASP-DV-007) (2012). [ONLINE] Available at: https://www.owasp.org/index.php/Testing_for_ORM_Injection [Accessed 13 September 2013]

[27] Open Web Application Security Project: Testing Guide Project Testing for SQL Injection (OWASP-DV-005) (2013). [ONLINE] Available at: https://www.owasp.org/index.php/Testing_for_SQL_Injection_%28OWASP-DV-005%29 [Accessed 11 September 2013]

[28] Open Web Application Security Project: Testing Guide Project Testing for Format String(2009). [ONLINE] Available at: https://www.owasp.org/index.php/Testing_for_Format_String [Accessed 11 September 2013]

[29] 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]

[30] Open Web Application Security Project: Top 10 2013-A1-Injection (2013). [ONLINE] Available at: https://www.owasp.org/index.php/Top_10_2013-A1-Injection [Accessed 11 September 2013]

[31] Open Web Application Security Project: Top 10 2013-A2-Broken Authentication and Session Management (2013). [ONLINE] Available at: https://www.owasp.org/index.php/Top_10_2013-A2-Broken_Authentication_and_Session_Management [Accessed 13 September 2013]

[32] Open Web Application Security Project: Top 10 2013-A7-Missing Function Level Access Control (2013). [ONLINE] Available at: https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control [Accessed 13 September 2013]

[33] RASEN: Baseline for Compositional Risk-Based Security Testing. RASEN project deliverable D4.1.1 (2013)